SOFTWARE TRANSACTIONAL MEMORY FOR
MULTICORE EMBEDDED SYSTEMS

A Thesis Presented
by
Jennifer Mankin
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Master of Science
in
Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts
August 2009
© Copyright 2009 by Jennifer Mankin

All Rights Reserved
Abstract

When general purpose computing hit the power wall, manufacturers turned to symmetric multicore processors in order to achieve performance gains while maintaining or lowering the clock frequency. Embedded processors can also benefit from the performance and power advantages offered by parallel execution on symmetric multicore processors. Unfortunately, management of concurrent accesses to shared memory is a common problem in shared memory parallel programs, protection from which is traditionally only offered through mutual exclusion, or locking. Coarse-grained locking is simple to implement and robust but limits performance, while fine-grained locking results in better parallelization but is error and deadlock-prone. Transactional Memory (TM) presents a solution to the problem, but has thus far been investigated only in general-purpose systems. In this thesis, we present Embedded Software Transactional Memory (ESTM), a novel solution to the concurrency problem in parallel embedded applications. We investigate several points in the Software Transactional Memory (STM) design space, and determine which decisions are best for an embedded platform. We present a full implementation of embedded STM, including both a non-real-time version (ESTM) and a real-time version (RT-ESTM), which contains added features to allow integration with real-time embedded applications. We compare the performance of the applications parallelized with ESTM to serial and parallel
versions utilizing coarse and fine-grained locking. For the non-real-time ESTM, we find that we can meet or beat the performance of fine-grained locking over a range of application characteristics. We attribute the performance advantage of ESTM to its efficient use of L1 memory and an optimization which allows customized memory protection of shared variables, increasing the transaction commit rate. These optimizations allow us to achieve the theoretical 2x speedup on a dual-core processor over a serial implementation on some benchmarks. We also evaluate RT-ESTM using an application with soft real-time deadlines, and show that it beats both coarse-grained and fine-grained locking. We show that by maintaining the performance level, power consumption can be reduced by over 26% over a serial application using RT-ESTM to manage concurrent accesses to shared memory.
Acknowledgements

First and foremost, I would like to thank both Dana and my family for all their unconditional love and support over the years.

I would also like to acknowledge my advisor at Northeastern, Dr. David Kaeli, and my technical supervisor at Draper, John Ardini, for their insightful, unending help throughout this entire process, culminating with this thesis. Finally, I would like to thank my colleagues at Northeastern and Draper for their invaluable feedback and discussions.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Internal Research and Development (IR&D), charge number 22949-008.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.
Contents

Abstract iii

Acknowledgements v

vi

1 Introduction 1

1.1 Introduction ............................................. 2

1.1.1 Distributed Memory .................................. 3

1.1.2 Shared Memory ...................................... 4

1.2 Contributions of Thesis ................................. 8

1.3 Organization of Thesis .................................. 10

2 Background 11

2.1 Multicore Synchronization .............................. 11

2.1.1 Lock-Free Synchronization Techniques .............. 13

2.2 Hardware and Hybrid Transactional Memory .......... 15

2.3 Software Transactional Memory ....................... 16

2.3.1 Embedded Transactional Memory .................... 20
## List of Figures

1.1 Coarse-Grained Locking .............................................. 5
1.2 Fine-Grained Locking ................................................. 6
1.3 Transactional Memory .................................................. 7

3.1 Embedded Software Transactional Memory API .................... 32
3.2 State Machine Representation of ESTM ............................. 33
3.3 High-level overview of ESTM transaction phases .................. 34
3.4 Execution of setup phase of ESTM transactions ................. 35
3.5 Execution of arbitration and commit/abort phases of ESTM transactions 36
3.6 Execution of all four transaction phases. .......................... 38

4.1 Memory hierarchy of the ADI BF561 dual-core embedded processor 45
4.2 Speedup as a function of size of shared data objects ............. 49
4.3 Breakdown of execution of critical section and overhead of STM calls 50
4.4 Speedup of critical section due to ESTM memory management relative to fine-grained execution .......................... 51
4.5 Transaction aborts per 1000 commits as a function of size of shared data objects .......................................... 52
4.6 Speedup as a function of percentage of execution time spent in the critical section ................................. 53
4.7 Transaction aborts per 1000 commits as a function of the execution time spent in the critical section ................................. 54
4.8 Speedup as a function of probability of contention in the critical section ...................................................... 56
4.9 Transaction aborts per 1000 commits as a function of probability of contention in the critical section ................................. 57

1 Pseudocode for function stm_start(). This function is called once per transaction. .............................................. 75
2 Pseudocode for function stm_open_mem(). This function is called once for each shared memory object to be accessed in the transaction. ...................................................... 76
3 Pseudocode for function stm_end_setup(). This function is called once per transaction. .............................................. 77
4 Pseudocode for function stm_end(). This function is called once per transaction. .............................................. 78
5 Pseudocode for function commit_transaction(), called by stm_end() .............................................. 79
6 Pseudocode for function abort_transaction(), called by stm_end() .............................................. 80
List of Tables

4.1 Potential power savings of the parallelized applications, relative to the lower bound of power consumption of the serial implementation. . . . 62
4.2 Potential power savings of the parallelized applications, relative to the upper bound of power consumption of the serial implementation. . . . 63
Chapter 1

Introduction

The recent availability of commodity off-the-shelf homogeneous embedded multicore processors has opened up the embedded design space to the same parallel programming paradigm that has given general purpose computing the opportunity for both performance and power savings. However, the use of these shared-memory multicore processors presents a challenging problem in dealing with interactions of threads on shared resources. Particularly, concurrent accesses to shared memory must be managed; we call this the shared-memory management problem. Traditionally, this problem has been handled with mutual exclusion; more recently, in general purpose computing, transactional memory has been proposed as a solution. In this chapter, we introduce the advantages and disadvantages of multicore processing, and discuss traditional and state-of-the-art solutions, including our own solution, to solving the shared memory management problem.
1.1 Introduction

In the last four decades, the computer industry has seen technology keep pace with Moore’s Law with an approximate doubling of the transistors on a chip every two years. Until around 2002, there was a proportional increase in performance; since then, however, there has been a lag in the performance improvement achieved relative to transistor counts. This can be largely explained by heat dissipation problems caused by increased power consumption of modern processors. Decreased transistor sizes results in non-negligible static power consumption, and increased clock frequencies contribute to higher dynamic power consumption. For these reasons, multicore processors have become a promising solution to the performance and power problem. With two or more processing cores, the operating frequency of each core can be reduced, and still there will still be an overall performance improvement. Since a slower clock frequency requires less supply voltage, this will result in a further reduced dynamic power consumption.

While general purpose and high-performance computers have only recently embraced multicore architecture, embedded systems have been employing heterogeneous multicore processing architectures in the form of Multiprocessor System-on-Chips (MPSoCs) for some time. These heterogeneous architectures use multiple cores of different types, and use a different parallel programming model in which tasks are assigned to cores based on the type of work they do, and there is little interaction between tasks on different cores. However, it is a less flexible architecture, and it is difficult to create a balanced load across all cores, since allocation of tasks to cores is strictly dependent upon the type of work each task performs.

Recently, embedded chip manufacturers have started looking at Symmetric Multiprocessors (SMPs) for the same reasons that the general purpose domain turned
to SMPs. First, threads can be allocated (statically or dynamically) to cores to achieve a balanced load and thus higher parallelism. With work divided across cores, higher performance can be attained for a given clock frequency due to the increased parallelism. Additionally, the same performance level can be maintained by increasing parallelism but decreasing the clock frequency. Decreasing the clock frequency allows for a decreased core voltage, and since $P_{DYN} \propto V_{DD}^2 f$, there is a great potential for reduction in power. This second benefit is especially useful in embedded systems, which do not generally employ advanced heat dissipation technology like fans or area-expensive heat sinks. Additionally, embedded processors are often used for portable, battery-powered devices, and thus are more energy sensitive. The advantage of a symmetric multicore embedded processor is obvious: It will allow for increased performance due to parallelism or decreased power consumption given the same performance level.

Unfortunately, converting a serial application into a parallel application for execution on a multicore processor is not often a simple or easily-automated task. A critical problem arises when threads on different cores need to access or operate on the same shared data. How threads manage operation on shared data depends on the memory architecture of the system. The memory architectures of multiprocessing can be coarsely divided into two categories: Distributed Memory and Shared Memory.

1.1.1 Distributed Memory

In a distributed memory platform, each processing core is associated with its own memory, and therefore, its own exclusive address space [35]. The advantage to this system is scalability: there is no bus contention or cache coherency problems since each processor is directly connected to its own memory. However, other cores which
need access to this data must request the data from the owner using message passing. Thus, it is critical for an application on a distributed memory system to have relatively independent threads which do not frequently require data stored in memory on other cores.

1.1.2 Shared Memory

The work in this thesis deals with concurrency on a shared memory system. In a shared memory system, there is a shared memory address space accessible by all processing cores [35]. A small but arbitrary number of processors are connected to a single bus, and thus any processing core can directly access any part of shared memory. Higher rates of interaction between threads can be tolerated than with the distributed memory model. However, there are two main problems with the shared memory architecture. First, the use of a single bus leads to contention and thus limits scalability. Secondly, there needs to be a management mechanism for controlling accesses to shared resources. That is, special care must be taken to conserve a coherent and consistent view of memory while different threads on different cores are concurrently reading and writing to shared memory. Coherence is usually managed through a cache coherence protocol, though it may require manual management by the programmer. To maintain consistency, the ordering of memory operations must be maintained from a thread’s point of view as memory instructions of different threads cannot be interleaved—that is, each thread’s critical section must be maintained.

Mutual Exclusion

One solution to protecting critical regions in a thread’s execution is with mutual exclusion, or locking. However, this method has severe limitations. In coarse-grained
locking, one-to-few locks are used to protect all of shared memory.

// Coarse-grained locking

// Thread A:
acquire_lock(&lock);
shared_a++;
shared_c++;
release_lock(&lock);

// Thread B:
acquire_lock(&lock);
shared_b = local_b;
release_lock(&lock);

Figure 1.1: Coarse-Grained Locking

Consider Figure 1.1, an example of coarse-grained locking. Threads A and B are both attempting to acquire the same lock simultaneously, so one thread will stall until the first thread can release the lock. However, they are actually accessing disjoint memory locations, so it is a false conflict—there is no need for one thread to stall waiting for the lock. This example illustrates the strengths and weaknesses of coarse-grained locking. It is very simple to implement and robust (i.e., there is little to no opportunity for deadlock). However, it can serialize code when threads conflict while attempting to acquire the same lock; therefore, the use of coarse-grained locks can limit the performance gains offered by a parallel execution.

With fine-grained locking, more locks are used to protect shared memory, such that each lock protects a smaller region of memory. Figure 1.2 shows the same example as Figure 1.1, now implemented with fine-grained locks. As the example shows, fine-grained locking can reduce or eliminate contention by reducing false conflicts, thus it achieves better performance from the increased parallelism. However, as the number
of locks and threads increases, fine-grained locking becomes notoriously difficult to implement and can result in hard-to-find concurrency bugs and deadlock. Additional steps, such as always acquiring locks in order or utilizing two-phase locking (in which a thread is either acquiring but not releasing locks, or releasing but not acquiring locks), must often be used to reduce the possibility of deadlock, but these in turn also reduce the performance of the parallel implementation.

// Fine-Grained Locking

// Thread A:
acquire_lock(&lock_a);
shared_a++;
release_lock(&lock_a);

acquire_lock(&lock_c);
shared_c++;
release_lock(&lock_c);

// Thread B:
acquire_lock(&lock_b);
shared_b = local_b;
release_lock(&lock_b);

Figure 1.2: Fine-Grained Locking

Transactional Memory Models

Transactional Memory, first proposed by Herlihy and Moss [43], addresses the problems associated with locking by allowing code to be divided into coarse-grained transactions, but detecting conflicts on a finer granularity. Consider Figure 1.3, the same example now delimited with transactional calls.

The critical regions are protected with coarse-grained transactional calls, but
Figure 1.3: Transactional Memory

within the call to end the transaction, the memory conflicts are detected with a fine-grained precision by monitoring the shared data which is actually modified. The stated goal of transactional memory, therefore, is to provide performance on the order of fine-grained locking but with the ease-of-programming and robustness of coarse-grained locking.

A transaction is a sequence of operations which executes speculatively and is guaranteed to commit or abort only as an atomic unit. That is, if the transaction does not read or write to memory locations which have been subsequently written to by another transaction, then it will commit, and all its changes will become permanent to shared memory. However, if a transaction has memory conflicts with another transaction, it will abort, and all the changes it made to shared memory will be discarded and the transaction will roll-back and re-execute.

Transactional memory approaches can be broadly classified into one of three types: Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM). HTM implementations provide lower
overhead than STM implementations, but have architectural limitations which make them unlikely to be adopted. 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 STM if transactions do not fit the limitations of HTM.

In this thesis, we will present an *Embedded Software Transactional Memory* model as a means to solve the concurrency problem for shared-memory symmetric multicore embedded systems. We will discuss the model, its implementation, and its effect on application execution. We will discuss the challenges involved in converting from a serial to parallel application. We will target a specific class of applications—the *Guidance, Navigation, and Control (GNC) System*—and will present our results on two synthetic GNC benchmarks which simulate the range of behavioral characteristics of a real GNC application.

### 1.2 Contributions of Thesis

The contributions of this thesis are summarized below:

- We investigate the design space of Software Transactional Memory models and analyze the decisions which are unique to the embedded platform. We describe the motivation behind the decisions made for the embedded platform.
• We implement Embedded Software Transactional Memory (ESTM), a fully-functional STM which controls concurrent accesses to shared memory for a commercial embedded multicore platform, the Analog Devices Dual-Core BF561.

• We demonstrate how to effectively utilize ESTM for parallelizing applications, and show how to effectively balance the load across cores to minimize memory conflicts between transactions using the Weighted Graph Partitioning model.

• We investigate and implement optimizations unique to an embedded STM and demonstrate that these design-space-specific optimizations help ESTM beat the performance of coarse-grained locking, and even approach or beat the performance of fine-grained locking.

• We implement features for the ESTM model in order to support applications with soft real-time deadlines and call this Real-Time ESTM, or RT-ESTM. We also investigate the challenges of utilizing a multicore platform for a real-time embedded application.

• We perform experiments to compare our ESTM to both forms of traditional locking and demonstrate that it can approach and even beat the performance of fine-grained locking using coarse-grained transactions. We also investigate the power ramifications of parallel execution and demonstrate that we can reduce the power consumption of an embedded processor by using ESTM over locking or sequential execution.
1.3 Organization of Thesis

In Section 2, we discuss related work in concurrent programming and shared memory management mechanisms; specifically, we discuss mutual exclusion, and hardware, software, and hybrid transactional memory implementations. In Section 3 we introduce Embedded Software Transactional Memory, analyzing all relevant design decisions which are made for any STM implementation and detailing the motivations behind the decisions made specifically for ESTM. We then discuss the implementation of ESTM and RT-ESTM, including the data structures, Application Programming Interface (API), and usage of the ESTM library. Additionally, we discuss the optimizations which can be made to ESTM given the nature of the GNC applications and unique characteristics of the embedded platform. In Section 4, we present our benchmarks, and our parallelization methodology. We compare ESTM to both fine and coarse-grained locking and demonstrate that the use of ESTM results in a performance approaching fine-grained locking with effort and robustness closer to coarse-grained locking. Additionally, we show how the optimizations for real-time embedded applications made RT-ESTM a more viable solution for parallelization of real-time application than locking. Finally, we discuss the power ramifications of parallelization using both traditional locking and ESTM. Section 5 presents a discussion of Software Transactional Memory (in general) and our embedded implementation (in particular). We conclude the thesis in Section 6.
Chapter 2

Background

2.1 Multicore Synchronization

Synchronization between tasks or threads has been a problem since even before multicore or multiprocessors when single core processors became multi-threaded or had multiple tasks. In this case, a task preempting another task could still result in concurrency problem if both were accessing a critical section.

Mutual Exclusion restricts access to memory by preventing more than one thread from accessing it concurrently. Given hardware support for atomic locking instructions, locking has become a favored way to provide mutual exclusion. Coarse-grained locking divides memory into large chunks, each of which is protected by a single lock; this mechanism is simple to implement, and given that only a limited number of locks are used, the application developer can trivially prevent against deadlock. However, it overly protects regions of memory, unnecessarily blocking parallel accesses to disjoint memory locations, limiting parallelism opportunities and performance benefits. Fine-grained locking operates on a much smaller region of memory. The disadvantage
is that it is notoriously difficult to implement and maintain as bug-free, as deadlock is common and hard to debug.

Transactional Memory corrects the problems of locking and provides the potential for performance on the order of fine-grained locking, but with the ease-of-programming and robustness of coarse-grained locking. A transaction is a sequence of operations which executes and is guaranteed to commit or abort only as an atomic unit. At the end of a transaction, the transaction attempts to commit or abort depending on whether there were any memory conflicts with another transaction. A conflict occurs when one transaction accesses (reads or writes) data which is written to by another transaction. If there are no conflicts, a transaction will commit, and its changes will become permanent to shared memory. If memory conflicts occurred, the transaction must abort, discard any changes it made, roll back to the start of its execution, and re-run.

Transactional Memory can be broadly divided into two categories: Hardware Transactional Memory (HTM) and Software Transactional Memory (STM). Additionally, Hybrid Transactional Memory (HyTM) combines the advantages of both types of transactional memory. HTM is typically faster than STM. However, it requires modifications to the hardware, and has architectural limitations. Specifically, transactions must be small enough to fit in cache, and short enough to not be preempted by the scheduler. Since the goal of transactional memory is to provide a clean interface for the programmer to exploit concurrency, the requirement that a programmer understand the characteristics and memory behavior of their application to ensure it fits within specific architectural constraints is not an acceptable solution. While spatially and temporally unbounded techniques exist, they have been deemed too complex to be implemented in future processors, and in some implementations
there is still a heavy performance penalty when such architectural limitations are exceeded.

STM, on the other hand, is known to be slower than HTM, but provides a means to support transactions without hardware modifications. The performance loss is due to the STM library calls which manage the transactions; however, this overhead can be minimized through STM optimizations and through compiler support. Additionally, STM does not suffer from the architectural limitations of HTM; as transactional data is not stored in transactional caches, it can be unbounded in both size and execution time. Large transactions, in turn, amortize the overhead of the STM library calls, minimizing the effect of STM on performance.

### 2.1.1 Lock-Free Synchronization Techniques

Because locking synchronization mechanisms often have a high Worst-Case Execution Time (WCET), due to the potential for deadlock and priority inversion, it is difficult to provide real-time guarantees. In response to this problem, researchers began investigating lock-free synchronization techniques [10, 16, 44, 46]. Lock-free techniques rely on the concept of a *retry* or *read-and-check* loops. In such an implementation, a thread must ensure that none of the memory accesses it made were interfered with (that is, written to by another thread) during the execution on shared memory values. A typical way to check this is by wrapping the shared memory access in a loop which executes indefinitely, until a check (often a Compare-and-Swap (CAS)) indicates that the value had not been interfered with.

Anderson et al. consider hard real-time tasks sharing lock-free objects and demonstrate that interferences can be bounded by considering task scheduling [10]. Cho et al. also characterize the behavior of lock-free synchronization [16]. Since tasks which
are the most urgent (i.e., have the shortest deadlines) may not be the most important, they consider utility and aim to determine an upper bound on retries caused by lock-free interferences. Lai et al. then extend the utility-based scheduling algorithm to help dynamically alter the supply voltage and clock frequency for reduced energy consumption [46]. Huang et al. present and evaluate several lock-free algorithms; these algorithms, however, only operate on the assumption that the tasks consist of multiple readers but only a single writer [44]. The authors trade off some algorithm complexity of read-and-check loops for additional space requirements in presenting their double buffer implementation, and also present optimizations to their double buffer mechanism to reduce some of the space requirements.

Several implementations of the \textit{Multiword Compare-and-Swap} (MCAS) provided concurrent wait-free or lock-free access to shared objects on multiprocessor systems [8, 9, 31]. The MCAS, as the name suggests, compares the contents of a range of memory to its old values; if no changes have occurred, the contents are updated with the new values. An original implementation by Herlihy et al. limited parallelism with overly restrictive comparisons [40]; Anderson et al. remedied this problem with their MWCAS, which permits operations on disjoint sets of objects to occur in parallel [8]. They also presented a “helping” mechanism by which one processes helps a competing process to complete a transaction. Anderson et al. presented several helping schemes are presented in the hope that they could be applied to objects on priority-based systems [9].

These lock-free synchronization mechanisms require a great deal of effort on the part of the programmer to maintain a consistent view of memory, as all shared memory accesses need to be manually transformed into the retry loops. Additionally, an inherent problem with MCAS operations is that they do not account for concurrent
read-level parallelism; that is, 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 its operation. Our STM implementation avoids this problem by managing shared memory reads in addition to shared memory writes.

\section{2.2 Hardware and Hybrid Transactional Memory}

Hardware Transactional Memory was first proposed by Herlihy and Moss \cite{Herlihy94}; their elegant solution added extensions to the multiprocessor cache coherence protocols and a \textit{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 \textit{unbounded} HTMs; unbounded implementations provide mechanisms for transactions to commit even if they exceed a system's architectural resources and/or scheduling quantum \cite{Lin2012, Hammond13, Hammond15, Lin15}. With \textit{Transactional Memory Consistency and Coherency (TCC)}, Hammond et al. proposed an unbounded HTM architecture which maintained transactions in existing caches (adding extra bookkeeping bits) and relied on broadcasting at transaction commit to keep memory consistent and coherent \cite{Hammond13}. However, in order to allow arbitrary transactions to commit, it had to maintain a separate victim cache for large transactions, or stall execution until that transaction can execute and commit. Given the nature of the broadcast and that commits must be serialized, TCC is not scalable for large numbers of cores, and the hardware overhead is too large to be practical. The LogTM
of Moore et al. used eager version management, performing all modifications in-place in shared memory and using undo-logs to roll-back memory to a consistent state after a transaction commit [56]. However, in many cases this “eager versioning” does not perform as well as a lazy-versioning scheme such as that employed in TCC.

Another approach to managing arbitrary transaction sizes is Hybrid Transactional Memory (HyTM) [12, 20]; 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. For example, the HyTM of Baugh et al. [12] used a simple best-effort HTM implementation, similar to the original HTM proposal of Herlihy and Moss [43], to run most transactions in hardware. If a transaction aborted, the abort handler decided whether to retry the transaction in hardware or failover to the STM. This is a strong approach because, as shown by Hammond et al. [36], most transactions will fit in even the smallest of modern caches. Thus, the common case utilizes the fast HTM hardware, and does not suffer a performance loss associated with executing small transactions with STM. By resorting to slower STM when running transactions overrun the architectural resources on HTM, HyTM avoids the main pitfall of HTM systems. Thus, HyTM can provide the advantages of both systems, and reduce the disadvantages.

2.3 Software Transactional Memory

Early Software Transactional Memory models were static. In a static implementation, the data set to be accessed by the transaction, and the transactions themselves, are
known in advance. The first STM model addressed the problems of MCAS operations; it was created by Shavit and Touitou [63]. Their STM was non-blocking and could be implemented on machines with a *Load Linked/Store Conditional* primitive.

The work of Herlihy et al. ushered in a new era of Software Transactional Memory with its DSTM: Dynamic Software Transactional Memory [42]. DSTM was developed in both C++ and Java. This implementation was non-blocking: It required that a thread which halts for an indefinite period of time or fails cannot prevent other threads from making progress. Like many successive implementations, it required the weakest form of non-blocking guarantees, *obstruction freedom*, which simplified the non-blocking requirement by guaranteeing progress only in the absence of contention [50]. Because data is accessed through a *locator* object, which is in turn referenced from a *Transactional Memory Object*, one criticism of this implementation was that the double-level of indirection created to manage data objects results in a performance degradation due to resulting cache misses [24].

The authors admit 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 sought to correct this with DSTM2 with *transactional factories* which allow programmers to “plug in” their own synchronization and recovery mechanisms [41]. In their paper, the authors demonstrate the flexibility of transactional factories by presenting and evaluating two very different factory classes: the obstruction-free DSTM of [42], and lock-based *Shadow Factory*, an implementation that seeks to avoid the indirection costs of the original DSTM implementation.

Fraser et al. present three unique APIs to aid in concurrent programming [31]; all three seek to decouple a memory location’s *physical contents* from its *logical contents*
and ensure that the logical contents of a set of locations can only be updated atomically through the use of a commit operation. The first, Multiword Compare-and-Swap or MCAS, is reminiscent of early STM implementations. It provides a low-level API which can result in cumbersome code as the program must maintain a list of read locations as successive MCAS operations do not know which locations have been read. It does, however, have a very low overhead when used in the right situations or by an expert programmer. The second abstraction is a Word-based STM or WSTM which solves the read-parallelism problem of the MCAS implementation, but all memory accesses must be wrapped in either WSTMRead or WSTMWrite function calls and this implementation often results in reduced performance compared to MCAS. Their final 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, though transactions with a large number of writes may be faster in DSTM with its simpler compare-and-swap operation [50].

Saha et al. discuss and evaluate STM tradeoffs, and implement their own STM, called McRT-STM, based on their findings [60]. Their STM is non-blocking with a two-phase locking protocol, and to prevent deadlock, they use a self-aborting strategy for transactions which wait a certain time to acquire a lock. Their STM implements two-phase locking for writes and versioning for reads, and rolls back changes by aborted transactions using undo-logs. They find that with certain applications they can approach fine-grained locking with enough processors. In other applications, they can beat fine-grained locking. 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) and show that it
scales as well as STM and better than HTM, and approaches HTM performance in
the single-thread case \[61\].

Another interesting body of work is called *Transaction Locking* or *TL* \[22\]. With
TL, Dice et al. analyze and evaluate several existing STM implementations to de-
termine the strongest design decisions, and use this information to create their own
STM. They agree with Ennals \[27\] in making the case that blocking STMs achieve
superior performance than their non-blocking counterparts, and that deadlock avoid-
ance is the only justifiable reason for using a non-blocking implementation. They
compare lock acquisition at both *encounter-time* and *commit-time*, and find that
commit-time algorithms have the best scalability across contention ranges. Finally,
they show that code generated mechanically with TL has superior scalability to other
handcrafted data structures. In their followup work they improve upon TL by intro-
ducing a global version clock \[23\]. Their new *TL2* eliminates the requirement of a
"safe" running environment, so that a thread will not be operating on inconsistent
memory states.

Much of the STM research has been done in conjunction with the Java language
\[1, 41, 42\], but some work has also been done creating STM libraries in C or C++
\[17, 19, 51, 57\]. Without memory management like garbage collection built into the
language, there are additional challenges that these works encounter. Marathe et
al. designed a low-overhead, obstruction-free STM for non-garbage collected systems
called RSTM \[51\]; they demonstrated a performance gain which is better than their
Java-based system \[48\]. The authors admit that the API is cumbersome and prone
to programmer error, and thus offer RSTM2 [19]. RSTM2 uses macros, multiple inheritance, templates, and operator overloading including *smart pointers* to provide ease-of-use.

Crowl et al. presented an “experimental” C/C++ implementation in order to gain experience for a more comprehensive solution in the future [17]. They choose to use a *control* statement to introduce transactions, designated with the keyword *transaction*. They discussed several design decisions, including nesting, interactions between transactional and non-transactional memory accesses, and means of handling I/O, system calls, and libraries. Finally, Ni et al. described new first-class language constructs for C/C++, an optimizing C++ compiler to optimize these extensions, and a new STM runtime library [57].

### 2.3.1 Embedded Transactional Memory

It is worth noting that, though has been no work on embedded *software* transactional memory to date, Ferri et al. explored *hardware* transactional memory for MPSoCs [28]. They modeled their implementation after the original TM of Herlihy and Moss [43], 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; the scheme also requires two additional bits to the cache coherence tag vector and two additional signals on the cache snooping device. In this work, the authors consider both performance and power, and in their best benchmark find a 71% better energy-delay product (EDP) than a locking mechanism on an eight-core system.
Chapter 3

Embedded Software Transactional Memory

The implementation of Software Transactional Memory (STM) for an embedded system presents different challenges than are faced in the development of an STM for a general purpose system. While embedded transactional memory implementations and discussions are rare, there has fortunately been a great body of work dedicated to STMs in general-purpose systems. We can look at the decisions and the results of these implementations and extrapolate conclusions which are specific to the embedded platform. Some of the challenges which are different between general-purpose and embedded STMs are:

- Low Overhead Requirements: Embedded applications are typically so performance constrained that application developers fine-tune the code to achieve maximum performance. Thus an STM implementation must maintain a very low overhead to manage the transactions. Fortunately, given that the application developers are so well acquainted with their applications, there is some
flexibility in programmer knowledge and application assumptions which aid in keeping the overhead of STM calls low.

• Less Operating System (OS) Support: Embedded systems typically do not have the feature-rich, fully-implemented OS support enjoyed by general purpose systems. Typically, there is only a small kernel which provides the functionality of threading and some basic synchronization mechanisms. There is little support for scheduling; generally scheduling is manually simulated in the threads using semaphores and timers and direct kernel calls. Additionally, there is even less support for multicore processors, including a lack of operating systems able to manage an SMP processor. Therefore, a separate kernel image needs to be placed on each core, rather than a single OS managing resources on all cores, thus limiting synchronization and threading options.

• Less Architectural Support: The embedded processor ISA may not feature support for such non-blocking synchronization mechanisms popular with many non-blocking implementations, such as Compare-and-Swap (CAS) and Load-Linked/Store-Conditional (LL/SC). Our chosen processor, the ADI BF561, provides only a single atomic primitive, the Test-and-Set Lock, thus eliminating non-blocking implementations from consideration.

• Unique Memory Hierarchy: While general-purpose multiprocessor systems feature memory hierarchies with fast caches and automatic cache coherency protocols, embedded systems lack complex and fast memory hierarchies. While there may be L1 cache present in the system, it is often configured as standard SRAM for power efficiency, performance improvement, or deterministic execution. Furthermore, there is no cache coherency protocol: developers are required
to manually flush the shared data from the cache and invalidate the correspond-
ing cache entries any time they access shared data that may be subsequently
used by the other core. Since memory is shared at the L2 and external memory
levels, the inability to use the L1 memory as cache means that the L1 memory
level would go largely unused unless the memory architecture is considered in
the development of the STM model.

In this chapter, we introduce Embedded Software Transactional Memory (ESTM).
We begin with an introduction to common STM design decisions and the right de-
cisions for an embedded implementation. We discuss the ESTM implementation, its
internal data structures, its API, and the optimizations which support GNC algo-
rithms on an embedded platform. Finally, we introduce modifications which can be
made to enable ESTM to work in soft real-time applications, providing a more robust
way for threads to meet their real-time deadlines than can be done with traditional
locking.

3.1 STM Design Decisions

Given that this is the first implementation of Embedded Software Transactional Mem-
ory, 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 em-
bedded multi-core system. This section discusses these design decisions and motivates
the choices made in our implementation.
3.1.1 Static vs. Dynamic STM

Transactions can be broadly classified into two categories: static and dynamic. Static transactions [54, 63] require that data accessed within a transaction, and the transaction itself, are defined in advance. In dynamic STMs [1, 31, 41, 42, 51, 60, 61], 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 [31].

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. While dynamic STM implementations provide greater ease-of-use for the programmer, they can add overhead in both implementation complexity and time. Since embedded systems software developers fine-tune their applications, they already have intimate knowledge of its memory behavior. Additionally, we are not as concerned with dynamic-sized data structures; our targeted embedded applications typically have strictly deterministic (i.e., static) data structures. 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.

3.1.2 Granularity of Conflict Detection

An important consideration in the STM implementation is the granularity of the memory accesses in a transaction. A word-based or cache-line-based scheme will detect conflicts across a range of memory locations [17, 31, 60], or, as described by Herlihy et al., by intercepting direct memory accesses [41]. A word-based STM [17, 31]
requires its own metadata to be maintained separate from the data itself, and often
code must be inserted around every memory access to indicate that it is a transactional
read or write.

Object-based conflict detection [31, 41, 42, 51, 48, 61] operates at a coarser granu-
larity. In an implementation done in an object-oriented language like Java [41, 42, 61],
conflict detection is done over an abstraction of memory—the object—rather than the
range of memory itself. Two transactions writing to disjoint elements of the same
array 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 ma-
nipulate pointers makes C suitable for a word-based approach. Further, through the
use of address books (described in Section 3.2.1), we do not require that code is inserted
around every memory access, as in previous word-based STM implementations.

3.1.3 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 [31, 41, 42, 51]. Later research demonstrated the merit
of a blocking or lock-based approach [1, 24, 27, 60, 61]. Lock-based approaches are less
complex and have less overhead, and research has shown them to be faster than their
non-blocking counterparts [24, 27, 60]. Dice et al. cite the maintenance of publicly
shared records as a primary source of overhead in non-blocking implementations. The
downside of using a locking implementation is the possibility of deadlock. However,
locks are only used in the STM itself, not in application code, so the programmer
does not need to consider them in their application design. Further, deadlock can be
eliminated with a timeout mechanism within lock acquisitions.

As embedded applications are generally fine-tuned and optimized for best per-
formance, we do not feel the overhead of a non-blocking implementation is justified.
We take steps, described in Section 3.1.4, to minimize the effects of the locks. Addition-
ally, due to the limited resources of the Blackfin processor, there is no hardware
support for an atomic read-modify-write or compare-and-swap primitives, eliminat-
ing most obstruction-free implementations from consideration. As the hardware does
provide an atomic lock instruction, the decision to implement a blocking STM is an
easy one.

### 3.1.4 Object Acquisition

The acquisition of an object occurs when a transaction asserts ownership of an object
in a non-blocking implementation [48] or when a transaction acquires the lock or
version number of an object in a blocking implementation. There are two variations:
*eager acquire* and *lazy acquire*. In eager acquisitions [27, 41, 42, 60], the objects
are acquired as memory locations are encountered; in lazy acquisition, the locks are
acquired only at commit time [23, 31]. The benefit of an eager acquisition is that
conflicts between transactions are detected early, so transactions which will eventually
abort do not perform useless work. With a lazy acquire, a transaction which will
eventually abort does not hold the lock for the duration of its execution [51].
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—once very briefly at the start of the transaction, and then longer at commit 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 main read lock.

### 3.1.5 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 this local copy, and the data is written back to shared memory only at transaction commit. In an undo-log implementation, all writes occur 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 [60], 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 [11]). To address this issue, there has been a great body of research [26, 30, 45, 58] 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.

3.1.6 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 using a contention manager [27, 31, 42, 60]. In DSTM [42], 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. Several policies were analyzed by Scherer et al. [66]; the Polite contention manager is cited as one of the best [27].

Despite the prevalence of contention managers in STM implementations, Dice et al. claim that they are unnecessary and can be replaced by a timeout [24]. 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. In our ESTM, the programmer denotes the maximum number of times a transaction should abort before being called “starving”. When a transaction becomes starving, the conflict manager steps in to ensure that no conflicting transactions will commit until the starving transaction can commit; this contention manager is discussed in more detail in Section 3.2.3.

### 3.2 Embedded STM

For each transaction, the programmer must declare two objects: 1) A Transaction object and 2) A local AddressBook object, as described in Section 3.2.1. 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 the implementation.
3.2.1 Data Structures

ESTM Address Book Structure

For our ESTM, we introduce 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 will contain 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 [31, 38] around each memory access 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 transform is more intuitive and readable and does not add to the lines of code:

matmult(addr_book->result, addr_book->mat1, addr_book->mat2);
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 was 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.

**ESTM Transaction Structure**

The Transaction struct maintains all necessary information for the transaction; it contains a unique ID for the transaction, a flag for 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, whether or not that variable is read-only, the size of the variable or array, and the version number at the time the data was copied into local memory.

### 3.2.2 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 3.2.2.
stm_start(Transaction* trans, void** local_address_book,
    void* global_address_book, int address_book_size,
    int max_aborts);

stm_open_mem(Transaction* trans, void** address, int size_bytes,
           int read_only);

stm_end_setup(Transaction* trans);

stm_end(Transaction* trans);

Figure 3.1: Embedded Software Transactional Memory API

3.2.3 ESTM Implementation

In this section, we step through the execution of a transaction, describing the operations that are performed in each function call of the ESTM API. Figure 3.2 shows a state machine representation of the execution of a transaction, specific to the ESTM implementation.

Figure 3.3 shows the execution of two conflicting transactions on a high level. For each transaction, there are four phases: Setup, Execution, Arbitration, and Commit/Abort. Transaction A and Transaction B are running concurrently on two different cores, and both write to the same shared variable (var2). Since Transaction A enters arbitration first, it commits, while Transaction B, entering arbitration second, aborts. The phases are described in more detail in the following subsections.

Transaction Setup Phase

Since each transaction must have a consistent snapshot of memory for use during execution, another transaction cannot write its results to shared memory while another transaction is in the setup phase. Thus, when entering the setup phase, a transaction
must acquire a global lock and increment the number of **readers** (i.e., the number of transactions currently reading from shared memory). This **reader** variable is incremented in the within the call to **stm_start**, as shown in Figure 3.4.

Additionally, within the call the **stm_start**, the **Transaction** object member variables 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.

Next, the programmer makes one call to **stm_open_mem** for each shared variable or array which will be 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 repointed to the new local copy. Thus, for the rest of the transaction, all accesses to that address will be directly 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 size of the data, the version number, and whether or not it is read-only.

The last step of the setup is a call to \texttt{stm\_end\_setup}. At this point, the \texttt{readers} counter is decremented so that a transaction which 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.

**Transaction Execution Phase**

With our setup complete, the program continues with its original execution. The body of the original code does not need to be transformed, except that variables are now accessed through the local address book, as described in Section 3.2.1.
Figure 3.5: Execution of arbitration and commit/abort phases of ESTM transactions

Transaction Arbitration Phase

At the end of the transaction, the programmer makes a call to the function `stm_end` and enters the *arbitration* phase. In this phase, the transaction must first ensure that no other transactions are actively in the setup phase, at which point they would be acquiring consistent copies of shared memory. The transaction acquires the global lock, then checks the number of readers. If the *readers* counter is greater than zero, the transaction releases the global lock to allow the other transactions to complete their setup phases. Once the *readers* counter is zero, the transaction retains the global read lock to ensure that no other transaction can begin 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 have changed since the versions recorded in the setup phase, it is allowed to commit. Otherwise, it must abort. The arbitration phase is shown in Figure 3.5.

**Transaction Commit/Abort Phase**

After the arbitration phase, the transaction enters either the *commit* or *abort* phase, as shown in Figure 3.5. 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 the local memory to the shared memory using the information stored in the Transaction object, and the version numbers of modified data are incremented in shared memory. The private data is 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 3.2.3) 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 from execution from just after `stm_start` left off.

The complete execution of the two conflicting transactions is shown in Figure 3.6.

**Contention Manager**

The *Contention Manager* prevents a transaction from starvation. This occurs when a transaction, which 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 and provide a mechanism for fairness, we propose a contention manager which gives the programmer some control over starving transactions: the `max_aborts` member variable of the `Transaction` object. At the start of the transaction, the programmer provides the `max_aborts` value which indicates how many times a transaction is allowed to abort before it blocks commits for other 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. Note that, if it is required that a transaction have a `max_aborts` value of zero, the `Real-Time` conflict manager must be used, as discussed in Section 3.4.
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. By the time the contention manager is invoked for an aborting transaction, any conflicting transactions will have already committed. Since the conflict manager cannot step in and abort the transaction which has already committed, it 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 thread has successfully committed.

### 3.3 ESTM-Optimized

Transactional Memory offers the opportunity for optimizations not present in traditional locking. One such optimization is the unique opportunity to customize the level of protection of a shared variable. We propose allowing *stale-reads*, a concept similar to the *early-release* optimization used in other transactional memory implementations [42].
In this thesis 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 writing 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, the programmer can denote read-only data as a stale-read, and it will not cause a conflict regardless of version number changes. This allows for faster commits of read-only data, and automatic commits for read-only transactions, such as in threads which read the filter state to control output. When we use this optimization in our ESTM implementation, we call it ESTM-Optimized or ESTM-Opt. It is the programmer’s responsibility to ensure correctness when using stale reads (as is the case for usage of early-release implementations as well), though the transaction is still guaranteed a consistent copy of memory since the setup phase is protected regardless of whether the data is marked stale. If this policy is not appropriate for the application, a stricter policy can be used by declaring all data as read/write; we refer to this as standard ESTM or just ESTM.

There is no safe way to operate on “stale reads” with traditional locking without violating the rules of memory consistency. In locking, a memory location must be either locked or unlocked: either there is exclusion from other threads writing to the location or there is not. If data is not protected by a lock, there is a finite probability that a thread will write to the locations concurrent to another thread reading from it. If this happens, the reading thread will view an inconsistent view of memory, as part of the reads will have occurred on the old copy and part on the new copy. Therefore,
the stale-read optimization is an optimization which is only possible in STM, and not in traditional locking, thus providing a performance advantage.

3.4 Real-Time ESTM (RT-ESTM)

We added two features to ESTM to enable its use in periodic real-time embedded applications; we call this modified implementation Real-Time ESTM, or RT-ESTM. The first is the relocation of the contention manager to the start of the transaction. Section 3.2.3 described the behavior of the standard contention manager of ESTM, which it is invoked only upon transaction abort. In the standard contention manager, if a transaction was aborted more times than was necessary, the contention manager would block commits on any conflicting transaction in order to allow the starving transaction to commit. This methodology is preferable for non-real-time applications, as it is acceptable to define a transaction as “starving” only after it has aborted at least once.

However, a problem arises in a real-time system with priority driven threads, given that the standard contention management policy has the restriction that a transaction abort at least once before the contention manager is invoked. In a real-time system with priority-driven tasks, it may be necessary for a high-priority transaction to be guaranteed to commit on its first attempt (this is also important for I/O driven tasks, if the I/O data needs to be committed to shared memory as it arrives). The solution to this problem is to allow a transaction to consult the contention manager at the start of the transaction. In this case, if the transaction needs to commit on its first attempt, the contention manager will block all conflicting transactions from committing while the high-priority transaction runs to completion. After it commits,
the contention manager unblocks commits.

The downside to this modification is that it performs an additional check on the transaction each time the transaction runs—rather than checking only at transaction abort. This overhead is justifiable in a real-time embedded system, in which there is a requirement for transactions to commit on its first attempt, though not in a non-real-time system. For this reason, the contention manager is easily relocatable.

The second modification is applied to the acquisition of the global lock to facilitate transaction commits. In the standard implementation, a transaction attempts to acquire the lock until it is successful. However, in a preemitting real-time system with priority-driven threads, it is possible that a low-priority thread will be preempted for a higher priority thread, while it is holding onto the read lock. To account for this possibility, in the real-time version of ESTM extra data is stored along with each lock acquisition. Since the lock consists of 32 bits, and only one of which is used to declare a lock as acquired, the remaining bits can be used for additional bookkeeping information. In this case, we store the core identifier in the lock. When a transaction attempts to acquire a lock, and is unsuccessful, it checks to see whether the lock is held by a competing thread on the same core or by a thread on an opposing core. If the lock is held by a thread on the same core, it yields the processor to allow that thread to complete and release the lock. If the lock is held by a thread on the other core, it spins, waiting for the lock to be released so that it can immediately acquire it.

RT-ESTM can be used with or without the “stale-read” optimization discussed in Section 3.3; if the optimization is used we refer to the system as RT-ESTM-Optimized.
Chapter 4

Experimental Results

In this section we discuss the results of our experiments with STM on the embedded multicore platform.

4.1 Hardware Platform

We have implemented our Embedded STM in the C language on the Analog Devices Blackfin Embedded Symmetric Multiprocessor, the BF561 [4]. The BF561 comes from the low-power Blackfin family of embedded processors, but with its two cores provides higher performance than other processors in the Blackfin family. Blackfin processors feature a RISC-like instruction set for control operations, but also the dual-MAC signal processing engine necessary for DSP functionality.

The BF561 memory hierarchy features Level 1 (L1) memory which typically operates at the full processor speed, with a larger Level 2 (L2) memory which operates at a lower latency, and addressable external memory accesses via the External Bus Interface Unit.

Each BF561 processor contains two cores, each with private L1 memory and shared
L2 and external memory. The L1 memory space is divided into data and instruction memories; there is 100K bytes of L1 memory, of which 68K bytes can be used for data. Of the 68K bytes of internal L1 memory, 4K bytes is scratchpad memory, commonly used to store the system stack, and 32K bytes can be configured as cache or standard SRAM. We configure all L1 data memory as standard SRAM. Additionally, 16KB of 32KB of instruction memory can be configured as cache or instruction SRAM; we configure it as cache.

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. There are multiple independent DMA controllers to automate transfer of data from L2 and external memories to internal L1 memory in order to reduce overhead of data copies. While the DMA channels are not used in this work, they are a necessary component to future implementations. The relevant portions of the Blackfin memory hierarchy are shown in Figure 4.1.

The BF561 supports both static and dynamic voltage and frequency scaling. There are eight discrete voltage levels, varying between 0.85 V and 1.35 V, in order to support power management. Voltage levels can be lowered to the smallest value which will support the given frequency level; voltage and frequency scaling can be performed at startup time or during runtime through the use of the ADI power libraries.

We used the Analog Devices VDSP++ Kernel, or VDK [6], 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 to manage concurrency.
4.2 Target Application

The motivating application for our work was a Guidance, Navigation, and Control application, an Unmanned Aerial Vehicle (UAV). The UAV is a real-time application featuring periodic tasks running at different task rates, where the higher-rate tasks are assigned a higher priority and the lower-rate tasks are assigned a lower priority. High-priority tasks preempt low-priority tasks, as per the Rate Monotonic Scheduler (RMS)—the optimal static priority scheduler for real-time systems.

The execution of the UAV application is spent primarily on the execution of the
Kalman Filter, which recursively estimates the state of a process from noisy measurements, which may include position, velocity, rotational attributes, and acceleration. The state and error estimates and updates, as well as supporting and intermediate data, are stored as matrices and vectors and are calculated largely using common matrix and vector operations. Based on this characterization, we created two synthetic benchmarks to evaluate the performance of ESTM against the two traditional locking mechanisms, as well as a serially-executing version. The non-real-time version uses the matrix multiply as its core operation and is discussed further in Section 4.3.1. The evaluation of RT-ESTM on a real-time system features a more detailed synthetic benchmark that models the behavior of a UAV benchmark. It features combinations of matrix and vector operations resembling that of a real GNC application with I/O, Kalman filter execution, and real-time deadlines. This version of the synthetic benchmark is discussed in more detail in Section 4.4.

4.3 Evaluation of ESTM

4.3.1 Non Real-Time ESTM Benchmark Characteristics

In this section, we investigate the standard ESTM implementation, which does not include the real-time modifications. We determine the performance benefit gained from using ESTM over a sequential execution and compare it to the performance benefit of using a parallel implementation of both coarse and fine-grained locking. In these tests, we do not use preempting or priority driven threads, and there are no real-time deadlines to be met—we are merely looking to determine the speedup of a the parallel application over the serial 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 to represent a generic GNC system, and provide a thorough evaluation of our ESTM.

As discussed in Section 4.2, the computationally expensive part of a GNC algorithm is the Kalman Filter. 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 matrices is varied; since a transaction operates on at most three 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%.

### 4.3.2 Experimental Results

In this section, we compare the speedup of ESTM to locking, both fine-grained and coarse-grained. 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 use the stale-read optimization, discussed in Section 3.3, to reduce the number of memory conflicts. We vary the parameters of our microbenchmark—datasize, contention, and time spent in the critical section—for each of the concurrency mechanisms.
Varying Size of Shared Data

In the first round of tests, we compare the speedup of the three concurrency mechanisms as a function of the size of the shared data. For these tests, we hold the contention rate constant at 30%. Figure 4.2 shows the effect that datasize has on speedup over a sequential implementation for the four concurrency mechanisms.

![Figure 4.2: Speedup as a function of size of shared data objects](image)

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 4.3, 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 4.3 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
Figure 4.3: Breakdown of execution of critical section and overhead of STM calls

code) increases. While the cost of the \texttt{stm\_open\_mem} function calls increases as the square of the matrix row size due to the \texttt{memcpy}, the resulting computation increases as the cube of the row size, and the overhead of the STM calls is overshadowed by the execution of the computations. 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 (we investigate the role of the size of the critical section in Section 4.3.2).

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 4.4 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 L2. This is explained by L1 memory accesses occurring in a single cycle but L2 accesses requiring at least seven cycles [3]. 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 4.2 and Figure 4.4. 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 datasizes. 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. The advantage of ESTM is evident in Figure 4.5, which shows that many more aborts occur on the standard ESTM implementation, thus accounting for the improved performance of ESTM-Optimized.

**Varying Time Spent in Critical Sections**

Figure 4.6 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 datasize 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

Figure 4.6: Speedup as a function of percentage of execution time spent in the critical section
Figure 4.7: Transaction aborts per 1000 commits as a function of the execution time spent in the critical section

the speedup for both types of locking. Figure 4.7 confirms that increasing the time spent in a critical section results in increasing conflicts, and thus more retries in STM and blocking in the locking mechanisms. From these results, we can infer that 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, and there is little performance loss compared to fine-grained locking.

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, it is worse than even a sequential implementation. There is too much conflict between 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 the fine-grained locking is lower, the 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 a sequential execution. With speedup approximately equal for fine-grained locking and ESTM, we can safely claim that ESTM is a better option for applications if the majority of execution is spent in critical sections as they are easier to implement and more robust, but have little performance loss over fine-grained locking.

STM-Optimized performs the best of all concurrency mechanisms, as is evident in Figure 4.7, which shows the rate of transaction aborts increasing slowly. With few transactions re-running due to abort, the application is parallelized effectively, and if the stale-read model is appropriate for the application, this is the best concurrency mechanism.

Varying Shared Memory Contention

Finally, in Figure 4.8 we look at how contention between threads affects the concurrency mechanisms.

For all of these tests we use matrices of 15x15 (225 4-byte words), and 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. Also not surprisingly, fine-grained locking and ESTM both perform the best with less contention, with the benefit of parallelizing the application decreasing as more conflicts occur. Fine-grained locking and ESTM perform approximately the same across all conflict ranges. 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
Figure 4.8: Speedup as a function of probability of contention in the critical section concurrency control across all contention ranges.

4.4 Evaluation of Real-Time ESTM

4.4.1 Real-Time Benchmark Characteristics

As discussed in Section 4.2, our target application is actually a real-time application like an Unmanned Aerial Vehicle (UAV). Thus, we evaluated our RT-ESTM using a benchmark which models the behavior of the UAV system using periodic threads facing real-time deadlines. We created a synthetic benchmark which has the communication and computation patterns of the UAV application. It features functions which read input from an I/O device, model the types of matrix operations found in a Kalman filter, perform image-processing similar to a feature tracker, and output calculated data to I/O. Since the real UAV functions operate on shared matrices and vectors, we modeled the benchmark to have the same communication patterns and
shared memory accesses as UAV to accurately represent collisions to shared memory that would occur in a dual-core system.

We can classify the UAV benchmark by the benchmark characteristics studied in Section 4.3.1. The datasize of the shared objects varies from between 9 words (3x3 matrices) to very large (45x90). We can expect poorer performance on transactions using these two extremes—the transactions operating on very small shared memory objects will not have time amortize the cost of the transaction overheads, and the transactions operating on the very large datasizes will occasionally need to work out of external memory. However, those transactions operating on shared data objects which are between those two extremes will benefit from the L1 memory allocation technique of RT-ESTM and will have time to amortize the cost of the STM overheads.

The UAV spends over 95% of its time in the critical section. Therefore, we can expect coarse-grained locking to perform very poorly, fine-grained locking to perform
better, but RT-ESTM-Optimized to perform the best as it is reducing the number of conflicts by making some shared objects read-only. Finally, we can classify UAV as having medium-to-high contention. There are 15 shared objects; two transactions access one shared object, two transactions access two shared objects, and the remaining 5 transactions access five or six shared objects. Thus we can expect coarse grained-locking to perform very poorly, fine-grained locking and standard RT-ESTM to perform approximately the same, but RT-ESTM-Optimized to perform the best because it reduces the probability of conflict in the critical section by marking some objects as read-only.

4.4.2 Load Balancing Across Cores

An integral part of the conversion from a single to dual-core embedded implementation is properly assigning threads to cores such that the load is balanced across cores and communication between cores is minimized to reduce the number of shared-memory collisions. Weighted graph partitioning provides an intuitive way to represent the problem to find a good assignment of threads to cores. In the weighted graph partitioning problem, nodes represent threads and edges represent shared memory conflicts. Specifically, the problem is defined as follows:

Given $N$ threads and $M$ cores, statically allocate threads to cores such that the weight of all conflicts between threads is minimized and the CPU utilization is balanced across all cores. The threads have the following properties:

- All threads are periodic, having a period of $T$. The $i$th iteration of a thread has a deadline equal to $iT$.
- All threads are assigned a priority based on their rate of execution, and a higher
rate task has a higher priority than a lower weight task.

- The least-common multiple of the periods of all the tasks is called the hyperperiod.

The problem can be modeled as a graph $G = (V, E)$, where $V = \{t_1, t_2, \ldots, t_N\}$, the percent of execution time of one hyperperiod which is spent in the $n$th thread, and $E = \{c_1, c_2, \ldots, c_N\}$, the set of weights of all conflicts between threads. A conflict is defined as the percent of time during one hyperperiod that two threads may be operating on the same shared data, where at least one thread is writing to that shared data.

As there are only 9 threads and 2 cores, and therefore only 127 possible combinations of allocations of threads to cores, we wrote an exhaustive algorithm to find all possible solutions. The input to this program is a representation of the graph and two ranges (one specifying the acceptable range of total conflicts across cores, the other specifying the acceptable load imbalance), and outputs all solutions which fall within those ranges. We then selected the partition which resulted in the balanced load with the least conflict between cores.

4.4.3 Experimental Results

Performance Impact

We ran four versions of the UAV benchmark in order to evaluate the performance and power characteristics of the benchmark. We created a serial version of the benchmark to run on a single core; since the UAV benchmark features pre-empting tasks operating on the same set of data, we protected critical sections using fine-grained locking. For the parallel version, we divided the threads across cores using the load-balancing
algorithm described in Section 4.4.2, and protected access to shared memory using coarse-grained locking, fine-grained locking, and Real-Time ESTM.

The goal in this periodic real-time application is not to make the application run as quickly as possible, but rather to set deadlines which each thread has to meet at each period. Specifically, the high-rate tasks must complete 100 times per second and the low-rate tasks must complete 10 times per second. Therefore, instead of measuring performance in terms of speedup, we will look at the minimum clock frequency that the processing cores can be lowered to which will still allow threads to meet their real-time deadlines. Thus, for each project, we lowered the clock frequency as far as possible to save power but still require that threads meet their deadlines.

The serial version of the UAV benchmark can safely run at 500 MHz; at this point, the core is idle 7.2% of the time, but any slower clock frequency results in threads missing their real-time deadlines. The coarse-grained locking benchmark actually performs worse than the serial benchmark, as its clock frequency cannot go lower than 550 MHz. This result is not surprising given the benchmark characteristics and the results of Section 4.3.2. Section 4.3.2 showed that coarse-grained locking will perform worse than serial execution when the probability of conflict in the critical section is high, and when over 90% of the execution time is spent in the critical section. These are both characteristics of the UAV benchmark.

As expected, fine-grained locking and standard RT-ESTM perform about the same, with fine-grained locking able to safely operate at 450 MHz and standard RT-ESTM at about 400 MHz. The RT-ESTM is able to slightly outperform fine-grained locking due to the effective use of L1 memory, given that most shared data can be stored in L1 for operation. The speedup achieved is appropriate given the results of Section 4.3.2, given an application with high conflict rates and large amounts of
execution time spent in the critical section.

Not surprisingly, RT-ESTM-Optimized performs the best due to the elimination of conflicts by marking memory accesses as read-only. RT-ESTM-Optimized can run safely at 350 MHz.

The additional benefit of RT-ESTM is the management of task priorities. With traditional locking, the first task to acquire a lock holds on to the lock until it has finished execution—another thread with a higher priority may be blocked waiting for the same lock, or a thread may starve indefinitely waiting for all the locks it needs. With RT-ESTM, this problem is reduced. All transactions are allowed to run, and then the conflict manager can determine which transaction should commit based on priority or starvation (represented by the num_aborts member variable). This results in a fairer execution than locking can provide.

4.4.4 Impact on Power Consumption

Since a given clock frequency requires a specific core voltage level, reducing the clock frequency allows us to also reduce the core voltage. Since power is proportional to both clock frequency and supply voltage, the power savings is greatly reduced by improving the performance of an application.

By using the reduced clock frequencies determined in Section 4.4.4, we can determine power savings for each parallel version of the application compared to the sequential single-core version.

We approximated the power consumption of the application using the Analog Devices spreadsheets from the white paper “Estimating Power for ADSP-BF561 Blackfin Processors” [5]. These calculations determine the power consumption of the processor’s core logic, including both dynamic power (which is a function of both supply
voltage and clock frequency, as well as instructions executed) and static power (which is a function of temperature and supply voltage, but not related to processor activity or clock frequency). The static power consumption is calculated based on a 100°C junction temperature.

Unfortunately, there is a challenge in representing the power consumption of the serial version of the benchmark. It is unfair to use the power consumption data for the BF561 as-is, with processor activity on both cores, since the second core is not in use. For this reason, we present two sets of data for the sequential benchmark (and thus two sets of numbers relating power consumption to those of the sequential benchmark). The first is a lower bound on expected power consumption of the serially-executing benchmark and attributes half of all power consumed to each core, and thus assumes that a single-core implementation would use half the estimated power consumption for the whole processor. This lower bound is derived by taking both static and dynamic power consumption values and dividing by 2 (the number of cores). The resulting power savings based on this approximation for serial execution is presented in Table 4.1.

<table>
<thead>
<tr>
<th></th>
<th>(V_{DD}) (V)</th>
<th>(I_{DYN}) (mA)</th>
<th>(I_{STAT}) (mA)</th>
<th>(P_{TOT}) (mW)</th>
<th>(%_{REL_TO_SER})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>1.19</td>
<td>196</td>
<td>214</td>
<td>487</td>
<td>100</td>
</tr>
<tr>
<td>Coarse</td>
<td>1.19</td>
<td>384</td>
<td>428</td>
<td>964</td>
<td>198</td>
</tr>
<tr>
<td>Fine</td>
<td>1.05</td>
<td>297</td>
<td>334</td>
<td>660</td>
<td>135</td>
</tr>
<tr>
<td>RT-ESTM</td>
<td>0.95</td>
<td>235</td>
<td>284</td>
<td>493</td>
<td>101</td>
</tr>
<tr>
<td>RT-ESTM-Opt</td>
<td>0.86</td>
<td>179</td>
<td>243</td>
<td>361</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 4.1: Potential power savings of the parallelized applications, relative to the lower bound of power consumption of the serial implementation.

The reason this value is presented as a lower bound is because it assumes that
Table 4.2: Potential power savings of the parallelized applications, relative to the upper bound of power consumption of the serial implementation.

<table>
<thead>
<tr>
<th></th>
<th>$V_{DD}$ (V)</th>
<th>$I_{DYN}$ (mA)</th>
<th>$I_{STAT}$ (mA)</th>
<th>$P_{TOT}$ (mW)</th>
<th>$%_{REL_TO_SER}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>1.19</td>
<td>196</td>
<td>428</td>
<td>741</td>
<td>100</td>
</tr>
<tr>
<td>Coarse</td>
<td>1.19</td>
<td>384</td>
<td>428</td>
<td>964</td>
<td>130</td>
</tr>
<tr>
<td>Fine</td>
<td>1.05</td>
<td>297</td>
<td>334</td>
<td>660</td>
<td>89</td>
</tr>
<tr>
<td>RT-ESTM</td>
<td>0.95</td>
<td>235</td>
<td>284</td>
<td>493</td>
<td>67</td>
</tr>
<tr>
<td>RT-ESTM-Opt</td>
<td>0.86</td>
<td>179</td>
<td>243</td>
<td>361</td>
<td>49</td>
</tr>
</tbody>
</table>

the power consumed by all components is split between both cores. However, only some of the core logic is duplicated; more of the processor logic, including L2 SRAM, timers, and emulator logic, is not duplicated and thus the power consumption for one core is more than half of total power consumption. To satisfy this, we also compute an upper bound on power consumption for the single-core implementation. For this calculation, dynamic power is assumed to still be attributable to Core A, as it is the only core actively switching, however the static power consumption is not divided by two. This results in higher nominal power consumption for the serial benchmark, so by comparison it results a greater-than-expected power savings for the parallel benchmarks relative to the serial benchmark. The upper bound for possible power savings is presented in Table 4.2.

Tables 4.1 and 4.2 show that parallelizing the application using coarse grained locking results in higher power consumption than running it in serial on a single core. This is because the performance is not improved by running it on two cores with coarse-grained locking, but both cores are consuming power to do the same work as was done with the serial version. Thus, the power consumption for the parallel benchmark is actually 1.98x to 1.3x the power consumption of the serial
application. The fine-grained locking application performs slightly better, consuming at best 0.9x the power and at most 1.3x the power of the serial benchmark. Again, the cost of powering two cores overcomes the performance advantage of running the application in parallel. However, using the Optimized STM, eliminating read-only conflicts, the performance gain from the increased parallelism is enough to offset the power consumption of the second core. For RT-ESTM-Optimized, the power consumption is 0.5x to 0.74x the power consumption of the single-core application. These results show that parallelizing an application using RT-ESTM provides can result in both increased performance or decreased power consumption.
Chapter 5

Discussion

The usage of transactional memory in the embedded system design space is a new one, both because of the relative novelty of software transactional memory in the general-purpose design space, but also because embedded systems manufacturers have only recently begun producing homogeneous multicore processors. In this chapter, we discuss some challenges in the development of ESTM, the current status of the project, and how we see STM fitting into the embedded design space in the future.

5.1 Challenges

The implementation of an embedded STM required solutions to unique challenges.

A common tradeoff encountered during the design phase was between making the implementation generic enough to work on a variety of embedded platforms and applications, and keeping the overhead low. For example, the choice to make it a static STM may be problematic in its acceptance, as recent implementations of general-purpose STMs are dynamic. However, STMs on a general purpose processor must necessarily be dynamic: they must support a variety of applications, including
those using dynamic-sized data structures. However, this comes at an overhead in the STM internals, and in many cases may be unnecessary for the embedded application. For example, our GNC application did not require the use of dynamic sized data structures like linked lists and trees, and thus the overhead of a dynamic STM was not worth the benefit.

Another tradeoff between generality and speed is in the conflict manager. In the general case, the fastest time to invoke the contention manager is at transaction abort; since aborts are relatively rare, the extra code will not often be executed, and contention management is likely only needed after consecutive aborts. However, this is impractical for real-time applications, in which it might be necessary for a transaction to commit each and every time it runs. In this case, invoking the contention manager at abort is already too late: contention management needs to performed in the setup phase, at the start of every transaction, before the transaction executes original application code. Thus, an effort to be more general (to support all types of applications, including real-time) causes an overall slowdown in the average case. These types of tradeoffs were common, and the tradeoffs between making the ESTM generic for many applications or fast for a specific application were not always obvious. For this reason, there is room for more evaluation and interpretation.

A big implementation challenge was in the locking mechanisms required for concurrency control in the STM internals. The decision to use locking in the internals is a difficult one because, at first glance, it seems to contradict the goal of eliminating traditional mutual exclusion from parallel programs. Fortunately, past research has shown that that the use of locks in the STM internals can greatly simplify management of the transactions and reduce overhead.
The introduction of locks in the STM internals does introduce some of the same design questions that are asked in application parallelization using traditional locking. Specifically, lock granularity and deadlock-prevention mechanisms must be investigated and implemented. The use of fine-grained locks to protect individual data introduces its own problems, and STM implementations which use fine-grained locks must rely on two-phase locking or acquiring the locks in order to prevent deadlock [24, 60]. However, there is still an advantage over traditional fine-grained mutual exclusion. The deadlock prevention strategy only needs to be implemented once (within the STM), so the application programmer does not need to test and debug for deadlock in every application which uses it. Further, there is a possible failover solution in the rare event that a deadlock is encountered: An offending thread can be simply aborted and restarted.

Using coarse-grained locks within the STM internals presents its own advantages and disadvantages as well. A global lock, such as that used in our ESTM, is simple to implement and error-free, and as an added benefit, provides a consistent view of memory for transactions to operate on. This eliminates the need to periodically “validate” a transaction’s read-set, as must be done in STM implementations which do not utilize a global lock [24]. The disadvantage of the global lock is clearly increased contention, though the lock is actually held for a short period time relative to the execution of original code so it is still an improvement over traditional coarse-grained mutual exclusion. However, without evaluating ESTM on embedded processors with more than two cores, it is difficult to claim the global lock as the superior implementation choice without investigating its scalability.
5.2 Current Status

Currently, ESTM is a fully functional software transactional memory implementation which can, in some cases, beat the performance of fine-grained locking. However, given that it is tailored to a specific class of applications, certain assumptions are made about the application which can be relaxed in future implementations. First of all, it assumes a *closed memory allocation model*. This means that memory allocated and freed by a transaction cannot be reused by nontransactional code. Consider the case of a transaction allocating memory in its body using the `malloc` call: If the transaction aborts, it will re-execute and potentially re-allocate the memory, resulting in a memory leak. The problem also arises in the use of the `free` call: If a transaction frees memory while other threads continue to access the memory, either an error will occur due to accessing invalid memory, or worse, the memory will have been recycled and the data will be silently corrupted. A simple way to get around this is through specialized `malloc` and `free` calls, but again, it creates an extra overhead in the average case to manage the heap-allocated memory, while a specialized embedded application (such as the GNC algorithm) may rely exclusively on statically-allocated data such as arrays.

ESTM allows data to be accessed in one of only two ways: *read/write* and *stale-read*. In the stale-read case, as discussed in Section 3.3, read-only data modified by another transaction will not cause a conflict or transaction abort. A read/write access is safe for all accesses, but performs a copy at the end of the transaction assuming that the data has changed. The introduction of a third state, a *read-only* state, will not provide the optimization of a stale-read (that is, read-only data modified by another transaction will result in a conflict and abort). The addition of this third state will save execution time by not performing a copy at the end of the transaction, and is
safe enough to use on all applications.

An interesting possibility for improvement is the use of DMA to perform the memory copies both in the transaction setup and commit phases. By moving the copy to the background, instead of utilizing valuable processing cycles, this will help amortize the overhead of the STM calls, allowing ESTM to be a viable alternative in applications with shorter transactions.

5.3 Usage

Transactional memory is a controversial solution to the shared memory management problem in general purpose systems, despite ongoing research in the field since 1993. Some researchers think that threading in general is impractical for future many-core processors and we need to develop an entirely different paradigm for parallel execution. Thus, transactional memory provides a solution to a problem which, to some, should be eliminated altogether. Even among transactional memory supporters, there is little consensus on the best means of implementation. Generally, bounded HTM systems are considered to be impractical for use because they require too much underlying knowledge of the architecture in order to ensure transactions don’t exceed resources. Unbounded HTM systems are unlikely to be adopted because of the high additional hardware cost, on top of the cost of the HTM itself. STM systems, on the other hand, present more of a chicken-and-egg problem. Programming language developers are wary of building a language around transactional memory primitives until an implementation can be agreed upon, but library based STMs (with or without compiler support) are still too cumbersome to be considered practical solutions and thus no one can agree on the implementation details.
Given the difficulty in finding acceptance for transactional memory in general purpose systems, it at first seems unlikely that it will be accepted in embedded systems. With the tight area and power budget, an HTM solution is not likely to be implemented in the near future. However, there still may be place for STM in the embedded design if it is implemented in the same way that many other embedded decisions are made: through customization. Because embedded applications are inherently special purpose, the solution to acceptance of STMs in embedded systems would require customized implementations. By allowing the STM to maintain assumptions about the application, and not requiring that it work for all classes of applications, it can remain simple and result in a low overhead. In this manner, it can remain simple to use, while meeting or beating the performance of fine-grained locking, and thus may have a chance to succeed as a viable alternative to traditional locking.
Chapter 6

Contributions and Future Work

In this chapter we discuss the contributions of this thesis and present areas of future work.

6.1 Summary of Contributions

Parallel execution of an application on multiple symmetric cores provides an opportunity for performance improvement and/or power consumption reduction. However, shared memory requires protection from concurrent access by different threads. Traditionally, this protection has been provided through mutual exclusion in the form of locks. Transactional Memory has been proposed as a solution, but has thus far only been investigated in high-performance and general-purpose computing. With embedded chip manufacturers now turning to symmetric multicore processors, there is a new motivation to find new concurrency mechanisms for parallel embedded applications. The stated goal of transactional memory is to achieve a performance equal to parallelism with fine-grained locking, but with the programmer effort and robustness of code of coarse-grained locking.
The primary goal of this thesis was to demonstrate the feasibility of Software Transactional Memory in the multicore embedded design space. To that end, the contribution of this thesis are as follows:

- We evaluated six key design points for an embedded implementation: Static vs. dynamic STM, granularity of conflict detection, blocking vs. non-blocking, object acquisition time, write-buffering vs. undo-logging, and starvation prevention.

- We presented three unique advantages that ESTM has over traditional locking. The first is a novel use of L1 memory; since transaction memory with lazy versioning requires a copy be made of shared data, we bring that copy into L1 memory, effectively utilizing L1 as a software cache. Executing code using data from L1 memory provides at minimum an 8% speedup over data in L2 memory (with an ever greater benefit provided over L3 memory). This effectively increases the capacity of the L1 memory, while the L1 memory may go largely unused in an application using traditional locking. The second optimization is allowing transactional accesses to be marked as “stale reads”. Due to the use of a global read counter, all data to be used in a transaction is guaranteed to be consistent, so if it acceptable for a transaction to operate on data which is slightly outdated, it may safely do so without violating memory consistency. In this case, conflicts between transactions are reduced, and thus there are fewer aborts and retries to use up execution time. Finally, the introduction of a “Contention Manager” introduces fairness to transactions and allows higher-priority or starving transactions to commit ahead of low-priority transactions.

- We implemented ESTM using the findings of our evaluation of STM design
decisions, and presented the details of the baseline ESTM implementation. We demonstrated the four simple functions of its API and explained the operation of ESTM. We presented optimizations which can help performance and modifications to make it compatible with real-time applications, and called implemented these features for our Real-Time ESTM (RT-ESTM).

- We created two synthetic benchmarks simulating a range of behaviors of Guidance, Navigation, and Control systems. One featured a synthetic matrix multiply with varying application characteristics, the other featured periodic threads with soft real-time deadlines with differing priorities, in which high-priority threads preempted low-priority threads.

- We demonstrated that ESTM approached the performance of fine-grained across most of the studied benchmark characteristics including size of shared data, amount of time spent in the critical section, and probability of memory conflict. In our synthetic benchmark, ESTM maintained a near-optimal 2x speedup in our benchmark across a range of application characteristics. Even with 99% of time spent in the critical section—when contention over shared resources is very high—it still achieved 1.5x speedup, on par with fine-grained locking. ESTM also maintained the performance level of fine-grained locking when varying the probability of contention in the critical region, achieving more than 1.4x speedup while the probability of conflict is less than 20%. When allowing optimizations for stale reads, ESTM vastly outperformed fine-grained locking across all application characteristics studied in our benchmark. Excepting the case of small data size (25 words), Optimized ESTM achieved at least a 1.6x speedup over sequential execution across the range of studied application characteristics.
• We evaluated our RT-ESTM on a real-time application modeled after a real GNC application, the Unmanned Aerial Vehicle. We demonstrated how to effectively balance the load across cores using a *Weighted Graph Partitioning* representation. We showed that the application was most effectively parallelized using RT-ESTM over traditional locking.

• We demonstrated that, if the goal was not reduced performance but reduced clock frequency at the same performance level, RT-ESTM could result in a lower clock frequency than either form of locking. The reduced clock frequency allowed for reduced power consumption; optimized RT-ESTM was found to reduce power consumption by more than 26%.

6.2 Future Work

There are many opportunities for future work on Embedded Software Transactional Memory. First, improvements can be made to the model to reduce overhead. This could include the inclusion of a third state of declaration for memory accesses, which is that they are read-only but cannot be stale-reads, or a more advanced L1 memory allocation strategy. These improvements could expand the usefulness of ESTM to applications which have small transactions. Additionally, a common next-step for library-based STMs is to provide compiler support to reduce programmer effort and provide automatic optimizations. More research into real-time modifications will additionally strengthen the case that STM is a good candidate for applications with real-time deadlines. Finally, we can investigate the scalability of ESTM by evaluating ESTM on a processor with additional cores, beyond the dual-core processor studied in this work.
Appendix A: ESTM Pseudocode

```c
stm_start_setup(Transaction* trans, void** local_address_book,
                 void* global_address_book, int address_book_size,
                 int max_aborts)
{
    /* Initialize transaction struct variables */
    trans->tid = get_trans_id(thread_id);
    trans->read_only = READ_WRITE;
    trans->max_aborts = max_aborts;
    trans->num_accesses = 0;

    /* Increment the read counter to prevent transactions from committing */
    acquire_lock(&read_lock);
    read_counter++;
    release_lock(&read_lock);

    /* The address of the local address book pointer (ie, a double pointer)
     * is passed in. Dereference the pointer to malloc the actual local
     * address book. */
    (*local_address_book) = malloc(address_book_size);
    memcpy((*local_address_book), global_address_book, address_book_size);

    /* Save this information in Transaction struct for use in stm_end() */
    trans->local_address_book = *local_address_book;
    trans->shared_address_book = shared_address_book;
    trans->booksize = address_book_size;
}
```

Figure 1: Pseudocode for function `stm_start()`. This function is called once per transaction.
stm_open_mem(Transaction* trans, void** address, int size_bytes, int read_only)
{
    access_id = trans->num_accesses;
    addr_shared = *address;
    trans->rw_set_mods[access_id] = read_only;
    trans->rw_set_shared[access_id] = *address;

    /* Now malloc the local copy to the lowest level of memory which is available */
    addr_local = heap_malloc(L1_HEAP, size_bytes);
    if(addr_local == NULL)
        /* Not enough room in L1, try L2 */
        addr_local = heap_malloc(L2_HEAP, size_bytes);
    if(addr_local == NULL)
        /* Not enough room in L2, try SDRAM */
        addr_local = heap_malloc(SDRAM_HEAP, size_bytes);

    /* Copy the data to the new private location */
    memcpy(addr_local, addr_shared, size_bytes);

    /* Fill in transaction struct information */
    trans->version_numbers[access_id] = get_current_version(addr_shared);
    trans->rw_set_local[access_id] = addr_local;
    trans->rw_set_sizes[access_id] = size_bytes;
    trans->num_access++;

    /* Repoint original pointer to new private address */
    *address = addr_local;
}
stm_end_setup(Transaction* trans){

    if(IS_REAL_TIME)
        /* Detect conflicts here, at the start of the transaction */
        if(trans->num_aborts >= trans->max_aborts)
            prevent_commits(trans);

    acquire_lock(&read_lock);
    read_counter--;
    release_lock(&read_lock);

    /* Transaction is read only if all transactional accesses are read-only */
    for(idx = 0; idx < trans->num_accesses; idx++)
        trans->read_only = trans->read_only & trans->rw_set_mods[idx];
}

Figure 3: Pseudocode for function \texttt{stm\_end\_setup()}. This function is called once per transaction.
stm_end(Transaction* trans) {
    if(trans->read_only)
        committed = TRUE:

        /* No need to memcpy back to shared memory because all access are by
         * definition read only and thus weren’t modified */

        /* Free local memory */
        for(idx = 0; idx < trans->num_accesses; idx++)
            free(trans->rw_set_local[idx]);

    else
        ok_to_update = wait_for_zeroed_read_counter();

        /* Can only get this far when no other transaction is in setup phase */
        can_commit = check_versions(trans);

        if(can_commit){
            committed = TRUE;
            commit_transaction(trans);

        } else
            committed = FALSE:
            abort_transaction(trans);

            /* For correct stack maintainence, setjmp is called in calling
             * function */
            longjmp(trans->env);
}

Figure 4: Pseudocode for function stm_end(). This function is called once per trans-
action.
commit_transaction(Transaction* trans) {

    reset_num_aborts();

    increment_versions(trans);

    /* Copy data back to shared memory */
    for(idx = 0; idx < trans->num_accesses; idx++)

        /* No need to copy if read_only */
        if(trans->rw_set_mods[idx] != READ_ONLY)
            memcpy(trans->rw_set_shared[idx], trans->rw_set_local[idx],
                   trans->rw_set_sizes[idx]);

    allow_new_transaction_setup();

    /* Check to see if this transaction was blocking commits, because now it has
    * committed so it is safe for others to commit */

    is_blocker = check_if_blocking_commits(trans->tid);
    if(is_blocker)
        unblock_commits(trans->tid);

    for(idx = 0; idx < trans->num_accesses; idx)
        free(trans->rw_set_local[idx]);

    free(trans->local_address_book);
}

Figure 5: Pseudocode for function commit_transaction(), called by stm_end()
abort_transaction(Transaction* trans) {

    /* Not committing after all, so allow other transactions to begin */
    allow_new_transaction_setup();

    increment_aborts(trans->tid);

    if(!IS_REAL_TIME) {
        /* Invoke contention manager here */
        trans_aborts = get_num_aborts(trans->tid);
        if(trans_aborts >= trans->max_aborts)
            prevent_commits(trans);
    }

    for(idx = 0; idx < trans->num_accesses; idx)
        free(trans->rw_set_local[idx]);

    /* Now do the work of stm_start */
    acquire_lock(&read_lock);
    read_counter++;
    release_lock(&read_lock);

    trans->num_accesses = 0;
    trans->read_only = READ_WRITE;
    memcpy(trans->local_address_book, trans->shared_address_book,
            trans->booksize);
}

Figure 6: Pseudocode for function abort_transaction(), called by stm_end()
Bibliography


