USING SOFTWARE TRANSACTIONAL MEMORY IN INTERRUPT-DRIVEN SYSTEMS

by

MICHAEL J. SCHULTZ, B.S.

A Thesis submitted to the Faculty
of the Graduate School, Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Milwaukee, Wisconsin, USA

May 2009

Abstract

Transactional memory presents a new concurrency control mechanism to handle synchronization between shared data. Dealing with concurrency issues has always been a difficulty when writing operating system software and using transactions aims to simplify matters. This thesis presents a framework for understanding how interrupt-driven device drivers can benefit from using transactional memory. A method for integrating software transactional memory (STM) into an operating system kernel is also developed and applied. This kernel uses STM over hardware transactional memory (HTM) because HTM requires modifications only implemented in simulated systems. By using STM, it is possible to build upon existing kernels and deploy operating systems onto commodity machines with communication peripherals.

At the core is a modernized version of the Embedded Xinu operating system that has been ported to the Intel IA-32 architecture and modified to use a publicly available, production quality compiler and STM library available from Intel Corporation. The implementation of the Embedded Xinu kernel required several modifications to make use of the transactions offered by a library designed for use with user-level threads executing in Linux. Integrating the STM library into the kernel presents several challenges when dealing with a system that allows interrupts to enter at any time. By using transactions in device drivers, synchronization can be performed automatically and with greater granularity than traditional synchronizations methods. This may be used to to reduce the jitter—variations in interrupt handling—occurring in interrupt-driven device drivers when sharing data between the upper and lower halves, however this implementation does not show any conclusive results.

This thesis discusses and presents the prototype implementation of "Transactional Xinu." The prototype runs on standard hardware that exists today and provides a framework for future experimentation.

Acknowledgments

Many thanks are due to the people who have been involved with me and this thesis over the past few years, so I would like to give thanks to the following people:

- Dennis Brylow, my imperious leader, a great researcher, friend, and advisor who gave me the opportunity to work on this interesting project;
- Praveen Madiraju and Craig Struble, my two committee members from Marquette who succeeded in pushing me to finish on time;
- Adam Welc, my external committee member who, despite living two timezones away, provided crucial support in integrating Intel's STM library with Embedded Xinu;
- Timothy Blattner, Aaron Gember, Paul Hinze, Adam Koehler, Zachary Lund, Adam Mallen, Mohammad "Meraj" Molla, Justin Picotte, and everyone else affiliated with the systems laboratory for listening to me and providing 100% awesome entertainment;
- My parents, Charles and Theresa, for offering support when I needed it; my sister and brother-in-law, Jennifer and Stefan, for giving me a place to sleep when I needed a break; and the rest of my family for not mocking me too much as I continue my educational endeavors; and finally,
- Lyndsie Schwanebeck, my fiancée and best friend, for her understanding when I worked late, constant support, and persistent belief that I could finish successfully and on time.

Contents

List of Figures

List of Terminology

2PL	**Two-Phase Locking**—A locking protocol that ensures the order of lock acquisition/release cannot cause deadlock.
ACI	**Atomicity, Consistency, and Isolation**—A subset of the ACID properties that TM system provide.
ACID	**Atomicity, Consistency, Isolation, and Durability**—Properties that ensure a transaction is executed reliably.
ANSI	**American National Standards Institute**—Defines standards for various products, such as the C programming language.
API	**Application Programming Interface**—A set of routines provided by the operating system and software libraries.
BIOS	**Basic Input/Output System**—First stage of booting a personal computer, can run very basic services.
CAS	**Compare-and-Swap**—An operation that atomically compares a single known value with the stored value and swaps in a new value if there is a match.
DCAS	**Doubleword Compare-and-Swap**—A CAS operation that works on two machine words.
DHCP	**Dynamic Host Configuration Protocol**—A mechanism to automatically configure a network interface.
DMA	**Direct Memory Access**—A mapping of physical memory addresses that is shared between the operating system and hardware devices.
ELF	**Executable and Linkable Format**—Standard format for storing binary files.
HTM	**Hardware Transactional Memory**—Transactional memory that is implemented entirely in physical hardware.
IA-32	**Intel Architecture, 32-bit**—The 32-bit version of Intel's standard architecture, sometimes called x86, x86-32, or i386.
ILS	**Interrupt-Local Storage**—A private block of memory that every interrupt handler can access.
I/O	**Input/Output**—Communication between a computer and external manipulator.
LL/SC	**Load Linked/Store Conditional**—A set of opcodes that guarantees an atomic update of a memory location.
micro-op	**Micro-operation Cycle**—A subdivision of an operation that the processor recognizes.
opcode	**Operation Code**—A single machine language instruction that specifies what operation to perform.
O/S	**Operating System**—The interface between hardware and application level code.

POSIX	**Portable Operating System Interface for Unix**—A collection of operating system standards, available through an API.
SGLA	**Single Global Lock Atomicity**—A transaction semantic that allows a transaction to operate as though there is a single global lock in the system.
SIMD	**Single Instruction, Multiple Data**—Allows a streaming processor to execute a single instruction over many separate pieces of data, in parallel.
SSE	**Streaming SIMD Extensions**—An additional set of instructions on the IA-32 architecture that enables vector operations.
STM	**Software Transactional Memory**—Transactional memory that is implemented entirely in software libraries.
TFTP	**Trivial File Transfer Protocol**—Simple tool to send files across a network.
TLS	**Thread-Local Storage**—A private block of memory that every thread can access.
TM	**Transaction Memory**—A concurrency control mechanism that allows a set of memory instructions to occur in an atomic fashion.
TSC	**Timestamp Counter**—A performance measure on IA-32 that counts the number of micro-operation cycles since the processor reset.

Chapter 1

Introduction

1.1 Thesis Statement

Software transactional memory can be used in interrupt-driven device drivers as a method to automate and provide fine-grained synchronization between the upper and lower halves.

1.2 Overview

In this thesis, a discussion of using *software transactional memory* (STM) in interrupt-driven device drivers is presented. This is substantial because software transactional memory has not (to this author's knowledge) been used at the operating system level before. One of the primary goals of this research is to implement a prototype operating system augmented with STM on actual hardware that can be purchased at the time of writing. Prior work has used *hardware transactional memory* to manage concurrency at the operating system level. These works, while novel, make use of hardware simulators and still rely on spinlocks in certain instances. One difficulty that arises from using transactions in interrupt-driven systems is non-linear code execution, where the operating system can arbitrarily change the code that the processor is executing.

This research presents three major outcomes. First, there is a discussion of how operating systems can benefit from applying a software transactional memory (STM) library to interrupt-driven device drivers. Transactions were developed with multi-core systems in mind, but their properties also present a natural union with device driver structure in uniprocessor systems. Sec-

ond, there is an explanation of methods for integrating an existing Linux based STM library into a non-Linux operating system kernel and drivers. While STM introduces a new set of problems to operating system design, it can minimize the difficulty in thinking about device drivers while increasing the scalability of operating system code as computing moves toward multi-core processors. Finally, there is a presentation of the performance results of using software transactional memory in an interrupt-driven device driver versus the non-transactional equivalent. These results demonstrate that transactional device drivers can be used without a significant impact to performance times.

The overall structure of this document is as follows.

- Chapter 1 is an introduction to the topics presented in this document and outlines the contributions of this work.
- Chapter 2 provides an overview of related work.
- Chapter 3 presents a framework for handling atomic sections in normal execution mode, interrupt handling, and input/output (non-revocable) portions of the O/S.
- Chapter 4 details the specifications and implementation process for "Transactional Xinu."
- Chapter 5 provides results and a discussion of the performance of the prototype system.
- Chapter 6 summarizes this work and discusses future research and contributions.

1.3 Operating Systems

An *operating system* (O/S) is software that runs at the lowest level of a computer once the system has completed the boot process. This software consists of two important sections: device drivers and the application programming interface.

A *driver* is the software that facilitates interaction between high-level O/S functions and low-level hardware communications. As such, the driver is broken down into two parts: the upper half and the lower half. The lower half interacts directly with the physical hardware and the upper half presents a standard programming model for an application developer to use to interact with the device. Drivers must communicate incoming data from the lower half to the upper half and vice versa for outgoing data.

The *application programming interface* (API) provides the standard high-level methods that an application programmer can use as a means to interface with the O/S, to interact with a user of the system, or to obtain data that has been generated for the application to use.

1.3.1 Interrupt-Driven Systems

One class of operating systems is called *interrupt-driven* systems, which includes almost every modern system from desktop computers to embedded routers to computerized thermostats. These are capable of receiving signals from outside the processor and causing the operating system to change the current path of execution to the code for handling interrupts. The operating system is able to ignore an incoming interrupt if the O/S is processing something that cannot be interrupted. However, the incoming interrupt request may be more important than the non-interruptible process the O/S is currently executing, and this behavior can lead to a crucial interrupt handler being deferred for an arbitrary amount of time.

In operating system terms *jitter* refers to the variations in the delay of interrupt handling caused by deferral. Jitter should be avoided in systems programming because it adds unpredictable behavior to the system. Unpredictable behavior is particularly troublesome in *real-time* systems, when results must be achieved within some fixed amount of time; jitter in interrupt handling may cause a necessarily predictable system to miss deadlines.

One common avoidance method is to minimize the length of time and/or amount of code that must disable interrupts in the system. This takes the time and effort of a "good" systems programmer who may still make errors in situations that are difficult to reason about or in a sufficiently complex system. Since the definition of a "good" systems programmer is loose, computer scientists aim to find more concrete solutions to these problems to reduce programmer errors. One obvious solution is to simply never disable interrupts in the O/S, but this can cause problems in certain cases; the simple piece of code in Figure 1.1 exemplifies an error that is likely to occur with that approach.

On line (1), the programmer declares the variable `our_var` to be global (i.e. it can be modified by any process at any time). On line (4), that variable is then incremented by 1. From a high level, this function may look safe. However, if the corresponding machine code (Figure 1.2) runs on a multi-threaded or multi-processing O/S, there is the possibility of a contention error. This

```
( 1) global integer our_var = 1
( 2)
( 3) function increment(void)
( 4)     our_var++
```

Figure 1.1: Example of a potentially unsafe memory access

```
( 1) .data
( 2)     .globl  our_var
( 3) our_var: 0x0001          # stored at memory location 0x4c
( 4)
( 5) .text
( 6) increment:
( 7)     lw    s0, 0x4c(zero) # load 'our_var' into register 's0'
( 8)     addiu s0, s0, 1      # increment value in 's0' by 1
( 9)     sw    s0, 0x4c(zero) # save register 's0' to 'our_var'
(10)     j     ra             # return to calling functions
```

Figure 1.2: Code from Figure 1.1 in MIPS assembly

error is referred to as a "race-condition" where two separate instances of the code try modifying the same variable at the same time with unpredictable results.

In line (3), `our_var` is created with a (fictional) location of `0x4c` in main memory. In order to perform the increment operation, the processor must load the variable from main memory into a register, increment the register, and finally store the updated value back into main memory. In the interrupt-driven paradigm, if an interrupt is raised anywhere between lines (7) and (9), it is possible that the value stored in location `0x4c` has changed. If this happens when the code from Figures 1.1 and 1.2 complete, the value saved in memory is not correct.

1.4 Contributions

This thesis provides a discussion about using software transactional memory in the interrupt-driven device drivers of an operating system. Software transactional memory (STM) provides a simple interface for handling concurrency issues between separate threads of execution while still allowing interruptions. Adding software transactions to an operating system allows the exploration of new

transactional technology in combination with existing operating system constructs with the goal of finding a balance between the two. Specifically, by using STM in device drivers, it may be possible to reduce jitter introduced by entering critical sections of code that have disabled interrupts. This allows the arrival time of interrupts into the system to be more predictable—a very useful property when dealing with system deadlines.

This theory is examined by developing a method for integrating a publicly available, production quality STM library—designed for Linux—into a small embedded operating system. This author ports Embedded Xinu—a modernized revision of the Xinu kernel—to the IA-32 architecture and augments the necessary components to transform the system into "Transactional Xinu." Transactional Xinu is a proof-of-concept prototype that is designed to actually run on real hardware and provides a strong framework for ongoing and future experiments on integrating transactional memory with core operating system components.

Finally, there is a direct evaluation of jitter reduction in asynchronous device drivers using STM-based language constructs to eliminate interrupt-blocking critical sections in the upper half of driver code. This evaluation is done through various performance measuring techniques to show differences between transactional and non-transactional code.

Chapter 2

Related Work

Concurrency control mechanisms are used to avoid inconsistency in shared data. In practice, the most common method of protecting shared data is to lock it, execute a read-modify-write sequence, and finally release the lock [11, 15, 22]. This forces any other process to wait until it can acquire the lock. In a uniprocessor system, waiting is not a significant problem because the operating system executes each process in a serialized fashion and continue progress. However, in the device drivers of multi-core systems the problem becomes more pronounced because every core must cease execution of device driver code and wait for the lock/unlock cycle to complete. It is in this context that lock-free implementations of concurrency control must be considered [26, 29, 34, 51].

2.1 Atomic Regions

Several methods for dealing with atomic regions of code have been developed over the years. The most popular is the use of the proberen (P-, wait) and verhogen (V-, signal) operations of semaphores [15]. These P- and V-operations provide a basis for monitor and mutual exclusion synchronization by acquiring and releasing locks as needed. Mutual exclusion, semaphores, and monitors can be considered as *traditional* synchronization mechanisms that provide the semantics needed to protect shared data from being accessed by multiple threads in the system. However, the P- and V-operations introduce problems such as deadlock, priority inversion, and an increase in programmer responsibility for correct code. A *deadlock* situation occurs when the programmer attempts to acquire two different semaphores in two processes in the opposite order, thus preventing

either process from continuing. Typically this is avoided through careful programming practices (such as always acquiring locks in a predefined order). *Priority inversion* is a much more cunning problem that occurs when a low priority process holds a lock that a high priority process must obtain before it can continue. Typical implementations of semaphores allow a low priority process to take processing time away from a high priority process. This falsifies the system invariant that at all times the highest priority process is executing. To avoid this scenario, priority inheritance protocols were developed [16, 47]. These allow a low priority process to be temporarily given a higher priority so the semaphore can be released and given to the higher priority process. While these methods hold true to the properties of the system, they do not run the way the programmer intends with the original high priority process taking precedence.

The above practices are sub-optimal solutions to the problems of deadlock and priority inversion. By using traditional synchronization as a way to lock and unlock data, the programmer must take the time to carefully acquire and release all the locks before reading or modifying the data. This greatly constrains the granularity of accessing data; if shared data is accessed, the programmer must create and acquire a lock for any read or write operation. If this is not done, then the data becomes corrupt and the system is unstable or incorrect. To prevent this the programmer can denote an atomic region that can be preempted at any point and safely restart from the beginning [3, 53]. The idea of preemptible atomic regions is important because it allows programmer to assume that contention does not occur; but, if contention does arise, then the atomic operations are unseen until the entire region of code is allowed to occur atomically. This has been shown to reduce jitter and run faster when used with varying priority processes within the Java virtual machine [32, 53]. However, since these are build using the Java virtual machine, they cannot be directly applied to the operating system, as it does not have garbage collection or other high level operations.

2.2 Transactional Memory

A *transaction* (in computer science) was first defined in the realm of database management systems [18] as a way to safely update records in a shared environment. This is done be keeping track of changes and, at commit time, ensuring that all shared data still has the expected values and updates each value. Moreover, a transaction was defined to have the following properties: atomicity,

consistency, isolation, and durability (or the ACID properties) [17, 35]. *Atomicity* means that the process executes one or more operations that are indivisible from each other and occur "instantaneously" (or appear so) [36]. In a transactional system, this means that an atomic block either commits the changes or aborts the changes. The *consistency* property guarantees that a transaction takes a current consistent state and moves to a new consistent state. *Isolation* means that a transaction does not depend on or affect a separately executing process in the system. Finally, *durability* makes the transaction irrevocable once the commit has completed.

2.2.1 Hardware Transactional Memory

Herlihy and Moss brought the idea of arbitrary sized transactions into the systems community as a way to safely update shared data in a multiprocessor system [29]. Their simulated implementation showed promising results for hardware transactional memory (HTM) but relied on extending the underlying hardware to support the system. A similar implementation by Stone et al. extended the existing single word load-linked/store-conditional (LL/SC) operations into a multiple reservation system that effectively enabled multi-word atomic operations [51].

These hardware solutions place several restrictions on transactional memory. One such restriction is that the size of an object is bounded by hardware constraints. Several authors have developed potential solutions to this problem [2, 21, 40, 42].

Ananian et al. describe an "Unbounded Transactional Memory" system that allows a transaction to grow as large as virtual memory. However, their implementation does not allow a transaction to grow as large as virtual memory and restricts the transaction size to that of physical memory [2]. Rajwar et al. created a virtual transactional memory system that is transparent to the user [42]. This system provides a consistent programming model to hide the details of hardware implementation from the software developer; this is promising because the programmer no longer must be concerned with the implementation details of the underlying system. Though these systems provide the functionality to have large transactions, which an O/S needs, they are still built using hardware simulators.

Hammond et al. developed a coherency and consistency protocol for transactional memory that extends existing hardware systems, in a simulator [21]. This system allows safe input/output (I/O) 'transactions' by guaranteeing that an I/O transaction never be allowed to revert to a previous

state. Moore et al. use a log-based system to track old values of data to provide safe reversion of arbitrary sized transactions [40]. Each of these systems differ in implementation of version management and conflict detection algorithms. Again, these systems each make extensive use of hardware simulators to implement extra opcodes that do not exist normally.

2.2.2 Software Transactional Memory

Hardware implementations of transactional memory have only existed as simulations; also, they are either bounded in transaction size or require a complex, inflexible system. As a solution to these problems, Shavit and Touitou proposed a software transactional memory (STM) system [48]. With this STM, the machine must implement the LL/SC operations—everything else to handle a transaction is done at the software level. As with hardware implementations, using software transactions is a non-blocking operation which allows software written using transactions to scale up to multi-core systems.

From this point there have been many improvements to software based transactional memory [1, 12, 14, 27, 28, 50]. Herlihy et al. built the first dynamic STM system to allow the size of a transaction to be defined during the run of a program. Both Saha et al. [1, 41, 45] and Harris et al. [23–25] have presented several papers exploring blocking STM implementations that are still quite efficient. Though the use of a blocking scheme seems to go against what transactional memory was developed for, the authors argue that by integrating their blocking implementation with a run-time scheduler and contention manager they are able to guarantee progress by aborting the transaction of another process if lock acquisition fails. This system also introduces either word- or object-sized conflict detection, allowing for less overhead when creating a transactional object because only one ownership of the object must occur instead of obtaining ownership of every item within the object. Dice et al. introduced improvements to how the system handles commits and version management and additionally implemented their STM system to work with `malloc`/`free` to allow an open memory system where transactional code can use the same memory as non-transactional code [14].

An article in *Communications of the ACM* discusses why STM has not become widely adopted by the programming community [8]. While some of the conclusions—lack of support for legacy code, baseline performance, and compiler instrumentation—are valid, a recent paper by Ni et

al. has provided some answers to the proposed problems [41]. These authors discuss the design and implementation of supporting legacy code, improving performance, and language constructs, for the Multi-Core RunTime STM system they developed [1,45]. In this McRT-STM environment, several modes of operation have been built to allow the safe integration of legacy code with STM-aware systems.

2.2.3 Hybrid Transactional Memory

Unsurprisingly, STM requires higher time and space requirements when compared to HTM. However, HTM needs hardware that has only existed in simulated environments and would need transactional semantics integrated into physical components of the system. Also, HTM would impose a certain level of machine-specific knowledge on the programmer, unless a software based virtual system is in place. As a compromise, several groups have developed hybrid transactional systems which take the advantages of both HTM and STM systems [9, 13, 33, 39]. Each of these systems puts a primary emphasis on performing the transaction at the hardware level to provide for fast transactions. If the hardware fails for some reason (large number or size of transactions occurring in parallel) then the system falls back to a software solution. While these systems provide the advantages of both HTM and STM they also get the disadvantages: code size must now incur the cost of software overhead and physical systems must have costly hardware.

In a similar line of thinking Saha et al. propose a purely software transactional system and add architectural acceleration to create a faster STM system [46]. This solution is interesting because it allows an existing STM system to remain unchanged while providing acceleration for common portions of library code. This gives a consistent system that can be improved transparently as improved hardware becomes available.

2.3 Operating Systems

Developing an operating system and associated structures utilizing transactional memory that are scalable and can perform on the same level as current systems is not a trivial task. One goal for these projects is to use lock-free structures so the kernel can execute on multi-core machines without running into a deadlock situation or becoming an inefficient, serially executing system. One

example of this is the Synthesis kernel that makes use of specialized data structures and the atomic compare-and-swap (CAS) operation of the system [37]. At the time there were only a few atomic operations available, so they fit the most common data structures (stacks, queues, and linked lists) into one- or two-word pieces of data and used the atomic CAS or double-CAS operations. While the novelty of this system demonstrated the possibility of lock-free operating systems, the difficulty of being able to use arbitrarily sized structures remained a large challenge to scalable operating system structure.

A similar idea appears in the Cache Kernel implemented by Greenwald and Cheriton [19]. Here the authors emphasize the usage of non-blocking synchronization to mutate O/S structures and the use of double-CAS operations to make this a feasible task. This work differs from the Synthesis kernel because Cache tries to optimize against a general linked list structure instead of O/S specific structures, allowing application programmers to program against a more generic interface. Both the Synthesis and Cache kernels differ from Transactional Xinu in that they concentrate on using a single operation to atomically change one- or two-word sized memory locations. Unlike these kernels, Transactional Xinu allows an arbitrary amount of processing and data modification to occur in an atomic section.

In the transactional realm, Rossbach et al. have implemented TxLinux—a version of Linux that makes use of a hardware TM system [44]. This is the most comprehensive version of a transactional-based operating system to date; the system makes use of "cooperative transactional spinlocks" (*cxspinlocks*) to deal with managing I/O operations within a transaction. Cxspinlocks fall back to standard locking mechanisms so the I/O operations can complete safely and successfully. This system is based on MetaTM, the underlying hardware simulator that implements a hardware transactional memory design to work with TxLinux.

2.4 Tools

The core of Transactional Xinu is a stable, modern implementation of the Xinu operating system [4, 5, 7, 10]. Xinu has a long history in a university setting. Dr. Douglas Comer developed it at Purdue University over 25 years ago on the DEC LSI-11 for teaching operating systems courses. From there, Xinu made its way into research projects, commercial appliances, and has been ported

to several different architectures including the VAX platform, Sun-2 and Sun-3 workstations, and finally the Intel i386 architecture. After a period of disuse, the Embedded Xinu project was born with the fundamental idea of maintaining simplicity and updating the source code to ANSI C compliance. With this updated version, embedded MIPS based wireless routers were chosen as the target architecture; later this author built a modernized port for the Intel IA-32 architecture based on Embedded Xinu.

By using the minimalist design philosophy and extending key O/S components, Transaction Xinu builds up new device drivers and modifies interrupt handling routines. Now, Xinu interacts with an existing transactional memory library and allows the exploration and analysis of effects a STM has on O/S design.

For the transactional memory implementation, this author has chosen a publicly available, production quality STM library and C/C++ compiler developed by Intel Corporation [31]. This provides a well written, stable STM library. By using these freely available components future implementations can be build and extended with relative ease.

2.5 Summary of Related Work

With the transactional model of sharing data becoming more prolific, computer scientists are interested in seeing what advantages and disadvantages result [30]. This thesis is specifically focused on using transactional memory as a means of reducing the use of traditional locks at the operating system level. Of the several types of transactional memory systems available, this system makes use of the software transactional memory system built at Intel [1, 41, 45, 54]. This software solution provides a working implementation that can be used on current generation machines that are readily available without the use of a hardware simulator. Also, the library provides a mode that can execute a serialized transaction (for use with I/O operations), and provides an obstinate mode for use in interrupt handlers that occur in such a way that no other transaction can execute in parallel.

Using this version of Intel's STM library raises a few issues with using a non-standard operating system such as Embedded Xinu [5, 7] because the library is targeted towards the Linux operating system. By developing a system to show that Intel's library can be adapted to work with an embedded operating system, this author aims to demonstrate the possibility of using STM with only

slight programming overhead. This implementation of Transactional Xinu is significantly different than TxLinux which builds a simulated hardware transactional memory model and adds extensions to the Linux kernel. This author has developed Transactional Xinu to work with hardware and software solutions that exist today and that can be implemented on any platform that complies with the hardware requirements of Intel's library.

Chapter 3

Framework

The goal of this thesis is to develop an operating system kernel that supports device drivers with software transactional memory as a concurrency control mechanism. Specifically, this system is aimed at interrupt-driven device drivers and jitter caused by entering and exiting critical sections. A *critical section* is a piece of code that accesses shared data in the system and must not be accessed by two or more threads simultaneously. Within a critical section any action or state change in the system cannot be seen by a separate thread of execution during the changes, as this may cause the system to become unstable. These critical sections present an interesting problem with respect to the handling of interrupts. When the kernel enters a critical section, some action must be taken to prevent shared data from being accessed by separate threads of execution; in an interrupt-driven system it is possible for the kernel to switch threads of execution at any point.

In a transactional system there is the possibility that input/output operations occurring within a transaction are interrupted. This presents a dangerous situation. When a separate physical device has already begun reading and processing shared data, if a transaction fails these changes to shared data cannot be safely *rolled back* (so called “irrevocable” operations). A rollback refers to the act of reverting the state of memory to one that existed before any actions of the transaction had taken place. Rollbacks can be implemented by keeping a log of changes that have been made or by not modifying memory until a commit operation takes place.

This chapter discusses options for handling critical sections in interrupt-driven devices drivers. Building upon these ideas, this author discusses how to apply transactional memory concepts to these device drivers to avoid the pitfalls of irrevocable operations.

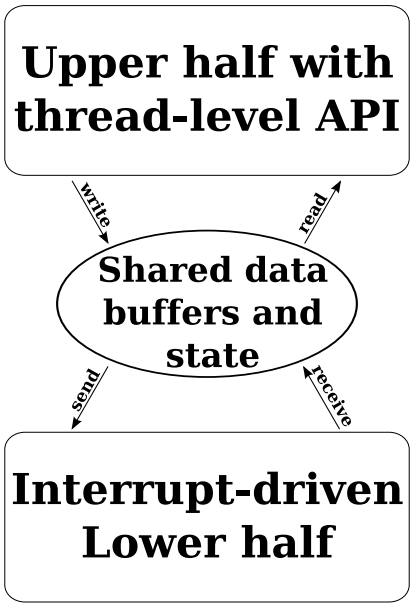

Figure 3.1: Logical separation of device driver halves

3.1 Device Driver Structure

Typical device drivers are separated into two logical halves: an upper half and a lower half. Interaction between the two halves occur as seen in Figure 3.1.

In the upper half of the device driver thread-level API calls are available to create a simple and standard way for the programmer to interact with device drivers. The programmer calls `read` and `write` on specific instances of a device which results in a transfer of data either from or to the device driver. Since `read` and `write` calls are done at the thread-level, the execution is temporally decoupled from the interrupt-driven lower half; thus, the shared data can be accessed at any point during execution.

In order for a thread to send data through the device, several steps must take place. The data begins by entering the device driver from the upper half. It is placed in a buffer shared between the two halves of the driver and the state of the driver is changed to reflect the amount of data now available to the lower half. When an interrupt enters the system the data can be transferred out of the shared buffer and sent via a hardware interface. This process is reversed when data arrives—the data is copied from the physical hardware to a shared buffer. The driver state is updated. Then, when the programmer wants to read data it is copied from the shared buffer into a user buffer. And again the driver state is updated. It is within the device driver that data is shared between the two

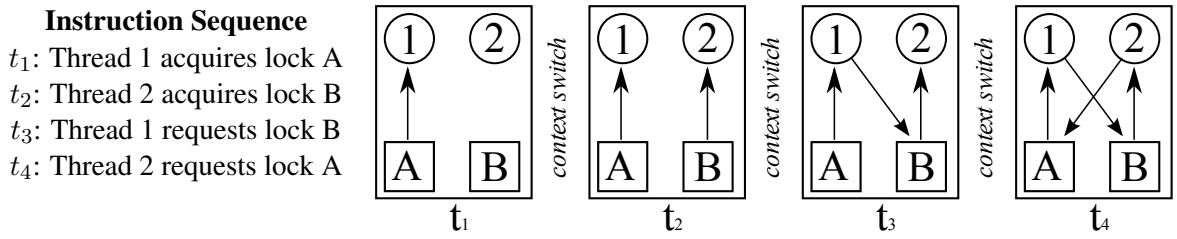

Figure 3.2: Circular deadlock caused by incorrect lock acquisition

separate halves and concurrency issues must be handled to maintain a correct state and provide the correct data to either half when needed.

3.2 Critical Sections

As touched on at the beginning of this chapter, critical sections of code exist when the machine must appear to execute multiple lines of code in a single operation. While the actual processor takes several instruction cycles to complete different operations, from the perspective of the kernel and other threads, the operations occur instantaneously or *atomically* within system. At the completion of a critical section, the kernel maintains a *consistent* state—no illegal or unknown states exist. The critical section occurs in *isolation* so that no other thread of execution is able to see or use an intermediate state. While not applicable to the systems community, *durability* ensures that once the data is committed it is not possible to return to a previous version of the data. In database systems, this collection is referred to as the ACID properties [20]; from an operating systems perspective only the ACI properties are necessary to guarantee the system remains in a safe and known state upon completion of the critical section.

In essence, a critical section requires some sort of concurrency control mechanism to ensure that the kernel does not enter an inconsistent state. Existing concurrency controls—including mutual exclusion, semaphores, and disabling/restoring interrupts—can provide the ACI properties in certain cases.

In the normal case when the kernel is switching between threads of execution, mutual exclusion and semaphores allow critical sections to follow the ACI properties because there is no way for a thread to execute code in a critical section when another thread has locked that data. However, this paradigm introduces the possibility of deadlock in the system if two threads acquire separate

locks in the opposite order. For example, thread 1 acquires lock A, followed by thread 2 acquiring lock B. Now thread 1 wants lock B and thread 2 wants lock A, but neither can proceed since the other has the necessary data lock. This instance of deadlock can be seen in Figure 3.2, where the circles represent threads, squares represent locks, an arc from a lock to a thread shows lock acquisition, and an arc from a thread to a lock is a request. Mutual exclusion also brings priority inversion into the system by allowing a low priority thread to be given processor time. When a low priority thread has acquired a lock that a high priority thread wants, the high priority thread yields the processor to the lower priority thread. Moreover, when interrupts are added into a system, it is possible for a thread of execution to acquire a lock, get interrupted, and have the important interrupt handler yield to the thread holding the lock. Alternatively, the handler could not defer to the thread, fail to acquire the lock correctly, and violate the ACI properties. Either of these cases result in an unpredictable system that cannot guarantee timeliness or even correctness. While there are ways to avoid these problems—like acquiring locks in a consistent order [52] or implementing a priority inheritance scheme [16, 47]—there are still problems when dealing with a system that can handle interrupts.

For an interrupt-driven system, it is clear that mutual exclusion brings many problems and greatly increase the complexity of the system in order to protect a critical section. To ensure the ACI properties, the kernel must use a more aggressive tactic—disabling interrupts. By disabling interrupts when entering a critical section, it is guaranteed that nothing can cause the processor to change threads of execution while executing code in a critical section. Once the critical section is completed, the kernel can restore the interrupts and replay any that were delayed while processing the critical section. While this solution works, it introduces interrupt jitter and does not scale well as multi-core processors enter into the computing field.

3.3 Interrupt Handling

As mentioned, interrupt-driven systems complicate matters in handling concurrency issues, since using locking synchronization methods introduces various problems and disabling interrupts introduces *jitter* into the system. Jitter is created when there are variations in the amount of time the system takes to respond to interrupts entering the system. If interrupts are disabled during critical

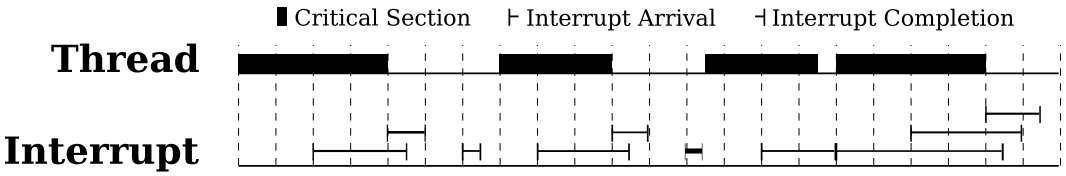

Figure 3.3: Jitter introduced by disabled interrupts

section execution, jitter occurs when an interrupt is raised but cannot begin processing until the critical section completes. This can be seen in Figure 3.3, where interrupts consistently arrive every two time intervals, but the completion time is not predictable due to critical code execution at arrival time. In an interrupt-driven system, the handling of interrupts is of utmost importance since a delay in handling could result in out-of-date or incorrect data. To avoid this issue the kernel can implement a fine-grained interrupt system that only disables certain interrupts when entering corresponding critical sections. Having fine grained control only minimizes the number of interrupts that experience jitter and increase the complexity of the code the programmer has to think about and develop.

Exploring software transactional memory (STM) as a concurrency control mechanism for interrupt-driven device drivers is an intriguing idea. STM offers a different method of concurrency control that allows the programmer to annotate the source code by inserting an “atomic” block around code in a critical section. By using an atomic block in a critical section, the STM library provides the assurance that the ACI properties are obeyed upon the completion of the transaction. This assurance can be extended to interrupt-driven system with ease. If an interrupt enters the system, it is implicitly given the highest priority so it is allowed to execute through to completion. By using this fact, it is possible to reason that the entire handler can be labeled as a critical section. Thus, if the interrupt handling code is annotated as an “atomic” block, it too maintains the ACI properties. Additionally, when the interrupt handler begins execution, it works with the STM library to invalidate any competing transactions that occur in the upper half of the driver.

Unlike mutual exclusion, STM does not transfer execution away from the interrupt handler, so the interrupt still completes in a predictable amount of time. Using STM also does not require the programmer to disable interrupts when entering a critical section. This allows the kernel to begin handling an interrupt immediately when the interrupt enters the system. Combining these two

observations, this thesis argues that using software transactional memory eliminates a major source of jitter from interrupt-driven device drivers.

3.4 Programming considerations

Adding transactional memory to any piece of software comes with some programming considerations. One appealing aspect of transactions is that the programmer no longer has to carefully reason about how the concurrency control works, or if the system can deadlock under certain conditions. The STM library and compiler inserts the code and synchronize data at runtime. The programmer is then freed from dealing with tedious synchronization issues and can concentrate on developing a better system.

Another difference is that transactions provide automatic and more fine-grained concurrency control to the programmer that traditional synchronization. If a programmer needs to access shared data, then a lock for that data must exist and the system must acquire that lock before reading or writing the data. With transactional memory, the locks exist at the machine-word level and do not need to be acquired when a read occurs. This allows a write transaction to acquire only the locks that are needed for that block of code, not a lock on the entire object or a region of memory. Also, when contention does occur on a lock the TM system is able to execute faster since it does not have to wait for the thread holding the lock to release allowing the current critical section to begin execution. This contrasts with traditional synchronization where an overzealous lock protects more data than needed in the critical section. When lock contention does arise under traditional synchronization, the thread must wait until the lock is released before it can enter the critical section.

3.5 Non-Transactional Input/Output

One problem that manifests by using transactions with interrupt-driven device drivers is input/output (I/O) operations. I/O operations make use of direct memory access (DMA) space that shares memory addresses between the operating system and a physical hardware device. This makes for more efficient data transfer because the processor does not have to write every word to the physical hardware, but can instead use memory to write to a common space. Once the data is written to DMA

space, the kernel can send a signal to the physical hardware to begin using the data stored at a specified memory location.

In a transactional system, this causes problems if a transaction fails during the commit phase because the library rolls back any changes made. If the physical hardware had already been notified that new data was available and began reading when the rollback occurs, there are no guarantees that the data is correct. This has long been a problem with merging transactional memory and operating systems, because one of the common places for shared data is in device drivers that share memory space with independent hardware systems. When an interrupt enters the system it is implicitly given the highest priority and does not allow other threads to run. Thus, any conflicting transaction in the system have already begun and no new conflicts can arise during interrupt processing.

From this fact, all that the STM library needs to know is that when an interrupt handler enters the critical section, it must allow the transaction to succeed and invalidate all other related transactions. If both the upper and lower halves of a device driver are written with critical sections protected by an atomic block, then in an interrupt-driven device driver the upper half does not lock the data or delay interrupts allowing the interrupt to begin processing immediately. In the lower half of the driver, the atomic block always beats other competing transactions and invalidate any transactions occurring in the upper half when the interrupt entered. This means that interrupts can be handled immediately by the operating system, thereby reducing the jitter for the device driver, while still performing correctly with thread-level API calls that are interacting with the upper half of the driver. As a consequence of reducing the jitter and adding a STM library, the time it takes to perform upper half reads and writes, as well as lower half sending and receiving, is increased. Despite this slow down, the increase in system predictability and simplification of reasoning about device driver concurrency make software transactional memory a useful tool for interrupt-driven systems.

3.6 Summary of Framework

This chapter has discussed various concurrency control mechanisms that are used in operating system design and how those traditional methods work with respect to an interrupt-driven device driver. A device driver consists of two halves—an upper and lower—that interact with each other through

shared memory buffers and state information. In this device driver the shared data must be protected against inconsistency. This is done through critical sections of code that must complete atomically, end in a consistent state, and occur in isolation. By disabling interrupts in device driver critical sections these properties are maintained, but this action does not scale well and introduces jitter into the system.

Transactional memory is a different form of concurrency control that has not been explored in this context and can guarantee that the ACI properties are maintained. This thesis has presented a method to use STM-based atomic blocks to coordinate between the upper and lower halves of the device driver. Because the interrupt-driven lower half always wins if a transaction is occurring in the upper half at the same time as the lower half, the interrupt still succeeds. Since the interrupt succeeds, the upper half must rollback the changes and try again—this is acceptable because the upper half does not cause the lower half to wait, thereby theoretically reducing interrupt jitter.

Chapter 4

Implementation Notes

This chapter presents specific information regarding the integration of the Embedded Xinu operating system with a publicly available, production quality STM library and compiler from Intel [31]. The Intel STM library was designed for software running under the Linux kernel so several modifications must be made to Embedded Xinu to mimic the expectations of the runtime library.

4.1 Original System

At the core of Transactional Xinu is the Embedded Xinu kernel [5, 7]. This thesis makes use of the Xinu kernel because it is an excellent research platform with under 20,000 total lines of code making it simple to understand and extend, while still providing a rich environment for experimentation. As a system, Embedded Xinu has a lightweight thread model with a shared memory space, a fully-preemptive multitasking priority scheduler, synchronization primitives, inter-process communications, and a variety of device drivers.

4.2 Transactional System

Extending the base system to include transactional components requires careful design and implementation. Since Transactional Xinu revolves around a software transactional memory implementation, the first step was to find a STM library capable of dealing with low level operating system tasks. This author chose a publicly available, production quality STM library and compiler from

Intel because the library offers several modes of operation and is capable of working with legacy code (code that was not written with transactions in mind). Because this library was designed for Linux, Transactional Xinu must be capable of supporting the threading functionality and transparent thread-level storage similar to Linux.

4.2.1 Transactional Library

Intel's STM library and compiler were selected because they offer different modes of operation: optimistic, pessimistic, obstinate, and serial [41]. Optimistic transactions make the assumption that no conflict occurs during the lifetime of the transaction, but if contention arises a rollback provides the correctness guarantees required of the library. Conversely, a pessimistic transaction must assume a conflict occurs and take precautions against such a case (like typical locking systems). The Intel STM library is the first STM implementation to allow both these modes of operation at the same time. Additionally, there is an obstinate mode that allows certain transactions in the system to be stubborn. If an obstinate transaction conflicts with a non-obstinate transaction the contention manager lets the obstinate transaction win. Lastly, serial mode allows integration with legacy and input/output code.

In Transactional Xinu, serial mode allows interrupt handlers to take advantage of certain properties in the STM library. As mentioned, the serial mode of transaction handling exists for legacy (non-transactional) functions to be used in a transaction. This ensures that global variables used in the legacy code safely update without risking data inconsistencies with other transactions. Once a serialized transaction begins execution, it runs through to completion—even if another transaction competes for the shared data.

Single Global Lock Atomicity

Intel's library provides single global lock atomicity (SGLA) semantics to all transactions that are in the system [38, 41]. SGLA semantics create an equivalence between every `__tm_atomic` code block and the same program with every atomic section occurring after a global lock was acquired. This SGLA equivalence property can be seen in the code segments found in Figure 4.1. Thus, a program that uses transactions behaves the same as if each atomic block were guarded by a single

```
( 1) __tm_atomic {          ( 1) wait(global_lock);
( 2)   Statements;    ⟶    ( 2) Statements;
( 3) }                      ( 3) signal(global_lock);
```

Figure 4.1: Code segment showing transactional to SGLA semantics

global lock. By doing this, the library can guarantee that if the program is race free under a single global lock, then the transactional equivalent is also race free. It is important to understand precisely what this property is saying. The guarantee is only valid if the program is race free under a single global lock, if a race condition exists—say between transactional and non-transactional code—then no guarantee is made. While the transactional code may see consistent values for shared variables, the non-transactional code accessing the same variable sees intermediate or inconsistent values. Moreover, the non-transactional code can violate the isolation property of a transaction by modifying data the transaction accesses. Thus, Intel's STM library and compiler can guarantee that no race conditions are introduced to the code, assuming there were no race conditions before.

Obstinate Transactions

Similar to serialized transactions, Intel's STM library provides an obstinate mode. An obstinate transaction does not make use to the SGLA properties; rather it tells the library that it should win over all other transactions in the system. This allows a long running transaction to be preferred over conflicting transactions in the system. Under the default configuration the library automatically tries to switch a transaction that has been aborted 100 times to obstinate mode. However, the library also provides an interface to allow the programmer to declare a transaction as obstinate immediately. Transactional Xinu makes use of this during the interrupt handling phase through the use of Intel's binary interface between the compiler and library. This can be seen in Appendix C beginning on lines 28, 61, and 100. By causing the library to switch the transaction to obstinate mode, the Transactional Xinu interrupt handler is able to beat any other transaction in the system.

Versioning and Logging Properties

Read versioning and undo logging are used by Intel's STM library for handling transactional reads and writes [41, 45]. Read versioning works as follows: the reader makes sure no writer owns the

Reader code

```
(1) global integer our_var
(2) local  integer my_var
(3)
(4) atomic
(5)     if ( our_var == 1 )
(6)         my_var = our_var + 1
(7)     else
(8)         my_var = 1
```

Writer code

```
(1) global integer our_var
(2)
(3) atomic
(4)     our_var = our_var + 1
```

Figure 4.2: Reader and writer code examples

write lock, it then adds the lock-word to its read set, stores the version number, and finally, upon commit the reader validates that the version numbers in its read set have not changed. Taking Figure 4.2 as an example code base, when the atomic section begins `our_var` is read with a lock-word version of 2 and added to the read set $\{our\_var : 2\}$. Similarly, when a writer acquires a lock it adds the version number to its write set and when the lock is being released an updated version number is written to the lock. Using Figure 4.2 as the base, the atomic block gets `our_var` at version 2 and add the lock-word to its write set $\{our\_var : 2\}$. Now, assume the reader loads the value of `our_var` for the `if` statement, then is delayed to let the writer commit a new version of `our_var`; when the reader returns to processing and tries to commit, it fails. This is caused by the version of `our_var` increasing after the writer makes the commit, then when the reader attempts to commit it sees that the version number stored in its read set is out of date and the transaction fails.

This read versioning, write locking system works because if a write occurs after a thread has read the data it is invalidated at commit time since the version number monotonically increases with every write. By using read versioning as opposed to reader locks the library is able to perform faster since a reader does not need to update the lock word during a read. Also, if the reader becomes a writer, there is no need to release the reader lock and acquire a writer lock. The use of undo logging gives the STM library a faster commit path since values do not need to be copied from a private buffer to shared memory. This trivializes the read after write case because the STM does not need to intercept a memory instruction and provide a value from some private memory; instead, it can simply let the read occur in physical memory and wait until commit time to invalidate any data.

Two Phase Locking

Another interesting property of Intel's STM library is that it makes use of a strict two-phase locking (2PL) protocol for its contention manager [45]. Strict 2PL means that every read, write, and unlock action is covered by a shared lock or exclusive lock, this ensures that a transaction maintains the atomicity, consistency, and isolation properties described in Chapter 3 [18]. This is different than other STMs that are designed to be completely non-blocking [21]. In this case, 2PL means that the STM makes a mapping of each memory location to a unique lock, then when a commit takes place the library must acquire all the necessary locks. The mapping of every accessed memory location to an associated lock and the action of acquiring these locks can be seen in both code size increases and the execution time of running Transactional Xinu. The mapping is built at runtime and consumes approximately 4 megabytes of memory to hold the entire table. Note that this size increase is not shown in the code size of the compiled image discussed in Section 4.5, and it is not a trivial amount of space. Keep in mind that the library has also not been carefully crafted to minimize spacial requirements. Since the runtime must also acquire all the locks associated with a running transactions write set, the execution time of transactions is increased. So, while it is acceptable to write transactional code by simply placing every contended function within an atomic block, this greatly increases the runtime since every piece of shared data that is accessed must acquire a lock. Transactional Xinu makes the best effort to minimize the amount of memory mutating code inside a transaction so the number of locks to acquire before a commit is small.

Though the 2PL approach taken by Intel seems to have several performance hits, it greatly simplifies the library code and provides a few interesting benefits. For example, if an interrupt handler is waiting for a commit lock that is held by another thread, then the interrupt handler can ask the contention manager to abort the competing thread's transaction and safely continue. By having a centralized contention manager, normal blocking problems can be avoided and a runtime scheduler can be used to handle conflicts. Deadlock, however, can still be problematic since a transaction waits for locks to be released before continuing and a particularly stubborn transaction may hold the lock for an extended period of time. Intel's STM acknowledges this and has devised a timeout scheme such that a transaction waits a finite amount of time for a lock to be released and then aborts [45]. While this may seem dangerous—as it can lead to situations where a transaction

incorrectly aborts—the database community has shown that this risk is minimal. Gray and Reuter showed that the probability of a single transaction entering a deadlock is approximately $nr^4/4R^2$, where n is the number of processes, r is number of records accessed, and R is the total number of records in the system [18].

From these properties, Intel's STM library is able to make data conflicts appear as lock contention. When a reader or writer attempts to access shared memory that another transaction is using, it finds that a write lock has been acquired and a conflict arises upon commit. Thus, when a conflict occurs the contention manager is able to determine what thread can complete and which must wait.

4.2.2 Interrupt-local storage

While Intel's STM library provides several useful properties, it was designed to work on user-level processes which presents some problems working with kernel-level tasks. Under Linux, a programmer has a segment of memory for "thread-local storage" (TLS). This segment is unique to every active thread on the system and provides a simple way to maintain a variable that is global at the thread (not process) level. Intel's STM library makes use of TLS to maintain a transaction descriptor that identifies what transaction is active in the thread. Specifically, with every thread context switch, Linux updates the segment registers to point to a portion of memory that only the active thread can access. With Transactional Xinu, the same behavior must be matched. To do this, every thread control block has a segment of memory available to the thread via the GS register, so when a context switch occurs the old value is discarded and the new thread's memory segment is loaded.

While this form of TLS provides a safe method of maintaining a unique value in a known variable to a user-level thread it does not interact well with interrupt handling. To integrate STM with interrupt-driven device drivers, proper interrupt handling is essential. Under Transactional Xinu the interesting behavior is not in user-level transactions—it lies in processing transactional interrupts. In the normal case, when an interrupt is raised, the current thread state is pushed onto the stack, and processor execution jumps to a specific interrupt handler. However, the segmentation registers are not updated when an interrupt begins processing because it adds a significant amount of overhead to the interrupt handler, which degrades system performance.

Library Call	Function
`pthread_key_create`	create a thread specific data key.
`pthread_setspecific`	set a key value unique to the currently executing thread.
`pthread_create`	create a thread (for testing)
`pthread_join`	wait for thread termination (for testing)

Figure 4.3: POSIX Thread API under Transactional Xinu

In order to avoid this pitfall, Transactional Xinu is augmented with "interrupt-local storage." Under this scheme, when an interrupt enters the system, the state of the currently executing thread is pushed onto the stack like before. The system also switches a single segmentation register, GS, to use a known memory block. This memory block is declared statically, and is unique to every interrupt request the system can handle. This allows the interrupt context to be distinct from the thread context as far as the transaction is concerned. Since each interrupt request maintains a private memory block, a system that allows prioritized interrupts is still able to make use of transactional device drivers because no memory is shared between interrupt handlers. Once the interrupt handler completes execution, the segmentation registers are reverted to the original thread-local storage context.

4.2.3 Additional Components

The Intel library also relies on several POSIX standard functions for accessing thread variables, beginning, and ending a thread. Since one of the major goals of this thesis is to maintain a small kernel size, these functions are implemented as simply as possible—providing wrapper functions to existing operating system functions. As a result, Transactional Xinu includes a compact POSIX thread API while maintaining its lightweight threading model. These library calls appear in Figure 4.3.

4.3 Thread Level Transactions

As a demonstration of correctness three simple programs were developed to test different types of concurrency control. For each snippet of code seen in Figures 4.4, 4.5, and 4.6, two threads are created with each executing the same code with only the `id` value differing. In Figure 4.4, absolutely no synchronization is used between the competing threads. While this may perform quickly, an incorrect result is always generated (by construction of the example). In Figure 4.5, synchronization

```
( 1) for (i = 0; i < 100; i++) {
( 2)     local_counter = global_counter;
( 3)
( 4)     if (id % 2 == 0) {
( 5)         yield();
( 6)         global_counter = local_counter + 1;
( 7)     } else {
( 8)         global_counter = local_counter - 1;
( 9)     }
(10) }
```

Figure 4.4: Code snippet without synchronization

```
( 1) for (i = 0; i < 100; i++) {
( 2)     wait(counter_lock);
( 3)         local_counter = global_counter;
( 4)
( 5)         if (id % 2 == 0) {
( 6)             yield();
( 7)             global_counter = local_counter + 1;
( 8)         } else {
( 9)             global_counter = local_counter - 1;
(10)         }
(11)     signal(counter_lock);
(12) }
```

Figure 4.5: Code snippet using traditional synchronization

between the two threads is done using mutual exclusion to lock the critical section of code while modification of the global variable is performed. This slows the system down, but guarantees a correct answer every time. In Figure 4.6, software transactional memory is used to automatically perform synchronization. Again, this slows the system down from the overhead caused by the automatic synchronization of global data; however, it shows that Intel's STM library and compiler produces and executes correctly under Transactional Xinu.

```
( 1) for (i = 0; i < 100; i++) {
( 2)     __tm_atomic {
( 3)         local_counter = global_counter;
( 4)
( 5)         if (id % 2 == 0) {
( 6)             yield();
( 7)             global_counter = local_counter + 1;
( 8)         } else {
( 9)             global_counter = local_counter - 1;
(10)         }
(11)     }
(12) }
```

Figure 4.6: Code snippet using transactional memory

```
( 1) __tm_atomic {
( 2)     packet = eth_ptr->input[eth_ptr->start];
( 3)     eth_ptr->start = (eth_ptr->start + 1) % ETH_BUFFERLEN;
( 4)     eth_ptr->count--;
( 5) }
```

Figure 4.7: Transactional version of upper half Ethernet device

4.4 Transactional Device Drivers

When writing transactional device drivers, the two halves (upper and lower) must be constructed with critical sections offset with `__tm_atomic`. A simplified example, showing a code snippet from the upper half of a transactional device driver can be seen in Figure 4.7, while the lower half is in Figure 4.8. The lower half of the driver must make use of transactional blocks to invalidate the memory locations referred to in upper half code. A complete listing of the transactional Ethernet

```
( 1) __tm_atomic {
( 2)     eth_ptr->input[eth_ptr->start + eth_ptr->count] = packet;
( 3)     eth_ptr->count++;
( 4) }
```

Figure 4.8: Transactional version of lower half Ethernet device

read, write, and interrupt handlers appear in Appendices A, B, and C, respectively. These listings are stripped of extraneous comments. The only comments appearing show differences between the transactional and non-transactional versions.

One thing to bear in mind is that any function call in a critical region of normal system code or interrupt code must be re-instrumented with transactions in mind. This is necessary because the runtime library must be able to both properly schedule each transaction and keep track of what memory locations have been invalidated during the course of a transaction. If the compiler detects a function call that has not been instrumented with transactions, it automatically generates code to tell the runtime library that this transaction should be serialized, so as to not allow conflicts to occur. If this type of code is generated in the upper half of a device driver, the system is incorrect since the upper half is serialized and forces the lower half to wait until it completes. Thus, upper half code must be completely transactional so the transaction never becomes serialized, and when an interrupt enters it does not have to wait. Within the interrupt handling code, transactional code should execute in obstinate mode so any non-interrupt code does not take priority over the interrupt handler. This ensures that I/O operations are not reverted when they should not be.

With this in mind, every function call and incidental function call that occurs while in a critical section was examined in both the upper and lower halves of the device driver. From this point, every function that can be called within a transaction was built with an additional attribute (`tm_callable`) to tell the compiler that both a non-transactional and transactional version should be built. The non-transactional version is used outside of device driver code and run at normal speeds, while the transactional version is used when called from an atomic block and executes code to invalidate memory locations.

4.5 Code Size Differences

Adding software transactional memory, and all the changes it entails, does increase the overall kernel size, as summarized in Figure 4.9. Originally, the kernel is approximately 342 kilobytes for the stripped ELF encoded image. Once the STM library and compiler generated code was added to the image, the size jumped to 527 kilobytes—over a 50% code size increase. This increase is not insignificant, especially with respect to resource-constrained embedded systems, but it should be

	Non-STM	STM	Increase	%
raw kernel image	365,628	595,251	229,623	62.80%
stripped kernel image	351,048	540,504	189,456	53.97%
excluding STM library (170,869 bytes)				
raw kernel image	365,628	424,382	58,754	16.07%
stripped kernel image	351,048	369,635	18,587	5.29%

Figure 4.9: Kernel code size overhead incurred, in bytes

noted that the library is of production quality and has not been optimized for embedded systems. If the STM library is ignored and only the compiler generated code is counted, the code size increase is only around 5%.

This implementation of Transactional Xinu is intended as a proof-of-concept demonstration that an STM-aware kernel can reduce interrupt jitter with negligible *runtime* overhead. As future work, tuning the STM library and compiler for embedded applications could allow the code size and memory overheads to be reduced to acceptable values.

4.6 Summary of Implementation Notes

Transactional Xinu is the first O/S designed to work with a publicly available, production quality software transactional memory library implementation. This differs from other transactionally aware operating systems by using STM instead of a simulated HTM implementation. Transactional Xinu required several modifications of the Embedded Xinu kernel to complete the build process. To accommodate Intel's STM library and compiler, this author implemented a small POSIX thread API and thread-local storage. For thread-local storage to work with interrupt-driven device drivers, this author added "interrupt-local storage" to provide correct transaction identifiers during interrupt handling. Once the runtime environment is configured, the compiler needs special instrumentation for function calls occurring in an atomic block. This instrumentation tells the compiler to create two versions of a function: one to work with non-transactional code and one to work with transactional code. By adding this special instrumentation, the amount of space the kernel takes on the platform has increased, although the majority of this increase is seen in the runtime library and not additional code overhead.

Chapter 5

Performance Analysis

One major emphasis of Transactional Xinu was to develop a transactional system that runs on real hardware that is available today. This differs from other transaction-based systems that have been developed using simulators to build efficient hardware solutions that do not exist in a usable form. As such, there must be recognition of the hardware that was used for building the implementation (front-end) and the hardware used for running the implementation (back-end).

Transactional Xinu was developed on a front-end machine running Linux and having the associated GNU compiler collection and toolchain. To build the Xinu kernel, the Intel C/C++ STM Compiler, Prototype Edition 2.0 was used. This compiler is used in place of the standard `gcc` compiler and makes use of the GNU linker (`ld`). Once the compiler collects all the C and assembler source files, it generates a complete (unstripped) kernel image in ELF format. This image is then run through the GNU `strip` command to reduce the size and compressed using the standard gzip algorithm. From there, the stripped and compressed kernel file is inserted into a Linux bootstrapping program and is placed in a TFTP folder for transfer to the back-end machine. The front-end machine is running an up-to-date version of Fedora 8 Linux with a kernel version of 2.6.21.7-3.fc8xen.

Intel's STM library requires a processor that supports the Intel IA-32 architecture (also known as x86 or x86-32) with streaming SIMD extensions (SSE). As a back-end system, this role was filled with a mid-model Pentium 4 processor ("Northwood") clocked at 3.0 GHz without hyper-threading technology and based on the NetBurst architecture. This system is equipped with a standard asynchronous serial port and device driver for user interaction. Additionally, the system has a 100 megabits per second Ethernet device and driver for use with high-speed I/O operations.

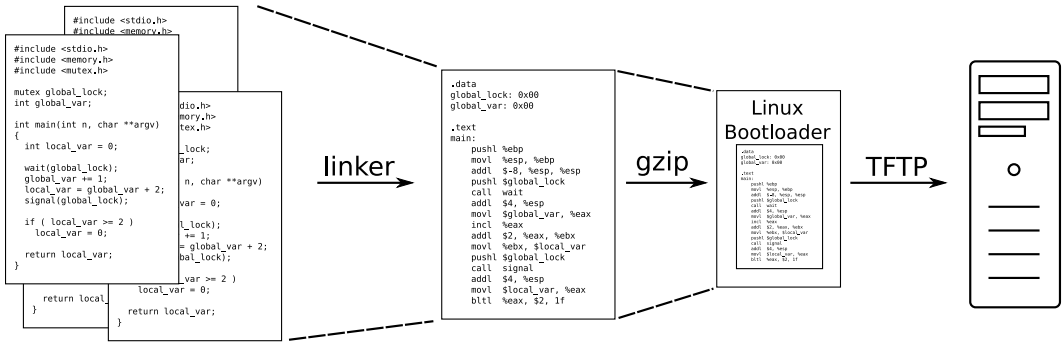

Figure 5.1: Transferring the Transactional Xinu kernel to the back-end machine.

In order to run Transactional Xinu on the back-end system there is a bootstrapping process that begins with the stripped and compressed kernel built on the front-end machine. This process of building and transferring the Transactional Xinu kernel is summarized in Figure 5.1. When the system first boots the BIOS chains off to a netboot bootloader, which uses DHCP to lease a network address and downloads the Linux bootstrapping program in the TFTP directory. Once the program is in place, it uncompresses the kernel image and begins execution of Transactional Xinu. After this point the only code running on the system is that of Transactional Xinu, no BIOS calls or Linux code is used beyond the initial bootstrapping process.

For the rest of this chapter there is discussion of how measurements are gathered and reported. An analysis of a simple program that uses transactional memory for synchronization of a single global variable follows. Next, a description is presented of how testing was done for an interrupt-driven device driver and how performance was measured and taken along with a presentation and discussion of the results. Finally, there is a review of the performance of the system with both thread-level contention and interrupt-driven contention.

5.1 Measurements

One of the challenges of performing tests on a real system is that it is difficult to monitor performance at various levels of operation. Since a goal of Transactional Xinu was to target a existing hardware system, these difficulties were overcome by using a combination of on-chip performance measures, kernel level data, and some results from external test data.

Part of the Intel NetBurst architecture includes a 64-bit built-in performance counter accessible through the `readtsc` (read timestamp counter) opcode. This counter is reset to zero when the processor receives a reset signal and increments once for every *micro-operation* that the processor executes. A micro-operation is defined as sub-division of each opcode of the Intel IA-32 instruction set, including every no-op that occurs. From a programming point of view, this performance counter provides a high-granularity, monotonically increasing number that can provide a sense of time spent between timestamp counter reads. Within the operating system, there are counters that keep track of how long a thread has been running (in microseconds) and a counter that keeps track of how long the kernel has been running (in hundreds of microseconds). External measures include the minimum, maximum, average, and mean deviation or the round-trip time of a ping packet.

5.2 Ethernet Device

5.2.1 Ping Testing Methodology

Testing Transactional Xinu under more realistic conditions was done by performing several different ping tests. These ping tests show how quickly the system is able to perform an interrupt-driven receive, user-level read and write, and finally transmit the response back to the host. Performance measurements are taken at four separate points during the execution of device driver code:

- `RX_TSC` occurs during the receive phase of interrupt handling, when the packet enters the machine and raises an interrupt.
- `READ_TSC` happens in the upper half of the device driver and shares data with `RX_TSC` to move the data to the user buffer.
- `WRITE_TSC` also happens in the upper half, but is instead sharing data with the transmission portion of the device driver.
- `TX_TSC` is the final portion of transmission in the lower half of the device driver.

Each measure is taken in the same place for both the non-transactional and transactional versions of the kernel.

From the operating system perspective, the device driver is executing at the thread level, but within a single system process. The operating system is running in protected mode but applies a flat memory model, so no segmentation or virtual memory is used. The only difference between threads of execution from a memory prospective is the changing of the GS register to point to a "private" memory block for thread- and interrupt-level storage. While any kernel thread is able to access the private data, no thread purposefully exploits this detail.

Due to the nature of the system, the data gathering occurs during execution and stores the results in statically allocated memory locations until it is requested. This reduces the overhead of performing 64-bit calculations, while in an interrupt handler, to a minimum. Once testing is complete, the data is transferred from the back-end machine to the front-end machine via serial port. By doing this, the interrupts that are triggered during testing are limited only to the Ethernet device and timer interrupt. The timer interrupt occurs every $1/10$ of a millisecond with minimal overhead; it does not update the segment registers and does not occur during an Ethernet interrupt. An Ethernet interrupt is raised when a packet is received from the physical line and placed in a shared receive buffer, and another interrupt is raised once the device completes sending a packet to inform the operating system that it can reallocate the shared transmit buffer.

Testing is done by using a front-end machine to send ping packets to the back-end machine on a private, closed network. Each test represents the average value (time in milliseconds or micro-operation cycles) over the course of 100 ping packets. In all cases every ping packet was both sent and received by the front-end machine.

Results

Results from the ping tests as seen from the front-end machine running ping are shown in Figures 5.2, 5.3, and 5.4 with ping intervals of 1000 milliseconds, 500 milliseconds, and under a flood ping, respectively. Numeric data is presented in Figure 5.5 with data averaged over several runs. These results show a small, expected overhead in the STM version of the code. From an external client's perspective, processing time in the STM version only increases by (on average) 45 thousandths of a millisecond.

Similarly Figures 5.6, 5.7, and 5.8 show the results of the timestamp counter performance measure at ping intervals of again 1000 milliseconds, 500 milliseconds, and under flood ping, re-

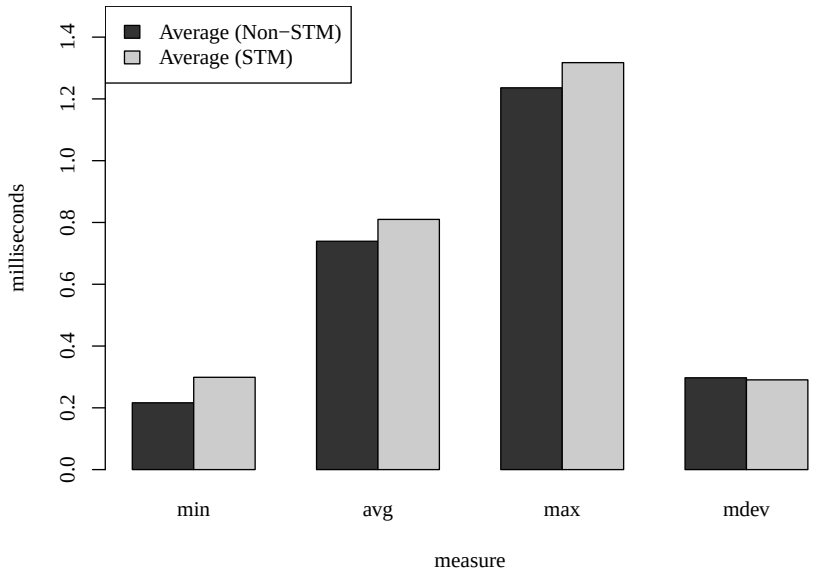

Figure 5.2: Ping results after 100 pings with a 1000 millisecond interval

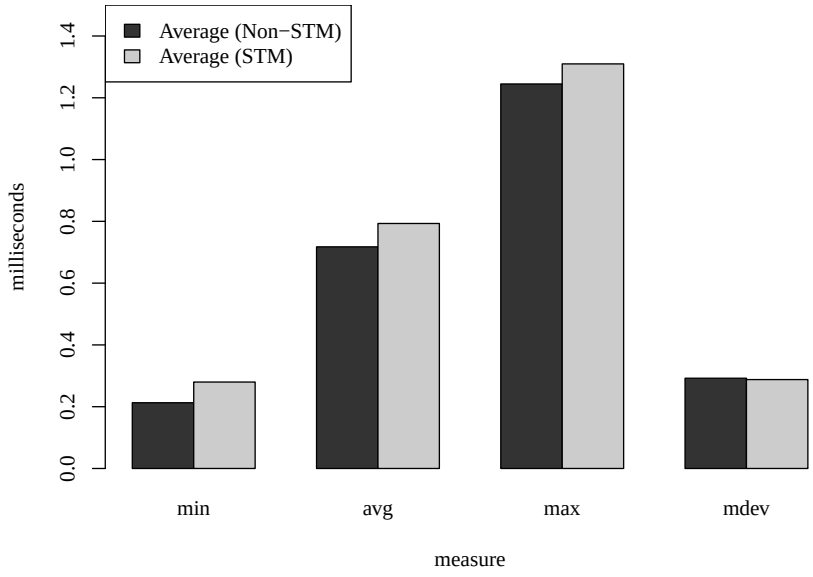

Figure 5.3: Ping results after 100 pings with a 500 millisecond interval

spectively. As can be seen from the results, in all cases the STM kernel takes longer to complete than the non-STM version. The additional overhead of a segment register switch to interrupt-local storage, plus the overhead of updating the transaction accounting information, guarantees that Transactional Xinu implementation experiences a slightly longer, but predictable lag to complete interrupt processing.

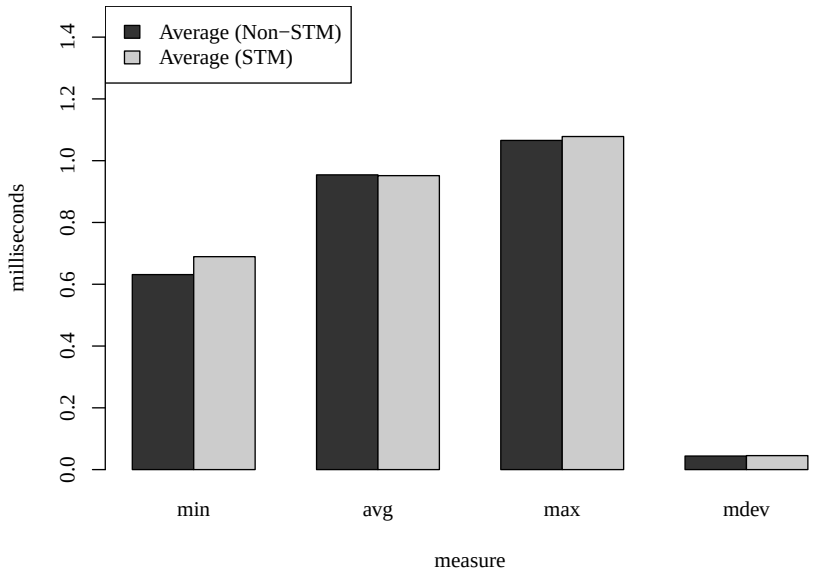

Figure 5.4: Ping results after 100 pings with a minimal interval (flood ping)

1 packet per second stream				
	min	avg	max	mdev
Without STM	0.254	0.749	1.262	0.289
With STM	0.286	0.817	1.479	0.304
STM increase	0.032	0.068	0.217	0.015
2 packets per second stream				
	min	avg	max	mdev
Without STM	0.213	0.706	1.198	0.291
With STM	0.273	0.768	1.292	0.290
STM increase	0.060	0.062	0.094	-0.001
ping flood (max speed)				
	min	avg	max	mdev
Without STM	0.557	0.879	1.070	0.054
With STM	0.609	0.885	1.025	0.051
STM increase	0.052	0.006	-0.045	-0.003

Figure 5.5: Roundtrip ping times measured in milliseconds

Since the micro-operation cycles can vary amongst Intel processors (as the measure is dependent on the internal clock speed of the system) it should be noted that on the back-end machine the time it takes to execute 100,000 micro-operation cycles is about 33 microseconds. On average, this means it takes about 66 microseconds longer to execute the interrupt handler than in non-transactional code. This fixed overhead favorably compares with what has been eliminated—interrupt jitter resulting from the expense of disabling interrupt in the critical sections of upper half

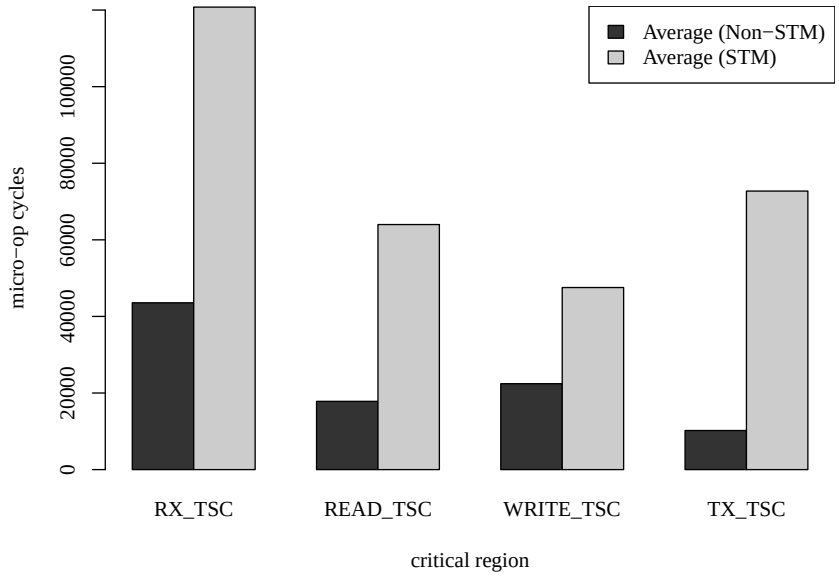

Figure 5.6: Timestamp counter measures for ping with a 1000 millisecond interval

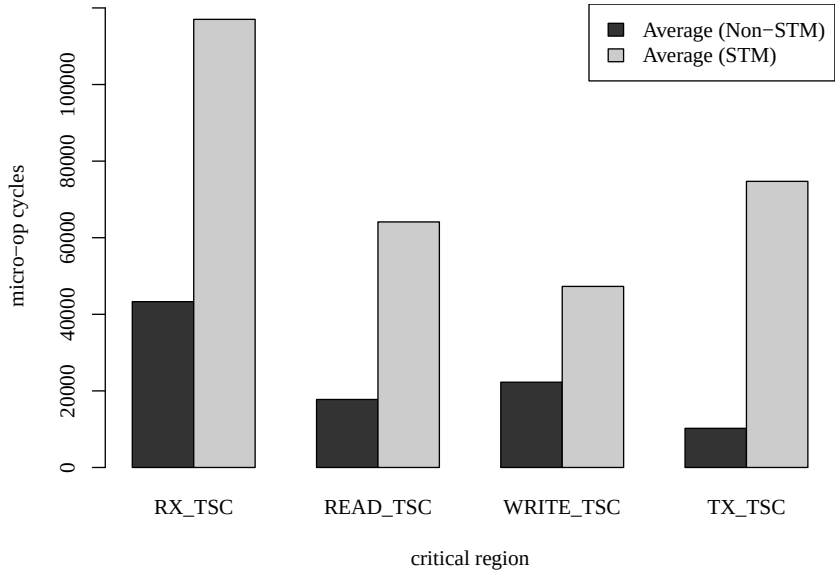

Figure 5.7: Timestamp counter measures for ping with a 500 millisecond interval

code of the drive to prevent the interrupt-driven lower half from running while mutating shared data and driver state.

Thus, while the number of micro-operation cycles executed in lower half processing of the STM system has predictably increased, both external and internal timing measurements show that the total overhead adds only a few fractions of a millisecond to round-trip time and processing in all cases.

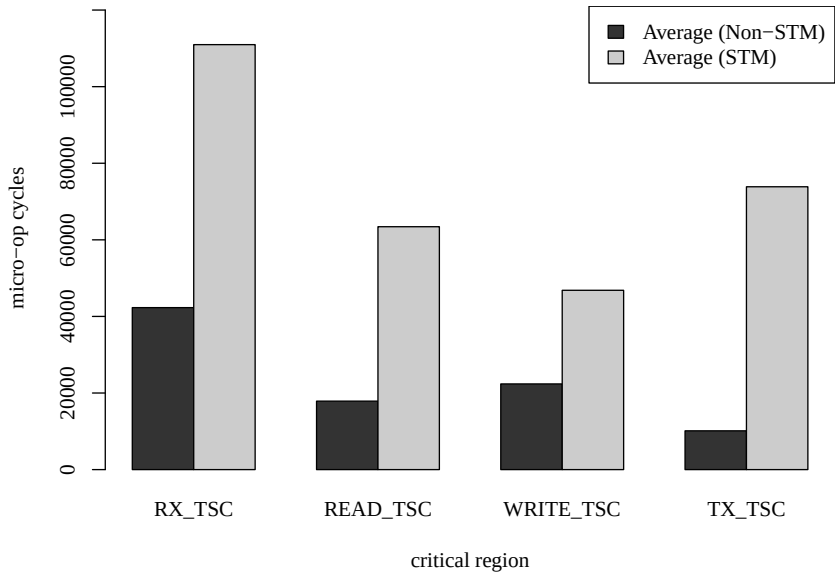

Figure 5.8: Timestamp counter measures for ping with a minimal interval (flood ping)

5.2.2 Timing Test Methodology

Measuring the jitter in interrupt handling is extremely difficult in a real system; hardware counters simply cannot track this kind of information, and having interrupts disabled in critical sections, by definition, makes it impossible to interrupt and record arrival times. Additionally, many practical interrupt sources cannot be developed to generate precisely timed requests, and internal software measuring can distort jitter and lag measurements by adding directly to interrupt handling code.

In order to observe the effects of STM-based jitter reduction, a lightweight timestamping mechanism was added to the head of the interrupt handler to record the time of arrival for each interrupt generated, and a system to provide a precisely timed stream of packets to the Ethernet device was developed. This timestamping mechanism is based off the `readtsc` opcode, and reads the timestamp counter, place the 64-bit value in an array of interrupt arrival times, and continue processing the interrupt. As a packet generator, an embedded networking device running a specialized, non-STM version of Embedded Xinu was used to create a stream of network packets with precise, periodic hardware clock timing and minimal NIC buffering. With a known period between packet transmissions on a directly connected, closed LAN, the arrival time variance was measured by the timestamp difference in inter-packet arrival times on the back-end machine under both a non-STM and STM version of the Xinu kernel.

	Average	Minimum	Maximum	Std. Dev.
Non-STM	30552911.53	30525638	30582946	13847.91
STM	30553291.98	30534954	30576856	13223.17
Difference	380.45	316	-6090	-624.74

Figure 5.9: Jitter Reduction measured in micro-operation cycles

Results

As mentioned, the data is gathered using Intel IA-32 micro-operation cycles as measured by the back-end system. The results presented in Figure 5.9 show the average, minimum, maximum, and standard deviation of the difference of inter-packet arrival times. Across hundreds of these precisely-timed packets, the STM-enabled Ethernet driver show a negligible increase in lag—approximately one thousandth of a percent—but a more meaningful 4.5% reduction in standard deviation. This demonstrates an end result of a reduction of interrupt jitter with very little overhead. Unfortunately, this reduction does not show a statistically significant difference from the original system. This is based on an upper one-tailed F-test using the null hypothesis that the standard deviations are equal, $\alpha = 0.05$, and a sample size around 180.

However, it should be noted that the Ethernet driver can still experience jitter caused by other portions of the system disabling interrupts. Transactional Xinu only has the Ethernet device driver instrumented with transactions. Other device drivers still disable and restore interrupts, but during testing these drivers should only run a small amount of time. The timer interrupt is still active during testing. Though it does not take much overhead, it does execute every 100 microseconds and cause Ethernet interrupts to be deferred. Other non-driver critical sections of the operating system still disable and restore interrupts when needed, though again, these should not execute frequently during testing. Additionally, while the testing is done on a closed network, it still exists in the physical world where collisions and transient congestion can delay packets for short periods of time, causing slightly inconsistent packet arrival times.

5.3 Summary of Performance Analysis

Measuring performance on a real system is difficult because inserting separate physical performance evaluators between hardware components to take timing measurements is expensive and may skew

results. This thesis has developed low overhead methods to measure the performance overhead and jitter associated with interrupt arrival times. Several different methodologies were developed and implemented to gather the data. These include using the on-chip timestamp counter, internal operating system clock time, and externally calculated round-trip times. By placing the internal measures in key portions of code the differences in running non-transactional and transactional code can be seen. While it is not surprising that the transactional form of code takes more time to execute on the system, what Transactional Xinu aims to examine is the jitter (or variability) introduced by disabling interrupts versus using transactions in critical sections of code. The Transactional Xinu kernel and these experiments have shown that is is possible to develop a kernel using software transactional memory with interrupt-driven device drivers and that the transaction-aware kernel is able to perform with only small overheads and with slightly less variance than the non-transactional kernel.

Chapter 6

Summary and Future Work

This thesis has presented a discussion of various transactional memory systems, their relations with operating system structure, a framework for integrating transactional memory with the operating system kernel, implementation notes for developing Transactional Xinu from Embedded Xinu, and an analysis of the performance results within Transactional Xinu. In this chapter a brief summary of what has been discussed and shown is presented, highlighting the contributions of this work, and at the conclusion is a discussion of future work that has been made possible.

6.1 Summary

Various transactional memory systems have been developed over the past two decades based on the idea of database transactions [17, 29]. These original transactional memory systems were implemented at the hardware level and in simulated environments. However, fabrication of the hardware is both expensive and forces the system to have an upper bound on both the size and number of transactions that are live in the system. Several hardware/software hybrid systems have been developed in order to get around the space restrictions, but these physical systems are still built in simulation. Other researchers have developed purely software-based transactional memory systems that have no space limitations and can be used on hardware that exists today [28, 45, 48]. These systems run with slightly higher overhead than their hardware based counterparts, but over time these overheads can be minimized by developing more efficient algorithms, or by leveraging as-yet-unavailable hardware acceleration for STM [1, 46, 49].

Transactional Xinu uses a publicly available, production quality version of Intel's C/C++ Compiler and associated software transactional memory library as described in [1,41,45,54] is used. This software based solution provides a working implementation of transactional memory that runs on any Intel IA-32 processor with SSE enabled. As this is an entirely software system, no hardware simulators need to be used to run the code. Intel's STM library provides several features that are relevant for operating system structure, such as the serialization of transactions that call legacy code, an obstinate mode of operation, and the ability to switch from optimistic transactions to pessimistic transactions.

It should be noted that this is not the first time integrating transactional memory has been tried. In 2007, Rossbach et al. implemented TxLinux by extending the Linux kernel to use *cxspinlocks* and their MetaTM hardware transactional memory system [44]. TxLinux uses hardware transactions when possible, but falls back to spinlocks and other synchronization primitives when dealing with stubborn pieces of code. Other attempts have been made to reduce or eliminate the use of blocking synchronization from the operating system. These can be found in the Synthesis and Cache kernels [19, 37]. Synthesis makes use of the atomic compare-and-swap (CAS) and double-CAS (DCAS) operations by building up operating system structures to fit in one- to two-word sized memory chunks. This novel approach demonstrated the feasibility of a lock-free kernel, but was restricted to structures that were able to fit into only a few word-sized regions. The Cache kernel followed a similar approach, but aims to make it possible for general linked list structures to perform atomic operations. Again, this approach makes use of the CAS and DCAS operations so several restrictions still existed.

Transactional Xinu is built to use software transactions that allow arbitrarily sized data structures to be updated atomically. Because this system makes use of a freely available STM library targeted for the Linux platform, modifications had to be made to the kernel to integrate the library properly. This version of Xinu was based off of an IA-32 port of Embedded Xinu and extended with a small POSIX thread API and "interrupt-local storage." Interrupt-local storage (ILS) is a version of thread-local storage that allows a unique instance of a global variable be available to every thread running in a process, and a unique instance for every interrupt handler. ILS is distinct to Transactional Xinu as a way to allow a software transaction running in an interrupt to be different from the transaction running at the thread-level. In a typical STM platform, only thread-level trans-

actions exist and ILS is not needed. Additional code instrumentation is needed to interact with the STM-aware C/C++ compiler that Intel provides. For critical sections of code to run atomically they must be enclosed in a `__tm_atomic` block of code. Any function calls that occur within the atomic block must also be modified. Since certain functions can exist either inside or outside of an atomic block they are amended with a `tm_callable` attribute that informs the compiler to generate both a transactional and non-transactional version of the function.

In addition to the modifications made to the Xinu kernel, device drivers must be instrumented with transactions in mind. A device driver consists of two halves—an upper and a lower—that interact through shared memory buffers and state. Traditionally, this shared memory is protected by disabling interrupts, mutating the data, and restoring interrupts—this ensures that atomicity, consistency, and isolation are maintained in the state of the driver. While by disabling interrupts in device driver critical sections these properties are maintained, the solution does not scale well and also introduces jitter into the system. By using transactional memory as the concurrency control mechanism, the system is guaranteed to maintain the ACI properties of an atomic block. With this in place, the interrupt-driven lower half is always able to win a transaction even if it conflicts with the upper half—thus the interrupt always runs through to completion. Though the interrupt succeeds, the upper half must rollback the changes and retry. This is acceptable because the upper half transaction can wait while the more important lower half executes, thereby reducing the interrupt jitter of disabling interrupts.

To show differences between the Embedded and Transactional Xinu kernels, this author devised a methodology using several data gathering techniques on a physical system. The first measure was to use the on-chip timestamp counter that measures micro-operation cycles which provide a fine-grained unit of measurement that increments as long as the processor is running. Another measure uses the operating system's timer that increments every $1/10$ of a millisecond, while a third measure is the round-trip time of a ping packet as measured in milliseconds. As expected, the transactional kernel increases the runtime of the interrupt handler since it must invalidate every memory address for competing transactions. However, this thesis is primarily concerned with the jitter (variability) of interrupt handling. In this aspect, transactional memory shows a decrease of approximately 4.5% in beginning an interrupt handler on time. While this is not a huge improvement, it should be noted that this is still a prototype system and various improvements to both the STM

library and Xinu kernel can provide speedups. Additionally, even with these slight improvements, it has been shown that integrating an STM into the operating system kernel is possible and incurs minimal overhead. Transactional Xinu also has shown that device drivers can be written to make use of transaction memory, removing some complexities of writing drivers from the programmer and automating them in the compiler.

In summary, this thesis has provided a framework for using STM in the interrupt-driven device drivers of an operating system. Based on this framework, Transactional Xinu was implemented—based on a modernized IA-32 port of the Xinu operating system—as a proof-of-concept designed to run on existing hardware components. This provides a platform for further research integrating STM into the O/S. Also, a method of measuring jitter and STM overhead was developed and used for a performance evaluation of the system.

6.2 Future Work

This version of Transactional Xinu is designed to work on a single-core processor and has shown that STM can provide advantages in an interrupt driven system, but one of the strengths of transactional memory is automatic concurrency control for multi-core and parallel computations. With this in mind, Transactional Xinu should be extended to work with multi-core processors to explore any advantages transactions offer in interrupt-driven device drivers on a multi-core system. On a multi-core processor, when an interrupt enters the system any core can begin interrupt handler execution. If a different core is running a thread-level critical section, the handler's core must wait until completion. With transactional memory the thread-level code is notified that the transaction fails and can move to some other process. Once the interrupt-level transaction commits, the handler core can begin executing the thread-level code. This can provide higher system throughput even though the individual process throughput is reduced.

Extending the idea of multi-core processors to multiple network interface processors is also an interesting idea. Embedded systems are becoming increasingly network aware by implementing different protocols such as ZigBee, WiFi, Bluetooth, and Ethernet. Merging these networking cores with a multi-core processor and transactional memory could again increase system throughput. Future network devices must be able to handle these protocols efficiently. In order to do this they

increasingly become multi-core and the communication protocols begin sharing memory between devices and the operating system. Building a Transactional Xinu with multiple network devices would be able to show improvements to the network throughput of a multi-core networking system.

Another research track involves building a better STM library with an interest in improving the operating system kernel. Currently, Intel's STM library is targeted at handling concurrency issues within the threads of a process. Transactional Xinu took this library and treated the entire operating system as a single process with several threads of execution for various components. By more tightly integrating the STM runtime with the operating system, it would be possible to build special modes suited for interrupt handling and sharing special memory regions. Other improvements include giving the STM library an ability to interface with the scheduling algorithm to allow better utilization of resources, and developing software to automatically change non-transactional critical sections to become transactional.

Finally, by changing critical sections to not heavy-handedly disable interrupts, this work has made possible new analysis of interrupt-driven code. The work presented by Brylow and Palsberg developed a static analysis tool that determines if an interrupt-driven system is able to meet real-time deadlines before it is deployed [6]. Transactional Xinu allows an interrupt to enter the system at any point. When the interrupt handler begins execution it could be designed to check if any deadlines are approaching, and either defer or handle the interrupt immediately. While this is not a solution to timeliness, it can provide different methods of analysis at runtime.

Also, since interrupts are less frequently disabled, the system can now experience "interrupt overload"—a form of livelock where more time is spent handling interrupts than on making progress on operating system code [43]. If interrupt overload begins occurring, a scheduler for interrupt handling is consulted and can either allow or forbid an interrupt from running. Transactional Xinu can be outfitted with such a system and further be integrated with the transactional memory library to determine if the interrupt should proceed or not, and can be used to improve the timeliness of the system.

This thesis has developed a modernized port of Embedded Xinu targeted at the IA-32 architecture, then extended it to be capable of using a pre-compiled STM library. Transactional Xinu is aimed at exploring an interrupt-driven environment and the prototype system allows device drivers to be written with some degree of automatic concurrency control. By doing this, it is possible to

minimize possible concurrency deadlock issues the programmer must deal with while creating a device driver that is less prone to experiencing jitter. While building a small, safe, secure, stable, and scalable operating system is still in the far distant future, this author believes that Transactional Xinu is a step in the right direction.

Bibliography

[1] ADL-TABATABAI, A.-R., LEWIS, B. T., MENON, V., MURPHY, B. R., SAHA, B., AND SHPEISMAN, T. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (2006), pp. 26–37.

[2] ANANIAN, C. S., ASANOVIC, K., KUSZMAUL, B. C., LEISERSON, C. E., AND LIE, S. Unbounded transactional memory. *High-Performance Computer Architecture, International Symposium on* (2005), 316–327.

[3] BERSHAD, B. N., REDELL, D. D., AND ELLIS, J. R. Fast mutual exclusion for uniprocessors. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1992), ACM, pp. 223–233.

[4] BRYLOW, D. Embedded Xinu. URL `http://www.mscs.mu.edu/~brylow/xinu/`.

[5] BRYLOW, D. An Experimental Laboratory Environment for Teaching Embedded Operating Systems. In *SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (2008), pp. 192–196.

[6] BRYLOW, D., AND PALSBERG, J. Deadline analysis of interrupt-driven software. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2003), ACM, pp. 198–207.

[7] BRYLOW, D., AND RAMAMURTHY, B. Nexos: A next generation embedded systems laboratory. *SIGBED Review: Special Issue from the Workshops on Embedded System Education 6*, 1 (2009).

[8] CASCAVAL, C., BLUNDELL, C., MICHAEL, M., CAIN, H. W., WU, P., CHIRAS, S., AND CHATTERJEE, S. Software transactional memory: why is it only a research toy? *Communications of the ACM 51*, 11 (2008), 40–46.

[9] CHUNG, J., MINH, C. C., MCDONALD, A., SKARE, T., CHAFI, H., CARLSTROM, B. D., KOZYRAKIS, C., AND OLUKOTUN, K. Tradeoffs in transactional memory virtualization. *SIGPLAN Not. 41*, 11 (2006), 371–381.

[10] COMER, D., AND FOSSUM, T. *Operating System Design: The XINU Approach*, PC ed., vol. 1. Prentice Hall, 1988.

[11] COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. Concurrent control with "readers" and "writers". *Communications of the ACM 14*, 10 (1971), 667–668.

[12] DALESSANDRO, L., MARATHE, V. J., SPEAR, M. F., AND SCOTT, M. L. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing* (Portland, OR, August 2007).

[13] DAMRON, P., FEDOROVA, A., LEV, Y., LUCHANGCO, V., MOIR, M., AND NUSSBAUM, D. Hybrid transactional memory. *SIGPLAN Notices 41*, 11 (2006), 336–346.

[14] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II. In *In Proceedings of the 20th International Symposium on Distributed Computing* (2006).

[15] DIJKSTRA, E. W. The structure of the "THE"-multiprogramming system. *Communications of the ACM 11*, 5 (1968), 341–346.

[16] GOODENOUGH, J. B., AND SHA, L. The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks. In *IRTAW '88: Proceedings of the second international workshop on Real-time Ada issues* (New York, NY, USA, 1988), ACM, pp. 20–31.

[17] GRAY, J. The transaction concept: virtues and limitations. In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases* (1981), VLDB Endowment, pp. 144–154.

[18] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, first ed. Morgan Kaufmann, 1993.

[19] GREENWALD, M., AND CHERITON, D. The synergy between non-blocking synchronization and operating system structure. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation* (New York, NY, USA, 1996), ACM, pp. 123–136.

[20] HAERDER, T., AND REUTER, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (1983), 287–317.

[21] HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional memory coherence and consistency. *SIGARCH Computer Architecture News 32*, 2 (2004), 102.

[22] HANSEN, P. B. Concurrent programming concepts. *ACM Computer Surveys (CSUR) 5*, 4 (1973), 223–245.

[23] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. *SIGPLAN Notice 38*, 11 (2003), 388–402.

[24] HARRIS, T., MARLOW, S., PEYTON-JONES, S., AND HERLIHY, M. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2005), ACM, pp. 48–60.

[25] HARRIS, T., PLESKO, M., SHINNAR, A., AND TARDITI, D. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), ACM, pp. 14–25.

[26] HERLIHY, M. A methodology for implementing highly concurrent data structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming* (New York, NY, USA, 1990), ACM, pp. 197–206.

[27] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), ACM, pp. 253–262.

[28] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER III, W. N. Software transactional memory for dynamic-sized data structures. In *Principles of Distributed Computing* (2003), pp. 92–101.

[29] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture* (1993).

[30] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, 2008.

[31] INTEL CORPORATION. C++ STM Compiler, Prototype Edition. URL `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/`.

[32] JAGANNATHAN, S., PROCHAZKA, M., PIZLO, F., AND VITEK, J. Transactional lock-free objects for real-time Java. In *Workshop on Synchronization and Currency in Java Programs (CSJP)* (2004).

[33] KUMAR, S., CHU, M., HUGHES, C. J., KUNDU, P., AND NGUYEN, A. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2006), ACM, pp. 209–220.

[34] LAMPORT, L. Concurrent reading and writing. *Communications of the ACM 20*, 11 (1977), 806–811.

[35] LARUS, J., AND KOZYRAKIS, C. Transactional memory. *Communications of the ACM 51*, 7 (2008), 80–88.

[36] LOMET, D. B. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software* (1977), pp. 128–137.

[37] MASSALIN, H., AND PU, C. A lock-free multiprocessor OS kernel. Tech. Rep. CUCS-005-91, Columbia University, June 1991.

[38] MENON, V., BALENSIEFER, S., SHPEISMAN, T., ADL-TABATABAI, A.-R., HUDSON, R. L., SAHA, B., AND WELC, A. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2008), ACM, pp. 314–325.

[39] MINH, C. C., TRAUTMANN, M., CHUNG, J., MCDONALD, A., BRONSON, N., CASPER, J., KOZYRAKIS, C., AND OLUKOTUN, K. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (2007), pp. 69–80.

[40] MOORE, K., BOBBA, J., MORAVAN, M., HILL, M., AND WOOD, D. LogTM: log-based transactional memory. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on* (Feb. 2006), 254–265.

[41] NI, Y., WELC, A., ADL-TABATABAI, A.-R., BACH, M., BERKOWITS, S., COWNIE, J., GEVA, R., KOZHUKOW, S., NARAYANASWAMY, R., OLIVIER, J., PREIS, S., SAHA, B., TAL, A., AND TIAN, X. Design and implementation of transactional constructs for C/C++. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2008).

[42] RAJWAR, R., HERLIHY, M., AND LAI, K. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 494–505.

[43] REGEHR, J., AND DUONGSAA, U. Preventing interrupt overload. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2005), ACM, pp. 50–58.

[44] ROSSBACH, C. J., HOFMANN, O. S., PORTER, D. E., RAMADAN, H. E., ADITYA, B., AND WITCHEL, E. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (2007), pp. 87–102.

[45] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., MINH, C. C., AND HERTZBERG, B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (2006), pp. 187–197.

[46] SAHA, B., ADL-TABATABAI, A.-R., AND JACOBSON, Q. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 185–196.

[47] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computing 39*, 9 (1990), 1175–1185.

[48] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Symposium on the Principles of Distributed Computing* (1995).

[49] SMARAGDAKIS, Y., KAY, A., BEHRENDS, R., AND YOUNG, M. General and efficient locking without blocking. In *MSPC '08: Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness* (New York, NY, USA, 2008), ACM, pp. 1–5.

[50] SPEAR, M. F., MICHAEL, M. M., AND VON PRAUN, C. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2008), ACM, pp. 275–284.

[51] STONE, J. M., STONE, H. S., HEIDELBERGER, P., AND TUREK, J. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology 1*, 4 (1993), 58–71.

[52] TANENBAUM, A. *Modern Operating Systems*, 3rd ed. Prentice Hall, Englewood Cliffs, 2007.

[53] WELC, A., HOSKING, A. L., AND JAGANNATHAN, S. Preemption-based avoidance of priority inversion for Java. In *International Conference on Parallel Processing, 2004* (2004), pp. 529–538.

[54] WELC, A., SAHA, B., AND ADL-TABATABAI, A.-R. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2008), ACM, pp. 285–296.

Appendix A

Ethernet Read Source Listing

```
(  1) #include <stddef.h>
(  2) #include <device.h>
(  3) #include <ether.h>
(  4) #include <string.h>
(  5) #include <interrupt.h>
(  6) #include <bufpool.h>
(  7) #include <stdlib.h>
(  8) #include <performance.h>
(  9)
( 10) devcall etherRead(device *pdev, char *buf, ushort len)
( 11) {
( 12)     static int read_count = 0;
( 13)     unsigned long long start, end;
( 14)     struct ether *peth;
( 15)     struct ethPktBuffer *epb = NULL;
( 16)     irqmask ps;
( 17)     ushort length = 0;
( 18)
( 19)     if (NULL == pdev) { return SYSERR; }
( 20)     peth = (struct ether *)pdev->controlblk;
( 21)     if (NULL == peth) { return SYSERR; }
( 22)     if (ETH_STATE_UP != peth->state) { return SYSERR; }
( 23)
( 24)     wait(peth->isema);
( 25)
( 26)     __tm_atomic {
( 27)         epb = peth->in[peth->is];
( 28)         peth->in[peth->is] = NULL;
( 29)         peth->is = (peth->is + 1) % IBLEN;
( 30)         peth->icnt--;
( 31)     }
```

```
( 32)
( 33)     if (NULL == epb) { return 0; }
( 34)     length = (epb->length < len) ? epb->length : len;
( 35)     memcpy(buf, epb->data, length);
( 36)     freebuf(epb);
( 37)
( 38)     return length;
( 39) }
```

Appendix B

Ethernet Write Source Listing

```
(  1) #include <stddef.h>
(  2) #include <stdlib.h>
(  3) #include <device.h>
(  4) #include <bufpool.h>
(  5) #include <ether.h>
(  6) #include <e100.h>
(  7) #include <interrupt.h>
(  8) #include <string.h>
(  9) #include <performance.h>
( 10)
( 11) devcall etherWrite(device *pdev, uchar *buf, ushort len)
( 12) {
( 13)     ulong  head;
( 14)     struct ether *peth;
( 15)     struct e100csr *pecsr;
( 16)     struct e100tcb *ptcb = NULL;
( 17)     irqmask ps;
( 18)
( 19)     if (NULL == pdev) { return SYSERR; }
( 20)     peth = (struct ether *)pdev->controlblk;
( 21)     if (NULL == peth) { return SYSERR; }
( 22)     if (ETH_STATE_UP != peth->state) { return SYSERR; }
( 23)     pecsr = pdev->csr;
( 24)     if (NULL == pecsr) { return SYSERR; }
( 25)
( 26)     if (len > TX_BUFFER_SIZE) { return SYSERR; }
( 27)
( 28)     __tm_atomic {
( 29)         ptcb = &peth->txBufs[peth->txHead];
( 30)         peth->txHead =
( 31)             (peth->txHead + 1) % peth->txPending;
```

```
( 32)
( 33)          memcpy(ptcb->data, buf, len);
( 34)
( 35)          ptcb->tcb_bytes = len;
( 36)
( 37)          while ( 0x00 != pecsr->cmd_lo ) { }
( 38)          pecsr->gen_ptr = (uint)ptcb;
( 39)          pecsr->cmd_lo = ETH_SCB_CUC_START;
( 40)      }
( 41)
( 42)      return 0;
( 43) }
```

Appendix C

Ethernet Interrupt Source Listing

```
(  1) #include <stddef.h>
(  2) #include <stdlib.h>
(  3) #include <device.h>
(  4) #include <ether.h>
(  5) #include <e100.h>
(  6) #include <platform.h>
(  7) #include <string.h>
(  8) #include <bufpool.h>
(  9) #include <performance.h>
( 10) #include <clock.h>
( 11) #include <itm/itm.h>
( 12) #include <itm/itmuser.h>
( 13)
( 14) ulong ether_errors = 0;
( 15)
( 16) void rxPackets(struct ether *peth, struct e100csr *pecsr) {
( 17)     struct e100rfd *perfd;
( 18)     struct ethPktBuffer *pepb = NULL;
( 19)
( 20)     perfd = &peth->rxBufs[peth->rxHead];
( 21)
( 22)     if ( (perfd->count & ETH_RFD_CNT_MASK) > ETHERNET_MTU
( 23)       || (perfd->status & ETH_RFD_STAT_ERR) != 0x0000 ) {
( 24)         peth->rxErrors++;
( 25)     }
( 26)     else {
( 27)         __tm_atomic {
( 28)             _ITM_changeTransactionMode(_ITM_getTransaction(),
( 29)                                        modeObstinate,
( 30)                                        NULL);
( 31)             if (peth->icnt < ETH_IBLEN) {
```

```
( 32)                 pepb = getbuf(peth->inPool);
( 33)                 if ( SYSERR == (int)pepb ) {
( 34)                     peth->rxErrors++;
( 35)                 }
( 36)                 else {
( 37)                     pepb->length
( 38)                         = perfd->count & ETH_RFD_CNT_MASK;
( 39)                     memcpy(pepb->data,perfd->data,pepb->len);
( 40)
( 41)                     peth->in[(peth->is+peth->icnt)%ETH_IBLEN]
( 42)                         = pepb;
( 43)                     peth->icnt++;
( 44)                 }
( 45)             }
( 46)             else {
( 47)                 peth->ovrrun++;
( 48)             }
( 49)         }
( 50)     }
( 51)
( 52)     allocRxBuffer(peth, peth->rxHead);
( 53)     peth->rxHead = (peth->rxHead + 1) % peth->rxPending;
( 54)     signaln(peth->isema, 1);
( 55) }
( 56)
( 57) void txPackets(struct ether *peth, struct e100csr *pecsr) {
( 58)     ulong index = peth->txHead;
( 59)
( 60)     __tm_atomic {
( 61)         _ITM_changeTransactionMode(_ITM_getTransaction(),
( 62)                                    modeObstinate,
( 63)                                    NULL);
( 64)         index = (index - 1) % peth->txPending;
( 65)         allocTxBuffer(peth, index);
( 66)     }
( 67) }
( 68)
( 69) interrupt etherInterrupt(void) {
( 70)     struct ether *peth;
( 71)     struct e100csr *pecsr;
( 72)     uchar status, mask;
( 73)     uchar acks = 0x00;
( 74)
( 75)     peth = &ethertab[0];
( 76)     if (!peth) { continue; }
( 77)     pecsr = peth->dev->csr;
( 78)     if (!pecsr) { continue; }
( 79)
```

```
( 80)     mask   = pecsr->cmd_hi;
( 81)     status = pecsr->stat_ack & ~(mask);
( 82)     peth->interruptStatus = (status << 8) | pecsr->status;
( 83)     peth->interruptMask = ~(mask);
( 84)
( 85)     if (!status) { continue; }
( 86)
( 87)     if (status & ETH_SCB_SACK_TX) {
( 88)         peth->txirq++;
( 89)         txPackets(peth, pecsr);
( 90)     }
( 91)
( 92)     if (status & ETH_SCB_SACK_FR) {
( 93)         peth->rxirq++;
( 94)         rxPackets(peth, pecsr);
( 95)
( 96)     }
( 97)
( 98)     if (status & ETH_SCB_SACK_RNR) {
( 99)         __tm_atomic {
(100)             _ITM_changeTransactionMode(_ITM_getTransaction(),
(101)                                        modeObstinate,
(102)                                        NULL);
(103)             while ( 0x00 != pecsr->cmd_lo )
(104)                 ;
(105)             pecsr->cmd_lo = ETH_SCB_RUC_RESUME;
(106)         }
(107)     }
(108)
(109)     pecsr->stat_ack = status;
(110)
(111)     return;
(112) }
```

Marquette University

This is to certify that we have examined this copy of the thesis by

Michael J. Schultz, B.S.

and have found that it is complete and satisfactory in all respects.

The thesis has been approved by:

Dr. Dennis Brylow
Thesis Director, Department of Mathematics, Statistics and Computer Science

Dr. Praveen Madiraju
Committee Member, Department of Mathematics, Statistics and Computer Science

Dr. Craig Struble
Committee Member, Department of Mathematics, Statistics and Computer Science

Dr. Adam Welc
Committee Member, Research Scientist, Intel Laboratories

Approved on