

Instruction Set and Simulation Framework for Transactional Memory

by

Vinson Lee

B.S. Electrical Engineering and Computer Science
University of California at Berkeley, 2000

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

© Massachusetts Institute of Technology 2003. All rights reserved.

Author

Department of Electrical Engineering and Computer Science
May 23, 2003

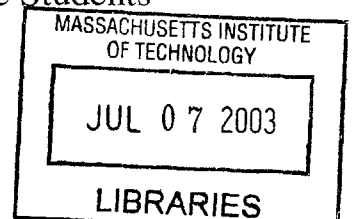
Certified by

.....
Larry Rudolph
Principal Research Scientist
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students

BARKER



Instruction Set and Simulation Framework for Transactional Memory

by

Vinson Lee

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis presents an instruction set extension to support transactional memory. Programmers can use these instructions to write lock-free applications. This thesis also presents a simulation framework that allows a programmer to write, compile, and simulate programs using the new transactional instructions. Benchmarks using transactional instructions and conventional lock schemes are run on the simulation framework, and the resulting performance numbers are compared.

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Acknowledgments

I would like to acknowledge my advisor Larry Rudolph for giving me the opportunity to be a graduate student in the Computation Structures Group. I thank him for his support and advice throughout my graduate career. Larry also provided me the ideas for this thesis.

I am extremely grateful to Derek Chiou for constantly keeping tabs on me. Derek generously offered me advice and help, and his encouragement was essential to finishing this thesis. Derek would make a great teacher, mentor, or parole officer anywhere.

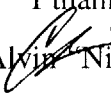
Many of my fellow graduate students deserve thanks. I thank Prabhat Jain for his tons of useful and useless (but mostly useful) advice about research, graduate school, and life in general. I thank Blaise “Blaaaaaise” Gassend for being my human Google, always providing instantaneous answers on programming and \LaTeX . Peter Portante* has been an excellent source for anything related to operating systems. The insane and stupid antics of my semi-officemates Byungsub “Crazyman” Kim and Alfred “Coolman” Ng entertained me for hours on end. I also thank David Chen, Jaewook “Evil Jae” Lee, Daihyun Lim, Daisy Paul, Enoch Peserico, Daniel “Evil Dan” Rosenband*, Edward “Evil Ed” Suh* for their help and friendship, as well as for fantasy baseball, indoor wiffleball, and joining me in my never-ending search for food.

Working with the enthusiastic engineers at MIPS Technologies is what sparked my interest in computer architecture. I thank Chinh Tran for giving me the opportunity to work at MIPS. I thank Art Stamness and Kevin Lau for patiently teaching me many useful computer skills which I still use today.

I thank many people from my undergraduate days at UC Berkeley without whom I would never have ended up at MIT in the first place. I received a glimpse of research life from Andy Neureuther, who gave me the opportunity to work in the LAVA group, and Jan Rabaey, who gave me the opportunity to work with Fred Burghardt at the BWRC. I thank my diligent lab and project partners from my classes, especially Timothy Chan, for putting up with me and helping me get all those good grades.

I thank my MIT friends for making my past few years special and memorable. I especially thank Joe Aung, Li-Wei Chen, Lillian Dai, Irina Medvedev, Tony Ko, Dennis Lee,

Quinton Ng, Anne “strwbrry” Pak, André Puong, Etty Shin, and Andrew Wang for free food, being my chauffeur, road trips, birthday parties, lunches and dinners, Celtics games, and providing me a life outside graduate school.

I thank all my longtime friends, including Melanie Hsu, Jonathan Ko*, “Master Jedi”  ^{Lau, M.D.} Alvin “Nivla” Lau, King Bond Lee, Matthew Lee, Melinda Lee, Braden Leung, Andrew Toy, and Jonathan “Yoda” Woo for staying in touch, making the effort to visit, the care package, and being loyal and supportive fans.

John Wong was the big brother I never had. I thank him for his advice and daring me to achieve.

Finally and most importantly, I thank my family, my mother Jenny Lee, my father Sam Lee, and my sister Winny Lee, for their love, support, and encouragement. This thesis is dedicated to my parents.

*These wonderful English-speaking people also helped proofread this thesis.

Contents

Contents	7
List of Figures	11
List of Tables	15
1 Introduction	17
1.1 Background	17
1.2 Thesis Contributions	18
1.3 Thesis Organization	18
2 Related Work	19
2.1 Transactional Memory	19
2.2 Existing Instruction Set Architectures	20
2.3 Other Related Work	20
3 Transactional Memory	21
3.1 Definitions	21
3.2 Transactional Memory Instructions	21
3.2.1 Begin Transaction	22
3.2.2 Commit Transaction	22
3.2.3 Load Transactional	22
3.2.4 Release	23
3.2.5 Reserve	23

3.2.6	Reserve Conditional	23
3.2.7	Store Transactional	23
3.2.8	Validate	24
3.3	Regular Memory Instructions	24
3.3.1	Load	24
3.3.2	Store	24
3.4	Intended Use	24
3.5	Hardware Implementation Issues	25
4	Simulations	27
4.1	Simics	27
4.2	Benchmarks	29
4.2.1	Counter Benchmark	30
4.2.2	Producer-Consumer Benchmark	34
4.3	Results	34
4.3.1	Single Processor Machine	34
4.3.2	Dual Processor Machine	38
4.3.3	Eight Processor Machine	40
4.4	Benchmark Overhead	43
4.5	Comparison of Critical Section and Transaction with Equal Number of In- structions	51
4.6	Simics Experimental Error	55
4.6.1	Single Processor Machines Comparison	55
4.6.2	Dual Processor Machine Comparison	60
4.7	Discussion	65
5	Conclusion	67
A	Source Code	69
A.1	Transactional Memory API	69
A.1.1	transactional-memory_api.h	69

A.2	Simics Transactional-Memory Module	71
A.2.1	transactional-memory.h	71
A.2.2	transactional-memory.c	71
A.2.3	commands.py	76
A.2.4	Makefile	76
A.3	Benchmarks	77
A.3.1	lock.h	77
A.3.2	Counter Benchmark (counter.c)	77
A.3.3	Consumer-Producer Benchmark (queue.c)	81
B	Simics with Transactional-Memory Module Guide	87
B.1	Simics Installation	87
B.2	Transactional-Memory Module Installation	88
B.3	Virtual Machine Configuration	88
B.4	Programming with Transactional Instructions	89
B.5	Modeling New Instructions	90
	Bibliography	93

List of Figures

4-1	Pseudocode of LT and ST macros.	28
4-2	Pseudocode of spin lock implementation.	29
4-3	Pseudocode of yield lock implementation.	30
4-4	Pseudocode of counting benchmark using lock.	30
4-5	Pseudocode of counting benchmark using transactional memory.	31
4-6	Pseudocode of counting benchmark using transactional memory with exponential backoff.	31
4-7	Pseudocode of consumer part of producer-consumer benchmark using lock.	32
4-8	Pseudocode of consumer part of producer-consumer benchmark using transactional memory with exponential backoff.	33
4-9	Results of counter benchmark from Simics single processor machine.	35
4-10	Results of producer-consumer benchmark from Simics single processor machine.	36
4-11	Results of counter benchmark from Simics dual processor machine.	39
4-12	Results of producer-consumer benchmark from Simics dual processor machine.	41
4-13	Results of counter benchmark from Simics eight processor machine.	42
4-14	Results of producer-consumer benchmark from Simics eight processor machine.	44
4-15	Counter benchmark overhead on Simics single processor machine.	45
4-16	Producer-consumer benchmark overhead on Simics single processor machine.	46

4-17	Counter benchmark overhead on Simics dual processor machine.	47
4-18	Producer-consumer benchmark overhead on Simics dual processor machine.	48
4-19	Counter benchmark overhead on Simics eight processor machine.	49
4-20	Producer-consumer benchmark overhead on Simics eight processor machine.	50
4-21	Results of counter benchmark with additional <code>nop</code> instructions using yield lock and counter benchmark using transactional memory from Simics single processor machine.	52
4-22	Results of counter benchmark with additional <code>nop</code> instructions using yield lock and counter benchmark using transactional memory from Simics dual processor machine.	53
4-23	Results of counter benchmark with additional <code>nop</code> instructions using yield lock and counter benchmark using transactional memory from Simics eight processor machine.	54
4-24	Comparison of results of counter benchmark using spin lock from Simics single processor machine and <code>storm.lcs.mit.edu</code>	56
4-25	Comparison of results of counter benchmark using yield lock from Simics single processor machine and <code>storm.lcs.mit.edu</code>	57
4-26	Comparison of results of producer-consumer benchmark using spin lock from Simics single processor machine and <code>storm.lcs.mit.edu</code>	58
4-27	Comparison of results of producer-consumer benchmark using yield lock from Simics single processor machine and <code>storm.lcs.mit.edu</code>	59
4-28	Comparison of results of counter benchmark using spin lock from Simics dual processor machine and <code>dosx.lcs.mit.edu</code>	61
4-29	Comparison of results of counter benchmark using yield lock from Simics dual processor machine and <code>dosx.lcs.mit.edu</code>	62
4-30	Comparison of results of producer-consumer benchmark using spin lock from Simics dual processor machine and <code>dosx.lcs.mit.edu</code>	63
4-31	Comparison of results of producer-consumer benchmark using yield lock from Simics dual processor machine and <code>dosx.lcs.mit.edu</code>	64

B-1	Simics dual processor machine configuration.	89
B-2	ADD macro.	90
B-3	Code to simulate ADD macro.	91
B-4	Sample program using ADD macro.	92

List of Tables

3.1	Transactional memory instructions.	22
4.1	Parameters for Simics single processor virtual machine.	37
4.2	Parameters for Simics dual processor virtual machine.	38
4.3	Parameters for Simics eight processor virtual machine.	40
4.4	Configuration comparison between Simics single processor virtual machine and <code>storm.lcs.mit.edu</code>	55
4.5	Configuration comparison between Simics dual processor virtual machine and <code>dosx.lcs.mit.edu</code>	60

Chapter 1

Introduction

1.1 Background

When writing parallel applications, programmers need to manage shared data structures to ensure program correctness. Sections of code that access shared data structures, called critical sections, must often be made mutually exclusive. When one process is accessing a shared data structure, no other process is allowed access to it.

Using locks around critical sections is one of the mechanisms used to provide mutual exclusion. Locks are typically implemented with hardware instructions to guarantee that only one process can atomically obtain and hold it. Before entering a critical section, a process acquires the lock. After executing its critical section, the process releases the lock. All other processes wait for the lock until the process releases it.

Processes that use locks suffer from many ill effects. Deadlock occurs when a process cannot progress because it is waiting on a lock that will never be released. This may happen when multiple processes attempt to acquire a set of locks in different orders, a process fails to release a lock after completing its critical section, or a process dies while holding a lock. Convoying arises when a process holding a lock is descheduled, preventing other processes that are capable of running from progressing. Priority inversion arises when a medium-priority process preempts a low-priority process holding a lock needed by a high-priority process. The high-priority process is forced to wait indefinitely for the lock to be released by the low-priority process. In addition, locking is sometimes overly conservative and can

reduce parallelism. Multiple processes may read or write different fields of a shared data structure that can only be determined at runtime.

Herlihy and Moss introduced transactional memory to support lock-free synchronization [2, 3]. In transactional memory, programmers can write critical sections as transactions using primitive transactional instructions. Unlike using locks, multiple processes can be executing their critical sections concurrently. At the end of its critical section, a process attempts to commit the effects of its transaction. If successful, the results of its transaction are made visible to other processes. A process that unsuccessfully commits must discard its changes and retry its transaction.

1.2 Thesis Contributions

This thesis presents an instruction set architecture extension and simulation framework for transactional memory. The instruction set is similar to the one proposed by Herlihy and Moss, but we provide two additional features. First, our transactional instructions allow a process to have more than one active transaction. Secondly, we provide instructions allowing a programmer to explicitly manage a transaction’s data set. We also present a simulation framework for transactional memory. Programmers can write and compile applications using our transactional instructions and test their functionality using our simulator. Finally, we use our simulation environment to compare the performance of benchmarks using our transactional instructions to conventional locking schemes.

1.3 Thesis Organization

Chapter 2 describes related work. In Chapter 3, we present our instruction set architecture extension. Chapter 4 describes our simulation environment and the results of our experiments. Chapter 5 concludes this thesis.

Chapter 2

Related Work

2.1 Transactional Memory

Transactional memory by Herlihy and Moss was the initial proposal to provide hardware support for lock-free data structures. Programmers are provided with special memory instructions to implement transactions. Instead of using locks, critical sections are written as transactions using these instructions. While multiple processes may execute the same transaction, only one process is allowed to commit its changes. Other processes that do not successfully commit must discard its changes and retry its transaction.

Transactional memory can be implemented by adding a separate fully-associative transactional cache and by extending the ownership-based cache coherence protocol of a multiprocessor system. Data that is read or written using transactional instructions is placed into the transactional cache. The transactional cache uses the cache coherence protocol to monitor to reads and writes of other processors. If the transactional cache detects a conflicting read or write by another processor, the transaction is aborted. The limitations of this scheme is that a processor may only have one active transaction and the size of a transaction is limited by the size of the transactional cache.

2.2 Existing Instruction Set Architectures

Hardware instructions for transactions is a generalization of the load-linked and store-conditional instructions in the Alpha [1], MIPS [6, 7], and PowerPC [9] instruction sets. Load-linked and store-conditional instructions can implement atomic read-modify-write operations, but only on a single word. The Intel IA-32 instruction set provides exchange, exchange-and-add, and compare-and-exchange instructions [4]. The SPARC architecture provides the hardware primitives swap, compare-and-swap, and load-store unsigned byte for mutual exclusion [12]. The M68000 family instruction set has test-and-set and compare-and-swap instructions [8]. One of the compare-and-swap instructions performs an atomic operation involving two sets of independent locations.

2.3 Other Related Work

Rajwar and Goodman have proposed Speculative Lock Elision (SLE) [10] and Transactional Lock Removal (TRL) [11]. SLE is a microarchitectural technique that allows multiple processes to execute critical sections protected by the same lock. The processor speculates on which instructions are synchronization instructions and executes the critical section without acquiring or releasing the lock. On misspeculations, the process is retried a finite number of times before actually acquiring the lock. Building on SLE, TRL is a technique to preserve lock-free execution in the presence of conflicts. TRL uses timestamps to order conflicting processes and avoid deadlocks.

Chapter 3

Transactional Memory

3.1 Definitions

We use the same definition of a transaction as Herlihy and Moss. A transaction is a sequence of instructions executed by a single process that satisfies the properties of serializability and atomicity. Serializability requires that the results of concurrent transactions appear as if the transactions were executed serially. The observed order of committed transactions must be the same for all processes. Atomicity requires that the observability of all reads and writes of a transactions appear atomically or not at all to other processes. When a transaction commits, all its changes are made visible atomically. If a transaction aborts, it appears to not have executed at all.

A transaction's read set is defined to be the set of all memory locations read from by the transaction. A transaction's write set is defined to be the set of all memory locations that are written to by the transaction. A transaction's data set is defined to be the union of the read and write sets.

3.2 Transactional Memory Instructions

This section describes the semantics of the instruction set additions listed in Table 3.1 to support transactional memory.

All transactional memory instructions take as one of its inputs a tag. The tag is a

Instruction	Name
BEGINT	Begin Transaction
COMMIT	Commit Transaction
LT	Load Transactional
RELEASE	Release
RESERVE	Reserve
RESERVEC	Reserve Conditional
ST	Store Transactional
VALIDATE	Validate

Table 3.1: Transactional memory instructions.

memory location that acts as a transaction identifier. Variables in a transaction’s data set are tagged with the `tag`. The value of the `tag` holds the status of a transaction and can have one of four possible values: `None`, `Active`, `Committed`, or `Aborted`.

3.2.1 Begin Transaction

Format: `status = BEGINT(tag)`

`BEGINT` attempts to start a new transactions with identifier `tag`. If successful, the value of the `tag` is set to `Active` and `true` is returned. Otherwise, the value of the `tag` is not changed and `false` is returned.

3.2.2 Commit Transaction

Format: `status = COMMIT(tag)`

`COMMIT` attempts to make the tentative changes of the transaction with identifier `tag` permanent. A transaction can commit only if the value of the `tag` is `Active`. If `COMMIT` succeeds, all of the transaction’s changes become globally visible to other processes. The `tag` value is set to `Committed` and `true` is returned. If `COMMIT` fails, all of the transaction’s changes are discarded and `false` is returned.

3.2.3 Load Transactional

Format: `r = LT(addr, tag)`

LT returns the value at memory location `addr` if `addr` is in the data set of transaction with identifier `tag` and the value of `tag` is `Active`. Otherwise, the return value is undefined.

3.2.4 Release

Format: `status = RELEASE(addr, tag)`

RELEASE removes the memory location `addr` from the data set of the transaction with identifier `tag`. Returns `true` if successful and `false` otherwise.

3.2.5 Reserve

Format: `status = RESERVE(addr, tag)`

RESERVE adds the memory location `addr` to the data set of the transaction with identifier `tag` if the value of `tag` is `Active`. If `addr` is in the data set of another transaction with identifier `tag'`, then the transaction with identifier `tag'` is aborted and `tag'` is set to `Aborted`. Any uncommitted change to memory location `addr` by the transaction with identifier `tag'` is discarded. RESERVE returns `true` if successful and `false` otherwise.

3.2.6 Reserve Conditional

Format: `status = RESERVEC(addr, tag)`

RESERVEC adds the memory location `addr` to the data set of the transaction with identifier `tag` if the value of `tag` is `Active` and if `addr` is not in the data set of another transaction. RESERVEC returns `true` if successful and `false` otherwise.

3.2.7 Store Transactional

Format: `ST(addr, value, tag)`

ST writes `value` into memory location `addr` if `addr` is in the data set of transaction with identifier `tag` and the value of `tag` is `Active`. Otherwise, ST performs no operation.

3.2.8 Validate

Format: `status = VALIDATE(tag)`

`VALIDATE` checks to the status of the transaction with identifier `tag`. `VALIDATE` returns `true` if the transaction is `Active` and `false` otherwise.

3.3 Regular Memory Instructions

In addition to transactional memory instructions, regular memory instructions are also supported. However, regular memory instruction must check for conflicts with the data sets of active transactions. If a conflict arises, a transaction may need to be aborted. We can view regular memory instructions as transactions that always commit and abort conflicting transactions.

3.3.1 Load

Format: `r = LOAD(addr)`

`LOAD` returns the value at memory location `addr`. If `addr` is in the write set of a transaction, the conflicting transaction is aborted and `LOAD` returns the value at `addr` before any changes were made to it by the transaction.

3.3.2 Store

Format: `STORE(addr, value)`

`STORE` writes `value` into memory location `addr`. If `addr` is the data set of a transaction, the conflicting transaction is aborted.

3.4 Intended Use

A programmer can implement a transaction by writing code in the following way:

1. Use `BEGIN` to signify the beginning of a transaction.

2. Use `RESERVE` or `RESERVEC` to reserve memory locations to be read and written by the transaction.
3. Use `LT` and `ST` to read and modify memory locations.
4. Use `COMMIT` to make the transaction's changes permanent.
5. Use `RELEASE` to unreserve memory locations of the transaction.
6. If `COMMIT` fails, then the process should return to Step 1 to retry the transaction.

A process can detect a transaction's failure earlier than the `COMMIT` instruction by using `VALIDATE` or by checking the return values of `BEGIN`, `RESERVE`, and `RESERVEC`.

3.5 Hardware Implementation Issues

Hardware additions are necessary to support our instruction additions. Since any data may be involved in a transaction, we propose two additions to each cache line, a transaction valid bit and transaction tag field. The transaction valid bit indicates whether the cache line is involved in a transaction. If this bit is set, the transaction tag field contains the physical address of the transaction tag. A transaction is atomically committed or invalidated by setting the transaction tag. Any memory references to a cache line with the transaction valid bit set must also check the value of the transaction tag.

We also must have additional storage for uncommitted values. To support small sized transactions, we can use a transaction cache similar to Herlihy and Moss. However, some other schemes are necessary to support scalable hardware transactions.

For the remainder of this thesis, we do not take into account the hardware needed to support our transactional instructions.

Chapter 4

Simulations

4.1 Simics

To evaluate our transactional memory instructions we used Simics [5], a full system simulation platform developed by Virtutech. More specifically, we used `simics-1.6.4` with the Linux/x86 host platform and x86 target architecture.

We created a Simics transactional-memory module to simulate transactional memory instructions. This module utilizes the Simics magic instruction to interface with a simulated program. For the x86 target architecture, the Simics magic instruction is `xchg %bx, %bx`. This instruction is equivalent to a `nop` and has no effect on the simulated program behavior. When this magic instruction is encountered in the simulated program, a `Core_Magic_Instruction` event is generated. The transactional-memory module registers a callback function to this event, such that whenever a simulated program executes `xchg %bx, %bx`, the transactional-memory module is invoked. Within the transactional-memory module, we can read and modify registers and memory locations of the simulated program by using the provided Simics API.

By leveraging this magic instruction, we created a simple protocol using the x86 registers to write programs using transactional memory and have Simics recognize when to simulate a transactional memory instruction. To indicate a transactional memory instruction, the simulated program writes values into x86 registers and then calls the magic instruction. When the Simics module is invoked, it reads the values from the registers and

```
int LT(addr, tag) {
    int result;
    movl OP_LT, %eax;
    movl &result, %ebx;
    movl addr, %ecx;
    movl tag, %edx;
    xchg %bx, %bx;
    return result;
}

void ST(addr, value, tag) {
    movl OP_ST, %eax;
    movl addr, %ebx;
    movl value, %ecx;
    movl tag, %edx;
    xchg %bx, %bx;
}
```

Figure 4-1: Pseudocode of LT and ST macros.

uses them to simulate the desired transactional memory instruction. In our protocol, the simulated program writes the transactional memory operation into `%eax`, the destination address into `%ebx`, the source address into `%ecx`, and the tag address into `%edx`. We encapsulate assembly instructions using this protocol into transactional memory macros that can be easily used in a program. Figure 4-1 shows pseudocode for the LT and ST macros. The full source for the transactional memory macros can be found in Section A.1 of the Appendix.

In addition to emulating transactional memory instructions, the transactional-memory module also monitors all regular read and write operations by the simulated program to check for possible conflicts with active transactions.

Simics is not a cycle-accurate simulator, and we do not model caches or memory latency. According to Simics, each instruction takes exactly one cycle. For multiprocessor simulations, each processor is simulated for 1000 cycles at a time. The overhead of a transactional memory instruction is three or four x86 integer instructions in addition to memory spills caused by the clobbering of register values. To measure performance in our simula-

```
void lock(lock_t *l) {
    int val = LOCKED;
    do {
        xchg val, *l;
    } while (val != UNLOCKED);
}

void unlock(lock_t *l) {
    *l = UNLOCKED;
}
```

Figure 4-2: Pseudocode of spin lock implementation.

tions, we use the Unix `time` function from within a Simics virtual machine. In Section 4.6, we show the accuracy of Simics by running our test programs on a real machine and on a Simics virtual machine with a similar configuration.

4.2 Benchmarks

This section describes the two benchmarks used for simulations, the counter benchmark and the producer-consumer benchmark. There are four versions of each benchmark, each with a different synchronization scheme: spin lock, yield lock, transactional memory, and transactional memory with user-level exponential backoff. In a spin lock, a process repeatedly attempts an atomic exchange operation until it successfully acquires the lock. In a yield lock, a process attempts to acquire the lock through an atomic exchange but yields the processor if it is unsuccessful.

The benchmarks are written in C and compiled to x86 binaries using `gcc 2.96` without any optimization. Shared memory regions are implemented using the Unix system call `shmget`. Locks are implemented using the x86 atomic exchange instruction `xchg` (Figure 4-2 and Figure 4-3). Transaction memory instructions are implemented with x86 assembly macros and simulated with Simics as described in Section 4.1.

The complete source code of the benchmarks is listed in Section A.3 of the Appendix.

```
void lock(lock_t *l) {
    val = LOCKED;
    while (TRUE) {
        xchg val, *l;
        if (val != UNLOCKED)
            yield();
        else
            break;
    }
}

void unlock(lock_t *l) {
    *lock = UNLOCKED;
}
```

Figure 4-3: Pseudocode of yield lock implementation.

4.2.1 Counter Benchmark

In the counter benchmark, p different processes increment a shared counter a total of n times. Each process increments the shared counter n/p times. Figure 4-4, Figure 4-5, and Figure 4-6 show pseudocode for the different versions of the counter benchmark.

```
shared int *counter;
shared lock_t *l;

while (success < work) {
    lock(l);
    counter++;
    unlock(l);
}
```

Figure 4-4: Pseudocode of counting benchmark using lock.

```
shared int *counter;
tag_t tag;

while (success < work) {
    BEGINT(&tag);
    RESERVE(counter, &tag);
    ST(counter, LT(counter,&tag)+1, &tag);
    if (COMMIT(&tag)) {
        success++;
    }
    RELEASE(counter, &tag);
}
```

Figure 4-5: Pseudocode of counting benchmark using transactional memory.

```
shared int *counter;
tag_t tag;

while (success < work) {
    BEGINT(&tag);
    RESERVE(counter, &tag);
    ST(counter, LT(counter,&tag)+1, &tag);
    status = COMMIT(&tag);
    RELEASE(counter, &tag);
    if (status) {
        success++;
        backoff = BACKOFF_MIN;
    } else {
        wait = random() % (0x1 << backoff);
        while(wait--);
    }
}
```

Figure 4-6: Pseudocode of counting benchmark using transactional memory with exponential backoff.

```
typedef struct {
    int deqs;
    int enqs;
    int items[QUEUE_SIZE];
} queue_t;

shared queue_t *q;
shared lock_t *l;

int queue_deq(queue_t *q) {
    done = FALSE;
    while(TRUE) {
        lock(l);
        if (q->enqs != q->deqs) {
            result = q->items[q->deqs % QUEUE_SIZE];
            q->deqs++;
            done = TRUE;
        }
        unlock(l);
        if (done) break;
    }
    return result;
}
```

Figure 4-7: Pseudocode of consumer part of producer-consumer benchmark using lock.

```

shared queue_t *q;
tag_t tag;

int queue_deq(queue_t *q) {
    while(TRUE) {
        done = FALSE;
        backoff = BACKOFF_MIN;
        BEGINT(&tag);
        RESERVE(&(q->enqs), &tag);
        RESERVE(&(q->deqs), &tag);
        tail = LT(&(q->enqs), &tag);
        head = LT(&(q->deqs), &tag);

        if (head != tail) {
            result = LT(&(q->items[head%QUEUE_SIZE]), &tag);
            ST(&(q->deqs), head+1, &tag);
            done = TRUE;
        }

        status = COMMIT(&tag);
        RELEASE(&(q->enqs), &tag);
        RELEASE(&(q->deqs), &tag);

        if (status && done) break;

        wait = random() % (0x1 << backoff);
        while (wait--);
    }
    return result;
}

```

Figure 4-8: Pseudocode of consumer part of producer-consumer benchmark using transactional memory with exponential backoff.

4.2.2 Producer-Consumer Benchmark

In the producer-consumer benchmark, p different processes perform n operations on a shared bounded FIFO buffer. $p/2$ processes produce items while $p/2$ processes consume the produced items. A total of $n/2$ items are produced and consumed. Each process produces or consumes n/p items. Figure 4-7 and Figure 4-8 show pseudocode for different versions of the producer-consumer benchmark.

4.3 Results

We ran our benchmarks on three different Simics virtual machines, each having a different number of processors. The first subsection reports the results from a single processor machine, the second subsection shows benchmarks results from a dual processor machine, and the third subsection shows results from an eight processor machine. The configurations for the single processor and dual processor virtual machines were chosen to closely match real machines in our laboratory.

For the counter benchmark, the number of operations n was set to 2^{20} and the number of processes p was varied from 1 to 32. For the producer-consumer benchmark, the number of operations n was set to 2^{14} and the number of processes p was varied from 2 to 32 in increments of two. `QUEUE_SIZE` was set to 1024. For the exponential backoff version of both benchmarks, `BACKOFF_MIN` was initialized to `0x2`.

Each counter benchmark result is composed of an average of the data from four runs. Each producer-consumer benchmark result is composed of data from a single run.

4.3.1 Single Processor Machine

Our single processor virtual machine was created using the Enterprise configuration with the `x86-p3` processor model and the `hippie3-rh62.craff` disk dump. Table 4.1 shows the summary of this virtual machine.

Figure 4-9 shows the results of the counter benchmark on the single processor machine. Each version the benchmark, except for the spin lock version, takes a consistent amount of

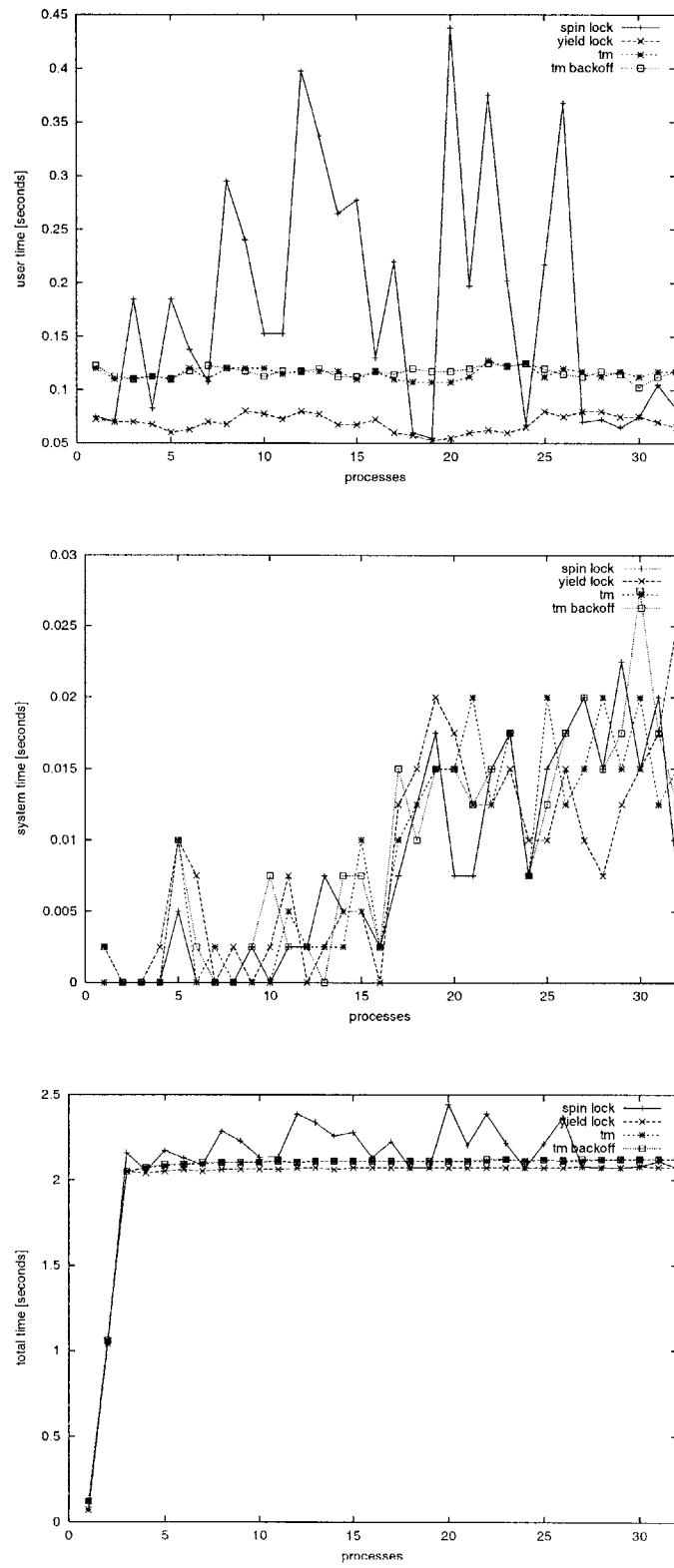


Figure 4-9: Results of counter benchmark from Simics single processor machine.

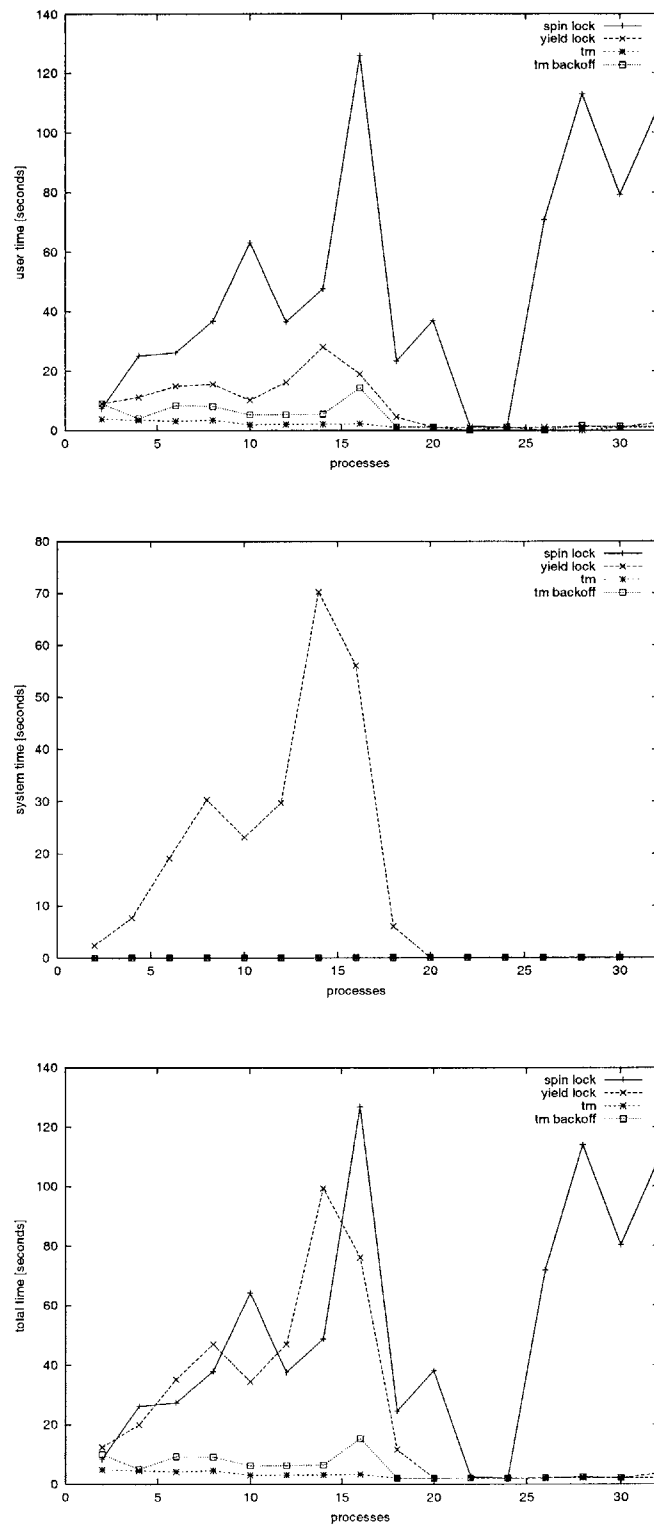


Figure 4-10: Results of producer-consumer benchmark from Simics single processor machine.

	Simics configuration
Number of processors	1
Processor model	x86-p3
Clock speed	600 MHz
Memory	512 MB
Operating system	Red Hat Linux 7.3
Linux kernel	2.4.18-3

Table 4.1: Parameters for Simics single processor virtual machine.

user time for all number of processes. The spin lock version, however, has a big variance in user time, ranging from 0.08 seconds to 0.4 seconds. These variations in user time of the spin lock version are a result of convoying. The process is descheduled while holding the lock and leaves the next active processes to spend their entire time quantum unsuccessfully attempting to acquire the lock. This behavior directly results in the variance seen in the total time graph for the spin lock version. In all the other versions, a sleeping process cannot impede the progress of the active process. Each version of the counter benchmark has nearly zero system time. On average, both transactional memory versions perform approximately 0.8% better than the spin lock version and 4.1% worse than the yield lock version.

Figure 4-10 shows the results of the producer-consumer benchmark on the single processor machine. We see that locks perform much worse than transactional memory. The majority of the total time for the spin lock is user time while the majority of the total time for the yield lock is system time. The queue size limits the progress of the benchmark because the active process does not yield the processor and release the lock if it cannot produce or consume. For the lock versions, the benchmark can only progress if the active process is a producer holding the lock with a queue that is not full or a consumer holding the lock with the queue that is not empty. In the spin lock case, the active process often wastes its entire time quantum spin-waiting for the lock or, if it does hold the lock, waiting for space in the queue to produce or an item in the queue to consume. In the yield lock case, even if the active process can produce or consume, it must yield the processor if it does not hold the lock. Often, the process holding the yield lock is a producer with a full queue or a consumer with an empty queue. In a single processor system, the producers and

	Simics configuration
Number of processors	2
Processor model	x86-p3
Clock speed	500 MHz
Memory	256 MB
Operating system	Red Hat Linux 7.3
Linux kernel	2.4.18-3smp

Table 4.2: Parameters for Simics dual processor virtual machine.

consumers cannot run simultaneously, and the queue is often either full or empty. Transactional memory performs better because it does not need to acquire the lock. The benchmark can proceed if the active process is a producer with a queue that is not empty or a consumer with an item in the queue. On average, the transactional memory version performs 79% better than the spin lock version and 52% better than the yield lock version. The transaction memory with backoff version performs 72% better than the spin lock version and 44% better than the yield lock version.

4.3.2 Dual Processor Machine

We configured a dual processor virtual machine using the Enterprise configuration along with the x86-p3 processor model and the `enterprise3-rh73.craff` disk dump. Table 4.2 summarizes this machine.

Figure 4-11 shows the results of the counter benchmark from the dual processor virtual machine. The results are similar to results from the single process machine with two exceptions. First, the variance of the user time for the spin lock version is greater, and secondly, the yield lock version spends more system time than all the other versions. In the spin lock version, with two processors, the cost of having a lock being held by an inactive process is greater than in the single processor case since two processors are spin-waiting. In the yield lock version, one process is always yielding because only one of the two active processes can obtain the lock, causing system time to be spent for context-switching. On average, the transactional memory version performs 2.3% better than the spin lock version and 2.3% worse than the yield lock version. The transactional memory with backoff ver-

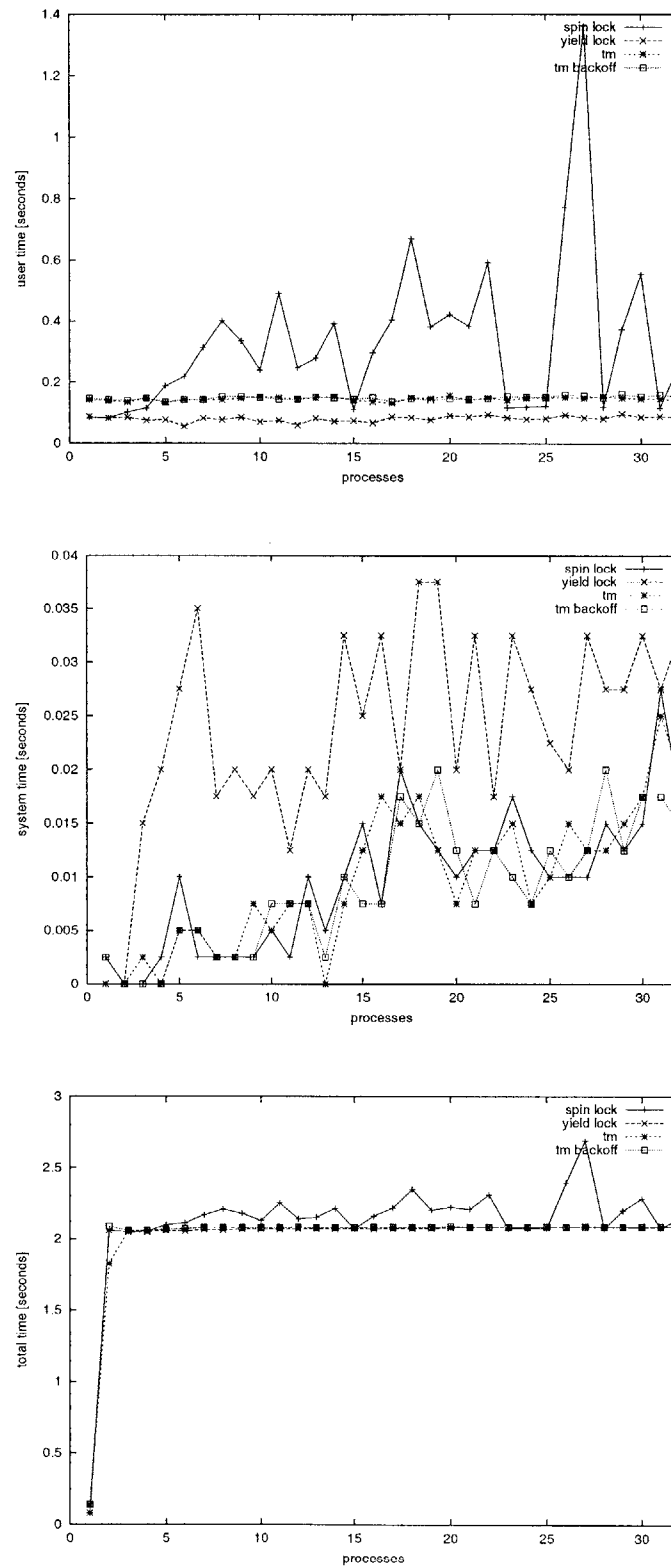


Figure 4-11: Results of counter benchmark from Simics dual processor machine.

	Simics configuration
Number of processors	8
Processor model	x86-p3
Clock speed	500 MHz
Memory	2048 MB
Operating system	Red Hat Linux 7.3
Linux kernel	2.4.18-3smp

Table 4.3: Parameters for Simics eight processor virtual machine.

sion performs 1.9% better than the spin lock version and 2.7% worse than the yield lock version.

Figure 4-12 shows the results of the producer-consumer benchmark on the dual processor virtual machine. Compared to the single processor results, the lock versions perform much better. With two processors, a producer process and consumer process can be active simultaneously, and the queue is neither full nor empty most of the time. On the single processor machine, an active process often could not proceed because the queue was empty or full. In the spin lock case, we can see that convoying occasionally happens, resulting in high user times in a few cases. Transactional memory with backoff performs worse than without backoff when the number of processes is less than 18. Since the only difference is exponential backoff, we conclude that conflicts occur when the number of processes is low, and the overhead of exponential backoff is high. On average, transactional memory performs 25% better than the spin lock and 4.8% worse than the yield lock. Transactional memory with backoff performs 3.7% worse than the spin lock and 47% worse than the yield lock.

4.3.3 Eight Processor Machine

The eight processor machine has a similar configuration to the dual processor machine in the previous subsection except for having eight processors, instead of two, and more main memory (Table 4.3).

Figure 4-13 shows the results of the counter benchmark on the eight processor virtual machine. The user time for the spin lock version is greater than the single and dual proces-

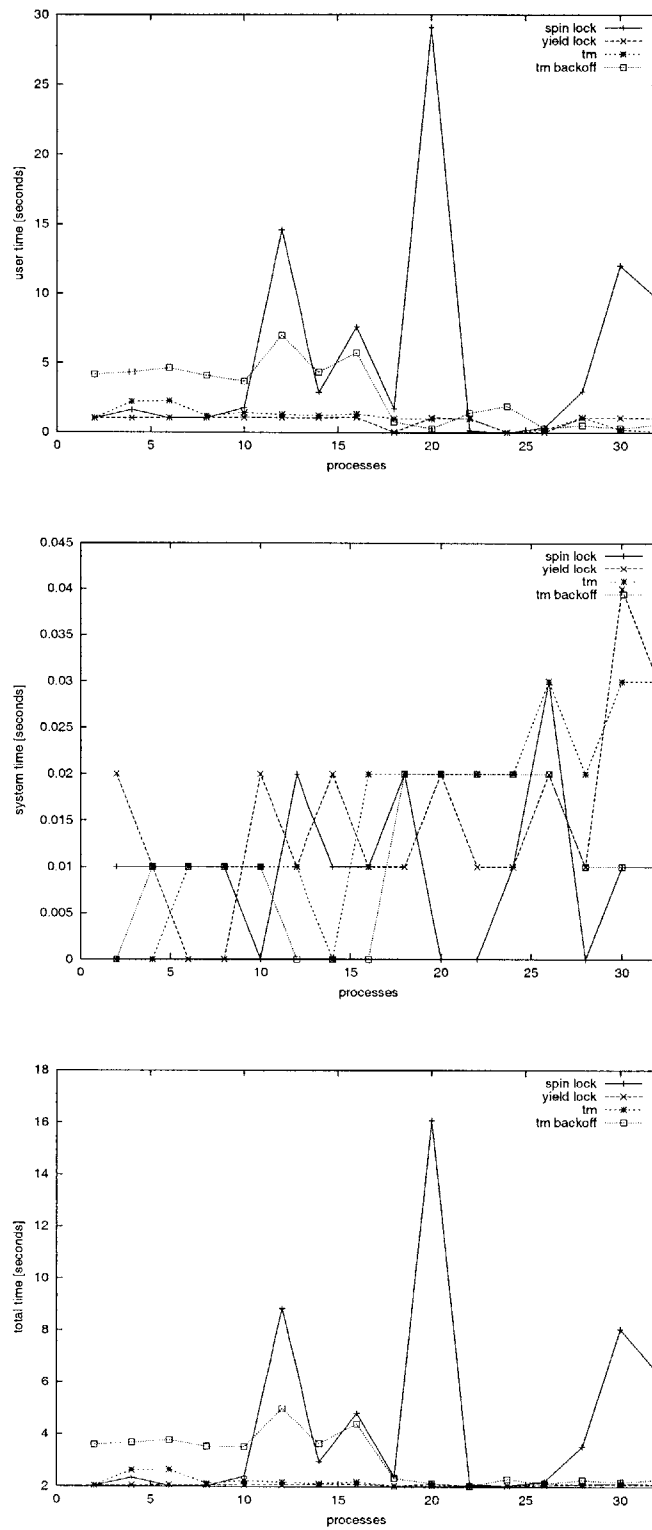


Figure 4-12: Results of producer-consumer benchmark from Simics dual processor machine.

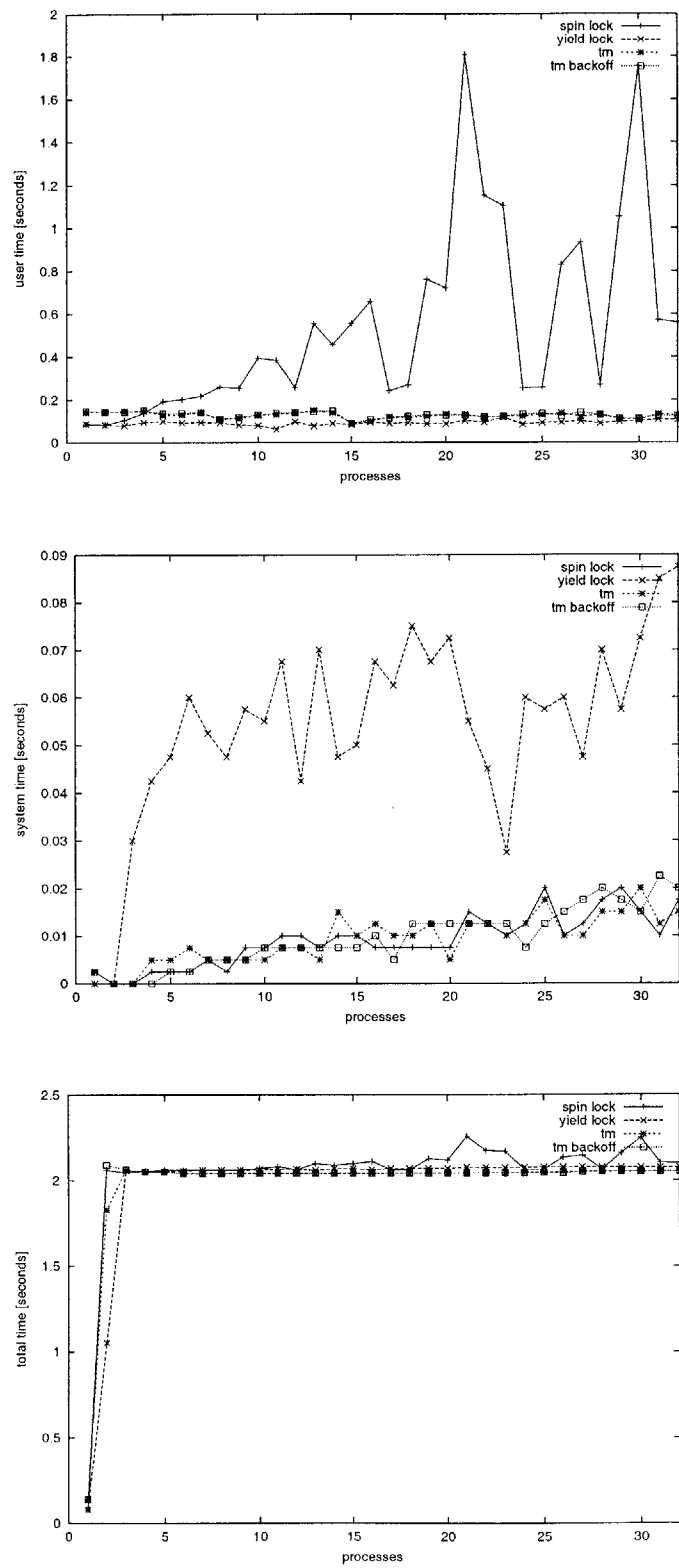


Figure 4-13: Results of counter benchmark from Simics eight processor machine.

sor case. The user time for the spin lock is much greater than the other versions because at least seven processes are spin-waiting for the lock. The system time for the yield lock is much greater than all the other versions because at least seven processes are yielding its processor. On average, transactional memory performs 0.5% better than the spin lock and 3.6% worse than the yield lock. Transactional memory with backoff performs 0.1% better than the spin lock and 4.4% worse than the yield lock.

Figure 4-14 shows the results of the producer-consumer benchmark on the eight processor virtual machine. The high user time for the spin lock shows that the majority of processes are spin-waiting. The high user time for transactional memory with backoff shows that there are a high number of conflicts. On average, transactional memory performs 5.3% better than the spin lock and 2.0% better than the yield lock. Transactional memory with backoff performs 33% worse than the spin lock and 36% worse than the yield lock.

4.4 Benchmark Overhead

We removed the main code sections from all versions of both benchmarks. The remaining code for each benchmark is code common to all versions, such as shared memory initialization and barrier synchronization ensuring all processes start at the same time. We ran the resulting code on the different Simics virtual machines to measure the benchmark overhead.

Figure 4-15, Figure 4-16, Figure 4-17, Figure 4-18, Figure 4-19, and Figure 4-20 show the results. All the graphs are nearly identical. For one process, there is almost no overhead. As the number of processes increases, the overhead quickly rises to a steady amount. In each case, the steady overhead is approximately 2.0 seconds with a small linear growth as the number of processes increase. User and system time comprise little of the overhead.

The observed overhead is a result of the way the benchmarks are written. The first thing occurring in both benchmarks is initialization of the shared data structures. Afterward, a process waits and sleeps until all other processes have finished initialization before beginning the actual benchmarking section. Thus, a benchmark consisting of one process has almost zero overhead because it does not need to wait for any other processes.

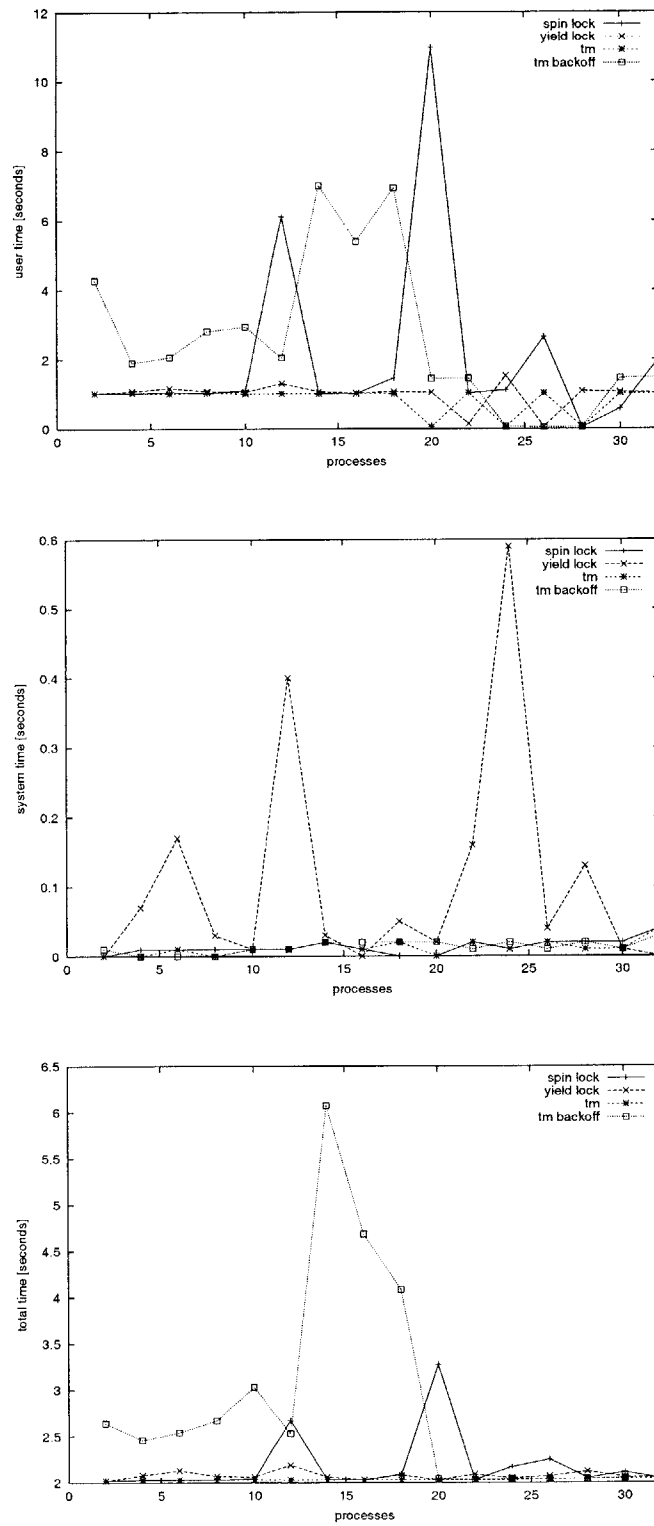


Figure 4-14: Results of producer-consumer benchmark from Simics eight processor machine.

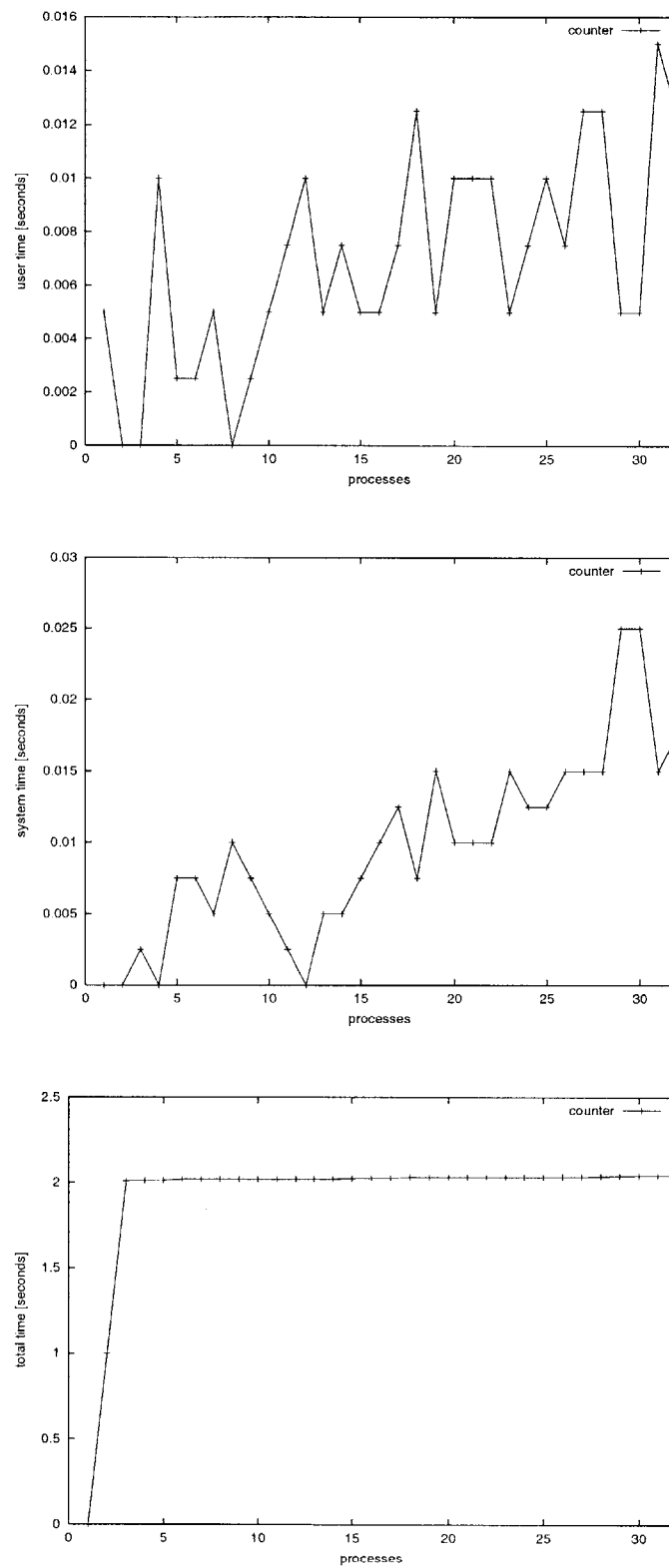


Figure 4-15: Counter benchmark overhead on Simics single processor machine.

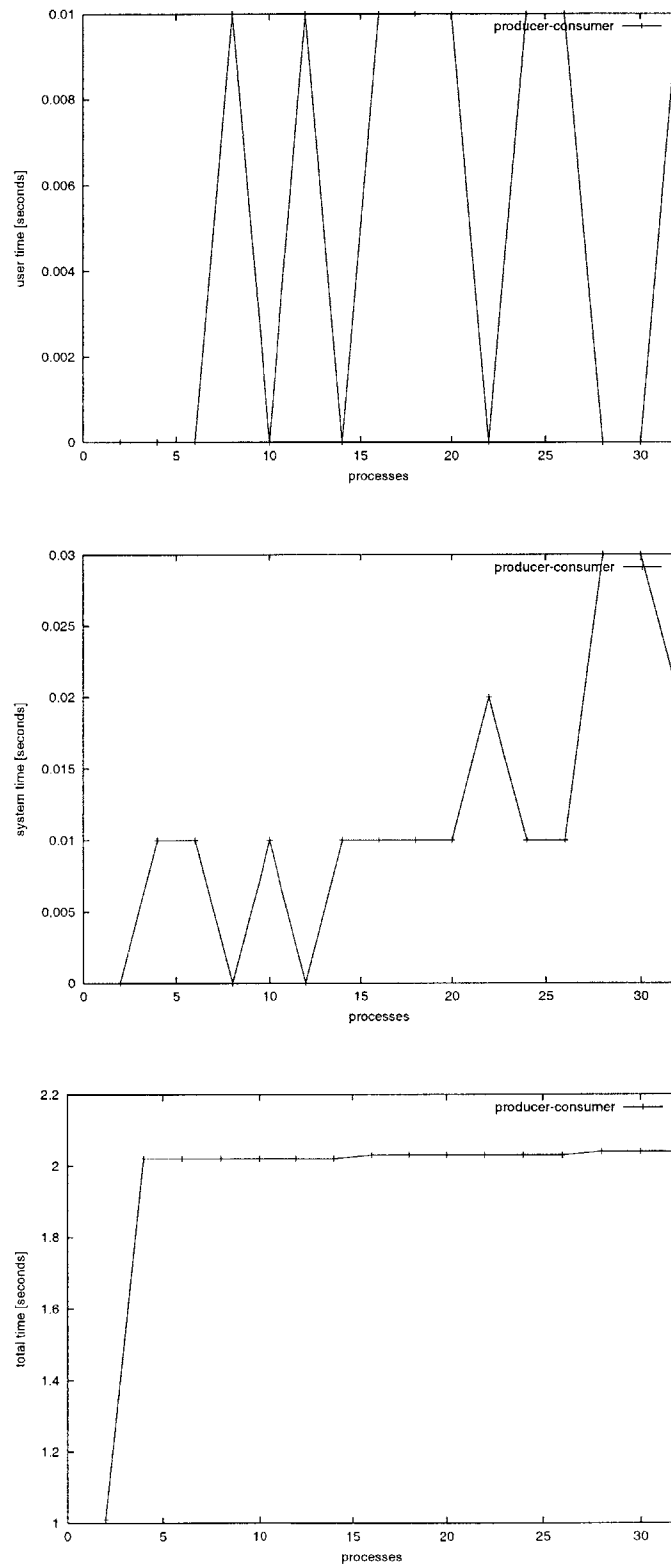


Figure 4-16: Producer-consumer benchmark overhead on Simics single processor machine.

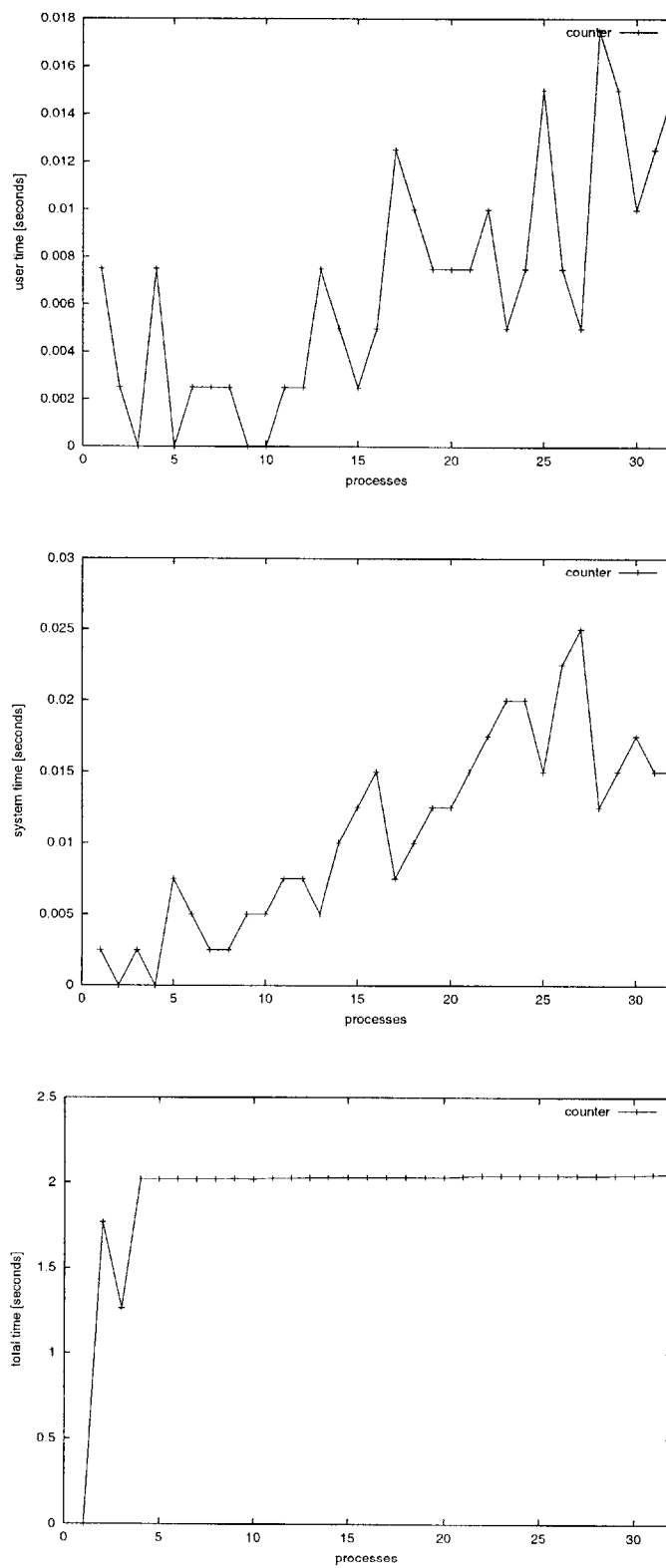


Figure 4-17: Counter benchmark overhead on Simics dual processor machine.

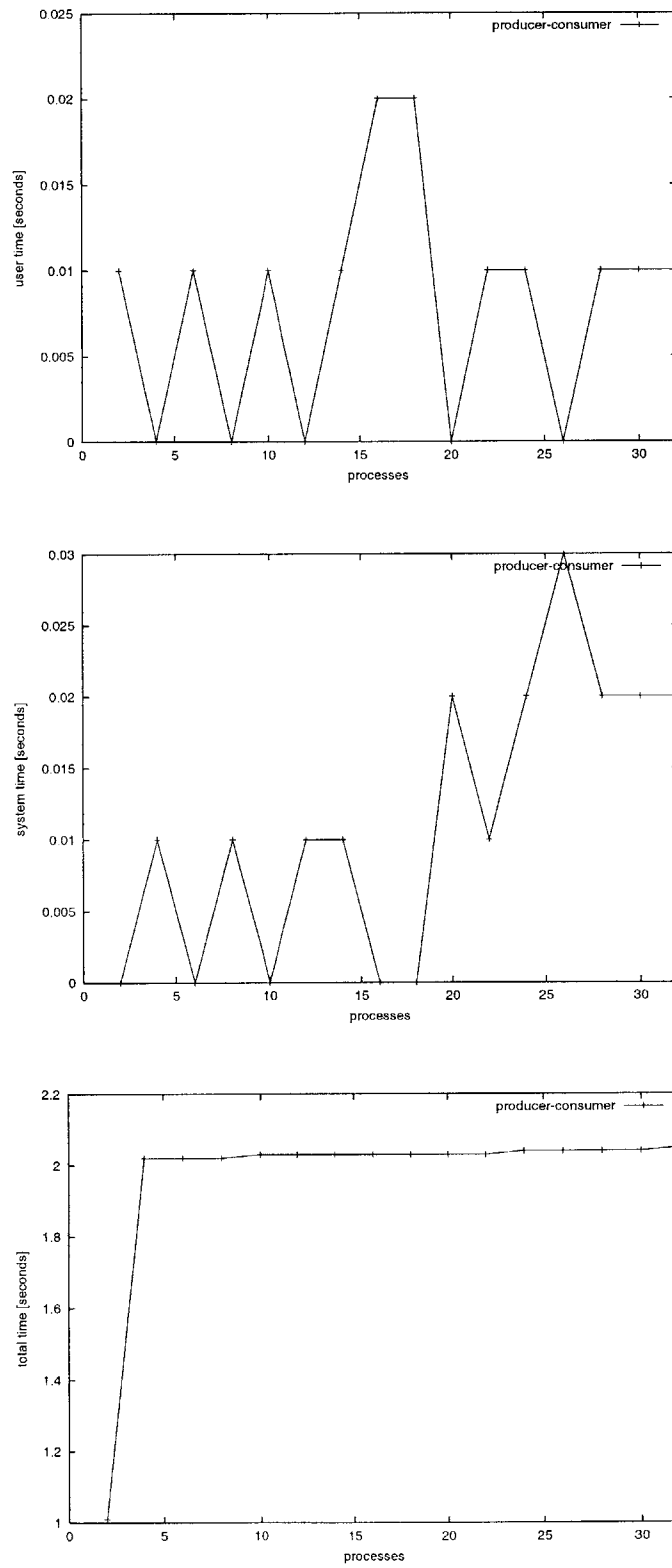


Figure 4-18: Producer-consumer benchmark overhead on Simics dual processor machine.

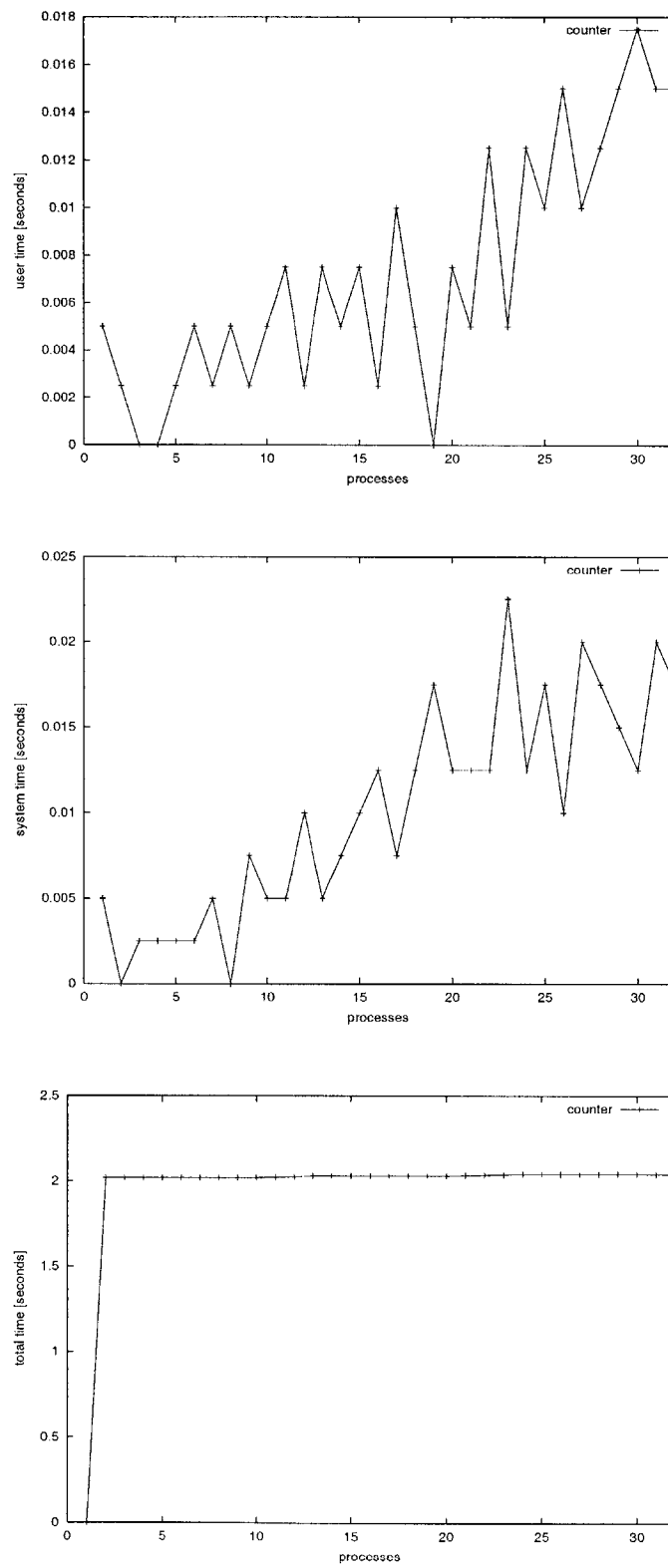


Figure 4-19: Counter benchmark overhead on Simics eight processor machine.

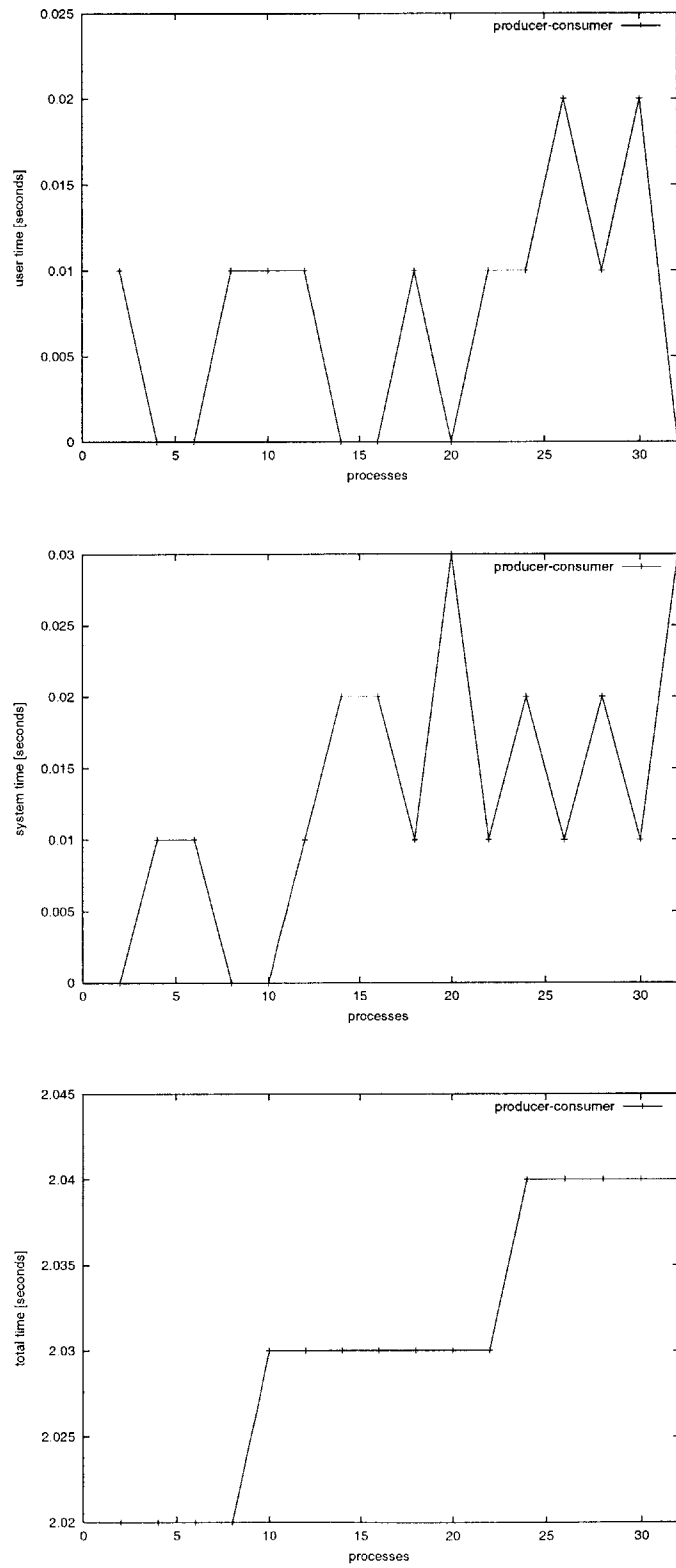


Figure 4-20: Producer-consumer benchmark overhead on Simics eight processor machine.

4.5 Comparison of Critical Section and Transaction with Equal Number of Instructions

The simulation of the transactional instructions makes the transaction longer than it will be on a real machine. Therefore, the total running times of transactional memory would be lower if there was no overhead in simulating transactional instructions. In this section, we show the results of an experiment where we compared the results of a benchmark using different synchronization schemes but having the identical number of instructions in its critical section or transaction.

Using `objdump`, we disassembled the yield lock and transactional memory versions of the counter benchmarks. We counted the number of x86 instructions required to increment the counter by one. For the yield lock, successfully acquiring the lock on the first attempt takes 14 assembly instructions. The critical section of incrementing the lock is 15 instructions and releasing the lock takes 6 instructions. The total number of x86 assembly instructions is 35. For transactional memory, a transaction that successfully increments the counter by one takes 62 x86 assembly instructions. A single transaction in the counter benchmark uses six transactional memory instructions. However, simulating those six instructions takes 28 x86 assembly instructions. Extra memory instructions are also added due to register spills. We added 27 `nop` instructions to the critical section of the yield lock version of the benchmark to equalize the number instruction it takes to increment the counter by one. We simulated this modified version of the benchmark and compared the results to the transactional memory version from Section 4.3.

Figure 4-21, Figure 4-22, and Figure 4-23 show the results. The only result adding extra `nop` instructions had was increasing the total times of the yield lock benchmark. The shape of the graphs remained the same. The yield lock is no longer better than transactional memory. In the single processor case, transactional memory performs 0.1% worse than the yield lock. In the dual processor case, transactional memory performs 1.4% better than the yield lock, and in the eight processor case, transactional memory performs 2.7% better than the yield lock. The graphs show that as the number of processors increase, the advantage of transactional memory over the yield lock also increases.

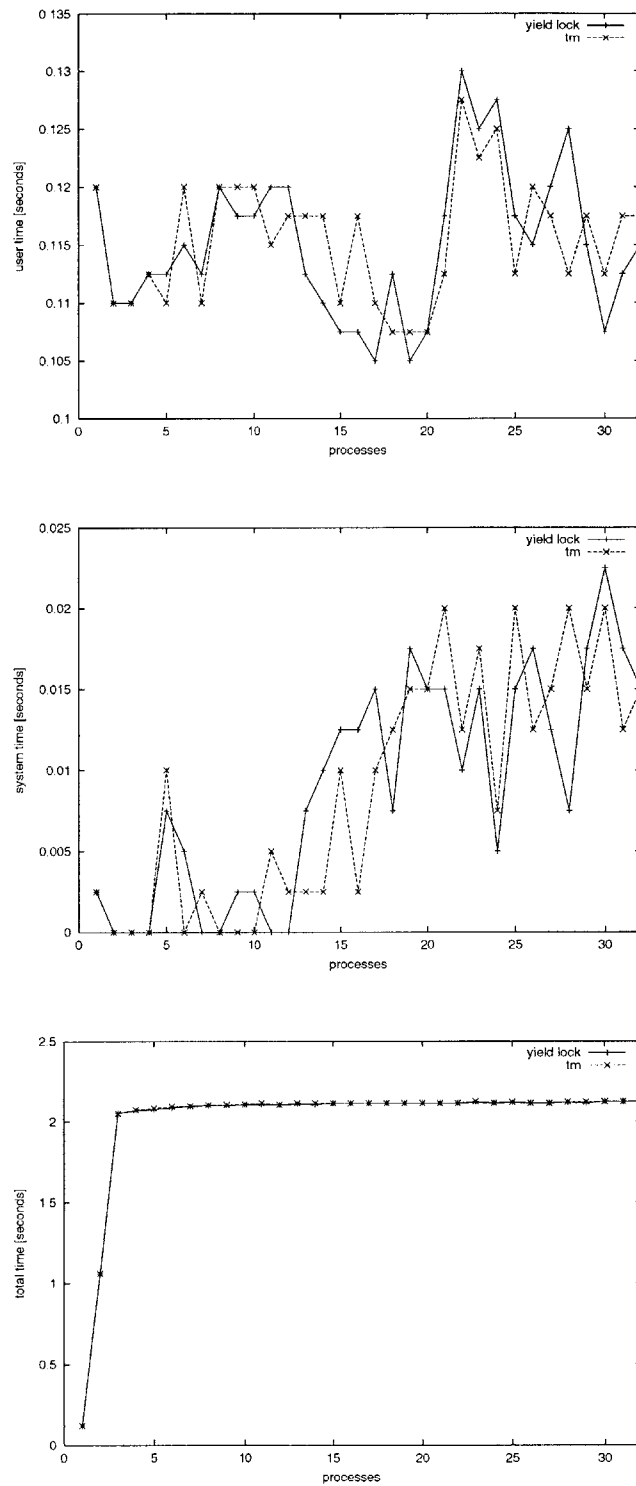


Figure 4-21: Results of counter benchmark with additional `nop` instructions using yield lock and counter benchmark using transactional memory from Simics single processor machine.

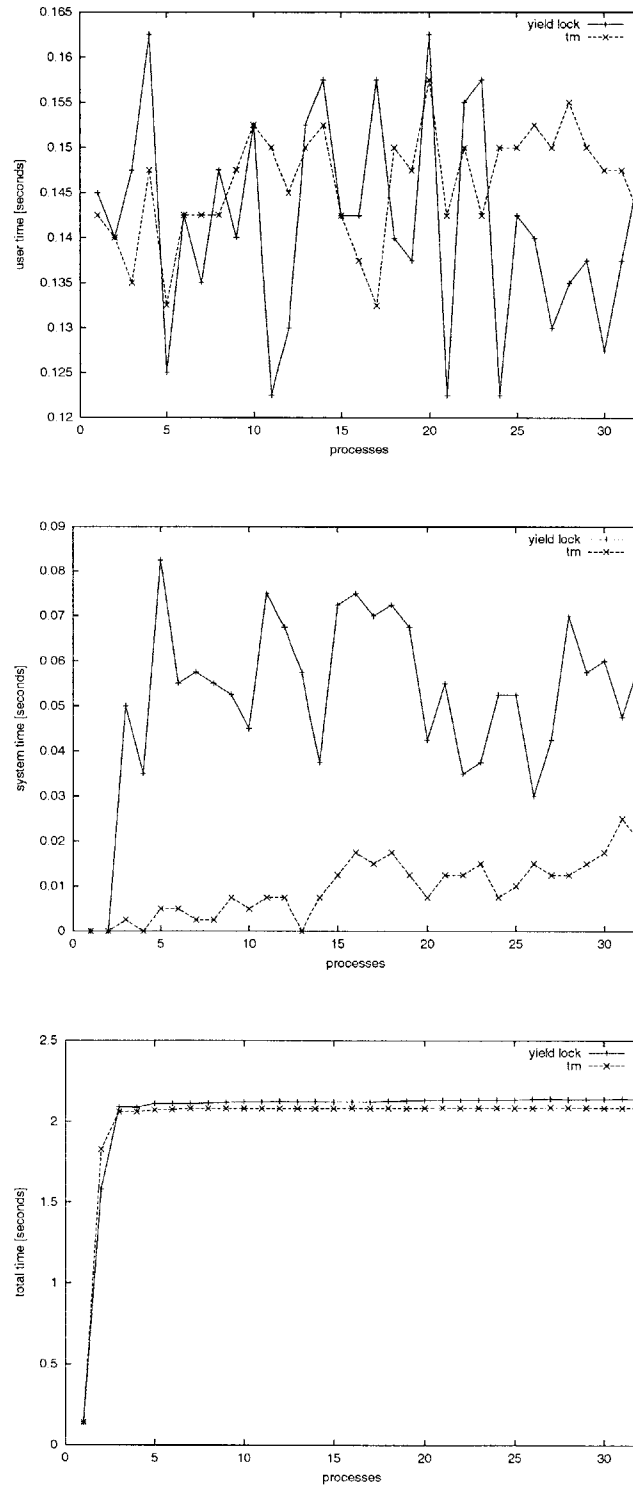


Figure 4-22: Results of counter benchmark with additional nop instructions using yield lock and counter benchmark using transactional memory from Simics dual processor machine.

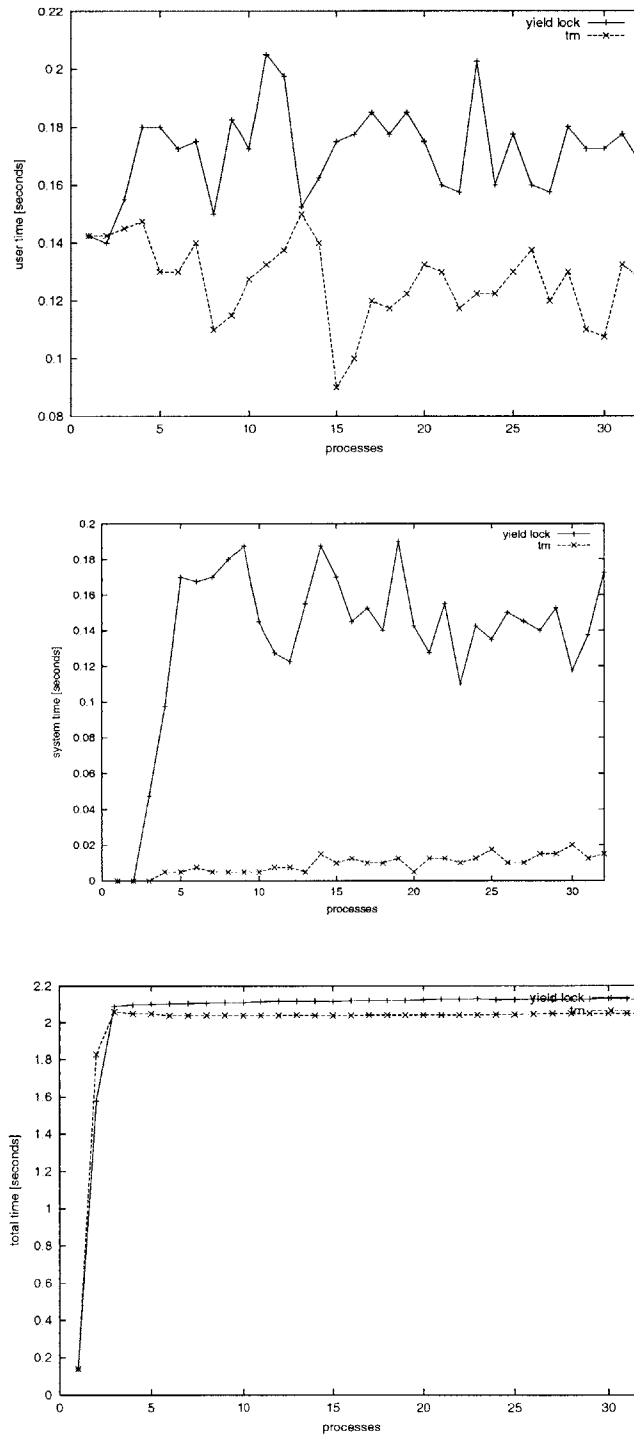


Figure 4-23: Results of counter benchmark with additional `nop` instructions using yield lock and counter benchmark using transactional memory from Simics eight processor machine.

	Simics virtual machine	<code>storm.lcs.mit.edu</code>
Number of processors	1	1
Processor model	x86-p3	AMD Athlon K7
Clock speed	600 MHz	600 MHz
Memory	512 MB	512 MB
Operating system	Red Hat Linux 7.3	Red Hat Linux 7.2
Linux kernel	2.4.18-3	2.4.7-10

Table 4.4: Configuration comparison between Simics single processor virtual machine and `storm.lcs.mit.edu`.

4.6 Simics Experimental Error

Since Simics is not a cycle-accurate simulator, we ran the spin lock and yield lock versions of our benchmarks on actual machines to obtain an estimate of experimental error. In this section, we present the results of our Simics accuracy experiments for the single processor and dual processor virtual machine.

The graphs of experimental runs from Simics machines are taken from Section 4.3. For data from runs on actual machines, each counter benchmark graph is created from the average of 100 runs and each producer-consumer graph is created from the average of 20 runs. In addition, we ran and plotted the results of the benchmarks for up to 64 processes. We present results in both average difference and average absolute difference, using the graphs from actual machines as the references.

4.6.1 Single Processor Machines Comparison

We used `storm.lcs.mit.edu` as our reference single processor machine (Table 4.4). `simics-1.6.4` does not have a processor model for the AMD Athlon K7 so we use the processor model for the Intel Pentium III instead.

Figure 4-24 and Figure 4-25 shows the results for the counter benchmark. In both cases, the slope of the system time graph is much greater in the results from `storm`. The total time given by the Simics machine, however, closely matches the results of `storm`. For the spin lock, the average difference is 1.9% and the average absolute difference is 6.0%. For the yield lock, the average difference is -2.3% and the average absolute difference is also

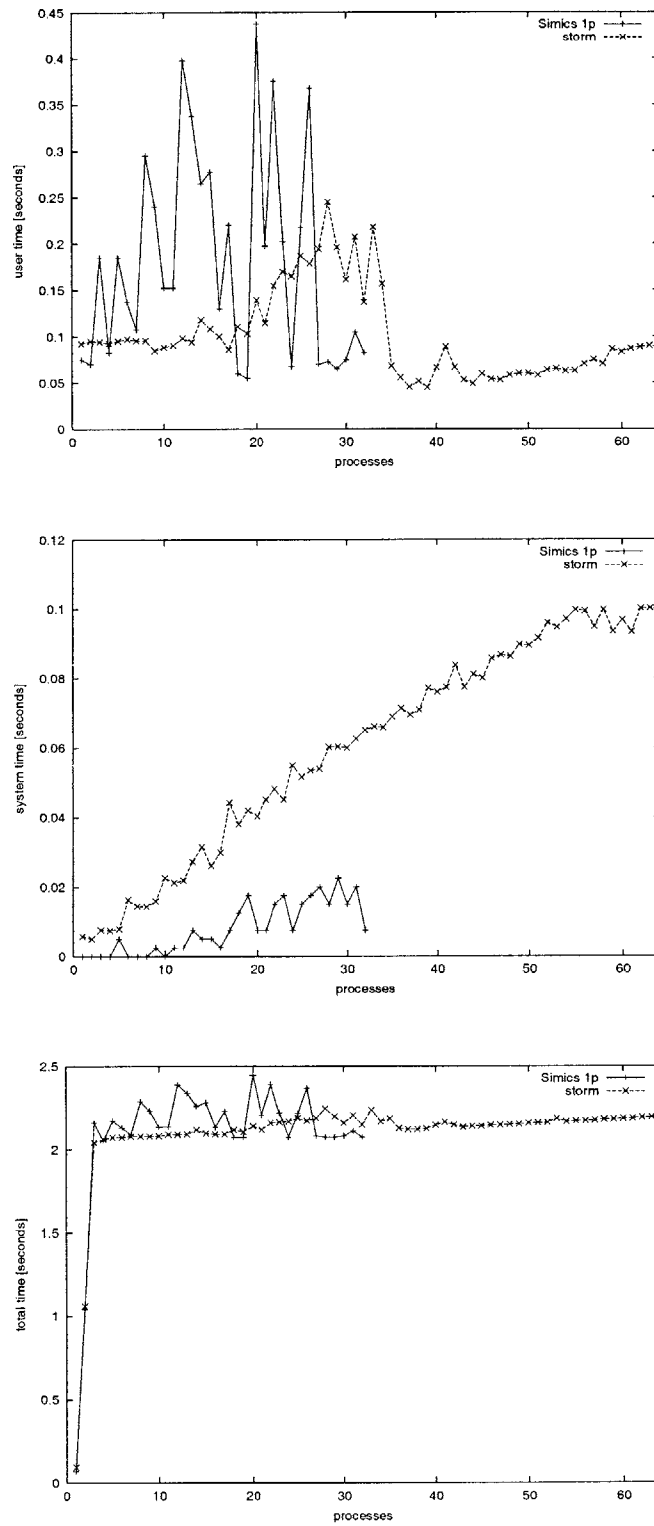


Figure 4-24: Comparison of results of counter benchmark using spin lock from Simics single processor machine and `storm.lcs.mit.edu`.

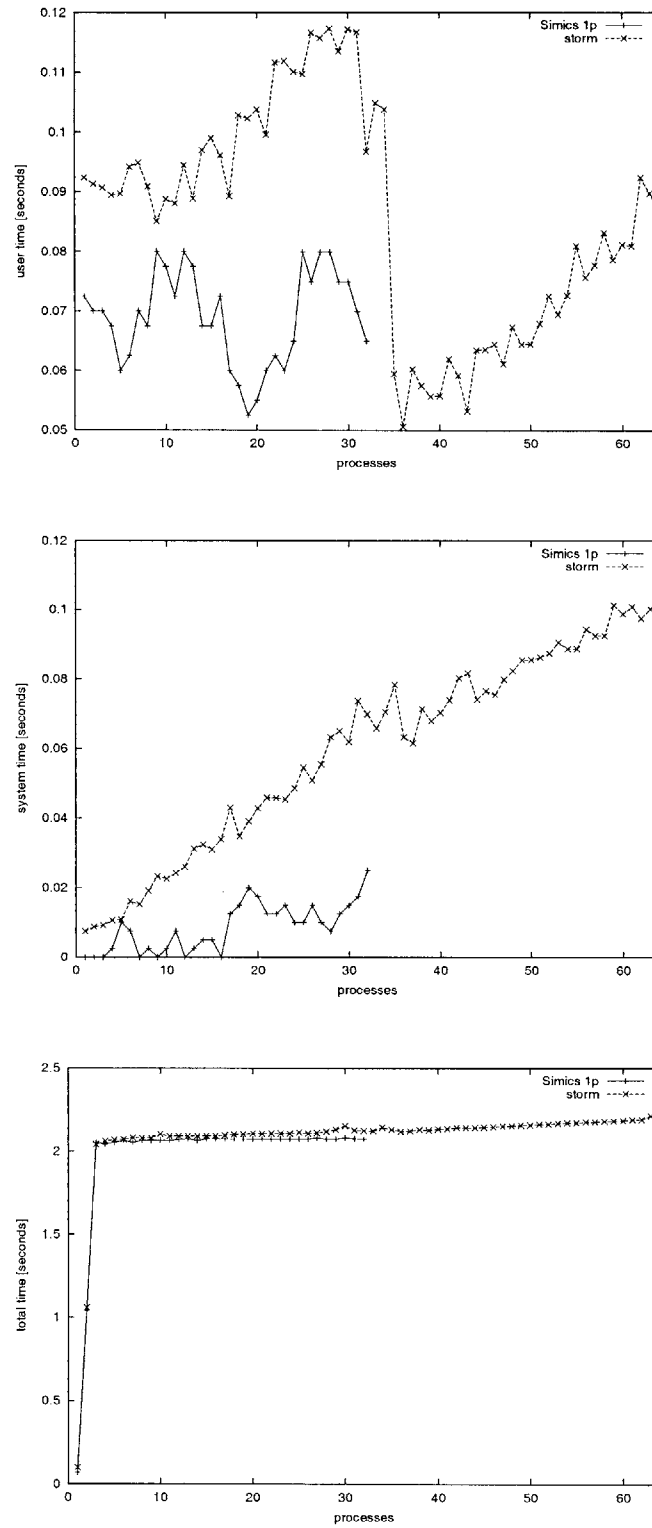


Figure 4-25: Comparison of results of counter benchmark using yield lock from Simics single processor machine and `storm.lcs.mit.edu`.

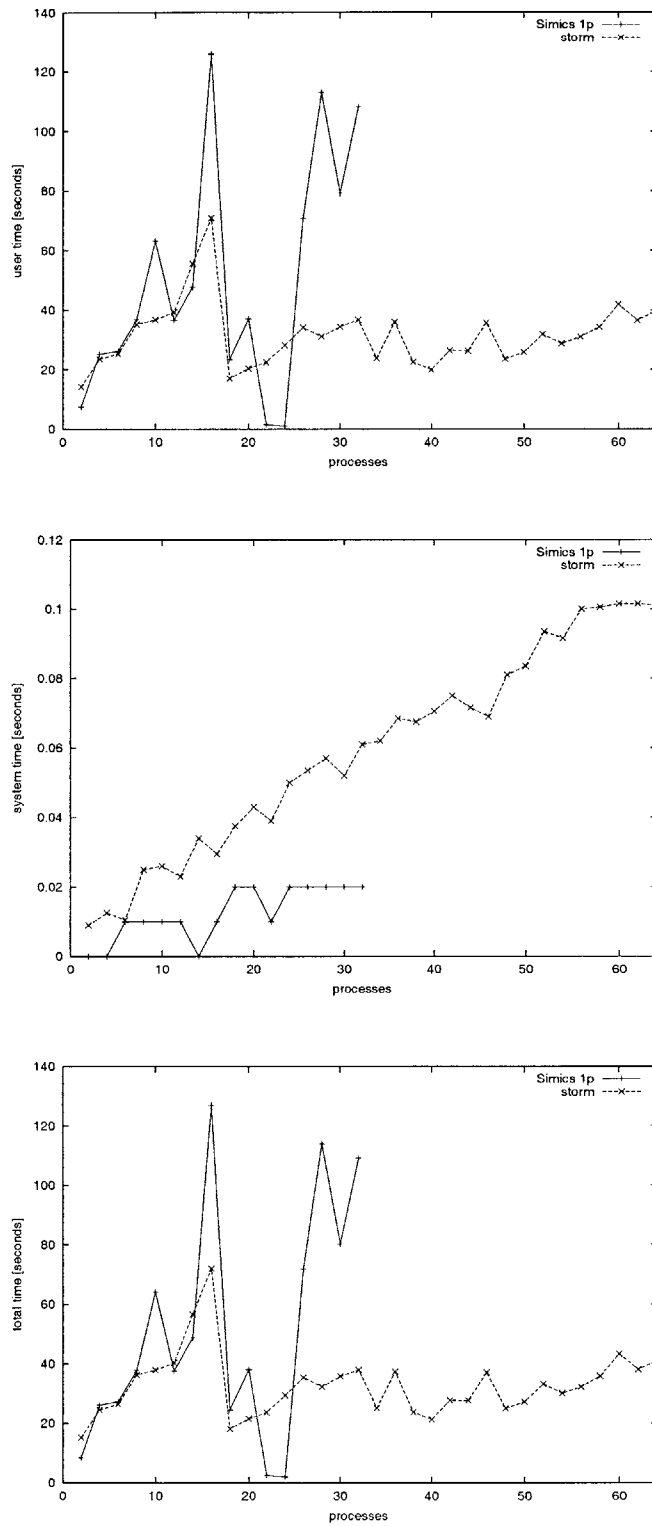


Figure 4-26: Comparison of results of producer-consumer benchmark using spin lock from Simics single processor machine and `storm.lcs.mit.edu`.

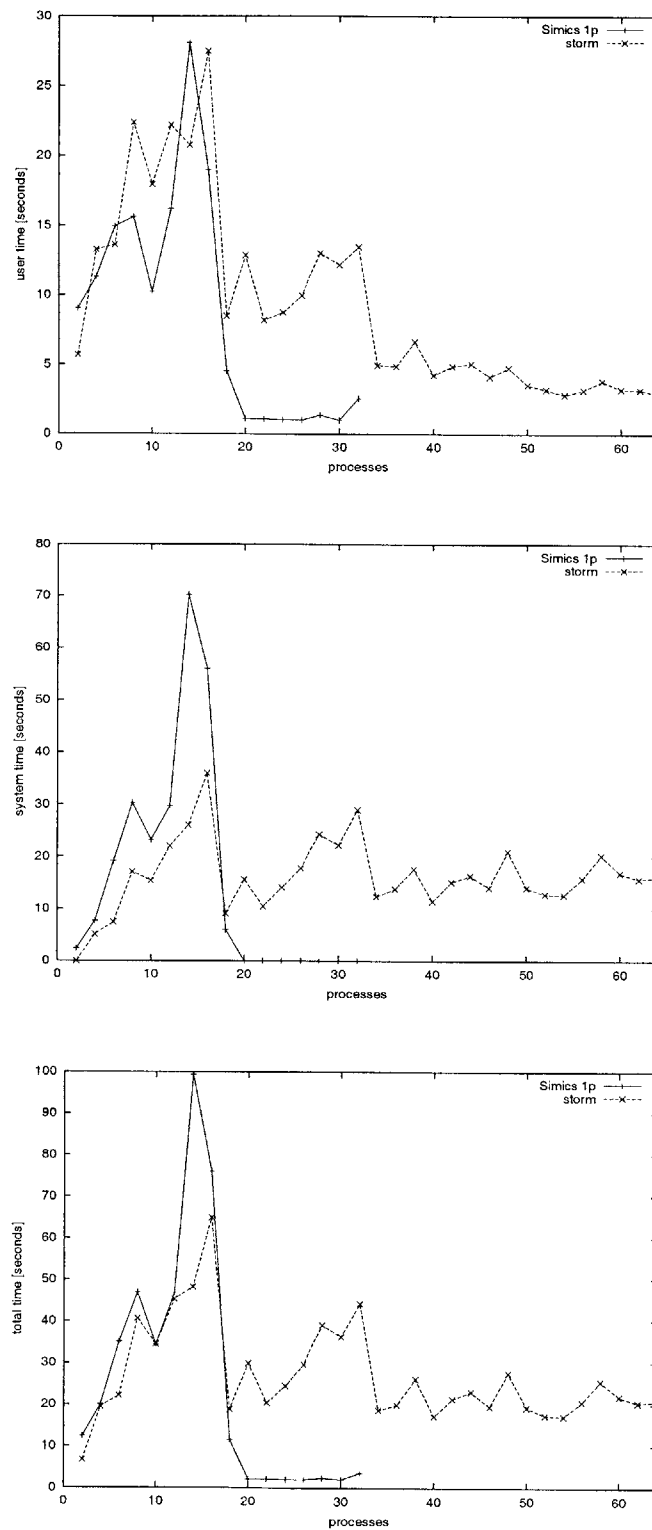


Figure 4-27: Comparison of results of producer-consumer benchmark using yield lock from Simics single processor machine and `storm.lcs.mit.edu`.

	Simics virtual machine	<code>dosx.lcs.mit.edu</code>
Number of processors	2	2
Processor model	x86-p3	Pentium III Katmai
Clock speed	500 MHz	500 MHz
Memory	256 MB	256 MB
Operating system	Red Hat Linux 7.3	Debian
Linux kernel	2.4.18-3smp	2.4.18 #6 SMP

Table 4.5: Configuration comparison between Simics dual processor virtual machine and `dosx.lcs.mit.edu`.

2.3%.

Figure 4-24 and Figure 4-25 show the results for the producer-consumer benchmark. For the spin lock, the average difference is 43% and the average absolute difference is 74%. For the yield lock, the average difference is -25% and the average absolute difference is 61%.

4.6.2 Dual Processor Machine Comparison

We used `dosx.lcs.mit.edu` as our reference dual processor machine (Table 4.4). `dosx` has two identical Pentium III processors, each operating at 500 MHz.

Figure 4-28 shows the results of the counter benchmark using a spin lock. The Simics machine shows a constant user time and system time across all number of processes. Results from `dosx` show that as the number of processes increases, both user time and system time increase linearly. For the total time, the Simics machine has a constant time starting from three processes whereas the real machine shows a linear increase in total time as the number of processes increases. Comparing the two total time graphs, the average difference is -49% and the average absolute difference is 52%.

Figure 4-29 shows the results of the counter benchmark using a yield lock. For system time, the results from `dosx` show a faster increase in comparison to the results from Simics. The graphs of total time from both machines are similar except that `dosx` consistently has a greater total time than Simics. The average difference is -6.3% and the average absolute difference is also 6.3%.

Figure 4-30 shows the results of the producer-consumer benchmark using a spin lock.

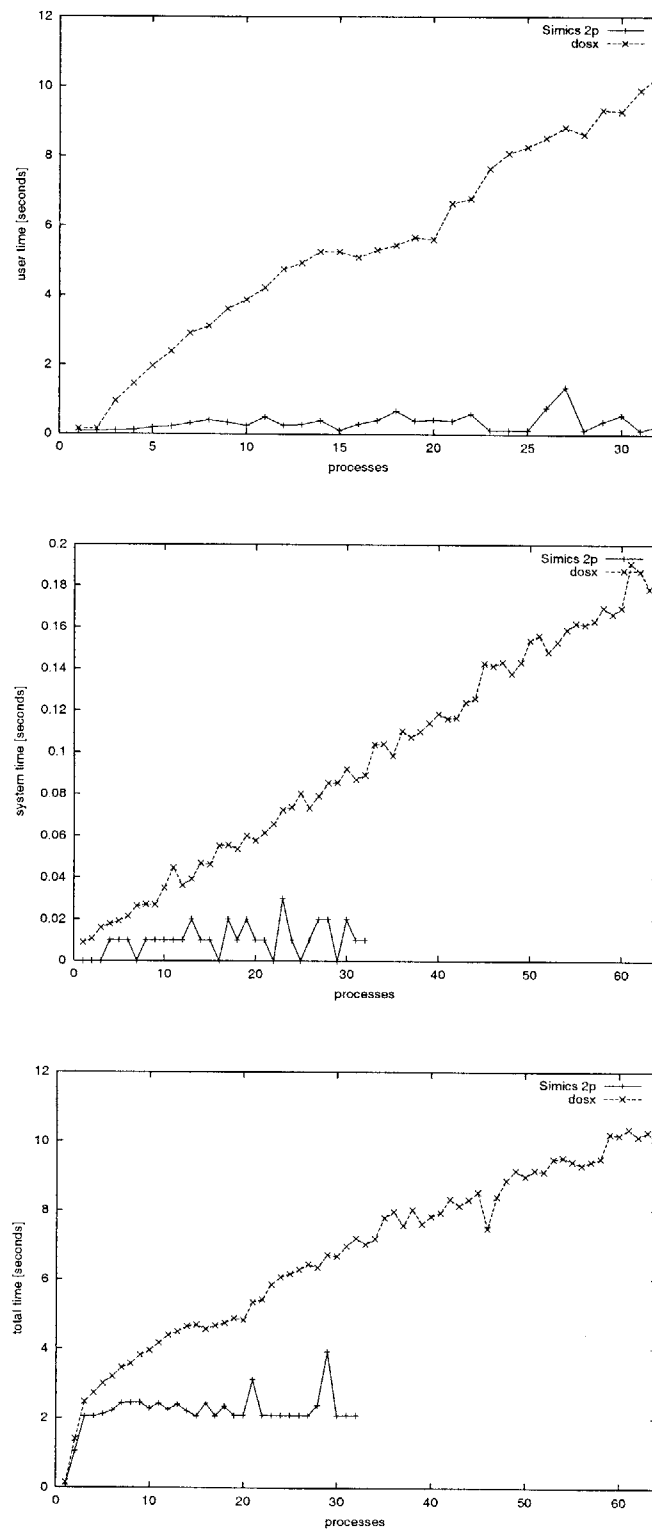


Figure 4-28: Comparison of results of counter benchmark using spin lock from Simics dual processor machine and `dosx.lcs.mit.edu`.

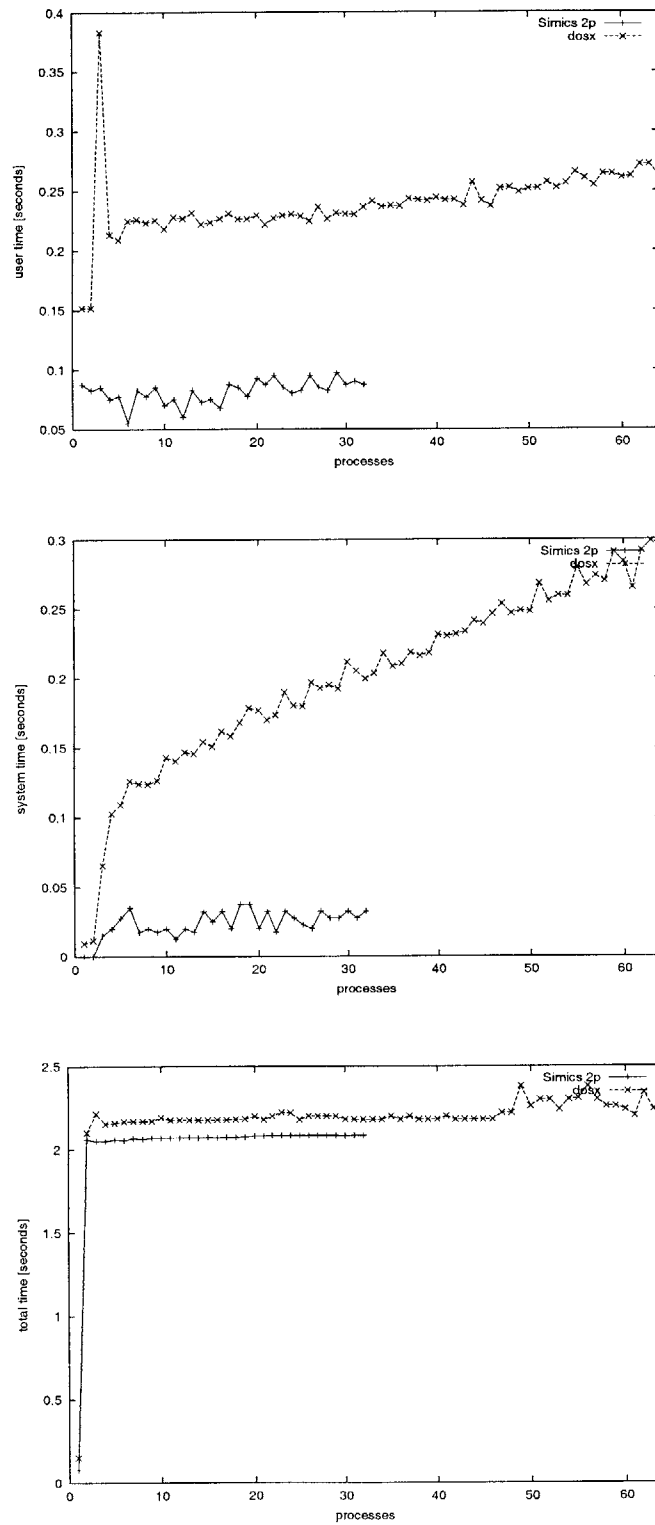


Figure 4-29: Comparison of results of counter benchmark using yield lock from Simics dual processor machine and `dosx.lcs.mit.edu`.

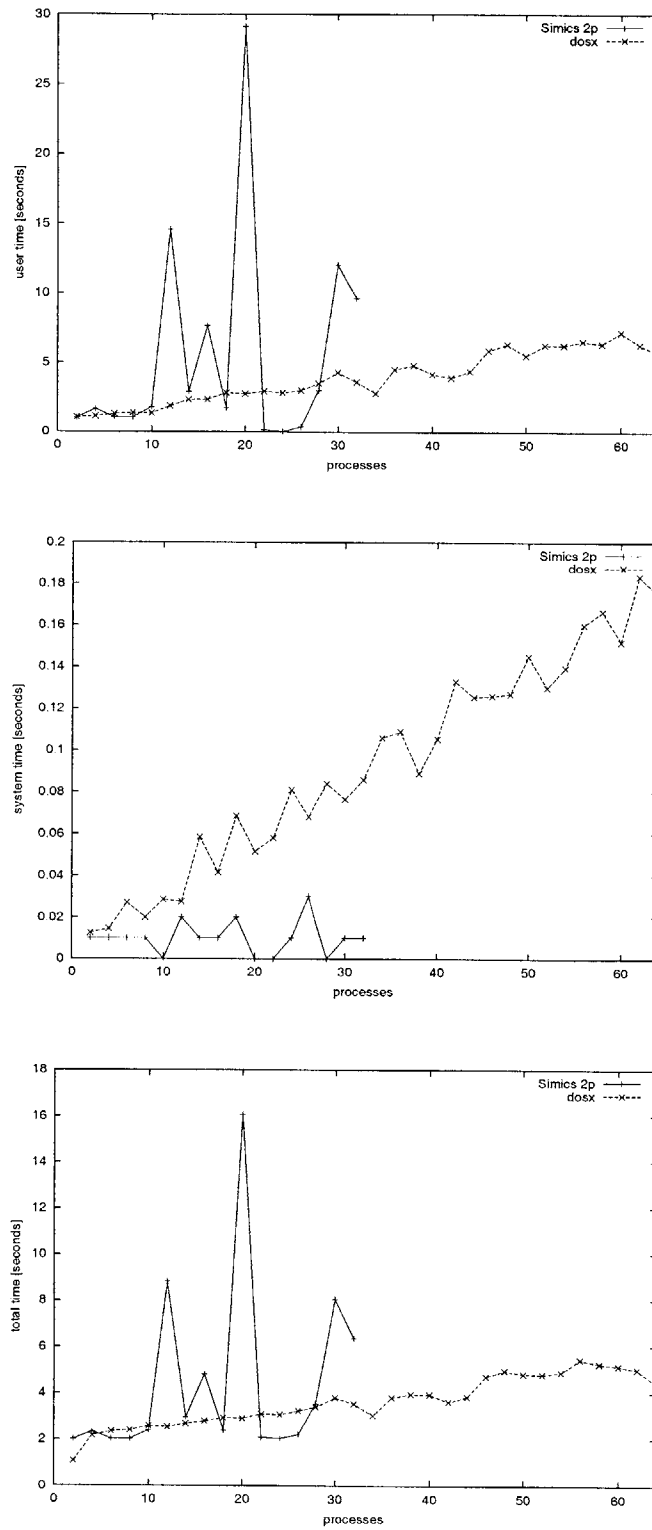


Figure 4-30: Comparison of results of producer-consumer benchmark using spin lock from Simics dual processor machine and `dosx.lcs.mit.edu`.

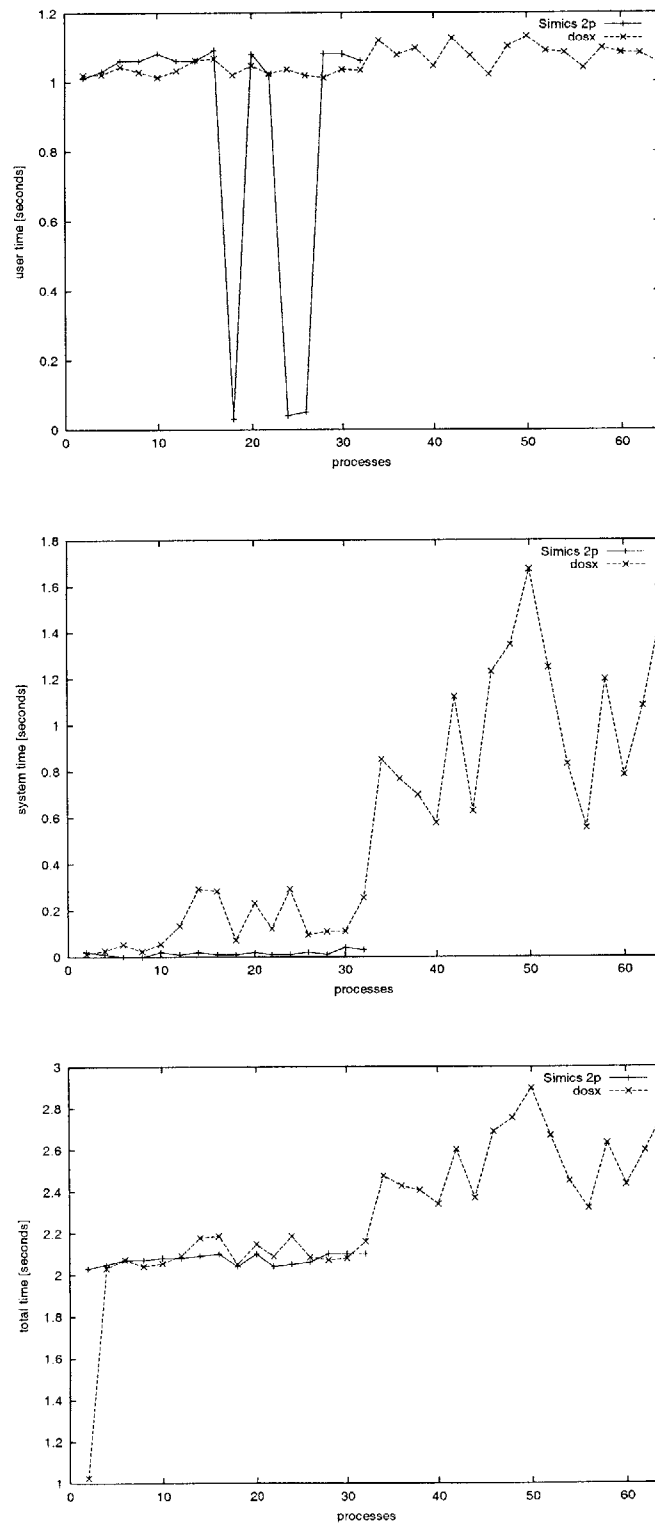


Figure 4-31: Comparison of results of producer-consumer benchmark using yield lock from Simics dual processor machine and `dosx.lcs.mit.edu`.

For some number of cases, the Simics machine gives a much greater user time and total time than `dosx`. The average difference is 58% and the average absolute difference is 77%.

Figure 4-31 shows the results of the producer-consumer benchmark using a yield lock. The two total time graphs are similar. The average difference is 5.0% and the average absolute difference is 8.0%.

4.7 Discussion

Additional experiments show that the percentage difference of results from our simulation framework and from real machines is greater than the percentage improvement of transactional memory over yield locks when the size of a transaction and critical section are equal. Therefore, our simulation framework is not accurate enough to distinguish the performance of those two synchronization schemes. Our simulation framework is useful to test the functionality of programs written with transactional instructions and to provide a rough estimate of their performance. However, we cannot reliably conclude that transactional memory is better than yield locks.

The initial results showed that adding exponential backoff to transactional memory decreased performance because the additional overhead was high. To calculate the amount for backoff, we used the `random` function, which requires many instructions. Benchmarks using spin locks showed that convoying is often a problem. A process is often descheduled while holding the lock, preventing any active process from progressing.

When the size of the critical section and the size of the transaction are equal, our results show that transactional memory performs better than yield locks. The advantage of transactional memory over yield locks increases as the number of processors increase. For a program using yield locks in a multiprocessor system, only the process holding the lock progress. All other active processes are yielding. In transactional memory, all processes can progress as long as they do not conflict. In summary, our results show that transactions never perform worse than yield locks.

Chapter 5

Conclusion

This thesis proposed an instruction set architecture extension to support hardware transactions. With these instructions, programmers can write critical sections that normally require locks as transactions. The transactional instructions allow a process to have multiple active transactions and a transaction to explicitly add variables to or remove variables from its data set.

There are some trade-offs in using transactions over locks. Transactions avoids the problems of locks, such as deadlock, convoying, priority inversion, and reduction in parallelism. Transactions are also less prone to programmer errors. A major problem in large programs which use locks is that programmers make mistakes keeping track of which locks protect which data structures. The primary problem of transactions is livelock, which can be handled in software.

We presented a simulation framework built upon Simics that allows programmers to write, compile, and simulate programs using the transactional instructions. Using our simulation framework, we compared the performance of benchmarks using transactional instructions to conventional locking schemes. The results from our simulations show that transactional memory and yield locks have comparable performance. Benchmarks using spin locks suffered from convoying. On a single processor system, results showed that transactional memory offers no advantage over yield locks. As the number of processors increased, the advantage of transactional memory over yield locks also improves.

Additional experiments also showed that results from our simulation framework have

error rates that are higher than the percentage improvement of transactional memory over yield locks. Therefore, we cannot reliably conclude programs using our transactional instructions perform better than the same programs using yield locks. Nevertheless, our simulation framework provides a method to test the functionality and obtain an estimate of the performance of programs written with transactional instructions.

Appendix A

Source Code

A.1 Transactional Memory API

A.1.1 transactional-memory_api.h

```
/*
    Transactional Memory interface

    protocol
    %eax - opcode
    %ebx - dest
    %ecx - src
    %edx - tag
*/

#ifndef _SIMICS_TRANSACTIONAL_MEMORY_API_H
#define _SIMICS_TRANSACTIONAL_MEMORY_API_H

#include "simics-magic-instruction.h"

#define OPCODE_RESERVE          0x0DEAD000
#define OPCODE_RESERVEC        0x0DEAD001
#define OPCODE_RELEASE          0x0DEAD002
#define OPCODE_LT               0x0DEAD003
#define OPCODE_ST               0x0DEAD004
#define OPCODE_BEGIN           0x0DEAD005
#define OPCODE_COMMIT          0x0DEAD006
#define OPCODE_VALIDATE         0x0DEAD007

#define RESERVE(addr, tag) \
    ({ unsigned int __result; \
      unsigned int *__addr = (addr); \
      unsigned int *__tag = (tag); \
      asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%ecx; movl %3,%edx" :: \
        "g" (OPCODE_RESERVE), "g" (&__result), "g" (__addr), "g" (__tag): \
        "eax", "ebx", "ecx", "edx"); \
      MAGIC_BREAKPOINT; \
      __result; })

#define RESERVEC(addr, tag) \
    ({ unsigned int __result; \
```

```

    unsigned int *__addr = (addr); \
    unsigned int *__tag = (tag); \
    asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%ecx; movl %3,%edx" :: \
        "g" (OPCODE_RESERVEC), "g" (&__result), "g" (__addr), "g" (__tag): \
        "eax", "ebx", "ecx", "edx"); \
    MAGIC_BREAKPOINT; \
    __result; ))

#define RELEASE(addr, tag) \
    ({ unsigned int __result; \
        unsigned int *__addr = (addr); \
        unsigned int *__tag = (tag); \
        asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%ecx; movl %3,%edx" :: \
            "g" (OPCODE_RELEASE), "g" (&__result), "g" (__addr), "g" (__tag): \
            "eax", "ebx", "ecx", "edx"); \
        MAGIC_BREAKPOINT; \
        __result; })

#define BEGINT(tag) \
    ({ unsigned int __result; \
        unsigned int *__tag = (tag); \
        asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%edx" :: \
            "g" (OPCODE_BEGINT), "g" (&__result), "g" (__tag): \
            "eax", "ebx", "edx"); \
        MAGIC_BREAKPOINT; \
        __result; })

#define COMMIT(tag) \
    ({ unsigned int __result; \
        unsigned int *__tag = (tag); \
        asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%edx" :: \
            "g" (OPCODE_COMMIT), "g" (&__result), "g" (__tag): \
            "eax", "ebx", "edx"); \
        MAGIC_BREAKPOINT; \
        __result; })

#define LT(addr, tag) \
    ({ unsigned int __result; \
        unsigned int *__addr = (addr); \
        unsigned int *__tag = (tag); \
        asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%ecx; movl %3,%edx" :: \
            "g" (OPCODE_LT), "g" (&__result), "g" (__addr), "g" (__tag): \
            "eax", "ebx", "ecx", "edx"); \
        MAGIC_BREAKPOINT; \
        __result; })

#define ST(addr, value, tag) \
    ({ unsigned int *__addr = (addr); \
        unsigned int __value = (value); \
        unsigned int *__tag = (tag); \
        asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%ecx; movl %3,%edx" :: \
            "g" (OPCODE_ST), "g" (__addr), "g" (__value), "g" (__tag): \
            "eax", "ebx", "ecx", "edx"); \
        MAGIC_BREAKPOINT; \
    })

#define VALIDATE(tag) \
    ({ unsigned int __result; \
        unsigned int *__tag = (tag); \
        asm volatile ("movl %0,%eax; movl %1,%ebx; movl %2,%edx" :: \
            "g" (OPCODE_VALIDATE), "g" (&__result), "g" (__tag): \
            "eax", "ebx", "edx"); \
        MAGIC_BREAKPOINT; \
        __result; })

#endif /* _SIMICS_TRANSACTIONAL_MEMORY_API_H */

```

A.2 Simics Transactional-Memory Module

A.2.1 transactional-memory.h

```
#define TM_LINES 8 // number of entries in transactional memory

// possible values of a transaction tag
#define TAG_NONE      0x0
#define TAG_ACTIVE    0x1
#define TAG_COMMITTED 0x2
#define TAG_ABORTED   0x3

// transactional memory line
typedef struct {
    unsigned int valid;      // entry in use?
    unsigned int modified;   // data modified?
    physical_address_t pa;   // physical address
    physical_address_t tag_pa; // physical address of tag
    integer_t data;
} tm_line_t;
```

A.2.2 transactional-memory.c

```
#include "first.h"
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include "global.h"
#include "simics_api.h"
#include "simmalloc.h"
#include "c-utils.h"
#include "transactional-memory_api.h"
#include "transactional-memory.h"

/* struct for the transactional-memory-class */
typedef struct {
    conf_object_t obj;
    timing_model_interface_t *timing_interface;
    int tm_enabled;      // enable transactional memory
    tm_line_t *tm;       // transactional memory buffer
    int num_trans;       // number of transaction slots used
} transactional_memory_object_t;

timing_model_interface_t *transactional_memory_timing_interface;
static int mm_id = -1;

static void
tm_handler(const conf_object_t *obj)
{
    transactional_memory_object_t *tm_obj = (transactional_memory_object_t *)obj;

    if (!tm_obj->tm_enabled) {
        return;
    }

    processor_t *cpu = SIM_current_processor();
    int eax_no = SIM_get_register_number((conf_object_t *)cpu, "eax");
    int ebx_no = SIM_get_register_number((conf_object_t *)cpu, "ebx");
    int ecx_no = SIM_get_register_number((conf_object_t *)cpu, "ecx");
    int edx_no = SIM_get_register_number((conf_object_t *)cpu, "edx");
```

```

uinteger_t opcode = SIM_read_register((conf_object_t *)cpu, eax_no);
uinteger_t dest_la = SIM_read_register((conf_object_t *)cpu, ebx_no);
uinteger_t tag_la = SIM_read_register((conf_object_t *)cpu, edx_no);

physical_address_t dest_pa = SIM_logical_to_physical(cpu, Sim_DI_Data, dest_la);
physical_address_t tag_pa = SIM_logical_to_physical(cpu, Sim_DI_Data, tag_la);
integer_t tag_value = SIM_read_phys_memory(cpu, tag_pa, 4);

int i;
int hit;
int hit_index;
int result;
int tag_match;
logical_address_t la;
physical_address_t pa;
integer_t value;

switch (opcode) {
case OPCODE_RESERVE:
case OPCODE_RESERVEVEC:
    result = FALSE;
    la = SIM_read_register((conf_object_t *)cpu, ecx_no);
    pa = SIM_logical_to_physical(cpu, Sim_DI_Data, la);

    if (tag_value == TAG_ACTIVE) {
        // check if already reserved
        hit = FALSE;
        hit_index = 0xdeadbeef;
        tag_match = FALSE;

        if (tm_obj->num_trans != 0) {
            for (i=0; i<TM_LINES; i++) {
                if ((tm_obj->tm[i].valid == TRUE) && (tm_obj->tm[i].pa == pa)) {
                    hit = TRUE;
                    hit_index = i;

                    if (tm_obj->tm[i].tag_pa == tag_pa) {
                        tag_match = TRUE;
                    }
                    break;
                }
            }
        }

        if (hit == FALSE) { // if not reserved, find empty slot and reserve
            for (i=0; i<TM_LINES; i++) {
                if (!tm_obj->tm[i].valid) {
                    tm_obj->tm[i].valid = TRUE;
                    tm_obj->tm[i].modified = FALSE;
                    tm_obj->tm[i].pa = pa;
                    tm_obj->tm[i].data = SIM_read_phys_memory(cpu, pa, 4);
                    tm_obj->tm[i].tag_pa = tag_pa;
                    tm_obj->num_trans++;
                    result = TRUE;
                    break;
                }
            }
        }

        if (result == FALSE) {
            // abort transaction
            SIM_printf("Warning! Transaction aborted due to lack of space\n");
            SIM_printf("Trying to reserve PA:%x tag_pa:%x\n", pa, tag_pa);
            SIM_write_phys_memory(cpu, tag_pa, TAG_ABORTED, 4);
            SIM_printf("valid modified pa tag_pa data tag_status\n");
            for (i=0; i<TM_LINES; i++) {
                SIM_printf("%d %d 0x%x 0x%x %d %d\n",
                    tm_obj->tm[i].valid,

```



```

        tm_obj->tm[i].modified,
        tm_obj->tm[i].pa,
        tm_obj->tm[i].tag_pa,
        tm_obj->tm[i].data,
        SIM_read_phys_memory(cpu, tm_obj->tm[i].tag_pa, 4));
    }
}
) else if ((hit) && (!tag_match) && (opcode == OPCODE_RESERVE)) {
    // reserved by another transaction so abort the other transaction
    SIM_write_phys_memory(cpu, tm_obj->tm[hit_index].tag_pa, TAG_ABORTED, 4);
    tm_obj->tm[hit_index].tag_pa = tag_pa;

    if (tm_obj->tm[hit_index].modified) {
        tm_obj->tm[hit_index].data = SIM_read_phys_memory(cpu, pa, 4);
        tm_obj->tm[hit_index].modified = FALSE;
    }

    result = TRUE;
}
}

SIM_write_phys_memory(cpu, dest_pa, result, 4);
break;

case OPCODE_RELEASE:
    result = FALSE;
    la = SIM_read_register((conf_object_t *)cpu, ecx_no);
    pa = SIM_logical_to_physical(cpu, Sim_DI_Data, la);

    for (i=0; i<TM_LINES; i++) {
        if ((tm_obj->tm[i].valid) && (tm_obj->tm[i].pa == pa)) {
            if (tm_obj->tm[i].tag_pa == tag_pa) {
                result = TRUE;
                tm_obj->tm[i].valid = FALSE;
                tm_obj->num_trans--;
            }
            break;
        }
    }

    break;

case OPCODE_BEGIN:
    result = FALSE;

    if ((tag_value == TAG_NONE) || (tag_value == TAG_COMMITTED)) {
        SIM_write_phys_memory(cpu, tag_pa, TAG_ACTIVE, 4);
        result = TRUE;
    } else if (tag_value == TAG_ABORTED) {
        // clear entries from aborted transaction
        for (i=0; i<TM_LINES; i++) {
            if ((tm_obj->tm[i].valid) && (tm_obj->tm[i].tag_pa == tag_pa)) {
                tm_obj->tm[i].valid = FALSE;
                tm_obj->num_trans--;
            }
        }
        SIM_write_phys_memory(cpu, tag_pa, TAG_ACTIVE, 4);
        result = TRUE;
    } else {
        SIM_printf("Warning! Cannot begin transaction. Transaction is already ACTIVE.\n");
    }

    SIM_write_phys_memory(cpu, dest_pa, result, 4);
    break;

case OPCODE_LT:
    la = SIM_read_register((conf_object_t *)cpu, ecx_no);
    pa = SIM_logical_to_physical(cpu, Sim_DI_Data, la);

```

```

    if (tag_value == TAG_ACTIVE) {
        for (i=0; i<TM_LINES; i++) {
            if ((tm_obj->tm[i].valid) &&
                (tm_obj->tm[i].pa == pa) &&
                (tm_obj->tm[i].tag_pa == tag_pa)) {
                SIM_write_phys_memory(cpu, dest_pa, tm_obj->tm[i].data, 4);
                break;
            }
        }
    }
    break;

case OPCODE_ST:
    value = SIM_read_register((conf_object_t *)cpu, ecx_no);

    if (tag_value == TAG_ACTIVE) {
        for (i=0; i<TM_LINES; i++) {
            if ((tm_obj->tm[i].valid) &&
                (tm_obj->tm[i].pa == dest_pa) &&
                (tm_obj->tm[i].tag_pa == tag_pa)) {
                tm_obj->tm[i].data = value;
                tm_obj->tm[i].modified = TRUE;
                break;
            }
        }
    }
    break;

case OPCODE_COMMIT:
    result = FALSE;

    if (tag_value == TAG_ACTIVE) {
        for (i=0; i<TM_LINES; i++) {
            if ((tm_obj->tm[i].valid) &&
                (tm_obj->tm[i].modified) &&
                (tm_obj->tm[i].tag_pa == tag_pa)) {
                SIM_write_phys_memory(cpu, tm_obj->tm[i].pa, tm_obj->tm[i].data, 4);
                tm_obj->tm[i].modified = FALSE;
            }
        }
        result = TRUE;
        SIM_write_phys_memory(cpu, tag_pa, TAG_COMMITTED, 4);
    }

    SIM_write_phys_memory(cpu, dest_pa, result, 4);
    break;

case OPCODE_VALIDATE:
    result = (tag_value == TAG_ACTIVE);
    SIM_write_phys_memory(cpu, dest_pa, result, 4);
    break;

default:
    SIM_printf("Warning! MAGIC INSTRUCTION called with unknown opcode %x\n", opcode);
}
}

static conf_object_t *
transactional_memory_new_instance(parse_object_t *pa)
{
    transactional_memory_object_t *obj = MM_ZALLOC(1, transactional_memory_object_t);
    SIM_object_constructor((conf_object_t *)obj, pa);
    obj->timing_interface = transactional_memory_timing_interface;
    obj->tm = MM_ZALLOC(TM_LINES, tm_line_t);

    SIM_printf("Initializing transactional memory\n");
}

```

```

SIM_hap_register_callback("Core_Magic_Instruction", (str_hap_func_t)tm_handler, obj);

return (conf_object_t *)obj;
}

/* This function is called once for every memory operation. */
static cycles_t
transactional_memory_operate(conf_object_t *obj, conf_object_t *space, map_list_t *map,
                             memory_transaction_t *mem_op)
{
    transactional_memory_object_t *tm_obj = (transactional_memory_object_t *)obj;
    generic_transaction_t *g = (generic_transaction_t *)mem_op;
    int i;

    /* We want to see future references, so make sure the STC does
       not hide them from us. */
    g->block_STC = 1;

    if (!tm_obj->tm_enabled || (tm_obj->num_trans == 0)) {
        return 0;
    }

    // check transactional memory for possible violations
    for (i=0; i<TM_LINES; i++) {
        if ((tm_obj->tm[i].valid) &&
            (tm_obj->tm[i].pa == g->physical_address)) {
            SIM_write_phys_memory(SIM_current_processor(), tm_obj->tm[i].tag_pa, TAG_ABORTED, 4);
            break;
        }
    }

    return 0;
}

static set_error_t
set_enabled_attribute(void *dont_care, conf_object_t *obj, attr_value_t *val, attr_value_t *idx)
{
    transactional_memory_object_t *tm_obj = (transactional_memory_object_t *)obj;

    if (val->kind != Sim_Val_Integer)
        return Sim_Set_Need_Integer;

    tm_obj->tm_enabled = val->u.integer;
    return Sim_Set_Ok;
}

static attr_value_t
get_enabled_attribute(void *dont_care, conf_object_t *obj, attr_value_t *idx)
{
    transactional_memory_object_t *tm_obj = (transactional_memory_object_t *)obj;
    attr_value_t ret;

    ret.kind = Sim_Val_Integer;
    ret.u.integer = tm_obj->tm_enabled;

    return ret;
}

DLL_EXPORT void
init_local(void)
{
    class_data_t class_data;
    conf_class_t *conf_class;

    /* initialize and register the class "transactional-memory-class" */
    memset(&class_data, 0, sizeof(class_data_t));
    class_data.new_instance = transactional_memory_new_instance;
    class_data.description =

```

```

    "The transactional-memory class emulates the transaction memory "
    "instruction set extensions.";

    conf_class = SIM_register_class("transactional-memory", &class_data);

    /* initialize and register the timing-model interface */
    transactional_memory_timing_interface = MM_ZALLOC(1, timing_model_interface_t);
    transactional_memory_timing_interface->operate = transactional_memory_operate;
    SIM_register_interface(conf_class, "timing-model", transactional_memory_timing_interface);

    /* initialize attributes */
    SIM_register_attribute(conf_class, "enabled", get_enabled_attribute,
        0, set_enabled_attribute, 0, Sim_Attr_Session,
        "<tt>1</tt>|<tt>0</tt> Set to 1 to enable transactional memory, 0 to disable.");
}

DLL_EXPORT void
fini_local(void)
{
}

```

A.2.3 commands.py

```

from cli import *

def tm_enable_cmd(obj):
    SIM_set_attribute(obj, "enabled", 1)
    print "Transactional Memory enabled"

new_command("enable", tm_enable_cmd,
    [],
    type = "transactional-memory-commands",
    alias = "tm-enable",
    namespace = "transactional-memory",
    short = "enable/disable transactional memory",
    doc = """
The <b>enable</b> command turns on transactional memory support and the
<b>disable</b> command switches off transactional memory support.
""")

def tm_disable_cmd(obj):
    SIM_set_attribute(obj, "enabled", 0)
    print "Transactional Memory disabled"

new_command("disable", tm_disable_cmd,
    [],
    type = "transactional-memory-commands",
    alias = "tm-disable",
    namespace = "transactional-memory",
    short = "disable transactional memory",
    doc_with = "<transactional-memory>.enable")

```

A.2.4 Makefile

```

MODULE_DIR = transactional-memory
MODULE_NAME = transactional-memory

MODULE_CFLAGS =
MODULE_LDFLAGS =

```

```

MODULE_CLASSES = transactional memory

SRC_FILES = transactional-memory.c

include $(SIMICS_BASE)/src/extensions/common/extension-makefile

```

A.3 Benchmarks

A.3.1 lock.h

```

#include <sched.h>

#define LOCK_LOCKED 0
#define LOCK_UNLOCKED 1

typedef volatile unsigned int lock_t;

static inline void spinlock_lock(lock_t *lock) {
    unsigned int __val = LOCK_LOCKED;
    do {
        asm volatile ("xchg %0,%2" : "=r"(__val) : "0"(__val), "m"(*lock));
    } while (__val != LOCK_UNLOCKED);
}

static inline void spinlock_unlock(lock_t *lock) {
    *lock = LOCK_UNLOCKED;
}

static inline void yieldlock_lock(lock_t *lock) {
    unsigned int __val = LOCK_LOCKED;
    while (1) {
        asm volatile ("xchg %0,%2" : "=r"(__val) : "0"(__val), "m"(*lock));
        if (__val != LOCK_UNLOCKED) {
            sched_yield();
        } else {
            break;
        }
    }
}

static inline void yieldlock_unlock(lock_t *lock) {
    *lock = LOCK_UNLOCKED;
}

```

A.3.2 Counter Benchmark (counter.c)

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#ifdef TM
#include "transactional-memory_api.h"
#else

```

```

#include "lock.h"
#endif

int main(int argc, char *argv[]) {
    key_t key;
    int shmid;
    int semid;
    struct shmid_ds buf;
    struct sembuf op[1];
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    } arg;

    int *counter;
    int success;
    int work;
    int processes;

#ifdef TM
    int status;
    int tag;
    int temp;
#endif
#ifdef BACKOFF
    int wait;
    int backoff;
#endif
#define BACKOFF_MIN 0x2
#endif
else
    key_t key2;
    int shmid2;
    lock_t *lock;
#endif

    key = 4000;
#ifdef TM
    key2 = 4500;
#endif

    if (argc == 1) {
        int a;
        int b;
#ifdef TM
        int c;
#endif
        printf("No arguments... performing cleanup\n");
        shmid = shmget(key, sizeof(int), IPC_CREAT | 0666);
        semid = semget(key, 1, IPC_CREAT | 0666);
        a = shmctl(shmid, IPC_RMID, &buf);
        b = semctl(semid, 0, IPC_RMID);
#ifdef TM
        shmid2 = shmget(key2, sizeof(lock_t), IPC_CREAT | 0666);
        c = shmctl(shmid2, IPC_RMID, &buf);
        printf("shmid:%d semid:%d shmid2:%d\n", shmid, semid, shmid2);
        printf("shmctl:%d semctl:%d shmctl:%d\n", a, b, c);
#endif
        printf("shmid:%d semid:%d\n", shmid, semid);
        printf("shmctl:%d semctl:%d\n", a, b);
    }
    exit(1);
} else if (argc != 3) {
    printf("Usage: %s work processes\n", argv[0]);
    exit(1);
}

work = atoi(argv[1]);
processes = atoi(argv[2]);

```

```

    success = 0;
    op[0].sem_num = 1;
    op[0].sem_flg = 0;

#ifdef TM
    tag = 0;
#endif

    shmid = shmget(key, sizeof(int), IPC_CREAT | IPC_EXCL | 0666);
    semid = semget(key, 1, IPC_CREAT | IPC_EXCL | 0666);
#ifdef TM
    shmid2 = shmget(key2, sizeof(lock_t), IPC_CREAT | IPC_EXCL | 0666);
#endif

    if (shmid != -1) {
        if ((int)(counter = (int *)shmat(shmid, NULL, 0)) == -1) {
            perror("shmat counter failed");
            exit(1);
        }

        *counter = 0;
    } else {
        if ((shmid = shmget(key, sizeof(int), 0666)) == -1) {
            perror("shmget failed getting counter shmid");
            exit(1);
        }

        if ((int)(counter = (int *)shmat(shmid, NULL, 0)) == -1) {
            perror("shmat counter failed");
            exit(1);
        }
    }

#ifdef TM
    // Simics hack
    // reference counter so that we don't get a Simics translation error
    temp = *counter;
#endif
}

if (semid != -1) {
    arg.val = processes;

    if (semctl(semid, 0, SETVAL, arg) == -1) {
        perror("semctl cannot set semaphore value.\n");
        exit(1);
    }
} else {
    if (semid = semget(key, 1, 0666) == -1) {
        perror("semget failed trying to get semid");
        exit(1);
    }
}

#ifdef TM
if (shmid2 != -1) {
    if ((int)(lock = (lock_t *)shmat(shmid2, NULL, 0)) == -1) {
        perror("shmat lock failed");
        exit(1);
    }
}

#endif

#ifdef YIELDLOCK
    spinlock_unlock(lock);
#else
    yieldlock_unlock(lock);
#endif
} else {
    if ((shmid2 = shmget(key2, sizeof(lock_t), 0666)) == -1) {
        perror("shmget failed trying to get lock shmid");
    }
}

```

```

        exit(1);
    }

    if ((int)(lock = (lock_t *)shmat(shmid2, NULL, 0)) == -1) {
        perror("shmat lock failed");
        exit(1);
    }
}
#endif

// wait for everyone else before starting
while (1) {
    shmctl(shmid, IPC_STAT, &buf);
    if (buf.shm_nattch == processes) {
        break;
    }
    sleep(1);
}

while (success < work) {
#ifdef TM
    BEGIN(&tag);
    RESERVE(counter, &tag);
    ST(counter, LT(counter,&tag)+1, &tag);
    status = COMMIT(&tag);
    RELEASE(counter, &tag);
    if (status) {
        success++;
    }
#endif
#ifdef BACKOFF
    backoff = BACKOFF_MIN;
#endif
    } else {
#ifdef BACKOFF
        wait = random() % (0x1 << backoff);
        while (wait--);
        backoff++;
    }
#endif
    }
    #else
#ifdef YIELDLOCK
        spinlock_lock(lock);
    }
    yieldlock_lock(lock);
    #endif
    *counter = *counter + 1;
    #ifndef YIELDLOCK
        spinlock_unlock(lock);
    }
    yieldlock_unlock(lock);
    #endif
    success++;
    #endif
}

op[0].sem_num = 0;
op[0].sem_op = -1;
semop(semid, op, 1);

// wait for everyone else to finish
// semctl returns
// 0 - everyone is done
// >0 - still processes working
// -1 - error meaning everyone should be done and someone already
//      deleted semaphore and shared memory
while (semctl(semid, 0, GETVAL) > 0) {
    sleep(1);
}

```



```

    shmctl(shmid, IPC_RMID, &buf);
    semctl(semid, 0, IPC_RMID);
#ifdef TM
    shmctl(shmid2, IPC_RMID, &buf);
#endif

    return 0;
}

```

A.3.3 Consumer-Producer Benchmark (queue.c)

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#ifdef TM
#include "transactional_memory_api.h"
#else
#include "lock.h"
#endif

#define QUEUE_SIZE 1024

typedef struct {
    int deqs;
    int enqs;
    int items[QUEUE_SIZE];
} queue_t;

int main(int argc, char *argv[]) {
    key_t key;
    int shmid;
    int semid;
    struct shmid_ds buf;
    struct sembuf op[1];
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    } arg;

    queue_t *q;
    int result;
    int state;
    int work;
    int processes;
    int success;

#ifdef TM
    int status;
    int tag;
    int head;
    int tail;
    int temp;
    void *ptr;
#endif
#ifdef BACKOFF
    int wait;
    int backoff;
#endif
#define BACKOFF_MIN 0x2
#endif

```

```

#else
    key_t key2;
    int shmid2;
    lock_t *lock;
#endif

    key = 5000;
#ifdef TM
    key2 = 5500;
#endif

    if (argc == 1) {
        int a;
        int b;
#ifdef TM
        int c;
#endif
        printf("No arguments... performing cleanup\n");
        shmid = shmget(key, sizeof(queue_t), IPC_CREAT | 0666);
        semid = semget(key, 1, IPC_CREAT | 0666);
        a = shmctl(shmid, IPC_RMID, &buf);
        b = semctl(semid, 0, IPC_RMID);
#ifdef TM
        shmid2 = shmget(key2, sizeof(lock_t), IPC_CREAT | 0666);
        c = shmctl(shmid2, IPC_RMID, &buf);
        printf("shmid:%d semid:%d shmid2:%d\n", shmid, semid, shmid2);
        printf("shmctl:%d semctl:%d shmctl:%d\n", a, b, c);
#endif
    } else {
        printf("shmid:%d semid:%d\n", shmid, semid);
        printf("shmctl:%d semctl:%d\n", a, b);
    }
    exit(1);
} else if (argc != 4) {
    printf("Usage: %s produce|consume work processes \n", argv[0]);
    exit(1);
}

state = atoi(argv[1]);
work = atoi(argv[2]);
processes = atoi(argv[3]);
success = 0;
op[0].sem_num = 1;
op[0].sem_flg = 0;

#ifdef TM
    tag = 0;
#endif

    shmid = shmget(key, sizeof(queue_t), IPC_CREAT | IPC_EXCL | 0666);
    semid = semget(key, 1, IPC_CREAT | IPC_EXCL | 0666);
#ifdef TM
    shmid2 = shmget(key2, sizeof(lock_t), IPC_CREAT | IPC_EXCL | 0666);
#endif

    if (shmid != -1) {
        if ((int)(q = (queue_t *)shmat(shmid, NULL, 0)) == -1) {
            perror("shmat q failed");
            exit(1);
        }

        q->enqs = 0;
        q->deqs = 0;
    } else {
        if ((shmid = shmget(key, sizeof(int), 0666)) == -1) {
            perror("shmget failed getting q shmid");
            exit(1);
        }
    }

```

```

        if ((int)(q = (queue_t *)shmat(shmid, NULL, 0)) == -1) {
            perror("shmat q failed");
            exit(1);
        }
    }

#ifdef TM
    // Simics hack
    // reference q so that we don't get a Simics translation error
    // expects q to be page aligned address with page size of 4kb (2^12)
    for (ptr=q; (unsigned int)ptr<(unsigned int)q+sizeof(queue_t); ptr+=1<<12) {
        temp = *((unsigned int*)ptr);
    }
#endif

    if (semid != -1) {
        arg.val = processes;

        if (semctl(semid, 0, SETVAL, arg) == -1) {
            perror("semctl cannot set semaphore value.\n");
            exit(1);
        }
    } else {
        if ((semid = semget(key, 1, 0666)) == -1) {
            perror("semget failed trying to get semid");
            exit(1);
        }
    }

#ifdef TM
    if (shmid2 != -1) {
        if ((int)(lock = (lock_t *)shmat(shmid2, NULL, 0)) == -1) {
            perror("shmat lock failed");
            exit(1);
        }
    }

#ifdef YIELDLOCK
    spinlock_unlock(lock);
#else
    yieldlock_unlock(lock);
#endif
    } else {
        if (shmid2 = shmget(key2, sizeof(lock_t), 0666)) == -1) {
            perror("shmget failed trying to get lock shmid");
            exit(1);
        }

        if ((int)(lock = (lock_t *)shmat(shmid2, NULL, 0)) == -1) {
            perror("shmat: shmat failed");
            exit(1);
        }
    }
#endif

    // wait for everyone else before starting
    while (1) {
        shmctl(shmid, IPC_STAT, &buf);
        if (buf.shm_nattch == processes) {
            break;
        }
        sleep(1);
    }

    if (state == 0) { // producer
        while (success < work) {
#ifdef TM
            BEGINT(&tag);
            RESERVE(&(q->enqs), &tag);

```

```

    RESERVE(&(q->deqs), &tag);
    tail = LT(&(q->enqs), &tag);
    head = LT(&(q->deqs), &tag);

    if ((tail+1)%QUEUE_SIZE != head%QUEUE_SIZE) {
        RESERVE(&(q->items[tail%QUEUE_SIZE]), &tag);
        ST(&(q->items[tail%QUEUE_SIZE]), tail, &tag);
        ST(&(q->enqs), tail+1, &tag);
    }

    status = COMMIT(&tag);
    RELEASE(&(q->enqs), &tag);
    RELEASE(&(q->deqs), &tag);
    RELEASE(&(q->items[tail%QUEUE_SIZE]), &tag);

    if (status && ((tail+1)%QUEUE_SIZE != head%QUEUE_SIZE)) {
        success++;
#ifdef BACKOFF
        backoff = BACKOFF_MIN;
#endif
    }
#ifdef BACKOFF
    else {
        wait = random() % (0x1 << backoff);
        while (wait--);
        backoff++;
    }
#endif
    #endif

    #else
    #ifndef YIELDLOCK
        spinlock_lock(lock);
    #else
        yieldlock_lock(lock);
    #endif
    if ((q->enqs + 1) % QUEUE_SIZE != q->deqs % QUEUE_SIZE) {
        q->items[q->enqs % QUEUE_SIZE] = q->enqs;
        q->enqs++;
        success++;
    }
    #ifndef YIELDLOCK
        spinlock_unlock(lock);
    #else
        yieldlock_unlock(lock);
    #endif
    #endif

    #endif
    }
    ) else if (state == 1) { // consumer
        while (success < work) {
#ifdef TM
            BEGINT(&tag);
            RESERVE(&(q->enqs), &tag);
            RESERVE(&(q->deqs), &tag);
            tail = LT(&(q->enqs), &tag);
            head = LT(&(q->deqs), &tag);

            if (head != tail) {
                RESERVE(&(q->items[head%QUEUE_SIZE]), &tag);
                result = LT(&(q->items[head%QUEUE_SIZE]), &tag);
                ST(&(q->deqs), head+1, &tag);
            }

            status = COMMIT(&tag);
            RELEASE(&(q->enqs), &tag);
            RELEASE(&(q->deqs), &tag);
            RELEASE(&(q->items[head%QUEUE_SIZE]), &tag);

```

```

        if (status && (head != tail)) {
            success++;
#ifdef BACKOFF
            backoff = BACKOFF_MIN;
#endif
        }
#ifdef BACKOFF
        else {
            wait = random() % (0x1 << backoff);
            while (wait--);
            backoff++;
        }
#endif

    #else

#ifdef YIELDLOCK
        spinlock_lock(lock);
    #else
        yieldlock_lock(lock);
    #endif
        if (q->enqs != q->deqs) {
            result = q->items[q->deqs % QUEUE_SIZE];
            q->deqs++;
            success++;
        }
#ifdef YIELDLOCK
        spinlock_unlock(lock);
    #else
        yieldlock_unlock(lock);
    #endif
    #endif
    }
}

    op[0].sem_num = 0;
    op[0].sem_op = -1;
    semop(semid, op, 1);

    // wait for everyone else to finish
    /*
        if positive, then ppl still in critical section
        if 0, then no one
        if -1, error someone already deleted the semaphore and shared block
    */
    while (semctl(semid, 0, GETVAL) > 0) {
        sleep(1);
    }

    shmctl(shmid, IPC_RMID, &buf);
    semctl(semid, 0, IPC_RMID);
#ifdef TM
    shmctl(shmid2, IPC_RMID, &buf);
#endif

    return 0;
}

```


Appendix B

Simics with Transactional-Memory Module Guide

This is a guide to using Simics with the transactional-memory module. We briefly describe how to setup Simics with the transactional-memory module, write programs using transactional memory instructions, and modify the transactional memory module to simulate new instructions. More details about Simics can be found in the user guide included with the Simics distribution.

B.1 Simics Installation

An academic user can obtain a free personal license and copy of Virtutech Simics from <http://www.simics.net>. The academic license allows one user to run Simics on one machine and is renewable on a yearly basis. The transactional memory module used in this thesis runs only with the x86 target and was tested only on Linux/x86 hosts. We recommend the user to obtain a Simics license with host Linux/x86 and target Simics/x86. For the remainder of this guide, we assume this target-host combination.

Virtutech usually approves and emails the user an academic license in a few business days. Instructions on downloading, installing, and getting started with Simics are also included in the email. We use the directory `[simics]` to refer to where Simics is installed.

After installing Simics, the user needs to download disk dumps from the Simics web-

site. Experiments in this thesis used the Redhat 7.3 Linux `enterprise-rh73.craff` and Redhat 6.2 Linux `hippie3-rh62.craff` disk dumps. Place the disk dumps in the `[simics]/import/x86` directory.

B.2 Transactional-Memory Module Installation

We use `[path]` to refer to the directory where the required files are residing.

Copy `transactional-memory_api.h` to the Simics header files directory.

```
cd [simics]/x86-linux/obj/include
cp [path]/transactional-memory_api.h .
```

Create a subdirectory for the transactional-memory module and place the files there.

```
cd [simics]/src/extensions
mkdir transactional-memory
cd transactional-memory
cp [path]/transactional-memory.h .
cp [path]/transactional-memory.c .
cp [path]/commands.py .
cp [path]/Makefile .
```

Add the transactional-memory module to the build list by adding the following line to the file `[simics]/config/modules.list-local`:

```
transactional-memory | BIT2 | x86
```

Setup the build environment and compile the transactional-memory module.

```
cd [simics]/x86-linux
../configure -q
cd [simics]/x86-linux/lib
gmake transactional-memory
```

B.3 Virtual Machine Configuration

The Simics distribution provides machine configurations in the directory `[simics]/home`. The virtual machines used in this thesis are based on the supplied Enterprise configurations.

In the configuration file, we must attach the transactional-memory module to the physical memory of each processor in the system. We can also change machine parameters

```
@num_processors = 2
@clock_freq_mhz = 500
@memory_megs = 256

@def user_config():
    global conf_list
    conf_list += [OBJECT("tm", "transactional-memory")]
    set_attribute(conf_list, "phys_mem0", "timing_model", "tm")
    set_attribute(conf_list, "phys_mem1", "timing_model", "tm")

run-command-file enterprise-common.simics
```

Figure B-1: Simics dual processor machine configuration.

such as number of processors, processor frequency, and memory in the system. Figure B-1 shows the configuration file `dosx.simics` that was used to create the Simics dual processor machine.

To start Simics with the dual processor machine configuration and the x86-p3 processor model, type at the command prompt:

```
cd [simics]/home/enterprise
./simics x86-p3 -x dosx.simics
```

The default behavior of the transactional-memory module is not to simulate transactional instructions. Simulating transactional instructions reduces the performance of Simics because the module must check for possible transaction conflicts on every memory reference. To enable transactional memory simulation, type `tm-enable` at the Simics prompt:

```
simics> tm-enable
```

B.4 Programming with Transactional Instructions

A C program must include the file `transactional-memory_api.h` in order to use the transactional instructions.

```
#include "transactional-memory_api.h"
```

```

#define OP_ADD 0xDEADBEEF

#define ADD(a, b) \
({ unsigned int __c; \
   unsigned int *__a = (a); \
   unsigned int *__b = (b); \
   asm volatile ("movl %0,%%eax" :: "g" (OP_ADD) : "eax"); \
   asm volatile ("movl %0,%%ebx" :: "g" (&__c) : "ebx"); \
   asm volatile ("movl %0,%%ecx" :: "g" (__a) : "ecx"); \
   asm volatile ("movl %0,%%edx" :: "g" (__b) : "edx"); \
   MAGIC_BREAKPOINT; \
   __c; })

```

Figure B-2: ADD macro.

A program with transactional instructions can be compiled with `gcc` like any other C program.

B.5 Modeling New Instructions

In this section, we demonstrate how to create and simulate a new instruction.

In this example, we model a new instruction that performs a memory-to-memory add. The macro `ADD(a, b)` adds the value at memory location `a` to the value at memory location `b` and returns the result.

First, we write the `ADD` macro in `transactional-memory_api.h` (Figure B-2). Find an unused opcode. Use assembly code to put the source and destination operands in the appropriate registers. In this example, we put the opcode in `%eax`, the address to put the result `c` in `%ebx`, `a` in `%ecx`, and `b` in `%edx`. Call the magic instruction and return the result.

The second step is to modify the transactional-memory module to simulate the `ADD` macro (Figure B-3). The function `tm-handler` in `transactional-memory.c` is called when Simics encounters the magic instruction. Add a case statement for the `ADD` macro. Read the logical addresses of the operands and destination from `%ebx`, `%ecx`,

```
case OP_ADD:
    c_la = SIM_read_register(cpu, ebx_no);
    a_la = SIM_read_register(cpu, ecx_no);
    b_la = SIM_read_register(cpu, edx_no);

    c_pa = SIM_logical_to_physical(cpu, Sim_DI_Data, c_la);
    a_pa = SIM_logical_to_physical(cpu, Sim_DI_Data, a_la);
    b_pa = SIM_logical_to_physical(cpu, Sim_DI_Data, b_la);

    a = SIM_read_phys_memory(cpu, a_pa, 4);
    b = SIM_read_phys_memory(cpu, b_pa, 4);

    SIM_write_phys_memory(cpu, c_la, a+b, 4);
    break;
```

Figure B-3: Code to simulate ADD macro.

and %edx. Translate the logical addresses to physical addresses. Read the source values, perform the add operation, and store the result.

Finally, write, compile, and simulate a program that uses the new ADD instruction (Figure B-4).

```
#include <stdio.h>
#include "transactional-memory_api.h"

void main() {
    int a = 2;
    int b = 3;
    c = ADD(&a, &b);

    printf("The result of c is %d.\n", c);
}
```

Figure B-4: Sample program using ADD macro.

Bibliography

- [1] Compaq Computer Corporation. *Alpha Architecture Handbook, Version 4*, October 1998. <http://www.support.compaq.com/alpha-tools/documentation/current/alpha-archt/alpha-architecture.pdf>.
- [2] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report 92/07, Digital Equipment Corporation Cambridge Research Lab, December 1992. <http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-92-7.html>.
- [3] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–301, May 1993. <http://www.cs.brown.edu/people/mph/HerlihyM93/herlihy93transactional.pdf>.
- [4] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 2003. <http://www.intel.com/design/pentium4/manuals/245471.htm>.
- [5] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002. http://www.virtutech.com/simics/Computer_200202.html.
- [6] MIPS Technologies. *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*, 1 September 2002.

- [7] MIPS Technologies. *MIPS64 Architecture for Programmers Volume II: The MIPS64 Instruction Set*, 29 August 2002.
- [8] Motorola. *Motorola M68000 Family Programmer's Reference Manual*, 1992. <http://e-www.motorola.com/collateral/M68000PRM.pdf>.
- [9] Motorola. *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, December 2001. <http://e-www.motorola.com/brdata/PDFDB/docs/MPCFPE32B.pdf>.
- [10] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, December 2001. <http://www.cs.wisc.edu/~rajwar/papers/micro01.pdf>.
- [11] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002. <http://www.cs.wisc.edu/~rajwar/papers/asplos02.pdf>.
- [12] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. Prentice Hall, Englewood Cliffs, New Jersey, 1994. <http://www.sparc.com/standards/SPARCV9.pdf>.