Synchronization on Multicore Architectures

by

Rachael Harding

B.S., Electrical and Computer Engineering, Carnegie Mellon University, 2010

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

February 2014

Signature of Author: ______________________________
Rachael Harding
Department of Electrical Engineering and Computer Science
January 24, 2014

Certified by: ______________________________
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ______________________________ [illegible]
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Committee for Graduate Students

Synchronization on Multicore Architectures

by Rachael Harding

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

Abstract

The rise of multicore computing has made synchronization necessary to overcome the challenge of sharing data between multiple threads. Locks are critical synchronization primitives for maintaining data integrity and preventing race conditions in multithreaded applications.

This thesis explores the lock design space. We propose a hardware lock implementation, called the lock arbiter, which reduces lock latency while minimizing hardware overheads and maintaining high levels of fairness between threads. We evaluate our mechanism against state-of-the-art software lock algorithms and find that our mechanism has comparable performance and fairness.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Srini Devadas for his support and guidance on this project. I would also like to thank George Bezerra for his feedback on this work and Hornet for being a sounding board during this thesis's earliest stages.

To my friends, my unending gratitude for your support and understanding throughout this entire project. I would especially like to thank Sruthi, Fabián, Francisco, Boris, Bernhard, and Steven. Without you, this thesis would not have been possible.

I would also like to thank the National Science Foundation Graduate Research Fellowship Program for their financial support (grant #1122374).

Contents

Abstract **3**

Acknowledgments **4**

List of Figures **9**

1 Introduction **11**

1.1 Synchronization in the Multicore Era 11

1.2 The Importance of Synchronization 11

1.3 Lock Characterization 13

1.3.1 Timeline 13

1.3.2 Physical Structure 14

1.4 Implementation Challenges 14

2 Related Work **17**

2.1 Software-based Synchronization 17

2.1.1 Early Locks 17

2.1.2 Complex Software Locks 19

2.1.3 Machine Learning Approaches 21

2.2 Hardware Locks 21

2.3 System-on-a-Chip and Embedded Multiprocessors 22

3 A Cycle-accurate Simulator for Multicore Processors **23**

3.1 Simulator Overview 24

3.2 Core Model 25

3.2.1 Memory Traces 25

3.3 Network 25

3.3.1 Routing 26

3.4 Caches and Cache Coherence 26

4 A Centralized, Fair and Fast Lock Arbiter **29**

4.1 Lock Arbiter 29

4.1.1 Lock Arbiter Algorithm 30
4.1.2 Overflow Mechanism 32
4.1.3 Hardware Additions 33
4.1.4 Instruction Set Architecture Additions 34
4.2 Evaluation 34
4.2.1 Simulation Setup 35
Microbenchmarks 35
Multi-threaded Scientific Benchmarks 36
4.2.2 Performance 37
4.2.3 Network Traffic and Contention 45
Cache Coherence Requests 45
4.3 Analysis and Conclusions 48

5 Future Directions for Synchronization **49**
5.1 Lock Arbiter Viability 49
5.2 Other Synchronization Primitives 50
5.2.1 Conditional Wait 50
5.2.2 Barriers 50
5.3 Smart Synchronization 51

List of Figures

1.1 An execution with two threads revealing a race condition. 11
1.2 Adding a lock prevents the race condition. 12
1.3 Timeline of a lock. 13

3.1 A single node. 24
3.2 The mesh network of a 16-node instantiation of the simulator. 26

4.1 Flow chart of *LOCK* process. 30
4.2 Performance of our system versus software lock algorithms. 38
4.3 Performance of our system versus software lock algorithms for the PARSEC benchmark suite. 38
4.4 Effect of lock queue length on performance. 39
4.5 Effect of lock queue length on average acquire time. 40
4.6 Short critical section handoff time. 41
4.7 Long critical section handoff time. 41
4.8 Short critical section acquire time. 42
4.9 Long critical section acquire time. 43
4.10 PARSEC acquire time. 43
4.11 Average lock acquire time by thread. 44
4.12 Average lock acquire time by thread for different lock queue lengths. . . 45
4.13 Network traffic breakdown by request type. 46
4.14 Network traffic breakdown by request type, focused on LA, MCS and CLH. 46
4.15 PARSEC network traffic breakdown by request type. 47

Chapter 1

Introduction

1.1 Synchronization in the Multicore Era

THE past decade has seen a steady rise in the number of cores on a single die. Commercial chips already have dozens of cores, such as Intel's 80-core Teraflops chip prototype and Tilera's 72-core TILE-Gx [4] [5]. Forecasts predict hundreds or thousands of cores on a single chip in the next decade.

The trend towards multicore processing stems from power constraints, scalability concerns, and processor complexity. Application developers have also looked towards parallelism for performance gains in their applications. Parallelism allows independent tasks in an application to be performed by threads simultaneously. This allows performance gains to be achieved in accordance with Amdahl's law. These threads may be executed on different cores. Thus, hardware designers move towards multicore.

Threads working in parallel are not entirely independent. Threads typically share data in order to communicate intermediate or final results from their tasks. These threads must synchronize with each other so that any shared data is modified consistently. In the next section, we illustrate why synchronization is critical for multithreaded applications.

1.2 The Importance of Synchronization

Thread A	Thread B
read(x) $\rightarrow$ 0	
add(x,1) $\rightarrow$ 1	
	read(x) $\rightarrow$ 0
	add(x,1) $\rightarrow$ 1
	store(x,add_result)
store(x,add_result)	

Figure 1.1: An execution with two threads revealing a race condition.

Although tasks in applications can often be performed independently, there are some data dependencies that need to be carefully handled to preserve correct program

Thread A	Thread B
lock(m)	
read(x) $\rightarrow$ 0	
add(x,1) $\rightarrow$ 1	
store(x,add_result)	
unlock(m)	
	lock(m)
	read(x) $\rightarrow$ 1
	add(x,1) $\rightarrow$ 2
	store(x,add_result)
	unlock(m)

Figure 1.2: Adding a lock prevents the race condition.

execution. Algorithm 1 shows a classic example of a race condition that may occur if two threads running the *thread_work*() function perform operations on the same data.

```
integer x = 0;
thread_work() {
  x = x + 1;
}
```

Algorithm 1: A simple, multi-threaded application in which threads increment a shared variable x.

Threads A and B attempt to increment the shared variable integer x, which is initialized to 0. Intuitively, the reader may think that either A or B increments x first, then the other increments x, resulting in a final value of 2. However, A and B may be executing in parallel, and $x = x + 1$ is not an atomic operation. In fact, $x = x + 1$ is composed of three distinct operations: read x, compute $x + 1$, and store the computed value to x. These three operations may be executed by A and B in any interleaved order. Figure 1.1 shows a possible execution in which A reads x and performs the computation. However, before A stores the value back to x, B reads the old value of x, performs the computation and stores the value to x. When A finally stores its new value to x, it overwrites B's value, resulting in a final value of 1.

In order to prevent race conditions, programmers use locks as a way to synchronize threads and preserve correctness. Locks guarantee mutual exclusion. That is, any one thread can own a lock at any given moment. A thread can only move beyond the initial lock request when it owns the lock. Locks provide a way to allow only one thread to access a shared variable.

Algorithm 2 performs the same task as Algorithm 1, but uses a lock to protect x. Once a thread acquires the lock, the other thread cannot acquire the lock or move

```
integer x = 0;
lock_variable m;
thread_work() {
  lock(m);
  x = x + 1;
  unlock(m);
}
```

Algorithm 2: A multi-threaded application with a shared variable x protected by lock m.

forward in the code until the lock is released. Thus, only one thread increments x at a time, and the result stored to x is always 2, as expected.

1.3 Lock Characterization

Locks are primitives that maintain mutual exclusion in applications by allowing only one thread to own the lock at a given time. Locks protect data from being shared in ways that may destroy the integrity of an application. In this section, we go into more detail about the elements that compose a lock in order to better understand how locks work.

1.3.1 Timeline

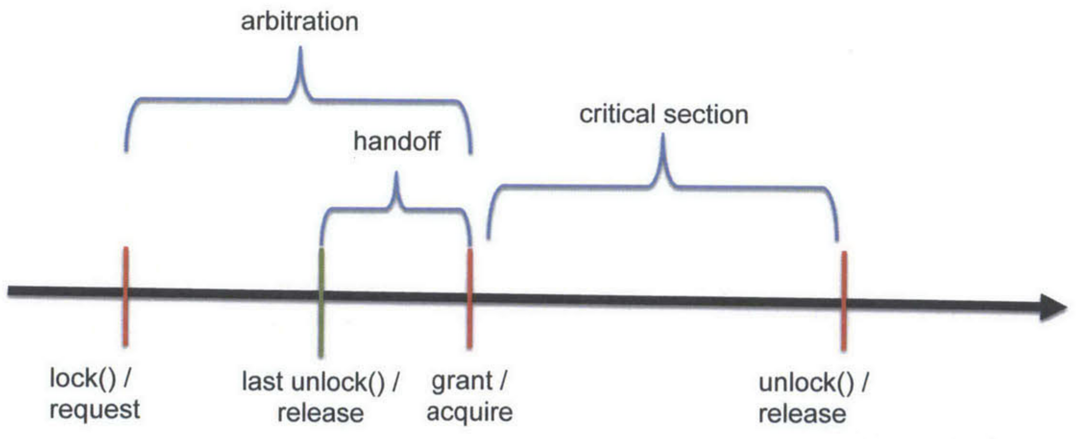

Figure 1.3: Timeline of a lock.

Locks have two phases, *arbitration* and the *critical section*, as shown in Figure 1.3. Arbitration is the process of choosing the next lock owner, and is performed by a mechanism called the arbiter, which may be centralized or distributed, depending on the implementation. The *critical section* is the execution of the code protected by the lock. Only one thread will be in the critical section at any point during the application execution, while any number of threads may be in arbitration.

These phases are separated by three events: the lock *request* by the requesting thread, the lock *acquire* (also called *grant* from the perspective of the arbiter), and the lock *release*. After a thread requests a lock, the request goes into arbitration until the arbiter grants the lock to the requester, at which time the requesting thread acquires the lock. During arbitration, the arbiter determines which requesting thread should be granted the lock. After a thread acquires the lock, it enters its critical section. When the thread completes the critical section, it releases the lock and resumes normal execution. The arbiter can then grant the lock to another thread that is in arbitration. The time between the last lock release and the next grant when the lock is contended is called the *handoff*.

The lock timeline reveals several opportunities for performance improvement when synchronizing. First, the arbitration period can potentially be shortened by simplifying or reducing the overall timing of the arbitration algorithm. However, shortening the arbitration only improves performance if the lock is uncontended or critical sections are too short to hide the arbitration latency.

Additionally, during the handoff period no useful computation towards the completion of the application is performed. Therefore, in order to optimize locks it is important to reduce the latency of the handoff.

Also, after the request, release, and grant the thread must communicate with the arbiter and vice versa. The communication introduces additional overheads in the arbitration and handoff periods.

■ 1.3.2 Physical Structure

How does the system know which thread, if any, owns the lock? Locks require some memory to maintain information about the lock. At a minimum, a flag that indicates whether or not the lock is owned is necessary. But as we'll see in Chapter 2, algorithms with more complex arbitration may require more metadata about the lock, resulting in an even larger data structure.

Size of the lock data structure is a major concern for multicore processors due to scalability. For example, if the lock data structure depends on the number of cores, as the number of cores on chip increases, so does the memory required by the lock.

■ 1.4 Implementation Challenges

There are many implementation challenges associated with locks. First, a lock needs to guarantee mutual exclusion. An incorrect lock implementation could lead to incorrect execution with race conditions such as the one in Figure ??. Thus, verification that the lock is correct is crucial. The number of execution orders increase dramatically as the number of cores increases, due to increased parallelization and, as a result, the number of possible interleavings of operations. As a result, verification is a real and challenging problem. Second, since guaranteeing mutual exclusion requires communication between participants in the process, performance is dependent on the communication substrate between the cores. Thus, the communication substrate provides a natural bound on the lock handoff latency.

As the number of threads in an application increases, the potential for synchronization to become a bottleneck increases. If more than one thread needs to acquire the same lock at one particular instance, the lock is said to be *contended*. Contended locks may result in poor performance because the lock can only be issued to one thread at a time. All the other threads must wait for the lock. During this time, the other threads are not doing any useful work. Only the thread holding the lock is making forward progress in the application. Therefore, it is important for the lock handoff between threads to be done as quickly as possible. It is also important that locks that are not contended are granted to requesting threads quickly. Acquiring a lock, although necessary for correctness, is not useful work in the application. Therefore, the process of acquiring the lock, and handing the lock off to other waiting threads, should be made as fast as possible.

Another challenge for synchronization is fairness. Fairness is the equal opportunity for threads to obtain lock. An unfair lock may be granted to the same thread repeatedly, despite other threads attempting to acquire the lock at the same time.

Chapter 2

Related Work

IN Chapter 1 we described the layout of a lock and ways in which locks could be improved. In this chapter we describe existing synchronization solutions found in the literature. We examine each lock's arbitration mechanism and handoff latency, and compare them qualitatively.

2.1 Software-based Synchronization

Various software-based locks, which rely on cache coherence to synchronize, are summarized in Table 2.1. Cache coherence is the most common mechanism to support data sharing in modern multiprocessors. As we will see in this section, when managed by the coherence protocol, synchronization data structures are continually replicated and invalidated. When the lock is contended, this leads to significant network traffic overhead and overall inefficiency.

2.1.1 Early Locks

The first locks used in multiprocessors were basic spin locks. These locks operate by repeatedly accessing a memory location, or *spinning*, until the memory location contains a specific value. The locking process is performed as a software algorithm, and the lock data structure is maintained through the cache coherence protocol. Test&Set (T&S) is an early spin lock algorithm that performs an atomic read-modify-write operation

Lock Algorithm	Local Spinning	Queue-based	Fair (FIFO)
T&S	No	No	No
T&T&S	Yes	No	No
MCS	Yes	Yes	Yes
CLH	Yes	Yes	Yes
M	Yes	Yes	Yes
QOSB	Yes	Yes	Yes

Table 2.1: Summary of software locking algorithms.

on the lock address.When T&S performs the atomic instruction, it unconditionally sets the lock data to a non-zero value and returns the previous value. A non-zero return value means that the lock was already taken. The thread repeats the atomic operation until a zero value is returned, indicating that thread now owns the lock. Every atomic operation requires the thread to have exclusive ownership of the cache line containing the lock data structure. Thus, when the lock is contended by many threads, T&S produces significant network overheads as the lock data ping-pongs between threads requesting the lock.

Test&Test&Set (T&T&S) is an extension of T&S by Rudolph and Segall in which requesters waiting for the lock spin on read-only copies of the lock [20]. When the lock is released, the read-only copies are invalidated by the thread releasing the lock. When the requesters obtain another read-only lock copy, they find that the lock has been released. All the requesters then perform the T&S atomic operation, at which only one thread will succeed. The threads that fail to acquire the lock resume spinning on read-only copies. T&T&S improves on T&S by introducing local spinning reducing network traffic. However, when the lock is released, there is still high network traffic as all requesters obtain a read-only copy and then upgrade to an exclusive copy of the lock. In addition, the handoff time between when the last thread releases the lock and the next thread acquires the lock is longer than T&S due to the next thread having to acquire a read-only copy before obtaining the lock.

Exponential back-off locks reduce the network contention in T&S and T&T&S by inserting delay between unsuccessful attempts to obtain a lock [15]. When a T&S is unsuccessful, the thread waits for a pre-determined number of cycles before moving ahead in the algorithm. Every additional unsuccessful attempt increases the wait cycles exponentially. While exponential back-off can improve network contention by spacing the lock requests, it requires significant fine-tuning in practice to avoid unnecessarily-long handoff times. In some cases it may make handoff significantly worse than T&T&S.

Lamport improved on the T&T&S lock with his bakery algorithm, also known as ticket locks [11]. In this algorithm, each requester performs an atomic read-increment-write on a shared variable. The return value from the operation is the thread's ticket number, and the thread spins on a read-only copy of a second variable. This variable is incremented by the thread releasing the lock. When the variable equals the spinning thread's ticket number, the thread owns the lock and enters the critical section. While the bakery algorithm does not reduce network contention every time the available variable is incremented, which results in all requesters' local variables being invalidated, the algorithm removes the contention due to T&S at the handoff stage. As soon as the next thread receives a copy of the ticket, it ascertains that it owns the lock. The bakery algorithm also introduces fairness into synchronization. The algorithm maintains a queue of requesters, unlike T&S and T&T&S. The baker algorithm grants locks in order of acquires instead of to the first thread to get an exclusive copy of the lock data after the lock has been released.

■ 2.1.2 Complex Software Locks

```
my_prev_addr;
my_lock_addr;
lock() {
  prev_addr = ATOMIC_SWAP(lock_addr,my_prev_addr)
  if(prev_addr == 0)
    return;
  [my_lock_addr] = 1;
  [prev_addr] = [my_lock_addr];
  while([my_lock_addr] != 0) {
    }
  return;
}
unlock() {
  next_lock_addr = [my_prev_addr];
  if(next_lock_addr == 0) {
    prev_addr = COMPARE_AND_SWAP(lock_addr,my_prev_addr,0)
    if(prev_addr == my_prev_addr)
      return;
    while((next_lock_addr = [my_prev_addr]) == 0)
      {}
  }
  next_lock_addr = 0;
  [my_prev_addr] = 0;
}
```

Algorithm 3: MCS Algorithm

There are several software-based lock implementations that incorporate both local spinning and queue-based locking. These locks attempt to achieve the benefits of the early locks, but require more complex software algorithms. They typically require a single atomic compare-and-swap operation.

Mellor-Crummey and Scott proposed the MCS queue lock [16]. We show pseudo-code for MCS in Figure 3. Each thread that requests a lock appends itself to a global queue by swapping a pointer to its own local node with a shared tail pointer to the last node in the queue. If it is at the front of the queue, it owns the lock. If there is a previous node in the queue, it modifies the previous node in the queue to create a doubly-linked list. When the previous thread releases the lock, it sets a flag at the next node to indicate the lock is available. Due to the queue, MCS offers fair, first-come-first-served locking, unlike T&S and T&T&S. MCS also avoids a flood of invalidate and

```
my_lock_addr;
lock() {
  [my_lock_addr] = 1;
  prev_addr = ATOMIC_SWAP(lock_addr,my_lock_addr)
  while([prev_addr] != 0) {
    }
  return;
}
unlock() {
  [my_lock_addr] = 0;
  return;
}
```

Algorithm 4: CLH Algorithm

line acquires, since the local flag used to pass ownership is only accessed by two threads: the previous lock owner and the next lock owner in the queue. This means that the handoff latency is equivalent to two cache line acquires, with invalidation.

Craig, Landin and Hagersten independently proposed a queue-based lock which is now known as the CLH queue lock [6] [14]. We show pseudo-code for CLH in Figure 4. In the CLH algorithm, a thread requesting a lock first creates a local queue node with its own ID and a Boolean variable. When true, the thread either owns the lock or is waiting to acquire the lock. When false, the lock has been released. After creating its own node, the thread adds itself to the tail of the queue by swapping its node with the current tail as in MCS. The thread then spins on the Boolean value of the previous thread in the queue, which it knows from the tail pointer it swapped. CLH has a similar handoff time to MCS. However CLH simplifies arbitration because a thread does not need to modify the previous node in the queue before it begins local spinning. CLH is also fair: locks are granted in the same order that threads add themselves to the tail of the queue.

Magnusson proposed the M lock to improve upon the CLH lock [14]. This lock is similar to the CLH lock except it reduces handoff latency when the lock is uncontended by detecting when there is no queue. As a result, the code for the M lock is even more complex than CLH. When the lock is indeed contended, the handoff time is slightly longer.

Goodman et. al. proposed QOSB, a queue-based primitive which uses a coherence protocol extension [9]. Invalidated lock cache lines are recycled as nodes in the queue, using unused bits in the invalid line to store queue information. A requester first adds itself to the queue, then spins on its own local queue node. The releasing thread overwrites the node of the next thread in the queue to notify that the lock is available. The algorithm must detect when one of the queue nodes is overwritten by another valid cache line. In this case, the entire queue must be rebuilt.

Hierarchical locks transform existing lock algorithms by clustering groups of threads. Each individual cluster can utilize a different locking algorithm. A higher-level lock algorithm arbitrates the lock between clusters. The lock algorithms at each level and each cluster can be different. Implementations include the hierarchical back-off locks [19], and hierarchical CLH queue locks [12]. These algorithms exhibit better performance and fairness for architectures with non-uniform communication containing multiple nodes.

■ 2.1.3 Machine Learning Approaches

Eastep et. al. developed Smartlocks which uses application heartbeats [8] [10] to adapt its internal lock implementation at runtime. The goal of Smartlocks is to determine when the properties of a particular lock would be better suited for an application. However, Smartlocks only utilizes a spin lock library and does not take advantage of more sophisticated software locks such as those in Section 2.1.2.

Research has also been done to accelerate critical sections on heterogeneous architectures by migrating critical sections to large, complex cores with more resources to compute the critical sections faster [21]. In this process, locks surrounding the critical section are also located at the large cores to ease the handoff process.

■ 2.2 Hardware Locks

Vallejo et. al. propose a hardware solution which utilizes a lock control unit (LCU) at each core and a centralized lock reservation table (LRT) to handle lock requests [23]. As locks are requested, a queue builds in the LCUs by allocating LCU entries to the threads. The LCUs then communicate with the LRTs, located at the memory controllers, which form the queue by maintaining head and tail pointers. If the lock is owned, the LRT forwards the request to the last queued LCU, and when the lock is eventually released, that LCU forwards ownership to the next LCU. Vallejo's proposal requires complex hardware mechanisms to maintain cross-chip queues. When overflow at the LRT occurs, the LRT begins to overflow entries into a pre-allocated hash table in main memory.

Zhu et. al. proposed the Synchronization State Buffer (SSB) which is a fine-grained locking mechanism [25]. A small buffer at each memory controller maintains synchronization state information for all active synchronized data. Systems dependent on SSBs require careful programming in order to avoid deadlock, as locks are maintained at the cache line granularity. When overflow occurs, the SSB traps to software and stores overflowed entries in a software table.

While hardware approaches to synchronization often have faster arbitration and handoff than software approaches, they are limited by finite resources. If the number of active synchronizations in the system is greater than the available resources to monitor them, then the system could deadlock. Thus, any hardware mechanism needs to support overflow, which can be expensive in terms of performance and hardware. Also, both centralized and distributed hardware synchronization mechanisms are prone to the same network traffic issues as software mechanisms.

2.3 System-on-a-Chip and Embedded Multiprocessors

While this thesis addresses only synchronization on chip-multiprocessors (CMPs), there has been some research in synchronization for System-on-a-Chip (SoC) and embedded multiprocessors. SoCs often run applications which require timing guarantees or predictability, which affects synchronization algorithms. Embedded multiprocessors also have real-time performance requirements and prioritize energy efficiency.

Akgul et. al. proposed a SoC lock cache that distinguishes between tasks waiting to perform a long or a short critical section [1].

Yu and Petrov proposed using distributed queues to synchronize embedded multiprocessors [24]. Their proposal assumes that each processor can monitor a common memory bus to construct a state for each synchronization variable including the number of processors which are queued before a particular processor. Whenever a processor wants to queue for a lock, it broadcasts its request onto the bus, allowing other processors to count how many processors are queuing for a particular lock. Upon releasing the lock, the processor also broadcasts the release on the bus for all other processors to see. This method effectively eliminates handoff time, because a processor can immediately acquire the lock once it observes the same number of releases as there are processors queued before it. However, this approach relies on access to a common bus, which is not feasible for CMPs with dozens or hundreds of cores.

Chapter 3

A Cycle-accurate Simulator for Multicore Processors

BEFORE we describe and evaluate our lock arbiter design, we must first present our evaluation framework. Due to the introduction of new hardware in our design and the difficulty of integrating new hardware on existing platforms, we chose to evaluate our system using hardware simulation.

There are some simulator properties that are highly desirable for lock evaluation. First, the simulator must be able to model a multicore processor based on machines one would likely find in use in the future. Second, cycle-accurate simulation is critical, due to the fine-grain nature of locks. Locks are on the critical path of an application, and a few cycles inaccuracy may lead to significantly different simulation results. Therefore in order to compare lock implementations reliably, we need to have a simulator that models them as accurately as possible.

Existing simulators often have to trade off accuracy and performance. Since simulators model multicore processors that are different and often larger than those the simulator itself run on, simulators can take a long time to complete their tasks. Therefore simulators often sacrifice cycle accurate simulation in order to complete faster simulations. Some simulators, such as Graphite [17], can easily model hundreds of cores, but are not cycle-accurate. Other simulators, such as HAsim [18], use field-programmable gate arrays to speed up modeling. While simulating multicores can take a long time, and a fast simulator can be more convenient, in the accuracy versus performance tradeoff, we require better accuracy for some system components in our case.

To these ends, we developed a trace-driven, cycle-accurate simulator using the Verilog hardware description language. The simulator allows detailed and accurate analysis of lock performance and overheads in a multicore processor. Verilog is a widely-used tool to model hardware, and allows our simulator to accurately model both timing and functionality of the circuits [22]. Our simulator is trace-driven in order to provide the necessary cycle-accuracy in the network and memory subsystem, while maintaining a reasonable run time by losing some detail in the execution of non-memory instructions. The simulator is roughly modeled after the architecture of a Tilera multicore processor.

The simulator supports two modes: with the lock arbiter and without the lock arbiter. Except for the addition of the lock arbiter in the first mode, the simulator is the same. This allows us to evaluate our design against locking schemes that rely solely on the memory system to manage synchronization. In this chapter we describe the design of the simulator with the lock arbiter.

3.1 Simulator Overview

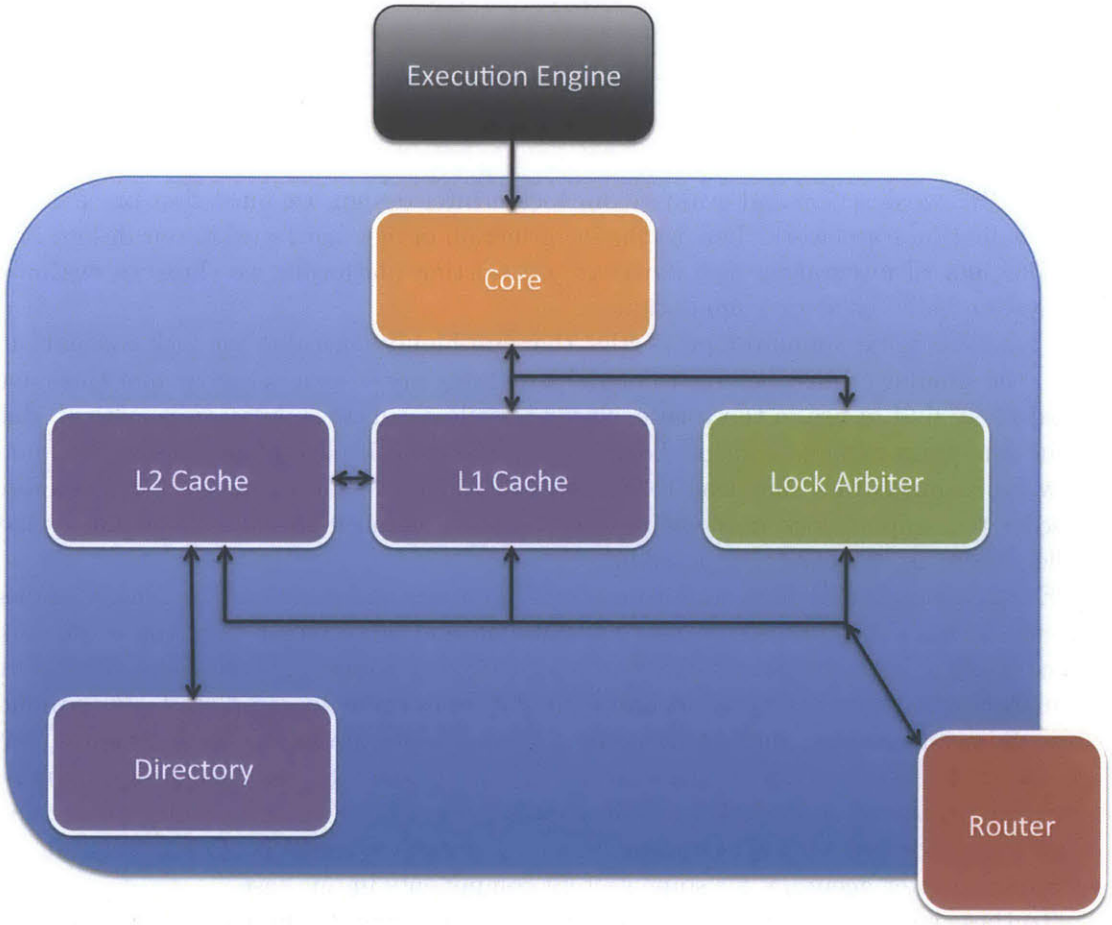

Figure 3.1: A single node.

The system is comprised of homogeneous nodes, or tiles. The architecture of a single node is shown in Figure 3.1. Each node contains an execution engine and core, a private cache, a shared cache slice and directory, a lock arbiter, and a router. The execution engine drives the simulation by reading in traces containing memory and lock operations, as well as the number of arithmetic instructions between them, and feeding them to the core model, which is detailed in Section 3.2. The core issues arithmetic instructions at a steady rate, and issues memory and lock operations to the memory subsystem. The simulator has detailed memory and network models. The cache hierarchy, described in Section 3.4, is comprised of a private cache and shared cache slice at each node. A distributed directory maintains coherence across the private caches. The nodes are arranged in a grid connected by a mesh network, as described in Section 3.3.

■ 3.2 Core Model

The core models an in-order processor. Before issuing a memory instruction from a trace, the core issues any arithmetic instructions. Arithmetic instructions are issued at a rate equal to the *issue width* of the core per cycle. We assume that arithmetic instructions can be issued every cycle, except when waiting for a memory instruction to complete.

Once all preceding arithmetic instructions have been issued, the memory or lock instruction can be issued. Only one of these instructions can be issued per cycle. To ensure that the simulator does not introduce any race errors in the original application, and preserve in-order processing, the core stalls until it receives a response that the memory or synchronization instruction has been completed. The only exceptions to this rule is the unlock instruction, which does not need a response to continue correct execution.

■ 3.2.1 Memory Traces

We generate memory traces by running an application binary with Pintool [13]. Pintool generates memory addresses, operation types (*READ*, *WRITE*, *LOCK*, *UNLOCK*), and counts how many non-memory operations were executed since the last memory instruction.

In lock arbiter mode, *LOCK* and *UNLOCK* operations are forwarded to the lock arbiter mapped to the lock defined by their lock address. In normal mode, a finite state machine intercepts these operations and replaces them with a sequence of *READ* and *WRITE* operations dependent on the baseline software lock algorithm we are evaluating.

■ 3.3 Network

The backbone of the multicore processor is the network, which connects the nodes and allows communication across the processor. Our simulator uses a mesh network topology in which n nodes are arranged as a $\log n$ by $\log n$ matrix, as shown in Figure 3.2. Coherence requests and data, encapsulated as packets or flow control digits (*flits*), are transported over the channels, or *links*, from their source node to destination node. We describe how the path the packets take from their source to their destination is determined in Section 3.3.1.

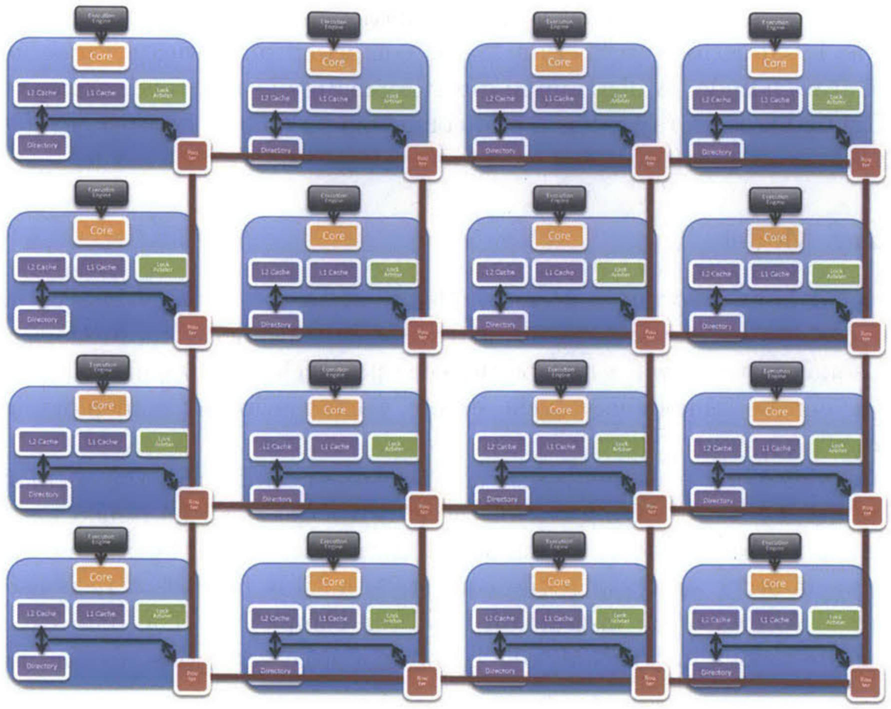

Figure 3.2: The mesh network of a 16-node instantiation of the simulator.

3.3.1 Routing

Our simulator uses X-Y routing, a deterministic, deadlock-free routing protocol [7]. Each router has an ID based on its location in the mesh.The coordinates are calculated from the ID based on the node's position in its row and column, respectively. Packets entering the network have a destination ID $< x, y >$. Packets are routed either east or west until they reach the router with the same x coordinate as the packet destination. Then packets are routed along the north-south axis until they reach the router with the same y coordinate as the packet destination. At this point, the packet is removed from the network and sent to the core, cache, or lock arbiter, depending on the request.

Routing is prioritized in the following order: east, west, north, south, node. This means that packets already in the network are prioritized over those not in the network.

The network has link-level flow control. Routers buffer each packet until the channel on its destination path is available and it can proceed to the next link. Routers also buffer incoming packets at their input until the output buffer is cleared.

3.4 Caches and Cache Coherence

Our simulator has a detailed memory model with a private cache and shared cache slice per node. Coherence is maintained through a distributed directory.

The first-level cache is a write-back cache. That is, the contents of a modified cache line due to a write is only written back to the last-level class in case of eviction, whether due to an invalidation request from the directory or thrashing.

While the capacity and associativity of the first-level cache is configurable, we assume a large last-level cache that can fit the working set of any application for simplicity.

We use the *MESI* protocol to maintain cache coherence. MESI is a widely used cache coherence protocol. Any line in the level 1 caches must be in one of the following states.

1. Modified: Line can be read or written. Only one valid copy of the line is allowed in all the first-level caches. A line in Modified state is dirty (i.e., has been written to), and thus must be written back to shared memory if the line is invalidated or evicted.

2. Exclusive: Line can be read. Only one valid copy of the line exists in all the first-level caches. A *WRITE* to a line in Exclusive state causes the thread to transition to Modified state.

3. Shared: Line can be read. Any number of copies of the line in Shared state is allowed in all first-level caches.

4. Invalid: Line is not valid in this cache.

At the directory, a directory entry for each address maintains a record of requests for various states from the different threads. If a line can be upgraded, it grants a request to upgrade the line immediately. If necessary, the directory sends invalidation requests to all the private caches with copies of the line. The directory then issues the upgrade when all private caches have responded that their copies have been invalidated.

If a cache line misses in the first-level cache, a request is sent to the directory and, if necessary, the data is fetched from the corresponding L2 slice. The node at which the line is located in the shared structure is determined by the address. The addresses are striped across the nodes by the $\log n$ lowest order bits of the address. This allows accesses to be relatively evenly distributed across the cache slices. This prevents performance bottlenecks at individual caches and on the network.

Chapter 4

A Centralized, Fair and Fast Lock Arbiter

IN Section 1.3 we described the lock structure and identified arbitration, handoff, and acquire time as areas in which lock algorithms can improve performance. In Chapter 2 we discussed existing lock implementations and their pro et contra with respect to the lock structure. In this chapter we present a hardware lock implementation for multicore systems and evaluate it against state-of-the-art software algorithms using the simulator from Chapter 3.

Our design seeks to improve on the shortcomings of both software and hardware lock algorithms. Software lock algorithms often result in high network traffic overhead due to passing lock flags between cores. In addition the cache coherence overheads of invalidating and writing back lock data result in high latencies. Software algorithms also often experience high acquire times under low contention and high handoff latencies under high contention due to code complexity. On the other hand, hardware locking algorithms improve upon handoff latency and acquire times, but at the cost of hardware overhead. To these ends, we designed a hardware locking algorithm that seeks to:

- Minimize handoff time
- Minimize network traffic
- Minimize hardware overhead
- Maximize fairness

4.1 Lock Arbiter

Our algorithm revolves around a hardware structure called the lock arbiter. The lock arbiter processes *LOCK* and *UNLOCK* requests from cores across the processor. The lock arbiter maintains information about each lock in order to process lock requests quickly with minimal hardware overhead.

Each node in the system contains a lock arbiter, which communicates with the local core, router, and shared cache slice. Each lock arbiter consists of a series of lock arbiter entries. A single lock arbiter entry is utilized by one lock system-wide. The lock arbiter receives *LOCK* and *UNLOCK* requests from the thread running on the local core and from remote threads via the router. Lock placement is determined by the same striping scheme that maps memory addresses to shared cache slices. This means that locks must be identified by an address, although the lock never enters the memory hierarchy, except in case of overflow, which we will discuss in Section 4.1.2. Before we discuss overflow, however, we will go into more detail about how the algorithm works.

4.1.1 Lock Arbiter Algorithm

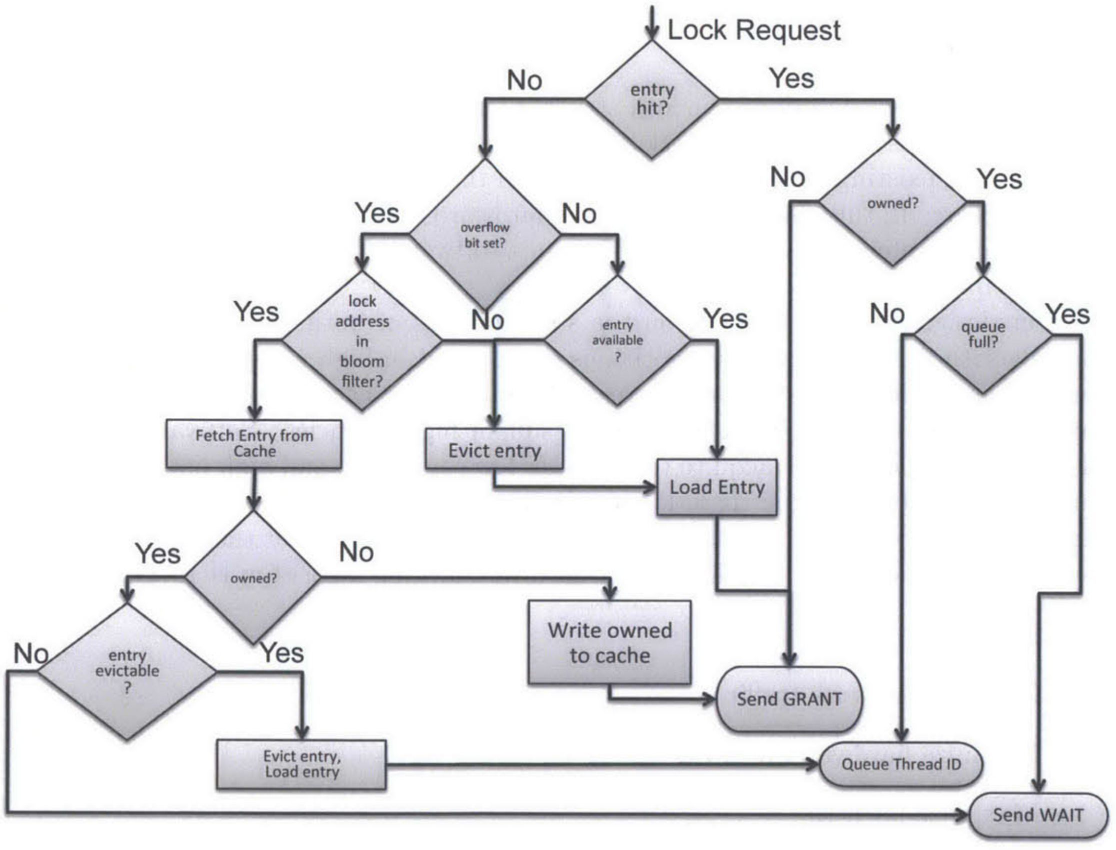

Figure 4.1: Flow chart of *LOCK* process.

In this section we describe the arbitration algorithm, which we summarize in the flow chart in Figure 4.1. When a thread needs to acquire a lock, it sends a *LOCK* request to the lock arbiter at the node with the ID equal to the low order bits of the lock's address, according to the address striping scheme. Once the *LOCK* request arrives at the correct arbiter, the lock arbiter checks if it already has a lock arbiter entry allocated for that lock address. Entries are fully associative, to fully utilize the capacity of each lock arbiter. To distinguish between locks, each lock arbiter entry has a tag, comprised of the high-order address bits, uniquely identifies the lock for which it is allocated. If there has been overflow, it is possible that the lock arbiter entry was evicted from the lock arbiter and is stored in the memory system. We describe how the lock arbiter entry is recovered in Section 4.1.2.

If no entry is currently allocated, the lock arbiter allocates a new entry for that address by loading the upper bits of the address into the tag of an available lock arbiter entry and resetting all components of the lock arbiter entry. When no entry is available for allocation, the contents of a lock arbiter entry are evicted to the shared cache as per the overflow mechanism in Section 4.1.2 and that lock arbiter entry is allocated to the new lock.

If the lock is currently unowned, the lock arbiter sends a *LOCK_GRANT* response to the requester thread and sets the *owned* bit. If the *owned* bit is already set, the lock arbiter enqueues the thread's ID in the lock arbiter queue to arbitrate locks as fairly as possible. If the thread ID is enqueued, no response is sent yet to the thread.

When the lock arbiter receives a *UNLOCK* request, if that lock's entry's queue is not empty, then the next thread ID is dequeued and a *LOCK_GRANT* is sent to the next thread. The *owned* bit remains set. If the queue is empty, the *owned* bit is cleared.

The lock arbiter queue promotes fairness by imposing a first-come-first-served order on requesting threads. However, guaranteeing this strict fairness requires the queue to be as long as the number of threads that can execute on chip, which is proportional to the number of cores on chip. A queue that depends on the number of cores does not scale well as the number of cores increases. To enable the lock arbiter to scale well, it is desirable for the queue depth to be significantly less than the number of cores. However, if a lock is highly contended, this may result in more *LOCK* requests than can be handled by the queue and the queue may become full. When the queue fills, the lock arbiter sends a *LOCK_WAIT* response to the requester with a timestamp of when the thread should try to lock again.

We utilize a timestamp so that threads do not send another *LOCK* request until it is likely that it will queue for the lock successfully. This reduces network traffic caused by retries. The time stamp is calculated based on the average critical section, since after that amount of time has elapsed the current owner will likely have released the lock and the next owner will have been dequeued, leaving an available queue slot.

The average critical section length is calculated at every *UNLOCK* request. First, the total time spent in critical sections is updated, as depicted in Equation 4.1. Then the average critical section length is calculated using the freshly-updated total critical section time and the number of grants for this lock, as shown in Equation 4.2.

$$total\_critical\_section = total\_critical\_section + current\_time - last\_grant\_time \tag{4.1}$$

$$average\_critical\_section = \frac{total\_critical\_section}{num\_grants} \tag{4.2}$$

The timestamp for when the thread should return is calculated using *average_critical_section* as shown in Equation 4.3.

$$time\_stamp\_return = average\_critical\_section + current\_time \tag{4.3}$$

If a core receives a *LOCK_WAIT* response, a small finite state machine loads the *time_stamp_return* value. When the core's current time is greater or equal to *time_stamp_return*, the core re-issues the *LOCK* request to the lock arbiter.

■ 4.1.2 Overflow Mechanism

As described in Section 4.1.1, when a lock request arrives at the lock arbiter, the lock arbiter first searches among its entries for a lock with the same tag and allocates a new entry if necessary. If no free lock arbiter entries are available (i.e., all lock arbiter entries are currently used by other locks), the lock arbiter enters overflow mode.

In overflow mode, the lock arbiter entry with the longest *average_critical_section* is evicted, and the hardware resources are recycled for use by the new lock. However, some critical information about the lock must be retained. For example, if information about whether the lock is owned or which threads are queued is lost, the application will experience race conditions or deadlock. Therefore, we store some of the data from the evicted entry in the shared cache slice at the same node where the lock arbiter resides. The lock address serves as the memory location where the overflow data is kept, which is why lock locations are striped in the same manner as data on the multiprocessor. This way the lock address can safely be used to handle overflow data, and the data can be kept at the same node for fast recovery.

In order to keep all the data on one cache line, we only store data necessary to continue processing lock requests. Therefore, we only keep one *owned* bit and the lowest 31 bits of *average_critical_section*. While the lock arbiter entry is in overflow mode, the *average_critical_section* is not updated.

When overflow does occur, a bit called the *overflow* bit is set. The address of all overflowed lock entries are also hashed into a Bloom filter [3]. The filter allows overflowed lock entries to be accurately identified by address. Bloom filters guarantee no false negatives, so if a lock address is added to the filter, it is guaranteed to be overflowed. However, the Bloom filter may result in a false positive, which means that it may say that a lock address that has not actually been placed in the shared cache has overflowed. This rare case is naturally handled by the memory hierarchy. Although false positives will result in a cache lookup, and subsequently a memory access, we believe that this case is rare enough that this expense will be inconsequential.

If an *UNLOCK* request arrives at the lock arbiter, the lock arbiter first searches the hardware lock entries for a match. If the corresponding entry is not found, the entry is retrieved from the shared cache. The *owned* bit is updated and the line is written back to the shared cache.

If a *LOCK* request arrives and the lock address tag does not match any hardware entries, the lock arbiter looks up the address in the Bloom filter. If the address is a hit in the Bloom filter, the entry is fetched from the shared cache slice. In case of a false positive, which is rare, the lock arbiter will still look up the lock address in the L2 cache, which may result in a cache miss and subsequent memory access. However, when the cache access does return, it will show that the entry is unowned, and the lock arbiter will allocate a new lock entry for the lock.

When an owned lock entry is retrieved from the shared cache, the lock arbiter checks if it can replace one of the existing hardware lock entries. If an entry has a longer *average_critical_section* than the overflowed entry, that entry is overflowed and the hardware lock entry is reallocated. The requester's thread ID is enqueued, the *average_critical_section* register is loaded from the cache line, and the *owned* bit is set. The register *num_grants* is initialized to 1 to enable *average_critical_section* updates.

4.1.3 Hardware Additions

Each lock entry contains several components to track and arbitrate locks.

1. Lock Queue – A queue of thread IDs who have requested the lock.
2. *owned* – One bit indicating if the lock is currently owned by a thread.
3. *num_grants* – Running total of the number of times a lock has been granted.
4. *last_grant_time* – Time stamp of the last time the lock was granted.
5. *total_critical_section* – Running total of the number of cycles when the lock was owned.
6. *average_critical_section* – Register with the average critical section length for this lock.

Our system requires a static number of lock arbiter entries at each core. Each lock arbiter entry has a queue of length *lock_queue_length* with width $\log n$ to store requester IDs, where n is the number of cores. Each lock arbiter entry also requires 1 bit to indicate whether the lock is currently owned. We use 32-bit registers for *num_grants*, *last_grant_time*, *total_critical_section*, and *average_critical_section*.

Thus, every lock arbiter entry requires

$$lock\_queue\_length * \log n + 1 + 4 * 32$$

bits.

We also use 100 bits for our Bloom filter, as well as 2 32-bit registers loaded with random numbers used to calculate the hash function into the filter. Only one Bloom filter is needed per node.

We also require a 32-bit register to keep *time_stamp_return* at the requester core in case of a *LOCK_WAIT* response. Therefore, for n nodes with e lock arbiter entries per node, our system requires

$$n * 32 + n * (e * (lock\_queue\_length * \log n + 1 + 4 * 32) + 100 + 2 * 32)$$

Algorithm	Operation	READ	WRITE	Atomic Operation	Total
T&S	lock	0	0	1	1
	unlock	0	0	1	1
T&T&S	lock	1	0	1	2
	unlock	0	0	1	1
MCS	lock	1	1	2	4
	unlock	2	2	1	5
CLH	lock	1	1	1	3
	unlock	0	1	0	1

Table 4.1: Software locking algorithms used in evaluation with the minimum number of operations.

bits. In a configuration with 16 cores, 4 lock entries, and a *lock_queue_length* of 4, this results in 9956 bits extra hardware across the entire chip, or just over 1KB.

4.1.4 Instruction Set Architecture Additions

The lock arbiter requires two additions to the Instruction Set Architecture (ISA). Since we have already explained how the lock arbiter handles these requests, we summarize them here.

1. *LOCK* – An instruction that indicates that a thread would like to acquire the lock at a particular address. *LOCK* returns either a *LOCK_GRANT* response or a *LOCK_WAIT* response. *LOCK_GRANT* indicates that this thread is now the owner of the lock and can continue execution. A *LOCK_WAIT* response indicates that the lock arbiter could not queue the request, and the thread should resend the *LOCK* request when the current clock time is greater than the return time.

2. *UNLOCK* – An instruction that indicates that a thread would like to release the lock at a particular address, which is currently owns. *UNLOCK* does not receive a response from the lock arbiter.

4.2 Evaluation

We evaluate our lock arbiter implementation using the trace-driven, cycle-accurate simulator from Chapter 3. We compare our results to several state-of-the-art lock implementations from Chapter 2 that rely on the cache coherence protocol to manage locks. The locks we implement, along with the number of each kind of memory operation for their *LOCK* and *UNLOCK* operations, are listed in Table 4.1. Pseudo-code descriptions of MCS and CLH can be found in Figures 3 and 4, respectively.

Paramter	Value
Issue Width	3 instructions
	(Max. 1 memory instruction)
Cache Line Size	32 bits
L1 Cache Capacity	32 KiloBytes
L1 Associativity	4
L2 Cache Capacity	Unbounded
Network Channel Delay	1 cycle
Flit size	72 bits
Lock Arbiter Capacity	4 entries
Lock Queue Depth	4

Table 4.2: System configuration for evaluation

■ 4.2.1 Simulation Setup

The default configuration we used for simulation are listed in Table 4.2. We chose parameter values based on the Tilera Gx series of microprocessors [5]. We modify the default parameters for some studies.

Microbenchmarks

```
lock lk = 0;
counter = 0;
microbenchmark_0() {
  for(int i=0;i<NUM_LOCKS;i++) {
    LOCK(lk);
    counter++;
    UNLOCK(lk);
  }
}
```

Algorithm 5: Microbenchmark with short critical sections.

We use microbenchmarks to demonstrate some properties of our algorithm and the basic performance and fairness of our system compared to other algorithms.

Our first microbenchmark, shown in Figure 5, is a simple multi-threaded program in which all threads repeatedly lock the same lock, performing a short critical system (incrementing a shared counter). The purpose of this microbenchmark is to demonstrate the lock handoff and acquire times for a highly contended lock. The microbenchmark in Figure 6 is similar to the first microbenchmark, except that delay is introduced to the

```
lock lk = 0;
counter = 0;
microbenchmark_1() {
  for(int i=0;i<NUM_LOCKS;i++) {
    LOCK(lk);
    counter++;
    for(int i=0;i<WAIT_TIME;i++) ;
    UNLOCK(lk);
  }
}
```

Algorithm 6: Microbenchmark with long critical sections.

Benchmark	Description	Lock Density	Distinct Locks
dedup	parallel-pipelined kernel using deduplication data compression	low	low
fluidanimate	data-parallel application simulating fluids for real-time animation	high	high
bodytrack_weight	particle weight calculation kernel in the bodytrack application	low	low
streamcluster	clustering algorithm with data streaming	medium	low
raytrace	tracks light in an image for animation	low	low

Table 4.3: PARSEC benchmark descriptions and lock properties.

critical section. This microbenchmark shows how the lock handoff and acquire times differ when the cost of arbitration can be hidden by a longer critical section.

Our third microbenchmark, shown in Figure 7, has threads execute several different critical sections of various lengths. This microbenchmark demonstrates how our lock algorithm adapts to critical section length at an individual lock level to reduce network traffic while still reducing lock handoff time.

Multi-threaded Scientific Benchmarks

We demonstrate the effectiveness of our system with real-world multi-threaded scientific benchmarks. We use a selection of benchmarks from the PARSEC benchmark suite

```
lock lk1,lk2,lk3,lk4 = 0;
counter1,counter2,counter3,counter4 = 0;
microbenchmark_2() {
  for(int i=0;i<NUM_LOCKS;i++) {
    LOCK(lk1);
    counter1++;
    for(int i=0;i<WAIT_TIME1;i++) ;
    UNLOCK(lk1);
    LOCK(lk2);
    counter2++;
    for(int i=0;i<WAIT_TIME2;i++) ;
    UNLOCK(lk2);
    LOCK(lk3);
    counter3++;
    for(int i=0;i<WAIT_TIME3;i++) ;
    UNLOCK(lk3);
    LOCK(lk4);
    counter4++;
    for(int i=0;i<WAIT_TIME4;i++) ;
    UNLOCK(lk4);
  }
}
```

Algorithm 7: Microbenchmark with varied critical sections.

which demonstrate real lock usage [2]. Table 4.3 lists the selection of benchmarks we used and some of their properties. Lock density indicates the relative number of lock instructions to other arithmetic and memory instructions. Distinct locks are the number of unique locks useds by the application. In all benchmarks except fluidanimate, the number of distincet locks was less than 4.

In our experiments we execute PARSEC benchmarks with test inputs for 500,000 cycles, except for dedup, which we ran to completion, and fluidanimate, which we ran to 200,000 cycles.We only compare our lock arbiter algorithm to MCS and CLH in our PARSEC experiments.

■ 4.2.2 Performance

In this section we present the performance and analyze various aspects of our system on our microbenchmarks and the PARSEC benchmark suite. We use the simulator from Chapter 3 to perform our simulations.

In Figure 4.2 we compare the performance of our lock arbiter (LA) compared with

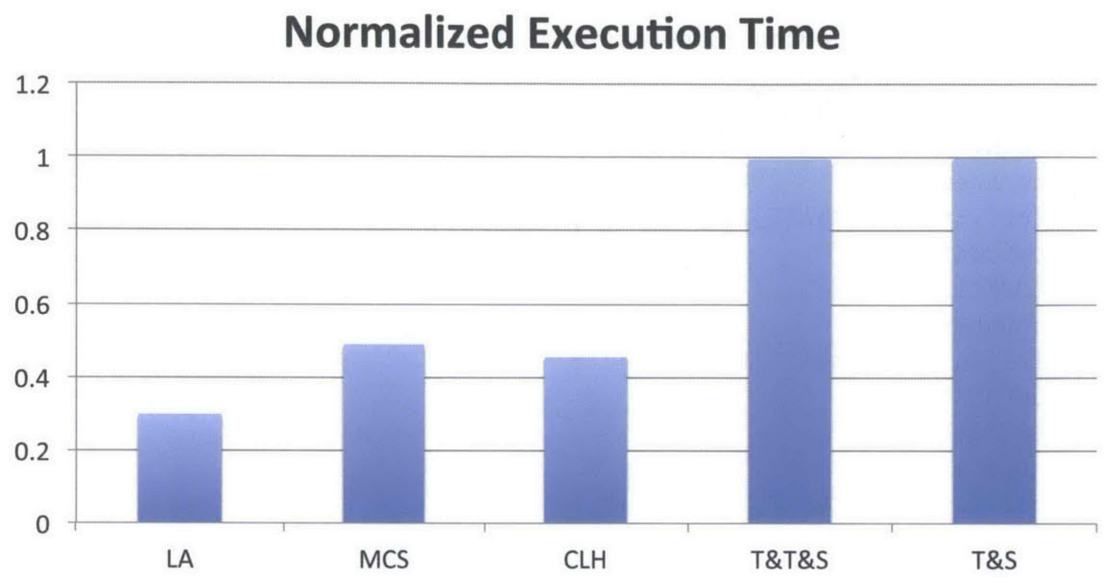

Figure 4.2: Performance of our system versus software lock algorithms.

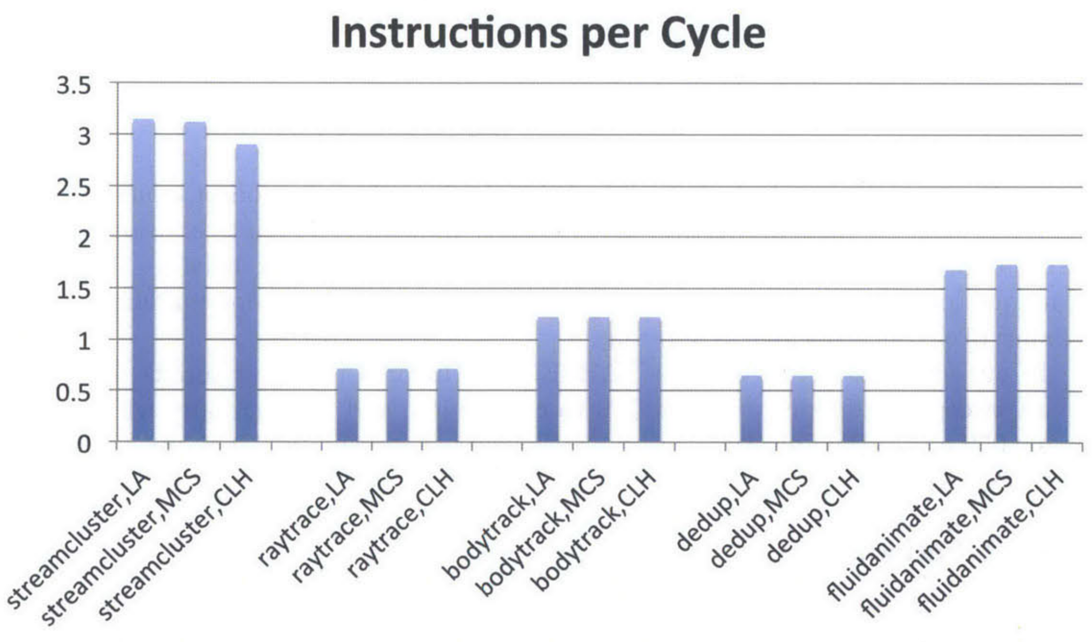

Figure 4.3: Performance of our system versus software lock algorithms for the PARSEC benchmark suite.

T&S, T&T&S, MCS and CLH on the first microbenchmark. We normalize the execution time to T&S, which takes the longest to complete the first microbenchmark. As hypothesized in Chapter 2, T&T&S performas similarly to T&S. Our lock arbiter performs best due to its fast handoff times. While MCS and CLH require several cycles for cache coherence, the lock arbiter processes locks in a single cycle.

In Figure 4.3 we compare the instructions per cycle (IPC) of LA, MCS and CLH for the PARSEC benchmarks. We use IPC to measure performance for the PARSEC benchmarks because we do not run the benchmarks to completion, but rather only run them for a predetermined number of cycles. We do not include instructions introduced by software lock algorithms in the IPC calculation. Thus, the IPC is a measure of real work accomplished by the system for the benchmark. Note that because most benchmarks have few locks, the overall performance of most benchmarks is not severely affected by the lock algorithm. An exception is streamcluster, which has high lock density and contention compared to most of the PARSEC benchmarks. The lock arbiter outperforms both MCS and CLH for streamcluster. Fluidanimate, the other PARSEC benchmark with high lock density, does not see performance improvement with the lock arbiter. This is because fluidanimate experiences lock overflow, which slightly degrades performance.

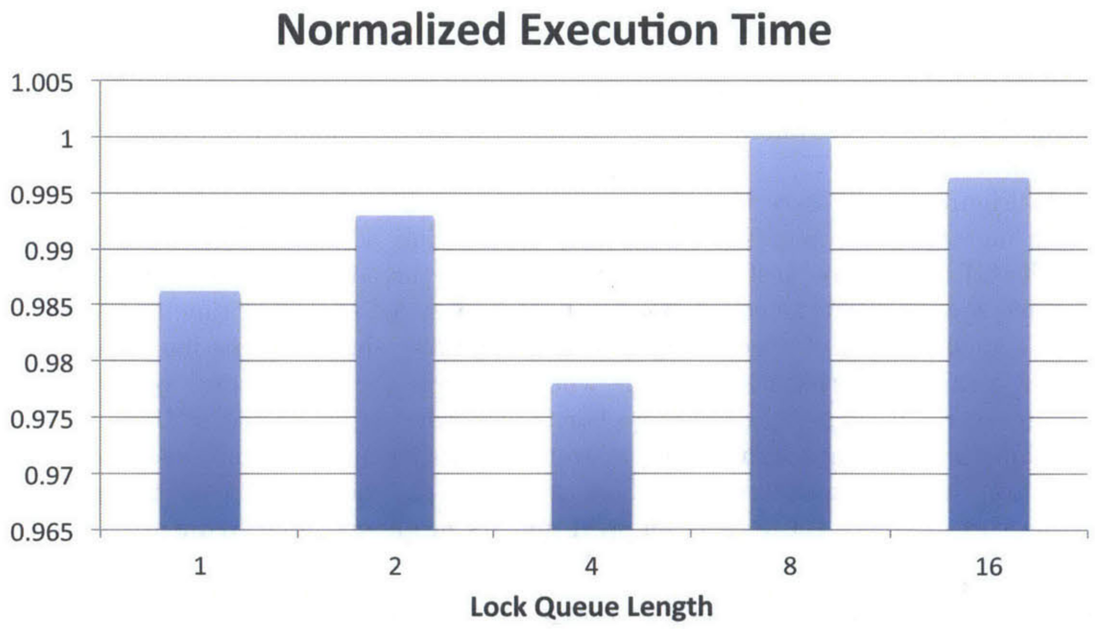

Figure 4.4: Effect of lock queue length on performance.

Lock Arbiter Entry Queue Length In Figure 4.4 we observe how the lock queue length affects the performance of the lock arbiter system when running 16 threads. While

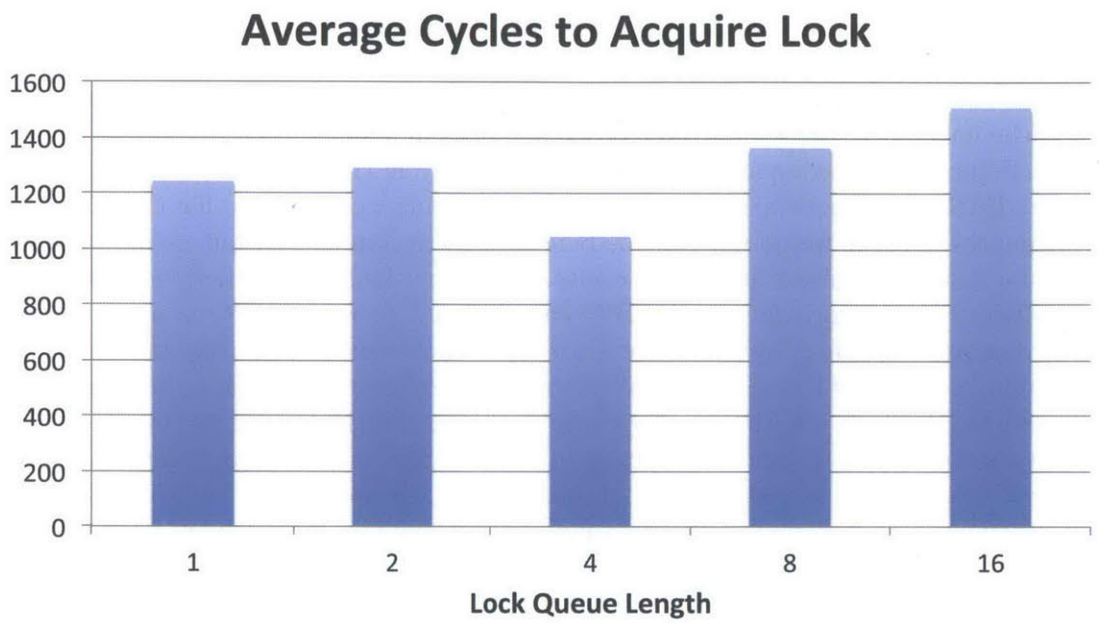

Figure 4.5: Effect of lock queue length on average acquire time.

execution time is similar for all queue lengths, we see subtle differences when we compare them. To explain the differences, we examine the average number of cycles to acquire each lock.

In Figure 4.5 we show how the average acquire time changes as the length of the queue increases. Acquire time is defined by the time a thread takes from the time the thread initially requests the lock to when the thread actually acquires the lock. The average acquire time correlates well to the normaled execution time in Figure 4.4. A lock queue length of 4 performs well because unlike shorter queue lengths, there is always a thread waiting in the queue when a lock is released. Longer queue lengths perform worse because they are more fair. Lock ownership is passed to threads across the chip, introducing communication overheads to the acquire time and the overall execution time. Lock arbiters with short queues, on the other hand, tend to grant locks to the thread where the lock is mapped and its neighbors more frequently. We will discuss fairness in more detail later in this section.

Handoff Time Figure 4.6 shows the average handoff time for the microbenchmark with highly-contended short critical sections. In this experiment we use a queue depth of 4 for the lock arbiter. We observe from this data that the our lock arbiter mechanism improves on the handoff time over MCS and CLH, which are each significantly better that T&S and T&T&S. We can attribute the improvement in handoff latency to the fact that in high contention, as is the case here, the lock arbiter's requester queue

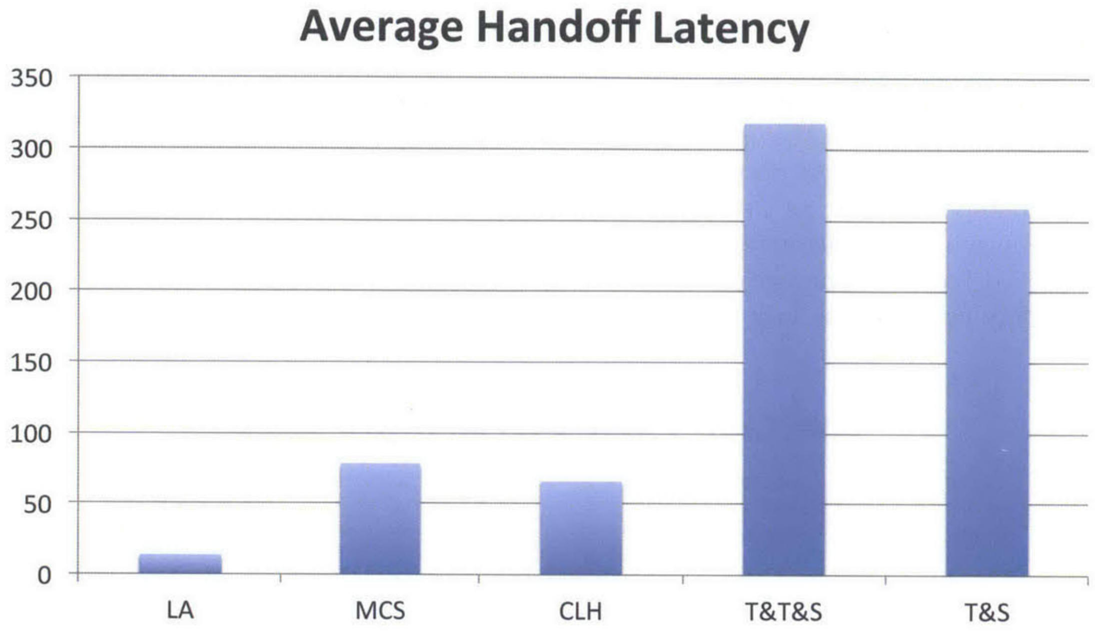

Figure 4.6: Short critical section handoff time.

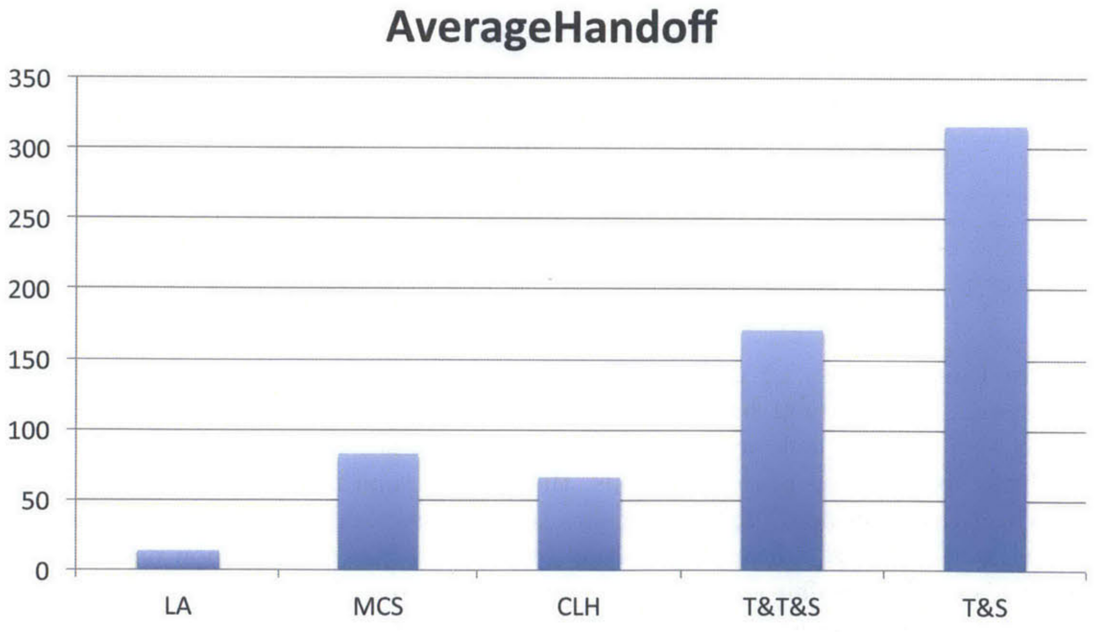

Figure 4.7: Long critical section handoff time.

holds the next thread that will acquire the lock. Then the lock arbiter only requires a single cycle to process a *LOCK* request and grant the lock to the next waiting thread. Although MCS and CLH both form queues, each requires a data access which results in a first-level cache miss in order for the owned flag to pass from one thread to another.

Figure 4.7 shows the average handoff time for a microbenchmark with highly-contended, long critical section. The handoff times for LA, MCS, and CLH are approximately the same as the first microbenchmark. We attribute this to the fact that queue formation is adequately hidden by the short critical section. T&T&S experiences improved handoff time because contending threads are able to reset and obtain their read-only copies of the lock data during the critical section.

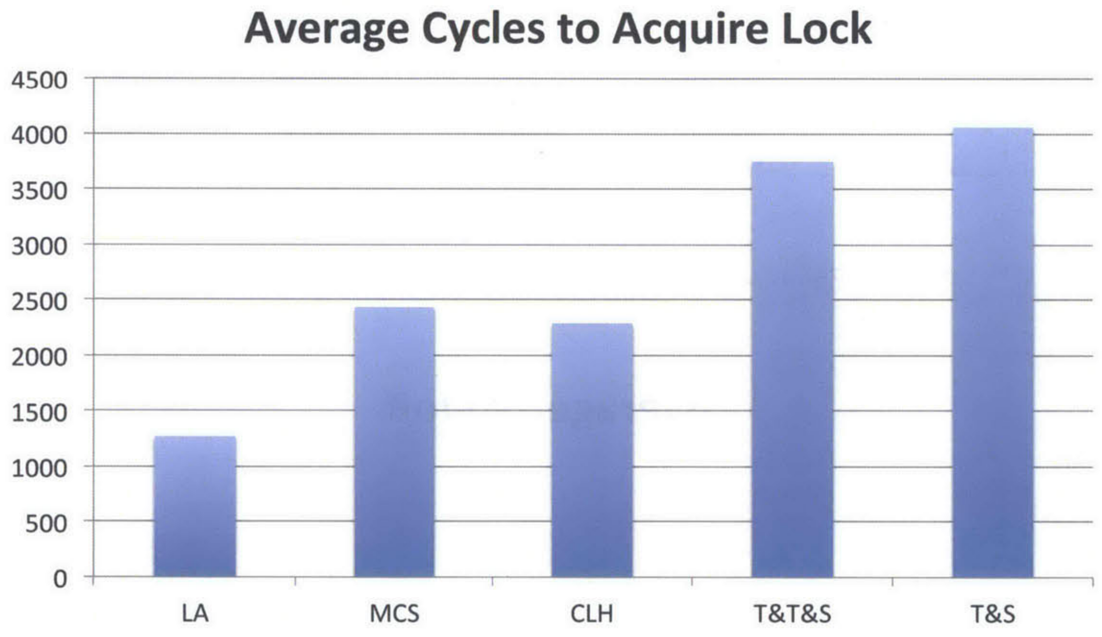

Figure 4.8: Short critical section acquire time.

Acquire Time Figures 4.8 and 4.9 show the average acquire time for the microbenchmarks with short critical sections and long critical sections, respectively. With short critical sections, the lock arbiter has lower acquire times than any of the software algorithms. In the long critical sections, the acquire time is dominated by the length of each critical section. T&T&S appears to be far superior than any algorithm with long critical sections. Closer observation reveals that this phenomenon occurs due to T&T&S's extreme unfairness, which we will discuss later in this section.

We show the average acquire time for locks in the PARSEC benchmark suite in Figure 4.10. As most of the applications do not have lock contention, the acquire times are very similar for all benchmarks. We observe that, with the exception of fluidanimate,

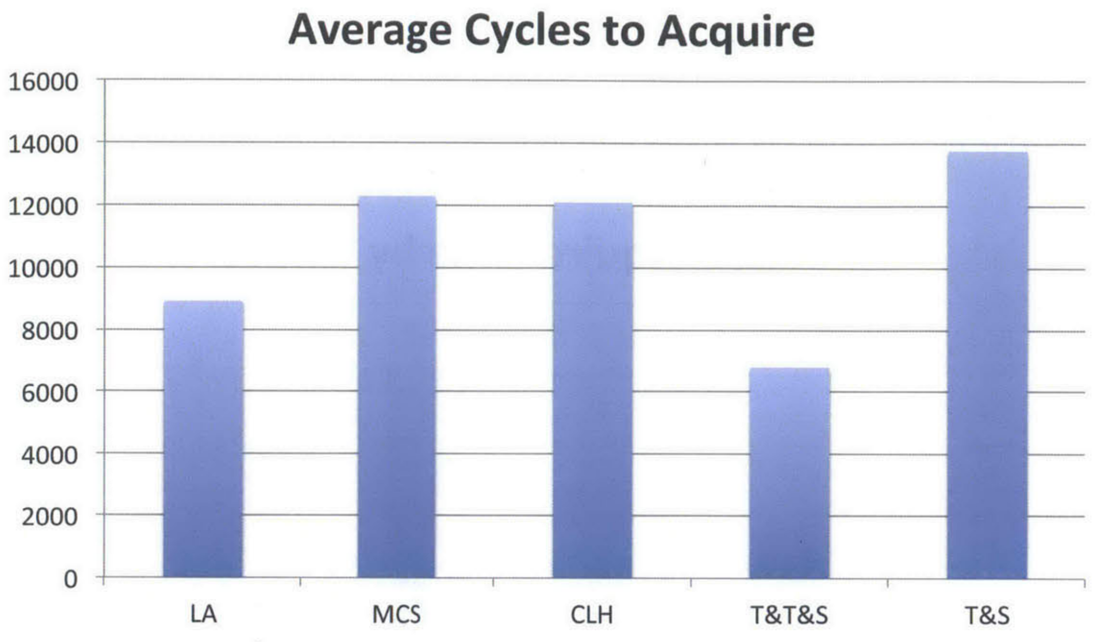

Figure 4.9: Long critical section acquire time.

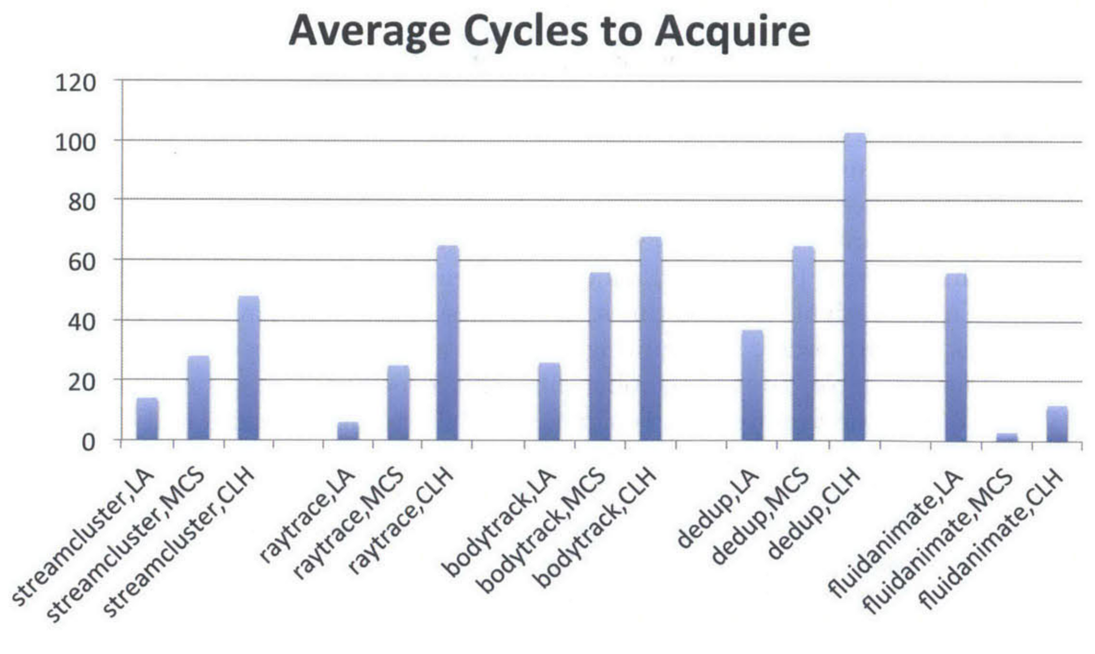

Figure 4.10: PARSEC acquire time.

the lock arbiter consistently delivers better acquire times than either MCS or CLH. We attribute this to the arbitration latency of MCS and CLH due to their code complexity in comparison to LA. The lock arbiter has high acquire time for fluidanimate due to overflow. Lock entries must frequently be fetched from the last-level cache, which adds significant delay to the lock process even if the lock is not contended.

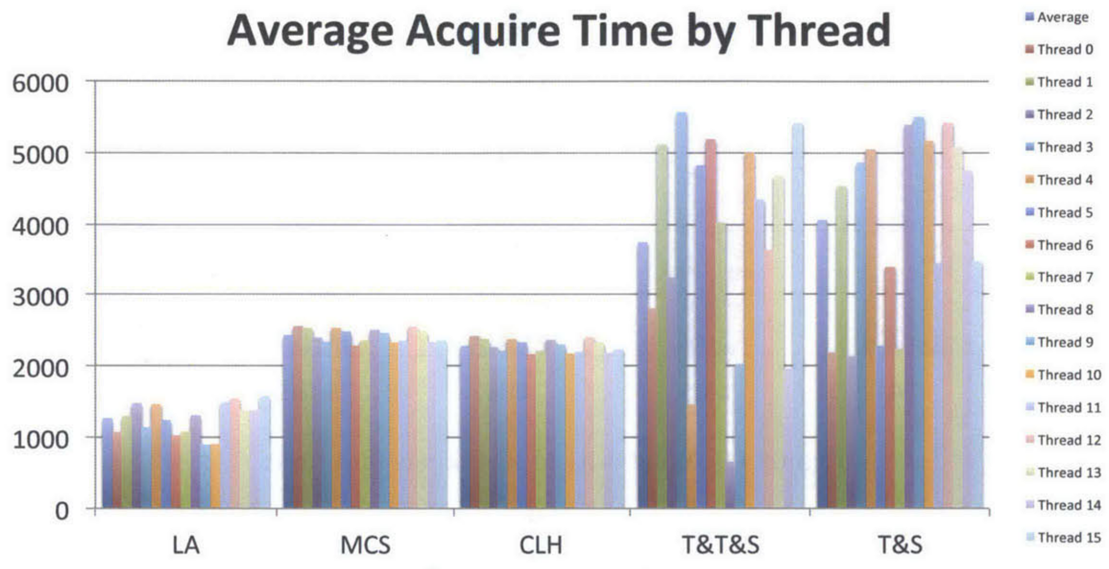

Figure 4.11: Average lock acquire time by thread.

Fairness We observe fairness examining the average acquire time per thread. If the average acquire times are roughly equal, then the lock algorithm is fair. If there is significant difference between acquire times, the lock is not fair.

We see in Figure 4.11 the fairness of each locking algorithm when used by the first microbenchmark with 16 cores and a lock queue length of 4. As expected, T&T&S and T&S have quite variable acquire times. Both these algorithms have no mechanism to control lock acquire order, so acquiring a lock is a free-for-all process every time the lock becomes available. In this microbenchmark, the lock address was mapped to node 8. We note that the acquire time is significantly shorter for thread 8 and the threads mapped close to 8, such as threads 4 and 9.

MCS and CLH are less variable, as we would expect from algorithms which form whole queues for the lock. The lock arbiter has more spread than MCS and CLH, but much less than T&T&S and T&S. The standard deviation between average thread acquire times is 225 cycles for LA, compared with 93 and 89 cycles for MCS and CLH, respectively. We would expect that increasing the queue length would lower LA's standard deviation even more.

Figure 4.12 supports the thesis that fairness and lock queue length are proportional.

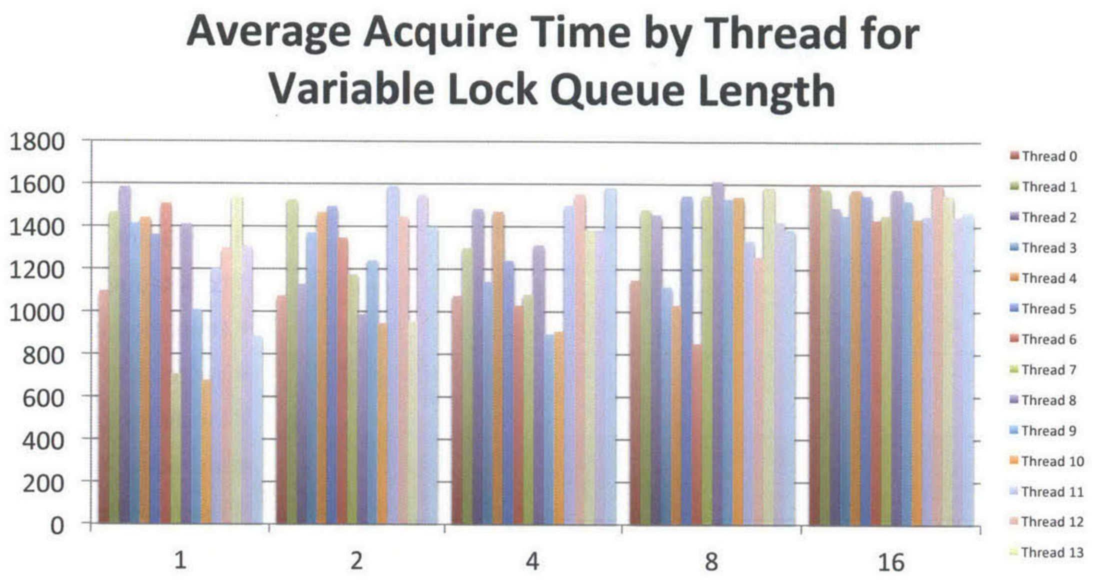

Figure 4.12: Average lock acquire time by thread for different lock queue lengths.

That is, as queue length increases, the lock arbiter's fairness improves. We observe that as the queue length increases, the spread between the thread acquire times decreases. The standard deviation for queue length 16, which can hold all requesting threads' IDs, is only 63 cycles. A queue length of 16 results in a more fair algorithm than MCS and CLH.

■ 4.2.3 Network Traffic and Contention

Cache Coherence Requests

Figure 4.13 breaks down the network traffic by request type and normalizes the proportion of each kind of request for the first microbenchmark. We observe that T&T&S has the most requests and is dominated by invalidate requests due to passing the lock flag between cores. In Figure 4.14 we zoom in to LA, MCS and CLH. We see that the lock arbiter significantly reduces network traffic due to cache coherence requests. However, these requests are replaced largely by lock requests. We also observe that a significant number of *LOCK_WAIT* messages are passed due to the high contention. In this simulation, the lock entry has only the 4 queue slots to service 16 threads, which leads to many *LOCK_WAIT* requests. While lock arbiter has similar network traffic to MCS and CLH, it does not improve upon network traffic.

Figure 4.15 shows the network traffic breakdown for the PARSEC applications. Due to the low number of lock requests in most benchmarks, we see little difference in total network traffic.

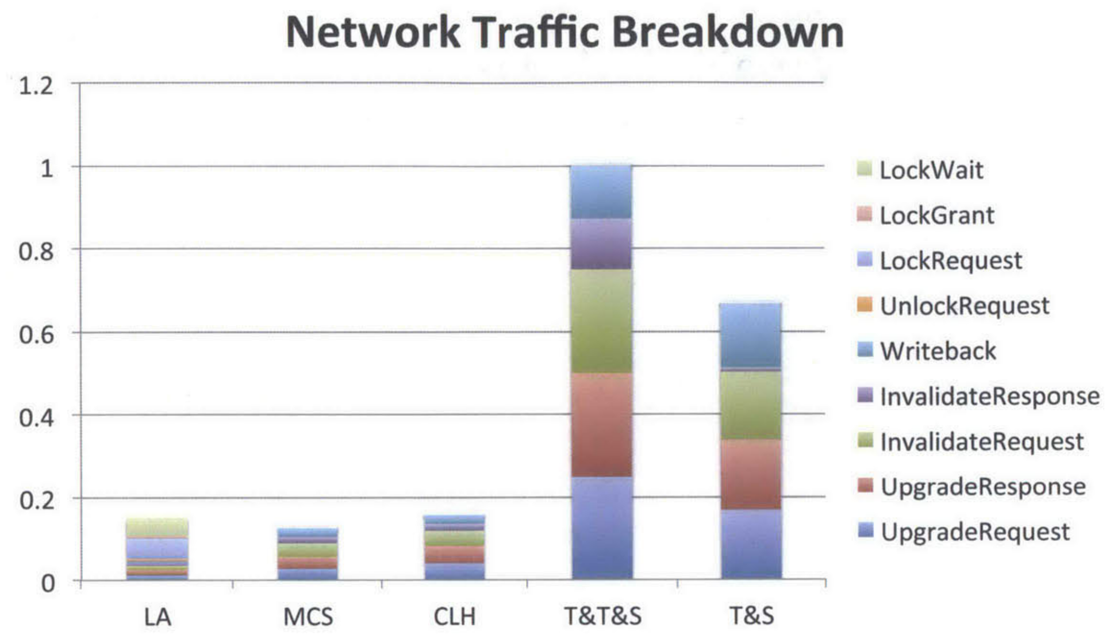

Figure 4.13: Network traffic breakdown by request type.

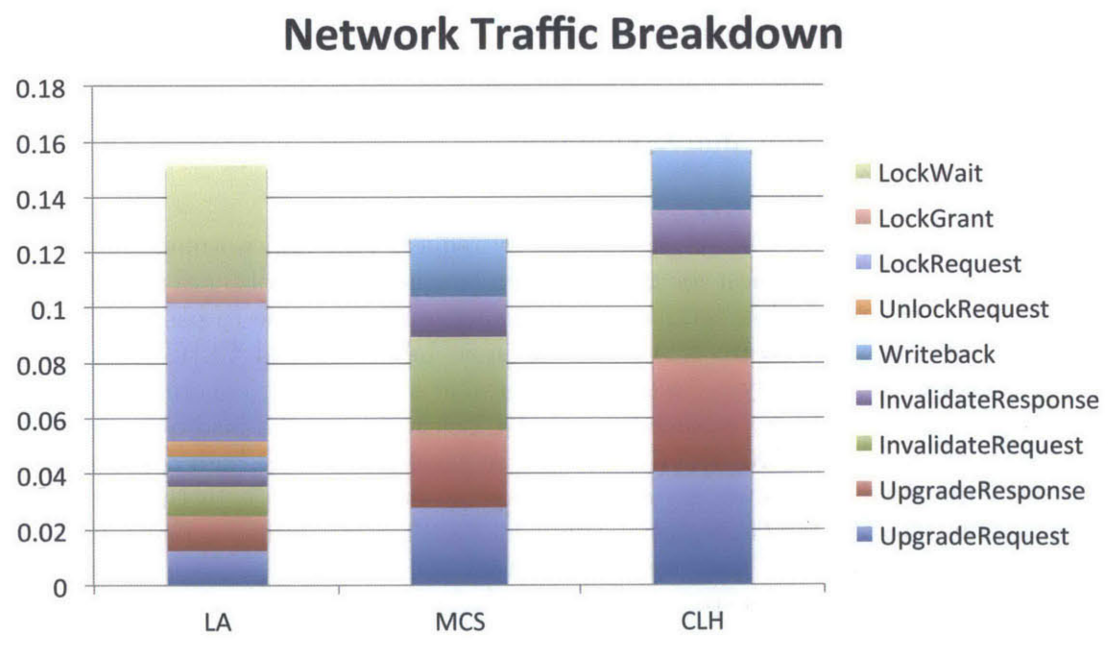

Figure 4.14: Network traffic breakdown by request type, focused on LA, MCS and CLH.

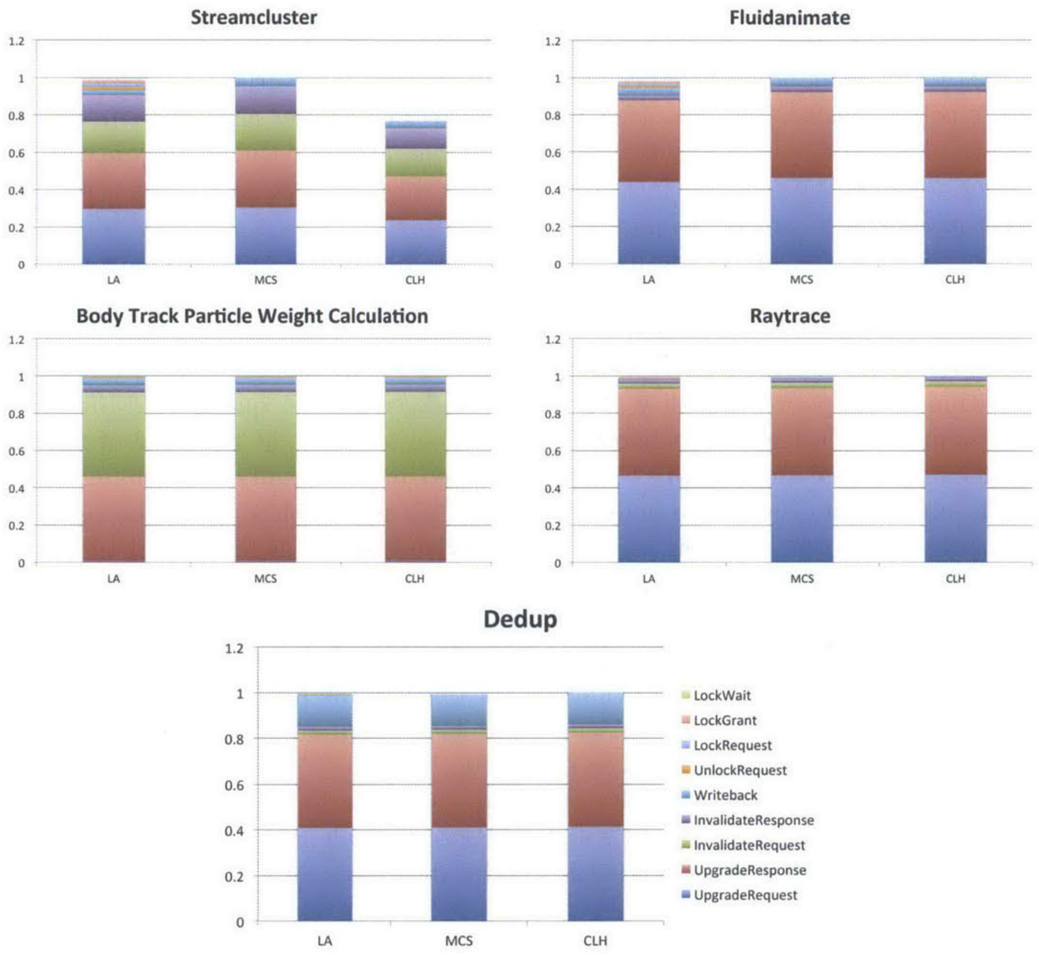

Figure 4.15: PARSEC network traffic breakdown by request type.

4.3 Analysis and Conclusions

As seen in our results from various experiments throughout this chapter, our locking mechanism has several advantages over the software algorithms. First, the lock arbiter only requires a single cycle to process *LOCK* and *UNLOCK* requests. This shortens the arbitration period of the lock when the lock is uncontended, since the requester receives a grant almost immediately. Compared to other schemes like CLH and MCS, which make several cache requests, this is a large improvement. However, if the lock is only requested by a single thread over and over again, CLH, MCS, and even T&S and T&T&S have an advantage because all the relevant lock variables are cached in the thread's private cache. On the other hand, in the lock arbiter case, the thread always has to send a request over the network to the lock arbiter. Unless the thread shares the same node as the relevant lock arbiter, acquiring the lock could cost a round trip across the chip.

Our second advantage is the hardware queue. The queue improves fairness by imposing some order to the thread requests. However, as we saw in our results, our mechanism trades off fairness and hardware overhead. A completely fair mechanism would have a queue depth to support the number of threads on chip. However, that proposition yields poor scalability. Therefore, we used shorter queues and instead simulated a queue by estimating when threads unable to queue at the time of the initial request could queue successfully. As we observed in our results, this change did not result in significant changes to acquire times. This means that, from the perspective of the threads themselves, their request was serviced almost as fairly as if there were a physical queue. The queue also has the potential to reduce network traffic. The software algorithms require multiple memory requests, which, when the lock is contended, result in increased network traffic. For example, CLH needs to pass around a copy of the tail pointer and, when the previous owner releases the lock, it needs to recover its lock flag from the other thread's cache. Our lock scheme handles this with a single remote request while there is room in the queue.

Overall, we find in these results that our hardware lock arbiter has comparable performance to the state-of-the-art software-based lock algorithms CLH and MCS and imposes reasonable fairness levels with low hardware overhead, even while locks are highly contended.

Chapter 5

Future Directions for Synchronization

THIS thesis has examined several locking schemes that have different approaches to synchronization. But what does the future have in store for synchronization? Is the lock arbiter we proposed the future of synchronization?

■ 5.1 Lock Arbiter Viability

In Chapter 4 we compared our lock arbiter algorithm against the most promising software algorithms. We found that our algorithm stacked well against the leading software algorithms in terms of overall performance, handoff time, acquire time, and fairness for both contrived microbenchmarks and real applications. So if the lock arbiter performs so well, what are the chances of it being adopted by chip manufacturers and integrated into multicore processors?

One issue we have not addressed so far in this thesis is general-purpose processing. Many servers manufactured today are designed to handle many different kinds of workloads. Our lock algorithm performs well under high contention. Its performance benefits are more easily apparent when locks are accessed frequently. However, many parallel applications do not have highly contended locks, due to their natural tendency to create bottlenecks and reduce the application to a serial execution. We observed in Chapter 4 that most PARSEC benchmarks did not benefit from the lock arbiter due to sparse lock usage.

Another issue is portability. Software locking algorithms are convenient because they can be run on virtually any multicore processor. Therefore, an application that uses these algorithms is highly portable. An application that is designed using the lock arbiter API, on the other hand, is restricted to only processors which have built-in lock arbiters.

The benefits of a hardware or hybrid hardware algorithm for locking is apparent in our results in analysis. However they may only be realized in machines developed for specific workloads.

■ 5.2 Other Synchronization Primitives

There are other synchronization primitives which this thesis did not investigate, but for which the lock arbiter could be used to improve performance.

■ 5.2.1 Conditional Wait

Conditional wait is a synchronization primitive that allows threads to wait until they are signalled by another thread to continue execution. A thread enters a conditional wait while it owns a lock. It then unlocks the lock and waits until another thread signals that the threads can continue. All the threads waiting for that specific signal then perform a lock operation, obtaining the same lock they had when they entered. Thus, conditional wait is prone to high lock contention when a signal occurs.

Lock arbiter entries could be modified to implement conditional wait primitives. In addition to maintaining locks, the lock arbiter could note which threads are waiting on a certain condition with a queue of thread IDs or bit vector. When the condition is signaled, the lock arbiter can automatically offload the threads into the lock queue to wait for the lock. This avoids the frenzy of threads attempting to acquire the lock.

■ 5.2.2 Barriers

A barrier is a synchronization primitive that prevents threads from passing the point in the execution where the barrier is called until enough threads have reached that point. It has two basic phases: collection and notification. In the collection phase, threads that reach the barrier are counted. The method of counting depends on the algorithm. For example, all threads can atomically update a centralized counter or the threads could be arranged as a tree and as threads reach the barrier, each thread notifies its parent until the root of the tree is reached. The notification phase occurs once the last thread necessary to pass the barrier reaches the barrier. In this phase all the waiting threads are notified that they can continue executing.

The lock arbiter could be modified to implement barriers as well. The lock arbiter could assist in the collection phase by being a central point for accumulating threads that have reached the barrier. The thread IDs could be added to a queue at the lock arbiter until enough threads reach the barrier. Then, the lock arbiter could dequeue the thread IDs and send each enqueued thread the notification that it can continue. This implementation would mean that each lock arbiter entry that can support barriers would need to have at least as many queue entries as the number of threads in the system. The hardware for this implementation may not scale well.

A common barrier implementation is with a lock and a conditional wait. In this implementation a thread that reaches the barrier executes a lock, increments a counter for the number of threads, and checks whether all the threads the barrier expects have arrived. If not, then the thread performs a conditional wait and releases the lock. If all the threads have arrived, the thread signals the waiting threads, releases the lock, and continues executions. The other threads then wake up, obtain the lock, then release it.

This implementation would be more suitable to our lock arbiter mechanism.

■ 5.3 Smart Synchronization

One key takeaway from the results in this thesis is that no one lock implementation is overwhelmingly better than any other. Different lock algorithms suit different applications. Factors which affect lock performance are the length of the critical section and the amount of lock contention. Therefore, as we look forward to a world with more cores on chip and more general-purpose chip multi-processors, we need to use smarter algorithms to adapt locks to applications or even phases of applications and individual locks.

Bibliography

[1] Bilge Saglam Akgul, Jaehwan Lee, and Vincent John Mooney. A system-on-a-chip lock cache with task preemption support. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '01, 2001.

[2] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), July 1970.

[4] Intel Corporation. Intel research advances 'era of tera'. 2007. URL `http://www.intel.com/pressroom/archive/releases/2007/20070204comp.htm`.

[5] Tilera Corporation. Tilera announces tile-gx72, the worlds highest-performance and highest-efficiency manycore processor. 2013. URL `http://www.tilera.com/about_tilera/press-releases/tilera-announces-tile-gx72-worlds-highest-performance-and-highest-effic`.

[6] Travis S. Craig. Building fifo and priority-queuing spin locks from atomic swap. Technical report, 1993.

[7] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* 2003.

[8] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, 2010.

[9] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ASPLOS III, 1989.

[10] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, 2010.

[11] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8), August 1974.

[12] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical clh queue lock. In *Proceedings of the 12th international conference on Parallel Processing*, Euro-Par'06, 2006.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[14] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, 1994.

[15] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1), February 1991.

[16] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, 1991.

[17] Jason Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings from the 16th Annual International Symposium on High Performance Computer Architecture*, HPCA '10, 2010.

[18] Michael Pellauer. *HAsim: cycle-accurate multicore performance models on FPGAs.* PhD thesis, Massachusetts Institute of Technology, February 2011.

[19] Zoran Radovic and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, 2003.

[20] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, 1984.

[21] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009.

[22] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language (4th Ed.)*. 1998.

[23] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. Architectural support for fair reader-writer locking. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[24] Chenjie Yu and Peter Petrov. Distributed and low-power synchronization architecture for embedded multiprocessors. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, 2008.

[25] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, 2007.