Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

____________________________  
                Jimmy Cleary  

____________________________  
                Date  


Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

____________________________  __________
Jimmy Cleary                  Date
Acknowledgements

I would like to thank my supervisor, Dr David Gregg, for the invaluable advice and guidance he has given me throughout this project, and for the extraordinary level of dedication he has for his students.

I would also like to thank all my friends and family, who have supported me this year while undertaking this project.
Abstract

This project discusses the shared-memory concurrency problem, analyses the challenges faced when implementing correct, concurrent algorithms on modern architectures, and explores the special case of lopsided access patterns to a critical section (that is, when one thread is dominant i.e. has far more accesses to the critical section than any other thread). In particular, biased and asymmetric locking algorithms described by Vasudevan et al (PACT, 2010) are implemented and experimented with and a new variation to their asymmetric algorithm is presented. Also, a new, novel approach to this particular situation and associated biased locking algorithms are described, discussed and similarly implemented and experimented with.
# Table of Contents

Declaration .......................................................................................................................... 2
Permission to Lend .................................................................................................................. 3
Acknowledgements ................................................................................................................. 4
Table of Contents ..................................................................................................................... 6
Chapter 1 Introduction ........................................................................................................... 9
  1.1 Motivation ....................................................................................................................... 9
  1.2 Project Aims ................................................................................................................... 10
  1.3 Project Outline .............................................................................................................. 10
Chapter 2 Background .......................................................................................................... 11
  2.1 The Mutual Exclusion Problem ..................................................................................... 11
  2.2 Mutual Exclusion Techniques ....................................................................................... 12
  2.3 Dekker’s Algorithm ....................................................................................................... 12
  2.4 Peterson’s Algorithm ..................................................................................................... 13
  2.5 Spinlocks ....................................................................................................................... 13
  2.6 MCS Lock ...................................................................................................................... 14
Chapter 3 Concurrency on Modern Architectures ................................................................. 16
  3.1 Caches .......................................................................................................................... 16
  3.2 Cache Coherency Protocols ......................................................................................... 17
  3.3 Impact of Memory Layout ............................................................................................. 18
  3.4 Store Buffering .............................................................................................................. 20
  3.5 Memory Barriers ........................................................................................................... 21
  3.6 Atomic Instructions ....................................................................................................... 23
  3.7 Simultaneous Multithreading (SMT) and Busy Waiting .............................................. 24
Chapter 4 Case Study: Multicore Packet Processing .............................................................. 25
  4.1 Introduction .................................................................................................................... 25
  4.2 Specification of Problem ............................................................................................... 25
  4.3 Single Threaded Approach ......................................................................................... 26
  4.4 Naive Multithreaded Approach .................................................................................... 26
  4.5 Approach Using Flow Pinning and Pipelining ............................................................. 28
Chapter 5 Lopsided Access Patterns ................................................................. 32
  5.1 Introduction ............................................................................................... 32
  5.2 Common Situations................................................................................... 33
  5.3 Lock Reservation Work ........................................................................... 33
  5.4 Relevance to Concurrent Packet Processing ......................................... 34
Chapter 6 Simple and Fast Biased Locks ....................................................... 35
  6.1 Introduction ............................................................................................... 35
  6.2 General Biased Locking Scheme ............................................................. 35
  6.3 Asymmetric Locks .................................................................................. 36
  6.4 Variation of Vasuveden et al's Asymmetric Locks .................................. 39
  6.5 Cache Considerations ............................................................................. 41
Chapter 7 Combining Message Passing with Biased Locks ............................ 44
  7.1 Initial Algorithm ....................................................................................... 44
  7.2 Asynchronous Version ........................................................................... 46
  7.3 Integrating Function Pointer Passing with a Spinlock ........................... 47
  7.4 A Thread-Safe Single-Reader Single-Writer Queue ............................... 49
  7.5 Algorithm Using Queues ....................................................................... 50
  7.6 Algorithm Using Messages .................................................................... 51
Chapter 8 Experiments/Results .................................................................... 54
  8.1 Beaker ..................................................................................................... 54
    8.1.1 Single Variable Incrementing ......................................................... 55
    8.1.2 Single Variable Incrementing with Medium Added Delay ............. 56
    8.1.3 Single Variable Incrementing with Long Added Delay ................. 58
  8.2 Zooey ....................................................................................................... 59
    8.2.1 Single Variable Incrementing ......................................................... 59
    8.2.2 Single Variable Incrementing with Medium Added Delay ............. 61
    8.2.3 Single Variable Incrementing with Long Added Delay ................. 63
  8.3 Shawbox ................................................................................................. 64
    8.3.1 Single Variable Incrementing ......................................................... 64
    8.3.2 Single Variable Incrementing with Medium Added Delay ............. 65
  8.4 Cube ........................................................................................................ 66
    8.4.1 Single Variable Incrementing ......................................................... 66
    8.4.2 Single Variable Incrementing with Medium Added Delay ............. 68
    8.4.3 Single Variable Incrementing with Long Added Delay ................. 70
Chapter 9 Conclusions..............................................................72
  9.1 Maximum Theoretical Speedup .........................................72
  9.2 “Tipping Point” ..............................................................75
  9.3 General Overview of Benchmark Results ..............................76
  9.4 Assessment of Individual Algorithms .................................77
    9.4.1 Lock Acquiring Algorithms ........................................78
    9.4.2 Synchronous Work Passing Algorithms ...........................78
    9.4.3 Non-Queue Based, Asynchronous Work Passing Algorithms ....79
    9.4.4 Queue Based Algorithms ............................................79
  9.5 Overall Assessment and Conclusions ..................................80
Appendix A Code Listing for Thread-Safe Single-Reader Single-Writer Queue ...........82
Bibliography .............................................................................84
Chapter 1

Introduction

1.1 Motivation

The problem of shared memory concurrency is over 40 years old and while there exist several algorithms which solve it in terms of correctness, the efficiency of different approaches varies drastically.

Different architectures exploit parallelism in a number of different ways, e.g. multiple processors on multiple chips, multiple cores on the same chip, simultaneous multithreading, static and dynamic instruction reordering, and vectorization. Memory models and memory organisation also vary in modern architectures, e.g. number of caches, degree to which caches are shared between hardware threads, strict vs. relaxed memory ordering and uniform vs. non-uniform memory access. Different applications will also exhibit different access patterns to shared memory.

What this all means is that determining the optimal approach to parallelizing an application relies on a number of factors involving the target architecture as well as the requirements of that application. The open ended nature of the problem makes shared memory concurrency an area with huge research potential, and the advent of multiple processors in the vast majority of modern computers make it a hugely important area of research. Both of these points were large motivating factors in deciding to perform research in this area.

This project specifically looks at the case of lopsided access patterns (explained in more detail in Chapter 5) by applications, and a class of locking algorithms called biased locks which exploit this scenario. A lopsided access pattern of accessing a critical section occurs when one thread accesses a critical section at a much higher frequency than any other thread, and a biased lock is a lock which allows a thread to “own” a lock, and access it with less overhead than other threads. Lopsided access patterns occur in a number of common applications, such as multicore packet processors using a pipelining and flow pinning approach (see sections 4.5 and 5.4), and increasing the speed of locking in these scenarios can be crucial to the efficient operation of important applications such as these.

The initial inspiration to specifically look at biased locking and lopsided access patterns came from a paper presented at PACT 2010 entitled “Simple and Fast Biased Locks”, by Vasuvedan et al. [I] This paper and the algorithms presented in it are discussed in depth in Chapter 6.
1.2 Project Aims

This project hopes to achieve the following:

- Present an analysis of issues for concurrency on modern architectures
- Explore examples of parallelization strategies geared towards modern architectures
- Present and discuss the special case of lopsided access patterns
- Discuss the biased locking algorithms from Simple and Fast, Biased Locks [1]
- Present and discuss results obtained from implementation of and experimentation with these locks
- Investigate the possibility of further optimisation of these locks or alternative biased locking strategies
- Present and discuss any results obtained from any further work completed

1.3 Project Outline

Chapter 2 explains the mutual exclusion problem, discusses approaches to solving it and presents several common algorithms used to solve it.

Chapter 3 explores the issues faced when attempting to solve the mutual exclusion problem efficiently on modern multiprocessor systems and presents several mini benchmarks run to evaluate the extent to which these issues hinder the implementation of efficient shared memory concurrency algorithms.

Chapter 4 presents a case study on multi-core packet processing, a situation in which a naïve solution is greatly outdone by a more intelligent method of parallelizing the process.

Chapter 5 discusses lopsided access patterns in detail, presents common situations in which they occur and discusses previous research on lock reservation, which targets these access patterns.

Chapter 6 discusses the Simple and Fast, Biased Locks paper and the algorithms contained in it and presents a novel variation of the Asymmetric algorithm from it.

Chapter 7 presents a novel approach to biased locking and several new algorithms based on this approach.

Chapter 8 presents results of experiments run involving the biased locking algorithms discussed in this project on several machines.

Chapter 9 presents conclusions resulting from the analysis and implementation of these algorithms and the results from Chapter 8.
Chapter 2

Background

2.1 The Mutual Exclusion Problem

Mutual Exclusion in Computer Science is a state where two or more concurrent processes do not access a common, shared resource at the same time. A region of code which accesses a shared resource is called a critical section. Critical sections in different threads involving the same shared resource cannot overlap, and thus if a thread wishes to enter a critical section involving a particular resource while another thread is in a critical section for that resource, it must either wait, or execute other work not in the critical section until the other thread has left its critical section. Otherwise, mutual exclusion would be violated.

Mutual exclusion is required for consistency of data in shared memory. This is quite easily illustrated with an example. Suppose one process wishes to increment an integer, x, and another process wishes to decrement it. The first process reads in the value of x, adds one to it, and writes it back to memory. The second process reads in the value of x, subtracts one from it and writes it back to memory. Theoretically, the result should be that x will be the same value as it was before either process, having been incremented and decremented exactly once. However, if both processes are operating simultaneously, the first process might read in the value of x, and the second process could read in the value of x in memory before the first had written the incremented value back to memory. So suppose that the value of x is 4. The following scenario could ensue:

- Process 1 reads the value of x as 4
- Process 2 reads the value of x as 4
- Process 1 adds one to the value of x it read in
- Process 2 subtracts one from the value of x it read in
- Process 1 writes the value 5 to the memory location where x resides
- Process 2 writes the value 3 to the memory location where x resides

Therefore, there needs to be a mechanism in place so that a process can have mutually exclusive access to shared memory. In the above example, process 2 should not be allowed read in the value of x until process 1 has written to it.
2.2 Mutual Exclusion Techniques

There are two approaches to ensuring mutually exclusive access to a critical section – busy waiting, or “spinning” and OS level synchronization. An important concept in mutual exclusion is that of a “mutex” or a “lock”, which is a program object which controls access to a critical section. Attempting to gain access to a critical section is often expressed as “trying to acquire a lock”. In mutual exclusion algorithms where there exists no concrete “lock” object, it is still quite common to refer to the process of attempting to access a critical section as “trying to acquire a lock”, or being in a critical section as “having a lock”.

OS level synchronization places responsibility for ensuring mutually exclusive access to a critical section on the OS. Typically, a program will interact with an OS’s multithreading library and attempt to acquire a lock object associated with a particular critical section. The OS will then suspend the process until the lock becomes available, at which point, it wakes it up. As the process is suspended, the process consumes no CPU time, and other processes can utilise this without interference from that process. However, in speed critical situations, the overhead involved in being suspended and reawakened by the OS can be significant.

Busy waiting, also called “spinning”, is a technique to ensure mutually exclusive access to a critical section without the use of OS level libraries. It involves repeatedly checking the state of a variable until it changes. Consider the following while loop:

```c
while (spinvar) ;
```

If spinvar is true, this is an infinite loop in a single threaded application. If, however, spinvar is a shared variable in a multithreaded application, then its value could be changed by another thread. Thus, it is a loop which does absolutely nothing but continually check the value of spinvar, and halts a thread’s execution until the value of spinvar is changed to false by another thread.

Algorithms using busy waiting without requiring any special atomic instructions are described in sections 2.3 and 2.4, while spinlocks, which rely on an atomic compare and swap operation, are described in section 2.5.

As their name implies, busy waiting loops are “busy” in that they consume a lot of CPU time due to the constant checking of the loop variable. Therefore, it is not advisable to use mutual exclusion protocols which rely on busy waiting unless the wait time is very short or in speed critical situations.

2.3 Dekker’s Algorithm

Dekker’s algorithm was published by Edgar Dijkstra in 1965 [2] and the first documented solution to the mutual exclusion problem for two processes. Pseudocode for the both processes is in separate columns below:

**Process 0**

```c
// lock
flag[0] = true;
while (flag[1]) {
```

**Process 1**

```c
// lock
flag[1] = true
while (flag[0]) {
```
if (turn != 0) {
    flag[0] = false;
    while (turn != 0) {
        flag[0] = true;
    }
}

// critical section

... 

// unlock

turn = 1;

flag[0] = false;

Listing 2-1 Dekker's Algorithm

2.4 Peterson’s Algorithm

Peterson’s Algorithm was conceived by Gary L. Peterson in 1981 [3] and is similar in approach to Dekker’s Algorithm. Pseudocode is provided below for two processes, though the algorithm can be generalized to N processes:

Process 0
flag[0] = 1;
turn = 1;
while (flag[1] == 1 && turn == 1) {
    // critical section
    ...
    flag[0] = 0;
}

Process 1
flag[1] = 1;
turn = 0;
while (flag[0] == 1 && turn == 0) {
    // critical section
    ...
    flag[1] = 0;

Listing 2-2 Peterson’s Algorithm

2.5 Spinlocks

A spinlock is a name given to a class of N-process mutual exclusion algorithms which involve spinning and atomic compare and swap operations. Pseudocode for an example spinlock, called a Test and Test and Set (TATAS) spinlock, is given below:

Listing 2-2 Peterson’s Algorithm

---

1 While the Dekker’s or Peterson’s Algorithm use busy waiting, and could arguably be classed as spinlocks, they are generally not referred to as such (and will not be for the purposes of this project), as they (generally) do not involve N processes or atomic operations.
Another spinlock implementation could look like this:

```c
spinlock::lock(int * lck)
{
    int success = 0;
    do
    {
        while (*lck != 0); //spin
        success = compare_and_swap(lck, 0, 1);
    } while(!success);
}

spinlock::unlock(int * lck)
{
    *lck = 0;
}
```

Listing 2-3 Test and Test and Set spinlock

In this instance, if not initially successful in acquiring the lock, a process spins on the local variable "i" for SPIN_COUNT iterations and then attempts to acquire the lock again. This can be faster than the Test and Test and Set spinlock, as it does not spin on shared variables, and thus does not cause potentially costly cache coherency traffic (cache coherency is discussed in section 3.2).

There are many different implementations of spinlocks, each one of which will have different merits and drawbacks in different situations.

### 2.6 MCS Lock

The MCS lock was described in [4] in 1991, and is a type of spinlock that scales very well to very large numbers of processors. It is a queue based lock. All threads possess a “qnode” struct which contains a “locked” Boolean variable, and a pointer to the next “qnode”. To acquire a lock, a thread checks if the queue is empty (indicated by the tail of the queue, L, being null). If it is, it inserts its qnode as the only node in the queue, and acquires the lock. If it is not, it inserts itself at the tail of the queue, sets its locked variable to true, and spins on it. When releasing a lock, a thread sets the locked variable of the qnode which is next in the
list to false, allowing its owner thread to proceed and acquire the lock. The description in [4] was studied and a resulting implementation is provided below:

```c
#define CAS __sync_bool_compare_and_swap
typedef struct _qnode {
    struct _qnode * next;
    bool locked;
} qnode;

void acquire_lock(qnode ** L, qnode * I)
{
    I->next = NULL;
    qnode * predecessor = __sync_lock_test_and_set(L, I);
    if (predecessor != NULL)
    {
        I->locked = true;
        predecessor->next = I;
        while(I->locked) ;
    }
}

void release_lock (qnode ** L, qnode * I)
{
    if(I->next == NULL)
    {
        if(CAS(L, I, NULL))
            return;

        while(I->next == NULL) ;
    }
    I->next->locked = false;
}
```

Figure 2-1 MCS Lock

---

2 This is an atomic builtin intrinsic in gcc. It is not a traditional test and set operation, but rather an atomic exchange instruction which returns the original value [7], and which functions similarly to the atomic fetch and store required by the MCS lock, as described in [4]
Chapter 3

Concurrency on Modern Architectures

Modern architectures come with several features which greatly improve speed of program execution, but which complicate how concurrency is implemented. These include caches, coherency between different CPU’s caches and store buffering. There also exist atomic instructions, which perform Read-Modify-Write operations which appear as if they were one instruction to other CPUs, as well as memory barriers (fences) which serve to help the programmer implement correct, concurrent algorithms.

3.1 Caches

A CPU cache is a small and fast memory, usually built into a processor chip, which stores copies of data from frequently accessed memory locations. It is much faster to fetch the contents of a memory address which is in a processor’s cache than to retrieve it from main memory, and generally programs tend to access the same addresses repeatedly, meaning that storing copies of these frequently accessed locations in a smaller, faster memory closer to the processor greatly improves performance.

When a processor attempts to access the contents of a memory location which is not currently in its cache, this is called a cache miss. When a cache miss occurs, the processor must fetch the data from main memory and the cache will allocate an entry for that memory location, and usually surrounding addresses, evicting another entry to make room (usually an approximation of the least recently used entry).

Factors which affect a cache’s performance include ways of associativity and cache line size. Ways of associativity means how many cache locations a copy of a region of memory can be stored, and cache line size means how large a region of memory can be stored in an entry in the cache (known as a cache line).

Modern processors tend to have multi-level caches. What this means is that rather than just having one cache, they have several which are arranged in different levels a cascaded fashion. When a processor wishes to read the contents of an address, it first checks its L1 cache, if there is a cache miss, it checks for it in an L2 cache, which is larger. On an L2 cache miss, if there is an L3 cache, that is checked, and so on.

While L1 caches are generally private to a single core (though potentially shared between two hardware threads if SMT is enabled), L2 and L3 caches are often shared between two or more cores.

When writing to a memory location, a processor can either automatically write the value to both its cache and through to main memory (write-through), or just to its cache (write-back),
only writing the value to main memory at a later stage. As writing through to memory is expensive, the latter is more efficient, and thus is most widely used.

Write-back caches will present a problem in multi-core situations, as CPUs generally do not share L1 caches, and thus if a CPU wishes to read the contents of an address, the correct value of which is only contained in another CPU’s cache, it will not be able to access it until it is written to memory by the other CPU. To address this problem, CPUs in multi-core machines need to obey a cache coherency protocol.

### 3.2 Cache Coherency Protocols

Cache coherency protocols are used to address the problem of copies of data stored in different CPUs’ private caches potentially not being consistent with the copy in main memory or the private caches of other CPUs.

For example, in situation with two CPUs, each with a write-back L1 cache, if one CPU reads a value from memory and writes to it, the value is written back to that CPU’s cache, and not through to memory. Therefore, should the other CPU read the value in main memory, it will be inconsistent with the updated value in the first CPU’s cache, and thus, incorrect.

Therefore, a mechanism to signal to other processors that the copy of a value in main memory is not the current value is necessary.

One cache coherency protocol, which is used in modern x86 processors, is MESI, which uses a bus snooping mechanism. In this protocol, processors monitor the address bus for reads and writes to addresses by other processors which are in their cache. Each cache line in a cache has a state associated with it, and can be in one of 4 states, which are detailed in the table below [5]:

<table>
<thead>
<tr>
<th>Cache Line State</th>
<th>M (Modified)</th>
<th>E (Exclusive)</th>
<th>S (Shared)</th>
<th>I (Invalid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This cache line is valid?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>The memory copy is...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Out of date</td>
<td></td>
<td>Valid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copies exist in caches of others?</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>A write to this line...</td>
<td>Does not go to the system bus.</td>
<td>Does not go to the system bus.</td>
<td>Causes the processor to gain exclusive ownership of the line.</td>
<td>Goes directly to the system bus.</td>
</tr>
</tbody>
</table>

Table 3-1 MESI Cache Line States

When the contents of a memory location which is not present in any other processors’ cache are initially read into a processor’s cache, the associated cache line is assigned an Exclusive state. When that memory location is read by another processor, the state of the associated cache line in both processors is set to Shared. If either processor writes to this cache line, the value is written through to memory, the writer’s cache line is set to Exclusive, and others’ lines are set to Invalid. A write to a line in the Exclusive state is written only to that
processor’s cache and changes its state to Modified. All writes to Modified cache lines are written only to that processor’s cache.

As is detailed in Table 3-1, reads to Invalid cache lines cause the processor to fetch a copy of that memory location from main memory, i.e. a cache miss. In all other states, reads to a memory location in the processor’s cache will result in a cache hit.

### 3.3 Impact of Memory Layout

It is quite apparent that frequent cache line invalidations are not desirable. However, it can seem as if they are vital for correctness. This is not always the case, however, and by taking into account the size of a cache line in a processor, great performance increases can be gained from how necessary variables in multi core situations are laid out in memory.

Consider the following code:

```c
#include <pthread.h>
#include <iostream>

typedef struct {
    int w;
    int x;
    int y;
    int z;
} threaddata;

void writex(threaddata * td) {
    for(int i = 0; i < 1000000000; i++)
        td->x = i;
}

void writey(threaddata * td) {
    for(int i = 0; i < 1000000000; i++)
        td->y = i;
}

void writez(threaddata * td) {
    for(int i = 0; i < 1000000000; i++)
        td->z = i;
}

void writew(threaddata * td) {
```

18
for(int i = 0; i < 1000000000; i++)
    td->w = i;
}

int main()
{
    pthread_t threads[4];
    threaddata * td = new threaddata;

    std::cout << "w: " << &td->w << std::endl;
    std::cout << "x: " << &td->x << std::endl;
    std::cout << "y: " << &td->y << std::endl;
    std::cout << "z: " << &td->z << std::endl;

    td->w = td->x = td->y = td->z = 0;

    pthread_create(&threads[0],NULL,(void* (*)(void*)) writew, td);
    pthread_create(&threads[1],NULL,(void* (*)(void*)) writex, td);
    pthread_create(&threads[2],NULL,(void* (*)(void*)) writey, td);
    pthread_create(&threads[3],NULL,(void* (*)(void*)) writez, td);

    for(int i = 0; i < 4; i++)
        pthread_join(threads[i], NULL);

    return 0;
}

Listing 3-1 Shared Cache Line Benchmark

On beaker (see section 8.1), compiled in g++ with no optimisations, this executes in approximately 18 seconds.

Now consider changing the threaddata struct to the following. Also note that the cache line size in the L1 caches of the CPUs in this machine is 64 bytes:

typedef struct {
    int w;
    byte padding[64];
    int x;
    byte padding2[64];
    int y;
    byte padding3[64];
    int z;
} threaddata;

Listing 3-2 Private Cache Line Benchmark
On the same machine, compiled in exactly the same way, this executes in 3 seconds. Six times faster.

The reason for this is that since there is padding of 64 bytes between the integers in the struct, each integer is guaranteed to be in a different cache line\(^3\). When we inspect what each of the threads are doing, we see that they are each writing to a different variable 1,000,000,000 times. If all four variables are in the same cache line, this means that every time any processor accesses any of these variables, it invalidates the cache line for all other processors, meaning these processors will have a cache miss when they try to access another variable in that cache line. In the above example, despite cache misses, the value of the variable each thread wishes to access every time is the same as the value in their cache.

By introducing padding, which g++ does not optimize away as we did not pass the O flag, each variable is in its own cache line, and thus cache misses are eliminated.

In contrast, both versions of the above program (modified to use 2 threads) execute in the same amount of time on a PPE of a cell processor. This is because the PPE is a simultaneous multithreaded processor, with two hardware threads available. As such, both threads share the same cache, and there is no difference between variables being stored in shared or private cache lines.

While the above example does not involve shared memory, it is important as it shows how memory layout is important in multithreaded situations. Unless the threads are running on the same core via SMT, data which is local to different threads should be in independent cache lines where possible. It also indicates that it could be beneficial to have lock variables in separate cache lines to the shared data they protect, and also in separate cache lines to locks protecting other shared data.

### 3.4 Store Buffering

Most modern architectures have a relaxed memory model \[^6\]. What this means is that reads from and writes to memory are not necessarily done in order and thus are not necessarily immediately visible to other processors.

The motivation behind relaxed memory models is that memory accesses incur a significantly higher latency than computation, and from the perspective of a single thread, it is possible to exploit non-strictly ordered memory accesses to gain speedups, while overall maintaining correctness.

For the purpose of this project, we are interested in the impact of write or store buffers. Whether writes to are immediately written to a processor’s cache or not does not affect the execution of a single threaded program. Instead, writes are put into a store buffer by the processor, where they wait to be potentially written to the processor’s cache. The latency of a write to the cache is therefore immediately avoided. Furthermore, should the processor read or write to the same value again, the entry in the store buffer can be removed / updated, and thus potential latency that might have been incurred by immediately writing to the processor’s cache is completely eliminated. Cache lines are also usually written to in

\(^3\) When programs are compiled in g++ without optimizations enabled, variables which are not used are not ignored, and memory is allocated for them.
chunks of one line at a time (on x86, 64 bytes), and therefore an optimization called “write-combining” allows multiple writes to be merged together in the store buffer.

A diagram of a typical modern memory architecture, taken from [6], is given below. Note that the diagram details a distributed memory architecture connected via an interconnection network. In a shared memory system, the last level of caches would be connected to the same main memory.

![Diagram of typical memory architecture](image)

Figure 3-1 Typical Memory Architecture

### 3.5 Memory Barriers

Relaxed memory models which use store buffering, as described above, have implications for concurrency on shared memory multiprocessors. Since a write is not visible at all to any other processor than the one that performed it until it is written to its cache (thus invalidating the cache lines of the other processors), if a write resides in a processor’s store buffer, another processor will not immediately be able to read the value of this write, and this can break many textbook parallel algorithms, for example, Dekker’s Algorithm:

**Process 0**

```c
// lock
flag[0] = true;
while (flag[1]) {
    if (turn != 0) {
        flag[0] = false;
        while (turn != 0);
    }
}
```

**Process 1**

```c
// lock
flag[1] = true
while (flag[0]) {
    if (turn != 1) {
        flag[1] = false
        while (turn != 1);
    }
}
```
If Dekker’s algorithm is being used to synchronize memory access between two processors, and writes are buffered and thus not instantly visible by all processors, it is easy to see that the above version of the algorithm will break. If neither process is in the critical section and process 0 wishes to enter it, it writes true to its flag variable and since process 1’s flag variable will be false, it continues on into the critical section. Its write to its flag variable is not instantly visible, however, and if process 1 attempts to access the critical section, it will see flag[0] as being false, skip over the while loop and enter the critical section, where process 0 may very well still be, and thus mutual exclusion is violated.

In order to remedy this problem, modern architectures provide memory barriers, or “fence” instructions. A fence instruction flushes store buffers, and writes all pending writes to memory. Therefore, a fence instruction can ensure visibility of all writes to all processors at a certain point in code. The above Dekker’s algorithm can therefore be fixed with the addition of memory barriers as shown below:

**Process 0**

```c
// lock
flag[0] = true;
fence();
while (flag[1]) {
    if (turn != 0) {
        flag[0] = false;
        while (turn != 0);
        flag[0] = true;
        fence();
    }
}
```

```c
// critical section
...
```

**Process 1**

```c
// lock
flag[1] = true;
fence();
while (flag[0]) {
    if (turn != 1) {
        flag[1] = false;
        while (turn != 1);
        flag[1] = true;
        fence();
    }
}
```

```c
// critical section
...
```

Listing 3-4 Dekker’s Algorithm with Memory Fences
Now when process 0 or 1 wishes to enter the critical section, after it has set its flag variable to true, it executes a fence instruction, ensuring that the value of flag[0] or flag[1] can be read by the other processor, before continuing on and potentially entering the critical section, thus avoiding the situation above.

Appropriate insertion of memory barriers in concurrency algorithms is essential for correctness. At the same time, as they tend to be costly instructions. To evaluate the cost associated with a memory barrier, a simple benchmark was implemented on beaker and zooey (see sections 8.1 and 8.2 for specs), in which the speed of a variable being written to 1,000,000,000 times was compared to a variable been written to that amount of times with a ‘mfence’ instruction following each write. On beaker, the version with the immediate fence was just under 4 times slower, whereas on zooey it was approximately 5 times slower.

Therefore, the number of memory barriers necessary for a correct implementation is an important consideration when designing modern concurrency algorithms.

### 3.6 Atomic Instructions

Atomic operations are operations which appear to execute instantaneously. An atomic instruction is an instruction which guarantees to perform an operation atomically. For example, an atomic read guarantees that the value being read will not be modified before it is fully read. Word, doubleword and quadword reads and writes are generally guaranteed to be atomic in most modern x86 architectures, especially on aligned boundaries [5], but older machines did not have this guarantee – for example a quadword read might actually have consisted of two doubleword sized reads, allowing for another thread to potentially modify the second doubleword before the whole quadword was read.

Other atomic instructions exist on modern architectures purely to aid concurrent programming, which perform a sequence of operations atomically. The most commonly used atomic instruction is a “Compare and Swap” or CAS. This compares a value with the contents of a memory location, and if they are the same, writes a new value to the memory location and indicates that the operation succeeded. In pseudocode:

```c
bool compare_and_swap (T * mem, T old, T new) {
    if (*mem == old) {
        *mem = new;
        return true;
    } else {
        return false;
    }
}
```

Listing 3-5 Compare and Swap

An atomic compare and swap instruction does the above as if it were instantaneous, so another processor could not, for example, change the value of the contents of mem at any point while another process is performing a CAS on it.
The reason an atomic CAS is useful for synchronization is if as part of a concurrency algorithm, if multiple threads wish to write to a variable, and only one can succeed, they can all attempt to write to it, but only the thread which executed its CAS first will succeed, because having changed the contents of the memory location involved, other processes’ CASs will fail. This is particularly useful in busy waiting/spinlocks (described in the previous chapter).

On x86 machines CMPXCHG is an atomic CAS instruction. GCC provides an intrinsic atomic CAS function [7]:

```c
__sync_bool_compare_and_swap(type* ptr, type oldval, type newval)
```

This avoids the need to write assembly code when writing synchronized code using atomic CAS instruction.

To investigate the speed of an atomic compare and swap instruction compared to a write, a simple benchmark was implemented on beaker and zooey (see sections 8.1 and 8.2 for specs), in which the speed of writing to a variable 1,000,000,000 times was compared with the speed of writing to a variable 1,000,000,000 times using a compare and swap instruction. On beaker, the compare and swap version was over 7 times slower than using normal writes, whereas on zooey it was over 3.5 times slower.

### 3.7 Simultaneous Multithreading (SMT) and Busy Waiting

Simultaneous Multithreaded (SMT) processors are processors which allow two threads to simultaneously execute. The result of this is that there can be two or more “logical processors”, but only one physical processor. It works by performing very fast context switching between threads, fetching instructions from them and executing them all in the same pipeline.

Busy waiting has very negative performance implications for SMT processors, as it consists of constantly performing a lot of useless work, i.e. checking the value of a variable repeatedly. This causes resources to be needlessly allocated to this thread that could have been used by another thread running on the same processor.

The solution is to sparingly use busy wait loops on SMT processors, and if used, they should be made less “busy” by adding NOPs into the body of the loop, which will delay the checking of the loop variable, and improve overall performance.

Intel provides a “pause” instruction [8] [5] which serves as a hint to the processor that the loop it is contained in is a busy wait loop, and it helps alleviate the strain the loop may put on an SMT processor by executing NOPs rather than constantly checking the loop variable.
Chapter 4

Case Study: Multicore Packet Processing

4.1 Introduction

As network speeds become faster and faster, it is imperative that the speed of processing incoming packets also increases to an acceptable level, such that it does not become a bottleneck when it comes to effective network throughput. As a result, improving the speed of packet processing by means of parallelization is very important. It is also a very relevant example of shared memory concurrency.

In his article, Understanding Packet Processing with Multicore Processors [9] (based on Intel’s “Supra-linear Packet Processing with Intel Multi-core Processors” white paper [10]), Edwin Verplanke details how packet processing was dealt with in single threaded versions of Snort [11], an open source packet analysis and intrusion detection application, and discusses strategies to take advantage of multicore processors to parallelize Snort and speed up processing. As will be seen, the best solution is not the simplest, most straightforward one.

In this chapter, packet processing and the approaches to multi-core packet processing described in Verplanke’s article are discussed. A fake packet processing simulator was designed and implemented for this project and was modified to use a single thread, to use multiple threads naively, and to use Verplanke’s pipelining and flow-pinning approach. This is described and results of experiments using it are presented and discussed. Results from experiments from Verplanke’s article, which involve the modification of Snort to use naïve and smart multithreading are similarly presented and discussed.

4.2 Specification of Problem

Packet processing involves the analysis of network packets arriving in real time. Typically, one of the most basic initial tasks that a packet processor must perform is to reassemble fragmented packets. This is usually done by means of a table containing entries for partially reassembled packets. Once packets have been reassembled, they can then be analysed as per the purpose of the particular packet processing application e.g. an intrusion detection application such as Snort.
4.3 Single Threaded Approach

The Snort dataflow is shown in Figure 4-1 (from [9]). In the single threaded approach, this simply runs on one core. Once packets are captured and decoded, they are passed to the pre-processors. The primary function of the pre-processors is to reassemble fragmented packets. Once reassembled (and also potentially having some other pre-processing performed on them), they are passed to the Detection Engine, which identifies suspect packets.

The fake packet processing simulator written for this project does not have a detection engine, and simply reassembles fake packet fragments in a reassembly table.

![Snort dataflow diagram](image.png)

Figure 4-1 Snort Dataflow

4.4 Naïve Multithreaded Approach

The diagram below details how the naïve multithreading approach works. It simply replicates the packet processing dataflow on each core. All cores pull packets off the network interface, decode them and put them through their pre-processors. Again, we focus on the example of packet reassembly. A reassembly table, which is shared between all cores, is used.

Intuitively, it might seem that since multiple cores are working on processing packets as opposed to one, that it will naturally be faster. However, it was found that this approach is no faster than the single threaded approach (as verified by experiments presented at the end of this chapter).

One reason for this is that the reassembly table is shared between several processors, and thus must be protected by a lock. Therefore, before a decoded packet can be inserted into
the table, the thread which is processing it must acquire the table’s lock. This involves both locking overhead, and waiting for other threads to relinquish the lock held by them, which effectively means that insertion into the reassembly table is not done in parallel. If insertion into the table is a significant percentage of the overall running time of each thread, then gains from parallelizing the process are significantly impacted.

Cache misses are another problem. An issue which also applies to the single threaded version of the application is that a reassembly table is likely to be large, and thus potentially span several cache lines. Any packet a thread reads could have to be inserted anywhere in the table, and thus there is a high potential that threads’ memory access patterns will not exhibit locality of reference, incurring cache misses and slowdowns.

The problem of cache misses is worsened in the naïve multithreaded version. The reason for this is the cache line invalidations which occur as a result of multiple threads inserting into the same reassembly table. For example, if one core fetches a packet, which is a fragment of a larger packet, from the network interface, decodes it and inserts it into the reassembly table, and another core fetches another packet, which is a fragment of the same larger packet, the latter thread’s insertion into the reassembly table will invalidate the former core’s cache line containing the location where it just inserted a packet fragment into. Every time this happens there will be one or more cache misses which would have been avoided in the single threaded version.

To combat the problems of all threads having to acquire a lock to insert into the reassembly table, lack of locality of reference and detrimental cache line invalidations, techniques called pipelining and flow pinning can be used.

Figure 4-2 Naïve Multithreaded Approach
4.5 Approach Using Flow Pinning and Pipelining

The flow-pinning and pipelining approach is described in [9] as being somewhat analogous to a common method of making registering attendees at a conference more efficient. It could also be similarly compared to how a university might speed up student registrations. Typically, N registrars of attendees sit behind N desks, and each registrant queues behind one of these desks. When a registrant reaches the top of a queue, the registrar looks up their name in a list of attendees and ticks off their name.

When N=1, this is directly analogous to the single threaded approach to packet reassembly. There is one queue behind one desk where a registrar sits. People represent packets, a person joining the queue represents a packet coming in on a network interface, the registrar represents a processor and the list of attendees represents the reassembly table. A person reaching the top of the queue represents a packet captured by the processor, and the registrar ticking their name off the list of attendees represents insertion into the reassembly table.

When N is greater than 1, intuitively the process should be faster. With more registrars, queues will be shorter and people can potentially be registered simultaneously.

However, there are two issues to consider with increasing the number of registrars. The first is how do people choose which desk to line up behind. The second is that only one registrar can hold the list of attendees at any one time.

One way this might be arranged would be to have registrants enter the room without guidance, and pick the shortest queue to line up in. When a person reaches the top of a queue, if the registrar does not have the list of attendees, they must signal to the other registrars that they need the list, and wait until it is passed to them. They then can look up the person's name and tick them off.

Of course, this approach presents serious issues which hinder efficiency. The first is also present when N=1. A person reaching the top of the queue could have a name which is not at all alphabetically close to the person before them, and the registrar would have to search for it in a completely different part of the list. It would be much more efficient if the person had a name which was on the same page or within a page or two of the last registrant.

The other issue is that registrars would be spending a lot of time waiting, with people at the top of their queues, for other registrars to finish with the list of attendees and pass it to them, which would potentially even decrease efficiency compared to having a single registrar.

These issues are directly analogous to the problems of lack of locality of reference in the single and naïve multithreaded approach to packet reassembly, and the need for threads to acquire a lock on the reassembly table before inserting a packet into in the naïve multithreaded approach.

Of course, what most ordinarily happens at registrations is that there is a person (or people) at the entrance to the registration hall, who asks for the registrant's name on entry, and directs them to a queue behind an appropriate desk. Rather than one large list of attendees, each registrar has a smaller list with names which lie in a certain alphabetical range. For example, one registrar might have a list of all surnames from A-C, another with a list of D-F and so on.
What this means is firstly, that rather than having all N registrars perform the same action, we pipeline the approach, and have one registrar performing the first stage of the pipeline, that is, directing people to the appropriate table, and N-1 performing the second stage, ticking names off their list as registrants reach the top of their queues. Secondly, this means that registrants are effectively being sorted into groups with alphabetically close surnames, and assigned to a registration desk associated with that group. This is called flow-pinning.

The problems of the initial approach when there are greater than one registrar are thus alleviated. As each registrar has their own, private section of the list of attendees, they do not need to pass one big one around, and do not have to wait for others to be finished with it. Also, since the list they have will be small, and registrants in their queue will have alphabetically close names, they will not have to search for entries in their list which are far away from the previous name they have ticked off.

We can apply the wisdom of this approach to conference attendee registration to packet processing very straightforwardly. An illustration of the process from [9] is shown below:

![Figure 4-3 Approach using Pipelining and Flow-Pinning](image-url)
A thread on one core is dedicated to capturing packets and directing them to a thread on an appropriate other core. This is done by means of a hash function. A simple (but certainly not optimal) example of such a function could be the packet’s destination address modulus the number of cores minus one. Each of the threads on the other cores behaves the same as the processor in the single threaded approach. They each have a smaller, private reassembly table which they do not need to share with other cores⁴.

With similar packets being directed to a thread on the same core, potential for locality of reference is greatly increased, and the cache invalidations which occur in the naïve multithreaded approach can be virtually eliminated if each thread’s private reassembly table does not share any cache lines with other threads’. As each thread’s reassembly table is local to that thread, it is not required that a lock need to be acquired and relinquished with each insertion into it⁴.

### 4.6 Fake Packet Processor

The fake packet processor designed for this project worked on fake packets with the following format:

<table>
<thead>
<tr>
<th>Source Address</th>
<th>Destination Address</th>
<th>ID Number</th>
<th>Fragment Number</th>
<th>Number of Fragments</th>
<th>Contents</th>
</tr>
</thead>
</table>

Therefore, an example packet would look like this:

A B 345 6 10 HELLO

This indicates a source address of A, a destination address of B, an ID of 345, that the packet is fragment number 6 of 10 of a larger packet and has the contents “HELLO”.

A program was written to generate packets, fragment them and write them to a file where they were separated by newlines. The order of the packets in these files was randomised using the unix “shuf” utility.

The packet processor read in a file containing packets line by line, parsing the lines into data structures representing packet fragments, and inserting them into a reassembly table. The data structure used for reassembly was initially a C++ STL map of ID numbers (integers) to arrays of packet fragments, but since finite numbers of packets were generated with IDs of 1 to N, this was replaced by an array of pointers to arrays of packet fragments, with a packet’s ID number serving as the array index, which gave faster lookup times over the STL map.

For the naïve multithreaded version, the file containing the packets was split into 4 files with an equal amount of packets in each file. 4 separate threads worked on a different file and inserted into the same reassembly table, acquiring a spinlock for each insertion.

For the pipelining and flow pinning version, one thread read in packets from a file, parsed them into fragment data structures, and inserted them into one of three queues, one for each

---

⁴ This may not always be the case and is further expanded on in Chapter 5 as an example of a lopsided access pattern
The queue used was a single-reader single-writer queue which requires little synchronization overhead. It is also used in a queue based biased locking algorithm and is described in section 7.4. A complex hash function was not used to divide packets between threads, but rather, packets were sent to different threads based on their ID number.

### 4.7 Results of Experiments

The following table shows the time taken for each version of the fake packet processing simulator to process 10,000,000 packets on beaker (see section 8.1 for specs).

<table>
<thead>
<tr>
<th></th>
<th>Time Taken</th>
<th>Compared to Single Threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Threaded</td>
<td>1m22s</td>
<td>-</td>
</tr>
<tr>
<td>Naïve Multithreaded</td>
<td>1m17s</td>
<td>~1x Faster (No speedup)</td>
</tr>
<tr>
<td>Pipelining &amp; Flow Pinning</td>
<td>27s</td>
<td>~3x Faster</td>
</tr>
</tbody>
</table>

Table 4-1 Results of Packet Processing Benchmark

The results clearly show pipelining and flow pinning to be by far the best solution, giving a 3x speedup over the basic single threaded version, with the naïve multithreaded version giving almost no speedup.

The experiments detailed in Verplanke’s article showed that a version of Snort using flow pinning exhibited a throughput of over 6.2x higher than the single threaded version of Snort when running with 250,000 TCP connections [9].
Chapter 5

Lopsided Access Patterns

5.1 Introduction

Thread safety incurs overhead. In a serial execution scenario, the running thread simply accesses and modifies data in memory as needed, with the only overhead being the time it takes the processor to access the data. In a concurrent scenario where locks are used, access to shared resources incurs the overhead of having to potentially wait for other threads to finish using the resources. The amount of contention for the resource, and the amount of time threads spend using the resource are the two most obvious and most impacting sources of overhead.

What must not be overlooked, however, is the overhead incurred from the locking mechanism itself.

Spinlocks and the MCS lock were discussed in Chapter 2. Spinlocks are simple and efficient for scenarios with low numbers of processors/hardware threads. However, in scenarios with higher numbers of competing hardware threads\(^5\), where the number of software threads does not exceed the number of hardware threads available, the performance of spinlocks begins to degrade, while the MCS lock performs significantly better [4]. This demonstrates how crucial lock choice can be in speed critical situations.

Other common scenarios, which are of particular interest in this project, are the very little contended and entirely uncontented cases.

In very little contended case, the vast majority of the time only one thread needs to access a resource, but crucially, not all the time, and thus it is necessary that it be protected by a lock. However, it is undesirable that the thread accessing the resource, most often without any other threads contending for it, should have to always incur the overhead of having to obtain a lock before accessing that resource. In the absence of contention, to ensure thread safety by conventional means, access to the resource will still be much slower than in a serial scenario, especially due to the fact that atomic operations and/or memory barriers are so often needed to ensure correctness. In the case of a TATAS spinlock protecting a resource,

---

\(^5\) By “higher numbers of competing hardware threads”, situations involving dozens or hundreds of hardware threads which are all very frequently accessing a critical section are meant. Basic TATAS spinlocks usually outperform MCS on situations with the small numbers (4-16) of hardware threads usually found on modern computers.
for example, the thread will need to check a lock variable, perform a compare-and-swap operation, and check if the operation succeeded before it can access the resource.

The entirely uncontented case presents the same issues as above, but the difference is no other thread needs access to a resource protected by a lock, and thus no other thread will ever try to acquire that lock. It might initially seem bizarre that such a scenario might arise, as logically, it seems that a lock would simply not be necessary in such an instance. However, this does indeed arise in certain situations, particularly in the design of thread safe libraries, which might be used in single threaded as well as multithreaded situations, and high level programming languages which give synchronization guarantees for access to data structures or objects. For example, this scenario arose in the development of the Java programming language, which is discussed later in this chapter.

A simple benchmark was implemented on beaker and zooey, (see sections 8.1 and 8.2 for specs) to demonstrate the difference in speed between accessing a variable in memory and writing to it and accessing a variable protected by a pthread spinlock without any contention, and similarly, writing to it. In each case, the variable was accessed 1,000,000,000 times and incremented each time (this is similar to a benchmark in [1]). No O flags were used. The case in which it was protected by a spinlock was over 7 times slower on beaker and 3 times slower on zooey, than the version without locks. While obviously incrementing a variable is a tiny task which maximizes the overhead of the spinlock, it is clear that such a slowdown is a cause for concern.

5.2 Common Situations

Lopsided access patterns are common in the design of thread safe libraries and in the design of languages with high support for thread-safety, such as Java, which was mentioned above.

They also occur in packet processing scenarios, in particular, in scenarios where a pipelining and flow-pinning approach, as was described in Chapter 4, is taken. This is discussed in further detail in section 5.4.

It is proposed that lopsided access patterns might also occur in real time stream reassembly applications, in which some kind of divide and conquer approach, or indeed, a flow pinning approach, is taken. In these applications, it might not be sufficient to split up work among different threads and merge it together at the end due to the demands of real time processing, and it could be optimal to allow threads occasionally access other threads’ section of the work and modify it on the fly.

5.3 Lock Reservation Work

Most work on biased locking up until now has been undertaken by Sun Microsystems in attempts to speed up the operation of synchronization primitives in the Java programming language [12] [13] [14] [15]. Generally, the focus of this work has been to build upon Bacon et al.’s Thin Locks for Java [16], which allow locking a thread safe object by the first accessor (which will often be the only accessor) to require only the overhead of a compare and swap operation. Should another thread access this object, the lock is converted to a “fat lock”, a
more heavyweight, but robust locking mechanism which allows access to all Java locking semantics.

This lock reservation work has sought to remove the necessity of this compare and swap operation by allowing a thread to reserve a lock, usually on the first access. Several approaches have been taken. The closest work to Vasuvedan et al. [1], which is of most interest for this project, is Onodera et al. [14], which proposes the KKO lock, a spinlock which allows a thread to reserve it and which tightly combines a Dekker-style lock with a CAS based one. The general scheme in [1] can be seen as a generalisation of the KKO lock in [14].

5.4 Relevance to Concurrent Packet Processing

In chapter 4, an approach to multicore packet processor was discussed, implemented and experimented with. It was shown that the idea of flow pinning, splitting the packet reassembly table into n-1 smaller tables private to n-1 threads, with one other thread dedicated to forwarding packets onto the appropriate worker thread mitigated the problems of a more naïve multithreading approach and effectively exploited multi-core hardware.

It should be noted that an advantage the single threaded and naïve multithreaded approaches have over the flow pinning approach, is that all threads have access to every entry in the reassembly table, whereas in the flow pinning approach, threads only have access to their private fraction of the reassembly table, and cannot access or modify packets outside their “group”. This is fine for basic reassembly, but it is reasonable to assume that in, for example, an advanced intrusion detection system, that a thread might occasionally have to update an entry in another thread’s private “group” at the pre-processor stage.

As such, each thread’s private reassembly table needs to be shared with other cores and made thread safe. By far the most frequent accessor of this table will be its owner, and yet, if protected by a lock, overhead is incurred on every access, even though the vast majority of the time there is no contention.

This kind of lopsided access pattern which arises in multicore packet processor software was one of the primary motivations behind Vasuvedan et al.’s 2010 paper, Simple and Fast Biased Locks. This paper is discussed in the next chapter, with further, novel, algorithms for reducing locking overhead in lopsided access scenario discussed in the one following that.
Chapter 6

Simple and Fast Biased Locks

6.1 Introduction

Simple and Fast Biased Locks [1] was a paper presented at the PACT conference in 2010. It outlines a simple, general scheme for an approach to biased locking, which divides lock accesses into two tiers – N threads competing for one lock, and 2 threads competing for the critical section – the holder of the N process lock (if there is one) and the dominant thread. Expanding on this, they outline 3 other locking algorithms based on this scheme – rebiasable locks, where the dominant thread can be changed on the fly, biased read-write locks, and asymmetric locks.

Their general biased locking scheme and their Asymmetric algorithm were of most interest for this project.

6.2 General Biased Locking Scheme

The general locking scheme, described as “Flexible, Fixed-Owner Biased Locks”, requires the selection of two locking algorithms – an N process algorithm and a 2 process algorithm. The basic idea is that an N process algorithm is likely to rely on expensive atomic operations such as compare and swap, whereas while a 2 process algorithm may not be as capable of scaling or dealing with contention well, it likely does not rely on the execution of such costly instructions in the uncontested case. This means that if a thread is very frequently accessing a resource with very little contention, then the overhead incurred by having to access it via a 2 process locking algorithm is likely to be less than the overhead from having to access it via an N process lock.

To exploit this fact in the case of one thread having much more accesses to a critical section than any other thread, the scheme defines this as the dominant thread and makes N non dominant threads compete for an N process lock. Holding this lock gives a thread the right to compete for a 2 process lock with the dominant thread. The holder of the 2 process lock can access the critical section. A pseudocode outline is given below (taken directly from [1]):

```c
typedef struct {
    ThreadId owner;
    Lock2 t; /* lightweight, 2-process lock */
```
Listing 6-1 Flexible, Fixed-Owner Biased Locking Scheme

As lockN and lock2 can call functions for any N or 2 process locking algorithms, it is easy to see why this is a flexible scheme. The most notable part of it, however, is the partitioning of the lock acquisition process into two tiers. This can be further exploited by the idea of “asymmetric” locking algorithms. The difference between symmetric and asymmetric locking algorithms and the Asymmetric algorithm described in [1] are explained in the next section.

6.3 Asymmetric Locks

In [1], a symmetric choice point is defined as follows:

A symmetric choice point in a mutual exclusion protocol is a state where two or more threads are waiting to enter a critical section and either thread can win the race by executing a sequence of its own actions.

They go on to explain that a “mutual exclusion protocol has the symmetric choice property if there is a reachable symmetric choice point”. This is true of most standard mutual exclusion algorithms, for example Dekker’s Algorithm, Peterson’s Algorithm and Spinlocks described in Chapter 2.

An asymmetric locking algorithm is one without the symmetric choice property. What this means in practical terms is that when two or more threads are waiting to enter the critical section, one thread can “decide” to enter the critical section or let another thread in. The
other threads cannot enter the critical section by executing a sequence of their own actions, but instead, rely on the actions of another thread to gain access to the critical section.

In [1], they also prove a theorem which states that all algorithms with the symmetric choice property require a “revealing operation”, i.e. a memory barrier, on a lock acquisition. One of the key factors in the speedup offered by using an asymmetric algorithm is the lack of the necessity of a revealing operation on acquisition by the dominant thread.

Vasuvedan et al.’s asymmetric algorithm follows the pattern of the flexible, fixed-owner biased locks in that there is an N-process lock that must be obtained by a non-dominant thread before it can compete for a 2-process lock. However, the 2-process lock in this instance cannot be any 2-process lock, but is a fixed, asymmetric lock.

The operation of the asymmetric lock is quite simple. It involves two Boolean variables shared between all threads – request and grant. When request is true, it indicates that a non-dominant thread wants access to the critical section. When grant is true, it indicates that the dominant thread has granted permission to a non-dominant thread to enter the critical section.

Before entering the critical section, the dominant thread checks the grant variable. If it is true, a non-dominant thread has been granted access to the critical section, and it spins until it has been set to false. On exiting the critical section, it checks the request variable. If true, a non-dominant thread has requested access to the critical section, and the dominant thread resets request to false and executes a memory fence to make all updates performed in the critical section visible to all threads. It then sets grant to true, allowing the non-dominant thread to proceed into the critical section. In the description of the algorithm in [1], a memory fence is executed directly after setting grant to true. It was confirmed by correspondence with the author, that this fence operation is not strictly necessary on most modern architectures. However, it may be wise to include it so that the waiting non-dominant thread can see the grant variable is set to true as soon as possible and proceed into the critical section.

When a non-dominant thread wishes to enter the critical section, it must first obtain the N-process lock. On obtaining this, it sets request to true and spins while grant is false. On exiting the critical section, it calls a memory fence to make any modifications in the critical section visible to the dominant thread, and sets grant to false. In the description of the algorithm in [1], no memory fence is placed after setting grant to false. It might be prudent to insert one at this point for the same reasons as one is placed after the dominant thread sets grant to true.

A pseudo code description of the algorithm is given below:

```c
struct lock
{
    bool request;
}
```

---

6 For this algorithm, and all algorithms in Chapter 6 and Chapter 7 from the Asymmetric algorithm onwards, pseudo code descriptions are not given in terms of biased_lock and biased_unlock functions as above, but rather, necessary locking overhead code is written inline inside dom_thread and other_thread functions, which respectively indicate work being done by the dominant and a non-dominant thread, with comments indicating where critical section code should be placed.
bool grant;
}

struct data
{
    lock l;
    int x;
    int y;
}

void dom_thread(data * d)
{
    ...

    while(d->l->grant) ;

    /* INSERT CRITICAL SECTION CODE HERE */

    if(d->l->request)
    {
        d->l->request = false;
        fence();
        d->l->grant = true;
        fence();
    }

    ...
}

void other_thread(data * d)
{
    ...

    lockN();
    d->l->request = true;
    while(!d->l->grant) ;

    /* INSERT CRITICAL SECTION CODE HERE */

    fence();
    d->l->grant = false;
    unlockN();

    ...

6.4 Variation of Vasuveden et al’s Asymmetric Locks

The locking overhead incurred by the dominant thread in the above algorithm, when no non-dominant thread has requested or has been granted access to the critical section, is the checking of two Boolean variables – grant before entry to the critical section, and request on exit.

It should be noted that as the non-dominant thread’s right to access the critical section is entirely at the “mercy” of the dominant thread, removal of the checking of both Boolean variables does not cause a violation of mutual exclusion, it simply causes starvation to any non-dominant thread which wishes to access the critical section.

In light of this, it can be said that the dominant thread is not acquiring or releasing a lock as such, but rather it polls for the request variable being set to true. Only when it sees request is true will it set grant to true. Therefore, only after having seen request is true and setting grant to true, will there be any potential for having to spin on the grant variable before the next critical section access.

If we make the assumption that if a thread is dominant, and biased locks are being employed to reduce locking overhead, that non-critical section code is insignificant, we can reduce the overhead of an uncontested critical section access from the checking of two Booleans to one by checking and spinning on the grant variable only when request has been seen to be true, as opposed to checking it before every critical section access. Pseudo code is shown below:

```c
struct lock
{
    bool request;
    bool grant;
}

struct data
{
    lock l;
    int x;
    int y;
}

void dom_thread(data * d)
{
    ...

    /* INSERT CRITICAL SECTION CODE HERE */
```
if(d->l->request)
{
    d->l->request = false;
    fence();
    d->l->grant = true;
    fence();

    while(d->l->grant) ;
}

...

void other_thread(data * d)
{
    ...

    lockN();
    d->l->request = true;
    while(!d->l->grant) ;

    /* INSERT CRITICAL SECTION CODE HERE */

    fence();
    d->l->grant = false;
    unlockN();

    ...
}

Listing 6-3 Variation 1 of Vasuvedan et al.'s Asymmetric Locking Algorithm

Although it was stated above that assumptions about the significance of non-critical section code are required for this variation, these assumptions are in fact not necessary. By duplicating any non-critical section code which might be executed and using a goto, it is possible to replicate the behaviour of the original asymmetric algorithm, with the dominant thread only needing to check one Boolean variable with each critical section access\(^7\).

void dom_thread(data * d)
{
    ...

\(^7\) Polling after every critical section access is a sensible frequency to poll at, but it is by no means strictly necessary that polling occur at this time and polling can be adjusted to be more or less frequent depending on the architecture and dominance level involved.
Listing 6-4 Variation 2 of Vasuvedan et al.'s Asymmetric Locking Algorithm

6.5 Cache Considerations

A subtle optimisation to Vasuvedan et al's Asymmetric lock, which is not feasible on current hardware, is provided below, as it is felt it is interesting:

```c
void dom_thread(data * d)
{

  ...)

  while(VALID_CACHE_LINE(&grant)) ;

  /* INSERT CRITICAL SECTION CODE HERE */

  if(INVALID_CACHE_LINE(&request))
  {
    REVALIDATE_CACHE_LINE(&request);
    fence();
```
Listing 6-5 Vasudevan et al.’s Asymmetric Locking Algorithm Using Cache Line States

VALID_CACHE_LINE() and INVALID_CACHE_LINE() are theoretical functions that might exist in future hardware that would allow a program to check the state of the cache line an address passed to it resides in without having to actually read the value itself. REVALIDATE_CACHE_LINE() is a theoretical function that would allow a program to change the state of the cache line of the address passed to it to a valid state.

The initial state for the algorithm to work is as follows. The grant variable and the request variables must each be in their own cache lines, and must be separate from any other data. Initially grant is set to true and request to false by the dominant thread, and thus the cache lines containing them are in the L1 cache of the core it is running on. The line containing grant must then be invalidated, while the line containing request must be valid.

In the ordinary version of the algorithm, grant is checked before entry to the critical section and request is checked on exit. It is almost certain that when uncontended, these variables will always be in the dominant core’s L1 cache. In the event of another thread setting the request variable, it will invalidate the cache line containing request, and will cause a cache miss and a resulting penalty for the dominant thread. If the dominant thread is spinning on the grant variable, waiting for a non-dominant thread to exit the critical section, when it does, modifying the value of grant will cause a cache miss for the dominant thread.

Thus, if grant and request are in independent cache lines, separate from any other data, the only time an invalidation will occur on the dominant core’s L1 cache. In the event of another thread setting the request variable, it will invalidate the cache line containing request, and will cause a cache miss and a resulting penalty for the dominant thread. If the dominant thread is spinning on the grant variable, waiting for a non-dominant thread to exit the critical section, when it does, modifying the value of grant will cause a cache miss for the dominant thread.

Therefore, in the standard, uncontested case, grant’s cache line will be invalid, so the while is passed over before the critical section, and request’s cache line is valid and thus the if is passed over after the critical section. When a non-dominant thread modifies request, its line becomes invalidated, and thus the dominant thread enters the if statement without any cache miss penalty incurred. Inside the if, both grant and requests’ lines are revalidated. Thus, when the dominant thread next encounters the while loop before the critical section, it spins while grant’s cache line is valid. When the non-dominant thread exits the section and modifies grant, its line is invalidated and the dominant thread proceeds into the section, again, without any cache miss penalty incurred.

Of course, this is all in the realms of fantasy at present, because current architectures do not expose cache line states, nor do they allow explicit control over cache line states to the programmer in this manner. Furthermore, the performance gains which might be gained from this are likely to be small for this algorithm, especially at high dominances, as cache
misses to grant and request only occur when a non-dominant thread accesses the critical section, a rare event. However, a theoretical inter-core communication mechanism using cache coherency lines has been outlined and demonstrated to theoretically work and potentially somewhat increase performance in this algorithm, and it is believed that a strong case could be made for including programmatic control over cache lines to potentially allow finely-tuned, efficient inter-processor synchronization in future architectures.
Chapter 7

Combining Message Passing with Biased Locks

The asymmetric algorithm presented in Vasuvedan et al. is fast and simple. The only overhead to the dominant thread required when uncontended is the checking of two Boolean variables, at the start and end of the critical section. When contended, the overhead is obviously greater. Aside from the obvious requirement of the dominant thread to spin while the non-dominant thread performs work in the critical section, memory barriers are required, and more critically, when the dominant thread resumes, there will be cache misses if the non-dominant thread has modified the shared variables.

In light of the above, another approach was considered. It was proposed that in the case of the dominant thread spinning, waiting to re-enter the critical section, which will be highly likely, as the dominant thread will naturally have a high frequency of accessing the critical section, that rather than allowing the non-dominant thread, running on another core, to access the section, that it would instead “push” work to the dominant thread, and spin itself. This would potentially eliminate the cache misses on the dominant thread regaining access to the critical section, and, as will be seen in the description below, only one variable needs to be polled by the dominant thread, with no “grant” variable needed.

7.1 Initial Algorithm

The first method devised to push work to the dominant thread was to use a function pointer. The scheme is simple and initially similar to Vasuvedan et al’s algorithm. As in Vasuvedan, the N non-dominant processes compete for an N process lock. At this point, however, rather than competing for a 2 process lock with the dominant thread, or setting a request variable, it sets a function pointer variable to contain the address of a function which contains the work the non-dominant thread wants performed in the critical section. The non-dominant thread then spins on a “done” variable. The dominant thread polls the function pointer at a frequency which should be tuned by the requirements of the program. If it is null, it simply continues on, if it contains the address of a function, that function is executed. When the function has been executed, the dominant thread resets the function pointer to null, executes a memory barrier to make all changes visible to other threads, and sets the “done” variable to true. The non-dominant thread then stops spinning, and releases the N process lock.
A pseudocode example of the algorithm in action is provided below:

```
struct lock
{
    bool done;
    void (*func)(data* d);
}

struct data
{
    lock l;
    int x;
    int y;
}

void dom_thread(data * d)
{
    ...
    /* INSERT CRITICAL SECTION CODE HERE */

    if(d->l->func != NULL)
    {
        d->l->func(d); //Execute Function
        d->l->func = NULL;
        fence();
        d->l->done = 1;
    }
}
void somefunc(data * d)
{
    /* INSERT CRITICAL SECTION CODE HERE */
}

void other_thread(data * d)
{
    ...
    lockN();
    d->l->func = &somefunc;
    while(!d->l->done);
    unlockN();
    ...
}
```

Listing 7-1 Function Pointer Passing Algorithm
7.2 Asynchronous Version

In the first algorithm in this section, the non-dominant thread must spin while its work is executed by the dominant thread. It was noted that in the event of no variables local to the non-dominant thread in question being modified in the critical section, that the non-dominant thread could continue on rather than waiting. There are some issues when attempting to apply this to the first algorithm, however.

The first issue which arises is how the non-dominant thread, which has just acquired the N process lock and placed a function address in the function pointer variable, knows when to release the lock. If it releases it immediately, then another thread is free to acquire the lock and overwrite the function pointer, very possibly before the dominant thread has executed it. A possible solution would be to make threads spin on the “done” variable after acquiring the N process lock, but before setting the function pointer. This means that while the dominant thread is executing a function “owned” by a non-dominant thread, the thread which owns the function can continue on executing, but any other thread which obtains the N process lock and wishes to pass a function to the dominant thread must wait until the currently executing function has completed before modifying the function pointer variable.

While the highlights the general approach, it can be optimised further. The done variable is unnecessary in reality, as whenever it is true, the function pointer variable is always null, and when it is false, the function pointer variable contains something. Therefore, it can be eliminated, and non-dominant threads can spin on the function pointer being non-null. Furthermore, if the function called immediately sets the function pointer variable to null, a non-dominant thread can place a pointer to its function in the variable as soon as the previous non-dominant owned function has been dispatched by the dominant thread, rather than when it has completed.

An example of the algorithm is provided below:

```c
struct lock
{
    void (*func)(data* d);
}

struct data
{
    lock l;
    int x;       //x and y are simply examples of shared data
    int y;
}

void dom_thread(data * d)
{
    ...
    /* INSERT CRITICAL SECTION CODE HERE */

    if(d->l->func != NULL)
```

46
Until now, the algorithms described has sought to improve the efficiency of Vasuvedan et al’s general scheme by modifying what how the holder of an N process lock interacts with the dominant thread in order to get its work completed. How the N process lock works, however, has not been considered. In Vasuvedan’s algorithms, as well as those described in this chapter so far, it can be any N process lock, and it is entirely decoupled from the interactions between its holder and the dominant thread. A simple and efficient lock for this purpose, and the one used in the experiments in this project, is a basic spinlock. Spinlocks are described in Chapter 2. The pseudocode given in that chapter is given below:

```c
void lockN (int * lck)
{
    int success = 0;
    do
    {
```
Listing 7-3 Basic Spinlock as N-Process Lock

Now let us look at a snippet of code from the Asynchronous Function Pointer algorithm from the previous section. This is the code for a non-dominant thread acquiring an N process lock, waiting for the function pointer to be null (i.e. the function having been dispatched by the dominant thread), setting the function pointer to contain its function and releasing the N process lock:

```c
void unlockN (int * lck)
{
    *lck = 0;
}

lockN();
while(d->l->func != NULL);
  d->l->func = &somefunc;
unlockN();
```

Consider what occurs when the N process lock is a spinlock. First a thread spins on the lck variable being non zero. The lck variable being 1 indicates that another thread holds the lock. When the thread holding the lock releases it, it sets lck to zero. At this point, all threads spinning on the lck variable proceed and execute a compare and swap on lck. One of these threads will successfully change lck to 1 and can proceed, as this means it now holds the lock, the rest go back to spinning. The proceeding thread now must spin on the function pointer being non-null. Once it is null, it proceeds to set it to its function and sets lck to 0, relinquishing the lock.

It was noticed that the lock holder spinning on the function pointer being non-null before proceeding and modifying that same function pointer bears some resemblance to the spinlock mechanism (the only difference is since there is only one thread spinning, waiting to modify the variable, an atomic compare and swap operation is not required). It was then noticed that the lck variable is superfluous, spinning can simply be done by all threads on the function pointer variable being non-null before attempting to write their own function pointer to it via a compare and swap:

```c
void pushWork (fp * func, fp work)  // fp - function pointer
{
    int success = 0;
    do
    {
        while (*fp != NULL) ; //spin
            success = compare_and_swap(fp, 0, work);
    } while(!success);
```
Listing 7-4 pushWork Function

Crucially, this eliminates the need for a call to an unlocking function, as the function pointer is already set to null by the dominant thread after dispatching the function pointed to by that function pointer. In light of the existence of the above pushWork function, we can now replace the other_thread function from the Asynchronous example with this:

```c
void other_thread(data * d)
{
    ...
    pushWork(d->1->func, &somefunc);
    ...
}
```

Listing 7-5 Non-Dominant Thread Code for Integrated Spinlock Algorithm

### 7.4 A Thread-Safe Single-Reader Single-Writer Queue

In order to describe the next variation of the algorithm, it is necessary to outline the operation of a thread-safe single-reader single-writer queue, which requires no locks, atomic operations or any explicit synchronisation mechanism besides two memory fences to ensure immediate visibility of certain variables.

The idea of using such a queue was prompted by it being mentioned by Verplanke [9]. Such a queue was described by the user “vanDooren” on the Microsoft MVP website [17] and the implementation was largely taken from it.

The key components of this queue are a fixed size ring buffer, a write pointer and a read pointer.

The write pointer always points to the element in the ring buffer after the last element written to (i.e. pushed to the back of the queue) by the writer. The read pointer always points to the element in the ring buffer after the last element read (i.e. popped off the front of the queue) by the reader.

As outlined by vanDooren [17], there are 3 possible cases:

- The read pointer is equal to the write pointer. This indicates that the queue is empty. There is nothing to read, but the queue can be written to.
- The read pointer points to the element after the element pointed to by the write pointer. This indicates that the queue is full. The queue cannot be written to, but can be read from
- Neither of the above cases is true. The queue contains elements but is not full. The queue can be read from or written to.

Before a read, the reader checks the first above condition. If true, the read fails. Otherwise, it reads the value pointed to by the read pointer and increments the pointer.
Before a write, the writer checks the second above condition. If true, the write fails. Otherwise, it writes to the location pointed to by the write pointer and increments the pointer.

The reader thread can only ever write to the read pointer and the writer thread can only write to the write pointer. Crucially, both threads only ever write to the respective read or write pointers after they have read from or written to the queue. This means that the read and write pointers will always reflect the state of the queue from the point of view of the respective reader or writer threads, and as such, there is no potential for the writer to overwrite a value before a reader has read it.

Writes to the reader and writer pointers must immediately be seen by the other thread, however. Memory fences are not included in vanDooren’s code listings [17]. As part of the work for this project, they were added for correctness in the implementation used for the following algorithms.

Code for this queue is given in Appendix A.

### 7.5 Algorithm Using Queues

The queue used is obviously the queue described in the previous section. The N non-dominant threads still compete for the N process lock, but rather than acquiring the right to write to the function pointer and the done variable, they become the only thread allowed to write to a queue of function pointers. It simply pushes a function pointer onto the queue and moves on. The dominant thread operates the same way as in the earlier algorithm, but rather than polling a function pointer, checking if it is null, it checks if the queue is empty. If it is not, it continually pops every function pointer off the queue and executes it until the queue is empty.

Pseudo code for the algorithm:

```c
struct lock
{
    Queue q; //Queue which holds function pointers
}

struct data
{
    lock l;
    int x;
    int y;
}

void dom_thread(data * d)
{
    void (*func)(data* d);
    ...
```
A downside of this approach is that checking if the queue is empty requires checking the equality of two integers, which involves more overhead than checking if a pointer is null.

### 7.6 Algorithm Using Messages

It was noticed after implementing the above algorithms, that this approach bears resemblance to message passing concurrency. Message passing is a method of concurrent programming whereby memory is not shared, but instead individual threads (or objects in OO message passing) have exclusive access to their own private memory. Other threads can manipulate the data in this memory by sending messages to the owner. Generally, a message will consist of some parameters and some value which indicates to the owner thread that it should invoke some handler.

Message passing is largely used in distributed systems, where logistically, there can be no shared memory. However, it is also used in parallel systems, where it most commonly manifests itself as the Actor Model, a concurrency model based on the exchange of messages between actors. An actor is a computational unit which is capable of sending messages and
reacting to received messages. Erlang is an example of a programming language which explicitly supports concurrency based on the Actor Model [18].

It is quite clear that in the algorithms described above, at a high level, there are two conceptual “actors” involved. The first actor is the dominant thread, while the other actor is the set of N non-dominant threads. The set of non-dominant threads send messages to the dominant thread. The dominant thread receives these messages and handles them. As the messages involved are pointers to functions, the handling is very simple – the dominant thread executes the function.

Function pointers are not messages in the traditional sense, however, as messages usually prompt the receiver to execute some predefined handler, as opposed to simply executing the function at the address passed to it.

Obviously, in one sense, function pointers are flexible and dynamic, allowing the non-dominant threads to push any work they please to the dominant thread. On the other hand, it can be argued that this is not a particularly safe way of doing things. There are also more pressing disadvantages to this approach. Function pointer calls in this context can obviously not be inlined, as the contents of the function pointer will change non-deterministically throughout the execution of the program. As a result of this, executing work passed to the dominant thread incurs the overhead of a function call, i.e. creation of a stack frame, saving registers, pushing the return address on to the stack etc. Another disadvantage is that it is unlikely that the dominant thread’s CPU’s instruction cache will contain any of the functions’ code, as they can be anywhere in memory.

In light of these issues, the idea of potentially adapting the algorithm to use a more traditional method of message passing was conceived. With this approach, instead of passing function pointers to the dominant thread, tokens would be passed (for example, integer values). The dominant thread would have a set of inlined, predefined handlers, one of which would be executed depending on the value of the current token received. In theory, this would mean the overhead involved in calling a function would be eliminated, and also, the code for the handlers would be much more likely to be in the CPU’s instruction cache.

This can be applied to all three variations of the algorithm above. Pseudocode for it applied to the queue example is shown below:

```c
struct lock
{
    Queue q; //Queue which holds integers
}

struct data
{
    lock l;
    int x;
    int y;
}
```
void dom_thread(data * d)
{
    int token;
    ...*/ INSERT CRITICAL SECTION CODE HERE */

    while(!d->l->q->empty())
    {
        token = d->l->q.pop();
        switch (token)
        {
            case 1:
                /* INSERT CRITICAL SECTION CODE HERE */
                break;
            case 2:
                /* INSERT CRITICAL SECTION CODE HERE */
                break;
            case 3:
                /* INSERT CRITICAL SECTION CODE HERE */
                break;
            ...
            }
        fence();
    }
}

void other_thread(data * d)
{
    .../* INSERT CRITICAL SECTION CODE HERE */
    lockN();
    while(d->l->q.full()); // spin while queue is full
    d->l->q.push(2); // the value 2 is an example token
    unlockN();
}

Listing 7-7 Queue-Based Message Passing Algorithm
Chapter 8

Experiments/Results

All benchmarks were compiled with g++ 4.3.x. No optimisations were used to prevent compiler reordering of reads and writes, and to prevent the folding of loops into simpler computations, as explicit execution of iterations was desired to measure locking performance. For more complex benchmarks, it is acknowledged that optimizations might be desirable, in which case, use of the “volatile” keyword would be necessary for correctness.

On beaker, zooey and shawbox, one dominant and 3 non-dominant threads were used, with one thread per hardware thread. On cube, one dominant and 7 non-dominant threads were used. The taskset utility was used to ensure that two SMT threads on each of 4 cores, two per chip, were used. On the IBM Wirespeed, 32 threads were used.

A pthread spinlock was used as the N-process lock in all applicable cases, and for the unbiased spinlock which served as the baseline comparison except for on shawbox, where this was not available (due to OSX), and in this case, an original implementation of a TATUS spinlock was used.

The benchmarks run all involved a light task in the critical section by both dominant and non-dominant threads, the incrementing of a single variable. To investigate differences in performance affected by the length of time spent in the critical section, a superfluous for loop was added before incrementing, with different levels of delay experimented with.

Where bars for an algorithm are absent for a particular dominance level in a benchmark, or where algorithms do not appear at all in a graph, this indicates that the “Tipping Point” was hit, and thus results for this algorithm in this benchmark at that dominance or those dominances were not included. This is discussed in more detail in section 9.2.

Some graphs contain a “Control”, which is simply one thread incrementing a variable the same amount of times as the others at that dominance, and thus indicates the maximum possible speedup that could be achieved were all locking overhead eliminated.

8.1 Beaker

Beaker contains 2 Intel(R) Xeon(R) 5138 @ 2.13GHz processors, which contain two cores each.
Each core has private, 8-way associative L1 data and instruction caches with a cache line size of 64B and a total size of 32KB.

Both cores on each chip share a 16-way associative unified L2 cache with a cache line size of 128B and a total size of 4096KB.

Cores on separate chips do not share any cache.

### 8.1.1 Single Variable Incrementing

![Graph](image)

**Figure 8-1** Single Variable Incrementing on beaker (99.999 – 90% Dominance)

![Graph](image)

**Figure 8-2** Single Variable Incrementing on beaker (85 -70% Dominance)
The above figures show the speedups offered by the various algorithms with one dominant and 3 non dominant threads running on beaker, all incrementing the same variable when they enter the critical section.

### 8.1.2 Single Variable Incrementing with Medium Added Delay

![Graph showing speedups of different algorithms](image)

**Figure 8-3 Single Variable Incrementing with Medium Delay on beaker (99.999-90% Dominance)**
Figure 8-4 Single Variable Incrementing with Medium Delay on beaker (85-70% Dominance)

The above figures show the speedups offered by the various algorithms with one dominant and 3 non dominant threads running on beaker, all incrementing the same variable when they enter the critical section. In addition, a ‘for (int i = 0; i < 10; i++);’ is executed in the critical section before incrementing the variable.
8.1.3 Single Variable Incrementing with Long Added Delay

The above figure shows the speedups offered by the various algorithms with one dominant and 3 non dominant threads running on beaker, all incrementing the same variable when they enter the critical section. In addition, a ‘for (int i = 0; i < 100; i++);’ is executed in the critical section before incrementing the variable.

Figure 8.5 Single Variable Incrementing with Long Added Delay on beaker (99.999 - 90% Dominance)
8.2 Zooey

Zooey has a dual core Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz, with 2-way SMT per core.

Both cores have a private, 8-way associative L1 data cache with a cache line size of 64B and a total size of 32KB, and a 4-way associative L1 instruction cache with a cache line size of 128B and a total size of 32KB. These are shared between both hardware threads which run on each core due to SMT.

Both cores have a private, 8-way associative L2 unified cache with a cache line size of 64B and a total size of 256KB. This is shared between both hardware threads which run on each core due to SMT.

A 16-way associative L3 unified cache with a cache line size of 64B and a total size of 4096KB is shared by both cores, and thus by all hardware threads.

8.2.1 Single Variable Incrementing

Figure 8-6 Single Variable Incrementing on zooey (99.999 - 90% Dominance)
Figure 8-7 Single Variable Incrementing on zooey (85 - 70% Dominance)

The above figures show the speedups offered by the various algorithms with one dominant and 3 non-dominant threads running on zooey, all incrementing the same variable when they enter the critical section.
8.2.2 Single Variable Incrementing with Medium Added Delay

Figure 8-8 Single Variable Incrementing with Medium Delay on zoeoy (99.999 -90% Dominance)
The above figures show the speedups offered by the various algorithms with one dominant and 3 non dominant threads running on zooey, all incrementing the same variable when they enter the critical section. In addition, a ‘for (int i = 0; i < 10; i++)’ is executed in the critical section before incrementing the variable.
8.2.3 Single Variable Incrementing with Long Added Delay

The above figure shows the speedups offered by the various algorithms with one dominant and 3 non dominant threads running on zooey, all incrementing the same variable when they enter the critical section. In addition, a ‘for (int i = 0; i < 100; i++);’ is executed in the critical section before incrementing the variable.

Figure 8-10 Variable Incrementing with Long Delay on zooey (99.999-90% Dominance)
8.3 Shawbox

Shawbox is a Mac Pro with 2 Intel Xeon 5150 "Woodcrest" Dual-Core processors @ 2.66 GHz.

Each core has private, 8-way associative L1 data and instruction caches with a cache line size of 64B and a total size of 32KB.

Both cores on each chip share a 16-way associative unified L2 cache with a total size of 4096KB. Cache line size could not be determined.

Cores on separate chips do not share any cache.

8.3.1 Single Variable Incrementing

The above figure shows the speedups offered by the various algorithms with one dominant and 3 non dominant threads running on shawbox, all incrementing the same variable when they enter the critical section.

Figure 8-11 Single Variable Incrementing on shawbox (99.999 -90% Dominance)
8.3.2 Single Variable Incrementing with Medium Added Delay

The above figure shows the speedups offered by the various algorithms with one dominant and 3 non dominant threads running on shawbox, all incrementing the same variable when they enter the critical section. In addition, a ‘for (int i = 0; i < 10; i++)’ is executed in the critical section before incrementing the variable.

Figure 8-12 Single Variable Incrementing with Medium Delay on shawbox (99.999 -90% Dominance)
8.4 Cube

Cube has two quad core Intel(R) Xeon(R) E5520 @ 2.27GHz processors, with 2-way SMT per core.

All cores have a private, 8-way associative L1 data cache with a cache line size of 64B and a total size of 32KB, and a 4-way associative L1 instruction cache with a cache line size of 128B and a total size of 32KB. These are shared between both hardware threads which run on each core due to SMT.

All cores have a private, 8-way associative L2 unified cache with a cache line size of 64B and a total size of 256KB. This is shared between both hardware threads which run on each core due to SMT.

A 16-way associative L3 unified cache with a cache line size of 64B and a total size of 8192KB is shared by both cores, and thus by all hardware threads per chip.

No cache is shared between cores on separate chips.

8.4.1 Single Variable Incrementing

Figure 8-13 Single Variable Incrementing on cube (99.999-90% Dominance)
Figure 8-14 Single Variable Incrementing on cube (85-70% Dominance)

The above figure shows the speedups offered by the various algorithms with one dominant and 7 non dominant threads running on cube, all incrementing the same variable when they enter the critical section.
8.4.2 Single Variable Incrementing with Medium Added Delay

Figure 8-15 Single Variable Incrementing on cube with Medium Delay (99.999-90% Dominance)
The above figure shows the speedups offered by the various algorithms with one dominant and 7 non dominant threads running on beaker, all incrementing the same variable when they enter the critical section. In addition, a ‘for (int i = 0; i < 10; i++);’ is executed in the critical section before incrementing the variable.

Figure 8-16 Single Variable Incrementing on cube with Medium Delay (85-70% Dominance)
8.4.3 Single Variable Incrementing with Long Added Delay

The above figure shows the speedups offered by the various algorithms with one dominant and 7 non dominant threads running on cube, all incrementing the same variable when they enter the critical section. In addition, a ‘for (int i = 0; i < 100; i++);’ is executed in the critical section before incrementing the variable.
8.5 IBM Wirespeed

A limited amount of time was gotten to run benchmarks on the new IBM Wirespeed processor, which has 16 cores with 4-way SMT per core. In this time, the speedups over an unbiased pthread spinlock offered by asymmetric algorithms and single function pointer algorithms were compared between 99.999 and 95% dominance levels. 32 threads were used. Time did not allow the testing of either queue based algorithm or any other dominance level. These results should be considered preliminary.

8.5.1 Single Variable Incrementing

The above figure shows the speedups offered by the various algorithms with one dominant and 31 non dominant threads running on the IBM Wirespeed, all incrementing the same variable when they enter the critical section.
Chapter 9

Conclusions

9.1 Maximum Theoretical Speedup

The maximum theoretical speedup for any concurrent, shared memory application using locks, is the overall running time of the program minus the time spent acquiring and releasing locks. This is not possible while we are constrained to using a nondeterministic parallelism model (work has been done investigating the possibilities of deterministic parallelism [19] [20]). However, it is very useful to consider this maximum theoretical speedup when investigating the worthwhileness of minimising lock overhead in a program.

Firstly, the composition of the program in terms of critical section accesses and non-critical section work must be considered. The only parts of the code which will be sped up by reducing locking overhead are critical section accesses, and thus they must make up a significant proportion of the program for a worthwhile overall speedup.

Note that the term “critical section access” is used here to refer to work done in the critical section as well as locking overhead required.

Biased locking is useful when both the percentage of critical section accesses in a program, and the result of the following formula are significant. This is the maximum theoretical speedup to critical section accesses that can be gained:

\[
\frac{CS + LO}{CS + W}
\]

Formula 9.1 Maximum Theoretical Speedup to Critical Section Accesses

Where:

- CS – Average time spent in critical section by dominant thread
- W – Average time waiting for lock held by non-dominant threads
- LO (Locking Overhead) – Average time dominant thread spends acquiring and releasing lock (or polling for requests in asymmetric algorithms).

For example, if the dominant thread spends on average 100ns in the critical section, locking overhead is 900ns and the average time spent waiting for non-dominant threads to exist the
critical section is 100ns, then the formula above indicates that were it possible to completely remove locking overhead, a 5x speedup to critical section accesses would be possible.

On the other hand, were the dominant thread to spend an average of 2000ns in the critical section, with the other variables identical to above, the maximum theoretical speedup of critical section accesses would only be $\sim 1.4x$, meaning any practically obtainable speedup from reducing lock overhead could not exceed 1.4x (and in reality, any speedup that could be obtained would probably be quite a bit less).

The numbers in the above examples do not relate to anything in practical terms, and are purely illustrative. However, comparing the results in Chapter 8 for different levels of delay in the critical section, i.e. different relative critical section lengths, it is clear to see that the speedups gained from biased locking are greatly reduced as the delay increases. For example, in section 8.1.1, where there was no added delay in the critical section, speedups of over 6x over an unbiased spinlock were recorded, whereas in sections 8.1.2 and 8.1.3, greater added delays in the critical section dramatically reduce the speedups gained by biased locking algorithms.

Furthermore, a bar representing “Control” was added to the graphs for medium and long critical section delays. This represents a single thread incrementing a variable the same amount of times as the other threads in its cluster, and thus is the theoretical maximum speedup for that benchmark. Notably, the control is very close to the unbiased spinlock speeds when critical section length is longer, especially at very high dominances, and there is not much room for the biased locking algorithms to improve on this. This clearly demonstrates the importance of considering the relative length of a critical section access versus locking overhead when evaluating the worthwhileness of using a biased locking algorithm.

A diagram illustrating the limitations of reducing locking overhead when there exists a relatively much longer critical section is shown in Figure 9-1 and Figure 9-2.

Information on how much time a program/specific thread in a program spends in locking functions and in critical sections is readily obtainable by means of profiling tools such as gcov and gprof.

This formula bears some similarity to Amdahl’s Law, and can be thought of as a law in itself. Simply put, whereas Amdahl’s Law states that the maximum speedup from parallelization is bounded by the amount of code that can be executed in parallel in a program, this states that the maximum speedup from reducing locking overhead is bounded by the amount of locking overhead in the program.
Figure 9-1 Speedup Gained by Halving Locking Overhead with a Small Critical Section

Figure 9-2 Speedup Gained by Halving Locking Overhead with a Large Critical Section
9.2 “Tipping Point”

Up until now, the term “dominant thread” has been defined as the thread which has much more accesses to the critical section than any other thread, and it has followed that the dominance percentage is the dominant thread’s percentage of accesses to the critical section.

What have not been discussed are the practical implications of maintaining this dominance percentage when employing biased locking techniques. If we increase locking overhead to the non-dominant threads, can they maintain an access frequency consistent with their theoretical access percentage? If we decrease locking overhead to the dominant thread, could the speed of its critical section accesses increase its effective dominance to a higher percentage than its theoretical level of dominance? In an asymmetric locking scenario, where polling by the dominant thread is required to grant access to the critical section to other threads, can the rate of polling versus the rate of requests by non-dominant threads be consistent with theoretical dominance levels?

The answers to these questions are very much dependant on a number of factors such as the algorithm used, the speed of the N process lock, the architecture involved etc.

Consider the following. We set up a benchmark using an asymmetric locking algorithm in which a dominant thread accesses a critical section 900 times, and a number of other, non-dominant threads access it 100 times. The theoretical dominance percentage is, therefore, 90%. We set the dominant thread to poll for access requests from non-dominant threads after each access of its own. Let \( t \) denote the amount of time it takes the dominant thread to perform a critical section access.

Suppose due to locking overhead, non-dominant threads are only able to make a request to the dominant thread every 10\( t \). This means that for every 10 times the dominant thread polls for non-dominant threads’ requests, it only sees one once. If it is to access the critical section 900 times, this means that it will only see requests from other threads 90 times before it finishes, leaving the non-dominant thread incomplete and starved of access to the critical section (as 90 < 100).

We call the point at which a scenario like this occurs the “Tipping Point”. It can be expressed mathematically as below:

\[
\frac{f_r}{f_r + f_p} < (1 - \text{Dom } \%)
\]

Formula 9-2 Tipping Point

Where:

- \( f_r \) is the frequency of requests by non-dominant threads
- \( f_p \) is the frequency of polling by dominant threads
- \( 1 - \text{Dom } \% \) is the theoretical percentage of critical section accesses by non-dominant threads
Simply put, if the ratio of request frequency to polling frequency is less than the theoretical percentage of critical section accesses by non-dominant threads, the dominant thread will finish before at least one of the non-dominant threads, leaving those threads starved.

This is alluded to briefly in [1], where they make the statement, in relation to their asymmetric locking scheme, that “The protocol as a whole is free from starvation provided the dominant thread checks for such requests infinitely often.”

As such, the tipping point problem can be alleviated by placing a loop at the end of the dominant thread’s execution which repeatedly polls for requests until all other threads have completed. This was considered to be unrealistic and including results from benchmarks using this technique was decided against for this project.

It is quite a bizarre situation, as the general focus of biased locking is to increase the speed of the dominant thread. It is rare that something going too fast would be an issue in optimization, but in this case, the dominant thread executing too fast for the non-dominant threads to keep up and maintain the theoretical dominance percentage is indeed an issue.

Of course, most situations where biased locking would be used, such as packet processing, are situations where threads run indefinitely, meaning that hitting the “Tipping Point” would not cause absolute starvation. Depending on the situation, it might have minimal impact or it could severely hinder the progress of non-dominant threads.

As will be explained in the next section, the algorithms most vulnerable to the tipping point are the synchronous function pointer/message passing algorithms, while the most robust against it are the algorithms involving queues. The “Tipping Point” tended to only be an issue at 95% and lower dominance percentages.

### 9.3 General Overview of Benchmark Results

The results of experiments run gave a variety of interesting results. In certain areas confirming expectations, and in others exhibiting interesting properties such as the “Tipping Point” and an algorithms’ susceptibility to it at differing dominance levels and critical section lengths.

Results varied hugely between architectures. The potential speedup for very high dominance was much higher (up to 6x) on beaker and shawbox than zooey or cube (up to 2.5x). It is suspected that this is partially a result of zooey and cube having SMT processors and possessing L3 caches, which mean they have a much greater level of cache sharing between hardware threads (all hardware threads on the same core naturally share L1 and L2 caches, while all cores on the same chip share an L3 cache). This means the modification of a lock variable is much less likely to require a main memory access as a result of a cache line invalidation. The speed of an atomic CAS operation relative to a write is also much faster (about 2x) on zooey and cube than on beaker and shawbox. Both of these factors potentially much lessen the overhead of a spinlock acquisition, which in turn, lessens the potential for speedup via other locking methods.

As is discussed in section 9.1, critical section lengths were hugely impacting on the potential speedups offered by biased locking. A notable finding was the degradation of speedups with
increasing critical section lengths. Interestingly, the level of degradation was much lower in zooey and cube (at very high dominance, ~2.5x to ~1.2x) than beaker and shawbox (at very high dominance, >6x to >~1.2x). As would be expected as a result of what was discussed in section 9.1, the longer the critical sections got, the less of an impact biased locks had.

Decreasing dominance levels, as expected, had a negative impact on the speedups offered by biasing locks. It was found that while great speedups can be obtained at very high dominances (99.999% - 99.9%) marginal reductions in dominance from these levels can have huge impacts on the speedups offered by biased locking. In the graphs in sections 8.1.1, 8.2.1 and 8.3.1, we see a marked difference in the speedups offered at between 99.9% and 95% dominance, for example (the exact rate of decreasing of speedups with decreasing dominance was very architecture dependant). In contrast, there is not much difference in the speedup offered by algorithms which have not hit the tipping point offer between 90% and 70% dominance. Thus we can conclude that, in general, speedups offered by these biased locking algorithms decrease exponentially with decreasing dominance levels.

The preliminary results obtained running the algorithms with 32 threads on the IBM Wirespeed (section 8.5) indicate that these asymmetric algorithms will potentially scale well to higher numbers of processors/hardware threads.

### 9.4 Assessment of Individual Algorithms

In total, 8 biased locking algorithms were implemented and experimented with. There are numerous ways to classify them and group them together. A table illustrating their properties is below:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lock Acquiring</th>
<th>Work Passing</th>
<th>FP Based</th>
<th>Message Based</th>
<th>Queue Based</th>
<th>N-process lock</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymmetric (Vasuvedan)</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Asymmetric Variation</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Function Pointer Passing</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Asynchronous FP</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Message Passing</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Integrated Spinlock</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Queue of Function Pointers</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Queue of Messages</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 9-1 Properties of Various Biased Locking Algorithms
For the purpose of this section, we classify algorithms according to the following categories:

- Lock Acquiring
- Synchronous Work Passing
- Non-Queue Based Asynchronous Work Passing
- Queue Based Work Passing

### 9.4.1 Lock Acquiring Algorithms

The algorithms in this category have non-dominant threads acquire a lock on the critical section perform necessary work in it themselves. They are the Asymmetric algorithm described in [1] and in section 6.3 of this report, and the variation to this algorithm described in section 6.4.

As it reduces the overhead in case where the dominant thread attempts to acquire a lock which is uncontended to only two comparisons, as opposed to needing an atomic operation or a memory fence, Vasuvedan's algorithm gives a considerable speedup over an unbiased spinlock or mutex solution. The variation improves upon this algorithm, lessening the overhead to just one comparison, and as such, was faster in all cases in all benchmarks run. At higher dominances, the variation's performance was similar to that of work passing algorithms with a single function pointer/message. As dominance lessened, its performance declined more quickly than the single function pointer/message algorithms.

Both algorithms were more immune to the “Tipping Point” than those involving work passing with a single function pointer or message. However, they generally exhibited a slowdown compared to the unbiased spinlock at lower dominances.

### 9.4.2 Synchronous Work Passing Algorithms

The algorithms in this category have non-dominant threads push work they need done to the dominant thread, and do not continue executing until the dominant thread has indicated to them that it has completed their work in the critical section. They are the Function Pointer Passing algorithm described in section 7.1, and an identical algorithm which uses messages instead of function pointers, the premise of which is described in section 7.6, which shall be referred to as the Message Passing algorithm.

The only overhead to the dominant thread for an uncontended critical section access was checking if a pointer was null.

Both of these algorithms exhibited similar performance across all benchmarks. Along with the asynchronous algorithms in the next section, they outperformed all other algorithms at very high dominance levels. However, they hit the Tipping Point earlier than any other algorithms (usually somewhere between 95 and 85% dominance).
9.4.3 Non-Queue Based, Asynchronous Work Passing Algorithms

The algorithms in this category are not queue based. They have non-dominant threads push work they need done to the dominant thread, and do not wait for the dominant thread to complete this work before resuming execution, meaning they can. They are the Asynchronous Function Pointer Passing algorithm described in section 7.2, and the Integrated Spinlock algorithm, which is also asynchronous, described in section 7.3.

These algorithms execute identically to the above algorithms from the perspective of the dominant thread. However, they focus on reducing the overhead to the non-dominant threads by allowing them to freely continue on after passing work to the dominant thread rather than waiting until that work has been completed, allowing them to get work to the dominant thread faster. Note that it is essential to the correctness of these algorithms that variables local to the non-dominant threads are not directly modified in the critical section.

As such, they perform similarly to the above algorithms at very high dominance, but are more immune to the Tipping Point, and their performance does not degrade as much as dominance levels decrease.

The Integrated Spinlock generally performs better than the Asynchronous Function Pointer Passing algorithm, because the N-process lock is eliminated and a function pointer is immediately placed in the function pointer variable after a thread “wins” the contest in the pushWork function. In some cases, this algorithm stood out among all other algorithms (e.g. 95% dominance in Figure 8-13 Single Variable Incrementing on cube (99.999-90% Dominance) Figure 8-13).

9.4.4 Queue Based Algorithms

There are two algorithms in this category. They are both queue based, as is discussed in section 7.5. One uses a queue of function pointers, while the other is message based (section 7.6) and uses a queue of messages. Both are asynchronous in operation.

At very high dominances, despite giving good speedups compared to an unbiased spinlock, these algorithms are the slowest of all algorithms experimented with. This is because the overhead to the dominant thread for an uncontested critical section access is checking the equality of two values (to check if the queue is empty).

As dominance lowers, the speedups offered by these algorithms exhibit a much lesser decrease than other algorithms. Notably, in Figure 8-2, they still give speedups at much lower levels of dominance than other algorithms.

They are very immune against the Tipping Point, the reason being that as they can queue up “requests”, one poll by the dominant thread can service many requests, as opposed to just one, and as such, Formula 9-2 does not apply to them.

The Queue of Messages was faster than the Queue of Function Pointers in nearly all cases.

While slower at very high dominances, these queue-based algorithms tend to scale very well to lower levels of dominance. It is proposed that could a faster implementation of the queue
data structure be used, that queue based algorithms could be the best out of all the algorithms experimented with.

9.5 Overall Assessment and Conclusions

Undertaking this project gave an insight into the difficulties faced in the area of shared memory concurrency, and prompted a deep level of analysis into the lopsided access scenario.

Several different approaches to producing fast, biased algorithms similar to the Asymmetric locking algorithm in [1] were considered, and in some cases implemented only to give poor performance and be scrapped. A notable example is a series of experimental algorithms involving the dynamic switching of the CPU affinity of threads. While a good idea in theory, in practice it suffered from severe slowdowns and the approach was abandoned and is not discussed in the main text of this project.

At very high dominance levels with very lightweight critical section accesses, Vasuvedan et al.’s asymmetric algorithm performs very well. The variation to it along with the various novel algorithms described in Chapter 7 perform even better in this situation, and this was a hugely positive outcome of the research from this project. The ability of the Asynchronous and Queue-Based algorithms to maintain speedups at lower dominances with similarly lightweight critical section accesses was also a very positive result.

The decreasing impact of biased locking with larger critical sections gives rise to concern over the potential impact of using a biased locking solution in practical scenarios. For example, if applying biased locking to a multi-core packet processor using pipelining and flow pinning (see section 5.4), the overhead of acquiring a lock in the uncontested case before modifying a thread’s private reassembly table would need to be significant compared to the time taken to perform the actual modification itself in order for a worthwhile speedup to be obtained. In this sense, the practical limitations of locking overhead reduction, as detailed in section 9.1 are extremely important.

Furthermore, one of the main sources of overhead in an unbiased locking scenario is often the requirement of costly atomic operations. However, it was demonstrated in the mini-benchmark in section 3.5 that these operations, such as a compare and swap, are not equally costly on all architectures, and as can be seen in chapter 8, and as was discussed in section 9.3, the less costly compare and swap operation on zooey correlated with lower speedups gained due to biased locking, as opposed to beaker. It is possible that faster atomic instructions on newer hardware will cause locking algorithms that rely on them to require less overhead.

On the other hand, it is unlikely that the cost of revealing operations, i.e. memory barriers/fences will become significantly reduced in newer hardware models, as the slowdowns they incur are a result of inhibiting the processor from performing optimizing reordering of writes. Therefore, the fact that asymmetric algorithms can avoid memory fences in an uncontested critical section access by a dominant thread is a very significant advantage.
Overall, it is felt that lopsided access scenarios are an important special case in the shared memory concurrency problem, and while improvements to new architectures may somewhat mitigate the overhead incurred to dominant threads in such scenarios, it is felt that biased locking, in particular asymmetric locking, is a very worthwhile and interesting area of research.

Contributions made in this project were analysis of the shared memory concurrency problem on modern architectures, evaluation of multi-core packet processing strategies, evaluation of existing biased locking techniques on many architectures, a more lightweight variation of Vasuvedan et al.’s Asymmetric algorithm, presentation of a new approach to biased locking using function pointer and message passing and several new algorithms based on this approach.

The novel algorithms presented in this project appear to perform extremely well and it is felt that a deep insight into this area has been gained, and that very valid contributions have been made.
Appendix A

Code Listing for Thread-Safe Single-Reader Single-Writer Queue

The below code was largely taken from vanDooren's description [17], the only change being the addition of memory fences. It should be noted that the fences are x86 “mfences”. On other architectures, the instruction will be differ.

```cpp
#ifndef MYQUEUE
#define MYQUEUE

template <class T>
class myqueue
{
    public:
    volatile int m_Write;
    static const int Size = 100;
    volatile T m_Data[Size];
    bool popElement(T * Element);
    bool pushElement(T * Element);
    volatile int m_Read;
};

template <class T>
bool myqueue<T>::popElement(T * Element)
{
    if(m_Read == m_Write)
        return false;

    int nextElement = (m_Read + 1) % Size;
    *Element = m_Data[m_Read];
    m_Read = nextElement;
    asm volatile ("mfence");
    return true;
}

template <class T>
bool myqueue<T>::pushElement(T * Element)
{
    int nextElement = (m_Write + 1) % Size;
```
if(nextElement != m_Read)
{
    m_Data[m_Write] = *Element;
    m_Write = nextElement;
    asm volatile ("mfence");
    return true;
}
else
    return false;
}

Listing A-1 Thread-Safe Single-Reader Single-Writer Queue
Bibliography

1 Vasuvedan, N. Simple and Fast Biased Locks. PACT (2010).

2 Dijkstra, E. Cooperating sequential processes. 1965.


