An Experimental Comparison of Concurrent Data Structures

Mark Gibson

**Introduction: (Thin)**

**My Work:**

**Context:**

**Structure:**

**Results:**

**Literature Review: (Fatish)**

**What Motivated You?**

I had been introduced to the idea of concurrency in the third year of my degree and it had piqued my interest. The solutions that concurrency provided for such computing problems as the memory and power wall to me seemed quite elegant. I saw the potential that this technology had and so I took another module based on concurrency in my final year so that I may learn about it in a greater depth. This proved useful to my understanding and so when it came to choosing a project for my final year, I decided to combine my new found interest in concurrency with data structures, something I had always liked since I was introduced to them due to my ability to visualise them in my mind and their inherent usefulness in Computer Science.

**Set the Scene**

**Produce a Critique**

**Method: (Fat)**

**What do you have to do?**

My work is as follows; firstly, I need to implement several different data structures, modifying them so that they can be run concurrently. Secondly, I need to run these data structures on several different computer architectures and gather data on their performance. Finally, I need to compile this data together and analyse it for anything of interest.

**How will you do it?**

After consulting with my supervisor, we agreed to take a modular approach to the project. I would select a data structure, implement it and gather data from it before moving onto the next data structure. I believed that this approach should help prevent confusion regarding different data sets, as I will only move on to another structure once the current one is done completely. In this way I would build my project up piece by piece.

**Ring Buffer**

For my first data structure I decided to go for a circular FIFO queue. I chose this due to its relative simplicity when compared to other data structures and I felt that it would give me an opportunity to get to grips with the atomic libraries I would be working with, as well as setting up the timing metric for testing purposes.

Initially, I implemented a lock using pthread mutexes, where a thread would lock the buffer when it wanted to access it and unlock it when it was done. I chose to do this as I felt that it would provide a good benchmark to test more complicated locking algorithms.

Once I had that up and working properly I decided to implement a TestandTestandSet lock (TTAS) [Herlihy et al, 2008, pg. 144]. This works by using the C++ 11 atomic exchange instruction [Atomic Operations Library. Available: <http://en.cppreference.com/w/cpp/atomic>] to atomically set and unset a lock. Each thread repeatedly checks the lock, if they find that it is equal to one then they sleep for a specified amount of time using the usleep instruction [usleep(3) – Linux man page. Available: http://linux.die.net/man/3/usleep]. After they wake up, the thread then checks the lock again to see if it is equal to one, if so then it starts the loop again, if not then sets the lock to one and enters the critical section [Herlihy et al, 2008, pg.22]. Upon finishing, the thread then sets the lock to zero and the process continues on. I decided to implement this algorithm as it is an efficient implementation of a spinlock as the sleep instruction stops the cpu traffic from becoming overwhelming [Herlihy et al, 2008, pg.147].

Next I decided to implement a less sophisticated spinlock when compared to the TTAS lock, the TestAndSet lock (TAS) [Herlihy et al, 2008, pg. 144]. This lock is less sophisticated, as while the TTAS lock tells a thread to sleep after it has failed to acquire the lock, the TAS lock does no such thing and simply allows the thread to continue polling. This leads to a dramatic increase in the amount of bus traffic between the CPUs in the machine and therefore results in fewer iterations/s when compared to the TTAS lock [Herlihy et al, 2008, pg.145].

Since the ring buffer is a FIFO queue, it is not possible to integrate much concurrency besides a spinlock say compared to a linked list as the push threads and pop threads are all working in their respective areas, the head and tail, there is no concurrent access of the elements between these two points [Circular Buffer. Available: http://c2.com/cgi/wiki?CircularBuffer]. However, I did investigate the possibility of implementing a spinlock written in assembly. Implementing pre-existing code [Spinlocks and Read Write Locks. Available: <http://locklessinc.com/articles/locks/> ] I integrated it into the buffer and compared it to the C++ atomic implementation to try and identify a performance difference. After comparing the two, I found the difference in performance to be negligible and so decided to stick with the C++ implementation, as I found it easier to work with.

The next lock I decided to implement was a lock based on the atomic instruction, ‘compare and swap’ which takes a value and compares it to another. If the first and second values are equal then the first value is replaced by a third value [Herlihy et al, 2008, pg.113]. This can then be placed within a loop, where threads continuously poll until one of them exchanges the lock successfully and breaks free into the critical section. This can create a lot of bus traffic however, similar to that of the TAS lock and so I added an exponential back off, similar in style to the TTAS lock, where a thread, upon failing to acquire the lock would sleep for a progressively larger time up to a defined maximum. The difference between the lock with and without the back off can be seen below: [ADD GRAPH].

The final mode of operation I added was that for a ticket lock, where each thread is given a ticket, and they are allowed to enter the critical section whenever their ticket is being served [Herlihy et al, 2008, pg.32]. This lock performs very poorly at higher thread counts, as due to the queue like nature of the threads when using the ticket lock, if a thread is de-scheduled as it is in the critical section then the entire queue is held up as a result, leading to a significant drop in performance.

**Experiments & Evaluation: (Fatish)**

**Does your method work?**

**Ring Buffer**

Since the ring buffer is a circular FIFO queue, the size of the queue is not important, as threads will be reading/writing from the same place anyway, hence, the following data was collected by using a buffer with 1000 elements.

INSERT GRAPH FOR RING BUFFER

For this, I am using the number of times that the critical section was entered, represented on the y-axis as iterations per second. The number of threads used is on the x-axis and goes from 1 to 128. Four different architectures were used, with each being subject to both modes of operation, Locked and Spinlock.

**Afterword: (Thin)**

**What happened?**

**Lessons learnt?**

**References:**

Herlihy, Shavit. 2008. The Art of Multiprocessor Programming.

N/A (17/07/2013). *Atomic Operations Library.* Available: http://en.cppreference.com/w/cpp/atomic. Last accessed 29/01/2014

usleep(3) – Linux man page. Available: <http://linux.die.net/man/3/usleep>. Last accessed 29/01/14

Michael Brady. 2013. Concurrent Systems II.

(10/02/2007). Circular Buffer. Available: <http://c2.com/cgi/wiki?CircularBuffer>. Last accessed 29/01/14

Lockless Inc. Spinlocks and Read Write Locks. Available: http://locklessinc.com/articles/locks/ . Last accessed 29/01/2014

**Acknowledgements:**

David Gregg

Read The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit

Read several articles on Locklessinc.com/articles

Including Spinlocks and Read Write locks

23/01/14

Agreed with David Gregg to implement a Ring Buffer as the first data structure. I decided to go for a simple design in C to begin with. I implemented it using pthreads and two modes of operation; the first used mutexes in a simple lock and unlock fashion while the second used a spinlock. However, the implementation of the spinlock proved tricky due to the low level nature of C and so I decided early on to start using C++, in order to access the higher level intrinsics, since my focus was not on the implementation but rather the testing of these structures.

<http://people.csail.mit.edu/edya/publications/OptimisticFIFOQueue-DISC2004.pdf> --Gives details on implementing a buffer using CAS.

27/01/14

Researched several papers on the topic of concurrent data structures, placed them into the research folder

Amended the output of the program to be easier to paste into excel for graph generation

Gathered data from stoker for both locked and spinlock modes of ring buffer

Need to gather data from my machine, spoon and ducss

28/01/14

Gathered data from ducss, netsoc and local machine. VPN has only 1 core so no point testing on that. Tested assembly spinlock against C++ version, performance negligible.

29/01/14

Added in several references to the report

30/01/14

Added in a CAS lock with back off to the ring buffer and collected data from all 4 machines

02/02/14

Added a TAS and ticket lock to all 4 architectures and collected data

03/02/14

Got perf + nice working. Came to the conclusion that a graph should be done for each size and mode of operation possible. In addition, started work on a linked list.

06/02/14

Added to the ring buffer section of the report, gathered data on the CAS lock with no back off.

11/02/14

Implemented an assembly version of the ticket lock to the ring buffer to compare against the c++ ticket lock. The difference was negligible.

Started gathering data from stoker using perf

Started gathering data from cube, perf not installed not, have requested

12/02/14

**Gathered data on the ring buffer from cube using perf. In addition I …**

…implemented a spsc lockless ring buffer and gathered data on it from stoker, cube and my local machine both normally and with perf

13/02/14

**Finished off the locked version of the linked list and messed around …**

…was CAS in test.cpp