An Experimental Comparison of Concurrent Data Structures

Mark Gibson

Introduction**: (Thin)**

**My Work:**

I have implemented three concurrent data structures, a ring buffer, linked list and a hash table. Each has one or two variations with regards to how they operate, such as the utilisation and placement of different pointers.

I have implemented them with a mixture of locked and lock free algorithms. Among the locked implementations are a simple pthread mutex, a compare-and-swap lock and a ticket lock. For the lockless implementations I used the C++ 11 atomic library.

I have gathered data and compared the performance of these three data structures over three different systems with varying thread counts.

**Context:**

There has not been much work done in this area and I am hoping to shed some light on this and hopefully discover some interesting patterns or results as I perform my comparisons on the data structures over the different architectures.

**Structure:**

Contents

[Introduction 1](#_Toc380851616)

[Literature Review 1](#_Toc380851617)

[Method 2](#_Toc380851618)

[Experiments & Evaluation 4](#_Toc380851619)

[Afterword 5](#_Toc380851620)

**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.

The problem that presents itself is that there does not seem to be a huge amount of data comparing the performance of concurrent data structures on different architectures. There is plenty of work done with regards to designing concurrent data structures [Moir et al. 2001] and implementing them [Herlihy. 1993], though considering the amount of research done on that topic, there is little to go on when it comes to comparing these structures across different architectures, to see how they affect the performance of the data structures.

Hence, I am hoping to add to what little has been done in this area by performing my tests and analysis.

**Set the Scene**

**Produce a Critique**

Method**: (Fat)**

**What do you have to do?**

My work is as follows; firstly, I need to implement the three different data structures, modifying them so that they can be run concurrently. This involves adding both locked and lockless modes of operation to the data structures.

Secondly, I need to run these data structures on the three computer systems and gather data on their performance. This data will be based on the number of iterations performed by each program per second, the number of threads running concurrently and size of the data structure in question.

Finally I need to gather this data together and analyse it for anything of interest. To aid in the collection and analysis of this data as accurately as possible I will be using tools like Perf [Perf Wiki. Available: <https://perf.wiki.kernel.org/index.php/Main_Page>] which can measure hardware counters and record such things as idle CPU cycles, cache misses and page faults.

**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 one. I believed that this approach would help prevent confusion regarding different data sets, as I would only move on to another structure once the current one is finished. 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]. Out of interest, I also decided to implement the TTAS lock without a pause instruction to investigate the effect that this would have on it.

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]. Like with the TTAS lock, I decided to add a pause instruction to the TAS lock to investigate what, if any difference it would have.

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 locked 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. As with previous locks, I decided to implement an assembly version of the ticket lock to compare its performance against the C++ implementation. However, as with previous times, I found the two implementations to be negligible in performance and so decided to stick with the C++ implementation.

For my lockless implementation of the ring buffer, I decided to implement a single producer – single consumer queue. To push, the front of the queue is taken and the index after it is examined. If the back of the queue is not pointing there, then an item is pushed to the front of the queue, and the index after it becomes the new front. Alternatively, to pop, the back of the queue checks that it does not share the current index with the front of the queue, and only then will it remove an item from the queue.

**Linked List**

For my next data structure, I decided to implement a singly linked list. I did this because I already had some experience with implementing this structure both serially and concurrently from previous assignments during my time in college. In addition, it is relatively simple and I had hoped that it would act as a stepping stone to the more advanced data structure when the time came.

As with the ring buffer I added both locked and lockless versions. The locked versions were all the same as the ring buffer, where a lock would be acquired, the add or remove code would be executed and the lock would then be released. The following locked modes of operation were implemented for the ring buffer: Simple pthread mutex lock, TestAndTestAndSet, TestAndTestAndSet with no pause instruction, Compare and Swap lock, Compare and Swap lock with no backoff, TestAndSet, TestAndSet with a pause and finally, a ticket lock.

I decided to base my lockess implementation of the atomic instruction ‘compare\_exchange’ from the C++ 11 atomic library. I declared a pointer of type Node to be atomic which represented the head of the linked list. I then used this to atomically switch the list’s pointers whenever a node was added or removed. Interestingly, I only had to use this atomic instruction a subset of the total cases in my code. For example, whenever the list was empty, or when the head had to be changed, I had to use the atomic instruction to change the head. However, for cases where node wanted to insert itself in the middle of the list, no atomic instruction seemed necessary. I tested this on a list with differing maximum sizes, with the lowest being of size 5 nodes and yet the list remained valid through every test.

For the lockless list, I decided to test it with a list of three different maximum key ranges, where, as each key was randomly generated, it would be subjected to the modulo of the key range and since the list was ordered and had no duplicates, this was put in place a hard cap on the maximum size of the list. I originally picked the values of 100, 100,000 and 100,000,000 to be the key ranges, but keeping in mind the cost of the modulo operation in terms of CPU cycles [reference] I decided to replace these with powers of two, namely 128, 131072 and 134217728 as I knew that the compiler I was using, g++ would optimise the modulo operation by replacing it with a bit shift operation, which requires far less CPU cycles [reference] and minimising the cost of using such large numbers.

**Linked List ALT**

After implementing the linked list and observing some of the data that was being gathered I saw that once the list started to get long, past 1,000 nodes, the performance dropped off significantly. I concluded that it was due to the time being spent by the threads traversing the list looking for an insertion point or a node to delete.

I felt that this was not optimal, as the size of the list was interfering with the comparison of the locking algorithms. Hence, I decided to remove the traversal issue all together and implemented a multi-consumer, multi-producer linked list buffer. This worked by always adding and removing from the head and tail respectively. There was no traversal of the list necessary and while this did mean that the list would no longer be ordered or free from duplicates, it was my opinion that this would provide clearer data from the locking algorithms.

As I was implementing it initially, I felt that to take my original linked list and to convert it into the FIFO buffer that I wanted I would need to introduce a second pointer to each node, which would point to the node previous to it, essentially making it a doubly-linked list, as after removing a node from the end, the tail would need to know what the previous node in the list was so that it could point to it.

It was only after I had finished implementing the buffer that I realised that I did not need the second pointer for each node if I simply rearranged the placement of the pointers. If I swapped the head and tail pointers around then I would again only need one pointer per node to implement the data structure. It worked as follows: the tail pointer would keep track of the oldest node in the list. Whenever a new node was added, the last node to be added would then be pointed to this node and the head pointer would move to the new head. It was in essence, flipping my initial implementation around but that small change reduced the complexity and size of the data structure as now, each node again only needed to store one pointer and all the code that was added to deal with the second pointer could be removed.

**Hash Table**

**Lockless**

For designing the lockless hash table I made the following decisions based on research done with regards to lockless hash tables; it would be a closed addressing hash table, each index in the table would point to a linked list, so any collisions would result in a node being added onto the relevant list. Finally, it would have a coarse-grained resize function, which involved transferring the lists or buckets to a new, larger table [reference].

To represent the buckets I decided to use the lockless linked list I had already implemented, as I had already tested it when I was collecting the data from it and it would save me time. I decided to go with my FIFO buffer implementation of the linked list to eliminate traversing the buckets as an issue. I gave each bucket two atomic variables, a head and tail pointer, which would reduce the time spent adding/removing nodes and would ensure that I could do it atomically through the use of the C++11 atomic library. This would be the only use of the atomic library; the hash table itself did not have any atomic variables.

After I had implemented the data structure I ran into two points of interest. The first was that as the program ran, it would sometimes post extremely low results for one of the iterations, usually the iteration using four threads in total. To try and discern the cause I added in a counter that tracked the failure rate of the atomic instructions in both the add and remove functions, but this proved to not be the cause of the problem as the resulting values I was getting were both quite low, no more than fifty failures per iteration and these did not correlate to the drop in performance I was observing. I decided to put it aside for a while, with the intention of returning and utilising the tool perf to try and find the cause of the performance drop.

The second point of interest I encountered was that the program occasionally caused a segmentation fault while it was running at high thread counts, around 32 threads or more, though sometimes it occurred at lower counts such as 8. As the problem’s frequency seemed to increase at higher counts my first thought was that it might be a contention issue. After reviewing the code, I noticed that I was accessing the hash table a lot during both the add and remove function calls in the form of “htable->table[hash]”. I believed that this may be the cause of the segmentation faults, as if a thread was halfway through an add, another thread may change the value of the hash among other things, leading to a segmentation fault. I tried to solve this by instead passing the bucket reference to a variable, tmpList which I would then use in the computation. In addition, I added several more checks into my code, checking that tmpList still pointed to the place it was supposed to and that if it was not then abort the operation and try again. To test to see if the problem had been fixed by my changes I set it to run twenty times, one after another, with the intention that if a segmentation fault would appear, indicating that the problem had not been fixed , that it would in these conditions. Luckily, this was not the case and my implementation seemed to be working correctly.

**Locked**

For the locked version of the hash table, I decided to have two implementations. The first implementation involved locking the entire hash table with a lock whenever a thread wanted to interact with the table. The second implementation differed from the first in that there was no global lock, but instead each bucket had its own lock. So whenever a thread wished to interact with a specific bucket, it would obtain the bucket’s lock and perform its work, in this way it allowed for the absence of a global lock and instead had a more modular approach which I would then compare to the first locked implementation.

To keep search times constant, I had to add in functionality to allow my hash table to resize itself when buckets got too full [reference]. I decided to implement a locked resize function first, which involved going through each bucket and rehashing each key. Then depending on the value of the hash, the key would either be left alone or transferred to another bucket. I was able to write this part of the implementation serially, as it is only called in inside the add function, where at which point a lock will have already been obtained, making the need for additional locks irrelevant.

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.

**Linked List**

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

(20/02/14). [Perf Wiki. Available: <https://perf.wiki.kernel.org/index.php/Main_Page>].

Moir, Shavit. 2001. Concurrent Data Structures

Herlihy, 1993. A Methodology for Implementing Highly Concurrent Data Objects

**Acknowledgements:**

David Gregg

Jeremy Jones