University of Dublin

TRINITY COLLEGE

An Experimental Comparison of Concurrent Data Structures

Mark Gibson

B.A.(Mod.) Computer Science

Final Year Project April 2014
Supervisor: Dr. David Gregg

School of Computer Science and Statistics

O’Reilly Institute, Trinity College, Dublin 2, Ireland

Acknowledgements:

David Gregg

Jeremy Jones

David Fennell

Michael Gallagher

Contents

1 Introduction: (Thin)

1.1 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 [reference] which contained the necessary atomic operations.

I have gathered data from these three data structures using varying thread counts and other variables, such as list or table size. The data was gathered from a total of three machines.

1.2 Context:

There has been much work done in the area of implementing concurrent data structures, with the area of concurrent programming expanding rapidly corresponding with the rise in multicore machines [reference]. However, despite all this research and work into concurrency, there is still a lack of data on the comparisons of different locking algorithms on these data structures. We are still unsure of the performance of different locking methods and if lockless algorithms are always preferred over locked alternatives. Hence, this project hopes to shed some light on the area by taking 3 concurrent data structures and testing them with several locking strategies to deduce if there is any correlation between certain algorithms and the data structures being used.

1.3 Results:

2 Background & Literature Review: (Fatish)

2.1 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.

2.2 Locked & Lockless Programming

A lock in terms of computer science is a synchronisation mechanism which is used to control access to a resource in an environment that contains more than one thread of execution [reference]. A common lock to implement using pthreads is a mutex, which is used to protect a shared resource. It works by only ever allowing one thread to 'own' it and only the thread who owns the mutex can access the resource [reference]. While this is a convenient way of ensuring mutual exclusion [reference], it does not scale with an increased amount of computing power or threads [reference], as only one thread can access the resource at any given time.

The term Lockless in computer science when referring to a non-blocking algorithm [reference] equates to an algorithm where threads who are competing for a shared resource do not have their execution postponed by mutual exclusion. These algorithms, while not using locks such as a mutex, still use atomic instructions as a means to protect a shared resource [reference].

The term Lock Free when discussing non-blocking algorithms, means that individual threads are allowed to starve, but progress is guaranteed on a system wide level. At least one of the threads will make progress when a program's threads are run for sufficiently long.

The term Wait Free when discussing non-blocking algorithms represents the strongest non-blocking guarantee of progress. It guarantees both system wide progress and starvation freedom for the threads [reference]. Every operation has abound on the number of steps the algorithm will take before the operation completes [reference]. It is for this reason that all wait free algorithms are also lock free [reference].

An Atomic Instruction is an operation that completes in a single step relative to other threads. When an atomic instruction is performed, no other thread can observe the instruction half completed, it will either complete entirely or not do anything, much like have transactions in databases work [reference]. Without this guarantee of completion, lockless programming would not be possible, as there would be no way, other than using a lock such as a mutex, to protect a shared resource used by multiple threads [reference].

A Concurrent Data Structure in computer science is a data structure that has been designed and implemented for use by multiple threads [reference]. As a result they are significantly more difficult to design and verify than regular, serial data structures, due to the asynchronous nature of threads. However, this added complexity can pay off as concurrent data structures can be very scalable if the shared resources of the data structure can be properly protected and utilised by the threads working on it [reference].

2.3 Previous Work

2.3.1 The Art of Multiprocessor Programming

When it came to researching what work had been done before now, I initially looked towards "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit. It covers much of the current state of multiprocessor programming, detailing some of the various problems that are encountered with concurrent programming, such as the Producer-Consumer problem [reference] and the ABA problem [reference]. It then delves into the foundations of shared memory [reference] and the basics of multithreaded programming, detailing the spin lock and the issue of contention, where many threads vie for control of the lock [reference]. It then goes through several data structures, such as the linked list and hash table, describing the different aspects of design and implementation and the problems one faces when attempting to implement locked and lockless forms of these structures. Considering how closely this book follows my own work, it was only natural that it inspired me and guided me while I was choosing, designing and implementing the data structures for my project.

2.3.2 Designing Concurrent Data Structures

"Designing Concurrent Data Structures" by Mark Moir and Nir Shavit, goes into depth on the processes required to successfully design a concurrent data structure, both in the general sense, talking about issues like blocking and non-blocking techniques [reference], performance and verification techniques [reference] while also going into detail for a range of specific data structures, like Stacks and Queues, Linked Lists and hash tables.

2.3.3 Implementing Concurrent Data Objects

"A Methodology for Implementing Highly Concurrent Data Objects" by Maurice Herlihy goes through the process of implementing a concurrent data structure, highlighting the issues with the conventional techniques of relying on critical sections [reference] and instead leaning towards a lockless approach, and the differences between lock-free and wait-free approaches.

2.3.4 Experimental Analysis of Algorithms

"A Theoretician's Guide to the Experimental Analysis of Algorithms" by David S. Johnson discusses the issues that can arise when attempting to analyse algorithms experimentally, where he goes over several principles which he feels are essential to properly and accurately analysing algorithms ranging from, "Use Reasonably Efficient Implementations"[reference] to, "Ensure Comparability"[reference]. In addition, he also goes over ideas and techniques of presenting data [reference] which I found to be most interesting.

2.3.5 A Lock-Free, Cache Efficient Shared Ring Buffer

2.3.6 Resizable Scalable Concurrent Hash Tables

3 Method: (Fat)

3.1 What do you have to do?

My work is as follows; firstly, I need to design and implement the three concurrent data structures. This involves adding both locked and lockless modes of operation to the data structures to allow them to be used by multiple threads.

Secondly, I need to run these data structures on the three different computer architectures I have at my disposal 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.

3.2 How will you do it?

After consulting with my supervisor, Dr. David Gregg, 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.

3.2.1 Recap of locked & lockless

As mentioned previously in the 'Background' section of the project, Locked algorithms use mutexes or other such constructs to provide a lock. Threads must acquire this lock to enter the critical section. Once a thread finishes in the critical section it releases the lock allowing another thread to enter. All threads without the lock are blocked, forming a bottleneck of execution [reference]. I will be implementing my locked variations as follows: I will implement different locks such as mutex, test-and-set etc. by using the pre-processor to define variables which will impact the path of execution. For example, when I implement the pthread mutex lock I will wrap any related code in an if-endif block using the pre-processor. Hence, when I execute the program and define the associated instruction, only the mutex code will be executed, leaving the rest of the lock unimplemented.

Lockless algorithms are defined by their use of atomic instructions to ensure thread-safe execution such as compare-and-swap which allows an object to atomically check if it contains a value and if so to swap it out for another value. Lockless algorithms do not used locks and so system wide throughput is guaranteed. To ensure that the locked and lockless algorithms are equal I plan to wrap each atomic instruction which can fail in a while loop so that if it fails it is forced to try again. I do this because with the locked variations, if a thread is trying to add an item to a linked list for example it will always succeed unless the list is full. However, if a thread attempts to do this with an atomic instruction and it fails then the node may not be added and so the work done by the two threads would not be equal.

3.2.2 List of locked modes

3.2.2.1 LOCKED

This lock would be composed of a pthread mutex [reference]. I chose it as I found it to be a very simple lock to implement. I also considered that, due to its simplicity, that it would provide an excellent baseline for me to test other locks against.

3.2.2.2TTAS

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].

3.2.2.3 TTASNP

This locked mode is the TTAS lock but without the sleep instruction after the second while loop. I added this as I was interested in the effect that the sleep instruction has on the performance of the lock in relation to my different data structures.

3.2.2.4 TTAS_RELAX

This is near identical to the normal TTAS lock but with one difference, the sleep instruction is replaced by the intrinsic _mm_pause() which is designed to reduce the performance impact that repeated thread polling can have on bus traffic and the CPU's pipeline [reference]. I added this mode as, like with the TTASNP mode, I was curious as to how the change would affect the lock's performance and if the intrinsic gave this mode an advantage over the sleep instruction.

3.2.2.5 TAS

I wanted to implement a TestandSet (TAS) lock because it is somewhat less sophisticated when compared to the TTAS lock [Herlihy et al, 2008, pg. 144]. It is more basic because, while the regular 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 per second and hence, a loss in performance when compared to the TTAS lock [Herlihy et al, 2008, pg.145].

3.2.2.6 TASWP

Like with the TTAS lock, I decided to add a sleep instruction to the TAS lock toinvestigate what, if any difference it would have on the lock's performance.

3.2.2.7 TAS_RELAX
Again, as with the TTAS lock, I decided to compare the sleep instruction implemented in TASWP and TTASWP with the intrinsic _mm_pause to investigate the difference, if any it would have on the different locks.

3.2.2.8 CASLOCK
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, but with each failed attempt, would sleep for a progressively larger time up to a defined maximum.

3.2.2.9 CASLOCKND
As with previous locking modes, I wanted to ensure that lock was implemented thoroughly, with different variations, and so I chose to implement the CASLOCK but without the exponential back-off to investigate if it was really necessary and if so when and in which scenarios it made a difference.

3.2.2.10 CASLOCK_RELAX
Similar to both the TAS_RELAX and TTAS_RELAX, the CASLOCK_RELAX mode replaced the exponential back-off, but instead of getting rid of it all together I replaced it with the intrinsic _mm_pause to investigate which had the better performance between it and the back-off.

3.2.2.11 TICKET
The final type of lock I added was 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 [reference]. As with the TTAS lock, if a thread polls and finds that it is not its turn in the queue yet it sleeps, where the amount of time sleeping is proportional to how far back in the queue the thread is, so if the thread is relatively close to the top of the queue it will sleep for less than if it was near the bottom of the queue.

3.2.2.12 TICKET_RELAX
As with the previous locks, I decided to compare the impact of the sleep instruction on the ticket lock by replacing it with the intrinsic _mm_pause and compare the two for performance.

3.3 Data Structures

3.3.1 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 give me a chance to finalise how I will be collecting data from the data structures.

3.3.1.1 Locked

For the locked variation I decided to go for a simple locking strategy where if a thread wished to interact with the buffer that it would acquire a lock, perform its operation and release the lock. This approach would only allow one thread to access the buffer at any one time and so would hopefully provide a nice contrast to the lockless implementation.

While implementing the different locked modes I came across the idea of implementing the locks in assembly, something which had already been done for some of the locks [Spinlocks and Read Write Locks. Available: http://locklessinc.com/articles/locks/]I decided to compare the performance of some of the locks I had already written to their assembly counterparts. If it was the case that the assembly implementations proved to have an advantage over the C++ versions then I would switch to them in order to procure more accurate results. Hence, I integrated them into the buffer and compared them to their C++ implementations to try and identify a performance difference. After comparing the locks, I found the difference in performance to be negligible between them and so decided to stick with the C++ implementation of the locks, as I found them easier to work with.

3.3.1.2 Lockless

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

I found this to be a good introduction to the C++11 atomic library as I was able to get to grips with declaring atomic variables and calling the library's functions, such as std::atomic_fetch_add which atomically increments a value by a given amount [reference].

3.3.2 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, I believe that it is relatively simple to design and implement and I had hoped that it would act as a stepping stone to the more advanced data structures when the time came for those to be implemented.

3.3.2.1 Singly Linked List

This variation of the linked list contains one class, the Node class which is used to make up the linked list. This class had two attributes and a constructor function. The first attribute, key, represents the value assigned to the node. The second attribute, next represents a pointer of type Node which is used to point to the next node in the linked list. Finally, the constructor takes two parameters, a value and a pointer of type Node and assigns them to their respective attributes within the node. This variation is implemented in such a way as to be ordered, so that the smallest values are at the head of the list and that there are no duplicate values in the list.

The head of the list, a pointer of type Node, is not part of any class as I decided to not add a List class for this variation as I encountered problems with calling the pthreads.

This variation contains three functions, Add, Remove and printList. Add works by randomly generating a value using the rand() function [reference]. It then creates a node using this key and attempts to add it to the list of nodes. To begin, it gets the current value of the head variable. If the head is equal to NULL then a list does not yet exist, so it sets up the list. If the list already exists but the node that has just been generated has a smaller value than the one at the head of the list, then the new node is inserted in front of the head of the list and the head is changed to the new node. If the list exists and the node to be added is not smaller than the head of the list then the list is traversed by getting a copy of the head pointer and repeatedly assigning the value of each node's next pointer. In this sense it can move down the list, checking the values of each node as it goes. If it finds a node that is larger than the new node in terms of key value then it inserts the new node before the larger node. It does this by pointing the new node's pointer to the larger node and by getting the node previous to the larger node and assigning its next pointer to the new node. If the end of the list is reached, marked by a node's next pointer being equal to NULL then the new node is simply added onto the end of the list, by assigning the next pointer of the last node in the list to the new node, making it the last node in the list.

The Remove function works in that it first generates a random number which will act as the value that it will search for and try to remove from the list. Firstly it takes a copy of the head of the list and checks if it is equal to NULL. If it is then there are no nodes in the list to remove. Alternatively if the key of the head of the list is equal to the key that the function is searching for then it will point the head to the next node in the list and remove the now isolated node. If neither of these cases is true then the list is traversed until either the node is found or the end of the list is reached. If the node is found in the list then the next pointer of the node before it is changed so that it points to the node that the node to be deleted points to, effectively removing that node from the list.

The printList function is relatively simple compared to the Add and Remove functions. It simply takes a copy of the head of the list and traverses the list until it reaches the end. For each node it prints out their key value followed by a comma.

3.3.2.1.1 Locked

The locked version of this variation would be similar to the locked version of the ring buffer I implemented, where any attempt to act on the list would require a thread to acquire the lock, which it would then release once it had completed its work. Since this was a locked variation I did not have to declare the head of the list as an atomic variable, so I was able to simply declare it as volatile which prevents the compiler from optimising any code that it is a part of [reference]. I declared the key attribute of the Node class to be volatile for the same reasons, along with any function level variables that dealt with the head or Node.

For the Add function I added in all the different locking modes to acquire the lock before the key value was randomly generated and added in the unlocking code at the end of the function, ensuring that only one thread could access the body of the function at any one time.

The same was done for the Remove function, the acquiring and releasing code was added before the key generation and after the body of the function respectively.

The printList function did not need to have any locking code added as it only called in the main function once the threads had finished their work and had been terminated.

3.3.2.1.1 Lockless

I decided to base my lockless implementation of this variation of the linked list on the atomic instruction ‘compare_exchange’ and its associated functions from the C++ 11 atomic library [reference]. To do this I would need to declare at least one atomic variable to call the necessary functions so I chose the head pointer of type Node. I did this because having an atomic head pointer would allow me to atomically change the head of the list. For this implementation I decided to remove the volatile keywords from the code and see if it made a difference to the validity or the performance of the data structure.

For the Add function, the code was somewhat smaller in size than the locked version as I did not need to add the different locking modes. Instead, the head is atomically loaded into a variable which is then checked to see if it is equal to NULL. If so then the atomic head pointer is changed from NULL to the new node that was created beforehand. This is done using the std::atomic_compare_exchange_weak function which acts as an atomic compare-and-swap instruction [reference], else if the head needed to be changed to another node than the atomic function would be called again, instead swapping the value of head from the old node to the new node.

It was at this point that I came across a point of interest in the code. I was unsure how to proceed with writing the code for atomically traversing the list so I decided to implement it serially and see what happened. I then ran the code several times and to my surprise the list it created was ordered with no duplicates and appeared to work locklessly for all intents and purposes. I repeated the procedure for the Remove function which was designed identically to the Add function, with atomic instructions for dealing with the head but serial code for dealing with list traversal and the results were the same.

To try force an error from my implementation I changed the maximum list size to five and ran it. Such a small list should have encountered a lot of contention considering the number of threads acting on it and yet no errors were found in the lists that were produced.

3.3.2.2 Doubly Linked Buffer

3.3.2.2.1 Locked

After implementing the singly 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 in length, 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.

To implement this I added a tail variable of type Node * which I declared using the volatile keyword, similar to the head variable. The tail would be used by having it point to the end of the list, recording where the end of the list and giving a location for the threads to remove nodes from. However, to implement this I realised that I would need to add a second pointer to the Node class, prev, as if the tail was pointing to the end of the list then whenever a node was removed the tail would need some way of then pointing to the previous node in the list.

In one sense this simplified the implementation as the code required for traversing the list was no longer required; all that was needed was code to set up the list if no node existed and to add/remove from the head and tail respectively.

3.3.2.2.2 Lockless

To implement the lockless version of the doubly linked buffer I started off by declaring the new tail pointer as an atomic object. This would allow me to atomically remove objects from the end of the list as the atomic head pointer allowed me to add things onto the front of the list.

Adding objects involved generating the node to be added then atomically switching the head pointer from what it was pointing at to the new node being added. Since the list no longer had to be traversed, the process of adding a node lucklessly became much simpler.

Removing a node was much the same as adding a node but in reverse, where the tail pointer was atomically switched to point to the previous node in the list using the old tail’s prev pointer to become the new tail of the list. The old tail was then discarded.

3.3.2.3 Singly Linked Buffer

It was only after I had finished implementing the doubly linked buffer that I realised that I did not need the second pointer for each node if I simply rearranged the placement of the head and tail 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 works 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.

In terms of implementation it was very similar to the doubly linked buffer with the only real differences being that there were no longer any references to a Node's prev pointer as that had been removed and the references to the head and tail would be mixed up as they had switched position and function with this latest implementation. This was the case with both the locked and lockless variations of the data structure.

3.3.3 Hash Table

3.3.3.1 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.

3.3.3.1.1 Global Lock

The premise for the globally locked hash table was simple, I wanted a baseline to compare my other two implementations on, the lockless and lock per bucket variations. In addition, I felt that it would be useful to get the add and remove functions working and tested in this implementation before moving onto more advanced variations.

As this was a baseline implementation, I decided to go for a very basic locking strategy, where a lock was acquired before a thread interacted with the table at all, and that the lock was global, in that only one thread could interact with the hash table at any given time, any other thread that attempted to interact with the table would be blocked.

3.3.3.1.2 Lock Per Bucket

This variation of my locked implementation of the hash table would be different in the sense that instead of threads acquiring a global lock, where only one thread would be able to access the table at any one time, each list in the table, or bucket, would have its own lock. In this way, multiple threads could work on the hash table at any given time and that they would acquire a lock for the bucket they were about to interact with so a thread would only be blocked if it attempted to interact with a bucket that another thread was already interacting with.

I felt that this implementation was more complex than the globally locked variety, I ran into some trouble when I attempted to implement the rest of the locking modes besides the basic pthread mutex lock, though I discovered that it was because I had mixed up a reference to one bucket's lock with another. After I had corrected this I was able to implement the rest of the locked modes, TAS, CAS, TICKET etc with no further delays.

3.3.3.2 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.

3.3.3.3. Resizing

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 the key would be transferred to the new table, based on its new hash. I was able to write this part of the implementation serially, as it is only called inside the add function, where at which point a lock will have already been obtained, making the need for additional locks irrelevant.

For my lockless implementation I investigated several potential methods, one of which involved leaving the keys where they were and forming new lists from them by dynamically creating each bucket [reference, concurrent hash table]. Another option from the same paper was to resize the table in place, where the current table was made bigger and the keys rehashed. Yet another option was to incrementally resize the table, where all adds started to add to another table, with remove and contain calls checking both tables and only switching to the new table when all the keys had been transferred from the old [reference]. I decided to try and implement the first solution and see how I got on. I immediately ran into problems with segmentation faults as I was unable to implement a necessary amount of atomicity to stop the threads from interfering with each other. This problem persisted for the two other solutions I attempted, each was plagued by segmentation faults which I was unable to get rid of. In the end I had to settle for using a lock, similar to my locked implementation, where only one thread was allowed access to resize the table.

To compensate for my inability to implement a lockless resize function, I planned to test my implementations with a large initial table size. I hoped that this would minimise the need for the table to resize and so have the smallest impact on the performance, allowing me to compare the locked and lockless algorithms almost purely based on what I had written already, the add and remove functions.

3.3.3.4 Contains Function

Before I began testing my hash table I decided that I wanted it to replicate a real world hash table as closely as possible. To do this I would need to add in a contains function, a function that took key and searched for it in the hash table [reference]. I would need to implement this functionality in all three of my hash table variations. The implementation itself was relatively easy, I randomly produced a key, got its hash and then retrieved the bucket associated with that hash. Once I had that I then iterated through the bucket until I had either found the key or I reached the end of the bucket.

3.3.3.5 Tracking Search Results

As a means to record positive and negative hash table searches I added in two variables, pSearches and nSearches to represent the total number of positive and negative searches each time the program ran. I did this because I planned to utilise these when I was testing the table to see if there were any correlations between the number of successful/unsuccessful searches and the table performance

3.3.3.6 Choose function

With the addition of the contains function in my hash table, I now encountered something which I had not done so far in the project. Whereas with the ring buffer and linked list there were just two functions, the hash table now had three. I could no longer simply assign half of the threads to adding and half to removing items. I had to come up with a better solution. An additional concern was that I wanted to replicate the function call ratios for hash tables, which are about 90% contains calls, 9% add calls and 1% remove calls [reference Art of Multiprocessor…]. In the end I decided to implement the choose function.

The choose function would be relatively simple, now, whenever a thread was spawned, it would call the choose function, instead of calling the add or remove function. Inside the choose function, a number would be randomly generated, initially I used the modulo operation to cap the number at 100 and then used an if-else block, where if the number was greater than 9 then the thread would call the contains function, else if it was greater than 0 it would call the add function, else it would call the remove function. After testing I found that this replicated the function call ratios I had encountered earlier, though I decided to change the cap of 100 to 128, so that the compiler would streamline the operation [reference], and hence, I changed the values in the if-else block to correspond to it.

4 Experiments & Evaluation: (Fatish)

-Evaluate locked vs lockless

-Evaluate differences between different locked modes, sleep vs cpu relax for example

4.1 Evaluation Strategy

To gather data from and evaluate the algorithms I was testing I decided to create the following evaluation strategy. Firstly, I would graph the data based on two main factors, the number of iterations per second generated by the algorithm being tested and the number of threads that were created for each iteration of the algorithm. This would allow me to graph the performance of each algorithm as the number of threads being generated increased. I decided on 128 as being the maximum number of threads spawned as this was twice the number of core of stoker which I believed at the time to be 64. I later found out that it actually had 32 cores but I decided to stay with the 128 figure as I had already collected some data with that thread count and it seemed to wasteful to throw it away.

To measure the iterations per second I had to first record how long the program took to finish. The system time was gotten at the start of the main function in each program; each thread was then created and run. After each iteration the thread completed it would check if a given amount of time had passed, in this case it was one second. If one second had passed the thread would stop and once all threads had stopped the time was again gotten from the system. The start and stop time were then used to calculate the running time and this was then divided by how many seconds each thread was allowed run which was ~1 and this produces the iterations per second for each algorithm.

In addition to the iterations per second and thread count I would be varying the size of the different data structures so investigate if size played a role in the performance of the locked and lockless algorithms.

Finally, when it came to collecting accurate data, I made sure to only collect data from each machine when CPU load was low so as not to jeopardise the data. In addition, while I initially collected data by running each algorithm once I felt the variance between the different iterations, while small was not negligible and so I changed my method to instead run each algorithm 7 times and then to get the median. I chose the median over say the average as it would give a better representation of the data [reference]. I calculated the median for each algorithm after it had run 7 times before moving onto the next algorithm. I ensured that it did not impact the performance of the algorithms by performing the calculations outside of the timed period of each program.

To speed up the time it took to gather results I wrote a simple bash script to automate the defining of the different locked modes of operation. Whereas before I had to manually enter the program and define/undefined each mode each time I ran the tests it was taking up a lot of time. The script was able to compile the program and define each variable separately using the –D option in the command line [reference]. This sped up the process of gathering results significantly.

To further increase the quality of my implementations to provide the best comparison between the different implementations I used the –O3 flag on the g++ compiler to turn on several optimisations supported by the compiler [reference].

4.2 Ring Buffer

4.2.1 Evaluation

Apart from the iterations per second and thread count I decided to vary the size of the buffer to investigate how this effected the locked and lockless variations if at all. The starting size of the buffer was 128; I initially had this at 100 but I decided to change it to be a power of two to minimise the effect that the modulo operation would have on the program's performance since the compiler would reduce it to a bitwise AND operation [reference].

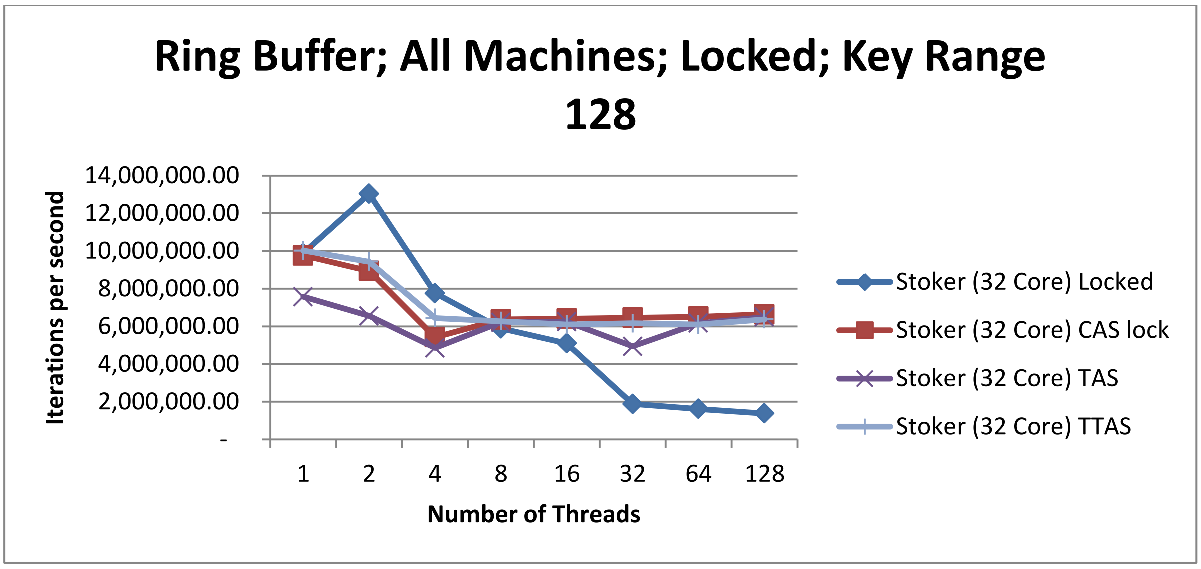
I did not expect the size of the buffer to have an impact on performance as items were added/removed from either end. There was no interaction between the threads and the items in the middle, hence the number of items between the front and back of the buffer should not matter, though I decided to confirm this and test it.

It was at this point in the project that I considered implementing assembly versions of the locked modes I had. http://locklessinc.com had several implementations in assembly such as a ticket lock and test-and-lock. I implemented them as described and then compared them against the C++ implementations I already had. I discovered that the performance between the two versions was negligible and so I decided to stick with the C++ versions I had as I felt more comfortable in my ability to understand and modify the locks as necessary, whereas with the assembly locks this was less so.

4.2.2 Results & Analysis

4.2.2.1 Locked Modes Comparison

I began my evaluation by first focusing on the different locked modes of operation before moving onto the lockless variation as I wanted to isolate the best performing locked modes to compare against the lockless ring buffer. The Graph below represents the best four locked modes of operation; the "Locked" mode represents a simple pthread mutex lock, "CAS lock" represents a Compare-and-swap lock, "TAS" represents a Test-and-set lock and "TTAS" represents a Test-and-test-and-set lock. These four modes had the best performance consistently across the four machines



To determine as to why these four modes did quite well I compared two of them, the LOCKED and TTAS modes against a mode that did not perform nearly as well, the TAS_RELAX mode which was similar to the TAS mode but it had a cpu_relax instruction after the poll. The hardware performance counter data is as follows:

	LOCKED	TTAS	TAS_RELAX
Cycles	2,102,076,100,442.00	76,583,801,948.00	2,589,400,577,492.00
Cache Misses %	38.35	37.37	92.12
Branch Misses %	0.09	0.06	0.33
Stalled Frontend Cycles %	95.10	67.82	99.31
Stalled Backend Cycles %	64.99	46.33	92.63

From the table we can see that TAS_RELAX has a ratio of cache misses to references of over twice that of both the LOCKED and TTAS modes. In addition it misses five times more branches and has a greater ratio of misses for front and backend cycles. From this we can see why TAS_RELAX did so badly while LOCKED and TTAS did relatively well when compared to it.

4.2.2.2 Lockless Comparison

Since the lockless ring buffer implemented was a Single Producer Single Consumer data structure (SPSC) There is only one result for each size on each machine as the thread count does not change, it remains at two, with one thread pushing items and the other popping them, the resulting table is below:

Machine	128 Size	131072 Size	134217728 Size	Total Iterations
Stoker (32 Core)	12,433,048	9,419,968	11,740,401	33,593,417

Local Machine (4 Core)	10,277,618	10,713,959	10,394,564	31,386,141
Cube (16 Core)	11,168,960	11,539,810	9,265,828	31,974,598

As seen above there does not appear to be a correlation between the maximum size of the buffer and the performance of the lockless algorithm, though we can see that Stoker performs the best overall with over 33 million iterations though it is in no way a clear leader

4.3 Linked List

4.3.1 Singly Linked List

4.3.1.1 Evaluation

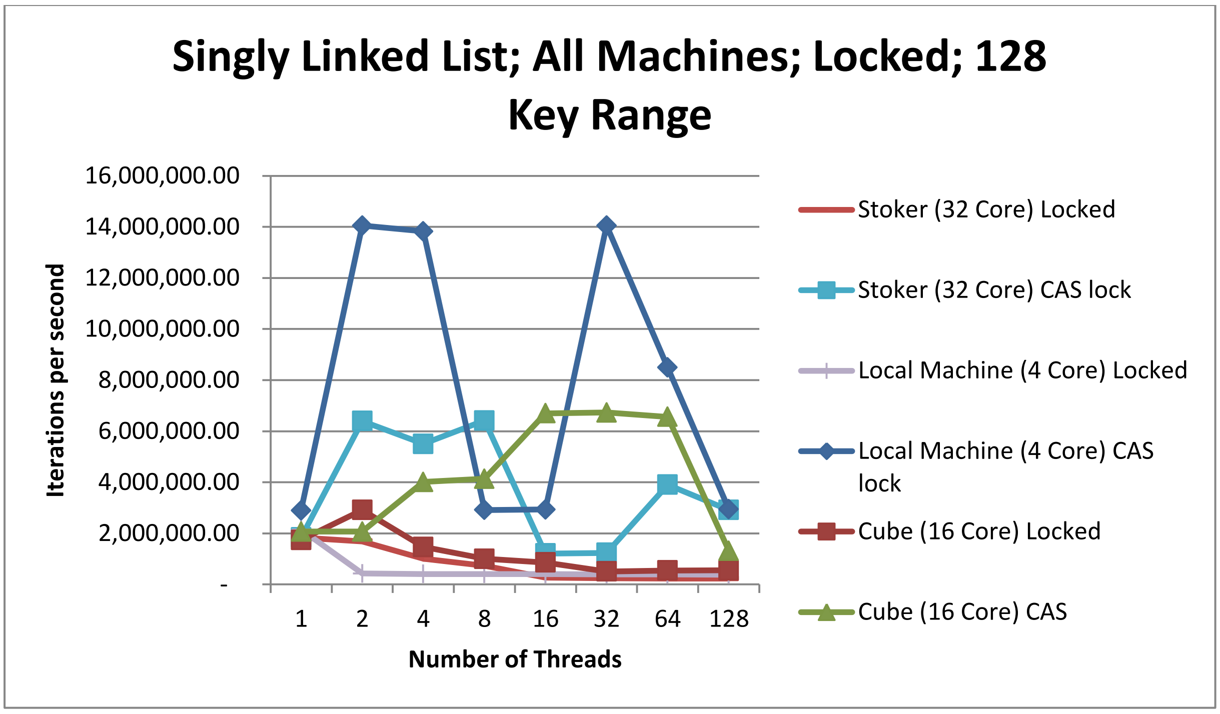
For the singly linked list I decided to vary it by changing the maximum size of the list to investigate if it had any effect on the performance of the locked and lockless algorithms. This was represented by the variable KEY_RANGE which I used with the modulo operation and the rand() function [reference] to produce key value for the nodes in the list. Since this list was ordered and there were no duplicates allowed, the value of KEY_RANGE was the largest value a node could have and since no nodes would be generated with a higher value, this acted as a hard cap on the maximum length of the list. I initially set out to test the list using the values 100, 100,000 and 1,000,000,000, however, as mentioned previously, to minimise the cost of calling the modulo operation so often I changed them to powers of two, namely 128 (2^7), 131072 (2^17) and 134217728 (2^27) so that the compiler would replace the modulo calls with a bitwise AND [reference] to minimise the performance impact.

4.3.1.1 Results & Analysis

4.3.1.1.1 Locked Comparison

Like with the ring buffer, I started evaluating the singly linked list with a size of 128 which was the maximum number of nodes allowed in the list. Again, as with the ring buffer I started off by comparing the locked modes of operation, though since both the locked and lockless modes were Multi Producer Multi Consumer I would then now be able to do a direct comparison between locked and lockless.

From the results gathered, the Compare-and-swap lock (CAS) had the best performance across the three machines as seen in the graph below where it is compared to the pthread mutex lock:



As seen, the three pthread mutex locks, while beginning around the same area as the CAS locks, the 2 million iterations per second mark, then fall into a lump. In comparison the CAS lock does much better, though even it falls off sharply when the thread count reaches 128. Below is the relevant hardware performance counter data gathered from stoker:

Counter	CAS Lock	Pthread Mutex Lock
Cycles	83,301,604,852	2,377,759,288,313
Ratio of cache references to misses (%)	73.16665369	39.75248062
Ratio of branches taken to branch misses (%)	0.264608023	0.177030999
Ratio of frontend cycles to stalled cycles (%)	64.26524113	94.59224774

From the table we can see that the mutex lock had a far lower rate of cache misses than the CAS lock with around half as many branch misses and almost 29 times more CPU cycles. However, the mutex lock blocks near 95% of those cycles, essentially wasting them while the CAS lock does a much better job of utilising its CPU time with only 64% of its CPU cycles wasted.

As I increased the size I noticed that the performance of the locks was dropping significantly, whereas as the iterations per second had been in the millions, it was now in the thousands. After going back and analysing my implementation I realised that it was due to the expanding size of the linked list. Due to the ordered nature of the list, every time a node needed to be added or removed the thread had to search the list. The longer the list was, the longer the search took and hence fewer nodes were added or removed, leading to a significant drop in performance.

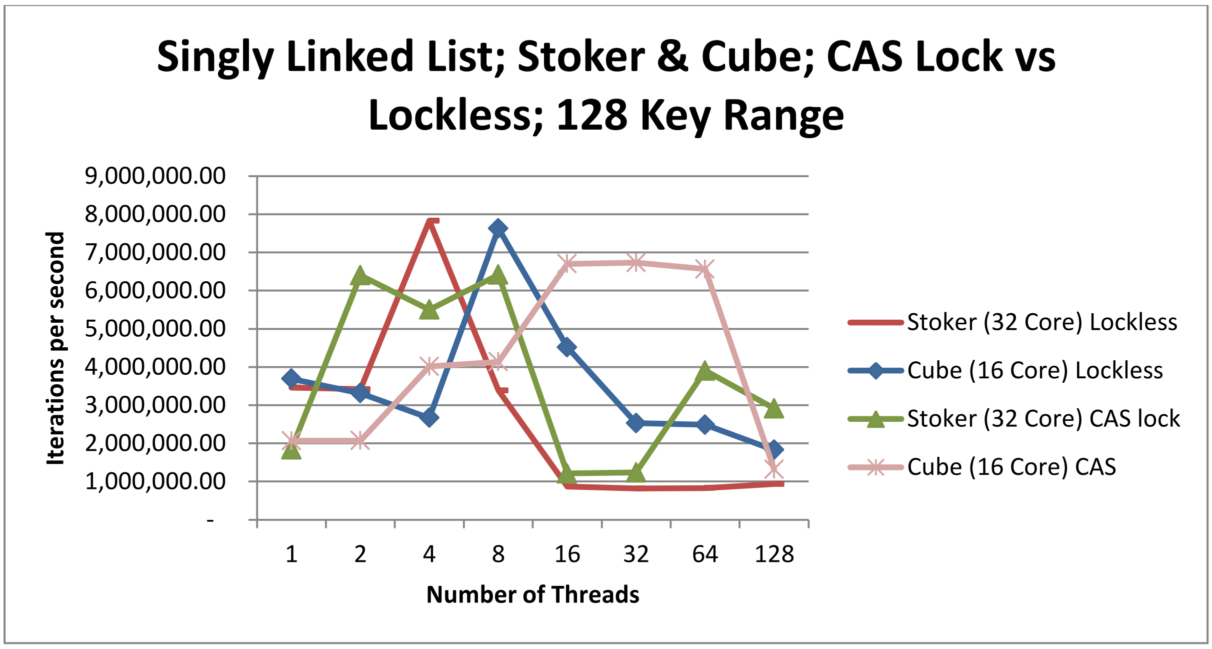
4.3.1.1.3 Lockless Comparison

For the lockless comparison I started with a size of 128 but as I increased the size of the list, as with the locked modes, the number of iterations being completed dropped sharply as the threads spent their time looking for nodes as opposed to adding or removing them.

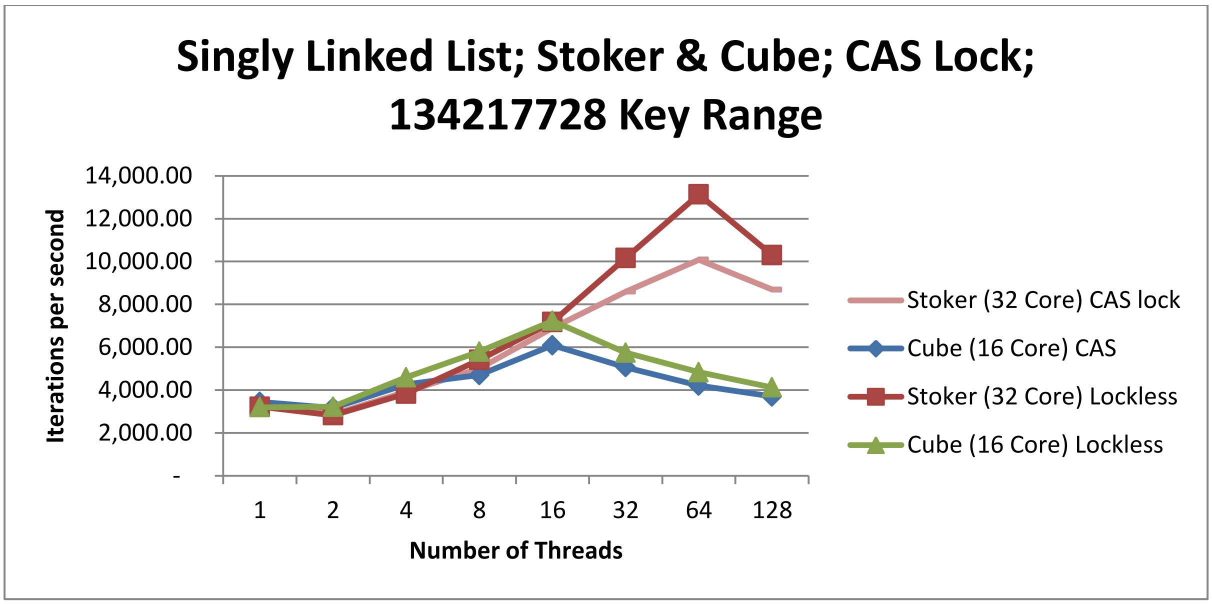
For this variation, the local machine did surprisingly well, outperforming both Stoker and Cube, with performance staying level even at high thread counts.

4.3.1.1.4 Locked vs Lockless Comparison

The lockless implementation performed well against the locked modes with a size of 128, though the CAS lock performed equally as well, beating the lockless implementation at the higher thread counts, though they both dropped sharply at 128 threads. The lockless implementation performed very similarly on both Stoker and Cube, rising sharply on the lower thread counts of 4 and 8 and then trailing off as the thread count increased.



With a higher size of 134217728 a different picture emerges when we compare the two again, both the CAS lock and the lockless implementation rise continually, reach a peak and then slump. However, the lockless implementation beats the CAS lock at every point this time, unlike with the smaller size.



4.3.2 Doubly Linked Buffer

4.3.2.1 Evaluation

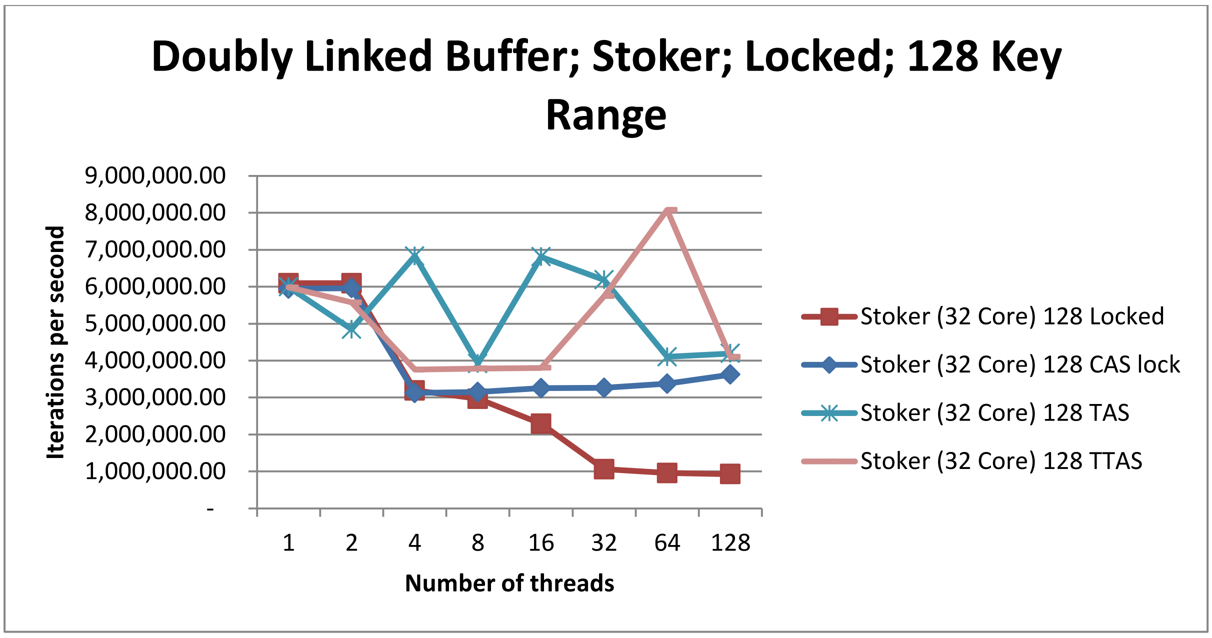
This variation of the linked list differed from the singly linked list due to the fact that this list is neither ordered nor does it prevent duplicates from being added to the list. In addition, nodes are added and removed from the head and tail respectively so that threads no longer have to spend any time searching the list, turning it into a buffer like object. This variation was implemented so that the locked and lockless versions could be compared as closely as possible, removing the randomness of the singly linked list where a thread may insert a node at the head of the list or may have to travel the full length based on the node that was randomly created.

As with before, the initial size to be tested is 128 with the size increasing up to 131072 to investigate if size impacts this version of the linked list at all.

4.3.2.1 Results & Analysis

4.3.2.1.1 Locked Comparison

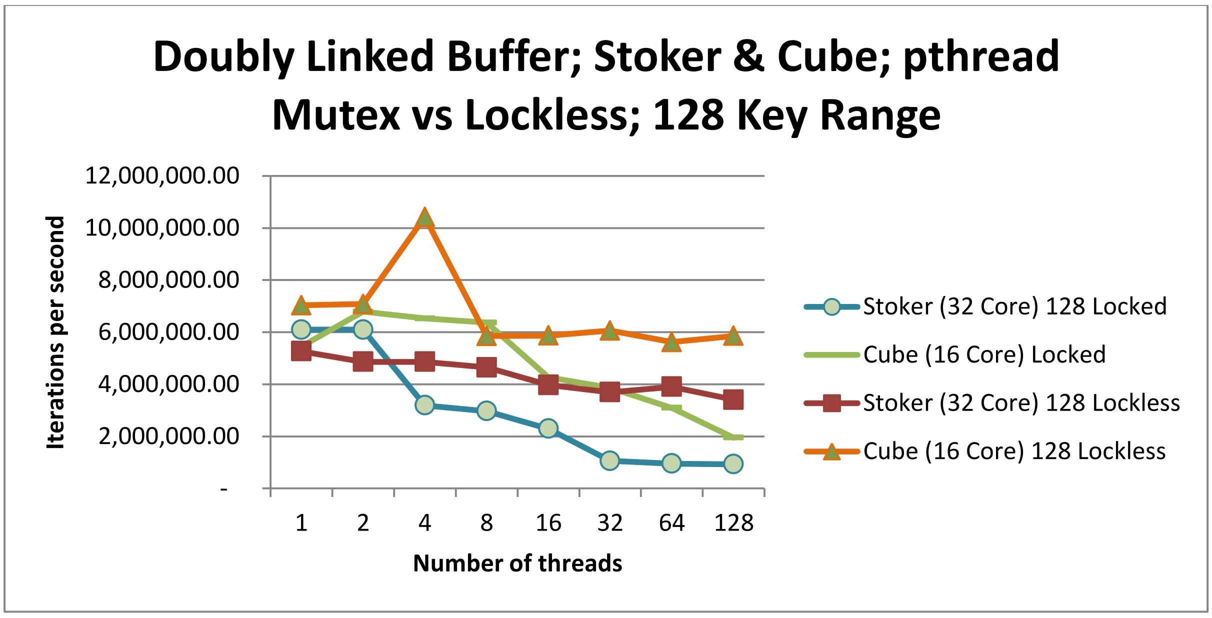
The four best locked modes of operation on Stoker were the pthread Mutex Lock, CAS lock, TAS and TTAS lock. In general, the TAS, TTAS and CAS lock along with their varieties did well across all machines.



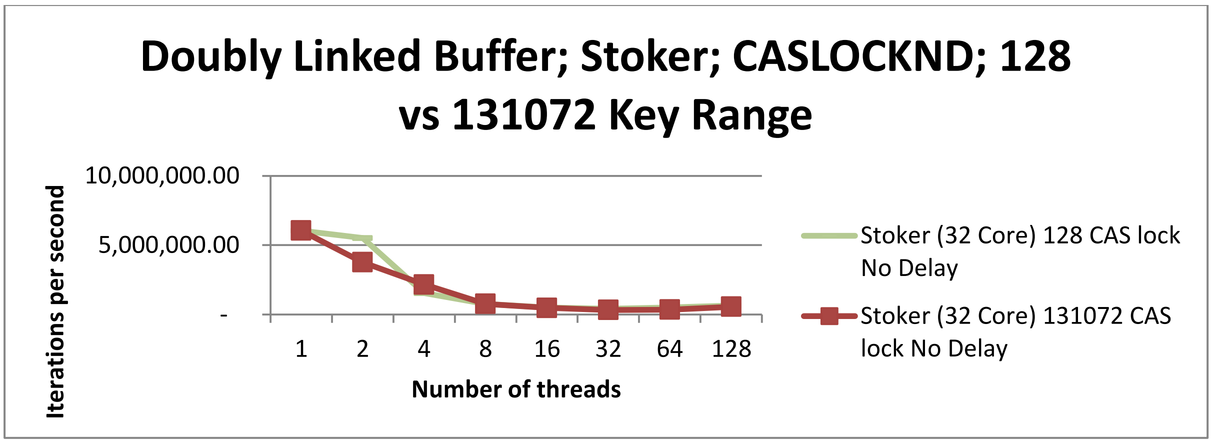
4.3.2.1.2 Lockless Comparison

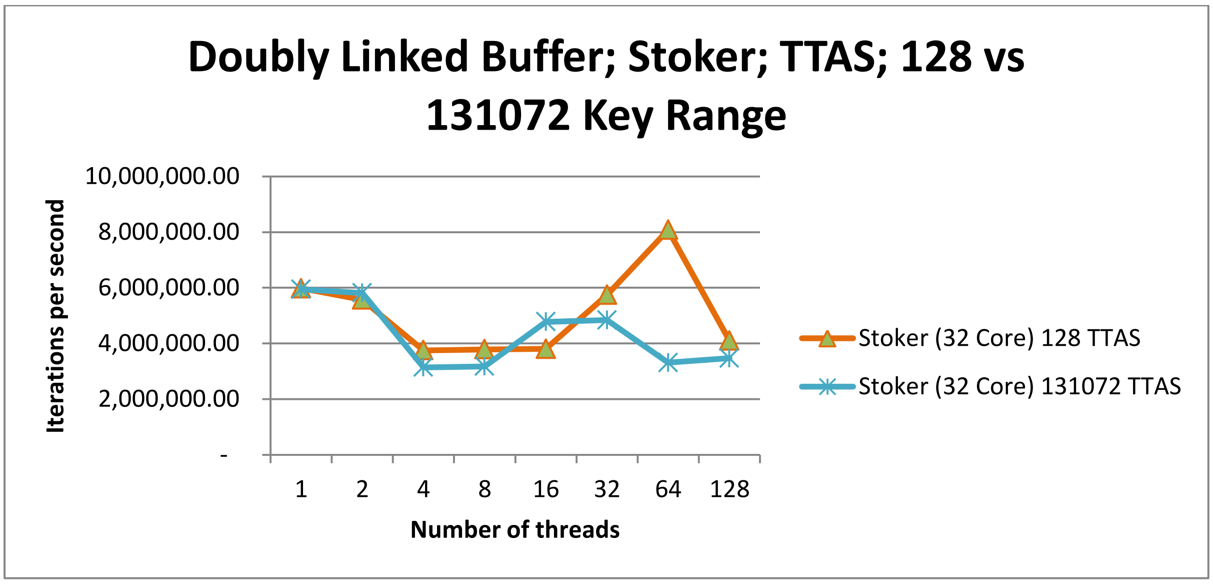
4.3.2.1.3 Locked vs Lockless Comparison

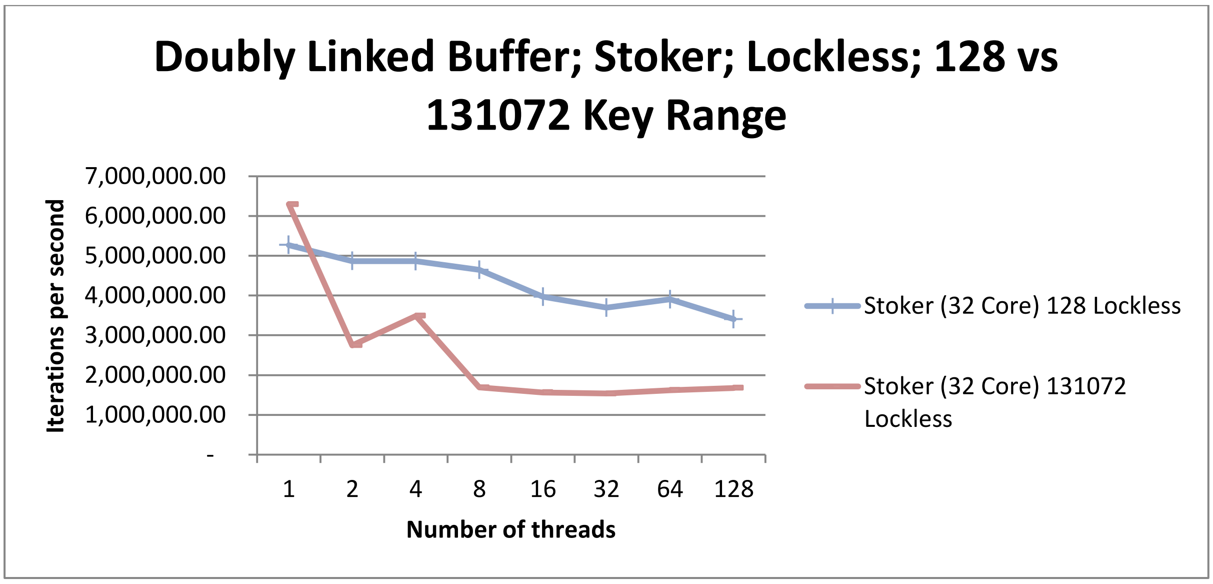
The lockless implementation did well against the locks with all machines reporting results to match or exceed the results from the best performing locks, the CAS, TAS and TTAS locks, especially around the early thread counts



I then ran the tests again, but this time used a size of 131072 to see if the size of the doubly linked buffer had an impact on the performance of the locks and on the lockless implementations:







From the above graphs it can be seen that for some modes of operation the size of the buffer makes no difference, as in the first graph comparing CASLOCKND. However in the subsequent two graphs we can see a performance difference, where the TTAS lock has a spike in performance at 64 threads and where the lockless version seems to have better overall performance than the version with the larger buffer.

4.3.3 Singly Linked Buffer

4.3.3.1 Evaluation

4.3.3.1 Results & Analysis

4.4 Hash Table

4.4.1 Evaluation

Came across an annoying issue where Cygwin on windows did not seem to be executing rand() correctly in the choose function. I had set the seed using the system time, yet the same number was always produced. My code worked fine on stoker & cube. In the end to fix the problem I created a global array, filled it with random numbers in main then got the threads in choose to pick an index of that array, this solved the issue without a noticeable drop in performance.

4.4.2 Results & Analysis

5 Afterword: (Thin)

5.1 What happened?

5.2 Lessons learnt?

5.3 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