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

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

_____ _____

Mark Gibson 23/04/2014

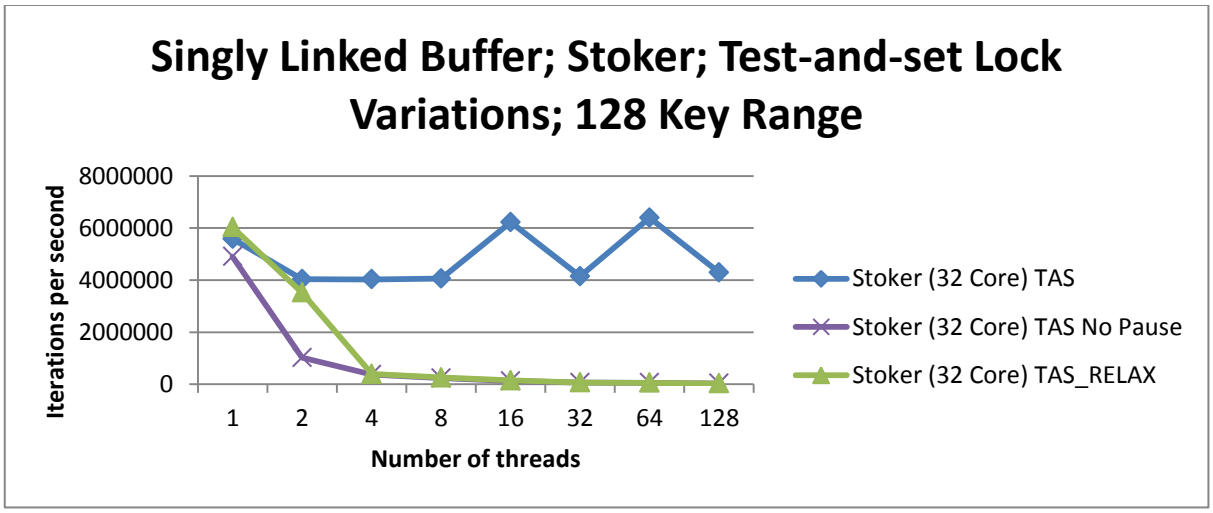
Acknowledgements:

Firstly, I would like to thank Dr. David Gregg for providing the inspiration for this project and giving me the opportunity to undertake it. This project would not have been possible without his input, support and optimism.

My second reader Melanie Bouroche for the time taken to review this project.

Fionnuala, Jo, Dave and Michael for their support through this project and my degree as a whole.

Contents



1 Introduction

1.1 My Work:

This purpose of this project is to determine and compare the differences between concurrent data structure implementations and investigate if the performance of these implementations is maintained across different architectures.

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

I have implemented each data structure with a mixture of locked and lock free algorithms. Among the locks used are a simple pthread mutex lock [Barney *et al*, 2013], a compare-and-swap lock and a ticket lock [Herlihy & Shavit, 2008]. For the lockless algorithms I use the C++ 11 atomic library [*Atomic Operations Library*, 2013] which contains the necessary atomic operations to implement lockless algorithms.

I have gathered data from these three data structures using varying thread counts and other variables, such as list or table size. The data structures are compared on a total of three different architectures to determine whether the performance of the algorithms is robust across architectures or not.

1.2 Context:

There has been much work done in the area of implementing concurrent data structures, with the field of concurrent programming expanding rapidly corresponding with the rise in multicore machines [Herlihy & Shavit, 2008]. 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 whether lockless algorithms are always preferred over locked alternatives. Hence, this project hopes to shed some light on the area by taking three concurrent data structures and testing them with several locking strategies to deduce if there is any correlation between certain algorithms and the relevant data structure's performance.

2 Background & Literature Review

2.1 What Motivated You?

I had been introduced to the idea of concurrency in the third year of my degree and it piqued my interest. The solutions that concurrency provided for such computing problems as the memory and power wall seemed quite elegant to me. I saw the potential that this technology had and 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 when it came to choosing a project for my final year, I decided to combine my new found interest in concurrency with data structures.

The problem that presents itself is that there seems to be little data comparing the performance of concurrent data structures using different locking algorithms. There is plenty of work done with regards to designing concurrent data structures [Moir et al. 2001] and implementing them [Herlihy 1993], but when it comes to practical implementation I have a difficult time in finding relevant work done in this area.

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. Locks works by allowing only one thread to 'own' it and only the thread who owns the lock can access the resource. While this is a convenient way of ensuring mutual exclusion, it does not scale with an increased amount of computing power or threads, as only one thread can access the resource at any given time [Herlihy & Shavit, 2008].

The term *starvation* when referring to multithreaded applications is the situation where a process is perpetually denied resources and as a result will never complete its assigned task, this can take place due to a poorly designed scheduler or in our case where several threads are competing for a lock. If a thread can never acquire a lock then it will never complete its task and so is subject to starvation [Herlihy & Shavit, 2008].

The term *lockless* in computer science when referring to a non-blocking algorithm equates to an algorithm where threads that are competing for a shared resource do not have their execution indefinitely 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 [Herlihy & Shavit, 2008].

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. Every operation has a bound on the number of steps the algorithm will take before the operation completes. It is for this reason that all wait free algorithms are also lock free [Herlihy & Shavit, 2008].

An *atomic instruction* is an operation that completes in a single step relative to other threads. When an atomic instruction is performed, much like a transaction in a database, it will complete entirely in one step or will not do anything. 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 pthread mutex, to protect a shared resource used by multiple threads [Herlihy & Shavit].

A *concurrent data structure* in computer science is a data structure that has been designed and implemented for use by multiple threads. As a result they are significantly more difficult to design and verify than sequential 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 [Herlihy & Shavit].

2.3 Sources Used

2.3.1 The Art of Multiprocessor Programming [Herlihy, Shavit]

When it came to researching what work had been done before now, I initially looked towards this book. 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 and the ABA problem.

It then delves into the foundations of shared memory and the basics of multithreaded programming, detailing the spin lock and the issue of contention, where many threads vie for control of the lock. 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 can face when attempting to implement locked and lockless forms of these data structures.

Considering how closely this book follows my own work, I draw on it heavily throughout the design and implementation phases of my project.

2.3.2 Designing Concurrent Data Structures [Moir, Shavit]

This work 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, performance and verification techniques while also going into detail for a range of specific data structures, like *stacks*, *queues*, *linked lists* and *hash tables*.

2.3.3 Implementing Concurrent Data Objects [Herlihy]

This book goes through the process of implementing a concurrent data structure, highlighting the issues with the conventional techniques of relying on critical sections. Instead he suggests using a lockless approach, going into detail on the differences between lock-free and wait-free approaches.

2.3.4 Experimental Analysis of Algorithms [Johnson]

This paper 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 using efficient implementations to ensuring comparability. In addition, he also goes over ideas and techniques of presenting data which I found to be most interesting.

2.3.5 A Lock-Free, Cache Efficient Shared Ring Buffer [Lee et al]

This paper goes into great depth and detail for designing and implementing a high performance, concurrent ring buffer by attempting to optimise cache locality when it comes to accessing control variables used for thread synchronisation.

2.3.6 Resizable Scalable Concurrent Hash Tables [Triplett et al]

This focuses on presenting algorithms for both shrinking and expanding a hash table while at the same time retaining wait-free concurrency.

3 Method: (Fat)

3.1 What do you have to do?

My work is as follows; firstly, I 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 run these data structures and gather data on their performance. This data is 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. I run these data structures on one or two additional machines to investigate if the data structure's performance carries over to other architectures.

Finally I 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 use tools such as Perf. Perf measures hardware performance counters and records such things as idle CPU cycles, cache misses and branches taken. I use this data to analyse the performance data I gather to explain why certain locks outperform others for example [*Linux Profiling with Performance Counters*, 2013].

3.2 Approach

My approach to this project is a modular one. I select a data structure I am interested in. I then research the chosen data structure and investigate what work has been done on designing and implementing the data structure concurrently. After this, I implement the data structure before gathering data from it. I graph and analyse the data I have gathered and generate several conclusions about the data structure and the relevant locking algorithms. Once this is done I choose another data structure and so the process continues as I 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. I am implementing my locked variations as follows: I implement different locks such as pthread mutex, test-and-set etc. by using the pre-processor to define variables which impact the path of execution. For example, when I implement the pthread mutex lock I use the –D option on the g++ command line to define a macro. In the case of the pthread mutex lock this macros is "LOCKED" just as the test-and-set lock's macro is "TAS".

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 use locks and so system wide throughput is guaranteed. To ensure that the locked and lockless algorithms are equal I wrap each atomic instruction which can fail in a while loop so that if it fails then 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

Below is a list of the different locks I use throughout the project. Each heading gives the name of the lock and the macro that I use to define it is given in brackets.

3.2.2.1 Pthread Mutex Lock (LOCKED)

This lock is composed of a pthread mutex. I choose it as I think that it is a simple lock to implement. In my opinion it also provides an excellent baseline for me to test other locks against due to its simplicity.

3.2.2.2 Test-and-test-and-set lock (TTAS)

The *test-and-test-and-set* lock [Herlihy & Shavit, 2008] lock works by using the C++ 11 atomic exchange instruction 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 sleep instruction [*sleep,* 1996]. 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 it sets the lock to one, acquiring it, and enters the critical section [Herlihy et al, 2008, pg.22]. Upon finishing, the thread then sets the lock to zero, releasing it and the process continues on. I 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 Test-and-test-and-set-no-pause lock (TTASNP)

This locked mode is the *test-and-test-and-set* lock but without the sleep instruction after the second while loop. I added this as I am interested in the effect that the sleep instruction has on the performance of the lock when compared to the normal *test-and-test-and-set* lock.

3.2.2.4 Test-and-test-and-set-relax lock (TTAS_RELAX)

This is near identical to the normal *test-and-test-and-set* 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 add this variation as, like with the *test-and-test-and-set-no-pause* lock, I am curious as to how the change affects the lock's performance and if the intrinsic gives this mode an advantage over the sleep instruction.

3.2.2.5 Test-and-set lock (TAS)

I implement a *test-and-set* lock because it is somewhat less sophisticated when compared to the *test-and-test-and-set* lock [Herlihy et al, 2008, pg. 144]. It is more basic because, while the regular *test-and-test-and-set* lock tells a thread to sleep after it has failed to acquire the lock; the *test-and-set* lock does no such thing and simply allows the thread to continue polling. This can lead to a dramatic increase in the amount of bus traffic between the CPUs in the machine and therefore can result in a loss in performance when compared to the *test-and-test-and-set* lock [Herlihy et al, 2008, pg.145]. Note that for my version, the *test-and-set* lock is implemented with a *sleep()* instruction. This is to better distinguish it from its two variations, the *test-and-set-no-pause* lock which would be traditionally considered a normal *test-and-set* lock and the *test-and-set-relax* lock which replaces the *sleep()* instruction with the intrinsic *_mm_pause()*,

3.2.2.6 Test-and-set-no-pause lock (TASNP)

This is identical to the previous lock, the *test-and-set* lock except that it discards the *sleep()* instruction.

3.2.2.7 Test-and-set-relax lock (TAS_RELAX)

This lock is identical to the *test-and-set* lock, but instead of utilising a sleep instruction it uses the intrinsic _mm_pause(). I add this lock as I am curious as to how this lock compares to the *test-and-set* lock in terms of performance.

3.2.2.8 Compare-and-swap lock (CASLOCK)

The next lock I implement is a lock based on the atomic instruction, *compare-and-swap* which takes an object and attempts to change the value it has. If the object contains an expected value, then this value is replaced with the new value, else the object remains unchanged [Herlihy et al, 2008, pg.113].

This can then be placed within a loop, where threads continuously poll until one of them acquires the lock successfully and breaks free into the critical section. However, this can generate a lot of bus traffic similar to that of the *test-and-set* lock and so I add an exponential back off, where a thread, upon failing to acquire the lock sleeps, but with each failed attempt, sleeps for a progressively longer time up to a defined maximum.

3.2.2.9 Compare-and-swap-no-delay lock (CASLOCKND)

This lock is the same as the *compare-and-swap* lock but where that lock has an exponential back-off to try and reduce bus traffic this lock has no such thing. Threads are able to constantly poll the lock when they are attempting to acquire it. As with previous lock variations, I am interested to see how the lack of a back-off impacts the performance of this lock compared to the regular *compare-and-swap* lock.

3.2.2.10 Compare-and-swap-relax lock (CASLOCK_RELAX)

This lock takes the exponential back-off present in the regular *compare-and-swap* lock and replaces it with the intrinsic _mm_pause(). This is done to compare the variations of the *compare-and-swap* lock and see how they perform compared to one another.

3.2.2.11 Ticket lock (TICKET)

The final type of lock I add is 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 once the number of threads exceeds the number of CPU cores, 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 *test-and-test-and-set* 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 lock(TICKET_RELAX)

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

3.3 Data Structures

3.3.1 Ring Buffer

I begin implementation of my project with a FIFO ring buffer. 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.

The previous implementation I decide to base mine on is located in “Designing Concurrent Data Structures”. In it they describe the design for a concurrent queue which utilises a head and tail pointer to allow for parallel execution. In it they use a dummy node to prevent deadlock. My locked implementation differs as I only use one lock to control the front and back of the queue. I do this as I want my implementation to be even simpler. The reason for this is that the ring buffer is more of a testing stage where I can easily implement my different locking methods and test them out. At the end of my project if I have enough time I will come back to the ring buffer and implement a more advanced locking algorithm but for now this I what I need.

My lockless implementation differs as I do not use a dummy node, but instead the producer and consumer threads look ahead to see if the next node is being used.

At this point in the project I consider implementing assembly versions of the locks I have. http://locklessinc.com has several implementations in assembly such as a *ticket* lock and *test-and-set* lock. I attempt to implement the assembly locks, which is successful. I now compare the assembly locks against the C++ locks I already have. I discover that the difference in performance is negligible and so I decide to stay with my C++ implementations as I feel more comfortable using and modifying them.

3.3.1.1 Locked

For the locked implementation of the ring buffer I choose a simply locking strategy. A thread wishing to interact with the buffer must first acquire a lock; all interactions with the buffer are mutually exclusive. Upon performing its operations, a thread would then release the lock; this approach allows only one thread to interact with the buffer at any given time.

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 implement a singly linked list. I choose a linked list because I have experience with implementing and testing linked lists both sequentially and concurrently.

The implementation I choose to base mine on is found in "The Art of Multiprocessor Programming. It is simple and it follows conventional designs which I am already familiar as mentioned previously. My lockless implementation does diverge slightly however, while Herlihy & Shavit use a *find* function to obtain the necessary details to add or remove a node I instead choose to implement this step as part of the *add* or *remove* functions due to problems I have encountered in the past with using pthreads and calling nested functions.

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 observe 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

I choose a hash table as my third and final data structure to implement because it is a data structure that I have always had an interest in. In addition, I will be able to utilise my work on linked lists by using them to represent the buckets in my hash table.

As to which implementation I am to base mine on, I again choose one from "Designing Concurrent Data Structures" by Moir and Shavit. In it they describe the design of a concurrent hash table which uses lockless linked lists as buckets, exactly how I wanted to design mine. I also draw from "Reizable, Scalable, Concurrent Hash Tables" by Triplett et al when I implemented the resize function.

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. To ensure that each iteration of the program is equal I create the hash table at the start of each iteration and delete it at the end so that each iteration works with the same data structure.

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 focus on other aspects of my project with the intention of coming back to the issue.

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 frequently 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.

A third 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 ran into segmentation faults immediately 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 encountered a problem with the structure of my program. 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. A better solution was needed. 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. The reason for this is that the compiler will convert the modulo 128 operation into a bitwise AND which is far less computationally expensive.

4 Experiments & Evaluation: (Fatish)

4.1 Evaluation Strategy

To analyse the data that I gather from the three data structures I implement in this project I am following the strategy outlined below.

Firstly, I 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 allows me to graph the performance of each algorithm as the number of threads being generated increases. I have decided on 128 as being the maximum number of threads spawned as I originally thought that this was twice the number of core stoker has which would be 64. However, I now know that stoker only has 32 cores, though I have decided to stick with the 128 figure.

To measure the iterations per second I have to first record how long the program takes to finish. The system time is retrieved at the start of the main function in each program; each thread is then created and run. Once all the threads are finished the system time is gotten again, this is the stop time. The start and stop time are then used to calculate the running time and this is then divided by how many seconds each thread is allowed run for which produces the iterations per second for each algorithm.

I have chosen one second to be the length that each thread should run for. I chose this amount of time as this is the case for many experiments of this type [reference] and makes the calculation for iterations per second trivial.

Each program starts by generating one thread which then works until one second has passed. The thread then terminates and the program restarts and generates two threads.

This process repeats, the thread count doubling every time until 128 threads are spawned at once. They then work for one second, at which point the program finishes.

In addition to the iterations per second and thread count I am varying the size of the different data structures to investigate whether the size of the relevant data structure plays a role in the performance of the locked and lockless algorithms.

To ensure that I am collecting accurate data, I am making sure to only collect data from each machine when CPU load is low so as not to jeopardise the data. In addition, while I initially collected data by running each algorithm only once I felt the variance between the different iterations, while small, was not negligible so I changed my method to instead run each algorithm 7 times and to get the median of each data set produced.

I have chosen the median over the average as it gives a better representation of the data [reference]. I calculate the median for each algorithm after it has run 7 times before moving onto the next algorithm. I ensure that the calculation of the median does not impact on the performance of the algorithms as it is done outside of the timed sections of the program.

To speed up the time it takes to gather results I have written several bash scripts to automate the defining of the different locked modes of operation. Without them I would need to manually enter each program and define/undefine each mode of operation each time I run the tests and that would take up a lot of time. The script compiles the program multiple times, defining a different mode of operation each time. I do this by using the –D option in the g++ compiler. It then runs the code which prints out the data before moving onto the next mode.

To further increase the quality of my implementations I use the –O3 flag on the g++ compiler to turn on several optimisations supported by the compiler [reference].

All code is written in C++ and compiled using g++ 4.7.2.

4.1.1 System Overview

4.1.1.1 Stoker

Stoker is a multicore machine owned by the School of Computer Science and Statistics. It has four processors, each of which has eight out-of-order, pipelined, superscalar cores. Each of these cores has two-way simultaneous multithreading [reference].

The architecture s Intel Ivy Bridge EX 22nm and each of its 32 cores runs at 2.00 GHz [reference].

4.1.1.2 Cube

Cube is a multicore machine owned by the Internet Society in Trinity College Dublin. It has 8 processors, with each using two-way simultaneous multithreading. [https://wiki.netsoc.tcd.ie/index.php?title=Cube].

The architecture is Gainestown 45nm and each of its 16 cores runs at 2.27 Ghz [reference].

4.1.1.3 Local Machine

Local Machine is a multicore machine owned by myself, Mark Gibson. Its architecture is Sandy Bridge 32nm. It has four cores running at 3.30 Ghz each [reference].

4.1.2 Hardware Performance Counters

As mentioned previously I am using hardware performance counters as part of my evaluation of the data structures to help me determine why certain implementations perform the way they do. Below are some of the counters I am using and what they record:

Cycles records the number of CPU cycles used by the program.

Cache References records the number of times that the cache was referenced during the execution of the program.

Cache Misses records the number of times that the cache was referenced but returned a cache miss [reference]. I commonly use this as a ratio, depicting what percentage of cache references reported misses.

Branches Taken records how many branches were taken during a program's execution [reference].

Branch Misses records how many branches were not predicted successfully. Used in a similar way to cache misses to provide a ratio of how many branches were misses out of all that are taken.

Stalled Frontend Cycles records how many CPU cycles were wasted in the first stages of the CPU's pipeline, the fetching and decoding of instructions. Stalls happen when the pipeline is waiting for a value to be read from memory among other things [reference].

Stalled Backend Cycles records how many CPU cycles were wasted in the final stages of the CPU's pipeline, the execution of instructions. Stalls happen when the pipeline is waiting for a value to be read from memory among other things [reference].

4.2 Ring Buffer

4.2.1 Evaluation

Apart from the iterations per second and thread count I vary the size of the buffer to investigate how this affects the locked and lockless variation. The starting size of the buffer is 128; I initially had this at 100 however I decided to change it to be a power of two to minimise the effect that the modulo operation may have on the program's performance since the compiler reduces it to a bitwise AND operation [reference].

4.2.2 Results & Analysis

4.2.2.1 Lock Comparisons

I begin my evaluation by first focusing on the performance of the different locks before moving onto the lockless. Since the lockless implementation is *single-producer-single-consumer* and the locked implementations are *multiple-producer-multiple-consumer* I am unable to compare the two thus I am interested in how the locks perform in relation to each other.

While I was performing my tests I observed that, the *compare-and-swap-no-delay* lock and the *compare-and-swap-relax* lock began exhibiting issues where they would randomly throw a segmentation fault. This took me by surprise as no such issues had arisen during implementation. To attempt to fix the problem I added several memory barriers to the code and removed the –O3 flag when I was compiling the code but to no avail. As I did not have the time to delve into the problem further I had to abandon the two locks for the remainder of the ring buffer evaluation.

The Graph below represents the best four locks; 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 locks have the best performance consistently across the three machines.

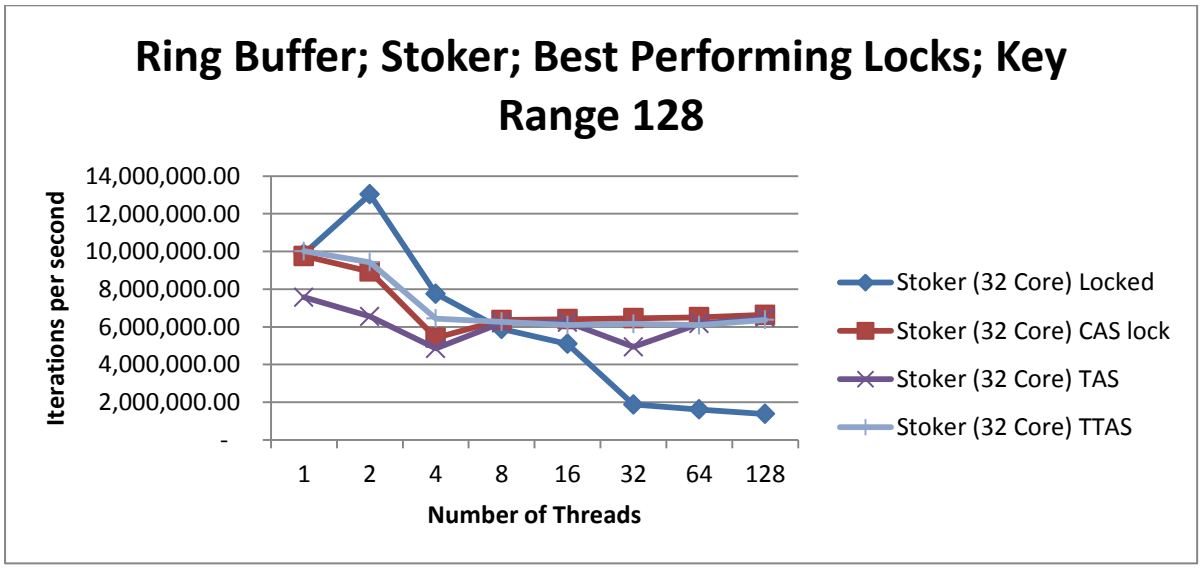


Figure 1

To determine why these four locks do quite well I compare two of them, the *pthread mutex* lock and *test-and-test-and-set lock* against a lock that does not perform as well, the *test-and-set-relax* lock which is similar to the *test-and-set* lock but it has a cpu_relax instruction after each thread attempts to acquire the lock in an attempt to reduce bus traffic. The hardware performance counter data is as follows:

	pthread mutex	*test-and-test-and-set*	*test-and-set-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 *test-and-set-relax* has a ratio of cache misses to cache references of over twice that of both the *pthread-mutex* lock and the *test-and-test-and-set* lock. In addition the lock has five times more branch misses with regard to the total amount of branches it took and has a greater ratio of misses for front and backend cycles. From this it can be seen why the *test-and-set-relax* lock does so badly while the *pthread-mutex* lock and the *test-and-test-and-set* lock do relatively well when compared to it.

4.2.2.2 Lockless Comparison

Since the lockless ring buffer implemented is a *single-producer-single-consumer* data structure it cannot be compared to the locks. The implementation only spawns two threads in total, one to push items onto the queue and one to pop the off. As a result the lockless implementation can only be compared to itself. For this I run it across the different architectures with different sizes to see how it performs. From the resulting table below we can see that the size of the buffer has a minor impact on the performance of the lockless implementation:

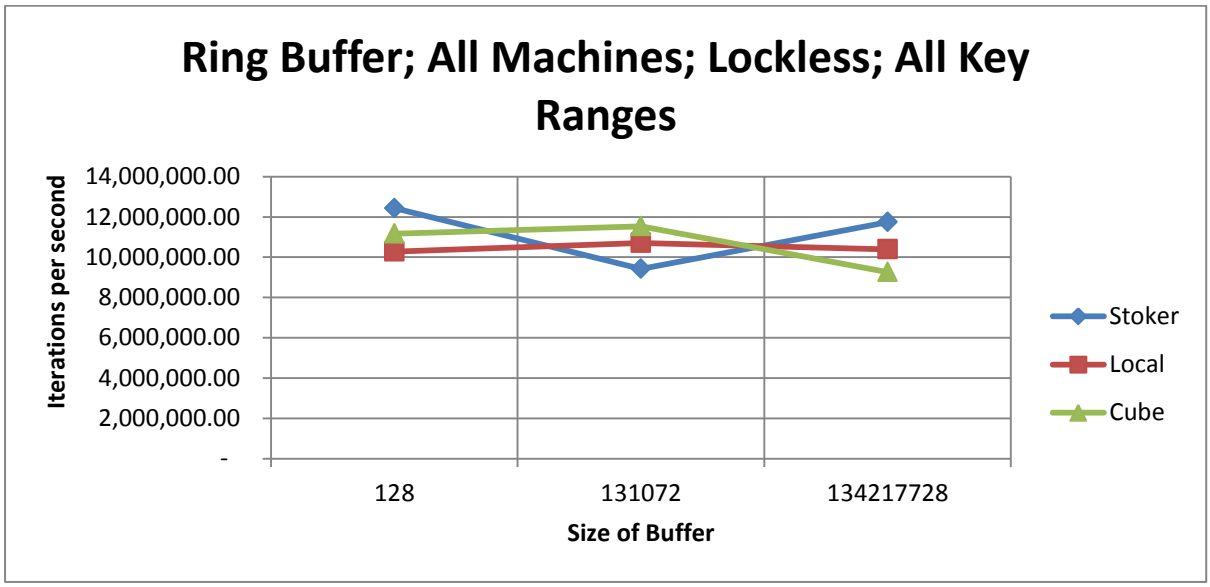


Figure 2

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 has the best performance at the first and last size it is by no means a clear winner in terms of performance.

4.2.2.3 Test-and-test-and-set Variations

I want to investigate the relative performance of the different *test-and-test-and-set* locks I have implemented. I have graphed the three variations below. It can be seen that the *test-and-test-and-set* lock using a sleep instruction has the best performance of the three. The *test-and-test-and-set-no-pause* seems to have a slight performance boost when the thread count is two while the *test-and-test-and-set-relax* lock is inferior for all thread ranges.

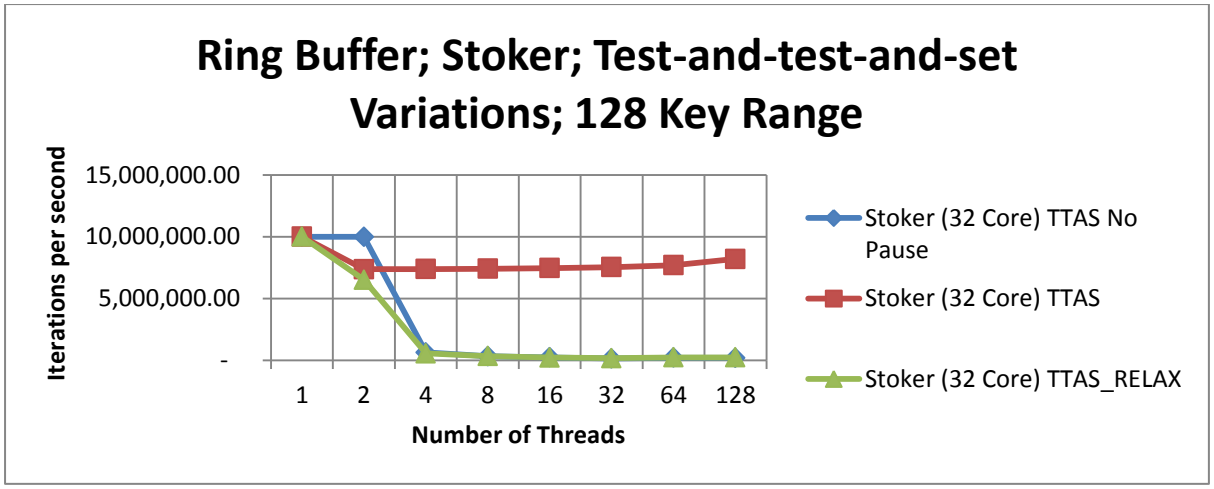


Figure 3

Analysing the data I gather from the hardware performance counters the reason for the regular *test-and-test-and-set* lock is clear, as can be seen from the table below:

Table 1

	Cycles	Cache References	Cache Misses	Stalled Frontend Cycles
Test-and-test-and-set	7.92E+10	1.39E+07	4.71E+06	5.41E+10
Test-and-test-and-set-no-pause	2.39E+12	3.75E+08	2.34E+08	2.21E+12
Test-and-test-and-set-relax	2.41E+12	3.09E+08	1.99E+08	2.35E+12

From the table it can be seen that the regular *test-and-test-and-set* lock uses far less CPU cycles than the other two variations. In addition, it has far fewer cache references, and a a lower number of cache misses than the other two variations . The *test-and-test-and-set-no-pause* lock has the largest number of cache misses which is likely due to its constant polling, which constantly invalidates cache lines, causing more bus traffic and more misses as a result [reference]. Finally the *test-and-test-and-set* lock has the lowest proportion of stalled frontend cycles meaning that it wastes the fewest CPU cycles. From the data above it can be clearly seen why the regular *test-and-test-and-set* lock has the best performance out of the three variations.

4.2.2.4 Test-and-set Variations

I turn my attention now to the *test-and-set* lock, of which I have implemented three variations. These are the *test-and-set –no-pause* lock, the *test-and-set* lock and the *test-and-set-relax* lock. Below is a graph showing their relative performances:

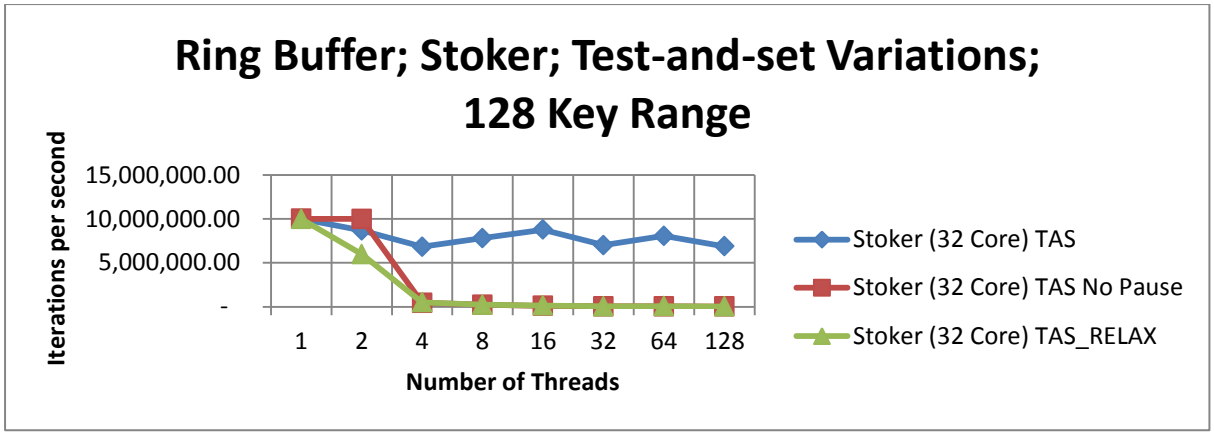


Figure 4

It can be seen from the graph that the *test-and-set* lock is the best performing lock once the thread count reaches four and onwards. Below is data from the hardware performance counters:

	Cycles	Cache references	Cache Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Test-and-set	8.98E+10	6.51E+06	1.33E+06	5.93E+10	4.50E+10
Test-and-set-no-pause	2.66E+12	2.05E+08	1.84E+08	2.65E+12	2.58E+12
Test-and-set-relax	2.67E+12	2.25E+08	2.07E+08	2.65E+12	2.48E+12

The *test-and-set* lock utilises far fewer CPU cycles and has a far lower proportion of stalled cycles both on the front and backend. Both the *test-and-set-no-pause* and *test-and-set-relax* locks utilise their CPU cycles extremely poorly, with both locks having 95% and over stalled cycles. Couple this with a far lower proportion of cache misses and it is clear why the *test-and-set* lock comes out on top in terms of performance out of the three locks.

4.2.2.6 Ticket Lock Variations

The final lock I wish to investigate for the ring buffer is the *ticket* lock and its variation, the *ticket-relax* lock. It is known that ticket locks perform poorly once the number of threads exceed the number of cores and I want to confirm that and see if the *_mm_pause()* intrinsic affects this in any way. Below is the graph detailing the two locks' performance:

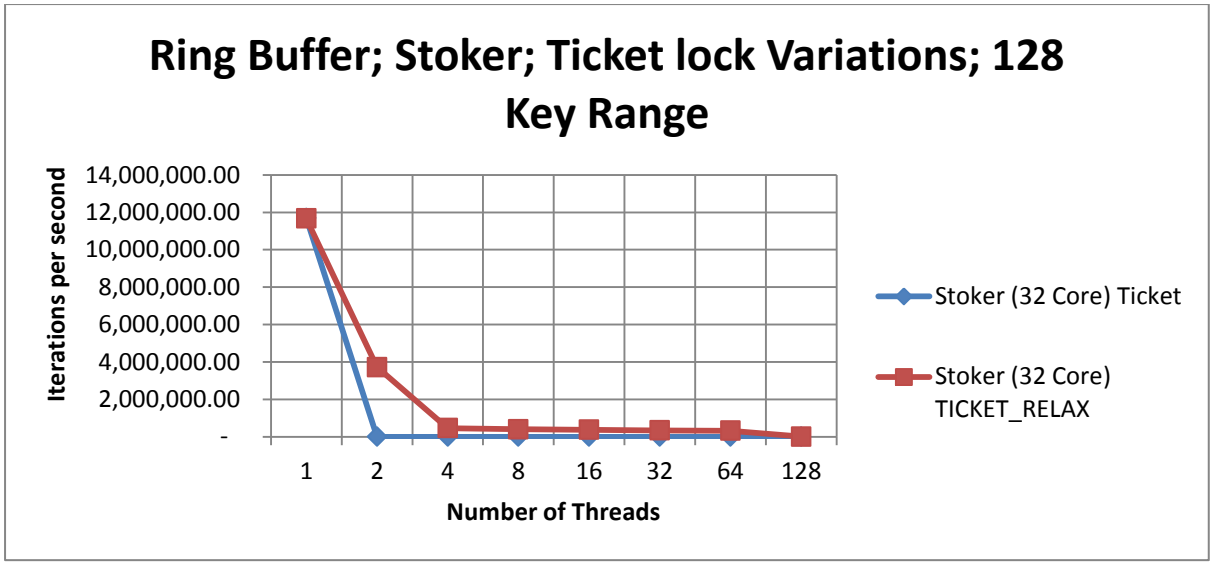


Figure 5

From the graph it shows that both locks drop sharply in performance, though somewhat earlier than expected considering that Stoker has 32 cores. The hardware performance counter data can be seen below:

4.2.2.7 Does size affect performance?

It has already been shown in the *lockless* comparison that the size of the buffer has little to no effect on the performance of the *lockless* implementation. Hence, I will now evaluate the *pthread mutex*, *test-and-test-and-set* lock and the *compare-and-swap* lock to investigate if this property carries over to the locked implementations.

From the three graphs below it is shown that while there are some performance differences between the buffer sizes used, it is inconclusive as to whether or not the size of the buffer directly affects the performance of the implementation.

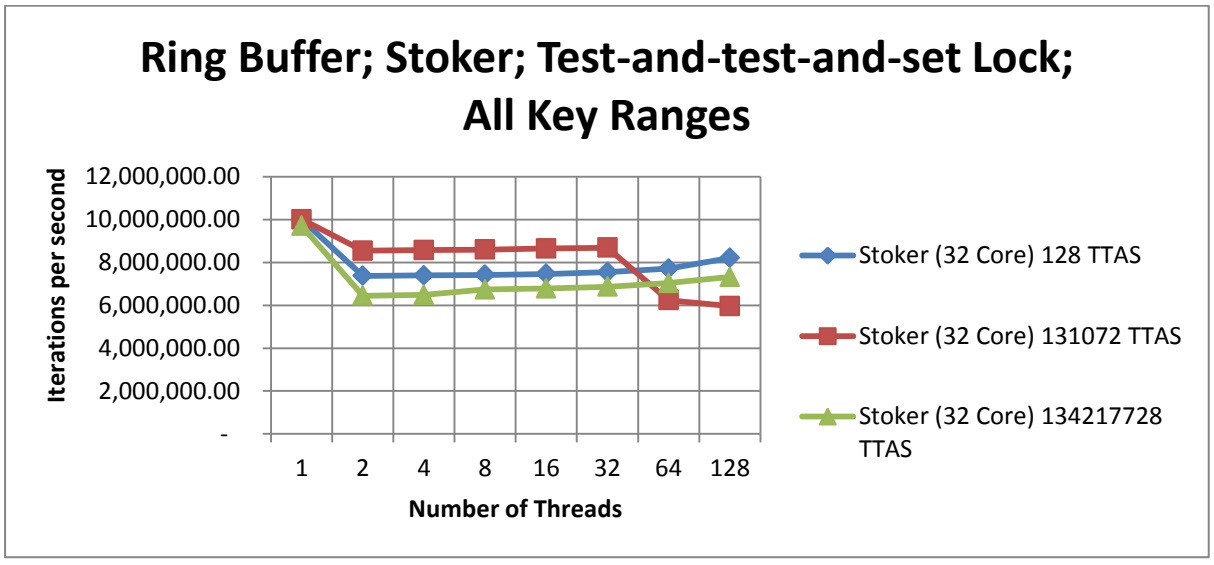


Figure 6

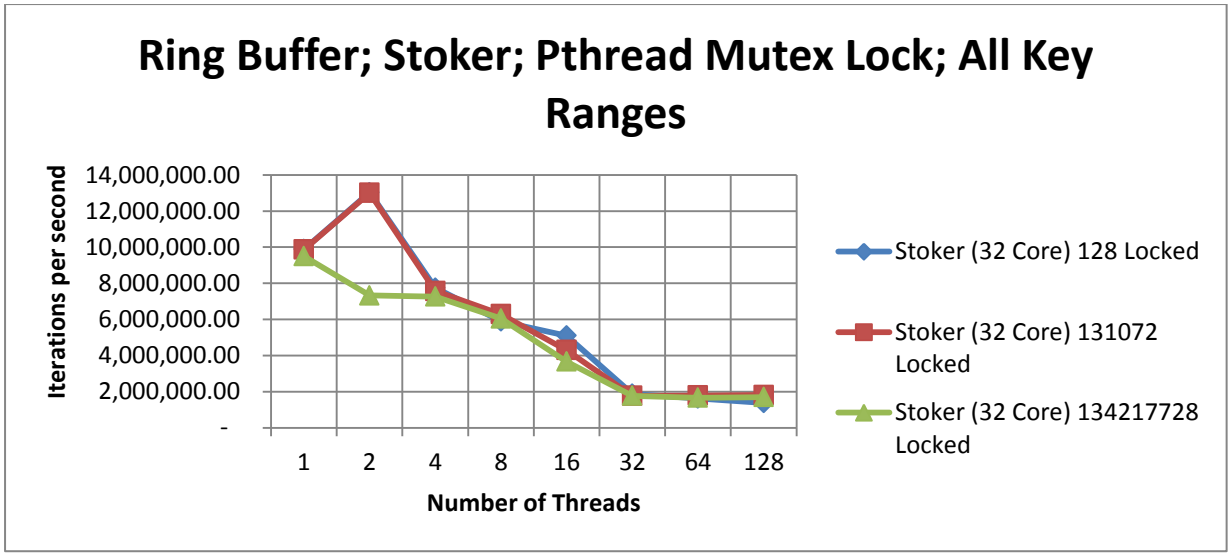


Figure 7

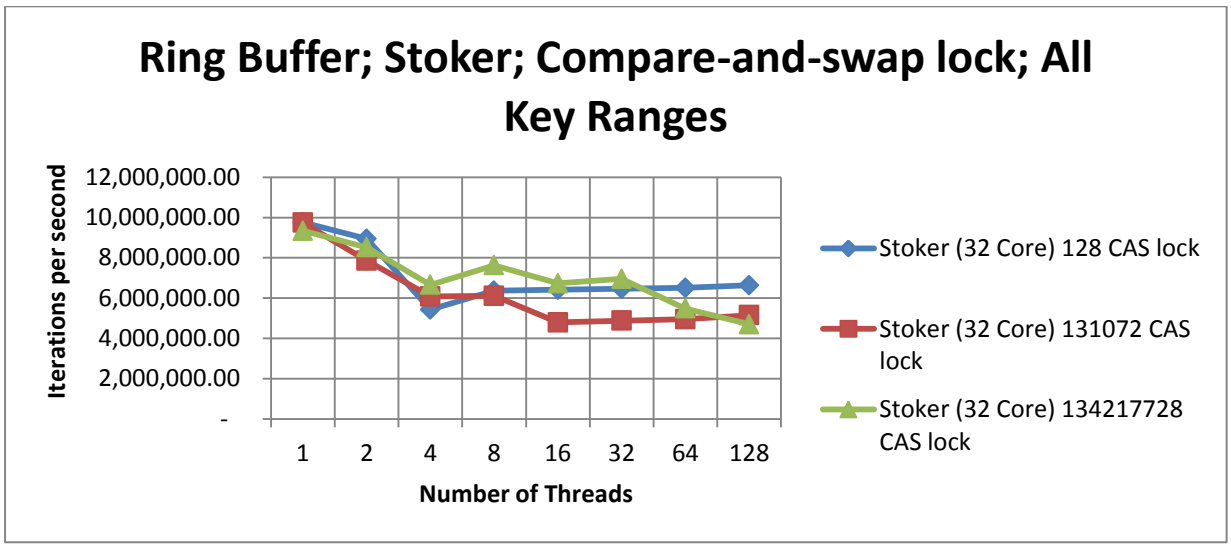


Figure 8

4.2.2.8 Is the ring buffer robust across architectures?

For the final piece of evaluation for the ring buffer I will now investigate whether or not the locked and lockless implementations maintain their relative performances across multiple architectures. The *pthread mutex* lock, *test-and-set* lock and *compare-and-swap* lock implementations will now be run across all three architectures. Below are a series of graphs, each representing one lock over the three architectures of Stoker, Cube and Local.

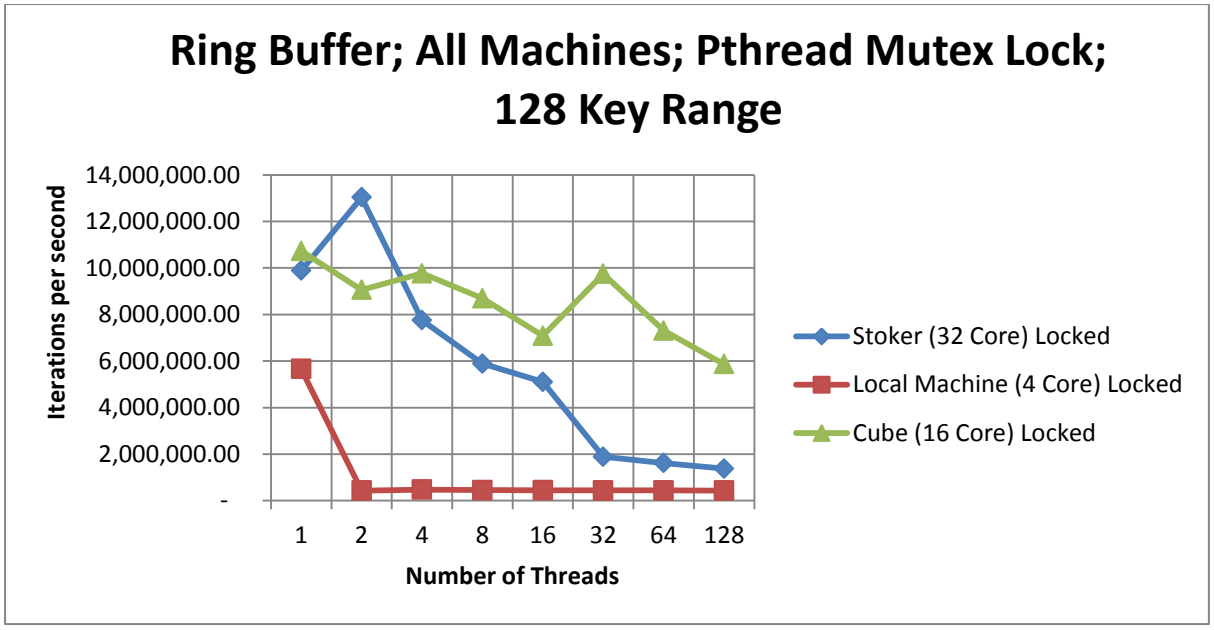


Figure 9

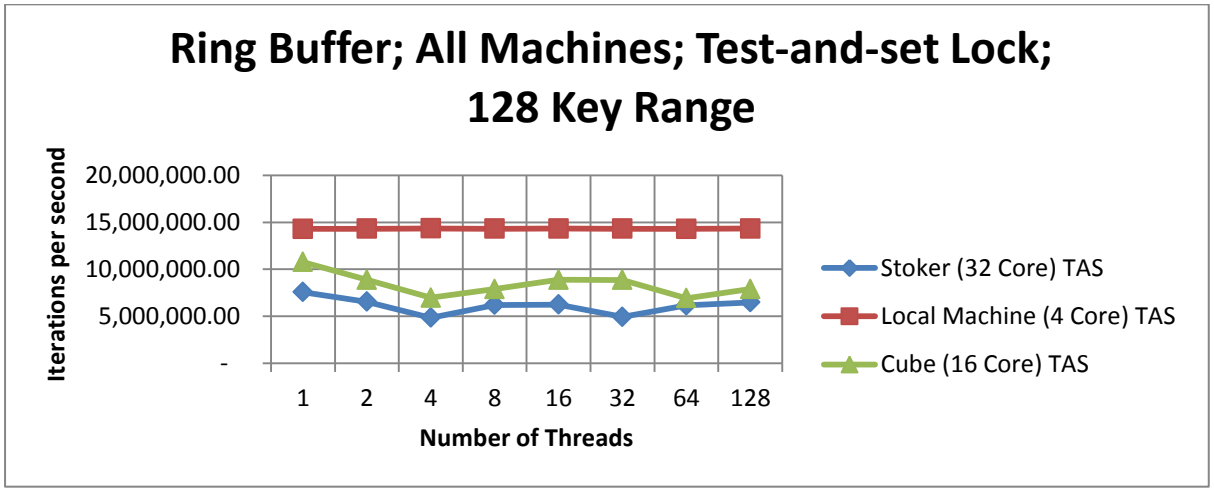


Figure 10

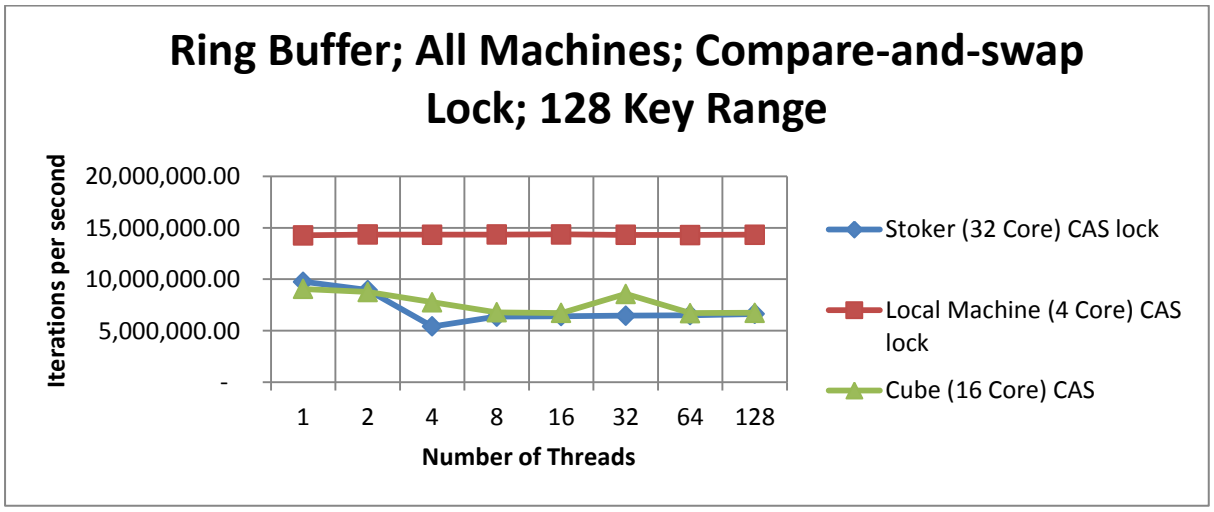


Figure 11

From the graphs above it shows that some locks are more robust across architectures than others. For example, the *test-and-set* lock's performance has a similar shape across the three architectures while the *pthread mutex* lock is quite different on the three architectures.

4.3 Linked List

4.3.1 Singly Linked List

4.3.1.1 Evaluation

For the singly linked list I vary it by changing the maximum size of the list to investigate if it has any effect on the performance of the locked and lockless algorithms. This is represented by the variable KEY_RANGE which I use with the modulo operation and the rand() function [reference] to produce key values for the nodes in the list. Since this list is ordered and there are no duplicates allowed, the value of KEY_RANGE is the largest value a node can have and since no nodes are generated with a higher value, this acts 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 will replace the modulo calls with a bitwise AND [reference] to minimise the performance impact of the instruction.

Finally to ensure that each thread count's seven iterations were as similar as possible, the head pointer is set to *null* at the end of each iteration, effectively deleting the list so that later iterations are not presented with a fuller list than the earlier iterations.

4.3.1.1 Results & Analysis

4.3.1.1.1 Locked Comparison

I start evaluating the singly linked list with a maximum allowed size of 128 nodes. Again, as with the ring buffer I start off by comparing the different locks against each other.

Again, both the *compare-and-swap-no-delay* and *compare-and-swap-relax* have run into issues, throwing segmentation faults where there was no issue before. Not having the time to discern why I exclude them from the rest of the evaluation.

Out of the locks it is the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* which perform the best, with the *compare-and-swap* lock vastly outperforming the other two as seen below:

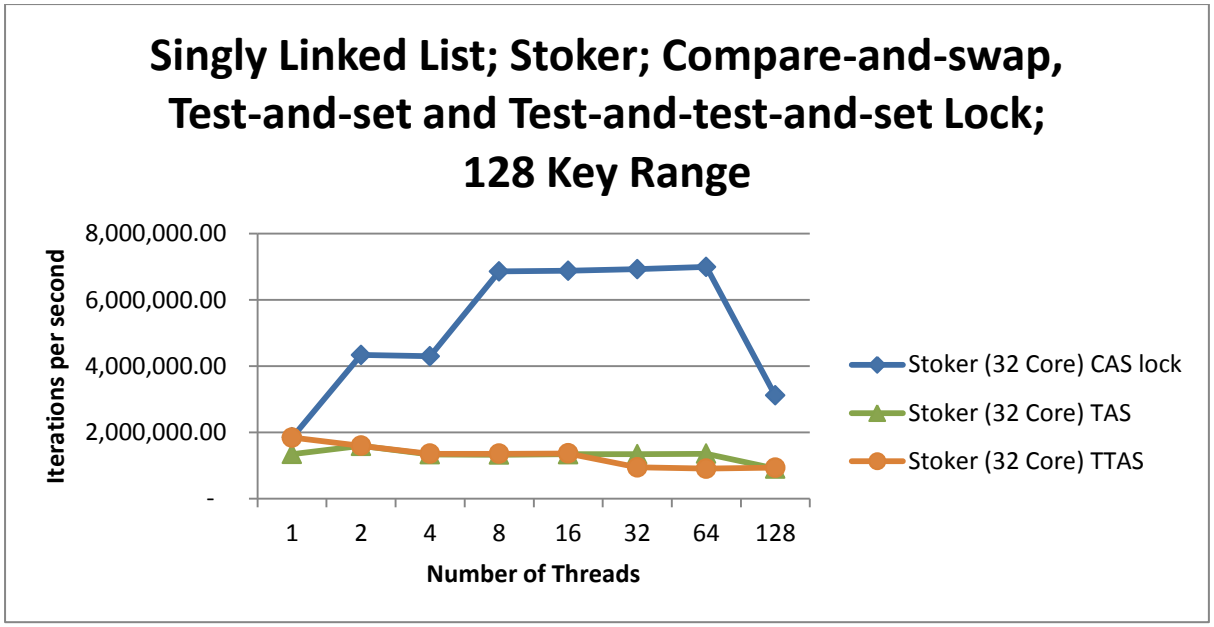


Figure 12

If the hardware performance data is examined it is clear why the *compare-and-swap* lock has such excellent performance in relation to the other two locks.

	Cycles	Cache References	Cache Misses	Branches	Branch Misses
Compare-and-swap	7.93E+10	2.53E+07	1.82E+07	1.37E+10	5.08E+07
Test-and-set	7.38E+10	3.80E+07	3.03E+07	1.30E+10	8.77E+07
Test-and-test-and-set	8.26E+10	3.70E+07	2.95E+07	1.46E+10	8.50E+07

Surprisingly, for such a large gap in performance, the *compare-and-swap* lock does not stand out from that data gathered. It has a similar amount of CPU cycles compared to the other two locks along with a comparable amount of cache misses. In fact the *test-and-test-and-set* lock has twenty percent fewer cache misses then the *compare-and-swap* lock. None of the data that I gather should make the *compare-and-swap* lock stand out as it does so this comparison requires further investigation.

4.3.1.1.2 Locked vs Lockless Comparison

I now compare the top performing locks from the previous section with the lockless implementation. The results of the comparison are shown below:

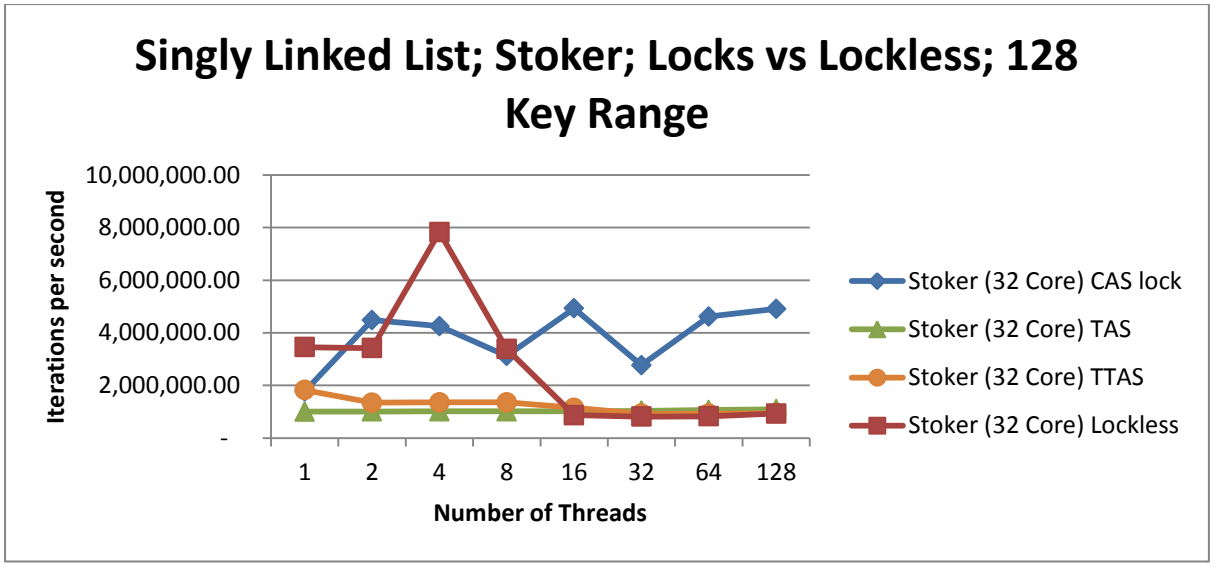


Figure 13

It can be seen that the lockless implementation performs very well at the earlier thread counts, especially at a thread count of four, however it then dips sharply, following the performance of the *test-and-set* and *test-and-test-and-set* locks.

	Cycles	Cache References	Cache Misses	Branches	Branch Misses
Compare-and-swap	7.93E+10	2.53E+07	1.82E+07	1.37E+10	5.08E+07
Test-and-set	7.38E+10	3.80E+07	3.03E+07	1.30E+10	8.77E+07
Test-and-test-and-set	8.26E+10	3.70E+07	2.95E+07	1.46E+10	8.50E+07
Lockless	2.32E+12	9.21E+08	3.76E+08	8.50E+10	1.77E+08

The hardware performance data gives some insight into why the lockless implementation may not perform as well as expected. As seen above, it references the cache much more than the other implementations and as a result also has many more cache misses. Additionally it has almost six times as many branches, which leads to a far higher number of branch misses. Both these factors when combined present a significant problem to the lockless implementation at higher thread counts as the contention between threads causes a higher proportion of CPU cycles to be missed and hence the performance drops.

4.3.1.1.3 TTAS

I will now test the different lock variations to analyse their differences. I start with the *test-and-test-and-set* lock and its two variations, the *test-and-test-and-set-no-pause* lock and the *test-and-test-and-set-relax* lock. Below is a graph showing their respective performances and the hardware performance data gathered for them:

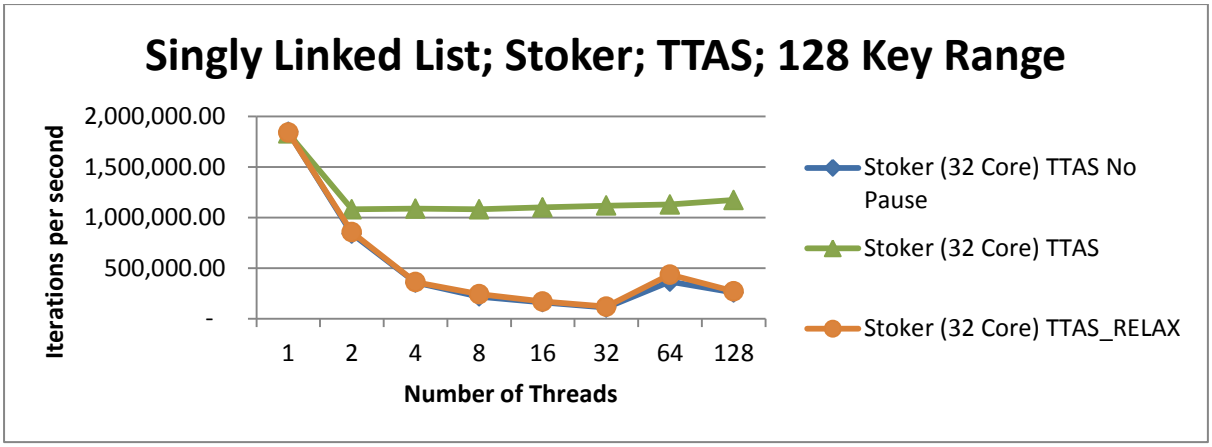


Figure 14

	Cycles	Cache References	Branches	Branch Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Test-and-test-and-set	8.11E+10	3.86E+07	1.42E+10	8.83E+07	5.22E+10	1.25E+10
Test-and-test-and-set-no-pause	2.67E+12	5.53E+08	1.46E+11	9.89E+07	2.38E+12	1.27E+12
Test-and-test-and-set-relax	2.69E+12	4.72E+08	4.02E+10	9.18E+07	2.60E+12	2.31E+12

From the data we can see that all three variations perform very similarly up to a thread count of four. At this point both the *test-and-test-and-set-no-pause* and *test-and-test-and-set-relax* locks fall off. This is perhaps due to increased contention between the threads since the gap between when a thread polls the lock before trying again is quite small or non-existent.

In any case, we see that the *test-and-test-and-set* lock has fewer CPU cycles along with fewer cache references, branches and far fewer stalled cycles both front and backend.

4.3.1.1.6 Size?

I now investigate if size has an impact on the performance of the locked and lockless implementations. I use the *pthread mutex* lock and the lockless implementation for comparison. Below are two graphs, each showing the comparison between their respective implementation with a maximum length of 128 nodes and a maximum length of 131072 nodes.

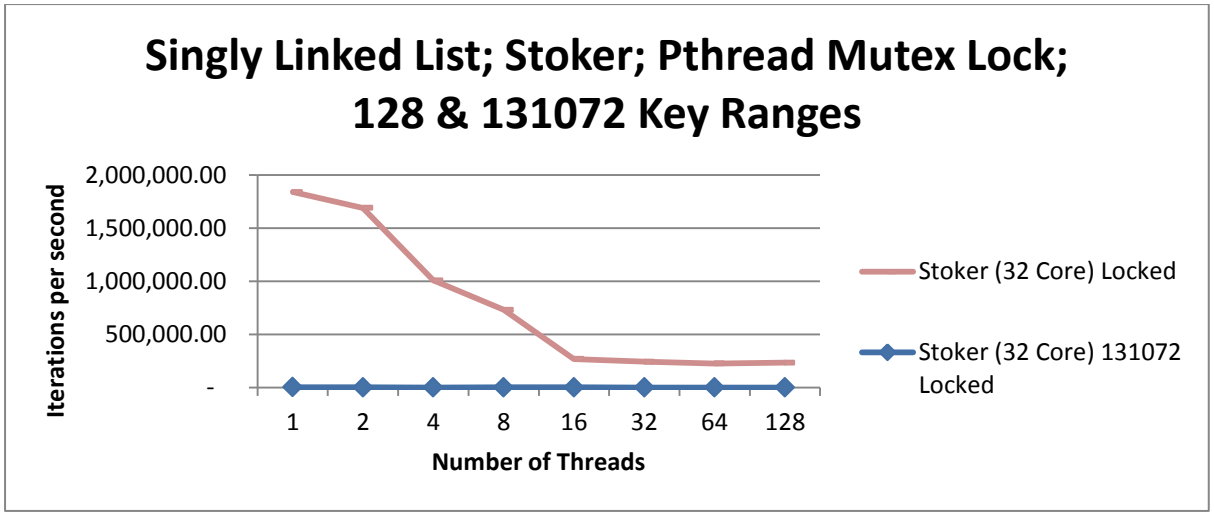


Figure 15

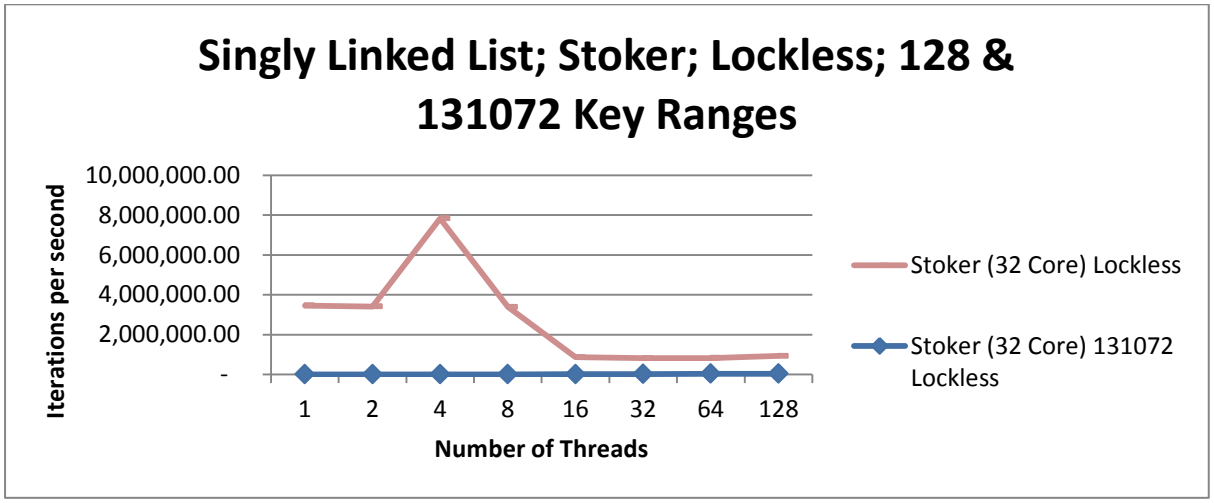


Figure 16

As we can see, the performance drop is staggering. In my opinion the reason for this huge drop in iterations per second is due to the time the threads spend searching for insertion points or nodes to delete. In a shorter list, a thread looking to delete a node will not have to spend as long on average looking as a thread searching through a longer list would. This added time spent traversing the list means that less time is spent adding and removing nodes and hence performance decreases.

4.3.1.1.7 Robust across machines?

For my final piece of evaluation on the singly linked list I will examine if the performance of the locked and lockless implementations is maintained across architectures or if it varies. Below are three graphs, detailing the performance of the *pthread mutex* lock, the *test-and-test-and-set* lock and the lockless implementation across the three architectures of Stoker, Cube and the Local Machine.

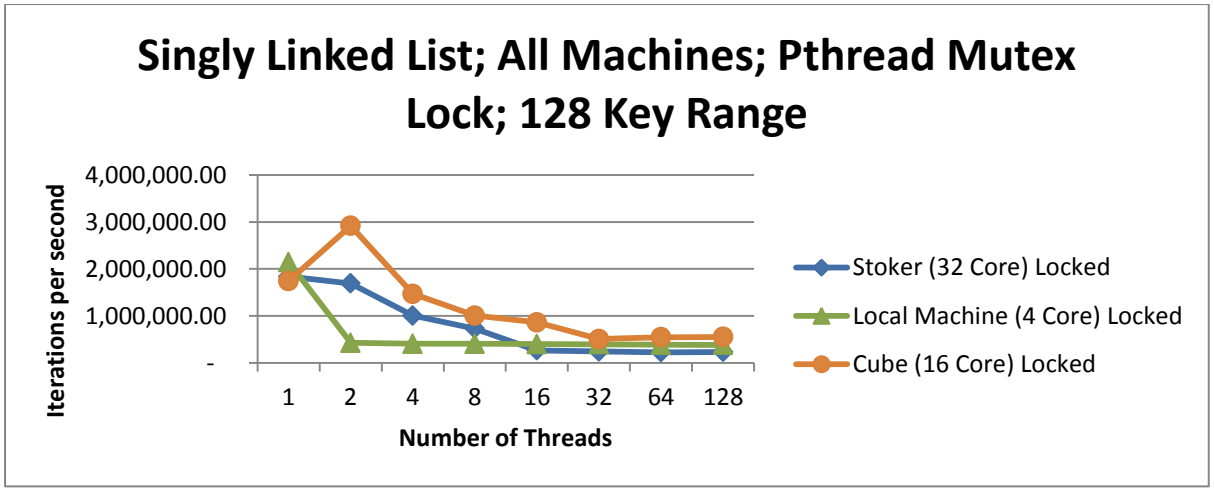


Figure 17

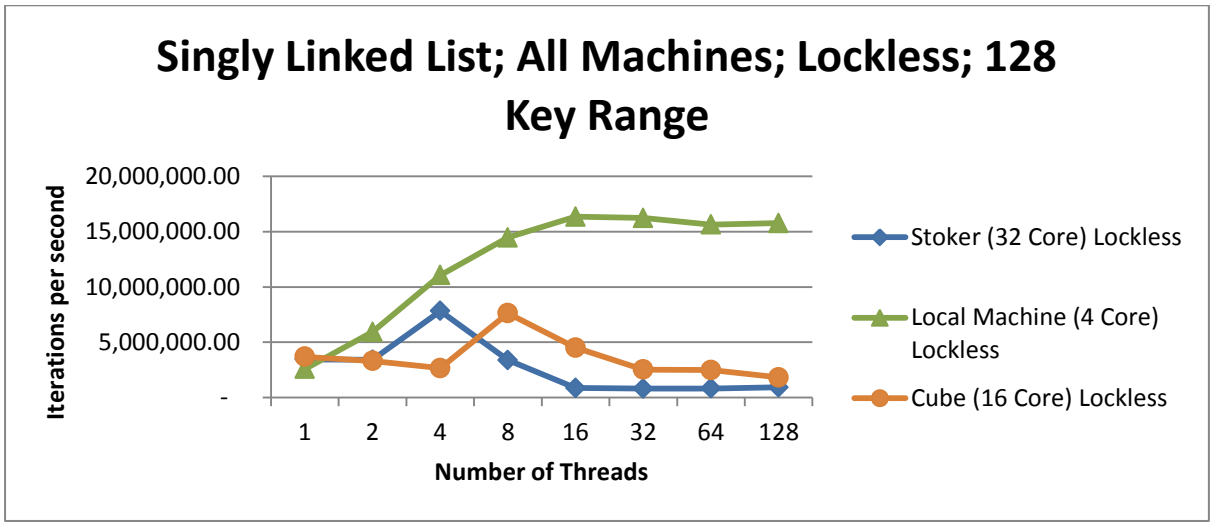


Figure 18

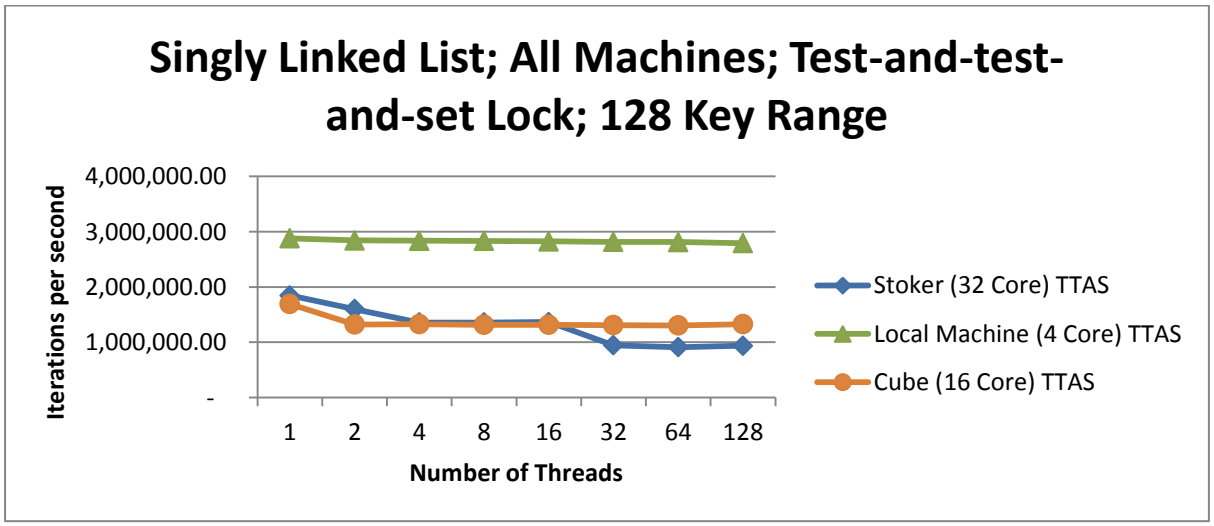


Figure 19

We see mixed results with the robustness of the implementation seemingly depending on which implementation is used. For example we see that the *pthread mutex* lock performs

similarly on the three architectures with the same going for the *test-and-test-and-set* lock. However, the lockless implementation breaks this trend with the Local Machine's performance acting as a stark contrast to both Stoker and Cube.

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. In addition, nodes are added and removed from the head and tail respectively so that threads no longer have to spend any time traversing the list. Due to these changes it is now more of a buffer, hence the name change.

This variation is implemented so that the locked and lockless versions can 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 is 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. I will also be investigating the performance of the lock variations and studying the robustness of the implementations across architectures.

4.3.2.1 Results & Analysis

4.3.2.1.1 Lock Comparison

The three best locked modes of operation on Stoker are the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* locks. This mirrors what has been seen from the other data structures tested such as the ring buffer and singly linked list.

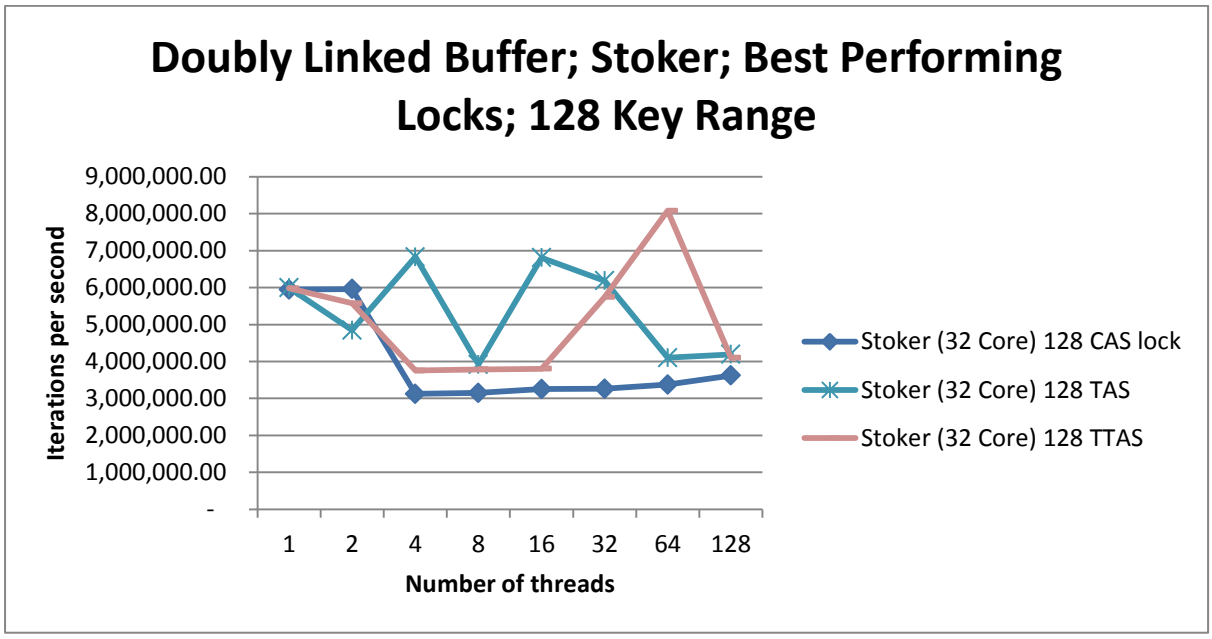


Figure 20

	Cycles	Instructions	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Compare-and-swap	8.37E+10	8.65E+10	1.11E+08	1.02E+08	4.81E+10	3.96E+10
Test-and-set	2.72E+12	3.55E+10	3.54E+08	2.86E+08	2.70E+12	2.63E+12
Test-and-test-and-set	8.53E+10	8.00E+10	8.60E+07	7.71E+07	5.24E+10	4.14E+10

From the hardware performance counters we see that the *test-and-set* lock has far more CPU cycles and fewer instructions to execute due to its simpler design when compared to the *compare-and-swap* or *test-and-test-and-set* locks. It also has the lowest proportion of cache misses to cache references at 80%. However, where the *test-and-set* lock is let down is its wastage of CPU cycles with a high percentage of both front and backend cycles stalling. This allows the *test-and-test-and-set* lock to pull ahead in terms of performance due to its better usage of CPU cycles which comes from its more sophisticated design.

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 for low thread counts. However, past four threads the lockless implementation's performance drops sharply below the locks and does not recover for the remainder of the test.

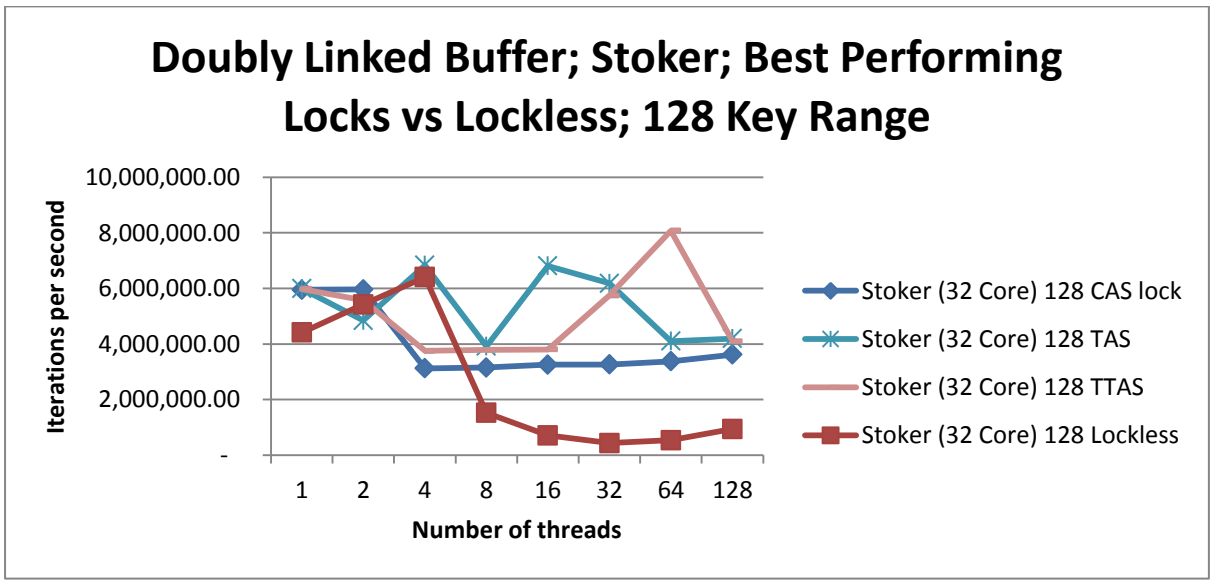


Figure 21

	Cycles	Instructions	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Backend Cycles

Compare-and-swap	8.37E+10	8.65E+10	1.11E+08	1.02E+08	4.81E+10	3.96E+10
Test-and-set	2.72E+12	3.55E+10	3.54E+08	2.86E+08	2.70E+12	2.63E+12
Test-and-test-and-set	8.53E+10	8.00E+10	8.60E+07	7.71E+07	5.24E+10	4.14E+10
Lockless	1.9E+12	1.34E+11	6.70E+08	4.17E+08	1.83E+12	1.54E+12

Now that the lockless data is added onto the other locks we can see why its performance is not what may have been expected. It has a high rate of stalled CPU cycles, comparable to the *test-and-set* lock indicating that the implementation relies heavily on memory.

4.3.2.1.4 Test-and-test-and-set Variations

As with previous data structures I now draw my attention to comparing the lock variations against each other to compare performance. I begin with the *test-and-test-and-set* lock and its two variations *test-and-test-and-set-no-pause* and *test-and-test-and-set-relax*. Figure 22 shows this comparison. As with previous tests, the *test-and-test-and-set* lock proves itself to be the best performing variation.

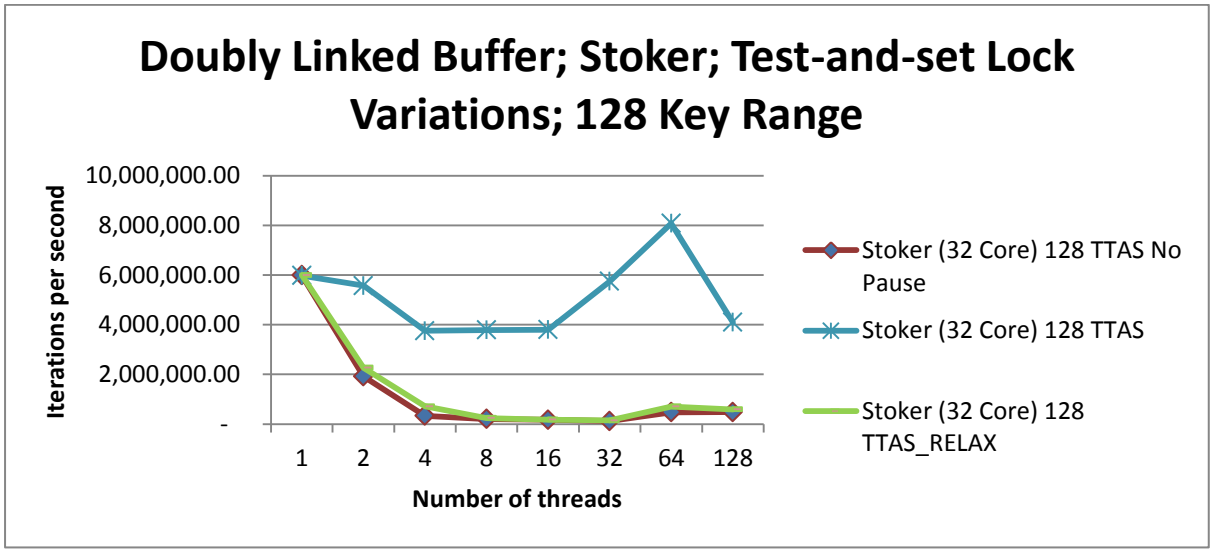


Figure 22

Table x below contains the relevant hardware performance data for the comparison in figure 22.

	Cycles	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Test-and-test-and-set	8.53E+10	8.60E+07	7.71E+07	5.24E+10	4.14E+10
Test-and-test-and-set-no-pause	2.22E+12	6.78E+08	3.02E+08	1.91E+12	9.26E+11
Test-and-test-and-set-relax	2.14E+12	4.20E+08	2.49E+08	2.09E+12	1.94E+12

From table x we can see that the *test-and-test-and-set* lock has the lowest number of CPU cycles, and the lowest proportion of stalled front and backend cycles. These two results

combined seem to overpower its high number of cache misses allowing it to achieve relatively good performance when compared to the two other variations.

4.3.2.1.3 Test-and-set Variations

Moving on to the *test-and-set* lock and its variations, figure 23 shows the relative performance of the three *test-and-set* locks.

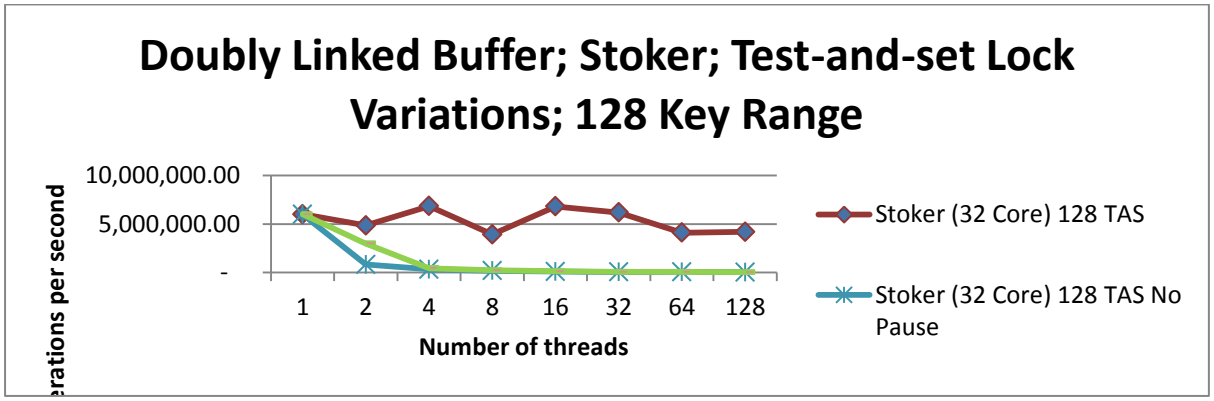


Figure 23

	Cycles	Branches	Branch Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Test-and-set	8.53E+10	1.62E+10	4.22E+06	5.24E+10	4.14E+10
Test-and-set-no-pause	2.22E+12	7.82E+09	1.65E+07	1.91E+12	9.26E+11
Test-and-set-relax	2.14E+12	7.91E+09	1.13E+07	2.09E+12	1.94E+12

From table x it can be seen how the *test-and-set* lock outperforms the other two variations by such a margin. Firstly, the *test-and-set* lock has at least seven times fewer branch misses than the alternate variations and a far lower proportion of its front and backend cycles are stalled. This allows for much greater efficiency and better use of the CPUs' pipelines, giving the *test-and-set* lock such a good margin of performance.

4.3.2.1.3 Compare-and-swap Variations

The doubly linked buffer is the first data structure that I have been able to successfully gather data from the two other variations of the *compare-and-swap* lock, the *compare-and-swap-no-delay* lock and the *compare-and-swap-relax* lock. Figure 24 shows their relative performance.

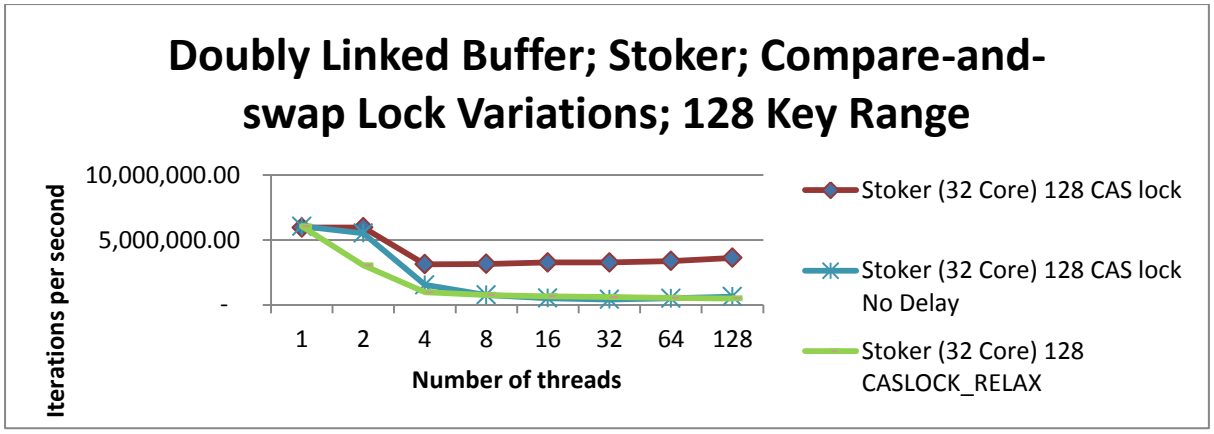


Figure 24

	Cycles	Branches	Branch Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Compare-and-swap	8.37E+10	1.74E+10	5.73E+06	4.81E+10	3.96E+10
Compare-and-swap-no-delay	2.24E+12	5.83E+10	9.40E+07	2.14E+12	1.70E+12
Compare-and-swap-relax	2.14E+11	7.68E+10	8.80E+07	2.22E+12	1.79E+12

From the table it can be seen how the *compare-and-swap* lock pulls ahead by the margin that it does. Like the *test-and-set* lock, the *compare-and-swap* lock's branch misses are far lower than its competitors, possibly due to the delay a thread encounters after failing to acquire the lock, leading to less failed attempts. In addition, the fewer CPU pipeline stalls combine with the branching to allow for a far more efficient use of the CPUs' pipelines, leading to an increase in performance over the *compare-and-swap-no-pause* and *compare-and-swap-relax* locks whose execution would be more disruptive with regard to the CPUs' pipelines.

4.3.2.1.3 Ticket?

I decide to investigate the *ticket* lock I feel that it goes against the trend of the other locks I am testing. Whereas the *compare-and-swap* or *test-and-set* locks all appear to perform better using the sleep instruction, the *ticket* lock does not follow this as can be seen from figure 25. Instead, the *ticket-relax* lock is the better performing of the two, through the use of the *_mm_pause()* intrinsic instead of the *sleep()* instruction.

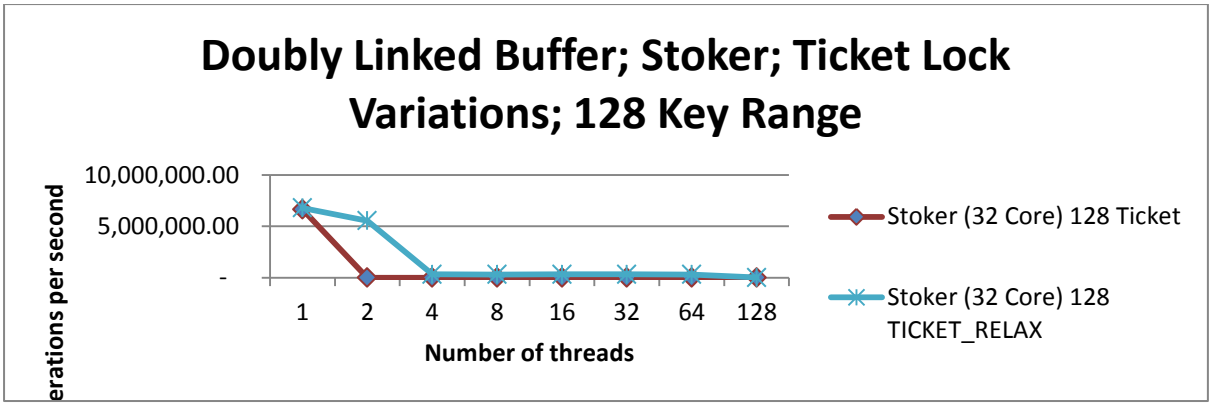


Figure 25

	Cycles	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Ticket	1.52E+10	3.06E+07	2.37E+07	8.12E+09	6.99E+09
Ticket-relax	2.87E+12	4.35E+08	2.28E+08	2.63E+12	2.25E+12

From table x the *ticket-relax* lock has far more CPU cycles, nearly two hundred times that of the regular *ticket* lock. This may explained by the *ticket* lock's use of the *sleep()* instruction. Whereas it used a proportional sleep, where the further down the queue a thread is the longer it sleeps, the *ticket-relax* lock simply uses the *_mm_pause()* instruction.

The *ticket-relax* lock also had a lower proportion of cache and branch misses than the *ticket* lock, though it has higher amounts of stalled cycles which may be why both of the locks fall off so sharply in performance past two threads.

4.3.2.1.3 Size?

As with previous data structures I now investigate how size affects this implementation of a doubly linked buffer with regards to performance. The following three figures, 26, 27 and 28 all graph the performance of one lock, run on the data structure using two different sizes.

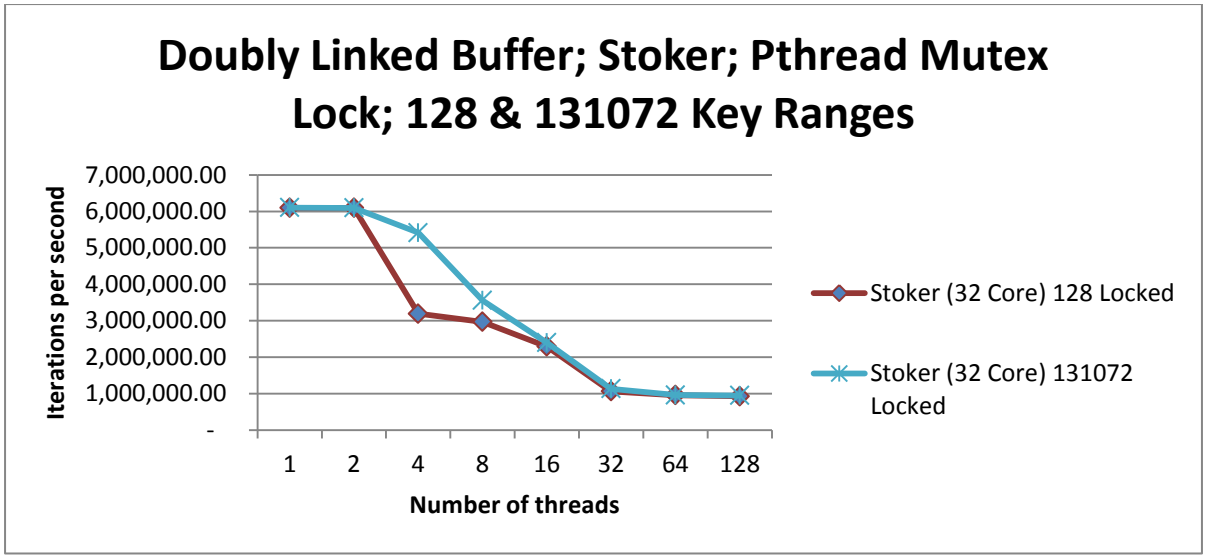


Figure 26

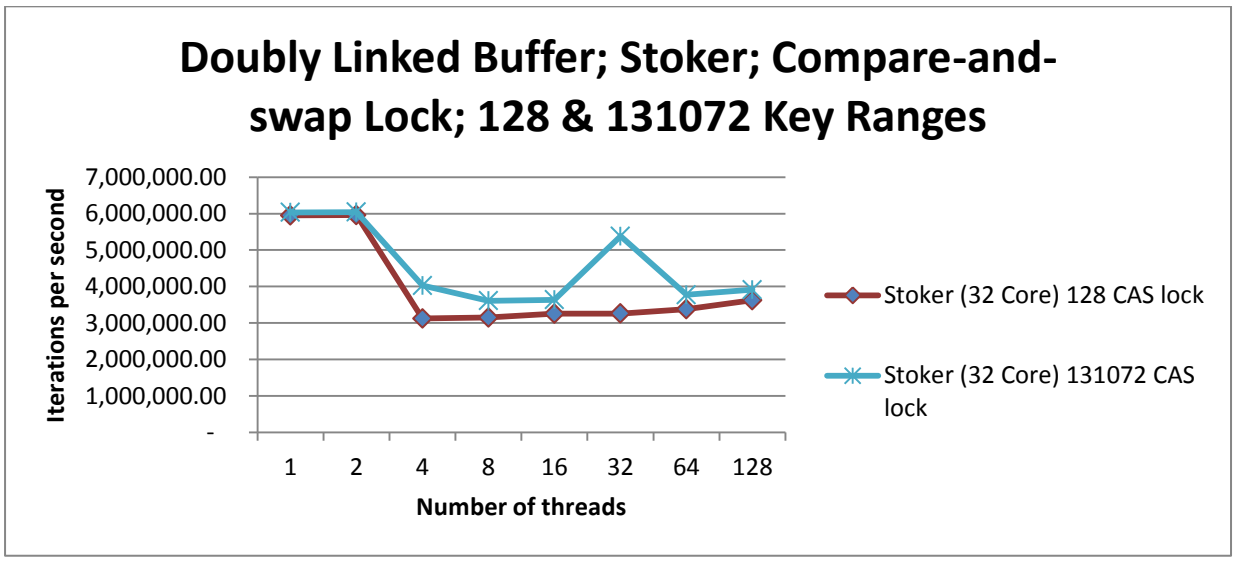


Figure 27

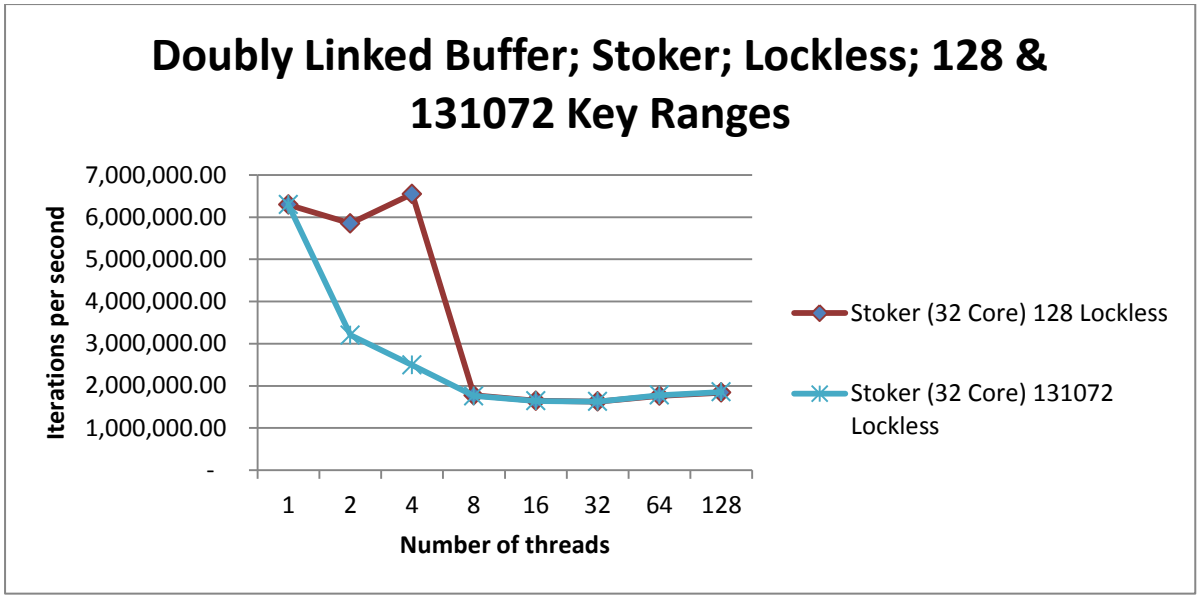


Figure 28

As with previous tests done with the ring buffer and singly linked list, different implementations appear to react differently. In figure 26 we see that the *pthread mutex* lock performs similarly with the two different sizes. Figure 27 shows much the same picture, with an exception occurring at a thread count of 32. Figure 28 shows us something different again with a significant difference between the two lockless implementations at lower thread counts.

4.3.2.1.3 Architectures?

For the final piece of evaluation on the doubly linked buffer I will now investigate how robust the implementations are across different architectures. Firstly, figure 29 shows us the *pthread mutex* lock which has a similar shape for both Stoker and Cube, yet the performance on Local is quite different. Then in figure 30 we see the *compare-and-swap* lock which appears to perform differently on each of the three architectures. Finally, in figure 31 we see the lockless implementation which again performs differently on each of the architectures.

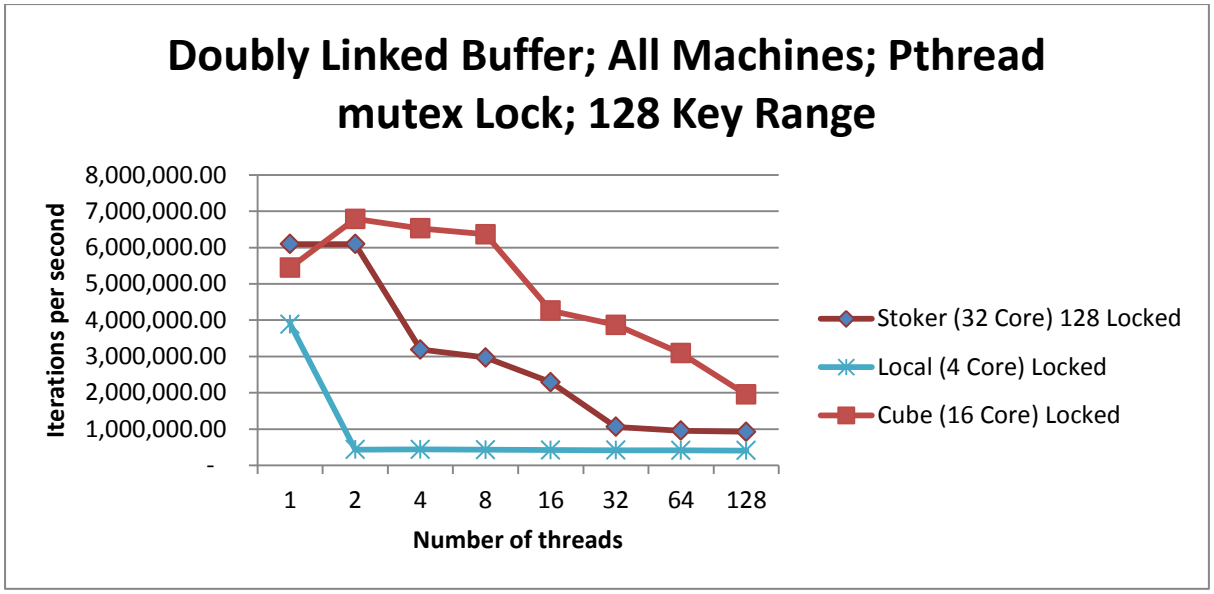


Figure 29

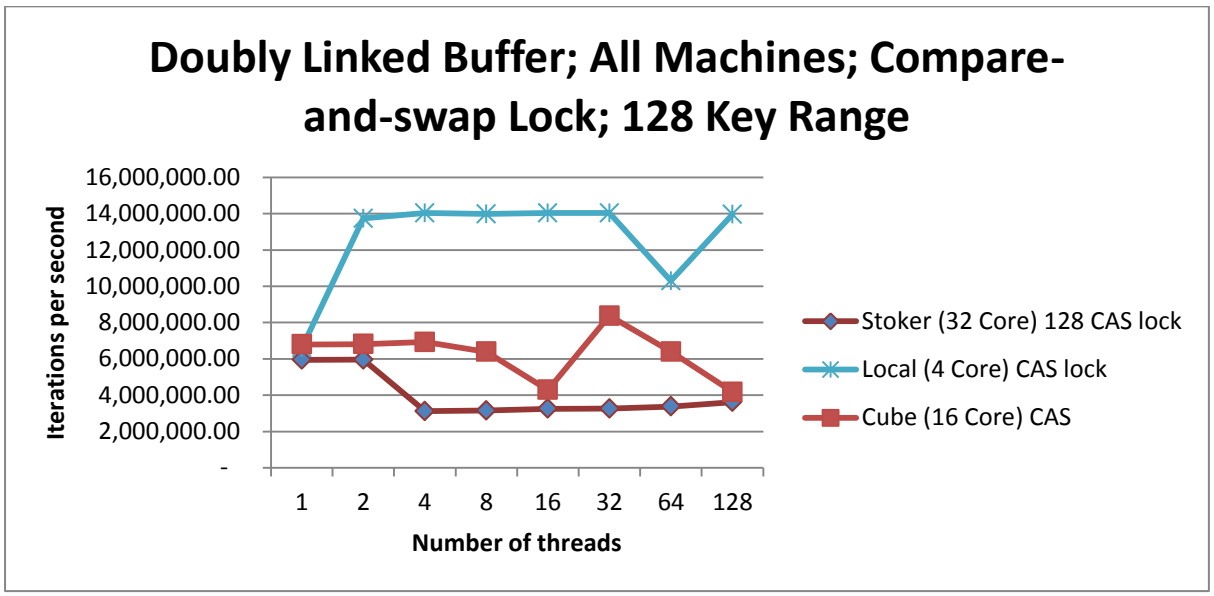


Figure 30

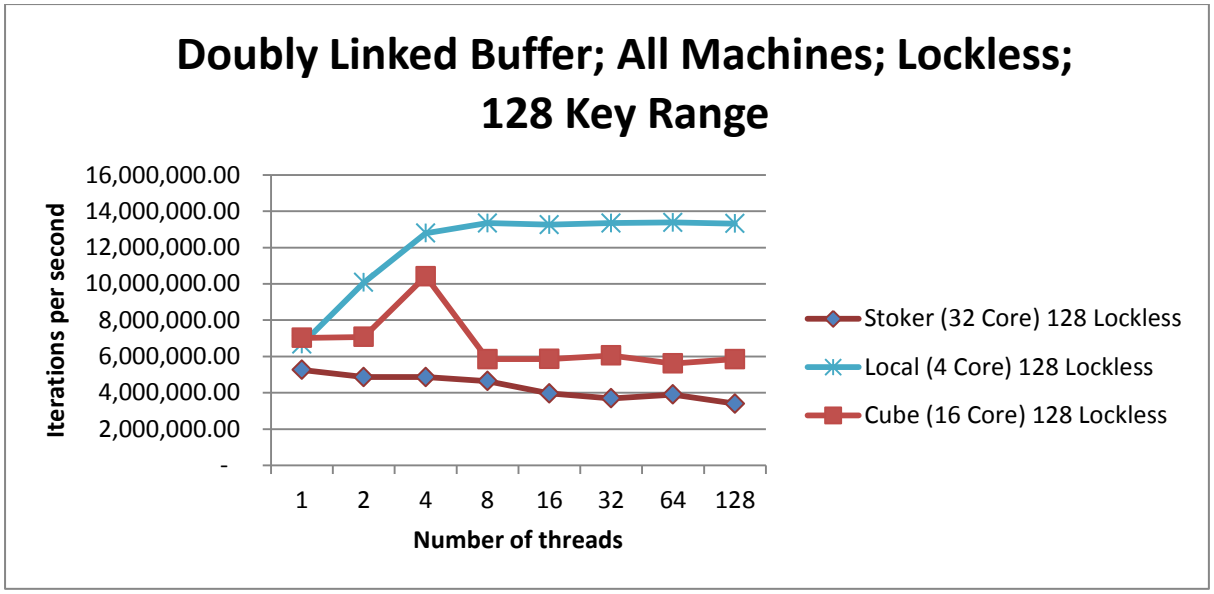


Figure 31

4.3.3 Singly Linked Buffer

4.3.3.1 Evaluation

The evaluation for the singly linked buffer is identical to the doubly linked buffer. I evaluate the locks first to determine which have the highest performance. I then move onto comparing the locks with the lockless implementation. Afterwards, I investigate the differences between different lock variations before finally examining the robustness of the implementations across different architectures.

The singly linked buffer is similar to the doubly linked buffer in that nodes are continuously added onto the head and removed from the tail except in this variation the placement of the head and tail pointer are switched thus eliminating the need for each node to have a second pointer. I am interested in seeing if this simpler implementation affects the performance of the locked and lockless varieties.

4.3.3.1 Results & Analysis

4.3.3.1.1 Locked Comparison

From figure 32 the best performing locks on the singly linked list are as follows, the *compare-and-swap*, *test-and-set*, *test-and-test-and-set* and *pthread mutex* lock.

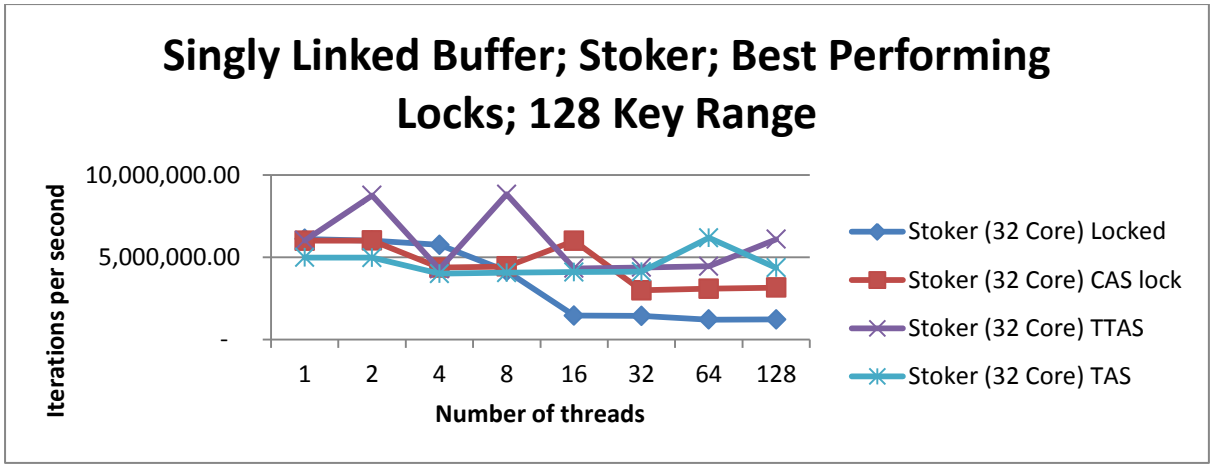


Figure 32

	Cycles	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Frontend Cycles
Pthread Mutex	2.20E+12	6.32E+08	2.92E+08	2.09E+12	1.42E+12
Compare-and-swap	8.59E+10	9.15E+07	8.33E+07	5.13E+10	4.15E+10
Test-and-set	8.78E+10	9.90E+07	8.94E+07	5.22E+10	4.19E+10
Test-and-test-and-set	9.14E+10	8.02E+07	7.18E+07	5.66E+10	4.48E+10

Table X shows the hardware performance data for the four best performing locks. The *pthread mutex* and *test-and-set* locks have equally low amounts of cache misses when compared to their total cache references. The *test-and-test-and-set* lock has the fewest frontend stalls and a low number of backend stalls which possibly contribute towards its impressive performance against the other locks.

4.3.3.1.2 Locked vs Lockless Comparison

Figure 33 shows the relative performances of the lockless implementation against two of the best performing locks, the *pthread mutex* and *test-and-test-and-set* locks.

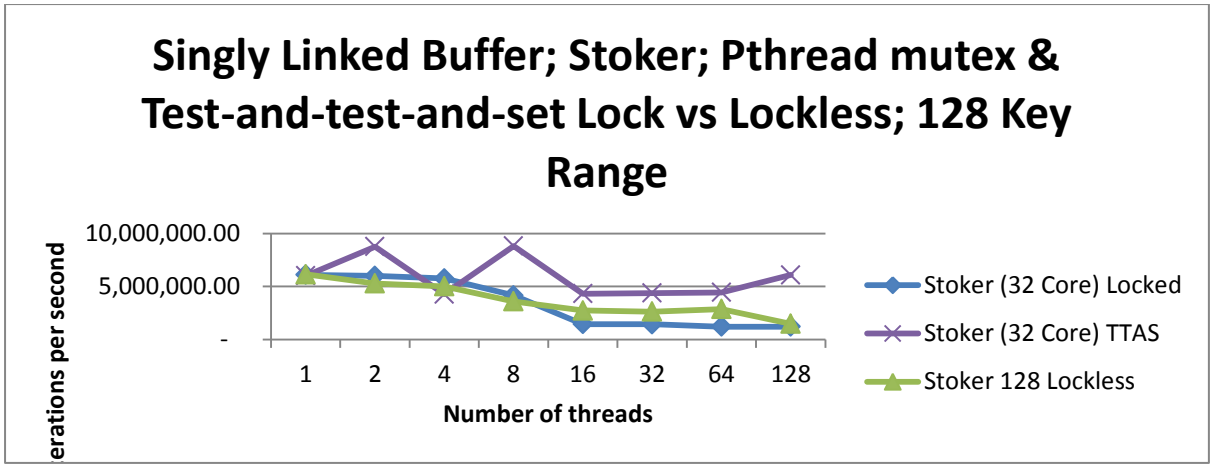


Figure 33

	Cycles	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Frontend Cycles
Pthread Mutex	2.20E+12	6.32E+08	2.92E+08	2.09E+12	1.42E+12
Test-and-test-and-set	9.14E+10	8.02E+07	7.18E+07	5.66E+10	4.48E+10
Lockless	9.42E+11	6.99E+08	4.68E+08	8.88E+11	6.53E+11

Table X displays the hardware performance data for the three implementations shown in figure 33. The lockless implementation performs about as well as the *pthread mutex* lock, with the lockless implementation losing in the early thread counts but pulling ahead of the *pthread mutex* lock as thread contention got higher. A possible explanation for lockless performance is the high contention around both the head and tail of the buffer. In such instances a lock seems to be preferable, suggested by the high branch miss rate and stalled cycles of the lockless implementation.

4.3.3.1.3 Test-and-test-and-set

Shown in figure 34 are the relative performances of the three variations on the *test-and-test-and-set* lock. Both the *test-and-test-and-set-no-pause* and *test-and-test-and-set-relax* locks have almost identical performance across all thread counts while the regular *test-and-test-and-set* lock spikes in performance at thread counts of 2 and 8 respectively.

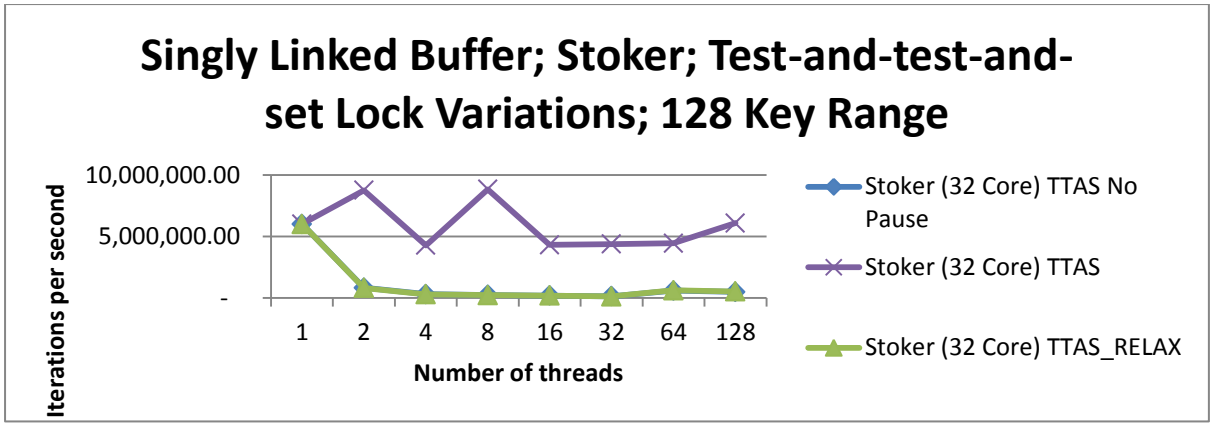


Figure 34

	Cycles	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Frontend Cycles
Test-and-test-and-set	9.14E+10	8.02E+07	7.18E+07	5.66E+10	4.48E+10
Test-and-test-and-set-no-pause	2.26E+12	6.34E+08	2.78E+08	1.88E+12	9.35E+11
Test-and-test-and-set-relax	2.26E+12	5.09E+08	2.32E+08	2.16E+12	1.96E+12

Table X further confirms the results with the regular *test-and-test-and-set* lock utilising its CPU cycles the most efficiently, sporting the lowest proportion of stalled cycles on both front and backend. The regular *test-and-test-and-set* lock does have a high number of cache misses but these seem to be outweighed from what is observed.

4.3.3.1.4 Test-and-set

Figure 35 shows us the different implementations of the *test-and-set* lock and their respective performances. The *test-and-set* lock outperforms the other two locks by a significant degree. To explain why, the hardware performance data is analysed.

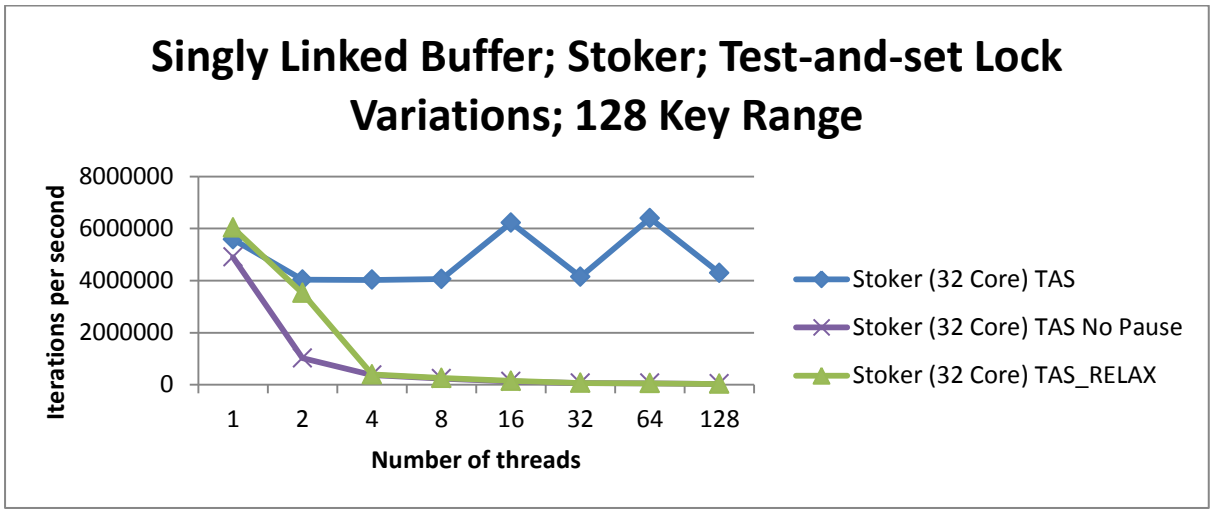


Figure 35

	Cycles	Branches	Branch Misses	Stalled Frontend Cycles	Stalled Frontend Cycles
Test-and-set	8.85E+10	1.66E+10	1.14E+07	5.37E+10	4.39E+10
Test-and-set-no-pause	2.63E+12	6.61E+09	2.04E+07	2.62E+12	2.55E+12
Test-and-set-relax	2.70E+12	8.15E+09	2.06E+07	2.67E+12	2.53E+12

The first item to note is the far lower rate of branch misses attributed to the *test-and-set* lock. Both the *test-and-set-no-pause* and *test-and-set-relax* locks have 0.25 to 0.3 percent of all branches resulting in a miss while the *test-and-set* lock only has a rate of 0.06. This results in a far less volatile pipeline [reference].

The second item that provides the *test-and-set* with such a lead with regards to performance is the number of CPU cycle stalls experienced by each lock. The *test-and-set* lock has a far lower rate of stalled cycles than the other two locks that stall upwards of 95% of all their CPU cycles.

4.3.3.1.5 Compare-and-swap

In figure 36 we see the three types of lock based on the atomic *compare-and-swap* instruction. The *compare-and-swap-no-delay* and *compare-and-swap-relax* locks do equally well in terms of performance but the regular *compare-and-swap* beats them both by a reasonable margin.

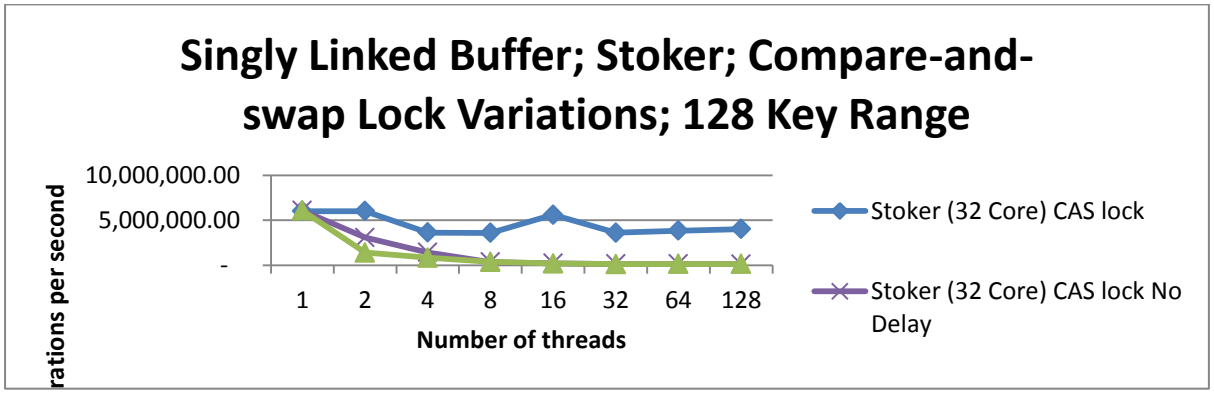


Figure 36

	Cycles	Branches	Branch Misses	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Frontend Cycles
Compare-and-swap	8.47E+10	9.88E+07	8.88E+07	1.70E+10	5.20E+06	4.99E+10	4.04E+10
Compare-and-swap-no-delay	2.29E+12	7.07E+08	4.57E+08	7.32E+10	8.54E+07	2.17E+12	1.76E+12
Compare-and-	2.28	7.23	4.70E+	5.78E+10	8.22E+	2.19E+12	1.82E+12

swap-relax	E+12	E+08	08		07		

To begin, the regular *compare-and-swap* lock has the largest amount of cache misses based on total cache references compared to the other two locks. However, where it pulls ahead is with its branch misses and CPU cycle utilisation. The regular *compare-and-swap* lock has a very low branch miss rate when compared to the other two locks and only wastes just over half of its CPU cycles. Compare this to the other two locks who miss over three times as many branches and waste around eighty five percent of their CPU cycles and it becomes clear why the regular *compare-and-swap* lock does so well.

4.3.3.1.6 Ticket

For the final lock comparison I examine the *ticket* lock and its alternative, the *ticket-relax* lock which replaces the *sleep()* instruction with the intrinsic *_mm_pause()*. Figure 37 shows how the two locks compare in terms of performance.

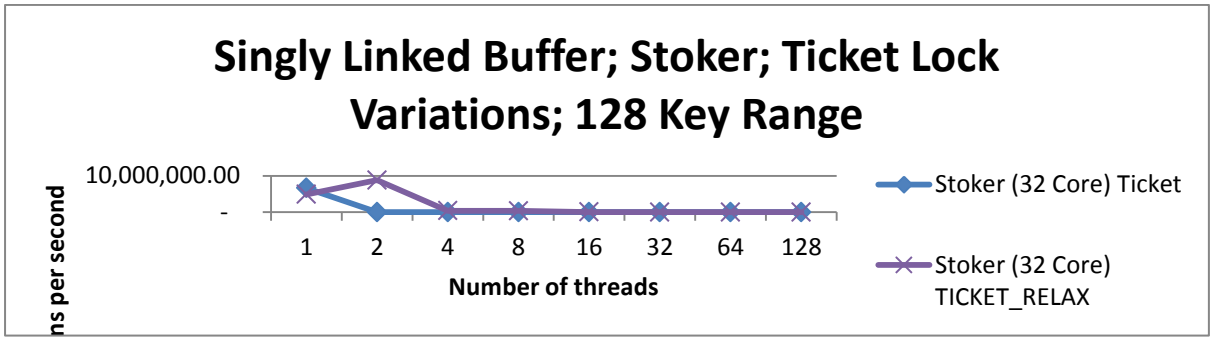


Figure 37

	Cycles	Cache References	Cache Misses	Branches	Branch Misses	Stalled Frontend Cycles	Stalled Frontend Cycles
Ticket	1.56E+10	3.09E+07	2.52E+07	3.66E+09	2.15E+06	8.52E+09	7.09E+09
Ticket-relax	2.28E+12	2.19E+08	1.16E+08	1.58E+11	1.91E+07	1.89E+12	1.40E+12

Table X shows the hardware performance data gathered for the two locks. The *ticket* lock utilises its CPU cycles more efficiently than the *ticket-relax* lock with a smaller number of stalled cycles, however, the *ticket-relax* lock has five times fewer branch misses and a smaller proportion of its cache references return a miss.

4.3.3.1.8 Architectures?

Figure 38, 39 and 40 show three implementations of the singly linked buffer run on the three architectures I use. From the graphs none of the implementations I test have similar performances on all three machines.

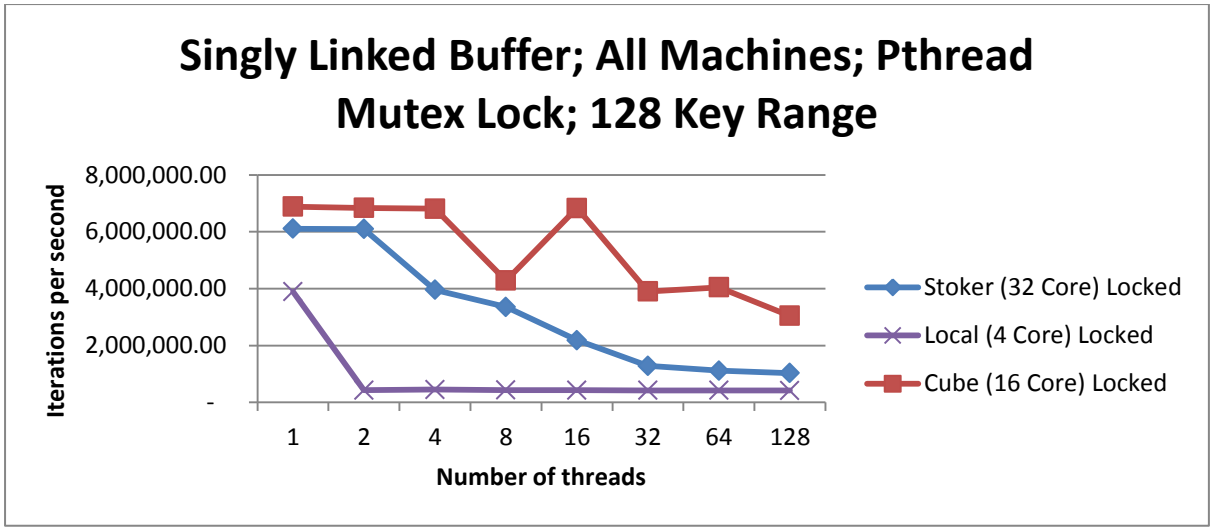


Figure 38

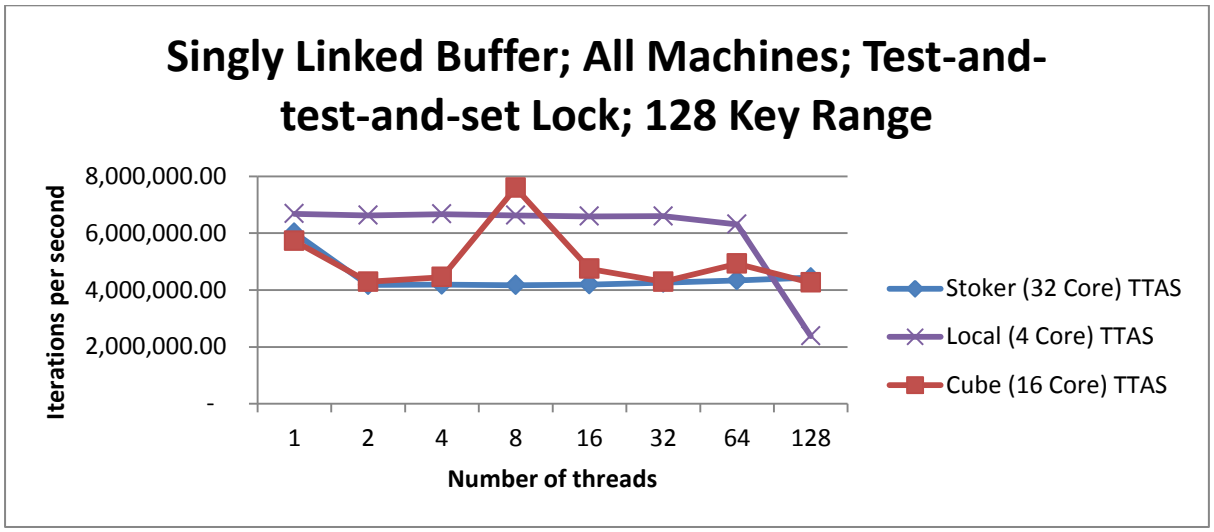


Figure 39

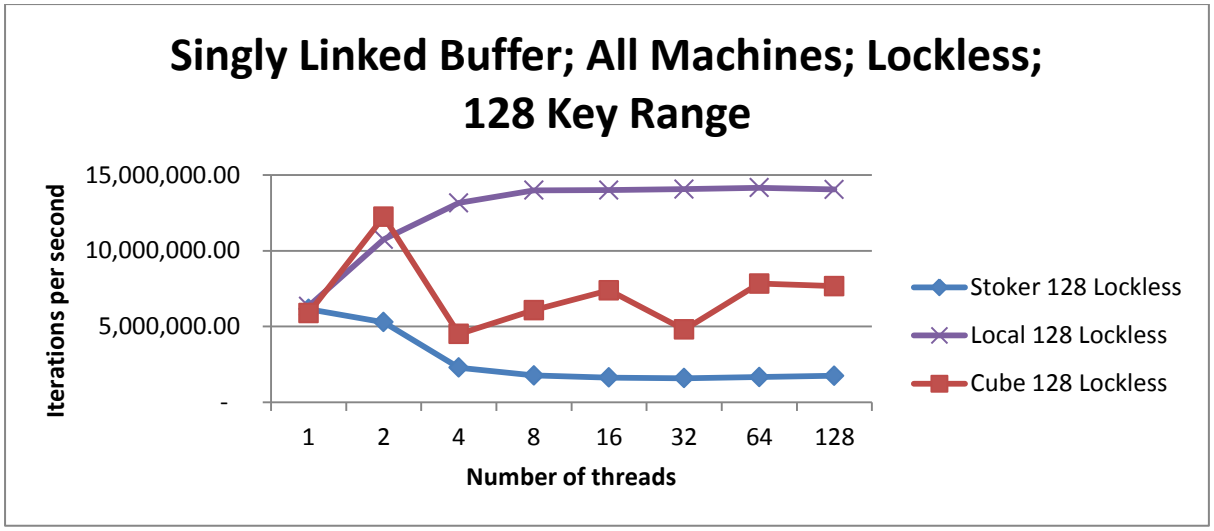


Figure 40

4.4 Hash Table

4.4.1 Evaluation

For the hash table I have two locked variations and the lockless variation. As discussed in the method section of the report, the globally locked version uses a single, global lock to grant mutual exclusion. The second locked variation, *lock per bucket*, is more granular and gives each bucket its own lock so that multiple threads can interact with the table but only on separate buckets. Finally the lockless version does not use any locks so multiple threads can interact with the same bucket.

To begin evaluation, I start with an initial table size of 128, with no resizing of the table. I then increase the table size to investigate how it impacts the performance of the three variations. After that I turn on resizing for the same purpose and finally I test the variations on Cube and my Local Machine to see whether the variations' performance changes with the architecture it runs on.

4.4.2 Results & Analysis

4.4.2.1 Global Lock Comparison

Figure 41 shows the top three best performing locks when using the *global lock* implementation of the hash table.

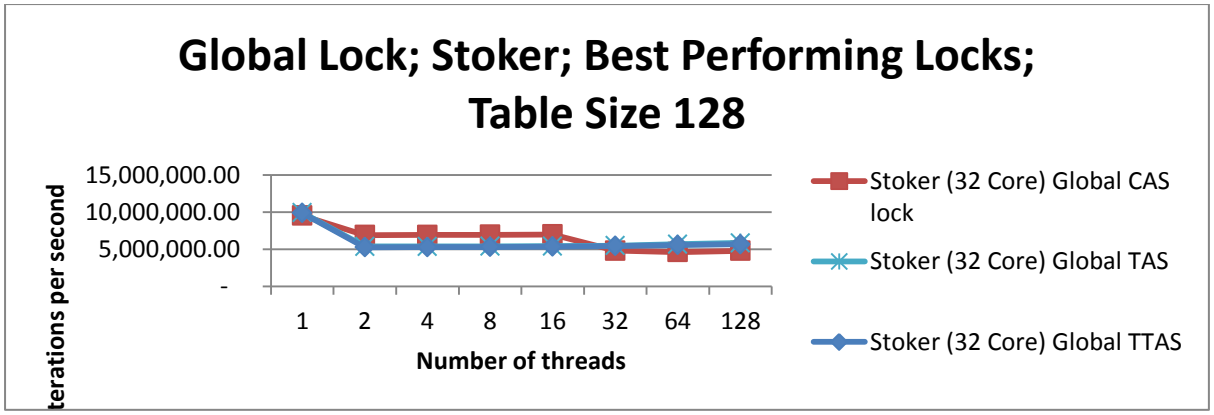


Figure 41

	Cycles	Cache References	Cache Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Compare-and-swap	8.80E+10	1.51E+07	6.95E+06	5.55E+10	4.01E+10
Test-and-set	2.69E+12	2.70E+08	2.18E+08	2.67E+12	2.61E+12
Test-and-test-and-set	7.64E+10	1.43E+07	6.47E+06	4.65E+10	3.47E+10

Table X shows us the hardware performance data from figure 41. The *compare-and-swap* lock and the *test-and-test-and-set* lock perform similarly and we can why as they both have roughly the same number of cycles, a similar ratio of cache misses and stall the same ratio of CPU cycles.

The *test-and-set* lock has over thirty times as many cycles as the other two locks, yet its performance is almost identical. This could be due to the fact that a far higher rate of cache misses and it stalls far more of its cache cycles which cause its performance to dip to the same level as the two other locks.

4.4.2.2 Lock per Bucket Comparison

Figure 42 shows the top three locks when using the *lock per bucket* implementation of the hash table. As with the *global lock* implementation, the same locks show the best performance.

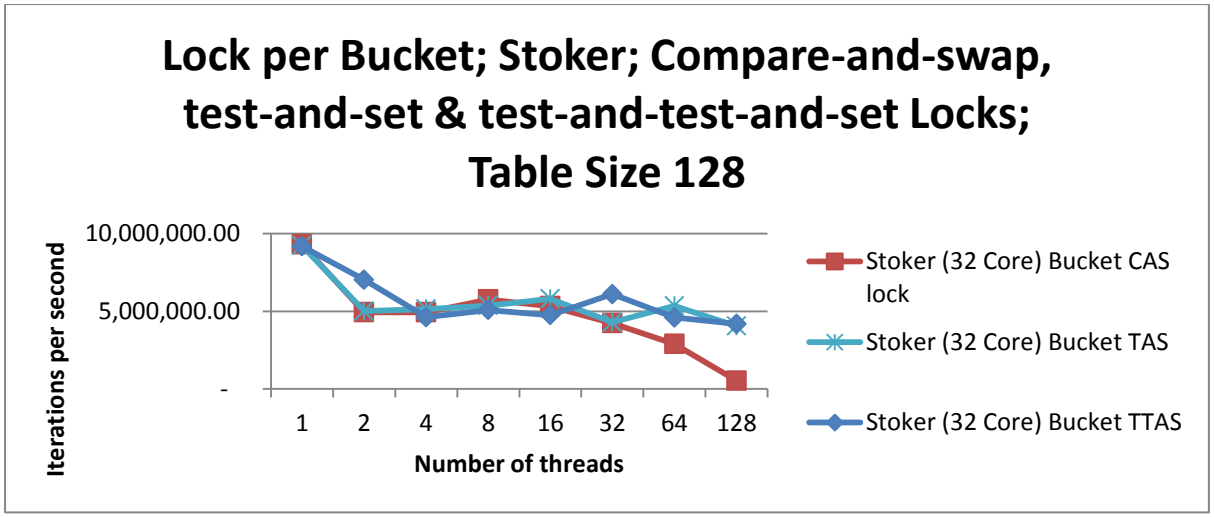


Figure 42

	Cycles	Stalled Frontend Cycles	Stalled Backend Cycles
Compare-and-swap	2.33E+12	2.21E+12	1.32E+12
Test-and-set	1.33E+11	9.95E+10	6.56E+10
Test-and-test-and-set	2.39E+12	1.96E+12	1.09E+12

Table X shows the hardware performance counter data; all three locks are similar in their performance until the thread count reaches thirty two. At this point, the *compare-and-swap* lock drops off sharply, possibly due to the large amount of contention between threads and the high number of different locks in use on the data structure.

4.4.2.3 Global Lock vs Lock per Bucket Comparison

Figure 43 shows the *global lock* and *lock per bucket* implementations compared with two different locks. The two *test-and-test-and-set* locks perform relatively equally, however the *compare-and-swap* lock in the *lock per bucket* implementation drops in performance as the thread count rises while the *compare-and-swap* lock in the *global lock* implementation stays stable as the thread count rises with a slight dip at a thread count of thirty two.

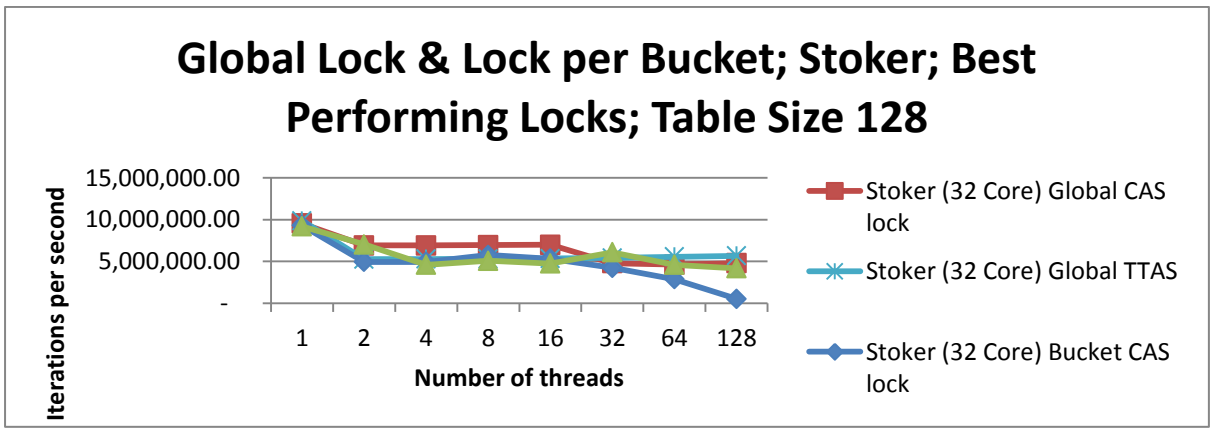


Figure 43

4.4.2.4 Locked Implementations vs Lockless Comparison

Figure 44 describes the performance differences between the *compare-and-swap* locks of both locked implementations and the lockless implementation which is based on the *compare-and-swap* atomic instruction.

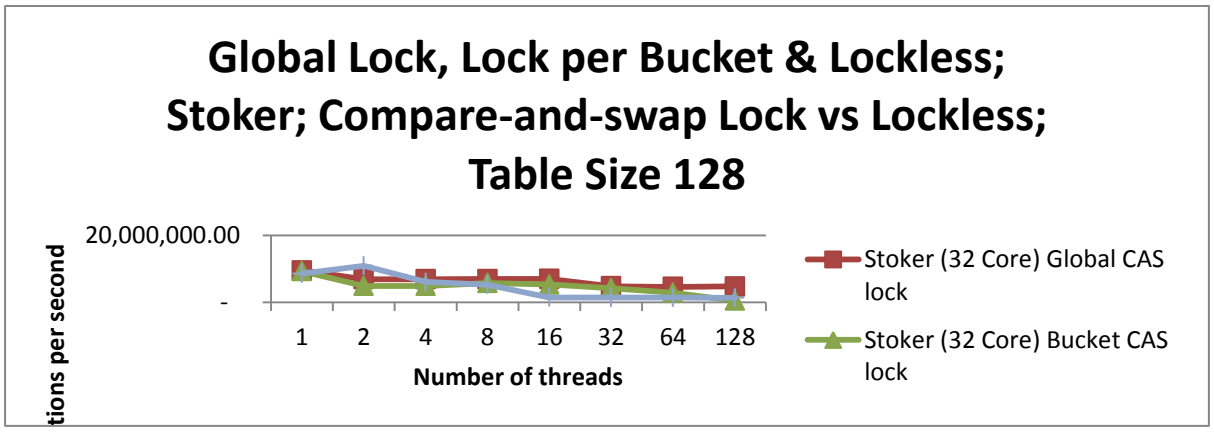


Figure 44

	Cycles	Cache References	Cache Misses	Branches	Branch Misses
Global compare-and-swap	8.80E+10	1.51E+07	6.95E+06	1.57E+10	1.67E+06
Bucket compare-and-swap	2.33E+12	5.87E+08	2.78E+08	7.53E+10	5.37E+07
Lockless	2.22E+12	6.21E+08	2.48E+08	6.09E+10	5.80E+07

Table X shows us the differences between the three implementations from the perspective of the hardware performance counters. The *lockless* implementation has the highest rate of branch misses out of the three, though it counters this with a particularly low rate of cache misses. However, it would seem that the *lockless* implementation suffers when contention between threads increases, with it peaking in performance at a thread count of two but then falling until it stabilises again at a thread count of 16.

4.4.2.5 What impact does resizing have?

To investigate the impact resizing has on the performance of the three variations I compare each variation with itself, putting the data gathered with no resize functionality beside the data with resize functionality enabled.

I choose the maximum length of a list allowed before resizing occurs to be four, as chosen by Herlihy and Shavit in “The Art of Multiprocessor Programming”.

Figure 45 show that the resize functionality causes a noticeable drop in performance whenever it is required. For the *pthread mutex* lock with resize in figure 45 it drops sharply, at thread count 2 and thread count 8 at which point it stays very low. For the *test-and-test-and-set* lock with resize it appears to perform almost identically to the regular *test-and-test-*

and-set lock until thread count 32 at which point it drops sharply and continues to do so as it get to the 128 thread count.

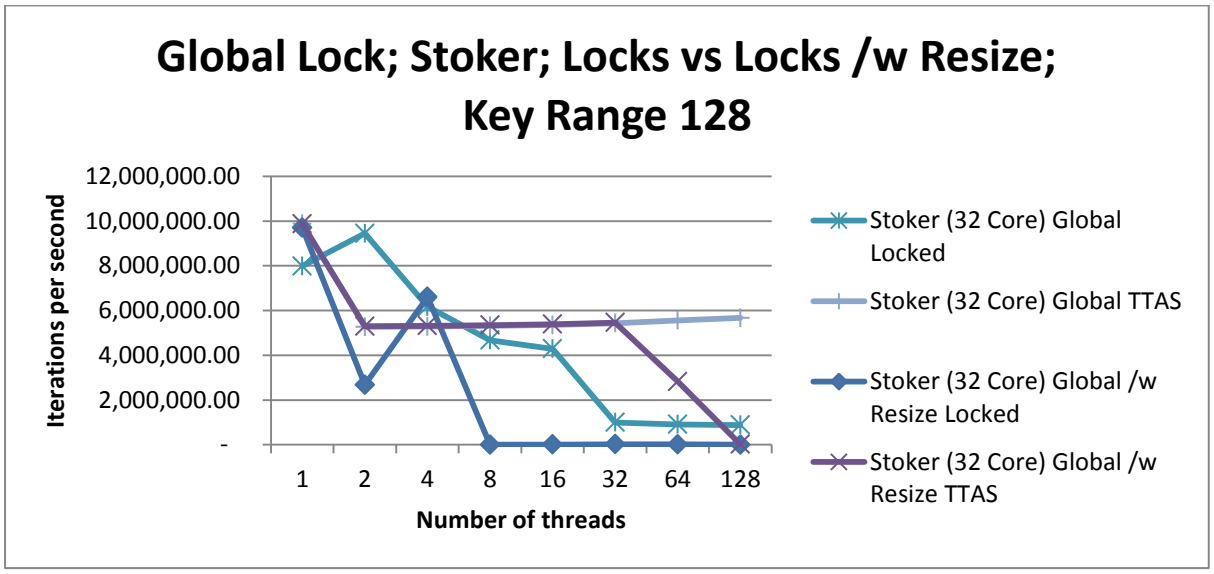


Figure 45

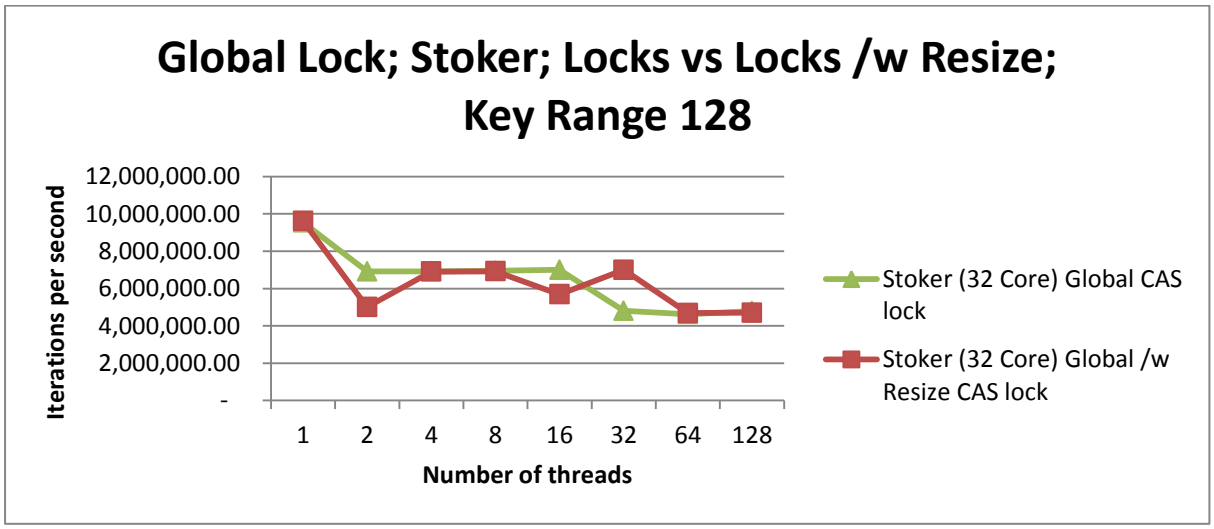


Figure 46

Relative to other locks, the *compare-and-swap* lock with resize performs quite well when compared to its resize-less version. A drop is seen at thread count 2, 16 but apart from those two points it performs equally if not better than the regular *compare-and-swap* lock.

4.4.2.3 How does the size of the table affect performance?

To evaluate the effect that size may have on the performance of the hash table implementations, I run several locks at the default size of 128 and then run them again at 131072. Figure 47 shows the comparison between the *pthread mutex* lock when using the *global lock* implementation of the hash table with two different table sizes while figure 48 shows the same thing but with the *compare-and-swap* lock.

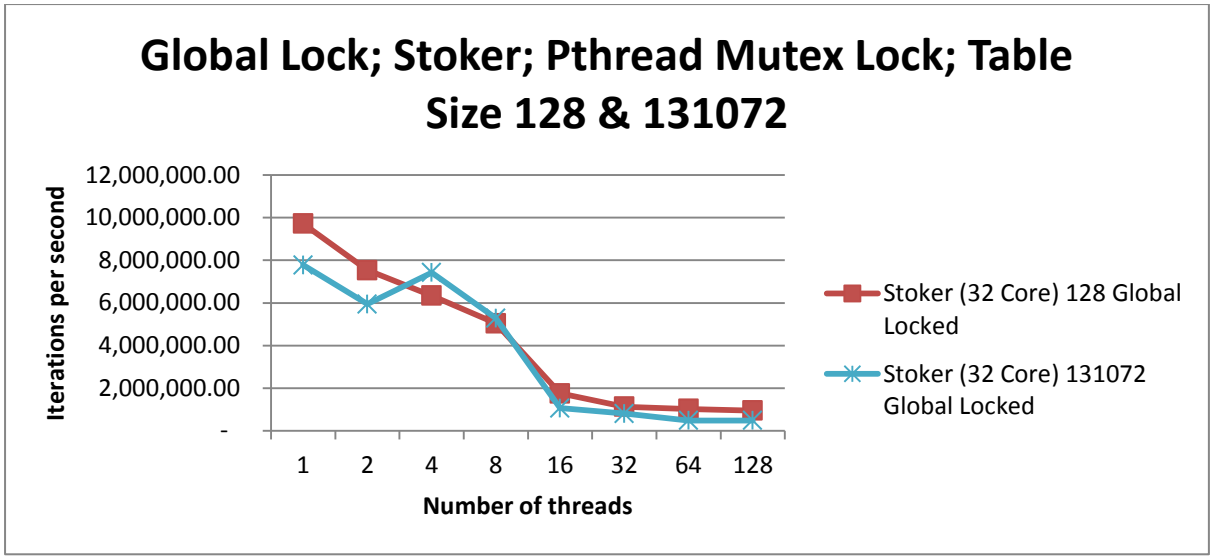


Figure 47

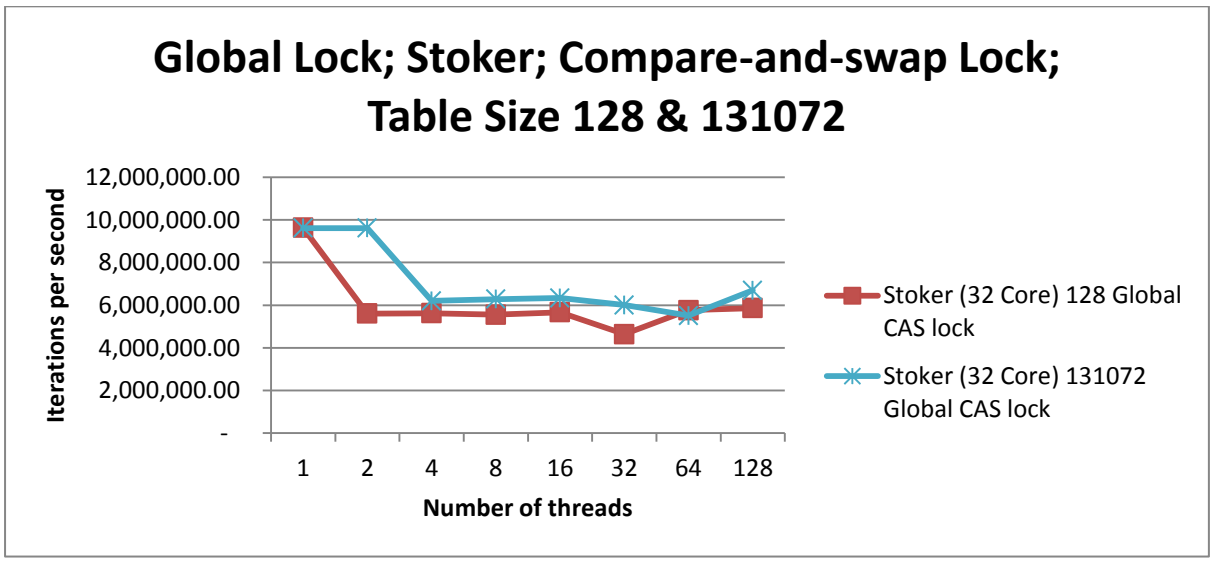


Figure 48

4.4.2.4 Test-and-test-and-set Lock Comparison

Beginning with the *global lock* implementation of the hash table, figure 49 shows that the *regular test-and-test-and-set* lock outperforms both of its variations by some margin, maintaining its performance as the thread counts increases.

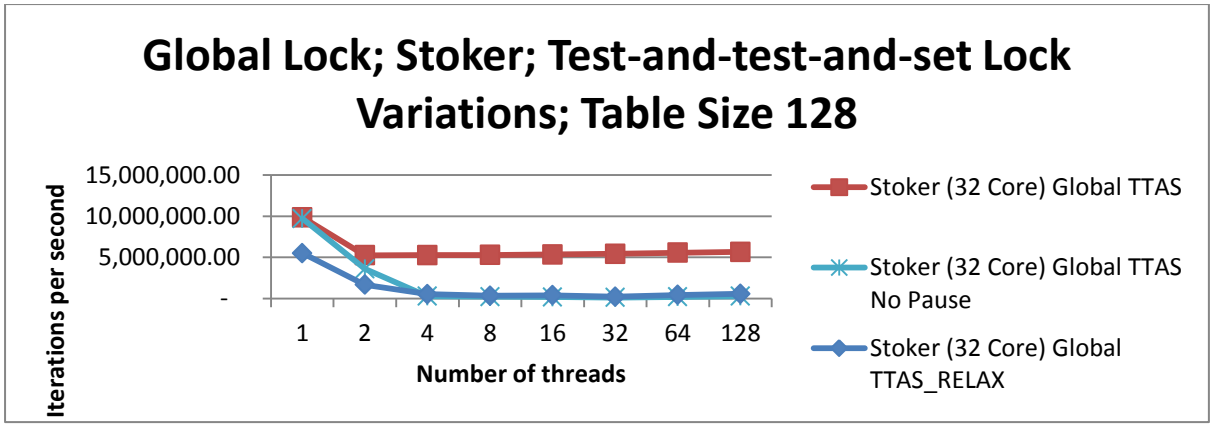


Figure 49

	Cycles	Branches	Branch Misses	Stalled Frontend Cycles	Stalled Backend Cycles
Test-and-test-and-set	7.64E+10	1.40E+10	1.59E+06	4.65E+10	3.47E+10
Test-and-test-and-set-no-pause	1.96E+12	9.43E+10	5.99E+07	1.77E+12	6.40E+11
Test-and-test-and-set-relax	2.39E+12	2.47E+10	4.34E+07	2.33E+12	2.15E+12

One of the reasons for this performance comes from the regular *test-and-test-and-set* lock's utilisation of its CPU cycles, wasting the fewest out of the three locks shown in table X. Combine that with the lowest number of both cache and branch misses and it is clear why it comes out on top.

4.4.2.5 Test-and-test-and-set Lock Comparison

Figure 50 shows the relative performances of the three *test-and-set* variations when using the *global lock* implementation. The regular *test-and-set* lock produces the best performance across all thread counts, with the *test-and-set-no-pause* and *test-and-set-relax* locks performing almost identically to each other.

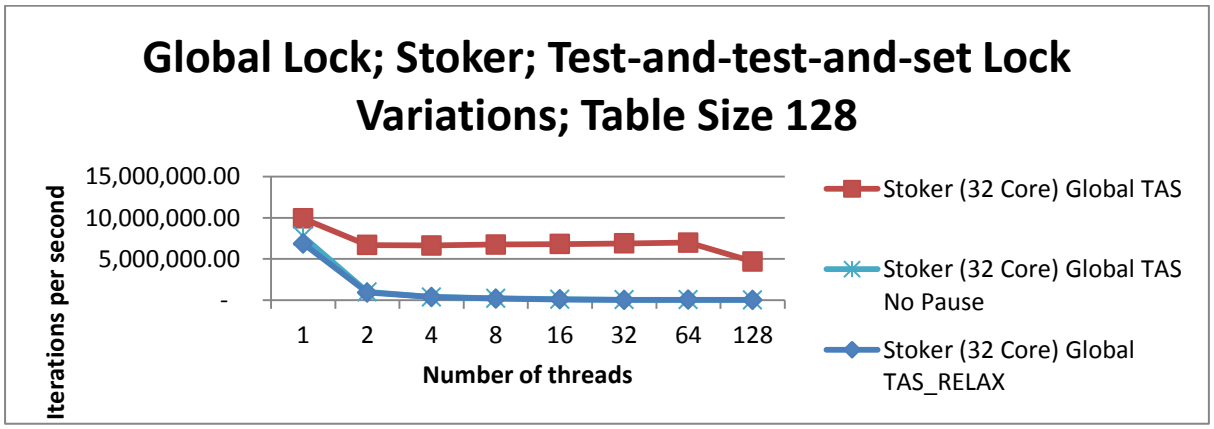


Figure 50

	Cycles	Cache	Cache	Stalled	Stalled

		References	Misses	Frontend Cycles	Backend Cycles
Test-and-set	8.92E+10	2.93E+07	1.73E+07	5.31E+10	4.11E+10
Test-and-set-no-pause	2.67E+12	3.51E+08	2.84E+08	2.66E+12	2.60E+12
Test-and-set-relax	2.69E+12	2.35E+08	2.01E+08	2.67E+12	2.49E+12

From table X it can be seen that the *test-and-set* lock has a clear cut advantage over the other two variations. A lower cache miss rate, combined with a far higher CPU cycle utilisation rate gives the regular *test-and-set* lock a healthy performance margin.

4.4.2.6 Do the variations' performances changes with different architectures?

To investigate this I first examine the *global lock* variation of the hash table. Seen in figure 49 is the performance of the *pthread mutex* lock on the three different architectures of Stoker, Cube and Local. Stoker and Cube have performance patterns which are roughly similar, however, the Local Machine does not follow this trend.

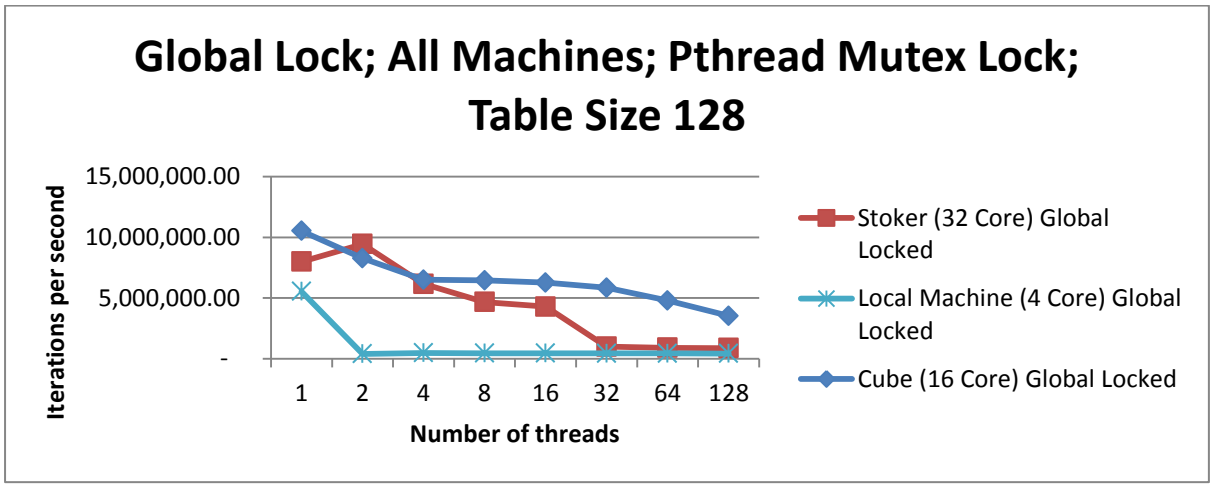


Figure 51

Figure 50 shows the *global lock* variation again but this time with the *test-and-test-and-set* lock. If the first data item with a thread count of one is ignored then all three architectures display a similar performance pattern of a straight line, indicating that the *test-and-test-and-set* lock is relatively robust when used with this implementation across different architectures.

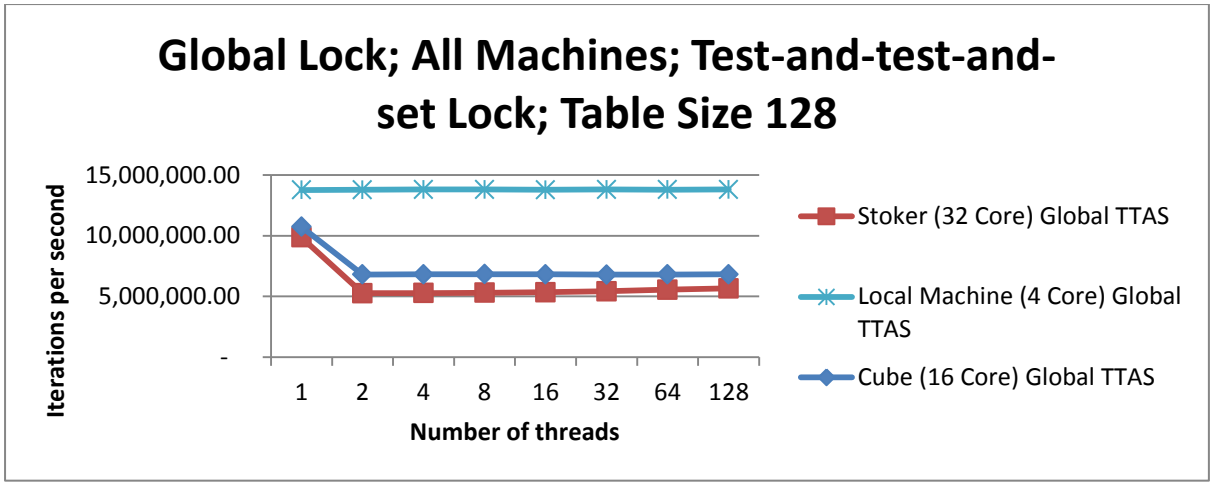


Figure 52

Moving now to the *lock per bucket* variation of the hash table we see in figure 51 that the *pthread mutex* lock is less robust across architectures than with the *global lock* variation, displaying different patterns for all three architectures.

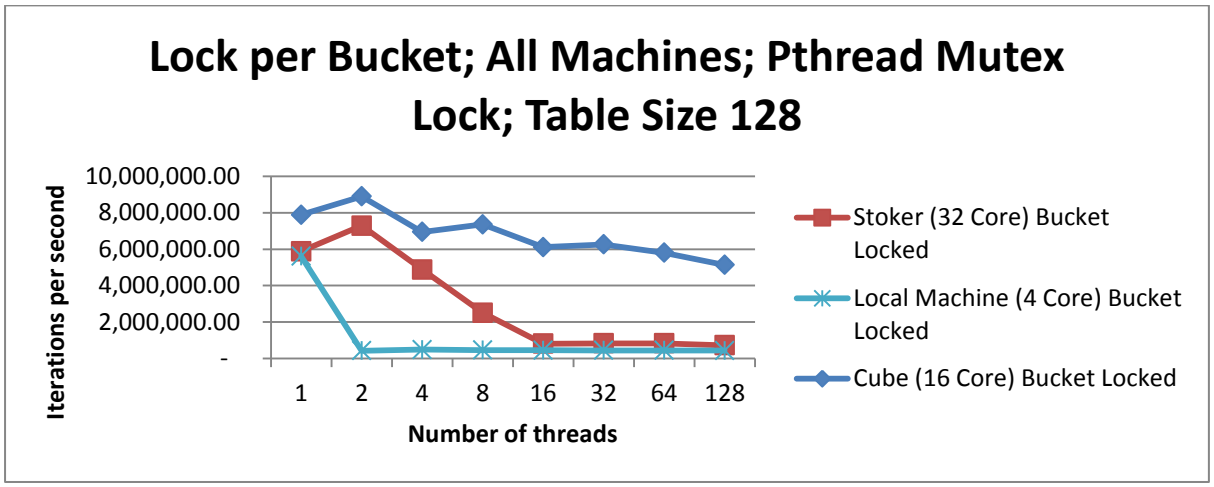


Figure 53

The *test-and-test-and-set* lock also shows less robustness with the *lock per bucket* implementation shown in figure 52. This may indicate that the *lock per bucket* implementation is less effective at preserving performance than its alternative the *global lock* implementation, though Cube and Local Machine do have similar performance patterns.

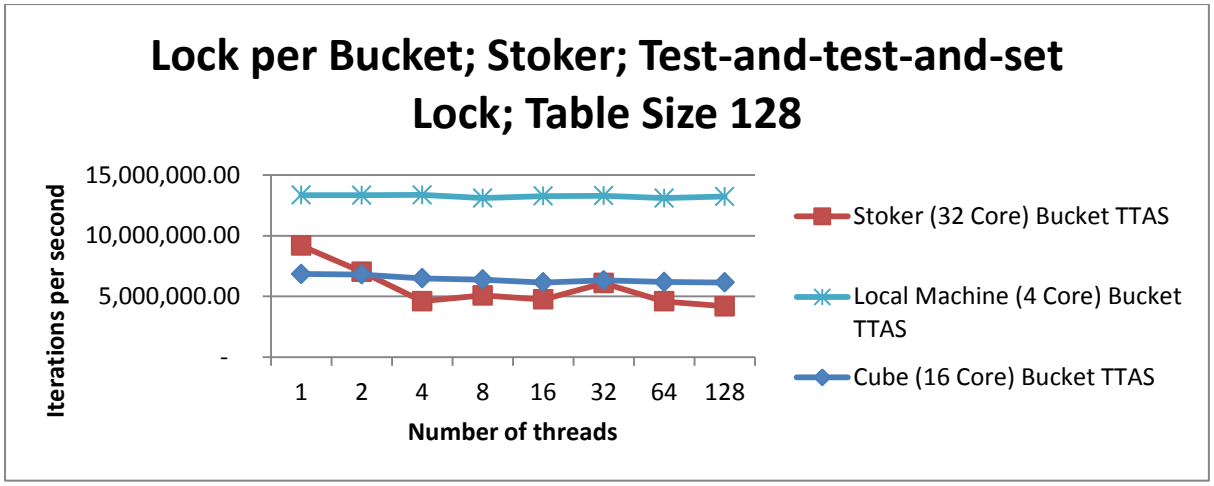


Figure 54

Finally, examining the *lockless* variation of the hash table, in figure 53, each of the architectures produces a different level of performance from the *lockless* implementation, indicating that this hash table variation is perhaps the worst out of the three at preserving performance across architectures.

5 Afterword: (Thin)

5.1 Conclusions

Relate to questions asked in introduction and if they have been answered and what new questions have arisen

The purpose of this project is to determine and compare the differences between concurrent data structure implementations and investigate if the performance of these implementations is maintained across different architectures.

What conclusions can I draw about hardware performance counters? What is the most important attribute to do well in?

5.1.1 Ring Buffer

I have concluded that the best locks for my implementation of the ring buffer are the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* locks based on their performance across the range of thread counts I use.

The regular *test-and-test-and-set* lock using a sleep instruction is the best performing out of itself and the two other variations, *test-and-test-and-set-no-pause* and *test-and-test-and-set-relax* locks.

The same goes for the *test-and-set* lock which out-performed its two variants, *test-and-set-no-pause* and *test-and-set-relax*.

The results gathered to determine if the size of the buffer affects the performance of the implementation is inconclusive. While the buffer size appears to affect the performance of

some locks, such as the *test-and-set* lock it seems to have no impact on other locks such as the *pthread mutex* or the *compare-and-swap* lock.

Finally, I have concluded that while most locks do not retain their performance across architectures, this is not the case for some. Again, the *test-and-set* lock behaves similarly across the three architectures but others are not so robust.

5.1.2 Linked List

5.1.2.1 Singly Linked List

From my tests and analysis I have discerned the following conclusions from my implementations of the *singly linked list*.

As with the ring buffer previously, the best locks used in the *singly linked list* in terms of performance are the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* locks.

The lockless implementation performed well compared to the different locks at lower thread counts but dropped off sharply at once the thread count exceeded four.

Out of the *test-and-test-and-set* variations the regular *test-and-test-and-set* lock utilising the *sleep* instruction proved to be the best in terms of performance at all thread counts.

Size heavily impacts the performance of the *singly linked list* due to the increased time threads have to spend traversing the list when adding and removing nodes.

Finally, as with the ring buffer, an implementation's robustness across architectures depends on the details of the implementation with some implementations maintaining performance across architecture while others do not.

5.1.2.2 Doubly Linked Buffer

The following are conclusions I have gathered from implementing and testing the doubly linked buffer.

Best Locks - CAS, TAS, TTAS

Lockless matches locks in performance until 4 then drops hard

TTAS best variation

TASWP best variation

CAS best variatiomn

TICKET_RELAX better

Size makes a difference to some, not all, especially lockless

Mutex performs similarly on different architectures, CAS and lockless do not

5.1.2.3 Singly Linked Buffer

Best Locks - TTAS, TAS, CAS, locked

Lockless performs similarly to pthread mutex lock, outperformed by TTAS

TTAS best

TASWP best

CAS best

Ticket relax outperforms ticker early

None of the locks tested performed similarly on architectures

5.1.3 Hash Table

5.2 Future Work

How my work could be improved and developed. What disadvantages are there in my approach (lack of memory management) etc.

Could have used perf while implementing designs to ensure best performing implementations

Could have done more analysis with more time.

More comparisons

Used perf on a thread count per thread count basis

6 Bibliography & Appendix: (Thin)

6.1 References:

Atomic Operations Library. (2013). *Atomic operations library.* Available: http://en.cppreference.com/w/cpp/atomic. Last accessed 17/04/2014.

Blaise Barney, Lawrence Livermore National Laboratory. (2013). *POSIX Threads Programming.* Available: https://computing.llnl.gov/tutorials/pthreads/. Last accessed 17/04/2014.

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

Linux profiling with performance counters. (2013). *Linux profiling with performance counters.* Available: https://perf.wiki.kernel.org/index.php/Main_Page. Last accessed 17/04/2014.

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

sleep. (1996). *sleep(3) - Linux man page.* Available: http://linux.die.net/man/3/sleep. Last accessed 17/04/2014.

6.2 Appendix: