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 Date

# 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 Dr. Mélanie 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:

The purpose of this project is to determine and compare the differences between concurrent data structure implementations and investigate whether 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 lockless 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 on 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 whether 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.

By performing the tests and analysis on the concurrent data structures that I implement I hope to shed more light on this area of computer science.

## 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, 2008]

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, 2001]

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, 1993]

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, 2001]

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, 2009]

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, 2011]

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

# 3 Method

## 3.1 Overview

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 whether 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.

As mentioned previously in the 'Background' section of this 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*. This allows me to quickly change which data structure implementation I am using without having to open the relevant file and modify the code.

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 whether 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 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 is not equal.

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. In addition I give an example of the code used to implement each lock.

### 3.2.1 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.

```
pthread_mutex_lock(&lock);//Acquire Lock

//Perform Work

pthread_mutex_unlock(&lock);//Release Lock
```

### 3.2.2 Test-and-test-and-set lock (TTAS)

The *test-and-test-and-set* 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. 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 & Shavit, 2008].

```
do{
        while(lock.load() == 1)sleep();//While lock is taken sleep

}while(lock.exchange(1));//Attempt to acquire lock

//Perform Work

lock = 0;//Release lock
```

### 3.2.3 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 add 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.

```
do{
        while(lock.load() == 1);//Check that lock is still taken

}while(lock.exchange(1));//Attempt to acquire lock

//Perform Work

lock = 0;//Release lock
```

### 3.2.4 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 [*Intel Corporation*, 2011]. 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 whether the intrinsic gives this mode an advantage over the sleep instruction.

```
do{
        while(lock.load() == 1)_mm_pause();//Check that lock is still taken

}while(lock.exchange(1));//Attempt to acquire lock

//Perform Work

lock = 0;//Release lock
```

### 3.2.5 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. 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 & Shavit, 2008].

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()*.

```
while(lock.exchange(1))sleep(PAUSE);//Attempt to acquire lock

//Perform Work

lock = 0;//Release lock
```

### 3.2.6 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.

```
while(lock.exchange(1));//Attempt to acquire lock

//Perform Work

lock = 0;//Release lock
```

### 3.2.7 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.

```
while(lock.exchange(1))_mm_pause();//Attempt to acquire lock

//Perform Work

lock = 0;//Release lock
```

### 3.2.8 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 & Shavit, 2008].

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.

```
int delay = MIN_DELAY;

while(true){

        if(lock.compare_exchange_strong(0, 1))break;//Attempt to acquire lock

                sleep(rand() % delay);//Sleep on failure
```

```
            if(delay < MAX_DELAY)delay = 2 * delay;
}

//Perform Work

lock = 0;//Release lock
```

### 3.2.9 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.

```
while(true){

    if(lock.compare_exchange_strong(0, 1))break;//Attempt to acquire lock
}

//Perform Work

lock = 0;//Release lock
```

### 3.2.10 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.

```
while(true){

    if(lock.compare_exchange_strong(0, 1))break;//Attempt to acquire lock

        _mm_pause();//Pause on failure
}

//Perform Work

lock = 0;//Release lock
```

### 3.2.11 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. 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 [Herlihy & Shavit, 2008]. 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.

```
int myTicket = ticket.fetch_add(1);//Increment ticket value

while(myTicket != nowServing)sleep(myTicket - nowServing);//Sleep if not served

//Perform work

nowServing++;//Move onto next thread in queue
```

### 3.2.12 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.

```
int myTicket = ticket.fetch_add(1);//Increment ticket value

while(myTicket != nowServing)_mm_pause();//Pause if not served

//Perform work

nowServing++;//Move onto next thread in queue
```

## 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 that I decide to base mine on is located in “Designing Concurrent Data Structures”. It describes a design for a concurrent queue which utilises a head and tail pointer to allow for parallel execution and uses 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. In addition, I construct my ring buffer using an array of values, not a linked list as other implementations do. 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 whether the next node is being used and act accordingly.

#### 3.3.1.1 Locked Implementation

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 [sfuerst, n.d] 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 ring 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 Implementation

For my lockless implementation of the ring buffer, I decide to implement a *single producer – single consumer* model. To push an item onto the buffer, 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 an item off the buffer, 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 find this to be a good introduction to the C++11 atomic library as I am 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.

### 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 with 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 has 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 on the *List* class.

This variation contains three functions, *add, remove* and *printList*. The *add* function works by randomly generating a value using the *rand()* function [*rand(3) – Linux man*

*page*, n.d]. It then creates a node using this key and attempts to add it to the list of nodes.

To begin, the *add* function gets the current value of the head variable. If the head is equal to *NULL* then a list does not yet exist, so it creates a node and points the head pointer to the newly generated node.

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 point 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 to it. In this sense the pointer 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 *next* pointer to the larger node and by getting the node previous to the larger node in the list 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 acts as the value that it searches for and tries to remove from the list. Firstly the *remove* function takes a copy of the head of the list and checks whether 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 *remove* function is searching for then the function 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 the node to be deleted in the list is changed so that it points the next node in the list after the node to be deleted. Now nothing is pointing to the node for deletion so it can be removed safely.

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 in the list the function prints out their *key* value followed by a comma to distinguish between node's *key* values.

#### 3.3.2.1.1 Locked Implementation

The locked version of this variation is similar to the locked version of the ring buffer I implement, where any attempt to act on the list requires a thread to acquire the lock, which it then releases once it has completed its work. Since this is a locked variation I do not have to declare the head of the list as an atomic variable, so I am able to simply declare it as volatile which prevents the compiler from applying certain optimisations to it [*volatile (C++)*, 2013]. I declare the *key* attribute of the *Node* class to be volatile for the same reasons, along with any function level variables that deal with the head or *Node* class.

For the *add* function I add in all the different locking modes to acquire the lock before the key value is randomly generated and add in the code to release the lock at the end of the function, ensuring that only one thread can access the body of the function at any one time.

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

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

#### 3.3.2.1.1 Lockless Implementation

I decide to base my *lockless* implementation of this variation of the linked list on the atomic instruction *compare-and-swap* and its associated functions in the C++ 11 atomic library. To do this I need to declare at least one atomic variable to call the necessary functions in the atomic library so I choose the head pointer of type *Node*. I do this because having an atomic head pointer allows me to atomically change the head of the list from node to node. For this implementation I decide to remove the volatile keywords from the code and see whether it makes a difference to the validity of the data structure.

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

It is at this point that I come to a point of interest in the code. I am unsure how to proceed with writing the code for atomically traversing the list so I decide to implement it serially and observe what happens. I then run the code several times

and, to my surprise, the list it created is ordered with no duplicates and appears to work for all intents and purposes. I repeat this procedure for the *remove* function which is 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 are the same.

To try and force an error from my implementation I change the maximum number of nodes allowed in a list to be five nodes and run the program. Such a small list should encounter a high level of thread contention considering the number of threads acting on it at any given time and yet no errors are found in the lists that are produced.

#### 3.3.2.2 Doubly Linked Buffer

##### 3.3.2.2.1 Locked Implementation

After implementing the singly linked list and observing some of the data that is being gathered I notice that once the list starts to get long, past one thousand nodes in length, the performance drops off significantly. I believe that this is due to the time being spent by the threads traversing the list looking for an insertion point or a node to delete.

I feel that this is not optimal, as the size of the list is interfering with the comparison of the data structure implementations. Hence, I decide to remove the traversal issue all together and implement a *multi-consumer multi-producer* linked list buffer. This works by always adding and removing from the head and tail respectively. There is no traversal of the list necessary and while this does mean that the list is no longer ordered or free from duplicates, it is my opinion that this provides clearer data from the different implementations.

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

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

#### 3.3.2.2.2 Lockless Implementation

To implement the *lockless* version of the doubly linked buffer I start off by declaring the new tail pointer as an atomic object. This allows me to atomically remove objects from the end of the list as the atomic head pointer allows me to add items on to the front of the list.

Adding objects involves generating the node to be added then atomically switching the head pointer from what it is pointing at to the new node being added. Since the list no longer has to be traversed, the process of adding a node in a *lockless* fashion becomes much simpler.

Removing a node is much the same as adding a node but in reverse, where the tail pointer is 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 is then discarded.

#### 3.3.2.3 Singly Linked Buffer

It is only after I finish implementing the doubly linked buffer that I realise that I do not need the second pointer for each node if I simply rearrange the placement of the head and tail pointers. If I swap the head and tail pointers around then I only need one pointer per node to implement the data structure. It works as follows: the tail pointer keeps track of the oldest node in the list. Whenever a new node is added, the last node to be added is then pointed to this node and the head then points to this new node, making it the new head of the list. It is in essence, flipping the pointer placement of the doubly linked buffer around but this change reduces the complexity and size of the data structure as now, each node again only needs to store one pointer, as with the singly linked list and all the code that is added to deal with the second pointer can be removed.

In terms of implementation the singly linked buffer is very similar to the doubly linked buffer with the only real differences being that there are no longer any references to a *Node's prev* pointer as that has been removed and the locations of the head and tail pointers are swapped.

### 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 am 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". It describes the design of a concurrent hash table which uses lockless linked lists as buckets, exactly how I want to design mine. I also draw from "Resizable, Scalable, Concurrent Hash Tables" when I implement the *resize* function.

For the locked version of the hash table, I decide to have two implementations. The first implementation, *globally locked* involves locking the entire hash table using a lock whenever a thread wants to interact with the table. The second implementation *lock per bucket* differs from the first in that there is no global lock, but instead each bucket has its own lock. So whenever a thread wishes to interact with a specific bucket, it obtains the bucket's lock and perform its work, in this way it allows for the absence of a global lock and instead has a more modular approach which I then compare to the first locked implementation. To ensure that each of the program's iterations is equal I create and delete the hash table for each run of the program.

#### 3.3.3.1 Globally Locked Implementation

The premise for the *globally locked* hash table is simple, I want a baseline to compare my other two implementations to, the lockless and *lock per bucket* variations. In addition, I feel that it is useful to get the *add* and *remove* functions working and tested in this implementation before moving onto the more complex variations.

As this is a baseline implementation, I decide to go for a very basic locking strategy, where a lock is acquired before a thread interacts with the table, and that the lock is global, so that only one thread can interact with the hash table at any given time, any other thread that attempts to interact with the table is blocked by the lock.

#### 3.3.3.2 Lock per Bucket Implementation

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

I feel that this implementation is more complex than the *globally locked* variety, specifically with regard to the number of locks present in the implementation as now multiple threads are locking and unlocking buckets concurrently.

#### 3.3.3.3 Lockless Implementation

For designing the *lockless* hash table I make the following decisions based on research done with regards to lockless hash tables; it is a closed addressing hash table, each index in the table points to a linked list which is referred to as a bucket, so any collisions result in a node being added onto the relevant bucket. Finally, the hash table has a coarse-grained *resize* function, which involves transferring the buckets to a new, larger hash table [Triplett *et al*, n.d].

To represent the buckets I decide to use the lockless linked list I have already implemented, as it is already tested and I know that it works. I decide to go with my singly linked buffer implementation of the linked list to eliminate traversing the buckets as an issue. I give each bucket two atomic variables, a head and tail pointer, to allow me to atomically add and remove nodes concurrently.

As I am implementing this data structure I run in to two points of interest. The first is that as the program runs, it sometimes posts extremely low results for one of the iterations, usually the iteration with a thread count of four. To try and discern the cause of this I add in a counter that tracks the failure rate of the atomic instructions in both the *add* and *remove* functions, but this proves to not be the cause of the problem as the resulting values I get are all quite low. The counter reported that no more than fifty failures per iteration were occurring indicating that the atomic instructions are not to blame for the drop in performance I am observing. I decide to focus on other aspects of my project with the intention of coming back to the issue.

The second point of interest I encounter is that the program occasionally causes a segmentation fault while it is running at high thread counts. The fault usually occurs around a thread count of thirty two, though sometimes the thread count is lower. As the problem's frequency seems to increase at higher counts my first thought is that it might be a thread contention issue. After reviewing the code, I notice that I am accessing the hash table frequently during both the *add* and *remove* function. I believe that this may be the cause of the segmentation faults, as if a thread is halfway through an *add*, another thread may change the value of the hash, so that the pointers may no longer be pointing to the same bucket in the hash table. I try to solve this by instead passing the bucket reference to a variable, *tmpList* which I then use in the computation. In addition, I add several more checks into my code, ensuring that *tmpList* still points to the place it is supposed to and that if it is not then the operation is aborted as the hash has changed. To test whether the problem has been fixed by my changes I set it to run twenty times, one after another. I do this so that if the problem is not resolved then it should throw a segmentation fault during

one of the twenty runs of the program.  A segmentation fault did not occur during the execution of the program twenty times and I so conclude that my additions to the code have fixed the issue.

#### 3.3.3.4 Hash Function

For my hash function I initially choose to go for a very simply variety. I pass in a value and modulo it with the size of the table. I then return this as the hash. I do consider implementing a more complicated hashing function, however, from what I observe, items appear evenly distributed in my table indicating that the hash function is performing as needed and so I decide to keep my simple implementation.

#### 3.3.3.5 Design & Implementation of the Contains Function

Before I begin testing my hash table I decide that I want to replicate a real world hash table as closely as possible. To do this I add a *contains* function into the code, a function that takes a key and searches for that key in the hash table. I implement this functionality in all three of my hash table variations, the *globally locked*, the *lock per bucket* and the *lockless* implementations. The writing of the *contains* function is relatively easy; I generate a random key, get its hash by passing it through my hash function and then retrieve the bucket associated with that hash. Once the relevant bucket is retrieved I can iterate through it until I either find the key, meaning that the search was successful or the end of the list is encountered, meaning that the key does not exist in the table.

#### 3.3.3.6 Design & Implementation of the Resize Function

To keep search times constant, I have to add in functionality to allow my hash table to resize itself when buckets become too full. I decide to implement a locked *resize* function first, which involves going through each bucket and rehashing every key. The key is then transferred to the new table. I am able to write this part of the implementation serially, as it is only called inside the *add* function, so a lock is already acquired by a thread, making the need for additional locks irrelevant.

For my *lockless* implementation I investigate several potential methods, one of which involves leaving the keys where they are and forming new lists from them by dynamically creating each bucket around them [Triplett *et al*, n.d].

Another option from the same paper is to resize the table in place, where the current table is made bigger and the keys rehashed to redistribute them as necessary.

A third option is to incrementally resize the table, where calls to the *add* function start adding nodes to a new, larger table or transferring them from the old if they already exist. Both the *remove* and *contains* function calls then check both the current table

and the new, larger table. The old table is only discarded when all the keys are transferred from it to the new table.

I decide to implement the first solution as it seems to be the simplest of the three in my opinion. However, I run into difficulty immediately as I am unable to implement a necessary amount of atomicity to stop the threads from interfering with each other. I try several approaches but I am unable to prevent threads from interfering with each other. I run into the same issue with the other two potential solutions and so I decide to implement the locked variety of the *resize* function that I implement in the two locked hash table varieties.

#### 3.3.3.7 Tracking Search Results

As a means to record positive and negative hash table searches conducted by the *contains* function I add in two variables, *pSearches* and *nSearches* to represent the total number of positive and negative searches carried out each time the program runs. I do this because it allows me to test my *contains* function to ensure that it is working correctly. At the end of program execution I then print out the relevant counters.

#### 3.3.3.8 Design & Implementation of the Choose Function

With the addition of the *contains* function in my hash table, I encounter a problem with the structure of my program. Whereas with the ring buffer and linked list there are just two functions, *add* and *remove*, the hash table now has a third function, *contains*. The result of this is that I cannot just assign half of the threads to adding and half to removing items. A better solution is needed. An additional concern is that I want to replicate the function call ratios for hash tables, which are about 90% *contains* calls, 9% *add* calls and 1% *remove* calls [Herlihy & Shavit, 2008]. To solve this problem I create the *choose* function.

The *choose* function works as follows: whenever a thread is spawned, it calls the *choose* function, instead of calling the *add* or *remove* function. Inside the choose function, a number is randomly generated from 0-128. I choose 128 as it is a power of two and so the modulo operation will be changed to a bitwise AND in the compiler. If the number generated is greater than or equal to 12 then the *contains* function is called which roughly equates to 90% of all the numbers generated. Else, if the number is greater than or equal to two then the *add* function is called which roughly equates to 9% of all numbers generated. Finally, if none of the previous two conditions are met then the *remove* function is called which roughly equates to 1% of all numbers generated.

This allows me to generate threads and have them assign themselves work in the correct ratios to replicate the workings of an actual hash table.

# 4 Experiments & Evaluation

## 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 the value 128 as being the maximum number of threads spawned as I originally thought that this was twice the number of cores that 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 as I feel that it will provide data on how the data structure implementations perform as the number of threads begins to greatly exceed the number of cores in the machine.

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 total running time and this is then divided by how many seconds each thread is allowed to run for which produces the iterations per second for each implementation.

I have chosen one second to be the length that each thread should run for. I choose this amount of time as this is the case for many experiments of this type [Triplett *et al*, n.d] 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 implementations.

To ensure that I am collecting accurate data, I am making sure to only collect data from each machine when the 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 version of the code 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. 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 which is time consuming. Each 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 [GCC Administrator, 2001].

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.

The architecture s Intel Ivy Bridge EX 22nm and each of its 32 cores runs at 2.00 GHz [*Intel Xeon Processor E7-4820*, n.d].

#### 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. [*The Internet Society*, n.d].

The architecture is Gainestown 45nm and each of its 16 cores runs at 2.27 Ghz [*Intel Xeon Processor E5520*, n.d].

#### 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 [*Intel Core i5-2500K*, n.d].

### 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. 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.

*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.

*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.

To represent the hardware performance counter data I decide to place it in a table with the type of implementation described on each row and the hardware counter on each column. Since much of the data gathered is in the form of very large numbers I have adopted the following system, whereby these numbers will be shortened in length and a letter representing the scale of the number will be added. "M" is used to represent millions and "B" is used to represent billions. For example the number 85,619,355,140 is shortened to 85.62 B, while the number 31,279,956 is shortened to 31.28 M.

## 4.2 Ring Buffer

### 4.2.1 Evaluation

Apart from the iterations per second and thread count I vary the size of the ring buffer to investigate how this affects the locked and lockless variation. I set the maximum length of the ring buffer array to be 128; I initially had this at 100 however I decided to change it to 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 [Horvath, 2012].

### 4.2.2 Lock Comparisons

I begin my evaluation by first focusing on the performance of the different locked implementations before moving onto the *lockless* version of the code. 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 only interested in how the locked variations 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 during execution. 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 evaluation of the two locks for the remainder of the ring buffer evaluation.

In order to determine which locks have the highest levels of performance on the ring buffer I compare all of the locks with each other. *Figure 1* represents the three best locks out of the twelve tested; the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* locks. As seen in *figure 1*, all three locks perform similarly from the thread count of sixteen and onwards. For earlier thread counts the *test-and-test-and-set* lock appears to have the advantage in terms of performance.

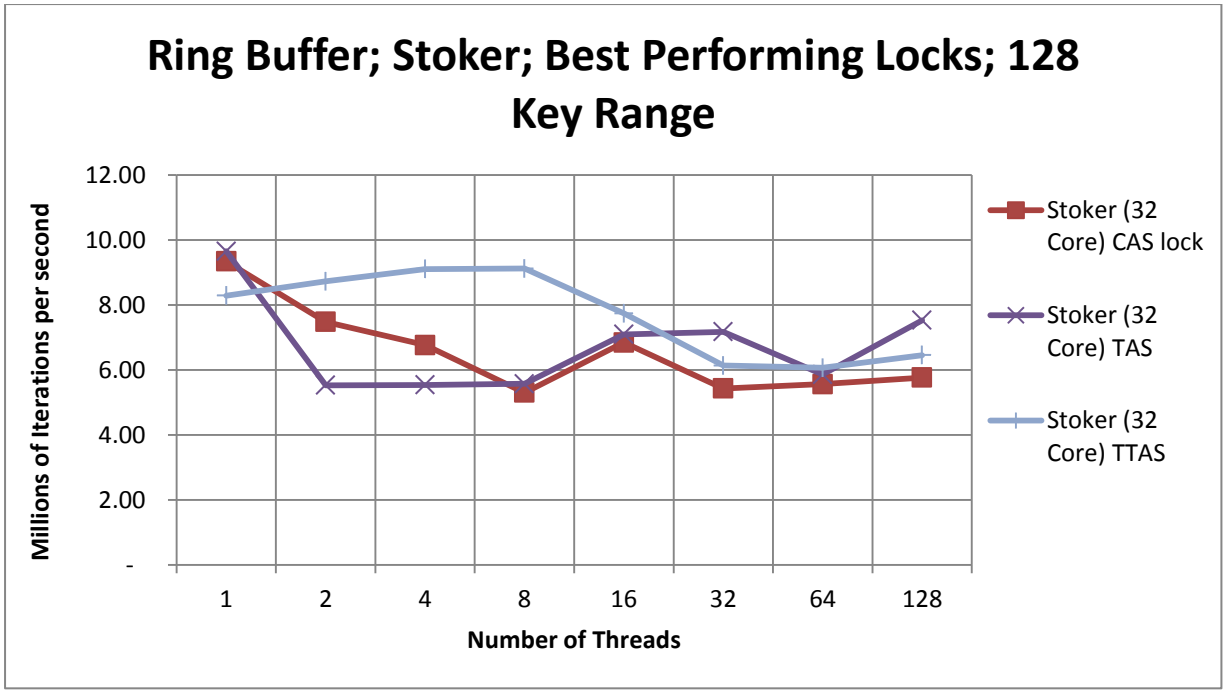


**Figure 1: Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "128 Key Range" indicates the maximum length of the array used in the buffer. "CAS lock", "TAS" and "TTAS" indicate the performance of the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* locks respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Compare-and-swap* | 85.62 B | 31.28 M | 24.67 M | 55.69 B | 46.92 B |
| *Test-and-set* | 79.32 B | 32.51 M | 25.79 M | 51.88 B | 45.5 B |
| *Test-and-test-and-set* | 79.75 B | 20.98 M | 12.11 M | 53.79 B | 35.03 B |

**Table 1: Hardware performance data gathered from the three ring buffer implementations tested in *figure 1*, the *compare-and-swap, test-and-set* and *test-and-test-and-set* locks.**

From *table 1,* each lock has a roughly equal number of cycles, with similar rates of stalled cycles. The *test-and-test-and-set* lock does have a slight advantage over the other two locks from the thread counts of two to sixteen. This can be explained by the fact that the *test-and-test-and-set* lock has a lower rate of cache misses than the other locks and a slightly smaller rate of stalled backend cycles. *Figure 1* shows that for lower thread counts, the *test-and-test-and-set* lock is the lock of choice to implement, however, at later thread counts all three locks perform quite well.

### 4.2.3 Lockless Comparison

Since the lockless ring buffer implemented is a *single-producer-single-consumer* data structure it cannot be compared to the locked variations. The implementation only spawns two threads in total, one to push items onto the queue and one to pop them off. As a result the *lockless* implementation can only be compared to itself. For this I run it three times, modifying the maximum length of the ring buffer's array each time. This is to investigate whether there is a relationship between the size of the ring buffer's array and the performance of the *lockless* implementation. From *Figure 2* we can see that the size of the buffer has a minor impact on the performance of the *lockless* implementation.

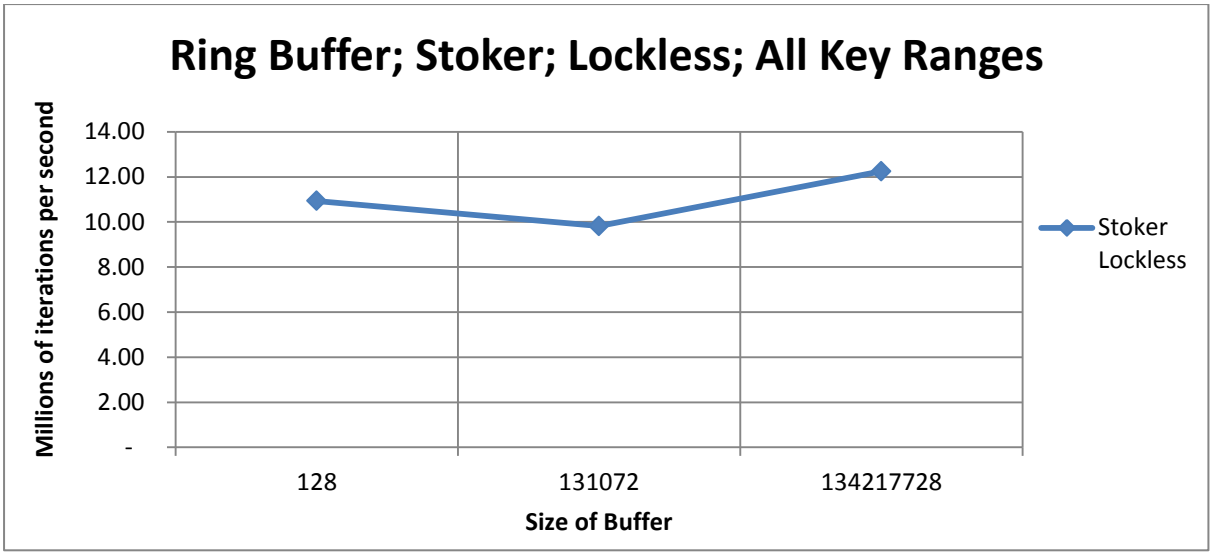


**Figure 2: Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "All Key Ranges" indicates that all three key ranges, 128, 131072 and 134217728, are represented in this graph. "Lockless" indicate the performance of the *lockless* implementation of the ring buffer.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| Stoker 128 | 15.231 B | 4.27 M | 3.85 M | 10.141 B | 7.287 B |
| Stoker 131072 | 16.01 B | 8.84 M | 8.24 M | 10.574 B | 7.884 B |
| Stoker 134217728 | 18.566 B | 18.15 M | 17.12 M | 14.2 B | 11.825 B |

**Table 2: Hardware performance data gathered from the three ring buffer lockless implementations tested in *figure 2*. The *lockless* ring buffer with an array of maximum length 128, 131072 and 134217728.**

*Table 2* shows that as the size of the buffer increases, so does the number of CPU cycles used as well as the number of cache references performed. Interestingly, the rate of cache misses stays relatively equal between the three implementations. It would appear that the size of the buffer does affect the performance of the ring buffer, though a larger size is not necessarily better as the performance does dip at the second size before rising with the third size.

### 4.2.4 Test-and-test-and-set Lock Comparison

I want to investigate the relative performance of the different *test-and-test-and-set* lock implementations of the ring buffer. It can be seen in *figure 3* that the *test-and-test-and-set* lock which uses a *sleep()* instruction has the best performance of the three at a thread count of four and onwards. The *test-and-test-and-set-no-pause* appears to have a slight performance boost over the other two implementations when the thread count is two while the *test-and-test-and-set-relax* lock is performs the worst for all thread ranges.

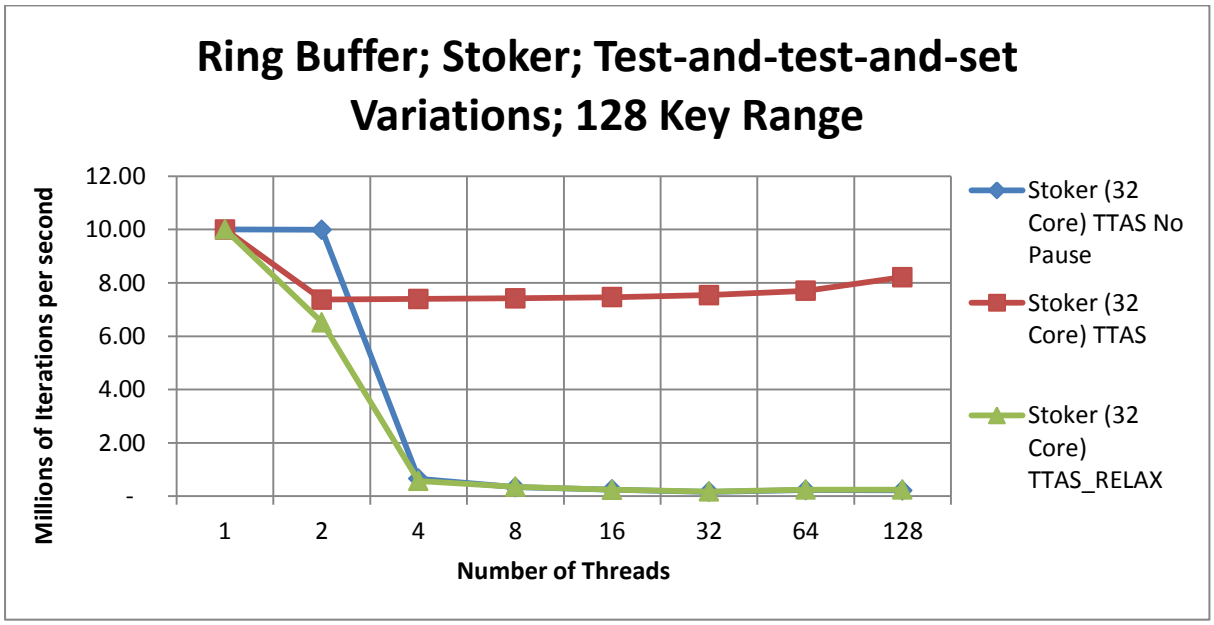


**Figure 3: Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "128 Key Range" indicates that the maximum length of the ring buffer array is 128. "TTAS No Pause", "TTAS" and "TTAS_RELAX" represent the *test-and-test-and-set-no-pause,* the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles |
|---|---|---|---|---|
| *Test-and-test-and-set* | 79.186 B | 13.94 M | 4.71 M | 54.076 B |
| *Test-and-test-and-set-no-pause* | 2387.58 B | 375.38 M | 234.27 M | 2208.663 B |
| *Test-and-test-and-set-relax* | 2410.88 B | 309.4 M | 199.46 M | 2348.28 B |

**Table 3: Hardware performance data gathered from the three ring buffer *test-and-test-and-set* implementations tested in *figure 3*. The *test-and-test-and-set-no-pause,* the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations**

From *Table 3,* the regular *test-and-test-and-set* lock implementation uses far fewer CPU cycles than the other two variations. In addition, it has far fewer cache references, and 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 of the lock, which constantly invalidates cache lines, causing more bus traffic and more cache misses as a result [Herlihy & Shavit, 2008]. Finally the *test-and-test-and-set* lock has the lowest proportion of stalled frontend

cycles meaning that it wastes the fewest CPU cycles on the frontend of the CPU pipeline.

This indicates that with a low thread count of two, it is better not to have any kind of sleep or pause instruction as the thread contention is low enough for the constant polling of the lock to matter. However, as the number of threads grows so does the contention between them and we see in *figure 3* how much of a difference a *sleep()* instruction can make as the *test-and-test-and-set* lock performs steadily as the thread count increases while the other two implementations drop drastically in performance.

### 4.2.5 Test-and-set Lock Comparison

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. It can be seen from *figure 4* that the *test-and-set* lock is the best performing implementation once the thread count reaches four and onwards whereas the other two implementations drop off sharply.

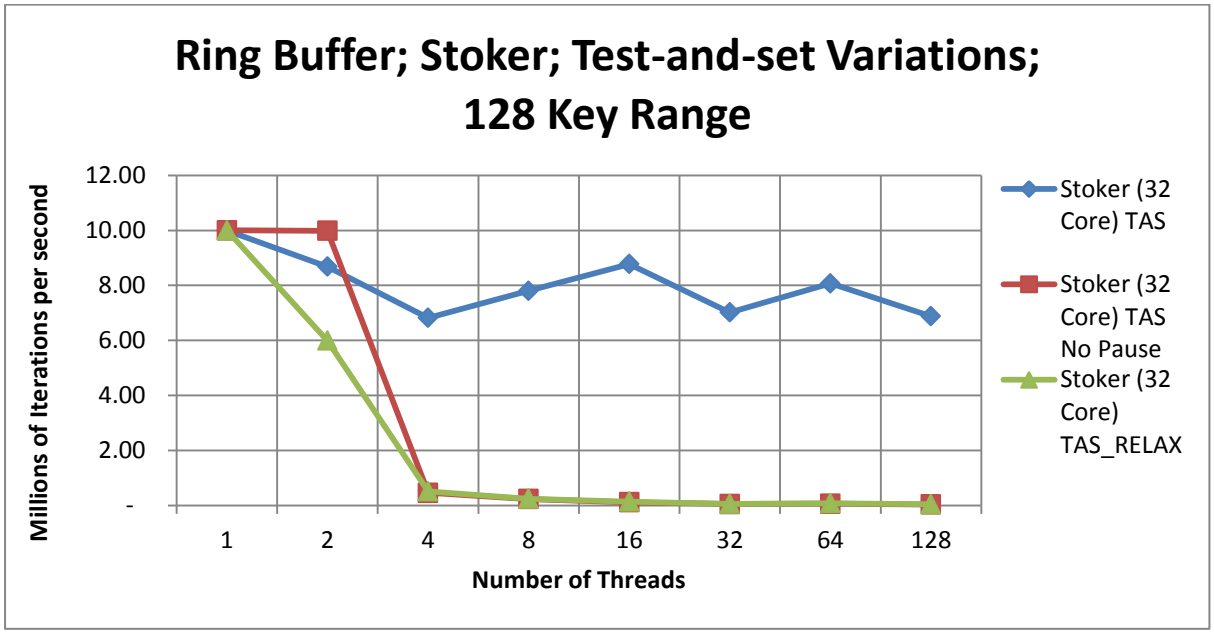


**Figure 4: Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "128 Key Range" indicates that the maximum length of the ring buffer array is 128. "TAS", "TAS No Pause" and "TAS_RELAX" represent the *test-and-set, the test-and-set-no-pause* and the *test-and-set-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Test-and-set* | 89.787 B | 6.51 M | 1.33 M | 59.32 B | 44.995 B |
| *Test-and-set-no-pause* | 2658.295 B | 204.92 M | 183.85 M | 2645.12 B | 2581.133 B |
| *Test-and-set-relax* | 2666.912 B | 225.39 M | 207.15 M | 2648.995 B | 2479.6 B |

**Table 4: Hardware performance data gathered from the three ring buffer test-and-set implementations tested in figure 4. The *test-and-set, the test-and-set-no-pause* and *the test-and-set-relax* lock implementations.**

*Table* 4 shows that 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 stalling over 95% of all their CPU 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 variations.

As shown with the *test-and-test-and-set* lock implementations, it appears to be beneficial to have an implementation with no sleep instruction if the thread count will be at one or two, however, as thread contention becomes higher with increased thread counts the *sleep()* instruction becomes vital in preserving performance as seen in *figure 4* where the *test-and-set* lock implementation performs well at the higher thread counts ,whereas the other two implementations drop in performance very quickly.

### 4.2.6 Ticket Lock Comparison

The final lock I wish to investigate for the ring buffer is the *ticket* lock and its alternate implementation, 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 both confirm that and see whether the *_mm_pause()* intrinsic affects the performance of the implementation in any way. From *figure 5* we see that both locks drop sharply in performance, though somewhat earlier than expected considering that Stoker has 32 cores and both implementations have dropped off significantly in performance by a thread count of four.

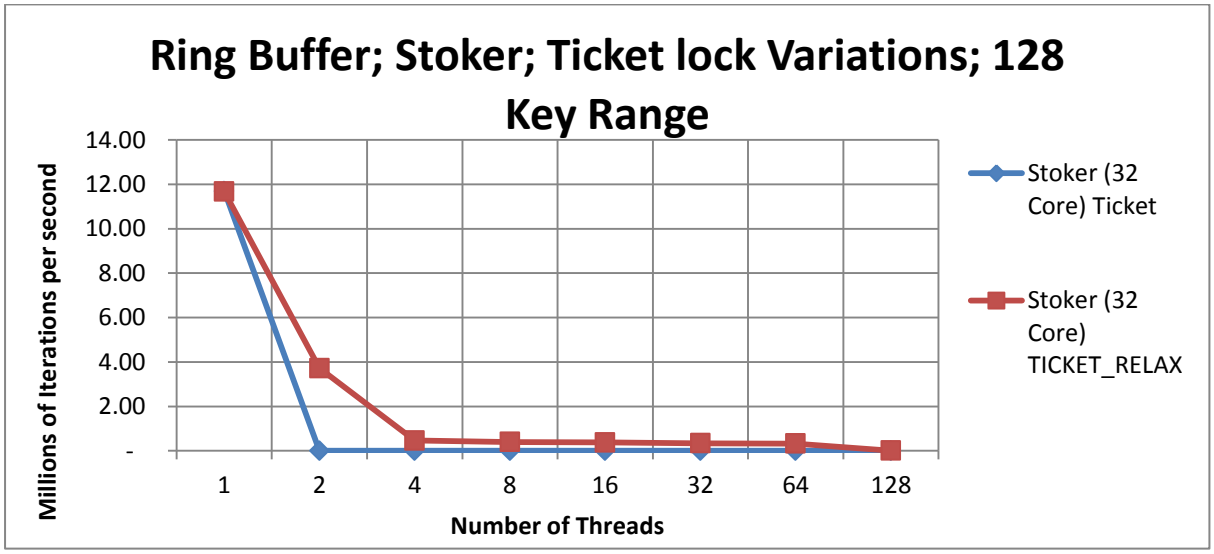


**Figure 5 : Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "128 Key Range" indicates that the maximum length of the ring buffer array is 128. "Ticket" and "TICKET_RELAX" represent the *ticket* and *ticket-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Ticket* | 89.787 B | 6.51 M | 1.33 M | 59.32 B | 44.995 B |
| *Ticket-relax* | 2934.05 B | 399. M | 218.06 M | 2681.776 B | 2282.696 B |

**Table 5 : Hardware performance data gathered from the two ring buffer *ticket* lock implementations tested in figure 5. The ticket *and* ticker-relax lock implementations.**

The two implementations have a huge difference in CPU cycles used as shown in *table 5*, with the *ticket-relax* lock utilising over thirty two times as many CPU cycles. However, this huge increase in CPU use does not appear to correlate to that much of a performance improvement. While the *ticket-relax* lock implementation does not drop off as quickly as the *ticket* lock implementation it still drops off far quicker than expected. Even for a thread count of two the performance is still poor compared to other implementations tested such as the *test-and-set* lock implementations and so I would be cautious about utilising this implementation of the *ticket* lock unless its performance can be improved.

### 4.2.7 The Size of the Ring Buffer Array & Performance

It has already been shown in the *lockless* implementation of the ring buffer that the size of the array in the ring buffer can have a minor impact on the performance of the data structure as a whole. Hence, I will now evaluate the *pthread mutex*, *test-and-test-and-set* lock and the *compare-and-swap* lock to investigate whether this relationship carries over to the locked implementations of the ring buffer.

From *figure 6, 7* and *8* it is shown that while there is 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 with some sizes performing better in some implementations but then performing worse in others. Therefore, the impact that size has on the performance on an implementation should be conducted on a case by case basis and a larger size should not be assumed to bring an increase in performance.

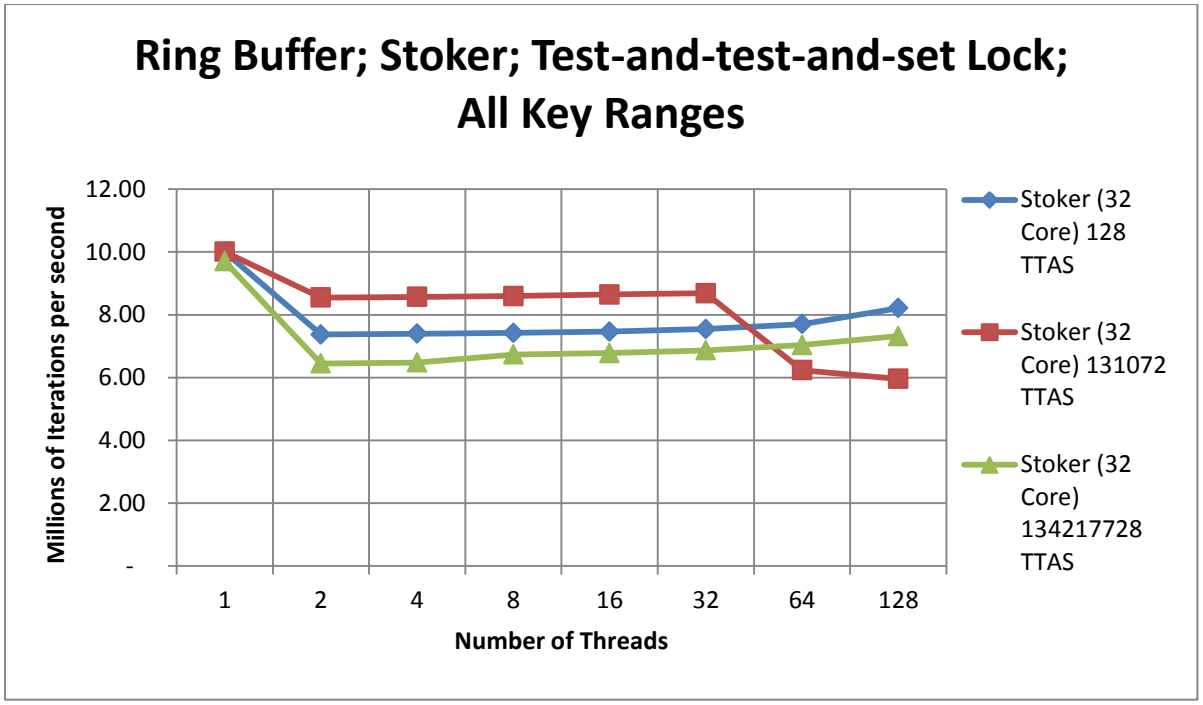


**Figure 6: Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "All Key Ranges" indicates that all three key ranges, 128, 131072 and 134217728 were used as the maximum length of the ring buffer array. "128 TTAS", "131072 TTAS" and "134217728 TTAS" represent the *test-and-test-and-set* lock implementations and which sized buffer was used when they were tested respectively.**

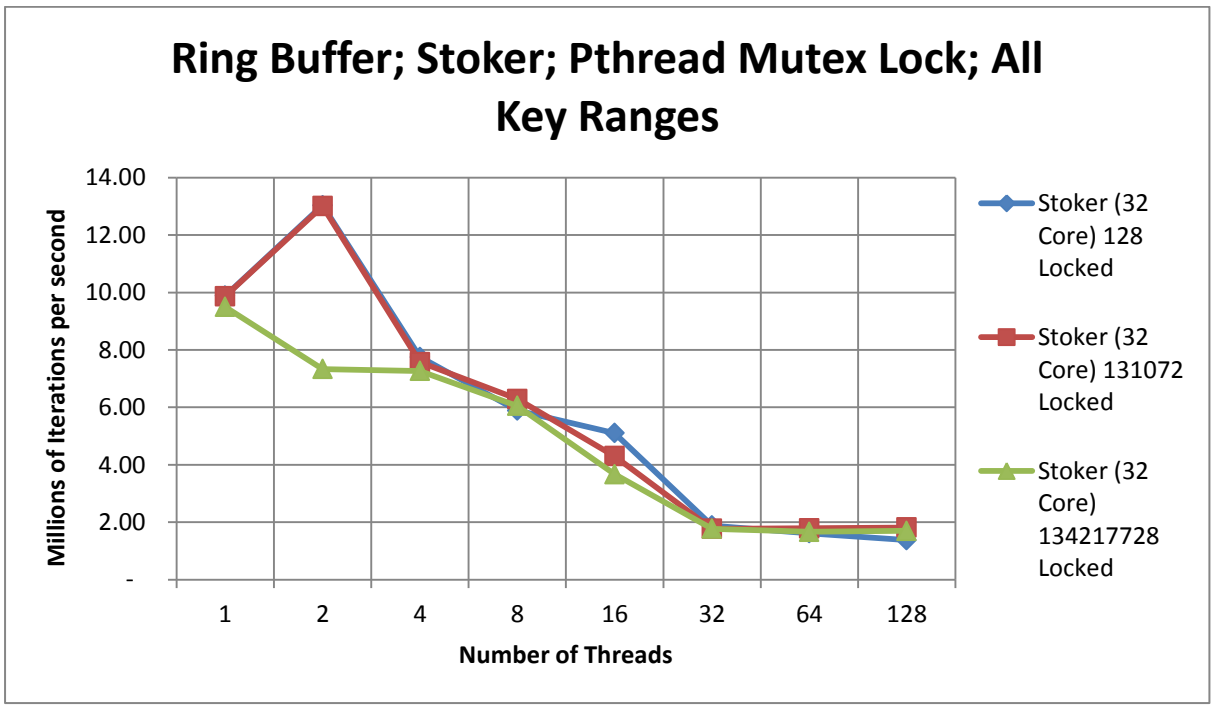


**Figure 7: Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "All Key Ranges" indicates that all three key ranges, 128, 131072 and 134217728 were used as the maximum length of the ring buffer array. "128 Locked", "131072 Locked" and "134217728 Locked" represent the *pthread mutex* lock implementations and which sized buffer was used when they were tested respectively.**

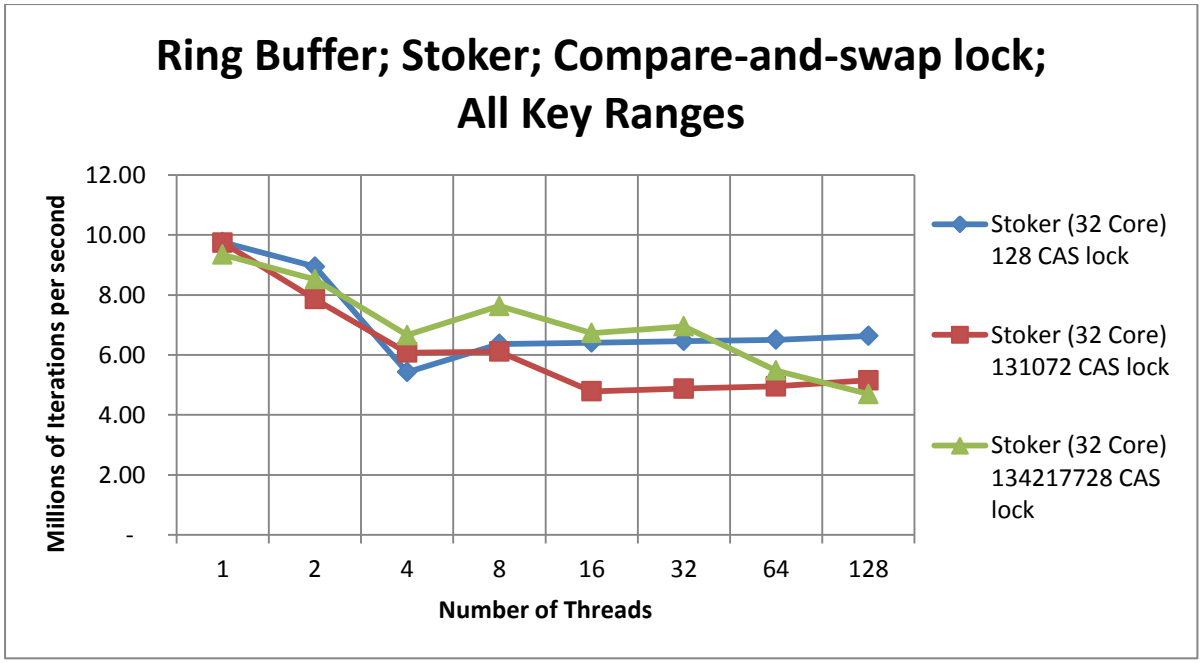


**Figure 8: Millions of iterations/second by number of threads for the ring buffer on the machine Stoker. "All Key Ranges" indicates that all three key ranges, 128, 131072 and 134217728 were used as the maximum length of the ring buffer array. "128 CAS lock", "131072 CAS lock" and "134217728 CAS lock" represent the *compare-and-swap* lock implementations and which sized buffer was used when they were tested respectively.**

### 4.2.8 The Ring Buffer's Performance across Architectures

For the final piece of evaluation on the ring buffer I will now investigate whether or not the locked and lockless implementations of the ring buffer 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 different architectures. *Figure 9, 10* and *11* each represent one lock over the three architectures of Stoker, Cube and Local Machine.

The results show that some locks are more robust across architectures than others. For example, the *test-and-set* lock's performance in *figure 10* has a similar shape across the three architectures while the *pthread mutex* lock in *figure 9* performs quite differently on the three architectures. This indicates that if these implementations of the ring buffer are to be run on different architectures then there performances should be measured on the architectures they are to be run on as their performance cannot be guaranteed.

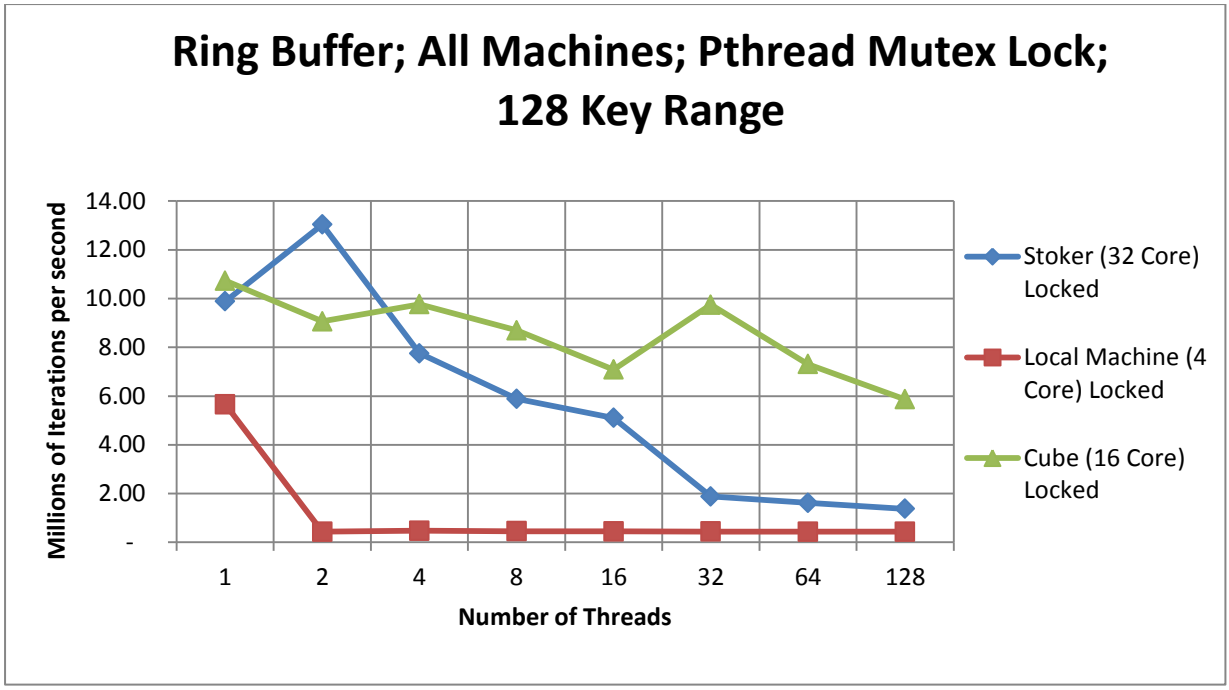


**Figure 9: Millions of iterations/second by number of threads for the ring buffer on the machines Stoker, Cube and Local Machine. "128 Key Range" indicates that the maximum length of the ring buffer array is 128. "Stoker (32 Core) Locked", "Local Machine (4 Core) Locked" and "Cube (16 Core) Locked" represent the *pthread mutex* lock implementation, which architecture the implementation is run on and how many CPU cores the architecture has.**

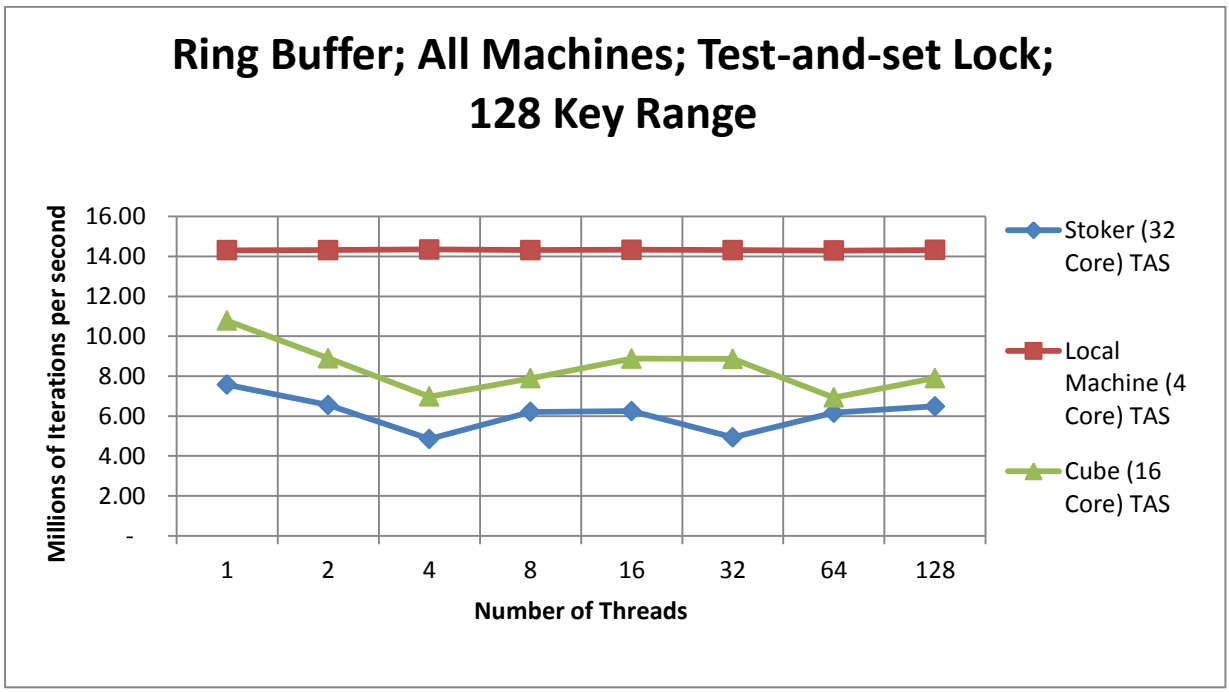


**Figure 10: Millions of iterations/second by number of threads for the ring buffer on the machines Stoker, Cube and Local Machine. "128 Key Range" indicates that the maximum length of the ring buffer array is 128. "Stoker (32 Core) TAS", "Local Machine (4 Core) TAS" and "Cube (16 Core) TAS" represent the *test-and-set* lock implementation, which architecture the implementation is run on and how many CPU cores the architecture has.**

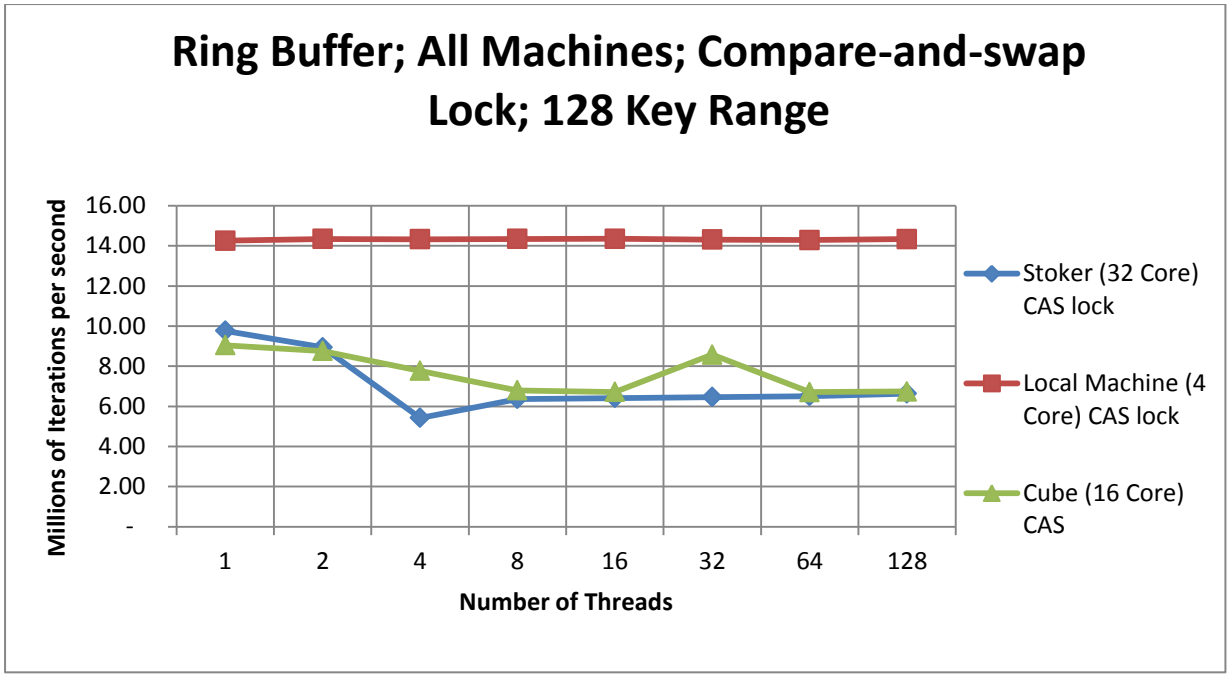


**Figure 11: Millions of iterations/second by number of threads for the ring buffer on the machines Stoker, Cube and Local Machine. "128 Key Range" indicates that the maximum length of the ring buffer array is 128. "Stoker (32 Core) CAS lock", "Local Machine (4 Core) CAS lock" and "Cube (16 Core) CAS lock" represent the *compare-and-swap* lock implementation, which architecture the implementation is run on and how many CPU cores the architecture has.**

## 4.3 Linked List

### 4.3.1 Singly Linked List

#### 4.3.1.1 Evaluation

The maximum length of the singly linked list in terms of nodes is represented by the variable *KEY_RANGE* which I use with the modulo operation and the *rand()* function to produce key values for the nodes in the list. Since the singly linked 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 do this because I want to investigate whether the size of the singly linked list affects the performance of the implementations. By defining a maximum size for the linked list I can make it larger and smaller at will, allowing me to evaluate the relationship between size and performance.

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), 131,072 (2^17) and 134,217,728 (2^27) so that the compiler will replace the modulo calls with a bitwise AND 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.

#### 4.3.1.2 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 locked implementations against each other in order to determine which has the highest performance.

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 this evaluation.

Out of the locked implementations 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 in *figure 12*.

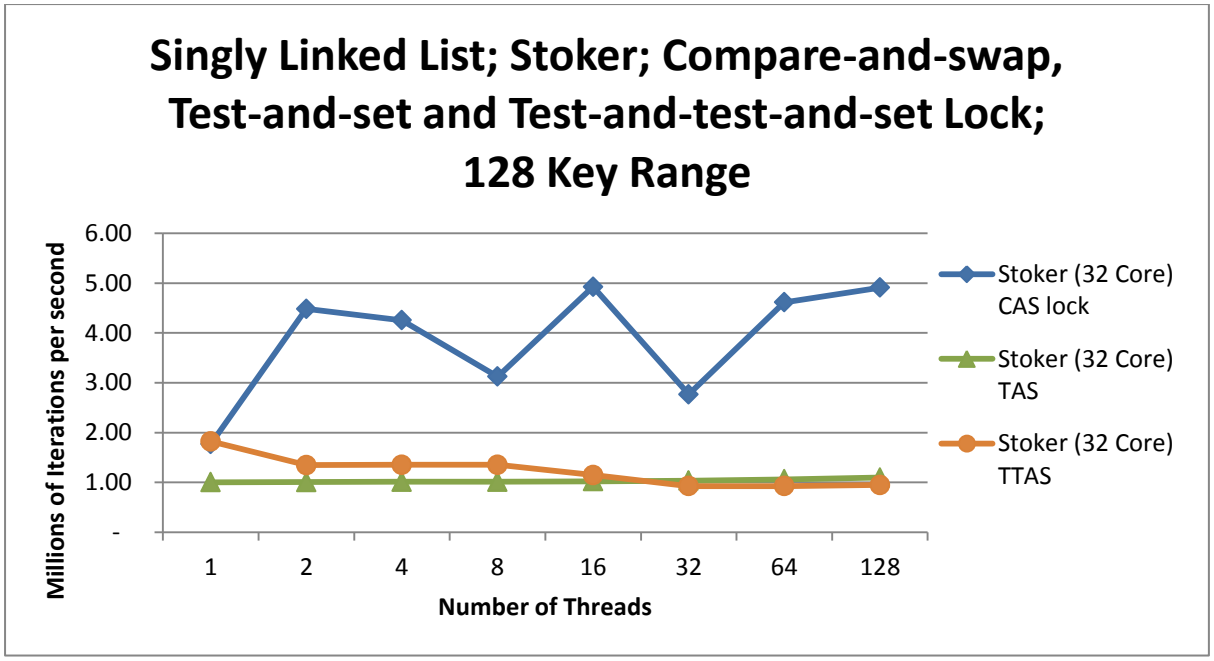


**Figure 12: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "CAS lock", "TAS" and "TTAS" represent the *compare-and-swap*, the *test-and-set* and the test-and-test-and-set lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Branches | Branch Misses |
|---|---|---|---|---|---|
| *Compare-and-swap* | 79.28 B | 25.29 M | 18.17 M | 13.72 B | 50.79 M |
| *Test-and-set* | 73.75 B | 37.97 M | 30.27 M | 12.99 B | 87.69 M |
| *Test-and-test-and-set* | 82.56 B | 36.96 M | 29.46 M | 14.58 B | 85. M |

**Table 6 : Hardware performance data gathered from the three singly linked list locked implementations tested in figure 12. The *compare-and-swap*, the *test-and-set* and the test-and-test-and-set lock implementations.**

Surprisingly, for such a large gap in performance as seen in *figure 12*, the *compare-and-swap* lock does not stand out when the data in *table 6* is examined; 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. The *test-and-set* lock implementation has a better utilisation of its CPU cycles with fewer stalls than the *compare-and*-swap lock implementation but it does have over twice as many branch misses. These could be offsetting the performance gains it should be making.

In comparison, the *test-and-test-and-set* lock does not suffer from a high number of branch misses, having only one fifth as many as the *compare-and-swap* lock implementation. Yet the *test-and-test-and-set* lock is far more wasteful with its CPU cycles, having far more stalled cycles than the other two implementations. This could be the reason that the *test-and-test-and-set* does not display better performance.

This is an interesting case, as I feel that the performance difference displayed between the *compare-and-swap* lock implementation and the other two implementations is too large for what the performance data gathered is revealing. I believe that this case needs further investigation in the future. In any case, the *compare-and-swap* lock implementation appears to be by far the best locked implementation to choose for this type of concurrent data structure, as shown by its impressive performance.

#### 4.3.1.3 Locked vs Lockless Comparison

I now compare the top performing locked implementations from the previous section with the *lockless* implementation of the singly linked list. It can be seen in *figure 13* 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 while the *compare-and-swap* lock implementation performs consistently across thread counts.

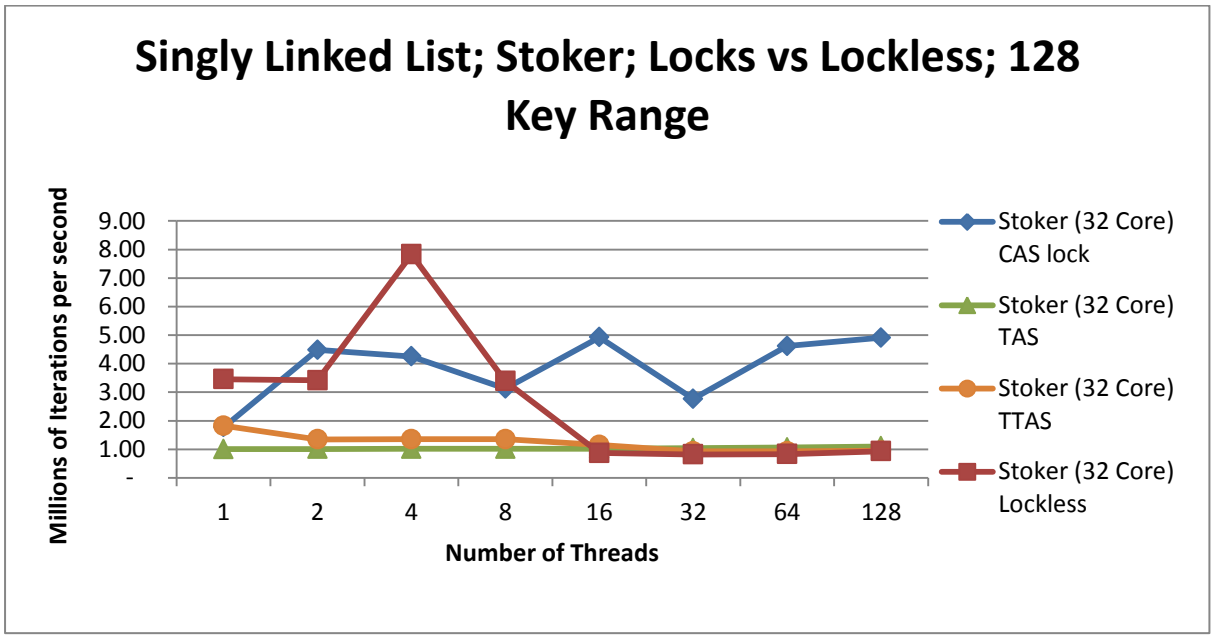


**Figure 13: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "CAS lock", "TAS", "TTAS" and "Lockless" represent the *compare-and-swap* lock, the *test-and-set* lock, the *test-and-test-and-set* lock and the lockless implementations respectively.**

| | Cycles | Cache References | Cache Misses | Branches | Branch Misses |
|---|---|---|---|---|---|
| *Compare-and-swap* | 79.28 B | 25.29 M | 18.17 M | 13.72 B | 50.79 M |
| *Test-and-set* | 73.75 B | 37.97 M | 30.27 M | 12.99 B | 87.69 M |
| *Test-and-test-and-set* | 82.56 B | 36.96 M | 29.46 M | 14.58 B | 85. M |
| *Lockless* | 2320.25 B | 920.91 M | 376.46 M | 85.04 B | 177.46 M |

**Table 7: Hardware performance data gathered from the four singly linked list locked implementations tested in figure 13. The *compare-and-swap* lock, the *test-and-set* lock, the *test-and-test-and-set* lock and the lockless implementations.**

The hardware performance data in *table 7* 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 recorded, 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.

This indicates that the *lockless* implementation is not effective at dealing with high amounts of thread contention but does perform very well at lower thread counts, outperforming the locked implementations by some margin at a thread count of four as seen in *figure 13*. Therefore, the *lockless* implementation should only be used in an environment with low amounts of contention; else the *compare-and-swap* lock implementation should be used as it deals more effectively with thread contention.

#### 4.3.1.4 Test-and-test-and-set Lock Comparison

I will now compare the different variations of the *test-and-test-and-set* locked implementations of the singly linked list. I do this because I want to investigate which implementation performs the best and why that is. The three types of implementation are the *test-and-test-and-set* lock, the *test-and-test-and-set-no-pause* lock and the *test-and-test-and-set-relax* lock. *Figure 14* shows that all of the implementations dip in performance from a thread count of one to two. However, the *test-and-test-and-set* lock implementation then levels out and performs consistently for the remainder of the thread counts. The other two implementations do not follow this trend and continue to drop in performance.

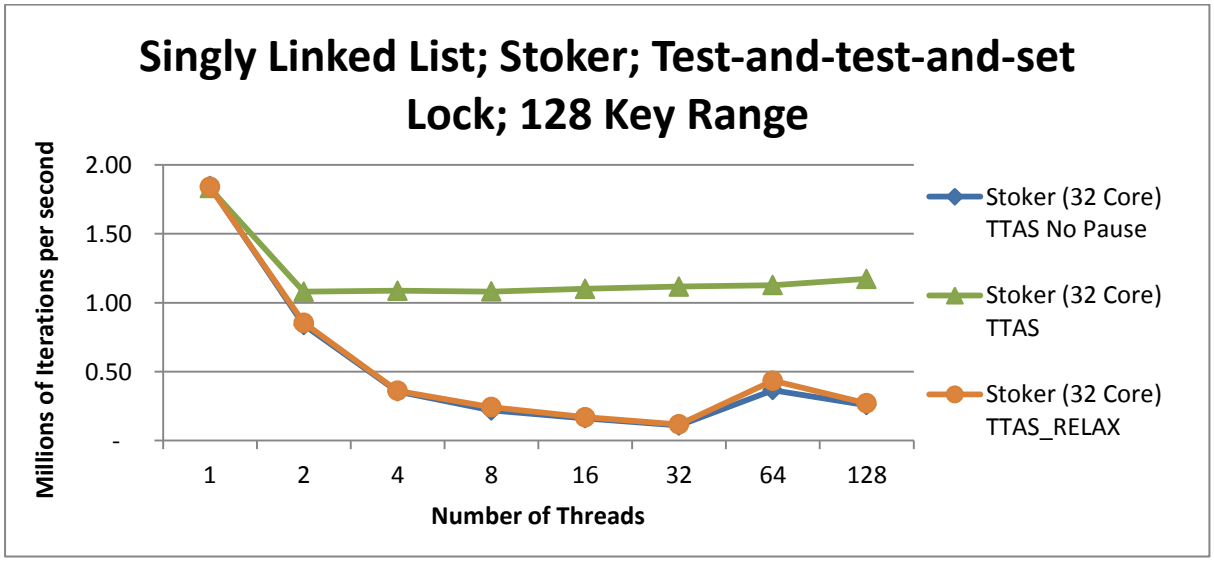


**Figure 14: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "TTAS No Pause", "TTAS" and "TTAS_RELAX" represent the *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Branches | Branch Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|---|---|
| *Test-and-test-and-set* | 81.13 B | 38.56 M | 30.64 M | 14.21 B | 88.34 M | 52.23 B | 12.48 B |
| *Test-and-test-and-set-no-pause* | 2668.94 B | 553.21 M | 279.02 M | 146.42 B | 98.95 M | 2380.56 B | 1268.95 B |
| *Test-and-test-and-set-relax* | 2694.26 B | 471.66 M | 243.42 M | 40.19 B | 91.78 M | 2597.36 B | 2314.94 B |

**Table 8: Hardware performance data gathered from the three singly linked list locked implementations tested in figure 14. The *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations.**

From the data in *table 8* we see that the *test-and-test-and-set* lock has far fewer CPU cycles along with fewer cache references, branches and far fewer stalled cycles both front and backend. These differences allow the *test-and-test-and-set* implementation to make better use of its environment and waste fewer of the resources it is provided. Both the *test-and-test-and-set-no-pause* and *test-and-test-and-set-relax* lock implementations seem unable to deal with thread contention, as shown in *figure 14* by their continuous downwards slope as the thread count increases. There is a slight bump in performance at a thread count of sixty four for a reason I am unable to discern but it is inconsequential.

The data shows that the best implementation out of the three implementations tested is the *test-and-test-and-set* lock implementation due to its ability to handle thread contention at a range of different thread counts. This indicates that the *test-and-test-and-set* implementation should be the only variety considered out of the three tested to be implemented in a concurrent data structure.

#### 4.3.1.5 The Relationship between the Size of the List & Performance

I now investigate whether the maximum allowed size of the singly linked list in terms of nodes has an impact on the performance of the locked and *lockless* implementations. I use the *pthread mutex* lock and the *lockless* implementation for comparison. *Figures 15* and *16*, each show the comparison between their respective implementation with a maximum length of 128 nodes and a maximum length of 131,072 nodes.

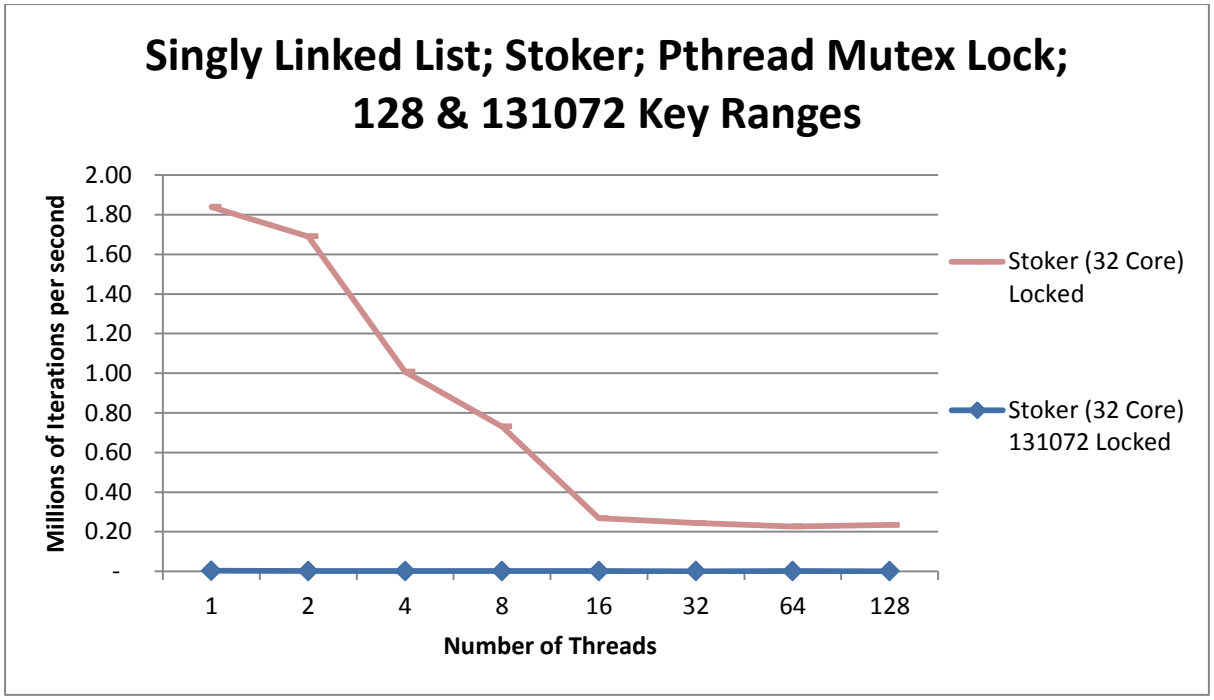


**Figure 15: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 & 131072 Key Ranges" indicates these are the two maximum sizes being compared. "Locked" and "131072 Locked" represent the *pthread mutex* lock and the maximum size of the singly linked list allowed for that implementation, with the default being 128.**

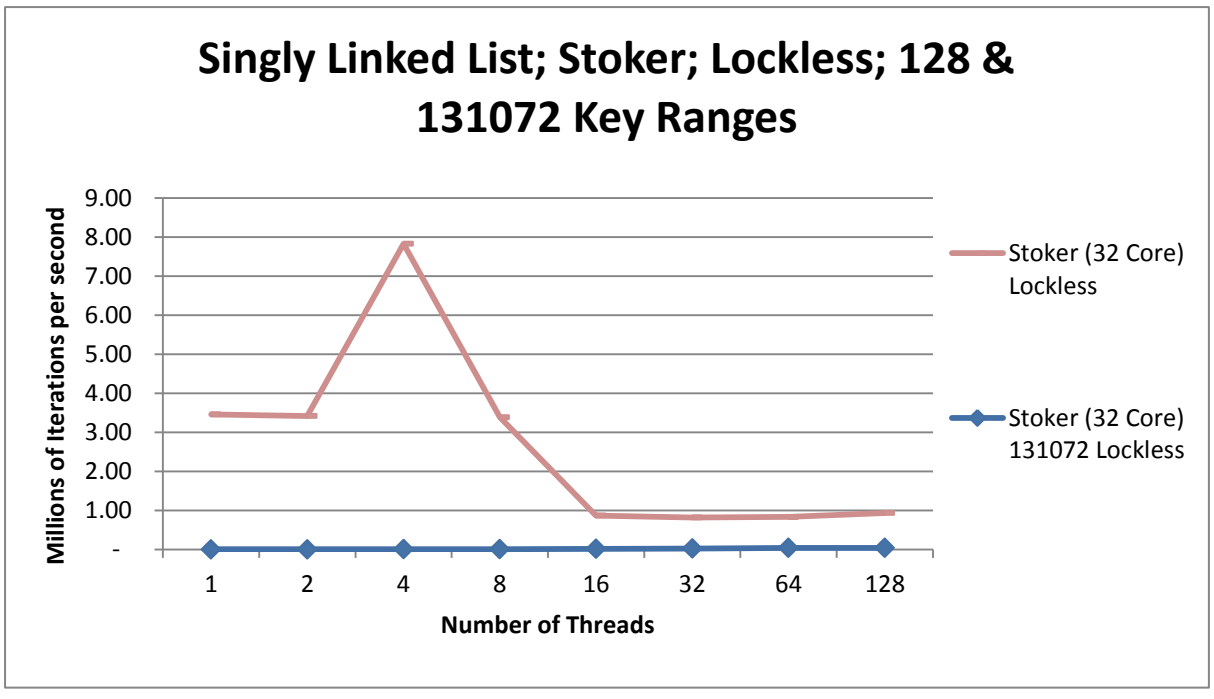


**Figure 16: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 & 131072 Key Ranges" indicates these are the two maximum sizes being compared. "Lockless" and "131072 Lockless" represent the *lockless* implementation and the maximum size of the singly linked list allowed for that implementation, with the default being 128.**

As we can see, in *figures 15* and *16* the performance drop of the singly linked list implementation with the maximum size of 131,072 nodes is staggering when compared to the implementation with a maximum size of 128 nodes. The performance drop is so significant that the performance of the larger list appears as a line at the bottom of each graph. However, the increase in size also drastically changes the performance of the *lockless* implementation, not just in iterations per second, but how it performs at different thread counts as seen in *figure 17*. Compare this to the same implementations, but with a smaller size and the shape of the *lockless* implementation's performance changes as seen in *figure 18*.

The drop in performance between sizes can be explained by the amount of time threads spend traversing the list. The longer the list, the more time threads have to spend looking for the correct insertion point for a node, or searching for a node to delete.

The change in size means that instead of hitting its performance peak at a thread count of four, as seen in *figure 18,* the *lockless* implementation continues to gain in performance all the way to a thread count of sixty four before dipping. This means that if this *lockless* implementation were to be utilised, it should be noted that the longer list the slower the implementation will get as a result of having to traverse the list, however its performance in relation to higher thread counts versus a locked implementation will increase.

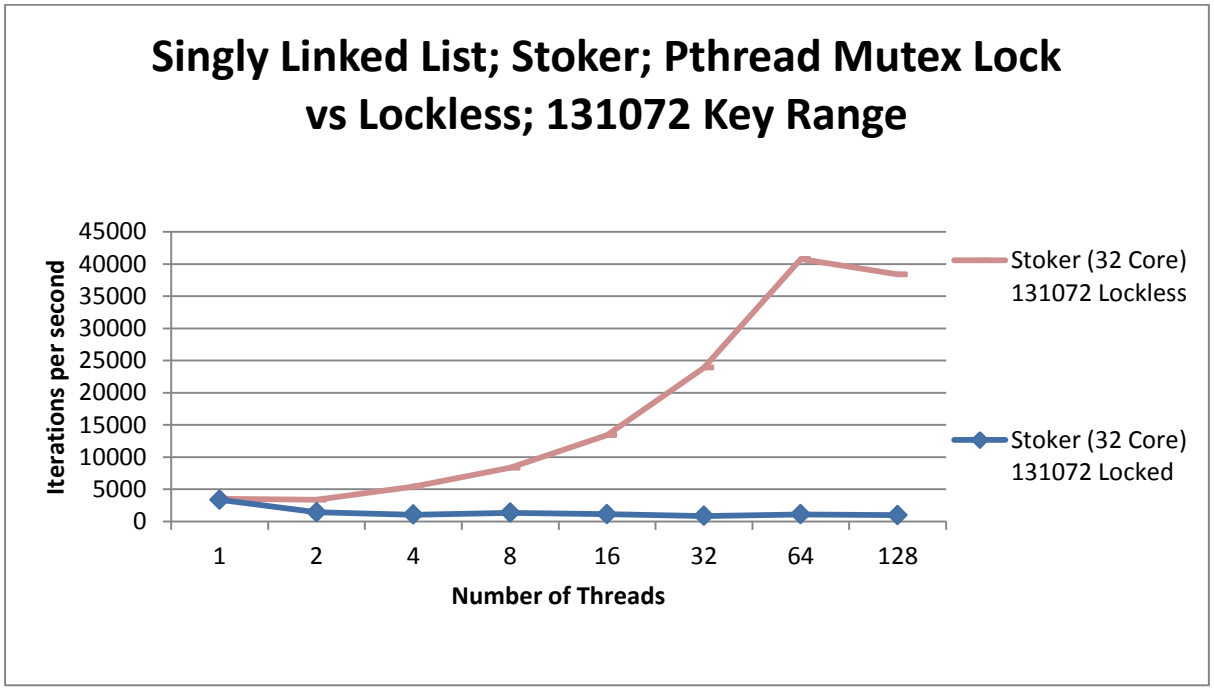


**Figure 17: Iterations/second by number of threads for the singly linked list on the machine Stoker. "131072 Key Range" indicates that this is the maximum number of nodes allowed in the list at any one time. "131072 Lockless" and "131072 Locked" represent the *lockless* and *pthread mutex* lock implementations respectively with the number indicating the maximum allowed size of the list in that implementation.**

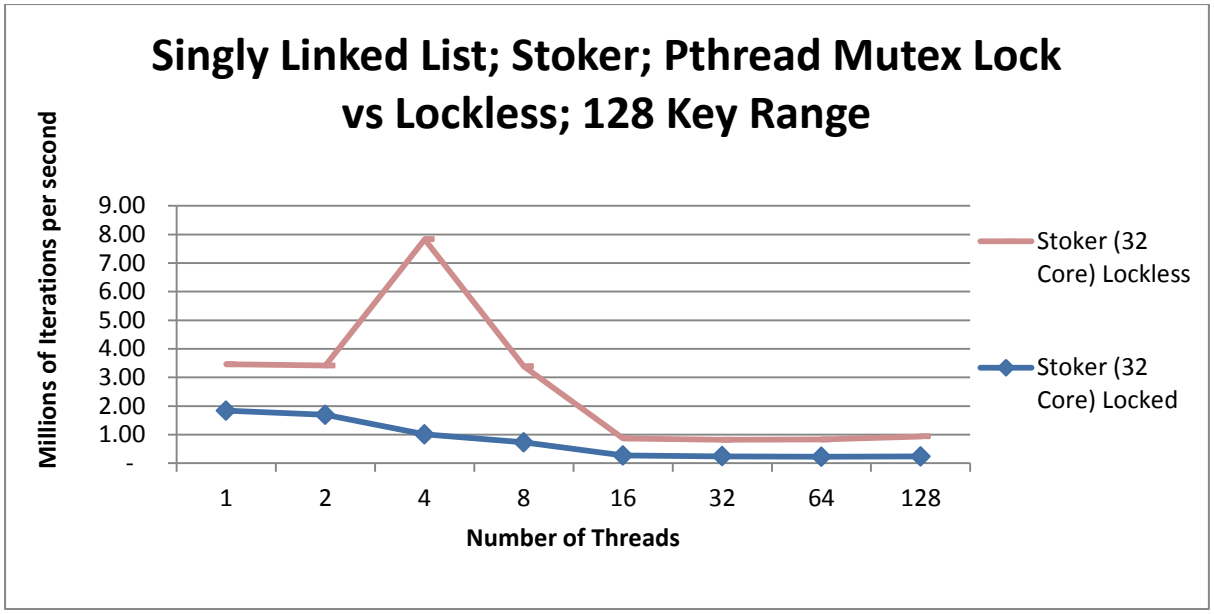


**Figure 18: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128" indicates that this is the maximum number of nodes allowed in the list at any one time. "Lockless" and "Locked" represent the *lockless* and *pthread mutex* lock implementations respectively.**

#### 4.3.1.6 The Singly Linked List's Performance across Architectures

For my final piece of evaluation on the singly linked list I will examine whether the performance of the locked and lockless implementations is maintained across architectures. *Figures 19, 20* and *21*, detail 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.

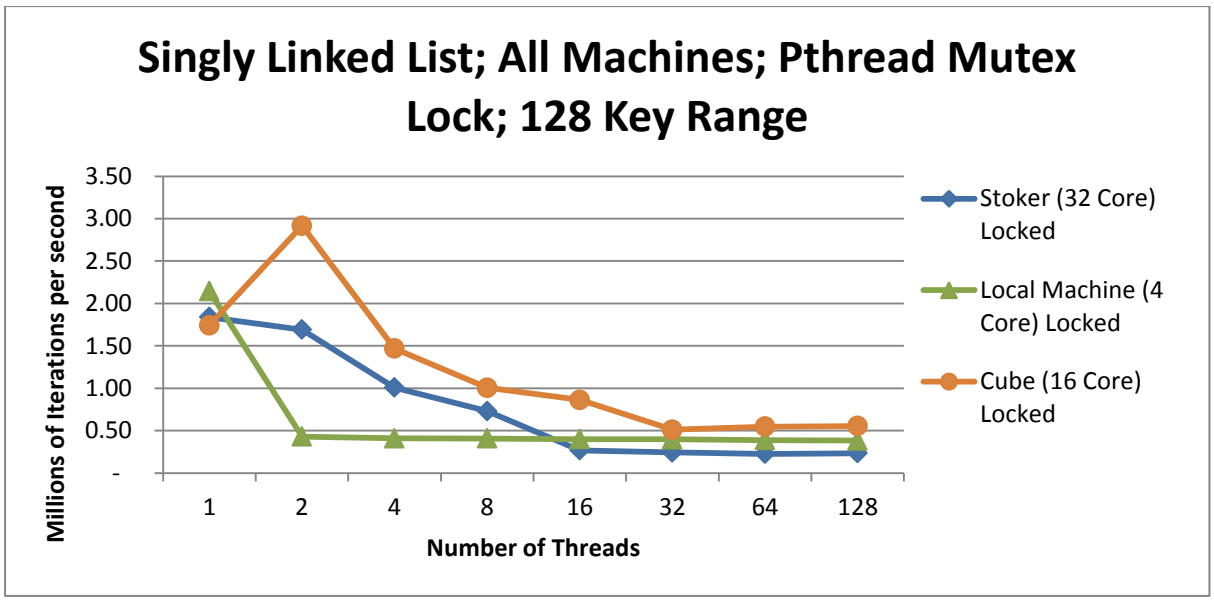


**Figure 19: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "Stoker Locked", "Local Machine Locked" and "Cube Locked" represent the *pthread mutex* lock implementation, with the architecture name representing which architecture the implementation was run on.**

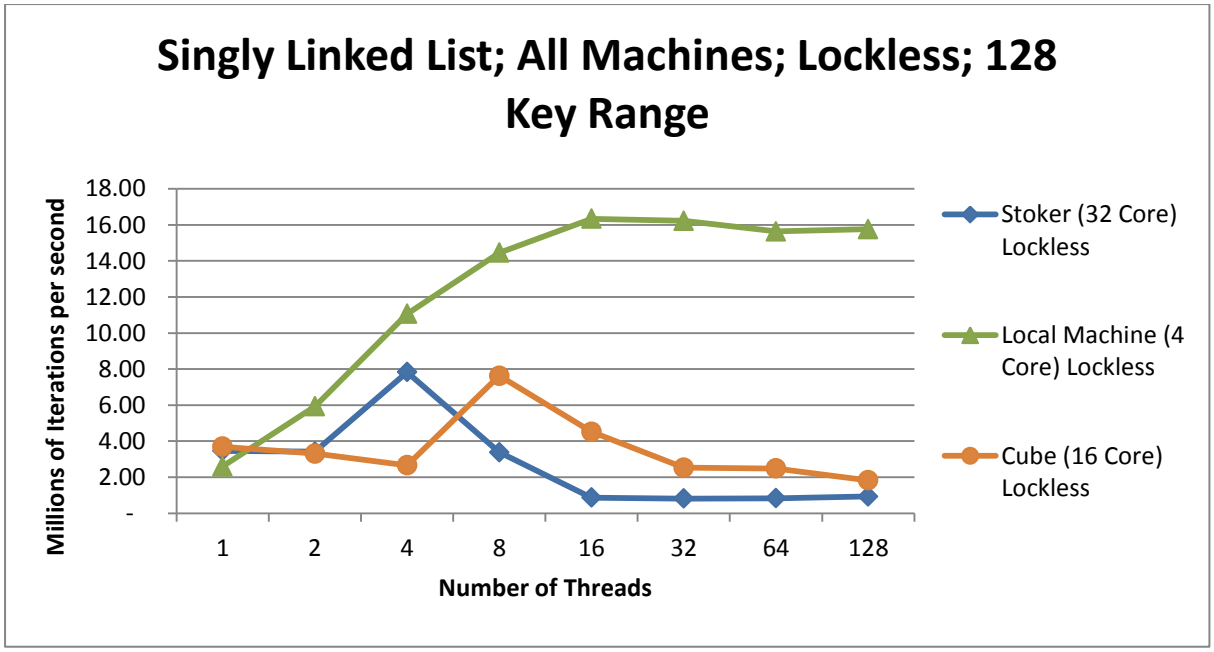


Figure 20: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "Stoker Lockless", "Local Machine Lockless" and "Cube Lockless" represent the *lockless* implementation, with the architecture name representing which architecture the implementation was run on.

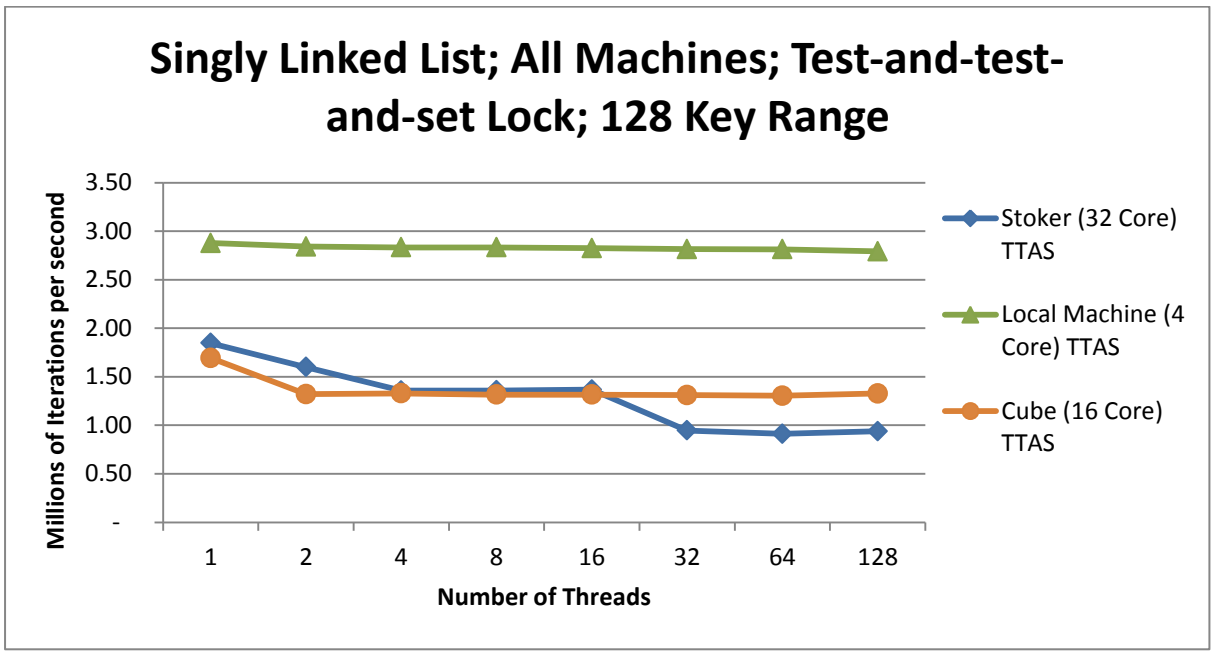


Figure 21: Millions of iterations/second by number of threads for the singly linked list on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "Stoker TTAS", "Local Machine TTAS" and "Cube TTAS" represent the *test-and-test-and-set* lock implementation, with the architecture name representing which architecture the implementation was run on.

We see mixed results with the robustness of the implementation seemingly depending on which implementation is used. For example in *figure 19* we see that the *pthread mutex* lock implementation performs similarly on the three architectures with the same going for the *test-and-test-and-set* lock implementation in *figure 21*. However, the lockless implementation breaks this trend in *figure 20* with the Local Machine's performance acting as a stark contrast to both Stoker and Cube.

This indicates that while some singly linked list implementations may perform similarly on certain architectures, there are some that do not, such as the *lockless* implementation and so any utilisation of this data structure should be tested on its target architecture to ensure that its performance is as expected.

### 4.3.2 Doubly Linked Buffer

#### 4.3.2.1 Evaluation

This variation of the linked list differs 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.

I will also be investigating the performance of the lock variations and studying the robustness of the implementations across architectures.

Finally, the key range for the doubly linked buffer does not modify the maximum number of nodes allowed into the buffer at any given time. It merely controls the maximum value of the keys generated for the nodes and is included for the sake of continuity.

#### 4.3.2.2 Locked Comparison

I want to identify the three most effective locked implementations of the doubly linked buffer so I test all of the locked implementations and select the best performing out of them. The three best locked implementations on Stoker are the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* locks seen in *figure 22*. This mirrors what has been seen from the other data structures tested such as the ring buffer and singly linked list.

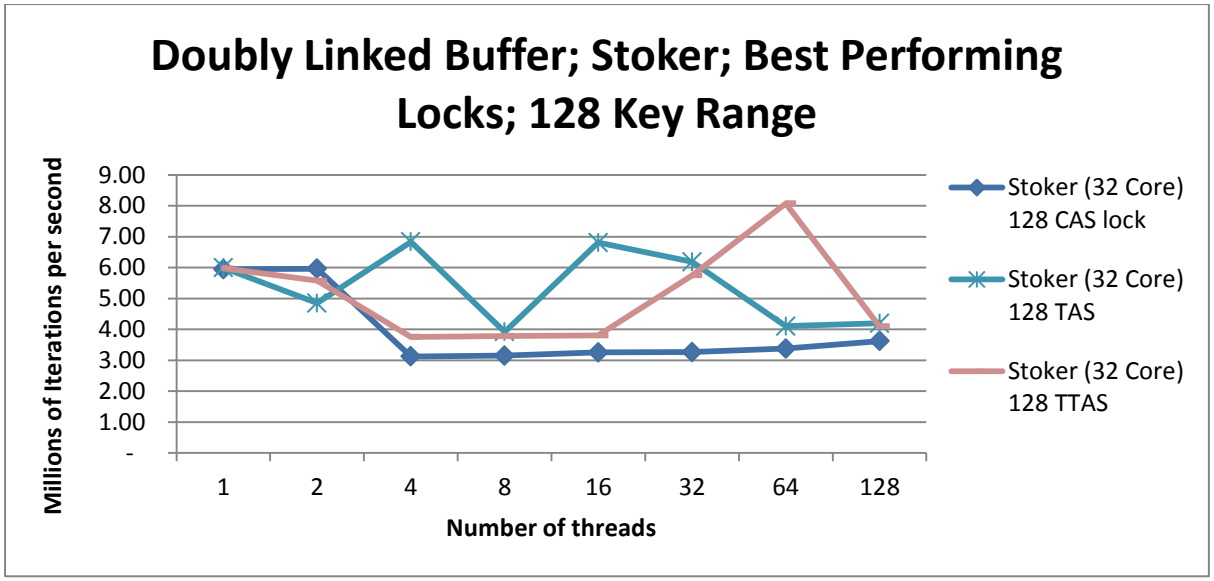


**Figure 22: Millions of iterations/second by number of threads for the doubly linked buffer on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "CAS lock", "TAS" and "TTAS" represent the *compare-and-swap*, the *test-and-set* and the *test-and-test-and-set* lock implementations respectively.**

| | Cycles | Instructions | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|---|
| *Compare-and-swap* | 83.65 B | 86.51 B | 111.46 M | 102.28 M | 48.08 B | 39.63 B |
| *Test-and-set* | 2720.11 B | 35.47 B | 354.39 M | 285.7 M | 2702.87 B | 2634.23 B |
| *Test-and-test-and-set* | 85.31 B | 79.99 B | 86.04 M | 77.07 M | 52.36 B | 41.37 B |

**Table 9: Hardware performance data gathered from the three doubly linked buffer locked implementations tested in figure 22. The compare-and-swap, the test-and-set and the test-and-test-and-set lock implementations.**

From *table 9* we see that the *test-and-set* lock has far more CPU cycles and fewer instructions to execute 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.

The result of this is that the doubly linked buffer could be implemented with any of the three locked implementations in *figure 22*. This is due to the fact that they all perform best at different points. The *compare-and-swap* lock implementation performs the best relative to the other two implementations at a thread count of two, whereas the *test-and-set* lock implementation performs the best at the thread counts of four and sixteen. For higher thread counts the *test-and-test-and-set* lock appears to perform the best, peaking at a thread count of sixty four.

#### 4.3.2.3 Locked vs Lockless Comparison

I now want to compare the best locked implementations against the lockless implementation of the doubly linked buffer to see how they perform relative to each other. The *lockless* implementation does as well as the locked implementations for low thread counts. However, past four threads the *lockless* implementation's performance drops sharply below the locked implementations and does not recover for the remainder of the test as shown in *figure 23*.

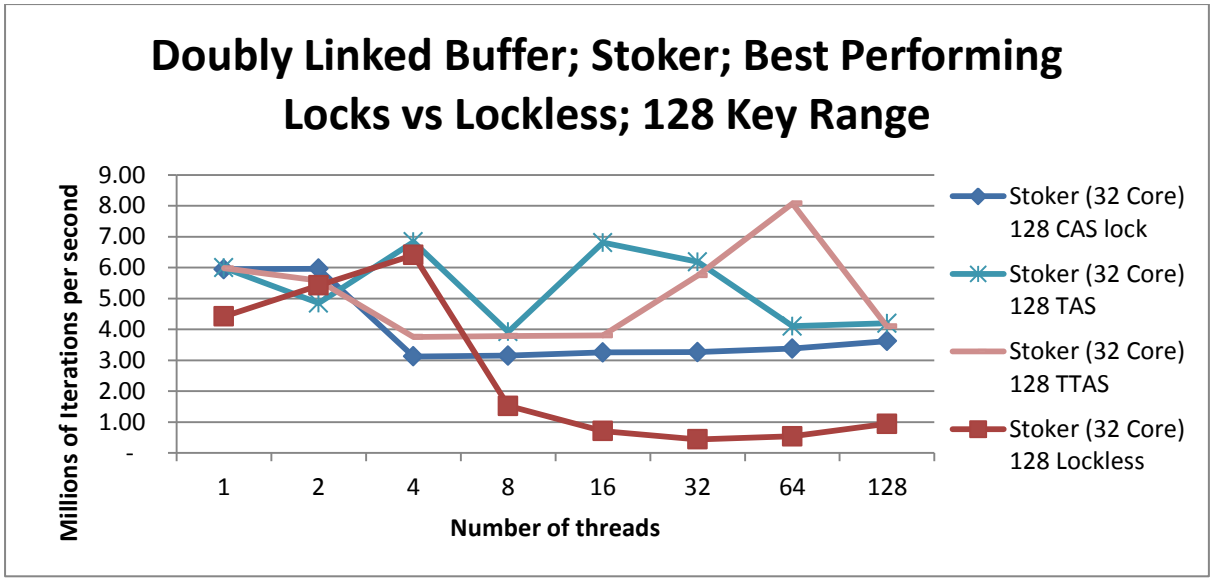


**Figure 23: Millions of iterations/second by number of threads for the doubly linked buffer on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "CAS lock", "TAS", "TTAS" and "Lockless" represent the *compare-and-swap* lock, the *test-and-set* lock, the *test-and-test-and-set* lock and the *lockless* implementations respectively.**

| | Cycles | Instructions | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|---|
| *Compare-and-swap* | 83.65 B | 86.51 B | 111.46 M | 102.28 M | 48.08 B | 39.63 B |
| *Test-and-set* | 2720.11 B | 35.47 B | 354.39 M | 285.7 M | 2702.87 B | 2634.23 B |
| *Test-and-test-and-set* | 85.31 B | 79.99 B | 86.04 M | 77.07 M | 52.36 B | 41.37 B |
| *Lockless* | 1901.83 B | 134.36 B | 670.12 M | 417.19 M | 1825.85 B | 1539.81 B |

**Table 10: Hardware performance data gathered from the three doubly linked buffer locked implementations and the *lockless* implementation tested in figure 23. The *compare-and-swap* lock, *the test-and-set* lock, the *test-and-test-and-set* lock and the *lockless* implementations.**

Now that the l*ockless* implementation data is added onto the other implementations in *table* 10 we can see why its performance is not what may have been expected. It has a high rate of stalled CPU cycles, below that of the *test-and-set* lock implementation but much higher than the other two implementations indicating that it relies heavily on memory as cycles are stalled while values are retrieved from memory.

The data indicates that the *lockless* implementation does very poorly at higher thread counts due to the high amounts of contention at the head and tail of the doubly linked list. Where the other locked implementations appear to handle the contention as the thread count increases, the *lockless* implementation falters. If the doubly linked buffer were to be implemented, the *lockless* implementation should not be used as the locked implementations can perform better across all the thread counts tested.

#### 4.3.2.4 Test-and-test-and-set Lock Comparison

As with previous data structures I now draw my attention to comparing the lock variations against each other to determine which implementation performs the best. 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*. *The test-and-test-and-set* lock proves itself to be the best performing variation from *figure 24*, outperforming the other two implementations at every thread count and peaking in performance at a thread count of sixty four.

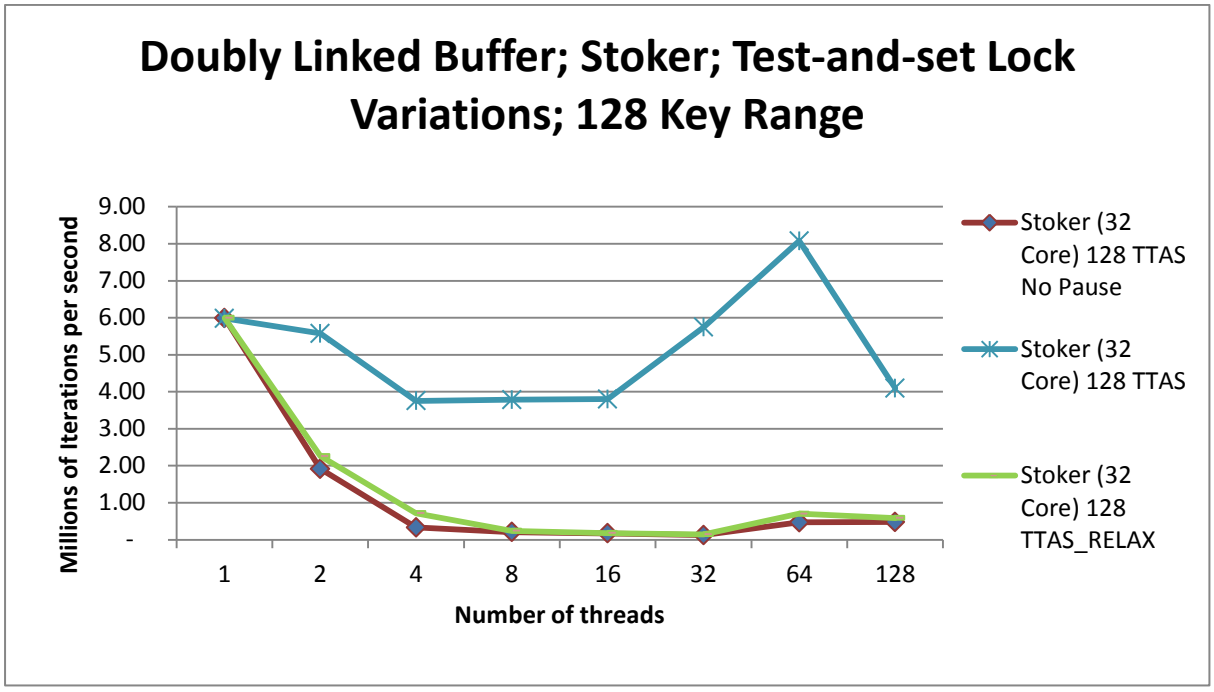


**Figure 24: Millions of iterations/second by number of threads for the doubly linked buffer on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "TTAS No Pause", "TTAS" and "TTAS_RELAX" represent the *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Test-and-test-and-set* | 85.31 B | 86.04 M | 77.07 M | 52.36 B | 41.37 B |
| *Test-and-test-and-set-no-pause* | 2224.45 B | 678.33 M | 301.95 M | 1914.52 B | 926.14 B |
| *Test-and-test-and-set-relax* | 2135.44 B | 420.03 M | 248.51 M | 2087.39 B | 1936.75 B |

**Table 11: Hardware performance data gathered from the three doubly linked buffer locked implementations tested in figure 24. The *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations.**

From *table 11* 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.

This test shows that the other two variations of the *test-and-test-and-set* lock implementation should be ignored when considering how to implement the doubly linked buffer. They are outperformed when compared to the *test-and-test-and-set* lock implementation which performs extremely well at the higher thread counts of thirty two and sixty four.

#### 4.3.2.5 Test-and-set Lock Comparison

Moving on to the *test-and-set* lock and its variations, I now want to test the three related implementations to discern which one is the best performing. From *figure 25*, the *test-and-set* lock implementation stands out, outperforming the other two varieties across all thread counts.

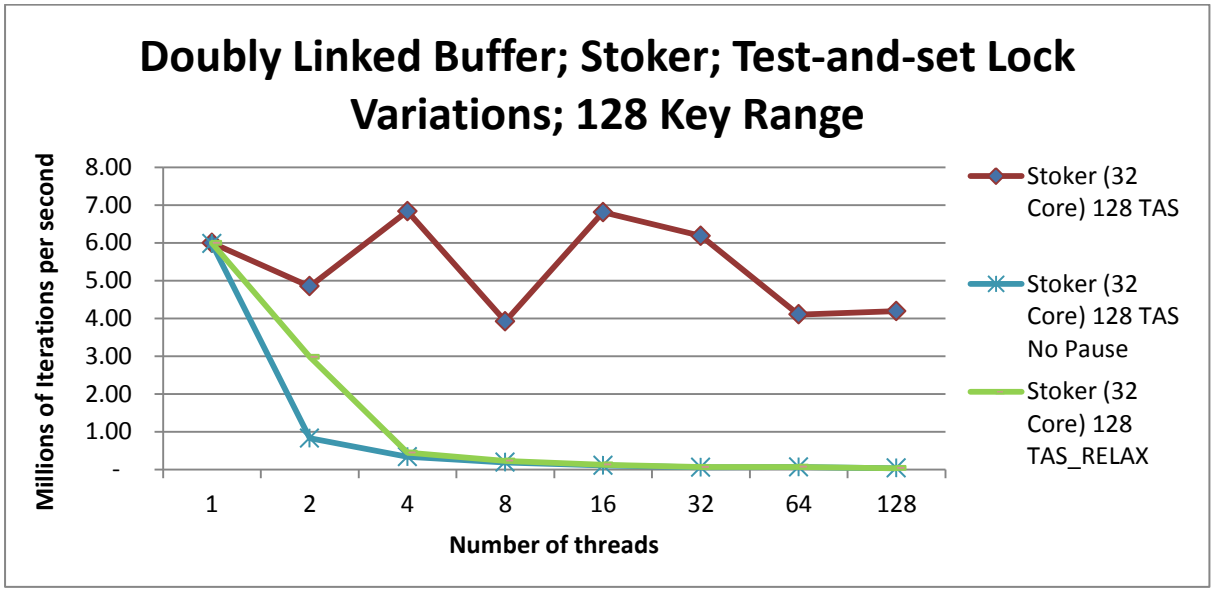


**Figure 25: Millions of iterations/second by number of threads for the doubly linked buffer on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "TAS", "TAS No Pause" and "TAS_RELAX" represent the *test-and-set*, the *test-and-set-no-pause* and the *test-and-set-relax* lock implementations respectively.**

| | Cycles | Branches | Branch Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Test-and-set* | 85.31 B | 16.17 B | 4.22 M | 52.36 B | 41.37 B |
| *Test-and-set-no-pause* | 2224.45 B | 7.82 B | 16.49 M | 1914.52 B | 926.14 B |
| *Test-and-set-relax* | 2135.44 B | 7.91 B | 11.28 M | 2087.39 B | 1936.75 B |

**Table 12: Hardware performance data gathered from the three doubly linked buffer locked implementations tested in figure 25. The *test-and-set*, the *test-and-set-no-pause* and the *test-and-set-relax* lock implementations.**

From *table 12* it can be seen that 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 it to make better of its CPU cycles and of the CPU pipeline, giving the *test-and-set* lock such a good margin of performance over the other two variations.

This test shows that the *test-and-set* lock implementation is best performing out of the three and the other two should not be considered whatsoever when it comes to implementing the doubly linked list. There is no point where the *test-and-set* lock implementation is outperformed, though it performs better at lower thread counts, with its performance dipping at the sixty four thread count mark.

#### 4.3.2.6 Compare-and-swap Lock Comparison

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. I now investigate which variation performs the best out of the three. *Figure 26* shows the *compare-and-swap* lock implementation to be the best, outperforming the other two implementations at all thread counts.

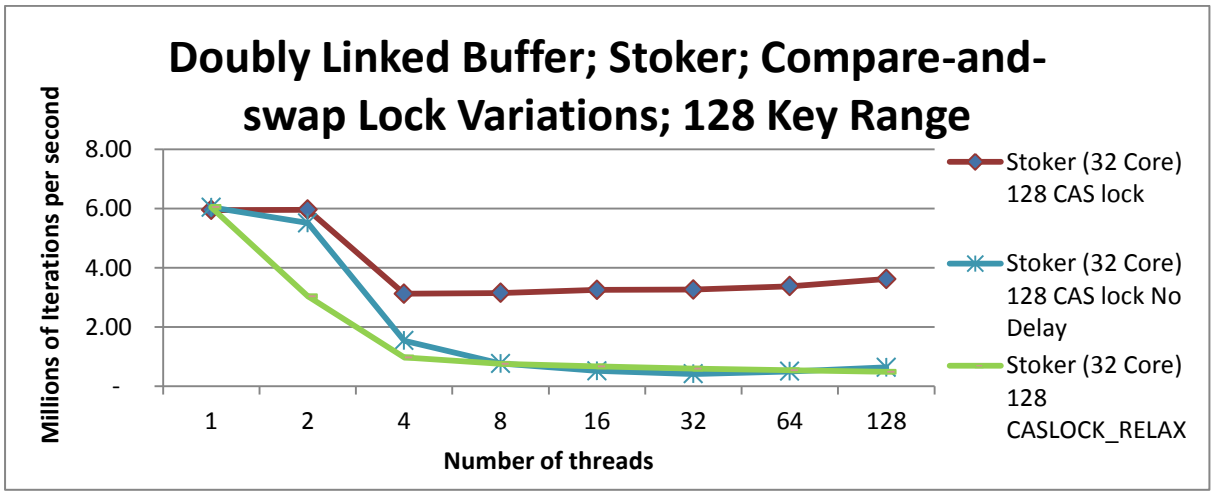


**Figure 26: Millions of iterations/second by number of threads for the doubly linked buffer on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "CAS lock", "CAS lock No Delay" and "CASLOCK_RELAX" represent the *compare-and-swap*, the *compare-and-swap-no-delay* and the *compare-and-swap* lock implementations respectively.**

| | Cycles | Branches | Branch Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Compare-and-swap* | 83.65 B | 17.43 B | 5.73 M | 48.08 B | 39.63 B |
| *Compare-and-swap-no-delay* | 2237.15 B | 58.3 B | 94. M | 2141.91 B | 1702.9 B |
| *Compare-and-swap-relax* | 214.14 B | 76.81 B | 88. M | 2224.24 B | 1794.57 B |

**Table 13 : Hardware performance data gathered from the three doubly linked buffer locked implementations tested in figure 26. The *compare-and-swap*, the *compare-and-swap-no-delay* and the *compare-and-swap* lock implementations.**

From *table 13* 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.

With this in mind, the *compare-and-swap* lock implementation should always be used over the two alternate implementations as it achieves the most iterations per second. However, it should be noted that it dips in performance at a thread count of four, though it then remains level for the remainder of the thread counts, despite increasing contention among the threads.

#### 4.3.2.7 Ticket Lock Comparison

I evaluate the *ticket* lock as I am curious as to the difference in performance between the *ticket* and *ticket-relax* lock implementations. From *figure 27* we see that the *ticket-relax* lock is actually the better performing of the two.

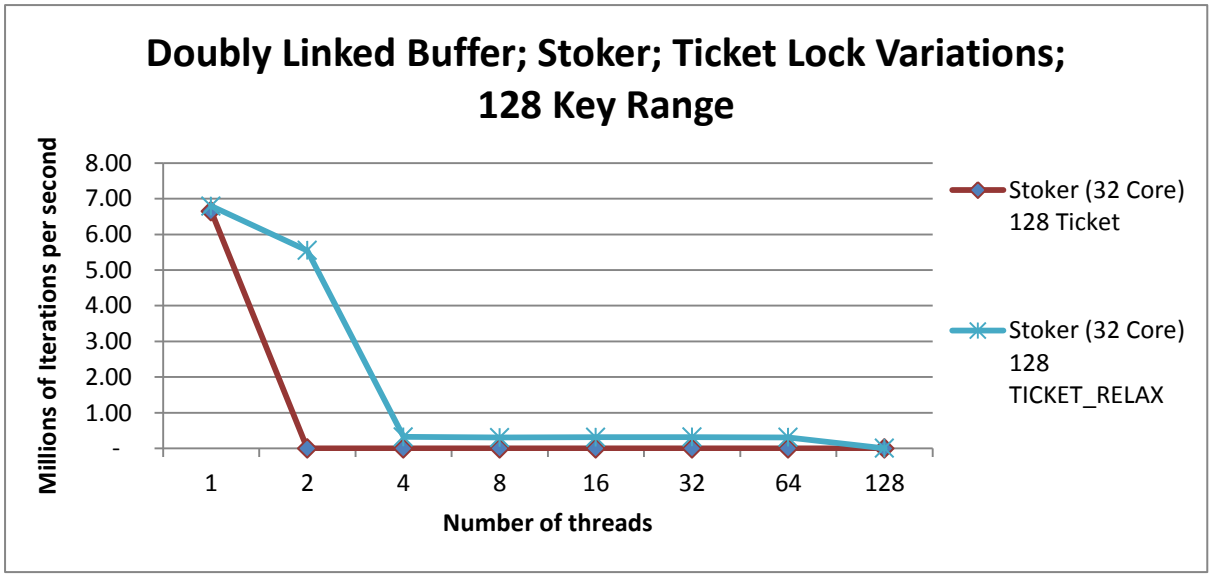


**Figure 27 : Millions of iterations/second by number of threads for the doubly linked buffer on the machine Stoker. "128 Key Range" indicates that maximum number of nodes allowed in the list at any one time is 128. "Ticket" and "TICKET_RELAX" represent the *ticket* and the *ticket-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Ticket* | 15.24 B | 30.61 M | 23.68 M | 8.12 B | 6.99 B |
| *Ticket-relax* | 2868.61 B | 435.09 M | 228.33 M | 2630.76 B | 2247.46 B |

**Table 14 : Hardware performance data gathered from the three doubly linked buffer locked implementations tested in figure 27. The *ticket* and the *ticket-relax* lock implementations.**

From *table 14* the *ticket-relax* lock version of the code executes 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 uses 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 has 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 a thread count of two.

While the *ticket-relax* lock implementation is the better performing of the two, it is only really effective at a thread count of two. As a result, the *ticket-relax* implementation should probably not be used as there are other implementations that perform equally as well for a thread count of two, such as the *compare-and-swap* lock implementation in *figure26* and don't drop off sharply for higher thread counts.

#### 4.3.2.8 The Doubly Linked Buffer's Performance across Architectures

For the final piece of evaluation on the doubly linked buffer I will now investigate how robust some of the doubly linked buffer implementations are across the architectures of Stoker, Cube and Local Machine. Firstly, *figure 28* shows us the *pthread mutex* lock which has a similar shape for both Stoker and Cube, yet the performance on Local is quite different, dropping sharply at a thread count of two and then remaining at a similar performance for the remainder of the thread counts compared to the slope-like appearance of Stoker and Cube.

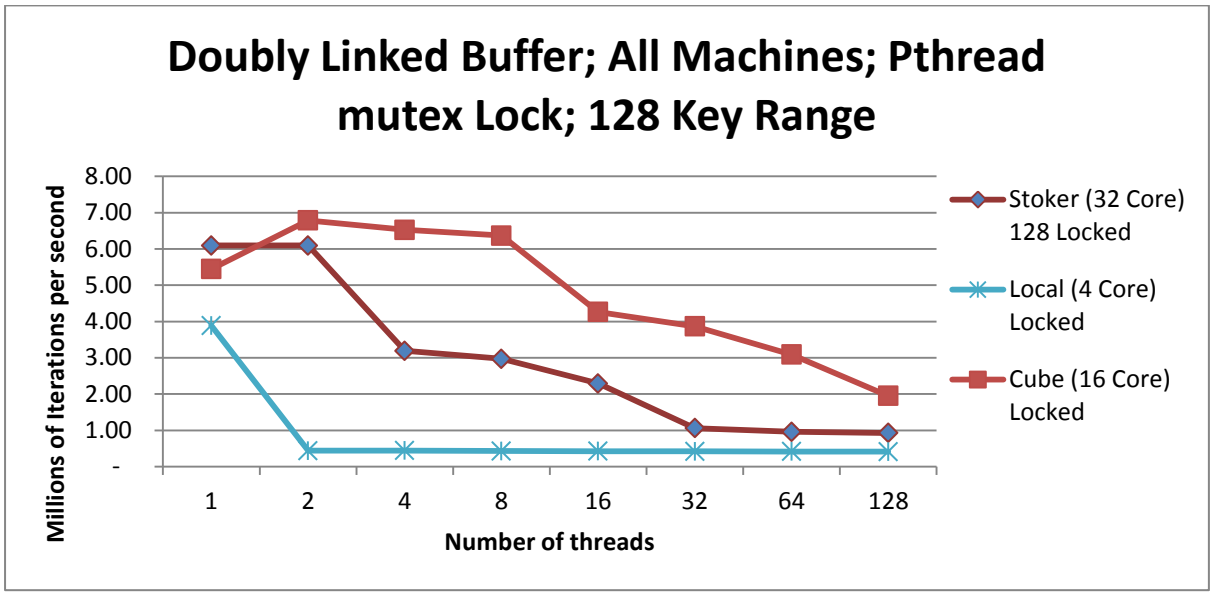


**Figure 28: Millions of iterations/second by number of threads for the doubly linked buffer on the three architectures of Stoker, Cube and Local Machine. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Stoker Locked", "Local Locked" and "Cube Locked" represent the *pthread mutex* lock, with the architecture representing which machine the implementation is run on.**

Then in *figure 29* we see the *compare-and-swap* lock which appears to perform differently on each of the three architectures, with each of the architectures reporting a different number for each thread count.

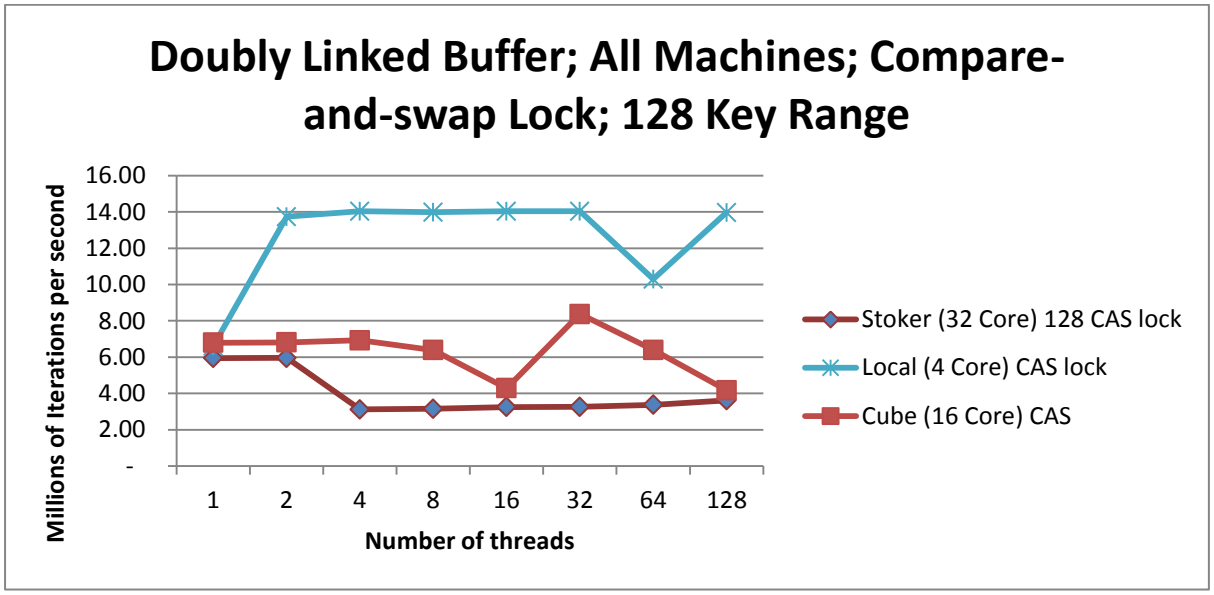


**Figure 29: Millions of iterations/second by number of threads for the doubly linked buffer on the three architectures of Stoker, Cube and Local Machine. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Stoker CAS lock", "Local CAS lock" and "Cube CAS" represent the *compare-and-swap* lock, with the architecture representing which machine the implementation is run on.**

Finally, in *figure 30* we see the lockless implementation which again performs differently on each of the architectures, with each of the architectures giving back different performances for each thread count.

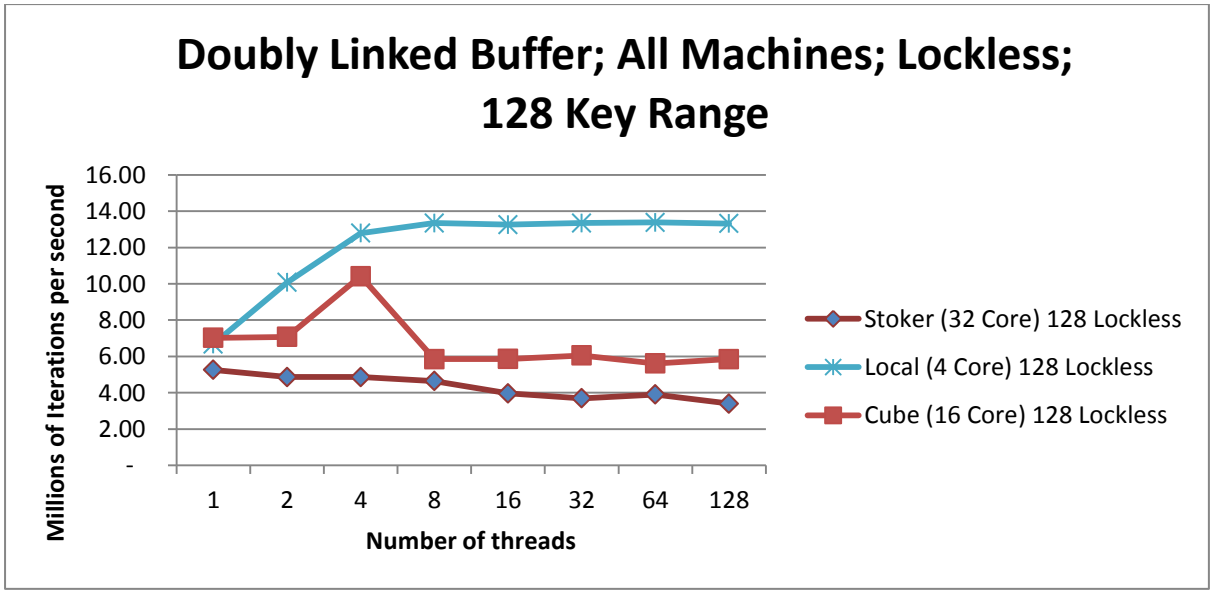


**Figure 30: Millions of iterations/second by number of threads for the doubly linked buffer on the three architectures of Stoker, Cube and Local Machine. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Stoker Lockless", "Local Lockless" and "Cube Lockless" represent the *lockless* implementation, with the architecture representing which machine the implementation is run on.**

This evaluation shows that any implementation of the doubly linked buffer should be checked on its target architecture as the performance of an implementation can vary wildly depending on the type of target architecture being used.

### 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 locked implementations first to determine which have the highest performance. I then move onto comparing the locked versions of the code with the *lockless* implementation. Afterwards, I investigate the differences between different locked 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.

#### 4.3.3.2 Locked Comparison

I now compare all the different locked implementations of the singly linked buffer to determine which implementations achieve the best performance. From *figure 31* 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.

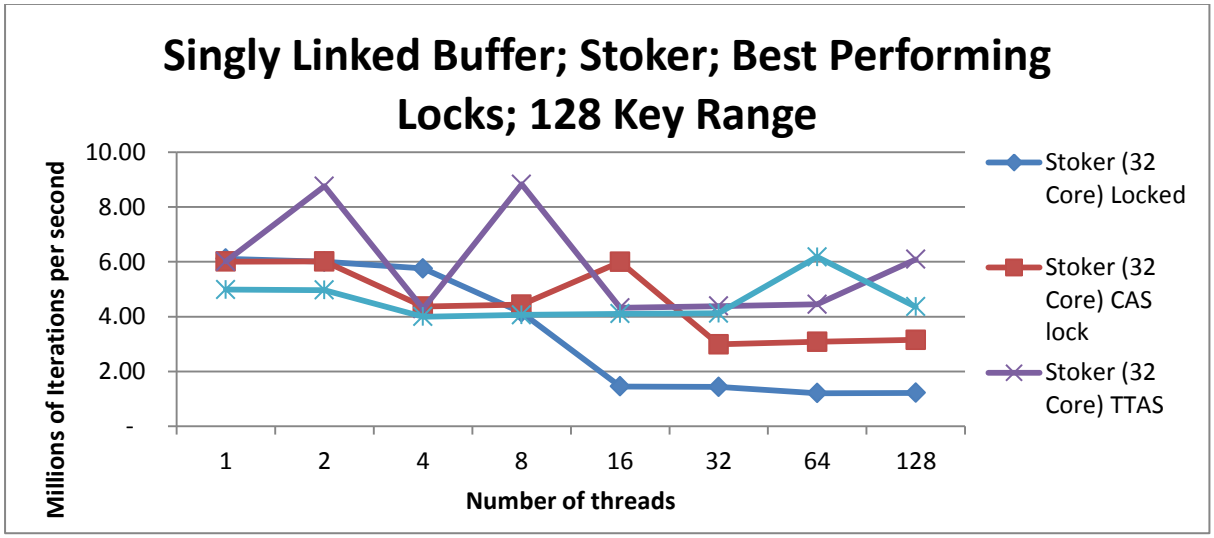


**Figure 31: Millions of iterations/second by number of threads for the singly linked buffer on the machine Stoker. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Locked", "CAS lock", "TTAS" and "TAS" represent the *pthread mutex*, *compare-and-swap*, *test-and-test-and-set* and *test-and-set* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Frontend Cycles |
|---|---|---|---|---|---|
| *Pthread Mutex* | 2202.59 B | 632.18 M | 292.22 M | 2089.12 B | 1419.93 B |
| *Compare-and-swap* | 85.86 B | 91.46 M | 83.33 M | 51.31 B | 41.53 B |
| *Test-and-set* | 87.83 B | 98.98 M | 89.44 M | 52.21 B | 41.94 B |
| *Test-and-test-and-set* | 91.38 B | 80.17 M | 71.77 M | 56.57 B | 44.79 B |

**Table 15: Hardware performance data gathered from the three singly linked buffer locked implementations tested in figure 31. The *pthread mutex*, *compare-and-swap*, *test-and-test-and*-set and *test-and-set* lock implementations.**

*Table 15* 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.

From this data we can see that different implementations perform the best for different thread counts. For example, the *test-and-test-and-set* lock implementation performs the best out of the other three locks at thread counts two, eight and one hundred and twenty eight, whereas the *compare-and-swap* lock implementation achieves the best performance at a thread count of sixteen. If one was to implement the singly linked buffer with one of these implementations, they would need to discover how many threads they would be using as the choice of locked implementation changes with the thread count.

#### 4.3.3.3 Locked vs Lockless Comparison

I now want to examine the differences in performance between some of the locked implementations of the singly linked buffer and the lockless implementation. To do this I take two of the locked implementations and compare them against the *lockless* implementation. *Figure 32* shows this comparison, however, the *lockless* implementation performs quite poorly, outperformed by the *pthread mutex* lock for the first half of the thread counts and beaten at every thread count bar one by the *test-and-test-and-set* lock implementation.

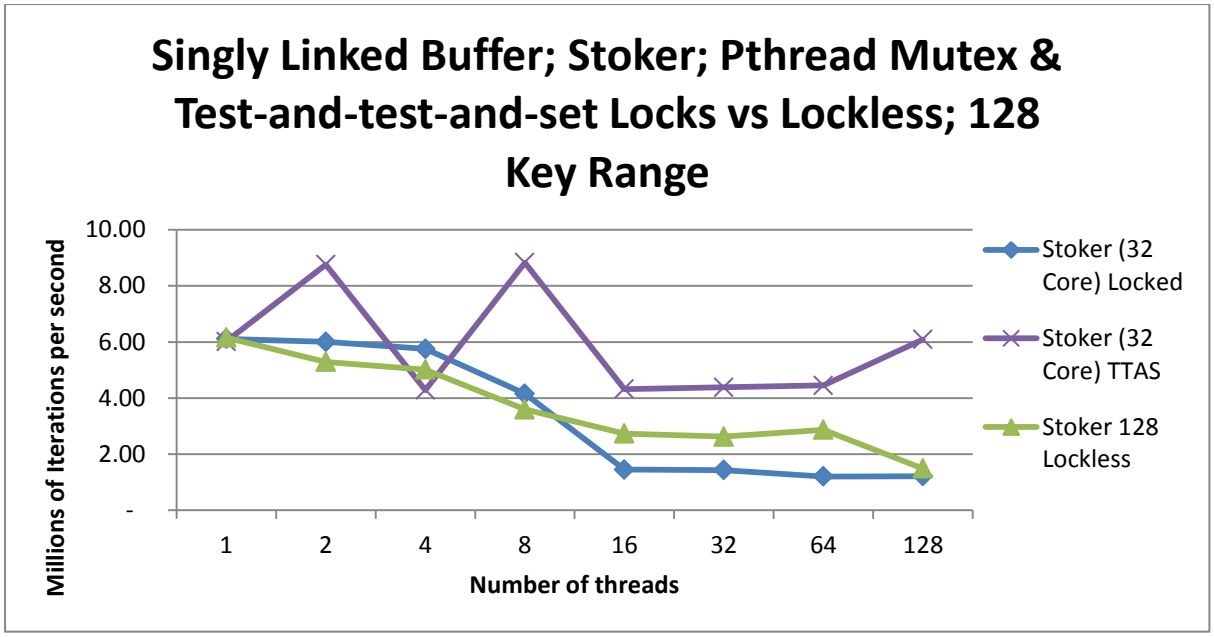


**Figure 32: Millions of iterations/second by number of threads for the singly linked buffer on the machine Stoker. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Locked", "TTAS" and "Lockless" represent the *pthread mutex* lock, the *test-and-test-and-set* lock and the *lockless* implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Frontend Cycles |
|---|---|---|---|---|---|
| *Pthread Mutex* | 2202.59 B | 632.18 M | 292.22 M | 2089.12 B | 1419.93 B |
| *Test-and-test-and-set* | 91.38 B | 80.17 M | 71.77 M | 56.57 B | 44.79 B |
| *Lockless* | 942.32 B | 699.22 M | 468.13 M | 887.68 B | 653.2 B |

**Table 16: Hardware performance data gathered from the three singly linked buffer locked implementations tested in figure 32. The *pthread mutex* lock, the *test-and-test-and-set* lock and the *lockless* implementations.**

*Table 16* displays the hardware performance data for the three implementations shown in *figure 32*. The *lockless* implementation performs about as well as the *pthread mutex* lock implementation, with the *lockless* implementation losing in the early thread counts but pulling ahead of the *pthread mutex* lock implementation 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 when compared to the *test-and-test-and-set* lock implementation.

This data suggests that for this concurrent data structure, the singly linked buffer, where thread contention is very high at large thread counts, that a locked implementation is preferable to a *lockless* one. This could be due to the fact that there are only two points of interaction with the buffer, the head and the tail, whereas with the singly linked list, each node could potentially be a point of interaction. In any case, from what has been observed in *figure 32*, the *lockless* implementation is not suited to the singly linked buffer.

#### 4.3.3.4 Test-and-test-and-set Lock Comparison

I now move to examining the differences in performance between the *test-and-test-and*-set lock implementations, of which there are three to compare. Shown in *figure 33,* 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 though still outperforming the other two variations by a sizeable margin for all thread counts.

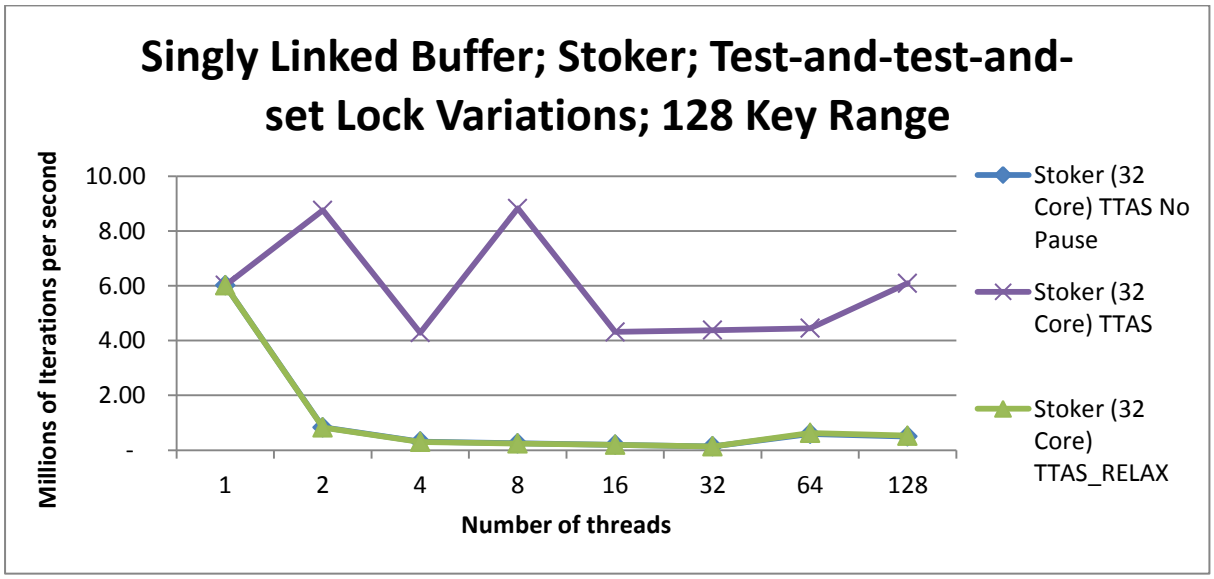


**Figure 33: Millions of iterations/second by number of threads for the singly linked buffer on the machine Stoker. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "TTAS No Pause", "TTAS" and "TTAS_RELAX" represent the *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Frontend Cycles |
|---|---|---|---|---|---|
| *Test-and-test-and-set* | 91.38 B | 80.17 M | 71.77 M | 56.57 B | 44.79 B |
| *Test-and-test-and-set-no-pause* | 2262.46 B | 633.98 M | 278.34 M | 1876.67 B | 935.14 B |
| *Test-and-test-and-set-relax* | 2264.25 B | 509.25 M | 232.24 M | 2157.47 B | 1960.32 B |

**Table 17: Hardware performance data gathered from the three singly linked buffer locked implementations tested in figure 33. The *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations.**

*Table 17* 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.

Interestingly, the *test-and-test-and-set-no-pause* and the *test-and-test-and-set-relax* implementations perform almost exactly the same according to *figure 33*. This indicates that the addition of the intrinsic *_mm_pause()* has almost no effect on the performance of this implementation of the singly linked list when compare to not having a pause instruction of any kind.

Finally, the regular *test-and-test-and-set* lock is clearly the winner in terms of performance and any implementations of the singly linked list using a *test-and-test-and-set* lock based mechanism should implement it as opposed to the two other variations which perform poorly in comparison.

#### 4.3.3.5 Test-and-set Lock Comparison

Moving on to the *test-and-set* lock, as with the *test-and-test-and-set* lock I want to investigate the relative performances of the different implementations to try and discern which one achieves the highest performance. The *test-and-set* lock implementation outperforms the other two locked variations by a significant degree as seen in *figure 34*.

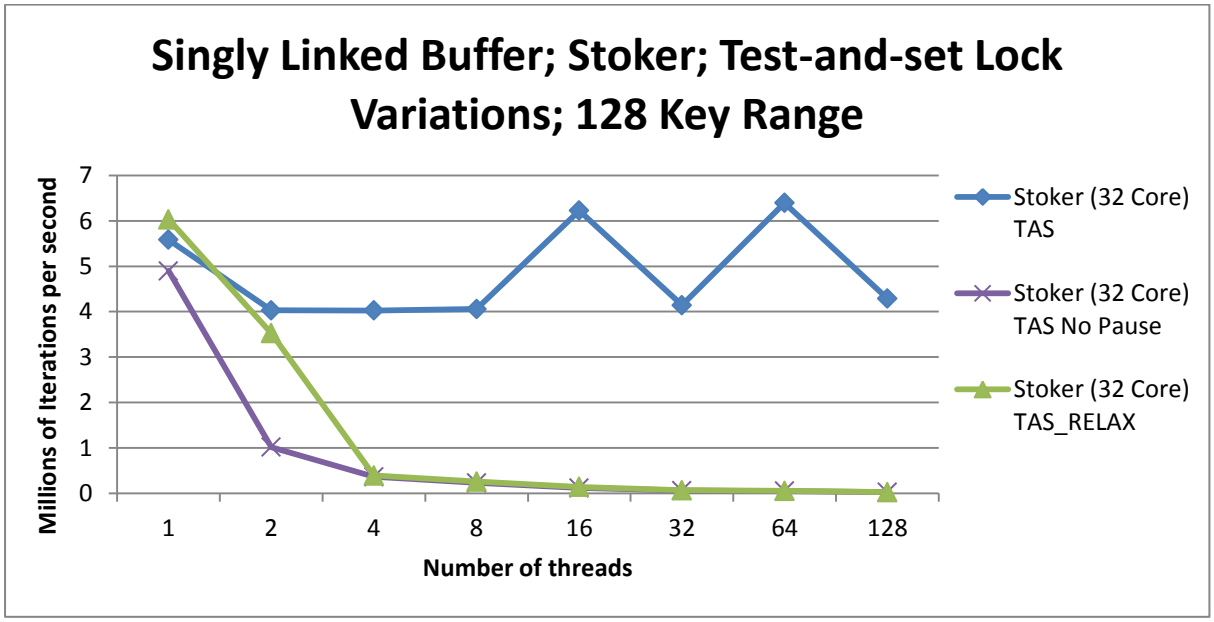


**Figure 34: Millions of iterations/second by number of threads for the singly linked buffer on the machine Stoker. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "TAS", "TAS No Pause" and "TAS_RELAX" represent the *test-and-set*, the *test-and-set* and the *test-and-set-relax* lock implementations respectively.**

| | Cycles | Branches | Branch Misses | Stalled Frontend Cycles | Stalled Frontend Cycles |
|---|---|---|---|---|---|
| *Test-and-set* | 88.53 B | 16.58 B | 11.41 M | 53.68 B | 43.92 B |
| *Test-and-set-no-pause* | 2630.48 B | 6.61 B | 20.44 M | 2616.28 B | 2550.95 B |
| *Test-and-set-relax* | 2698.35 B | 8.15 B | 20.64 M | 2674.64 B | 2525.91 B |

**Table 18: Hardware performance data gathered from the three singly linked buffer locked implementations tested in figure 34. The *test-and-set*, the *test-and-set* and the *test-and-set-relax* lock implementations.**

The first item to note is the far lower rate of branch misses attributed to the *test-and-set* lock in *table 18.* 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 lower rate of branch misses results in a far less volatile pipeline [Eles, n.d], which in turn results in fewer stalled cycles which equates to better performance for the implementation.

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.

This indicates that the addition of the *sleep()* instruction in the *test-and-set* lock implementation proves to be an extremely effective measure of maintaining performance across different thread counts and handling thread contention. This also shows that the difference between using the *_mm_pause()* instruction and not having a pause instruction in the lock is hardly noticeable past a thread count of four.

#### 4.3.3.6 Compare-and-swap Lock Comparison

The different *compare-and-swap* lock implementations are now compared to try and determine which one achieves the highest performance. 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 as shown in *figure 35.*

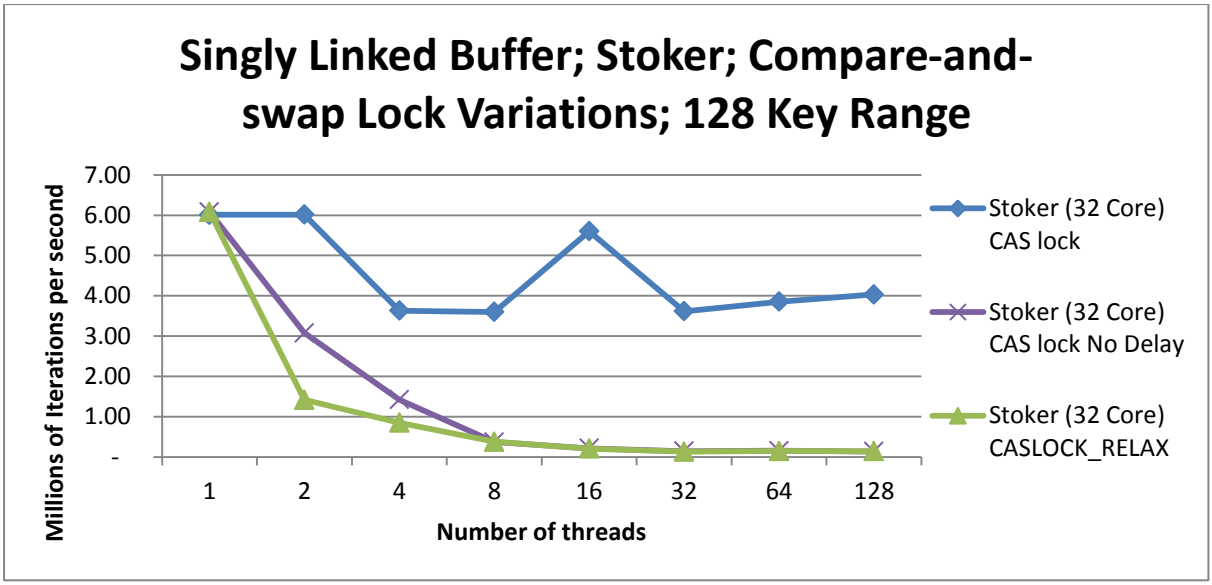


**Figure 35: Millions of iterations/second by number of threads for the singly linked buffer on the machine Stoker. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "CAS lock", "CAS lock No Delay" and "CASLOCK_RELAX" represent the *compare-and-swap*, the *compare-and-swap-no-delay* and the *compare-and-swap-relax* lock implementations respectively.**

| | Cycles | Branches | Branch Misses | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Frontend Cycles |
|---|---|---|---|---|---|---|---|
| *Compare-and-swap* | 84.67 B | 98.81 M | 88.84 M | 16.98 B | 5.2 M | 49.9 B | 40.44 B |
| *Compare-and-swap-no-delay* | 2288.58 B | 706.78 M | 456.88 M | 73.21 B | 85.37 M | 2172.82 B | 1762.62 B |
| *Compare-and-swap-relax* | 2283.02 B | 723.28 M | 470.27 M | 57.79 B | 82.22 M | 2188.95 B | 1820.84 B |

**Table 19 : Hardware performance data gathered from the three singly linked buffer locked implementations tested in figure 35. The *compare-and-swap*, the *compare-and-swap-no-delay* and the *compare-and-swap-relax* lock implementations.**

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, shown in *table 19*. However, where it pulls ahead is with its branch misses and CPU cycle utilisation. The regular *compare-and-swap* lock implementation has a very low branch miss rate when compared to the other two lock varieties and only wastes just over half of its CPU cycles. Compare this to the other two implementations who miss over three times as many branches and waste around 85% of their CPU cycles and it becomes clear why the regular *compare-and-swap* lock implementation does so well.

This test allows the regular *compare-and-swap* lock variation to be declared the best performing of three and such be used over the other two in any implementation of the singly linked buffer that wishes to use a *compare-and-swap* lock based method.

#### 4.3.3.7 Ticket Lock Comparison

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 36* shows how the two implementations compare, with the *ticket-relax* variation performing over nine times as well as the *ticket* implementation at a thread count of two.

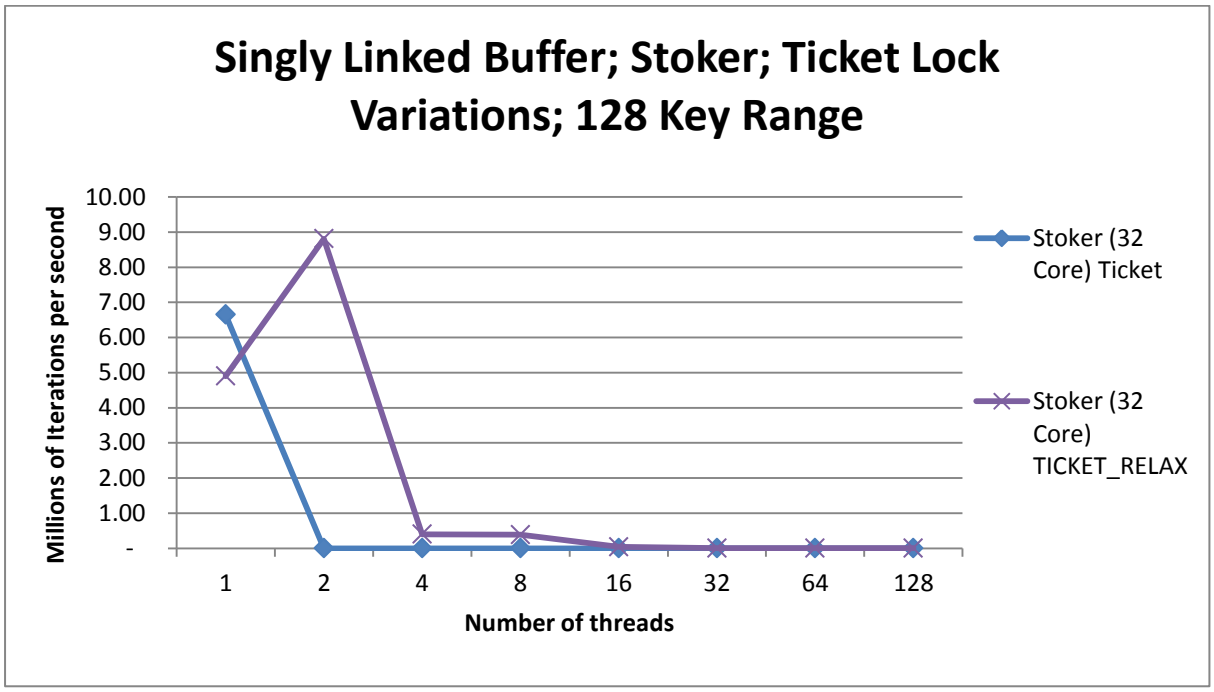


**Figure 36: Millions of iterations/second by number of threads for the singly linked buffer on the machine Stoker. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Ticket" and "Ticket-relax" represent the *ticket* and the *ticket-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Branches | Branch Misses | Stalled Frontend Cycles | Stalled Frontend Cycles |
|---|---|---|---|---|---|---|---|
| *Ticket* | 15.64 B | 30.92 M | 25.2 M | 3.66 B | 2.15 M | 8.52 B | 7.09 B |
| *Ticket-relax* | 2280.46 B | 218.9 M | 115.93 M | 157.71 B | 19.14 M | 1889.73 B | 1404.3 B |

**Table 20: Hardware performance data gathered from the two singly linked buffer locked implementations tested in figure 36. The *ticket* and the *ticket-relax* lock implementations.**

*Table 20* shows the hardware performance data gathered for the two locks with the *ticket* lock utilising 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.

Due to the limits of the data collection method, the exact reason for the *ticket-relax* implementation's huge performance margin at a thread count of two cannot be fully explained, however, for a general purpose lock neither of these should be used as they are too susceptible to massive performance losses past two threads.

#### 4.3.3.8 The Singly Linked Buffer's Performance across Architectures

For the final piece evaluation of the singly linked buffer, I investigate as to whether the different implementations of the data structure maintain their performance across the different architectures of Stoker, Cube and Local Machine. To begin, *figure 37* shows that the *pthread mutex* lock implementation is not robust across architectures, with its performance differing on each. Most noticeable is the difference between the implementations run on Local Machine and Stoker.

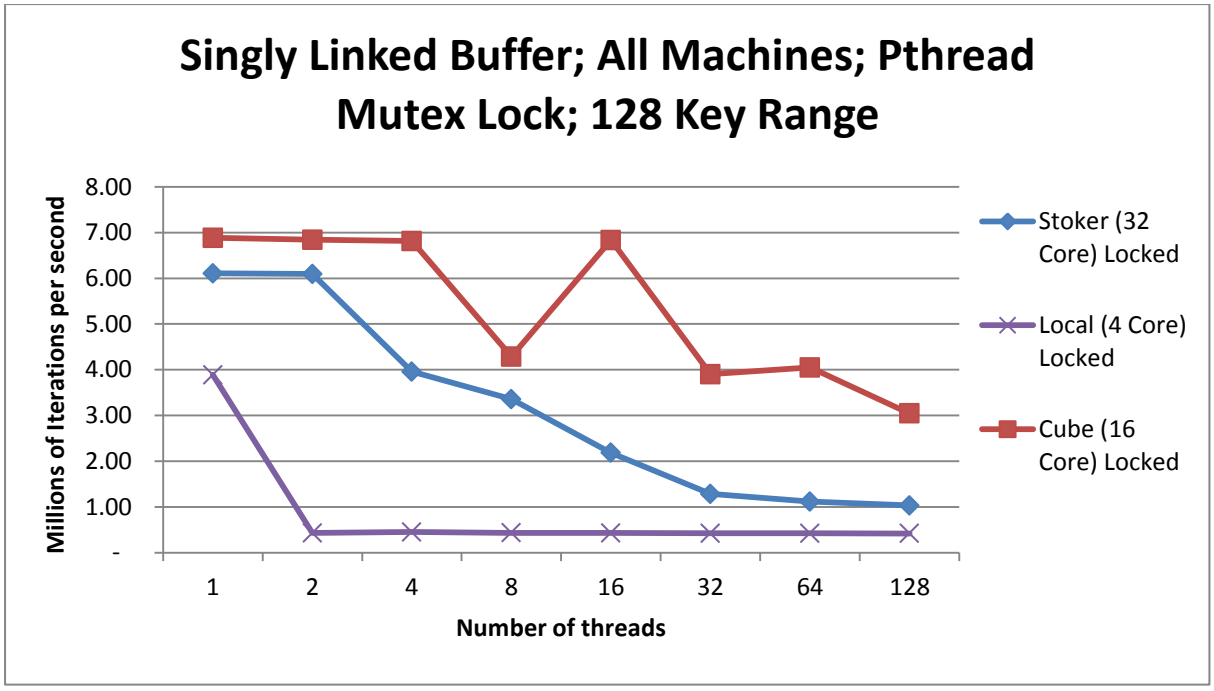


**Figure 37: Millions of iterations/second by number of threads for the singly linked buffer on the three architectures of Stoker, Cube and Local Machine. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Stoker Locked", "Local Locked" and "Cube Locked" represent the *pthread mutex* lock implementation, with the architecture representing which machine the implementation is run on.**

Next, we again see that the performance of the *test-and-test-and-set* lock implementation is different for all three of the architecture it is run on, shown in *figure 38*.

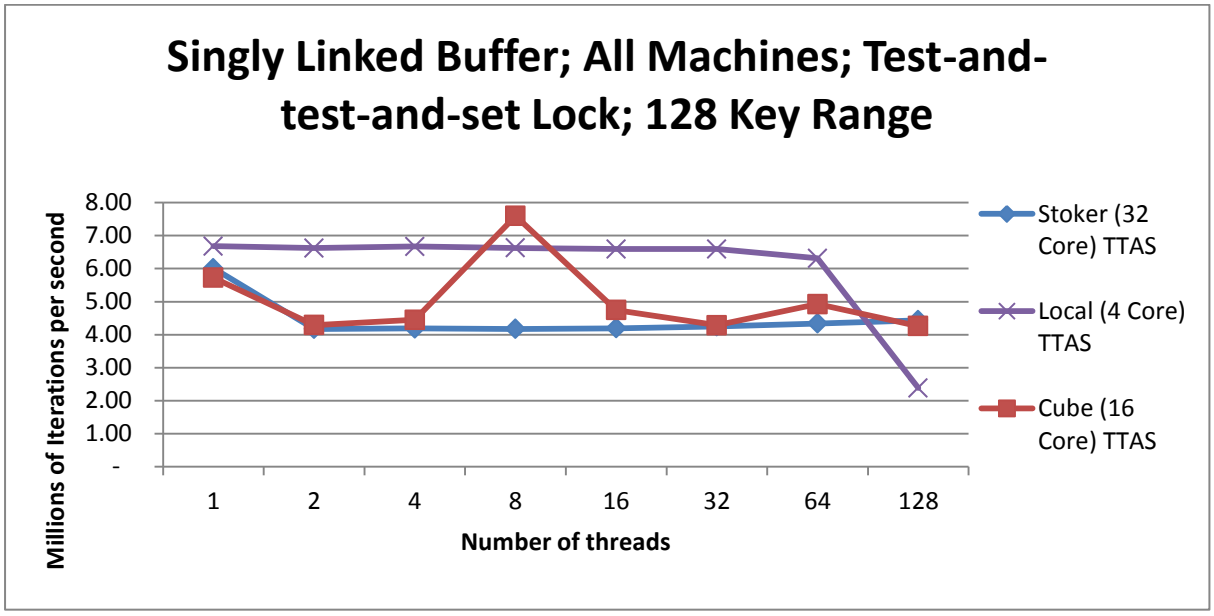


**Figure 38: Millions of iterations/second by number of threads for the singly linked buffer on the three architectures of Stoker, Cube and Local Machine. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Stoker TTAS", "Local TTAS" and "Cube TTAS" represent the *test-and-test-and-set* lock implementation, with the architecture representing which machine the implementation is run on.**

Finally, the *lockless* implementation is examined and it too, shows no robustness between the different architecture, seen in *figure 39*.

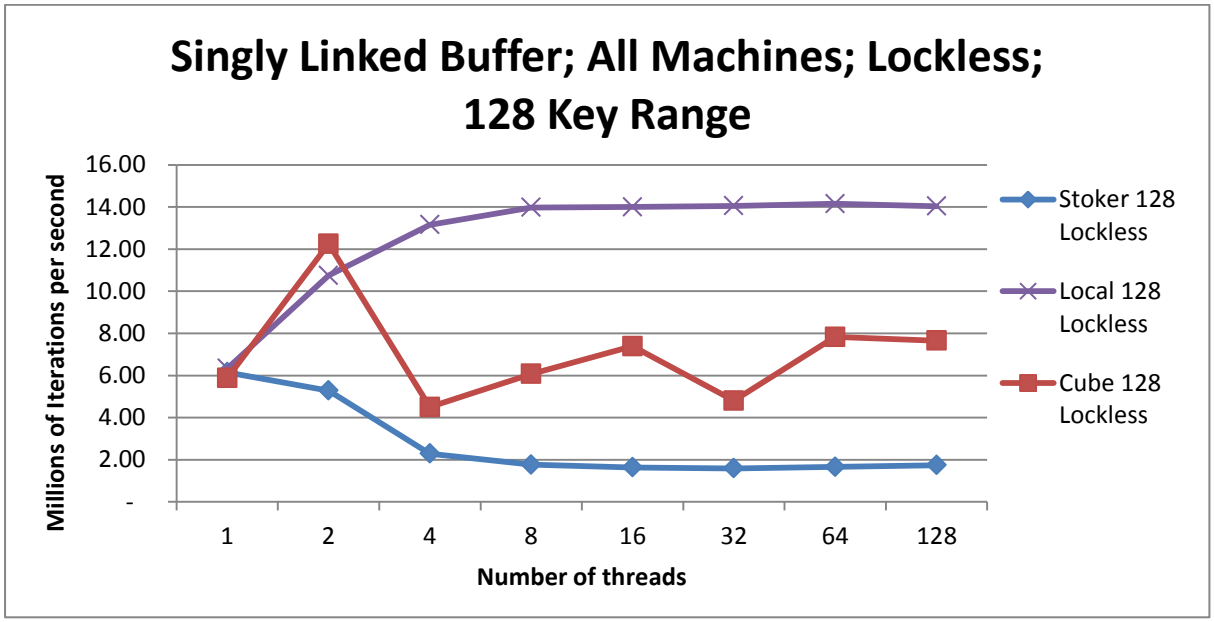


**Figure 39: Millions of iterations/second by number of threads for the singly linked buffer on the three architectures of Stoker, Cube and Local Machine. "128 Key Range" indicates the largest key that can be generated for a node in the list at any one time, though this is not a hard cap on the number of nodes. "Stoker Lockless", "Local Lockless" and "Cube Lockless" represent the *lockless* implementation, with the architecture representing which machine the implementation is run on.**

From *figure 37, 38* and *39*, it can be shown that these implementations, *pthread mutex* lock, *test-and-test-and-set* lock and *lockless* are not robust across architectures. Hence, any implementation of the singly linked list on a different architecture should be tested before use, as its performance could vary depending on both the implementation used and the target architecture.

## 4.4 Hash Table

### 4.4.1 Evaluation

For the hash table I have two locked variations and a *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 allows multiple threads to 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 investigate how the resizing functionality affects the performance of the implementations tested. Finally I test the variations on Cube and my Local Machine to see whether the variations' performance changes with the architecture they run on.

### 4.4.2 Globally Locked Comparison

To begin the evaluation, I start by attempting to find the best performing locked implementations when using the *globally* locked version of the hash table. All three of the best performing locked implementations perform quite similarly in that they all dip in performance at a thread count of two, as seen in *figure 40*. All three implementations then perform stably until a thread count of thirty two, where the *compare-and-swap* lock implementation dips in performance again, though the *test-and-set* and *test-and-test-and-set* lock implementations remain steady with regards to their performance through this period.

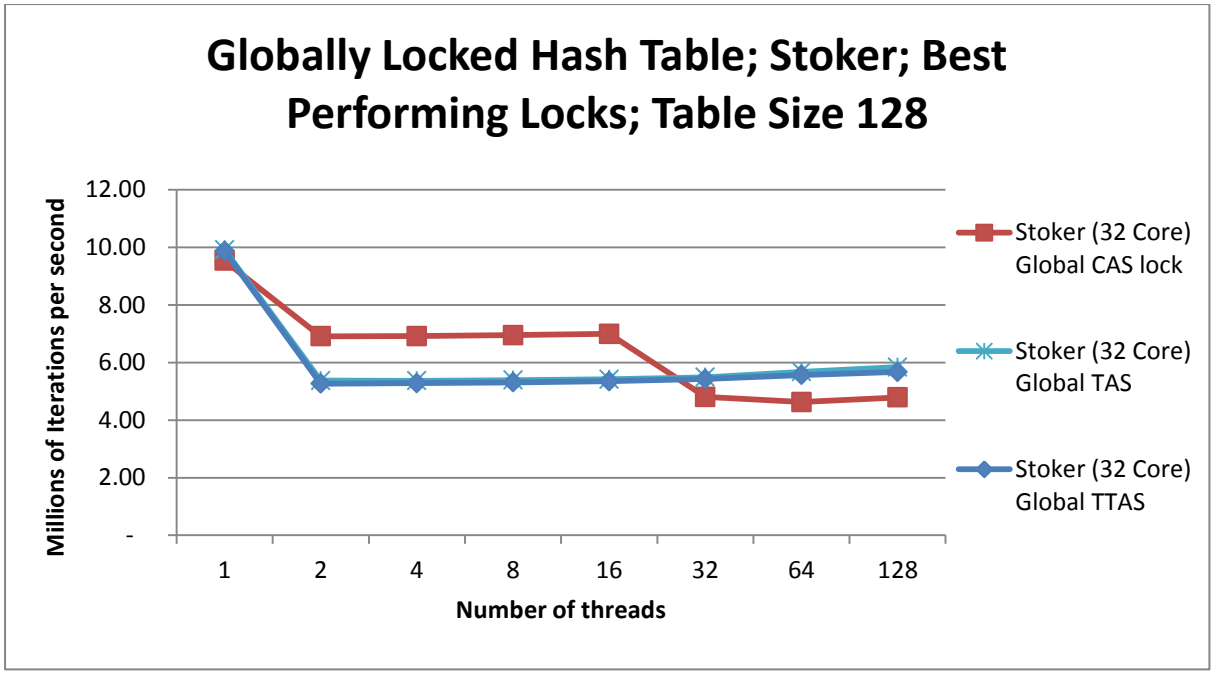


**Figure 40: Millions of iterations/second by number of threads for the *globally locked* hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "CAS lock", "TAS" and "TTAS" represent the *compare-and-swap*, the *test-and-set* and the *test-and-test-and-set* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Compare-and-swap* | 87.98 B | 15.11 M | 6.95 M | 55.51 B | 40.13 B |
| *Test-and-set* | 2688.13 B | 270.05 M | 218.31 M | 2671.17 B | 2610.23 B |
| *Test-and-test-and-set* | 76.35 B | 14.3 M | 6.47 M | 46.51 B | 34.75 B |

**Table 21: Hardware performance data gathered from the three *globally locked* hash table implementations tested in figure 40. The *compare-and-swap*, the *test-and-set* and the *test-and-test-and-set* lock implementations.**

*Table 21* shows that the *compare-and-swap* lock and the *test-and-test-and-set* lock both have roughly the same number of cycles; a similar ratio of cache misses and have comparable amounts of stalled cycles.

The *test-and-set* lock has over thirty times as many cycles as the other two locked implementations, yet its performance is almost identical, especially when compared to the *test-and-test-and-set* lock. This could be due to its far higher rate of both cache misses and pipeline stalls which bring its performance down to the level of the other two implementations.

These results show that any three of the locked implementations tested in *figure 40* could be used to implement the *globally locked* hash table. However, it is important to note that the *compare-and-swap* implementation does creep ahead in performance with the smaller thread counts while the *test-and-set* and *test-and-test-and-set* implementations perform better at higher thread counts, so this should be taken into account.

### 4.4.3 Lock per Bucket Comparison

I now need to measure the performance of the locked implementations on the *lock per* bucket hash table to both see how the different versions perform and so I can then compare the *lock per* bucket and *globally* locked hash tables. The three implementations that achieved the best performance all act similarly up to a thread count of thirty two, at which point they start to diverge as seen in *figure 41.* The addition of more threads causes the *compare-and-swap* lock version of the code to start falling behind in performance compared to the two other implementations, whose performance remains level as the thread count increases.

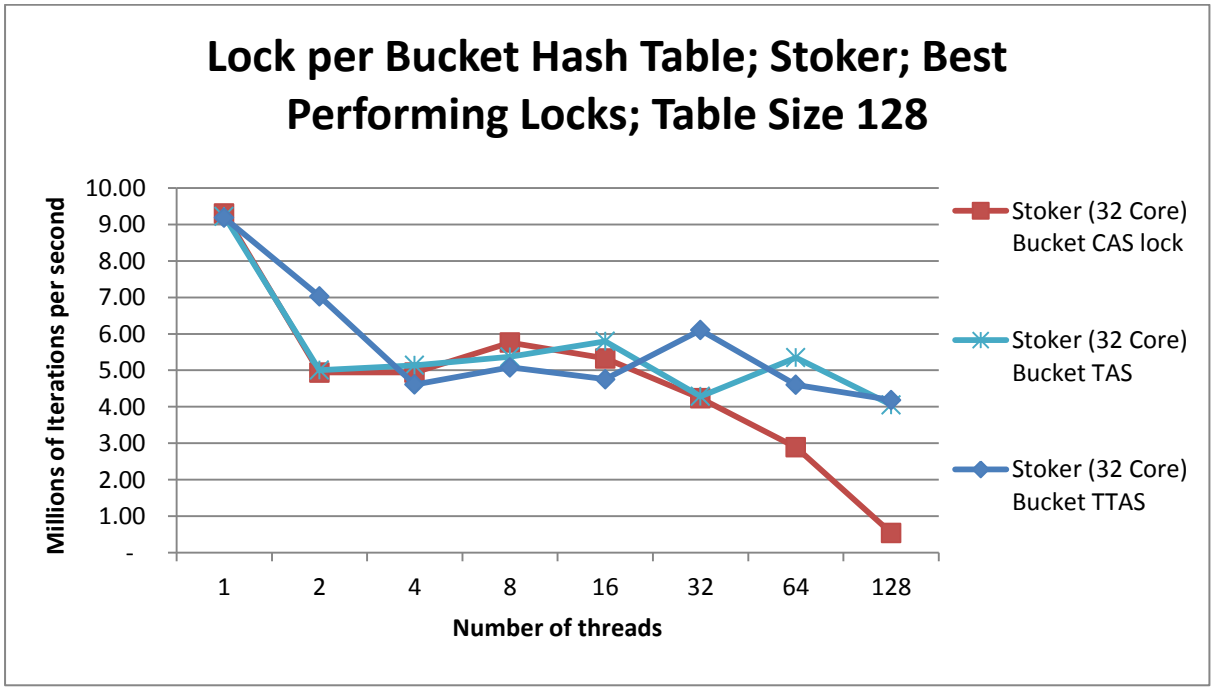


**Figure 41: Millions of iterations/second by number of threads for the *lock per bucket* hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "CAS lock", "TAS" and "TTAS" represent the *compare-and-swap, the test-and-set* and the *test-and-test-and-set* lock implementations respectively.**

| | Cycles | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|
| *Compare-and-swap* | 2328.5 B | 2209.03 B | 1323.89 B |
| *Test-and-set* | 133.08 B | 99.53 B | 65.63 B |
| *Test-and-test-and-set* | 2389.94 B | 1964.04 B | 1093.77 B |

**Table 22: Hardware performance data gathered from the three *lock per bucket* hash table implementations tested in figure 41. The *compare-and-swap*, the *test-and-set* and the *test-and-test-and-set* lock implementations.**

*Table 20* shows that the *test-and-test-and* and *compare-and-swap* implementations are comparable, in CPU cycles, though they diverge slightly when it comes to stalled cycles. The *compare-and-swap* version has a higher proportion of stalls, indicating that it is less resilient to thread contention than the other two implementations. This is shown in *figure 41* as from the thread count of eight; the *compare-and-swap* implementation's performance starts to slump and continues to do so for the remainder of the thread counts.

The *test-and-set* version performs so similarly to the *test-and-test-and-set* version because they are so comparable with regards to their ratios for cache misses and stalled cycles. For implementation purposes, the *compare-and-swap* implementation should be ignored for the *lock per bucket* hash table as its performance is overwhelmed by the other two locks for all thread counts but one.

### 4.4.4 Globally Locked vs Lock per Bucket Comparison

Having compared the locked implementations of both the *globally locked* and *lock per* bucket versions of the hash table, I am now able to compare the two directly to investigate how they differ with regards to performance and to discern which implementation is better. I choose to compare the *compare-and-swap* and *test-and-test-and-set* lock implementations of both versions of the hash table as for both cases, the *test-and-set* implementation performed very similarly to the *test-and-test-and-set* lock implementation, so I felt that it could be removed.

From *figure 42* it appears that the *compare-and-swap* implementation on the *globally locked* hash table is more resistant to thread contention than the *lock per bucket* variety, as it maintains it performance for the final three thread counts whereas the *lock per bucket* variety slumps progressively at each thread count. The *test-and-test-and-set* lock *globally* locked implementation also appears to have the edge in the final thread counts, outperforming its *lock per bucket* counterpart by a slim margin.

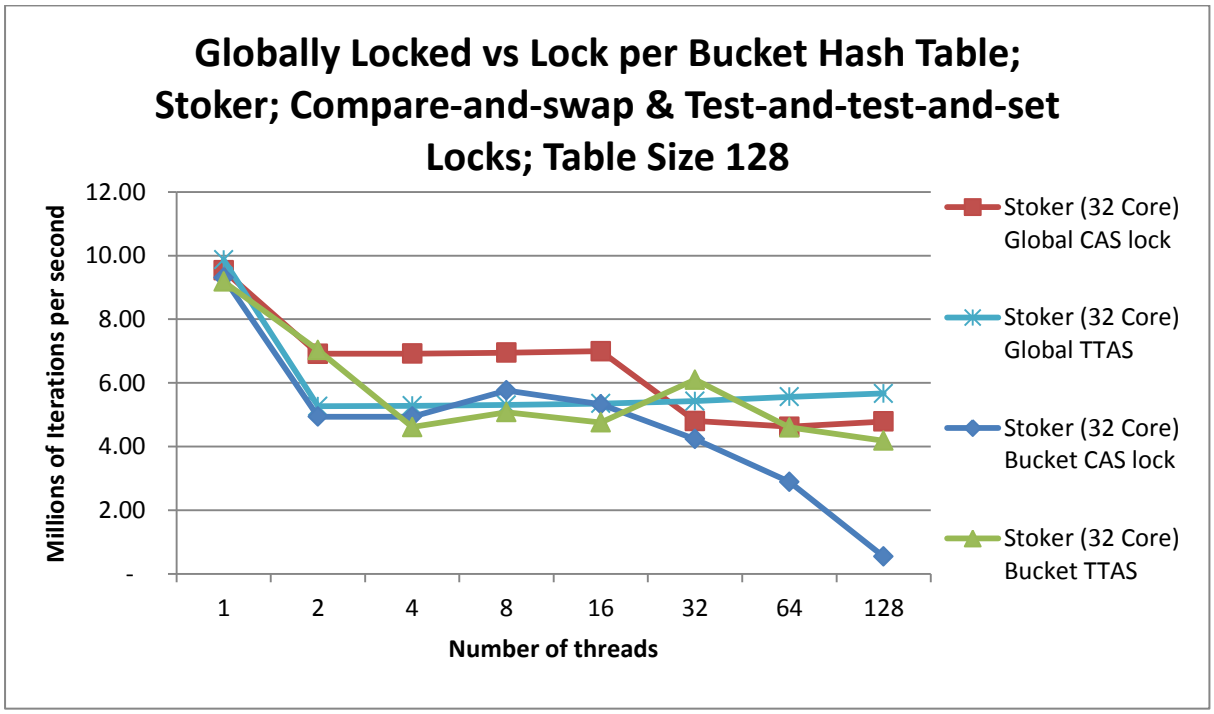


**Figure 42: Millions of iterations/second by number of threads for both the *globally locked* and *lock per bucket* varieties of the hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Global CAS lock", "Global TTAS", "Bucket CAS lock" and "Bucket TTAS" represent the *compare-and-swap* and the *test-and-test-and-set* lock implementations respectively with the name describing which variety of the hash table the implementation was run on.**

| | Cycles | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|
| *Global Compare-and-swap* | 87.98 B | 55.51 B | 40.13 B |
| *Lock per Bucket Compare-and-swap* | 2328.5 B | 2209.03 B | 1323.89 B |
| *Global Test-and-test-and-set* | 76.35 B | 46.51 B | 34.75 B |
| *Lock per Bucket Test-and-test-and-set* | 2389.94 B | 1964.04 B | 1093.77 B |

**Table 23: Hardware performance data gathered from two *globally locked* and two *lock per bucket* hash table implementations tested in figure 42. The *compare-and-swap* and the *test-and-test-and-set* lock implementations.**

*Table 23* shows that a major difference between the two hash table varieties is the number of CPU cycles utilised by their respective implementations. Both *lock per bucket* versions have over twenty five times as many CPU cycles as their *globally locked* counterparts in addition to have a far higher rate of stalled frontend cycles.

This indicates that where the *lock per bucket* implementations may utilise more system resources, they waste much of the resources that they are given, compared with the *globally locked* versions which are much more efficient. In addition, the effects of thread contention appear to hit the *lock per bucket* versions harder, leading to poorer performance at the higher thread counts.

These results show that, interestingly, the *globally locked* variety of the hash table appears to perform better at higher thread counts and is more resource efficient, which should be taken into account when considering the two varieties.

### 4.4.5 Locked vs Lockless Comparison

Having now determined that the *globally locked* variation of the hash table is the better of the two locked versions, I now compare the lockless implementation to the two in an effort to investigate how it measures up to them. This comparison is shown in *figure 43* where the *lockless* implementation does well at a thread count of two, but then slumps until it reaches a thread count of sixteen.

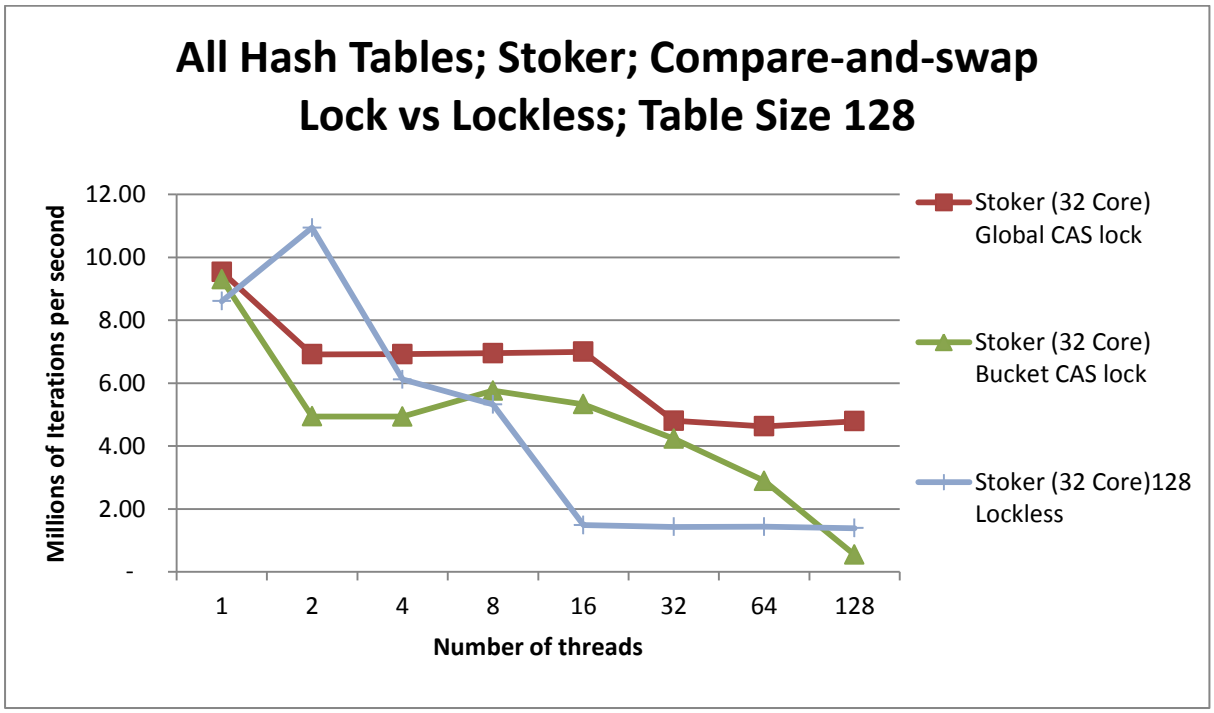


**Figure 43: Millions of iterations/second by number of threads for the *globally locked, lock per bucket* and *lockless* versions of the hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Global CAS lock", "Bucket CAS lock" and "Lockless" represent the *compare-and-swap* lock and the *lockless* implementations respectively with the name of representing which version of the hash table the implementation was run using.**

| | Cycles | Cache References | Cache Misses | Branches | Branch Misses |
|---|---|---|---|---|---|
| *Global compare-and-swap* | 87.98 B | 15.11 M | 6.95 M | 15.69 B | 1.67 M |
| *Bucket compare-and-swap* | 2328.5 B | 586.73 M | 278.27 M | 75.34 B | 53.68 M |
| *Lockless* | 2215.8 B | 620.97 M | 248.19 M | 60.86 B | 58.04 M |

**Table 24: Hardware performance data gathered from the three versions of the hash table and their respective implementations tested in figure 43. The *compare-and-swap* lock and the *lockless* implementations.**

*Table 24* shows us that 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, peaking in performance at a thread count of two but then falling until it stabilises again at a thread count of 16.

It is also noted that the *lockless* implementation has a magnitude more CPU cycles than the *globally locked compare-and-swap* implementation, much like the *lock per*

*bucket* version. The *lockless* implementation also has a comparable number of cache references to the *lock per bucket*. However, the *lockless* implementation also seems to mirror the problems of the *lock per bucket* version, performing poorly at high thread counts.

The results indicate that the *lockless* variety of the hash table, while good at a thread count of two, is consistently outperformed by the *globally locked* hash table, due to its poor resource utilisation and weakness to thread contention. This shows that, surprisingly, the *globally locked* hash table is the best performing of the three at mid to high thread counts.

### 4.4.6 The Effect of the Resize Functionality

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. This allows me to investigate how the *resize* functionality effects the different implementations.

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”. After a node is added to a bucket, the length of that bucket, the number of nodes contained in the bucket, is inspected. If the number of nodes is greater than the maximum length allowed then the *resize* function is called within the *add* function of the implementation.

From *figure 44* the impacts of the *resize* functionality can be seen, with two large drops in performance at the thread counts of two and eight where the resizing took place. After the first drop in performance, the *pthread mutex* lock with *resize* functionality regains its throughput at a thread count of four, before falling into the second drop. This would indicate that implementations suffer heavily in performance whenever a *resize* is required and so any implementations of the code should be designed to minimise the need for resizing or else attempt to make the process as efficient as possible as a large portion of performance is lost as a result of every call to the *resize* function.

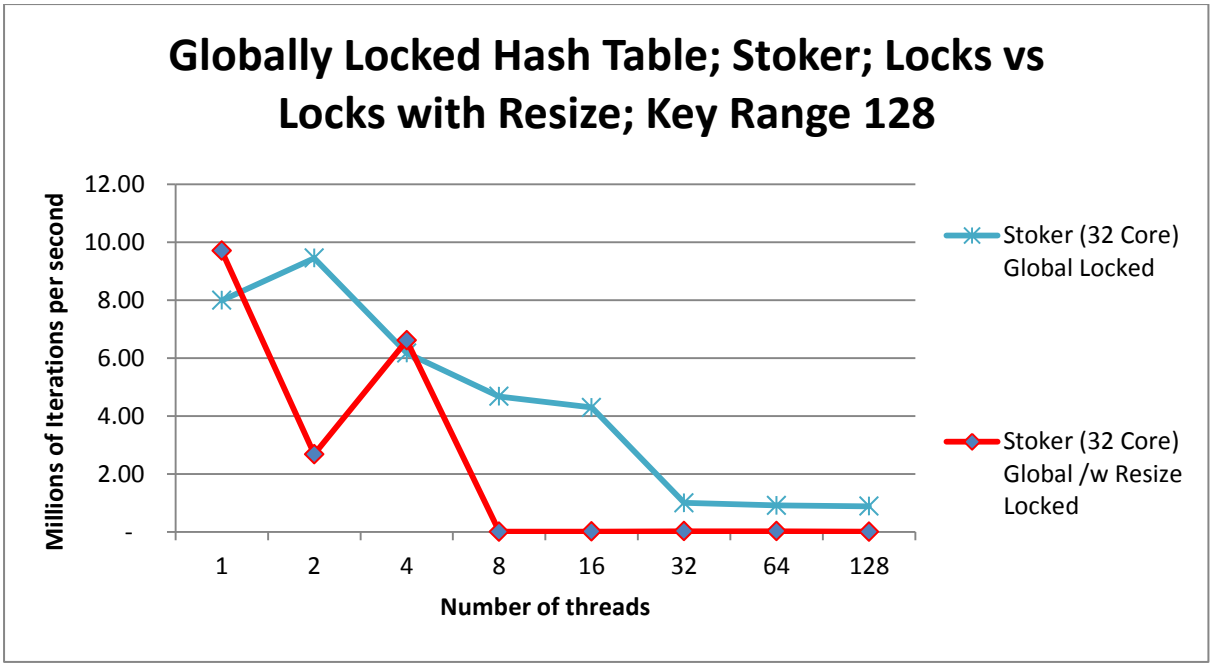


**Figure 44: Millions of iterations/second by number of threads for the globally locked hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Locked" and "/w Resize Locked" represent the *pthread* mutex lock and the *pthread* mutex lock with resize functionality implementations respectively.**

From *figure 44* it can be clearly seen that the *resize* functionality has a potent effect on the performance of the implementation that it is used. I choose not to do further evaluation of this on the *lock per bucket* or *lockless* versions of the hash table due to time constraints, though I feel that it is enough to show that the *resize* functionality does have an impact on the performance of an implementation.

### 4.4.7 The Relationship between Table Size and Hash Table Performance

To evaluate the effect that the maximum size of the Hash Table's table may have on the performance of the hash table implementations, I run two implementations, the *pthread mutex* and *compare-and-swap* lock, twice. For the first run, they each use a maximum table size of 128. For the second run, this is increased to 131,072.

The performance of the *pthread mutex* lock across the two different table sizes is quite similar. The smaller size appears to perform slightly better at the start with the larger size edging ahead for the thread count of four, before both implementations slump as shown in *figure 45.*

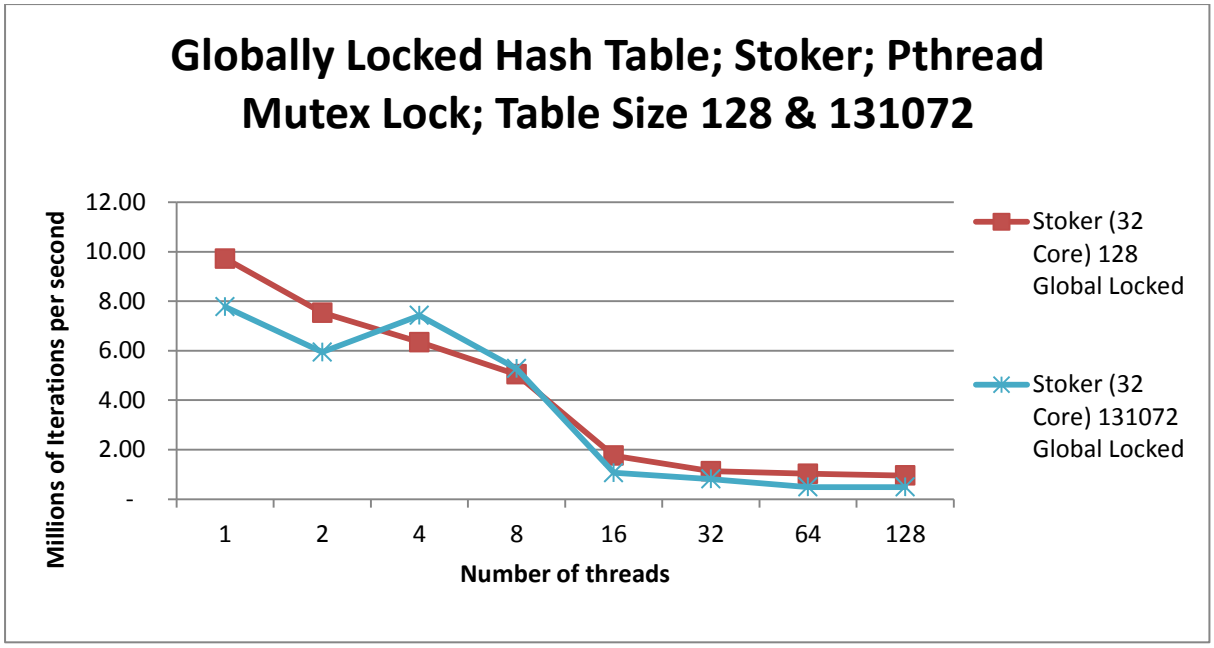


**Figure 45: Millions of iterations/second by number of threads for the globally locked hash table on the machine Stoker. "Table Size 128 & 131072" indicates the maximum number of buckets allowed in the hash table at any one time. "128 Global Locked" and "131072 Global Locked" represent the *pthread mutex* lock implementations respectively with the number representing what table size was used for that implementation.**

The second of the locked implementations, the *compare-and-swap* lock again performs comparably over the two table sizes, though this test is more one sided with the larger table size achieving the better performance of the two on all but one occasion at the thread count of sixty four as seen in *figure 46*.

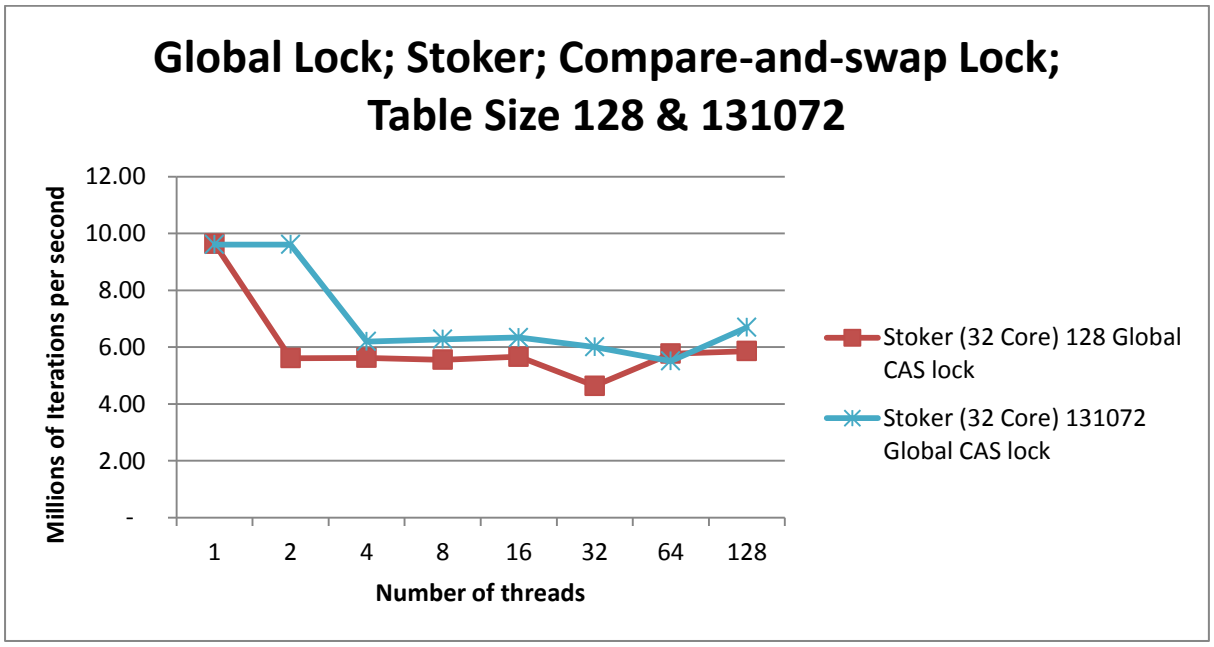


**Figure 46: Millions of iterations/second by number of threads for the globally locked hash table on the machine Stoker. "Table Size 128 & 131072" indicates the maximum number of buckets allowed in the hash table at any one time. "128 Global CAS lock" and "131072 Global CAS lock" represent the *compare-and-swap* lock implementations respectively with the number representing what table size was used for that implementation.**

As a result of these tests, I can confirm that the changing of the table size does seem to affect the performance of the locked implementations of the *globally locked* hash table. However, this performance gain appears to be neither consistent nor particularly large between the two table sizes used and so I would not highlight it as a major concern when implementing the concurrent hash table.

### 4.4.8 Test-and-test-and-set Lock Comparison

I now investigate the relative performances of the different varieties of *test-and-test-and-set* lock implementations with both the *globally locked* and *lock per bucket* versions of the hash table. I do this as I want to investigate how the different locked implementations perform and which one achieves the highest performance across the different thread counts.

For the *globally locked* variety of the hash table, the *test-and-test-and-set* lock implementation is the best performing variation for all thread counts. Though dipping in performance at a thread count of two it then proceeds to stay level for the remainder of the test while the other two locked implementations fall much further in performance as seen in *figure 47*.

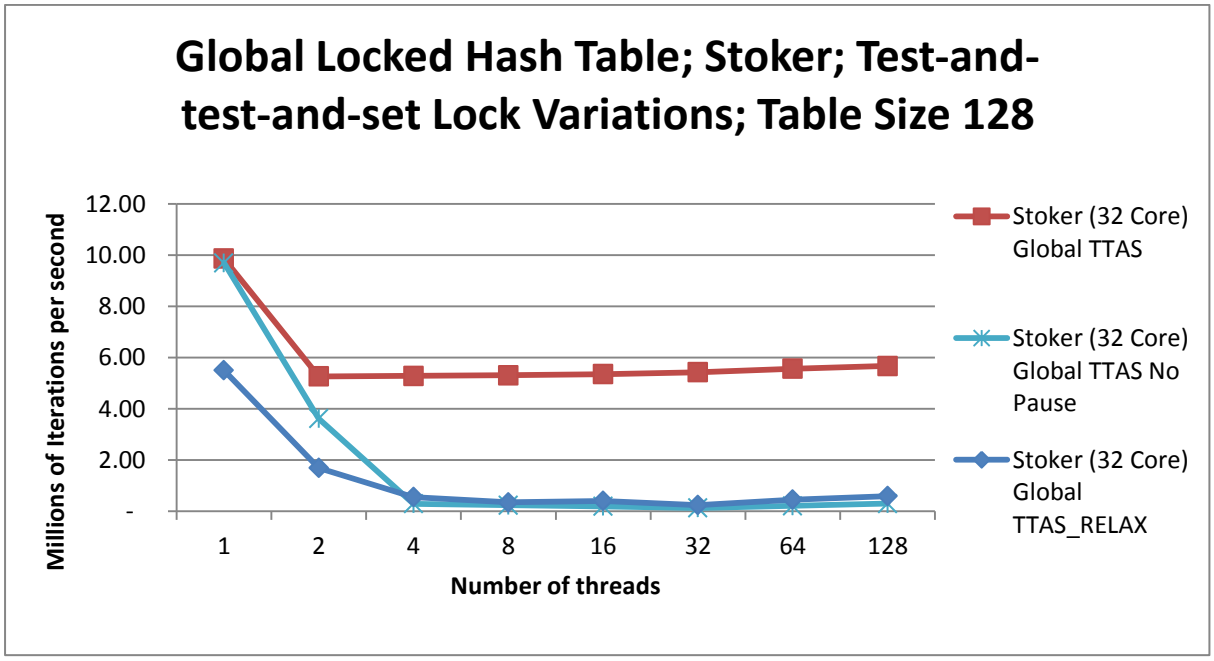


**Figure 47: Millions of iterations/second by number of threads for the *globally locked* version of the hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "TTAS No Pause", "TTAS" and "TTAS_RELAX" represent the *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations respectively.**

| | Cycles | Branches | Branch Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Test-and-test-and-set* | 76.35 B | 13.96 B | 1.59 M | 46.51 B | 34.75 B |
| *Test-and-test-and-set-no-pause* | 1955.67 B | 94.3 B | 59.93 M | 1769.87 B | 639.78 B |
| *Test-and-test-and-set-relax* | 2392.19 B | 24.72 B | 43.37 M | 2334.01 B | 2148.76 B |

**Table 25 : Hardware performance data gathered from the three *globally locked* hash table implementations tested in figure 47. The *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations.**

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 locked implementations shown in *table 25*. Combine that with the lowest number of both cache and branch misses and the *test-and-test-and-set* lock is the most efficient implementation out of the three.

*Figure 48* shows the three *test-and-test-and-set* locks again but this time with the *lock per bucket* implementation of the hash table. Surprisingly, it is the *test-and-test-and-set-no-pause* lock which performs the best up to a thread count of four, at which point the regular *test-and-test-and-set* lock takes over for the remainder of the test.

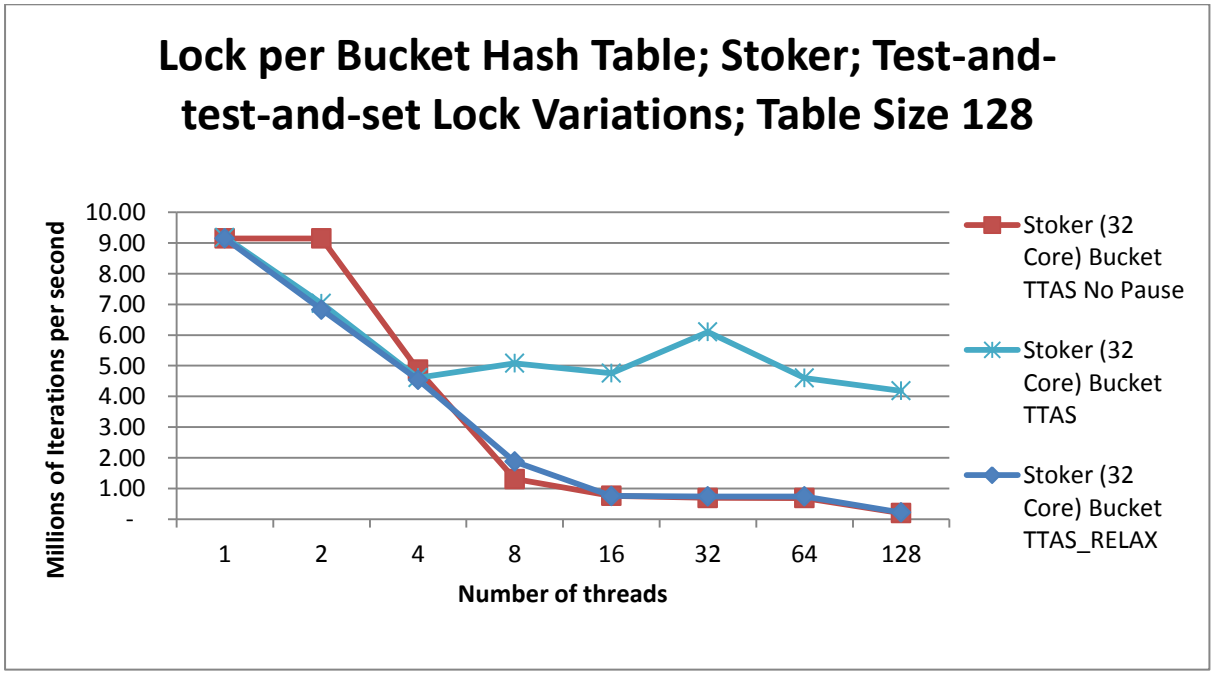


**Figure 48: Millions of iterations/second by number of threads for the *lock per bucket* version of the hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "TTAS No Pause", "TTAS" and "TTAS_RELAX" represent the *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations respectively.**

| | Cycles | Branches | Branch Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Test-and-test-and-set* | 2389.94 B | 224.02 B | 58.82 M | 1964.04 B | 1093.77 B |
| *Test-and-test-and-set-no-pause* | 646.03 B | 39.08 B | 32.68 M | 579.03 B | 354.76 B |
| *Test-and-test-and-set-relax* | 2296.4 B | 58.6 B | 61.85 M | 2198.13 B | 1304.11 B |

**Table 26: Hardware performance data gathered from the three *lock per bucket* hash table implementations tested in figure 48. The *test-and-test-and-set-no-pause*, the *test-and-test-and-set* and the *test-and-test-and-set-relax* lock implementations.**

From *table 28*, the regular *test-and-test-and-set* lock has the lowest branch miss rate, along with the smallest ratio of CPU pipeline stalls. It is due to these reasons that the *test-and-test-and-set* lock does so well at higher thread counts than the other two locks.

An interesting point to note is that the *test-and-test-and-set-no-pause* and *test-and-test-and-set-relax* implementations of the *globally locked* hash table drop in performance much quicker than their *lock per bucket* counterparts. At a thread count of two they are both performing at fewer than four millions iterations per second, however, they don't reach this point until a thread count of eight with the *lock per bucket* variety.

All this indicates that the *test-and-test-and-set* lock implementation should be used for both of the locked variations of the hash table due to its performance in both *figure 47* and *48*. However, for lower thread counts, the *test-and-test-and-set-no-pause* lock in *lock per bucket* achieves the highest performance but this ends at higher thread counts which should be taken into account when designing a concurrent hash table.

### 4.4.9 Test-and-set Lock Comparison

I now want to examine the differences between the different *test-and-set* lock variations on the *globally locked* implementation of the hash table. From *figure 49,* 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.

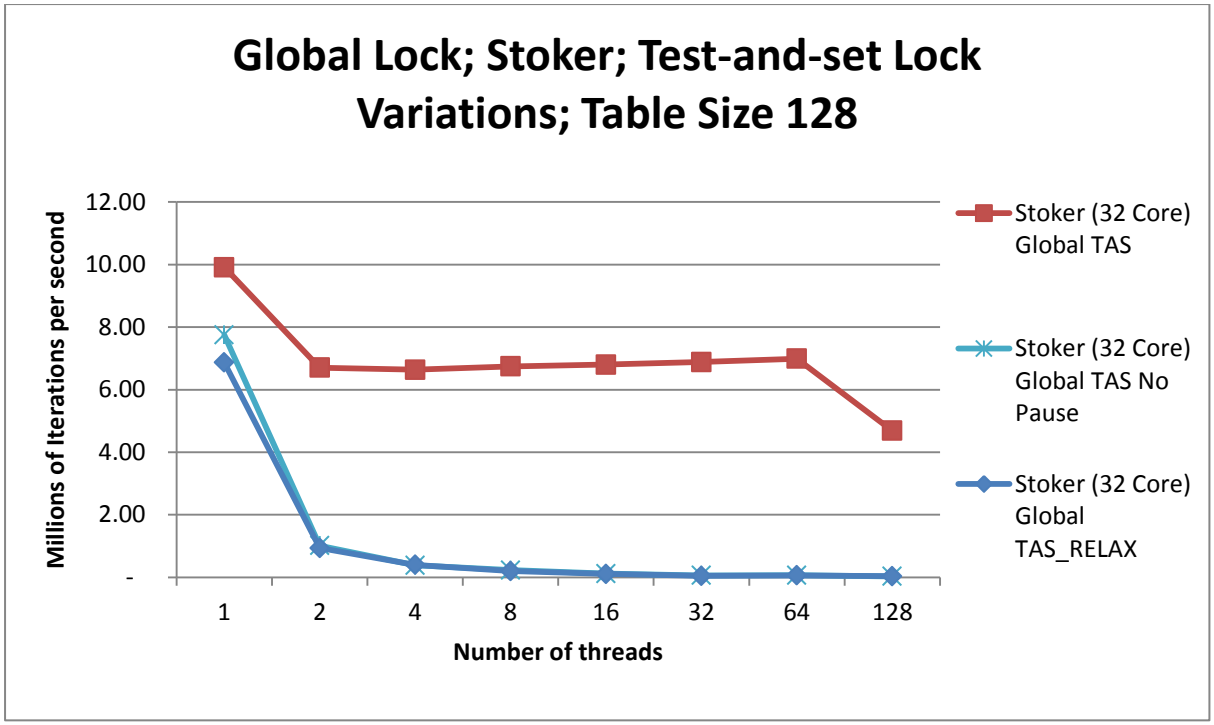


**Figure 49: Millions of iterations/second by number of threads for the *globally locked* hash table on the machine Stoker. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "TAS", "TAS No Pause" and "TAS_RELAX" represent the *test-and-set*, the *test-and-set* and the *test-and-set-relax* lock implementations respectively.**

| | Cycles | Cache References | Cache Misses | Stalled Frontend Cycles | Stalled Backend Cycles |
|---|---|---|---|---|---|
| *Test-and-set* | 89.17 B | 29.33 M | 17.27 M | 53.05 B | 41.09 B |
| *Test-and-set-no-pause* | 2674.89 B | 350.87 M | 283.6 M | 2658.39 B | 2601.32 B |
| *Test-and-set-relax* | 2687.66 B | 235.21 M | 200.51 M | 2668.06 B | 2488.81 B |

**Table 27: Hardware performance data gathered from the three *globally locked* hash table implementations tested in figure 49. The *test-and-set*, the *test-and-set* and the *test-and-set-relax* lock implementations.**

From *table 27* 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. It is also important to note the sheer difference in CPU cycles used. The *test-and-set* lock implementation uses almost thirty times fewer CPU cycles in its execution, thanks to its pause instruction which prevents the threads from constantly polling the lock in its implementation.

This evaluation indicates that for the implementation of a concurrent hash table, only the *test-and-set* lock implementation need be considered out of the three implementations tested in *figure 49*.

### 4.4.10 The Hash Table's Performance across Architectures

To investigate whether the various hash table implementations are robust in their performance across architectures I first examine the *globally locked* variation of the hash table. Seen in *figure 50* 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.

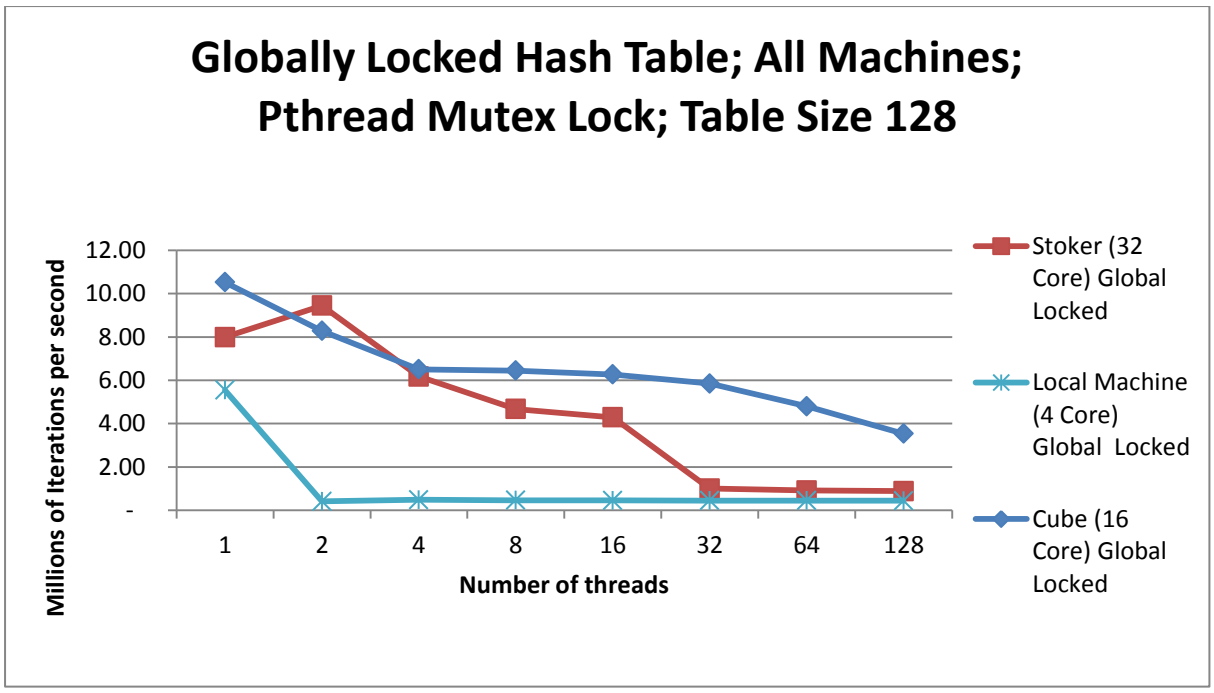


**Figure 50: Millions of iterations/second by number of threads for the *globally locked* hash table on the three architectures of Stoker, Cube and Local Machine. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Stoker Global Locked", "Local Machine Global Locked" and "Cube Global Locked" represent the *pthread mutex* lock implementation, with the architecture name representing which machine the implementation is run on.**

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, whereas the *globally locked* version was somewhat similar for both Stoker and Cube.

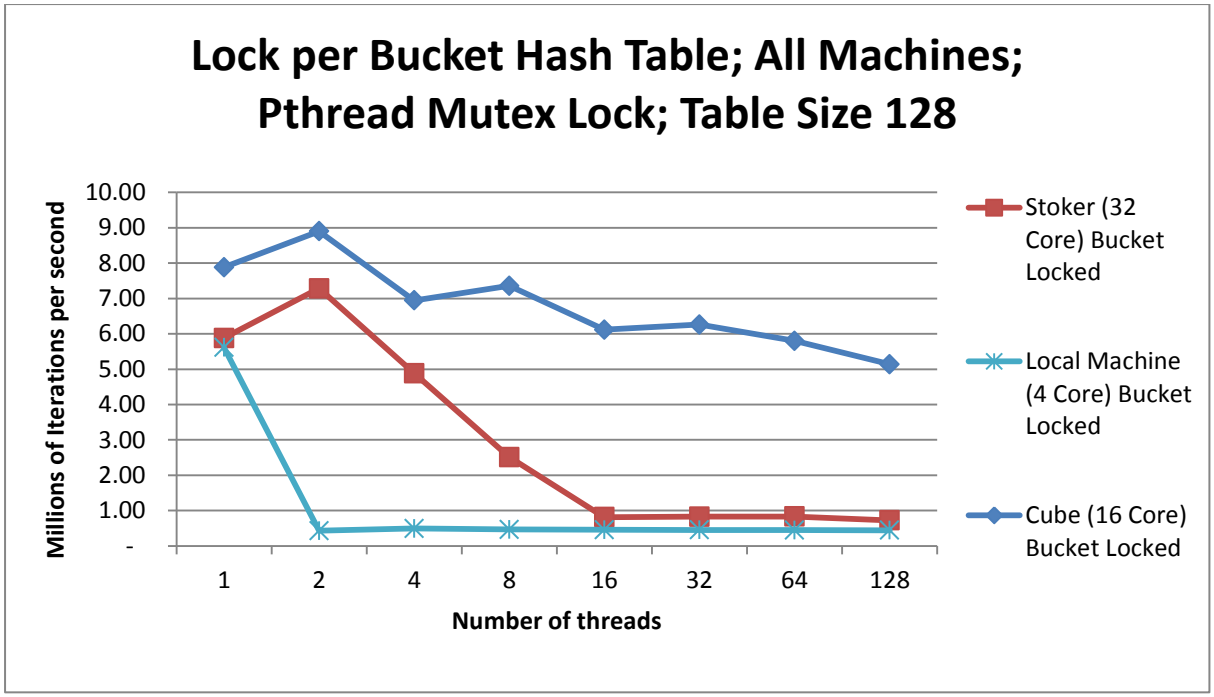


**Figure 51: Millions of iterations/second by number of threads for the *lock per bucket* hash table on the three architectures of Stoker, Cube and Local Machine. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Stoker Bucket Locked", "Local Machine Bucket Locked" and "Cube Bucket Locked" represent the *pthread mutex* lock implementation, with the architecture name representing which machine the implementation is run on.**

I now move to investigate another implementation between the two locked hash table varieties. The *test-and-test-and-set* implementation on the *globally locked* version of the hash table, shown in *figure 52*, appears to be reasonably robust between the machines of Stoker and Cube. The same cannot be said however, for the Local Machine which produces a straight line.

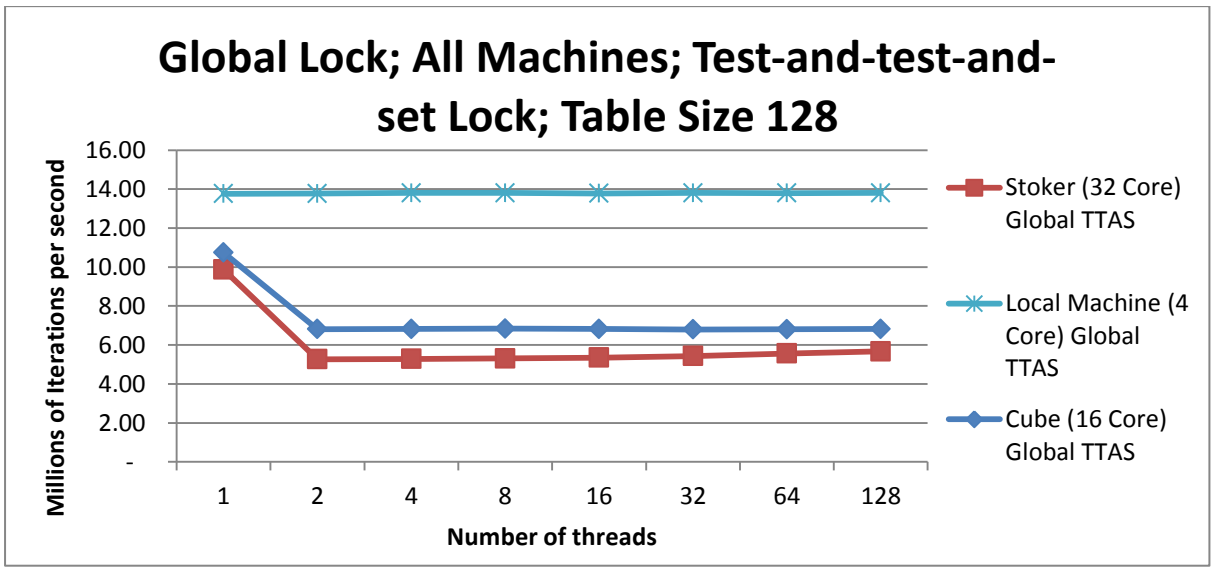


**Figure 52: Millions of iterations/second by number of threads for the *globally locked* hash table on the three architectures of Stoker, Cube and Local Machine. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Stoker Global TTAS", "Local Machine Global TTAS" and "Cube Global TTAS" represent the *test-and-test-and-set* lock implementation, with the architecture name representing which machine the implementation is run on.**

The *test-and-test-and-set* lock implementation also shows less robustness with the *lock per bucket* implementation shown in *figure 53*. This may indicate that the *lock per bucket* implementation is less effective at preserving performance than its alternative the *globally locked* implementation, with Stoker and Cube having more diverse performance patterns for the *lock per bucket* version of the hash table than the *globally locked* version.

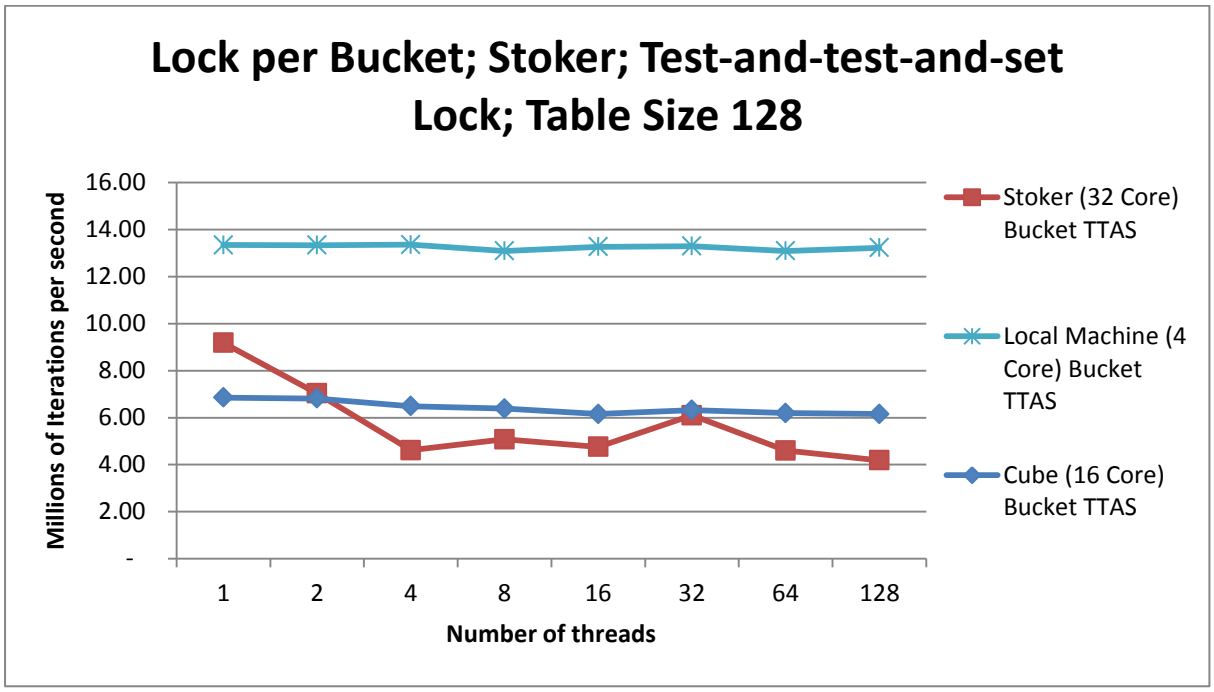


**Figure 53: Millions of iterations/second by number of threads for the *lock per bucket* hash table on the three architectures of Stoker, Cube and Local Machine. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Stoker Bucket TTAS", "Local Machine Bucket TTAS" and "Cube Bucket TTAS" represent the *test-and-test-and-set* lock implementation, with the architecture name representing which machine the implementation is run on.**

Finally, I examine the lockless version of the hash table to discern how robust it is across different architectures compared to the other two varieties of hash table. Examining the *lockless* variation of the hash table, in *figure 54*, 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.

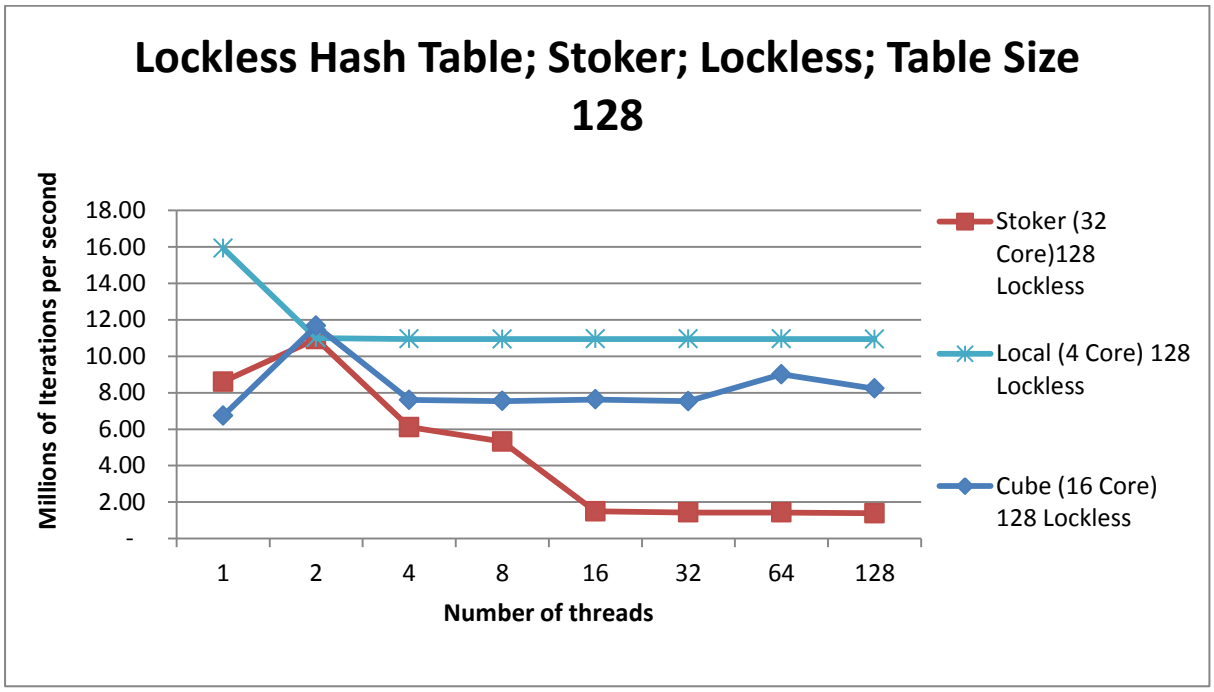


**Figure 54: Millions of iterations/second by number of threads for the *lockless* hash table on the three architectures of Stoker, Cube and Local Machine. "Table Size 128" indicates the maximum number of buckets allowed in the hash table at any one time. "Stoker Lockless", "Local Machine Lockless" and "Cube Lockless" represent the *lockless* implementation, with the architecture name representing which machine the implementation is run on.**

# 5 Afterword

## 5.1 Conclusions

Overall I am quite pleased with how the project went. I feel that I was able to conduct accurate research into the performance differences between concurrent data structure implementations. In addition, I have investigated whether the performance of these concurrent data structure implementations is maintained across three different architectures.

When I started this project I had already had some experience with concurrent programming, however, I could have benefited from more time actually designing and implementing concurrent programs. This project has provided that experience and I now feel much more confident dealing with the issues of multithreaded applications and writing code designed for concurrent access. I look forward to further reading on the subject and I hope that I will get another chance to apply what I have learned about concurrency and algorithmic evaluation in the future.

I will now describe the general conclusions and contributions that can be taken from this project.

Across all concurrent data structure tested, the best locked implementations proved to be the *compare-and-swap*, *test-and-set* and *test-and-test-and-set* locks which consistently outperformed the other locks, such as the *ticket* lock.

For every case, 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 for medium to high thread counts of four threads and above. There are cases where the *test-and-test-and-set-no-pause* lock implementation outperforms the other two, but these instances are confined to thread counts of one or two, where the lack of a *sleep()* instruction does not impact the performance of the implementation to a large degree.

The same can be said for the *test-and-set* utilising the *sleep()* instruction, outperforming both the *test-and-set-no-pause* and *test-and-set-relax* lock implementations, especially at the higher thread counts of eight and onwards.

Continuing the trend, the *compare-and-swap* lock implementation achieves the highest performance out if it, the *compare-and-swap-no-pause* and the *compare-and-swap-relax* lock implementations.

The various *lockless* implementations usually perform well at early thread counts. However, they appear to be heavily impacted by the effects of high thread contention and so mostly drop in performance after a thread count of four.

In general, none of the implementations tested, both locked and *lockless* proved to be consistently robust across any of the architectures.

A summary of the specific findings for each concurrent data structure implementation is detailed below:

### 5.1.1 Ring Buffer

The performance of the *lockless* implementation of the ring buffer appears to be affected by the size of the array used. It is inconclusive however, as to whether an increased size infers increased performance.

The results gathered to determine whether the maximum length of the buffer affects the performance of the locked implementations 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, while most locks do not retain their performance across architectures, this is not the case for some. The *test-and-set* lock behaves similarly across the three architectures but others are not so robust.

### 5.1.2 Singly Linked List

The *lockless* implementation performs extremely well compared to the locked implementations; however, this is only the case when the maximum number of nodes allowed in the list is increased to 131,072. This is due to the multiple threads being able to traverse the list concurrently compared to the locked implementations where only one thread is allowed to traverse at any one time. This particular implementation proves to be the best performing *lockless* implementation relative to the performance of the locked varieties out of all the *lockless* implementations tested during this project.

### 5.1.3 Doubly Linked Buffer

The *ticket-relax* implementation achieves a far better performance at the thread count of two when compared to the *ticket* lock implementation, though this then dips sharply at a thread count of four.

### 5.1.4 Singly Linked Buffer

The *lockless* implementation performs similarly to the *pthread mutex* lock, all the while being outperformed by the *test-and-test-and-set* lock implementation. This proves to be the worst performing *lockless* implementation out of all tested as at no point does it perform better than a locked implementation.

The *ticket-relax* lock outperforms the *ticket* lock in much the same way as with the doubly linked buffer, with a huge performance margin at a thread count of two but then falling sharply afterwards.

### 5.1.5 Hash Table

The *compare-and-swap* lock implementation with the *lock per bucket* variety of the hash table appears to be far more susceptible to performance loss at higher thread counts than its counterpart in the *globally locked* hash table.

The *globally locked* variety of the hash table consistently performs equal to or better than the *lock per bucket* variety.

The *resize* functionality has a large impact on the performance of the implementations tested, with major dips in throughput whenever a *resize* occurs during execution.

The *lockless* implementation performs poorly against the locked implementations of the hash table.

## 5.2 Future Work

A topic to address in future work would be the issue of memory management. I decided not to implement memory management as part of my data structure implementations because I felt that this would have added another degree of difficulty to the project. Time spent implementing memory management solutions for the data structures would have detracted from the main focus of the project. Further work could be linked to taking the data structures I have implemented and designing memory management solutions for them. These could then be tested in order to see how they affect the performance of the concurrent implementations.

Another issue with my work is that I have a limited understanding of the implementations' performances due to the fact that I gather aggregated hardware performance data for each implementation, in that for example, each implementation I test returns one figure which represents all thread counts, as opposed to figures for each thread count of the implementation. While the aggregated approach gives me a snapshot of an implementation's performance as a whole, a thread count by thread count hardware counter analysis could be useful. This would allow performance to be analysed for different thread counts and may help explain some of the behaviour I have observed during this project.

In relation to hardware performance counters again, I feel that my implementations could have been improved if I had gathered hardware counter data during the implementation phase of my project. This would have allowed me to observe how my implementations were performing as they were being designed and may have

helped me identify any parts of my code that were resource heavy to allow for optimisation or redesign to take place.

I gathered a third size for many of the concurrent data structure implementations of 134,217,728, though due to time constraints I was unable to use it for anything useful, except for comparing the ring buffer's *lockless* implementation. It may be interesting to evaluate each implementation using this third value and see how it compares with the value of 128 and 131,072, the other values used.

Due to time constraints I feel that I was unable to fully evaluate the three hash table versions that I implemented. For example, I was unable to do a comparison between the different *compare-and-swap* lock implementations or the *ticket* lock implementations as I did for previous data structure evaluations. I feel that additional work could be done on analysing more of the data collected from the hash table implementations than I was able to do.

Finally, while I used C++ for implementation, it would be interesting if my project were to be implemented using python or nodejs, to see how the performance of the data structures changes when exposed to an asynchronous environment.

# 6 Bibliography & Appendix

## 6.1 References


1. *Atomic Operations Library*. (2013). *Atomic operations library.* Available: http://en.cppreference.com/w/cpp/atomic. Last accessed 13/04/2014.
2. Blaise Barney, Lawrence Livermore National Laboratory. (2013). *POSIX Threads Programming.* Available: https://computing.llnl.gov/tutorials/pthreads/. Last accessed 23/02/2014.
3. Eles. (n.d). *Pipeline Branch Penalities.* Available: http://www.ida.liu.se/~TDTS57/info/lectures/lect7-8.frm.pdf. Last accessed 17/03/2014.
4. GCC Administrator. (2001). *GCC Command Options.* Available: http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_3.html. Last accessed 05/03/2014.
5. Herlihy (1993). *A Methodology for Implementing Highly Concurrent Data Objects*. USA: ACM.
6. Herlihy & Shavit (2008). *The Art of Multiprocessor Programming*. USA: Elsevier Inc.N/A (17/07/2013).
7. Horvath. (2012). *Faster division and modulo operation - the power of two.* Available: http://blog.teamleadnet.com/2012/07/faster-division-and-modulo-operation.html. Last accessed 08/02/2014.
8. Intel Corporation. (2011). *Pause Intrinsic.* Available: https://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/intref_cls/common/intref_sse2_pause.htm. Last accessed 09/04/2014.
9. Intel® Xeon® Processor E7-4820. (n.d). *Intel® Xeon® Processor E7-4820 v2 (16M Cache, 2.00 GHz).* Available: http://ark.intel.com/products/75246. Last accessed 17/04/2014.
10. Intel® Xeon® Processor E5520. (n.d). *Intel® Xeon® Processor E5520 (8M Cache, 2.26 GHz, 5.86 GT/s Intel® QPI).* Available: http://ark.intel.com/products/40200/Intel-Xeon-Processor-E5520-8M-Cache-2_26-ghz-5_86-gts-Intel-qpi. Last accessed 17/04/2014.
11. Intel® Core™ i5-2500K Processor . (n.d). *Intel® Core™ i5-2500K Processor (6M Cache, up to 3.70 GHz).* Available: http://ark.intel.com/products/52210/intel-core-i5-2500k-processor-6m-cache-up-to-3_70-ghz. Last accessed 17/04/2014.
12. Linux profiling with performance counters. (2013). *Linux profiling with performance counters.* Available: https://perf.wiki.kernel.org/index.php/Main_Page. Last accessed 15/04/2014.
13. Lockless Inc. Spinlocks and Read Write Locks. Available: http://locklessinc.com/articles/locks/ . Last accessed 29/03/2014
14. Moir & Shavit (2001). *Designing Concurrent Data Structures*. USA: CRC Press.

15. rand(3) - Linux man page. (n.d). *rand(3) - Linux man page.* Available: http://linux.die.net/man/3/rand. Last accessed 03/04/2014.
16. sleep. (1996). *sleep(3) - Linux man page.* Available: http://linux.die.net/man/3/sleep. Last accessed 27/03/2014.
17. sfuerst. (n.d). *Spinlocks and Read-Write Locks.* Available: http://locklessinc.com/articles/locks/. Last accessed 2/04/2014.
18. The Internet Society. (n.d). *Cube.* Available: https://wiki.netsoc.tcd.ie/index.php?title=Cube. Last accessed 21/04/2014.
19. Triplett et al (n.d). *Resizable, Scalable, Concurrent Hash Tables*. S.l.
20. usleep(3) – Linux man page. Available: http://linux.die.net/man/3/usleep. Last accessed 29/01/14.
21. volatile (C++). (2013). *volatile (C++).* Available: http://msdn.microsoft.com/en-us/library/12a04hfd.aspx. Last accessed 19/02/2014.