Fair data structures for stronger performance isolation guarantees

Rachel Somerset

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2024

Abstract

In shared environments such as operating systems, database servers, and hypervisors we often come across situations where entities with varied requirements compete to access shared resources. Ensuring fair resource allocation between the users requires careful scheduling and proportionate opportunity for each user.

In this thesis we introduce data structures as one such resource, and ask if we can fairly allocate this resource between entities. We successfully taxonomise data structures into the classifications: Shared, Correct, and Performance and discuss the foundation that this builds for fair allocation. We discuss what fair allocation looks like for a data structure and define fairness for this shared resource as 'each entity bares to cost for it's own actions' or performance isolation.

We propose a fair data structure that, unlike existing structures, guarantees performance isolation through partitioning the data on a per user or function basis. Using microbenchmarks, we show that our fair data structure is efficient and achieves high performance with minimal overhead under extreme workloads. We also show that our solutions are easily adaptable to the implementation of other complex data structures.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Rachel Somerset*)

Acknowledgements

I would like to fairly allocate my thanks to several entities (people).

To my supervisor Dr Yuvraj Patel for all his guidance and help. Thank you for the many sheets of paper covered in red ink, never taking a holiday, and your analogies. I have thoroughly enjoyed working on this project, will forever be thankful for having an excellent supervisor.

I would also like to thank my parents, my boyfriend, and, to keep it fair, my sisters. Thank you for your constant support throughout my years at University. I am very grateful for all the proof reading you have done, the meals you have made and, when needed, leaving me alone.

Table of Contents

Chapter 1

Introduction

Data structures serve as the foundational framework for organizing and manipulating data efficiently, playing a pivotal role in various computational tasks and algorithmic implementations. The significance of data structures in shared systems lies in their ability to facilitate efficient and organized data management, enabling collaboration and resource allocation among multiple users within a networked environment. Today, shared systems play an important part in promoting collaboration, resource sharing, and in coordinating users or entities. In these systems the scheduling of resources between these users is important to promote their popularity and growth, and we must strive to constantly keep up with the growing demands placed upon them.

One such way to do so - concurrency - refers to the ability of a system to execute multiple tasks simultaneously, providing the illusion of parallelism. Concurrency is essential for improving system performance, responsiveness, and resource utilization. It allows programs to efficiently handle multiple tasks which is crucial in various domains such as operating systems, distributed systems, and parallel computing. Concurrency is typically achieved through mechanisms such as multithreading, multiprocessing, and asynchronous programming. Multithreading involves executing multiple threads within the same process, allowing different tasks to run concurrently. Multiprocessing, on the other hand, involves running multiple processes simultaneously, leveraging multiple CPU cores for parallel execution. Asynchronous programming enables non-blocking execution of tasks, allowing programs to perform other operations while waiting for certain tasks to complete. These enable programs to efficiently utilize modern hardware with multiple cores and support the development of highly responsive and interactive applications.

However, concurrency also introduces challenges and limitations. One of the major challenges is the complexity of concurrent programming, which can lead to issues such as race conditions, deadlocks, and thread synchronisation problems. These issues are notoriously difficult to debug and can lead to unpredictable behavior and system failures. Another limitation of concurrency is the potential for performance bottlenecks, especially in scenarios with shared resources. Contention for shared resources can degrade performance and scalability, requiring careful design and optimization strategies to mitigate. Additionally, achieving optimal concurrency often requires careful

consideration of trade-offs between parallelism and overhead. Fine-tuning concurrency mechanisms and synchronisation primitives to balance performance, scalability, and complexity can be challenging and time-consuming.

The scheduling of these resources so that all users or tenants are operating at their highest potential has led to optimising memory usage, network traffic, CPU schedulers, and even focused into common/significant spots of contention within such systems such as locks. optimisations look at allocating these resources fairly between the entities so that all have equal or proportional opportunity and the behaviour of each entity has no impact upon the performance of any others —performance isolation.

We add to this by now considering shared data structures. In such systems as mentioned above shared data structures are used to communicate between entities, provide access to global information, and for performance enhancing purposes whilst saving on memory. We find that in addition to access to such structures as a point of contention, the behaviour of entities within the data structure is intertwined, and malicious or expensive actions within such structures can seriously damage the performance of unassuming threads by taking advantage of, or inadvertently causing, unfairness within.

In this thesis we ask the question: Can we start treating data structures as a resource and ensure fair allocation to entities?

1.1 Overview

- We start this thesis with some essential background into concurrent systems, mentioning on cloud based systems and operating systems where shared data structures are often used. Then we move to where contention comes from within these systems and synchronisation methods used to ensure the correctness of shared data when such contention arises.
- To study contention in data structures we first work with a linked list to highlight the extent of the damage unfairness can do and discuss known attacks that manipulate the unfairness that we bring to light.
- We classify all data structures into Shared, Correct, and Performance and describe these in detail. By producing a detailed taxonimisation we begin the discussion into data structures as a resource to be shared.
- We propose a solution of a fair data structure that removes the unfairness through partitioning the data in such a way as to guarantee performance isolation for those structures within the Non-Shared and Shared Correct classifications.
- Using micro benchmarks we show that in a variety of synthetic workloads, our fair data structure achieves the desired behaviour. We also investigate the scalability of the solution, showing that it is adaptable and provides high performance when used by many applications concurrently with high workloads.
- Finally we evaluate our solution and discuss the future work stemming from this thesis.

1.2 Outline

this thesis contains the following sections: We start with a background chapter presenting important preliminary topics to the project, then there is a motivation chapter on the reasons and drive behind the research we have performed into fairness within data structures. Then we discuss the design and implementation details of our main contribution - our fair data structure in the design chapter. This is followed by the evaluation chapter where we discuss our experiments and empirical evaluations of our solution. Finally we summarise, conclude, and describe potential future research in the chapter Conclusions and Future Work.

Chapter 2

Background

2.1 Concurrent Systems

Concurrency is an optimisation used within computers to improve performance [18] . Every task to be completed by a computer can be broken up into sections of executable code that don't necessarily need to be performed sequentially. These can be run by different CPUs in parallel.

2.1.1 Cloud Systems

Many tasks these days are data driven and require working on data sets of significant sizes it is common practice now to use a cloud-based system as they are scalable, adaptable, and much better equipped than a single computer to run such a task. A key aspect and benefit of cloud computing is distributed processing allowing a task to be executed across multiple computers. This approach enables large amounts of data to be processed in parallel taking a fraction of the time it would sequentially.

There are three different service models for cloud computing [1]; The Infrastructure-as-a-Service (IaaS) which uses virtualization to share hardware resources, the Platform-as-a-Service (PaaS) model which hosts the applications of customers, and the Software-as-a-Service (SaaS) which provides an up and running hosted application. In all three of these systems, we encounter the same concurrency, performance, and security concerns. Multi-tenant Applications will share one instance of an application between many tenants and provide each tenant a dedicated share of this instance that is isolated from the others. This isolation is easy as these issues are handled by the application domain. In most cases, all the tenants will use an application in a similar way however the number of users and the peak times might defer, and the tight coupling of tenants results in strong interference of non-functional system properties. A hypervisor runs several virtual machines on the same hardware. A virtual machine is a computer which is not directly accessing the hardware by leveraging virtualization. Thus, several virtual machines can run in parallel and share the resources. However, research done by [13] shows that different virtual machines can interfere with each other whilst running at the same time.

Isolation between these tenants and applications in such systems is vital, to provide security and reduce potential loss of performance [26]. Sometimes the system itself can have some control over the performance of a single user if the minimum level service agreement is not violated. An extreme solution is to isolate the computational resources; however, this leads to a lot of idleness within the system. Other solutions are to utilise the scalability of the cloud system to scale itself up or down depending on the necessary computational power to handler tenant requests, however malicious users can find ways around this and they do not ensure complete isolation or fairness between all parties.

2.1.2 Operating Systems

Operating systems are another place where we see the constant use of concurrency. In operating systems concurrent tasks are assigned to a CPU by a scheduler. Schedulers are a separate process that organises the running of tasks based upon their priority, length of time or some other feature [10]. On some architectures a CPU may also have more than one thread assigned to it – multithreading – and must switch between the threads to keep them all progressing forward [17].

Schedulers define how resources are allocated using a scheduling algorithm There are many different schedulers, and the vastness of these implementations mirrors its importance in ensuring a well performing system [3], [10], [11]. Scheduler fairness is extremely well researched. Fair schedulers focus on ensuring that processes get ‘fair’ CPU time, whether that be an equal amount, based upon task priorities, or task workload. The most common schedulers known are those currently used by major operating systems such as the Completely Fair Scheduler[3] used by linux. The paper ‘Proportionate Progress: A notion of fairness in resource allocation’ defines a new fairness p-fairness which is related to resource allocation by the scheduler [2]. The approach is to maintain a proportionate progress: ‘each task is scheduled resources in proportion to its weight’. We call this proportionate fairness or P-fairness. This measurement of fairness is used in other papers which compare the fairness of different current operating systems [3]. The paper compares current scheduler implementations to the ‘idealised’ fair scheduler and, uses the notion of ‘lag’ to measure fairness within these schedulers and in resource allocation. Lag measures the difference between the number of resource allocations that task x ”should” have received in the set of slots [0, t] and the number that it received. So fair resource allocation and therefore a fair scheduler come from reducing this lag.

2.2 synchronisation

Now we are more familiar with concurrent systems we can discuss concurrent data structures. This is organised data that is stored in shared memory and accessible to multiple tenants [24]. They are very useful and are used in all areas from data centres, to hypervisors, to operating systems. Modern day computer systems and data centres have hundreds of processes running concurrently and there is the potential that all could interact with a single shared data structure. As the performance of computers is pushed to achieve greater and greater speeds each task, process, and thread is subject to

individual performance requirements that must be met. Ensuring that each application, tenant, or thread does not detrimentally hold back others to meet its own requirements is known as performance isolation [27]. Accessing a shared data structure is a likely way to fall short of performance isolation requirements so systems must be fast and correct when accessing these structures and extra mechanisms need to be put in place to ensure no errors occur if two or more threads try to use the data simultaneously. Cloud servers and data centres often contain many shared data structures which can cause a bottleneck of interaction. Tenants can be unregulated in such systems so these structures could be shared with malicious users or even genuine users performing heavy work on the shared system.

If a data structure is accessed at the exact same time the execution of different applications/threads can become interleaved [14]. A simple example of this is a counter. We start with two threads (referred to as thread A and thread B) on a single CPU, both want to increment an integer counter variable. The instruction for this is broken down into:

a. Load the counter into a local register (at its current value).

b. Increment the counter (current value + 1).

c. Store it back into the memory.

If we do not put restrictions on the threads, we may have thread A load the counter into its registers, and then increment the counter. However, before it can store the new value back to memory the CPU switches to thread B. Thread B then loads the old counter value from memory into its registers, and increments it, and then stores back to the memory. Thread A then does a store and overwrites the value written by thread B. Instead of incrementing the counter by two, the value in memory will now only show an increase of one. As you increase the number of threads and CPUs this interleaving of the instruction is only exacerbated, and the value stored in memory from the counter becomes wildly incorrect. In this scenario the scheduler oversees which thread is running on the CPU and so the amount of CPU time each thread has is usually predetermined based upon previously mentioned factors i.e. threads with higher priority will get more CPU time. In terms of performance, this is a clear example of how the two threads are not isolated.

2.2.1 Locks

One method to prevent tenants read and writing to a memory location simultaneously is to use a lock [14], [16]. The sections of code the threads execute that write to shared variables and data structures are defined as critical sections. Locks prevent two threads from entering overlapping critical sections at the same point by ‘locking’ the saved variable when a thread accesses it. Adding a simple lock to the counter variable, when thread A calls increment it first acquires the lock to stop other accesses, and then performs instructions a. and b. The CPU then switches to thread B, which tries to acquire the lock and cannot as it is still held by thread A. Thus, it remains at this stage trying to acquire the lock. Once the CPU switches back to thread A, it carries on and stores the value back to the memory location and now releases the lock. Thread B can acquire the lock once the CPU switches back and can perform the increment resulting

in the counter correctly being incremented by two. This basic example is inefficient as it 'blocks' threads that are waiting to acquire the lock, so performance is not good, but correctness is guaranteed. It offers no guarantee of equality between threads as the same process can acquire the lock continuously if it has a short non-critical section and others need to be woken up before they themselves will try to acquire it. Deadlock [4] can happen where two threads hold two locks, and each want the others lock but can't move forward and release the lock they hold so neither thread progresses. Priority inversion [23] is where a high priority thread is stuck waiting for a low priority thread that holds the lock and so it cannot move forward, and this affects the performance. Various popular versions of locks have been designed to combat these issues, such as a ticket lock [21] which records the order that threads attempt to acquire the lock and allows them access in that order. A priority lock [23] allows threads of a higher priority to acquire the lock before those of a lower priority to combat the priority inversion problem. We follow with some relevant examples of locks.

2.2.2 Pthread spinlock

A spinlock is a basic version of a lock and is commonly found in systems. We use a version of this, the pthread spinlock, which is provided by the pthread library [22]. The lock is made up of a single integer variable that can be set to 0 (unlocked) or 1 (locked). Atomic commands are used to perform the updates to this integer. It is initialised to 0 so the first thread to attempt a to acquire the lock, used the atomic command test-and-set to set the value to 1, indicating it is locked. If it is successful, it carries on into the critical section, if it fails, it spins, checking the value of the shared integer until it returns 0, then it tries atomically to set it to 1 again. If it fails it continues spinning following the same pattern as above. To evaluate the spinlock, we can see its simplicity, the ease in implementing it, and it has minimal memory overhead, however there is no fairness between threads trying to acquire the lock. We provide some psuedocode of a basic implementation in Algorithm 1 in Appendix A.

2.2.3 Scheduler Cooperative Lock

A lock that is relevant to this thesis is the Scheduler Cooperative Lock [19]. This advanced lock guarantees 'lock usage fairness' and 'lock acquisition fairness'. They develop a new measurement 'lock opportunity' which is defined as 'the amount of time a thread holds a lock or could acquire the lock, because the lock is available'. The SCLs achieve fairness by the following components:

1. **Lock usage accounting:** Each thread has an allocated (fair or proportional) amount of lock opportunity. Each lock keeps track of its usage across each thread that can access it.
2. **Penalizing threads depending on lock usage:** Each thread receives a quota and once that quota has been reached the thread is forced to sleep if they prematurely try to reacquire the lock.
3. **Dedicated lock opportunity using a lock slice:** This is a window to time dedicated to a thread in which it can acquire and release the lock when it would

like without interference or competition.

Returning to the scheduler unfairness example above with the addition of a scheduler cooperative lock on the shared variable. If thread A holds the lock and is within its allocated time slice it can increment the counter variable. If the CPU switches threads during thread A's critical section, then thread A holds the lock but only for the remainder of its time slice. Thread B cannot immediately acquire the lock, it must wait for thread A time slice to be up and then it can acquire the lock and its own time slice will begin. If thread B continues the attempt to acquire the lock more than its fair share it will be penalised and put to sleep. Thus, fairness has been guaranteed and a solution to the problem known as scheduler subversion has been provided. SLCs performance is comparable with other implementations of locks with the added benefit of this fairness. One drawback however is that in systems where many different platforms and applications are integrated together there is no way to guarantee which lock is being used to protect concurrent data structures.

2.2.4 Non-blocking algorithms

Non-blocking or lock-free alternatives are available that use the hardware and atomic instructions to ensure that sequential ordering of threads is preserved but does not put to sleep threads that are waiting. This alternative guarantees forward-progress of 'some' thread even if others are delayed. Whilst these have benefits over locks, they are logically more difficult to design and implement and can often be slower than a well performing correct lock. While little research has been done into how fair these lock-free alternatives are or how to guarantee fairness one paper Fair synchronisation by Gadi Taubenfeld [25] provides a solution to the fair synchronisation problem. He defines this by the following three points:

1. **Progress:** If a process is trying to enter its fair section, then some process eventually enters its fair section.
2. **Fairness:** A process cannot complete its fair section twice before a waiting process complete its fair and exit sections once.
3. **Concurrency:** All the waiting processes which are not enabled become enabled at the same time.

He then provides a fair synchronisation algorithm that can integrate into a data structure to provide a fair synchronisation data structure. This guarantees that processes get the same amount of access to the data structure and that once a process has been given access, it cannot then access it again until all other processes have had access. The paper defines any data structure combined with this algorithm as a fair data structure.

2.3 Data Structures

2.3.1 Linked List

A linked list is a simple data structure of linked nodes that contain at the least a piece of data and a pointer to the next node in the list. We can perform three operations on the linked list: insert, find, and delete. The insert() function creates a new node struct, with the data entered as a parameter put into the variable. The nodes *next pointer is then updated to the head of the list and the linked list structure is then altered to point at the new node. The find() function starts at the head of the linked list and iteratively checks if the data stored in that node matches the value being searched for. If so, it returns that node, if not it moves to the next node until it reaches the end of the list. If it has not found the value it returns NULL. The delete function is similar to the find function however if it finds a node with the value it then points the previous node to the next node around the found value and then releases the memory where the node is stored. The psuedocode for a linked list can be found in Algorithm 3 in Appendix A.

There are many implementations that may vary slightly. Some implementations have a key for each node, which can be searched for instead of the data value. Some implementations will not allow duplicates of data (or keys), others may act as queues and add the new nodes to the end of the linked list and other variations.

2.3.2 Hash table

The psuedocode is shown below in Algorithm 4 in Appendix A. A hash table consists of a data structure and a hash function. The hash function takes in a key value and maps it to a fixed set of values. It may be a simple function such as for a key X and a fixed set [0, .., N] we may have X mod N or it may need to be purposed for the use of the hash table for example if it needs to be secure, or needs some keys to be hashed to the same number. The hash table is made up of a predefined N buckets. The mapping of the hash function defines which bucket a node will be stored in. A collision occurs when two nodes are hashed to the same bucket. Again, different implementations deal with this differently, we choose to map these nodes to the same bucket using a linked list. When a new node is inserted at a bucket with entries already it is treated as a linked list insert and added to the head of the bucket. Other implementations include rehashing to move around bucket entries. A hash table has the same three functions as the linked list, insert(), find() and delete(). The insert function takes a key and value and creates a node to be inserted. The hash function is used to calculate which bucket the entry is added to, and within the bucket the node is inserted to a linked list using the linked list functionality. The find function takes in the key parameter and uses the hash function to calculate which bucket to search. It then iterates through the entries in this bucket as a linked list. The delete function acts similarly to the find function however it deletes the node if it finds it.

Chapter 3

Motivation

3.1 Unfairness within Data Structures

'Each entity bares the cost of its own actions.'

Within concurrent systems data structures are used to store and share information between entities, it is their key method of communication (users/applications/threads). These are a global structures maintained in the shared memory that rely on scheduling, accounting and namespaces to prevent interference. Fair schedulers consider CPU time as a resource, and fairness as splitting this resource equally or proportionally between entities. The Scheduler Cooperative Lock also takes this approach with fairness in the lock, it defines lock opportunity as a resource and allocates this fairly between the interacting entities. The purpose of this thesis is to ask: Can we start treating data structures as a resource and ensure fair allocation to entities? Fair opportunity and access are provided by a fair lock so we in this thesis look at fairness not as equality but as striving to performance isolation i.e. the performance of each entity whilst in the structure is only affected by its previous actions.

3.2 Simulated Example

3.2.1 Scenarios

To start to answer the question we need to understand the action that can occur within data structures.

We choose to focus at this stage on a linked list. This is because it is a simple data structure with only a couple of operations, that is commonplace in cloud based and operating systems. Linked lists form the base for more complex data structures such as queues, hash tables, and graph structures so by looking at fairness of a linked list we can understand fairness in many other data structures. These scenarios provide a comprehensive list of what behaviour we might expect to see within a linked list, and cover the different purposes and features that a data structure might require that could affect these scenarios. From these scenarios we are able to uncover the behaviour of

these structures, decide what is fair or unfair, and then start to look at what would make it fair. There are many cases of unfairness as shown in table B in Appendix B, that can be solved with something like the scheduler cooperative lock, however we uncover unfair cases where there the fairness cannot be resolved —this is where we want to work.

In all of these scenarios what we see as unfair is an interaction between the threads of the linked list, this is exacerbated by the threads showing increasingly different behaviours such as one having a higher insertion rate or one not accessing the data structure until later on in it's task.

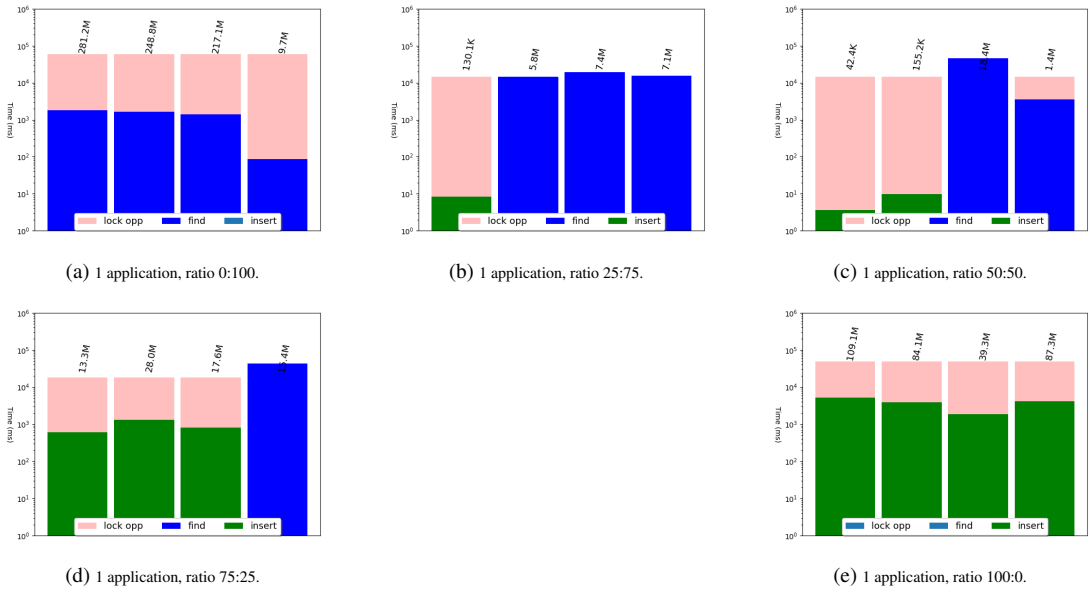
3.2.2 Results

Base case In order to show the unfairness in the scenarios we run some simulated examples on a linked list. Following the example of [15] we group our threads in 4 as applications that share data within them. These threads perform one of two functions, they either continually insert entries or they perform a find on the linked list. We introduce an insert ratio parameter which is the ratio percentage of threads that are insert threads for each application. This allows us to simulate the above scenarios by changing the ratios of each thread, and thus alter the behaviour of each application.

We run a single application initially and then two applications in parallel. To ensure there is no interference we run the experiments on Cloudlab [7], a cloud based computer cluster, which provides hard isolation from other users. We use a single Intel Xeon CPU E5-2660 v3 @ 2.60GHz, with twenty CPUs split evenly between two sockets. We pin each thread to single CPU on the same socket and set the priority of each thread to default so that they are all the same. We run each experiment for 64 seconds. We measure throughput - the number of operations each thread completes, the latency - total time spent by each thread within the critical section, and lock opportunity time - the total time that the lock is free and that thread is available to acquire it PLUS the time that the thread spends inside the critical section. This differs between different locks.

Figure 3.1 shows the critical section length and throughput of a single application with increasing ratios of insert to find threads. When running experiments with 100% find threads, throughput is high as the linked list is empty and so all operations are quick without any traversal of the list. When we have a single insert thread, we see a drop in the throughput of the find threads and as the number of insert threads increase, we see a continued decrease. The total critical section time for each thread within the application in a) when there are no insert threads is very similar. Images b), c) and d) show that find threads have a much higher total critical section time than the insert threads. This is because they must spend time traversing the growing linked list in order to find the entry. These graphs show a significant jump in the critical section time of the find threads as the insert threads increase, the only change here is the length of the linked list so we can conclude that this increase is directly linked. We see some contention between the insert threads and the find threads, in image b) and c) the insert threads spend very little (in comparison to the find threads) in their critical section.

The discrepancy in our expectation in this experiment is partly that the results are



(a) 1 application, ratio 0:100.

(b) 1 application, ratio 25:75.

(c) 1 application, ratio 50:50.

(d) 1 application, ratio 75:25.

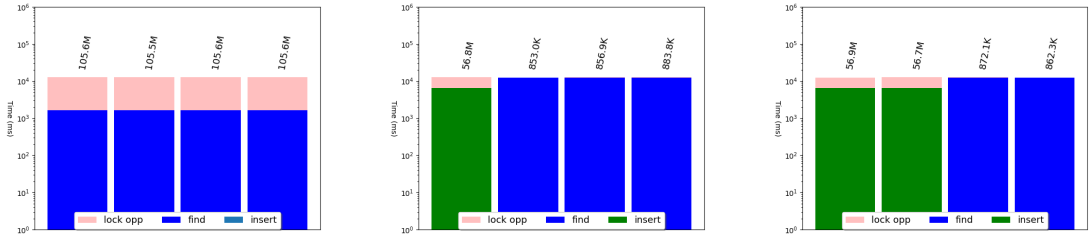
(e) 1 application, ratio 100:0.

Figure 3.1: ***The throughput and total critical section of a single application with increasing insert ratios***. *We label them in the following way: Ratio (Insert:Find). The columns measure the total critical section time and the values above the column refer to the throughput. We display the time and throughput in a log graph output to account for the wide difference in values between the two types of threads.*

skewed by lock contention. The spinlock is known to unfairly favour the thread that held it last, leading to one thread outperforming others. To remove this unfairness and improve our results we run these experiments with the scheduler cooperative lock (SCL). Figure 3.2 displays the results with the SCL instead. Here we again show the results of running a single application for reference. We now expect when running a single application that:

- The find threads have a higher total critical section time than the insert threads.
- The throughput of the insert threads is higher than that of the find threads despite a difference in critical section time.
- As the number of insert threads increase, we see a decrease in the throughput of the find threads.

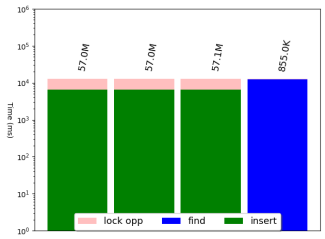
One point of note in figure 3.2 is that the insert threads are not using their full lock opportunity, this behaviour was unexpected and currently we are unsure of why this is, however it does not affect our simulation.



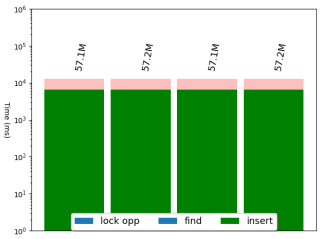
(a) 1 application, ratio 0:100.

(b) 1 application, ratio 25:75.

(c) 1 application, ratio 50:50.



(d) 1 application, ratio 75:25.



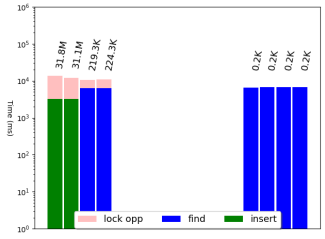
(e) 1 application, ratio 100:0.

Figure 3.2: ***Throughput and total critical section time when we change the global structure lock to a SCL.*** *A single application with increasing insert ratios. We label them in the following way: Ratio (Insert:Find). The columns measure the total critical section time and the values above the column refer to the throughput. We display the time and throughput in a log graph output to account for the wide different in values between the two types of threads.*

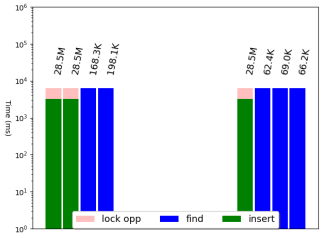
Figure 3.2(a) shows a higher and equal critical section time of all of the threads than in figure 3.1(a). As we increase the insert thread ratio, we notice that the find threads now fully use their allocated lock opportunity slice, the SCL does not allow the thread to go over this opportunity (as that would reduce the opportunity of another thread). This indicates that the slice is limiting the throughput of the threads in this duration. However by allowing this we are able to see the effect on the number of operations each thread performs over the same time period with no lock contention or unfairness so we have removed extenuating factors, and now we can clearly analyse the data and use it as the base case when multiple applications. As we change the ratio in favour of insert threads we notice a slight decrease in the throughput of the find threads. Now that we have an understand our base case, we can add another application that shares the linked list and monitor the changes.

2 Applications:Fixed Ratio Figure 3.3 displays running two applications on 8 CPUs. When running two applications we expect to see the following results:

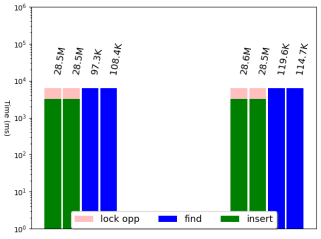
- Due to the workings of the scheduler cooperative lock, we expect to see a reduction in the total critical section time, as each thread is assigned a slice of opportunity and this is now divided between 8 threads, not just 4.
- When application 2 contains no insert threads, application 1 looks proportional to the base case figure 3.2 where it runs using the linked list alone.
- As we increase the number of insert threads in the second application, the throughput of the find threads in application 1 decreases.



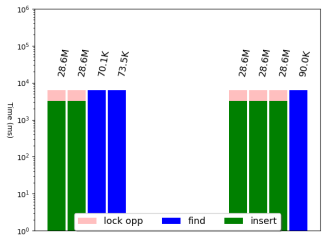
(a) 2 applications, app 2 ratio 0:100.



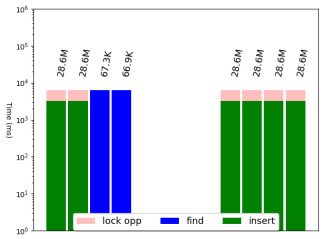
(b) 2 applications, app 2 ratio 25:75.



(c) 2 applications, app 2 ratio 50:50.



(d) 2 applications, app 2 ratio 75:25.



(e) 2 applications, app 2 ratio 0:100.

Figure 3.3: ***Throughput and total critical section time running two concurrent applications.*** *We keep application 1 the same with a ratio of 50:50 and increase the ratio of application 2. We label them in the following way: Ratio (Insert:Find). The columns measure the total critical section time and the values above the column refer to the throughput.*

- As we increase the number of insert threads in the second application, the throughput of the insert threads in application 1 is not affected. This is because the thread only interacts with the first entry in the data structure when it performs an insert.

As expected we see these points above which indicates interference between the two applications. Additionally the graphs also show:

- In 3.3(a) we see very little throughput in application two. This is due to the fact the as it has no entries in the linked list so it traverses the full length each find operation that it performed.
- The insert threads never use their full lock opportunity.
- The throughput of the insert threads is slightly reduced when insert threads are added to the second application, as seen between 3.3(a) and 3.3(b).

This example has shown a direct interference between these two applications and their critical section length. It also exhibits the correlation between the critical section length and the throughput.

Application 1			Application 2		
Ratio 1	Insert *throughput*	Find *throughput*	*Ratio 2*	Insert *throughput*	Find *throughput*
0:100	0	210M	0:100	0	211M
0:100	0	1K	100:0	120M	0
25:75	28M	264K	25:75	28M	425K
25:75	28M	153K	75:25	85M	166K
75:25	85M	149K	25:75	28M	158K

Table 3.1: ***Notable combinations of two application ratio showing total throughput and critical section time*** *separated by find and insert threads. We show the ratio and then the total throughput of each type of thread, and then the latency of each type of thread. Each application is four threads.*

2 Applications:All Ratios We ran an exhaustive set of experiments of all ratios with two applications, Table 3.1 shows the notable results that we want to comment on. By running all the different ratios we are able to simulate all possible scenarios, and conclude that unfairness is present. For ease of reading we only show significant

results, the table displays the ratio of each application and the throughput of the find threads combined and the insert threads. From this table we can see the extent of the unfairness as it applies to every experiment we perform. In no column do we see a result proportional to the base case i.e. no interference. In this table we are able to show the results of our edge cases, where the danger of unfairness can be seen.

- Row 1 shows an experiment with application 1 running only find threads while application 2 runs only find threads also. We see high throughput for both cases since there are no entries added to the linked list.
- In row 2, application 1 runs all find threads while application 2 runs all insert threads. This case shows a very low throughput for the find threads. This is because the find threads must traverse the whole linked list each time it performs an operation, this is the worst case behaviour. We see here the effect such behaviour has on the insert application also, it only completes half the operation it did previously, showing the interference between the applications.
- Row 3 shows running two applications with the same ratio. The throughput of the insert threads is very similar between applications but we see a difference in the find threads throughput, displaying clear unfairness between applications.
- Finally rows 4 and 5 show the ratios 75:25 and 25:75 run on each application. This shows that regardless of which application we run the ratios on we receive similar results.

3.3 Real World Examples

We now know there is unfairness within the data structures and the scenarios that can lead to this unfairness, however we don't yet know the potential damage that can occur in real systems through this unfairness. The following papers identify some actual scenarios where this unfairness is exploited.

3.3.1 Taming Adversarial synchronisation

Using Trātr to tame Adversarial synchronisation [20] describes some examples of unfair data structures used maliciously within the Linux kernel. The paper defines some vulnerabilities of the operating systems concurrent data structures notably an input parameter attack and a framing attack. An input parameter attack is 'when an attacker targets a synchronisation primitive and uses input parameters to increase the critical section size'. A framing attack is a type of synchronisation attack that targets weak complexity guarantees of a data structure. synchronisation attacks make the victims stall longer; framing attacks additionally make them spend more time in the critical section. A fair data structure would prevent such an attack by providing isolation and a solution to actively block such an attack within the data structure.

An example from this paper is the futex table [9][6] in the Linux kernel. A futex is a certain type of lock that when there is no contention can be handled in the user-space without having to enter the kernel. This is possible as the futex is simply a memory

address in the user-space e.g. a 32-bit lock variable field. If the lock has already been acquired and another thread attempts to acquire it then the lock is updated to note there is someone waiting for it and the sys_futex(FUTEX_WAIT) syscall is used. This triggers the kernel to add the waiting process to a wait queue for that lock.

Rather than have a wait queue for each futex variable, the kernel contains a futex table which is a hash table that hashes the futex variable address to a hash bucket that holds the wait queue. As multiple futex addresses may hash to the same bucket it stores a shared wait queue. Once the thread holding the futex is finished it checks the lock to see if there are threads waiting, if so it sends a syscall sys_futex(FUTEX_WAKE) to wake up waiting threads. This traverses the shared queue, and then releases the first thread waiting or all the threads waiting for this futex. Within the paper the authors successfully create an attack on the futex table where a malicious thread spawns a few thousand futex variables and then probes the hash buckets and identifies the busy wait queues based upon the time taken to complete the syscall. It then creates thousands of threads that wait upon that futex variable elongating these busy queues and affecting performance. This is the framing attack mentioned above.The attacker spends a small amount of effort adding entries and then sits quietly whilst the damage occurs.

The input parameter attack is also relevant to our thesis and is performed on data structures such as the inode cache. An inode stores metadata about a file such as its size and permissions. Also in the Linux kernel the inode cache stores inodes maintained by the Virtual File System to minimise expensive reads to the disk to for file metadata. This is stored as a hash table where the hash combines the unique inode number and the address of the file system super block. When an inode is created, it is added to the inode cache. An attack is possible which allows unprivileged users to arbitrarily create inodes and target a shared bucket within the hash table, elongating the list stored and thus critical sections.

3.3.2 Algorithmic Complexity Attacks

The paper ‘Surge Protector’ [1] gives a simple example of an attack in a shared data structure between clients. Pigasus is a hybrid FPGA+CPU, 100Gbps IDS, and it implements partial TCP reassembly to detect attacks that span across multiple packets in a TCP bytestream. This system uses a linked list to store incoming packets from an out of order system. A reassembly engine traverses its linked list when a packet arrives to sort it based upon its packet sequence number into the correct place in the list i.e. next to the packet that ends with that sequence number. When segments are in order they are released. As each packet reordering can take up the full size of the linked list n cycles to find the correct placement, if the system cannot keep up with the load of arriving packets it is forced to drop some.

A relevant attack on this system comes under the branch of Algorithmic Complexity Attacks [5]. These are attacks on the algorithms or data structures of a system to trigger their worst behaviour scenarios. Here, an attacker targets this effort of cycles per packet by maliciously sending packets that require a significant reorganisational effort. This causes the system to lag, so it drops some packets to compensate. In this way a malicious packet that takes perhaps five times as long to sort can cause the system to

drop five genuine packets. Such as attack manipulates the data structure by unfairly inserting in such a way that genuine users are denied service.

The above papers identify a gap in fairness within concurrent systems and show how this gap can be exploited. In this thesis we want to build on the work of the scheduler cooperative lock and the fair synchronisation algorithm and look at making more than just access to the data structure fair but look at extending this fairness to the structure itself, and in doing so provide a data structure that is immune to the attacks brought to light by the Trātr paper and more.

3.4 Gap in the Research

Relevant papers look at performance isolation in concurrent systems and the fairness of opportunity in these concurrent systems. One such paper is Wait-free synchronisation [12]. This paper defines a wait-free concurrent data object as 'one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes.' This is a very similar definition to performance isolation, as we can guarantee that the operation will only take at most some n steps regardless of the actions of other steps. The variable n however is dependent upon the number of threads and the consensus number of the data structure. The consensus number of a concurrent object is defined to be the maximum number of processes in the system which can reach consensus by the given object in a wait-free implementation. This means that the number of tasks that can assure that the shared object is correct control whether the implementation of the data structure will be wait free. Simple structures such as lists, queues and b-trees have a consensus number of 2, so with only up to two threads can we successfully implement a wait free structure. This paper goes on to prove it is impossible to construct wait-free implementation of some of the most basic data structures such as a set, queue, and list when run with a consensus of more than 2 processes. Thus, if we consider wait free to be the same as isolated performance this paper conclusively shows that we cannot design a perfect fair data structure, therefore we must strive for as close to isolated as possible.

A similar paper [8] also discusses lock-free linked lists and provides implementations that are on par performance wise with locks. Still their average performance is linear in the length of the linked list plus some contention. As we have discussed above, the length of the linked list is dependent on other threads, thus we cannot describe such implementations as fair or isolated. From this paper we can see that other implementations to improve such data structures look at the lock aspect, rather than the data structure.

Both papers Scheduler Cooperative Locks [19] and Fair Synchronisation [25] look at fair entry usage as fair access to the data structure. If each process is offered a fair opportunity to access the data structure, then it is fair. However, the SCL paper acknowledges that the length of the critical section introduces a different aspect to guaranteeing fairness. Consider the case of a linked list that has entries from two threads A and B. All the entries of A are at the head of the linked list and thread B's are at the end. For thread B to access any entries that it has inserted it must first traverse all of A's entries. If A has thousands of entries before the entries of B then thread B's

critical section is going to take significantly longer than that of A or another thread C that has perhaps added a single entry to the front. Performance is not isolated here as even if the access to the shared variable is as fair as can be, thread B is still affected by the entries that thread A has added within the list.

3.5 Summary

In this section we have given the motivation behind our work. We first give a simulated example displaying the unfairness within a data structure. We then move to some real life examples of malicious behaviour that target the unfairness shown above. Finally we discuss the gap in research into fair concurrent data structures, looking briefly at papers that highlight this.

Chapter 4

Design

4.1 Design Goals

1. **Controlled fairness within the data structure.** Our design should guarantee a level of performance isolation/fairness for each thread. In the general cases this is complete isolation where required, or in specific cases we look for no interference from malicious threads.
2. **A solution for ALL scenarios.** Our solution should be correct and fair for all use cases that may arise. It may adapt to fit these cases when necessary.
3. **No significant overhead and has a scalable performance.** When used in the place of a regular data structure our design should not be too complex to implement, with no major changes to increase overhead and as we increase the number of threads or users the performance it provides should scale.

4.2 Scenarios

Now that we move from motivation to designing the new data structure framework we go over all possible scenarios that could occur when using a concurrent linked list. This is to highlight any scenarios where unfairness with the structure can be seen. In table B in Appendix B we give a comprehensive list of scenarios including a description, if the scenario is fair, what would make the scenario fair, and how this can be achieved. Some scenarios are already fair. Others are the scenarios that we want our design to address. From our list of scenarios, we are able to pull out features of the data that shape the purpose of the data structure. We aim to develop a solution that can cover all scenarios with either small tweaks or on/off flags if a feature does or does not apply. We divide our scenarios and design into four quadrants and show that all scenarios fit into one of these four quadrants.

4.3 Data structure taxonomy

We taxonomise data structures to better understand how to address data structures as a resource and to determine best how to address unfairness. In some cases, sharing may be incidental, and partitioning the data structure is best, similar to per-thread variables. In others, sharing may be central to the use of the data structure, so other approaches are needed.

The first observation we make is that shared data structures differ on whether the *contents* of the structure are shared. For some structures, such as caches used by the file system, the structure elements represent global data (files in a global namespace) and are thus logically shared and cached data is accessible to all entities. For other structures, the shared data structure is used as a convenient mechanism to manage and group data that is only accessed by a single entity. For example the ESX page sharing hash table pages are typically private to a process and only shared if and when a process requires an identical physical page. We term data structures with shared content *Sharing data structures* (S), and those without shared content *Non-sharing data structures* (NS).

For sharing data structures, contents cannot be effectively partitioned by the entity, so a fair solution must look at a different method to ensure fairness such as managing the ability of an adversarial user to extend the structure. For non-sharing structures, it may be possible to partition data.

Our second observation is another two major classes of data structures. Several structures are designed as performance optimisations, where the contents are used to speed up important operations, but the presence of elements is not required for correctness. Caches fall in this category, such as the inode cache. Similarly, kernel same-page merging (ksm) maintains a tree with pages that are candidates for deduplication, but this tree is used only for performance optimization (sharing memory) and the presence or absence of a page in the tree does not affect correctness. We term these *Performance data structures* (P)

In contrast, for many kernel data structures, the contents must be present for correctness. For the futex table from the previously described attack in the motivation section, threads must be present on the waitlist to correctly implement synchronisation. We term these *Correctness data structures* (C).

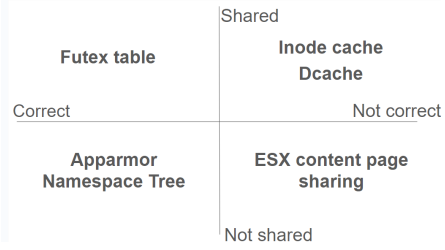


Figure 4.1: ***The three features placed in a quadrant.*** *This figure is to highlight how the three features we describe interact with each other. Applications of data structures can only be in one of the four quadrants.*

The distinction between these structures dictates how we can implement fairness in the

structure and we design our solutions based upon these four quadrants.

Taking both dimensions together, as shown in Figure 4.1, helps pick a strategy for addressing fairness on different types of data structures. For any performance structure (P), we have the flexibility that fairness may come from discarding elements from the structure to reduce the complexity of access. For correctness structures that do not provide sharing (NS+C), it is possible implement separate data structures or to partition the structure elements so that each entity accesses only the data it uses. For structures that support sharing and are necessary for correctness (S+C), it is not be possible to implement fairness by removing entries, so it is important to instead prevent unfairness through partitioning or some other method.

We believe that the $S + C$ quadrant is the target to address the isolation concerns. As multiple entities share the data, the presence or absence of an entry impacts the correctness of multiple containers/applications. Hence, one needs to prioritize isolating these global data structures.

For data structures in quadrant $NS + C$, when data is not shared among users (NS), the performance isolation concerns subside even though correctness is paramount. Often for such structures supporting something like a per-container/user structure will need more memory and managing individual structures can be difficult. Thus, we believe the simplicity of using the existing global data structures and the benefits of aggregating resources outweighs the security concerns that may arise. For data structures designed for performance purposes belonging to the other two quadrants - $S + P$ and $NS + P$, the same benefits of simplicity and aggregating the resources subside the security concern. The presence or absence of an entry is not going to impact correctness. Consider the example of the inode cache used to cache the inodes to prevent frequent access to the disk and support global namespace. Supporting separate caches need more memory and is difficult to manage as the containers need to access common files.

4.4 Non-shared solution

Algorithm 1 Psuedocode for a non-shared link list.

```
1:  Structure:
    Node:
        data
        next

    Thread Node:
        id
        next                                   ▷ points to the next Thread Node
        list                                   ▷ points to the head of the sub list

2:  Insertion:
3:  function INSERT(head, targetId, data)
4:      newNode ← createNode(data)
5:      curr ← head
6:      while curr ≠ NULL do                   ▷ search through the Thread Nodes
7:          if curr.id = targetId then
8:              newNode.next ← curr.list       ▷ add the Node to the head of the sub list
9:              curr.list ← newNode
10:             return
11:         end if
12:         curr ← curr.next
```

```
newIdNode ← createIdNode(targetId, newNode)          ▷ create a new Thread Node
newIdNode.next ← head                                ▷ insert it to the head of the list
head ← newIdNode


Find:
function FIND(head, targetId, target)
    curr ← head
    while curr ≠ NULL do                             ▷ search through the top list
        if curr.id = targetId then
            currNode ← list
            while currNode ≠ NULL do                 ▷ search through the sub list
                if currNode.data = target then
                    return currNode
                end if
                currNode ← currNode.next
            end while
        end if
        curr ← curr.next
    end while
    return head
end function=0
```

When the data is not shared between applications the solution is straightforward. We partition the data according to the application that inserts it, by creating sub-lists within the list. We create an additional type of node structure that we call the thread node from which the linked list is now made. Each of these thread nodes has a pointer variable to a node object which is the entry point to the sub-list of nodes that have been added by that thread. We show the psuedocode for this new thread node in Algorithm 1 which displays the changes to the functions as described in table 4.1 below to complete our design. The differences can be seen by comparing to the psuedocode for the linked list in 2. Here we have contained all our changes providing fairness to the data structure implementation. We show the design is fair as per the goals specified in section 1.1.

Name	Changes
Structure	Create an additional thread node structure containing an id variable pointer to the sub-list and a pointer to the next thread node
Insert	When a thread inserts it's first entry it passes in the key, value, and it's id. Initially the process traverses the linked list of thread nodes, looking for the thread id. As this is it's first entry it will not be found, so once the whole list has been checked and the find unsuccessful, it then creates a new thread node with that thread id, pointing to the new node entry and inserts the thread node at the front of the list. When a thread now inserts an entry again, when searching for the thread node by the thread id it will perform a successful find and insert the new entry at the head of the sub-list stemming from that thread node.
Find	To find an entry, we again first find the thread node for that thread id, and then perform the find as before on the sub-list
Delete	To delete an entry. We follow the steps of a find, however when finding the entry in the sub-list, we point the entry a head of it to the one after it and free the memory, deleting it.

Controlled fairness within the data structure We have provided controlled fairness through the isolation of the data per thread by separating into sub-lists. Now each

thread will only traverse the entries they have added without the additional critical section time spent moving through other threads entries. Each thread may add as many entries as they need without any effect on other threads when they access the linked list. This particular solution covers many scenarios of the above, and works for many data structures beyond our linked list. This solution effectively covers at least a half of the quadrant above, which hosts a wide range of the uses of shared data structures.

No significant overhead and has a scalable performance There is some overhead during the insert operation as it must first traverse and find the correct thread node before the insertion into the sub list. This overhead depends on the number of threads that are using the shared linked list, if there are many it may be a considerable overhead but remove this with a simple optimisation described below. An application will only have a sub list if it contains an entry so, in the worst case it is still the number of threads interacting which is less than or equal to the number of total entries so we have improved the best and worst cases with our fair data structure. Performance is improved as we no longer have to search through the entries of other threads, when the the number of these entries is large (potentially in the millions as shown in the real life example section). Removing their traversal time has a significant effect on the performance during the critical section.

optimisations and Limitations Further improvement for this design is made by using individual locks per sub-list. This will speed up performance as it will increase parallelism between the threads as the critical section spent holding the global lock is reduced, and the majority of the operation can be placed inside a different lock with no other contenders.

We can also counteract the potential overhead of a thread traversing many thread nodes of many threads to find its own which could potentially impede performance. To protect from this we can have a hash table that stores the thread id and a pointer to the address of the thread node, allowing for an average time to find the thread node of $\mathcal{O}(1)$. Cases with many threads and few entries are rare and there in these cases the optimum solution may be to have separate linked lists per thread.

4.5 Shared solution

When the data is shared between multiple applications we must consider that others may need to access the data added by another thread. This initially seems to contradict with our saying ‘bare the weight for what you add’ as now they may need to look through some ”weight” from other threads. How do we then apply our definition?

4.5.1 Shared – Correct

In the cases where the data cannot be separated by thread we need to think deeper about the solution to a fair linked list so we consider another approach utilising our base solution. With correct data structures we can find a common point that connects the data to be shared between a group of users i.e. when data is shared between this group of users the reason for this leads to us being able to isolate the groups from each other. For

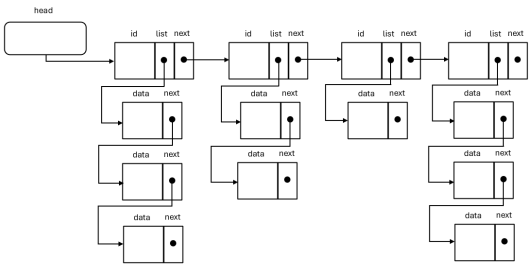


Figure 4.2: ***The set up of the non-shared solution.*** *In this diagram we can see the changes that would be made for a fair data structure. The head node shows the entry point, and the new node structure can be seen in the top row with the additional pointer. The sub lists then come from these new nodes, linked together as they were previously.*

example we consider again the futex hash table. The threads are placed in the hash table as they wait on futexes - our common point - and so within the wait queues they can be grouped by the futex that they wait upon. There cannot be duplicates (they cannot wait upon more than one futex at once) and the nodes here cannot be lost. The futex then is our common factor as we can group users by this for a very effective solution. This solution is successful for applications serving different purposes, some applications need to interact to progress and we can group these in such as way, whilst still allowing others to have isolation from them. If then a heavily used futex shares a wait queue with a lightly used futex, the release of threads waiting on the lightly used futex does not need to be impeded by the other waiting applications - thus still providing performance isolation.

Another example is a message queue. An incoming message can be for a specific user, or it can be for a group of processes, or it can be for anyone.User often wait for a message to come through before they can continue. In this case we can split the data structure into sub lists based upon the message. We can have sub lists for single users, each of the different groups, and then for everybody. Users can then search the sub lists that are relevant to themselves, thus they do not have to consider messages that are not meant for them. These users want to communicate with each other through the data structure and are not interested in the entries of those outwith their group so we manipulate this to the advantage of our solution. There cannot be performance isolation Where data is shared between these users as they must be able to access the entries of each other. However here the linked list is fair as the users or applications aren't adversely holding each other back, but providing a route forward by communicating through their entries. We again can split the data into sub lists by this common factor rather than by user. To implement this we can rename the thread node to be purpose specific, and where the parameter passed in is the thread id we change this also to the common factor between these users. The implementation of functions as described in table 4.1 do not have to change any functionality with the different names and parameters, displaying the ease and flexibility of our solution.

Controlled fairness within the data structure This implementation is fair for these shared data scenarios as it provides performance isolation between groups and applications. If the data is shared then the users 'agree' to take on the cost of other users in the group, there is no unexpected cost from anyone outwith the group.

No significant overhead and has a scalable performance As the implementation is the same as the non-shared solution we have already shown no significant overhead and scalable performance. As we add more groups to share the linked list we will not see an effect on the other groups (other than through lock contention). If we add more threads to our group then we will see longer critical sections to account for their entries, but still this performance is no worse than the original implementation of the linked list.

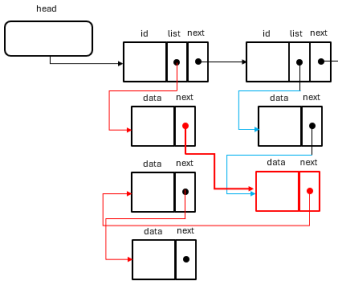
Optimisations and Limitations Our limitations with this design are that we must be able to group the data and the users in some way and this does not apply to all scenarios as we can see from those listed in B. If we end up with one single group which we must account for we will then simply have a linked list. Now that we have partitioned the data we can optimise with some simple additions to reduce the lock contention and improve both fairness and performance further. We can remove our global SCL lock and implement a SCL lock per sub-list. This change of position removes the contention of many threads for one lock to a wide dataset to a few threads on each lock to the relevant dataset. We then can use a simple spin lock on the global structure, so that when ThreadNodes are inserted or deleted there is no danger of de-sychronisation. The spin lock is only held until the SCL lock is reached so a spin lock is effective enough here as although it is not fair it will allow threads to reach the fair locks without too much overhead. Many threads will be waiting on the internal SCL lock at any one moment which will reduce the lock contention of the spin lock considerably.

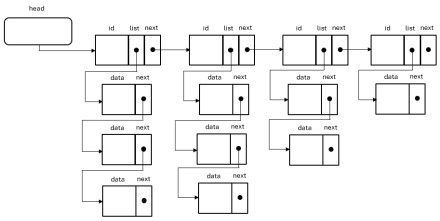
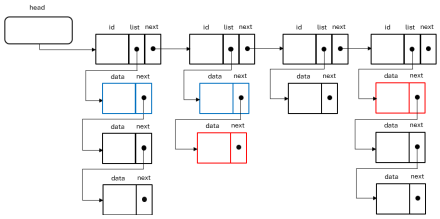
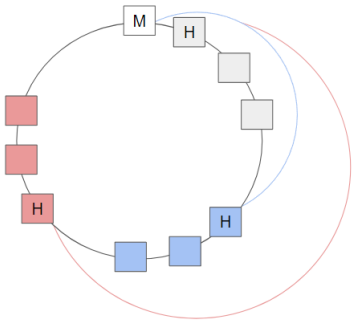
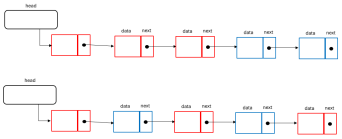
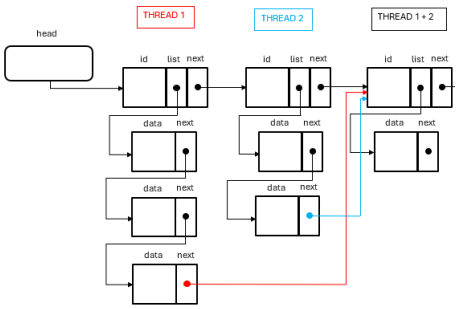
4.5.2 Shared – Performance

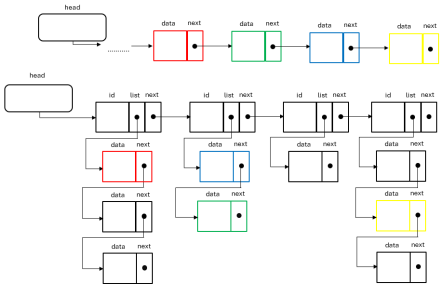
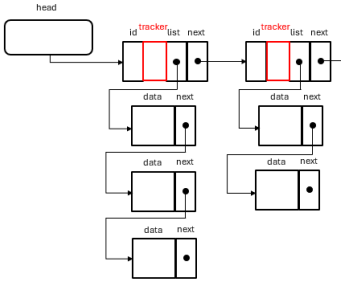
Due to time constraints we only list here some possible solutions that need some development before they can be implemented. When combining Shared and Performance we note:

- We can consider an unfair thread as an actor/user whose actions are negatively impacting the others to a significant extent. This may be malicious or it may not.
- The goal of any such data structure is to be fast i.e. to speed up some action that would have a significant overhead, whilst not adding too much time itself if the action much be completed (a cache) or to reduce the time is takes to search the whole data structure as a precaution such as a tree.
- We must consider here the greater fairness of entities. Thus, punishing threads that are being unfair to others could be acceptable.

Below we list some ideas for this section:

Idea	Potential	Limitations	Image
Allow pointers between sub lists.	Allow the sub lists to point at entries in other sub lists. We need to provide a way to point back at the rest of the other sub list through the shared entry.	We need a way to know where/if a previous entry is stored in the data structure. This solution could work for applications with high find rates.	

Sort the substructures by insertion rate, so that is they must traverse the whole data structure they can look through the shorter sub lists first.	This removes the harm from looking through long sub lists of other threads which are more likely to be malicious.	We may then penalise threads that need entries in genuine but long sub-list.	
Allow duplicates of entries in each sub list when using the solution described above.	If we allow duplicates of entries then each thread/user can be kept separate and do not need to look through the entries of others.	This takes up (slightly) more memory and we must ensure some way that values are kept synchronised.	
We make the structure circular, so that each application can search it's own entries first and then the entries of other applications.	This provides the opportunity for each user/thread to search it's own entries first, and then the entries in the rest of the structure.	As we still may search the rest of the structure we do not provide isolation from unnecessary entries/users.	
Rebalance the data structure when detecting unfairness.	Spread entries fairly within the list so that no thread/user spends more time searching than others.	This does not fit our definition of fairness, but if we have no way to achieve performance isolation or have as much isolation as can be given. Then we may have to look at fairness as equality.	
Create further sub lists for data shared between threads	If an entry is to be shared by a	This does not fit our definition of fairness, but if we have no way to achieve performance isolation or have as much isolation as can be given. Then we may have to look at fairness as equality.	

We can control the size of the data structure to reduce sizable interference, each of these data structures will have a mechanism for removing old/ obsolete data such as the least recently used(LRU) queue in cache tables or they will prohibit adding new entries until some have been deleted.	This solution has good potential within cache-like structures, if we can allow each thread to add a proportional amount of entries that is can control whilst other entries can look through these without being impeded by looking through an excessive amount of entries.	The main limitation is that we may penalise a genuine thread/user that needs to add many entries, if we cut them off or starve them then we may do some serious damage.	
+ We can combine the above with tracking the data structure usage of each thread or application. We may track the insertion rate, the number of entries they have added. We may want to look at insert to find ratio and penalise threads based upon these factors.	The adds to the above solution, and would allow us to be more exact and make informed decisions when setting limits for each thread or user. The overhead could be kept low for a solution such as this.	We still may penalise a genuine thread.	

4.6 Summary

In this chapter we outline our design for a fair data structure that acts as a resource. We begin by taxonimising data structures and determine that different solutions are required for different data structures. We then discuss the design of these solutions, beginning with the non-shared category where we design a solution that partitions the data based upon each entity. Our shared-correct solution follows this pattern or partitioning the data guarantee performance isolation and fair resource allocation. Finally we list viable solutions for the section of data structures and discuss their potentials and limitations.

Chapter 5

Evaluation

Here we evaluate the effectiveness of our solution for non-shared and correct data structures. Using the same micro-benchmarks as we used to show unfairness in the Motivation section we show scalable performance and controlled fairness within our linked list. We also show that the solution can apply to other data structures by implementing the solution with a hash table.

We use a synthetic workload to stress different aspects of the default linked list side by side with our solution. The workload consists of a multi-threaded program; each thread executes a loop and runs for a specified amount of time. Each loop iteration consists of two elements: time spent outside a shared lock, i.e., non-critical section, and time spent with the lock acquired, i.e., critical section. Unless explicitly specified, the priority of all the threads is the default thereby ensuring each thread gets an equal share of the CPU time according to the CFS default policy.

We use the following metrics to show our results: **(i) Throughput:** For synthetic workloads, throughput is the number of of operations completed (e.g., inserts or finds). This metric shows the bulk efficiency of the approach but is also used to show how isolated a thread or applications performance is. **(ii) Lock Hold Time:** This metric shows the time spent holding the lock, broken down per thread. This shows whether the lock is being shared fairly.

When we refer to performance here we move away from our definition in the taxonomy section back to the typical definition. We measure performance by throughput as through the fair lock we are confined to the time slice or opportunity to perform operations.

5.1 Fairness and Performance

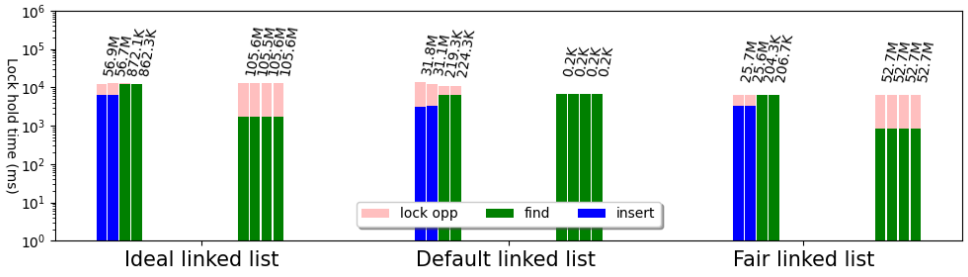
Intra-application To gauge the fairness and isolation of threads in our solution we run this synthetic workload with two applications. Each application has four threads that can either continually insert into the structure or continually find and the combination creates synthetic workloads that we use to run test. The desired result or ideal data structure is where our applications will show similar throughput, latency and lock usage to our base case in figure 3.2 in Motivation when we run a single experiment. The

ultimate goal is to achieve such isolation that there is absolutely no interference between applications. To show performance isolation we want to see that the application's threads are performing a similar amount of operations at each ratio of insert to find regardless of any other applications running.

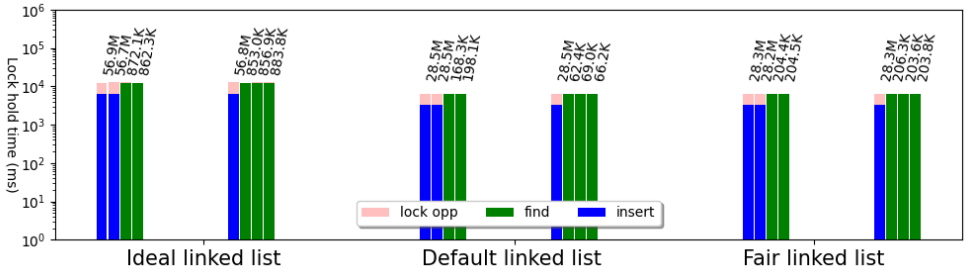
Figure 5.1 shows five graphs with increasing insert to find ratios. We plot our base case first on each graph, which shows the ideal results which come from running single applications no separate linked list with no interaction. Then we show the throughput and lock hold time of the default linked list with two applications of four threads each running. Finally we have the throughput and lock hold time of our solution with two applications.

Figure 5.1 a) b) c) d) and e) show the ideal result as a comparison for the default linked list and our solution. The ideal data structure here is the result of running two single applications, as each of two applications in the most fair case would have no contention with each other. We can see in all figures that the lock hold time of all applications is either using all or almost all of their lock hold opportunity, so time is split equally between all applications and threads. Throughput in the find and insert threads in applications in the default linked list is about half of the base case value, and in our solution although improved is still only slightly more than half. This is because of the contention on the global lock. We look at addressing this in the next section.

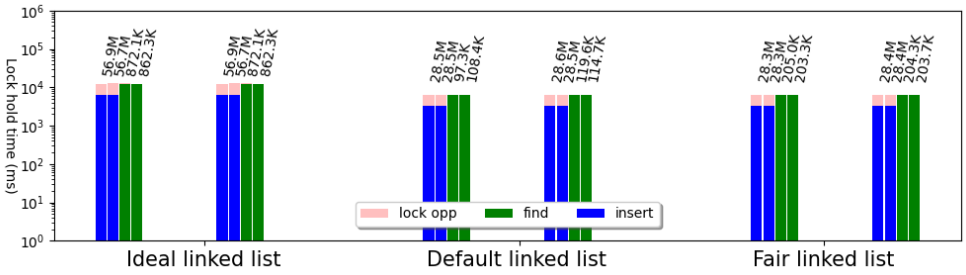
Figure 5.1 a) shows that when we run a 50:50 application next to an application full of find threads, the default linked list manages more inserts and finds than our solution because these find operations are very short as there are no entries in the sub list. We focus here on the results of a 50:50 application as the experiments of other ratios show similar results in fairness and performance. Figures 5.1 b) c) d) and e) show that when a 50:50 application is run beside an application with an increasingly higher insert:find



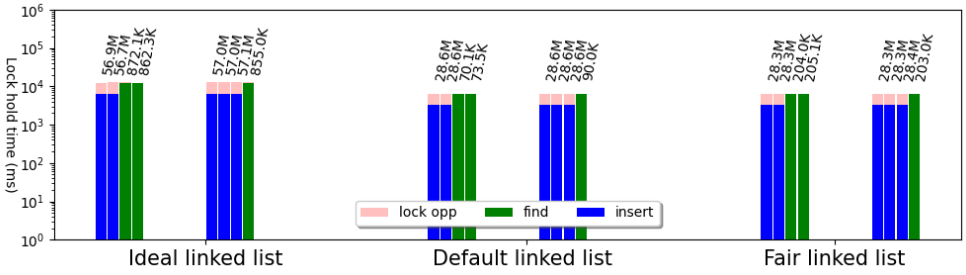
(a) 2 applications, app 2 ratio 0:100.



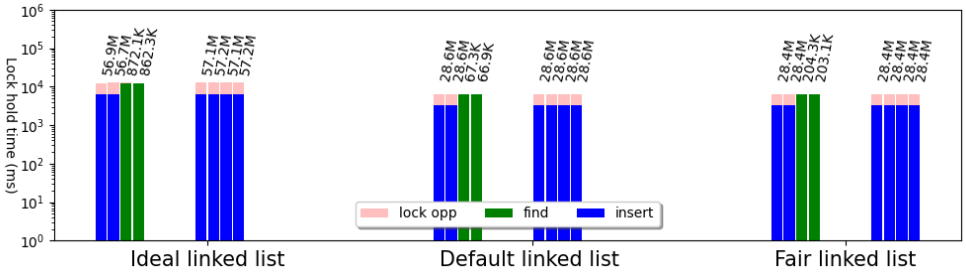
(b) 2 applications, app 2 ratio 25:75.



(c) 2 applications, app 2 ratio 50:50.



(d) 2 applications, app 2 ratio 75:25.



(e) 2 applications, app 2 ratio 0:100.

Figure 5.1: ***Throughput and total lock hold time running two concurrent applications.*** *We keep application 1 the same with a ratio of 50:50 and increase the ratio of application 2. We label them in the following way: Ratio (Insert:Find). The columns measure the total critical section time and the values above the column refer to the throughput.*

ratio, the default linked list shows a reduction in throughput in application 1 at every ratio increase of application 2. However in our solution as the insert ratio increases, the throughput in application 1 per thread stays the same. The lock hold time of each thread does not change in all solutions showing that our solution does not result large overheads that cut into the lock opportunity of each thread. In order for each thread to perform the maximum throughput it must utilise as much of it's lock opportunity as possible. Thus between threads within applications our solution shows controlled fairness whilst maintaining performance.

Inter-application Figure 5.2 considers the above experiment focusing on fairness between applications by the total find thread throughput of an application when run in parallel with another, both performing operations on the shared linked list. We can discard the insert throughput here as it is unaffected by the changing length of the linked list or sub lists and by our solution as seen in figure 5.1 we see no change in the

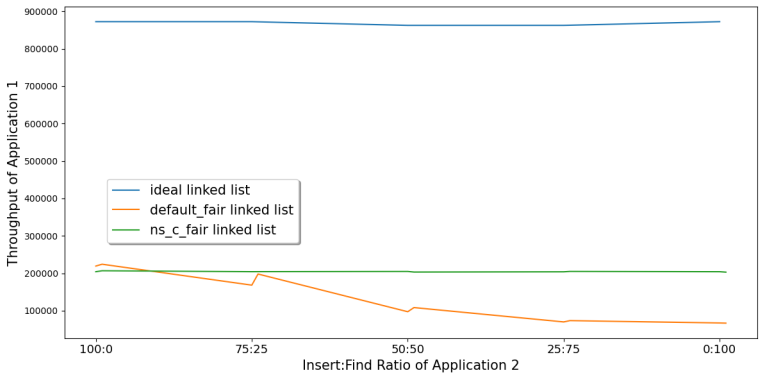
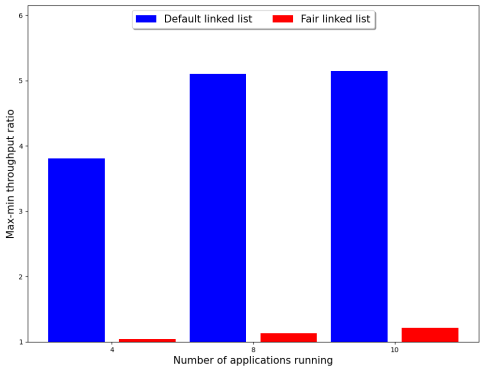


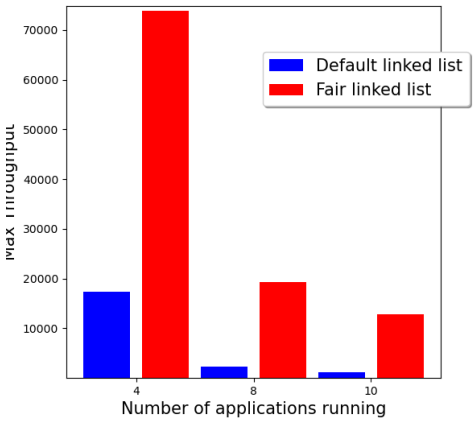
Figure 5.2: ***Throughput of find threads in linked lists.*** *We show the ideal case, the throughput of the default linked list, and then our solution. We we want to see here is a straight line to show no changes to the throughput regardless to changes to the x-axis.*

values. We plot the throughput of the 50:50 ratio application (Application 1) against the changing ratio of the other application (Application 2). We start by plotting the ideal fair linked list where throughput is high (as we have no lock contention) we see no change as the other application changes. The default linked list starts lower due to lock contention and shows a decrease in throughput as we increase the insert ratio of application 2, showing interference between the two applications, as more entries are added to the shared linked list. Our solution however is much fairer than this. We see changing the ratio of application 2 has no effect on the throughput of application 1 as the applications are now isolated from each other.

Figures 5.2 and 5.1 all show improved overall throughput combined of the two applications. Since each of the find threads in all applications at all ratios are performing more operations whilst running. The overall throughput and and performance is greater than in the applications running on the default linked list.



(a) ***Max-min throughput ratio of find threads in applications*** *On the x axis we have the number of applications running. On the y axis we have the ratio between the highest and lowest throughput of a find application over all ratios.*



(b) ***Max throughput of a find thread in an application*** *On the x axis we have the number of applications running. On the y axis we have the maximum throughput of a find thread within an application over all ratios.*

Figure 5.3

Scalable performance To show scalable performance we run experiments with more than two applications. We tighten our parameters (since running 10 applications with a possibility of 5 different ratios for each is too much data to analyse). We now consider two types of applications, find applications which have a ratio of 25:75 (insert:find) and high insert applications which have a ratio of 100:0. We choose the ratio of 25:75 for the find threads so that there are entries in the data structure for the application to find. The ratio of 100:0 for the insert threads is chosen as it is the worst case of behaviour that could occur. Figure 5.3 shows the impact on throughput to the find applications as we increase the number of high insert applications running concurrently. We run this experiment on 4, 8, and 10 applications. 10 is the maximum we are able to run, as each application has 4 threads, and we can only assign to 40 CPUs. The CPU architecture is made up of 20 CPUS but has the added functionality of hyper-threading, allowing two threads to be run on the same CPU concurrently.

Figure 5.3 a) shows max-min total find throughput of each experiment as we change the find:high-insert application ratio. For example with four applications we have four experiments, with eight applications we have eight experiments. From this figure we see that for the default linked list we have a large max-min ratio throughput indicating there is interference between applications as changes are made to the ratio of find applications to high insert applications. However for our solution this ratio is much smaller, never reaching more than 1.5 and indicating less interference between applications. Thus we have also shown achieved our fairness goal on a scalable level, and shown it is possible between threads in applications and between applications. Figure 5.3 b) also shows the max find throughput for each of the applications run over their experiments. For the default linked list we can see that the maximum throughput is not very high in comparison to our solution, and combined with the max-min ratio from b) we can see that the max and minimum performance of find applications on the default linked list is less than the results of our solution. Thus with solution we show an much improved performance.

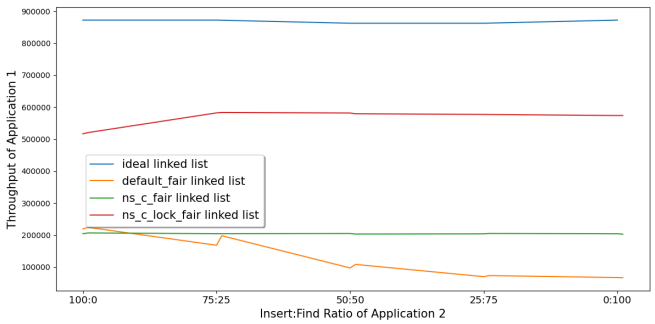
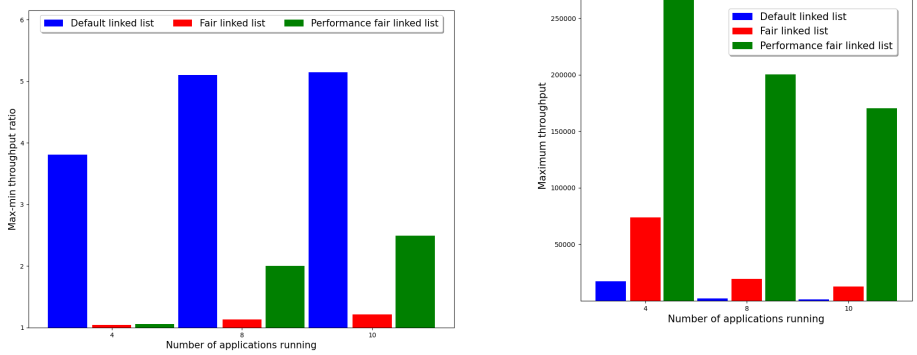


Figure 5.4: ***Throughput of find threads in linked lists.*** *We show the ideal case, the throughput of the default linked list, our solution, and then our solution with internal locks. What we want to see here is a straight line to show no changes to the throughput regardless to increase along the x-axis.*

5.2 Internal Locks

Design Now that we have our base fair solution the bottle neck once again becomes our fair lock so we implement internal locks as mentioned in the Design chapter.

Inter-application With this change we reevaluate the fairness of the solution. In figure 5.4 we add the throughput of the fair linked list with the internal SCL locks on each sub list to show the improvement to performance to the graph above. The ns_c__lock_fair solution with the internal locks clearly outperforms our solution and the default linked list. Moving the locks inside allows multiple operations to be completed on parts of the linked list at once so we can perform more than just the ideal case of twice a single application's performance. To evaluate fairness we can show the change to throughput of the ns_c__lock_fair solution as we change the ratio of application two. As we increase the ratio we see little to no change in the throughput of application 1 yet again, which again clearly indicates performance isolation between the applications.



(a) ***Max-min throughput difference of find threads in applications*** *On the x axis we have the number of applications running. On the y axis we have the difference between the highest and lowest throughput of a find application over all ratios.*

(b) ***Max throughput of a find thread in an application*** *On the x axis we have the number of applications running. On the y axis we have the maximum throughput of a find thread within an application over all ratios.*

Figure 5.5

Scalable Performance To evaluate the scalable performance of the internal lock solution we run the same test with multiple applications as for our original solutions. In figure 5.5 a) we show the max-min ratio when running 4, 8, and 10 applications as different find:high insert ratios. The performance solution with the internal locks shows some variation in throughput amount of the find threads however this is still less than is seen by the default linked list. For the performance of this solution over multiple applications can be seen in figure 5.5 b) against the performance of the base solution and the default linked list, we can see that the maximum throughput is significantly higher than either of the other linked lists.

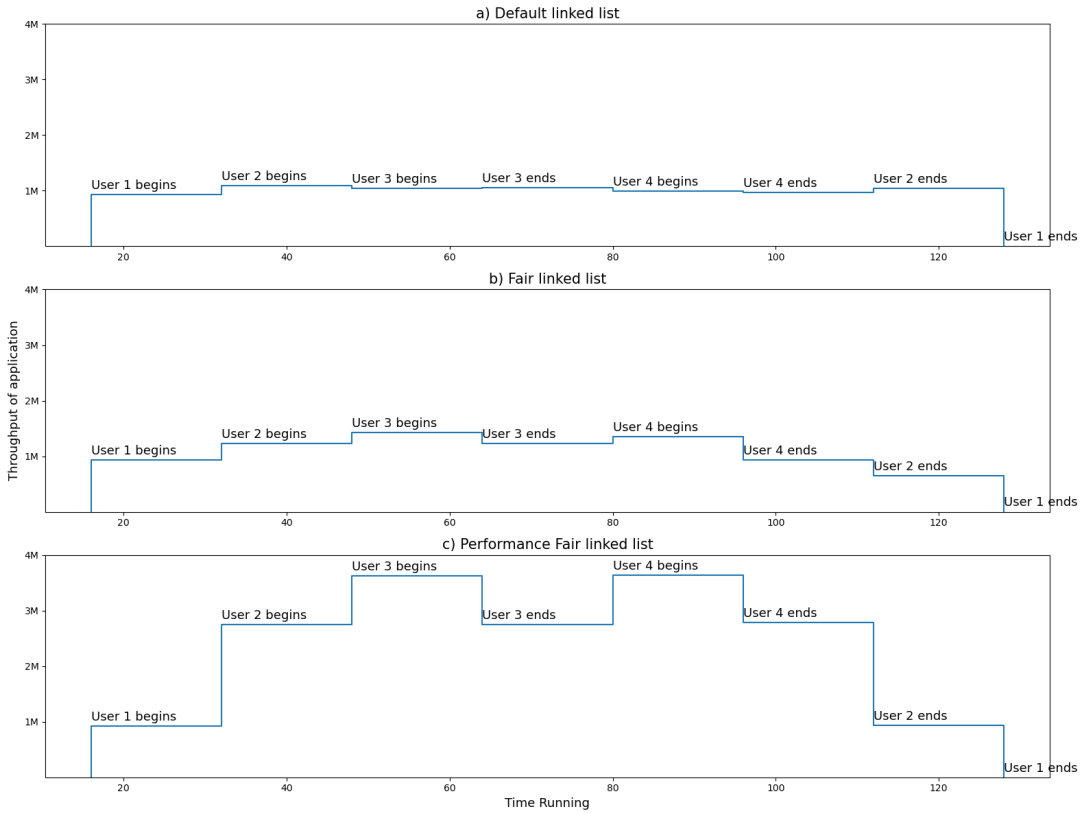


Figure 5.6: **Throughput of linked lists in a dynamic environment.** Initially running one application consisting of one find and one insert, after 16 second intervals we add or remove applications until we once again each a single running application.

5.3 Dynamic Scenario

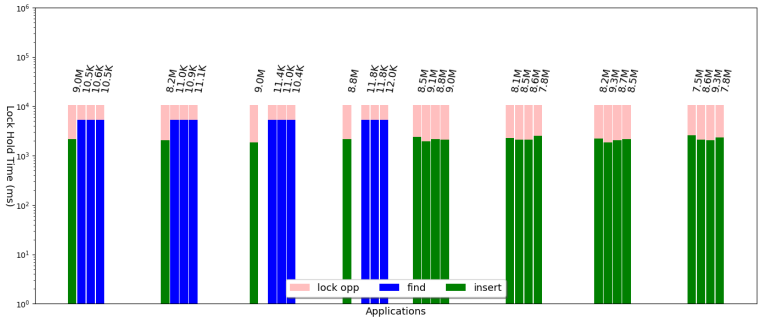
We also want evaluate the solutions in a dynamic environment to show they are adaptable to changes whilst maintaining fairness and performance. We simulate a shifting environment by growing and shrinking the number of applications accessing the shared structure. Our simulation builds up and removes threads over a period of time. Each application is identical with one find thread and one insert thread for simplicity.

In figure 5.6 we plot the throughput of these applications running in a dynamic environment. We plot for each of the default linked list, the fair linked list, and our fair linked list with improved performance. Within the default linked list we see that as we add applications we do not get an even increase in throughput. After the initial thread we do not note any significant change in throughput as we move through the simulation.

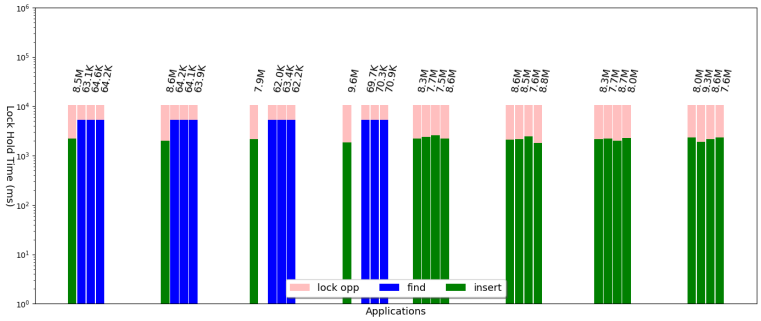
For our fair solutions as we add new applications that interact with the shared linked list, we see an equivalent increase in throughput, and as we remove applications an appropriate decrease.

The difference between our two solution is that once we add the internal locks we see a much higher total throughput, and that as we add and remove applications there is a much more equal step between the throughput as it increases. This displays the contention that we remove in the dynamic scenario from changing from a singular

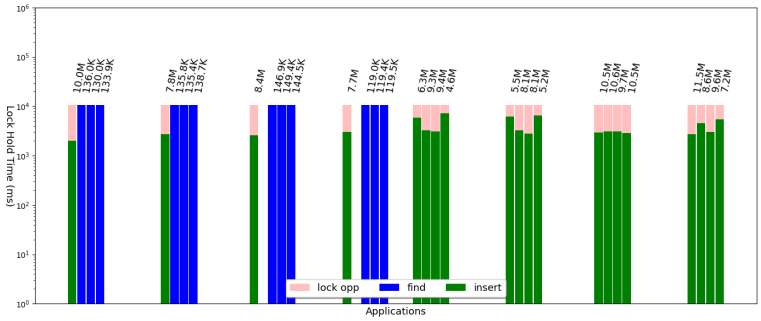
global lock to multiple internal locks.



(a) Default hash table



(b) Fair hash table



(c) Performance fair hash table

Figure 5.7: ***Throughput and total critical section time running eight concurrent applications.*** *We show eight applications running concurrently on each implementation, four are find applications (with a thread ratio of 25:75(insert:find) and the others are high inserts (100:0). A pair of each share a bucket and so interactions occur between these applications only. We show lock hold time through bar locks, with the lock opportunity of each thread also shown for context, with the throughput shown above each bar.*

5.4 Hash table

We also give a fair implementation of a hash table and show that through using our fair linked list as a base we can guarantee the same design goals for more complex data structures that build off of this linked list.

Design The hash table builds on the linked lists by implementing these fair linked lists as collision-prevention mechanisms within the hash buckets. We use a base design with

four buckets in the hash table to allow us to compare first the fairness and performance of the individual buckets and then to look at collisions within the buckets. We use the same experimental design as described above for the linked list, with applications continually running operations on shared data, this time within our hash table.

Inter-application To show inter-application fairness we now add collisions to the 4 buckets by running an experiment with eight applications. This leads to two applications accessing each bucket.

Collisions/Intra-application Figure 5.7 shows eight applications performing applications on a shared hash table. We hash two applications to each bucket, in the configuration of application 0 and 4, 1 and 5, and 2 and 6, 3 and 7. Per bucket we have one high-insert application (described above) and one find application. Each sub figure shows the lock hold time, with the lock opportunity shown behind it and throughput of each application shown above the graph. We see from these graphs that the lock hold time of the find threads in all hash table implementations is very nearly identical, although in our fair performance table we use all of the lock opportunity for the find threads. The lock hold times of the insert threads are also very similar, with some variation in our fair performance application (which likely some contention introduced from the spin lock). The throughput values also show performance improvement between these implementations as our fair hash tables perform significantly more operations than the default hash table.

Figure 5.8 shows the find throughput of an application accessing the hash table against the changing ratios of the other seven applications running. We choose to focus on a application with the same ratio in each experiment so that we can analyse its throughput as other the applications change. The default hash table shows throughput decreases when we reach four high insert threads. This is expected because buckets are shared by two threads, so we have shown that there is no interactions between threads in different buckets but that once they share a bucket, we can clearly see interference from the drop in throughput. For our fair solutions we do experience this drop in throughput, showing isolation between applications. For our ns_c_lock solution which uses internal locks per sub list we see a small amount in intra-application interference from the spin lock, however as shown the throughput is much better than the default hash table so we can accept this.

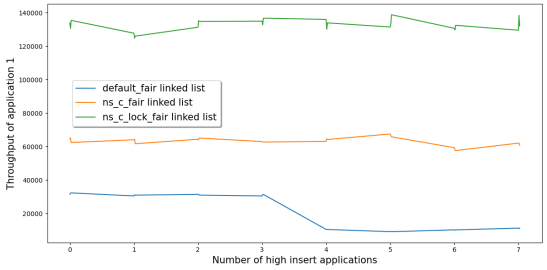
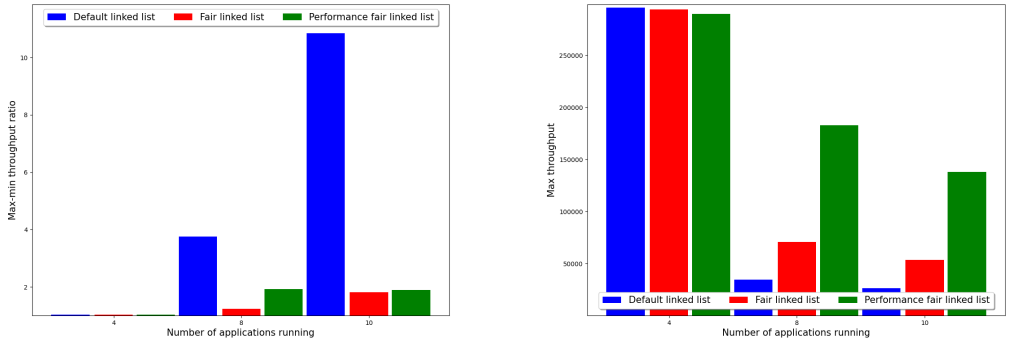


Figure 5.8: ***Throughput of find threads of each hash table.*** *We show the throughput of the default hash table, our solution, and then our solution with internal locks. What we want to see here is a straight line to show no changes to the throughput regardless to increase along the x-axis.*



(a) ***Max-min throughput ratio of find threads in applications*** *On the x axis we have the number of applications running. On the y axis we have the ratio between the highest and lowest throughput of a find application over all find:high insert ratios.*

(b) ***Max throughput of a find thread in an application*** *On the x axis we have the number of applications running. On the y axis we have the maximum throughput of a find thread within an application over all find:high insert ratios.*

Figure 5.9

Performance To look at the performance of our hash table over an increasing number of applications we plot the results from 4, 8, and 10 applications running in parallel. In figure 5.9 a), we see the max-min ratio of find throughput in the applications. We see that when running 4 applications as mentioned above, each application has it's own bucket which gives us very fair results. From figure 5.9 b) we see that the max throughput for each hash table is very high and with little variation so we can conclude all solutions provide excellent throughput here. When we move to 8 applications the results are more interesting, in a) we see that the default solution provides the highest variation in throughput over the experiments indicating variations in the results and interference between applications. When we reach 10 threads we see that this interference is very significant. When we also compare with (b) we can see that the performance for the default hash table decreases also as we add applications. For our solutions however we show a small amount of variation with the throughput of the applications over the different experiments. We see an expected reduction in performance but it is not as severe as the default hash table showing we have saved some performance with our isolated solutions.

5.5 Summary

In this section we show that our solution is fair as it guarantees performance isolation both between threads inter-application and intra-application. We show also they meet the design goal of scalable performance, outperforming the default linked list (and hash table). We perform further experiments with dynamic applications to show that our solution continues to guarantee performance isolation and allocate the linked list as a resource between joining applications. Finally we show the solution is adaptable to more complex data structures though running tests on a fair hash table that builds on our linked lists.

Chapter 6

Conclusions and Future Work

6.1 Contributions

We summarise our contributions from this thesis below:

- We identified a gap in research into fairness of data structures in concurrent systems. Research has been undertaken into the fair access of such shared structures but we take this further by considering fairness within these structures.
- We have defined a fair data structure in time as a guarantee of performance isolation between entities that interact with the structure. We have shown unfairness exists in the basic implementation of data structures by showing it can be seen in the simple structure of a linked list through interference with the operations of threads and applications that interact with the linked list.
- We have shown the dangers of this unfairness and highlighted how it could be maliciously exploited to attack [19][5] an operating system or shared cloud system. Vital shared structures such as the futex table are at risk of exploitation that takes advantage of this unfairness to extend the critical section of users and stunt their progress. Algorithmic Complexity Attacks are defined in [5] which covers attacks that exploit this unfairness.
- We develop a taxonomy of data structures in which every data structure has a place under the classifications of Shared, Correct, and Performance. By creating such classifications we can group data structures and consider their purpose as a resource within these. This allows us to now discuss such structures as resources and answer our question.
- We have developed a system for solving this unfairness that encompasses all scenarios based upon the use cases and taxonomy of the data structures.
- We give a solution for the non-shared and shared-correct scenarios implemented on a linked list. This solution importantly meets all the goals we set out at the beginning of the design section. The solution - partitioning the linked lists into sub lists based upon a shared feature or variable such as a futex variable or id -

provides fairness through isolation of entities that do not need to interact whilst maintaining the same or better overall performance.

- We have shown that this solution is scalable to many concurrently running applications, showing even with high contention and potential interactions between many applications our solution conserves both its fairness and performance qualities.
- We have shown the solution is adaptable to dynamic workloads, allowing it to be applicable in any environment where a shared data structure may be required. We show fairness and isolation is instantaneous in these variable scenarios.
- We have shown that this solution is adaptable to other complex data structures such as a hash table. Through implementing a fair hash table that builds on our linked list solution we show that it exhibits the same features of controlled fairness and scalable performance intra-application (between threads) and inter-application.

6.2 Future Work

Implementation of fair kernel data structures The next step following the evaluation of our Shared Correct solution is to implement it in the place of such data structures in the kernel. Focusing on the futex table here would show the real life application and potential of the fair data structure in defending against the framing attack described in [19] and [5] and other algorithmic complexity attacks.

Final quadrant In the future a solution should be developed for the final quarter of the quadrant - Shared/Performance. This section will need a different approach to the correct design as described in the data taxonomy of the design section, and must ensure the performance factor is as good as, if not better, (which we have shown is a really possibility) than the default structures used. There are several viable solutions listed in our design section with a brief initial evaluation into their design, potential and feasibility that was discussed. More development of these ideas is required as whilst it initially seems simple, the solution is more complex than originally thought. A development or combination of these would lead to the final nail in the coffin for unfairness in data structures.

Fairness over space We have shown conclusively in this thesis that we can achieve fairness in data structures over time. This is just one dimension of concurrent systems, and some consideration should be given to fairness over space also where this is particularly relevant to the system. Our partitioning solution for example does not necessarily ensure fairness over space if the unfairness is being injected by a malicious thread which voids entries. Fairness over space would require a different definition for fairness that was not related to performance isolation, and such solutions would need to consider this change to provide fairness over space. Solutions would focus more heavily on preventing unfairness in the size of the data structure rather than adapting to it or spreading it evenly.

Expansion to complex data structures Developing and sharing implementations for more complex data structures that are commonly shared in concurrent system is an important task for the future. Structures such as the binary tree used for searching and

manipulating directories or database indexing, n-ary trees, or graphs are very important for big data manipulation and machine learning. We believe these structures have the potential to play an crucial part in the future of shared systems and envision a future where they are easily implemented or inserted into systems in the place of their default predecessors.

Bibliography

[1] N. Atre, H. Sadok, E. Chiang, W. Wang, and J. Sherry, "Surgeprotector," *SIGCOMM '22: Proceedings of the ACM SIGCOMM 2022 Conference*, Aug. 22, 2022. DOI: 10.1145/3544216.3544250. [Online]. Available: https://doi.org/10.1145/3544216.3544250.

[2] S. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, Jun. 1, 1996. DOI: 10.1007/bf01940883. [Online]. Available: https://doi.org/10.1007/bf01940883.

[3] J. Bouron J., S. Chevalley, B. Lepers, *et al.*, "The battle of the schedulers: Freebsd ule vs. linux cfs," Jul. 11, 2018. [Online]. Available: https://www.usenix.org/system/files/conference/atc18/atc18-bouron.pdf.

[4] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, Jun. 1, 1971. DOI: 10.1145/356586.356588. [Online]. Available: https://doi.org/10.1145/356586.356588.

[5] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C.: USENIX Association, Aug. 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks.

[6] U. Drepper and Red Hat, Inc., "Futexes are tricky," Nov. 5, 2011. [Online]. Available: https://www.akkadia.org/drepper/futex.pdf.

[7] D. Duplyakin, R. Ricci, A. Maricq, *et al.*, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19.

[8] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pp. 50–58, Jul. 25, 2004. DOI: 10.1145/1011767.1011776. [Online]. Available: https://doi.org/10.1145/1011767.1011776.

[9] H. Fuss, IBM Thomas J. Watson Research Center, R. Russell, IBM Linux Technology Center, M. Kirkwood, and matthew@hairy.beasts.org, "Fuss, futexes and furwocks: Fast userlevel locking in linux," *Ottawa Linux Symposium*, pp. 480–481, 2002. [Online]. Available: https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

[10] A. Gupta, A. Tucker, and S. Urushibara, "The impact of operating system scheduling policies and synchronization methods of performance of parallel

applications," *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, Apr. 2, 1991. DOI: 10.1145/107971.107985. [Online]. Available: https://doi.org/10.1145/107971.107985.

[11] S. Haldar and D. K. Subramanian, "Fairness in processor scheduling in time sharing systems," *Operating Systems Review*, vol. 25, no. 1, pp. 4–18, Jan. 2, 1991. DOI: 10.1145/122140.122141. [Online]. Available: https://doi.org/10.1145/122140.122141.

[12] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, Jan. 1, 1991. DOI: 10.1145/114005.102808. [Online]. Available: https://doi.org/10.1145/114005.102808.

[13] N. Huber, M. Von Quast, M. Hauck, and S. Kounev, "Evaluating and modeling virtualization performance overhead for cloud environments," *Proceedings of the 1st International Conference on Cloud Computing and Services Science*, Jan. 1, 2011. DOI: 10.5220/0003388905630573. [Online]. Available: https://doi.org/10.5220/0003388905630573.

[14] H. P. Katseff, "A new solution to the critical section problem," *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, Jan. 1, 1978. DOI: 10.1145/800133.804335. [Online]. Available: https://doi.org/10.1145/800133.804335.

[15] R. Krebs, C. Momm, and S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments," *Science of Computer Programming*, vol. 90, pp. 116–134, Sep. 1, 2014. DOI: 10.1016/j.scico.2013.08.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642313001962.

[16] L. Lamport, *The mutual exclusion problem: part II—Statement and solutions*. Oct. 9, 2019. DOI: 10.1145/3335772.3335938. [Online]. Available: https://doi.org/10.1145/3335772.3335938.

[17] P. Moorhead, "Intel's newest core processors: All about graphics and low power," Jun. 4, 2013. [Online]. Available: https://www.forbes.com/sites/patrickmoorhead/2013/06/04/intels-newest-core-processors-all-about-graphics-and-low-power/.

[18] "Operating systems: Three easy pieces." (), [Online]. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/.

[19] Y. Patel, L. Yang, L. Arulraj, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Avoiding scheduler subversion using scheduler-cooperative locks," *Fifteenth European Conference on Computer Systems (EuroSys '20)*, Apr. 15, 2020. DOI: 10.1145/3342195.3387521. [Online]. Available: https://doi.org/10.1145/3342195.3387521.

[20] Y. Patel, C. Ye, A. Sinha, *et al.* "Using Trātṛ to tame adversarial synchronization." (2022), [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/patel.

[21] "Pthread$_{s}pin_{l}ock(3) - linuxmanualpage$." (), [Online]. Available: https://man7.org/linux/man-pages/man3/pthread_spin_lock.3.html.

[22] $pthread_spin_lock(3) - Linux manual page$. [Online]. Available: https://man7.org/linux/man-pages/man3/pthread_spin_lock.3.html.

[23] L. Sha Jr., R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," 9, Sep. 1990, pp. 1175–1176. [Online]. Available: https://www3.nd.edu/~dwang5/courses/spring17/papers/real-time/pip.pdf.

[24] G. Taubenfeld, *Contention-Sensitive data structures and Algorithms*. Jan. 1, 2009, pp. 157–171. DOI: 10.1007/978-3-642-04355-0_17. [Online]. Available: https://doi.org/10.1007/978-3-642-04355-0_17.

[25] G. Taubenfeld, "Fair synchronization," *Journal of Parallel and Distributed Computing*, vol. 97, pp. 1–10, Nov. 1, 2016. DOI: 10.1016/j.jpdc.2016.06.007. [Online]. Available: https://doi.org/10.1016/j.jpdc.2016.06.007.

[26] B. Tekinerdoğan and A. Oral, *Performance isolation in Cloud-Based big data architectures*. Jan. 1, 2017, pp. 127–145. DOI: 10.1016/b978-0-12-805467-3.00008-9. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/B9780128054673000089#.

[27] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation," *Sigplan Notices*, vol. 33, no. 11, pp. 181–192, Oct. 1, 1998. DOI: 10.1145/291006.291044. [Online]. Available: https://doi.org/10.1145/291006.291044.

Appendix A

Important Algorithms

A.1 Pthread spinlock

Algorithm 2 Spinlock

```
1:  Variables:
2:  lock                                   ▷ Boolean variable indicating if the lock is acquired

3:  Initialization:
4:  lock ← false                           ▷ Initialize lock as unlocked

5:  Acquire Lock:
6:  function ACQUIRELOCK
7:      while lock = true do               ▷ Spin until lock is acquired
8:          do nothing
9:      end while
10:     lock ← true                        ▷ Lock acquired
11: end function

12: Release Lock:
13: function RELEASELOCK
14:     lock ← false                       ▷ Unlock the lock
15: end function
```

A.2 Scheduler Cooperative Lock

Algorithm 3 Scheduler Cooperative Lock

```
1:  procedure FAIRLOCK_ACQUIRE
2:      if current thread is the owner of this lock slice then
3:          if no thread waiting on lock then
4:              atomically mark lock as acquired
5:          else
6:              atomically change next thread from READY_TO_ACQUIRE to NEXT_TO_ACQUIRE
7:          end if
8:          if atomic operation succeeds then
9:              record time and return
10:         end if
11:     end if
12:     if current is banned then
13:         wait until ban expires
14:     end if
15:     while True do
16:         get the last thread waiting on lock and atomically swap it with current thread
17:         retry if atomic operation fails
18:         if no thread waiting on lock then
19:             set current thread as ACQUIRED
20:             set next thread of lock as current thread
21:         else if next thread is holding the lock then
22:             set current thread as NEXT_TO_ACQUIRE
23:             set next thread of previous thread as current thread
24:         else
25:             set next thread of previous thread as current thread
26:             wait for notification from previous thread
27:         end if
28:         wait until the current slice expires
29:         wait until state becomes READY_TO_ACQUIRE and atomically swap it with ACQUIRED
30:         if current thread has next thread waiting behind then
31:             set next thread of lock as next thread of current thread
32:             set next thread as NEXT_TO_ACQUIRE and notify
33:         else
34:             atomically mark lock as acquired
35:         end if
36:         set new lock slice and claims ownership
37:         record time and return
38:     end while
39: end procedure
40: procedure FAIRLOCK_RELEASE
41:     if no thread waiting on lock then
42:         atomically mark lock as free
43:     else
44:         set next thread of lock as READY_TO_ACQUIRE
45:     end if
46:     measure critical section time and accumulate usage metric
47:     if overuse lock time proportional to thread weight then
48:         mark current thread as banned
49:         relinquish lock slice
50:     end if
51: end procedure
```

A.3 Linked list

Algorithm 4 Linked List Operations

```
1: Structure:
   Node:
       data
       next

2: Insertion:
3: function INSERT(head, data)
4:     newNode ← createNode(data)
5:     newNode.next ← head
6:     head ← newNode
7: end function

8: Find:
9: function FIND(head, target)
10:    if head = NULL then
11:        return head
12:    end if
13:    if head.data = target then
14:        return head
15:    end if
16:    curr ← head.next
17:    while curr ≠ NULL do
18:        if curr.data = target then
19:            return curr
20:        end if
21:        curr ← curr.next
22:    end while
23:    return head
24: end function
```

A.4 Hash table

Algorithm 5 Hashtable Operations

```
Structure:
Hashtable:
    array[size]

function HASHFUNCTION(key, size)
    return key mod size
end function

function INSERT(table, key, value)
    index ← HASHFUNCTION(key, sizeof(table.array))
    node ← new Node with key and value
    node.next ← table.array[index]
    table.array[index] ← node
end function

function FIND(table, key)
    index ← HASHFUNCTION(key, sizeof(table.array))
    current ← table.array[index]
    while current ≠ NULL do
        if current.key = key then
            return current.value
        end if
        current ← current.next
    end while
    return NULL
end function
```

Appendix B

Table of Scenarios

Scenario	Fair?	Why	Desired Behaviour
Two threads are inserting into a shared data structure			
2 threads are adding items to a linked-list at an even rate.	No	The rate of insertion is even with a fair lock there is not much contention but still some interference.	No contention.
Threads are taking turns on the same CPU.	No	One thread may hold the lock and starve the other thread when it is running on the CPU.	We can use a Scheduler Cooperative Lock.
Thread A adds entries at x times the rate of Thread B.	No	The thread with the lower rate may hold the lock for a disproportionate time.	We can use a Scheduler Cooperative Lock.
Thread A changes its insertion rate at some point in time.	No	The lock opportunity time may not change to reflect this.	We can use a Scheduler Cooperative Lock.
Thread A makes inserts until a point of time and then stops, thread B continues.	No	The lock opportunity time may not change to reflect this.	We can use a Scheduler Cooperative Lock.
There are entries already in the list and two threads want to insert more.			
Thread A has x entries in the list and thread B now wants to add an entry.	-	Depending on the data structure and the insert functionality it may add to the front of the struct, or need to traverse some entries.	Thread B does not interact with the entries of thread A.
Thread A has x entries in the list and thread B now wants to add y entries.	-	Depends upon the functionality of the insert.	Thread B does not interact with the entries of thread A.

Thread A has x entries in the list and thread B and A now wants to add y, z entries.	-	Depends upon the functionality of the insert.	The threads do not interact with the entries of the other thread.
Each add x entries (“fairly”) then Thread A does a lookup.	No	Thread A must traverse some or all entries of thread B.	Thread A does not interact with the entries of thread B.
Both threads have inserted entries and now want to lookup data.			
Thread A has inserted x more entries before Thread B goes in to do a find().	No	Thread A must traverse some or all entries of thread B.	Thread A does not interact with the entries of thread B.
Thread B has one entry that it will lookup every access while Thread A is still adding entries.	No	Thread B must traverse some or all entries of thread A, as the list grows it may get longer, increasing the find time.	Thread B does not interact with the entries of thread A.
Threads perform finds at an equal rate on shared entries.	Yes	The entries are shared by both threads so access to all entries is needed.	-
Threads perform finds at an equal rate on only the entries they have added.	No	The threads must traverse entries added by the other thread.	There is no interaction with the entries they have not added.
Threads perform finds at an unequal rate on only the entries they have added. One thread performs x more finds th an the other.	No	The thread performing more finds may not have a proportional opportunity for this, and within the structure it has to interact with the entries of the other thread.	

Threads perform finds at an unequal rate on only the same entries. One thread performs x more finds than the other.	No	The thread performing more finds may not have a proportional opportunity for this.	
A thread only performs finds on data added by other threads, it performs no insertions.	Yes	The entries are shared by both threads so access to all entries is needed.	-
External factors have an effect on the threads.			
One thread has a higher priority than the other thread and so should have a proportional access to the data.	No	A simple lock will have no regard for the priority of the thread.	We can use an Scheduler Cooperative Lock.
Threads must find an entry within an within a certain time. They must provide some performance guarantees.	No	Another thread's entries are the cause of not providing these guarantees.	
A thread wants to delete entries from the structure.			
A thread wants to delete all entries that were added by this thread.	No	They must traverse the entries of other threads in order to find all of their own entries.	There is no interaction with the entries they have not added.
We want to flush the data structure of all entries.	No	One thread here is responsible for all the deletions	The threads share this responsibility.
External factors have an effect on the data structure.			
We must limit the size of the data structure due to memory constraints.	Yes	Some threads will be prevented from inserting more entries or a section on entries will be deleted.	

The primary purpose is storing data, and the structure is expected to be large.	Yes	-	-
Entries must be unique.	-	-	-
Entries must be sorted.	-	-	-
The structure needs to be re-balanced.	-	-	-
The structure uses some probability within . it's find function.	-	-	-

Table B.1: Scenarios of uses cases for a data structure.