ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Performance and Scalability Analysis of Concurrent Data Structures

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χαράλαμπου Ε. Στυλιανόπουλου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Performance and Scalability Analysis of Concurrent Data Structures

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χαράλαμπου Ε. Στυλιανόπουλου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Νοεμβρίου 2014.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Λέκτορας Ε.Μ.Π.	Νικόλαος Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2014.

.....

Χαράλαμπος Ε. Στυλιανόπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στις μέρες μας, οι πολυπύρηνοι επεξεργαστές χρησιμοποιούνται ευρέως και έχουν εισαχθεί σε πολλά προγραμματιστικά περιβάλλοντα. Ο παράλληλος προγραμματισμός δεν αφορά πλέον μόνο επιστημονικές εφαρμογές για υπερυπολογιστικά συστήματα, αλλά καλύπτει ένα μεγάλο φάσμα εφαρμογών, που περιλαμβάνει και εφαρμογές καθημερινής χρήσης σε desktops ή ενσωματωμένα συστήματα.

Ένα σημαντικό και καθοριστικό κομμάτι για την επίδοση κάθε εφαρμογής είναι οι δομές δεδομένων που χρησιμοποιεί. Η μετάβαση από αρχιτεκτονικές ενός πυρήνα σε πολυπύρηνες αρχιτεκτονικές, σηματοδοτεί την ανάγκη εκσυγχρονισμού και παραλληλοποίησης των βασικών δομών δεδομένων, ώστε να ακολουθούν τις τάσεις του μέλλοντος και να προσφέρουν υψηλή κλιμακωσιμότητα.

Η διπλωματική αυτή αφορά τις δομές δεδομένων, με ιδιαίτερη έμφαση στις ουρές και τους πίνακες κατακερματισμού, και μελετά διάφορους τρόπους παραλληλοποίησης τους με βάση τα προβλήματα που καλούνται να επιλύσουν, τα ιδιαίτερα χαρακτηριστικά τους και την συμπεριφορά τους με βάση το υλικό.

Λέξεις κλειδιά: παράλληλες δομές δεδομένων, ταυτόχρονη πρόσβαση, αμοιβαίος αποκλεισμός, ατομικές εντολές, transactional memory, FIFO ουρές, πίνακες κατακερματισμού, κλιμακωσιμότητα, επίδοση.

Abstract

Nowadays, multicore processors have become mainstream and are being used by many programming environments. Parallel programming is no longer about scientific applications run in supercomputers, but covers a wider range of environments, including applications on desktops and embedded systems.

An important and crucial factor of every application is the set of data structures it is build on. The transition form single core to multicore systems marks the need to refactor and parallelize basic data structures in order to support higher scalability.

In this thesis we study concurrent data structures, particularly focusing on FIFO queues and hash tables, with respect to the way the are synchronized, the problems they are dealing with, their special characteristics and their effects on hardware.

Keywords: concurrent Data Structures, concurrent access, mutual exclusion, atomic primitives, transactional memory, FIFO queues, hash tables, scalability, performance.

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Καθηγητή Νεκτάριου Κοζύρη.

Θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Κοζύρη για την ευκαιρία να δουλέψω στο εργαστήριο και την έμπνευση που μου έδωσε μέσα από τις διαλέξεις του. Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Λέκτορα Γιώργο Γκούμα για την επίβλεψη της δουλειάς μου και τις πολύτιμες συμβουλές του. Θερμά ευχαριστώ αξίζουν και στον Υποψήφιο Διδάκτωρ Δημήτρη Σιακαβάρα για την συνεχή καθοδήγησή του και το χρόνο που αφιέρωσε. Χωρίς τη συμβολή του η ολοκλήρωση αυτής της εργασίας δε θα ήταν εφικτή.

Θα ήθελα επίσης να ευχαριστήσω τους φίλους μου και τους συμφοιτητές μου για τα όμορφα χρόνια των σπουδών μου και τη βοήθειά τους σε επιστημονικό και προσωπικό επίπεδο.

Τέλος θα ήθελα να ευχαριστήσω, την οικογένειά μου, του γονείς μου και τα αδέρφια μου για τη στήριξη τους όλα αυτά τα χρόνια και την εμπιστοσύνη τους στις επιλογές μου.

Στυλιανόπουλος Χαράλαμπος

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Overview

In 1965, Moore predicted, based on observations, that the number of transistors on integrated circuits will double approximately every two years. This was the case for many years after that, with the industry pushing hardware performance forward to meet that goal.

Advancements in transistor technology made it possible to integrate more of them in the same silicon chip. Their frequency also increased, presumably doubled every 18 months. At the same time, emerging architecture techniques made it possible to further exploit the continuously increasing cpu power. The use of deeper pipelines superscalar architectures made it possible to increase the operation throughput. Out of order execution and branch predictors helped to make sure that most of the available transistors are utilized, at any given point of the execution. Moreover, hardware manufacturers were able to keep pushing for better performance, without worrying about memory consumption. They were able to achieve higher frequencies fore more transistors, by reducing the voltage supply needed, thus keeping power consumption at controlled levels. In the meantime, cache hierarchies grew bigger and faster to meet the demands of CPU. All those factors made it possible to keep a steady rate of increase in computing power, without any demand for new programming models.

That staggering rate of advancement reached a peak around 2004. At that point, manufacturers were not able to keep making transistors smaller and faster, without sacrificing power consumption. At the same time, memory performance was now too small compared to CPU frequencies, creating and unbridgeable CPU - memory gap. In order to meet the increasing demands for computer power, computer science shifted towards multicore CPUs. After

that point, improvements in performance were no longer achieved exclusively by advancements in architecture, but required fundamental software support. The excess of cores and transistors would mean nothing, if the overlaying software was not able to effectively break execution path down to independent path that could be run in parallel. From user applications, to compilers and operating systems, software needed to be redesigned in way that could exploit the extra cores and the available architecture. Nowadays, the demand for multicore CPUs is no longer an exclusive characteristic of super computers and data centers, but has become rather mainstream, entering the field of desktop computers and embedded systems. Over that time, parallel software techniques have made great steps of progress towards a more efficient utilization of every aspect of modern computer architectures. Despite that progress, parallel programming continues to be a challenging task, and a field of ongoing research.

1.2 Parrallel Architectures

According to Michael J. Flynn, computer architectures are classified in 4 different categories, according to their instruction and data stream.

- **Single Instruction Single Data stream (SISD)**.
- **Single Instruction Multiple Data streams (SIMD)**.
- **Multiple Instruction Single Data streams(MISD)**.
- **Multiple Instruction Multiple Data streams(MIMD)**.

SIMD and MIMD are the architectures that attract the most interest in parallel programming. SIMD machines execute the same commands over multiple data. GPUS and accelerators are characteristic examples of that architecture. In MIMD machines on the other hand, multiple autonomous processors execute different instructions on different data. Multicore and Distributed Systems are MIMD machines.

MIMD machines are further classified into 3 categories, according to their address space:

1.2.1 Shared memory

In shared memory architectures, every processor has its own hierarchy of cache memory and all processors share the same main memory. Interconnection between processors and memory is typically done through a memory bus, but more sophisticated interconnection networks can be used.

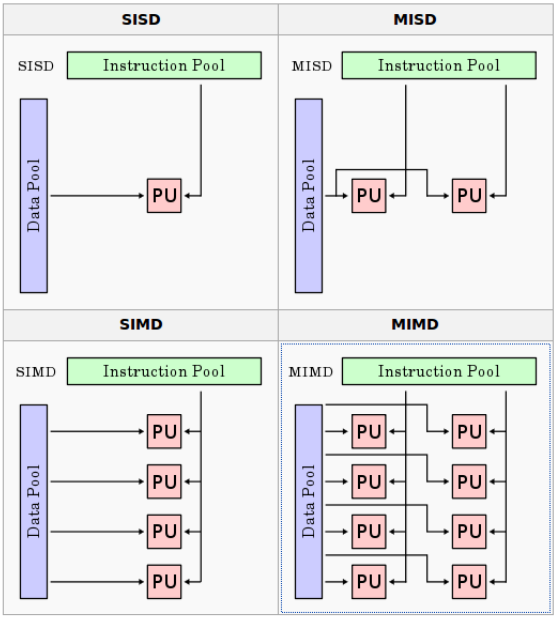


Figure 1.1: Flynn's taxonomy

All working threads work on the same address space and accessing a memory location previously modified by another processor can be as simple as an access to any other variable. This makes programming in shared memory architectures seem easy, but concurrent access from different processors on the same memory location can lead to unexpected results if not a synchronization scheme is not used. Moreover, shared memory architectures wont typically scale beyond a few thousand nodes, because the bush and memory bandwidth cannot keep up with the increased traffic.

Access to memory can take the same amount of time for all processors(Uniform Memory Access -UMA) or can vary depending on the processor and the memory location(Non Uniform Memory Access).

Non Uniform Memory Access is an architecture design where memory access time depends on the relative location of the memory location and the processor. In NUMA models, it is typical to create NUMA nodes or packages each with it's own fragment of main memory, while separate nodes are connected through a slower network. Processors can quickly access their package's memory but require more time to access a memory location that resides on a different package. This model is beneficial to workloads where threads, or small groups of threads access the same data, a scenario quite common in servers.

On the other hand, NUMA might perform badly when threads from dif-

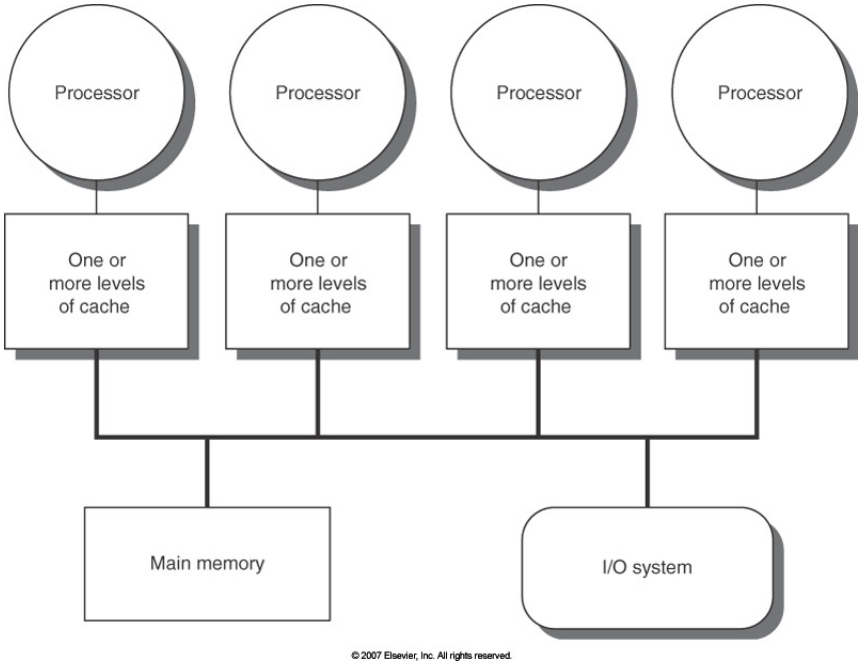


Figure 1.2: Shared memory Architecture

ferent packages have access to the same data, partially because of the need for cache coherence which is mentioned later. In NUMA, cache coherence traffic, such as the one produced by MESI protocol, is exchanged between cache controllers and the need to keep several packages coherent may come in great cost. For this reason, some operating systems try to implement NUMA-friendly scheduling to keep communication between packages to a minimum.

1.2.2 Distributed memory

In distributed memory architectures, every processor has it's own cache and local memory, and it has access only to that memory hierarchy. All processors are connected on an interconection network(Ethernet, Mirinet), consist of complex switching topologies. Communication is done using messages from processor to processor and usually being served by memory controller with direct memory access(DMA). Accessing data stored on memory outside the processor, may require several messages to be past between nodes.

Programming in a distributed memory environment can be quite challenged, because memory locations needed by a program may not be accessible locally but have to be requested in advance. Efficient parallelization

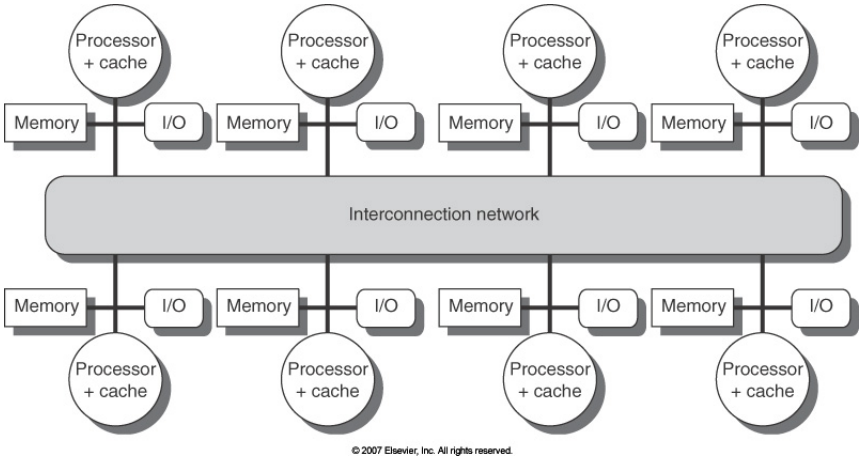


Figure 1.3: Distributed memory Architecture

over distributed memory, requires understanding the memory dependencies of the program ability to effectively distribute memory in advance in a way that will minimize communication between distant processes. However, distributed memory can achieve far greater scalability than shared memory, up to thousands of node that can be dynamically inserted and removed from the network.

1.2.3 Hybrid Memory

In a hybrid memory architecture, a groups of processors share a common local memory and, possibly some cache levels, similar to the shared memory architecture, whereas these groups of processors communicate with each other over a an interconnection network. This a typical topology used in clusters, where several processor share a common address space and create nodes that can be inserted in the interconnection network.

1.3 Parallel Architecture and Programming Challenges

1.3.1 Memory coherence

Multiple levels of cache memory are quite common in processing systems, as they reduce the cost of memory references by keeping copies of recently uses memory locations close to the processor. This replication of memory locations can cause many problems, even in uniprocessor systems when a Direct

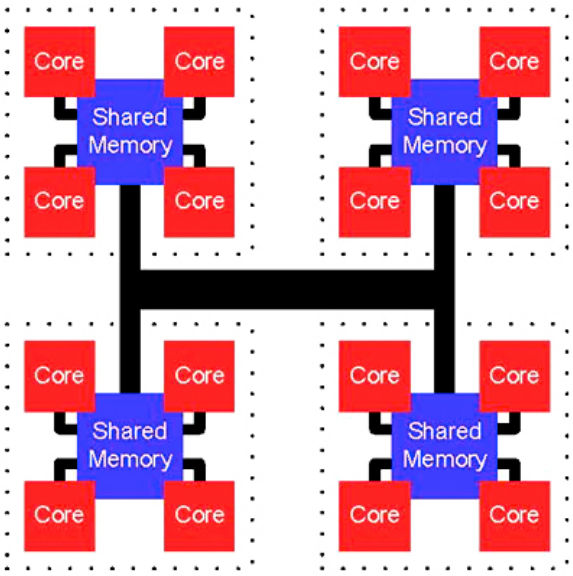


Figure 1.4: Hybrid memory Architecture

Memory Access(DMA) mechanism is involved. In multiprocessor systems, several processor may access the same memory location concurrently, creating many copies that reside in multiple caches. If that memory location is only read, it can be shared through the processor without a problem. If however that location is modified by one of the processors, all the others must be notified of the change, or else they might end up using an outdated image of that memory location. Therefore, a certain protocol , called memory coherence protocol, must be followed to ensure that all processors are accessing the update value of a memory location. The most widely used memory coherence protocol is the MESI protocol.

In MESI protocol, every cache line is marked with one of the following states:

Modified. The cache line is present only in the current cache and it's value has been modified from the value in the main memory. The cache is required to write the value back to the main memory at some time in the future, before allowing any other thread to read that memory location from the memory.

Exclusive. The cache line is present only in this processor and it's value is the same as in main memory. The cache line will change to shared state, if read by another processor, or to modified state if written by that processor.

Shared. The cache line may exist in many caches and it's value is the same as in the main memory.

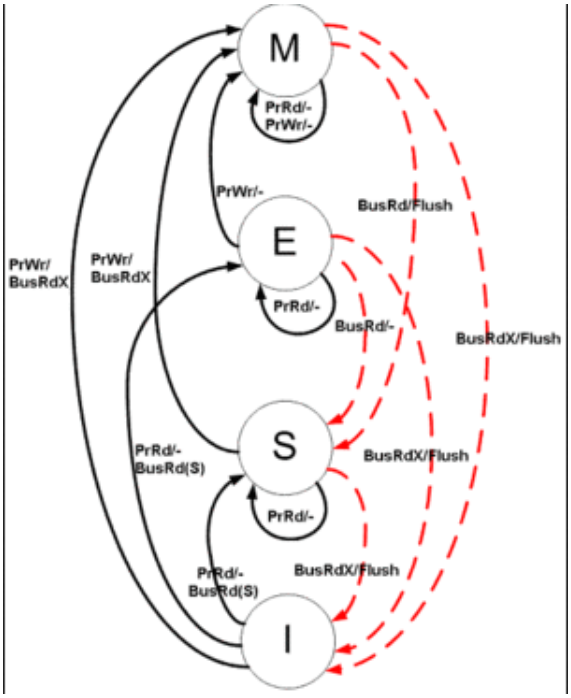


Figure 1.5: State diagram of the MESI protocol. The transitions are labeled "action observed / action performed"

Invalid. The cache line is invalid.

According to the state diagram depicted in figure 1.5, the MESI uses the above mentioned 4 states for every cache line to allow all processor to have a consistent view on all memory locations.

1.3.2 Memory consistency

In a program running on a multithreaded machine, the order between operations is not always guaranteed, and operations issued concurrently by many processors may actually take effect in an oder that cannot be determined in advance. In this way, no guaranteed ordering can be assumed when writing parallel programs. In an even more relaxed model, it is even possible for operations on the same core to be rearranged by the compiler or the processing unit, for various performance reasons. In that case, memory barriers can be used to make sure that operations all operations before the barrier have been completed.

1.3.3 Amdahl's Law

In an effort to achieve better performance, parallel programming aims to exploit more cores to speed-up sequential problems. Amdhal's law is a famous theoretical formula used to determine the maximum expected speedup achieved

Consider speedup S as the ratio between the time it takes on processor to finish a job and the amount of time it takes for n processor to finish that job. Also consider f to be the fraction of the job that cannot be parallelized and must be done sequential. Amdahl's law dictates that

$$S = \frac{1}{f + \frac{1-f}{n}}$$

where n is the amount of available processors. The effects of that law can be understood through an example. If we manage to parallelize as much of 90% of an application and have only 10% executed sequential, using 10 processors to run that application will yield, according to Amdahl's law an speedup of 5.2, meaning only halve of our processing power is utilized.

Figure 1.6 visualizes the effect's of Amdahl's law. What it really means is that the sequential part of the algorithm must be as little as possible, if we hope to achieve high speedup. In most of the cases, the sequential part of a multithreaded application is the communication between the threads and the input/output operations that are all served by a single memory channel with limited bandwidth. Parallelizing more and more of that sequential but is not always easy but it is , in many cases, the main focus of parallel programming and they key to improve performance.

1.3.4 Synchronization and Progress

In parallel applications, threads will eventually need to communicate with each other or perform operations on a common data structure. In the shared memory model where every shared variable is easily accessible, it would be tempting to simply modify the shared memory as if threads were running in isolation, each with access to its own memory space. This would be entirely wrong since synchronizing concurrent accesses on shared memory locations is extremely important in parallel programming. The next figure shows a simple example of how the absence of synchronized access on a counter leads to inconsistencies. The counter is incremented once by every one of the two threads, but due to the patter of operations executed, it is eventually incremented only once.

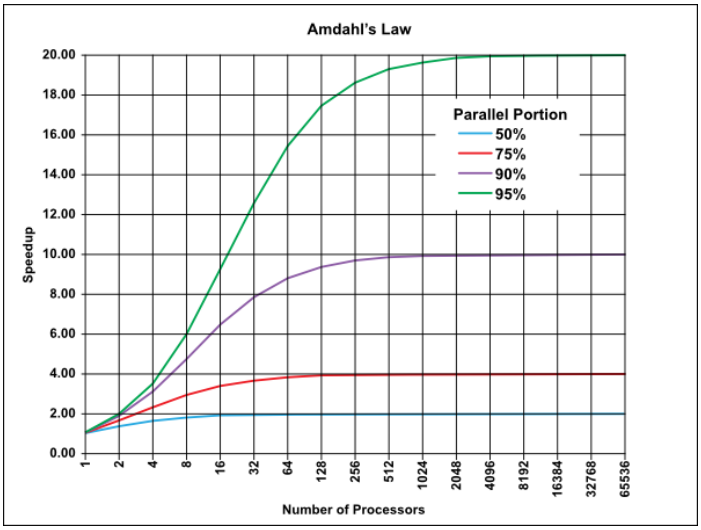


Figure 1.6: The effects of Amdahl's law

```
1
2  Thread1                                  Thread2
3
4                                           var2 = counter
5  var1 = counter
6
7                                           var2 = var2 +1
8                                           counter = var2
9
10 var1 = var1 +1
11 counter = var1
```

Listing 1.1: Inconsistencies caused by the lack of syncronization

This problem cannot be solved without hardware support, specifically atomic primitives that will ensure that an operation will be executed without being interrupted by a concurrent thread. With the use of Fetch -And-Inc, incrementing a shared variable can be implemented safely, at the extra cost of locking the memory bus until the variable is read and incremented. Other useful atomic operations, provided in most systems, are Test And Set(TAS) and Compare And Swap(CAS). Test and Set will atomically set a variable to 1 and return its previous value. Compare And Swap will compare the value store at a memory location with a giver value and only if they are equal, CAS will update that location to a new given value and return true. If not the operation will fail.

Synchronization schemes in a parallel program can be classified in one of three categories : **blocking**, **lock-free** and **wait-free synchronization**.

- Blocking approaches, usually use mutual exclusion to allow only one thread to access the critical section at a time.
- Lock-free approaches allow several threads to access the same data and introduce overheads mostly when there are actual conflicts.
- Wait-free approaches guarantee that every thread will finish it's operation within a finite number of steps, although that number can be high, and are better used in real-time applications where a maximum operation latency must not be surpassed.

Non blocking approaches, meaning either lock-free or wait-free approaches, generally allow access by multiple concurrent threads without mutual exclusion. In cases like that, threads will access share data locations and attempt to perform changes without blocking access to other threads. Although individual operations may fail and need to be started over , even indefinitely, it is guaranteed that some other will succeed its operation during that time.

1.3.5 Synchronization

In most concurrent programming models, synchronization is achieved by one of the following techniques:

Mutual Exclusion

Most of the times, blocking synchronization is achieved by implementing mutual exclusion, usually with the use of semaphores and locks. Almost all locks use a Test-And-Set atomic primitive to set a memory location. If that memory location is set, and the thread that sets that location form 0 to 1 is considered as the lock owner. The use of a lock over a section of code is the easiest way to ensure that only one thread will access that section of code (critical section) at a given time.

In it's simplest form, setting a lock consists of continuously executing TAS on the same memory location until the value return(the value in the memory location before setting it) is zero. This will cause heavy bus traffic, since every TAS will lock the buss until it is executed, along with heavy cache coherence protocol traffic, since every TAS is essentially a write that will invalidate all other copies of the lock, including the owner's. For this reason , a basic improvement is the uses of a Test and Test And Set (TTAS)

lock, where the value of the lock is first read locally and a thread will attempt to set the lock with a TAS only if the value read is zero. Even in that case though, constantly reading the value of shared variable will cause substantial buss and cache coherence traffic. That's why it common to implement some sort of back off mechanism, where a thread that found the lock taken, will wait some time before attempting to check it again. This however makes the lock quite unfair, since some threads may spend more time waiting than others. Other locking schemes such as queue locks may improve fairness, but usually the overhead involved is too high.

Locks make synchronization easy, but sometimes we need more than one locks to allow threads to access several locations in parallel. In that case, it might be proven very difficult to come up with a fine grained synchronization scheme that will avoid deadlocks. Deadlocks appear when two thread hold a lock each, with every thread trying to acquire the lock held by the other. In that case no thread will processes and execution reaches a dead end. Even when deadlocks are meticulously avoided, blocking applications suffer from the effects of preemption. If a thread holding a lock looses the process for a while, every thread waiting at that lock will grind aimlessly, creating a "convoy" effect where the slow thread holding the lock will keep all the rest behind it.

Atomic Operations

Most of the times, non blocking approaches are implemented using the Compare And Swap atomic primitive. A thread will need to commit its change using a single CAS. If it is successful, the operation is complete. If not, this is a sign that another thread managed to complete an operation prior to that thread, so the currents thread's view of the shared memory is outdated and the operation needs to restart from the top. Alternatively, Link-Store Conditional (LL/SC) can be used. LL/SC will mark a memory location and update it only if no other concurrent updated have occurred in the meantime.

There are times when one CAS is not enough to successfully complete an operation, because several locations must be updated in atomic manner. In those cases, a typical approach to execute the operation incrementally and have other threads help complete the work of others. In lock free approaches, it is possible that a thread will take an infinite amount of time to complete its operation because it endlessly helps complete the work of others, but the shared object is guaranteed to progress in general.

Correctness of non blocking approaches is generally difficult to prove. Its based on find linerization points, i.e. code lines where we can consider that

the operation is instantly completed. The order by which these points are reached, dictates the state of the shared object.

One of the problems that occurs during the implementation of this algorithm in particular and of algorithms that use C-A-S in general, is the so called ABA problem. In summary, the problem is described by the following scenario: A process observes that a memory location is in a state A and then is halted for a while. In the meantime, another process alters the state of the memory location to B, then back to A again. The initial process will find that the state of the memory location is A and a CAS will succeed, without knowing that the state has changed in the meantime. This problem is related to the lack of a garbage collector that would ensure that we could not release a memory segment that is still being referenced by a thread.

A standard way to solve the problem is to incorporate a counter along with the memory location we are trying to update. CAS is now performed not on the memory location itself but on the pair <memory location, counter<and every succesful CAS will increment the counter. Thus, a delayed CAS will find the memory location altered because of the different counter and it will fail. This does not solver the problem, because the counter may track only a finite amount of updates. Instead it lowers the probability that the ABA problem will happen, at the expense of leaving fewer bits available for the variable. There is, in fact, a tradeoff between the number of useful bits in the memory location and the probability that the ABA problem will happen.

Alternatively, there have been put forward many ways to solve the ABA problem, for example with the use of reference counters, that doesn't require merging counters and variables.

Transactional Memory

Transactional memory is a potential alternative method of synchronization that promises ease of programming. It is main principle is derived form database systems and it is based on the concept of transaction: a section of code that must be performed atomically. Transactional memory monitors the critical sections run by all processors and try to detect conflicts. A conflict is detected when a processor in a transaction reads a shared memory location that is modified in another concurrent transaction and vice versa. If no conflicts are detected, the effect of a transaction take place and are visible to all other processors (transactional commit) . If however a conflict is detected, the system tries to return back to its state before the transaction (transactional abort). Transactional memory mechanisms can implemented either on software or on hardware.

Software transactional memory does not rely on any hardware support.

It instead implements a software subsystem that keeps track of transaction, detects conflicts and performs commit and aborts. Although portability is achieved, the cost of monitoring and resolving conflicts is usually to hight and the applications where it can have an advantage are limited. On the other hand, in hardware transactional memory, conflict detection and resolution is done by appropriate hardware, achieving higher performance than software transactional memory. Even in this case, hight contention may result in frequent aborts and performance degradation, but it is possible to achieve fine grain synchronization in cases where it would be extremely difficult without the use of transactional memory.

1.4 Concurrent Data Structures

Data structures, as a way of storing and organizing information is one the most important factor in any programming application. Especially today where projects grow larger and larger, the need for sophisticated data structure that will provide meaningful organization and easy of access is essential. In fact, the performance of the underlying data structure is a big part of the overall performance, and in many cases it may become a bottleneck for the whole application.

With the introduction of parallel programming, available data structure had to be re-factored respectively, to provide safe, synchronized access to multiple threads. Challenging as it was, for sequential data structure, to ensure safety and consistency of the structure against the effects of any operation, the inherited difficulties of parallel programming the way any thread can unexpectedly perform operations on common data, made keeping concurrent data structures safe even more difficult. Even further, it is not enough that the structure is kept safe and nothing bad will never happen; there is also the demand that something good will always happen and the data structure as a whole will keep progressing and serving requests.

Coming up with fast concurrent data structures is important, in parallel programming, for one more reason. According to Amdahl's law, explained previously, the part of an application that is sequential, seriously prohibits it's maximum gain from parallelization. It just so happens that operations on a shared data structure usually belong to that sequential part. Even in applications that are naturally parallel and every thread can execute its operations independently, some sort of data organization in a shared structure that is accessible by all will be eventually needed. It is therefore of paramount importance that a concurrent data structure will try to allow more work to be done in parallel and reduce sequential parts, that usually consist of synchro-

nization costs. In addition to that, concurrent accesses to the data structure must not create bus congestion and cache coherence traffic, since this would further degrade overall performance and introduce more bottlenecks. All this matters are in fact the major points of focus in the study of concurrent data structures.

1.5 System Configuration

The system we used to benchmark our implementations was the "Sandman" platform, a 32-core NUMA architecture with the following characteristics.

- 4 packages (Intel(R) Xeon(R) E5-4620 @ 2.20GHz)
- 8 Cores per packages(16 threads with hyperthreading)
- 32KB L1 cache per core
- 256KB L2 cache per core
- 16MB L3 cache per package
- 256 GB RAM

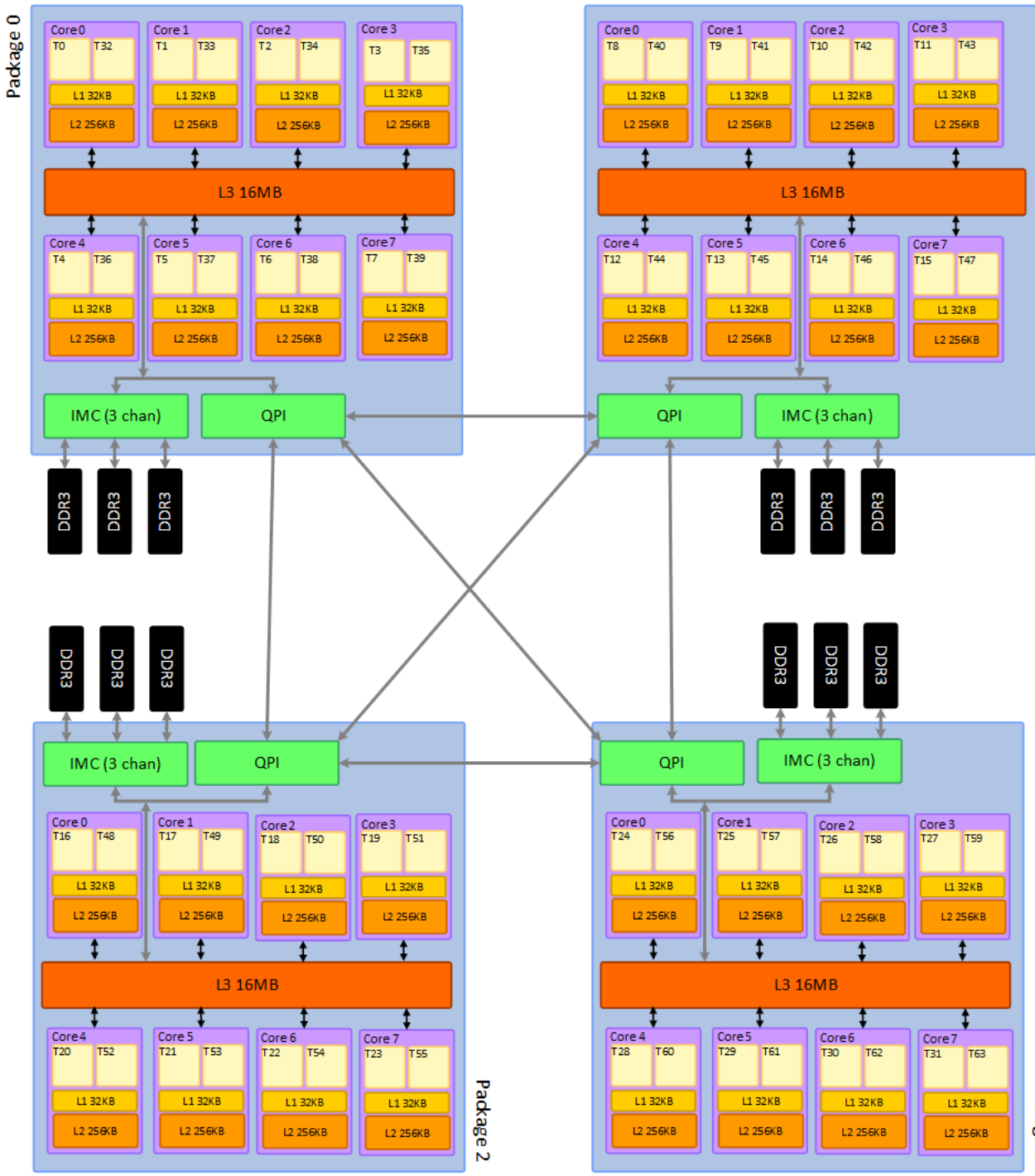


Figure 1.7: Sandman Platform

Chapter 2

FIFO Queues

2.1 Introduction

FIFO Queues are one of the most widely used data structures. They have been studied thoroughly and are used in many projects. Applications of queues range from web servers to interruption managing, inter-process communication, CPU schedulers and many more. They are essential on any application that requires some sort of First in First out ordering of data.

Queues are usually implemented by linked lists and support two simple operations; enqueue and dequeue, shown in figure 2.1 and 2.2

In many applications, there is the need of having multiple threads, communicating through a shared queue, that is often needed to serve millions of operations per second. Concurrent queues can be quite handy in a scenario where multiple producer threads create work items, that need to be consumed by several worker threads. Producers may create many items in a burst, while consumer threads are receiving them at a slower rate, so the excess of times need to be organized in an efficient manner. Although situations like that can be quite common, implementing an efficient queue that allows concurrent access from multiple threads can be quite challenging. For this reason, concurrent FIFO Queue implementations attract theoretical, as well as practical interest.

In order to maintain the coherence of the data structure in a multi-threaded environment, synchronization between threads is needed. An example of what could go wrong without proper synchronization is shown in figure 2.3 .Queues are a structure that, by its nature, allows low levels of concurrency, since all reads and writes are applied on Head and Tail, rendering these locations as hot-spots. Therefore, we expect low scalability, as the number of parallel threads increases.

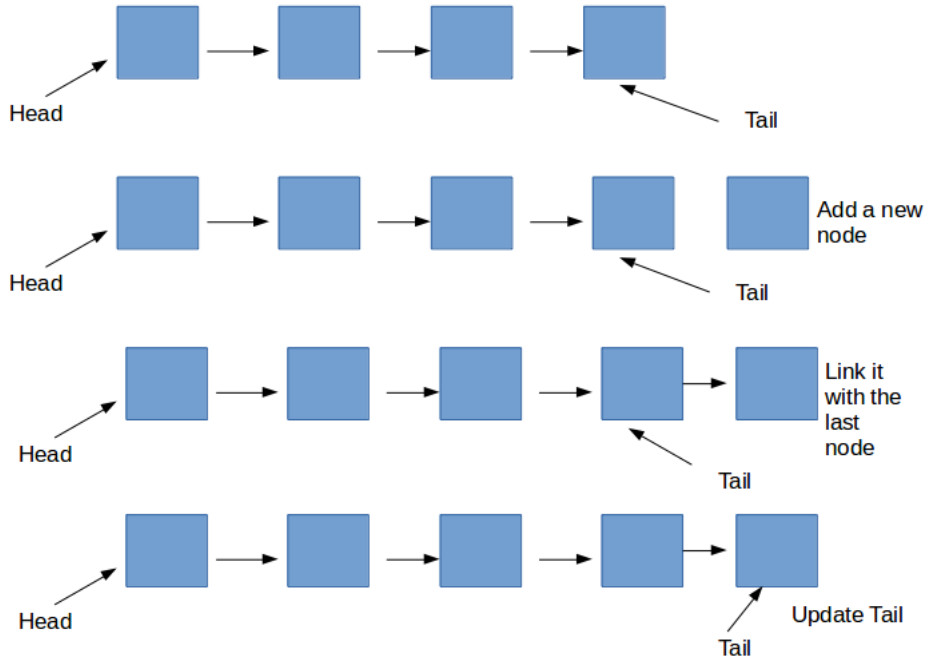


Figure 2.1: The basic enqueue operation

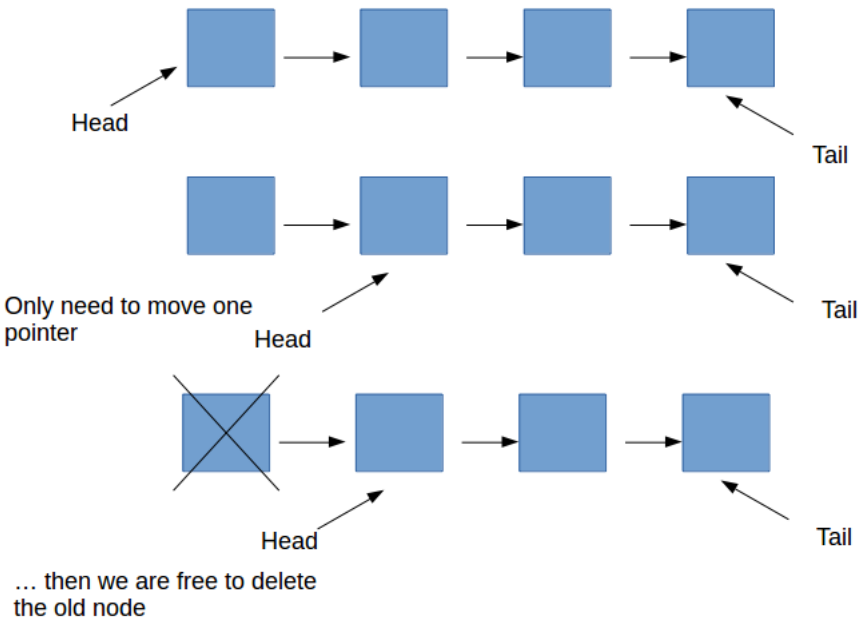


Figure 2.2: The basic dequeue operation

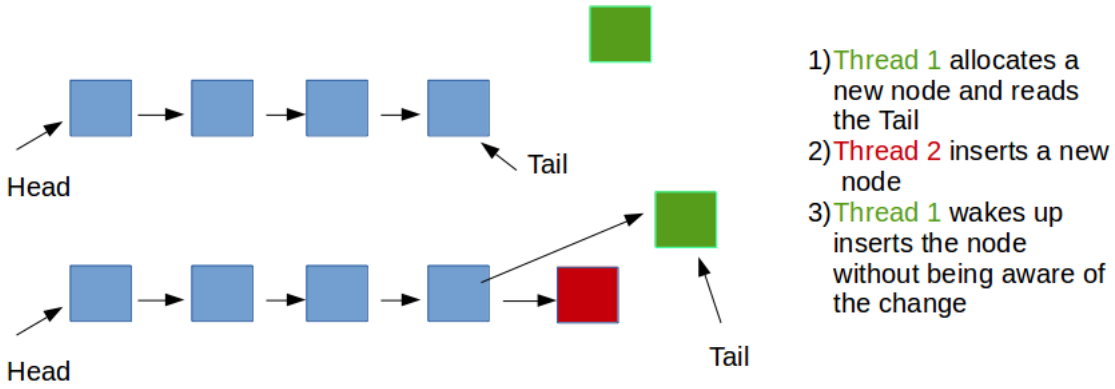


Figure 2.3: An example of how lack of proper synchronization can make a queue inconsistent

Given the low level parallelization offered by the structure, the problem of high performance, essentially becomes the problem of finding a low cost synchronization scheme, as well as achieving better cache utilization.

2.1.1 Global Lock

In our first implementation we adopted a naive, coarse grained strategy. We introduce a global lock, which every thread is trying to set at the beginning of an enqueue or a dequeue. As expected, performance shown in figure2.4 is disappointing and the implementation does not scale. Since only one thread can execute an operation on the structure every given time, while all the other threads spin on the lock, we would expect performance to remain unchanged for any given number of threads. Instead, we can see an imediate and rapid decrease in throughput.

The reason for this poor performance is that the naive global lock implementation cause heavy cache coherence protocol traffic and bus congestion. Even after we implemented and used a TTAS(Test and Test And Set) lock that reduces traffic due to the cache coherence protocol, delays were still high. The main reason is that constantly reading the lock to check it's state produces heavy traffic. Also, since the lock is changing owners frequently, pointers on the list will ping pong back and forth across different caches, causing a lot of cache misses.

A solution to that problem is the introduction of backoff: when a thread finds a lock taken, it wont try to check its state again for a while, but will instead wait on loop for a few iterations. This way, the shared memory

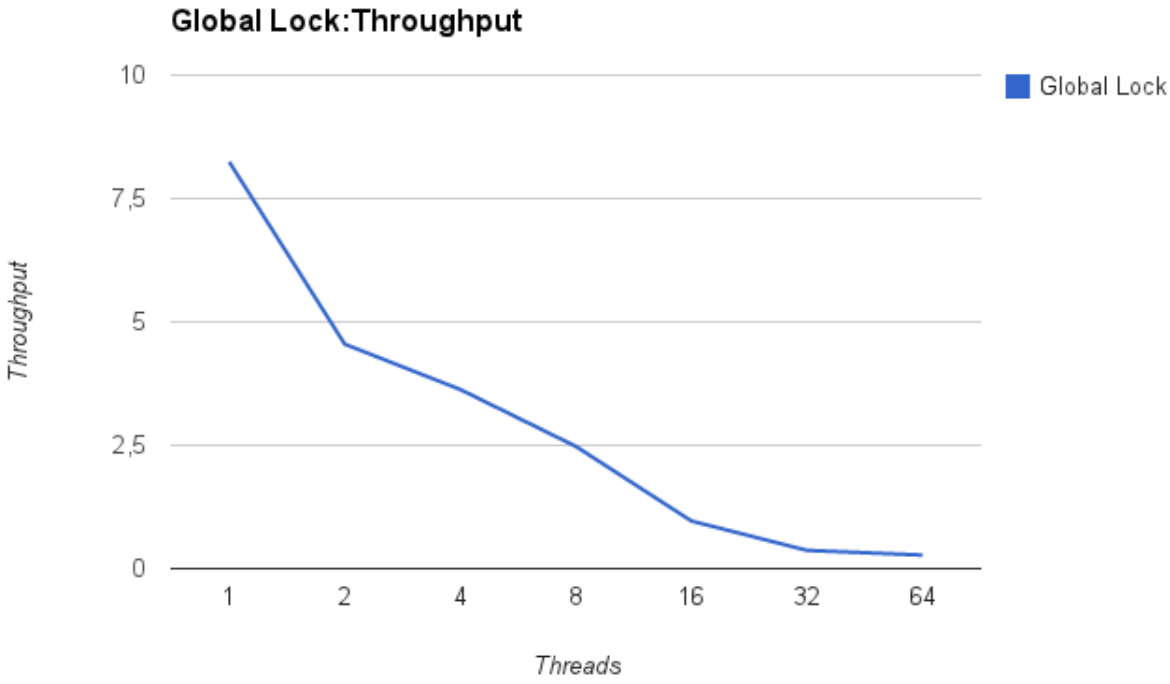
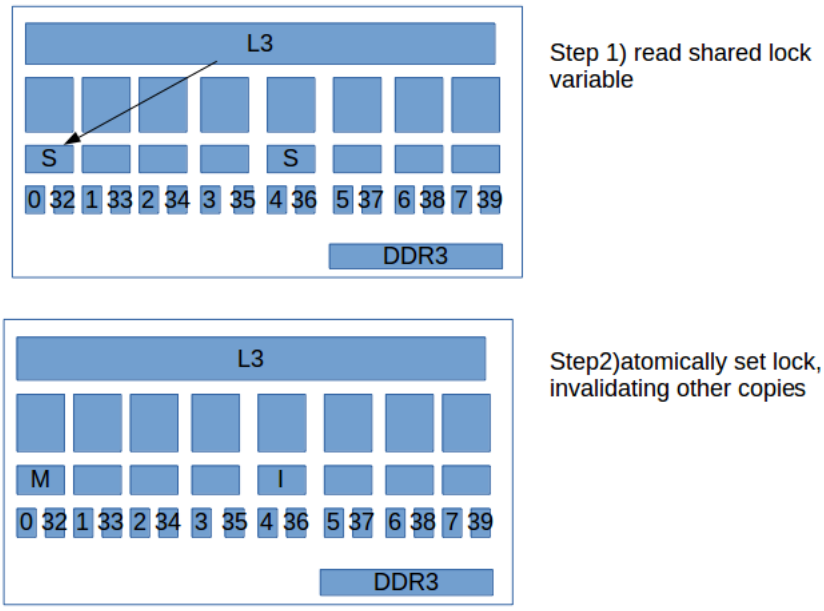


Figure 2.4: Performance of the naive global lock approach



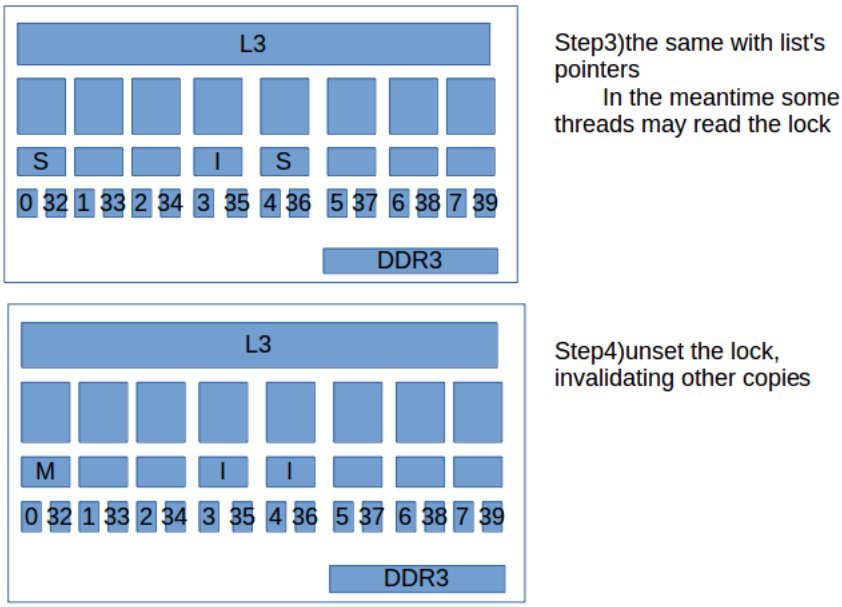


Figure 2.5: Cache coherence traffic on the simple lock queue

location is not polled so often,benefiting the general throughput as shown in figures 2.6 and 2.7.

We can easily see that in general, longer backoff generally increases overall throughput. The reason is more than the reduced traffic caused from polling on the lock. A high backoff means that threads that not manage to take the lock will become inactive for a while, but the thread that takes the lock will quickly complete the critical section (which is very small) and will probably return to attempt another operation very soon. At that time, the other threads will be inactive, spending time backing off, and that thread will probably take the lock again, hitting L1 cache for the memory location of the lock, the Head and the Tail. This certainly improves performance by reducing cache misses, but makes the lock quite unfair: a thread that finds a lock taken may have to wait a long time before setting it to perform a single operation, while a thread that takes the lock will probably retake it several times in a row, as shown in figure 2.8.

Another characteristic behavior is that throughput declines dramatically when the number of parallel threads exceeds the number of available cores. In this case, more than one threads share the same core and if a thread holding the lock is scheduled out, no other thread is advancing until that threads regains the CPU and unsets the lock. This is a general problem with locking implementations: If a thread inside the critical section is delayed, the

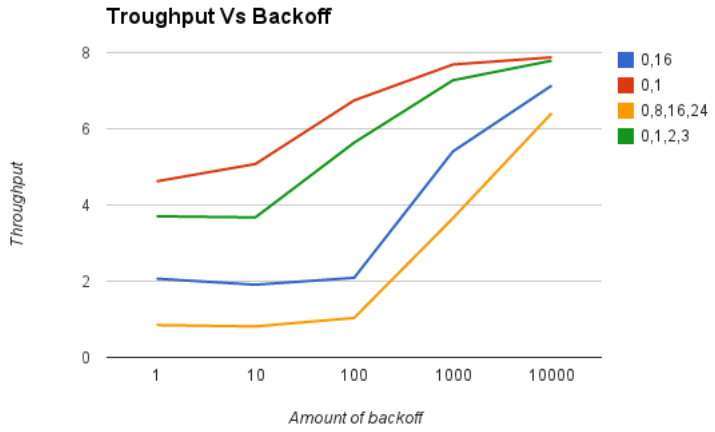


Figure 2.6: The amount of backoff affects throughput. Points on the horizontal axis are the number of iterations spend waiting on a loop. Red and green lines represent of threads inside the same node, while in blue and yellow lines, threads are placed across multiple nodes

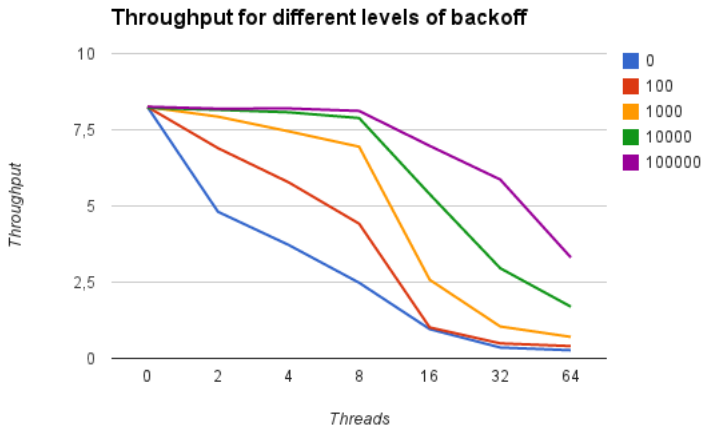


Figure 2.7: The overall scalling of global lock implementation for various amounts of backoff

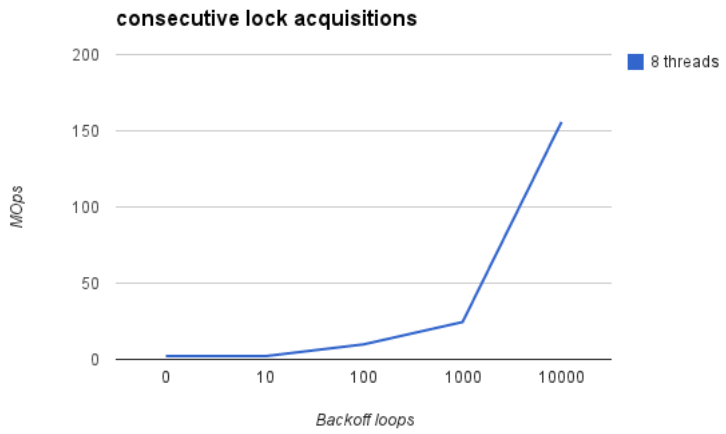


Figure 2.8: As the amount of backoff increases, the same thread will most likely take the lock more times in a row

progress of all other threads is halted and performance suffers significantly, making blocking algorithms not suitable for real-time applications.

For this reason, in the field of data structures in general and FIFO Queues in particular, great effort has been applied to come up with efficient, lock free implementations. One of the first, well known, successful implementation of a lock-free FIFO Queue is from Michael and Scott [8] and it serves as a base line for all further efforts.

2.2 Michael Scott Queue

2.2.1 Description

In the Michael Scott approach the data structure is implemented as a simply linked list, with a pointer Head referencing the start of the list, where dequeues are applied and a pointer Tail at the end where we can add new nodes. The node pointed by Head is considered a dummy node and is used to ensure that the list is never left empty.

In its core, the algorithm uses atomic operations (in particular Compare And Swap operations) to atomically modify the appropriate pointers. Every time, we run checks to make sure that we have a consistent view of the pointers we are trying to modify.

As depicted in figure 2.9, an enqueue requires 2 atomic operations: one to link the last node with the new node we are trying to insert and one to

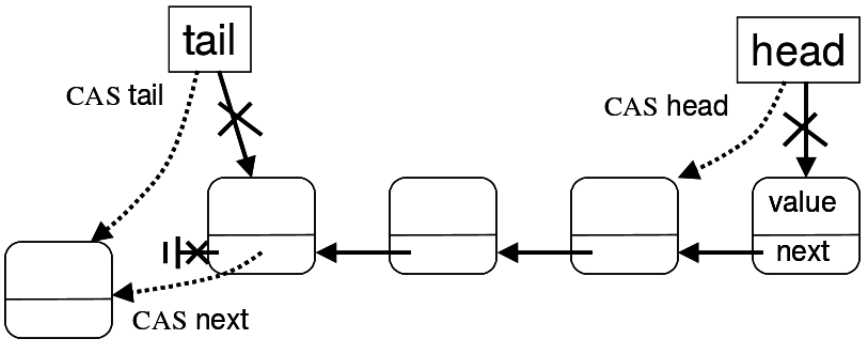


Figure 2.9: The basic layout of the link list used in Michael Scott implementation

swap the Tail pointer to the new node.

On the other hand, we need only one C-A-S on the value of Head to perform a dequeue.

2.2.2 Challenges

At any given time during the execution of a thread, the values of Head and Tail can change unexpectedly, causing the atomic operations to fail and the execution to start over from the top: read the new value of Head/Tail and try to change it atomically. Moreover, in order for an enqueue to finish, both two required C-A-Ss need to succeed. This in turn makes it possible for the new node to be added to the list, without updating the value of Tail accordingly, i.e. Tail doesn't point to the last node. For this reason, during enqueue or dequeue, it is necessary to check for this inconsistency and correct it.

In order to solve the ABA problem, we chose to use modification counters, which we increase on every successful C-A-S and we incorporate them along with every pointer on the data structure. Atomic operations are now performed, not on the pointer but on the pair <pointer, modification counter>which is now treated as a single variable. Thus, we now have to treat pointers in a non traditional way, extracting them from the variable using bit shifting which causes overheads and make programming more difficult and error prone.

The basic outline of the algorithm is shown bellow:

```
1
2  void enqueue (Queue_t * Q , int value){
3
4    node = allocate new node
5    node->value = value
6    node->next.ptr = NULL
7    while (1){
8      tail = Q->Tail
9      next = tail.ptr->next
10     if tail == Q->Tail
11       if next.ptr == NULL
12         if CAS(&tail.ptr->next,next, <node, next.count+1>)
13           break
14       else
15         CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
16   }
17   CAS(&Q->Tail, tail <node, tail.count+1>)
18 }
19
20 boolean dequeue (Queue_t *Q, int * pvalue){
21
22   while (1){
23     head = Q->Head
24     tail = Q->Tail
25     next = head->next
26     if head == Q->Head
27       if head.ptr == tail.ptr
28         if next.ptr == NULL
29           return False
30         CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
31       else
32         *pvalue= next.ptr->value
33         if CAS(&Q->head, head, <next.ptr, head.count+1>)
34           break
35   }
36   free(head.ptr)
37   return True
38 }
```

Listing 2.1: Michael Scott queue pseudocode

The result is a lock-free implementation, that does not require central locking of the data structure and allows all threads to advance. The absence of a lock leaves us with one less, heavily contested shared memory location to worry about. Performance is not affected by random delays that a thread can have, achieving robustness. Especially during over subscription, performance does not degrade like global locks' does.

On the other hand, CAS operations have a substantial cost, greater than

Figure 2.10: Performance of the Michael Scott queue compared to the global lock queue

a simple store. This store is even greater when the atomic operation is performed over different NUMA node, hence the sudden drop in performance when threads leave the package. Moreover, failing a CAS means that the operation needs to start over, stalling this particular thread, while others progress.

The number of CASs needed to perform an operation increases with contention, as shown in figure 2.11. Again, we try to reduce contention by introducing backoff after a threads fails to perform a CAS. The results shown in figure 2.12 suggest a modern amount of speedup, once again, at the expense of fairness.

2.3 Optimistic Queue

2.3.1 introduction

One of the drawbacks of lock-free approaches is that, each time an atomic operation fails, the execution start back from the top. Moreover, failed atomic operations are costly, due to the synchronization barrier they introduce. Especially in Michael Scott Queue, since both two atomic operations need to be successful in order to complete an enqueue, it is quite common for a thread to repeat execution again and again until it is done correctly. For this reason, we would like to reduce the number of synchronization points ,i.e. the number of atomic operations.

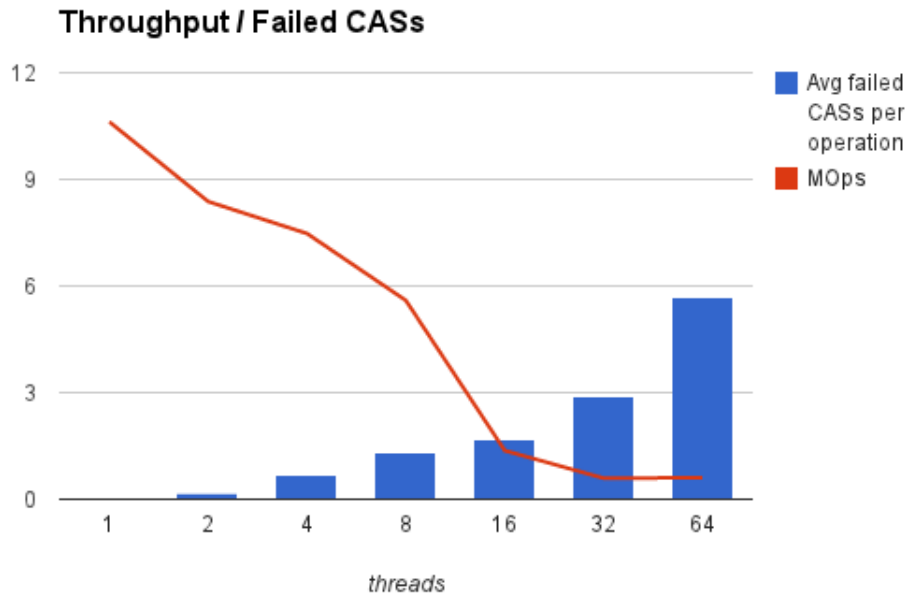


Figure 2.11: Performance and number of CASs per operation

A solution to this problem is introduced by the next algorithm we implemented, by Ladan-Mozes and Shavit [6]. This implementation follows an optimistic approach (which is why we will refer to this implementation as optimistic queue), in a sense that it runs quickly on the common case where there is no conflict and leaves the costly operations for the case where an inconsistency is spotted. In particular, one of the two C-A-S operations, during enqueue, is replaced with a simple local store, making sure that we correct the data structure in case it is inconsistent.

2.3.2 implementation

Practically, the link list becomes a doubly linked list, with the "next" directions being from Tail to Head. In this way, we only need a single C-A-S to Tail to make it point to the new node, in order to successfully complete an enqueue. However, in order to have access to node from head when we dequeue, we need pointers in the reverse order, as seen in figure 2.13.

Pointers in both directions are updated with simple local stores , without synchronization, which makes inconsistencies possible, in the "prev" direction. For this reason, as soon as an inconsistency is spotted, function FixList

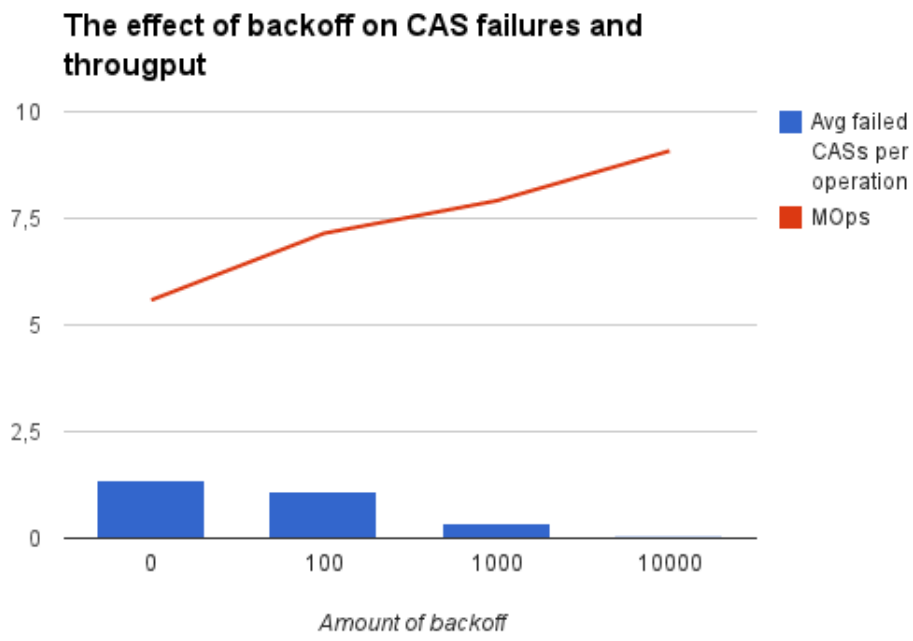


Figure 2.12: The effects of backoff on the performance of the Michael Scott Queue

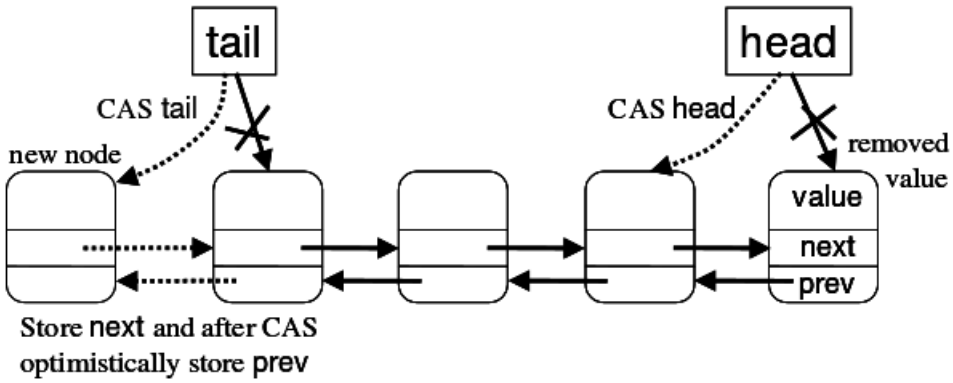


Figure 2.13: The basic layout of the doubly linked list used in the Optimistic Queue

is called to traverse the list and fix all the pointers. However, the reason for inconsistencies in the "prev" direction are the long delays a single thread might take and not contention. Therefore, we expect the number of calls to FixList to remain low, even when the number of parallel threads increases.

Inserting a new node in the list includes 3 steps: 1) Set the next pointer of the new node we are trying to insert 2) Compare And Swap on Tail, to make it point to the new node 3) Change the prev pointer of the next node.

```
1  void enqueue ( Queue_t * Q, int value){
2
3    node = allocate new node
4    node->value = val
5    while (1){
6      tail = Q->tail
7      node->next = <tail.ptr, tail.tag +1>
8      if CAS(&(Q->Tail), tail, <node, tail.tag+1>)
9        (tail.ptr)->prev = <node, tail.tag>
10       break
11   }
12
13 }
```

Listing 2.2: Enqueue operation of the optimistic queue

2.3.3 ABA and consistency

In order to avoid the ABA problem and spot inconsistencies, this implementation also uses modification counters, that are merged along with the pointers and are incremented in every successful C-A-S

Any thread might take arbitrary time between steps 2 and 3 and during this time more nodes might be inserted in the queue. Note however that, every time a node is successfully inserted in the queue(after successful C-A-S), the modification counter to be inserted next is incremented by one. Thus, pointers of consecutive nodes in the queue, will have consecutive modification counters. In this way, during a dequeue, if a prev pointer does not have the expected modification counter, FixList is called and the pointers are repaired.

```
1  void fixList(Queue_t Q, tail, head){
2    curNode = tail
3    while((head == Q->Head) && (curNode != head){
4      curNodeNext = (curNode.ptr)->next
5      if (currNodeNext.tag != curNode.tag)
6        return
7
8      nextNodePrev = (curNodeNext.ptr)->prev
9      if (nextNodePrev != <curNode.ptr, curNode.tag - 1>)
```

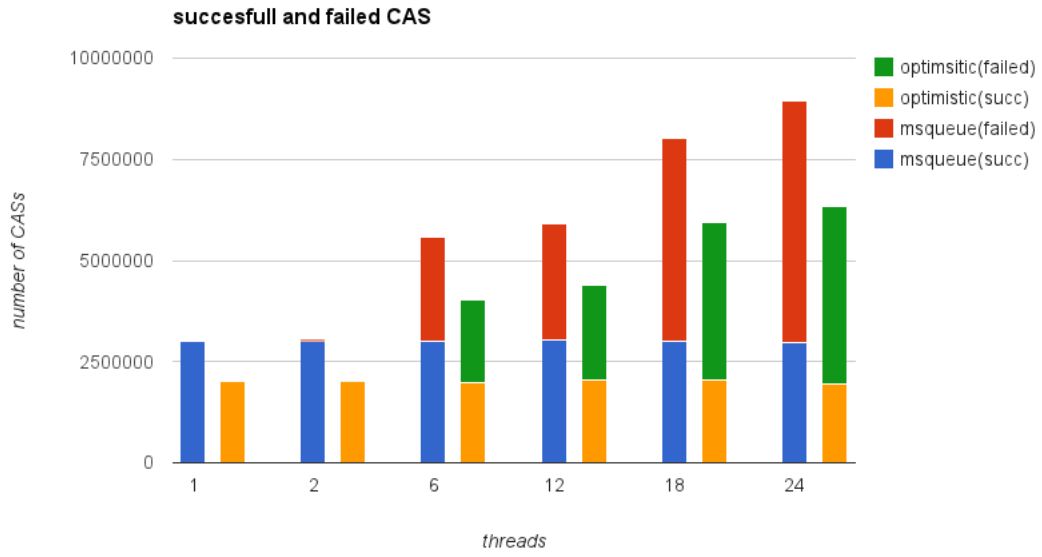


Figure 2.14: Total successful and failed C-A-Ss for the two lock-free implementations

```
10          (curNodeNext.ptr)->prev = <currNode.ptr, currNode.tag -1>
11
12        curNode = <curNodeNext.ptr, curNode.tag -1 >
13      }
14  }
```

Listing 2.3: The function used to fix pointers along the prev direction

Note that there must be special care taken to ensure that there is always one dummy node in the list and Tail never goes past that node.

2.3.4 Failed C-A-S operations

The next diagram compares the number of C-A-S operations(both successful and failed) needed to execute 1 million pairs of enqueue/dequeue, across these two lock-free implementations. We can see that the optimistic approach, as promised, requires less C-A-Ss and has, in total less costly, failed C-A-Ss as the level of concurrency increases.

We can see in figure 2.15 that during over-subscription, the global lock's performance suffers whereas lock-free approaches do not seem to be affected, since they are more robust against preemptions.

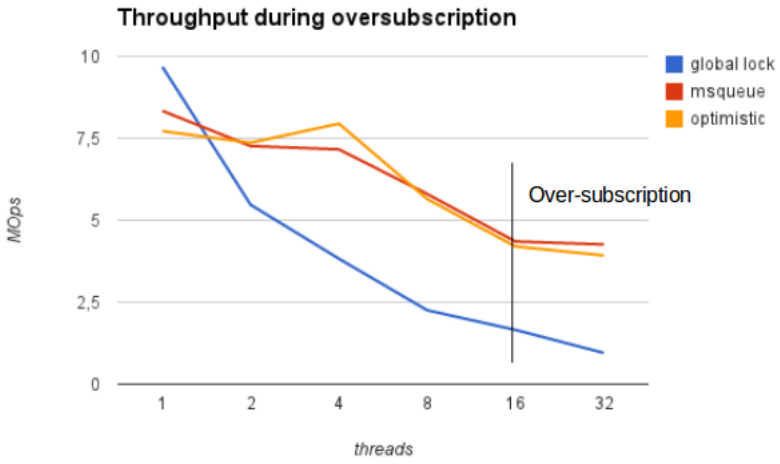


Figure 2.15: The effects of over-subscription on locking and lock free algorithms

All in all, the optimistic optimization seems to contribute a slight amount of speedup, compared to the Michael Scott queue, as shown in figure 2.16. The overall performance of the concurrent queue has significantly increased since the naive global lock algorithm, but many problems still remain. The very small critical section makes synchronization a high fraction of the total cost. Adding backoff was a step forward but it would potentially introduce heavy overheads for some threads.

2.4 Flat Combining

2.4.1 Introduction

The two previous implementations followed a fine grain approach, where every thread has access to the data structure and they are trying to achieve performance through high parallelization. As more treads are free to operate on the data structure, performance is expected to be better that locking approaches that block the progress of some threads. We next present the principles of flat combining, a programming approach by Hendler, Incxe and Shavit [2] that goes against the above mentioned statements.

In particular, the authors claim that the point at which the cost of synchronization between threads exceeds the benefit from high parallelization, is at a lower lever of concurrency than expected. Flat combining is based on

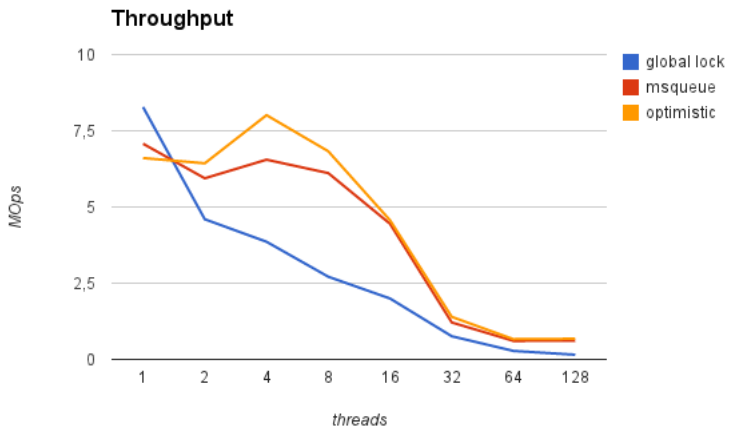


Figure 2.16: The performance of the optimistic approach

a synchronization scheme where each thread locks the data structure in an extremely low-cost way, gathers information on the operations trying to be executed on the queue by the other threads and then does the operations in their place. The result is an implementation with low synchronization cost and better cache performance, overcoming the drawbacks of blocking and low parallelization.

2.4.2 Implementation

Flat combining is a layer of abstraction that can be used over a sequential structure and its basic functions is the following: 1) Every thread publishes the operations it is trying to perform on the structure, along with any parameters, on its corresponding public record. These records can be in an array, for fast read/write or in linked list with dynamic size, proportional to the number of active threads. 2) Every thread checks the state of the locks and if it finds it unlocked, it tries only once to atomically set the lock. If the lock was locked already, the thread spins in its public record, waiting for a response. 3) If it takes the lock, this thread is considered the new combiner. It then traverses the public records and executes the request on the data structure, one by one, writing back the results. Finally the thread releases the lock.

The definition of the public record structure, as well as the main steps take by every thread are shown bellow:

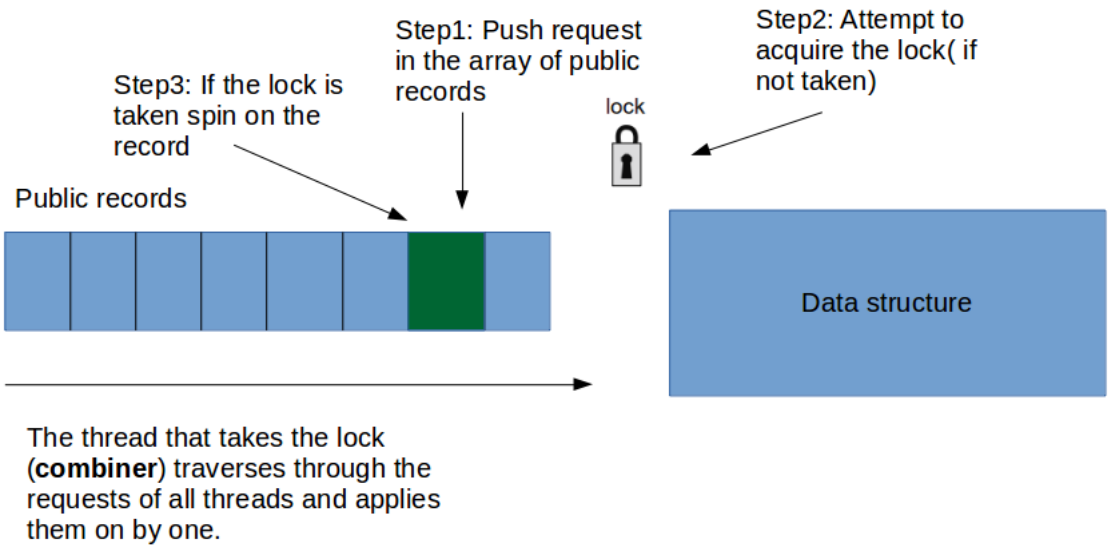


Figure 2.17: The basic synchronization scheme of flat combining

```
1
2  struct pub_record{
3    int pending
4    int operation
5    int value
6    int response
7  }
8
9  int try_access(Queue_t Q, struct pub_record * pub, int operation
       , int value){
10   int thread_id
11   pub[thread_id].operation = operation
12   pub[thread_id].value = value
13   pub[thread_id].pending = 1
14   while (1){
15     if ( Q->lock){
16
17       while (!pub[thread_id].response) do nothing
18
19       pub[tid].response = 0
20       return pub[tid].value
21     }
22     else{
23       if (__sync_lock_test_and_set(&(Q->lock), 1))
24         continue;
```

```
25        else{
26          for i=0 to number of threads
27            if pub[i].pending{
28              if pub[i].op ==1
29                do the actual enqueue
30              else
31                do the actual dequeue
32              pub[i].pending=0
33              pub[i].response=1
34            }
35          }
36
37          pub[thread_id].response=0
38          Q->lock = 0
39          return pub[thread_id].value
40      }
41    }
42  }
```

Listing 2.4: Basic layout of flat combining

2.4.3 Implementation characteristics and Benefits

From the way flat combining works, certain advantages can be concluded:

1. Since only the combiner has access to the structure, operations can be optimized with the best sequential algorithm, without minding synchronization and concurrency. This even allows us to implement concurrent structures, such as pairing heaps, that are otherwise difficult to implement using fine grain synchronization, by applying flat combining over already existent sequential data structures.

2. In some cases, the combiner can use some smart way to group the request and make the access to the structure faster and more efficient. For example, in concurrent stacks, a combiner can perform elimination between a concurrent enqueue and a concurrent dequeue. In our queue implementation, we used fat nodes to group data, as we explain later on.

3. The combined way of accessing the structure by a single thread can lead to better cache utilization.

4. Using a global lock to isolate the structure makes programming, as well as debugging, much easier.

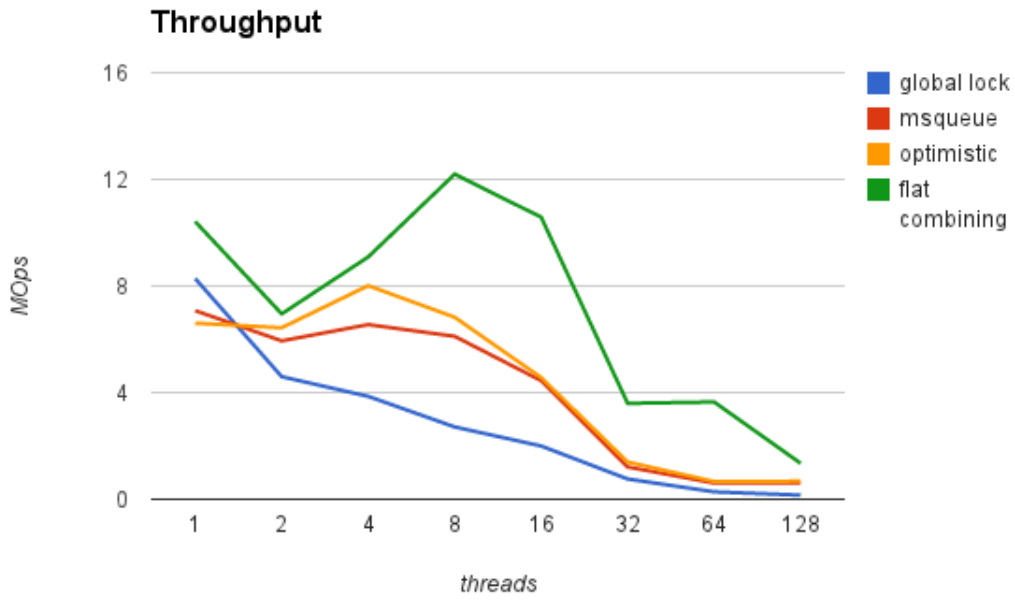


Figure 2.18: Perfromance of flat combining, compared to the other approaches

It is also important to note that flat combining, as a layer of abstraction that ensures synchronization, can bee used as it is over different data structured. Of course, not all data structures are benefited by flat combining. For example, if the cost of a single operation in a search tree is Θ (logn), using flat combining, the cost of k operations is in general Θ (klogn), while we could use parallel threads that execute operations independently on different parts of the tree in Θ (longn) total time.

FIFO Queues in particular, seem to benefit by flat combining, given that they already allow low levels of concurrency (only two access points). In our implementation, we organized public records in an array and used fat nodes, where every node can hold up to 16 values. Therefore, we only need to add one node in the queue and swap the Tail pointer only once, for every 16 values inserted. We also used two independent instances of flat combining, one for enqueues and one for dequeues, in order to exploit the maximum concurrency allowed.

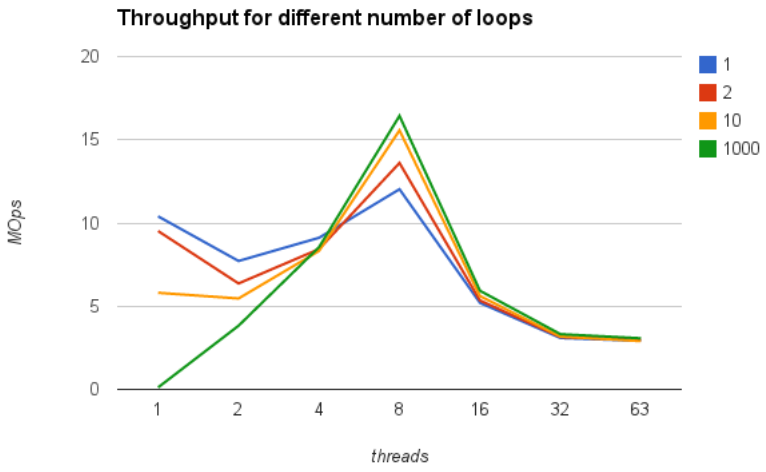


Figure 2.19: The effects of keeping the combiner iterating more than once over the public records

2.4.4 First results

Figure 2.18 shows the overall performance of flat combining compared to the other approaches. Performance seems to benefit greatly from the way batches of work are done by a single thread and the light-weight synchronization scheme. Especially inside the node, a fairly good scaling is noted. Trying to utilize that benefit even more, we keep the combiner iterating over the the public records more than once. Figure 2.19 shows that assigning more work to every combiner, although harmful when only a few threads are utilized, offers a slight speedup when more than 4 threads are used. Note however that this further increases the potential latency of the thread that becomes a combiner.

2.4.5 Optimizations

From the analysis of the algorithm we can deduct that every thread can, in theory, access the queue and alter its state. That means that the nodes of the queue can be in the cache memory of any thread, which in turn leads to increased cache misses and long delays.

By extending that idea, we can see that it might be better to have a dedicated combiner, residing in a specific core that will serve all requests. This way references on the lists nodes will always hit

The result, shown in figure 2.20, is a constant improvement in perfor-

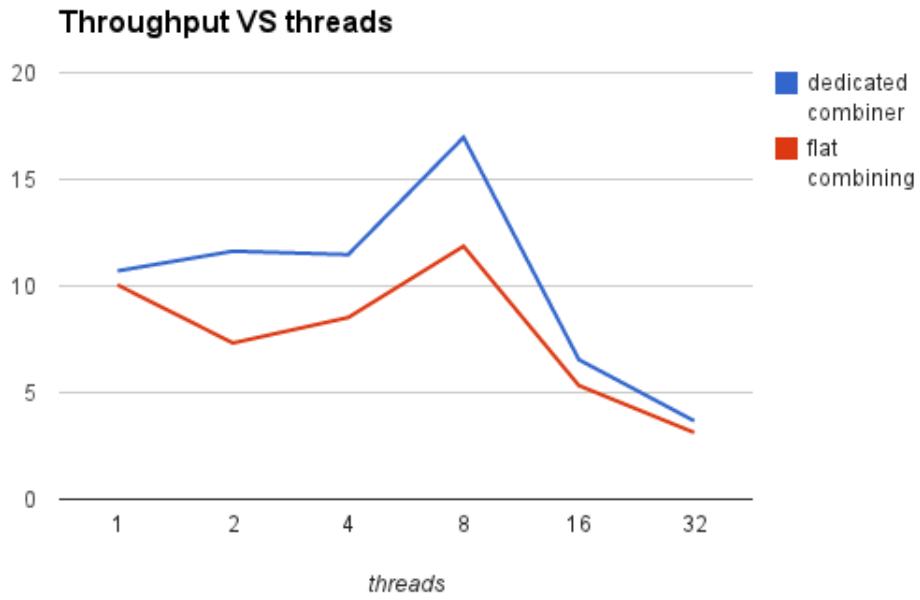


Figure 2.20: The speed up achieved by using a dedicated combiner

mance, that corresponds to less cache misses and better locality. Again, however, leaving the package significantly degrades performance and in fact, the overhead of communication between packages quickly becomes the dominant cost.

Addressing that problem,we can see that the flat combining synchronization scheme, as it is, introduces a lot of communication over different nodes, as the combiners need to read and modify the publication record of a thread that is on a different package.

The previously implemented flat combining algorithm is not NUMA-aware and cannot deal with the problems mentioned above. For this reason, we designed and implemented a hybrid implementation that combines previous algorithms, taking architecture into account, in order to achieve better performance.

The implementation that was eventually chosen consists of two stages. In the first stage, we use flat combining in every NUMA node. Threads of the same node are synchronized to come up with a combiner. In the second stage, combiners from each node are further synchronized to access the queue. In the second stage we tried several synchronization schemes (Michael-Scott queue, a second level of flat combining). Eventually, the best performance

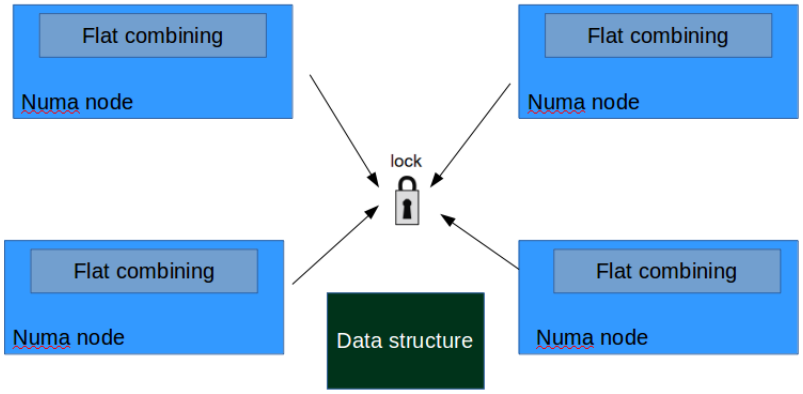


Figure 2.21: The use basic layout of the hybrid aproach

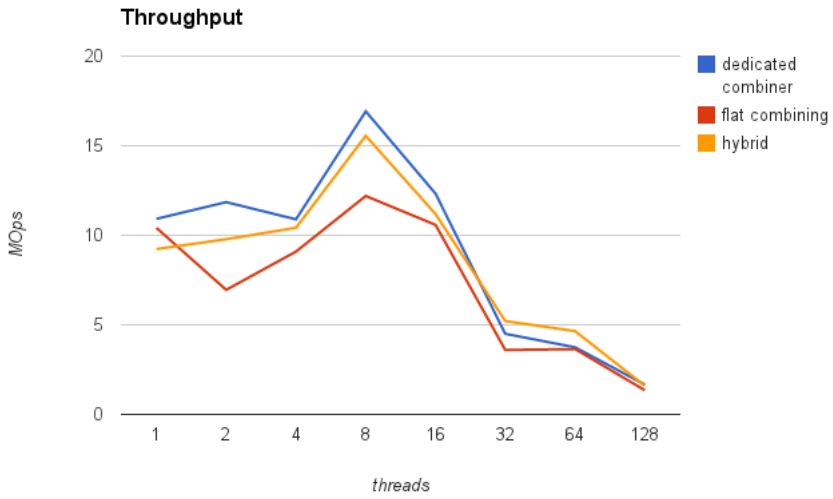


Figure 2.22: The use performance of the hybrid aproach compared to other flat combining schemes

for the given architecture was achieved by a simple TTAS global lock.

The result was a limited improvement of the performance when threads reside across several nodes, shown in figure 2.22.

Chapter 3

Hash Tables

3.1 Introduction

Hash Tables are a fundamental data structure that provides fast store and lookup operations, and it is used in various programming applications. The ability ,for example ,to create sets and quickly perform searches on them, depends on the efficiency of the underlying Hash Table. For this reason, many algorithms and approaches, both sequential and concurrent have been put forward, each with its own distinctive strengths and weaknesses.

The purpose of a Hash Table is to efficiently associate a given value with a key and use that key to rapidly store or search that value among other values. It usually consists of an array or list of buckets, which can hold one or more values, as well as a hash function that maps a value on the table.

Ideally, every different values will hash to different buckets, making it easy to insert and search them, However, it is improbable that there will be no collisions, in fact for a large number of operations we expect collisions to be quite common. For the purpose of avoiding collisions, one must consider the size of the hash table (more buckets means more ways to distribute the keys) and the hash function (a hash function that will evenly distribute keys among the available buckets will reduce collisions). Even so, collisions are unavoidable, and the way they are handled is one the most important factor among the various implementations.

3.1.1 Collision Resolution

Hash tables can be divided into two main categories: Closed Addressing and Open Addressing.

Closed addressing hash tables allow more than one values to be stored on the same bucket. This is usually implemented by attaching a linked list

at the start bucket , and each new value is added on the list. The length of the list must be kept bellow a constant number (called load factor) , to keep operations on the list fast. This way the average lookup / insertion time is dependent only on load factor. It is a common scenario to keep the list ordered , which reduces the average lookup time in half. This is an easy to implement data structure and it has been proven to perform well in practice.

Open addressing hash tables allow no more than a single value per bucket. When a collision is detected, the buckets are traversed in order to find an empty bucket to store the new value, even though it's hash value does not correspond to that bucket. The sequence according to which the buckets are traversed is typically:

1. Linear probing, where an empty bucket is searched within a given number of steps from the mapping bucket

2. Quadratic probing, where the buckets is searched at increasingly bigger intervals, according to successive values of a quadratic polynomial.

3. Double hashing, where interval is the outcome of a new hash function.

Some other important techniques used in resolving conflicts in open addressing hash tables is cuckoo hashing and hopscotch hashing. Cuckoo hashing, as explained later in detail, employs a second , or more hash functions. The value is first hashed using the first hash function and if it corresponds to an non-empty bucket the second hash function is used. If both buckets are empty, then one of the two previously hashed values is evicted and then hashed again using the other hash function, possibly triggering a series of evictions until all values are hashed.

Hopscotch hashing combines linear probing and cuckoo hashing. First , buckets are traversed until an empty bucket is found. If that bucket is in the neighborhood of the initially mapped bucket, the value is placed there, just as in linear probing. If the empty bucket is outside the neighborhood, values are moved in a sequence of hops, effectively moving the empty slot closer and closer to the neighborhood of the initial bucket. An example is shown at figure 3.1.

3.1.2 Resizing

Resizing is important to maintain constant average insertion and lookup time. In closed addressing algorithms, buckets may become too full, making their traversal slow, while in open addressing algorithms, the table may become too full to easily find empty buckets . In either case, the size of the

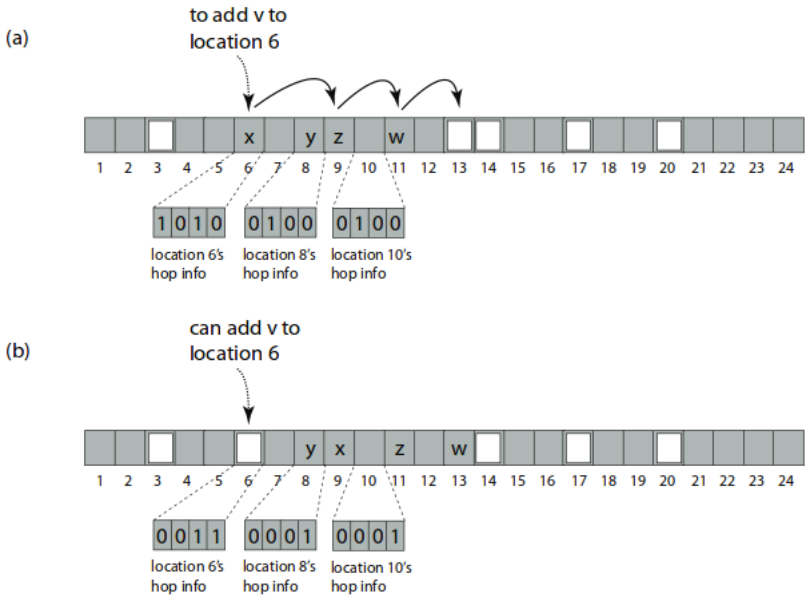


Figure 3.1: Example of an insertion in hopscotch hashing. Here we try to insert value 6 and the first empty bucket is found at index 13, outside the neighbourhood of bucket 6. We then find out that w an index 11 can be displaced to the empty bucket and we place it there. Now the empty bucket is at index 11 wich is still far from 6. Subsequently, by switching places with buckets 9 and 6, the empty bucket is finally at index 6 and the new value can be inserted

table needs to be expanded and all the values from the old, small hash table must be transferred to the new bigger one. This can be done by rehashing every value of the old table to the new one (possibly causing a high delay which may not be acceptable in a real time application) or incrementally , by moving every new inserted value to the new table , along with a few elements from the old table each time, until the old table is empty.

3.1.3 Concurrent hash tables

In a multiprocessor environment it is quite common for multiple threads to require concurrent access to the same hash table. Access on disjoin locations on the hash tables, suggests hash tables may allow a much higher level of concurrency, compared to FIFO queues as studied above. However, keeping the data structure fast and consistent despite contention, introduces many challenges and many diverse concurrent algorithms have been proposed to face them.

3.2 Locking Approaches

We start by implementing the hash table as a table of nodes, with each node being the head of a simple linked list that we keep ordered. We protect the table with a simple spinlock. Each thread needs to acquire the spinlock before performing any operation (insert, lookup or delete).

In order to ensure that every operation takes constant time, we need to keep the average number of items in every link list bellow a minimum. This is done by resizing the table , according to a certain policy. The resizing mechanism may trigger when the number of buckets that have grown beyond a certain threshold is large, or when the total ratio of inserted elements to the number of buckets exceeds a threshold. To perform the resize operation, a thread acquires the lock, checks if another thread has already resized the table and if not, proceeded to allocate a new table twice as big as the old one and then rehash every value from the old table into the new one.

The global lock mechanism is simple to implement but it introduces a sequential bottleneck and performance is low, as shown in figure 3.2. All threads spin on the same lock, even they are trying to access disjoint locations on the table . Moreover , since the critical section inside the lock is very small, the overhead of acquiring and releasing the lock becomes a large proportion of runtime.

We then try to permit more concurrency by used a fixed number of locks, instead of a single one. We introduce an array of spinlocks with length L,

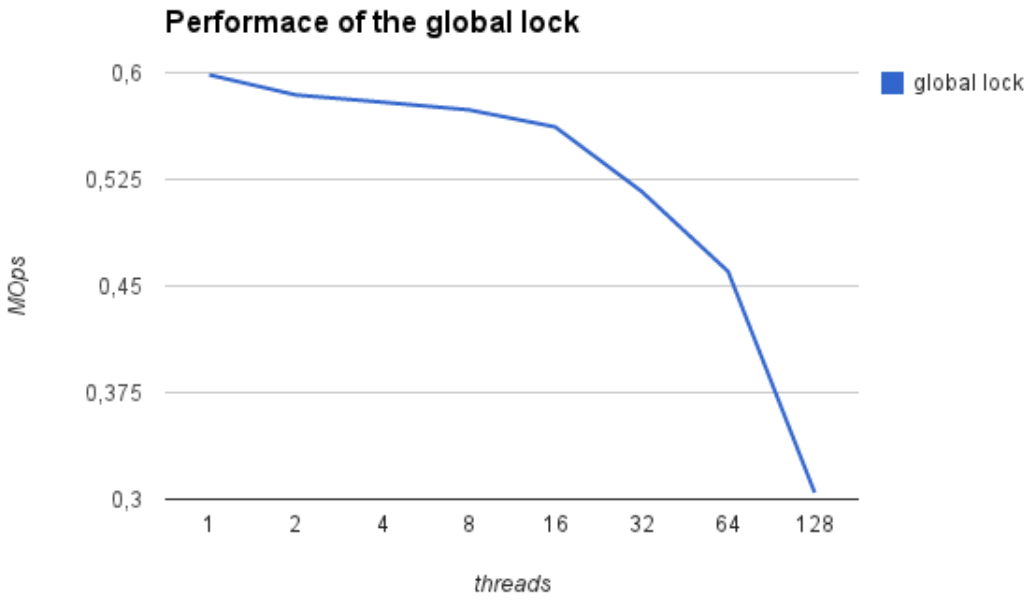


Figure 3.2: The performance of the naive global lock approach

with each lock protecting a number of buckets. We determine which lock is responsible for each bucket by simply mapping the index of the bucket on the array of locks. As the table grows, the number of locks remains the same, so each lock is responsible for more buckets, as shown in figure 3.3.

Having more than one locks means more than one threads can proceed successfully and this results to less contention, more concurrency and improved performance. Increasing the number of locks , closer and closer to the number of buckets, seems to improve throughput and reduce latency , as depicted in figures 3.4 and 3.5. However, if we want to keep the number of locks equal to the number of buckets, as the table increases in size, we need to be able to resize the lock array as well, which is not straightforward.

For this reason we then try to implement a refinable locking scheme, where locks can be dynamically increased to match the number of buckets. Here, we require mutual exclusion between resizing and updating, so we introduce a marked field owner, containing the id of the thread that is currently resizing. When a thread is trying to resize, it first atomically sets the mark bit and writes its id in the owner field. It then waits until no more updates are being performed(e.g. all the locks are unlocked). That way, all other threads that are trying to update, will find the marked bit of the owner value set

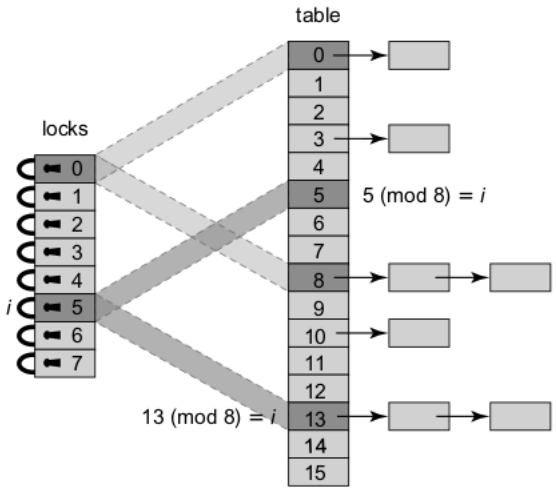


Figure 3.3: A representation of the striped hash set

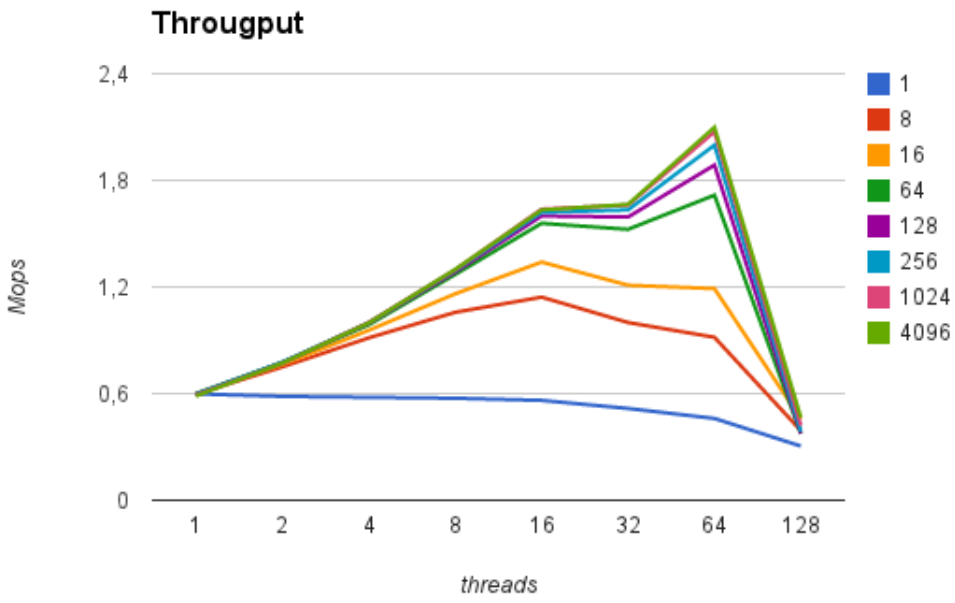


Figure 3.4: Throughput for various numbers of locks

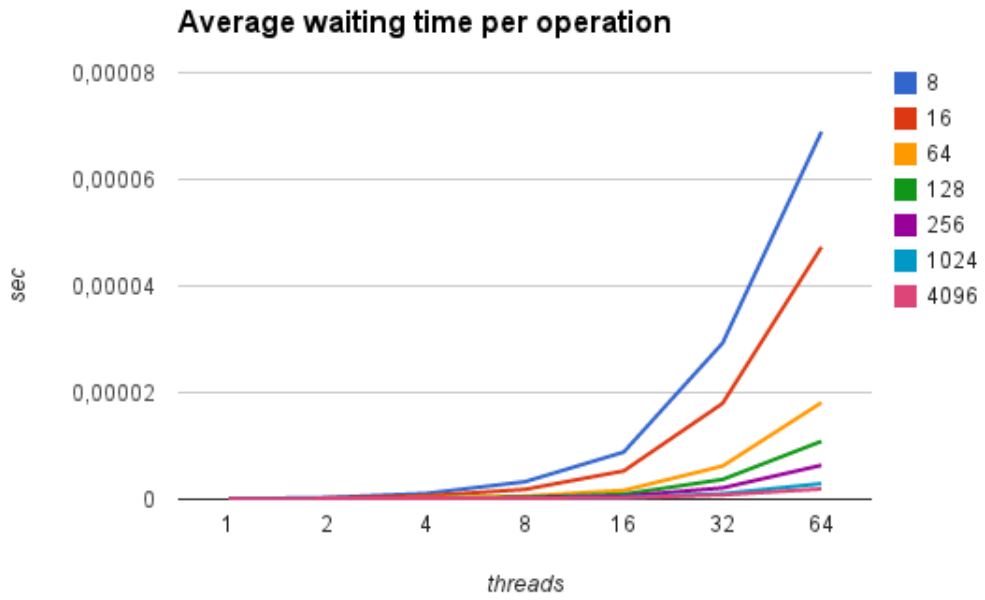


Figure 3.5: Latency for various numbers of locks

and will not proceed to take any lock. The thread is now free to resize the table and the lock array, rehash every value into the new table and unset the marked bit. In this implementation, we also allow lookups to first search the value without taking a lock. This allows to sometimes successfully perform lookups, even during resizing. If the value is found, lookup returns that value. If not, it might be possible that the value was inserted but the thread could not access it yet, so the thread tries searching again, this time holding the lock.

The outline of the function used by every thread to acquire a lock is shown here, where we can see how the when the mark bit of the owner field is set, no one can acquire any locks.

```
1
2  void acquire(HashTable_t T , int x){
3
4    me = current thread index
5    while (1){
6      do{
7        <who, mark> = owner
8      }while (mark && who != me)
9
10     Locks[] oldLocks = locks
```

Figure 3.6: The amount of time needed for every resize

```
11      lock the appropriate lock on oldLocks array
12      <who, mark> = owner
13      if ((!mark || (who == me)) && locks = oldLocks)
14        return
15      else
16        unlock the lock from oldLocks array
17    }
18  }
```

Listing 3.1: Acquiring the lock for the refinable hash table

The result is an increase in performance, because we exploit the finer grained synchronization. However, this improvement comes with the cost of slower resizing. Increasing the number of locks means that we need more time to allocate the extra locks and more time to wait until all locks are free. The extra amount of time introduced is shown in figure 3.6. In fact, figure 3.7 suggests that it would be better to stop resizing the locks array after a certain point.

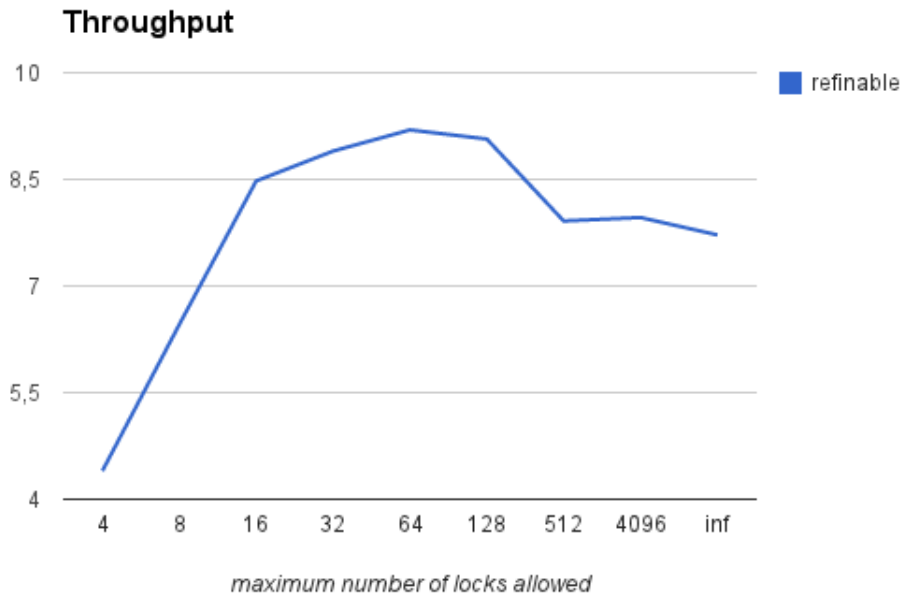


Figure 3.7: Setting a maximum granuality level affects performance

3.3 Split Ordered List

All the above mentioned algorithms use locks to enforce synchronization and therefore inherit all the disadvantages of blocking algorithms such as low performance when the number of threads exceeds the number of available cores. Moreover, these algorithms perform resizing in a "stop -the - world" manner, meaning that a single thread resizes the table and during that time no other thread can proceed. We now focus on a lock free algorithm by Shalev and Shavit [13] ,that uses atomic operations for synchronization and the hash table can grow incrementally without having to rehash any value or introduce a thread barrier.

The algorithm is based on a lock-free linked list , implemented by Michael [7] to store the values. The buckets, kept in a single array, are now references to specific nodes in the list, representing the start of the particular bucket, effectively working as short-cuts into the list. As the list grows, we introduce new buckets that split the older ones in half, keeping their size bellow a maximum value. In order to do this however, the list must be kept ordered according to a recursive split order, as described next.

Split ordering is in fact the reverse bit representation of values, kept in

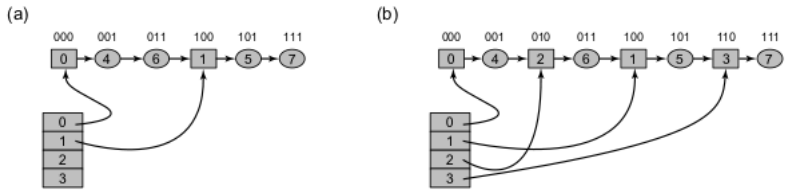


Figure 3.8: This figure explains how split ordering manages to effectively split a bucket in half. In part a , the list consists of two buckets. Above each node is it's split-ordered key, which is the reversed bit representation of each value. Square nodes are sentinel nodes, depicting the start of each bucket. When the table capacity grows from 2 to 4 in part b, two new buckets are inserted, splitting the older one in half

ascending order. The goal here is to split each bucket by inserting a new one, without having to rearrange anything else. Figure 3.8 explains how split ordering achieves just that.

There are some extra things to note here that illustrate why this specific type of ordering is the ideal for our need. Imagine a thread trying to search for the value 6 in the hash table depicted in part a. The capacity (the number of buckets is 2) so values 6 hashes to bucket zero. The thread starts traversing the list and while being at node value 4, a delay occurs. During that delay, capacity is double and a new bucket (bucket 2) is inserted between values 4 and 6, splitting bucket zero in half. As seen above, 2 is prior to 6 according to split ordering, and that mean that traversal well go past 2 and keep looking until it successfully finds the value 6. This depicts a key idea of the algorithm: when the table's capacity is doubled from 2^i to 2^{i+1}, a buckets b is split in half and those elements with values k for witch k mod 2^{i+1} =b remain in bucket b while the others migrate to bucket b + 2^i. The algorithm ensures that these two bucket are positioned one after an other, so that in order to split the bucket we don't really have to move any node around, but instead we only need to let the new bucket start after the first group of items and before the second.

To avoid the case where a node referenced by a bucket is deleted, we use sentinel nodes, inserted at the lists locations pointed to by every bucket. We use the Least Significant Bit of the reversed bit representation to differentiate between normal node and sentinel node and we do not de-allocate these sentinels node during deletion operation, in order to avoid corner cases.

In order to keep track of the load factor, we update a shared variable on every insert/delete operation that depicts the number of values currently stored in the table. If the ratio of inserted values to number of bucket ex-

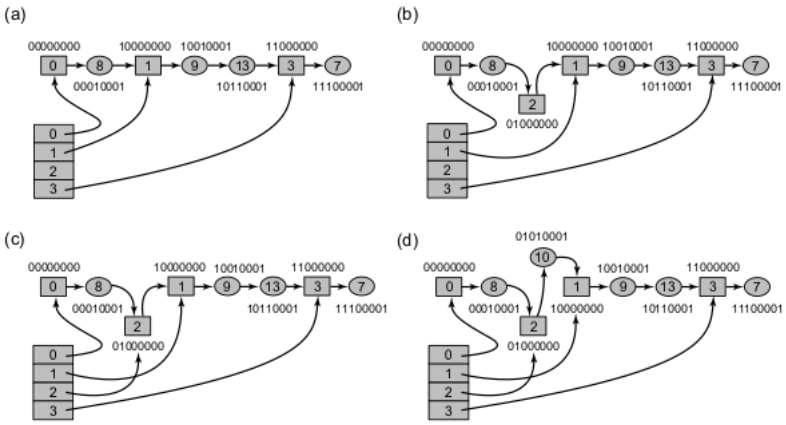


Figure 3.9: Example of an insertion in the split ordered list. At part a , the bucket 2 is uninitialized. Inserting value 10, initializes the bucket, inserting a sentinel node in the list, before the actuall insertion of 10 at part d

ceeds a certain limit, we double the capacity, thus introducing new available buckets. Note that, in order to avoid complexity, we allocate a large array of buckets in advance. By introducing an extra level of indirection, namely an array of array of buckets, we could be able to adaptively increase the maximum number of available buckets if needed.

Figure 3.9 is an example of how adding a new value in the hash table increases capacity and introduces new buckets.

Note that unused buckets are uninitialized. The sentinel nodes for each bucket are only inserted when it is actually needed, that is when this buckets holds at least one value. When initializing a bucket, we might also need to recursively initialize other buckets that come before it as shown in figure 3.10

The insert operations looks very simple, simply initializing the bucket if needed and then inserting the split-ordered key in the list starting from the node referenced by the appropriate bucket. Finally, we check if the capacity needs to be doubled. The same outline is followed by the lookup operation.

```
1
2  int insert(HashTable_t T, int key){
3    node =  allocate new node
4    node->key = split_ordered_representation(key)
5    bucket = key \% size
6    if T[bucket] == uninitialized
7      initialize(bucket)
8    if (! list_insert(&T[bucket], node)){
9      delete node
10     return 0
```

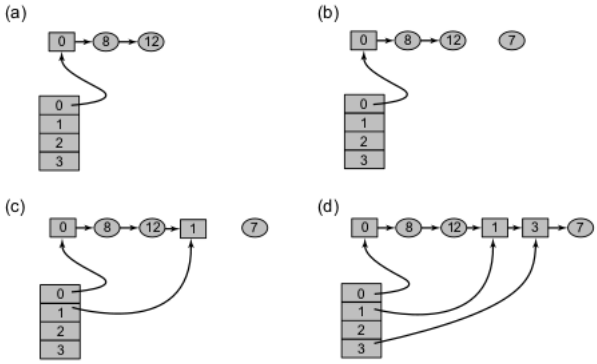


Figure 3.10: Example of recursive initialization of buckets, before insertion of value 7

```
11    }
12    csize = size
13    if (fetch_and_add(&count ,1) / cize > MAX_LOAD)
14      CAS(&size ,csize , 2 * csize)
15    return 1
16  }
17
18  int find(HashTable_t T, int key){
19    bucket = key \% size
20    if T[bucket] == uninitialized
21      initialize(bucket)
22    return list_find(&T[bucket], split_ordered_representation(key)
          )
23  }
```

Listing 3.2: Insert and Loopkup operations of the split ordered algorithm

At its lower level, this implementation utilizes a lock free list. Each thread hash a set of three private variables curr, prev and next, each consisting of a pointer to a node, along with a mark bit. These pointers traverse the list to find the appropriate locations for an insertion or deletion and then the thread attempts to atomically modify the list, using Compare And Swap operations

The result, shown in figure 3.11 is a lock free hash table that allows high concurrency, fast operations on the list and most importantly , does not require resizing and rehashing the entire hash table.

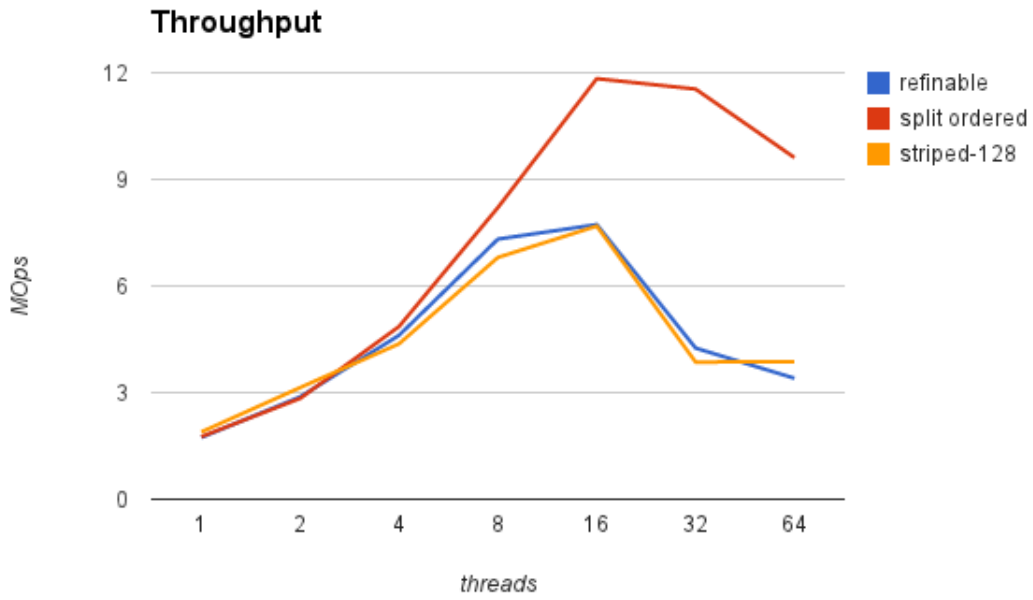


Figure 3.11: Performance of the split ordered algorithm

3.4 Cuckoo Hashing

3.4.1 Sequential version

We now turn our attention to an open-addressing hash table, called cuckoo hashing. Open-addressing algorithms allow for only one value to be stored at each bucket. In its sequential version, cuckoo hashing uses two arrays of buckets and two hash functions, although it is possible to use only a single array.

Lookup operations are quite simple. The value is hashed using the first hash function and the corresponding bucket on the first array is checked. If the value is not found, the second hash function is used and the second array is checked. If the value its not there either, the lookup operation returns that the value does not exist on the hash table.

The basic idea behind cuckoo hashing is better shown during the insertion operation. We first hash the value using the first hash function and insert it on the appropriate bucket. If that bucket was initially empty, the process is over. If not, the previous value stored in that bucket is evicted, and we then insert it on the other hash table, using the other hash function, possibly evicting another value in the process and so on, until an empty bucket is

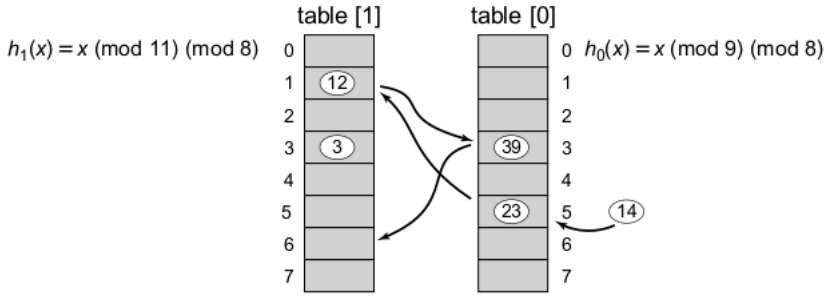


Figure 3.12: Insertion in cuckoo hashing. Trying to insert value 14, we find both apropriate bucket taken by values 3 and 23. Thus, a sequence of displacements is triggered, ending when value 39 is inserted to the previously empty bucket at Table[1][6]

found.

The chain of evictions and insertions can grow too long if either the table is too full and no empty buckets can be found easily, or the sequence of displacements form a circle, looping over the same buckets. For this reason, we set an upper limit on the number of evictions triggered by a single insertion. If that limit is exceeded, the table needs to be resized.

There are many variations of the sequential cuckoo hashing algorithms some using more than two different hash function. In general, sequential cuckoo hashing has been proven to work well in practice for a small to medium load factor.

3.4.2 Concurrent version

In order to implement a concurrent cuckoo hash table, a few changes were maid to the original sequential version. Instead of single element buckets, we use probe sets , whose size is not allowed to grow beyond a certain upper limit. Every set also has a threshold, which is the number of items that can be normally stored inside a bucket. The number of elements stored in a bucket at a given time may exceed the threshold, but the extra elements are marked for eviction and reallocation using the other hash function.

The main principle of the algorithm remains the same. During insertion, the value is added in the set and if the size of the set hash exceeded the threshold, we try to reallocate items using the other hash function and into the other hash table. In our implementation, we choose to evict the first item on each set, although several other strategies can be chosen instead.

In order to ensure synchronization, we choose to associate every set with

its own lock. During any operation (insert, lookup or delete) on a value x, we lock the sets with index h1(x) on table 1 and h2(x) on table 2 accordingly, always in this order, to avoid deadlocks. During resizing, a thread take all the locks on table 1 in ascending order, allocates two new tables, rehashes everything from the old tables to the new ones and releases the locks in the same order. Note that it is possible to implement a striped approach, as discussed in previous implementations, whereas there is a lock-free implementation of cuckoo hashing in the literature [11].

The basic outline of the insert function is shown bellow.

```
1  int insert(HashSet * table, int x)
2    acquire the locks on both buckets that x maps to
3    h0 = hash0(x)
4    h1 = hash1(x)
5    mustResize = false
6    search(x) if found return false
7    set0 = table[0][h0]
8    set1 = table[1][h1]
9    if set0.size < THRESHOLD {
10     add(set0, x)
11     return true
12   }
13   else if set1.size < THRESHOLD {
14     add(set1, x)
15     return true
16   }
17   else if set0.size < PROBE_SIZE {
18     add(set0, x)
19     i=0
20     h=h0
21   }
22   else if set1.size < PROBE_SIZE {
23     add(set1, x)
24     i=1
25     h=h1
26   }
27   else{
28     mustResize = true
29   }
30   if mustResize{
31     resize()
32     insert(T,x)
33   else if (! realocate(i,h)){
34     resize()
35   }
36   return true
```

Listing 3.3: Insert methods of the cuckoo algorithm

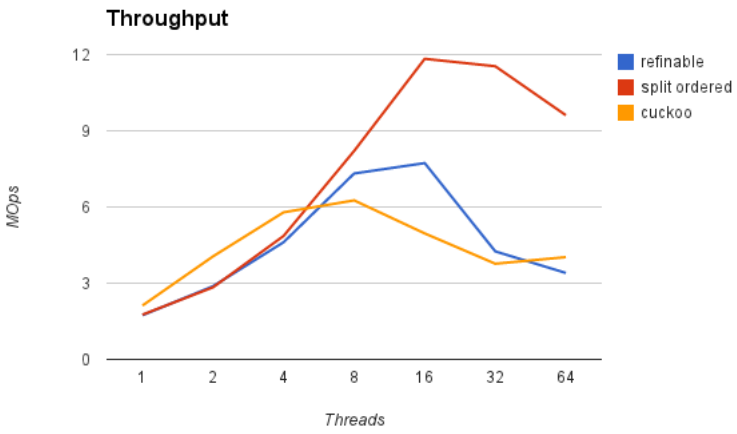


Figure 3.13: Performance of the cuckoo hashing algorithm

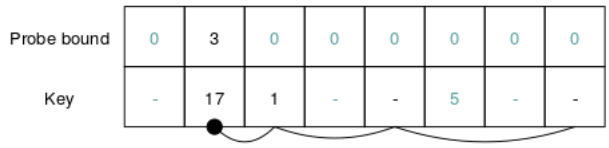
In the cuckoo algorithm, search is faster because only two buckets with limited number of items must be checked. However, cuckoo hashing usually requires more than half of the table to be empty in order to perform properly. Moreover, some insertions may cause a realocation chain to loop, forcing the table to resize early, meaning that we cannot easily keep the table from expanding without altering the hash functions. As a result we have bigger tables that are resized more often and during resize, memory allocation is undertaken by a single thread while all the others wait doing nothing.

3.5 Non Blocking open addressing

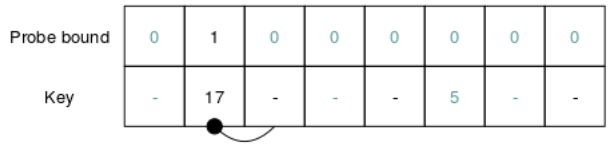
Last of all, we implemented another open addressing algorithm , introduced by Purchell and Haris [12] that, unlike cuckoo hashing, achieves non blocking access by using simple atomic operations.

This algorithm resolves conflicts by using quadratic probing, along the subsequent values of $\left[\frac{1}{2}\left(i^2+i\right)\right]$. In this implementation, every bucket is accompanied by a probe bound, a value depicting the number of collision on the probe sequence. For example, when searching for a value on a bucket with a probe bound of 4 , we need to take up to 4 steps in the quadratic sequence to search for that value. Keeping a consistent view state of the probe bound can be challenging when multiple threads are inserting or deleting on the chain of collisions, as seen in figure 3.14.

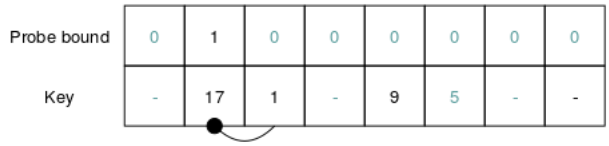
For this reason, we add a scanning bit for each probe bound, that we are able to update atomically, along with the probe bound. During insertion,



After a collision is removed, a thread scans for the previous collision.



If a concurrent erasure is missed, the bound may be left too large.



Worse, if a concurrent insertion is missed, the bound may be made too small.

Figure 3.14: Problems maintaining a shared bound, after a collision is removed from the probe sequence

threads just attempt to clear the bit and increase the bound if necessary. During deletions, a thread that is trying to erase a collision uses this bit to make sure that no other concurrent updates have been made and that the probe bound has decreased correctly.

Every individual bucket holds the value and a state variable that helps resolve conflicts and synchronize concurrent operations, as described later. In order to prevent the ABA problem, a version count is integrated along with each state variable.

In order to perform an insertion, a thread takes the following steps: First, it finds an empty bucket, stores the value and sets the state of that bucket to inserting. Then , it scans the probe sequence, looking for other threads inserting the same key, or a completed insertion of that key (characterized with the state member) . If a completed insertion is found, the thread sets its own working bucket to empty and the operation fails. If any other unfinished insertion operations are detected, the thread assists in the process of setting the first insertion found in the process to member and setting all the rest to collided. This operation is done concurrently by all threads working on the probe sequence, resulting in a lock –free algorithm where the delay of a single thread will not stall the others.

The main outline of the Insert and Assist operations are shown bellow.

```
1
2  bool Insert(int key){
3    h = Hash(k)
4    i=-1
5    do
6      if ++i>= size
7        table is full
8      <version , state> = Bucket(h,i)->vs
9    while (! CAS (&Bucket(h,i)->vs,<version , empty>,<version , busy
       >))
10   Bucket(h,i)->key=key
11   while (!) {
12     Bucket(h,i)->vs = <version , visible>
13     ConditionallyRaiseBound(h,j)
14     Bucket(h,i)->vs = <version , inserting>
15     r = Assist(k,h,i,version)
16     if Bucket(h,i).vs != <version , collided>
17       return true
18     if !r
19       ConditionallyLowerBound(h,i)
20       Bucket(h,i)->vs = <version+1, empty>
21       return false
22     version ++
23   }
```

```
24 }
25
26 bool Assist(int k, int h, int i , int ver_i){
27   max = GetProbeBound(h)
28   for (j=0; j<max ;j++){
29     if i!=j {
30       <ver_j,state_j> = Bucket(h,j)->vs
31       if state_j = inserting && Bucket(h,j)->key = k {
32         if j<i{
33           if Bucket(h,j)->vs = <ver_j,inserting){
34           CAS(&Bucket(h,i)->vs, <ver_i,inserting>, <ver_i,
                 collided>)
35           return Assist(k,h,j,ver_j)
36           }
37         }else{
38           if Bucket(h,i)->vs = <ver_i,inserting>
39             CAS(&Bucket(h,j)->vs, <ver_j,inserting> ,<ver_j,
                   collided>)
40         }
41       }
42       <ver_j, state_j> = Bucket(h,j)->vs
43       if state_j =  member && Bucket(h,j)->key = k{
44         if Bucket(h,j)->vs = <ver_j, member>{
45           CAS(&Bucket(h,i)->vs, <ver_i,inserting>,<ver_i,
                 collided>)
46           return false
47         }
48       }
49     }
50   CAS(&Bucket(h,i), <ver_i, inserting>, <ver_i, member>)
51   return true
52 }
```

Listing 3.4: Insert and Assist operation of the non blocking open addressing algorithm

Figure 3.15 is an example of how the algorithm detects and avoids collisions.

The impact of the ABA problem is better understood from the following example of bad synchronization between inserting threads, in figure 3.16 , that can be avoided with the use of version counters.

This open addressing algorithm also requires bigger, partially empty hash tables and resizing is again slow. However, if we know the approximate size of the values inserted in advance, or if the workload consists mainly of lookups while insertions are rare, this implementation may perform better than the others, mainly due to low it's small cache footprint that works very well with NUMA architectures. Figure 3.17 demonstrates that behavior.

Probe bound	0	2	0	0	1	0	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	empty	member
Key		9	1		17	12		7

Initial state

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	inserting	member
Key		9	1		17	12	12	7

Write key and raise probe sequence bound

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	collided	member
Key		9	1		17	12	12	7

Earlier 'inserting' entry found; move bucket into 'collided' state.

Probe bound	0	2	0	0	1	1	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	member	collided	member
Key		9	1		17	12	12	7

Assist completion of earlier entry

Probe bound	0	2	0	0	1	0	0	0
Version	18	2	3	6	4	3	25	7
State	empty	member	member	empty	member	member	empty	member
Key		9	1		17	12		7

Empty bucket, lower probe sequence bound and return `false`.

Figure 3.15: Inserting 12 using the lock-free algorithm.

State	empty	inserting
Key	-	12

A single thread is about to complete its insertion of key 12. The next step is to atomically move the cell from inserting to member state.

State	empty	member
Key	-	12

The thread is suspended, and its insertion assisted to completion by another thread.

State	member	inserting
Key	12	12

The key is now removed, and two other threads are concurrently attempting to reinsert key 12. One has just succeeded, and the other is about to remove itself. If the first thread wakes up at this point, it will still atomically move the cell from inserting to member state, duplicating key 12.

Figure 3.16: The effect of the ABA problem on concurrent assisting of operations.

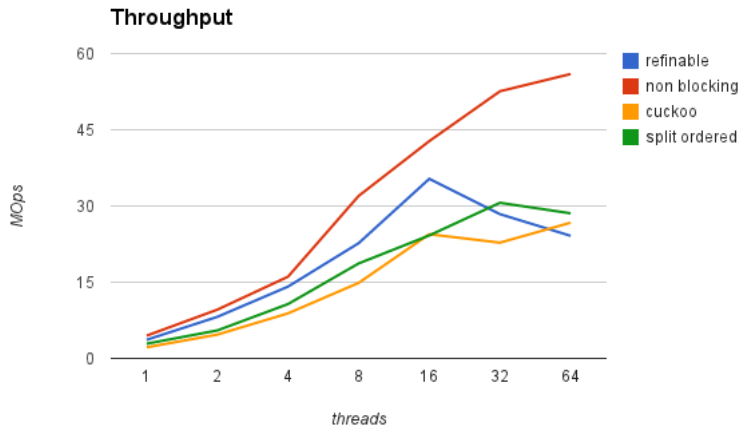


Figure 3.17: Performance of the non blocking open addressing implementation

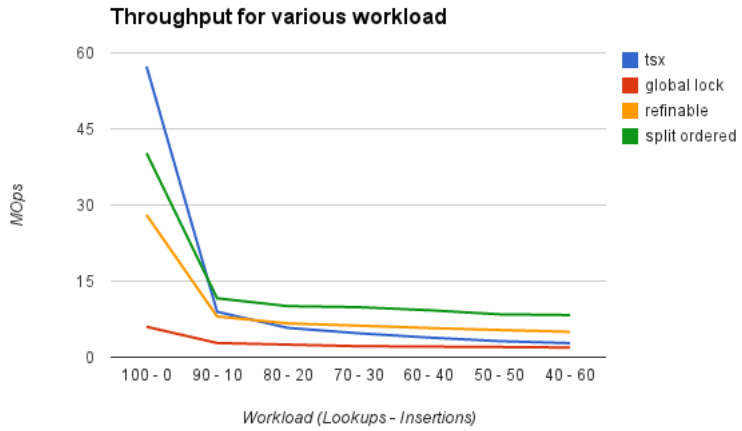


Figure 3.18: Performance of various implementations as the workload changes

3.6 Transactional memory

For completeness, we also present a simple implementation using hardware transactional memory on Intel's Hashwell multiprocessor. Operations on the table are executed inside a transaction. After a few consecutive aborts, a global lock is used as a fallback mechanism.

Figure 3.18 suggests that the transactional memory implementation perform very well, only when the workload consists only by lookups.In that case threads, dont modify memory locations and the absence of locking overhead leads to good performance. However, even a small percentage of insertions cause a significant amount of aborts and the advantage of transactional memory is quickly lost. In fact, as insertions become a bigger proportion of the workload, more and more transaction use the fallback mechanism and the algorithm gradually degrades down to the naive global lock approach.

Chapter 4

Conclusions

4.1 Synopsis

In this thesis we studied concurrent data structures, in particular FIFO queues and hash tables, two data structures with many inherent difference that reside on opposide ends of the spectrum of concurrency level.

FIFO queues proved to be inherently sequential data structures with little room for scalability. The simple locking aproach was inefficient, whereas lock - free approaches provided a more robust alternative. The study of flat combining suggested that the key to achieve performance is low synchronization overhead and high cache utilization.

On the other hand, hash tables allowed more threads to operate concurrently and scaled better. Performance was greatly affected by each algorithm, collision resolution policy, resizing mechanism and syncronization scheme. Locking and non blocking approaches were examined, each with it's own strengths and weaknesses, while transcational memory provided a worth mentioned alternative.

4.2 Future Work

There are still many interesting implementations of concurrent queues and hash tables to be studied, each with its own set of characteristics and innovations.

Regarding hash tables, it would be interesting to pursue an adaptive, dynamic approach where, the type of hash table used is determined by the workload and may change dynamically. Frequent resizes would suggest that a split ordered list may be more fitting, so during a resize, after we have stopped every operation, we can rehash everything to a split ordered list

and stop worrying about the cost of resizing. If the size stabilizes and the workload ends up consisting mainly of lookups, transferring the keys to an open addressing hash table would decrease lookup time.

Other than the two data structures we studied, there is a vast number and diversity of structures. Search trees, for example, are one of the most studied and frequently used and parallelizing them introduces many, new challenges.

Bibliography

[1] Daniel Cederman: *Concurrent Algorithms and Data Structures for Many-Core Processors.* Doktorsavhandlingar vid Chalmers tekniska högskola. Ny serie, no: 3184Technical report D - Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, no: 76. Department of Computer Science and Engineering, Networks and Systems, Distributed Computing and Systems (Chalmers), Chalmers University of Technology,, 2011, ISBN 978-91-7385-503-7.

[2] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir: *Flat combining and the synchronization-parallelism tradeoff.* In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0079-7.

[3] Maurice Herlihy and Nir Shavit: *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008, ISBN 0123705916, 9780123705914.

[4] Maurice Herlihy, Nir Shavit, and Moran Tzafrir: *Hopscotch hashing.* In Gadi Taubenfeld (editor): *Distributed Computing*, volume 5218 of *Lecture Notes in Computer Science*, pages 350–364. Springer Berlin Heidelberg, 2008, ISBN 978-3-540-87778-3.

[5] Alex Kogan and Erez Petrank: *Wait-free queues with multiple enqueuers and dequeuers.* In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0119-0.

[6] Edya Ladan-mozes and Nir Shavit: *An optimistic approach to lock-free fifo queues.* In *In Proceedings of the 18th International Symposium on Distributed Computing, LNCS 3274*, pages 117–131. Springer, 2004.

[7] Maged M. Michael: *High performance dynamic lock-free hash tables and list-based sets*. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM, ISBN 1-58113-529-7.

[8] Maged M. Michael and Michael L. Scott: *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM, ISBN 0-89791-800-2.

[9] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit: *Using elimination to implement scalable and lock-free fifo queues*. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM, ISBN 1-58113-986-1.

[10] Adam Morrison and Yehuda Afek: *Fast concurrent queues for x86 processors*. SIGPLAN Not., 48(8):103–112, February 2013, ISSN 0362-1340.

[11] Nhan Nguyen and P. Tsigas: *Lock-free cuckoo hashing*. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 627–636, June 2014.

[12] Chris Purcell and Tim Harris: *Non-blocking hashtables with open addressing*. In Pierre Fraigniaud (editor): *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin Heidelberg, 2005, ISBN 978-3-540-29163-3.

[13] Ori Shalev and Nir Shavit: *Split-ordered lists: Lock-free extensible hash tables*. J. ACM, 53(3):379–405, May 2006, ISSN 0004-5411.