12-15-2018

Efficient and Practical Composition of Lock-Free Data Structures

Neha Bajracharya
nehabajracharya@gmail.com

EFFICIENT AND PRACTICAL COMPOSITION OF LOCK-FREE DATA STRUCTURES

By

Neha Bajracharya

Bachelor's in Computer Engineering (B.E.)
Kathmandu Engineering College
2012

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2018

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 20, 2018

This thesis prepared by

Neha Bajracharya

entitled

Efficient and Practical Composition of Lock-Free Data Structures

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Ajoy K Datta, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

John Minor, Ph.D.
Examination Committee Member

Yoohwan Kim, Ph.D.
Examination Committee Member

Venkatesan Muthukumar , Ph.D.
Graduate College Faculty Representative

Abstract

A concurrent data object is lock-free if it guarantees that at least one, among all concurrent operations, finishes after a finite number of steps. In other words, a lock free technique guarantees that some thread always makes progress. Lock-free data objects offer several advantages over their blocking counterparts, such as being immune to deadlocks and priority inversion, and typically provide high scalability and performance, especially in shared memory multiprocessor architectures.

Composition of data structures is a powerful approach to combine simple data structures to create more complex ones. It works as a building block for many advanced useful data structures. In this thesis, we study the composition of lock-free data structures. We specifically consider a simple yet fundamental problem of combining two stacks to create a queue in a lock-free concurrent setting. Thus, we simulate the enqueue and dequeue operations of queues using the push and pop operations of stacks. Our work explores the challenges in combining the lock-free data structures with respect to ensuring the consistency as well as the lock-freedom of the composed data structure. We also implement the algorithm and experimentally evaluate its performance.

Acknowledgements

"Firstly, I would like to express my sincerest gratitude to my thesis advisor, Dr. Ajoy K. Datta for his continuous support and encouragement throughout my thesis work as well as my study here at UNLV.

I am very grateful to my committee members Dr. John Minor, Dr. Yoohwan Kim and Dr. Venkatesan Muthukumar and would like to thank them for providing their valuable time to serve in my thesis committee and for reviewing my work.

I would also like to express my deepest appreciation and gratitude to Dr. Bapi Chatterjee, Institute of Science and Technology Austria, previously with IBM India Research Lab, for taking time out of his busy schedule and imparting his knowledge and expertise in this study.

Most of all I am greatly indebted to my parents without whom I would not be here. Their love, support and trust have always encouraged me to pursue my dreams. And finally I would also like to thank all my friends and family, here in Vegas as well as those back at home for their continuous support, love and guidance."

Neha Bajracharya

University of Nevada, Las Vegas

December 2018

Table of Contents

List of Figures

Chapter 1

Introduction

Not so long ago, the central processing units (CPU) in the commonly available computers used to have only a single computing unit or compute core. With the development in computation, the need for faster processors was achieved by increasing the number of transistors on the processor which resulted in a higher clock speed. However, there was only a certain limit up to which the clock speed could be increased because the speed-up achieved by increasing clock speed also caused an increase in power dissipation and overheating of the transistors. To address this issue, the design of computers inevitably shifted to multi-core systems.

A multi-core processor comprises of two or more cores on a single chip. These multiple cores operate multiple instructions or programs in parallel resulting in better processing speed and greatly improves the overall performance. In addition to this, they have a lower clock frequency than the single core and thus produce lower power dissipation and are more energy efficient. Thus, the multi-core processors are overtaking single core machines nowadays as the world makes more advancement in technology and computation. The number of cores per computer is also rising rapidly. However, a significant challenge faced by multi-core processors is the need for synchronization due to the concurrency achieved by multiple cores working together to handle multiple tasks simultaneously. Since the multiple cores share the same memory, there may be interleaving of steps of the many threads working concurrently on each core. So there must be proper co-ordination among the threads to access the shared memory so that the computation produces the correct result. This can be achieved through a concurrent data-structure. A concurrent data-structure is a fundamental building block of concurrent programs and algorithms. These data structures are shared by concurrent processes and thus need to handle concurrent access to share memory. We can modify a sequential data-structure to create a concurrent data-structure that can work in a

concurrent environment. One of the simplest ways to make a data-structure concurrent is through mutual exclusion by means of a lock that provides the lock owner exclusive access to the shared memory [AS01]. This is also called the blocking method and it does not provide high scalability and performance due to deadlock, priority inversion and convoying. Another method that overcomes this disadvantage is the non-blocking method which guarantees progress of the operation even on a failure or suspension of a thread. Depending on the level of progress guarantee, a non-blocking data structure can be categorized as lock-free, obstruction-free or wait-free. Lock-freedom is achieved through the use of atomic primitives instructions available in the hardware machines such as compare-and-swap, test and set, load-linked/store-conditional.

1.1 Objective

The objective of our thesis is to study the composition of lock-free data structures. The term composing refers to the task of nesting or making one type of lock-free data-structure a part of another. In the literature, there are many non-blocking data structures like queue, stack, heap, binary search tree, etc. Using such a type of data-structure to create another lock-free and concurrent data-structure is a non-trivial problem.

We can easily compose sequential data structures, but it is a challenging task to compose concurrent lock-free data structures. The fact that the design and implementation of a lock-free data structure itself is complicated makes it difficult to trivially alter them in order to make them suitable to combine efficiently. As a result, in the current context, a general, efficient method for composing data structures in a concurrent and lock-free setting is still not available.

Therefore the main contribution of our work is to provide a base for the composition of data structures. It manages this while preserving the lock-freedom and progress guarantee of the data object. In our work, we explicitly consider a simple yet fundamental problem of combining two stacks to create a queue in a lock-free concurrent setting. In the composition of a queue using two stacks, we use the basic operations of the stack, i.e. its push and pop operations to implement the enqueue and dequeue operation of the queue. To ensure that the resulting data-structure provides a lock-free progress guarantee for their operation executions, we have used the non-blocking synchronization primitive compare-and-swap, which is provided by the multiprocessor system.

Study and research in this field of composition of data structures is limited, but it has begun nonetheless. Our work, therefore, contributes to providing a fundamental framework for similar algorithmic designs to develop advanced lock-free data structures using the existing basic ones. To

the extent of our knowledge, we believe that this is the first work which presents an algorithm and its implementation of the composition of lock-free data structures to obtain another lock-free data structure.

1.2 Outline

In Chapter 2, we will present some necessary concepts and basic background terminologies of the multi-core system required for this study. We will give a brief overview of the multi-core processor and its shared memory systems, synchronization methods as well as give a brief description of the composition of data structures.

In Chapter 3, we will give a brief survey of related work. Study and implementation works of concurrent data-structure are presented in this chapter.

In Chapter 4, we will present our proposed algorithm and its implementation of the composition of data structures. We will explain in detail the implementation of constructing a queue by using two stacks.

In Chapter 5, we will present the correctness of our algorithm in terms of progress guarantee and linearization.

In Chapter 6, we will present the experimental results on the performance comparison between various concurrent queues in terms of throughput and fairness.

Finally, in Chapter 7, we will conclude with an overall summary of our work and provide a scope of improvement and future recommendations to extend this study.

Chapter 2

Background

Initially, to obtain faster processors, the number of transistors was increased. The speed of the processor is determined by the clock frequency. The increase in the clock frequency of the processor started leading to an increase in high power dissipation and therefore caused overheating. After a certain limit was reached, the clock speed could not be exceeded beyond that point. As a result, a new design of processor known as a **"Multi-core processor"** was introduced by the manufacturers to address this issue.

2.1 Multi-core Processor

A multi-core processor can be defined as a single integrated circuit that consists of two or more independent computing/ processing units, commonly known as cores. These multiple cores located on the single chip operates simultaneously at a comparatively lower clock frequency to reproduce the same performance as that of a single processor. It is not necessary that the performance of individual cores on multi-core processor is as fast as the highest performing single-core processor. But since they are able to handle more tasks in parallel, they provide overall better performance with much less power dissipation. In a single-core processor, multiple processes are assigned different time-slices. If the time taken to complete a task by any process gets longer or they get delayed due to some reason, then this affects other processes as they start to lag behind too. However, in the instance of multi-core processors, each of the core has multiple processes assigned to it. So if the multiple cores works in parallel, the multiple tasks also run in parallel which will ultimately boost the performance.

2.2 Shared Memory System

In a multi-core processor, each core shares a main memory. Each core is associated with its own cache and they all share the same system bus. Such a system is known as a shared memory system. Such a system where different processors can simultaneously access the same main memory for the purpose of communication among them is know as a **Shared Memory System**. A shared memory system can be divided into:

2.2.1 Uniform Memory Access (UMA) Architecture

In UMA, every processor is located at equal distance from each shared memory. As a result, the memory access time is same all over the system, independent of the processor that request it.

2.2.2 Non Uniform Memory Access (NUMA) Architecture

In NUMA, the shared memory location may be at a different distance from the processors i.e. the distance is not uniform. Hence, the access time of the memory by different processors varies.

2.3 Synchronization

Data structures are an essential component of efficient and well-structured programs. "A data structure is a way to store and organize data in order to facilitate efficient access and modifications to the data" [CLRS09]. Access, insert, delete, find and sorting the data are some of the basic operations that one can perform using a data structure.

A thread, also called a lightweight process, is a sequential program and each thread runs at a different speed. Thus a thread is asynchronous and its delays are also unpredictable. These threads communicate with each other through the shared memory. Since threads are executed concurrently on different processors, and because of their asynchronous nature, the steps of different threads can interleave arbitrarily. So to properly coordinate the access to the shared data in order to produce a correct computational result, we need to synchronize the asynchronous concurrent processes.

The synchronization methods for a concurrent system are of two types. They are:

2.3.1 Blocking method

Blocking method of synchronization is the traditional way of synchronizing access to shared resources by the use of locking to achieve mutual exclusion. This method provides the lock owner

exclusive access to the shared resources. All the other threads will have to wait for the current thread holding the lock to finish its execution and release that lock. Monitors, mutex, semaphores, lock are some examples of locking primitives. A blocking synchronization is also called a lock-based implementation.

If a single lock is used for the entire shared resource, then it is called Coarse grained locking. Instead of using a single lock, if multiple locks are used then it is called Fined-grained locking.

Blocking synchronization method is simple, easier to design and implement. However, it suffers from many drawbacks. It does not tolerate process failure. If a thread holding a lock fails, then all other threads that are waiting for the lock get halted forever. Also, this method is vulnerable to deadlocks, priority inversion and convoying. Deadlock is a situation when there are two or more threads that get blocked while waiting to obtain locks which are being held by other threads in the deadlock. This prevents the threads from making progress. Priority inversion occurs when a lock is hold by a low priority thread and this causes a high priority thread not to be able to make progress. Convoying is a performance problem which occurs when there are multiple threads of equal priority that are repeatedly contending for the same lock. The thread that is holding the lock becomes inactive due to being scheduled out that results in other threads waiting for that lock to queue up and not be able to progress their operations. The performance of a blocking algorithm decreases significantly in an asynchronous multi-core processing system whenever there is a delay or suspension of a thread or process. Some causes for such delays are page faults, cache misses, processor scheduling preemption etc.

2.3.2 Non-Blocking Method

A non-blocking method allows all threads to access the shared resource without blocking the threads involved. In a more general term, if the suspension of one thread cannot lead to the suspension of other threads involved, then such a method is called non-blocking. There are three types of non-blocking method :

Lock-freedom

An implementation is said to be lock-free if it guarantees that some among its concurrent operations succeed to complete its operation after a finite number of steps. Lock-freedom ensures system-wide progress. However, it allows some threads to starve [Dan14]. It can tolerate any number of process failures, is deadlock-free and livelock-free, but is not starvation-free [Dan14].

Wait freedom

An implementation is wait-free if each process must complete an operation after taking a finite number of steps [Her93]. The wait-free condition guarantees that all non-halted processes make progress [Her93]. It ensures per-process guarantee and has the strongest progress guarantee [Dan14]. It can tolerate any number of process failures, is starvation-free, deadlock-free, and livelock-free [Dan14].

Obstruction freedom

An implementation is obstruction-free if a process completes the execution of an operation after it has executed in isolation a finite number of steps [Dan14]. It has the weakest non-blocking progress guarantee of the three. It ensures progress only in the absence of contention.

A lock-free data-structure can be implemented using atomic primitives that are made available by the hardware [Dan14].

2.3.3 Atomic Primitives

Atomic primitives, also known as hardware primitives, are machine instructions which can atomically access and modify one or more memory locations. Below are some commonly used atomic primitives:

Compare-and-Swap (CAS)

CAS takes three arguments: an address to a memory location, an expected value and an update value. It returns a boolean. When executed, CAS loads the value present in the address and compares that value with the expected value. If both the two values are found to be equal, then the value at the memory location is replaced by the update value and the CAS returns true. If the value read from the memory location and the expected value are different, then the memory is left unchanged and the operation returns false. Many architectures such as AMD, Intel, and Sun supports the compare-and-swap (CAS) atomic instruction [HS08].

Test and Set (TAS)

Test-and-Set is an instruction which is used to write a value to a memory address. It returns the old value in that memory. All these steps are executed as a single atomic operation. If multiple

processes can access the same memory location, then only one process can begin the test-and-set instruction. All other processes will have to wait until the first process has performed finishing the test-and-set instruction.

Load Linked/ Store Conditional (LL/SC)

This is a pair of instructions. The LL/SC instructions pair together is used for implementing an atomic Read/Write. The load-linked (LL) instruction first loads the value from a memory location. Then the store-conditional (SC) will check if the value in that memory address has changed or not since the LL instruction was last issued by that thread. If the value is still the same, the SC will store a new value at that address. Otherwise, it fails.

2.4 Concurrency and Correctness

The safety and liveness properties are used for defining the behavior of concurrent objects. They are also referred as correctness and progress respectively [HS08].

The key to the correctness property of a concurrent object is to somehow map their concurrent executions to their sequential ones, and limit our reasoning to these sequential executions [HS08]. According to the different mappings , there are three correctness conditions : *Quiescent consistency*, *Sequential consistency* and *Linearizability*.

In case of the liveness or progress property, it is of two types: blocking, in which the delay caused by a thread can delay other threads; and non-blocking in which the delay caused by any thread does not affect the progress of other threads.

2.4.1 Quiescent consistency

We can define Quiescent consistency by the following two principles [HS08] :

1. The method calls should appear to happen in a one-at-a-time, sequential order.
2. The method calls separated by a period of quiescence should appear to take effect in their real-time order. An object is quiescent if it has no pending method calls.

It provides high-performance at the cost of weaker constraints on object behavior.

2.4.2 Sequential consistency

This property requires that the result of any execution be the same as if the operations of all the processors were executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program [Lam79]. It is useful for describing low-level systems such as hardware memory interfaces [Lam79].

2.4.3 Linearizability

This property states that each method call should appear to take effect instantaneously at some moment between its invocation and response [HS08]. That is, there must be the preservation of the real-time behavior of the method call. Every linearizable execution is sequentially consistent, but not vice versa [HS08]. Linearizability is the strongest property among the three and is useful for describing higher level systems composed from linearizable components [HS08].

Formal Definition

An execution of a concurrent system is modeled by a *history* which is defined as a finite sequence of method invocation and response events. A *response* matches an invocation if they have the same object and thread. A *method call* in a history H is a pair consisting of an invocation and the next matching response in H.

In some histories, method calls do not overlap: A history $\boldsymbol{H}$ is *sequential* if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response.

The basic idea behind linearizability is that every concurrent history is equivalent, in the following sense, to some sequential history. The basic rule is that if one method call precedes another, then the earlier call must have taken effect before the later call. By contrast, if two method calls overlap, then their order is ambiguous, and we are free to order them in any convenient way.

We can show that the concurrent implementation of an object is linearizable by identifying the linearization point of each method. The linearization point is the time-point between invocation and response where the method effect is considered to take place. For lock-based implementation, the critical section of each method can be considered as its linearization point. For lock-free implementation, the linearization point is the single step where the effect of the method call becomes visible to other method calls [HS08].

2.5 Composition

The term composing refers to the task of combining concurrent data structures to create another type of concurrent one. It can also refer to the task of combining operations of multiple concurrent data structures to create a new one that can be performed atomically. Example of the composition of operations of data structures is combining the remove operation of one data structure with the insert operation of another to create a move operation. However, in our context, we will be dealing with the first definition of composability only. Composability is an important aspect of object oriented programming and a much desired quality for software artifacts.

Composing concurrent data structures:

There are many basic lock-free data structures like linked-list, queue, stack, tree etc. Each such data-structure has their own equivalent implementation of a basic set of operations like insert, delete, find etc. We can create a new advanced type of lock-free data-structure using these existing lock-free data structures. This is called as the Composition of data-structure. We achieve this by using the basic operations of the composing components.

To compose a lock-free Queue data-structure using two lock-free stack data structures and composing lock-free Stack data-structure using two lock-free queue data structures are fundamental yet non-trivial example of data-structure composition. In such types of composition, we use the enqueue and dequeue operation of the queue to compose push and pop operations of the stack and vice versa.

To design and implement a concurrent data-structure is an arduous task, but due to the growing popularity of multi-core systems writing them have become of practical importance. Since it is already a challenging task to design a single data-structure, composing a concurrent data-structure using multiple data structures, is even more difficult and has been an open problem in the area of multi-core processing.

Chapter 3

Literature Review

3.1 General Nonblocking Methodologies

Herlihy has presented in his paper [Her93] a general methodology for constructing a lock-free or wait-free implementations of concurrent objects. In this methodology, a given sequential implementation of an object is transformed into a non-blocking concurrent implementation by the use of synchronization and memory management algorithm. Synchronization is implemented using compare-and-swap or load-linked/store-conditional atomic primitives. The basic technique is as follows: the shared data that needs to be modified is first read and copied into the cache. Any updates are made to this copy. Then the updated copy is used to replace the shared data by means of atomic primitive operation compare-and-swap (CAS) of load-linked/store-conditional (LL/SC) which ensures that linearizability. This methodology suffered from two drawbacks: the overhead caused by the copying of large shared data for each operation and the ABA problem. To address these issue, Herlihy also proposed an optimization. The data-structure is made up of blocks consisting of pointers. To reduce the overhead of copying large shared data, optimization approach suggests to only copy only those blocks that require to be modified or those blocks with pointers that are pointing to the blocks that need to be modified. This removed the necessity to copy the entire data. Yet still the problems of the need of lots of copying persist as well as the need to incorporate logic for breaking the structure into proper blocks.

Barnes [Bar93] has also presented a methodology to generate a lock-free caching algorithm for any data-structure. He presented a technique in which exclusive access to shared data can be achieved without using mutual exclusion but through the caching method that uses co-operative technique where any modifications to the shared data are recorded and time-stamped. Whenever

any conflict arises, the processes co-operate to resolve that conflict.

3.2 Non-blocking methodologies for Queue and Stack

Treiber [Tre86] has presented a simple and efficient non-blocking algorithm of a stack data-structure. The proposed stack is a singly-linked list which consists of a *Top* pointer. The atomic primitive instruction compare-and-swap is used to modify the top pointer in an atomic way. The paper does not provide any results on the performance of the stack. However, the algorithm is widely considered to be very simple and is expected to work very efficiently as well.

However, Michael and Scott in [MS98] have shown the performance results comparing implementation of four different stacks. The four stack implementation were the common single lock algorithm using ordinary and preemption safe locks, Triebers non-blocking stack algorithm and an optimized non-blocking stack algorithm based on Herlihys general methodology [MS98]. The result from the experimentation showed that Treibers stack performed the best among the four stacks, even the system. Was a dedicated one.

Michael and Scott have presented two simple and practical algorithms for concurrent queue. One is a non-blocking queue and the other is a blocking queue that uses a pair of locks [MS98].

The first algorithm is a lock-free queue algorithm. The queue is represented as a singly-linked list. It consist of two pointers: *Head* and *Tail*. Head is a dummy node and it is made to point to the first node in the list [MMM96]. The last or second to last node in the list is pointed to by the Tail [MMM96]. During the enqueue operation, the nodes are only inserted at the tail i.e. at the rear end of the linked-list. Nodes are always deleted from the head during the dequeue operation. This algorithm uses Compare-and-Swap atomic instruction. It also make use of a modification counter in order to avoid the problems of ABA. This algorithm ensures that the values from various pointers are consistent by reading the values in sequences to rechecks the previous values are still the same.

The other algorithm that Michael and Scott have presented is a two-lock queue algorithm. This queue only allows one enqueue and one dequeue that proceed concurrently [MS98]. The queue consist of a separate Head lock and a separate Tail lock. The Head points to a dummy node which is always at the beginning of the linked-list. The main purpose of the dummy node is to avoid the deadlock problem that may be caused by different processes trying to acquire locks in different order. The dummy node ensures that the Head is never accessed by the enqueuers and the Tail is never accessed by the dequeuers.

The performance results on comparing different blocking and non-blocking implementation of

queues showed that the non-blocking algorithm gave better performance. Moreover, their non-blocking queue implementation yielded the best performance; hence the Michael and Scott queue is a very popular queue and widely used in multiprocessors that support universal atomic primitives. It is generally regarded as the algorithm of choice for a concurrent non-blocking queue [NND11].

3.3 Composition of lock-free data structures

Cohen and Cambell [CC92] were the first to study about composite data structures. Composite data structures are such data structures that have been constructed by nesting or composing multiple basic data structures.

Cederman and Tsigas have presented a lock-free methodology for composing together highly concurrent linearizable objects by unifying their linearization points [DC13]. In this methodology the operations of different types of concurrent lock-free objects are composed together to produce move operations that are atomic in nature and is able to move elements between these objects. A lock-free data object is first checked to see if it fulfill a set of properties or requirements given by the methodology. If the requirements are fulfilled they are considered to be a move candidate. The insert and remove operation, or an equivalent set of operations, of the move candidates are then used to compose a new atomic operation. This is done by combining their linearization points which is defined as the time instant when an operation becomes visible to all other threads in the system. Atomic primitive compare-and-swap i.e. CAS is the linearization point for many concurrent lock-free data objects. Hence we can combine the CASs of the insert and delete operations of the data objects such that they perform simultaneously, thus resulting in a combined move operation.

Nguyen and Tsigas have presented in [NND11] a new synchronization mechanism that guarantees the shared lock-free data-structure composed of other lock- free data structures progress. They first give the observation that lock-free data objects are linearizable and hence can be composed, but they do not provide progress guarantee. To remedy this, they proposed their synchronization mechanism that act as an interface between calls from various lock-free objects. This mechanism guarantees that every shared lock-free data objects achieve progress. The proposed mechanism works by means of keeping track of all invocations made by the sharing object to the shared object [NND11]. The invocations that fail to execute many times are announced such that later invocation will help to first complete this announced operation before performing their own operation. In this way, the sharing objects are able to progress.

Chapter 4

Implementation

Queues and Stacks are one of the most commonly used data structures that allow us to dynamically store and retrieve large collection of data items.

A queue is based on the First-In-First-Out or FIFO principle, which means that the elements of a queue are removed in exactly the same order as they were inserted or added into the queue. Inserting an element to the queue is known as an Enqueue operation. Removing an element from the queue is know as a Dequeue operation. Similarly, a stack is implemented using the Last-In-First-Out principle. An element can be added or removed only from the top of stack. The delete operation of a stack is called a Pop while the insert operation is called a Push.

In this thesis, we have designed and implemented a lock-free Queue. For the composition of a Queue data-structure, we have used two lock-free stacks by Trieber [Tre86].

4.1 Queue using two stacks

A queue can be implemented using two stacks. A Queue using two stacks is a fundamental example of data-structure composition. It is a well-known concept in sequential data-structure. However, its equivalent concurrent algorithm does not exist in the literature so it is considered to be a fundamental non-trivial problem in the area of multi-core technology.

The sequential algorithm for it is given below:

1. For enqueue operation, push the element into stack1
2. For dequeue operation, if both the stacks are empty then it indicates an empty queue. If stack2 is empty, then all elements from stack1 are pushed into stack2 which reverses the order of elements in stack2. Then the top element is removed from stack2.

Listing 4.1: Sequential Algorithm to implement queue using two stacks

```
enqueue(q,x)
1) Push x to stack1

dequeue(q)
1) If both stacks are empty, then output   Empty Queue
2) If stack2 is empty
        While stack1 is not empty, push all elements from stack1 to stack2
3) Pop the element from stack2 and return it
```

We simulate the above sequential algorithm in a non-blocking manner to design our proposed queue. The Queue uses two stacks- stack1 and stack2. These stacks are shared by all the threads of the system. The stack used for the composition is the scalable lock-free stack introduced by Treiber [Tre86].

The stack is represented as a singly-linked list. The top node of the stack is initialized to a sentinel node. The value at the top node of the stack is modified using the atomic primitive compare-and-swap. The push operation of the stack is used to design the enqueue method of the queue while the pop operation of stack is used to design the dequeue method of the queue.

The push and pop operations of Treiber's stack is as follows:

Algorithm 1 Implementation of push operation on a Stack

```
1: function PUSH(item)
2:     StackNode newStackNode = new StackNode(item);
3:     StackNode oldTopNode ;
4:     while !top.compareAndSet(oldTopNode, newStackNode) do
5:         oldTopNode = top.get();
6:         if oldTopNode.isMarker() then
7:             return oldTopNode;
8:         newStackNode.next.set(oldTopNode);
9:     return null
```

The push operation as shown in Algorithm 1 takes an item to be inserted into the stack and returns a node. The only modification made to the push operation of Treiber's stack for the proposed Queue using two stacks algorithm is at line 6 of Algorithm 1. Here the top node of the stack is first checked whether it is a *Marker* class node. If the top node of the stack is a Marker class node, it will return that node without pushing the item into the stack. This is because a marker node on top of the stack means that push cannot be done right now because currently the stack is being used for a dequeue method by some other process. If it is not a marker node, then the push operation is carried out. After a successful push, we return null.

Here the return value signifies whether the item was successfully pushed into the stack and hence successfully enqueued in the queue. If the return value is null, it means enqueue in the queue is successful, otherwise enqueue cannot occur right now as some other thread is in the middle of its dequeue process. If the return value is not null, then the thread calling the enqueue will first help the unfinished dequeue of some other thread, then proceed towards its own enqueue operation.

Algorithm 2 Implementation of pop operation on a Stack

```
1: function POP
2:     StackNode oldTop, nextTop;
3:     while !top.compareAndSet(oldTop, nextTop) do
4:         oldTop = top.get();
5:         if oldTop.isMarkedStackNode() || oldTop.equals(sentinel) then
6:             return oldTop;
7:         nextTop = oldTop.next.get();
8:     return oldTop
```

The pop operation of Treiber's stack is shown in Algorithm 2 The pop operation takes no arguments. But it returns a node value. The modification to the pop operation of Treiber's stack is made is line 5 of Algorithm 2. The top node of the stack is checked if it is either a marked node or if it is empty. Empty top node means empty stack. A marked node at the top indicates there is another dequeue operation in progress which first needs to be completed. If either of these case is true, then the top node is returned without removing any node from the stack. So the current thread will first take necessary steps to complete the pending operation before continuing its own operation. If both cases are false, then the top node of the stack is returned after removing it from the stack i.e. the node is popped successfully.

4.1.1 Operations of Queue

ENQUEUE

The enqueue operation of the Queue takes a value and returns a void. We always perform enqueue on stack1. As shown in Algorithm 3 the enqueue operation first calls the push operation of stack1. If the push operation returns a null value, then enqueue of value into the queue is successful and the enqueue operation returns. If the push operation returns a node, it means that the top node of stack1 is a marked node. Marking a top node of stack indicates that the dequeue operation of the queue is in progress. As a result, the enqueue operation calls a shift operation to help finish the ongoing dequeue of some other thread. After the shift completes, enqueue loops back and calls

the push operation to continue with its own operation. Hence, enqueue returns only when a value is pushed successfully into stack1.

Algorithm 3 Implementation of enqueue operation on a queue

```
1: function ENQUEUE(value)
2:     StackNode s;
3:     while true do
4:         s = s1.push(value);
5:         if s == null then
6:             return
7:         else
8:             shift((Marker) s);
```

DEQUEUE

The dequeue operation of the Queue takes no arguments and returns a void. As we can see from Algorithm 4 the dequeue occurs from stack2. The dequeue operation first calls the pop operation of stack2 which returns the top node of the stack. The returned node can be of three types : a sentinel node , a marked node or simply the top node.

If the returned node from the pop operation is a sentinel node, it means that stack2 is empty. An announce operation is then called, which marks the top node of stack2. If the return from the announce is null, it implies that the stack is not empty and not marked so it is suitable to dequeue an item from it. So the current thread will loop back to call the pop operation and try to remove the top item from that stack. If the announce operation returns a marked node, it can mean either that the announce was successful or the top node of stack was already marked. In both the cases, the dequeue operation calls the help operation. A boolean true value is returned from help operation if stack1 is also empty. Since this help was called after stack2 was found empty and it returns true indicating stack1 is also empty, it means that the queue is empty. The top of stack2 node is rechecked to see if it is still the same before the dequeue operation returns. If the help operation returns false, it indicates that the transfer of nodes from stack1 to stack2 is completed. After the successful help, the marked node has also been removed from top of stack2. So the current dequeue invocation will loop back again to call the pop operation and try to remove the top node of stack2 , hereby completing the dequeue operation of the queue.

If the returned node from the pop operation is a marked node, it means that a dequeue was already in progress so a help operation is called. If the help returns boolean value true, it implies that the stack1 is empty. Since the stack1 contains all the enqueued items, an empty stack1

indicates an empty queue. Stack2 is rechecked again to see if it is still the same. If there has been no change to its state, the dequeue operation returns. If help returns false, then it implies that all the nodes from stack1 have been transferred to stack2 and the marked node has also been removed from stack2. The dequeue then loops back calling the pop operation of stack2 to remove the top node from the stack.

If the node returned is a simple stack node, it implies that the top node of stack2 was successfully removed. This, therefore, completes the dequeue operation of the queue.

Algorithm 4 Implementation of dequeue operation on a queue

```
 1: function DEQUEUE
 2:     StackNode tops2;
 3:     while true do
 4:         tops2 = s2.pop();
 5:         if tops2 == s2.sentinel then
 6:             StackNode s = announce(tops2);
 7:             if s != null then
 8:                 if help(s) then
 9:                     if s2.get.top() == s then
10:                         return
11:         else if tops2.isMarkedStackNode() then
12:             if help(tops2) then
13:                 if s2.get.top() == tops2 then
14:                     return
15:         else
16:             return
```

ANNOUNCE

The announce method takes a node as its argument and returns a node as shown in Algorithm 5. The announce operation basically announces that the stack is going to be used for a dequeue. The announce operation puts a marked node on top of the stack if that stack is empty. A *Marked* node is simply a node containing the $Integer.MAXVALUE$ as its value with its next pointer pointing towards the top node of the stack. So a marked node at the top of a stack indicates that a dequeue is in progress. Any thread that encounters a marked node on a stack will know that the stack is being used by some other thread for their dequeue operation so it will first take the necessary steps to complete the pending dequeue before performing its expected operation. This helping mechanism ensures that any operation in progress is first completed without using any blocking mechanism. If any thread encounters that the stack is no longer empty in the announce method

will return null from that operation. If the stack is already marked, the operation returns that marked node.

Algorithm 5 Implementation of announce operation on a Queue

```
1: function ANNOUNCE(s)
2:     StackNode tops2;
3:     while true do
4:         tops2 = s2.top.get();
5:         if tops2.isMarkedStackNode() then
6:             return oldTopNode;
7:         else if tops2.equals(s2.sentinel) then
8:              StackNode marker = new StackNode(Integer.MAX_VALUE,tops2);
9:             if s2.top.compareAndSet(tops2, marker) then
10:                return marker
11:        else
12:            return null
```

HELP

The help operation takes a node as an argument and returns a boolean value as shown in Algorithm 6. In help operation, the process of transferring nodes from stack1 to stack2 is performed until stack1 becomes empty. After the transfer of nodes completes, the nodes in stack2 will be reversed to that of stack1. This ensures that the earliest node inserted into a queue is at the top of the stack2 to be removed, thus maintaining the FIFO discipline of the queue data-structure.

The help operation first reads the top node of the stack1.

If it is empty, the help operation returns true.

If the top node of the stack1 is not empty, but is found to be marked, it calls the shift operation to move that node of stack1 to stack2. A marker node on top of the stack1 indicates that a transfer of a node is ongoing.

If the top node of the stack1 is not empty and is not marked, it will first be marked to begin the process of transfer of nodes between the two stacks. A *Marker* node is an object of class Marker that extends the stacknode class and it contains all the necessary information to complete the transfer of the top node from stack1 to stack2. It contains the current top node of stack1, the current next node to the top of stack1 and the current top node of stack2. This information helps any other thread that encounters a marker node to continue with the dequeue process thereby providing a progress guarantee. After the top node of stack1 is successfully marked, a shift operation is called to transfer that node to stack2. When the shift method returns, if the top node of stack2

is still marked, it implies that the stack1 is not empty so the help method loops back to continue the transfer process. If the top node of stack2 is not found to be marked after a successful shift operation, the help method terminates by returning boolean false.

Therefore, the help returns false after all nodes of stack1 has been pushed to stack2. The marked node at stack2 and the marker node at stack1 are also removed by the end of the help operation. The successful completion of help allows for the removal of the top node from stack2, thus completing the dequeue operation of the queue.

Algorithm 6 Implementation of help operation on a Queue

```
1: function HELP(markedNode)
2:     StackNode tops1;
3:     while true do
4:         tops1 = s1.top.get();
5:         if tops1 == s1.sentinel then
6:             return true
7:         else if topS1.isMarker() then
8:             shift((Marker) topS1);
9:         else
10:            StackNode next = topS1.next.get();
11:            Marker mark = new Marker(topS1, markedNode, next);
12:            if s1.top.compareAndSet(topS1, mark) then
13:                shift(mark);
14:        markedNode = s2.top.get();
15:        if !markedNode.isMarkedStackNode() then
16:            return false
```

SHIFT

The shift operation is performed as shown in Algorithm 7. The shift operation removes the top node of stack1 and inserts it into the top of stack2. We need to make sure that the removal of a node from stack1 is always followed by the insertion of that node to stack2. Also, at the end of a node transfer, we need to maintain markers at top of both the stacks. For this, we first put a temporary marker node on the top node of stack1 before the actual transfer of node is performed. The temporary marker node will contain the marked nodes to be placed on each stack top to complete the transfer of nodes i.e. the top on stack1 will need to point to the next node of the current top of stack1 which implies the current top being removed and the top of stack2 will need to point to the current top of stack1 which implies that the stack1 node is inserted into stack2.

To begin the transfer of the node, we first create a new marked node that is going to be inserted into the top of stack2. This node has its next pointer pointing the node that is about to be removed from stack1. In the next step, we create a new Marker node to be inserted into the top of stack1. Its next pointer is pointing to the node that follows the node being removed from stack1. This new marker class node contains all the necessary information required to perform the next transfer. Then we create a temporary marker class node to be placed on top of the stack1. It has $Integer.MINVALUE$ as its value with its next pointer pointing to the current top node of stack1. This temporary marker node contains the two marked nodes previously created that is to be placed on each stack top during the transfer of node. We use the atomic primitive instruction CAS to insert the temporary marker to the stack1 top. After the successful CAS, any thread that encounters a temporary marker node on the stack1 will be able to proceed with transferring the correct nodes in order.

For the actual transfer of node, we perform three successive CASs as shown lines 9,10 and 11 in Algorithm 8.

The first CAS of line 9 is performed on the next pointer of the node to be removed from stack1. A successful CAS will point this next pointer to the actual top node of stack2 i.e. the node that follows the marked node of stack2.

The second CAS of line 10 is performed on the top pointer of stack2. In the second CAS, the top pointer of stack2 is replaced with the new marked node that was previously created and which is now obtained from the temporary marker node of stack1. This new marked node has its next pointer pointing to the node of stack1 that is to be transferred to stack2.

The final CAS of line 11 is performed on the top pointer of stack1. In the third CAS, the top pointer of stack1 is updated with the new marker class node created previously and obtained from the temporary marker node of stack1. The next pointer of this new marker node is pointing to the node that follows the node being shifted from stack1 to stack2 i.e. this is the node that contains the value that is going to be at the top of stack1.

In the cleanTemporary method as shown in Algorithm 8 the CAS operation will replace the top node of stack1 with a marked node for stack1 from the temporary marker. Similarly, the CAS operation will replace the top of stack2 with a marked node for stack2 from the temporary marker. Thus after the third CAS succeeds, the transfer of the top node from stack1 to stack2 is completed i.e. the top node of stack1 has been removed and it has been successfully inserted into stack2. The top nodes of both the stacks after the completion of transfer is still marked with updated information for the next transfer of node.

In the shift method only a single node from stack1 is moved to stack2. So it is repeatedly called from the help operation until there are no more nodes left in stack1 to be moved. So, before each transfer in the shift method, the stack1 is checked to see if it will be empty after this transfer process. This check is performed by comparing the next pointer of the node being removed from stack1 with a sentinel node. If the result of this comparison is true, it indicates that the stack1 will be empty so we will no longer need to maintain marked nodes at the top of each stacks at the end of the transfer process. In this case, the temporary marker node is not used as before. The three successive CASs are directly performed which transfers the last node of stack1 to the top of stack2 and finally removes the marked nodes from both the stack tops. At the end of the transfer process, the stack2 will contain all the nodes from stack1 in reverse order which represents the nodes of the queue in order. The top node of stack2 will be the earliest inserted item of the queue. This is the value to be removed in the dequeue operation. The stack1 will be empty with a sentinel node as its top.

Algorithm 7 Implementation of shift operation on a Queue

```
 1: function SHIFT(markedNode)
 2:     if markedNode.isTempMarker() then
 3:         cleanTemporary(markedNode);
 4:     else
 5:         StackNode topFirst = markedNode.next.get();
 6:         StackNode nextFirst = markedNode.nextFirst;
 7:         StackNode topSecondMarked = markedNode.topOther;
 8:         StackNode topSecond = topSecondMarked.next.get();
 9:         if nextFirst == s1.sentinel then
10:             topFirst.next.compareAndSet(nextFirst, topSecond);
11:             s2.top.compareAndSet(topSecondMarked, topFirst);
12:             s1.top.compareAndSet(markedNode, nextFirst);
13:         else
14:             StackNode newMarkerSecond = new StackNode(Integer.MAX_VALUE,topFirst);
15:             Marker newMarkerFirst = new Marker(nextFirst,newMarkerSecond,
16:             nextFirst.next.get());
17:             Marker tempMarkerFirst = new Marker(Integer.MIN_VALUE, markedNode,
18:             newMarkerFirst, newMarkerSecond);
19:             topFirst = s1.top.get();
20:             if topFirst.isTempMarker() then
21:                 cleanTemporary((Marker) topFirst);
```

Algorithm 8 Implementation of cleanTemporary operation on a Queue

```
 1: function CLEANTEMPORARY(tempNode)
 2:     Marker markedNodeFirst = (Marker) tempNode.next.get();
 3:     StackNode topFirst = markedNodeFirst.next.get();
 4:     StackNode nextFirst = markedNodeFirst.nextFirst;
 5:     StackNode oldMarkSecond = markedNodeFirst.topOther;
 6:     StackNode topSecond = oldMarkSecond.next.get();
 7:     StackNode newMarkFirst = tempNode.topOther;
 8:     StackNode newMarkSecond = tempNode.nextFirst;
 9:     topFirst.next.compareAndSet(nextFirst, topSecond);
10:     s2.top.compareAndSet(oldMarkSecond, newMarkSecond);
11:     s1.top.compareAndSet(tempNode, newMarkFirst);
```

Chapter 5

Correctness and Lock-freedom

In this chapter, we will show that the proposed data-structure Queue using two stacks is linearizable and lock-free. We will show that data-structure is linearizable by defining the linearization points for each operation.

5.1 Determining linearization point

5.1.1 Enqueue Operation

The enqueue operation inserts an element into the queue. This is achieved by pushing the element to the top of stack1. During execution of line 4 of the push operation for a stack in Algorithm 1, if the compare-and-swap atomic operation returns true, it indicates that the new node is now linked to the top node of the stack, resulting in the enqueue operation taking effect. Therefore the linearization point for a successful enqueue is at line 4 of the push operation for a stack Algorithm 1.

5.1.2 Dequeue Operation

The dequeue operation removes the element from the queue in FIFO order. This is achieved by popping the top element of stack2. Dequeue begins by calling the pop operation for stack2 which returns a node as shown in Algorithm 4. There are three cases of linearization points for dequeue:

1. Both stacks are empty

   If the returned node from the pop operation is a sentinel node, it means that stack2 is empty. When the stack2 is empty, we place a marked node on top of it by calling the announce operation. If the announce was successful and the top node of the stack2 has been marked,

we call the help operation to transfer the elements from stack1 to stack2. If the help method returns true, it means that the stack1 is empty as well. Thus we have an empty queue.

Also if any thread finds a marked at the top of stack2, it calls the help operation. The help operation returns true if it finds the top of stack1 to be a sentinel node. This case also implies that both stack1 and stack2 are empty, thus the queue is empty.

After rechecking that the top of stack2 is still the same as before, i.e. empty, the dequeue operation returns. Therefore the linearization point of dequeue when the queue is empty is at line 4 of the help operation, i.e. Algorithm 6, when the operation reads the top of stack1 to find that it is a sentinel node.

2. Stack2 is not empty and not marked

   When the pop operation for a stack is called, the top of stack2 is checked to see if it is empty i.e. a sentinel node or if it is marked. When both of these cases fail, it means that stack2 is not empty and also there is no pending dequeue operation. So a CAS is executed to remove the top of stack2 and replace it with the next of top of stack2. Therefore the linearization point for a successful dequeue is at the successful execution of line 3 during a call of the pop operation for a stack as shown in Algorithm 2.

3. Stack1 and Stack2 are not empty and Stack2 is marked

   When both the stacks are not empty and stack2 is marked, it implies an unfinished dequeue operation by some other thread. In this case, the help method is called to complete the transfer of all elements from stack1 to stack2. After the successful help, the execution loops back to call the pop operation for a stack2. In the pop, if the CAS is executed successfully, then it specifies that the top node of stack2 is removed, resulting in the dequeue operation taking effect. Thus the linearization point for a successful dequeue is at line 3 during a call of the pop operation for a stack as shown in Algorithm 2.

5.2 Proving correctness

5.2.1 Lemma 5.2.1 :

The linearization point of the enqueue and dequeue operations of the queue lie between their respective invocation and response, satisfying their sequential specifications.

Proof :

Enqueue - In the section 5.1.1, we have determined the linearization point for enqueue is line 4 of the push operation. We can observe that the push operation is called from within the enqueue operation, therefore we can state that the linearization point of enqueue lies between its invocation and response.

Dequeue - In section 5.1.2, we have determined the linearization point for dequeue. When the queue is empty, the linearization point is at line 4 of the help method. Since the help method in invoked from the dequeue method, it lies between the invocation and the return. Also, the linearization point for the cases (a) when both stacks are non empty and the stack2 is marked and (b) when the stack2 is non empty and not marked, is at the successful execution of CAS in line 3 of the pop operation. The pop operation is called from dequeue so therefore, it lies between the invocation and return of the dequeue.

Therefore, both the enqueue and dequeue operations of the Queue using two stacks are linearizable.

5.3 Proving Progress

The two lock-free stacks that we use to compose the data-structure Queue is the commonly used Trieber stack. To make our implementation of queue non-blocking, we have used the commonly available atomic primitive compare-and-swap.

We simply push the element to be enqueued to the top of stack1 to simulate the enqueue operation of the queue. The enqueue completes when the element to be enqueued is successfully pushed to the top of stack1 by the successful execution of CAS. If another concurrent enqueue operation is able to push its element to the top of stack1 first, then the current threads CAS will fail and the thread will restart the push operation.

Similarly, for the dequeue operation, we pop the element at the top of stack2 which is achieved by the successful execution of CAS in the pop operation of stack2. If any concurrent operation changes the top node of stack2, the CAS fails and it loops back to restart the pop again.

Also, if any thread encounters a marked node at the top of stack1 or stack2, it indicates that there is an unfinished dequeue operation. The marked node stores necessary information to complete the ongoing dequeue. So any thread encountering a marked node will first help complete this unfinished dequeue operation, and then continue their own expected operation. This implies system-wide progress.

Chapter 6

Experimental Results

In this section, we provide a performance comparison among different types of concurrent queues. We have measured the performance in terms of **throughput** and **fairness**.

Throughput can be defined as the total number of operations completed by all the threads within a certain time i.e. the overall rate at which method calls complete. High throughput value is a highly desired property for a synchronization method. A high throughput value is achieved if there are high number of successful operations of all the threads at a certain instance of time. High throughput implies high efficiency of performance.

The second property that we are going to consider in our study is fairness of synchronization constructs. In our experiment, we have used a relevant definition of fairness introduced by Ha et al [HPT07]. Here fairness is defined by comparing the minimum number of operations a thread had with the average number of operations of all threads [CCD+13]. To detect any unfair behavior, we use as a fairness measure the minimum of the above values [CCD+13]. Fairness value ranges from 0 to 1. If the value is close to 1, it means fair behavior. Low value of fairness means there are some threads that are being served less than others. The fairness measure helps to distinguish threads that are starving or that are being served less.

The following data structures were used for comparison:

1. Our proposed concurrent lock-free Queue using two lock-free stacks
2. Non-blocking concurrent linked-list based queue introduced by Michael and Scott
3. Two-lock concurrent queue introduced by Michael and Scott

We have only used the basic data type integer values in our benchmark, but other data types like float, double or any other custom data types can also be used. We performed our evaluations on a 64-bit, Intel Core i7-7500U CPU @ 2.70GHz with 8GB DDR3 RAM having 4 logical cores. The test was conducted for different numbers of threads. We used 2,4,8 and 16 number of threads in our testing. For each test, we started off with a 2-second warm-up period to allow the Java compiler to initialize and optimize the running code.

For each queue data-structure, the experiment was carried out for 5 seconds. We used different number of threads for each set of experiments. Then an average over 5 trials was taken to display the final result. We kept count of all the operations successfully completed by all the threads and divided this total number by the duration of time for which the experiment was conducted which gave us the throughput value. For fairness measure, we selected the minimum value between two calculated values: the average number of operations successfully completed by all the threads and the minimum number of operations of a thread.

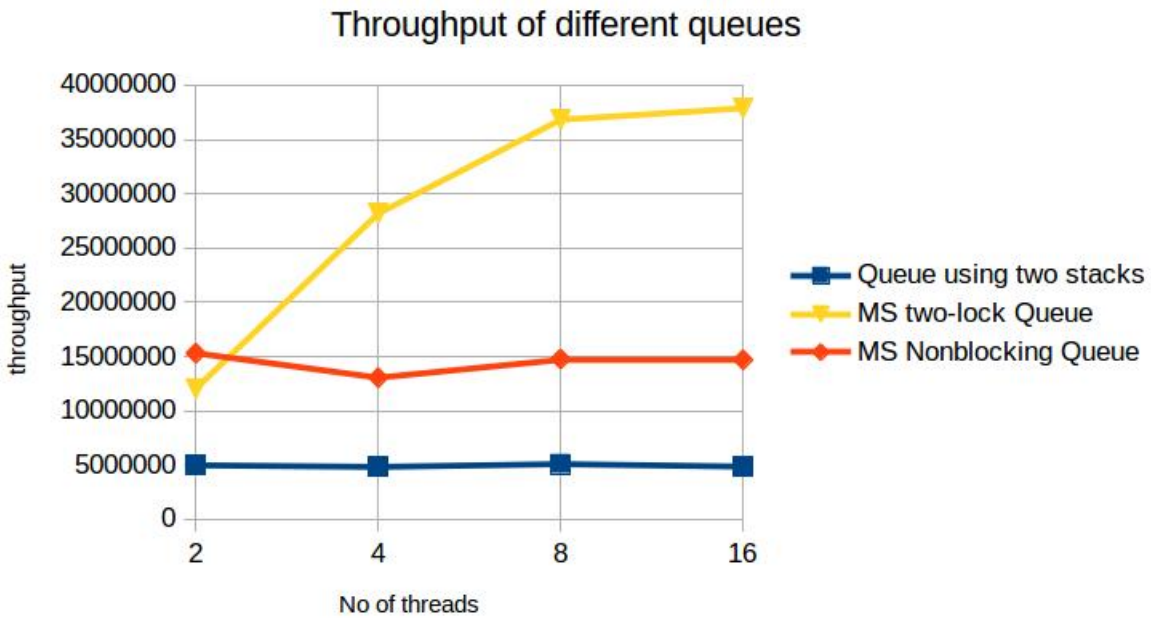

Figure 6.1: Throughput of concurrent queues

Figure 6.1 and 6.2 shows the throughput in Mops and fairness of the three concurrent queues with respect to the number of threads respectively.

We can see from the above figure, the two-lock concurrent queue has the highest of the throughput followed by the non-blocking concurrent queue and our proposed concurrent queue using two

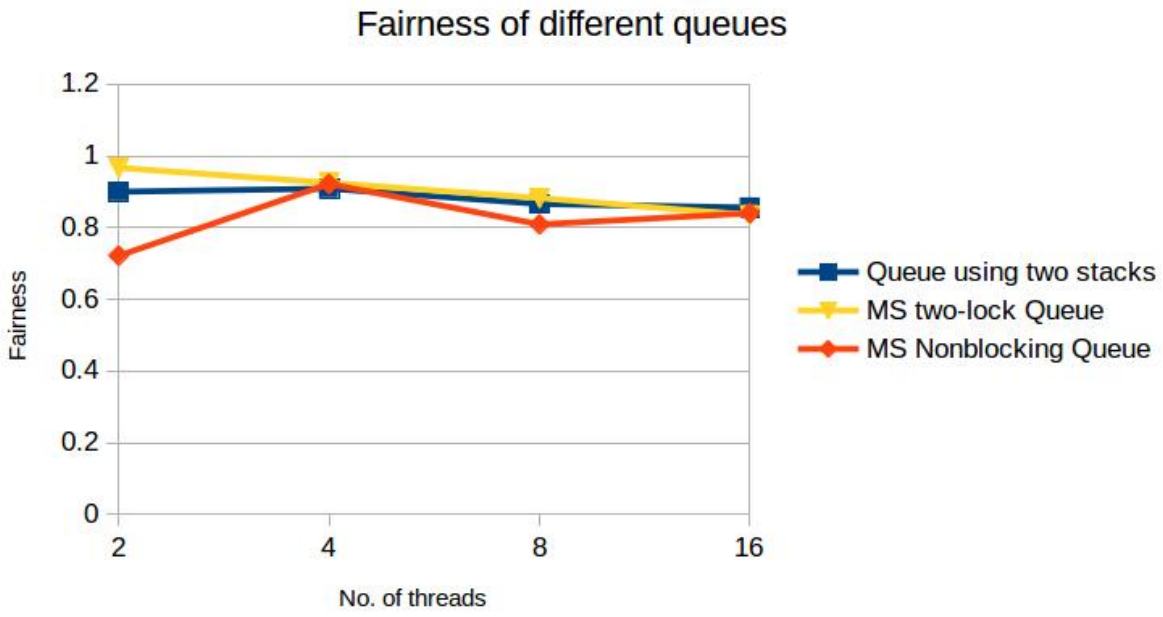

Figure 6.2: Fairness of concurrent queues

lock-free stacks.

The result of the experiment is expected. Reentrant locks were used to implement the two-lock concurrent queue. Reentrant lock is light weight and also has an inherent backoff mechanism. So the two-lock concurrent queue achieved higher throughput and fairness as compared to the non-blocking queue of Michael and Scott which uses CAS operations that are expensive.

Also we can see from the results, the performance of our proposed queue is the weakest. And it is reasonably so. The performance of our proposed queue was bound to be low as it was a composition of two separate non-blocking concurrent stacks. As we know, CAS operations are expensive. The need for synchronization using hardware primitives such as CAS to handle concurrency causes the performance of a non-blocking data-structure to drop. In the composition, since we need to introduce a lot more synchronization mechanisms to maintain its lock-freedom, it automatically decreases performance. Also the main objective to propose the construction of a concurrent queue using two lock-free stacks was not to present another lock-free queue with a better performance but to successfully design a composite data-structure in a concurrent setting which will provide a base for the composition of other advanced data-structure.

In terms of fairness also, the two-lock concurrent queue is the fairest among the three queues compared. However, as we can see from the Figure 6.2, our proposed concurrent queue using

two lock-free stacks has a good fairness measure as well and has value more or less similar to the two-lock concurrent queue for thread numbers 4, 8 and 16. For all the three queues, the fairness measure decreases with the increase in number of threads.

Chapter 7

Conclusion and Future Works

In summary, we have proposed a composition of lock-free data-structure by presenting a lock-free implementation of composing a concurrent lock-free queue using two concurrent lock-free stacks. We have used the push and pop operations of the composing stack components to implement the enqueue and dequeue operations of the resulting queue. We have used CAS operations and helping mechanism to ensure linearizability and guarantee progress of our implementation. We have also presented a performance comparison of our proposed concurrent queue with two other popular and commonly used queues in terms of throughput and fairness.

In our study, we ran our benchmarks on a machine with 4 logical cores and varied the number of threads from 2 to 16 only. So for future work, we can run the program on a more powerful machine having a higher number of logical cores and number of threads and analyze the performance. Also, we can use similar algorithm to design a concurrent lock-free stack using two lock-free queues which is another fundamental example of lock-free data-structure composition.

From the experimental results, we have observed that our proposed queue does not provide better performance output compared to the basic lock-free and lock-based queues. This is not surprising since we had to introduce more synchronization mechanisms in the process of composition to ensure linearizability. So for future work, we can introduce optimization in the synchronization mechanisms to provide a much better performance.

Our implementation of the non-blocking queue is a non-trivial yet a fundamental contribution to the slowly growing study in the field of composition of data structures. It can be used as a base for other advanced lock-free data structures to be created by combining different basic lock-free data structures using a similar approach.

Bibliography

[Adh14] Ashok Adhikari. Non-blocking Priority Queue based on Skiplists with Relaxed Semantics. 2014.

[AS01] Greg Gagne Abraham Silberschatz, Peter Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 6th edition, 2001.

[Bar93] G. Barnes. A method for implementing lock-free data structures. *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[CC92] Donald Cohen and Neil Campbell. Automatic composition of data structures to represent relations. pages 182–191, 1992.

[CCD+13] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen Dang, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems. pages 1309–1320, 2013.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Dan14] Nhan Nguyen Dang. On Composability, Efficient Design and Memory Reclamation of Lock-free Data Structures. 2014.

[DC13] Philippas Tsigas Daniel Cederman. Supporting Lock-Free Composition of Concurrent Data Objects: Moving Data between Containers. *IEEE Transactions on Computers*, 62(9):1866–1878, 2013.

[ERAEB05] Hesham El-Rewini and Mostafa Abd-El-Barr. Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing). *WileyInterscience*, 2005.

[Gee05] D. Geer. Chip Makers Turn to Multicore Processor. *Computer*, 38:11–13,05, 2005.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, pages 124–149, 1991.

[Her93] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[HPT07] Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Efficient self-tuning spin-locks using competitive analysis. *Journal of Systems and Software*, 80(7):1077 – 109, 2007.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28:690–691, 1979.

[MMM96] M. L. Scott M. M. Michael. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Proceedings of PODC 1996*, Part II:267–275, 1996.

[MS98] M. M. Michael and M. L. Scott. M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[NND11] Philippas Tsigas Nhan Nguyen Dang. Progress guarantees when composing lock-free objects. *Euro-Par'11 Proceedings of the 17th international conference on Parallel processing*, Part II:148–159, 2011.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. *Technical Report RJ 5118, IBM Almaden Research Center*, April 1986.

[Ven11] Balaji Venu. Multi-core processors - An overview. *CoRR*, abs/1110.3535, 2011.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Neha Bajracharya

nehabajracharya@gmail.com

Degrees:

Bachelor of Computer Engineering 2012

Tribhuwan University, Nepal

Thesis Title: Efficient and practical composition of lock-free data structures

Thesis Examination Committee:

Chairperson, Ajoy K. Datta, Ph.D.

Committee Member, John Minor, Ph.D.

Committee Member, Yoohwan Kim, Ph.D.

Graduate Faculty Representative, Venkatesan Muthukumar, Ph.D.