Scalable and Performance-Critical Data Structures for Multicores

Mudit Verma

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Johan Montelius

June 2013

Acknowledgments

I would like to express my gratitude to Prof. Luís Veiga from IST and Dr. Marc Shapiro from INRIA, who guided me throughout this thesis. Both of them became great source of inspiration and provided valuable feedback regularly. I would also like to thank Gaël Thomas and Lokesh Girda from INRIA, with whom I had brainstorming sessions on many occasions.

I would finally like to thank Prof. Johan Montelius, the coordinator of EMDC program without whom, the smooth journey of this master would not have been possible.

This work was supported by *International Internship Program 2013* at INRIA.

European Master in Distributed Computing, EMDC

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Community via the Erasmus Mundus program. My track in this program has been as follows:

First and second semester of studies: IST
Third semester of studies: KTH
Fourth semester of studies: IST (Internship at INRIA Paris)

Abstract

In this work, we study the scalability, performance, design and implementation of basic data structure abstractions, such as a queue, for next generation multicore systems. We propose two algorithms for concurrent queue. Our first algorithm, a wait-free queue, provides an efficient replacement to a lock-free queue. Lock-free queue is considered very efficient, but does not provide local progress guarantee for each thread. It also performs badly under stressed conditions. Our wait-free queue, not only provides local progress guarantee, but also depicts high performance and positive scalability under highly stressed conditions. Our second algorithm, a sequentially consistent queue, further achieves high performance by changing the consistency model. All the queue algroithms provide linearizability, which orders the operations on a global time scale. However, our sequentially consistent queue orders the operations in a program order, which is local to a thread. Our experimental results shows that our algorithms outperforms existing *state-of-the-art* algorithm by a factor of 10 to 15.

Resumo

Neste trabalho estudamos a escalabilidade, desempenho de estruturas de dados fundamentais como queues, para potenciar o seu desempenho e escalabilidade com vista à próxima geração de sistemas multicore. Propomos dois algoritmos para uma queue concorrente. O primeiro, uma queue wait-free, permite substituir com mais eficiência uma queue lock-free. Esta última é considerada muito eficiente mas não oferece garantias de progresso a todas as threads numa computação concorrente, além de ter mau desempenho sob carga elevada. A nossa queue wait-free, não só oferece as garantias de progresso referidas como também obtém alto desempenho e escalabilidade sob carga elevada. O nosso segundo algoritmo, uma queue com consistência sequencial, consegue atingir desempenho ainda superior através da modificação do modelo de consistência. Todos os algortimos cumprem linearizability, que ordena as operações globalmente. Contudo, a nossa queue com consistência sequencial ordena as operações de acordo com a ordem no programa, que apenas é local a cada thread. Os resultados experimentais revelam que os nossos algoritmos ultrapassam o desempenho do estado da arte por um factor de 10 a 15.

Keywords

Multicore

Linearizability

Sequential Consistency

Lock-Free

Wait-Free

Queue

Palavras-Chave

Multicore

Linearizability

Consistência seqüencial

Lock-Free

Wait-Free

Queue

Index

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Shortcomings of current solutions . . . 2
1.3 Thesis Objective . . . 3
1.4 Contributions . . . 5
1.5 Document Roadmap . . . 5

2 Related Work **7**
2.1 Multicore or Multiprocessing systems . . . 7
2.1.1 Symmetric Multiprocessing Machines (SMP) . . . 8
2.1.2 Non-Uniform Memory Access (NUMA) machines . . . 9
2.2 Consistency Model . . . 10
2.2.1 Linearizability . . . 10
2.2.2 Sequential Consistency . . . 11
2.3 Concurrent Data Structures . . . 13
2.3.1 Lock-based or Blocking . . . 14
2.3.2 Lock-free and Non-Blocking . . . 15
2.3.3 Wait-Free . . . 16
2.4 Concurrent Queues . . . 16
2.4.1 Blocking queues . . . 17
2.4.2 lock-free or non-blocking queues . . . 18
2.4.3 Wait-free Queues . . . 22
2.4.4 Semantically Relaxed Queues . . . 24
2.5 Summary . . . 25

3 Scalable and Performance-Critical Queues **27**
3.1 Wait-Free Linearizable Queue . . . 27

3.1.1 Background 27
3.1.2 Insight 28
3.1.3 Algorithm 29
3.1.4 Discussion on Correctness 36
3.2 Sequentially Consistent Queue 39
3.2.1 Background 39
3.2.2 Algorithm 39
3.2.3 Discussion on Correctness 42
3.3 Implementation 43
3.4 Analysis 45

4 Evaluation **47**
4.1 Experimental Setup 47
4.2 Performance and Scalability 48
4.2.1 Per operation Completion time 48
4.2.2 Total Execution Time 49
4.2.3 Throughput 50
4.2.4 Think Time 51
4.2.5 Fair Share 52
4.3 Graph Exploration 54
4.4 Analysis 55

5 Conclusion **57**
5.1 Future Work 58

Bibliography **59**

List of Figures

1.1 Maximum achievable speedup by Amdahl's Law with different fraction of parallel code . . . 3
1.2 Scalability of Michael and Scott's lock-free queue . . . 4

2.1 Symmetric Multiprocessing (SMP) architecture . . . 8
2.2 Non uniform memory access (NUMA) architecture . . . 9
2.3 A sequence of events (operations) on a queue by two different processes . . . 12

3.1 A visual example of the algorithmic steps in an enqueue operation for worker with id 4. . . . 31

4.1 A per operation completion time comparison with the increasing number of threads (workers) . . . 49
4.2 A total execution time comparison with the increasing number of threads (workers) . . . 50
4.3 Throughput comparison with the increasing number of threads (workers) . . . 51
4.4 Performance comparison with increasing Think Time (work done between two operations) . . . 52
4.5 Worker completion time in a pool of 32 workers. . . . 53
4.6 Graph exploration completion time on 16 and 32 cores respectively. 55

List of Tables

2.1 A comparison of Queues 26

3.1 General Safety Properties 36

5.1 A comparison of previous Queue algorithms with our Queue algorithms 58

Listings

2.1 Code skeleton for compare-and-swap (CAS) operation 15
2.2 Algorithm for Michael Scott's blocking queue 17
2.3 Algorithm for enqueue operation (Michael and Scott) 20
2.4 Algorithm for dequeue operation (Michael and Scott) 21
3.1 Global fields and queue initialization (Wait-Free Queue) 30
3.2 Algorithm for enqueue operation (Wait-Free Queue) 32
3.3 Algorithm for dequeue operation (Wait-Free Queue) 33
3.4 Algorithm for peek operation (Wait-Free Queue) 33
3.5 Algorithm of Helper (Wait-Free Queue) 34
3.6 Algorithm for enqueue operation (Sequentially Consistent Queue) 40
3.7 Algorithm for dequeue operation (Sequentially Consistent Queue) 41
3.8 Algorithm for merge operation (Sequentially Consistent Queue) . . 42

1 Introduction

1.1 Background and Motivation

Multicore systems have been in practice since its inception in early 80s. However, the usage was limited to scientific computing. On the verge of Moore's law [26], as it is constantly getting harder to put more transistors in a single processor, multicore machines are bound to represent the present and future of computing systems [7]. In last 5-6 years multicore systems have come to main stream and can be seen in mobile devices, desktop machines, laptops, network gears and enterprise servers.

Contrary to the rapid development in multicore hardware architectures, softwares are still not able take full advantage of these massively parallel working units. Much of the problem lies in the way applications are designed and the way they utilize basic data structures. Parallel and concurrent programming is naturally hard. Therefore, programmers tend to pursue conservative approaches in order to avoid race conditions, while working on shared memory systems on multicores [9]. In a multithreaded application, threads running on different cores access the globally shared data structures. Concurrent access to these data structure becomes a bottleneck if the access rate is very high, which consequently results in degraded application performance. This problem increases as the number of threads in an application and number of cores in a machine increases [15].

The ever growing challenge for programmers is to build applications that could benefit from these massive parallel machines. While the efforts are being made at hardware level to come up with better multicore architectures, computing industry should also aim to redefine the way software uses these machines.

To ease the multithreaded application programming and to achieve better performance, number of efforts have been made one of which is *java.utils.concurrent* library. This library provides sophisticated basic concurrent data structure abstractions such as queue, stack, maps and counters and implements few of the best known concurrent algorithms for these abstractions. These algorithms benefit the applications in two ways. First, they provide thread safe abstractions, where programmers do not need to worry about the synchronization and race conditions. Second, they provide lock freedom, which results in better performance.

1.2 Shortcomings of current solutions

Although existing algorithms for concurrent data structures have achieved good results, yet they are far from being ideal in terms of scalability and performance. Future promises us that multicore machines are here to stay and grow. With the increasing number of cores, dynamic interaction among threads and inter communication is only going to increase [27]. According to the famous Amdahl's law, scalability of a program is limited by the portion of the code that needs to be run sequentially [13]. If f be the fraction of code that can be run in parallel and n be the number of cores in a multicore machine, the achievable speedup e of an application is only limited to $\frac{1}{(1-f)+\frac{f}{n}}$. Figure 1.1 depicts the speedup with different values of n and f.

Linear scalability is the ideal desired outcome for any application running on multicore machine. However, our experimental results shows us that few of the sophisticated concurrent data structure algorithms are far from achieving ideal linear scalability. Infact, they do not even achieve positive scalability. Figure 1.2 presents the scalability of *state-of-the-art* Michael and Scott's lock-free queue algorithm which is also implemented in *java.util.concurrent* library. This algorithm shows negative scalability under high concurrent access rate. On 32 cores, this algorithm is only able to perform 200 operation/ms as compared to 6000 operation/ms on a single core machine where it needs to perform all the operations

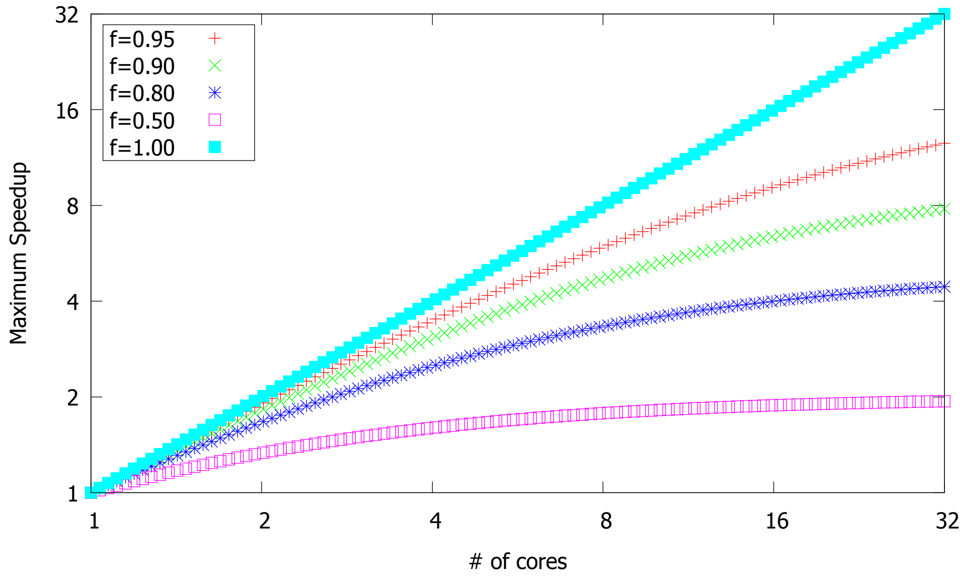


Figure 1.1: Maximum achievable speedup by Amdahl's Law with different fraction of parallel code

sequentially. Therefore, the problem is not with running the code sequentially, but with the inherent underlying communication overheads, which these data structures suffer from on a multicore machines.

1.3 Thesis Objective

Considering the scalability and performance issues with existing algorithms, we foresee a fundamental shift in data structure's design for next generation multicores. Therefore, we are interested in designing and implementing efficient algorithms that are more scalable and perform better under stressed conditions. Currently, all the practiced algorithms provide semantics such as linearizability and lock-freedom. However, we look into alternate directions. We explore wait-free algorithms which can perform better than lock-free algorithms. Wait-free algorithms have gained the attention only recently, and they are not widely covered in the literature [16].

Unlike lock-free algorithms, which provide progress guarantee for one worker, wait-free algorithms provide a progress guarantee for all the workers. We empha-

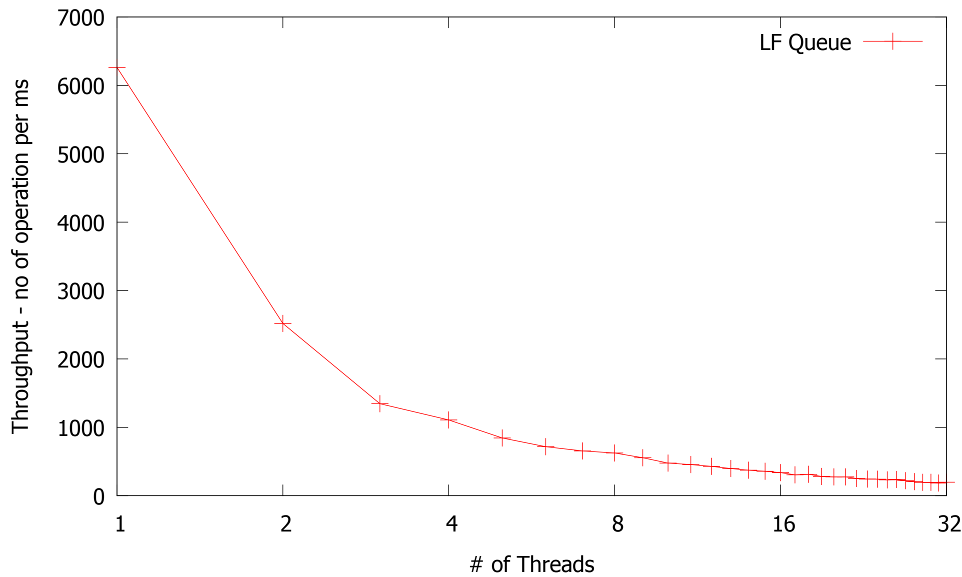


Figure 1.2: Scalability of Michael and Scott's lock-free queue

size that, with cloud computing being the future, where compute is governed by service level agreements (SLAs) and the quality of service (QoS), wait-free algorithms will replace lock-free algorithms. Therefore, we work on the design of efficient wait-free algorithms.

We also explore the trade-off between the correctness and performance. High performance can be obtained at the expense of correctness. Therefore, we consider the relaxation of consistency model from linearizability to sequential consistency and apply it to basic data structures. We apply our techniques to concurrent queue and introduce two new efficient algorithms for a wait-free concurrent queue and a sequentially consistent concurrent queue. However, same techniques can also be applied to other concurrent data structures such as stacks, linked lists and skip lists.

Our primary aim is to target situations, where applications heavily access the concurrent objects and cause high contention. For other cases, existing algorithms provide fair results and is out of the scope of this thesis.

1.4 Contributions

This thesis summarizes my contributions related to scalability and performance of concurrent data structures with queue as a design example. Our main contributions are:

- We present an overview of some of the *state-of-the-art* concurrent data structure's design techniques and algorithms.
- We present a wait-free linearizable queue as a replacement to *state-of-the-art* Michael and Scott's lock-free linearizable queue, which is also implemented in JAVA concurrent library. Our simple algorithm uses an external dedicated helper that performs operations on behalf of other worker threads while being fair everyone.
- We present a sequentially consistent queue. We achieve high performance by semantically relaxing the consistency model from linearizability to sequential consistency. In this algorithm, we try to reduce the synchronization among the worker threads by providing local enqueue operations.

1.5 Document Roadmap

The remainder of this document is organized as follows. Chapter 2 covers the related work in the concurrent data structures on multicores, consistency models and *state-of-the-art* queue algorithms. Chapter 3 covers the algorithmic design and implementation of two algorithms, a wait-free queue and a sequentially consistent queue. Chapter 4 covers the performance and scalability evaluation of both the algorithms. Finally, chapter 5 concludes the thesis with a brief discussion and a roadmap for future research.

2 Related Work

In this chapter, we analyze the multicore architectures, consistency models and various algorithms and techniques for concurrent data structures in multicore machines. In 1st section, we analyze different types of multicore architectures and their associated problems with regard to the concurrent data structures. In 2nd section, we audit different types of consistency models in the literature. In 3rd section, we examine the synchronization techniques and progress guarantees used for concurrent data structure. Finally, in 4th section, we study the various algorithms for concurrent queues.

2.1 Multicore or Multiprocessing systems

With the growth of multiprocessing machines [1], parallel and concurrent programming has become the part of main stream. In this section, we will consider two classes of multicore architecture. First and most common is *Symmetric Multiprocessing* (SMP) architecture. In this architecture two or more identical processing units are connected to a shared main memory. These types of machines can be found in mobile phones, laptops, desktops and servers. The second type of architecture is *Non-Uniform Memory Access* (NUMA) architecture. In NUMA architecture, many SMP machines are connected using a network. This provides scaling over SMP machines. In following subsections we will briefly analyze both the architectures are some of their known issues.

[1] There is a minor difference between the multiprocessor and multicore machines. However, in this document we use both the terms exchangeably to describe a machine that has two or more processing units. These processing units can be either be a processor or a core.

2.1.1 Symmetric Multiprocessing Machines (SMP)

SMP machines were first introduced by IBM in 1960s [2]. This main idea behind symmetric multiprocessing is to allow hardware level parallelism for the softwares running on these machines. In this architecture, all the processing units have global view of system's resources such as memory, I/O and the network. A single instance of operating systems governs the execution of application running on these machines. Figure 2.1 depicts the architecture of a SMP machine.

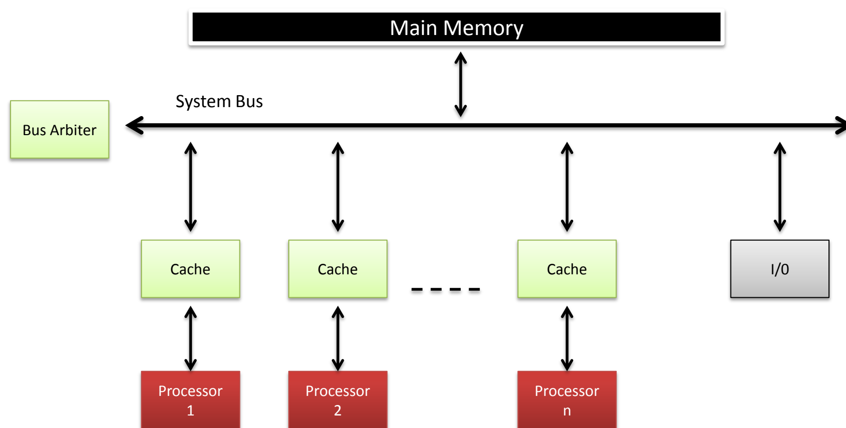


Figure 2.1: Symmetric Multiprocessing (SMP) architecture

Shared interconnect between the processors in SMP machines provides a homogeneous access to the resources. However, this design does not work well as the number of processors increases. The shared bus connecting the processors to memory becomes a bottleneck due to the limited bandwidth [31]. For example, a 4 processor SMP machines is only 2.7 times faster than a uniprocessor machine. This bottleneck at the hardware level does not allow applications to fully utilize the computational power of these machines.

2.1.2 Non-Uniform Memory Access (NUMA) machines

Non-Uniform Memory Access provides a scalable solution to limited bandwidth problem in SMP machines. A NUMA machine is a set of smaller SMP machines connected through a DSM network. There exists commercial NUMA machines which has more than 1000 cores. Figure 2.2 presents a general architecture of a NUMA machine. In a NUMA machine, the memory access time depends on its location with respect the to position of processing unit. Each SMP machine (also called node) within a NUMA, has its own local main memory. This memory is shared between all the cores within the node. Whenever, a core in a node need to access memory from some other node, it needs to send the request to the remote node using the long path through the network.

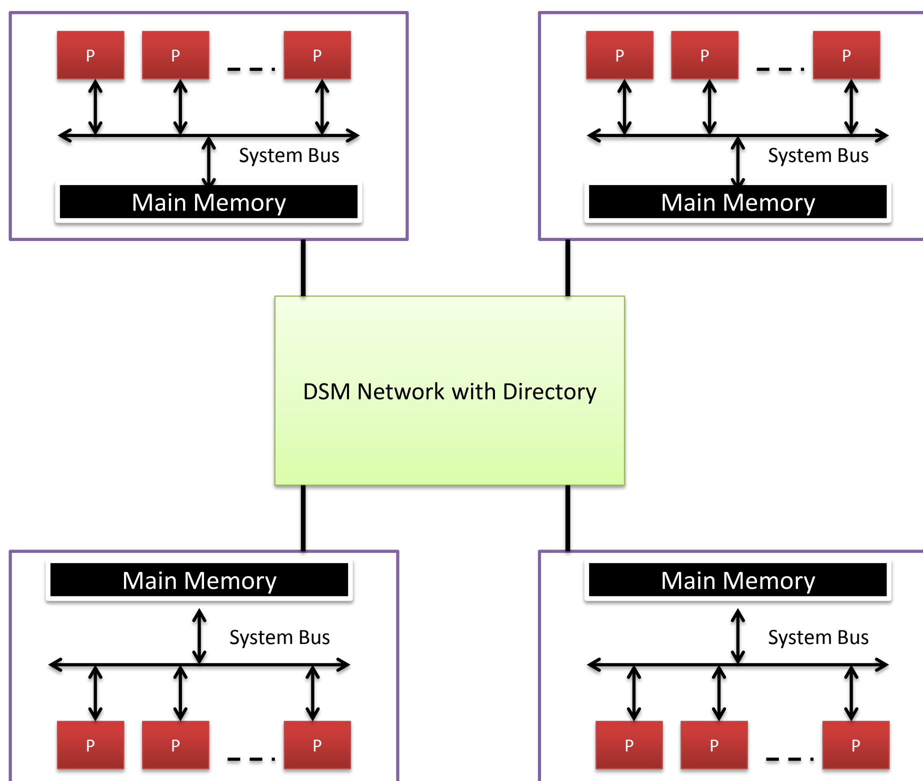


Figure 2.2: Non uniform memory access (NUMA) architecture

Although this architecture allows fast access to local memory, it becomes problematic for a multithreaded program running across the nodes which accesses a

globally shared data structure. The problem becomes worse in case of cache coherent NUMA machine where cache has to be kept consistent cross the nodes. If the threads running across the nodes, access the shared data structure too frequently, in becomes impossible for data to remain locally available. Therefore, it causes the remote memory access in almost every operation on the shared data structure. This creates a ping-pong type behavior which saturates the network. This increased contention results in severally reduced performance of an application [3].

The other issue with NUMA machines is the unfair access patterns. For example, if a two threads running on node 1 and node 2 respectively simultaneously access a memory location which belongs to node 1, there are high chances that the thread running on node 1 will win. This happens because the time taken for a request arriving from a remote node is longer than the time taken by a request arriving from local node.

2.2 Consistency Model

In a shared memory system, consistency model, provides a set rules for memory consistency. It these rules are followed by the programmers, system will provide a predictable outcome of memory operations. In this section we cover two types of consistency models, linearizability and sequential consistency.

2.2.1 Linearizability

Linearizability was first introduced by Herlihy and Wing in 1990 as a consistency model [12]. This model became widely popular in multiprocessor programming, where the globally shared data structures are accessed concurrently by processes running on different cores. Most of the implementations of basic concurrent data structures such as queue, stack, hash map, linked-list and trees provide linearizability as the correctness reasoning which provide execution equivalent to a sequential execution.

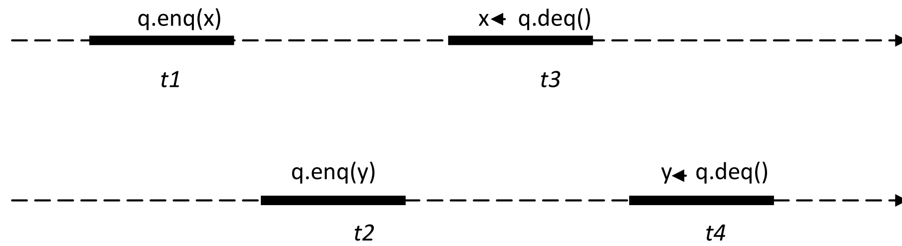
According to this consistency model, for an observer each operation applied by processes sharing a concurrent data structure (object), happens to take effect instantaneously at some point between the operation's invocation and the response. Therefore, all the operations can be given a pre-condition and a post-condition. Pre-condition defines the state of a concurrent object before the operation invocation event, whereas, post-condition defines the state of a concurrent object after the operation response event. Linearizability is considered one of the strongest level of consistency guarantee. It provides a global ordering of operations on a linear time scale.

Linearizability is commonly achieved with the help of mutual exclusions, semaphores, locks and primitives such as CAS (compare-and-swap). It gives an illusion that an instruction or a set of instructions came in effect instantaneously. This can be further understood with a formal example. Queue events presented in figure 2.3.a is linearizable because the enqueue and dequeue operations take effect on global time scale. Element x is enqueued first and therefore, it is dequeued first. However, events in Figure 2.3.b and 2.3.c violate the linearizability property.

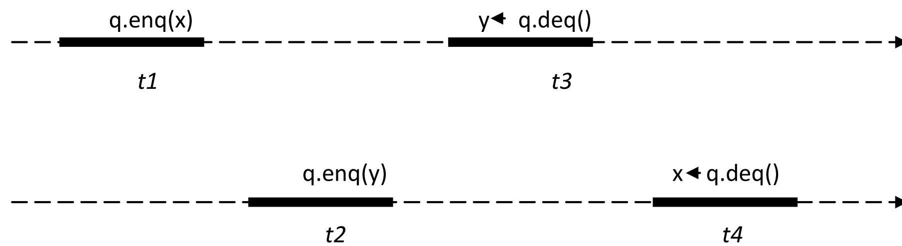
2.2.2 Sequential Consistency

Sequential consistency is a weaker consistency model as compared to linearizability. Unlike linearizability, which considers global time as the basis of ordering, sequential consistency considers program order as the basis of ordering. This model was introduced by Lamport as a consistency model for multiprocessor systems [18]. According to this model, for an observer, all the operations applied on a shared concurrent object, happens to come in effect according to program order.

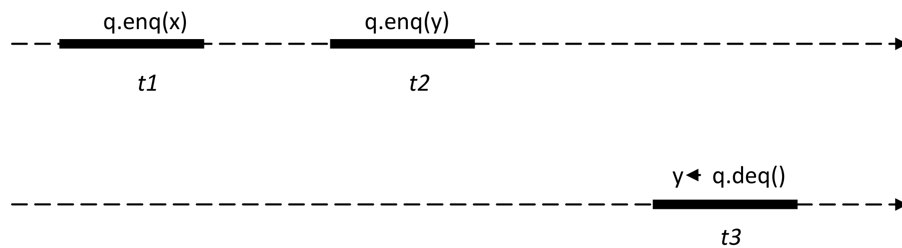
Program order is defined by the order in which a process (thread in case of multithreaded program) applies the operations to a concurrent shared object. The preservation of program order in sequential consistency restricts an observer to view an inconsistent ordering of the operations issued by the same process. However, it does not restrict a global order to come in effect in which operations between the processes are reordered.



a). Process 1 enqueues an element *x* at time *t1*. Process 2 enqueues element *y* at time *t2*. Process 1 dequeues element *x* at time *t3*. Process t2 dequeues the element y at time *t4*.



b). Process 1 enqueues an element *x* at time *t1*. Process 2 enqueues element *y* at time *t2*. Process 1 dequeues element *y* at time *t3*. Process 2 dequeues the element *x* at time *t4*.



c). Process 1 enqueues an element *x* at time *t1*. Process 1 enqueues element *y* at time *t2*. Process 2 dequeues element *y* at time *t3*.

Figure 2.3: A sequence of events (operations) on a queue by two different processes

This relaxed notion gives the system a freedom to interleave the operations coming from different processes in any order as long as the program order is preserved. Therefore, sequential consistency is preferred in scenarios where global time ordering is not required. Figure 2.3 gives a pictorial representation of the sequence of events on a queue. Events presented in Figure 2.3.a and 2.3.b are sequentially consistent as it follows the program order with respect to FIFO property of a queue. However, figure 2.1.c violates the program order as y is dequeued before x while x was enqueued first.

2.3 Concurrent Data Structures

In this section we will formally understand how the concurrent data structures are designed on a multicore machine. Concurrent data structures (objects) on a multicore machines are accessed simultaneously by different processes, and therefore requires careful consideration. Generally, the correctness and progress of a concurrent data structure is provided by defining two essential properties namely, safety and liveliness [14].

Safety property ensures that something incorrect such as the lost updates or the violation of happens-before relationship never occurs. The other property, liveliness guarantees that something productive continues to happens as the time elapses. Liveliness property is generally provided by the the implementation whereas safety property can be provided by the design.

Following subsections describes how the safety and liveliness can be achieved by employing different techniques such as by using the blocking access, lock-free access or the wait-free access.

2.3.1 Lock-based or Blocking

The traditional approach, to provide safe access to a concurrent shared data structure, is the usage of locks. Lock synchronizes the access to the shared data structure and provides mutual exclusion. Different synchronization primitives are used to provide a safe access to these shared data structures. Some of such primitive includes mutex, critical section and semaphore. These primitives are essential for safety because they restrict the processes from simultaneously access and modify a shared data structure. In absence of these means, processes that simultaneously act on these shared data structure might corrupt the memory of these data structures.

Despite the common usage in resource protection, many problems are associated with the locks. These problems are widely known in literature [5, 20, 28, 19]. Following are few such problems.

- Locks cause blocking. That means, at any given time, only one process will be able to continue the work, while others have to wait until the lock is released.

- Locks require careful usage and implementation. A poorly used lock can result in a deadlock causing the complete halt of execution progress.

- If, a process holding the lock gets blocked due to a byzantine fault or gets scheduled out, the other processes waiting for the lock might have to wait indefinitely.

- Locks increase the underlying communication overhead. This creates contention and limits the scalability of program.

- A low priority process, holding a common lock with a high priority process, might halt the progress of high priority process indefinitely. This behavior is unwanted and works against the notion of priority.

2.3.2 Lock-free and Non-Blocking

Considering the some of problems associated with locks, lock-free and non-blocking algorithms were introduced. Lock-freedom ensures that processes accessing the shared data structure will not have to block indefinitely due to the mutual exclusion. Also, unlike locking, lock-freedom guarantees program level progress at at any time.

Lock-freedom is generally achieved using the hardware *read-modify-write* primitives. One of the widely used *read-modify-write* primitive is *compare-and-swap* (CAS). CAS was first introduced in IBM 370 architecture [8]. CAS instruction takes three parameters, an address of a memory location, an expected value and a new proposed value. If, the value at the provided memory location is equal to the expected value, the value at memory location is changed to the new value. All these steps are combined and executed atomically by the hardware. Therefore, CAS behaves as a single atomic instruction. Listing 2.1 gives an exemplary code skeleton of a CAS operation.

Listing 2.1: Code skeleton for compare-and-swap (CAS) operation

```
1 bool CAS(addr,expected,new) {
2         if(valueAt(addr)==expected){
3                 valueAt(addr)=new;
4                 return true;
5         } else {
6                 return false;
7         }
8 }
```

Most of the lock-free algorithms use the CAS to provide safety and progress guarantee. With the help of CAS, lock-free algorithms overcome the problems associated with the lock, and as a result, provide an efficient alternative to lock-based or blocking algorithms. Although lock-free algorithms provide overall system level progress guarantee, they do not provide process or thread level progress guarantees. In many execution scenarios, one or more threads in a program can

be starved despite the overall progress.

2.3.3 Wait-Free

Wait-freedom provides local progress guarantee. That means, all the processes or threads in a program will be able to make progress irrespective of each other. This eliminates the possibility of starvation for any process or thread. It is a stricter progress guarantee compared to lock-freedom, which only guarantees the global progress. Wait-freedom is desired by live systems, which requires progress for all the processes. Different processes are generally associated to different clients which requires a minimum quality of service for each of them. Generally, wait-freedom ensures that any operation by a process or a thread will be completed in a bounded number of steps.

Despite the stronger and desired guarantees provided wait-free algorithms, it is extremely hard to design and implement an efficient wait-free algorithm. Lock-free algorithm uses primitives such as CAS for better performance. However, *Herlihy* proved that it is impossible to design a wait-free algorithm by only using the low level synchronization primitives such as CAS [11]. Therefore, many of the wait-free algorithms relies on complex out of the box techniques to provide wait-freedom. This results in a complex wait-free algorithm, which does not provide better performance when compared to the lock-free algorithms [23].

2.4 Concurrent Queues

In this section, we critically analyze various concurrent first-in first-out (FIFO) queue algorithms. First, we analyze the blocking queue algorithms which use locks for mutual exclusion, followed by the lock-free queue algorithms, which uses low level atomic primitive such as CAS. Finally, we analyze the wait-free queue algorithms.

A concurrent queue supports two fundamental operations, enqueue and dequeue. Enqueue operation appends the element to end (tail) of the queue, whereas a dequeue operation remove the element from the start (head) of the queue. Head holds the element which is present in the queue for the longest time, whereas, tail holds the element which is in queue for the shortest time.

2.4.1 Blocking queues

Just like other blocking algorithms, blocking queue synchronizes the access to the queue. Queue has two access points, head and tail. In a conservative approach, a single lock can be used for the queue which synchronizes the enqueue and dequeue operations. This gives a safe implementation of a concurrent queue. However, Amdahl's law suggest that concurrent data structures whose operations hold exclusive lock fails to scale [10].

Since, a queue has two access points, it is advantageous to hold separate locks for enqueue and dequeue operations. First such practical blocking algorithm for concurrent queue was presented by Michael and Scott [22]. The algorithmic steps for enqueue and dequeue operations are given in the listing 2.2. In both the operations, enqueue and dequeue, respective locks are acquired before modifying the head and tail references. This gives a thread safe implementation of the queue which is correct as per the sequential specification.

Listing 2.2: Algorithm for Michael Scott's blocking queue

```
1 enqueue(e) {
2       //create a new node
3       node = new_node(e,null);
4       //lock the tail
5       tail_lock();
6       //append new node the the tail
7       tail.next = node;
8       //update the tail reference to new node
9       tail = node;
```

```
10          // unlock the tail
11          tail_unlock();
12      }
13
14
15      E dequeue() {
16          //lock the head
17          head_lock();
18          // get the top node
19          node = head.next;
20          // queue is empty
21          if(node == null) {
22                  // unlock head
23                  head_unlock();
24                  return null;
25          }
26          // change the head pointer
27          head = node;
28          // unlock the head
29          head_unlock();
30          //return element
31          return node.e;
32      }
```

2.4.2 lock-free or non-blocking queues

Lock-free algorithms presents an efficient alternative to blocking algorithms. Not only the lock-free algorithms provide better performance, they also provide a global progress guarantees. Most of the practical lock-free algorithms are based on compare-and-swap (CAS) primitive.

One of the most efficient lock-free concurrent queue algorithm was presented by Michael and Scott [22]. In literature, it is considered as *state-of-the-art* lock-free algorithm [10, 30, 17]. Following is the brief description of the basic queue struc-

ture and the functioning of enqueue and dequeue operations as presented in this algorithm.

- **Basic Structure**: This algorithm implements an unbounded thread-safe queue based on linked nodes. The *head* of the queue is the element which has been in the queue for the longest time whereas, *tail* of the queue holds the element which is in the queue for the shortest time. Queue globally maintains the reference to *head* and *tail* of the queue and updates them as and when elements are enqueued or dequeued.
- **Enqueue operation**: The algorithmic step in enqueue operation is presented in listing 2.3. It has two atomic steps. First, to append the element to the tail of the queue (line 16). Second, to update the tail reference to the new appended element (line 18). Both of these steps are done atomically using the CAS. The algorithm loops indefinitely until it manages to apply first step (first CAS, line 16). Meanwhile, if it fails to append the element, it tries to fix the tail reference which is modified by some other thread (line 24).
- **Dequeue operation**: Dequeue operation also works similar to enqueue operation. It loops indefinitely until it manages to remove the first (top) element from the queue and update the head pointer. If the queue is empty it returns the null element (line 15-18). In other case, it tries to atomically get the element and update the head reference to the next element in the queue (line 23-31). The algorithmic steps are presented in listing 2.4.

There are other variants of lock-free queue algorithms, which have been proposed to solve different problems. One such variants uses doubly linked list and single CAS for dequeue operation as compared to two CAS which are used in this algorithm. However, this new variants still uses 2 CAS for the enqueue operation. Therefore, it is only optimized for dequeue operations [17]. Another variant uses elimination and random back-off techniques for the better performance [24].

Listing 2.3: Algorithm for enqueue operation (Michael and Scott)

```
1   bool enqueue(E e) {
2       //create a new node
3       n = new_node(e, null);
4     // keep trying until enqueue is done
5       while(true) {
6         //read tail reference
7           t = tail;
8           // read tail next
9           next = t.next;
10          // is tail consistent
11          if (t == tail) {
12              // is tail pointing to last node
13              if (next == null) {
14                // try to link the node at the end
15                // of linked list
16                  if (CAS(t.next, next, n)) {
17                      // update tail reference
18                      CAS(tail,t, n);
19                      // enqueue is done
20                      return true;
21                  }
22              } else {
23                  // correct tail reference
24                  CAS(tail,t, next);
25              }
26          }
27      }
28  }
```

Listing 2.4: Algorithm for dequeue operation (Michael and Scott)

```
1   E dequeue() {
2       // keep trying until dequeue is done
3       while(true) {
4           // read head
5           h = head;
6           // read tail
7           t = tail;
8           // read first element
9           first = h.next;
10          // is head consistent
11          if (h == head) {
12              // is queue empty or tail legging behind
13              if (h == t) {
14                  // is queue empty
15                  if (first == null) {
16                      // dequeue not possible
17                      return null;
18                  } else {
19                      // advance tail
20                      CAS(tail,t, first);
21                      }
22              // try moving head pointer to first
23              } else if (CAS(head,h,first)) {
24                  // read the element
25                  E item = first.getItem();
26                  if (item != null) {
27                      // free first element
28                      first.setItem(null);
29                      // dequeue done return element
30                      return item;
31                  }
32              }
33          }
34      }
```

```
35          }
```

2.4.2.1 Critical Analysis

A lock-free algorithm based on CAS works very efficiently as compared to lock based algorithms. It does not require any conservative locking while accessing the globally shared data structure, the queue. Also, it provides a global progress guarantee. However, the efficient functioning of queue depends a lot on the contention level and the multicore topology. Although it provides global progress guarantee, there can be cases where one of the threads will have to wait unfairly long to complete its operations as compared to other threads. This leads to starvation. One such case arises due to data locality.

Let's consider a case where a previous enqueue operation was successfully applied by a thread running on core 1 in a multicore systems. Now, another thread running on core 2 wants to perform an enqueue operation. If, at the same time, thread running on core 1 issues another enqueue operation, intuitively there are high chances that thread running on core 1 will be able successfully perform the enqueue operation using the CAS, whereas, the other thread will fail on CAS.

This happens mainly in a contended systems where the required data has be brought in from the cache line of some other core. The time taken for the thread running on core 1 to access the tail node of the queue, which is locally available to it, will be far less in comparison of the thread running on core 2, for whom it has to be brought in from remote memory (cache-miss). This local-access vs remote-access problem is very much prevalent in NUMA multicore machines [3].

2.4.3 Wait-free Queues

Although, Lock-free concurrent queue algorithms provide global progress guarantee, they do not provide process (thread) level progress guarantee. To overcome this problem, wait-free queue algorithms were proposed. However, as we described in section 2.3.3, constructing a wait-free algorithm is not easy and they

generally do not perform well as compared to lock-free queues.

One of the first wait-free algorithm was proposed by Lamport [1]. He presented a single producer (enqueuer) and single consumer (dequeuer) wait-free circular buffer. Subsequently, this circular buffer can also be used to implement a single producer and single consumer queue. Although, this implementation is wait-free, it limits the concurrency to only one consumer and one producer. Also, since the circular buffer is based on a static array, it limits the number of elements a queue can hold.

David proposed another wait-free algorithm that supports multiple concurrent dequeuers but only one enqueuer. This queue is based on an array which is infinitely large [4]. This requirement makes the algorithm impractical. There have been other proposals but none of them presented a multiple enqueuer and multiple dequeuer queue algorithm.

This first practical multiple enqueuer and multiple dequeuer concurrent queue algorithm is presented by Kogan and Petrank [16]. In this algorithm, faster threads try to help other slower threads in applying their operations. Each operation, enqueue or dequeue starts by choosing a phase number. This phase number is higher than all the phase numbers chosen previously by other threads. The thread performing the operation, records its operation information at a designated position in a state array.

The size of state array is proportional to the number of threads (each thread has a thread id ranging from *0* to *n-1*, where *n* is the maximum number threads which is predetermined). After storing its operation information in state array, thread traverses through the state array. It looks for the pending operation whose phase value is equal or lesser than its own phase value. If found, it completes those operations on behalf of the other threads. These pending operations include its own operation as well. After finishing the help, the thread can be sure that either its operation is applied by the thread itself or by some other thread. Same procedure applies for dequeue operation.

This algorithm has a very complex implementation. Though, it provides the wait-freedom, results shows that it does not perform better than the Michael Scott's lock-free algorithm.

2.4.4 Semantically Relaxed Queues

Despite of all the efforts, previously mentioned algorithms performs badly under high concurrency. This is due to the synchronization bottleneck. Though lock-free algorithms claim to be lock-free, still the CAS primitive in itself is a fine grained lock provided by the hardware.

An altogether different approach can be to allow the relaxation of the sequential specification of a concurrent FIFO queue [15]. The issue with the sequential specification of a concurrent FIFO queue is that it leaves very little space for optimization [22]. A queue has only two access points, head and tail. Therefore, if the number of threads accessing the queue is high, it is natural to have a poor performance and scalability.

One interesting solution in this direction is presented by Krisch *et al* [25]. They systematically change the sequential specification of concurrent data structures for better performance. They introduce k-FIFO queue, which is a relaxed version of strict FIFO queue. A FIFO queue dequeues the element which is in the queue for the longest time. In comparison of that, k-FIFO queue allows dequeue of any element from a queue, which is in the range from oldest to kth-oldest. The implementation of this k-FIFO queue is linearizable according to the relaxed sequential specification.

2.5 Summary

This section summaries the the related work and the relevant study. We have studied that:

- At hardware level, SMP machines provide homogeneous access to all the resources such as memory and I/O. However, SMP machines fails to scale beyond few cores because of the saturation in bus that connects the cores. NUMA machines provide a scalable solution, but non uniform memory access provide different read-write latency for threads running on different machines. This affects thread specific performance for multithreaded applications which run across the nodes in a NUMA machine.

- There are different consistency models. Linearizability provides execution, which is consistent with the sequential execution of a program. It orders the event based on a global time. Whereas, sequential consistency is a weaker form of consistency model. Instead of providing global ordering, it provides program ordering, which is process (or thread) specific.

- At software level, algorithms for concurrent data structures such as a concurrent queue, must provide safety and progress guarantee. Such algorithms that uses coarse grained locks, fail to achieve good performance on a multicore machine. They also do not guarantee progress. To avoid this problem, lock-free algorithms substitutes locks with atomic primitive such as CAS. However, lock-free algorithms only provide a global progress guarantee. A thread in a multithreaded application using lock-free algorithm can be starved for resource. This also depends on the multicore topology.

  Wait-free algorithms provide thread level progress guarantee. But, designing a wait-free algorithm is very complex and they do not perform as good as lock-free algorithms. Under high concurrent accesses to a shared data structure, both lock-free and wait-free algorithms performs badly and do not scale.

Table 2.1 summaries the different properties, performance and scalability of concurrent queue algorithms.

Table 2.1: A comparison of Queues

	Progress Guarantees		Performance under stress			
Queue Type	**Global Progress**	**Local Progress**	**High Stress**	**Medium Stress**	**Low Stress**	**Scalability**
Blocking Queue	No	No	Very Bad	Low	Medium	Negative
Lock-Free Queue	Yes	No	Very Bad	Good	Good	Negative
Wait-Free Queue	Yes	Yes	Very Bad	Good	Good	Negative

3 Scalable and Performance-Critical Queues

Queues are fundamental and ubiquitous in producer and consumer type computing scenarios. The very two fundamental operations a FIFO concurrent queue supports are *enqueue* and *dequeue*. An *enqueue* operation appends an element to the bottom of the queue. Similarly, *dequeue* operation removes the element from head of the queue. To avoid races and faults, access to a concurrent queue should be atomic. When multiple workers (threads) aggressively access the shared data structure such as a queue, performance deteriorates drastically. Not only the performance deteriorates but some of the workers may get easier access to the shared data structure as compared to others. This depends on which core the worker is running and what is the multicore architecture and topology. With the increase in number of workers, this problem gets worse.

In this chapter, we try to solve these problems at software level by designing and implementing two algorithms for concurrent queues which are optimized to work under heavy stress levels. The first algorithm, a *wait-free linearizable Queue*, provides an alternative to *lock-free* algorithms that fail to provide starvation free execution guarantee. The second algorithm, a *sequentially consistent Queue*, relies on relaxing the semantics of queue from *linearizability* to *sequential consistency*. Latter algorithm exploits the trade-off between the performance and correctness.

3.1 Wait-Free Linearizable Queue

3.1.1 Background

Lock-free algorithms for queues are well known and various such algorithms have been proposed in recent past. One such most popular algorithm is Michael and

Scott's non blocking *lock-free* queue, which is considered *state-of-the-art*. In the era of cloud computing where compute is governed by service level agreements (SLAs), it is more important than ever to provide a guarantee that a worker will complete its operations in a bounded number of steps. As mentioned previously, lock freedom only guarantees that at given point of time one of the workers will be able to make process. This may not be acceptable in many scenarios where all the workers or clients require progress guarantee. *Wait-free* algorithms provide such progress guarantees.

Having studied existing *wait-free* algorithms in literature, we have come to a conclusion that designing a simple and efficient *wait-free* algorithm is not easy. As covered in detail in related work sections, one of the most practical *wait-free* algorithm resort to use mutual helping during process execution. If few of the workers are not able to complete their operations, one of the workers which is fast enough, and is able to get hold of the object, helps other waiting workers in completing their operations. However, most of the *wait-free* algorithms are very complex in nature and do not perform better overall in throughput as compared to *lock-free* algorithms. Therefore, while designing this queue, our prime focus is to come up with a *wait-free* algorithm which is practical, simple and yet provides very efficient access to a concurrent queue.

3.1.2 Insight

Our queue is based on an underlying singly *linked list* which holds references to *head* and *tail* of the queue. Our idea extends the notion of mutual help in previous *wait-free* algorithms to an external help provided by a dedicated worker, whose job is to help other threads to complete their operations in a fair manner. Following are the two additions to our *wait-free* queue algorithm.

- **External Helper**: Queue maintains an external helper for queue operations. The workers (enqueuers and dequeuers) do not directly interact with the underlying queue structure which is a *linked list*. Instead, they submit their requests to *Helper* which, in turn, completes their requests by enqueuing or

dequeuing the elements to/from the queue.

- **State Array**: Queue internally maintains a *state array* that acts as a placeholder for incoming requests for *enqueues* and *dequeues*. The size of *state array* is equal to the number of workers in the program. Each position in the *state array* is dedicate to a worker. Whenever a worker needs to do an operation on the queue, it places the request at its designated position in the array.

3.1.3 Algorithm

The basic logic behind the algorithm is simple. Each worker puts its request for *enqueue* or *dequeue* in the *state array* and waits for its operation to be applied to the queue. *Helper* worker traverse through the *state array*, looking for new incoming requests. If a new request is found, it processes it on behalf of the requester. Workers wait until their operations are successfully picked up and applied by the *Helper*.

Queue also maintains an internal *linked list* of nodes where each node contains an element *e* and a pointer to the next node in the list. Queue keeps *head* and *tail* references for the *linked list* and updates them as the queue shrinks or grows. First node in the linked list is a sentinel (dummy) node. It also keeps a counter of total elements in the queue. Listing 3.1 presents the global fields and queue initialization.

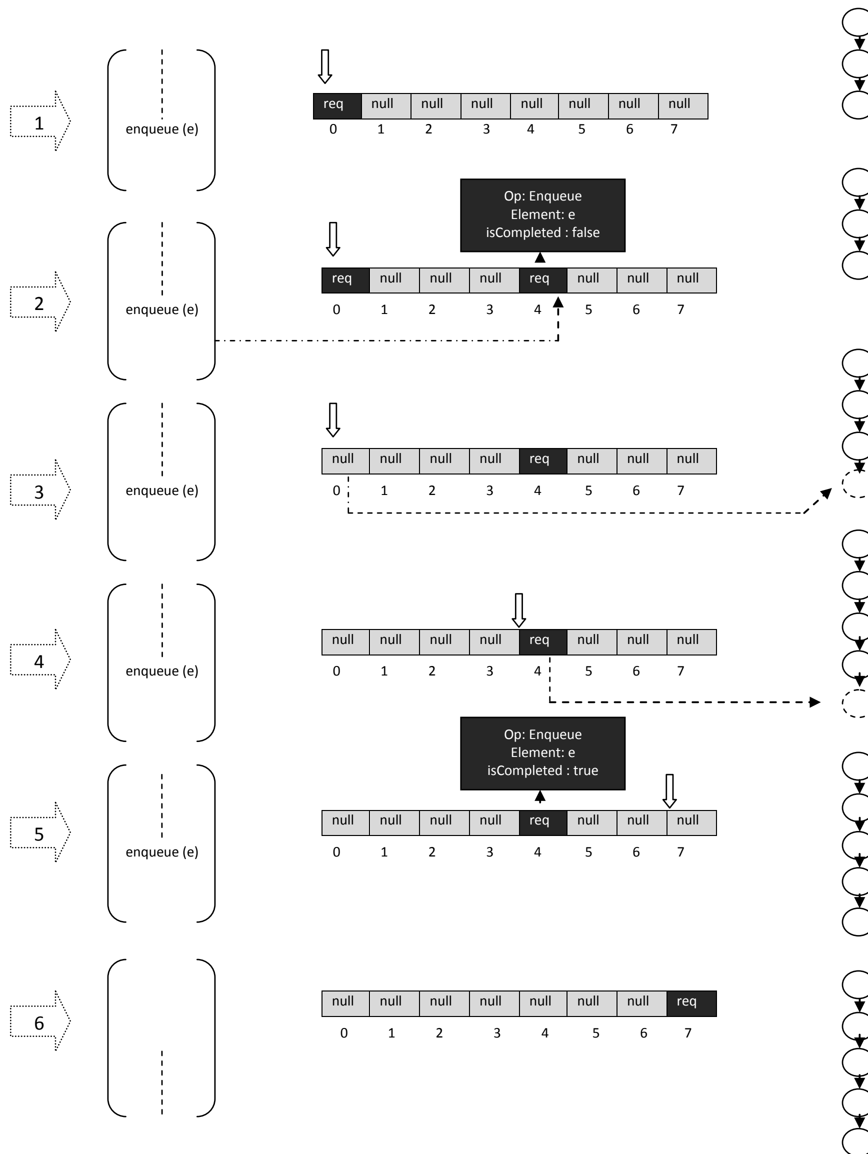
When an application creates an instance of queue, it can specify the number of workers. Accordingly, the *state array* is initialized. Requests are put using the *Request* structure. It contains the *type* of operation, *value* place holder for queue element and a *flag* specifying if the operation is completed. Figure 3.1 depicts an enqueue operation visually.

Listing 3.1: Global fields and queue initialization (Wait-Free Queue)

```
1        //global fields
2        Node head;
3        Node tail;
4        int workers;
5        Request[]  stateArr; // state array
6        int size;
7
8        //initialization
9        initializeQueue(int n) {
10               head = getNewNode(null);
11               tail = head;
12               workers = n;
13               stateArr = Request[n];
14               size = 0;
15               // start the Helper
16               startHelper();
17       }
18 }
```

Following is a brief description of how the enqueue and dequeue operations are performed. Peek is another operation which a queue provides. Peek operation returns the element from the top of the queue without removing it.

- **Enqueue**: Algorithm for enqueue operation is given in Listing 3.2. Worker creates a *Request* with the element e and operation type as ENQUEUE. It then places this request in its designated position in the *state array* and waits for the operation to be picked up by the *Helper*. It continuously checks the status of the *Request* by checking *isCompleted* flag. If it is set true that means the operation has been completed by the Helper. Following that, it clears the request from State Array and method returns as success. This operation never fails, assuming that the *Helper* will always complete the operation.



Bracket on left side shows the worker's flow, state array is in middle and on right side the queue structure is presented. Step 1: worker starts enqueue operation. Step 2: a request is put in the state array. Step 3:Helper applies previous request by worker 0. Step 4: Helper picks the request for worker 4. Step 5: Element is enqueued in queue and isCompleted flag is set to true. Step 6: Request is cleared from state array. Meanwhile a new request arrives from worker 7.

Figure 3.1: A visual example of the algorithmic steps in an enqueue operation for worker with id 4.

Listing 3.2: Algorithm for enqueue operation (Wait-Free Queue)

```
bool enqueue(e) {
        // get the thread id of the worker
        id = getThreadId();
        // create a new request with element e
        req = createRequest(e,ENQUEUE);
        // post this request in state array
        stateArr[id] = req;
        // wait until the operation is completed
        while(!req.isCompleted);
        // clear the request from state array
        stateArr[id] = null;
        // operation is completed, return
        return true;
}
```

- **Dequeue**: Just like the enqueue operation, dequeue also creates a *Request* with empty element field and operation type as DEQUEUE. It places this request in its position in *state array*. The *Helper* threads picks this request and processes it. Once completed, the element field in *Request* structure contains the top element from the queue and *isCompleted* flag is marked as true. When the operation completes, the request is cleared from *state array* and the element is returned. Listing 3.3 presents the algorithm.

- **Peek**: Peek operation works exactly similar to dequeue operation. However, it does not remove the top element from the queue. It reads and return the element, which is in the queue for the longest time. The algorithm for peek operation is presented the Listing 3.4.

Listing 3.3: Algorithm for dequeue operation (Wait-Free Queue)

```
1  E dequeue() {
2          // get the thread id of the worker
3          id = getThreadId();
4          // create request with empty value field
5          req = createRequest(null,DEQUEUE);
6          // put the request in state array
7          stateArr[id] = req;
8          // wait until the operation completes
9          while(!req.isCompleted);
10         // clear the request from state array
11         stateArr[id] = null;
12         //return the top value which has top element
13         return req.e;
14 }
```

Listing 3.4: Algorithm for peek operation (Wait-Free Queue)

```
1  E peek() {
2          // get the thread id of the worker
3          id = getThreadId();
4          // create request with empty value field
5          req = createRequest(null,PEEK);
6          // put the request in state array
7          stateArr[id] = req;
8          // wait until the operation completes
9          while(!req.isCompleted);
10         // clear the request from state array
11         stateArr[id] = null;
12         //return the top element
13         return req.e;
14 }
```

The algorithm for *Helper* is given by Listing 3.5. It traverse through the *state array* in a round robin fashion in a tight loop. If it sees a *Request* from a worker it fulfills it before moving to next index in *state array*. It first reads the operation field in *Request*. If it is an enqueue operation, it creates a new node with the ele-

ment in request field. It then appends the node to the *tail* and updates the *tail* of the *linked list*. In the end it marks the operation as completed (line 11-22).

If it is a dequeue operation, it checks if the *linked list* is empty. If it is empty, it returns null and marks the operation as completed. If the list is not empty, it removes the top element and updates the references to *head* and *tail* (if required) respectively. It changes the element field in *Request* from null to the removed element and marks the flag *isCompleted* as success (line 25-52). Similarly, if it is peek operation, it updates the element field in *Request* with the top element without removing it and marks the operation as completed (line 54-70).

Listing 3.5: Algorithm of Helper (Wait-Free Queue)

```
1  void helper() {
2          //index of state array
3          id = 0;
4          // infinite loop
5          while(true) {
6            // read the request from state array
7            req = stateArr[id];
8            // if there is a request
9            if(req!=null && !req.isCompleted) {
10                 // Enqueue or Offer
11                 if(req.operation == ENQUEUE) {
12                    //create new node with value e
13                    n = new_node(req.e);
14                    // append the node to the tail
15                    tail.next =n;
16                    // update the tail reference
17                    tail = n;
18                    // increase the size of queue
19                    size++;
20                    // mark the request as completed
21                    req.isCompleted = true;
22                 }        // enqueue completes
```

```
23
24              // Dequeue or Poll
25              if(req.operation == DEQUEUE) {
26                // check if the queue is empty
27                 if(head.next == null) {
28                 // queue is empty, update request
29                   req.e = null;
30                   // mark operation as completed
31                   req.isCompleted = true;
32                 }
33                 // queue has elements
34                 else {
35                   // get the top element
36                      n = head.next;
37                      // unlink top element from queue
38                      head.next = n.next;
39                      // queue will be empty after removal
40                      if(n.next==null) {
41                              //update the tail
42                              tail = head;
43                      }
44                      // update the request with element
45                      req.e = n.e;
46                      // decrease the size of queue
47                      size--;
48                      // mark the request as completed
49                      req.isCompleted = true;
50                }
51              } // dequeue completes
52
53              // Peek but not remove
54              if(req.operation == PEEK) {
55                       // check if queue is empty
56                       if(head.next == null) {
57                         // queue is empty
```

```
58                          req.e = null;
59                          // mark operation as completed
60                          req.isCompleted = true;
61                      }
62                      // queue is not empty
63                      else {
64                              // read the top element
65                          req.e = head.next.e;
66                          // mark operation as completed
67                          req.isCompleted = true;
68                      }
69                  } // peek completes
70              }
71              id = incrementId(); //increment index id++%worker
72          }
73  }
```

3.1.4 Discussion on Correctness

We claim that our algorithm implements a concurrent *wait-free* queue, which is *linearizable*, *wait-free* and *safe*. In this section we will cover a brief discussion on wait-freedom, linearizability and thread safety for our algorithm. Table 3.1 presents general safety properties which our algorithm holds. First, we start by discussing the computation model assumed in our design. Subsequently, we prove the wait-free and linearlizability property of our algorithm.

Property	Holds at
Linked list is always connected.	Listing 3.5, line 15 and 38
Node is only inserted at tail.	Listing 3.5, line 15
Node is only removed at head.	Listing 3.5, line 36 and 38
Tail always points to last node.	Listing 3.5, line 17
Head always points to sentinel node.	Listing 3.5, line 36

Table 3.1: General Safety Properties

- **Computation Model**: Our model of multithreaded concurrent system follows linearizability and assumes a shared memory system. Program is executed by n deterministic threads which are predetermined. One important assumption we make is that, the *Helper* runs on a dedicated core and never gets scheduled out. This assumption is made to receive maximum computational power for the *Helper*. However, the other threads can schedule out at any time. Thread scheduling is performed solely by operating system's scheduler. We also assume that each thread has a unique thread ID whose value is between 0 to n-1 where n is total number of threads. The id of *Helper* is n. A thread can perform series of operations (eneuque or dequeue) in program order and the thread waits until its operation is completed.

- **Linearizability**: An operation is linearizable, if it happens to be applied atomically at some point between the invocation and the response of the operation. For a queue to be linearizable, the enqueue operation where a worker enqueues an element e, has to be placed in the queue before the call returns to the worker. In our case, a worker places its request in the *state array*, and the call for enqueue only returns, when the element is placed on the *tail* of the queue. Similarly, the call for dequeue operation finishes, when the element has been successfully removed and return from the *head* of the queue. Linearization point for enqueue and dequeue are at line 15 and 36 (Listing 3.5) respectively.

- **Wait-Freedom**: Wait-freedom comes from the fact that an operation completes in a bounded number of steps. If we consider the number of workers predetermined and known, we can instantiate the size of *state array* to a size proportional to the number of workers. In our case, any enqueue or dequeue operation will complete in maximum n steps, given there are n workers. This is so, because *Helper* iterates over the state array. Let's consider the worst case, where a worker T*i* places a request in *state array* when the pointer of *Helper* is at location *i+1*. Now, the helper needs to complete the whole round by jumping *n* places before it comes back to location *i*. Therefore, the upper bound for enqueue and dequeue operation is *n*, which is fixed.

- **Thread-Safety**: We claim that our queue implementation is thread-safe, which means multiple threads can access the queue without the need of external synchronization. The thread-safe program execution in a multi-threaded environment is essential for any concurrent data structure. Since, at any given point of time, only one request is served by the *Helper*, there is no possibility of race conditions among threads. This abstraction helps programmers to not worry about the race conditions and synchronization issues which are covered in the internal implementation of queue. This makes programming easy and yet fault-proof.

3.2 Sequentially Consistent Queue

3.2.1 Background

It is practically impossible to provide a very fast FIFO Queue implementation that works on large multicore systems under stressed conditions. However, performance can be achieved by weakening the consistency. In our design, we explore the possibility of changing the consistency guarantees from linearizability to sequential consistency. There are cases such as graph exploration where strict ordering guarantees are not required, however, applications are very performance critical. In absence of any weaker practical implementations of queue, programmers either resort to use linearizable queue or use some complex algorithmic techniques for better performance. Some of such techniques includes graph partitioning and work stealing.

3.2.2 Algorithm

This algorithm is an optimized enhancement to our previous *wait-free* algorithm. The basic structural details of this algorithm remains similar to our previous design. Additionally, this queue maintains an internal *local linked list* structure for each worker. On enqueue, an element is appended to the local *linked list*. This local list is exclusive to the worker. Therefore, there is no possibility of contention and race among workers. This allows very fast enqueue operation. Queue also maintains a *global linked list* structure where workers perform dequeue operations. However, this global structure is not directly accessed by the workers. Instead, they submit their request to *Helper* with the help of *state array* and wait until the operation is completed.

The *Helper* continuously checks for dequeue requests in *state array*, and if found it processes them. The *Helper* also, in the background, periodically merges the *local lists* with the *global list*. The *Helper* freezes (locks) the local linked list for a worker when it merges it with *global linked list*. This is done to avoid updating incorrect or stale references by enqueue operation. Following, is the discussion on how enqueue and dequeue operations work for this *sequentially consistent* queue.

- **Enqueue** The algorithm for enqueue operation is presented in the Listing 3.6. The enqueue operation starts by setting the lock for enqueue operation so that *Helper* does not start merging its *local list* with *global list* at the same time. Once the lock is acquired, it retrieves the *local list*, creates a node with the element and appends to the tail of the local list.

- **Dequeue** Dequeue operation is performed by putting the dequeue request in the *state array*. Steps involved in dequeue operation are shown in Listing 3.7. Operation starts by creating a *Request* for dequeue operation with empty element. This request is put in its designated position in the *state array*. The worker waits until the operation is completed by the *Helper*. It continuously checks for *isCompleted* flag in the *Request* structure. If completed, it clears its request from *state array* and returns the element which is updated in *Request* structure.

Listing 3.6: Algorithm for enqueue operation (Sequentially Consistent Queue)

```
1  bool enqueue(E e) {
2          //get thread (worker) id
3          id = getThreadId();
4          // try to freeze the local linked list
5          while(!lock[id].CAS(false, true));
6          // read the tail of local linked list
7          tail = localTails[id];
8          // it should not be null: error
9          if(tail == null)
10                 return false;
11         // append new element to tail
12         tail.next = new_node(e);
13         // update the tail to new element
14         localTails[id] = tail.next;
15         // unfreeze local linked list
16         lock[id].set(false);
17         // operation completed
18         return true;
19 }
```

Listing 3.7: Algorithm for dequeue operation (Sequentially Consistent Queue)

```
E dequeue() {
        // get worker (thread) id
        id = getThreadId();
        // create new dequeue request
        req = createRequest(DEQUEUE);
        // put the request in state array
        stateArr[id] = req;
        // wait until helper completes the operation
        while(!req.isCompleted);
        // operation completed, clear request
        stateArr[id] = null;
        // return the dequeued element
        return req.e;
}
```

One of the important tasks of the *Helper* is to merge the *local linked lists* with the *global linked list*. How *Helper* performs this operation is shown in Listing 3.8. *Merge()* for each worker is called periodically by the *Helper*. The period of merge operation can be configurable. In our case, *Helper* performs merge, when it visit the worker's position in *state array* looking for dequeue requests. This is done to avoid a case, where the *global linked list* is empty, even though there are elements in *local linked list*.

To merge, *Helper* retrieves the local *head* and local *tail* reference for a worker. It then compares the local *head* and local *tail* reference to check if there are elements in the local linked list. If there are elements in the local linked list, it freezes the list for the worker and joins it with the global linked list. If the local linked list is already freezed for enqueue, it waits until the list is unfreezed. We should note, the merge is performed just by updating the references. *Tail* of *global linked list* is joined by the *localHead.next* (first node after the sentinel node) and global *tail* is updated to the local *tail*.

Once the *local linked list* is merged with the *global linked list*, it should not be directly accessible to worker for enqueue operation. Therefore, local linked list is

cleared by updating the *head* and *tail* for *local linked list*. All the steps in merge process are finite. Once the merge completes, the *Helper* unfreezes the *local linked list* which has no element in it. The workers keep on enqueuing in the local linked list until the merge happens. *Helper* also processes the dequeue requests from workers. However, we have not described it here as the procedure is exactly similar to the one shown in *wait-free* linearizable queue.

Listing 3.8: Algorithm for merge operation (Sequentially Consistent Queue)

```
1  void merge(id) {
2          // read head of local linked list
3          localHead = localHeads[id];
4          // read tail of local linked list
5          localTail  = localTails[id];
6          // to merge, list should not be empty
7          if(localHead!=localTail){
8                  // try to freeze the local linked list
9                  while(!lock[id].CAS(false, true))
10                 // update global tail with local head.next
11                 tail.next = localHead.next;
12                 // update global tail to local tail
13                 tail = localTail;
14                 // clear local linked list
15                 localTails[id]=localHeads[id];
16                 // unfreeze the local linked list
17                 lock[id].set(false);
18         }
19 }
```

3.2.3 Discussion on Correctness

As we have mentioned previously, the motivation behind this algorithm is to provider faster operations on queue in a multithreaded environment. We semantically relax the queue by changing the consistency model from linearizability to sequential consistency. In this subsection, we will discuss, why this algorithm

provides the sequential consistency. We will first briefly discuss the guarantees provided by sequential consistency model. A detailed discussion on the same is done in related work section.

Sequential consistency guarantees the program order. The order in which a worker (thread) issues operations is called program order. This order is unrelated to how and when operations are issued by different threads. If, on a linear time scale, a worker A performs *queue.enqueue(x)* at time 1 and worker B performs *queue.enqueue(y)* at time 2, it is possible that at time 3 when worker performs *queue.dequeue()* operation, it may returns y. This type of behavior is acceptable in program order and is sequentially consistent. This behavior might look unjustified as x was enqueued first and it should have been dequeued first. However, there are applications which do not require strict ordering guarantees as long as the program order is followed.

We claim that our algorithm is sequentially consistent but not linearizable. This is so because *enqueue()* call returns as soon as the element is appended in the local linked list. *Helper* does not guarantee which local list will be merged first, therefore, it is possible that an element, enqueued at a later time by different worker, may be dequeued first by some other worker. This violates the linearizability property. Nonetheless, our program still follows the program order as two elements enqueued by the same worker will be dequeued in same order irrespective of who performed the dequeue operation. Our algorithm preserves this property because the local ordering is maintained when a worker issues enequeue operations in *local linked list*.

3.3 Implementation

We have implemented these algorithms in Java. There are few implementation specific details, we would like to discuss in this section. Following are few of the implementation specific issues:

- **Volatile Variables**: On a multicore machine, each core has its own local cache. Therefore, a global variable also has cached copies at each core. If this variable is accessed concurrently by different threads running on different cores, there are chances that some of them may still see the old value available in local cache. This is particularly important in our implementation of queue, because, there are many internal variables which are accessed by different threads. This problem can be avoided using the *volatile* primitive of Java. Marking a field as *volatile* makes sure that every time a thread running on a core accesses the variable, it read the copy from main memory instead of local cache of that core [21].

  The *state array* and the request structure is accessed by different threads simultaneously, therefore, to avoid races we have set them *volatile*. Similarly, in case of sequentially consistent queue, *localHead* and *localTail* are accessed frequently by both the *Helper* thread and the workers. Therefore, these variables are also marked as volatile.

- **False Sharing**: Memory management on multicore generally provides cache coherence. False sharing is a phenomenon that occurs when threads running on different cores modifies different memory location that resides on same cache lines [29]. Cache coherence protocol forces whole cache line to be invalidated, even if different threads were accessing different memory locations in the cache line.

  In our implementation, this problem is natural to occur, because the entries in *state array* are accessed by threads running on different cores simultaneously. We have avoided this problem, by separating the entries of each thread in *state array* by 8 times. The other entries in between two correct entries are dummy place holders. This padding allows all the thread specific entries to fall into different cache line, thereby avoid the false sharing among core.

- **Memory Management**: We create *node* object for queue's internal *linked list* on *Helper* Thread. Alternatively, this could have been done inside the *enqueue* method call. If we have chosen the latter approach, *linked list* would

have its node residing on the main memory of different nodes of a multicore. This will make the *linked list* structure spanned across the multicore memory. It can cause extra overhead for garbage collector for memory management. Also, it will cause more remote accesses during the *dequeue* operation which is performed by the *Helper*. Instead, the *Helper* creates the node objects, which in most likelihood will remain on same node of multicore during its execution. This allows, the *linked list* structure to remain available locally for *Helper*, which increases the cache performance.

3.4 Analysis

We have presented two new algorithms for a wait-free linearizable concurrent queue and a sequentially consistent concurrent queue. However, there are few open questions which we have not addressed in our design and implementation. We have assumed that the *Helper* never stops and continues to help other workers. Though, it is practically possible to achieve this behavior, the *Helper* becomes single point of failure. If *Helper* thread stops or crashes, the queue becomes inaccessible. The more sound solution will be to have a pool of *Helpers* or at least a reserved backup *Helper* so that in case something goes wrong with the primary *Helper*, others can take over. We leave this enhancement as a future work. This enhancement will provide fault tolerance to our implementation.

We also assume a dedicated core for *Helper*. This is done to provide full computational power *Helper*. In case, the *Helper* does not get enough CPU, it will impact the performance of enqueue and dequeue operations. For example, if *Helper* gets only 50 percent CPU, it will increase the operation time by a factor of 2.

In wait-free algorithm, we are using single *state array* and single *Helper* for both enqueue and dequeue operations. Since, both the operations are done on different end of the queue, it can be advantageous to have separate *state array* and separate *Helper* for enqueue and dequeue operation. This will further increase the performance by a factor of 2. However, this will require 2 dedicated threads, one

of enqueue *Helper* and one for dequeue *Helper*. It will incur an extra computational overhead. We did not explore this trade-off between the performance and overhead computation in our implementation.

4 Evaluation

In this chapter, we evaluate the performance and scalability of both of our algorithms, the wait-free linearizable queue (WF Queue) and the sequentially consistent queue (SC Queue). We compare the performance and scalability of these algorithms with *state-of-the-art* lock-free queue algorithm presented by Michael and Scott (discussed in section 2.4.2).

4.1 Experimental Setup

We implemented both the algorithms in java in a controlled configurable environment. We have also implemented a benchmark framework for experiments. All the experiments are run on a 48 core NUMA machine with x86 64 bit architecture running Linux with kernel 3.0.0. The machine has 4 sockets and 8 nodes where each node has 6 cores. All the micro benchmarks are taken in a careful manner in order to minimize the noise. Our evaluation framework emulates a multi worker (producer-consumer) workload where each worker is allowed to perform both enqueue and dequeue operations.

To eliminate external influences such as migration of threads from one core to another and to avoid running multiple threads on same core, each worker is made to run to a separate dedicated core (using core affinity). Similarly, the *Helper* also runs on a separate core. This allows workers and the *Helper* to fully utilize the computational power available in machine. Each experiment is run 4-5 times and an average is taken. Standard deviation was minimal and is therefore, not shown for better readability. There are many different ways to evaluate the performance and the scalability of a queue. However, we have selected three benchmarks which we find relevant and important for the evaluation of our al-

gorithms. Following are the benchmarks used.

- **Enqueue-Dequeue pair**: The queue is initially empty. At each iteration, each worker enqueues a random integer in the queue followed by a dequeue operation. This emulates the 50-50 percent producer-consumer workload under the stressed conditions. Each experiment performs approximately 2.5 million operations divided equally between all the workers. The number of maximum workers is 32 which is unchanged in all the experiments. Helper runs on a separate core.

- **Think-Time**: Think-Time is nothing but the work done by the worker between the two operations. We measure the performance of our algorithms as a function of increasing Think-Time. Ideally, with the increasing Think-Time contention in the system should decrease. Think-Time is presented in number of cycles. Each experiment performs approximately 2.5 million operations divided equally between all the workers. The number of maximum workers are 32.

- **Application**:We also evaluate the performance of our wait-free queue and sequentially consistent queue for graph exploration and compare it with lock-free queue. Graph is represented as connected nodes, where a node in graph only knows about its neighbors. Graph contains 8.3 million nodes. We evaluated the performance on 16 and 32 cores respectively.

4.2 Performance and Scalability

4.2.1 Per operation Completion time

In this experiment we have measured average per operation completion time for each of the three algorithms, lock-free (LF) Queue, wait-free (WF) Queue and sequentially consistent (SC) Queue under very high contention. Results are shown in Figure 4.1. Under such stressed conditions, is it important to known how long

does each operation takes to complete. As we can see from the graph, it is evident that our algorithms takes far less time as compared to LF queue. With the increasing number of workers, per operation time increases for all the Queue implementations. For LF queue, by doubling the number of workers, per operation time increases by approximately 4 times, however, for WF queue and SC queue it increases approximately twice which is understandable.

With 32 workers, one single operation in LF queue takes as much as 0.1 ms. This converts to 300000 cycles of work, which LF queue does in a single enqueue or a dequeue operation. In comparison to that, our WF queue takes 9 times lesser time and our SC queue takes 16 times lesser time. SC queue perform better than WF queue because of weaker consistency semantics. If we increase the number of workers, this performance gap between LF queue and WF queue increases.

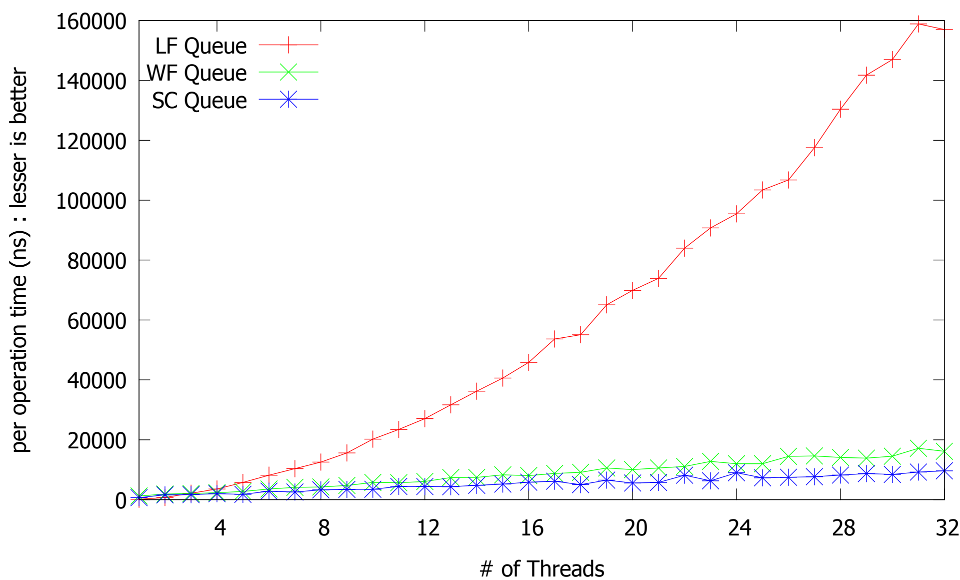


Figure 4.1: A per operation completion time comparison with the increasing number of threads (workers)

4.2.2 Total Execution Time

The performance of an application depends on how fast it completes a given task irrespective of how workers perform it individually. In this experiment, we eval-

uate the time taken to complete the whole execution of the program. Results are shown in Figure 4.2. We can see from the graph that upto 2 threads (workers), WF Queue and SC Queue take more time to complete the task. However, for LF queue, as the number of workers increase, the total execution time increases drastically. This depicts the worst kind of negative scalability and goes against the very purpose of parallel programming. On the contrary, WF depicts some positive scalability, where total execution time either remains stable or decreases. SC Queue performs best, where its total execution is least among all. Also, similar to WF Queue, it depicts positive scalability. With 32 workers, WF queue performs approximately 10 times faster and SC Queue performs approximately 15 times faster.

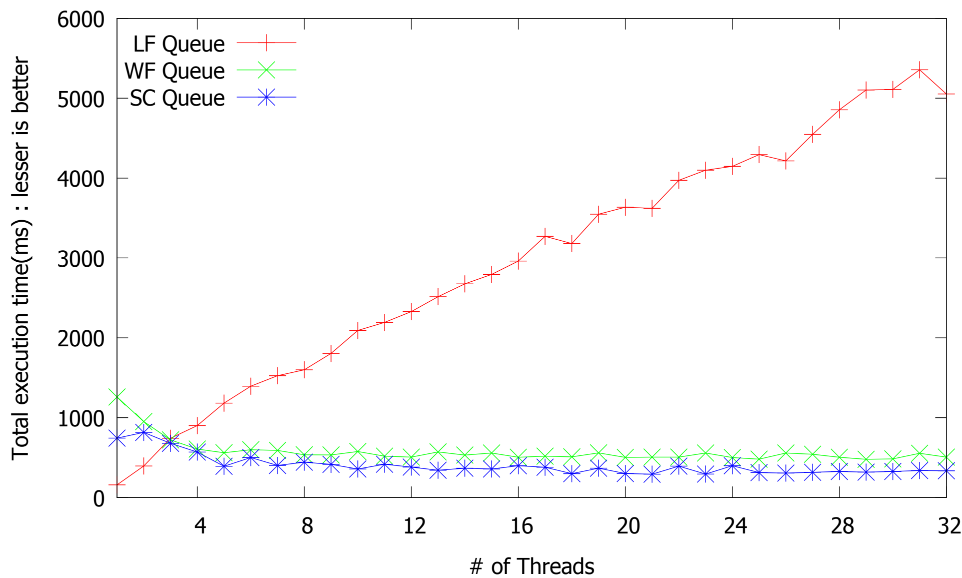


Figure 4.2: A total execution time comparison with the increasing number of threads (workers)

4.2.3 Throughput

Scalability and performance of a queue also depends on how many operations it can successfully perform in a unit time. In this experiment, we show the throughput as a function of increasing number of workers. Results are presented in Figure 4.3. In this case also, LF queue depicts negative scalability and the throughput

decreases with increasing number of workers. WF Queue and SC queue shows sharp positive scalability upto 6 workers, after which throughput remains stable with the increasing number of workers. However, in comparison with the LF Queue, throughput remains approximately 10 times more for WF queue and 15 times more for SL queue. This higher throughput directly translates to better performance for an application using the concurrent queue.

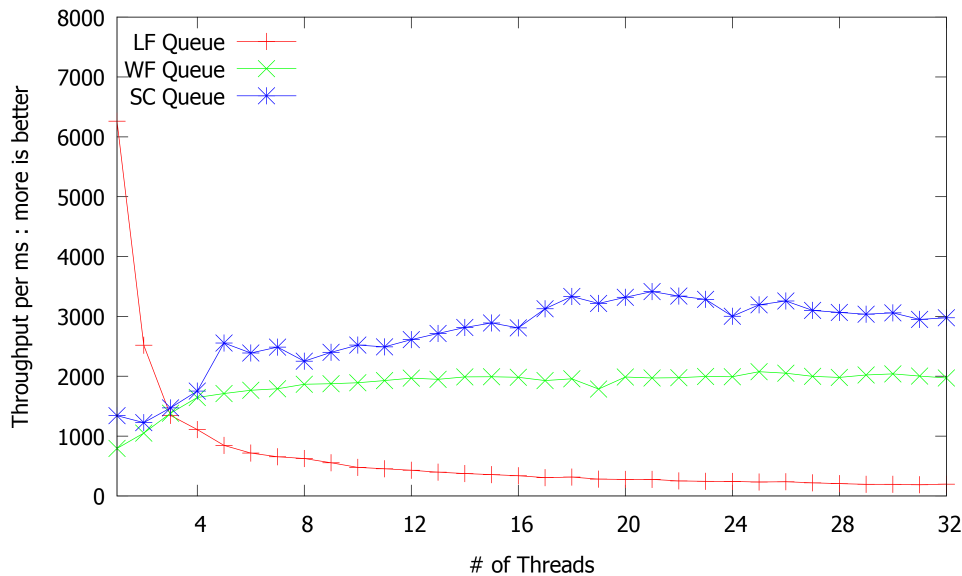


Figure 4.3: Throughput comparison with the increasing number of threads (workers)

4.2.4 Think Time

In real scenarios, applications do some work between two operations. For some applications, this work can be computationally intensive, for others it can be computationally low. We call this work as think time. In this experiment we compare the performance of the Queues with increasing think time. In our experiment, it ranges from 0 to 1 million cycles. The number of workers are fixed to 32. Helper runs on a separate core. Results are shown Figure 4.4. Up to 1000 cycles of think time, per operation time remains stable for all three queues. Also, in comparison with LF queue, WF queue and SL queue take 10 to 15 times lesser time to complete an operation. After 10,000 cycles of think time, contention starts to reduce

and performance improves for all three algorithms. Nonetheless, our algorithms still perform significantly better in comparison of LF queue.

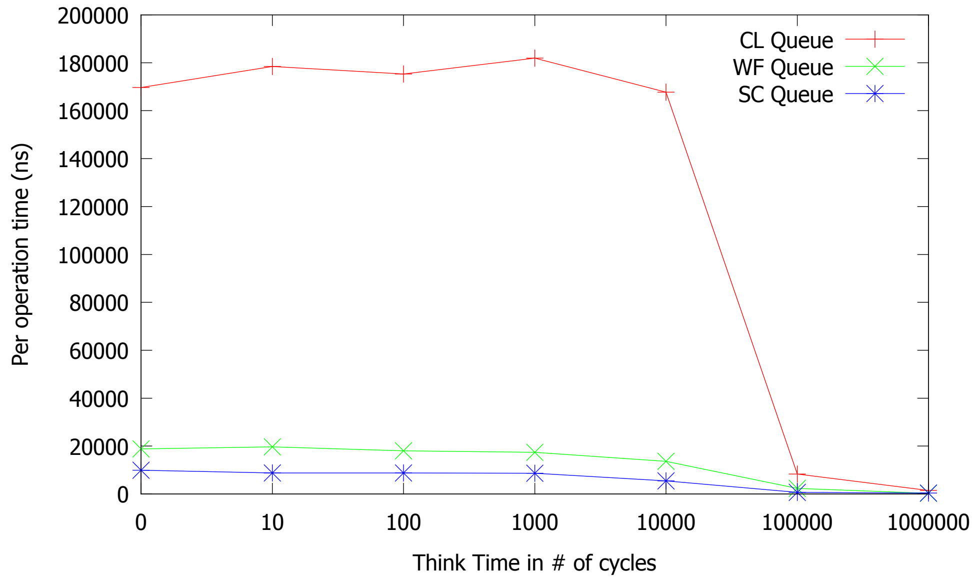


Figure 4.4: Performance comparison with increasing Think Time (work done between two operations)

4.2.5 Fair Share

A chain is as strong as its weakest link. For a better multithreaded program, it is necessary that all the workers perform equally well. Overall program completion time depends on the last finishing worker. Even if, a single worker takes very long to complete, the whole job gets delayed. Therefore, wait-freedom becomes crucial in such scenarios. It guarantees that all the workers get equal opportunity to access the shared data structure so that they will not have to wait for more than a fixed number of steps to apply their operations.

In this experiment, we compare LF queue with WF queue, to check how long does each worker take to finish its share of operations. Results are shown in Figure 4.5. As we can see, in LF queue, workers do not take same time to complete their operations. Whereas, in case of WF queue, all of the workers preciously takes equal amount of time to complete their share of operations.

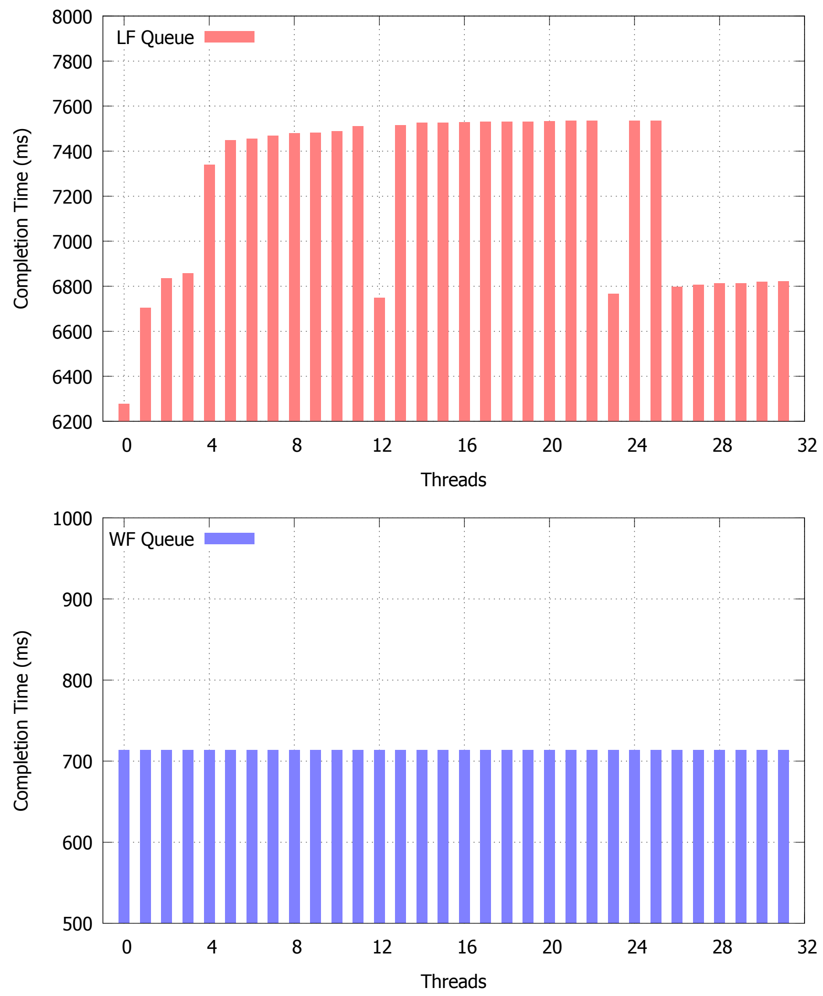


Figure 4.5: Worker completion time in a pool of 32 workers.

4.3 Graph Exploration

Queues are used in many graph exploration algorithms such as breath first search (BFS), level order traversal, spanning trees and topological sorting [6]. We have evaluated the performance of our wait-free queue and sequentially consistent queue by using them in a graph traversal. We have also compared them with lock-free queue.

The graph has approximately 8.5 million nodes. Our graph exploration visits a node and atomically marks it visited. If a worker, during the exploration, finds a node which is already visited by some other worker, it ignores it. At the end of execution, the algorithm has explored all the nodes and returns a count of the total size of graph. The exploration has used 16 and 32 workers (threads) in different executions.

Figure 4.6 presents the results. On 16 cores, in comparison of WF queue and SC queue, LF queue takes 2.5 and 3.75 times more time to complete the graph exploration. Similarly, on 32 cores, LF queue takes approximately 5 and 7.5 times more time. This shows that, for LF queue, as the number of cores (workers) will increase, the execution time of graph exploration (with same number of nodes) will increase exponentially. In case of WF queue and SC queue, it slightly decreases. Ideally, with the desired linear scalability, the execution time should decrease proportionally. However, graph exploration is not an embarrassingly parallel problem. Therefore, we must also account for synchronization such as the operations on shared concurrent queue used by the exploration. Even though, the graph exploration time do not decrease significantly by using WF queue and SC queue, there is a possibility to do computations during the exportation, that can be done in parallel.

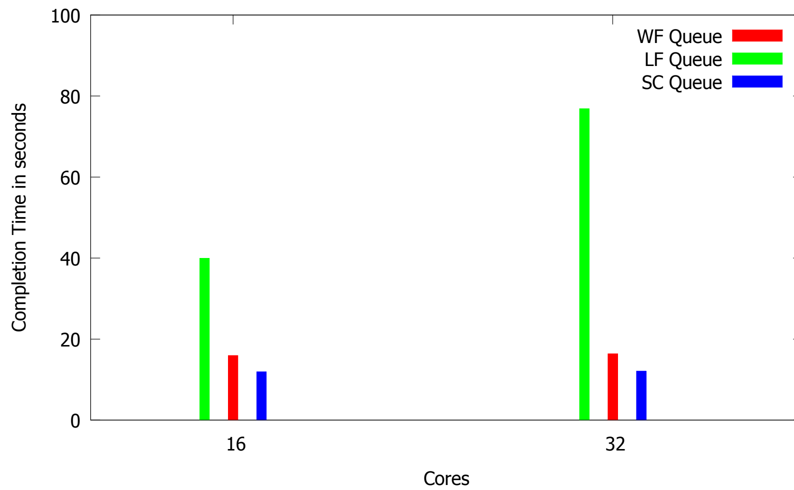


Figure 4.6: Graph exploration completion time on 16 and 32 cores respectively.

4.4 Analysis

A queue is a fundamental structure which has a variety of usages. There are many wait-free and lock-free queue implementations but none of them provide positive scalability under high concurrency. As we have seen in our results, one of the best known implementation of concurrent queue, fails at achieving good results with the increasing number of workers. With our algorithms, we have presented very simple, yet highly efficient queue implementations. Our algorithms are highly efficient and do not depict negative scalability.

High performance and wait freedom both are necessary properties required for next generation softwares running on multicore machines. Wait-free algorithms are gaining momentum and are being considered a replacement for lock-free algorithms. Yet, we did not find any wait-free algorithm in literature for queues, which can perform 8 to 10 times better than lock-free algorithms. We have achieved this performance gap with just 32 workers. If, we increase the number of workers, this gap will further increase. We believe that this is the first time an algorithm is able to provide wait-freedom and high performance at the same time. Similarly, we also did not find any sequentially consistent version of a queue. It is a known fact that high performance can be achieved by compromising strict notions of correctness. Yet there was no implementation, which exploit

the trade-off between the performance and correctness in terms of the consistency model.

We believe that there are many applications which do not require strong ordering guarantees and will function correctly with the weaker semantics, such as sequential consistency. For example, we tested our sequentially consistent queue algorithm for graph exploration, which uses queues internally. Graph exploration is widely used in social networks, computer networks and web crawling. Results shows that our algorithms reduce total exploration time significantly. We also, performed graph exploration using our wait-free algorithm. In this case also, our algorithm exhibits high performance.

Another application of a sequentially consistent queue can be a network switch or a router. Core routers and switches handle millions of packets per second. They use queues internally for header processing and forwarding. Generally, a switch or a router is connected to other switches and routers from whom it receives the packets. In this case, it does not need to provide global time ordering for the packets arriving from different connected routers and switches. However, it must provide ordering among the packets, which are coming from same switch or a router. In such scenario, sequentially consistent queue becomes invaluable.

There are other applications, which require FIFO guarantees and high performance for all the workers. Few of such applications are online ticket reservation systems, online inventory transactions (e-shopping) and banking systems. These systems, generally use backend applications running on a huge multicore machines, where client's requests are handled by different threads. In this case, it becomes necessary to provide a certain quality of service for each client, irrespective of which thread handles the client's request. In these systems, our wait-free algorithm can provide substantial performance benefits and local progress guarantees, without compromising the FIFO ordering.

5 Conclusion

This thesis summarizes my work on the scalability and performance of concurrent data structures on multicore machines. We showed that it is possible to implement a wait-free algorithm that can perform better than a lock-free algorithm. We also showed that it is possible to exploit the trade-off between the performance and the notion of correctness. We presented our algorithms for concurrent queue which is one of the widely used basic data structure. Our wait-free queue outperforms existing *state-of-the-art* Michael and Scott's lock-free queue while providing stricter local progress guarantee for each worker. We further achieved high performance by relaxing the FIFO ordering of a queue. Both of our algorithms also depict positive or stable scalability, which was non existent in previous algorithms.

In our algorithmic design, we introduced the concept of external *Helper*. This concept can be applied to other concurrent data structures such as stack, skip-lists and linked-lists. For example, a stack is very similar to a queue. The only difference is the order in which elements are inserted and removed. Queue provides FIFO ordering while stack provides LIFO ordering. We believe that our techniques and algorithms will become highly beneficial in future, as the number of cores continues to grow on multicore machines.

Table 5.1 summaries the comparison of different properties, performance and scalability of previous concurrent queue algorithms with our algorithms.

Table 5.1: A comparison of previous Queue algorithms with our Queue algorithms

Queue Type	Global Progress	Local Progress	High Stress	Medium Stress	Low Stress	Scalability
	Progress Guarantees		Performance under stress			
Blocking Queue	No	No	Very Bad	Low	Medium	Negative
Lock-Free Queue	Yes	No	Very Bad	Good	Good	Negative
Wait-Free Queue	Yes	Yes	Very Bad	Good	Good	Negative
Our wait-Free Queue	Yes	Yes	Good	Very Good	Good	Positive or Stable
Our Sequentially Consistent Queue	Yes	No	Very Good	Very Good	Good	Positive or Stable

5.1 Future Work

At present, our algorithms do no provide fault tolerance. As we mentioned previously, if the *Helper* dies, it becomes impossible to access the qeueue. In future, we would like to implement fault tolerance by replacing the *Helper* with a pool of *Helpers*.

We would also like to implement our techniques for other data structures. We are presently working on replicated maps. Maps are highly used and they do not scale on a multicore machine. For high scalability, we are replicating the map on each core. We are also utilizing the concept of *Helper*, replication, commutativity and eventual consistency for better performance. We find this work interesting as the future demands a radical change in software development on multicore machines.

Bibliography

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Euro-Par 2012 Parallel Processing*, pages 662–673. Springer, 2012.

[2] Gene M Amdahl, Gerrit A Blaauw, and FP Brooks. Architecture of the ibm system/360. *IBM Journal of Research and Development*, 8(2):87–101, 1964.

[3] W Bolosky, R Fitzgerald, and M Scott. Simple but effective techniques for numa memory management. In *ACM SIGOPS Operating Systems Review*, volume 23, pages 19–31. ACM, 1989.

[4] Matei David. A single-enqueuer wait-free queue implementation. In *Distributed Computing*, pages 132–143. Springer, 2004.

[5] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2):9–21, 1988.

[6] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.

[7] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[8] David Gifford and Alfred Spector. Case study: Ibm's system/360-370 architecture. *Communications of the ACM*, 30(4):291–307, 1987.

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

[10] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[11] Maurice P Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 276–290. ACM, 1988.

[12] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[13] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[14] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

[15] Christoph M Kirsch and Hannes Payer. Incorrect systems: it's not the problem, it's the solution. In *Proceedings of the 49th Annual Design Automation Conference*, pages 913–917. ACM, 2012.

[16] 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*, pages 223–234. ACM, 2011.

[17] Edya Ladan-Mozes and Nir Shavit. *An optimistic approach to lock-free fifo queues*. Springer, 2004.

[18] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[19] Doug Lea. *Concurrent programming in Java: Design principles and pattern*. Addison-Wesley Professional, 2000.

[20] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[21] Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*, volume 40. ACM, 2005.

[22] 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*, pages 267–275. ACM, 1996.

[23] Mark Moir and James H Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.

[24] 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*, pages 253–262. ACM, 2005.

[25] Hannes Payer, Harald Roeck, Christoph M Kirsch, and Ana Sokolova. Scalability versus semantics of concurrent fifo queues. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 331–332. ACM, 2011.

[26] Robert R Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[27] Nir Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, 2011.

[28] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[29] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *Computers, IEEE Transactions on*, 43(6):651–663, 1994.

[30] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143. ACM, 2001.

[31] Ronald C Unrau. *Scalable memory management through hierarchical symmetric multiprocessing*. PhD thesis, Citeseer, 1993.