A Comparison of Concurrent Correctness Criteria for Shared Memory Based Data Structure

2016

Dipanjan Bhattacharya
University of Central Florida

A COMPARISON OF CONCURRENT CORRECTNESS CRITERIA FOR SHARED MEMORY BASED DATA STRUCTURES

by

DIPANJAN BHATTACHARYA
B.Tech. Guru Gobind Singh Indraprastha University, 2013

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2016

Major Professor: Damian Dechev

ABSTRACT

Developing concurrent algorithms requires safety and liveness to be defined in order to understand their proper behavior. Safety refers to the *correctness criteria* while liveness is the *progress guarantee*. Nowadays there are a variety of correctness conditions for concurrent objects. The way these correctness conditions differ and the various trade-offs they present with respect to performance, usability, and progress guarantees is poorly understood. This presents a daunting task for the developers and users of such concurrent algorithms who are trying to better understand the correctness of their code and the various trade-offs associated with their design choices and use. The purpose of this study is to explore the set of known correctness conditions for concurrent objects, find their correlations and categorize them, and provide insights regarding their implications with respect to performance and usability. In this thesis, a comparative study of Linearizability, Sequential Consistency, Quiescent Consistency and Quasi Linearizability will be presented using data structures like FIFO Queues, Stacks, and Priority Queues, and with a case study for performance of these implementations using different correctness criteria.

I dedicate my thesis to my family and many friends. A special feeling of gratitude to my loving parents, Madhumita and Dipesh Bhattacharya whose words of encouragement and push for tenacity ring in my ears. My elder sister Monisha has never left my side and is very special.

ACKNOWLEDGMENTS

I wish to thank my committee members who were more than generous with their expertise and precious time. A special thanks to Dr. Damian Dechev, my committee chairman for his countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process. Thank you Dr. Gary Leavens and Dr. Mostafa Bassiouni for agreeing to serve on my committee.

I would like to acknowledge and thank my school division for allowing me to conduct my research and providing any assistance requested.

Finally I would like to thank my lab members for providing valuable input for my research and constant help with all the improvements that were made during the course of the research work. I would like to specially thank Christina Peterson for constantly helping me use the tools required for conducting the research and all the hours spent on reading and correcting the thesis. I would also like to thank Ramin Izadpanah for helping with the use of PAPI in my research, Deli Zhang for providing some of the materials used for research, and Kishore Debnath for all the hours spent on discussing concurrency that helped invoke new ideas for research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

With the introduction of multi-core processors, the programming paradigm has changed towards multi-threading and shared memory access. Multiple threads accessing shared memory in a system requires proper analysis and development of algorithms that prevent data races and deadlocks. In the past locks have been used to solve this problem, but due to their poor scalability, non-blocking algorithms [1] are now replacing the lock based algorithms. Analysis of these algorithms requires us to define the *safety* and *liveness* for these concurrent algorithms.

In this thesis, our focus will be on the safety, or correctness, of concurrent objects used to develop shared memory based data structures, and the different criteria that have been developed to solve this problem. The notion of correctness criteria was developed to analytically reason about whether a concurrent object is behaving correctly or not.

Concurrent data structures are expected to maintain the same properties as sequential data structures. Therefore, all of the correctness criteria try to verify the correctness by comparing a concurrent object with a sequential object. This ensures that the semantic meaning of the data structure is not violated and hence provides the validation that the concurrent implementation is correct. With interleaving of threads, it is often not possible to directly compare the concurrent objects with their respective sequential counterparts. This has led to the development of various correctness criteria like *Sequential Consistency*, *Linearizability*, *Quiescent Consistency* and *Quasi Linearizability*. Each defines different set of rules, which establish when a concurrent execution is considered correct. These rules have implications on performance and usability that is difficult to analyze. No previous thesis compares all the correctness criteria mentioned in this thesis, due to the vast difference in definition and implementation of the correctness criteria. However, with the growth of concurrent programming, it becomes necessary to analyze the different correctness criteria men-

tioned and their correlation. These correctness criteria are defined using a theoretical approach to modify existing data structures to follow different correctness criteria, followed by a case study on their performance analysis.

The case study is performed for several data structures like FIFO Queues, Stacks and Priority Queues. These are selected for the case study, since they already have an existing implementation following one of the correctness criteria mentioned above. In the case study, we use the existing implementation of the non blocking data structure as a basic implementation, and then modify the implementation to follow the different correctness criteria. This approach is taken as long as a valid and feasible implementation is possible under all possible correctness criteria being studied in this thesis. After all possible valid and feasible implementations, the data structures are studied using performance metrics like unhalted CPU cycles and level 1 cache misses, which provide more information than just throughput, to establish certain correlation and trade-offs among different correctness criteria. These metrics help in analysis of the different data structure implementations by plotting them for each data structure implemented using different correctness criteria. Using these metrics, we will be looking for patterns that will help us understand the correlation between the different correctness criteria and therefore establish the trade-off for using these correctness criteria. These trade-offs can be an increased throughput at the cost of higher memory utilization, or low throughput resulting in fewer cache misses giving an insight on limitations, restrictions, issues and implications.

Such trade-offs can also help us understand the complexity of each correctness criteria while designing concurrent data structures. This aspect of complexity is not subjective, but can be inferred from the steps taken to ensure that the data structure or algorithm is correct as per the correctness criteria being used. Such an analysis presents the complexity of designing data structures using relaxed correctness conditions. An execution of a concurrent data structure generates a concurrent history. This concurrent history is rearranged to form a sequential history, according to the correct-

ness condition. This rearrangement can be performed in a number of different ways, following the formal definition of a correctness criteria. All of the rearrangements have to conform to a correct sequential history, for the implementation to be correct. In this thesis, it is shown that as a correctness condition is more relaxed, the number of possible rearrangements of the concurrent history is increased. A developer of concurrent data structures and algorithms has to ensure that such a data structure built using a relaxed correctness criteria allows all the possible rearrangements for higher performance, and also verify the correctness of the rearranged sequential histories. This makes design of concurrent data structures using relaxed correctness criteria a complex task. An analysis of the complexity of design has not been performed in previous studies of individual correctness criteria, but it is an important aspect that relates the relaxation of correctness criteria to the complexity incurred while designing a concurrent program.

The approach used is to first check the correctness of an existing implementation of one of the above data structures using the *Correctness Condition Specification Tool* (CCSpec). Then the implementation of the data structure is modified to meet a different correctness criteria. Thereafter, the different implementations are compared using hardware metrics from *Performance Application Programming Interface* (PAPI) [3]. The hardware metric used are the unhalted CPU Cycles and the level 1 data cache misses. Upon measuring the hardware metrics, the results are analyzed to infer correlation between the different correctness criteria. Since these metrics are hardware specific, all the experiments are done on a single machine so that there are no differences in the metric values while inferring the results. Then the correctness criteria are categorized according to throughput, memory utilization and feasibility.

We make the following contributions by presenting:

1. The first thesis on correctness criteria, that are applicable for concurrent non blocking data structures, which analyzes the performance trade-offs among each of the correctness condi-

tions.

2. Coining the term *Complexity of Design* and finding its relation with designing a data structure to meet a particular correctness criteria.

3. Relationship between correctness criteria in comparison to the data structure type.

The contributions provide an overview to designers of concurrent programs to analyze the trade offs for designing a data structure using different correctness criteria.

The thesis is divided into discussing the background of each correctness criteria, including their formal definitions, explanation and examples in the next chapter. The third chapter presents the literature review for the thesis. The methodology used in the thesis is discussed in chapter four. Chapter five analyses the findings from this thesis, while the conclusions are presented in the last chapter.

CHAPTER 2: BACKGROUND

The Standard Template Library provides many data structures that are well documented based upon their usage and the functionality that they provide. However, they are designed to operate using a single thread, leading to sequential operations. But with processors incorporating multiple cores and running multiple threads, these sequential implementations cannot operate with multiple threads. A sequential implementation doesn't protect shared memory while multiple threads are accessing it, leading to problems like data races, torn reads and writes, and many other issues. For this reason, methods have to be defined that provide safe access to shared memory when multiple threads are operating. The two criteria used to define concurrent programs are correctness and progress assurance. In this thesis, the main focus is on correctness of a concurrent program. We now proceed to understand the basis of using a correctness criteria.

Correctness Criteria

Since concurrent programs are different than sequential programs, a different paradigm is required to infer their correctness. Sequential programs or data structures are inferred to be correct if they show the required behavior. For an example a FIFO queue requires preserving the First In First Out property for each element in the queue. In a sequential implementation of FIFO queue, since only one thread is operating, the operations of enqueue and dequeue will occur one at a time. Thus the correct implementation of a sequential FIFO queue depends only upon the correct implementation of the enqueue and dequeue operations, since there is no inter thread interaction. But with multiple threads interacting, it is not sufficient to provide correct implementation of enqueue and dequeue which only prevent data races. Along with this, there has to be a theoretical analysis of whether the interacting enqueue and dequeue operations can be arranged to satisfy a sequential

execution. This is the basis of correctness criteria that is used to analyze the safety of concurrent implementations. As an example, the Herlihy/Wing queue [2] is presented in Algorithm 1. This is a simple implementation of an array based concurrent FIFO queue. The enqueue operation reserves an index in the array for enqueue by using an atomic Get And Increment operation on the atomic variable called *tail*, and stores the current value of *tail* in a thread local variable *position*. It then places the value to be entered in the queue at the location of the array *items* indicated by *position*. The dequeue operation starts by atomically loading the value of *tail* into the variable *limit*. This specifies the current maximum possible length of the array based queue. Then starting from the first location of the array to *limit*, each position of the array is exchanged with NULL value using an atomic exchange operation. If the value exchanged from the array is not NULL, then it is returned by the dequeue operation, otherwise it continues the exchange of the array elements. However if the dequeue operation is unable to find any non NULL values in the array, it simply returns NULL, indicating that the queue is empty.

An example using different threads to perform different operations is given below. Let us assume that two threads perform operations on the Herlihy/Wing queue. Thread A first enqueues a value of 100, followed by a dequeue operation performed on the queue. Thread B performs an enqueue of value 200. The above scenario can be represented by a time based diagram of the operations that are performed by each thread in real time order. Such a diagram is presented in Fig. 2.1. In this figure, the blue line at the bottom represents time as it increases from left to right. The orange lines represented at the top are the execution of the enqueue and dequeue operations performed by thread A, and the green line represents the enqueue operation performed by thread B. Each operation begins and ends in a period of time represented by the length and position of the line for the given thread and operation. Within each line is written the name of the thread that called the operation, the name of the operation that was called and the arguments that are passed to the operation. For example, the first operation performed by thread A is an enqueue of value 100, and

is represented as $A.enq(100)$. This notation identifies each operation performed by each thread, enabling the extension of the diagram to represent any number of threads performing different operations.

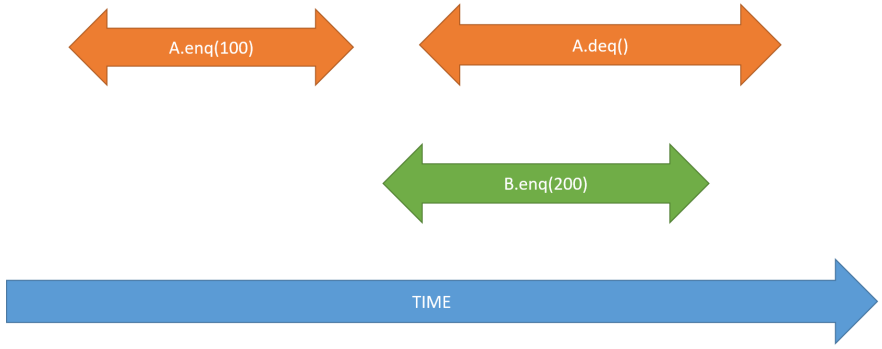

Figure 2.1: A time based diagram for the Hirlihy/Wing Queue

The safety of a concurrent data structure according to a correctness condition can be analyzed from such a diagram. The correctness condition followed by Hirlihy/Wing Queue is Linearizability. Thus according to Fig 2.1, the dequeue operation must return a value of 100, since linearizability is based upon the operations taking place in a real time ordering. If the dequeue operation returns a value of 200, then the Hirlihy/Wing Queue would not have satisfied linearizability. However it could have still been correct according to sequential consistency or quasi linearizability. The conditions for validating correctness of a concurrent execution is explained using formal definitions of each correctness condition. We now proceed to define these terms in the next section.

Definitions

In this subsection we define certain terms that will be helpful in understanding the formal definition of the different correctness criteria.

Definition 1: History

A history according to [1] is a finite sequence of method *invocations* and *responses*. An invocation and response are formally defined in [1] as tuples in the form of $< x.m(a^*)A >$ for invocation and $< x : t(r^*), A >$ for response respectively. In this definition, x is the object upon which the operation is taking place, A is the thread performing the operation, m is the name of the operation being performed, a^* are the arguments of the operation, t is either *Ok* or an exception, and r^* are the sequence of result values. A response matches an invocation if the object and thread are the same and there are no other response of the same object and thread between the given invocation and response pair. This is defined as a *method call*. Formally a *method call* is a pair of matching invocation and response in a history H. An invocation is said to be pending, if no matching response can be found in the history H. A history H is complete if for each invocation, there is a matching response.

Definition 2: Matching Pair

A matching pair is a set of invocation and response that belong to the same operation performed by a single thread. It matches the operation that has been completed by a thread. According to [5] let a history be denoted as H. Let $\#H$ be the number of sequences in H. The invocation is denoted as $inv?(e)$ and the response as $ret?(e)$. The set of return events, as denoted earlier, is r^*. $H(n)$ represents the n^{th} sequence in the history H. Also let $e.p \in P$ be the process executing the event e and $e.i \in I$ the index of the abstract operation to which the event belongs. Now [5] give a formal definition of matching pair as follows:

For a sequence of events *seq*, two positions in H, m and n form a matching pair called as

$mp(m, n, H)$, if

$$0 < m < n \leq \#H \wedge H(m).p = H(n).p \wedge H(m).i = H(n).i \wedge \\ \forall k \bullet m < k < n \Rightarrow H(k).p \neq H(m).p \tag{2.1}$$

Definition 3: Pending Invocation

As mentioned in Definition 1, a pending invocation is an invocation in a history H such that there is no matching response in the same history. A formal definition according to [5] is a location n in H known as a pending invocation, denoted as $pi(n, H)$ if

$$1 \leq n \leq \#H \wedge inv?(H(n)) \wedge \forall m \bullet n < m \leq \#H \Rightarrow H(m).p \neq H(n).p \tag{2.2}$$

Definition 4: Legal History

A legal history is one where each invocation and response belong to a matching pair, and/or there are pending invocations in the history. A history is said to be legal [5] if

$$\forall n : 1..\#H \bullet \textbf{if} inv?(H(n)) \textbf{ then } pi(n, H) \vee \exists m : 1..\#H \bullet mp(n, m, H) \\ \textbf{else} \exists m : 1..\#H \bullet mp(m, n, H) \tag{2.3}$$

Definition 5: Sequential History

A history H is said to be sequential if for all invocation, except for the last one, the next sequence

is the matching response of the previous invocation. Sequential histories are used to validate the correctness of concurrent algorithms by matching the concurrent histories produced by a concurrent algorithm with a sequential history. This is explained for each correctness criteria mentioned below.

Quiescent Consistency

Formal Definition:

In [5], before the definition of quiescent consistency, a definition of a quiescent history is presented as follows:

A legal history H is quiescent, written as *qu(H)* if $\nexists n \bullet pi(n, H)$

According to [5], if we consider a quiescent concurrent history H, and a sequential history HS, then H can be considered quiescently consistent, denoted as *qcons(H, HS)* if

$$\begin{aligned} &\exists bijective f : 1..\#H \mapsto 1..\#HS \bullet \\ &(\forall n : 1..\#H : H(n) = HS(f(n))) \wedge (\forall m, n : mp(m, n, H) \Rightarrow f(m) + 1 = f(n)) \\ &\wedge \forall m, n, k : m < n \wedge m \leq k \leq n \wedge qu(H[1..k]) \wedge ret?(H(m)) \wedge inv?(H(n)) \\ &\Rightarrow f(m) < f(n) \end{aligned} \tag{2.4}$$

An implementation I is quiescent consistent wrt. a specification S if for all quiescent histories H of I there is a sequential history HS of S such that qcons(H; HS).

Explanation

The formal definition is an equation that defines the behavior of a concurrent algorithm whose correctness can be proved using quiescent consistency. Its meaning can be understood with an example. In Fig. 2.1, we see that the enqueue by thread A is completely separated in real time from the the enqueue by thread B. This implies that there is a time frame in which no work is being done. Such a time frame is called a period of quiescence. Quiescent consistency develops on this idea and puts forth the notion that if two operations are separated by a period of quiescence, then the operations before and after the period of quiescence are sequentially ordered. Such a notion is difficult to understand initially, but if we look carefully at Fig 2.1, it is clear that there is a period of quiescence after thread A enqueues 100, and then thread B enqueues 200 and thread A dequeues an element. Now the latter two operations can have any order, but their total order with respect to the first operation has to be sequential in nature due to the period of quiescence.

Examples:

There are some data structures that follow quiescent consistency. For example, the priority queue in [7], a LIFO stack in [14], a queue implementation in [5] and other implementations of similar data structures as well. Some of these data structures will be discussed in the next section.

Sequential Consistency

Formal Definition:

In [9] there is a formal definition of sequential consistency. But first we present some of the terms used in [9].

Given an execution σ, let $ops(\sigma)$ be the sequence of call and response events appearing in σ in real time order, breaking ties for each real time t as follows: First, order all response events for time t whose matching call events occur before time t, using process IDs to break any remaining ties. Then order all operations whose call and response both occur at time t. Preserve the relative ordering of operations for each process, and break any remaining ties with process IDs. Finally, order all call events for time t whose matching response events occur after time t, using process IDs to break any remaining ties. Given a sequence of s of operations and a process p, we denote by $s \mid p$ the restriction of s to operations of p.

Definition: An execution σ is *sequentially consistent* if there exists a legal sequence H of operations such that H is a permutation of $ops(\sigma)$ and, for each process p, $ops(p) \mid p$ is equal to $H \mid p$.

Explanation:

Perhaps one of the most widely used correctness criteria is sequential consistency. In order to understand sequential consistency, we need to take a look at Fig. 2.1 again. Now let us assume a scenario in which the dequeue of thread A returns 200. According to quescient consistency and the sequential model presented before, this is not possible as the period of quiescence between the two enqueues will definitely prevent 200 from being dequeued first and this will be an incorrect implementation. However sequential consistency has nothing to do with the global real time order. It is only concerned with the partial time ordering of each thread. In other words, there is only an ordering of the intra thread operations, while the inter thread timings can be changed as long as the intra thread timings are not violated. Thus for the example in Fig. 2.1, the enqueue by thread B can be moved backwards in time as there is no other operation being performed by thread B. Now we can say that the enqueue by thread B of the element 200 happens before the enqueue by thread A of 100. Now this justifies the dequeue of 200 by thread A to be sequential in nature. In other

words, we can say that the enqueue of 200 by thread B occurs first, followed by the enqueue of 100 by thread A, and finally a dequeue by thread A of 200. A notion that one may have is to think that the sequential order can be an enqueue by thread B of 200, followed by the dequeue of thread A that returns 200, followed by enqueue of 100 by thread A. However this is not possible as each thread has to respect its own internal ordering of operations. This examaple is shown in Fig. 2.2.

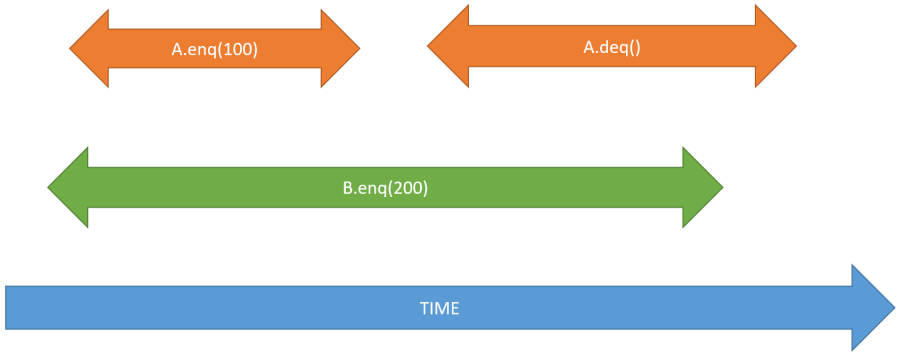

Figure 2.2: Example to show sequential consistency

Examples:

Sequential consistency has been used to determine the correctness of many concurrent objects such as the queue implementation in [2] as well as in [9], a stack implementation in [9]. Other data structures that follow linearizability would also be correct according to sequential consistency. In addition to the data structures that follow linearizability as well as sequential consistency, the next section also presents a novel queue implementation that is correct as per sequential consistency, but not according to linearizability. This queue implementation is compared against the other linearizable queue in [2].

Linearizability

Formal Definition:

In [2] the formal definition is preceded by definition of partial order given as follows:

A history H includes an irreflexive partial order $<_H$ on operation:

$$e_0 <_H e_1 \textbf{ if } ret?(e_0) \textit{ precedes } inv?(e_1) in H. \tag{2.5}$$

A history H is linearizable if it can be extended to some history H' such that:

$$complete(H') \textit{ is equivalent to some legal sequential history S } and \\ <_H \subseteq <_S \tag{2.6}$$

Another definition presented in [9] is as follows:

An execution σ is *linearizable* if there exists a legal sequence H of operations such that H is a permutation of $ops(\sigma)$, for each process p, $ops(p) \mid p$ is equal to $H \mid p$, and furthermore, whenever the response of operation op_1 precedes the call of operation op_2 in $ops(\sigma)$, then op_1 precedes op_2 in H.

In this definition, $ops(\sigma)$ is the sequential history S, and the rest of the definition is the same as in [2].

Explanation:

The notation $<_H$ is based on the real time order of events within a history H. It can also be termed as a *happens before* relationship. An event e_1 is said to happen before an event e_2, if the entire execution of e_1 is complete before the execution of e_2 begins. In a sequential history, since each invocation is followed by its response, all the events are implicitly ordered. This implicit ordering is not possible in a concurrent history, since there can be multiple invocations by different threads that occur before there is a response that completes an event. Therefore two events can only be ordered when the invocation of one follows the response of the other event, and they are said to follow a *happens before* relationship.

Establishing such happens before relationship among all events in a concurrent history is possible through the specification of *linearization points* for each operation with respect to other operations that occur concurrently with them. A *linearization point* is an instruction in an operation, where the entire operation completes with respect to other concurrent operations that are taking place. In other words, it helps order the concurrent operations to obtain a happens before relationship among all the operations. For a better understanding, let us look at the example in Fig. 2.3. In this example, there are many different outcomes that can be possible. One of the possible outcomes is that thread A enqueues 100, followed by the enqueue of 200 by thread B, and after that thread A dequeues 100 and thread B dequeues 200. The linearization points are shown in Fig. 2.4. From the linearization points, we can conclude that the concurrent history is equivalent to a sequential history. Another possible sequential history can start with thread B enqueuing 200 first followed by enqueue of 100 by thread A. If thread A dequeues first, it will return 200 followed by the dequeue of thread B returning 100. Thus we can see that there are different possible interleavings of the linearization points. If we look at the Algorithm 1, we can find the instructions that act as the linearization points. If two enqueues overlap, the one that performs the fetch and add on the tail first, can be said to linearize first, since it acquires a spot in the queue before the other thread's enqueue. Therefore the element is placed in the first position in the queue even if the actual insertion of

the value occurs later. In such examples, it is often easy to find one instruction that can act as the linearization point. The linearization point can change based upon the operations that overlap. In algorithm 1, the linearization point for two enqueues is at the moment the *AtomicFetchAdd()* operation completes. However if an enqueue overlaps with a dequeue, then the linearization point of the enqueue is after the completion of *AtomicStore()* operation. Another interesting observation is that a linearization point can be defined only when there are overlapping operations. If the enqueue of thread A happens in isolation as given in Fig. 2.1, there is no need to define the linearization point for the enqueue.

An advantage of linearizability over sequential consistency is the composability of linearizable objects. If two concurrent objects' correctness is individually defined using linearizability, then a third object built using only the previous two objects, is also correct in accordance to linearizability. This property is called *composability*. Concurrent objects that follow sequential consistency are not composable, while objects that follow linearizability are. This is an advantage of linearizability over sequential consistency, and the reason why linearizability is used in most of the concurrent objects. This helps to simplify software life cycle., where each individual component can be verified to follow linearizability, which in turn guarantees that the complete software is linearizable.

Examples:

Linearizability has been used extensively to check correctness of various data structures. Some examples include a queue implementation in [2] and [9], a stack implementation in [9], a priority queue implementation in [19], a wait-free hashmap in [18] and many other data structures. In addition to these data structures, we also implement a linearizable version of the priority queue in [7], to compare its performance with the quiescently consistent priority queue. This comparison is presented in the next section.

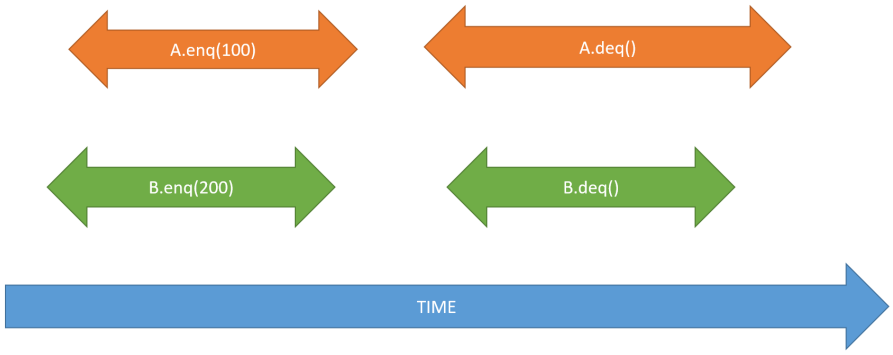

Figure 2.3: Another example of linearizabilty

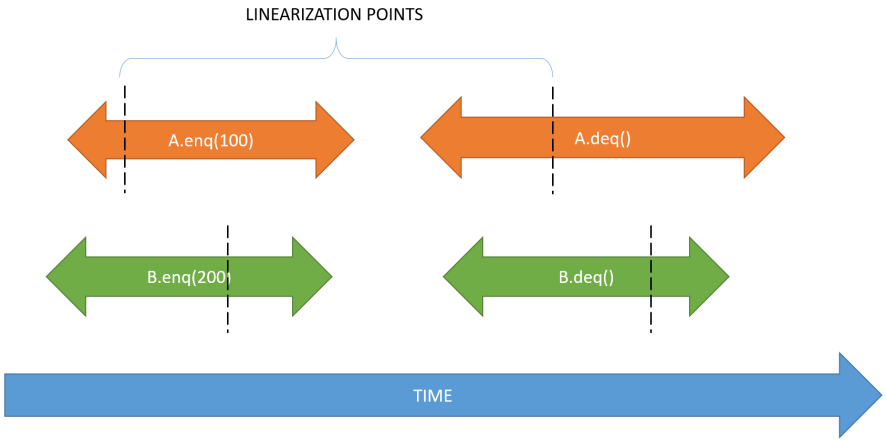

Figure 2.4: Linearization points for the example in Fig. 2.3. The dashed lines show the linearization points for each of the operation on the queue by the two threads A and B.

Quasi Linearizability

Formal Definition:

In [6] there are definitions that precede the definition of Quasi Linearizability. First we define

Quasi-linearization factor and Q_O-Quasi-Sequential specification.

Definition 6: Quasi-linearization factor

A function Q_O of an object O defined as $Q_O : D \rightarrow N$. D is the set containing subsets of the object's domain, formally $D = \{d_1, d_2, ...\} \subset Powerset(Domain(O))$.

Definition 7: Q_O-Quasi-sequential specification

The Quasi-sequential specification is a set of all sequential histories that satisfy the *distance* bound implied by the quasi-linearization factor Q_O of an object O. The *distance* between two histories H and H', denoted as $Distance(H', H)$ is defined such that H' is a permutation of H, and is $max_{e \in Events(H)}\{|\ H'[e] - H[e]\ |\}$..., and distance can only be defined for histories that are permutations of each other. Formally for each sequential history H in the set, there is a legal sequential history S of O such that H is a prefix of some history H' which is a permutation of S and $\forall$ subset $d_i \in D :\ Distance(H' \mid d_i, S \mid d_i) < Q_O(d_i)$

Definition 8: Q-Quasi-Linearizable history

Let *Objects(H)* be the set of all objects that H involves with. A history H is *Q-Quasi-Linearizable* if it has an extension H' and there is a sequential history S' such that:

1. $Q = \bigcup_{O \in Objects(H)} Q_O$
2. *Complete(H') is equivalent to S'*
3. *If method invocation $inv?(e_0)$ precedes method invocation $inv?(e_1)$ in H, then the same is true in S'.*

4. $\forall O \in Objects(H) : S' \mid O$ *is member of the* Q_O*-Quasi-Sequential specification.*

Definition 9: Q-Quasi-Linearizable object

An object implementation A is Quasi-Linearizable with Q if for every history H of A (not necessarily sequential), H is Q-Quasi-Linearizable history of that object.

Explanation

Therefore linearizability is one of the most common correctness criteria that is used by developers to infer the safety of their concurrent objects. Since it is compositional, it is easy to combine two or more objects that are linearizable. However, its strict nature prevents optimal utilization of the concurrent resources. Thus there is inherent need to look for correctness criteria that is more relaxed than linearizability, but at the same time is compositional. This introduces Quasi Linearizability, which is modified from linearizability to present a much more relaxed version of the correctness criteria. The formal definition presented above represents the criteria for a Quasi-Linearizable object. To understand this, let us revert back to Algorithm 1. As we have seen before, this algorithm linearizes concurrent enqueues based on the fetch and add operation on the tail. This implies that only one operation can linearize at a time. Quasi Linearizability on the other hand says that up to Q number of such operations can linearize in any order and all of them would be acceptable. For example, if $Q = 2$, then from the example in Fig. 2.3, we can infer that the enqueues and the dequeues can happen in any order, but since $Q = 2$, we can have only two operations fall into this criteria; that is, the enqueues will linearize first followed by the dequeues. However if we increase Q to 4, now all the four operations can happen in any order. This set of outcomes is different from all the other outcomes presented till this point of the thesis. One of the possible outcomes can be that both the dequeues linearize first, returning an exception that the queue is empty, followed by the enqueues which enqueue 100 and 200 respectively. Such a

result has not been presented by the former correctness criteria that we have studied. Moreover, this is a valid sequential history where two dequeues are performed on an empty queue, followed by two enqueues which lead to the queue containing two elements after all the four operations are completed.

Examples:

Quasi linearizability has several examples such as a k-FIFO queue implementation in [5] and a k-LIFO stack implementation in [1].

Eventual Consistency

Eventual consistency [10] is a weak correctness criteria used in databases, and therefore a formal definition of the same is out of the scope of the thesis. It is a form of consistency model more than just a correctness condition, and is used in large scale database designs in Amazon [10]. It was developed to meet the growing demands of increasing the size of the available databases, while still maintaining retrieval and update time to provide client satisfaction. In [12], the *CAP Theorem* is presented, which states that two out of the three properties of shared data systems, namely data consistency, system availability, and tolerance to network partition, can be achieved at any given time. Considering this aspect, if a system is made tolerant to network partitions, then one of the other two properties has to be sacrificed. If such a system additionally incorporates data consistency, that is, the data within the system is always consistent to the end user, then the system's availability decreases and the client may not be able to use the system when the load exceeds a certain limit. On the other hand, relaxing the consistency criteria for the data, increases availability of the system for the users. This led to the development of eventual consistency, which provides a relaxed consistency model to provide higher availability of the system to the client. However,

such a weak consistency model also implies that the data might not be the same when accessed by different clients. Eventual consistency only guarantees that if no updates are made to the system, eventually all accesses will return the last updated value [10]. Eventual consistency comprises of many other consistency models such as *Causal Consistency, Read-your-writes consistency, Session consistency, etc.* as given in [10]. One of the more prominent of these is causal consistency, whose applications are being tested in shared memory systems as well [13]. Causal consistency is based on causation and its effects on different entities based on the relation to the source of the cause. It is the basis of relativistic programming, which is based on the principles of relativity. If a database is using causal consistency, then a write initiated by a process A will be visible to process B, if process A has communicated to process B about the update. However another process C, which has no correlation to process A, will not reflect the updated value until it is available according to the existing consistency model [10]. But further discussion of the above models are beyond the scope of the thesis which focuses on the correctness conditions most widely used in shared memory data structures.

CHAPTER 3: LITERATURE REVIEW

Since concurrent objects need to follow rules for safety and liveliness, the correctness criteria and progress guarantee have been studied in detail throughout the literature for concurrent programming. Since the focus of this thesis is on safety or correctness criteria, we present a detailed review of the literatures used in this thesis by dividing them into different sections of their contributions.

Overview of Correctness Criteria

Concurrency has been studied in detail in [1] which deals with most of the basic concepts and gives the basis of study on correctness criteria. It provides detailed explanation on Linearizability, Sequential Consistency and Quiescent Consistency, with examples and definitions. The time based diagrams presented in this thesis are based upon the examples provided in [1]. These are used to analyze the modifications made to data structures to incorporate different correctness criteria.

A detailed explanation of each correctness criteria is provided in separate literatures. Linearizability is presented in detail in [2]. It provides the formal definition presented in this thesis and gives in dept analysis and examples to help understand linearizability better. The Herlihy/Wing Queue, used as one of the data structures for the case study, is a slight modification of the queue implementation presented in [2]. This queue implementation is then further modified to be sequentially consistent and not linearizable. The proof of linearizabe objects being composable is also presented in [2].

The basic definitions and examples for Sequential Consistency are provided in [1]. The examples are extensive and distinguish clearly between linearizability and sequential consistency. It also explains why sequentially consistent objects are not composable and the problems incurred in

designing these objects. A more detailed and mathematical explanation of sequential consistency and its formal definition is provided in [9].

Quiescent consistency is studied in [5] using a detailed formal definition and explanation of the terms used in the paper. Many of the definitions for quiescent consistency have also been presented in this thesis as a background to understand quiescent consistency. The composability of quiescently consistent concurrent objects is also presented in [5]. Additional examples and explanation for quiescently consistent objects is provided in [1]. It explains in a simple way the formulation of concurrent design based upon quiescent consistency, but doesn't delve into the mathematical analysis of the formal definition.

Quasi linearizability is presented and compared with linearizability in [6]. There is a detailed explanation of the concept of quasi linearizability and a formal definition of the same. The definition in [6] is also used to in this thesis for understanding quasi linearizability from a theoretical standpoint.

Eventual consistency is explained in detail in [10]. It is not analyzed in the case study performed in this thesis, but is still covered in the background to understand the basics and why it is not included in the case study.

Comparison of Correctness Criteria

Linearizability and sequential consistency are compared in [9] , but the approach is more theoretical and there are no practical examples for the same. Also the comparison is only on the basis of throughput, and the calculations are based upon the mathematical constructs derived from the formal definitions of linearizability and sequential consistency.

On the other hand quasi linearizability is compared against linearizability in [6] using a practical approach. However only throughput is compared in this approach using software metrics to show that data structures that follow linearizability have lower throughput than the ones that follow quasi linearizability.

Literature of Concurrent Data Structures

The case study used in this thesis uses different data structures to study their throughput, memory utilization and usability. The linearizable array based FIFO queue implementation is a slight modification of the implementation presented in [2]. The modification converts the algorithm into a non blocking implementation of a linearizable array based FIFO queue. The K-FIFO queue is presented in [20] and the same algorithm is compared with a linearizable and sequentially consistent FIFO queue in this thesis. The implementation of a quiescently consistent priority queue is adopted from [7] and is further modified to construct a linearizable priority queue for comparison in the case study. The implementation of a linked list based queue and stack is from the tervel library [15]. These are also modified to follow quasi linearizability and then compared in the case study.

Verifying Correctness Conditions

The verification of different correctness criteria is a challenging task that requires knowledge about both the data structure being analyzed and the correctness condition that we are verifying. Verification can be done either analytically or by using certain tools. In [16] many different methods for verifying linearizability are analyzed. Some of them are *Canonical Abstraction*, *Sequential Abstraction*, *Reduction*, *Shape Analysis*, *Augmented States* and a few other methods. The next sec-

tion discusses Correctness Condition Specification tool (CCSpec), which has been used to verify the correctness conditions of the different data structures presented in this thesis. Along with the verification methods provided in [16], it is also mentioned that model-based verification tools also exists, but is not covered in the thesis.

In this thesis, we use a model-based tool that can verify all the correctness conditions mentioned in Appendix B known as CCSpec.

CHAPTER 4: METHODOLOGY

This thesis studies the application of different correctness criteria for concurrent non blocking data structures. The formal definition of each correctness criteria, and their explanation provide theoretical background to study the correctness conditions in detail. A detailed study of the correctness condition is required to provide insights on performance, usability and trade-offs. These parameters are studied using a case study of the different correctness criteria using different data structures. Since the requirement is to compare the performance of different correctness criteria, we implement the same data structure using different correctness criteria, and gather the hardware metrics for the different implementations. This provides data to compare performance of the same data structure when implemented using different correctness criteria.

Overall Method

The case study is designed to analyze the throughput, memory utilization and usability of different correctness criteria, while implementing different data structures. This is achieved using the following method. First, we check the correctness condition of an existing data structures in literature, like a queue, stack or priority queue, using CCSpec. In order to obtain a fair comparison between the various correctness criteria, we perform minimal modifications to the existing data structure design to achieve a new target correctness criteria. CCSpec is used to verify whether the modified data structure meets the new correctness criteria as well as fails the previous one. CCSpec utilizes unit tests to verify the correctness criteria. The unit tests have been designed to enumerate through different interleavings of execution. These interleavings create differences between the correctness criteria, and are covered within the test conditions. This ensures that the modified data structure is indeed correct according to the correctness criteria employed to design the data structure. Further,

there is confirmation that the modification is sufficient to violate the previous correctness criteria. We follow the previously described methodology to modify the data structure to target each of the correctness criteria covered in this thesis. In certain cases this is not possible, since either the modification leads to unfair comparison, or there doesn't exist any method in literature to facilitate the change in correctness condition. Thus, if the change is not possible, then the comparison is performed using only the original and modified data structure. Once a data structure has been implemented using different correctness criteria, they are tested to measure the performance metrics. These performance metrics provide us with the data to compare the different correctness criteria, and infer the correlations in terms of throughput, memory utilization and usability. PAPI [3] is used to measure the two different hardware metrics used in the case study. These are the unhalted CPU cycles, and the level 1 data cache misses.

Detailed Method

In order to measure the hardware metrics, the different implementations of the data structure are used to perform a certain number of operations that are consistent for the various implementations. This method is derived from [17], where different queue implementations are compared. The comparison of different queue implementations is based upon performing one million enqueue and dequeue operations, starting with an empty queue. However in [17], the metric is time elapsed, which is a software metric. In the case study that we perform, we use hardware metrics instead of software metrics. Hardware metrics provide more accurate and detailed information than software metrics. These metrics are *Unhalted CPU Cycles* and *Level 1 Data Cache Misses*.

The hardware metrics are chosen to provide two different aspects for analysis of the correctness criteria. The CPU cycles measure the performance of the data structure since less CPU cycles for a set of operations indicates that the particular implementation has a higher throughput compared

to an implementation that takes more CPU cycles to complete the same set of operations. As an example, if an implementation of a concurrent queue using a particular correctness criteria takes 10^8 CPU cycles to complete 10^4 enqueue and 10^4 dequeue operations, while another takes 10^6 CPU cycles, it implies that the former implementation takes more time to finish the same set of operations than the latter implementation, indicating a higher throughput of the latter implementation than the former one. Thus CPU cycles as a hardware metric indicates the throughput of an implementation. The second metric is the number of level 1 data cache misses. This hardware metric represents the cache utilization for any given implementation. Since the level 1 data cache is the closest to the processor, if there is a hit in this cache, then the execution can take place within a few cycles. However, a miss indicates that the data has to be fetched from lower level cache or even the main memory. This can take up to hundreds of CPU cycles. It also creates a bottleneck on the memory bus if there are frequent request to lower level caches. Therefore in applications where memory utilization needs to be optimized, this hardware metric becomes most important. For this reason, this additional metric, along with CPU cycles, is also considered in this thesis to provide greater insights into each correctness criteria. This also helps developers look into different aspects of implementations using different correctness criteria.

An example of the case study is now provided. If we are to test the different implementations of a queue, then all the implementations could perform 10^4 enqueue operations and 10^4 dequeue operations. A lower number of operations are selected since hardware metrics can provide accurate information even while executing limited number of operations. Since the number of enqueue and dequeue operations are the same, there is a 50% ratio of enqueue to dequeue operations. Once the results are collected using 50% ratio, the ratio is changed to 75% enqueue operations and 25% dequeue operations to get another set of metrics. Upon further collection of data, the ratio is changed to 25% enqueue operations and 75% dequeue operations. In the above mentioned manner, 3 sets of metrics are collected for a given data structure implementation with different correctness

criteria.

In a similar way, the priority queue and stack implementations are also studied. For a priority queue, the operations are insert and deletemin. The insert adds an element, which includes a value and a priority, to the priority queue and the deletemin operation removes the element with the least priority from the priority queue. The stack has two operations, called pus and pop. The push operation adds an element to the top of the stack, while the pop removes the top most element from the stack.

Measuring Complexity of Design

In order to obtain a measurable parameter to assess the complexity of designing a data structure, we first define complexity of design. The complexity of design is defined as *the number of valid sequential histories obtained by rearranging the concurrent history of a given data structure for a fixed number of operations, according to the formal definition of the correctness condition*.

The definition is validated based upon an analysis of the data structures presented in Section 5 of this thesis, and also based upon the formal definition of each correctness criteria. A sequential history is constructed by executing a sequential implementation of the data structure under study. A sequential history is correct if it follows the logical constructs of the data structure. As an example, in a sequential history of a FIFO queue, an enqueue of value 100, followed by a dequeue should return the value 100 to be logically correct. For a concurrent program, the concurrent history is rearranged into a finite number of sequential histories, based upon the formal definition of the correctness criteria being used. Then each sequential history is checked according to the logical correctness of the data structure, and if all are correct, the implementation is considered correct. The higher the number of sequential histories obtained indicates that the designer has

more flexibility with the design such that it will meet the correctness condition. At first it may seem as though a relaxed correctness condition may result in a more simplistic design than that of a strict correctness condition such as linearizability. However, in order to truly take advantage of the larger number of allowable method call ordering, a designer should employ additional techniques to increase thread interleavings that would otherwise violate a strict correctness condition. This leads to higher complexity of design.

Now we present a mathematical proof of a relaxed correctness condition resulting in greater number of sequential histories. Let us assume that a concurrent data structure is implemented to perform N number of operations. These operations are divided among n threads, where each thread performs on average k number of operations. Thus we have $n * k = N$.

We first present a hypothetical correctness criteria, where the operations performed in a concurrent history can be rearranged in any order to form a sequential history. Thus we can use permutations to find the number of possible sequential histories denoted by H_s as follows:

$$H_s(H) = {}^{N}P_N = N! = (n * k)! \tag{4.1}$$

For sequential consistency, the equation changes, since each thread has to maintain its own order of operations. An enqueue of value 100 that precedes an enqueue of value 200 performed by the same thread, needs to maintain the same order in the sequential history. Therefore the equation for the number of sequential histories can be written as:

$$H_s(SC) = \sum_{j=1}^{n} M(k, kj) \tag{4.2}$$

Where

$$M(k, kj) = \sum_{i=1}^{kj-k+1} M(k-1, kj-i) \tag{4.3}$$

Substituting the value of $M(k, kj)$ in equation (8), we get

$$H_s(SC) = \sum_{j=1}^{n} \sum_{i=1}^{kj-k+1} M(k-1, kj-i) \tag{4.4}$$

This is an iterative equation, where $M(k, kj)$ depends on the previous values of k and j. The minimum value for M is obtained as $M(2, 2) = 1$. Using the value of $M(2, 2)$, the iterative equation can be modeled into a program to find the value of $H_s(SC)$ for any given n and k. A closed loop equation is not possible for the correctness criteria defined in this thesis, due to the dependency of each operation on other operations.

The number of terms in this equation are the number of total operations performed, or $n * k$.

For linearizability, equation (10) still holds, but the value of H_s can change as the formal definition of linearizability enforces real time ordering along with thread level ordering. Due to this reason, the value of H_s for linearizability cannot exceed the value of H_s obtained from equation (10), and in most scenarios, has a lower value. Thus we can write the equation of Linearizability as follows:

$$H_s(L) \leq \sum_{j=1}^{n} \sum_{i=1}^{kj-k+1} M(k-1, kj-i) \tag{4.5}$$

For Quasi Linearizability, the independent factor of Q_0, that is the quasi linearization factor, can be multiplied to equation (10), since each iterative change in equation (9) due to Q_0 is independent

of the original form of the equation. Thus the equation for the number of valid sequential histories for quasi linearizability are represented as:

$$H_s(QL) = \sum_{j=1}^{n} \sum_{i=1}^{kj-k+1} M(k-1, kj-i) * Q_0 \tag{4.6}$$

Equation (7), (10) and (12) cannot be directly compared. The approach for comparison is to take incremental numbers of n and k and show that the values for $H_s(H)$ and $H_s(QL)$ always exceed the values for $H_s(SC)$. The values for linearizability are always equal to or lower than the value of $H_s(SC)$. The values for $H_s(QL)$ are obtained using $Q_0 = 2$. A small value for Q_0 is used since the values for k and n are also small, and if the value of Q_0 is equal to the value of k, then $H_s(H)$ and $H_s(QL)$ become equal. The values are compared in table 4.1.

Table 4.1: Table comparing number of valid sequential histories that are possible for a Hypothetical correctness condition $H_s(H)$, sequential consistency $H_s(SC)$ and quasi-linearizability $H_s(QL)$ for incremental values of n and k

n	k	$H_s(H)$	$H_s(SC)$	$H_s(QL)$ $\{Q_0 = 2\}$
2	2	24	12	24
2	3	720	144	480
3	3	362880	15120	120960
3	4	479001600	198450	58060800

From table 4.1, we observe the difference between the values of $H_s(H)$, $H_s(QL)$ and $H_s(SC)$. For incremental values of n and k, $H_s(SC)$ is always lesser than $H_s(QL)$, which is lesser than $H_s(H)$, if Q_0 is less than k. As $H_s(L)$ is always lesser than or equal to $H_s(SC)$, the sequential histories obtained would be even lesser for $H_s(L)$.

Complexity of design, as defined above, correlates to practical implementation of concurrent data structure. As complexity of design increases, it becomes increasingly difficult to implement a

concurrent data structure. Each of the valid sequential histories need to be checked for logical correctness during design, leading to increased time for designing the data structure. Often the complexity can render a design infeasible. This observation is the effect of relaxation of correctness criteria on the complexity of the above mentioned correctness criteria in design of concurrent data structures. We now present the method used to determine complexity of design in this thesis.

A FIFO queue is implemented in the case study using three different correctness criteria. Each implementation of the queue is checked for verifying the correctness criteria using CCSpec. Since CCSpec uses test cases to verify the correctness condition, each test case is designed to check most of the possible interleavings, to ensure that the implementation meets the specified correctness criteria. The correctness is checked by checking the logical correctness of the sequential histories generated from the concurrent history based upon the formal definition of the correctness criteria. Since all the valid sequential histories are generated by CCSpec, the count of these histories are collected and presented as practical analysis of feasibility of design. The same is done for the priority queue, since it compares two different correctness criteria than the FIFO Queue.

CHAPTER 5: FINDINGS

The findings are divided into two parts. In the first section, we present the case study used to find the practical effects of changing correctness condition for a data structure and the effect thereof on the hardware metrics. The second part is the analysis that uses the information gathered from the case study to compare and contrast the effect on performance for using different correctness criteria.

Case Studies

The correctness conditions mentioned in Section 2 of this thesis are used in shared memory access in multicore systems. The analysis of these correctness conditions give us knowledge about the usability, performance and trade-offs. In this case study, a set of non-blocking data structures are selected, and then implemented using different correctness conditions like quiescent consistency, sequential consistency, linearizability and quasi linearizability. However, a data structure is not implemented using all the correctness criteria mentioned above as some of the implementations are not feasible. For example, it is a difficult and complex task to construct a quasi linearizable priority queue. Thus there are no feasible implementation of a quasi linearizable priority queue and we do not attempt to construct such a data structure. We use multiple types of data structures, such as, array and linked list based queues, a linked list based stack and a multi dimentional list based priority queue, in order to make assessments regarding the performance benefits associated with the above mentioned correctness criteria for these data structure types. For each of the data structures, we first start by explaining the original implementation. Next the modifications required to change the correctness conditions are explained, followed by the working details of the new correctness criteria. After all the modifications, we study the effect of the different correctness criteria on

performance. We use different metrics like total number of cycles, number of level 1 cache misses and task distribution among the multiple threads. These help us analyze not only throughput, but memory utilization and work distribution. Finally, all the results are analyzed to understand the usability and trade-off related with each correctness criteria.

An Array based Queue Implementation

Linearizable Herlihy/Wing Queue

The queue implemented in [2] is an array based queue which follows linearizability. The algorithm is presented in Algorithm 1. The functioning of the queue and a proof of linearizability has been established in Section 2 of this thesis. This queue is also sequentially consistent since any data structure that is linearizable is also sequentially consistent.

Sequentially Consistent Queue

In order to study the effect of relaxed correctness conditions, the Herlihy/Wing queue is modified to be only sequentially consistent and not linearizable. The algorithm of the modified Herlihy/Wing queue, presented in Algorithm 2, is incorrect as per linearizability, but correct in terms of sequential consistency is given in. An example for the correctness according to sequential consistency is presented in Fig. 5.1. The arrays corresponding to each thread are represented in this figure. Each thread has its own array that it enqueues elements into. While dequeueing elements, each thread first tries to dequeue from the arrays of all the other threads in the system. If it cannot dequeue an element, it tries to dequeue from itself and again from the other arrays. If still there is no element to be dequeued, then the algorithm concludes that the queue is empty and returns an empty exception.

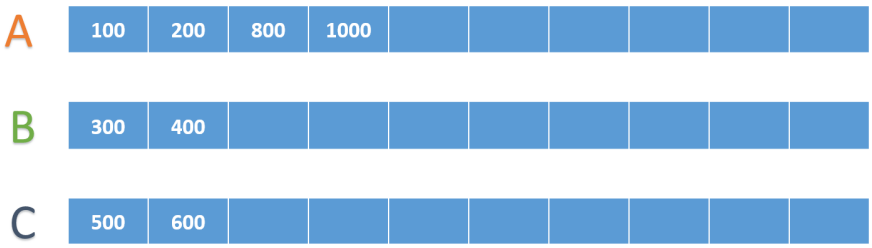

Figure 5.1: A sequentially consistent version Herlihy/Wing Queue

In Fig. 5.1 thread A has enqueued the values 100, 200, 800, 1000. Thread B has enqueued 300, 400 and thread C 500 and 600. These operations are followed by a dequeue by thread C. The following scenario is shown in Fig. 5.2. In this figure, the enqueue of values 800 and 1000 by thread A is not depicted as the two operations are not required in the counter example.

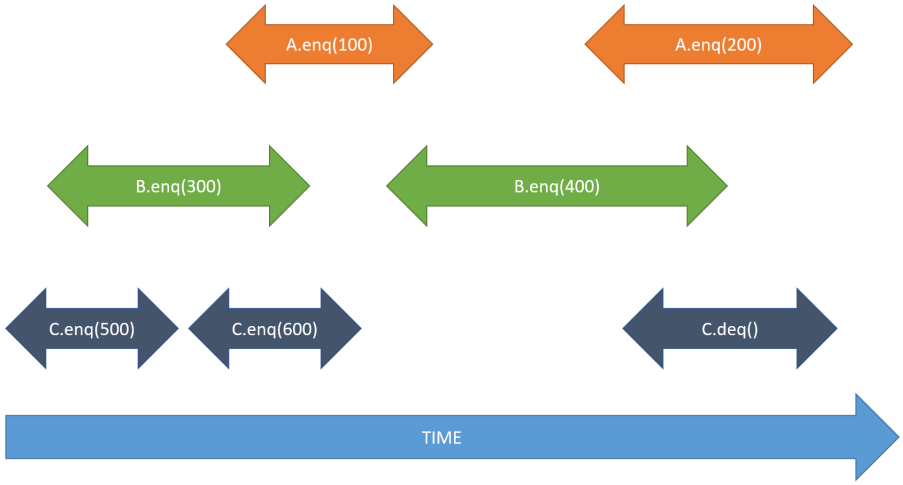

Figure 5.2: A timeline of executions based on Algorithm 2 and Fig. 5.1

According to linearizability, the dequeue by thread C should either return a value of 300 or 500,

since the enqueue of 300 by thread B and enqueue of 500 by thread C overlap, resulting in either one of them linearizing first. However, according to Algorithm 2, the dequeue operation performed by thread C will first check the array of thread A to dequeue an element. From the timeline, it is clear that the enqueue of value 100 by thread A has completed before the dequeue by thread C started. Thus thread C will dequeue 100 from the queue. This result invalidates linearizability because we would expect thread C to dequeue 300 or 500. But this is correct according to sequential consistency. The reason for correctness according to sequential consistency is based on the fact that the threads need to follow program order and not total order. Since the program order relates to ordering of operations within a thread, the enqueue of value 100 by thread A can be extended back in the timeline to overlap with the enqueue by threads B and C. This example is given in Fig. 5.3. This timeline is now consistent, even if linearization points are considered.

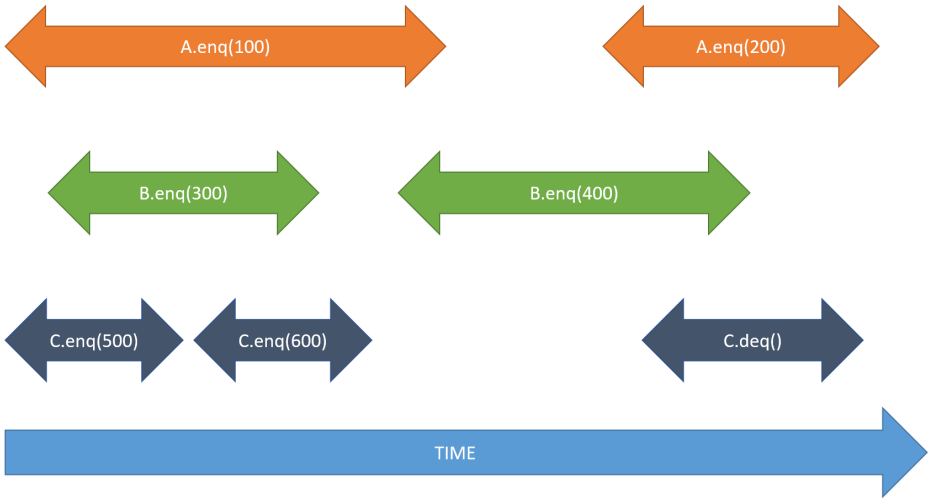

Figure 5.3: The timeline modified according to sequential consistency.

The queue being only sequentially consistent, becomes more relaxed when running concurrently. The enqueue is performed by each thread not on a single array, as done in the Herlihy/Wing queue. Instead, each array enqueues into its own array, reducing contention for enqueue on the tail of a

single array based queue. Also the dequeue operation has less contention on the queue. However, the dequeue operation is costly as well, as it has to traverse more elements than the Herlihy/Wing queue.

Quasi Linearizable K-FIFO Queue

Our next modification to the queue implementation is to make it quasi linearizable. The algorithm for the quasi linearizable K-FIFO queue is given in [20] as well as implementation details. In this thesis the segmented K-FIFO Queue has been implemented for comparison. This queue implementation is different from the other two implementations, containing a head element of the queue, which maintains the next index for dequeue. Due to this it does not have to traverse the entire queue to look for the first element to dequeue, giving it significant advantage over the previous two implementations. However this implementation bears a cost of synchronizing the head as well as the tail using atomic primitives. This provides an opportunity to observe the trade-off in using the different implementations.

Performance Results

The comparison for the performance in terms of clock or CPU cycles is presented in Fig. 5.4, Fig. 5.5 and Fig. 5.6. Each experiment was performed using an Intel Haswel processor with clock speed of 2.40 GHz with 16 GB of memory and 3MB Low Level Cache (LLC). The tests were run five times with each algorithm performing a total of 10^4 operations, and the average for each metric was calculated. Fig. 5.4 compares the CPU cycles required by each method to complete the 10^4 operations out of which 50% of the operations were enqueue and 50% were dequeue operations. Fig. 5.5 compares the CPU cycles for the same number of operations, but there are 75% enqueue operations and 25% dequeue operations, while in Fig. 5.6 there are 25%

enqueue operations and 75% dequeue operations. It is observed in all the three figures, that the number of CPU cycles required by the Herlihy/Wing queue and the Sequentially Consistent queue is significantly larger than the CPU cycles required by the K-FIFO queue. This result was expected since quasi linearizability is more relaxed than both sequential consistency and linearizability, leading to higher throughput.

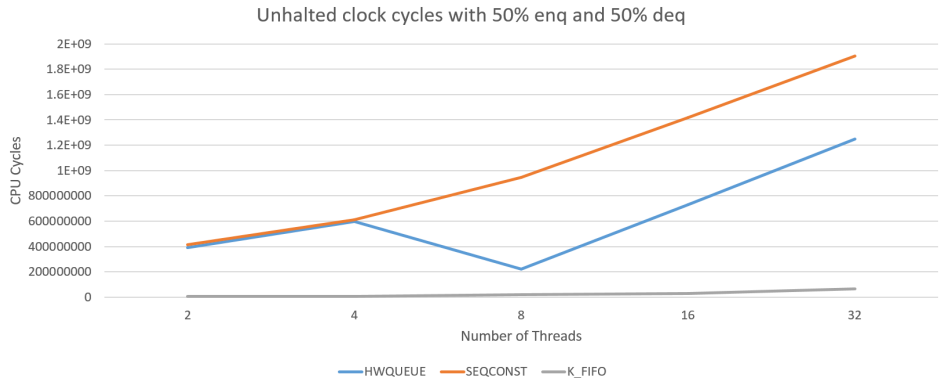

Figure 5.4: Comparision of CPU Cycles for the three different queue implementations with 50% enqueue and 50% dequeue operations

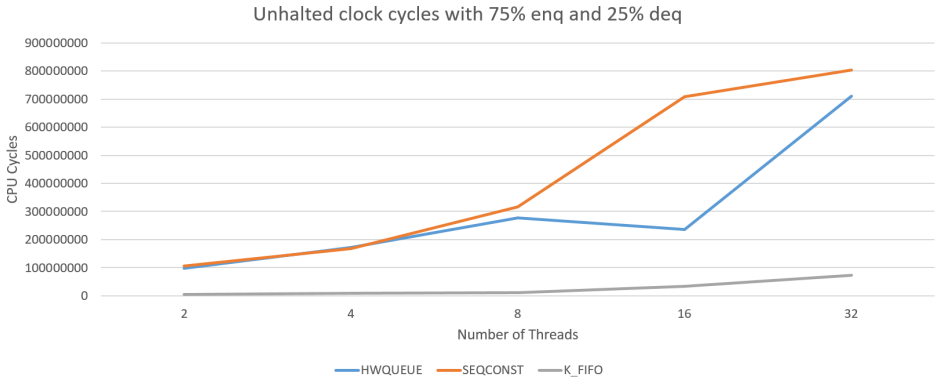

Figure 5.5: Comparision of CPU Cycles for the three different queue implementations with 75% enqueue and 25% dequeue operations

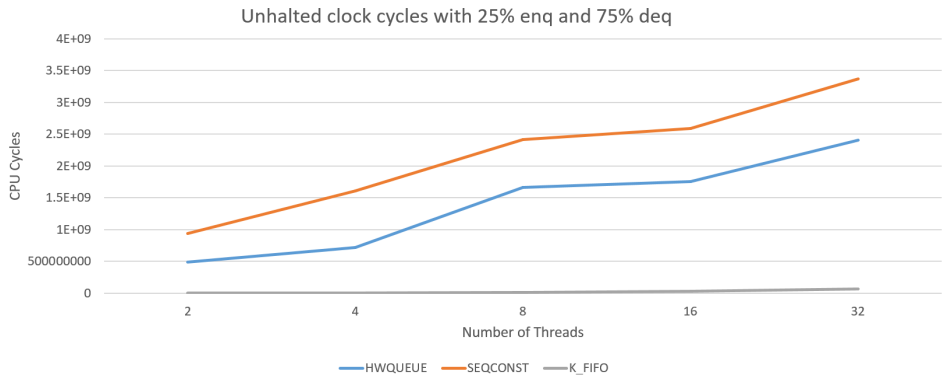

Figure 5.6: Comparision of CPU Cycles for the three different queue implementations with 25% enqueue and 75% dequeue operations

The next metric that is used for comparison is the level 1 data cache misses. This metric is useful for comparing the memory performance, since an algorithm that has fewer cache misses is well designed to utilize the cache lines and will have a significant performance increase if the number of operations being performed are the same. Recent development in the CPU performance has also increased the memory latency. Due to this the cache misses can severely affect the performance of an algorithm. As the level 1 cache is closest to the processor and smallest in size, it can be accessed within a single CPU cycle, while fetching data from the next level of cache due to a level 1 cache miss can increase the latency to 10 or more cycles depending upon the architecture.

The comparison for the level 1 cache misses are presented in Fig. 5.7, Fig. 5.8 and Fig. 5.9.The experiments were performed on the same machine and using the same configuration as mentioned for the CPU cycles. From all the three figures it is observed that the number of cache misses for the K-FIFO queue is significantly greater than the number of misses by the Herlihy/Wing queue and the Sequentially Consistent queue. Also the number of cache misses is similar for both the Herlihy/Wing queue and the Sequentially Consistent queue. This shows that providing the additional synchronization for the head of the K-FIFO queue as well as the quasi linearizability

correctness criteria leads to significant increase in the number of level 1 data cache misses. This can be an overhead in applications that are memory intensive rather than being computation intensive.

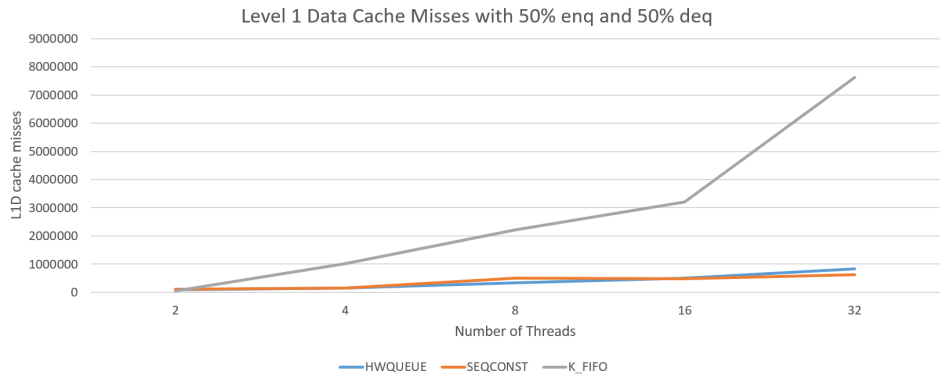

Figure 5.7: Comparision of level 1 data cache misses for the three different queue implementations with 50% enqueue and 50% dequeue operations

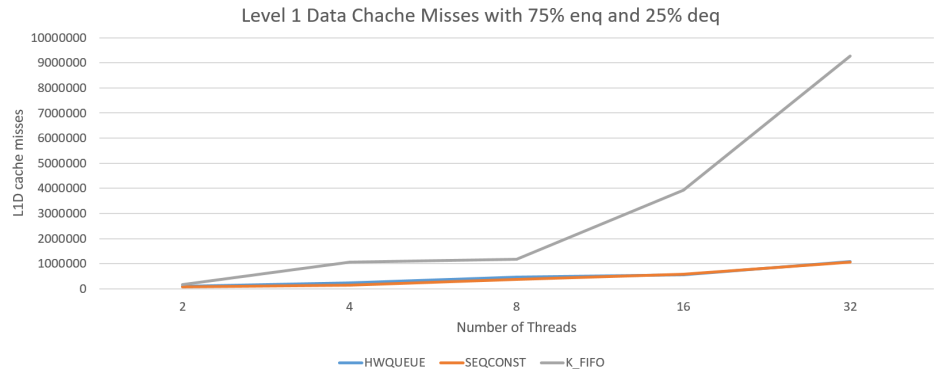

Figure 5.8: Comparision of level 1 data cache misses for the three different queue implementations with 75% enqueue and 25% dequeue operations

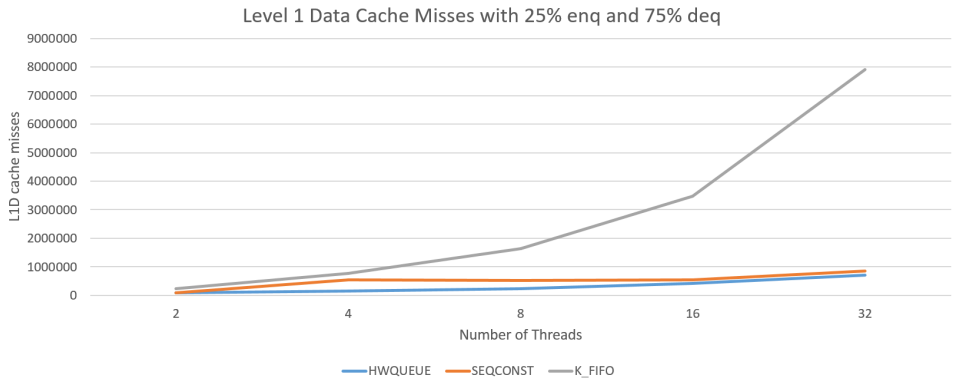

Figure 5.9: Comparision of level 1 data cache misses for the three different queue implementations with 25% enqueue and 75% dequeue operations

Complexity of Design

In the methodology section of this thesis, a method to analyze the complexity of design of concurrent data structures is presented. CCSpec has the ability to find the number of sequential histories that can be generated from the concurrent history of an execution of operations by a concurrent data structures. This has been calculated for each of the FIFO queue implementations for 2 enqueue operations and 1 dequeue operation performed by two threads. The values obtained are presented as follows:

Table 5.1: Table showing number of sequential histories obtained for different correctness criteria

Correctness Criteria	Number of sequential histories
Linearizability	3.66
Sequential Consistency	4.5
Quasi Linearizability	6.09

From Table 5.1, it is observed that the more relaxed is the correctness criteria is, the greater the number of possible legal sequential histories. The larger number of possible legal sequential his-

tories provides more opportunity for designers to employ additional synchronization techniques to increase thread interleaving in the data structure. This imposes greater complexity for the user to design an optimal data structure following a relaxed correctness criteria.

A Multi Dimensional Priority Queue

Quiescently Consistent Priority Queue

The priority queue implemented in this case study is presented in [7]. The priority queue implementation is based upon a multi-dimensional list whose performance has been shown to be better than skiplist based implementations [7]. However the implementation, even though based on a multi-dimensional list, is explained better if presented as a singly linked list. Thus for the rest of the study, we would like to consider it as a linked list based priority queue for simpler understanding. In a linked list based priority queue, after a number of insert and deletemin operations have been performed concurrently, there would be a discrepancy in the head and the actual location from where the deletemin operation would remove an element. This is similar to the problem in the Herlihy/Wing Queue, except that new elements could be added in front of the head, while a large number of elements after that have already been deleted. Therefore, a purge of the deleted elements is required to allow quicker deletemin operations.

However, this purge of the deleted nodes cannot be performed in a linearizable manner, without incurring significant overheads. In the quiescently consistent priority queue, the purge operation can be performed as it does not need to linearize with either the insert or the deletemin operations.

Linearizable Priority Queue

The implementation in [7] originally follows quiescent consistency, but for this study, has also been modified to be linearizable. This provides an opportunity to study the comparison between quiescent consistency and linearizablity, which is the main reason to study this particular data structure.

In the linearizable priority queue, the purge operation cannot be performed as the overhead to synchronize the purge is extremely costly. This leads to the inability to purge the deleted nodes, leading to larger time for the deletemin operations in a linearizable priority queue. This is the main difference between the two implementations.

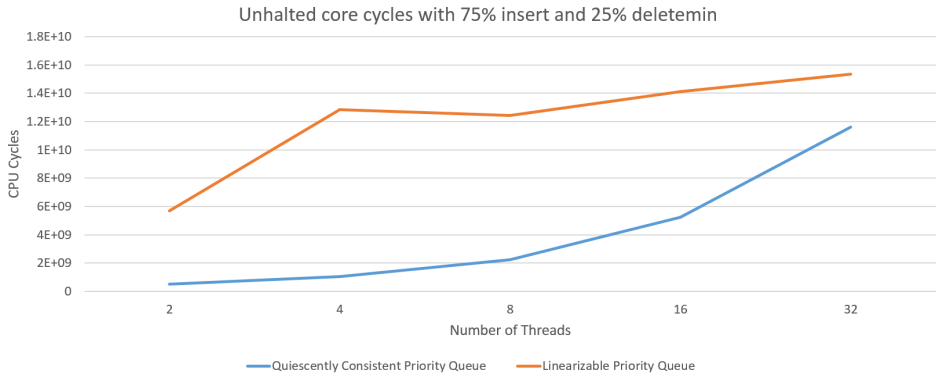

Figure 5.10: Comparision of CPU Cycles for the two different priority queue implementations with 75% inserts and 25% deletemin operations

Performance Results

The same analysis as that for queue implementation is also performed for the priority queue implementation. The graphs for the comparison of the CPU cycles is presented in Fig. 5.11, Fig. 5.10 and Fig. 5.12. All the experiments were performed using the same configuration given for

the queue comparision. It is observed that the linearizable priority queue takes significantly more cycles to complete the 10,000 operations compared to the quiesciently consistent priority queue. However in Fig. 5.10, it is observed that for large number of threads, the performance of the linearizable and quiescently consistent priority queue are almost the same. This is due to the large number of inserts and less deletemin operations, which leads to comparable performance. But there is significant difference in performance in Fig. 5.12, due to the large number of deletemin operations degrading the performance of the linearizable priority queue.

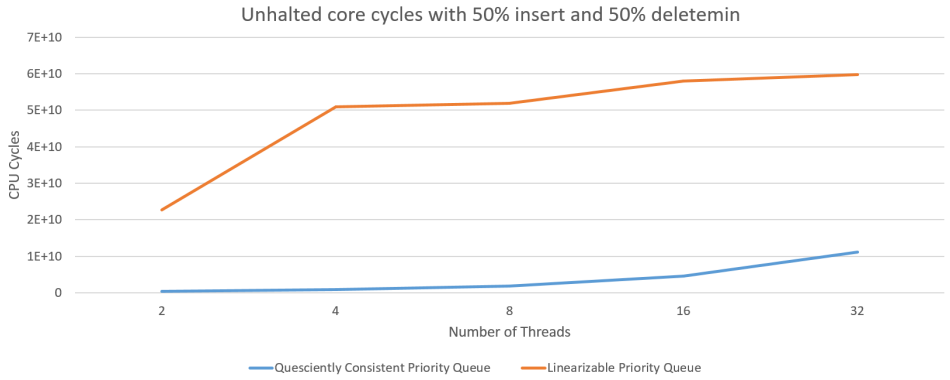

Figure 5.11: Comparision of CPU Cycles for the two different priority queue implementations with 50% inserts and 50% deletemin operations

The graphs comparing the level 1 data cache misses is presented in Fig. 5.13, Fig. 5.14 and Fig. 5.15. These graphs are also similar to the graphs for CPU Cycles, which indicate the performance dependency on the cache utilization. Again it is observed in Fig. 5.14 the increase in memory utilization for the linearizable queue. Thus we may conclude that a quiescently consistent priority queue has better memory utilization than a linearizable queue. However, this observation is not correct. An observation comparing Fig. 5.15 and Fig. 5.14, is explained further to reason the increase in cache misses for the linearizable queue. For 25% deletemin operations the number of cache misses for linearizable and quiescently consistent implementations are almost same.

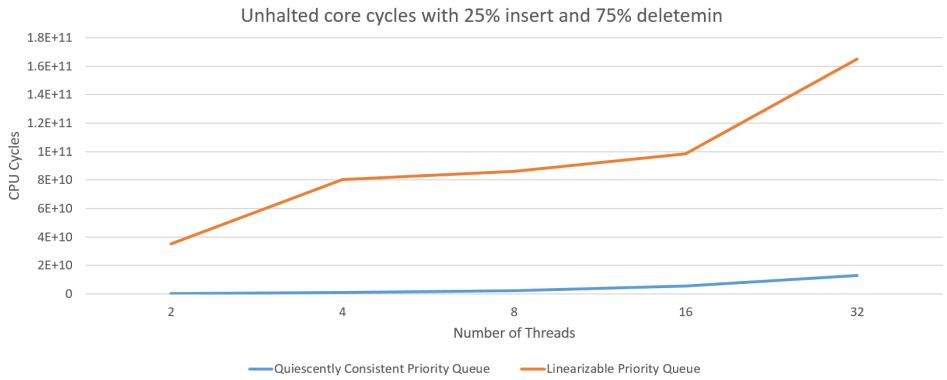

Figure 5.12: Comparision of CPU Cycles for the two different priority queue implementations with 25% inserts and 75% deletemin operations

This indicates the similarity of memory utilization for insert operations, while representing the discrepancy in cache misses for the deletemin operation. If we consider the implementation details, the lack of a purge operation for the linearizable queue increases the memory overhead for the deletemin operation. It is due to this reason that the memory utilization of the linearizazble priority queue is worse than quiescently consistent implementation.

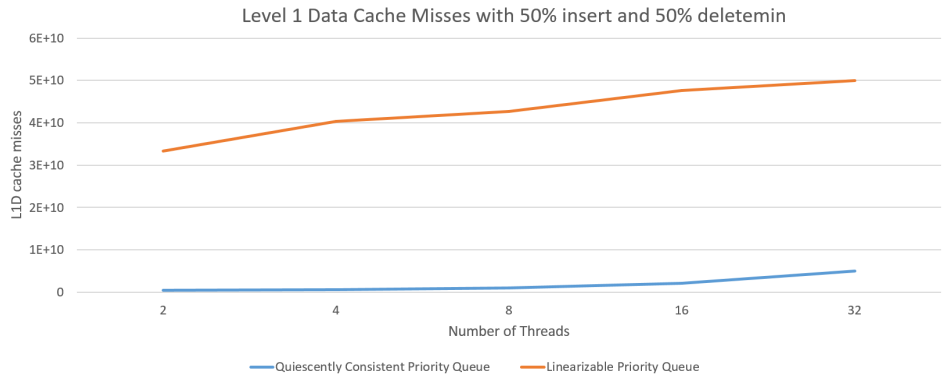

Figure 5.13: Comparision of level 1 data cache misses for the two different priority queue implementations with 50% inserts and 50% deletemin operations

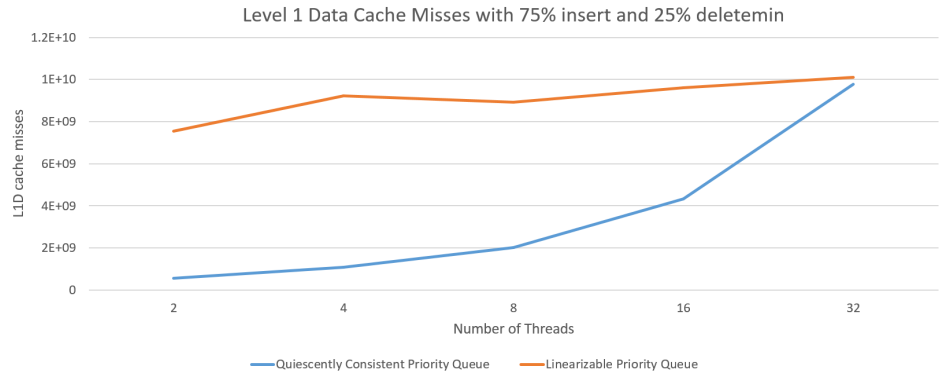

Figure 5.14: Comparision of level 1 data cache misses for the two different priority queue implementations with 75% inserts and 25% deletemin operations

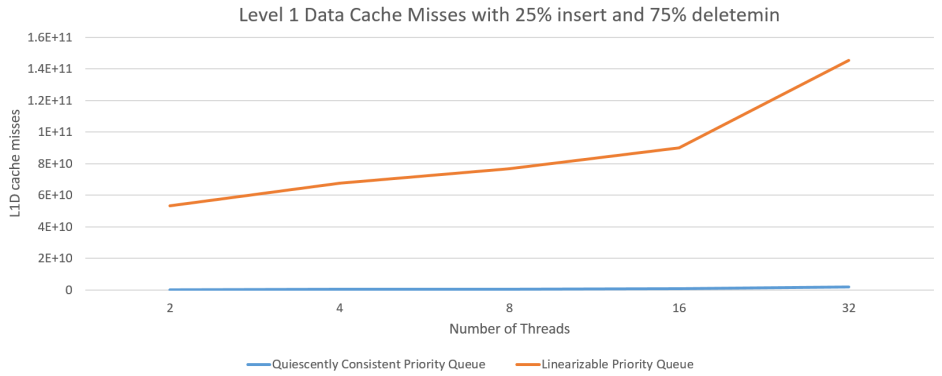

Figure 5.15: Comparision of level 1 data cache misses for the two different priority queue implementations with 25% inserts and 75% deletemin operations

Complexity of Design

The analysis performed for the FIFO Queue implementations, is also performed for the Priority queue implementation. Since the priority queue compares quiescent consistency and linearizability, it provides insights for quiescent consistency, even though it has not been studied using a mathematical analysis in the Methodology of this thesis. The number of sequential histories obtained

for 3 insert and 2 deletemin operations performed by two threads is presented in Table 5.2

Table 5.2: Table showing number of sequential histories obtained for different correctness criteria for the priority queue

Correctness Criteria	Number of sequential histories
Linearizability	6.989199
Quiescent Consistency	121.081078

From Table 5.2, it is observed that quiescently consistent implementations have much larger number of sequential histories generated as compared to linearizabile implementations. This makes quiescent consistency as complex an implementation as quasi linearizability, if not more.

A Linked List Based Wait Free Queue Implementation

Linearizable Queue Implementation

A simple linked list based queue implementation is available in [1]. A series of implementations are presented that vary from being lock based to lock free. However, a wait free implementation is not presented in the book. This implementation is given in detail in [15]. The linked list based queue implementation is based on the Tervel framework and provides a progress assurance scheme to ensure that the implementation is wait free. There is also a lock free implementation available in the Tervel framework, but the wait free queue implementation is used to compare the performance of two correctness criteria to consider the cumulative effect of the progress guarantee and correctness criteria on the design of a data structure.

The detailed implementation of the linked list based wait free queue is available in [15], and will not be covered in this thesis as it is out of the current scope of study. This implementation follows linearizability, and has been verified using CCSpec. The reason for following linearizability is

based upon either a thread being able to insert or remove the value into the queue by itself, or with the help of descriptor objects. Since the actual insertion or removal happens with the use of atomic instructions, the implementation is said to follow linearizability.

Quasi Linearizable Queue Implementation

The linearizable linked list queue was modified to follow quasi linearizability.The linearizable wait free queue does not incorporate the additional synchronization techniques necessary to allow for safe traversal through the linked list while ensuring the progress assurance and memory reclamation policies used in the implementation. Therefore, the strategy of a segmented queue as presented in the array based quasi linearizable queue is non-trivial. Due to this reason, a different approach is used to make the implementation quasi linearizable. Instead of using a single head and tail pointer, k, a number based upon the Q_0 factor for quasi linearizability, head and tail pointers were created, resulting in k different queues implemented using the Tervel framework. Whenever a thread enqueues or dequeues a value, it first performs an atomic *GetAndIncrement* operation on an atomic global counter limited from 0 to $k-1$. There are two separate counters, one for enqueue and another for dequeue. Based upon the value of the counter that the thread stores in its thread local storage, the enqueue or dequeue is performed on the head or tail corresponding to the value of the local counter. This method ensures that there are k separate linearizable queues upon which enqueue and dequeue operations are performed in a *Round Robin* technique. Thus at any instant of time, there can be at most $k-1$ operations that can happen before the the first operation is completed, resulting in k being the quasi linearization factor Q_0 as explained earlier. This implementation has also been tested using CCSpec, with the result that it passes all the unit tests while checking for quasi linearizability, while it fails linearizability tests using the same test cases. This ensures that the modified implementation of the linked list based wait free queue from [15] is quasi linearizable.

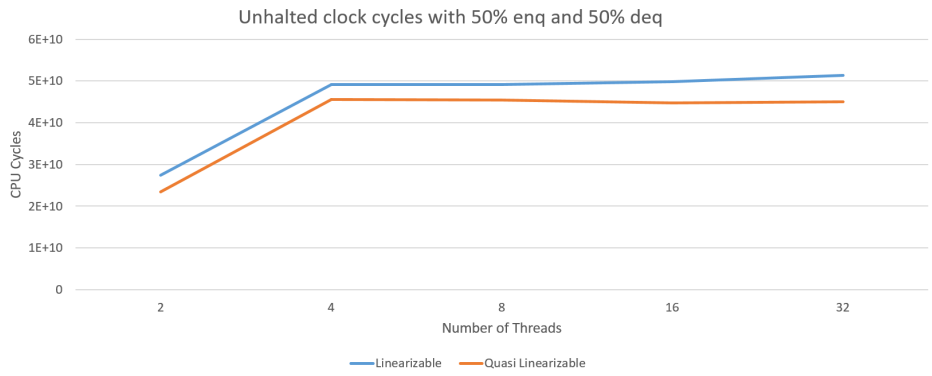

Figure 5.16: Comparision of CPU Cycles for the two different linked list based queue implementations with 50% enqueue and 50% dequeue operations

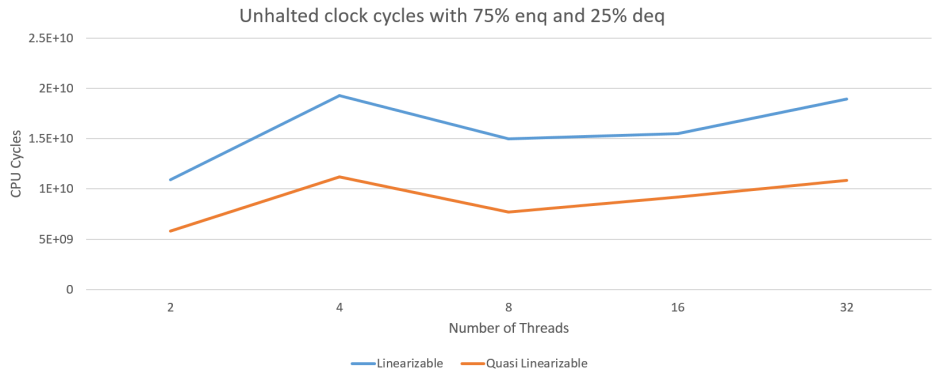

Figure 5.17: Comparision of CPU Cycles for the two different linked list based queue implementations with 75% enqueue and 25% dequeue operations

Performance Results

Now we proceed to present the results of the performance tests using the hardware metrics from PAPI. The results of measuring CPU cycles is presented in Fig. 5.16, Fig. 5.17, and Fig. 5.18. From the figures presented above, it is observed that for a linked list based queue implementation, there is not much difference between the CPU cycles utilized by the linearizable and quasi lineariz-

able queue implementations. This can be attributed to the overhead of maintaining two additional global counters for the quasi linearizable queue compared to the linearizable one.

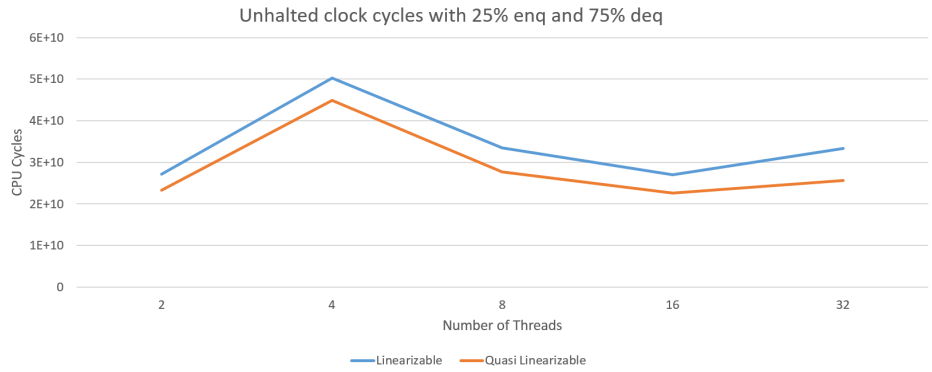

Figure 5.18: Comparision of CPU Cycles for the two different linked list based queue implementations with 25% enqueue and 75% dequeue operations

Fig. 5.19, Fig, 5.20 and Fig. 5.21 represent the level 1 data cache misses for the linearizable and quasi linearizable implementation of the linked list based queue.

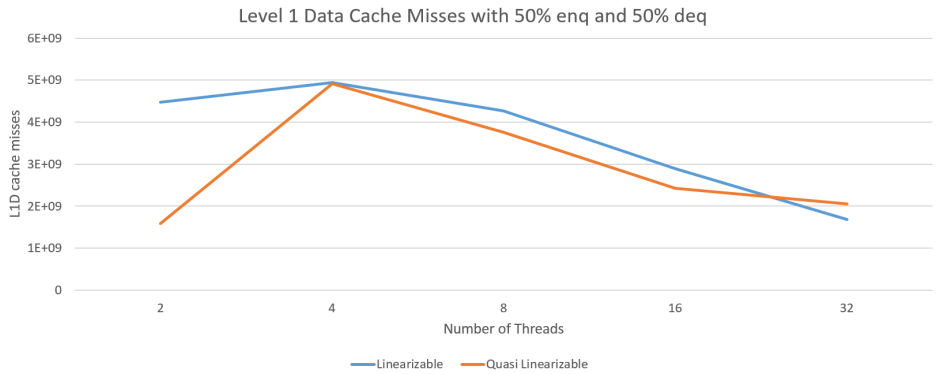

Figure 5.19: Comparision of level 1 data cache misses for the two different linked list based queue implementations with 50% enqueue and 50% dequeue operations

These graphs are similar to the ones for CPU cycles. However the number of cache misses for

two threads is much higher for the linearizable queue than the quasi linearizable one. This differs significantly than our previous observation of higher cache misses for the quasi linearizable array based queue when compared against the linearizable Herlihy/Wing Queue. A reason for this can be attributed to the value of k used for this implementation. Since k is kept constant at 2 for all the experiments, it might lead to better memory utilization with only two threads operating on the queue. Thus for the linked list based queue implementation we observe very similar results due to different overheads for each implementation.

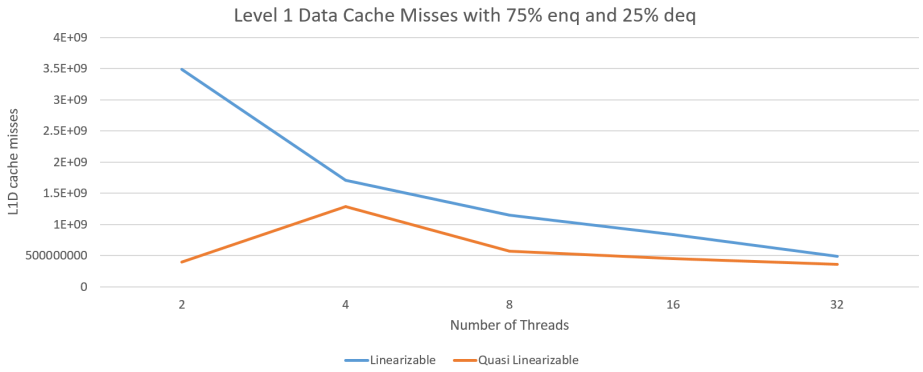

Figure 5.20: Comparision of level 1 data cache misses for the two different linked list based queue implementations with 75% enqueue and 25% dequeue operations

A Linked List Based Wait Free Stack

Linearizable Stack Implementation

The last data structure that is used for the case study is a stack. Stacks have always presented a challenge for concurrent programming due to the sequential bottleneck of the top pointer. In a linked list based implementation in [15], we find the same approach for the progress assurance scheme allowing a wait free implementation, as observed in the previous queue implementation.

The wait free implementation of the stack is also linearizable due to the *push* and *pop* operations appear to occur in a single atomic step. This is due to the same reason presented for the linked list based queue implementation explained previously. The correctness criteria has also been verified using CCSpec.

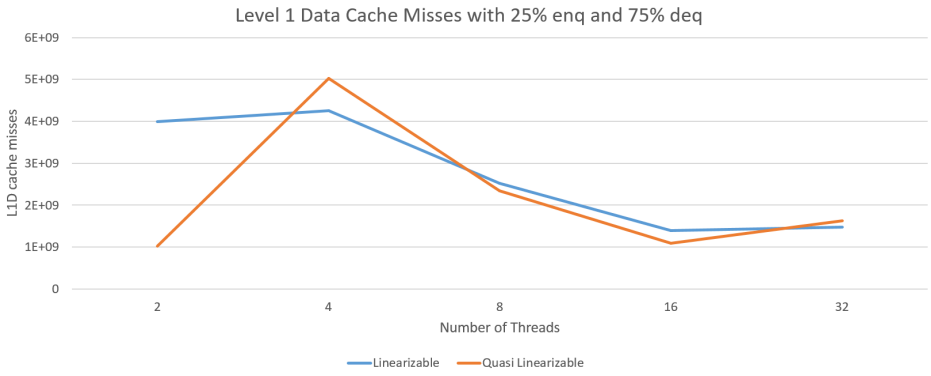

Figure 5.21: Comparision of level 1 data cache misses for the two different linked list based queue implementations with 25% enqueue and 75% dequeue operations

Quasi Linearizable Stack Implementation

The linearizable stack implementation is modified to change the correctness condition to quasi linearizability. In [1], an elimination array is used to construct a linearizable linked list based stack, which overcomes the sequential bottleneck offered by the single top pointer. However the implementation in [15] doesn't utilize the elimination array, since the implementation only assures lock freedom and not wait freedom. This linearizable wait free stack is then modified to follow quasi linearizability. The method used is similar to the one used for the linked list based queue implementation. For the quasi linearizable stack, k top pointers are used to represent k different stacks. Again k represents the quasi linearization factor Q_0. Two global counters are used to circularly assign one of the k stacks for a push or pop operation as requested by the different

threads. The verification for quasi linearizabilty was confirmed using CCSpec. Similar to the queue implementation, a similar test case was used to verify quasi linearizability, while refuting linearizability for the modified stack.

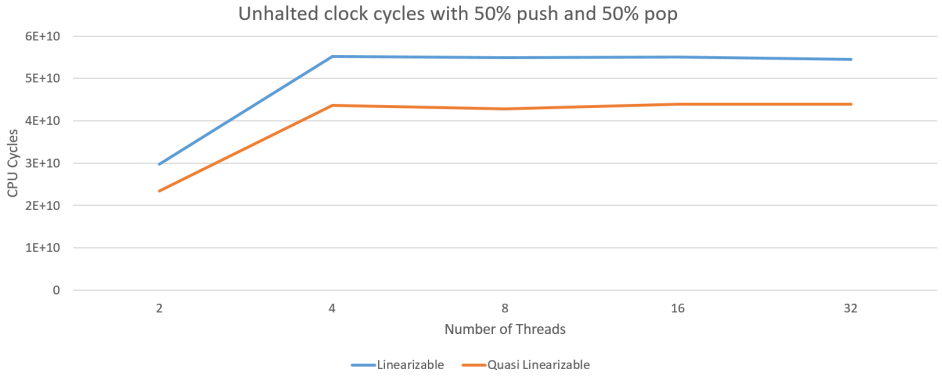

Figure 5.22: Comparision of CPU Cycles for the two different linked list based stack implementations with 50% push and 50% pop operations

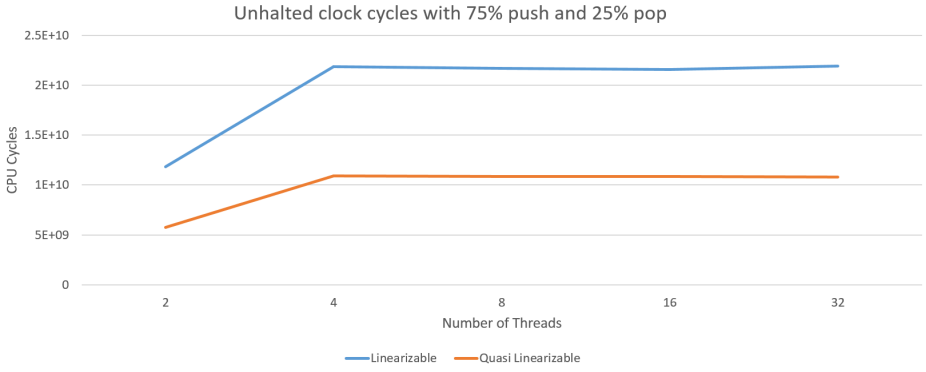

Figure 5.23: Comparision of CPU Cycles for the two different linked list based stack implementations with 75% push and 25% pop operations

Performance Results

Fig. 5.22, Fig. 5.23 and Fig. 5.24 represent the comparison of the CPU cycles used by the linearizable and quasi linearizable stack implementation. The figures represent the difference in performance of the two implementations. The quasi linearizable implementation is able to perform better than the linearizable stack. The improvement is not as significant as the array based queue implementation. However, the improvement is similar to the one observed for the linked list based queue implementation.

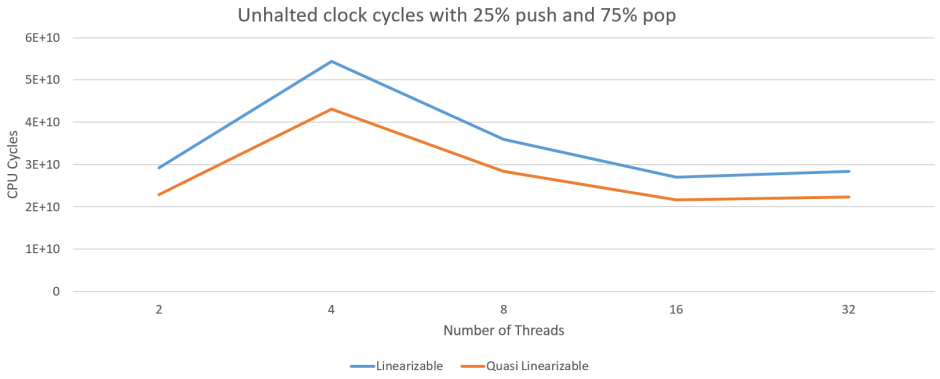

Figure 5.24: Comparision of CPU Cycles for the two different linked list based stack implementations with 25% push and 75% pop operations

This similarity will be further studied in Section 7, but it is an indication of similarity of performance and the implementation of the data structure. Even though the stack is a completely different data structure compared to the queue, when implemented using a linked list, their behavior is similar in terms of the atomic operations performed in both the data structures. In a queue, the atomic operations are performed on the head and tail pointers, and the nodes that they are pointing to. Similarly in a stack, the atomic operations are performed only on the top pointer and the node that it is pointing to. This leads to the time taken for completion of these operations to be similar. Thus we see a similarity between the plots for performance of the linked list based queue and stack. This

difference in performance of the linearizable and quasi linearizable stack shows improvement in performance if the correctness condition is changed to quasi linearizability from linearizability.

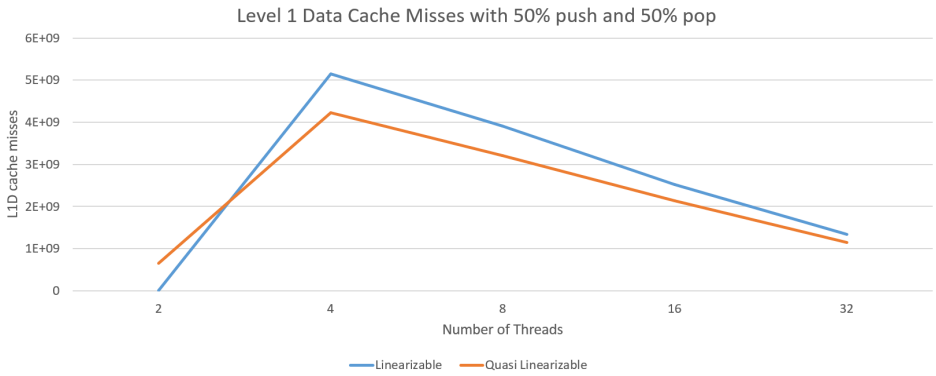

Figure 5.25: Comparision of level 1 data cache misses for the two different linked list based stack implementations with 50% push and 50% pop operations

Fig. 5.25, Fig. 5.26 and Fig. 5.27 represent the result for the level 1 data cache missed for the linearizable and quasi linearizable linked list based stack implementation. From these plots it is observed that the difference in the cache misses is not significant at all, except for the case when there are 75% push operations and 25% pop operations. In this particular case, we observe a large number of cache misses for the linearizable stack implementation as compared to the quasi linearizable one. Another observable fact is that there are significantly less number of cache misses for the linearizable implementation in the case of just two threads running the operations. This is exact opposite of the observation made using linked list based queue implementation. In the queue results, the number of cache misses for the linearizable linked list based queue is much more compared to the quasi linearizable implementation of the linked list. The reason behind this observation will be discussed in detail in Section 7. However, this contrast in the results of the queue and stack implementation is critical for finding out the exact reason for such behavior and provides additional information that could not have been reasoned by simply studying one data

structure. Thus in this thesis, different data structures have been used for the case study to present similarities and differences that are not observable by the analysis of a single data structure, or a single correctness condition.

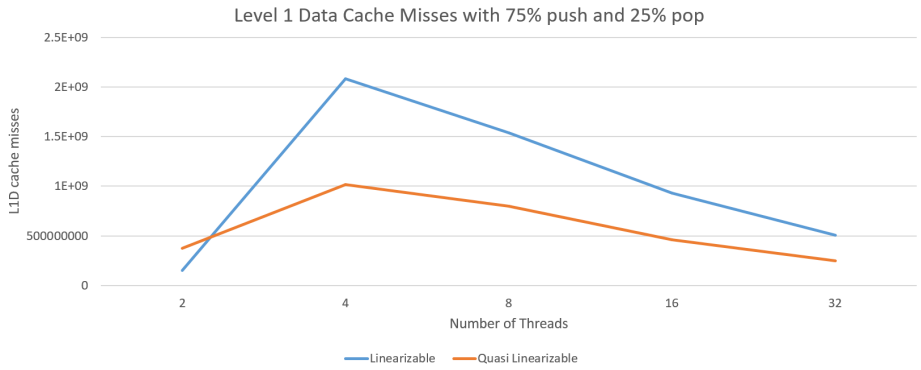

Figure 5.26: Comparision of level 1 data cache misses for the two different linked list based stack implementations with 75% push and 25% pop operations

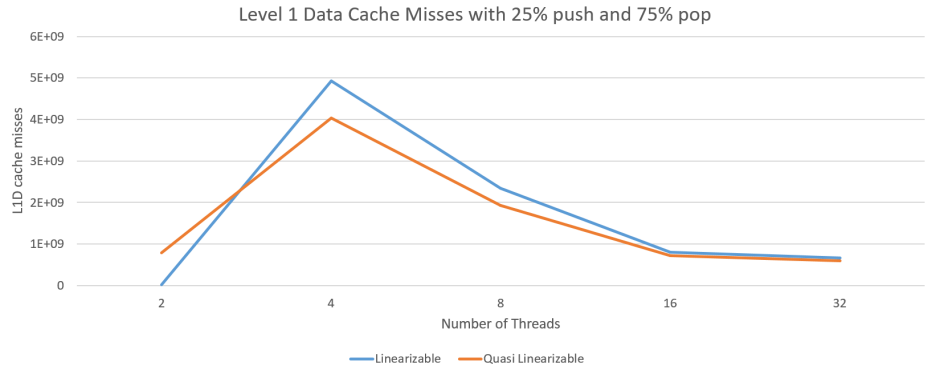

Figure 5.27: Comparision of level 1 data cache misses for the two different linked list based stack implementations with 25% push and 75% pop operations

Analysis

The four case studies of the different correctness criteria using various data structures, provides data for analysis and comparison of the correctness criteria. We now discuss the various implications of the results obtained in the case study to compare and contrast a pair of correctness criteria based upon the experiments performed in the case study. Then each correctness criteria is considered for its usability and performance.

Sequential Consistency and Linearizability

The array based queue implementation provides us with details about the comparison between linearizability and sequential consistency. From the results obtained for CPU cycles, there is no significant difference between the performance of the linearizable and sequentially consistent queue. Though in [9] there is theoretical proof of increase in performance while using relaxed correctness criteria, the practical use of relaxed correctness criteria doesn't always produce better results in performance. This is often due to some trade off that is required to maintain the correctness according to the formal definitions. The sequentially consistent array based queue was expected to perform better than the linearizable Herlihy/Wing Queue from a theoretical aspect, due to the reduced contention using multiple arrays to maintain the queue. However, this performance gain was not visible in the performance metrics using the CPU cycles. This shows a difference between the theoretically predicted behavior and the practical outcome of implementing a concurrent data structure using different correctness criteria. While comparing the level 1 data cache misses for the array based queue implementation, we again observe that there is not a significant difference between the number of cache misses for both the sequentially consistent and linearizable queue implementations. Again the memory utilization by the two algorithms are quite similar. Since cache misses haven't been studied for concurrent data structures, we can only conclude that for

the implementation used in this thesis, the cache misses for a linearizable and sequentially consistent queue are similar. Thus for all the performance metrics, there is not a significant difference between a linearizable and sequentially consistent data structure.

Complexity of Design

To assess the complexity of both the correctness criteria, it is evident that the linearizable Herlihy/Wing Queue has a much simpler design than the sequentially consistent version of the same. The complexity in designing the sequentially consistent queue to provide less contention using multiple arrays is unable to provide sufficient performance benefits. Moreover, since sequentially consistent concurrent objects are not composable, the queue cannot be used to construct another data structure, without complete testing of the new data structure to verify that it follows sequential consistency. Thus the complexity of a sequentially consistent data structure is much less compared to a linearizable one. Since the performance of a linearizable data structure is similar to a sequentially consistent one, there is not much performance benefit obtained by relaxing the correctness condition. Also the loss of composability of concurrent objects implies that it is better to use linearizability as the correctness condition to design data structures than sequential consistency. This allows for lesser code complexity, easier design methodology for developers, composable objects, and equivalent performance compared to sequentially consistent data structures.

Linearizability and Quasi Linearizability

Linearizability has been compared against quasi linearizability three times in the case studies. The first comparison comes from the array based queue implementation. In this implementation it is observed that the performance benefit in terms of throughput is significantly higher for a quasi linearizable implementation. But the trade-off is in terms of cache utilization, which is much worse

for the quasi linearizable implementation, when compared to the linearizable one.

In the linked list based queue implementation, we do not observe a significant increase in throughput of the quasi linearizable queue as compared to the linearizable queue, and the same is observed for the linked list based stack. These experiments show that throughput of a quasi linearizable implementation is higher than a linearizable one, but the difference in throughput can vary significantly based upon the data structure and their implementation details. For the array based queue implementation, there is a large overhead of maintaining the FIFO property using an array for the Herlihy/Wing Queue. This overhead leads to poor performance for the linearizable Herlihy/Wing Queue when compared to the K-FIFO queue implemented using quasi linearizability. But for the linked list based queue and stack implementations, the overhead for the linearizable queue or stack is comparable to the overhead for the quasi linearizable implementation. However, a quasi linearizable implementation is able to reduce the contention by allowing multiple atomic operations to succeed, while a linearizable queue can only allow a single atomic operations to succeed, in order to maintain the correctness condition. Thus we always see an improvement in throughput for the quasi linearizable implementations over the linearizable implementations. But if there is an additional overhead for the linearizable implementation, then we can expect to see a larger improvement in throughput for the quasi linearizability over linearizability.

For both the linked list based queue and stack, the number of cache misses are very similar. This is in contrast to the behavior observed for the array based queue implementation, where the number of cache misses for the quasi linearizable implementation is much higher than the linearizable Herlihy/Wing Queue. This can be explained due to the contiguous memory allocation of an array, which, if it properly fits into the cache memory, then the cache misses can be significantly reduced. The linearizable implementation's access of the array is in a contiguous manner, where it scans the array starting from the first location, till it reaches the tail, or it encounters an element for dequeue. This helps in the array being loaded once to the cache and then being read continuously from

the cache memory, thereby reducing the number of cache misses. But for the K-FIFO queue, the location for the enqueue and dequeue are continuously being updated, resulting in constant update of the cache memory. This leads to higher conflicts in the cache allocation and thereby leads to higher cache misses. Also since the K-FIFO queue uses a segmented array based approach, this leads to poor cache behavior. But this is not the case for a linked list based queue or stack implementation. Since a linked list is non contiguous memory allocation, the cache performance is much worse than an array based implementation. This also means that there is not much difference in the number of cache misses for the linearizable and quasi linearizable implementation. This explains the reason behind very similar number of data cache misses for both the implementations.

Another observation is the significant difference in the number of cache misses when there are only two threads operating in the linked list based queue and stack implementation. But for the queue, the number of misses for the linearizable implementation is much higher than the quasi linearizable one, while in the case of the stack the scenario is completely reversed, that is, the number of cache misses is significantly greater for the quasi linearizable implementation than the linearizable one. The explanation is available when examining both the situations together. The reason is that for the queue implementation, when two threads are operating, and the quasi linearization factor is 2, then the array used to store the head and tail locations provide a contiguous memory for better cache utilization when implementing the quasi linearizable queue. But for the stack, the same concept cannot be extended to predict similar behavior, since the push and pop using only two threads leads to access of the same location when there is a push immediately followed by a pop. Thus in this case the linearizable stack, this improves the cache performance, while for the quasi linearizable stack, this leads to poor cache behavior since the array of top pointers causes more conflict misses due to more number of locations being accessed.

Complexity of Design

In terms of complexity, a linearizable implementation is much simpler to design compared to a quasi linearizable implementation of the same data structure. This is evident when comparing the algorithms for the queue and stack implementations. But a quasi linearizable data structure performs better in terms of throughput when compared to a linearizable implementation. Also as both of the correctness criteria can be used to design composable objects, quasi linearizability is a strong alternative to linearizability when improved performance is desirable. The only exception to this case is when cache utilization is considered to be a more important metric than throughput, and the data structure uses contiguous memory allocation, it would be preferable to use linearizability. However, since it is always easier to design algorithms using linearizability, the usage of quasi linearizability is reserved for designers knowledgeable on synchronization techniques to reduce contention on concurrent resources.

Quasi Linearizability and Sequential Consistency

The only case study that provides insight into the comparison of quasi linearizability and sequential consistency is the array based queue implementation. From the results obtained by comparing the CPU cycles and cache misses, it is observed that the throughput of a quasi linearizable implementation is significantly higher than a sequentially consistent implementation of the same data structure. Even though both quasi linearizability and sequential consistency are supposed to be more relaxed than linearizability, the difference in throughput is significant to put sequential consistency at a disadvantage when throughput is of utmost importance while designing a data structure.

Both the implementations try to reduce contention for atomic operations when compared to linearizable implementation, but the effectiveness of quasi linearizability to show practical results is

much better than sequential consistency. Another interpretation of the results would be to conclude that the overhead of incorporating the rules of sequential consistency are far greater than that of quasi linearizability. Thus in terms of throughput and relaxation of rules for correctness, quasi linearizability is much better than sequential consistency. This can also be attributed to the later development of quasi linearizability as compared to sequential consistency. Since quasi linearizability was directly aimed at relaxing the conditions for correctness as compared to linearizability, it is able to address the major restrictions and relax the correctness condition to enhance throughput. However, sequential consistency was proposed as a correctness model in which it is possible to place a partial ordering on the methods called by an individual process, but not on all the processes as a whole. The motivation for sequential consistency is therefore focused on the capabilities of the concurrent system rather than performance. Thus the practical benefits of quasi linearizability can be directly observed, while the relaxation of correctness offered by sequential consistency remain theoretical at best.

In terms of cache utilization, sequential consistency is much better than quasi linearizability. This is again due to the allocation of contiguous memory and its utilization. The sequentially consistent queue used a matrix to implement the FIFO property of the queue. If the matrix is able to fit well into the level 1 cache, it can significantly reduce the cache misses. However, the cache misses is higher for the quasi linearizable implementation as explained in the previous part comparing linearizability and quasi linearizability.

Complexity of Design

Since the throughput of a quasi linearizable data structure is significantly greater than a sequentially consistent data structure, it is necessary to use the former for time critical applications. But if the application is memory critical, like in embedded systems, then it would be better to use

sequential consistency. However, these interpretations are only for data structures that utilize contiguous memory allocation. As observed in the case of linearizability and quasi linearizability, the performance of a linked list based data structure is completely different when compared to an array based implementation.

In terms of code complexity, both the correctness criteria can be difficult to implement. In terms of the array implementation, it is easier to construct the sequentially consistent queue than a quasi linearizable queue. But for other implementations, this interpretation might not hold. As a basic idea, it can be considered as hard to design a quasi linearizable data structure as it is to construct a sequentially consistent one. A note to be observed at this point is that when we mention sequentially consistent data structures, we refer to the data structures that follow only sequential consistency and not linearizability. Since we know that an implementation that is linearizable is also sequentially consistent, such an implementation is only considered as linearizable for this thesis. Usability is not only restricted to performance or design complexity, but also to composability. Since quasi linearizable objects are composable they are preferred as compared to sequentially consistent objects, that are not composable.

Linearizability and Quiescent Consistency

Linearizability and quiescent consistency have been compared in this thesis by studying the multi-dimensional list based priority queue as presented in [7]. In terms of both CPU cycles and cache misses, the quiescently consistent priority queue performs better than the linearizable one. The main reason for this is the inability to purge the deleted nodes in the linearizable implementation. Since the purge operation cannot take place in a linearizable manner, it is performed to follow quiescent consistency. This leads to improved performance for the quiescently consistent priority queue. Along with CPU cycles, the quiescently consistent implementation also shows lesser cache

misses as compared to the linearizable one. This is completely different than the results obtained for the queue implementation when comparing linearizability and quasi linearizability.

The practical results observed confirm according to the theoretical analysis of the two correctness criteria. The formal definition of quiescent consistency indicates a much more relaxed correctness condition as compared to linearizability. Therefore the performance of the priority queue declines when linearizability is enforced. The restrictions placed on a data structure designed for linearizability is the primary cause for a decrease in performance when compared to data structures designed for a different correctness condition.

In the previous comparisons, it has been observed that there is a trade-off between memory and throughput when implementing data structures with different correctness criteria. However, in the case of quiescent consistency alone, there is no trade-off between memory and throughput. In fact for both memory utilization and throughput, the quiescently consistent implementation outperforms the linearizable priority queue. In all the previous analysis, none of the high throughput implementations were able to outperform the other implementation in terms of memory utilization. Thus quiescent consistency appears to offer better performance without any practically visible trade-offs.

Complexity of Design

Since the quiescently consistent priority queue is better in terms of both throughput and cache performance, it is more suitable to use quiescent consistency as the correctness criteria while designing data structures for any application. It can be used in place of linearizability to design data structures for both time and space critical applications.

In terms of design complexity, a quiescently consistent data structure is significantly more complex

than a linearizable one as observed from Table 5.2. The number of sequential histories generated for quiescent consistent priority queue is much larger than the number of histories generated for a linearizable implementation. This shows a greater complexity for designing quiescently consistent objects than linearizable ones. Linearizability can be achieved simply by incorporating linearization points into the data structure methods. Since the operations of the method appear to take effect instantaneously between the invocation and response, such a design will clearly respect the real-time ordering required by linearizability. But in the case of quiescent consistency, the entire design needs to be changed to allow relaxation of the history thus produced by the concurrent implementation. It therefore increases design complexity and is one of the main reasons why linearizability is still used to construct concurrent data structure.

Final Remarks

In this section of the thesis, we have covered the analysis of pairs of correctness criteria, whose performance results were obtained from the case study. Now we present a combined analysis of the different correctness criteria. The analysis is summarized in Table 5.3

The first parameter, *Composability*, is obtained from the formal definitions of each correctness criteria. This parameter is available in the literature for each correctness criteria.

The *Throughput* has been analyzed for each pair of correctness criteria. We summarize the results based upon the data obtained from the case study. Data structures following linearizability are observed to have low throughput, as they perform significantly worse than quiescently consistent and quasi linearizable data structures. The same is true for sequentially consistent data structures. Quiescently consistent and quasi linearizable data structures show high throughput as they require significantly less CPU cycles to complete the same number of operations as a linearizable or sequentially consistent execution.

Table 5.3: Table comparing different correctness criteria

Correctness Criteria	Composable	Throughput	Memory Utilization	Complexity
Quiescent Consistency	Yes	High	Low cache misses	Complex implementation
Sequential Consistency	No	Low	Low cache misses	Simple implementation
Linearizability	Yes	Low	Low cache misses	Very simple implementation
Quasi Linearizability	Yes	High	High cache misses	Complex implementation

Memory Utilization is based upon the number of level 1 data cache misses. These results, inferred from the case studies, indicate that quiescently consistent, sequentially consistent and linearizable data structures have lower number of cache misses as compared to quasi linearizable data structures. Quasi linearizable data structures have high cache misses, which increases if the data structure is based upon contiguous memory allocation.

The last column represents the *Complexity of Design* of the data structures implemented using the correctness criteria mentioned in the table. The data for complexity for linearizability, sequential consistency, quasi linearizability and quiescent consistency are available in the case study for an array based concurrent FIFO queue and multi dimensional list based priority queue. The data in Table 5.1 and Table 5.2 are sufficient to decide the increasing complexity of design for linearizability, sequential consistency, quasi linearizability and quiescent consistency. A larger number of allowable method call orderings provides more opportunities for a design to meet the target correctness condition. In order to optimize a data structure to its full potential under a relaxed correctness condition, a designer must employ synchronization techniques to reduce contention on

shared resources. Thus, data structures implemented using linearizability have very simple implementations. Using sequential consistency is more complex, but simpler when compared to quasi linearizability and quiescent consistency, which have highly complex implementations.

When an overall comparison is performed, no single correctness criteria comes out as the best that can be used in all possible implementations. Linearizability has slower performance when compared to quiescent consistency and quasi linearizability, but better memory utilization and a more simplistic design. For this reason, it is preferred in many implementations of concurrent data structure. Sequential consistency has similar metrics as compared to linearizability, but is not composable, and is therefore not preferred over linearizability. Quasi linearizability gives better throughput, but has poor memory utilization and higher complexity of design. Quiescent consistency has higher throughput and better memory utilization, but the complexity of design often makes implementation of data structures using quiescent consistency infeasible. It therefore becomes necessary to consult the values given in Table 5.3, and the design requirements before selecting a correctness condition to implement a concurrent data structure or algorithm.

CHAPTER 6: CONCLUSION

This thesis presents an overview of the different concurrency correctness conditions for shared memory access. The different correctness conditions were studied in detail to understand the basic formal definitions and their informal explanation. This was followed by some example data structures that had been implemented using the different correctness criteria. Finally we did four case studies based on FIFO queues, priority queues, linked list based queues and stacks. Using all of these studies, the CPU and cache performance were observed for different implementation of the data structures using different correctness conditions. It was observed that the performance would improve when the correctness condition was relaxed. This is not true for the sequentially consistent queue implemented in the case study, but it is true for the other comparisons for the FIFO queue, priority queue, linked list based queue and stack implementations. It is observed that the K-FIFO queue performs significantly better than the Herlihy/Wing queue and the quiescently consistent priority queue performs significantly better than the linearizable priority queue. On the other hand it has been observed that the number of level 1 cache misses is significantly larger for the K-FIFO queue as compared to the Herlihy/Wing queue or the sequentially consistent queue. This is not observed in the comparison between the two priority queue implementations, or the linked list based queue and stack implementations which implies that the observed behavior is not consistent with relaxation of the correctness condition. However the trade-off between performance and cache misses does exist in certain scenarios, while in others the performance is based upon the cache utilization. Thus in this thesis we not only studied the different correctness conditions, but can also conclude that there are trade-off between performance and memory utilization among the different correctness conditions. Based upon use of contiguous memory locations, the more relaxed the correctness condition, the performance is better in terms of CPU Cycles; while the stricter is the correctness condition, there is better memory utilization as observed in the lesser

number of cache misses.

APPENDIX A: ALGORITHMS USED IN THE SURVEY

ALGORITHM 1: Herlihy/Wing Queue

```
Data: Array:items[capacity], AtomicInt:tail
Procedure Enqueue(q:queue, x:item)
    position := AtomicFetchAdd(q.tail);
    AtomicStore(q.items[position], x);
    return
Procedure Dequeue(q:queue)
    limit := AtomicLoad(q.tail);
    for i in range 0 to limit do
        x := AtomicExchange(q.items[i], NULL);
        if x is not NULL then
            return x
        end
    end
    return NULL
```

ALGORITHM 2: Modification to Herlihy/Wing Queue

```
Data: Array:items[capacity][num_threads], AtomicInt:tail
Procedure Enqueue(q:queue, x:item, t:thread_id)
    position := AtomicFetchAdd(q.tail[t]);
    AtomicStore(q.items[position][t], x);
    return
Procedure Dequeue(q:queue, t:thread_id)
    limit := AtomicLoad(q.tail[t]);
    for i in range t+1 to 2*(num_threads)+t do
        for j in range 0 to limit do
            x := AtomicExchange(q.items[j][i % num_threads], NULL);
            if x is not NULL then
                return x
            end
        end
    end
    return NULL
```

ALGORITHM 3: K-FIFO Queue Enqueue Function Implementation

```
Procedure Enqueue(q:queue, x:item, t:thread_id)
    while true do
        tail_old = get_tail();
        head_old = get_head ();
        item_old , index = find_empty_slot ( tail_old , k );
        if tail_old == get_tail () then
            if item_old . value == EMPTY then
                item_new = atomic_value ( item , item_old . version + 1);
                if CAS (& tail_old → segment [ index ], item_old , item_new ) then
                    if committed ( tail_old , item_new , index ) then
                        return true
                    end
                end
            end
            else
                if tail_old . value + k == head_old . value then
                    if segment_not_empty ( head_old , k ) then
                        if head_old == get_head () then
                            return false
                        end
                    end
                    else
                        advance_head ( head_old , k )
                    end
                end
                advance_tail ( tail_old , k )
            end
        end
    end
```

ALGORITHM 4: K-FIFO Queue commited Function Implementation

```
Procedure commited(tail_old, item_new, index)
    if tail_old → segment [ index ] != item_new then
        return true
    end
    head_current = get_head ();
    tail_current = get_tail ();
    item_empty = atomic_value ( EMPTY , item_new . version + 1);
    if in_queue_after_head ( tail_old , tail_current , head_current ) then
        return true
    end
    else
        if not_in_queue ( tail_old , tail_current , head_current ) then
            if ! CAS (& tail_old → segment [ index ], item_new , item_empty ) then
                return true
            end
        end
    end
    else
        // in queue at head
        head_new = atomic_value ( head_current.value , head_current.version + 1);
        if CAS (& head , head_current , head_new ) then
            return true
        end
        if ! CAS (& tail_old → segment [ index ], item_new , item_empty ) then
            return true
        end
    end
    return false
```

ALGORITHM 5: K-FIFO Queue Dequeue Function Implementation

```
Procedure Dequeue(q:queue, x:item, t:thread_id)
    while true do
        head_old = get_head ();
        item_old , index = find_item ( head_old , k );
        tail_old = get_tail ();
        if head_old == get_head () then
            if item_old.value != EMPTY then
                if head_old.value == tail_old . value then
                    advance_tail ( tail_old , k );
                end
                item_empty = atomic_value ( EMPTY , item_old.version + 1);
                if CAS (& head_old [ index ], item_old , item_empty ) then
                    return item_old.value
                end
            end
            else
                if head_old.value == tail_old.value && tail_old.value == get_tail () then
                    return NULL
                end
                advance_head ( head_old , k );
            end
        end
    end
```

APPENDIX B: TOOLS USED FOR VERIFYING CORRECTNESS AND PERFORMANCE MEASUREMENT

Correctness Condition Specification Tool

Complex concurrent systems generally must undergo performance optimizations throughout the course of their life cycle. Such optimizations may inspire designers to adopt a relaxed correctness for potential performance gains. In order to accommodate the evolution of a concurrent system, Peterson et al. propose a Correctness Condition Specification tool (CCSpec) for the verification of user-specified correctness condition. CCSpec can check any correctness condition that expects a concurrent history to exhibit equivalent behavior to a legal sequential history that is not necessarily constrained by real-time ordering. CCSpec is publicly released at http://ucf-cs.github.io/CCSpec/ under a BSD open-source license.

CCSpec generates the concurrent histories for a unit test using the model checker CDSChecker [4] and verifies correctness according to the correctness condition specification. The fundamental concept behind defining a correctness condition using CCSpec's custom annotations is the specification of the circumstances in which a method *happens-before* another method in a concurrent history. The happens-before relation places a partial ordering on the methods called in a concurrent history, which is extended to form a set of all possible total orderings of the methods. The set of total orderings is used to derive the set of legal sequential histories for the concurrent history by invoking the methods from the total ordering on the sequential counterpart from the C++ Standard Template Library. The unit test is verified to meet the correctness condition if each concurrent history can be mapped to a legal sequential history.

Performance Application Programming Interface

Measuring performance has been based upon software metrics which calculate the time that an execution takes to measure throughput. Other software metrics can also be developed to get information on a per thread basis. However there is always some cycles that are consumed while measuring the software metrics and for that reason, they are not as accurate as hardware metric. Measuring hardware counters and reporting the values obtained from them is more accurate and provides greater flexibility to measure different metrics form an execution.

Performance Application Programming Interface (PAPI) provides simple and easy to use APIs that can be used to collect information from hardware counters and combine the results to represent the metrics individually after an execution is complete. The programmer can select the hardware metrics which are to be monitored and the result is presented at the end of the execution separately. The metrics can be used as long as they are supported by the hardware, and different hardware support different metrics based upon the hardware counters that are available. The metrics provide in depth analysis that are used to measure throughput and cache misses in this thesis.

LIST OF REFERENCES

[1] Herlihy, Maurice and Shavit, Nir, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[2] Herlihy, Maurice P., and Jeannette M. Wing, *Linearizability: A correctness condition for concurrent objects*. ACM Transactions on Programming Languages and Systems, (TOPLAS) 12.3 (1990): 463-492.

[3] Mucci, Philip. *The performance api papi. White Paper of the University of Tennessee*. (2001).

[4] Norris, Brian, and Brian Demsky. *CDSchecker: checking concurrent data structures written with C/C++ atomics*. ACM SIGPLAN Notices. Vol. 48. No. 10. ACM, 2013.

[5] Derrick, John, et al. *Quiescent consistency: Defining and verifying relaxed linearizability*. FM 2014: Formal Methods. Springer International Publishing, 2014. 200-214.

[6] Afek, Yehuda, Guy Korland, and Eitan Yanovsky. *Quasi-linearizability: Relaxed consistency for improved concurrency*. Principles of Distributed Systems. Springer Berlin Heidelberg, 2010. 395-410.

[7] Zhang, Deli, and Damian Dechev. *A lock-free priority queue design based on multi-dimensional linked lists*. IEEE Transactions on Parallel and Distributed Systems 27.3 (2016): 613-626.

[8] Kirsch, Christoph M., Michael Lippautz, and Hannes Payer. *Fast and scalable, lock-free k-FIFO queues*. Parallel Computing Technologies. Springer Berlin Heidelberg, 2013. 208-223.

[9] Attiya, Hagit, and Jennifer L. Welch. *Sequential consistency versus linearizability*. ACM Transactions on Computer Systems (TOCS) 12.2 (1994): 91-122.

[10] Vogels, Werner. *Eventually consistent*. Communications of the ACM 52.1 (2009): 40-44.

[11] Gilbert, Seth, and Nancy Lynch. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. ACM SIGACT News 33.2 (2002): 51-59.

[12] Brewer, Eric A. *Towards robust distributed systems*. PODC. Vol. 7. 2000.

[13] Raynal, Michel, and Andr Schiper. *From causal consistency to sequential consistency in shared memory systems*. Foundations of Software Technology and Theoretical Computer Science. Springer Berlin Heidelberg, 1995.

[14] Shavit, Nir. *Data structures in the multicore age*. Communications of the ACM 54.3 (2011): 76-84.

[15] Feldman, Steven, Pierre LaBorde, and Damian Dechev. *Tervel: A unification of descriptor-based techniques for non-blocking programming*. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on. IEEE, 2015.

[16] Dongol, Brijesh, and John Derrick. *Verifying linearisability: A comparative thesis*. ACM Computing Thesiss (CSUR) 48.2 (2015): 19.

[17] Michael, Maged M., and Michael L. Scott. *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*. Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. ACM, 1996.

[18] Laborde, Pierre, Steven Feldman, and Damian Dechev. *A Wait-Free Hash Map*. International Journal of Parallel Programming (2015): 1-28.

[19] Shavit, Nir, and Asaph Zemach. *Scalable concurrent priority queue algorithms*. Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing. ACM, 1999.

[20] Kirsch, Christoph M., Michael Lippautz, and Hannes Payer. *Fast and scalable, lock-free k-FIFO queues*. International Conference on Parallel Computing Technologies. Springer Berlin Heidelberg, 2013.