A Comparative Study of Concurrent Queueing Algorithms & Their Performance

LUCA MUSCAT

Supervisor: Prof Kevin Vella

November, 2022

Submitted in partial fulfilment of the requirements for the degree of Bachelor of Science (Honours) (Computing Science).

L-Università ta' Malta
Faculty of Information & Communication Technology

Acknowledgements

Firstly, my utmost appreciation goes to Prof Kevin Vella for his continual support, which shaped and moulded this study to what it is today.
I would also like to thank my family for their encouragement, support, and for filling the deep cavity in my coffee mug.

Abstract

Queues are among the most ubiquitous data structures in the field of Computing Science. With the advent of multiprocessor programming, concurrent queues are at the core of many concurrent and distributed algorithms.
We were able to identify few studies surveying and replicating concurrent queuing algorithms from their seminal works. This work is an experimental study, with the aim of comparing the performance of multiple concurrent queues with one another, under different levels of contention, whilst replicating the results of other researchers. The production of a benchmarking framework for concurrent queues also forms part of this study's contributions.
We successfully replicate results to support the claims that non-blocking queues are superior to blocking ones under high contention, and remain competitive under low contention. We also expose a number of unreported, yet significant biases in a number of classical works.
A limiting factor of this study is that each algorithm is benchmarked on consumer-grade hardware, fortunately, the benchmarking framework is developed to run on multiple machines, making this limitation transient in scope.

Contents

List of Figures

1 Introduction

1.1 Problem Definition

The act of writing and testing multiprocessor programs is, in jest, oft-referred to as an art [19], due to the difficulty of ensuring the code's correctness. Due to the many ways to unknowingly cause performance penalties or data corruption, intimate knowledge of the CPU's architecture and memory is required.

Within the realm of Computer Science, queues are among the data structures that are utilized the most frequently. Since the development of multiprocessor programming, concurrent queues have become an essential component of a wide variety of concurrent and distributed algorithms.

1.2 Aims and Objectives

This study develops a benchmarking framework as well as a number of concurrent queuing algorithms. In addition to the validation of results, all measurements taken are compared to one another and to their original works. As a result, our research objectives are as follows:

O1. Implement a benchmarking framework for concurrent queueing algorithms capable of gathering measurements similar to prior works;

O2. Reasonably validate the benchmarking framework through metrics and experiments;

O3. Implement a variety of concurrent queueing algorithms, with the aim of replicating results from the original works.

O4. Critically compare each concurrent queueing algorithm's performance under a variety of synthetic benchmarks.

1.3 Document Layout

Chapter 2 offers a brief overview of multiprocessor programming concepts required to understand the contributions of this study; a brief survey of the field of concurrent queueing algorithms is provided. Chapter 3 describes the concurrent queueing algorithms in question and gives a high level overview of the benchmarking framework. Chapter 4 evaluates the performance of the concurrent queueing algorithms implemented in this study, together with a commentary on when one should use specific algorithms. The final chapter closes the study by giving an overview of the results obtained, discussing related work, and offering directions to improving and extending the study and the benchmarking framework.

2 Background & Literature Review

When developing concurrent programs, programmers often find that their intuitions about sequential programs do not apply. This chapter introduces the fundamental concepts necessary to comprehend the contributions of this study. Initially, the differences between *uniprocessors* and *multiprocessors* are presented. Second, concepts of multiprocessor programming are discussed. Lastly, a definition of queues and concurrent queues is provided.

2.1 Multiprocessors

The diminishing growth of *uniprocessor* speeds compelled programmers to consider parallelism as a means of improving performance [9]. Unlike *uniprocessors*, *multiprocessors* do not ensure that instructions appear in execution order [43], making multiprocessors hard to model.

Throughout the years, multiprocessor programming has become more practical and feasible, partly thanks to multiprocessors getting cheaper, and easier to program due to abstractions such as OpenMP, MPI (Message Passing Interface), implicit parallel programming environments such as SQL, and programming language standards that provide guarantees as to how parallel code will behave, such as Java's [39] and C++11's [1] standardized memory models [32, Chapter 2.2].

Multiprocessors execute a number of *processes* on distinct hardware processors [19, Appendix B.2], with each processor constantly executing and swapping out *processes*. Scott defines a *process* as a set of *threads*, where a thread is an active computation that has the potential to share variables with other, concurrent threads [43, p.6]. When a process's *time quantum* (fixed share of CPU time) expires, it is *context-switched*. Processes may be *de-scheduled* prematurely as a result of interrupts, system calls, or the voluntary yielding of control to the CPU [44, Section 3.2.3].

For the sake of optimization, compilers and CPUs do not guarantee deterministic results when multiple threads modify the same memory location concurrently [13]. To prevent incoherence across caches, explicit use of synchronization primitives (such as read-modify-write operations or memory barriers) is required.

2.2 Synchronization

2.2.1 CPU Cache

As the disparity between CPU frequency and memory access times grew, small, yet fast SRAM chips were inserted between the CPU and main memory in order to alleviate issues caused by the differences in speed [9, 13, 32]. Large quantities of SRAM is deemed uneconomical, hence the implementation of more cache layers, with each layer becoming cheaper, larger, and slower [13, 32].

Data is transferred in units of *cache lines*, which are power-of-two fixed-size aligned blocks of memory, commonly ranging from 32 to 256 bytes [33]. Cache lines are filled with data starting from the requested memory address up to an arbitrary amount of neighbouring bytes [33].

Cache-Incoherence occurs when distinct caches contain stale data [20]. A memory address can be safely cached in different processors if no updates on it occur; cache lines will be invalidated if the contents of the shared memory address are updated (preventing stale data from being read) [20]. The *MESI* cache-coherence protocol (Modified, Exclusive, Shared, or Invalid) attaches states to memory addresses [20]. State transitions notify the processor when a cache line should be invalidated [20].

Sharing occurs when one processor reads or writes to a memory address that is cached by another [20]. In some cases, sharing is unavoidable; closely located, yet logically distinct data may share a single cache line [20].

False Sharing takes place when cache lines are invalidated due to updates on logically distinct data [20]. As the rate of invalidations increases with each write to shared-memory, performance tends to degrade heavily, harming scalability. At the cost of a higher memory footprint, false sharing may be solved by segregating logically distinct data with unused memory to prevent the sharing of cache lines [43].

2.2.2 Interconnection

CPUs are linked to cache and memory via an interconnection medium. Two kinds of interconnection architectures exist: (1) *Nonuniform memory access (NUMA)* architectures [19, Appendix B.3] are typically associated with multi-CPU and distributed systems; (2) *Uniform memory access (UMA)* architectures link processors and memory using a bus interconnect [19, Appendix B.3].

Programmers require prior knowledge on the interconnection architecture they intend to deploy their high-performance programs on, as certain algorithms are not suited for specific interconnection architectures, due to the characteristics of the memory access overhead. This study will be conducted on a *UMA* architecture.

2.2.3 Atomic Operations

An operation is said to be *atomic*[1] if it is impossible to observe the operation in an intermediate state [32]. On most architectures, atomic operations may only affect the CPU's word size worth of memory.

Read-Modify-Write (RMW) operations read and write to memory atomically [32]. Most synchronization primitives may be implemented as RMW operations [19, Section 5.6], and typically need to be implemented in hardware [19, Appendix B.8]. Herlihy proved that not all synchronization primitives are equally powerful by associating a *consensus number* (i.e. the number of processes that the *consensus problem* can be solved for) to each synchronization primitive. *Non-atomic* operations are prone to *load* and *store tearing* [32, Section 4.3.4][2].

Scott lists a number of RMW operations [43, Table 2.2]. *Compare-and-swap* (*CAS*, also known as *Compare and Exchange*) is a ternary atomic operation that requires: A source, destination, and an update value. If the source and destination are equal, the source is set to the update value, otherwise, the destination is updated with the source's value [23]. Older studies may give different definitions of CAS [43, Table 2.2; 47, Appendix A], such that line five of Algorithm 1 is excluded.

2.2.4 ABA Problem

Dechev et al. [12] define the ABA problem as a false positive execution of a CAS on a shared location. The ABA problem may occur when the source's memory address has been changed from A to B, and back to A, bearing in mind that A has been freed and immediately allocated a logically distinct value, between the point in time when the destination was loaded and the CAS was executed [12].

[1]Modern CPUs only guarantee that atomic operations are semantically atomic, meaning that they do not have to execute atomically on hardware.

[2]The Intel and ARM architectures guarantee that correctly aligned loads and stores are atomic [24, Section 8.1.1; 6, Chapter 2.2].

Algorithm 1: x86 compare-and-swap pseudocode.

```
Input: ptr<T> source, ptr<T> destination, T updateValue
Output: boolean success
if $Source = Destination$ then
    $Source \leftarrow UpdateValue$;
    return True;
end
$Destination \leftarrow Source$;
return False;
```

2.2.5 Progress Conditions

A *progress condition* is a *liveness property* that describes how, and when a number of threads are able to make *forward progress* [43, Section 3.2].

A concurrent method is said to be *blocking* if there is some reachable state in the system where a thread cannot make progress until some other thread takes action [43, Section 3.2]. Algorithms that rely upon mutual exclusion are considered to be blocking [43, Section 3.2], however, non-lock-based algorithms may also be blocking [34].

A concurrent method is said to be *non-blocking* if there is no reachable state in the system where a thread cannot make progress [43, Section 3.2]. The non-blocking property is desired due to its immunity to random delays.

The following are a subset of non-blocking progress conditions, ordered by the strictness of the property (starting from the most relaxed) and the complexity of implementation (starting from the simplest):

1. *Obstruction-Freedom*: A method is obstruction free if a thread finishes execution in a finite number of steps after executing in isolation [19, Section 3.8.3];

2. *Lock-Freedom*: A method is lock-free if from all threads, a single (undetermined) thread finishes in a finite number steps [19, Section 3.8.2];

3. *Wait-Freedom*: A method is wait-free if every method call executes in finite number of steps [19, Section 3.8.1].

Stronger guarantees do not necessarily equate to higher throughput, as the overhead and complexity of the implementation tends to be positively correlated to the strength of the guarantee.

2.2.6 Correctness Conditions

Correctness conditions describe the pre and post conditions for a concurrent object's operations [19]. Such conditions decide whether a concurrent history is legal [21]. Correctness conditions are based on two requirements: (1) When an operation takes effect, and (2) how the order of non-concurrent operations should be preserved [21].

A method is said to *take effect* when the effects of its method call is seen by other method calls [19, Section 3.4.1].

Concurrent systems may be modelled as a *history*, which is a sequence of events, where each event is either an *invocation* or a *response* [19, Section 3.6.1].

A history is said to be *legal* with respect to a correctness property if it is *equivalent* to a history that respects said correctness property.

Lamport defines the *sequential consistency* correctness condition as a history in which the result of an execution is the same as if the operations had been executed in *program order* [31].

A *linearizable* concurrent computation gives the illusion that a method call takes effect instantaneously some time between the method's invocation and response; the point in time when the method takes effect is also known as the *linearizability point* [19, 21]. Processors perceive linearizable operations in a *total, and real-time order*, where overlapping method calls take effect in a non-deterministic order [19, Section 3.6.2].

2.2.7 Memory Consistency Models

Memory Consistency Models (referred to as *memory models* henceforth) are correctness properties that describe how threads perceive the order of other threads' effects on shared memory [19, Section 3.7]. Similarly to the correctness conditions described in section 2.2.6, memory models vary in strength; as the model used becomes more relaxed, more aggressive optimizations may be used, increasing performance at the cost of programmability and portability [15].

Relaxed memory models such as *weakly ordered models*[3] offer little to no constraints with respect to the ordering of reads and writes, requiring the explicit use of memory barriers for a finer grained control over the ordering [15]. Gharachorloo identifies relaxed memory models as *system-centric models*, since they tend to favour performance, as opposed to a programmer's intuition about shared-memory (sequential consistency) [15].

[3]There exists ARM processors that tend to be weakly ordered when interacting with main memory [6, Section A3.5.5]

Memory models—such as sequentially consistent models—are considered to be *strongly ordered*. Strongly ordered systems tend to heavily restrict the reordering of instructions, giving a simpler and higher-level interface between memory and the programmer.

2.2.8 Mutual Exclusion

Mutual exclusion (also known as a *mutex*, or a *lock*) is a synchronization protocol that *serializes* the concurrent execution of code inside *critical sections*. A *spin-lock* is a variant of a mutex, that *busy-waiting*, and can be released by any thread.

Every mutex has to decide what actions to take when a critical section has already been acquired; the three categories said action can fall under are:

- *Blocking*: Yield the remaining CPU time if the mutex is already acquired. Blocking is not suitable for small critical sections[4], as a thread typically has to wait tens of milliseconds to transition from a *ready state* to a *running state*.
- *Busy-Waiting (spinning)*: Polls an area in memory containing the state of a mutex. *Busy-waiting* is best suited for small critical sections, as contention caused by polling may lead to degraded performance;
- *Hybrid*: Utilizes a mix of *blocking* and *busy-waiting*, forgoing the inefficiencies associated with both waiting strategies.

Algorithms reliant on mutual exclusion are inherently blocking [19, Section 3.8]; *starvation freedom* and *deadlock freedom* are some examples of correctness conditions unique to *blocking* algorithms.

2.3 Queues

Knuth defines a queue as a linear list for which all insertions occur at one end of the list and deletions occur at the other [27]. A queue comes with two operations, enqueue (places an item at the head of the queue) and dequeue (removes and returns an item at the tail of the queue), following First in First out ordering (FIFO).

2.3.1 Concurrent Queues

Concurrent queues are types queues that remain consistent and correct when accessed simultaneously through a number of threads. Non-concurrent—or ill-synchronized

[4]A critical small critical section takes hundreds of nanoseconds or microseconds to execute.

queues—typically become inconsistent after being accessed through multiple threads, leading to hard-to-find bugs [48]. Concurrent queues are often the basis of scheduling algorithms [11] and many other concurrent algorithms [48]. Linearizability is the de-facto correctness condition for non-blocking queues, as the semantics of the queue's operations remain unchanged [34, 47].

The multiplicity of producers and consumers that can simultaneously interact with a concurrent queue is used as a taxonomy to categorize queues; furthermore, the underlying data structure used to implement a queue (such as a circular buffer, linked list, or both) affects if the queue is of *fixed capacity (bounded)* or not *(unbounded)*.

This study focuses on *unbounded, MPMC, blocking and non-blocking queues*, with chapter 2.4 covering the evaluated queues in further detail.

Figure 2.1: Possible configurations in the Producer-Consumer taxonomy

Label	Description	Reference
SPSC	Single-Producer/Single-Consumer	[4]
SPMC	Single-Producer/Multi-Consumer	[7]
MPSC	Multi-Producer/Single-Consumer	
MPMC	Multi-Producer/Multi-Consumers	[22, 37, 46]

2.4 Literature Review

2.4.1 Valois' Queue

Valois surveys several lock-free data structures and techniques together with the introduction of the *safe read* memory-reclamation scheme and an *MPMC lock-free queue* [46, 47]. Combining the memory-reclamation scheme with the queue leads to increased cache line churn, as enqueues traverse the queue's linked list for a bounded number of nodes, causing each node's reference counter to be modified several times.

2.4.2 Michael's and Scott's Queues

MS-Queue Similar to Valois' queue, the MS-queue's head and tail are separated, with the head always pointing to a dummy node. CAS retry loops, which are a common pattern in lock-free algorithms, tend to cause starvation and reduced parallelism. Michael and Scott adopt a thread-helping technique, where a thread that fails to commit a node to the linked list, may help other threads by doing part of their work (acting as a secondary back-off, reducing contention). The authors omit any discussion on the MS-queue's memory reclamation scheme, which as pointed out by Michael in [36] may lead to race conditions.

Two-Lock Queue Similar to the MS-queue, the head and tail are separated, allowing for concurrency between enqueues and dequeues [37]. The ABA problem does not exist in this algorithm, as the CAS operation is not used; furthermore, complex memory reclamation schemes are not needed, as access to nodes are mutually exclusive, ensuring that a node may never be freed when referenced by another thread.

2.4.3 Optimistic MS-Queue

Ladan-Mozes and Shavit improve upon the MS-queue by enabling enqueues to take effect in a single CAS [30]. With enqueues adding nodes to the beginning of the list, doubly-linked lists enable the deletion of a node through backwards traversals of the linked list. Pointers to previous elements are maintained using simple stores, and are fixed upon entering an inconsistent state (hence the *optimistic* replacement of CAS operations).

2.4.4 Hoffman et.al's Baskets Queue

Hoffman et al. present a variation of the *MS-queue* [22], which is formed using baskets (groups) of overlapping linearizable operations that are non-deterministically ordered among one another. Time spent backing off in CAS failing threads is spent inserting nodes in a basket, increasing parallelism across enqueuers. The baskets mechanism also doubles down as a secondary back-off, further reducing contention.

Tagged pointers are used for ABA avoidance; dequeued nodes are logically deleted by setting a flag bit packed inside the node's ABA counter. As the number of logically deleted nodes grows greater than the number of *max hops* (an arbitrarily chosen constant), or a logically deleted node points to the tail of the queue, the *free-chain* method is used to swing the queue's head to the next non-deleted node, and reclaims any logically deleted node between the head and the tail. Under high levels of concurrency, the authors boast of a 25% performance improvement when compared to the MS-queue.

2.4.5 Kogan-Petrank Wait-Free Queue

Wait-free queues offer benefits such as *starvation freedom* and predictable operation latencies at the cost of performance, practicality, and complexity. Based on the *MS-queue*, Kogan and Petrank introduce one of the first practical unbounded, MPMC wait-free queues [28] (henceforth known as the KP-queue).

2.4.6 Mechanically Creating Wait-Free Queues

Kogan and Petrank present a methodology to create fast wait-free queues [29] by making use of the *fast-path-slow-path* methodology; a lock-free queue is used as the *fast-path*, using a wait-free queue as a *slow-path* after a number of retries.

2.4.7 Wait-Free Queue as Fast as Fetch-And-Add

In [49], Yang and Mellor-Crummey presents a wait-free queue that is said to be as fast as fetch-and-add. Notably, the benchmarks and comparisons among queues are statistically rigorous.

3 Design & Implementation

This chapter marks out the design and implementation of the benchmarking framework (research objective **O1**). A discussion on the steps taken to validate the present study's results (research objective **O2**), together with efforts to contain the sources of error are discussed. The queues implemented in this study (as per research objective **O3**) are discussed[1].

3.1 Relevant Software Versions

Large runtime environments, such as Java's JVM, are regularly used in concurrent programming, as the responsibility of memory management falls on the garbage collector [28]. Fog notes that large runtimes are typically more resource-consuming than the executed algorithms [14]. Arguably, implementations of non-blocking algorithms in lock-based, garbage-collected environments are not truly non-blocking. The present study is implemented using the C programming language, since it uses a relatively light runtime without garbage collection. The clang *version 10.0.0-4ubuntu1* compiler, with *-O3 -march=native* optimisation flags are used for compilation. An *Intel Core i7 6700HQ (Skylake)*, with four cores and a base frequency of 2.60GHz, together with 16GB of RAM (8x2) is used throughout the rest of this study.

PAPI (Performance Application Programming Interface) is used to measure elapsed time in nanoseconds [45]. Although an entire library for measuring time may seem excessive, PAPI also provides an interface for sampling hardware counters, which can be used for debugging purposes.

[1]The artefacts presented in this study can be found in the following git repository: `https://github.com/lucamuscat/dissertation`

3.2 Methodology

Researchers introducing novel concurrent algorithms [22, 28, 37, 46, 49] frequently compare their contributions with existing algorithms. The *Pairwise* and *50% Enqueue* benchmarks are considered to be standard benchmarking procedures for concurrent queueing algorithms. Ten million operations[2] are executed amongst N threads ($t_0,\ t_1,\ \cdots,\ t_{N-1}$) as follows:

$$\text{iterations(i)} = \begin{cases} \lfloor \frac{10^7}{N} \rfloor, & \text{if } i < N-1 \\ 10^7 - (\lfloor \frac{10^7}{N} \rfloor \cdot (N-1)), & \text{otherwise} \end{cases}$$

The average of each thread's average elapsed time for ten repetitions, excluding time spent in artificial delays is used as the recorded sample.

3.2.1 Pairwise Benchmark

On an initially empty queue, each thread enqueues an item, executes an artificial delay (which Michael and Scott refer to as "other work" [37]), dequeues an item, and executes another artificial delay. The queue is destroyed and recreated with every repetition (i.e.every ten millions iterations). Michael and Scott use an artificial delay to reduce the bias introduced by long runs of queue operations by a single thread [37]. Quantity of threads, and the length of the artificial delay are the experiment's independent variables. The degree of contention is dependent on both the number of threads and the length of delay; furthermore, the time measured is also dependent on contention and operating system activities. This benchmark serves the purpose of simulating a producer-consumer workload, where each thread acts both as a producer and a consumer.

3.2.2 50% Enqueue Benchmark

Drawing from a uniform distribution, each thread is equally probable to execute an enqueue or a dequeue, followed by an artificial delay. The queue is prefilled with 1000 items to prevent long runs of empty dequeues (potentially introducing biases). Randomness aids with exercising a larger breadth of code paths.

Random numbers are stored as double-precision floating point variables, ranging from zero to one; furthermore, they are generated in the benchmark's setup phase as to

[2]Ten million operations reach a suitable equilibrium between test runtime, memory consumption, and amortized overheads.

calculate the number of enqueues (consequently the amount of memory that needs to be allocated), and to reuse the same random numbers with every repetition, maintaining consistency across measurements.

The *50% enqueue benchmark* executes at most half the number of enqueues and dequeues as the *pairwise benchmark*, as an iteration no longer consists of a pair of enqueues and dequeues, but either one of them.

3.2.3 Sources of Error

Researchers must be aware of any threats to the validity of their results. An *observational error* is the difference between what is measured—and the true result—which can be caused by *systemic* or *random errors* [3]. *Systemic errors* are caused by the inherent inaccuracies of measurement apparatus. With respect to the framework's measurements, a source of system errors is the resolution (granularity) of the clock used to measure elapsed time [3]. The CPU's clock speed is fixed at its maximum frequency by setting the CPU frequency governor to *performance* mode (using the '*cpufreq-set -g performance*' command) in order to restrict fluctuations in the CPU's clock speed, which may lead to inaccurate measurements and delays at the cost of higher energy consumption. Random errors cause inconsistencies in repeated measurements. Variance is commonly attributed to operating system activities, such as process migration across cores, and context-switching.

Hyperthreading is disabled as it leads to higher rates of cache misses due to each core's cache being split with an extra thread [14]. Turbo boost is disabled, since it dynamically controls processor performance according to the CPU's thermal status [24, Section 14.3.3]. Threads are pinned to specific cores using thread affinity to reduce *process migration*.

The *observer effect* describes the phenomenon where the behaviour of an observed experiment deviates from its true form when not under observation. Concurrent algorithms are highly timing-sensitive[3], as measurement overheads may influence the interleaving of threads. By measuring a significantly large number of operations, the mean time is derived and used as a sample, amortizing measurement costs. Aceto et al. note that data variability affects the repeatability of experiments [2]; the ratio of standard deviation to mean (*coefficient of variation*), $\mathrm{CV} = \frac{\sigma}{\bar{x}} \cdot 100$ is used to quantify the repeatability of each experiment.

[3]As a testament to the timing-sensitive nature of concurrent algorithms, the term *heisenbug* has been coined for bugs that tend to disappear when debugging overheads are introduced [32].

3.3 Design Decisions

As per research objective **O3**, the following concurrent queueing algorithms are implemented: Michael and Scott's Lock-Free Queue [37]; Michael and Scott's Two-Lock Concurrent Queue [37] (protected by a *test-and-test-and-set-lock* [35]); Hoffman et al.'s Baskets Queue [22]; and Valois's Lock-Free Queue [46]. Keen-eyed readers may observe that the chosen queues are *unbounded* and *MPMC*.

3.3.1 ABA Avoidance

Version tagging is used for ABA avoidance (not to be confused with ABA-Freedom) [12]: A *counter* (also known as a *tag*) is attached to a pointer, which increments with each successful CAS. Comparisons between pointers that have the same address, but do not match in tag value are expected to fail, as differing tag values hint at the possibility of a change in logical data. Counters may overflow, leading to tags wrapping back to their initial value, allowing for ABA violations to still occur (the chances of which are virtually nil)[4].

For systems with a significant number of processors (for instance multi-CPU NUMA architectures), larger counters are required to safely avoid the ABA problem. The *Double-Width Compare-and-Swap* instruction (*DWCAS*, *CMPXCHG16b* on *Intel x86_64*) may be used to attach a 64 bit counter to a pointer. Intel systems require that the pointer and counter are aligned to a 16 byte boundary.

```c
struct tagged_ptr{
    void* ptr;
    uint64_t tag;
} __attribute__((aligned(16))) tagged_ptr;
```

Listing 3.1: Struct aligned to 16 bytes, as required by the DWCAS instruction.

Alternatively, systems that do not provide the DWCAS instruction, suffer from an unacceptable degradation in performance due to DWCAS, or do not allow for the modification of pointers may employ a single-word CAS, by attaching a counter to the pointer itself (known as *pointer packing*, or *tagged pointers*) through the use of masking and shifting. Intel x86_64 architectures support 64 bit pointers, however, the architecture only makes use of the least significant 48 bits (known as the *linear address*) [24, Section 3.3.7.1], allowing for the most significant 16 bits to be used as a counter.

[4] Our implementation of *version tagging* can be found in the following resource: `https://github.com/lucamuscat/dissertation/blob/5e493261cb152c870551a360a5a859619bdfb773/src/tagged_ptr.h`

Pointer packing is flawed: The range of values a 16 bit counter can hold ranges from zero to $2^{17} - 1$ (which may easily overflow). Pointer packing is not portable, as there is no guarantee that the endianess, or the linear address size will remain the same. Pointers may only be dereferenced in their canonical form, meaning that the most significant 16 bits either need to be set to zero (user-space pointer) or one (kernel-space pointer), as canonical faults are thrown when dereferencing a non-canonical pointer.

3.3.2 The Essence of the Implemented Queues

3.3.2.1 Valois's Queue

Valois presents a lock-free, unbounded, MPMC queue, together with three different strategies for retrying failed CAS operations [46]. The first strategy strictly requires the tail to point to the last node in the linked list. The guarantee of pointing to the last node of the linked list comes at the cost of an extra CAS. The second strategy loosens this guarantee by only requiring the tail to provide a *hint* of where the last node of the linked list is. Lastly, a hybrid approach of the first and second approaches can be used. This study implements *Valois's lock-free queue* using the second retry strategy[5].

Segregated Head and Tail Irrespective of the retry strategy used, the head (which points to the last dequeued node, and initially points to a dummy node) and tail of the queue are separated, allowing for higher degrees of parallelism.

Traversal of the Linked List As the tail solely provides a hint to where the last item is in the linked list, the enqueueing thread traverses at most $2p - 1$ (p being the number of concurrent threads) nodes to reach the end of the linked list. Michael and Scott notes the that the tail may lag behind the head, possibly breaking the linked list when reclaiming a node [37].

3.3.2.2 MS Queue

The *MS Queue* [37][6] harnesses the non-deterministic outcomes of CAS-contended memory locations, employing thread-helping mechanisms in CAS-failing threads. In order to enqueue a node, a thread must commit the node into the queue by setting the true tail's (i.e., the last node in the linked list) next pointer to the enqueued node; next, the tail must be updated to point to the last node in the linked list. Consequently, two successful CASs are required to commit a node; Ladan-Mozes and Shavit reduces the number of required CASs to enqueue a node to a single CAS by employing a doubly-linked list [30]. As CASs may fail to update the tail, the tail may point to the true tail's preceding node. In the event an enqueueing thread takes a snapshot of the node preceding the true tail, the thread may update the tail to point to the true tail (this is referred to as *swinging*

[5]Our implementation of *Valois's lock-free queue* can be found in the following resource: `https://github.com/lucamuscat/dissertation/blob/master/src/queues/non-blocking/valois_queue.c`

[6]Our implementation of the *MS queue* can be found in the following resource: `https://github.com/lucamuscat/dissertation/blob/master/src/queues/non-blocking/ms_queue.c`

the tail). Failure to swing the tail implies that another thread has already swung the tail. A thread will endlessly retry adding a node to the end of the linked list (and consequently, remain enqueuing the node) until it succeeds to do so. Kogan and Petrank capitalize on the wasted parallelism of multiply-failing CAS threads by employing a wait-free enqueue (the slow path) after a number of failures in the lock-free enqueue (the fast path). Unfortunately, the degree of parallelism offered by these thread-helping mechanisms are restrained to simultaneously enqueuing threads, as a thread may only affect the queue if its snapshot of the tail is consistent (i.e., the snapshot of the tail is the true tail).

Similar to an enqueue, a dequeue may only take effect if the head is consistent. Similar to *Valois's queue*, the head points to the most recently dequeued node. If the snapshot of the head is consistent, and is equal to the snapshot of the tail, a queue is said to be empty if the head's next pointer is null, otherwise, the thread will attempt to swing the tail to the only node in the linked list. The dequeue method may only provide a response when it successfully manages to move the head to the next node.

3.3.2.3 Baskets Queue

The *Baskets Queue* [22][7] exploits the fact that two or more operations overlapping in time have non-deterministic ordering. An enqueueing thread which fails to commit its node into the linked list (due to a conflicting CAS) spends time building a *basket*, which is a group of non-deterministically ordered nodes. Building *baskets* doubles down as a secondary backoff. In case the tail does not point to the last node in the linked list, the enqueueing thread searches for the last node and swings the tail. The queue's head only serves as a hint of the last dequeued node, which is reason nodes are logically deleted before they are reclaimed. A node is said to be logically deleted if the flag bit inside the ABA counter is set to one. The dequeueing thread performs the *free chain* procedure if the newly deleted node is far away enough from the head; if the tail is encountered while searching for a non-deleted node, the head is updated, and the nodes between the old and the new head are safe to reclaim.

3.3.3 Nanosecond Accurate Delay

Delays of arbitrary lengths can be placed between operations, in order to control contention [46]. High-resolution timers are accurate up to a number of microseconds (10^{-6}

[7]Our implementation of the *Baskets Queue* can be found in the following resource: `https://github.com/lucamuscat/dissertation/blob/master/src/queues/non-blocking/baskets_queue.c`

seconds), making it unsuitable for nanosecond accurate measurements (10^{-9} seconds). With respect to CPUs, time may be described in different dimensions, where the number of CPU cycles can be extrapolated from wall-clock time and the CPU's clock speed, allowing for an instruction—with a known latency—to be executed as many times as required to consume an arbitrary amount of time. Unfortunately, extrapolating CPU cycles from wall-clock time requires a fixed CPU clock speed, hurting the validity of results in cases where CPU clock speeds are highly unstable.

Similar to Intel's *Architecture Agnostic Spin-Wait Loops* [26] and *pollDelay* function [25, Example 2.3], a nanosecond-accurate delay is implemented by constantly polling the CPU's timestamp counter until the extrapolated number of cycles are consumed [41].

Algorithm 2: Delay implemented through polling the CPU's timestamp counter.

```
Input: uint64_t time_ns
/* Pre-calculate to prevent inaccuracies. */
1 $cycles \leftarrow time\_ns \cdot cpu\_frequency\_ghz$
2 $start \leftarrow read\_timestamp()$
3 repeat
4     $stop \leftarrow read\_timestamp()$
5 until $stop - start < cycles$;
```

The delay's accuracy is significantly improved through calibration, where the delay function is put through a trial run, and the number of cycles to be executed are modified according to the error between the delay's average time elapsed, and the expected delay.

3.3.4 Test-and-Test-and-Set Lock with Exponential Delay

Rudolph and Segall propose an alternative to the *test-and-set (TAS)* spinning method called *test-and-test-and-set (TTAS)* [42]. The steps *TTAS* takes to reduce cache line invalidations caused by indiscriminate *TAS* instructions are two-fold: (1) Read the polled flag until a 'falsey' value has been read (lock has been released); (2) Acquire the lock using the *TAS* instruction, going back to the first step if the *TAS* operation returns a 'truthy' value (another thread has acquired the lock between steps one and two).

Exponential back-off is needed to control contention [46]. Anderson suggests a bound on the delay, as uncontended processors that back off a number of times will take longer to acquire a lock; furthermore, unfairness exacerbated under exponential delay [19, Section 7.4]. Intel's *Contended Locks with Increasing Back-off Example* was adapted in the lock's implementation [25].

```c
typedef struct ttas_lock_t
{
    // Do not share the cache line
    __attribute__((aligned(64))) atomic_bool busy;
} ttas_lock_t;
```

Listing 3.2: Test-and-Test-and-Set Lock Struct.

Algorithm 3: Pseudocode for Acquiring a Test-and-Test-and-Set Lock.

```
   /* Acquire */
1  mask ← 1 max_backoff ← 64 repeat
2      while atomic_load_explicit(lock.busy, memory_order_acquire) do
3          i ← mask for i → 0 do
4              PAUSE /* Reduces contention from spin-waiting */
5          end
6          if mask < max_backoff then
7              mask ← mask · 2
8          else
9              mask ← max_backoff
10         end
11     end
12 until atomic_exchange_explicit(lock.busy, 1,memory_order_acquire);
   /* Start Critical Section */
   /* Critical Section */
   /* End Critical Section */
13 atomic_store_explicit(lock.busy,0,memory_order_release)/* Release */
```

3.3.5 Memory Management

Due to time constraints, dynamic memory management of non-blocking data-structures is omitted, as it is an actively researched field which can single-handedly fit an entire dissertation [36, 47].

Memory is allocated before each benchmark, as to avoid degrading non-blocking progress conditions through the use of lock-based memory allocation. Pre-allocating memory prevents the ABA problem (as memory addresses may no longer be reused) [12], removes the overhead of memory allocation in measurements, and forgoes the need of memory management schemes, as memory is released at the end of the benchmark.

3.4 Benchmarking Framework Designs

3.4.1 Modularity

New queues and locks may be added to the framework with ease by implementing interfaces B.1 and B.2. Locks in the *src/locks* directory are statically linked with every blocking queue inside *src/queues/blocking*; each blocking queue variation and non-blocking queue (found in *src/queues/non-blocking*) are statically linked to every benchmark, generating a single executable binary for each queue's benchmark.

3.4.2 Lifecycle

Each benchmark goes through a common lifecycle, which can be categorized into four phases: Firstly, the master thread initializes synchronization primitives (such as locks and barriers), auxiliary data (such as sequences of random numbers for the *50% Enqueue benchmark*), and the queue to be benchmarked; Secondly, *N* threads are spawned, and are provided with the necessary arguments; Thirdly, a contiguous pool of memory proportional to the number of enqueue operations executed in the warm-up and benchmark are allocated[8]; Finally, each thread executes its share of operations and waits until each thread has finished, in order to safely free up the allocated memory. The entire cycle is repeated using a newly initialized queue.

3.4.3 Auxiliary Data

Upon the initialization of a queue, up to 64 bytes of memory—aligned as four byte integers—are passed to the queue, and are outputted the benchmark data. Throughout the study, these 64 bytes store several fine-grained statistics, such the number of failed Compare-And-Swaps, or the number of times a thread helping mechanism has been invoked.

3.4.4 Performance Considerations

Static linking is utilized as it provides performance benefits over dynamic linking; Fog notes that functions inside dynamically linked libraries take longer to execute, and make less efficient use of code and data caching [14, Section 14.11].

The alignment and padding variables reduces false sharing, preventing logically distinct data from being stored in the same cache line. Thread-specific data was stored in thread-local storage, as to further reduce false sharing. Struct fields were aligned to

[8]The number of enqueues are extrapolated from the pre-generated sequence of random numbers in the case a 50% enqueue benchmark is used.

128 bytes, due to Intel's spatial prefetcher loading data into L2 cache in chunks of 128 bytes [25, Section E.2.5.4].

Threads are synchronized to start at the same time using barriers, mitigating transient start-up effects [22]. Furthermore, a number of operations are executed before the start of each benchmark, as to warm up the cache and branch predictor, moving cache misses due to unpopulated caches and speculative branch mispredictions outside the recorded benchmarks.

3.4.5 Validation

In response to research objective **O2**, parts of the framework's components are validated, as to increase confidence in the measurements obtained. A constant value is enqueued, allowing for the comparison between the original and dequeued values to test for any corruption caused by race conditions.

The artificial delay's accuracy is validated through a separate benchmark where the average delay over ten million operations is recorded several times. A threshold for the coefficient of variance of one percent is used to validate suitability of the delay; such a test allows for immediate feedback loops, where changes to the delay's behaviour may be verified in a timely manner.

The benchmarking framework's auxiliary data stores the number of times an algorithm visits any specific line of code (visited lines are counted only if the user instruments the benchmark to do so). Counters act as crude indicators that an algorithm may not be functioning correctly; for instance, if a counter indicating a dequeue on an empty queue has occurred for the majority of the benchmark's iterations, one can deduce that the queue's linked list is corrupted.

A *Test Driven Development (TDD)* approach was used when implementing the pointer packing logic, which served the purpose of reducing the chances of regressions due to code changes or differences in CPU architecture. Atomic variables are asserted to be lock-free through the *atomic_is_lock_free* method, which may trigger due to compiler settings, or unavailability of specific atomic instructions on CPUs. Error checking is used to prevent the framework from reaching undefined states.

4 Evaluation

This chapter tackles research objective **O4** by congealing the benchmarking framework's results into a number of insights, which are used to compare the performance of the presently implemented queues and their seminal works. Each section is dedicated to the performance of each queue under a workload of a fixed number of threads, with varying delay between queue operations. Each queue's performance is quantified using the net runtime of each benchmark, together with the magnitude of performance degradation, which is calculated as the ratio of net runtime between the current and previous thread (where a value greater than one indicates degraded performance).

4.1 Workload Under One Thread

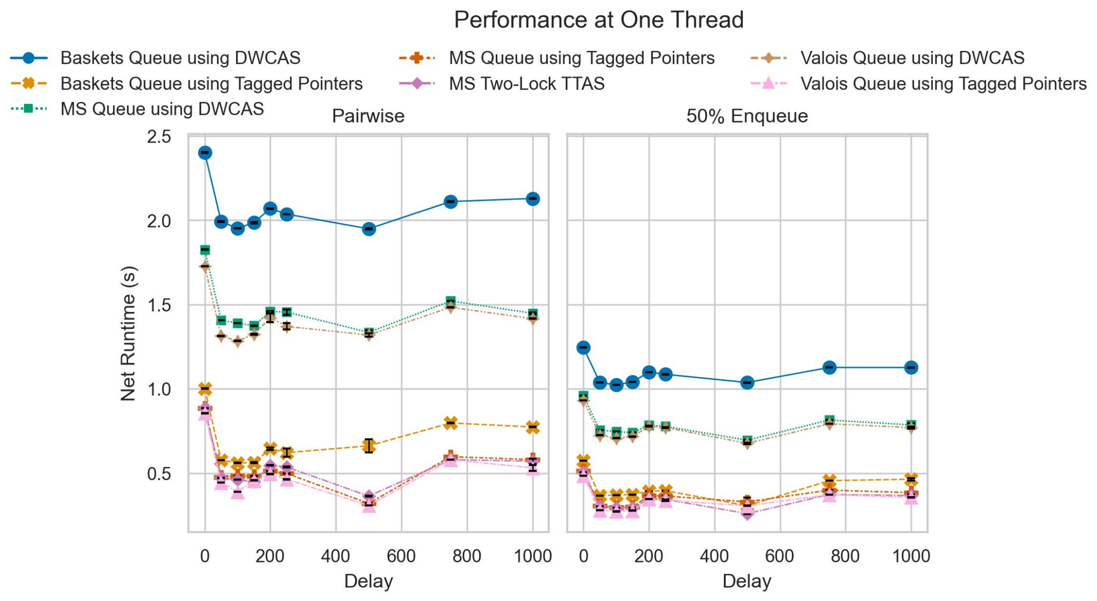

Figure 4.1: The pairwise and the 50% benchmarks on the left and right respectively.

The sequential latency of a queue is determined by its algorithmic complexity [47]. Contention-reducing mechanisms—such as thread helping—are not triggered under single threaded workloads, as they rely on failed CAS operations, adding extra overhead through the computation of predicates. Figure 4.1 shows the performance of each queue in a single threaded workload. Under the *pairwise benchmark*, the *MS-Queue* and the *Baskets Queue* using tagged pointers are at most **3.168** and **2.593** times faster than their DWCAS counterparts. The *Baskets Queue* is consistently outperformed by the *MS-Queue* and the *Two-Lock Queue* (respectively, at most **0.450** and **0.516** times slower). Although the *pairwise benchmark* does not show which queue is consistently faster, the *50% enqueue benchmark* shows that the *Two-Lock Queue* consistently outperforms the *MS-Queue* (by at most **0.27** times).

4.2 Workload Under Two Threads

Similar to [22, 30, 37], under a workload of two threads, a significant degradation in performance can be observed. Michael and Scott notes that as each queue's head and tail are shared across two processors, cache misses are more frequent. Figure 4.2 shows that queues using tagged pointers tend to experience higher contention, consequently leading to worse performance. In support of this claim, the *Baskets Queue using Tagged Pointers* with 50 nanoseconds of delay was **2.435** times more likely to re-attempt an enqueue than its DWCAS counterpart. For the remainder of this chapter, any discussions relating to the performance of queues using DWCAS are omitted, due to their inferior performance.

In the *pairwise benchmark*, the *Baskets Queue* is consistently outperformed by the *MS-Queue* and the *Two-Lock Queue* by at most **20.978%** and **34.672%**, with the *Two-Lock Queue* consistently outperforming the *MS-Queue* by at most **11.100%**.

At delays greater than 200 nanoseconds, the *two-lock queue* exhibits the best performance. Although outperformed, non-blocking queues using tagged pointers remain competitive under low-contention.

The performance of Valois's queue in this study highly conflicts with that of [37]. Although a queue without a memory reclamation scheme may share a common algorithm with one that has, it does not imply that they are algorithmically equivalent, making their performance unconnected; as this study does not include memory reclamation schemes in its implementations, valid comparisons cannot be made with [37]'s results of Valois's queue, however, insights may still be inferred. Michael and Scott omit a discussion on how their choice of memory-reclamation schemes introduces significant

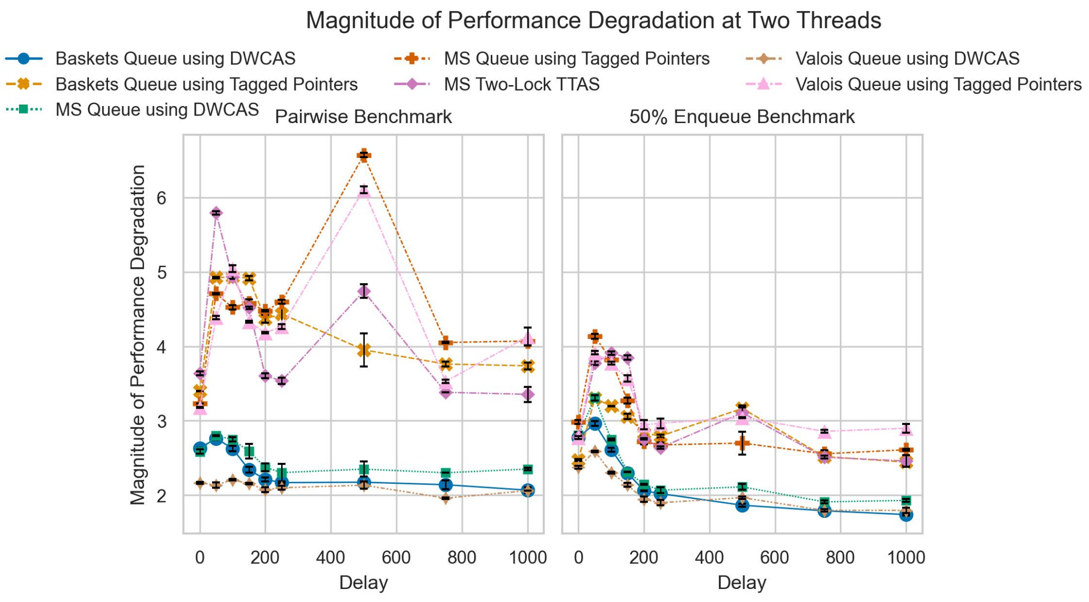

Figure 4.2: The magnitude of performance degradation between single and two-threaded workloads.

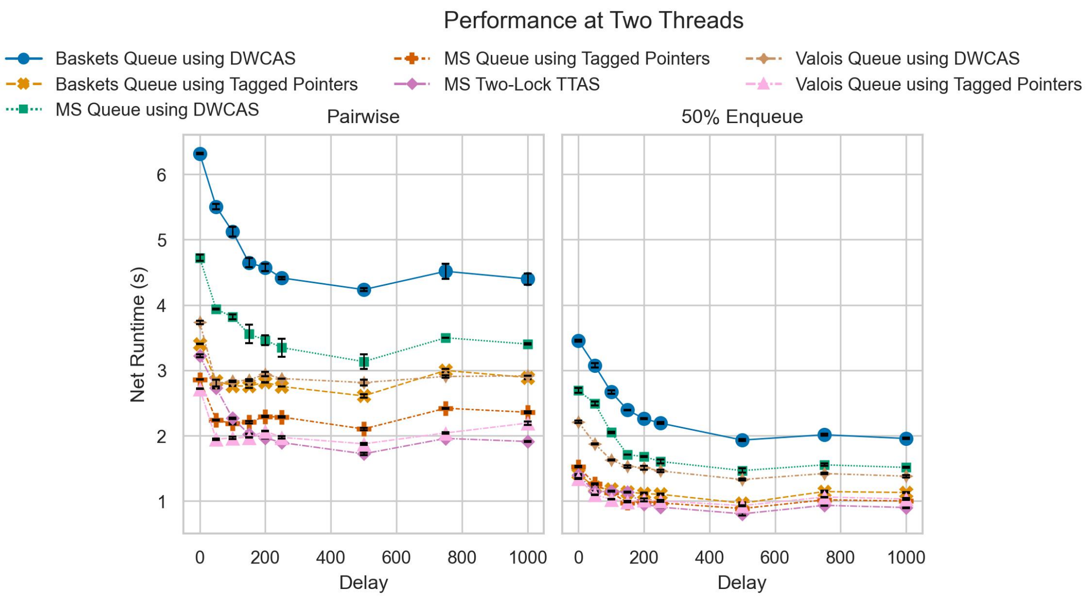

Figure 4.3: The pairwise and the 50% benchmarks on the left and right respectively.

biases in their favour. This is only made clear when inspecting the study's published source code [38], where Valois's queue makes use of the safe-read protocol, whilst the *MS-Queue* only keeps track of each threads' most recently recycled node non-atomically in thread-local storage (see Appendix A.1).

One may hypothesize that Valois's queue fitted with the *safe read* protocol would lead to a horribly inefficient algorithm, as each enqueueing thread is required to traverse a number of nodes in the linked list, incurring a high number of cache misses due to the reference counting system incurring additional loads and stores on visiting and moving on from a node. Consequently, any discussions related to *Valois' Queue* are omitted.

Figure 4.5 shows that at specific delays, speedups of up to **6.591%** can be obtained. Michael and Scott observe speedups of a factor less than $\frac{1}{3}$ [37], which is far more drastic than that observed in [22, 30] and this study. The 50% enqueue benchmark does not exhibit any speedups, further highlighting the effects of a benchmark's artificiality on the validity of its results.

4.3 Workload Under Three Threads

A minor speedup under a workload of three threads is commonly observed [22, 30, 37]. Michael and Scott link the boost in performance to fewer iterations per thread, in combination with a cache miss rate similar to that under two threads.

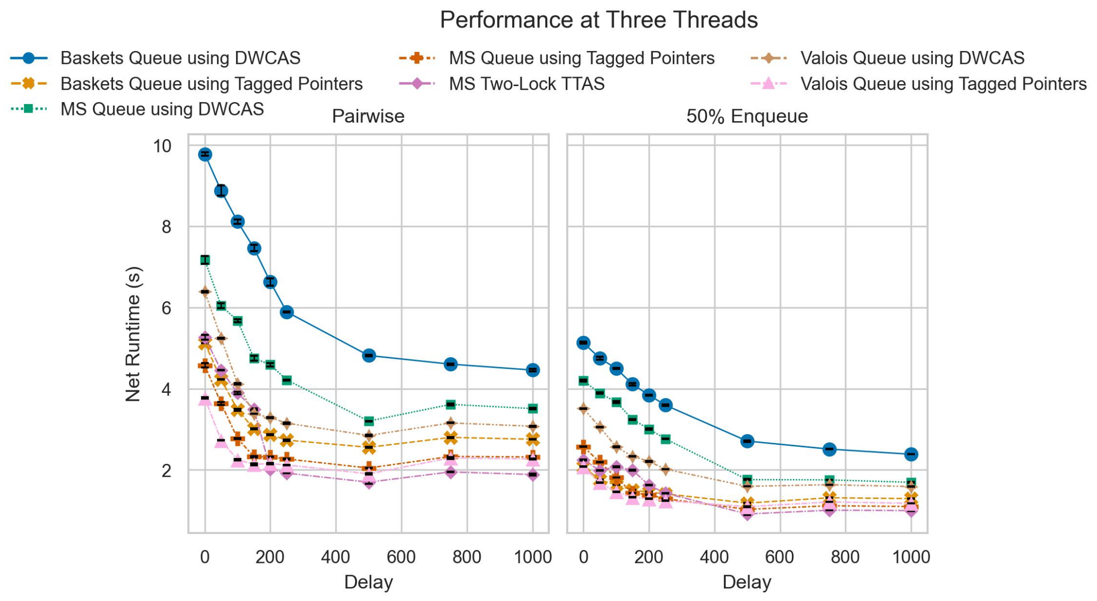

Figure 4.4: The pairwise and the 50% benchmarks on the left and right respectively.

Up to delays of 150 nanoseconds, every non-blocking queue using tagged pointers significantly outperforms the two-lock queue (*MS-queue* is **33.412%** faster than the two-lock queue at 150 nanoseconds of delay), however, as contention drops, the two-lock queue tends to become favourable. The *MS-queue* consistently outperforms the *Baskets Queue* by at least **10.778%**. As the degree of contention under three threads is higher than that at two threads, the performance penalty in using higher delays is reduced.

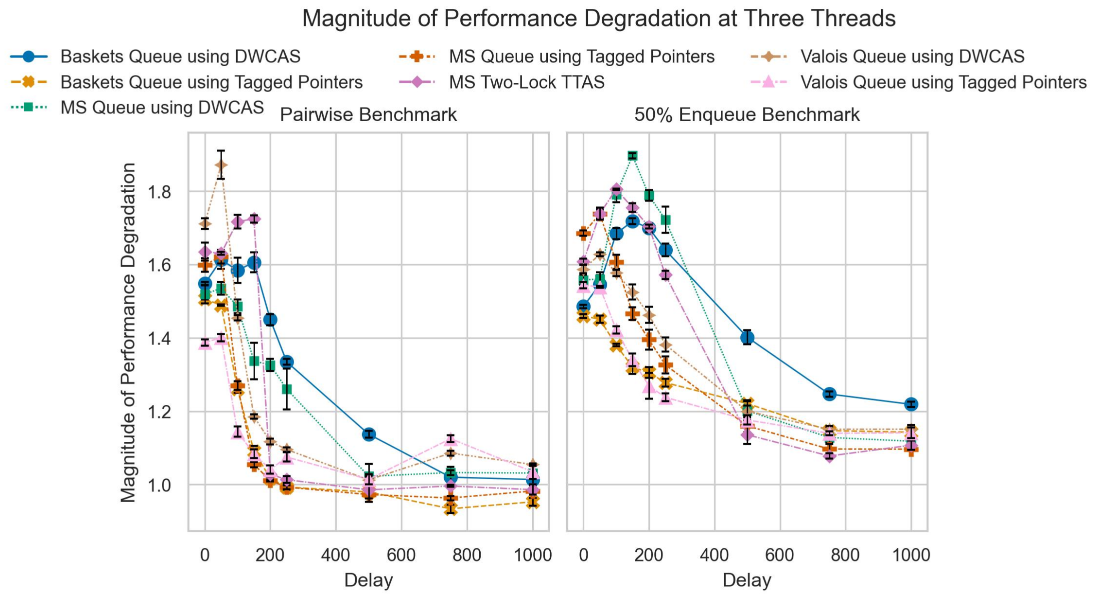

Figure 4.5: The pairwise and the 50% benchmarks on the left and right respectively.

4.4 Workload Under Four Threads

At four threads, [22, 30, 37] observe a slight dip in performance for both the *MS-queue* and the *Two-Lock Queue*. Hoffman et al. notes an increase in performance for the *Baskets Queue*. As each queue at every delay exhibits a magnitude greater than one, it can be concluded that performance has worsened.

Figure 4.6 shows that under the pairwise benchmark and no delay, the *Baskets Queue* is **0.259%** slower than the *MS-Queue*; At delays of 50 and 100 nanoseconds, the *Baskets Queue* is respectively **0.765%** and **3.668%** faster than the *MS-Queue*; Above delays of 100 nanoseconds, the *Baskets Queue* is up to **26.074%** slower than the *MS Queue*. On the other hand, the *50% Enqueue Benchmark* shows that the *Baskets Queue* is up to **42.047%** faster than the *MS-Queue*. The distinct characteristics of the *50% Enqueue Benchmark* allowed for the *Baskets Queue* to make use of the baskets thread-helping mechanism up to **110.964** times more than in the *Pairwise Benchmark*. Hoffman et al. use a variation of

Michael and Scott's *pairwise benchmark*, where each thread alternates between enqueues and dequeues. It is possible that this variation was purposely chosen, as the alternation of operations creates an environment where the *'baskets'* mechanism is used more frequently.

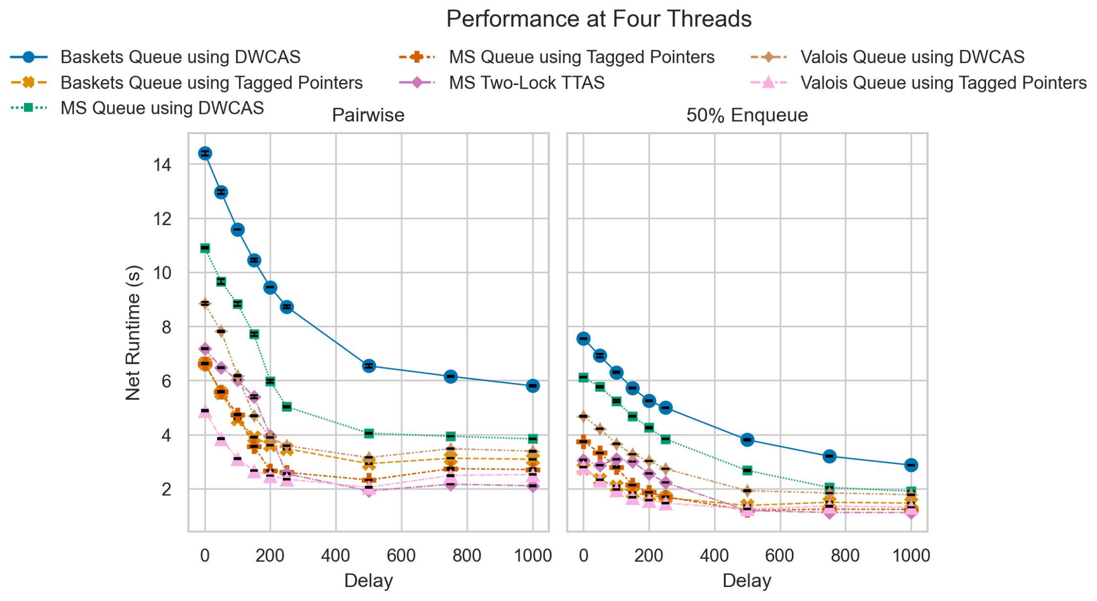

Figure 4.6: Pairwise and 50% Enqueue Benchmarks at four threads.

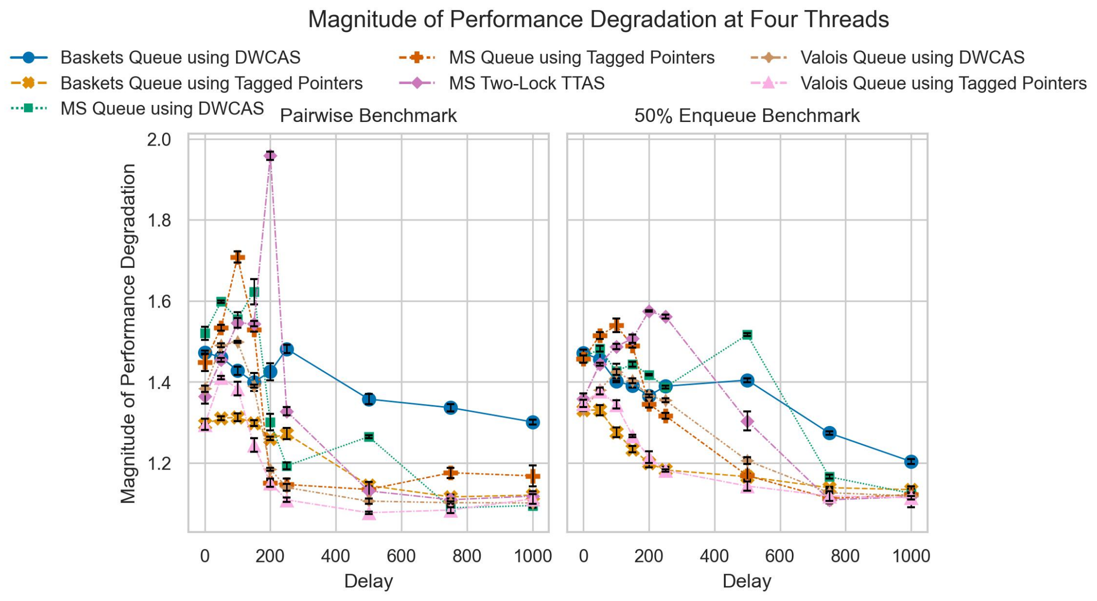

Figure 4.7: Degree of performance degradation at four threads.

4.5 Performance Under Oversubscription

4.5.1 Workload Under Five Threads

In this study, processors are oversubscribed by pinning more than one process to each processor, forcing the thread scheduler to increase the frequency of context switches.

Figure 4.9 shows that between 0 and 150 nanoseconds of delay, the magnitude of performance degradation is a linear function of delay; As delays greater than 150 nanoseconds are used, performance degradation explodes. One may hypothesize that the sudden explosion in performance degradation is a result of the significantly decreased time between context switches; as delay increases, the total CPU time available is further reduced.

In the pairwise benchmark, between 0 and 250 nanoseconds of delay, the *Baskets Queue* and the *MS-Queue* are at most **47.045%** and **60.604%** faster than the *two-lock queue*; Between 500 and 1000 nanoseconds, the *two-lock queue* is at most **3.351%** faster than the baskets queue.

Similar to the trends observed in section 4.4, the *Baskets Queue* significantly outperforms the *MS-Queue* (by at most **53.645%**) only in the *50% Enqueue Benchmark*. This repeating pattern shows that the performance of the *Baskets Queue* is heavily dependent on the utilization of the baskets mechanism.

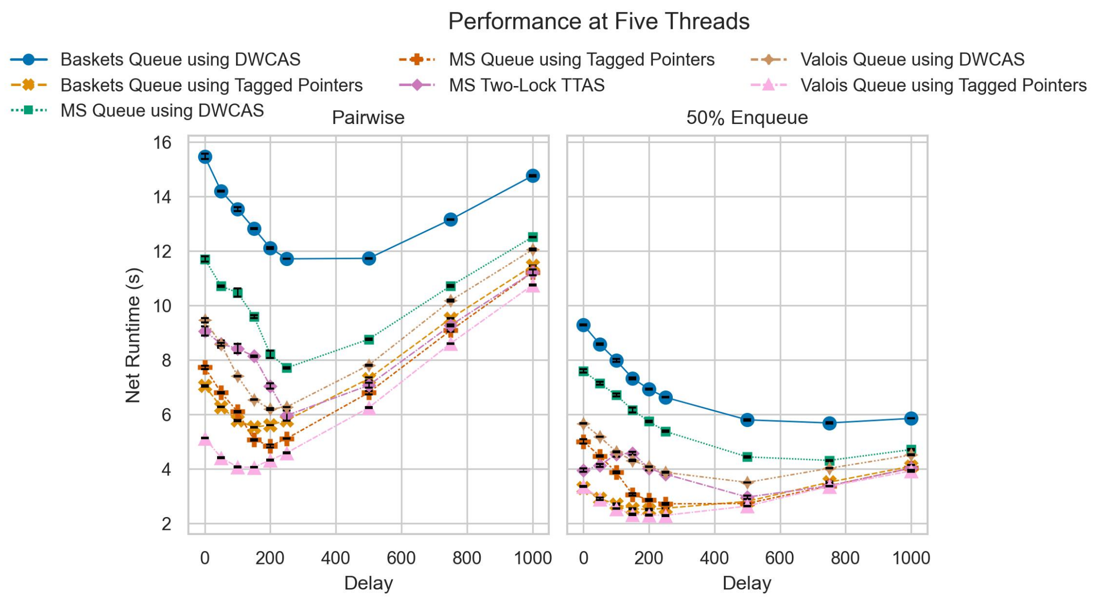

Figure 4.8: Pairwise and 50% Enqueue Benchmarks at five threads.

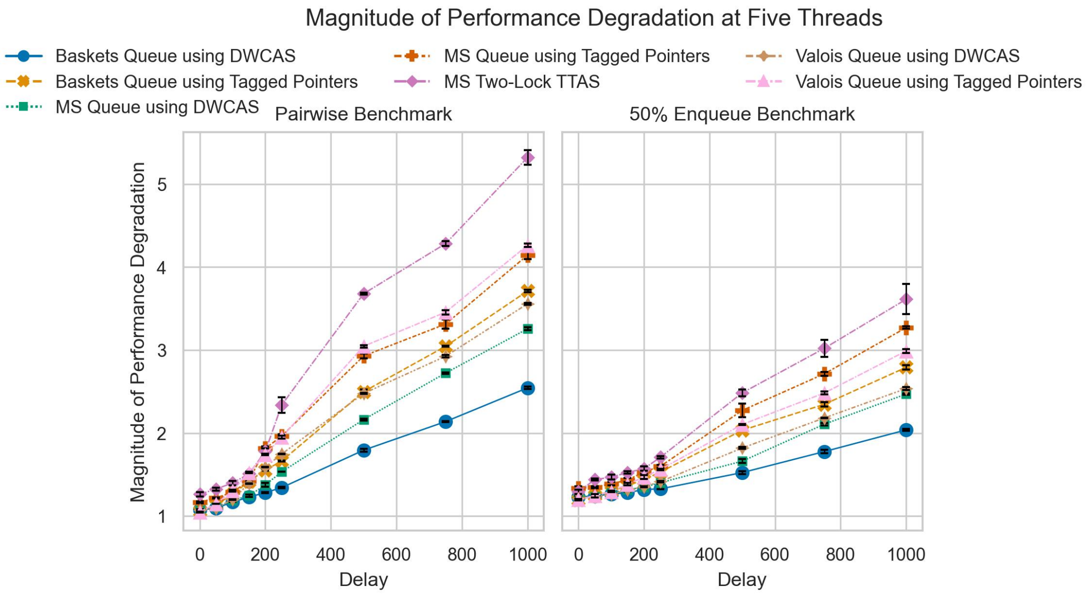

Figure 4.9: Degree of performance degradation at five threads.

4.5.2 Six Threads and Above

With six threads, the non-blocking queues significantly outperform the *two-lock queue* at every delay. This trend repeats itself up to 12 threads, with the degree at which the non-blocking queues outperform the *two-lock queue* increasing at every thread; the degree at which the *baskets queue* outperforms the *MS-Queue* also grows with every thread.

4.6 Effects of Delay on Performance

Valois and Hoffman et al. [22, 47] note that back-off algorithms are required to improve the efficiency of concurrent algorithms. More often than not, a queue's optimal delay tends to be similar in consecutive threads, with the pattern changing once oversubscription is present. Between one and four threads, 91.667% (under the *pairwise benchmark*) and 83.333% (under the *50% enqueue benchmark*) of all optimal net runtimes have a delay of **500 nanoseconds**. Between five and twelve threads, each queue's optimal delay slightly varies between zero and 250 nanoseconds.

4.7 Biases and Threats to Validity

Artificiality of Workloads In [17], Gregg claims that *micro-benchmarks* (type of benchmark adopted in this study) produce artificial workloads, rendering results which are solely obtained under various assumptions. Results obtained in this study represent the concurrent queues, whose operations are separated by fixed delays, inside a clean-room environment. Realistic workloads seldom follow such rigid patterns, making the results' validity under real-world scenarios undetermined.

High Variance Due to the quantitative nature of this study, variance plays a major role in determining the reproducibility of the study. An arbitrary coefficient of variance of 5% was chosen as the threshold for acceptable reproducibility. From the whole dataset, only the *Baskets Queue* using Tagged Pointers at one thread and 500 nanoseconds of delay (under the *pairwise benchmark*), and the *MS Queue* using Tagged Pointers at the same number of threads and delay (under the *50% enqueue benchmark*) had a coefficient of variance higher than 5% (5.700% and 7.424% respectively). The dataset's 99% quantile for coefficient of variance is **2.874%**.

Lack of Benchmarking Standard Unlike the field of databases—concurrent queueing algorithms do not have a standardized benchmarking methodology—allowing researchers to choose variations of benchmarks that provide subtle biases in favour of their agenda.

4.8 Final Remarks

This project was evaluated on a consumer-grade CPU with four cores, making over-subscription necessary to take readings over four threads. Although this limitation immediately affects the evaluation of this study, the benchmarking framework aids with collecting performance data on more powerful *x86_64 Intel* CPUs.

In this study, the concurrent queueing algorithms presented in [22, 37, 47] were implemented and compared with the results obtained from their seminal papers (as per research objective **O4**). Tagged pointers using atomic Double-Width-Compare-And-Swap (DWCAS) instructions (for ABA prevention purposes) were several times slower than their Single-Word-Compare-And-Swap (CAS) counterparts.

It was noted that Michael and Scott [37] omitted a discussion on the memory-reclamation scheme overheads of each of their implementations—which if brought to light—would have shown that the choice of memory-reclamation scheme for the *MS-Queue* (the authors' novel algorithm) potentially led to significant biases in their favour.

The following hypothesis was presented to the reader: The algorithm spawned from the marriage of *Valois' queue* and the *safe-read protocol* is significantly more inefficient than most other permutations of *Valois' queue*, due to the required linked-list traversals.

Under single-threaded workloads, *non-blocking* queues were consistently outperformed by Michael and Scott's *Two-Lock Queue* [37]. Workloads of two threads present that tagged pointers using CAS suffer from a harsher degradation in performance when compared to their DWCAS counterpart, as they are more likely to fail a CAS, requiring one or more retries to completely commit their changes. Yet again, the *Blocking Two-Lock Queue* outperformed the *non-blocking* algorithms. Similar to prior art [22, 30, 37], every queue's net runtime increased by several times under two threads. Michael and Scott claim that a speed-up of less than a factor of $\frac{1}{3}$ can be observed in the *MS-Queue*, under a workload of three threads [37], however, similar to [22, 30], the observed speedups were more modest (up to **6.591%**). Under high contention at three threads, the *non-blocking* queues were superior to the *blocking two-lock queue*. Under four threads, non-blocking queues outperformed the *two-lock queue* at high contention; the *Baskets Queue* only outperformed the *MS-Queue* in the *50% Enqueue Benchmark*, as it utilized the 'baskets' mechanism up to **110.964** times more than in the *Pairwise Benchmark*. Under workloads of five threads, the CPU was over-subcribed, causing the operating system to context-switch more frequently—reducing the CPU time slice allocated to each thread. *Non-blocking queues* remained superior at high degrees of contention, with the *blocking Two-Lock-Queue* only becoming competitive at lower degrees of contention. Every workload with six threads or more showed monotonically attenuating trends similar to those observed at five threads.

A few challenges encountered in this project included minimizing interference created by the operating system, and the creation of a simple memory-management system which could be used in a lock-free manner. Performance measuring tools, such as the instrumentation of counters and the measurement of time had to be amortized over several million iterations, as to dampen their overheads.

5 Conclusions & Future Work

5.1 Conclusion

Limited work explicitly focusing on surveying concurrent queueing algorithms exists. Consequently, we implement the algorithms in [22, 37, 46] and compare their performance through identical benchmarks. This study addresses the following research objectives:

O1. *Implement a benchmarking framework for concurrent queueing algorithms capable of gathering measurements similar to prior works;*

Section 3.2 offers a detailed description of the benchmarking methodologies used in this study; A high-level overview of the benchmarking framework's design and implementation is provided in section 3.3.

O2. *Reasonably validate the benchmarking framework through metrics and experiments;*

Section 3.4.5 describes the efforts taken in validating a number of the benchmarking framework's components. The absolute error of the artificial delay (i.e. the absolute difference between the actual average and the expected time) and its coefficient of variance are recorded and are kept to a minimum. Although every queue's implementation is not thoroughly tested (in terms of code coverage, due to the complexity of testing concurrent algorithms), the frequency at which specific code paths are executed is measured; specific distributions of these counters act as a tell-tale sign that each algorithm is either working or not working as expected. Finally, the repeatability of the benchmarking framework is quantified through the results' coefficient of variance, where the 99% quantile of the coefficient of variance for this study is 2.874%

O3. *Implement a variety of concurrent queueing algorithms, with the aim of replicating results from the original works;*

Four queues [22, 37, 46] (three non-blocking, and one blocking) are implemented in this study. The non-blocking queues require tagged pointers (pointers combined with a version counter) for ABA avoidance purposes; implementations of both 64 bit and 128 tagged pointers are presented in this study (making a total of seven queues).

O4. *Critically compare each concurrent queueing algorithm's performance under a variety of synthetic benchmarks.*

Chapter 4 solidifies the study by offering multi-faceted comparisons and interpretations of each queue's performance. None of the queues in this study were able to outperform every other queue under all circumstances, hinting at the fact there is no such thing as a one-size-fits-all queue. Blocking queues outperform non-blocking queues at low levels of contention (i.e. a combination of low thread count and high delay), however, non-blocking queues still remain competitive. At high levels of contention, blocking queues are inferior to non-blocking ones.

Under workloads of three threads, the Michael and Scott queue boasts an impressive speedup of a factor less than $\frac{1}{3}$ [37], however, Hoffman et al.'s reported speedup for the *MS-Queue* [22] and the present study's reported speedup are more modest.

We observe trends similar to [22, 30, 37], where every queue experiences a significant degradation in performance under two threads, and a small speedup at three.

Michael and Scott's evaluation [37] introduces significant biases in favour of their novel algorithm through the choice of memory-reclamation schemes (effectively increasing the magnitude by which the *MS-Queue* circumstantially outperforms every other queue).

The *Baskets Queue's* performance is highly dependent on the utilization of its 'baskets' mechanism, which is triggered more often under alternating enqueues and dequeues. Hoffman et al. [22] fail to discover this shortcoming due to their choice of benchmarking methodologies. Consequently, the present study is only able to replicate Hoffman et al.'s results under the *50% Enqueue Benchmark*.

Between one and three threads, the *MS-Queue* consistently outperforms the *Baskets Queue*, however, only under the *50% Enqueue Benchmark* does the *Baskets Queue* start to consistently outperform the *MS-Queue* at four threads or above.

Queues using 128-bit tagged pointers (through the *Double-Width Compare-and-Swap* instruction) are several magnitudes slower than queues adopting 64-bit pointers.

5.2 Related Work

Pourmeidani [40] evaluates the performance of a blocking and a non-blocking (array-based) queue on a GPU, and finds that similar to concurrent queues on CPUs, non-blocking queues require sufficient parallelism to outperform blocking queues.

Gilbert [16] uses several open source concurrent queueing algorithms and measures their performance with the aim of finding the queue best suited for DBMS page evictions.

5.3 Critique and Limitations

Our evaluation is limited by the number of cores on the benchmarking machine's CPU, although this does not affect the validity of the study, it potentially reduces the amount of insights found.

The queues evaluated in this study do not include memory-reclamation schemes, potentially leading to biases.

Although several validation techniques are adopted, none of them are exhaustive enough to ensure freedom from race conditions. The evaluation does not discuss the benchmarking framework's overhead, making the degree of its effects on the benchmarked code benchmarked unknown.

5.4 Future Work

- Add more algorithms with different progress conditions, such as obstruction-free or wait-free queue;
- Adopt Curtsinger and Berger's *stabilizer* [10], which allows for statistically sound performance evaluations by forcing benchmark executions to sample the space of memory configurations;
- Further extend the study by running the benchmarks on UMA machines with better processors;
- Make the benchmarks more organic by substituting different queues in real world code (similar to Boyd-Wickizer et al.'s analysis of non-scalable locks [8]);
- Add a variety of memory-reclamation schemes to the implemented queues, discussing their effects on each queue's performance and behaviour.

References

[1] Memory model. https://en.cppreference.com/w/cpp/language/memory_model, 03 2022. Accessed: 2022-05-01.

[2] Aceto, L., Attard, D. P., Francalanza, A., and Ingólfsdóttir, A. On benchmarking for concurrent runtime verification. *Fundamental Approaches to Software Engineering*, 12649:3, 2021.

[3] Adelstein Lelbach, B. Cppcon 2015: Bryce adelstein-lelbach "benchmarking c++ code", 2015. URL https://youtu.be/zWxSZcpeS8Q. Accessed: 08-06-2022.

[4] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. An efficient unbounded lock-free queue for multi-core systems. In *European Conference on Parallel Processing*, pages 662–673. Springer, 2012.

[5] Anderson, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[6] *Arm Architecture Reference Manual for A-profile Architecture*. Arm® Limited, 2022.

[7] Arnautov, S., Felber, P., Fetzer, C., and Trach, B. Ffq: A fast single-producer/multiple-consumer concurrent fifo queue. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 907–916. IEEE, 2017.

[8] Boyd-Wickizer, S., Kaashoek, M. F., Morris, R., and Zeldovich, N. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[9] Cantrill, B. and Bonwick, J. Real-world concurrency. *Communications of the ACM*, 51(11):34–39, 2008. Accessed: 2022-05-01.

[10] Curtsinger, C. and Berger, E. D. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News*, 41(1):219–228, 2013.

[11] Debattista, K. and Vella, K. High performance wait-free thread scheduling on shared memory multiprocessors. 2002.

[12] Dechev, D., Pirkelbauer, P., and Stroustrup, B. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192. IEEE, 2010.

[13] Drepper, U. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[14] Fog, A. Optimizing software in c++, 2020.

[15] Gharachorloo, K. *Memory consistency models for shared-memory multiprocessors*. PhD thesis, 1996.

[16] Gilbert, M. F. Performance evaluation of different open-source implementations of data structures and algorithms in the context of a dbms buffer manager. 2020.

[17] Gregg, B. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[18] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1): 124–149, 1991.

[19] Herlihy, M., Shavit, N., Luchangco, V., and Spear, M. *The Art of Multiprocessor Programming*. Newnes, 2020.

[20] Herlihy, M., Shavit, N., Luchangco, V., and Spear, M. *The Art of Multiprocessor Programming*, chapter B.5.1. Newnes, 2020.

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

[22] Hoffman, M., Shalev, O., and Shavit, N. The baskets queue. In *International Conference On Principles Of Distributed Systems*, pages 401–414. Springer, 2007.

[23] *Intel® 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, A-Z*. Intel®, 2021.

[24] *Intel® 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide*. Intel®, 2021.

[25] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel®, 2021.

[26] Intel®. Architecture agnostic spin-wait loops. `https://www.intel.com/content/www/us/en/developer/articles/technical/a-common-construct-to-avoid-the-contention-of-threads-architecture-agnostic-spi` `html`, 2018. Accessed: 30-05-2022.

[27] Knuth, D. E. The art of computer programming, vol 1: Fundamental. *Algorithms*, page 187, 1968.

[28] Kogan, A. and Petrank, E. Wait-free queues with multiple enqueuers and dequeuers. *ACM SIGPLAN Notices*, 46 (8):223–234, 2011.

[29] Kogan, A. and Petrank, E. A methodology for creating fast wait-free data structures. *ACM SIGPLAN Notices*, 47(8): 141–150, 2012.

[30] Ladan-Mozes, E. and Shavit, N. An optimistic approach to lock-free fifo queues. *Distributed Computing*, 20(5): 323–341, 2008.

[31] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers c-28*, 9:690–691, 1979.

[32] McKenney, P. E. *Is parallel programming hard, and, if so, what can you do about it?* Second edition, release candidate 11 edition, 2021.

[33] McKenney, P. E. *Is parallel programming hard, and, if so, what can you do about it?*, chapter 3.2.1. Second edition, release candidate 11 edition, 2021.

[34] Mellor-Crummey, J. M. Concurrent Queues: Practical Fetch-and-Phi Algorithms. Technical report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1987.

[35] Mellor-Crummey, J. M. and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[36] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[37] Michael, M. M. and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[38] Michael, M. M. and Scott, M. L. Fast concurrent queue algorithms. `http://www.cs.rochester.edu/research/synchronization/code/concurrent_queues/SGI.tgz`, 1996. Accessed: 2022-08-02.

[39] Oracle®. Chapter 17 Threads and Locks. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4`, 04 2014. Accessed: 2022-05-01.

[40] Pourmeidani, H. Performance evaluation of blocking and non-blocking concurrent queues on gpus. 2019.

[41] Ramalhete, P. Cpp. https://github.com/pramalhe/ConcurrencyFreaks/blob/c61189546805c67792df7931f9484e09a3cda3bf/CPP/pqueues/pfences.h#L33, 2019.

[42] Rudolph, L. and Segall, Z. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, 1984.

[43] Scott, M. L. Shared-memory synchronization. *Synthesis Lectures on Computer Architecture*, 8(2):1–221, 2013.

[44] Silberschatz, A., Gagne, G., and Galvin, P. B. *Operating System Concepts*. Wiley, 10th edition edition, 4 2018. ISBN 978-1-119-32091-3.

[45] Terpstra, D., Jagode, H., You, H., and Dongarra, J. Collecting performance data with papi-c. In Müller, M. S., Resch, M. M., Schulz, A., and Nagel, W. E., editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11261-4.

[46] Valois, J. D. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[47] Valois, J. D. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1995.

[48] Yahav, E. and Sagiv, M. Automatically verifying concurrent queue algorithms. *Electronic Notes in Theoretical Computer Science*, 89(3):450–463, 2003.

[49] Yang, C. and Mellor-Crummey, J. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

A Abridged Source Code

A.1 Original Implementation of the MS-Queue's Memory Reclamation Scheme

```
void
init_memory()
{
}

unsigned
new_node()
{
  return private.node;
}

void
reclaim(unsigned node)
{
  private.node = node;
}
```

Listing A.1: Memory management used in Michael and Scott's original implementation of the MS queue

A.2 Memory Management Used in Michael And Scott's Original Implementation of Valois' Queue

```
void
init_private()
{
  private.value = 1 + initial_nodes + (pid * iterations);
```

```
}

void
init_memory()
{
}


unsigned
dec_tas(unsigned* ptr)
{
  unsigned old;
  unsigned new;

  do {
    old = *ptr;
    new = old - 2;
    if (new == 0) {
      new = 1;
    }
  } while (cas(ptr, old, new) == FALSE);
  return (old == 2);
}

void
clear_bit(unsigned* ptr)
{
  unsigned old;
  unsigned new;

  do {
    old = *ptr;
    new = old - 1;
  } while (cas(ptr, old, new) == FALSE);
}

node_t*
alloc()
{
  node_t* pnode;
  unsigned success;

  for (success = FALSE; success == FALSE; ) {
    pnode = safe_read(&smp->Avail);
    assert(pnode != NULL);
    success = cas(&smp->Avail, pnode, pnode->next);
    if (success == FALSE) {
      release(pnode);
    }
  }
  faa(&smp->allocated, 1);
  return pnode;
}

void
reclaim(node_t* pnode)
{
  node_t* old;

  do {
    old = smp->Avail;
    pnode->next = old;
  } while (cas(&smp->Avail, old, pnode) == FALSE);
  faa(&smp->reclaimed, 1);
}

node_t*
new_node()
{
  node_t* pnode;

  pnode = alloc();
  faa(&pnode->refct, 2);  /* adjust */
  clear_bit(&pnode->refct);
  return pnode;
}

void
release(node_t* pnode)
{
  node_t* ptr;
  node_t* next;

  ptr = pnode;
  while(1) {
    if (dec_tas(&ptr->refct) == FALSE) {
      return;
    }
    next = ptr->next;
    reclaim(ptr);
    ptr = next;
  }
}

node_t*
safe_read(node_t** ptr)
{
  node_t* pnode;

  while(1) {
    pnode = *ptr;
    if (pnode == NULL) {
      return NULL;
    }
    faa(&pnode->refct, 2);
    if (pnode == *ptr) {
      return pnode;
    }
    release(pnode);
  }
}
```

Listing A.2: Memory-reclamation scheme used in Michael and Scott's original implementation of the Valois' Queue

B Interfaces

B.1 Queue Interface

Queues are expected to implement the following seven methods found in *src/queues/queue.h*:

bool create_queue(void** out_queue)

void out_queue** Queue to be initialized;

bool enqueue(void* queue, void* in_item)

void* queue Pointer to an initialized queue;

void* in_item Pointer to the item that is to be enqueued;

Return *true* if enqueue is successful, else returns *false*

bool dequeue(void* queue, void** out_item)

void* queue Pointer to an initialized queue;

void out_item** - Pointer to a pointer of the variable that will be assigned the dequeued value.

Return *true* if dequeue is successful, else, if the queue is empty, or an error occurs, return *false*

void destroy_queue(void** out_queue)

void out_queue** - Pointer to a pointer of a queue that is to be de-allocated (freed).

void register_thread(size_t num_of_iterations)

This function is to be called in each thread that is used during the benchmark in order to allocate the memory necessary.

size_t num_of_iterations - The number of iterations in which the enqueue function is called, which will determine how many nodes queue elements need to be initialized for each thread.

void cleanup_thread()

Frees the memory allocated by *register_thread*; It is the user's responsibility to call this function only after all the expected operations have been executed, as it is possible for an active thread to encounter a use-after-free error.

char* get_queue_name()

Return the name of the queue being benchmarked. The result of this function is appended to the results of the benchmark, in order to identify which results belong to a queue.

B.2 Lock Interface

bool create_lock(void lock)**

void destroy_lock(void lock)**

void wait_lock(void* lock)

void unlock(void* lock)

char* get_lock_name()