DESIGNING A HIGH THROUGHPUT BOUNDED MULTI-PRODUCER, MULTI-CONSUMER QUEUE

An Undergraduate Research Scholars Thesis

by

REGINALD A. FRANK

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

BACHELOR OF SCIENCE

Approved by
Faculty Research Advisor: Dr. Dmitri Loguinov

May 2021

Majors: Computer Science
Applied Mathematical Sciences

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Reginald A. Frank, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Designing a High Throughput Bounded Multi-Producer, Multi-Consumer Queue

Reginald A. Frank
Departments of Computer Science & Engineering and Mathematics
Texas A&M University

Research Faculty Advisor: Dr. Dmitri Loguinov
Department of Computer Science & Engineering
Texas A&M University

Multi-Producer Multi-Consumer (MPMC) Queues are the most natural way to solve the common parallel programming Producer-Consumer Problem. The problem arises when a group of entities need work to be done, and another group of entities are responsible for doing said work. The problem is then of how should work be allocated to the workers such that neither group is hindered by the communication required for work allocation. MPMC Queues can solve the problem of allocation by functioning as a global location for work requests to be posted by the former group and later removed to be acted on by the latter group joining the two, but as the amount of work to be done by a system increases, this singular connection can easily become a bottleneck preventing work from being done. Current high performance MPMC Queue implementations strictly enforce that work posted first will be scheduled to a worker first, and while this improves the latency of a system, it can greatly decrease the overall work throughput, crippling bulk data-processing application performance. This project aims to create an MPMC Queue that is focused on overall throughput and investigate what performance optimizations can be made by sacrificing the standard latency guarantees.

1. INTRODUCTION

Shared objects in general serve the purpose of allowing different processes or threads to communicate with one another. One of the earliest examples of this notion appeared in Dijkstra's 1965 manuscript [1] where there is one producer and one consumer process; the former continuously writes, or pushes, data to a shared buffer of unbounded size, and the latter continuously reads, or pops, the data written to the buffer. The data is read out in the same order as it is inputted making the buffer act as a way to queue up the producer's inputted data to be read by the producer. Because there is a single producer and a single consumer acting on a unbounded size buffer, the construction today is known as an unbounded Single-Producer Single-Consumer (SPSC) Queue.

This is simultaneously one of the simplest yet most pervasive instances of inter-process communication, and since its inception, various extensions have been made to the framework used. Some of these extensions include the ability for the queue to support multiple producers and or multiple consumers, providing queue buffers of a fixed or variable size, having mandates on the what order elements can be extracted from the queue, and providing guarantees on how quickly each thread can make progress in its goal of inserting or removing elements from the queue.

In this project, the aim is to identify which framework will allow the most throughput of data written and read when using modern hardware, implement a producer consumer queue in that framework, and compare how our solution performs when compared to existing queues in their frameworks.

1.1 Model Selection

In order to better understand what types of decisions go into designing a queue each decision will be covered now in more detail.

First, the most fundamental choice in the design is deciding on how many producers and consumers the queue can support. Modern systems have multiple cores which allow simultaneous execution of instructions, but in order to take advantage of this, the program must split the work being done across multiple threads. To provide a simple mental model for a best case calculation, let the processor have N cores where in a unit of time each core can allow X and Y push and pop operations respectively. This means that if the queue on a specific system supports the simulations operation of up to $P, C \leq N$ producers and consumers respectively, the number of elements going through the queue overall during a steady state operation is less than or equal to $2 \cdot \min(P \cdot X, C \cdot Y)$. This is because the slower operation is the bottleneck for elements going through the queue–there cannot be more elements popped than there are pushed, and pushing more elements than can be popped doesn't increase throughput through the queue. Using this mental model, we see that queues that are not MPMC restrict either P and or C to be at most one which in any case will bottleneck the max theoretical queue throughput. This motivates the creation of a MPMC Queue to avoid this bottleneck.

The question of if a bounded or unbounded buffer should be used during operation is also somewhat nuanced. Modern Operating Systems are intended to create the appearance of allowing programs infinite memory which then allows the use of a seemingly unlimited sized buffer, but in reality, each computer only has so much memory which can be used at once. Unbounded sized buffers then have the potential to exhaust this limited amount of memory, but in practice this is rather unlikely. More concerning is that the fastest memory in modern computers are the CPU's on chip L1, L2, and L3 caches, with RAM being larger but significantly slower. Unbounded sized buffers will be unable to fit in these small but extremely fast locations in memory which will largely decrease performance and therefore throughput as well. Even though unbounded buffers make producers able to work without waiting for space to free up in the queue, the extra cost of accessing slower memory outweighs this benefit nonetheless. For this reason this paper will consider a queue of bounded size.

Note that because we have decided on a bounded MPMC queue, there are now unsuccessful outcomes to both push and pop operations. On a push operation, the queue can be full meaning the producer is dependent on the consumer to push elements to make the queue non-full. On a pop operation, the queue may be empty meaning that the consumer is dependent on the producer to pop elements to make the queue non-empty. These dependencies will restrict the processes from making progress at different points during the execution. In a worst case or adversarial process scheduler, a producer or consumer may only be given resources when they are unable to make progress essentially forever blocking them. Many current queue implementations such as [2, 3] allow push and or pop operations to terminate if making progress is momentarily impossible due to a full or empty queue. This change allows the threads to not get blocked, however, this leaves open the problem of how or when threads should retry their queue operations as even though they are not blocked by the queue operation itself, they were prevented from pushing or popping an element. Section 2.4 shows the decision of how to implement retries can have a drastic impact on queue performance. Therefore, in this paper we will focus on making a queue with blocking operations to address this problem of retries. This means that producers and consumers will continue their current operation until they are successful, but in a general execution there is no guarantee on an operation being successful. The cause of this is because if consumers pop from a permanently empty queue, no operation will ever be successful no matter the type of queue implemented. The same can be said for producers which push to a permanently full queue. Despite the weak condition for a general execution, we provide that if the queue is not indefinitely full or empty then push and respectively pop operations should be *lock-free* meaning at least one thread will always be able to eventually complete its operation.

Lastly, the order that elements are removed from the queue is of interest which falls under the queue's consistency guarantees. Typically MPMC queues follow a *linearizable* consistency model where a queue maintains a total global order where each element is removed from the queue in the same order as they were put in regardless of what thread inserted or removed the element [4]. While this consistency is the most natural and easiest to program with, [5] shows that a linearizable queue shared by n threads has a worst case execution where each operation will $\Omega(n)$ steps to complete its operation. An extensive execution on a queue can be looked at as a series of smaller rapid smaller executions including up to n processes, meaning that even if the absolute worst case doesn't always occur, the slow sub-executions can lessen the overall queue throughput.

Loosing all consistency would be undesirable from a usability standpoint, so we propose the idea of using a *locally consistent* model which we define as guaranteeing that if a consumer pops any two elements a, b from the same producer, then it popped a before b *iff* the producer pushed a before b. Another way to justify this choice is that in an extremely high throughput environment, virtually all producers will be constantly pushing elements making the question "of all producers in the system concurrently pushing, who finished first?" somewhat arbitrary. In that situation, even a linearizable queue's ordering between different threads is largely a race condition due to the constant contention. However, the ordering within threads is always kept no matter what as each process must finish its current operation before starting its next operation. From this view point, in this environment local consistency keeps much of the ordering while avoiding the $\Omega(n)$ worst case lower bound on the number of operations for a push and pop.

Now as a result of the previous discussion, this paper will be designing a Locally Consistent, Multi-Producer Multi-Consumer, Bounded Queue, with blocking push and pop operations. Additional detail on the model is included in Section 3.1 which more formally defines the model, compares it to other consistency models, and states liveness properties.

2. Analysis of Related Work

Here, the design, performance, and application of prior works will be discussed. Many of the ideas introduced from both past SPSC as well as MPMC queues will be adjusted and restudied in our chosen model to guide development of this thesis's finished queue.

2.1 SPSC Queue Designs

2.1.1 SPSC Mutual Exclusion Queue

When the Producer Consumer problem was first introduced by Dijkstra [1] the intended solution was to write code to prevent producers and consumers from pushing and popping from the queue at the same time. When this solution was proposed, the difficulty was in producing code which properly excludes producers and consumers from mutually accessing the same location in code no matter what order the executions operate in. Nowadays, this problem is solved in highly optimized ways by mainstream operating systems, therefore we will be testing a queue that utilizes a kernel level mutual exclusion routine on an underlying single-threaded queue as seen in Listing 2.1.

One important thing to note about this solution is that it will serve as the baseline performance for all other designs. This solution only allows one thread to make progress at a given time, therefore it effectively can only make full use of one CPU core at any point in time, other designs may allow the producer and consumer to make progress concurrently. For SPSC queues this means that up to two cores can be concurrently used potentially doubling the queue performance.

2.1.2 Lamport's Concurrent Lock-Free Queue

One optimization of the previous solution was found by Lamport in [6], which proves that if the underlying Reads and Writes to the head and the tail satisfy a sequentially consistent memory model, then the prior algorithm in Listing 2.1 can remove the `mutex.lock()` and `mutex.unlock()` statements while maintaining a correct algorithm. Most all systems satisfy a sequentially consistent memory model, and that means that the mutual exclusion in the prior section which prevented

```
bool tryPush(const int element) {
  mutex.lock();
  int nextTail = (tail + 1) % queueSize;
  if (nextTail == head) {
    mutex.unlock();
    return PUSH_FAILED;
  }
  buffer[tail] = element;
  tail = nextTail;
  mutex.unlock();
  return PUSH_SUCCEEDED;
}

bool tryPop(int &elementOut) {
  mutex.lock();
  if (head == tail) {
    mutex.unlock();
    return POP_FAILED;
  }
  elementOut = buffer[head];
  head = (head + 1) % queueSize;
  mutex.unlock();
  return PUSH_SUCCEEDED;
}
```

Listing 2.1: Simple SPSC mutual exclusion queue.

simultaneous pushes and pops to occur is actually often unnecessary. This change will allow the queue to make progress on multiple cores at the same time which in theory should improve performance.

Furthermore, `mutex.lock()` and `mutex.unlock()` are both expensive operating system calls which eat up time that could have otherwise been spent pushing or popping elements. Removing these calls would intuitively incur less overhead than the prior solutions, but there are some competing effects which complicate this analysis. In order to maximize system performance, modern AMD and Intel CPUs may reorder read and write operations on different cores [7, 8]. This can cause an increase in throughput and latency as the queue may have had elements written to it by a producer, but when the consumer goes to read the queue, their reads may be reordered to occur before when the element was written to the queue making the consumer believe the queue is empty causing the pop to fail. The same can happen to producers where they may read the queue as full despite the consumer having written to the queue during completed pop operations. In the worst case, both producers and consumers may both believe they are blocked at the same time.

```
// all buffer elements initialized to EMPTY_SLOT
static constexpr int EMPTY_SLOT = NULL;

bool tryPush(const int data) {
  if (buffer[tail] != EMPTY_SLOT) {
    return PUSH_FAILED;
  }
  buffer[tail] = data;
  tail = (tail + 1) % queueSize;
  return PUSH_SUCCEDED;
}

bool tryPop(int &dataOutput) {
  dataOutput = buffer[head];
  if (dataOutput == EMPTY_SLOT) {
    return POP_FAILED;
  }
  buffer[head] = EMPTY_SLOT;
  head = (head + 1) % queueSize;
  return PUSH_SUCCEEDED;
}
```
Listing 2.2: FastForward cache optimized queue.

This won't continue forever, the writes will eventually be visible to all threads, but threads being blocked by the delayed visibility can decrease overall throughput. Then, notably, modern CPUs typically won't reorder Reads or Writes over assembly operations with a `LOCK` prefix which are used in kernel operations such as `mutex.lock()` and `mutex.unlock()` meaning the prior solution never has this issue. One way to get around this reordering over different cores is to fix the producer and consumer to the same core, but this then prevents different threads from both making progress on different cores which used to be an advantage of this solution. We can empirically see the impact of these competing factors in the performance review of these queues.

2.1.3 Fast Forward Cache Optimized Queue

One of the more recent academic studies of SPSC queues made an advancement by noticing that modern CPUs benefit heavily from caching variables, but this benefit can be handicapped if multiple threads all read from the same cached variable which is undergoing updates. The cause of this inefficiency stems from the CPU's cache consistency layer which on the write will attempt to give the reading thread the updated value from the writer thread, and since the fastest L1 and L2 caches aren't shared between cores, the update at best must go through the significantly slower shared L3 cache. If writes continuously happen then updates continuously go through, and these updates incur some overhead limiting queue performance. If a cached variable is only visible to one thread then all the updates will occur in the far faster L1 or L2 caches and no time is spent on pushing updates from one thread to the other. Then looking back at Listing 2.1 shows that the producer reads the head in line 4 which is continuously written to by the consumer in line 19, and the tail is read by the consumer in line 15 which is constantly written to by the producer in line 8. This means that the prior solution, there is constant implicit synchronization of variables between threads which can bring a large performance penalty to queue operation.

Giacomoni et al. address this issue by recognizing that any correct SPSC queue must at least synchronize what elements are stored in the buffer as this is where elements are written to and read from, therefore in [9], they propose a queue which only synchronizes using the buffer. Ingeniously their solution, showed in Listing 2.2 uses the buffer as the only shared variable which each process does only one read from the desired location followed by one write if the desired location is available. This method appears to use the minimal amount of synchronization to solve this problem by restricting one value from the queue to use as an empty slot token. The only potential problem is that despite what one may first think, lowest communication does not necessarily imply it has the highest throughput. The reason is similar to how removing the `mutex.lock()` and `mutex.unlock()` in the Lamport queue gives the potential to make the queue seem empty and full at the same time when it may be neither. Here having less communication could manifest in more failed pushes and pops reducing effective performance.

2.1.4 Facebook Folly ProducerConsumer Queue

A more modern SPSC queue used by high performance applications is the Facebook Open-source Library's (Folly) ProducerConsumerQueue [10]. This design provides a bounded, non-blocking, lock-free queue using a design algorithmically equivalent to the Lamport SPSC queue design. It's major difference is that it uses advanced c++ semantics to strictly define what order reads and writes to the `head` and `tail` occur between separate threads.

Using a sophisticated method of ordering of reads and writes will definitely outperform the implicit compiler chosen read/write ordering – this means that we would expect this to be more performant than the Mutual Exclusion design or the Lamport design. However the comparison to the FastForwardQueue becomes more complicated because a strict ordering doesn't change the issue that more updates will frequently have to be pushed through the L3 cache for each push and pop operation. Despite this disadvantage, the strict read/write ordering does give it the potential to let more pushes and pops to succeed because it limits the potential for updates to not be given to the other threads.

2.1.5 Moodycamel Open Source ReaderWriterQueue

This recent SPSC design [11] provides many modern features such as optional resizing, bulk operations, and optional blocking operations. Because none of the other designs implement blocking operations, of most interest is to see the potential benefits of its blocking strategy. However, its implementation of blocking relies on a counting semaphore which notates how many elements are present in the queue. This means that on every push the producer must release the semaphore, and on every pop the consumer must wait on the same semaphore. This design also doesn't provide blocking for push operations making this implementation likely bottlenecked by the overhead from touching a semaphore on every iteration.

Apart from this difference and its optional features, it has much similarity to the Folly ProducerConsumerQueue. Ignoring resizing, the queue is essentially a modified Lamport queue with modern C++ semantics for strict memory ordering between threads which can increase its performance compared to the more ambiguous original Lamport implementation.

2.2 SPSC Queue Performance

The benchmarking methodology goes as follows. A single Producer and Consumer would be created, without affinity to any specific core, each locally keeping count of how many successful push or pop operations it has successfully completed, occasionally atomically adding the local total to a global count. As seen in Listing 2.3, threads working on queues with blocking operations invoke a push or pop method and then increment its counter. Importantly the variables briefly

```
void nonBlockingProducerRoutine() {
    unsigned int localPushes = 0;
    while (true) {
        localPushes += queue->push(localPushes);
        if (localPushes > THRESHOLD_TO_SYNCHRONIZE) {
            atomicAdd(&totalPushes, localPushes);
            localSent = 0;
        }
    }
}

void nonBlockingConsumerRoutine() {
    unsigned int localPops = 0;
    int curElement;
    while (true) {
        localPops += queue->pop(curElement); // add true (1) or false (0)
        if (localPops > THRESHOLD_TO_SYNCHRONIZE) {
            atomicAdd(&totalPops, localPops);
            localPops = 0;
        }
    }
}
```

Listing 2.3: Benchmarking routine for nonblocking queues.

used in operation to prevent the compiler from optimizing out portions of the push and or pop routines. Then as seen in Listing 2.4, threads working on queues with nonblocking operations add the result of its push or pop operation to its local counter which prevents aggressive compiler optimization and also counts the number of successful operations at the same time. Once either thread's local number of operations surpass a threshold, it atomically update a global count of how many operations had completed, the global operation count would then be output by a third metrics thread.

Tests were repeatedly run at multiple different queue sizes until the results stabilized giving the results presented in Figure 2.1. The tests were run on a a system running Windows 10, code was compiled using Visual Studio 2019 set to the maximum optimization level. The system is running on a Core i9 9900K, an 8 core CPU with hyperthreading, overclocked to 4.75GHz equipped with 64GB of DDR4 RAM running at 3200MHz. This same benchmarking methodology is repeated throughout this document with the only difference being in how many producers and consumers are being used on the queue tested.

```
void blockingProducerRoutine() {
    unsigned int localPushes = 0;
    while (true) {
        localPushes += 1;
        queue->push(localPushes);
        if (localPushes > THRESHOLD_TO_SYNCHRONIZE) {
            atomicAdd(&totalPushes, localPushes);
            localSent = 0;
        }
    }
}

void blockingConsumerRoutine() {
    unsigned int localPops = 0;
    while (true) {
        localPops += (queue->pop() != 0);
        if (localPops > THRESHOLD_TO_SYNCHRONIZE) {
            atomicAdd(&totalPops, localPops);
            localPops = 0;
        }
    }
}
```

Listing 2.4: Benchmarking routine for blocking queues.

Then in Figure 2.1 many results predicted in the prior analysis are apparent. The MutexQueue is the worst performing of all implementations, slightly beat by the blocking Moodycamel ReaderWriterQueue which similar to the MutexQueue requires access of a synchronisation variable on each queue operation. The difference can be explained by the Moodycamel gaining additional throughput from allowing concurrent access despite the synchronization overhead. Further improving, the Lamport queue design gives the benefits of allowing concurrent access while also removing synchronization overhead which preforms almost three times as well as the prior methods. Then nearly doubling that throughput, the Facebook Queue performs even better likely due to its more deliberate synchronization. After that, despite having only implicit inter-core synchronization, the Fast Forward queue performs better than the prior methods due to improved cache use, and then scales incredibly with the buffer size far outperforming other methods likely due to further improved cache use. The larger the queue, the less likely they'll be acting on the same cache lines which improves performance. Additionally when an element is modified by one thread, the other thread must go through more elements to reach the modified element, this gives the cache

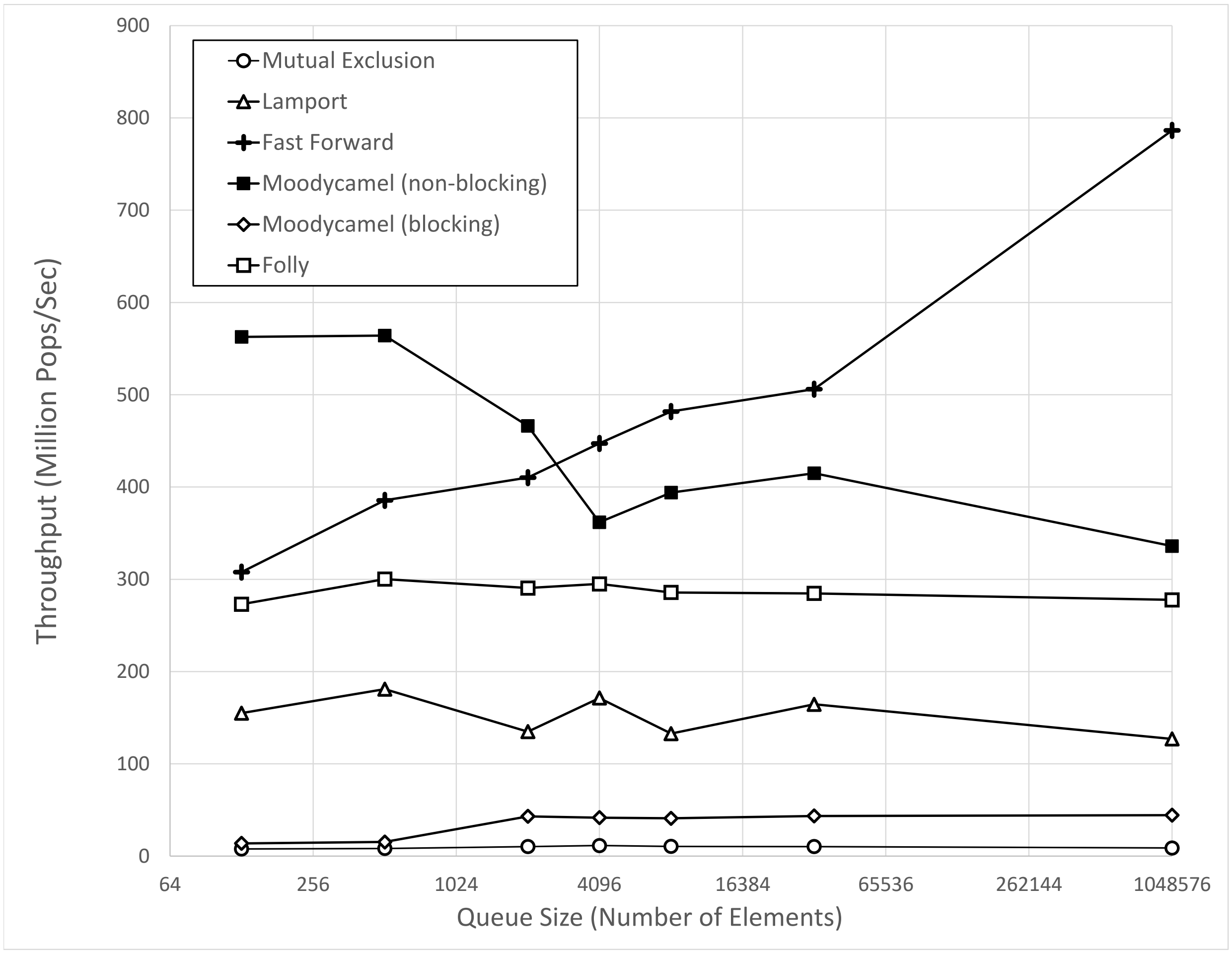

Figure 2.1: SPSC throughput vs queue size.

consistency mechanisms of the CPU more time to send updates reducing the amount of failed Queue operations. This impact is then further expanded as the retry mechanism for nonblocking queues, as shown in Listing 2.3, is busy-spinning which when combined with a lack of explicit synchronization can lead to many wasted cycles until writes become visible. One downside with increasing the queue size is that the increased memory leads to a decrease in locality which means that there is more movement of data through different levels of the computer's memory hierarchy which incurs a communication cost. Knowing this then makes the performance of the smaller sized Moodycamel ReaderWriter queues all the more impressive, even though they don't match the performance of the Fast Forward queue at larger sizes, they offer some of the best performance

```
bool tryPush(const int data) {
  producerMutex.lock();
  bool result = SPSCQueue.tryPush();
  producerMutex.unlock();
  return result;
}

bool tryPop(int &dataOutput) {
  consumerMutex.lock();
  bool result = SPSCQueue.tryPop();
  consumerMutex.unlock();
  return result;
}
```
Listing 2.5: Simple MPMC Mutual Exclusion Queue

at with 4 orders of magnitude less storage.

The differences between the many Lamport Queue implementations show the importance of memory synchronization. Leaving it all up to the compiler and system give the worst performance, the Facebook implementation gains some performance by its choice of semantics, and then the moodycamel ReaderWriterQueue acheives a threefold performance increase over the original Lamport queue using its semantics. One thing to note is that since these play at a level so low to hardware, these results may be a result of certain strategies doing well on our chosen benchmarking hardware while other systems may get better performance from potentially radically different strategies. However some amount of performance usually can be gained compared to the compiler chosen synchronization method by using application specific knowledge.

2.3 MPMC Queue Designs

2.3.1 MPMC Mutual Exclusion Queue

In the prior section, Mutual Exclusion enabled the construction of a SPSC queue by preventing the concurrent execution of threads which reduced the problem to implementing a standard single-threaded queue. Here we can see that we can use a similar technique in this section – the `producerMutex` and `consumerMutex` make it so only a single producer and single consumer can access the queue at once. This reduces the MPMC case to the SPSC case and thus by using one of the prior constructions we now have a correct MPMC queue. Furthermore, if the SPSC queue is linearizable, like all the discussed works, this queue is linearizable and has all of the same properties of the underling `SPSC` implementation.

This solution may seem incredibly trivial with much optimization to be had, but because of the step lower bound result for shared linearizable queues in [5] we actually know that the worst case for any linearizable implementation's operations in the worst case will take $\Omega(n)$ steps where n represents the number of producers along with consumers. This implementation's operations always takes exactly $\Theta(n)$ steps amortized because if $m \in \Theta(n)$ threads lock a mutex to access the queue, the i-th thread to enter has to wait for at least $i-1$ steps for the $i-1$ threads to lock and unlock it's mutex. If we sum these up and average them we find that the amortized cost per operation is $\frac{1}{m}\sum_{i=0}^{m-1} i = \Theta(\frac{m^2}{m}) = \Theta(n)$.

This essentially means that each linearizable MPMC queue, no matter the implementation will be about as slow as this implementation in a worst case execution, and the goal of these designs is to minimize how often these worst case executions occur. That leaves this queue as a good analysis queue to see how well the other linearizable queues do at avoiding the worst case performance.

2.3.2 Facebook Folly MPMCQueue

The Facebook Folly MPMCQueue [10] is a linearizable bounded queue with optional blocking that comes from the same library as the previous Folly SPSC queue.

At a high level, the design consists of a buffer of N blocking one element MPMC queues, where each push/pop operation requires a ticket. The tickets are some integer mod m, and if the residue of an operation's ticket is larger than the residue of the last operation's ticket, then the operation will succeed. The overall queue construction, similar to the Lamport SPSC queue then has a `head` and `tail` pointer but instead of pointing at elements it now points at the single element queues. The ticket is a shared variable between all threads, and it is atomically incremented on each operation, as are the `head` and `tail` variables.

Even though the costs of atomically incrementing the shared variables has a performance overhead, it allows multiple threads to concurrently access the queue which should give much better performance than the Mutual Exclusion solution on average.

2.3.3 MoodyCamel ConcurrentQueue

Of all the different queue implementations, the model used in this design [12] is the most similar to what is being studied in this paper. At a high level, their design allocates a Single-Producer Multi-Consumer (SPMC) Queue for each producer which has interacted with the queue, and then consumers will scan through the available queues until they find a queue which has an element to pop from. The specifics of their design provide a flavor of what we have defined as local consistency in their MPMC queue construction. They also do an excellent job of providing bulk push and pop methods, however these are not covered here as it hides the cost of synchronization between threads by getting N operations by synchronizing once, making it hard to compare implementations.

One potential issue with this design is that SPMC queues must incur more synchronization than a SPSC queue due to the $\Omega(n)$ lower-bound previously discussed. This likely restricts the performance of the overall despite it not being an atomic queue implementation.

2.3.4 Microsoft Concurrent Queue

Contrary to the previous designs presented, the Microsoft Concurrent Queue implements an unbounded Producer Consumer Queue [13]. The queue does support custom reallocation strategies so it would be possible to make the queue bounded with enough editing, but analysing this queue with its default reallocation strategy will show how unbounded designs effect performance.

2.3.5 Rigtorp MPMCQueue

This [14] is one of the more straightforward queue designs, it directly extends a Lamport queue to allow multiple producers and consumers by having them reserve slots by atomically incrementing shared `head` and `tail` variables. The biggest flaw of this design is that its performance is directly linked to the number of threads which can successfully increment the head and tail variables concurrently. Then because the queue is bounded by using wrap around a thread may get a slot after incrementing, but have to wait for the wraparound to unfold some number of times to make progress in addition to the potential that a slot may be assigned before the slot is ready to be consumed from or produced to. This design works around this problem by making each thread busy-spin until its slot is valid to be used. The combination of these factors will undoubtedly incur a performance hit as the number of threads increases.

2.4 MPMC Queue Performance

The benchmarking methodology follows what is laid out in Section 2.2. In Figure 2.2 an equal number of producers and consumers were created to act on the queue in question without affinity to any particular core. Each Queue here deals with storage differently, but shown here are the best results across a the same storage points chosen in Figure 2.1. The only unbounded design tested is the Microsoft concurrent queue which used its default provided resizing strategy.

One important facet to immediately point out about these results is how low all of the throughput numbers are compared to SPSC results, and how most all queues get worse performance when the number of Producers and Consumers increase. The first datapoint is the case of using the MPMC queue as a SPSC queue and we see that unlike how in Figure 2.1 with performance as high as 780 Million pushes and 780 Million pops per second, even the best queues here are at best an order of magnitude less than that result. One way to interpret this decrease even in a single producer single consumer setting is that the MPMC queues here don't know that no other threads are going to interact with the queue. This makes them have extra overhead to accommodate for the potential of new threads entering the queue which causes a large impact on performance, if they didn't go through this, then a thread interacting with the queue for the first time halfway through the test would break the queue. Then the performance of queues dropping as the number of threads increases is potentially due to the theoretical $\Omega(n)$ lower-bound, but a separate also applicable factor is that the synchronization variables used in each implementation decrease in performance as the number of threads increase as seen by the reduced performance of the Mutex Queue as the number of threads increase.

Additionally, many of the queues' performance drastically drops as the number of threads far exceeds the 16 concurrent threads supported by our benchmarking machine. For the Microsoft

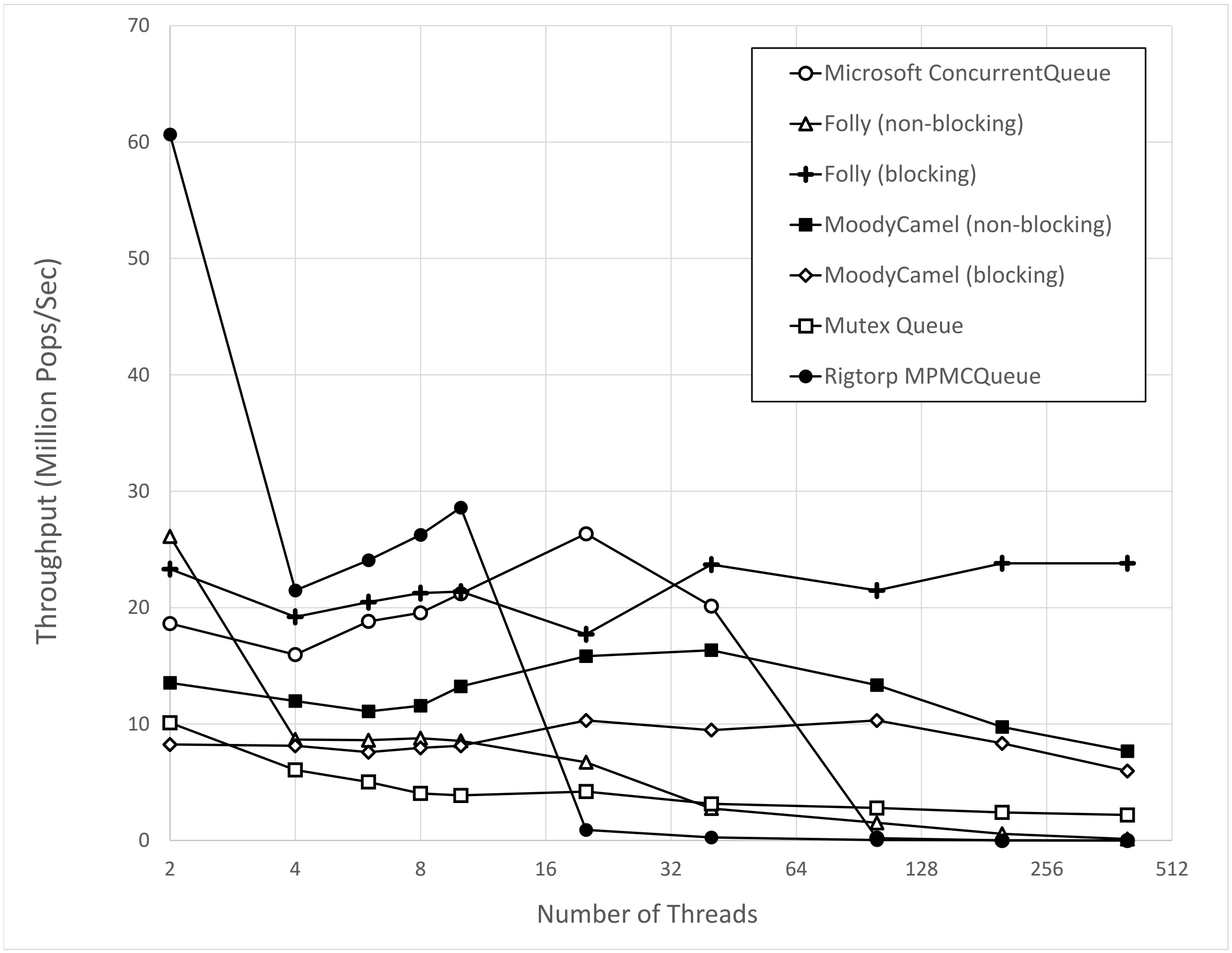

Figure 2.2: MPMC throughput vs queue size.

ConcurrentQueue this is likely due to frequent resizing which blocks pushes and pops from occurring until resizing is complete. For the MoodyCamel Concurrent Queue this is potentially due to a decrease of locality as increasing the number of threads linearly increases the storage used by the queue. The Rigtorp queue also has a steep drop-off as predicted. In the SPSC mode it far outperforms any of the other queue implementations as it is essentially a cautious SPSC Lamport Queue, but as the number of threads increase its contention of the head and tail and use of busy-spinning causes its performance to drop off as the number of threads start to outnumber the number of threads the CPU can concurrently execute.

One interesting performance is visible in the Facebook blocking vs non-blocking MPMC queue. The default retry mechanism in the benchmark is to effectively make each thread busy-spin on the queue making repeated operation accesses, this inherently can cause issues with scaling as it is possible for 16 consumers or 16 producers to be concurrently running on a empty or full queue which hinders throughput. One way to prevent this is by having a sophisticated retry mechanism which makes threads that cannot do work sleep to prevent them from hogging resources from other threads. The Facebook blocking method does exactly this and it allows it to have the most consistent performance of any queue tested here.

3. UNISON QUEUE

The main contribution of this work comes from the following observation from the analysis of past works. The fastest SPSC Queues have over ten times the throughput of the fastest MPMC Queues regardless of the number of threads used. This stands to reason that if one is able to make a MPMC queue that can operate as multiple SPSC queues with limited synchronization between them, then there is a large amount of throughput to be gained. Queues designs that provide linearizability inherently require a good amount of synchronization as proved by [5], but the same is not true for our local consistency model. Taking advantage of this will realize the aforementioned throughput gain.

3.1 Model

The strict definition of local consistency can be hard to reason with–in this section we will concretely show a difference between locally consistent and linearizable queues while additionally giving a simplified sufficient requirement to satisfy the formal local consistency definition. Additionally, blocking makes having deadlocks unavoidable in certain circumstances, here we introduce the idea of deadlock avoidance to address this.

First some background information is needed to more formally discuss the model and what is admissible in it. A produce-consumer queue is a shared object which supports the invocation of PUSH(X) and POP() operations. Producers are threads which can invoke push operations and consumers are threads which can invoke pop operations, for simplicity, we will first assume that no thread is both a producer and a consumer. Push operations will "insert" some element x to the queue, pop operations will return an element that had been "inserted" to the queue. For every element x which had been inserted to the queue, it can only be returned by at most a single pop operation. Operations done by a single thread are serialized by the linear order in which they occur. We shall say that one queue operation occurs before another on the same thread if the former operation precedes the latter operation in this time based serialization, this leads to a linear ordering on the set of each threads queue operations denoted as $\prec_t$ for thread t. An execution is loosely defined by a set of queue invocations by producers and consumers with a partial ordering $\preceq$ representing the parallel ordering of operations. Importantly, the partial ordering $\preceq$ respects the previously discussed linear thread specific ordering. A deadlock is then defined as an execution where the queue terminally stops returning to all outstanding PUSH(X) and or POP() operations. With these definitions, local consistency can now be better defined.

Definition 3.1.1 (Local Consistency)**.** *A producer-consumer queue is locally consistent if and only if the queue guarantees that in all executions, for any two threads p, c, if thread p pushed two elements x, y which were also popped by thread c, then x was pushed before y if and only if x was popped before y.*

A trivially locally consistent queue implementation is a queue which immediately deadlocks on any operation preventing the consistency condition from being violated. This implementation clearly has no throughput making it not of interest in our study, but to more strictly rule out the potential of avoidable deadlocks, an additional condition is imposed.

Definition 3.1.2 (Deadlock Avoidance)**.** *A bounded blocking queue is deadlock avoidant if and only if the queue satisfies the following for finite and infinite executions. First the queue only deadlocks in a finite execution if the number of pushes subtracted from the number of pops is negative or strictly greater than the size of the queue. Second the queue only deadlocks in an infinite execution if either the number of pushes or the number of pops is finite.*

The motivation behind this definition follows from the fact that a queue with blocking operations has unavoidable executions which will deadlock. If there are more pops than there are pushes then consumers will be blocked as there are no elements for the queue to give to blocked consumers. If there are more pushes than there are pops then the queue must store the additional elements, if the queue full then the queue must block the extra producers. In infinite executions, if one type of operation is bounded one of the previous cases will occur creating the same problem.

```
 1 int numThreads = 2;
 2 SynchronizationBarrier barrier(numThreads);
 3 QueueImplementation queue(2 * numThreads);
 4
 5 void work(int val) {
 6     queue.push(val);
 7     barrier.wait();
 8     queue.push(val);
 9 }
10
11 int main() {
12     CreateThread(work, 1);
13     CreateThread(work, 2);
14
15     int firstSum = q.pop() + q.pop();
16     int secondSum = q.pop() + q.pop();
17
18     if (firstSum != secondSum) {
19         // Queue Execution Not Linearizable
20     }
21 }
```

Listing 3.1: Linearizability vs local consistency.

Then because these are the only unavoidable deadlocks created by making the queue blocking, a deadlock avoidant queue would only deadlock in these situations.

One thing to note is that if a producer-consumer queue supports only one producer and one consumer, then it is identical to a linearizable queue. However, this is not true in general– a linearizable queue is trivially locally consistent, but the opposite does not hold. Listing 3.1 shows a concrete example of how local consistency is weaker than linearizability. In this example, a linearizable queue would guarantee that both sums are equal as the barrier ensures that both pushes on line 6 occur before the pushes on line 8. A locally consistent queue ignores the ordering between the two threads making the barrier effectively not exist meaning the queue may return any permutation of 1122 potentially changing the sums. While this example is trivial, it shows that using one locally consistent queue along with outside synchronization variables can lead to unexpected results because the queue may behave as if the outside variables were removed.

Even though at first glance local consistency gives the queue a lot of freedom in what elements consumers will receive when they pop an element from the queue–adding deadlock avoidance adds additional restrictions on what can be returned.

Lemma 3.1.1 (k-FIFO Decomposition)**.** *A bounded, blocking, locally consistent, deadlock avoidant queue is only correct only if for all pop operations in non-deadlocking executions, if x is returned to a consumer, where x was pushed by thread p, then x was at most $k - 1$ elements away from the minimal un-popped element specified by $\prec_p$ where k is the number of pop operations which were executing concurrently.*

Proof. For contradiction, assume that Q is a correct bounded, blocking, locally consistent, deadlock avoidant queue which does not satisfy the given condition. Consider an execution where the condition is violated, namely that k pop operations were executing concurrently, the list ordered by $\prec_p$ of elements pushed producer p but not yet popped is $(x_1, x_2, \dots, x_k, \dots)$, and a pop operation called by consumer c returned $x_m \mid m > k$. Note that in this execution, there is no way for x_k to be popped without additional invocations of pop operations. Therefore because the execution does not deadlock, there is an execution where no additional push or pop operations are called by any threads except not including thread c, all outstanding pushes and pops complete, and then consumer c attempts to pop all elements from the queue. If all the pops by thread c complete then it must have returned x_k which violates local consistency, if it doesn't return x_k then the execution must deadlock violating deadlock avoidance or an invalid value was returned as excluding x_k, there are no elements to return. Either way this contradicts the assumed correctness of Q. $\square$

This result means to satisfy the model, elements from each producer must be removed in a way where ordering deviates from a First-In First-Out (FIFO) by at most as the current number of calls to the queue. This amount of deviation may be as low as one, which leads to a natural implementation of a queue where all elements from each producer are consumed in FIFO order.

3.2 Designs

This section focuses on the two approaches taken which eventually led to our final designs. Even though our designs are focused on throughput, there are multiple methods described here and implemented in the appendix which have different pros and cons which are discussed in this section.

3.2.1 Helper Based MPMC Queue

Because we are dealing with implementing a locally consistent queue, drastically different design decisions can be made from many of the MPMC queues discussed in the prior section. Consumers solely need to read elements out in the same order that consumers pushed them in on a per-producer basis. Intuitively, this means that the Queue should keep track of what order Producers push items in, and conveniently, this can be done by giving each Producer a Single-Producer Zero-Consumer (SPZC) queue–a simple buffer where elements are added in to one-by-one. Symmetrically, Consumers can then be given a Zero-Producer Single-Consumer (ZPSC) queue buffer where they can read out elements in an order that doesn't violate the Consistency Requirements. Because each of these Queue Buffer will not empty or fill themselves, a Helper thread will be in charge of copying elements from a Producer Queue Buffer to a Consumer Queue Buffer.

These Queue Buffers are implemented as Lamport Queues as defined in Section 2.1. The SPZC queue will only partake in push operations meaning that the `tail` will get closer to the un-moving `head` stopping right before they touch. This allows the Helper to freely access all the pushed elements in the buffer between the `head` and the `tail`, give the elements to consumers, and then set the `head` to the `tail` effectively emptying the SPZC queue with a bulk pop operation. The ZPSC queue will similarly only partake in push operations removing elements between the `head` and `tail`. The helper can then freely copy in elements just after the `tail` before the `head`, and set the `tail` to the end of the new elements effectively filling the new queue with a bulk push. These bulk operations are advantageous to the SPSC push/pop operations from before as they occur independent of each other and in completely different parts of memory reducing the potential for false sharing on the buffer, and additionally happen much less frequently reducing sharing of the `head` and the `tail` pointers. A similar process can be taken to implement the queue using FastForward queue buffers as defined in Section 2.1. Note that both the Lamport and FastForward based ZPSC and SPZC queue designs avoid the frequent false sharing of variables present in SPSC Lamport queues because only one thread frequently acts on each queue.

```
class HelperBasedQueue {
  Semaphore helpProducerTrigger(0,1); // initalized to 0 with max size 1
  Semaphore helpConsumerTrigger(0,1);
  vector<ProducerQueue> producerQueues;
  vector<ConsumerQueue> consumerQueues;
  vector<Semaphore> producerSemaphores;
  vector<Semaphore> consumerSemaphores;

  void helperRoutine() {
    while (true) {
      helpProducerTrigger.wait();
      // find an input queue with non-empty elements
      for (ProducerQueue &prodQueue : producerQueues) {
        int copyAmount = prodQueue.size();
        offloadItems(prodQueue, copyAmount);
        producerSemaphores.release();
      }
    }
  }

  inline void offloadItems(ProducerQueue &prodQueue, int copyAmount) {
    static int lastConsumer = 0;
    while (copyAmount > 0) {
      helpConsumerTrigger.wait();
      for (int offset = 0; offset < consumerQueues.size(); offset++) {
        int curConsumerIndex = (lastConsumer + offset) % consumerQueues.size();
        ConsumerQueue& curQueue = consumerQueues[curConsumerIndex];
        int freeElements = curQueue.capacity() - curQueue.size();
        curQueue.bulk_push(prodQueue.bulk_pop(freeElements));
        copyAmount -= freeElements;
        consumerSemaphore.release();
        if (copyAmount == 0) {
          lastConsumer = curConsumerIndex;
          break;
        }
      }
    }
  }
};
```

Listing 3.2: Helper routine.

The general process goes then as follows. The Producer pushes elements into a Bounded Single-Producer Zero-Consumer Queue Buffer–if the Queue Buffer is full, then it will signal the helper that a producer needs help, and wait on a Semaphore until it is released by the Helper. The Consumer does the same except it signals the helper that a consumer needs help when its queue is empty. The helper then continuously runs the routine given in Listing 3.2 waiting for producers to need help, to then wait for consumers to need help and then finally moving elements between the two groups when they're both ready.

There's a lot of benefits to this method which makes it perform well despite its simplicity. The biggest factor is the reduced amount of synchronization, in a linearizable MPMC queue each thread normally would do some amount of synchronization with all other threads on each push and pop operation. Here each thread can fill or empty their entire buffer with no synchronization at all. Additionally, it only synchronizes with one party, the helper, and this thread doesn't do more synchronization as the number of producers and consumers grow. The main bottleneck then arises from two problems. The Helper itself is required to move all elements that make its way through the queue, capping the overall queue speed at the helper's speed. Then additionally, the elements need to be written to a producer buffer, read by the helper, written to a consumer buffer, and then read by a consumer, this overall doubles the amount of memory accesses from what a normal queue would have which implies there may be more room for improvement.

3.2.2 Helper Based SPSC Queue

Just because the previous queue supports multiple producers and consumers does not mean that it has to be used in that way. Interestingly, despite the previous design effectively operating as two Lamport Queues with an intermediary thread when used as an SPSC Queue, it performs significantly better than the previously benchmarked Lamport Queue. Revisiting the benefits of the Helper Based Queue in general explains this difference as the benefit of significantly isolating the Producer and Consumer reducing the number of times they share memory.

With a small design change we can change the Producer and Consumer to push and pop from the same memory buffer. In this queue, the Producer and Consumer each have their own set of `head` and `tail` variables which are local to themselves. When one thread gets blocked from the queue being either full or empty, it will then access the helper to give it more elements as normal, but in this design, the helper can skip the step of copying over elements as the elements are already written or read by the producers and consumers themselves. This dramatically increases the speed of the helper routine, and because both threads wait on the helper to push or pop elements, this removes what previously was a large bottleneck.

This then gives this queue altercation the following characteristics. The Producer and Con-

```
class SPSCUnisonQueue {
  Semaphore helpProducerTrigger(0,1); // initalized to 0 with max size 1
  Semaphore helpConsumerTrigger(0,1);

  void push(int element) {
    const int nextTail = (localTail + 1 == queueSize)? 0 : localTail + 1;
    while (true) {
      for (int retries = 0; retries < NUM_RETRIES; retries++) {
        if (notFull(nextTail,headEstimate)) {
          buffer[localTail] = element;
          volatileTail = localTail = nextTail;
          return;
        }
        // Update head estimate to see if space has been freed
        headEstimate = volatileHead;
      }
      consumerSema.release();
      producerSema.timed_wait();
      headEstimate = volatileHead;
    }
  }

  int pop() {
    const int nextHead = (localHead + 1 == queueSize)? 0 : localHead + 1;
    while (true) {
      for (int retries = 0; retries < NUM_RETRIES; retries++) {
        if (notEmpty(localHead,tailEstimate)) {
          const int currentElement = buffer[localHead];
          volatileHead = localHead = nextHead;
          return currentElement;
        }
        // Update tail estimate to see if elements have been added
        tailEstimate = volatileTail;
      }
      producerSema.release();
      consumerSema.timed_wait();
      tailEstimate = volatileTail;
    }
  }
};
```

Listing 3.3: SPSC Unison Queue

sumer act on a the section of elements bounded by their local `head` and `tail`, compared to the original Lamport Queue where these values are accessed and updated across threads on every operation, these reads and writes are delayed in this design to only occur when threads are blocked. Additionally whereas the Lamport Queue provides non-blocking operations, this queue makes the Producer and Consumer sleep on a semaphore until they can successfully complete work.

3.2.3 Unison SPSC Queue

The helper based SPSC design has the nice property of delaying the reads and writes of the head and tail variable between threads, but it relies on a separate helper thread to accomplish this. Ideally, removing this outside party would increase the performance of that previous queue design further. Listing 3.3 shows a design that accomplishes the delaying of reads. This queue has accurate `volatileHead` and `volatileTail` variables which are shared between the two threads. The **`volatile`** keyword in C++ incentives the compiler to not always cache values of variables as it may change from other threads. The Microsoft Visual Studio compiler actually goes one step further and gives virtual variables acquire and release semantics [13] meaning that all reads and writes will both access the variable directly and not be reordered with other reads or writes. This is important as it means updates to the volatile variables will quickly be visible between threads and because each volatile variable is modified by only one thread, the performance impact of using volatile is largely mitigated. Then to delay reads of the volatile variables, each thread has local variable estimates of the head or the tail which can then be read much faster than their volatile counterparts. keyword equal to their local counterparts which don't have delayed writes to them, but the threads delay their access of them by relying on their head and tail copies until they are blocked.

Because the volatile variables are efficiently updated between threads, adding the semaphore sleep may seem unnecessary, but it serves a dual purpose. First, this prevents a thread from permanently busyspinning on an empty or full queue, adding a sleep will free the resources the thread was using once the thread is convinced that it is sufficiently blocked. Second, aside from true balanced concurrent pushing and popping from the queue, one of the most efficient modes of this queue is having the producer wake up and fill the queue, then wake up the consumer to empty the queue, who then wakes up the producer to repeat the cycle. The order of the semaphores effectively allows this hand-off mechanism to take place. This sleep wake cycle then also has the property that the variable estimates will be guaranteed to be up to date as the semaphore implicitly calls LOCK prefixed assembly instructions which by the Intel and AMD development guides [8, 7] then means that when the thread wakes up it will necessarily see the most up to date state of the queue. The benefits of both of these effects will be seen in the following MPMC design.

This idea of having local estimates for global variables doesn't directly apply to other methods like the Fast Forward queue as the only variables needed to estimate are the queue itself, since the current queue element defines what work can be done, there is no point to save a local snapshot of the current element to know if the element can be popped or not.

3.2.4 Unison MPMC Queue

Returning to the MPMC design ideas, there is one clear improvement to the Helper MPMC design. The Helper design relies on an intermediary thread to copy elements from producers to consumers, this gave benefits for performance which was then replicated in the Unison SPSC queue without needing the helper or copying elements. A similar approach can then be made to remove the helper and copying from the MPMC helper queue design as well.

The basic design idea is that instead of having threads push or pop from SPZC or ZPSC queues, threads can be joined on Unison SPSC queues which have many benefits of the SPZC and ZPSC separation but without using copying or a helper. If there are an equal number of producers and consumers then the question of how many SPSC queues to create is simple, but then what should occur if there is an imbalance in the number of producers and consumers? One solution would be to have as many SPSC queues as the larger group, and then let the smaller group access multiple SPSC queues. This solution turns out to be problematic, as if one producer pushes in multiple SPSC buffers then it is easy to have a consumer read elements out of order which then can lead to a violation of Lemma 3.1.1 making the queue deadlock or violate local consistency. To remedy this, the proposed design gives each producer a SPSC queue, and this then guarantees that each producer's elements will be read in FIFO order preventing the previously mentioned Lemma from being violated. Then going beyond just satisfying the Lemma, it actually provides a locally consistent deadlock avoidant blocking queue because the SPSC Unison queue provides this, and the property is then inherited by this new construction.

4. RESULTS

The different designs laid out in Chapter 3 are now benchmarked and analyzed in this chapter. Note that the benchmarking methodology is identical to what is laid out in Section 2.2, thus direct comparisons for how queues preform in the benchmarking setting can be made.

4.1 SPSC Queue Performance

Going back over the SPSC results from Section 2.2, we see that most of the queues had throughput under 600 million elements per second regardless of the size of the queue. The only exception to this was the Fast Forward queue which reached a throughput of 780 million elements per second once the queue could hold 2^{20} (around one million) elements. While these numbers are impressive, given that the MPMC Unison design is built off of several SPSC queues, making each SPSC queue use one million elements will quickly consume extremely large amounts of memory.

Recalling that the new SPSC queue designs as well as the previous Fast Forward design avoid many of the pitfalls of the Lamport queue, it would be reasonable to expect the new designs to have similar throughput to the Fast Forward design. Looking at Figure 4.1 confirms this, and additionally shows that the new designs actually higher speeds using far less memory.

Focusing on the Unison SPSC queue implementations, we see that at smaller queue sizes, busy-spinning achieves higher throughput. This effect lessens and then even reverses as the size of the queue increases. As mentioned in the previous section, using the semaphore to wait when a thread is blocked can lead to a pattern where threads empty or fill the buffer and then sleep. For smaller queue sizes this is undesirable as each buffer fill or empty contributes less to the overall queue throughput, but at larger sizes, the larger number of elements per sleep lessens the impact of sleeping on throughput. Additionally, at extremely large queue sizes, the semaphore also helps separate which part of memory each thread accesses. If the two threads start accessing the same portion of memory, this is because either the producer or consumer has done operations so quickly that it caught up to the other relatively slower thread meaning the queue is nearly full or nearly

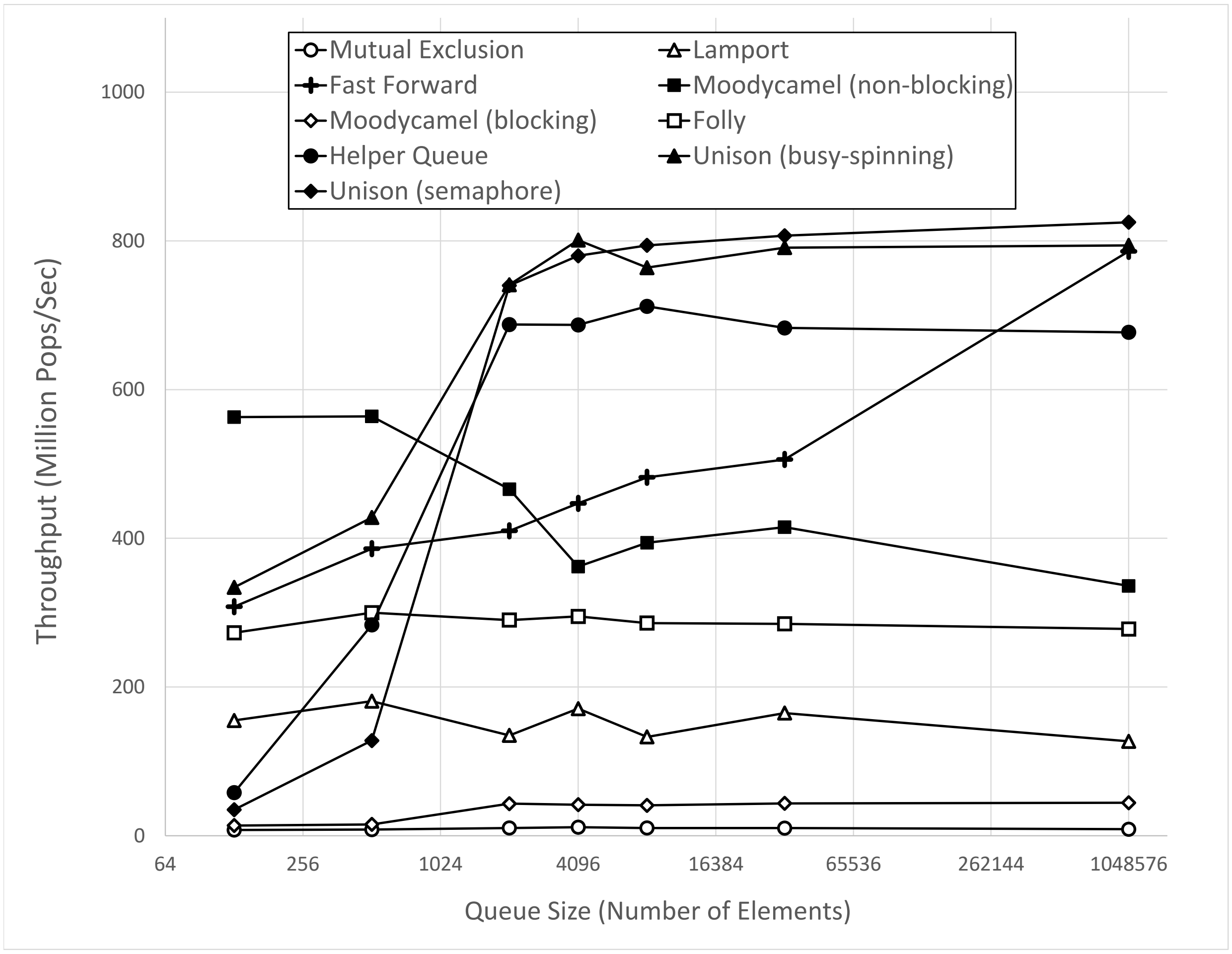

Figure 4.1: SPSC throughput vs queue size.

empty. After this occurs, it is extremely likely for the fast thread to get blocked as threads sleep once their queue estimates indicate the queue is either full or empty. For the faster thread to not get blocked in this state, both threads must somehow equalize to the same speed, and then the previously faster thread must repeatedly and indefinitely get correct estimates on most all queue operations. This is unlikely to happen for extended periods of time, so once the faster thread gets blocked, the slower thread then can continue working on the queue, distancing itself from the blocked thread which temporarily remains in the same portion of memory.

Another interesting result is that the SPSC Helper queue's performance follows a similar trend as the two Unison queues. This is surprising as the Helper queue delays both reads and writes

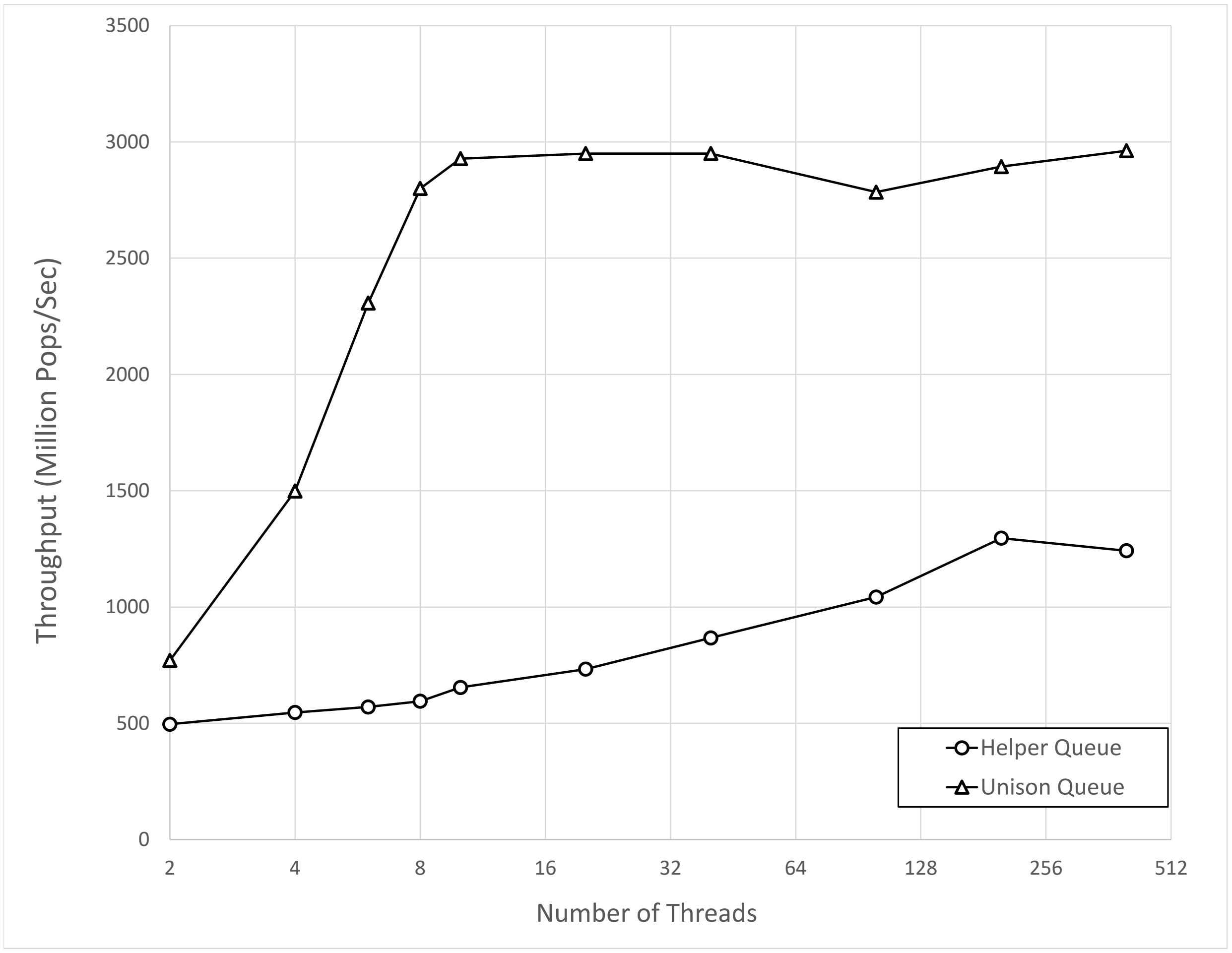

Figure 4.2: MPMC throughput vs queue size.

to queue variables while the Unison designs only delay reads of queue variables. The absolute difference in performance can likely be explained as a result of removing the helper thread, but potentially there is more performance to be gained by a helper-less design that delays both reads and writes.

4.2 MPMC Queue Performance

Now moving on to MPMC performance we can see how well the two previous methods do in the new setting. Figure 4.2 shows the impressive speeds of the two MPMC queue designs proposed in the previous chapter.

Comparing the two trends some important visual features are apparent. The y intercept of the charts show the two queues running with a single producer and a single consumer. As discussed in Section 2.4, linearizable MPMC queues had a severe performance hit compared to their SPSC counterparts due to the risk of new threads entering the system during execution. Here we can see that the performance of the locally consistent Unison queue is impacted much less, nearly matching the performance shown in Figure 4.1. The reason why the Helper queue's speed is decreased from Figure 4.1 is because the SPSC helper does not need to copy elements from producer to consumer, but the MPMC helper does decreasing performance. Then both queues scale quite well with threads. Many of the queues studied in Section 2.4 decreased in speed as the number of threads grew, but the Helper queue's performance increases even as the number of threads far exceeds the number of execution cores on the system. The MPMC Unison queue's performance scales nearly linearly up to 8 threads, however this performance then tops out at around three billion elements per second. Recalling that for memory updates from one core to become visible to another core, the updates must travel through L3 cache gives the possible explanation that hardware is bottlenecking the performance of the MPMC Unison queue. While memory updates within one core can be delivered using the CPU's much faster L2 cache, the size of this cache is extremely small meaning the best performing SPSC queues from the previous section are unable to bypass the possible L3 cache speed bottleneck.

Despite the potential for even greater throughput, the current speeds are a large improvement on what was studied in Chapter 2. Even only using two threads, the locally consistent Unison queue has an order of magnitude more throughput than any of the studied MPMC queues. For situations where high throughput work coordination is necessary, these designs allow enable queue based coordination to be used in situations where older designs would be several times too slow. However, in the search for pure throughput, some nice properties have been lost, Chapter 5 discusses these issues and their potential workarounds in detail.

5. FUTURE WORK

5.1 Local Consistency vs Sequential Consistency

The local consistency condition given in Definition 3.1.1 is a property strictly applying to queue operations. Its motivation was based around the idea that the common linearizable consistency condition is quite strong, and it could potentially lower the potential throughput that a queue is capable of. Even though local consistency successfully sidesteps the creation of a linearizable queue, there are more general consistency conditions for any type of shared object which are weaker than linearizability, and there is the open question of whether local consistency is equivalent to one of these more general conditions.

The consistency condition that seems most similar to local consistency is *sequential consistency*. This condition is defined in [15] as the property where an execution's operations can be arranged into a linear order which respects sequential queue semantics as well as the ordering of operations by individual threads. The distributed computing community has studied distributed sequentially consistent queues in works such as [16]. The throughput of these designs were not studied as this paper focuses on parallel environments, but in theory it may be possible to use the designs here in distributed environments as well. Additionally in general it is unclear how the strength of sequentially consistency and local consistency compare, if more work is done then perhaps the two notions could be combined.

5.2 Lazy Consumers

The lazy consumer problem was briefly touched upon in Chapter 3. This issue is caused by a consumer being allocated some amount of elements to read, but the consumer becomes lazy and ends up not reading them. The design of the Unison MPMC queue is to have a producer take exclusive control of a SPSC queue. If a consumer truly becomes lazy, then a third party, such as a helper thread, would need to give the exclusive control of the queue to a non-lazy thread. However, because a queue only interfaces with threads through push and pop methods, it is not possible for the queue to know if a thread has been inactive for an extended period of time as opposed to whether it has become lazy forever. If at any point in time the queue decides a thread is lazy and it gives the queue to a separate consumer, there is the possibility of the assumed lazy attempting to pop an element from the MPMC queue. In this situation, the consumer should be blocked from popping from its queue, and in order to get speeds equivalent to the SPSC queue by itself, threads cannot easily check synchronization variables to block its access to the queue. Even hardware clocks do not have the throughput to be checked on each push or pop operation for a thread to know if it itself had been lazy.

There are ways to remedy the lazy consumer problem, the simplest method is to have a helper suspend the thread which is suspected to be lazy, and if it is not in the middle of popping, then give its queue to another thread. This is problematic because even though a thread may be lazy from the perspective of the queue, it may be doing important work somewhere else. In a real time environment, this suspension could break timing guarantees by the system. Another way to address this problem is to have threads formally announce they are going to be leaving the queue for some period of time, and as part of this announcement it can return its queue for other threads to use. This is problematic as it gives the programmer an additional responsibility to ensure that each thread announces their departure to each queue that it may use during the course of an execution. A third method to approach this problem is to use the concept of zero cost exceptions to force lazy consumers to have an exception on their next pop, give the consumers' queue to another thread, and if the lazy consumer returns, it will have to handle the exception, part of which will involve find a new queue to attempt to pop from. This is similar to the suspension method as the exception will slow the thread down so significantly that it may as well be temporarily be suspended. However, the difference lies in the fact that it would be temporally stopped from making progress during a queue operation. Because the queue operations are blocking, this may have occurred even without the exception handling, so this approach doesn't create a new problem. There may be other solutions which create an even smaller slowdown which can be studied in the future.

5.3 Load Balancing and Starvation

One common benefit of using a linearizable queue is that if a element is pushed into the queue, then the element will be read from the queue in a finite number of pops. This is related to the latency of a queue, a factor not considered in this work. Additionally, the MPMC Unison queue is composed of several blocking SPSC queues, if a producer's element is not popped in a finite amount of pops, then this means that the producer's queue is not being popped from. This can lead to a queue becoming filled, permanently blocking the producer in a state known as starvation. This makes the issue of latency also manifest as starvation in the MPMC Unison queue design.

This issue can also occur in reverse where consumers are starved from the design choice that there are only as many SPSC queues as there are producers. Because SPSC queues can only support one consumer at once, it is possible for a consumer to starve other consumers by being very productive on its queue. Because productivity is related to throughput and the goal of this work was to maximize throughput, these behaviors were not prevented in this work. This means the current way to incur the lowest latency and starvation is to use a similar number of producers and consumers, if one group outweighs the other by a significant margin then it is almost guaranteed that some number of threads will be starved by the most productive threads in the group which also can increase latency.

Both of these problems can be solved, although likely at the cost of giving up some amount of performance. To help with load balancing, threads can rotate between queues after completing up to N operations on their current queue. However, this would mean that multiple threads would frequently be switching queues, and because each SPSC queue only supports one consumer and one producer, each producer and consumer would need to frequently synchronize on who gets which queue. Additionally checking if they should rotate queues on each pop operation will incur another branch miss on each operation, and require another variable to be touched on each operation which can severely impact the performance of the highly optimized push and pop methods. Latency would be helped from these rotations, but it is also unclear if this simple rotation mechanism would prevent starvation or not.

REFERENCES

[1] E. W. Dijkstra, *Cooperating Sequential Processes*, pp. 65–138. New York, NY: Springer New York, 2002.

[2] C. M. Kirsch, M. Lippautz, and H. Payer, "Fast and scalable, lock-free k-fifo queues," in *Parallel Computing Technologies* (V. Malyshkin, ed.), (Berlin, Heidelberg), pp. 208–223, Springer Berlin Heidelberg, 2013.

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

[4] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, p. 463–492, July 1990.

[5] F. Ellen, D. Hendler, and N. Shavit, "On the inherent sequentiality of concurrent objects," *SIAM Journal on Computing*, vol. 41, pp. 519–536, Jan. 2012.

[6] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst.*, vol. 5, p. 190–222, Apr. 1983.

[7] A. Corporation, *AMD64 Architecture Programmer's Manual Volume 1:Application Programming*. AMD, 2020.

[8] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2020.

[9] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, (New York, NY, USA), p. 43–52, Association for Computing Machinery, 2008.

[10] F. Inc., "Folly: Facebook open-source library." `https://github.com/facebook/folly`, 2020.

[11] Cameron, "A single-producer, single-consumer lock-free queue for c++." `https://github.com/cameron314/readerwriterqueue`, 2013.

[12] Cameron, "An industrial-strength lock-free queue for c++." `https://github.com/cameron314/concurrentqueue`, 2020.

[13] Microsoft, "Microsoft .net 5.0 api documentation." `https://github.com/dotnet/dotnet-api-docs`, 2020.

[14] E. Rigtorp, "Bounded multi-producer multi-consumer concurrent queue written in c++11." `https://github.com/rigtorp/MPMCQueue`, 2020.

[15] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, p. 91–122, May 1994.

[16] M. Feldmann, C. Scheideler, and A. Setzer, "Skueue: A scalable and sequentially consistent distributed queue," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1040–1049, 2018.