University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

High Throughput Queue Algorithms

Daniel Orozco[1] [2]
Elkin Garcia[1]
Rishi Khan[2]
Kelly Livingston[1]
Guang Gao[1]

CAPSL Technical Memo 103
January 14th

[1] University of Delaware and
[2] ET International
{orozco, egarcia, livingst, ggao}@capsl.udel.edu, rishi@etinternational.com

Abstract

Advanced many-core CPU chips already have few hundreds of processing cores (*e.g.* 160 cores in an IBM Cyclops-64 chip) and as computer architecture progresses, more and more processing cores become available. The underlying runtime systems of such architectures need to efficiently serve hundreds of processors at the same time, requiring all basic data structures within the runtime to maintain unprecedented throughput. In this paper, the design of concurrent queues is explored to meet the runtime system and algorithm demands for hundreds of simultaneous transactions being handled in real time.

We explore some of the basic standard queueing techniques, showing their weaknesses in a highly parallel environment. We then develop a simple high throughput queue. The algorithm is then extended to address all cases and to provide all the functionality commonly found in queues. We test all queueing algorithms on Cyclops-64, a many-core processor by IBM.

The following are the major contributions of this paper:

1. Queueing theory is used to provide a mathematical background for the inherent throughput of queueing algorithms.
2. The Circular Buffer Queue algorithm: A very high throughput, low latency queue implementation useful for most common cases.
3. The High Throughput Queue algorithm: A high throughput, low latency queue implementation that supports all traditional operations on a queue.
4. Experimental results that show that for highly parallel systems, both the Circular Buffer Queue and the High Throughput Queue allow larger scalability and higher performance than other famous, state of the art implementations.
5. We show that the queue algorithms presented are linearizable and behave like non-blocking, wait-free algorithms for practical implementations.

1 Introduction

New challenges in queue algorithm design are posed by the development of many-core architectures: Algorithms must support several hundreds of processors concurrently. There are existing algorithms that correctly allow access to queues by an arbitrary number of processors. However, correct access does not necessitate efficient access, and as shown throughout the paper, even the best non-blocking algorithms have a low amount of concurrency when used in an environment with hundreds of simultaneous requests. The issues have radically changed from execution in an environment dominated by virtual parallelism to an environment dominated by massive hardware parallelism. Although the new requirements of efficiency and parallelism seem challenging, we, however, were able to provide a surprisingly simple intuition that allows development of faster, more parallel, higher throughput queues.

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

The intuition behind the solution can be conceptually explained by an analogy between old-style train stations and queue algorithms: The enqueue process is analogous to passengers boarding a train. Traditional queue algorithms are in a way or another variants of passengers trying to serially enter the train through a main entrance. The algorithms presented here follow a different approach: passengers (processors) can obtain a ticket (an integer) and then board (enqueue) a train (the queue) directly to their seat (array location) in parallel.

Two concurrent queue algorithms, the Circular Buffer Queue and the High Throughput Queue algorithm, are presented and fully developed in this paper. Throughput is the design constraint. First In First Out (FIFO) behavior is provided by both algorithms. Throughout the paper, it is shown that the algorithms meet important properties such as linearizability [5] and behave for all practical purposes as wait-free [4] and nonblocking [9] implementations.

The development of fast concurrent queues is important because parallel applications frequently rely on concurrent queues to perform a variety of tasks such as scheduling, work distribution, graph traversing, etc. Previous work on concurrent queues [8] in the past decades suggested that using locks was sufficient or even better than other non-blocking algorithms to implement parallel queues. Our experiments provide evidence showing that locking algorithms are very slow when a large number of processors attempt to use the queue at the same time. We show that many popular non-blocking algorithms for concurrent queues are also slow because they rely on the Compare and Swap operation, which does not guarantee success for every single processor.

Queueing theory is used to reason about the performance of a queue algorithm. Latency of individual queue operations has been considered the most important factor from the point of view of the application, but, it does not consider the fact that latency is a direct consequence of the intrinsic throughput of a queue. Extensive research in the field of queueing theory [6] has shown that latency greatly increases when the available queue throughput is not sufficient to serve the incoming requests. In the past, the throughput of a queue was not an issue, since it was very rare that enough requests could be provided to saturate the throughput of a queue. For example, only 12 processors were used to test the performance of the famous concurrent queue implementation by Michael and Scott (MS-Queue)[9]. Other papers provide results for 32 or 64 processors [10, 12], but they fall short of reaching the limit set by the intrinsic queue throughput. However, as technology advances, more and more processors are used for computations stressing the need for concurrent algorithms that can theoretically scale to thousands or millions of processors.

The algorithms introduced in this paper are different from other implementations because they avoid the *inquire-then-update* approach when possible. The difference is significant: Other algorithms using the inquire-then-update approach (such as the MS-Queue algorithm or a spin lock implementation) require at least two full memory roundtrips to complete an operation. We follow an alternative path: Every single memory operation should be guaranteed to succeed if possible. The core of our implementations relies heavily on the Fetch and Add operation which always succeeds to change the state of the queues with *one* memory operation. Although the design of the High Throughput Queue has operations of the type inquire-then-update, the

occurrence of retrying an operation can be made arbitrarily low at the expense of a reasonable amount of memory.

The Circular Buffer will always complete queue operations successfully, waiting for queue elements or enough memory as needed. The High Throughput Queue allows all the functionality of traditional queues, including (1) inquiring about whether or not a queue is empty and (2) allocating more memory if it is full.

Experimental results shown in Section 5 show that our queue algorithms have better latency, scalability and throughput than the famous MS-Queue [8] (which is used in the Java Concurrent Class), and the common single-lock algorithm. We also compared our queueing algorithms to a high throughput algorithm by Mellor-Crummey (MC-Queue) [7] and we have found that our algorithms use less memory, have lower latency and equal or greater throughput.

In general, when the queue is designed properly, any queue operation will complete in a bounded, low amount of cycles. Although from a purely academical point of view, no strict guarantee of this statement is possible, in practice, it is possible to bound the time required to complete each queue operation if a bound for an individual memory operation is known, *regardless of the number of processors simultaneously accessing the queue.*

The performance of the Circular Buffer and the High Throughput Queue algorithms is discussed in detail in Section 5. The experiments show that the throughput of queue operations is only limited by the bandwidth of the memory controllers, which provides a much wider range of scalability than other approaches such as the MS-Queue or traditional queues based on locks.

The rest of the paper is organized as follows: Section 2 provides background on the properties of queues, Section 3 formally introduces throughput as an important design parameter of queues and Section 4 presents our high throughput queue algorithms. Experimental results showing the advantages of our implementations are shown in Section 5. The paper finishes with discussion on our findings and conclusions in Sections 6 and 7.

2 Background

This section provides a theoretical background on the concurrency properties of FIFO Queues.

The wide body work on implementations of FIFO Queues usually classify queue algorithms as either *wait-free* [4], *nonblocking* [9] or *blocking.*

Wait free implementations guarantee that each operation will complete in a bounded amount of time regardless of the state (or failure) of other processors. Wait-free implementations provide the strongest guarantees about performance, but in practice, they exhibit a lower performance than other implementations [4]. The introduction of Obstruction Freedom [3] partially alleviates this problem, but it requires support from the user application and is outside of the scope of this paper.

Nonblocking implementations of queues guarantee that at least one processor will finish

its operation in a bounded amount of time, but it provides no guarantees about starvation of individual processors.

Blocking implementations of queues do not guarantee that an operation will succeed in any finite amount of time. Many blocking implementations perform better under low contention than similar wait-free or nonblocking implementations.

The current paradigm about concurrent objects dictates that nonblocking implementations are the best choice to achieve high performance for queue operations. Several papers have shown that the very popular MS-Queue [8] and nonblocking queues variants of it [10, 12] based on the "Compare and Swap" operation provide a reasonably good performance in systems with 12, 32 or up to 64 processors. The reasonably good performance of the MS-Queue is well accepted in the community, and the algorithm is used, for example, as part of the kernel of the Java Concurrent Class. However, no evidence has been shown that the approach of the MS-Queue will continue to prove effective for future parallel systems with thousands or millions of processors.

The Wait-Free and Nonblocking properties are defined assuming arbitrary processor speeds, including a speed of zero instructions per second, or under the presence of hardware faults. These assumptions are not very useful for non-preemptive systems in the absence of thread failures. Such constraints, in general, do not pose a drawback since many systems such as Cyclops-64 [2] support disabling interrupts to allow truly non-preemptive execution of regions of code and in many cases, hardware failures take precedence over wait-free or nonblocking considerations.

The Circular Buffer and the High Throughput Queue algorithms are related to the wait-free and non-blocking properties in practice; queue operations complete in a bounded amount of time. However, from an academical point of view, they do not strictly meet the conditions to be classified as wait-free or non-blocking.

Linearizability [5] is a property related to the semantics of the operations. The concept of linearizability depends on the meaning of the operation, and in general, linearizability can be explained as a guarantee that concurrent operations appear to happen at an instantaneous point in time. Both the Circular Buffer and the High Throughput Queue are linearizable.

3 The problem with existing concurrent queues on many-core architectures

Queuing theory is used to support the main idea of this section: *queue throughput* is the single most important aspect of a queue in a large parallel system. This is a result of an intuitive performance requirement: In a parallel program, queue operations should have low latency.

The theoretical exposition is founded by the fact that the latency of a queue operation is mostly composed of two factors: The time necessary to access the queue over the network and the time taken by the queue to service the request. In highly parallel systems the time to

complete the request can be much larger than the network latency if there are many queue requests pending. For systems with a large number of processors, having few or no pending requests in the queue is of utmost importance: The total time to complete a queue operation will increase with the number of pending requests. Existing research in queueing theory [6] states that when the latency at the queue is low, the latency in the queue is proportional to the number of processors using the queue and inversely proportional to the total throughput of the queue squared.

Our studies are focused on non-preemptive systems, and they are motivated by recent developments in computer architecture, where the increased number of hardware threads allows the development of complete non-preemptive systems such as IBM Cyclops-64 [2]. The absence of an operating system, the ability to disable interrupts and the absence of a hardware managed cache on such many-core systems simplify our analysis of algorithms making statistical models unnecessary as would be the case for other architectures.

Common queue algorithms under non-preemptive systems are analyzed in detail on Subsections 3.1 to 3.4. The limitations of those queues in many-core architectures are exposed in subsection 3.5, allowing a formal statement of the problem to be solved.

3.1 Queue latency and throughput for large parallel systems

The following definitions are used in the theoretical analysis:

- P is the number of processors accessing the queue.
- r is the average amount of time taken between calls to queue operations by user code.
- μ is the *throughput* of the queue: The number of requests serviced per unit of time.
- λ is the average request arrival rate to the queue. When the latency at the queue is low, λ can be approximated as $\lambda \approx P/r$
- m is the average latency of a single memory operation.
- k is the amount of time (measured in cycles) that an atomic operation uses at the memory controller.
- z is the number of cycles a memory read or write uses the memory controller.
- N is the average number of pending requests in a queue.
- L is the average waiting time (latency) before a queue request is served.

The latency caused by calls to the queue has primarily two components: 1) The time taken by the network to deliver the request to the queue. 2) The time to complete the request. Modifying the first component is mostly a computer architecture problem and it is outside of

the scope of this paper. We will focus on minimization of the average time taken to serve a queue request.

The latency and the performance of the queue can be analyzed using the well-established field of queueing theory where queues are classified according to the probability distribution functions (PDF) of queue requests, the PDF of requests served, and the number of servers. Although the strict model of our queue is more complex, we use a reasonable approximation that focus on analyzing the bottlenecks of execution. The arrival intervals of requests from the users is approximated by a random Poisson process (this approximation is used in many fields and it is well studied [11]) and the service rate is approximately deterministic since the queue will serve requests as soon as possible at its maximum rate.

A queue with a poisson arrival rate, a deterministic service rate and one server is referred in the queuing theory literature as an $M/D/1$ queue [6]. It is proved that the average number of pending requests in a queue (N) grows asymptotically when the average queue request rate (λ) approaches the throughput of the queue (μ) as shown by Eq. 1 [6].

$$N = \frac{\lambda}{(\mu - \lambda)} \tag{1}$$

In situations where $\lambda \approx \mu$ (Eq. 1) the latency of queue accesses will increase due to increased number of pending requests at the queue. The queue will have few pending requests only when $\lambda << \mu$. Under the condition $\lambda << \mu$, Eq. 1 can be approximated by Eq. 2.

$$N \approx \frac{\lambda}{\mu} \approx \frac{P}{r}\frac{1}{\mu} \tag{2}$$

Average latency at the queue is related to the number of average pending requests at a given time. The intuition behind the latency at the queue is that if N requests are pending, it would take N/μ (Equation 3) units of time to get to the front of the queue.

$$L \approx \frac{N}{\mu} \approx \frac{P}{r}\frac{1}{\mu^2} \tag{3}$$

Eq. 3 shows that low latency is due to either a high throughput, or a low request rate by the processors. The request rate depends of the application so we are focusing on *achieving low latency by designing a queue with high throughput.*

The following sections present an analysis of several common queue algorithms.

3.2 Single Lock Queue

The algorithm followed by processors in a single-lock queue implementation is: (1) obtain a lock, (2) read the queue pointer, (3) update the queue structure, (4) release the lock. Figure 1 shows the data structure commonly used to implement this queue.

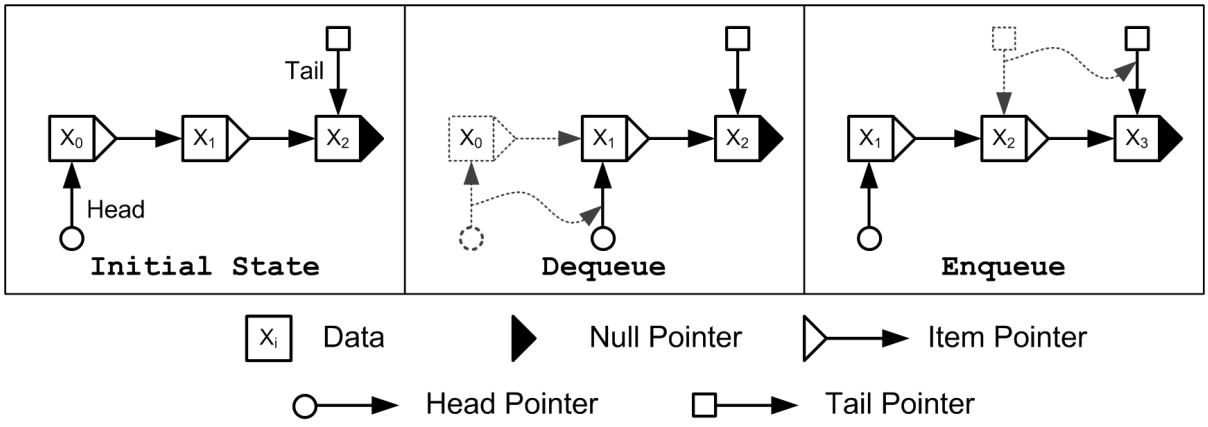

Figure 1: Traditional view of a queue

Completion of a queue operation takes at least 2 complete roundtrips to memory, even with optimal pipelining and scheduling (half a round trip to obtain the lock, 1 round trip to read the queue structure and half a round trip to update the queue and release the lock). Accordingly, the throughput of the single lock queue is:

$$\mu \approx \frac{1}{2m} \tag{4}$$

The analysis of the Single Lock queue and other subsequent analysis assumes that control flow instructions and other local instructions executed at the processor take very little time when compared to the memory latency.

Practical implementations of the Single Lock queue usually have a much lower throughput because optimal scheduling and pipelining are difficult due to library calls, or because thread preemption can not be disabled.

3.3 MS-Queue

The queue algorithm by Michael and Scott, commonly referred to as the MS-Queue [8], is a popular concurrent queue algorithm used in many commercial implementations, including the Java Concurrent Class. Its algorithm is described in detail in [8] and it is not included here due to space constraints.

The MS-Queue algorithm uses a data structure similar to that of Figure 1. Enqueues and Dequeues in the MS-Queue algorithm are based on successful execution of a Compare and Swap operation on the tail and the head pointers respectively. In general, a successful Compare and Swap operation on the MS-Queue requires that the memory location referred by the Compare and Swap remains constant for 3 memory roundtrips (One memory roundtrip to read the initial pointer, one memory roundtrip to dereference the pointer, and another memory roundtrip to complete the Compare and Swap operation).

The best throughput scenario (highest throughput) happens when the tail and head pointers are located in different memory banks, enjoying independent bandwidth and allowing simultaneous execution of Compare and Swap operations on the head and tail pointers. In that case, the throughput for the MS-Queue is given by Eq. 5: 2 queue operations can be executed every 3 memory roundtrips.

$$\mu \approx \frac{2}{3m} \tag{5}$$

The throughput of this algorithm is better than the single-lock queue and it is not affected by thread preemption due to its non-blocking nature.

3.4 Mellor-Crummey's Queue

Mellor-Crummey presented a concurrent queue (we refer to it as the MC-Queue) [7] that increases throughput by distributing queue requests over multiple traditional queues. With enough memory, the bottleneck for the throughput is given by operations on a shared variable that contains the total number of elements in the queue. The shared variable is atomically incremented during an enqueue and it is atomically incremented and read once during a dequeue. The throughput that this allows is limited by 2 queue operations every 2 atomic increments and one read. Eq. 6 gives an expression for its throughput.

$$\mu \approx \frac{2}{2k + z} \tag{6}$$

3.5 Limitations on existing queues in many-core architectures and problem formulation

Many-core architectures are particularly sensitive to low queue throughput. Any queue, even when highly optimized, can perform an average of 1 queue operation every μ^{-1} cycles, which in turn, limits the ability of individual processors to issue requests to at most:

$$\lambda^{-1} = \mu^{-1} P \tag{7}$$

Consider the case of Cyclops-64 [2] where 160 processor cores ($P = 160$) concurrently use an MS-Queue, and where each memory access to shared memory takes 30 cycles in the best case ($m = 30$). Under those conditions, each individual processor core is limited to issue at most one queue request every 7200 cycles in the most optimistic scenario if low latency at the queue is desired.

Traditional queues severely limit the usability of queues as a basic parallel construct, since for many applications that use queues, the workload associated with a queue is already in the range of few thousand cycles. The limitation is given specifically by the product $\mu^{-1}P$.

Equation 7 provides another insight in the importance of throughput for extreme scale algorithms.

3.6 Problem Formulation

The previous analysis is the foundation for the formulation of the following research questions: Is it possible to implement a fast, highly concurrent FIFO queue with enqueue and dequeue operations for many core architectures? if so, how can we implement these kind of queue algorithms? and what are the trade-offs of the algorithms in terms of their properties?

4 Solution Method

The main throughput limitation of current queue implementations is that they usually take an *"inquire-then-update"* approach on the queue, either by first acquiring a lock and then updating the queue, or by an unsynchronized read followed by a Compare-and-Swap operation.

The problem with the inquire-then-update approach is that in order to succeed, the queue must be locked during at least 2 memory roundtrips in the case of a locking implementation, or the queue must remain unchanged during at least 2 memory roundtrips for nonblocking implementations. For all practical throughput considerations in a non-preemptive environment, a locking implementation and a Compare-and-Swap based implementation will have approximately the same performance characteristics[1] and, depending on implementation, they may be subject to starvation of individual processors.

This paper takes a different approach: Queue operations (enqueue, dequeue) should succeed *immediately* if they can succeed at all. The word immediately is used in the context of not requiring multiple operations, instead, the queue structure should be changed with only *one* memory operation. In this sense, processors trying to access the queue will directly write to the queue, without first reading the state of the queue. This important distinction allows a significantly greater queue throughput than the throughput provided by an inquire-then-update queue.

The inquire-then-update is avoided by constructing the queue as an array of queue elements, in which a positive integer can be associated to a position in the array that contains a queue element. Processors performing enqueue or dequeue operations can claim a positions in the array using a single atomic increment.

Processors do not need to have exclusive access to the queue during a certain period of time, as is the case for other nonblocking implementations. Queue operations will proceed and complete after a fixed number of memory operations regardless of the state of other processors if

[1]This paper assumes that systems are non-preemptive either natively or because preemption has been disabled during queue operations. Due to practical considerations at implementation, however, Compare-and-Swap implementations usually exhibit better throughput and performance than lock-based implementations

there is enough memory during an enqueue operation, or if the queue is nonempty in a dequeue operation.

Two versions of the algorithm are presented here. Section 4.1 shows the Circular Buffer Queue algorithm: A simple queue algorithm used to demonstrate the main idea. The Circular Buffer Queue is non-blocking on dequeues of non-empty queues and enqueues of non-full queues. Otherwise, it waits until the queue becomes nonempty or not full. Section 4.2 extends the circular buffer to allow enqueue and dequeue operations regardless of the previous state of the queue.

4.1 Circular Buffer Queue

The Circular Buffer Queue (Figures 2 and 3) allows fast, concurrent, queue operations.

The main idea behind the Circular Buffer Queue is that an *atomic increment* can be used to claim a position in the queue. Enqueues can atomically increment the `WriterTicket` variable to obtain a position in the array where the user data and acknowledgement flag reside. In the same way, dequeues can atomically increment the `ReaderTicket` variable to obtain an array position where to write. A local modulo with the array size on the position is done locally by the processors to take advantage of the circular nature of the array. If the size of the array is conveniently set to be a power of 2, the modulo operation can be replaced by extraction of lower and upper bits. A *turn* (Figure 3) is a flag that indicates the status of the last operation at a particular location. Every time that an operation is completed the turn is increased by one.

A position claimed by an atomic increment (for both enqueues and dequeues) can be easily matched to a required flag to ensure correct result and ordering.

Figure 3 provides a visual example of a series of enqueues and dequeues on a Circular Buffer Queue. User data is represented by X_i. The first frame of the figure shows the initial state of the queue. The second frame shows the state of the queue after 5 enqueues. Note that enqueues first claim a position in the queue by an atomic increment on `WriterTicket`, and then compute the *turn* they must wait for. In the second frame of Figure 3 each array element has been used 0 times in the past so enqueues wait for a zero (given by the initial state) and then write the user data and set the turn to 1. Dequeues proceed similarly, incrementing `ReaderTicket`, waiting for the correct turn (that for the third frame of Figure 3 is 1), reading the data, and incrementing the turn.

The throughput of a circular buffer queue is only limited by the time required to execute an atomic increment *at the memory controller* where the variables reside. As can be observed in Figure 2 the dequeue operation is independent of the enqueue operation given enough queue space and enough items in the queue. Assuming that atomic operations take k cycles at the memory controller, the total throughput of the circular buffer queue is given by Eq. 8.

$$\mu = \frac{2}{k} \tag{8}$$

```
/* Type Definitions */
  typedef struct QueueItem_s
 {
   int64_t LastID;
   int64_t Value;
 } QueueItem_t;

/* Global Variables */
  QueueItem_t gQueue[QUEUESIZE]; // Array
  int64_t WriterTicket;          // Writer Counter
  int64_t ReaderTicket;          // Reader Counter

/* Enqueue */
void Enqueue( int value )
{
  int ticket, position, turn;
  QueueItem_t *pItem;

  ticket = atomic_increment( WriterTicket );
  turn = upper_bits( ticket ) * 2;
  position = lower_bits( ticket );
  pItem = &gQueue[ position ];

  while ( pItem->LastID != turn ) { ; } // Blocking

  pItem->Value = value;
  pItem->LastID = turn+1;
}

/* Dequeue */
int Dequeue( void )
{
  int ticket, position, turn, value;
  QueueItem_t *pItem;

  ticket = atomic_increment( ReaderTicket );
  turn = upper_bits( ticket ) * 2 + 1;
  position = lower_bits( ticket );
  pItem = &gQueue[ position ];

  while ( pItem->LastID != turn ) { ; } // Blocking

  value = pItem->value;
  pItem->LastID = turn+1;
  return( value );
}
```

Figure 2: A circular buffer implementation

The throughput of the Circular Buffer Queue is considerably larger than the throughput of other implementations where the throughput is on the order of $1/m$ queue operations per cycle (in most many-core architectures $k << m$). This paper assumes that atomic increments are executed at the memory controller. Although this is not true for processors of past generations, it is becoming the norm for current many-core architectures.

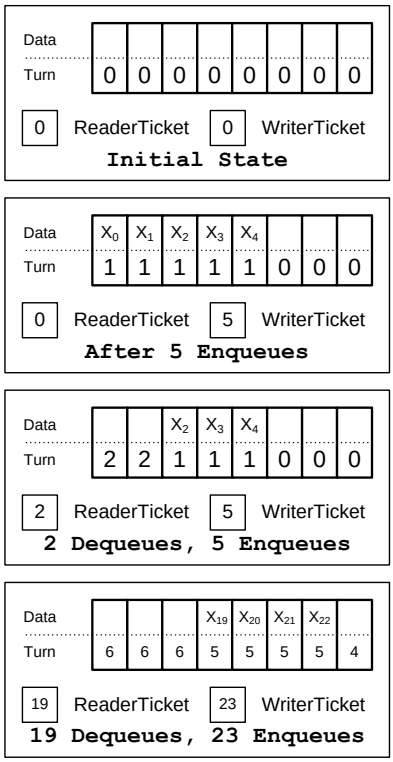

Figure 3: Circular Buffer Queue

The maximum number of elements in a Circular Buffer Queue is limited by its array size. However, in many applications this can be statically determined. *e.g.* In an operating system, the maximum number of active threads is limited by the available stack space and it is typically less than 65536. If a queue is used to schedule active threads, any array size of more than 65536 elements would be enough.

One of the disadvantages of the Circular Buffer queue is that it is not possible to reliably inquire whether or not the queue is empty (or full) at a particular point. The High Throughput Queue is an extension to the Circular Buffer and it addresses this constraint.

4.2 An extension of the Circular Buffer Queue: The High Throughput Queue

The High Throughput Queue is a fast queue that overcomes the limitations of the Circular Buffer Queue.

The main algorithm of the High Throughput Queue (Figures 4 to 6) is based on the Circular Buffer algorithm. The limitations of the Circular Buffer Queue, where operations block when the queue is empty (for a dequeue) or full (for an enqueue) are overcome by a clever idea: A counter keeps track of the available number of elements and another counter specifies whether

or not allocation of additional memory is necessary.

The High Throughput Queue avoids dereferencing dangling pointers, (*i.e.* pointers that are obsolete because the memory has been freed by another processor), a problem common to other concurrent queue implementations such as the MS-Queue.

A simple idea is used to avoid the use of old pointers that refer to deallocated memory: Obtaining the pointer and knowing whether or not the pointer is valid should be a single, atomic operation. The solution is to place the array counters with the pointers to the node structure in the same 64 bit word. This serves a double purpose: It allows claiming a position in the array (with a 64 bit atomic increment) at the same time that the array pointer is read and it allows the processor trying to claim the element to discover whether or not the queue is empty or full. Note that this technique also allows dereferencing the pointer *only* when it is guaranteed that there is available space for an enqueue or available queue elements for a dequeue. This is an important distinction that avoids the possibility of a segmentation fault, caused by a slow processor reading a pointer to a queue node that is about to be deallocated. Implementations such as the MS-Queue suffer from this weakness and are potentially prone to having a segmentation fault failure. The High Throughput Queue does not present problems with segmentation fault exceptions because the pointers are only dereferenced when a queue position has been claimed for reading or for writing.

The High Throughput Queue uses 64 bits for the tail structure (Figure 4) and 64 bits for the head structure to the queue. Each 64 bit value contains a pointer in the lower 32 bits and a count in the upper 32 bits. The queue is initialized by allocating two node structures and linking their `Previous` and `Next` pointers to each other as shown in Figure 4. Each node structure contains an array of size Q (A sample queue with $Q = 8$ is shown in Figure 4).

To enqueue an item, a processor needs to claim a position in one of the arrays in one of the nodes. To claim a position in an array, a processor atomically adds 2^{32} to the 64 bit value that contains the tail pointer, leaving the pointer unchanged and incrementing the count located in the upper 32 bits. The count returned (`Local_Counter`) uniquely identifies the node and the array location to use.
The `Tail->Previous` node is used when `Local_Counter` is less than Q, and `Tail` is used when the count is between Q and $2Q - 1$. In all cases, the count obtained (modulo Q) is used to index the array in the node. The data (X_i) is written first and then the flag is set to 1. If the count obtained is exactly $Q - 1$ the processor allocates and initializes a new node and atomically moves the tail pointer to the newly allocated node. The atomic movement can be done with a 64 bit atomic addition that effectively adds $-Q$ to the upper 32 bits and changes the pointer on the lower 32 bits to the new pointer. At the end of the operation, the processor atomically increments the `AvailableElements` counter to publish the existence of one more available element.

Dequeue operations are similar. Processors read the `AvailableElements` counter to find whether or not elements may be available. Atomic additions are used to claim one element, or to return if the queue is empty. If enough elements

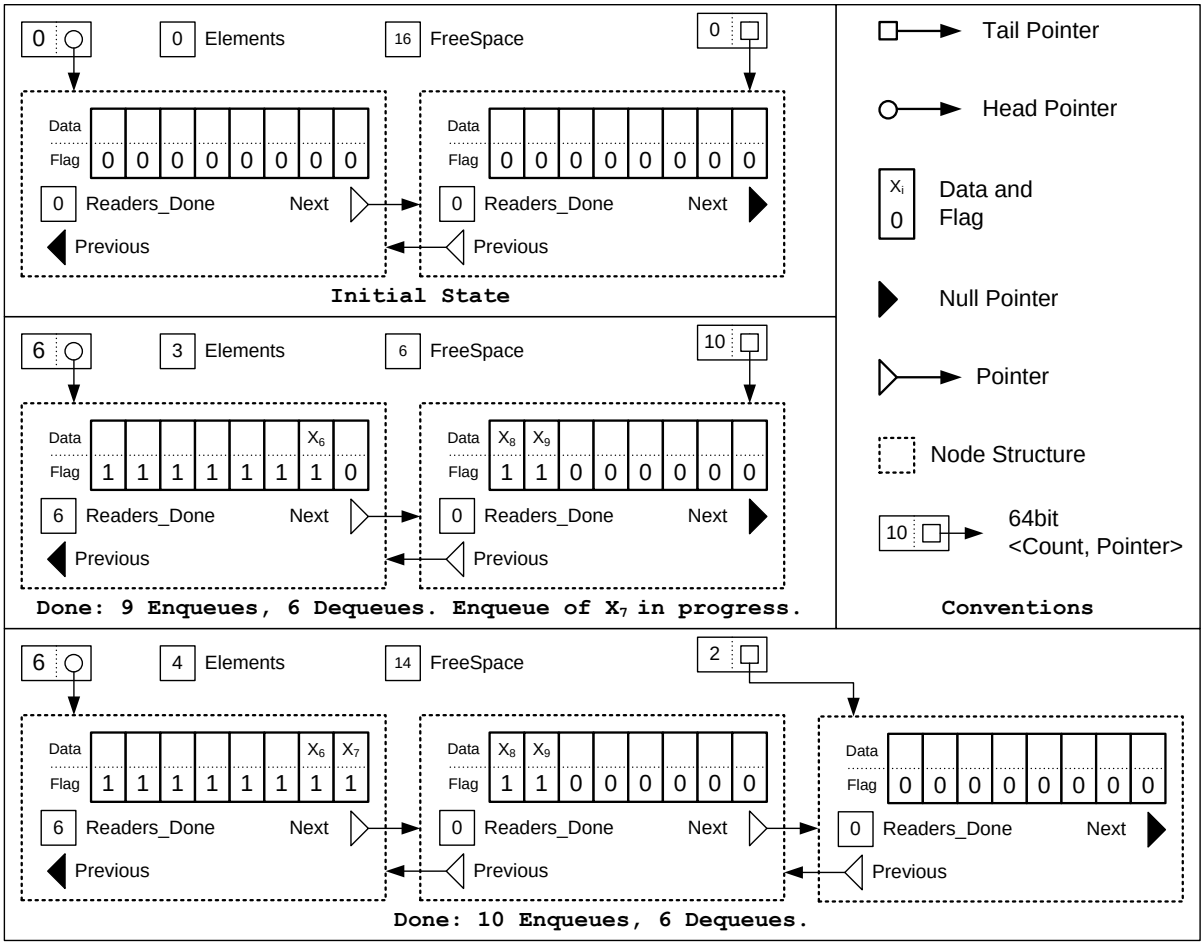

Figure 4: The High Throughput Queue

are available, processors use a 64 bit atomic increment to atomically get a pointer to the head node and a position in the array. The head node is advanced to the next node by the processor who obtains Q as the local count. Processors that obtain a value greater or equal than Q retry the atomic increment until the head variable has been advanced.

The design of the operations and the data structures in the High Throughput Queue aim to achieve a high throughput in a concurrent environment. The high throughput of the High Throughput Queue is achieved by designing the algorithm, when possible, in terms of operations that can be executed with single (atomic) memory operations.

An expression for the throughput of the High Throughput Queue algorithm is obtained by analyzing the bottlenecks of the algorithm. The total throughput is the combination of the enqueue and dequeue throughputs, with the restriction that the combined throughput can not exceed the rate at which shared resources are used. The throughput analysis assumes that in general, the values where atomic operations are required (`Elements`, `FreeSpace`, `HeadStruct`, `TailStruct`, and `Readers_Done`) are located in different memory banks that can execute the

```
1 void Enqueue( value )
2 {
3 Enqueue1:
4  Space = Atomic_Addition( &FreeSpace, -1);
5  if ( Space < 0 ) {
6   Atomic_Addition( &FreeSpace, 1 );
7   while ( FreeSpace <= 0 ) {;}
8   goto Enqueue1;
9  }
10
11  LocalTail = Atomic_Addition( &TailStruct, <1,0> );
12  Local_Counter = LocalTail.Count;
13  if ( LocalCounter % QUEUESIZE == (QUEUESIZE-1) ){
14   Node = NewNode();
15   if ( LocalCounter == QUEUESIZE -1 )
16   {  EndNode = LocalTail.Pointer }
17   else
18   {
19 Enqueue2:
20    EndNode = LocalTail.Pointer->Next;
21    if ( EndNode == NULL ) { goto Enqueue2; }
22   }
23   EndNode->Next = Node;
24   Node->Previous = EndNode;
25
26   // Consistency checks
27   ...
28
29   Atomic_Addition(&TailStruct, <-QUEUESIZE, Node-EndNode>);
30   Atomic_Addition(&FreeSpace, QUEUESIZE);
31  }
32
33  if ( LocalCounter < QUEUESIZE )
34  {  Tail_Pointer = LocalTail.Pointer->Previous; }
35  else
36  {
37   Tail_Pointer = LocalTail.Pointer;
38   LocalCounter = LocalCounter - QUEUESIZE;
39  }
40  Tail_Pointer->Array[ LocalCounter ].Value = value;
41  Tail_Pointer->Array[ LocalCounter ].Flag = 1;
42  Atomic_Addition( &Elements, 1 );
43 }
```

Figure 5: Enqueue (High Throughput Queue)

atomic operations independently.

Dequeues perform an atomic addition on the shared variable `HeadStruct`, and once per node, it is changed. The atomic addition takes k cycles at the memory controller, and every Q dequeues, the head has to be advanced, an operation that takes a full roundtrip to memory (m cycles). On average, dequeues have a throughput limit of 1 dequeue operation every $k + m/Q$ cycles.

Enqueues have a throughput limit of 1 enqueue operation every $k + k/Q$ cycles in average

```
1 int Dequeue( int *value )
2 {
3  if( Elements <= 0 )
4  { return( QUEUE_IS_EMPTY ); }
5  LocalElements = Atomic_Addition( &Elements, -1 );
6  if ( LocalElements <= 0 )
7  {
8   Atomic_Addition( &Elements, 1 );
9   return( QUEUE_IS_EMPTY );
10 }
11
12 Dequeue1:
13 // Claim item in queue
14 while( LocalHead.Count > QUEUESIZE )
15 { LocalHead=Atomic_Addition(&HeadStruct,<1,0>); }
16
17 // Move Head if necessary
18 if ( LocalHead.Count == QUEUESIZE ){
19  HeadStruct = < 0, LocalHead.Pointer->Next >;
20  Head_Pointer = LocalHead.Pointer;
21  pRD = &LocalHead.Pointer->Readers_Done;
22
23  // Free if necessary
24  Readers_Done = Atomic_Addition( pRD, 1 );
25  if ( Readers_Done == QUEUESIZE - 1 )
26  { Free( Head_Pointer ); }
27  goto Dequeue1;
28 }
29
30 LocalCounter = LocalHead.Count;
31 Head_Pointer = LocalHead.Pointer;
32 pRD = &LocalHead.Pointer->Readers_Done;
33
34 // Make sure enqueue has completed
35 while(Head_Pointer->Array[LocalCounter.Flag]==0) {;}
36
37 // Read value
38 *value = Head_Pointer->Array[ LocalCounter ];
39
40 // Free if necessary
41 Readers_Done = Atomic_Addition( pRD, 1 );
42 if ( Readers_Done == QUEUESIZE - 1 )
43 { Free( Head_Pointer ); }
44 return ( SUCCESS );
45 }
```

Figure 6: Dequeue (High Throughput Queue)

because one atomic addition per enqueue is required and one atomic addition is executed on `TailStruct` once every Q accesses.

Enqueues and Dequeues share a memory bank when incrementing the `Elements` variable to keep track of the total number of available elements. A Queue and Dequeue pair executes two atomic additions on `Elements` as well as one read. For that reason, the total throughput of the queue is limited to 2 queue operations in $2k + z$ cycles (because in the best case, the pair of

operations require 2 atomic operations plus one read on `Elements`).

The total throughput of the queue, measured in queue operations per cycle is:

$$\mu = \min(\frac{2}{2k+z}, \frac{1}{k+m/Q} + \frac{1}{k+k/Q}) \tag{9}$$

In many-core architectures, it is reasonable to expect that k and z are a small number of cycles, (usually $k < 5$ and $z = 1$). In addition, $k < m$ and Q is reasonable large ($m << Q$). Simplifying Eq. 9 and using $z = 1$ we obtain Eq. 10.

$$\mu = \frac{2}{2k+1} \tag{10}$$

5 Experiments

The performance of our queuing algorithms is compared against other state of the art queue implementations.

The experiments were designed to show latency of individual operations and total throughput of each queue implementation with varying number of processors.

The data used in this paper was gathered using ET International's Cyclops-64 simulator[1]. A Cyclops-64 chip exists, but the results had to be produced with a simulator because some of the queue techniques such as the MS-Queue [8] require the use of Compare and Swap, which is not available on the chip produced, but it can be accurately simulated.

The average latency of individual queue operations as a function of the total number of processors using the queue is shown in Figure 7. In the experiment, each processor performed 75000 Enqueue and Dequeue pairs.

The total queue throughput is shown in Figure 8. The throughput is defined as the total (aggregated) number of operations completed by all processors per unit of time. The throughput was measured by having each processor perform 20000 queue operations and by measuring the total time taken by the program.

Figure 9 zooms Figures 7 and 8 to show the performance of each queue when used with few processors. The Circular Buffer and the High Throughput Queue have a very competitive latency and throughput, and overall, perform very well at low and high processor counts.

The Circular Buffer Queue and the High Throughput Queue show significantly better performance in terms of throughput and latency than the MS-Queue and the Spin Lock implementations. In particular, as the number of processors increase, the latency of the High Throughput Queue and the Circular Buffer perform much better than the popular MS-Queue. The MS-Queue is the *de facto* industry standard, being used, among others, as part of the Java Concurrent Class. The High Throughput Queue has a throughput that is similar to the throughput

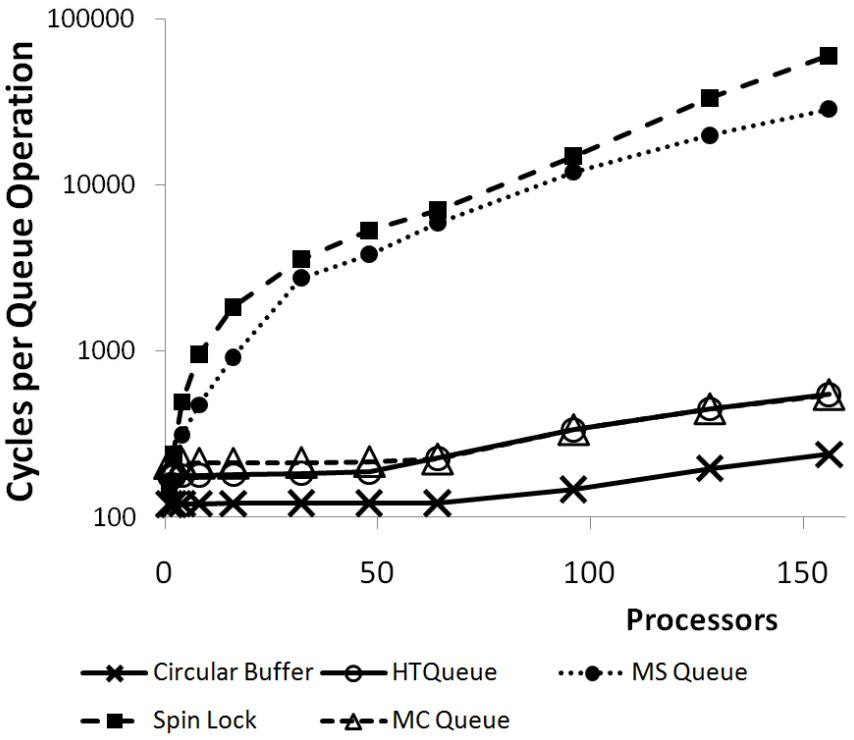

Figure 7: Latency for a single queue operation

of the MC-Queue. However, the High Throughput Queue has a lower latency because the algorithm requires less total memory roundtrips to complete each individual operation.

6 Discussion

The Circular Buffer queue and the High Throughput Queue algorithms are fast, they have a very high throughput, they have a very low latency and in general, they are excellent choices to implement queues: They have a much greater throughput than the MS-Queue and the Spin Lock queue, they have a lower latency and they use less memory than the MC-Queue.

The High Throughput Queue is preferable in a general case over the MC-Queue. Although both have similar throughput, the High Throughput Queue has lower latency, it uses less memory, and memory management is easier since arrays are allocated as opposed to individual queue items. The High Throughput Queue has a lower latency than the MC-Queue because its algorithm requires fewer roundtrips to memory to complete an operation. The High Throughput Queue uses less memory than the MC-Queue because pointers are used along each element in the MC-Queue, while in the High Throughput Queue elements are placed in an array, allowing only a pair of pointers per array.

The main drawback of the Circular Buffer Queue is the necessity to know, a priori, the

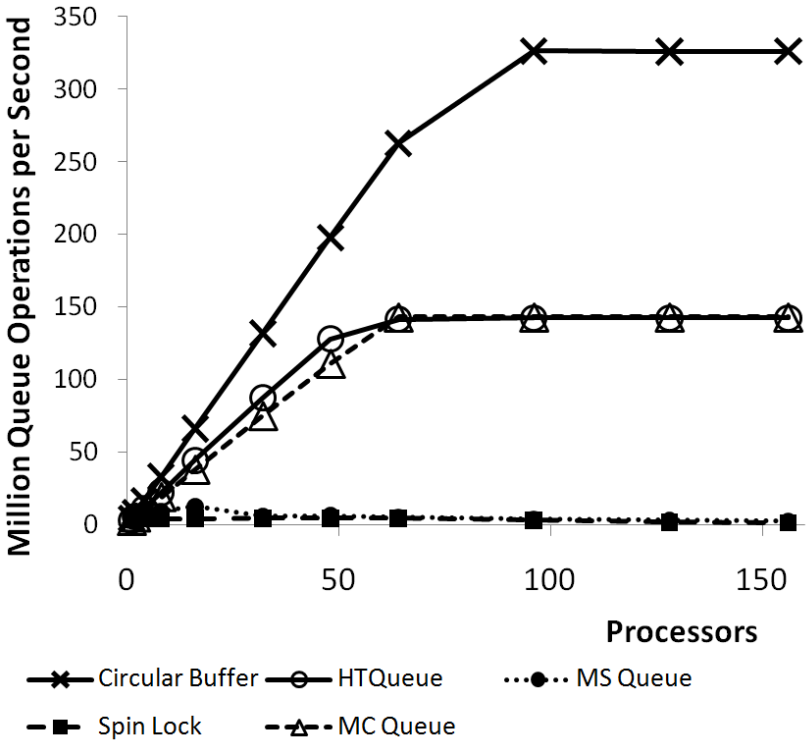

Figure 8: Queue Throughput

maximum number of elements that can be written to the queue, or accept its blocking behavior. For many high performance parallel systems, particularly extreme-scale dataflow-based systems, this is not a limitation, since the blocking behavior is usually desired and in general, the number of elements in the queue can be known a priori. For all other cases, the High Throughput Queue overcomes these limitations.

Implementation of the High Throughput Queue requires an architecture with a 64 bit atomic addition instruction as well as a 32 bit addressing space. Current trends in computer architecture suggest that 64 bit atomic additions will continue to be available. However, addressing spaces are likely to require more than 32 bits. This is not likely to be a limitation since it is relatively easy to provide a one to one mapping between the memory regions allocated for the queue and a 32 bit space (*e.g.* by allocating a large consecutive amount of memory and then representing the memory used by the queue as an offset in the allocated memory).

The High Throughput Queue algorithm presented in Section 4.2 relates to (but does not not strictly meet the requirements of) non-blocking and wait-free algorithms because for practical non-preemptive systems, where individual memory operations take a finite amount of time, individual forward progress happens in a bounded amount of time. Preemptive systems can also benefit from the non-blocking and wait-free similitudes because most modern processors support disabling of preemption for small regions of code.

The High Throughput Queue and the Circular Buffer queue algorithms result in *linearizable* queues because the queue operations appear to take place instantly at the moment where the atomic increment claiming a position in the array takes place. Linearizability [5] allows the High Throughput Queue and the Circular Buffer to be used in systems where conceptual FIFO ordering is desired.

The results of this paper may be counterintuitive with previous results [10, 12] that suggest that parallel algorithms perform faster with locks. This idea was influenced by experiments where the queue operation rate was considerably smaller than the queue's intrinsic throughput, either because they were done in environments with low contention, with low number of processors (less than 128) or because the experiments have an intrinsic bottleneck somewhere else (i.e. each node calls `malloc` which is a slow, serialized operation in the system).

7 Conclusions and Future Work

This paper introduced and developed two FIFO queue algorithms: The Circular Buffer Queue and the High Throughput Queue. Both algorithms have a large throughput and low latency. Dataflow runtime systems, and in general, operating systems can benefit from the Circular Buffer implementation. The High Throughput Queue serves as a viable replacement for tra-

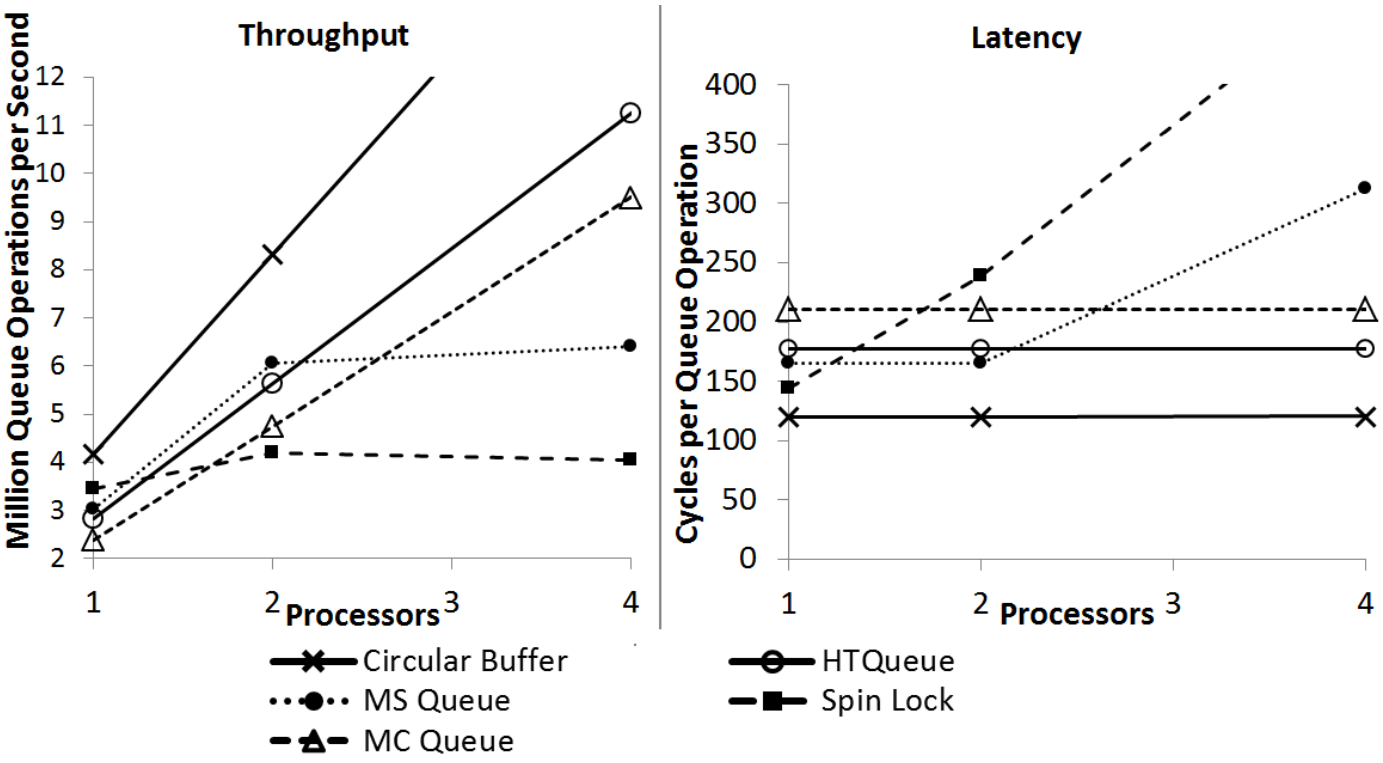

Figure 9: Performance with few processors

ditional queues because it matches their functionality with increased throughput and lower latency.

A theoretical analysis describing what is "a good queue" was presented in Section 3. where it is stated that any user program will experience less average latency if the queue has a greater throughput. A theoretical best case was obtained for common implementations of concurrent queues (a single lock queue, the MS-Queue and the MC-Queue) as well as for the queues developed in this paper (Circular Buffer Queue and High Throughput Queue).

Space limitations have forced the scope of this paper to focus on a few, selected implementations. The authors are aware of the existence of other queue algorithms. Future work will expand our analysis to include throughput models for other implementations as well as experiments comparing their performance to the Circular Buffer Queue and the High Throughput Queue.

The Circular Buffer and the High Throughput Queue are better than other queue implementations because they have a greater throughput and very low latency. The main reason for the greater throughput is that mostly, the queue operations are performed *in memory* through the use of atomic instructions as opposed to attempting inquire-then-update operations that are common on other implementations. The difference is important: An in-memory operation can complete in very few cycles, allowing more requests to be completed per unit of time than a read-modify-write approach, where at the very least, a roundtrip to memory plus some processor involvement is required for every access to the queue.

The use of a combining network, where the associativity of addition is exploited to group multiple atomic additions into a single atomic addition before they reach the memory controller is a current topic of research. A combining network would be likely to increase the throughput of atomic operations, but its cost in terms of die space are likely to offset its advantages in a production environment.

Future work will focus on how to use the queues presented here to support highly distributed execution of parallel programs. We believe that the intrinsic throughput advantages of the queues described here will help achieve better scalability of parallel programs.

8 Acknowledgements

This work was possible due to the generous support of the National Science Foundation through research grants CCF-0833122, CCF-0937907, CNS-0720531, CCF-0925863 and OCI-0904534. The authors express their gratitude to all CAPSL members at the University of Delaware for their support during the course of this research.

References

[1] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang Gao. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. *CAPSL Technical Memo 062*, 2005.

[2] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the cyclops-64 cellular architecture. In *High-Performance Computing in an Advanced Collaborative Environment, 2006.*, page 9, May 2006.

[3] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *Dist. Comp. Systems, 2003. Proc. 23rd International Conference on*, pages 522 – 529, May 2003.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[5] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[6] Leonard Kleinrock. *Queueing Systems. Volume 1: Theory.* 1975.

[7] John Mellor-Crummey. Concurrent queues: Practical fetch and phi algorithms. *Tech. Rep. 229, Dep. of CS, University of Rochester*, 1987.

[8] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[9] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51:1–26, May 1998.

[10] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proc. of 17th ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.

[11] Sheldon Ross. *First Course in Probability, A.* 2005.

[12] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proc. of the 13th ACM Symp. on Parallel Algorithms and Architectures*, SPAA '01, pages 134–143, New York, NY, USA, 2001. ACM.