Toward High Throughput Algorithms on Many Core Architectures

Daniel Orozco, University of Delaware
Elkin Garcia, University of Delaware
Rishi Khan, ET International
Kelly Livingston, University of Delaware
Guang R. Gao, University of Delaware

Advanced many-core CPU chips already have few hundreds of processing cores (*e.g.* 160 cores in an IBM Cyclops-64 chip) and more and more processing cores become available as computer architecture progresses. 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, we analyze the throughput requirements that must be met by algorithms in runtime systems to be able to handle hundreds of simultaneous operations in real time.

We reach a surprising conclusion: Many traditional algorithm techniques are poorly suited for highly parallel computing environments because of their low throughput.

We reach the conclusion that the intrinsic throughput of a parallel program depends on both its algorithm and the processor architecture where the program runs.

We provide theory to quantify the intrinsic throughput of algorithms, and we provide a few examples, where we describe the intrinsic throughput of existing, common algorithms. Then, we go on to explain how to follow a *throughput-oriented* approach to develop algorithms that have very high intrinsic throughput in many core architectures.

We compare our throughput-oriented algorithms with other well known algorithms that provide the same functionality and we show that a throughput-oriented design produces algorithms with equal or faster performance in highly concurrent environments.

We provide both theoretical and experimental evidence showing that our algorithms are excellent choices over other state of the art algorithms.

The following are the major contributions of this paper:

(1) Motivating examples that show the importance of throughput in concurrent algorithms.
(2) A mathematical framework that uses queueing theory to describe the intrinsic throughput of algorithms.
(3) Two highly concurrent algorithms with very high intrinsic throughput that are useful for task management in runtime systems.
(4) Extensive experimental and theoretical results that show that for highly parallel systems, our proposed algorithms allow greater or at least equal scalability and performance than other famous, similar, state of the art algorithms.

Categories and Subject Descriptors: C.4. [**Performance of Systems**]: Modeling Techniques

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.

DOI 10.1145/2086696.2086728 http://doi.acm.org/10.1145/2086696.2086728

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Intrinsic throughput, Manycore architectures, Throughput, Queue Algorithms, Concurrent Queues, Parallel Queue, FIFO Queue

ACM Reference Format:
Orozco, D., Garcia, E., Khan, R., Livingston, K., Gao, G. 2011. The Importance of Algorithm Throughput in Many Core Architectures. ACM Trans. Architec. Code Optim. 8, 4, Article 49 (January 2012), 21 pages.
DOI = 10.1145/2086696.2086728 http://doi.acm.org/10.1145/2086696.2086728

1. INTRODUCTION

The advent of many-core architectures poses new challenges to algorithm design, including the necessity to support several hundreds of processors concurrently. Algorithms that correctly allow access to queues by an arbitrary number of processors exist. However, correct access does not necessitate efficient access, and as shown throughout the paper, even popular non-blocking algorithms have a low amount of concurrency when used in an environment with hundreds of simultaneous requests. The driving issues in concurrent queue development have radically changed from execution in an environment dominated by virtual parallelism to an environment dominated by massive hardware parallelism. We address these challenges using a surprisingly simple intuition that allows development of faster, parallel, higher-throughput algorithms based on queues.

The intuition behind our solution can be explained with 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 a train through a main entrance (typically the head pointer). 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. The throughput of the queue can be viewed as the maximum passenger access rate. A good concurrent algorithm will allow more passengers to board (enqueue) a train (the queue) per unit of time.

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. 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. In addition, we show that many popular non-blocking algorithms for concurrent queues are also slow, among other things, because they rely on the Compare and Swap operation, which may or may not succeed in swapping its operands when it is executed, and may need to be executed many times.

Queueing theory is used to reason about the performance (*e.g.* the throughput) of a queue algorithm. The advantage of using queueing theory is that it allows a formal analysis of the interplay between factors such as latency of individual operations, number of processors and algorithm structure. Section 2.3 discusses the importance of queueing theory to analyze whether or not the number of requests in a system approaches the intrinsic threshold of an algorithm.

We present two algorithms characterized for their excellent throughput: The Circular Buffer Queue (CB-Queue) and the High Throughput Queue (HT-Queue). Both were designed to achieve high throughput while providing First In First Out (FIFO) behavior. The CB-Queue and the HT-Queue are able to scale to unprecedented levels because they avoid the *inquire-then-update* approach of other implementations. 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 commit a queue 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.

The CB-Queue is introduced first. The CB-Queue always completes queue operations successfully, waiting for queue elements or enough memory as needed. The HT-Queue expands the basic algorithm of the CB-Queue to allow 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.

The algorithms presented here are different to other algorithms using Fetch and Add operations in that (1) the inquire-then-update is avoided when possible, and (2) the algorithms were designed to allow the maximum possible throughput.

We compare our queue implementations against Mellor-Crummey's algorithm (MC-Queue), the algorithm with the highest throughput that we could find [Mellor-Crummey 1987] (it also uses fetch and add), the industry *de facto* standard by Michael and Scott (MS-Queue) [Michael and Scott 1996], and a simple implementation with locks. Our theoretical results show that our algorithms have a very high throughput, surpassing the MS-Queue and spinlock implementations and matching (in the case of the HT-Queue) or surpassing (in the case of the CB-Queue) the throughput of the MC-Queue implementation. Our experimental results (Section 5), that use carefully written microbenchmarks, have closely reproduced our theoretical throughput predictions (Section 4), and they show that the throughput of our queue operations is only limited by the bandwidth of the memory controllers, providing a wider range of scalability than other approaches such as the MS-Queue or traditional queues based on locks. Our experiments also show that the throughput of queues is critical to support the execution of programs that use more than 64 processors.

The rest of the paper is organized as follows: Section 2 provides relevant background, Section 3 introduces throughput as a 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. Finally, discussion on our findings, conclusions and future work are presented in Sections 6 and 7.

2. BACKGROUND

This section presents relevant background on the architecture used for our analysis and experiments, a brief overview of the issues faced by concurrent queue algorithms, and a description of relevant queueing theory concepts.

2.1. Cyclops-64 and Many-Core Architectures

Cyclops-64 (C64) is a many core architecture produced by IBM and extensively described by del Cuvillo et al. [del Cuvillo et al. 2006].

Each C64 chip has 80 cores, each containing 2 single-issue thread units. Each core in C64 has 60KB of local SRAM memory with its own memory controller amounting to 5MB of on-chip memory. C64 also has 1GB of external off-chip DRAM memory accessed through 4 DRAM banks, each with an independent memory controller.

All memory controllers in C64 support *in-memory* atomic operations: Each memory controller has an ALU that allows it to execute atomic operations in 3 clock cycles directly inside the memory controller (both SRAM and DRAM), without help from a thread unit.

C64 was designed to operate as a non-preemptive system. There is no virtual memory, there is no automatic cache and all memory is visible and addressable by the user.

2.2. Concurrent Queue Algorithms and Scalability

A considerable amount of attention has been paid to the nonblocking behavior of concurrent objects such as queues. However, no evidence has been shown to say that all nonblocking implementations are necessarily *effective* for large number of processors.

Several papers have shown that the MS-Queue and non-blocking queue variants of it [Moir et al. 2005; Tsigas and Zhang 2001] based on the "Compare and Swap" operation provide a reasonably good performance in shared memory systems 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, to implement the Java Concurrent Class.

The MC-Queue seeks to increase parallelism by distributing queue requests over multiple traditional queues. The MC-Queue has a high throughput because the load of enqueues and dequeues is distributed over an array of queues. The count of available elements in the queue remains centralized, making its update the bottleneck of the implementation.

2.3. Relevant Queueing Theory

Queueing theory is a mature field used to analyze the performance of queues. An excellent introductory work on queueing theory can be found in Kleinrock's 1975 textbook [Kleinrock 1975].

Queueing theory dictates that latency greatly increases when the available queue throughput is comparable to the rate of 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 intrinsic throughput of a queue. Current trends in computer architecture suggest that more and more processors will be used for computations, stressing the need for concurrent algorithms with throughputs that allow scaling to unprecedented numbers of processors.

The following conventions are used throughout the rest of the paper:

— μ is the *throughput* of the queue: The maximum number of requests that can be serviced per unit of time.
— P is the number of processors accessing the queue.
— r is the average amount of time taken between calls to queue operations.
— λ is the average request arrival rate to the queue. When the latency at the queue is low, λ can be approximated as $\lambda \approx P/r$
— $\rho = \lambda/\mu$ is the *utilization factor* of the queue.
— 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. This parameter arises from the fact that C64 has the ability to execute some operations at the memory controller without help from any thread unit.
— z is the number of cycles a memory read or write uses the memory controller. In general $z < k$ because the memory controller needs to do less work to complete a normal write (or read) than to compute an atomic operation.

By definition, μ, the throughput of the queue, specifies the maximum number of requests that can be serviced by the queue per unit of time. μ sets a bound on the intrinsic parallelism of the queue: The queue will scale until the request rate λ reaches the throughput μ because the queue can not service more than μ requests per unit of time.

A queue is defined as stable if $\mu > \lambda$, or $\rho < 1$. When $\rho > 1$, requests entering the queue accumulate faster than they can be served and, in theory, latency increases to infinity. In practice, the system saturates, limiting the request rate to be $\lambda = \mu$, (or $\rho = 1$) and stabilizes the queue with high latency.

Analyzing with precision the latency at the queue requires specifying the probability distribution function of the arrival rate to the queue and the service rate at the queue, a topic outside the scope of this paper. Instead, we will settle by saying that in general, as λ approaches μ (and ρ approaches 1) the waiting time at the queue *increases*.

3. THE PROBLEM WITH EXISTING CONCURRENT QUEUES ON MANY-CORE ARCHITECTURES

This section uses queueing theory to further advance an interesting result: *queue throughput* is the single most important aspect of a queue in a large parallel system. This result is a natural consequence of an intuitive performance requirement: In a parallel program, queue operations should have low latency.

The goal of the theoretical analysis is to find the maximum throughput (or intrinsic throughput) that a particular queueing algorithm can deliver in a particular machine. The maximum throughput of a particular queueing algorithm depends both on the hardware used and the implementation of the algorithm. In general, the intrinsic throughput of an algorithm can be found by analyzing the availability of resources, both logical and physical. In practical situations for FIFO queues, either communication between processors is present, memory is shared, or logical resources such as locks are shared, limiting the intrinsic throughput to a certain, finite value.

The intrinsic throughput of queueing algorithms (Section 2) is of utmost importance because it sets a bound to the maximum rate of requests that still result in a utilization factor ρ that is less than 1. The importance to the user becomes painfully obvious: When the queue operation request rate increases and approaches the intrinsic throughput, the latency of individual operations will increase because the queue can not serve more requests than its intrinsic throughput.

We analyze the intrinsic throughput of selected queueing algorithms starting with an example in Section 3.1 and presenting several algorithms in Sections 3.2 to 3.4. The limitations of those queues in many-core architectures are exposed in Section 3.5, allowing a formal statement of the problem to be solved.

As it will be seen in the following sections, the throughput of an algorithm is intrinsically related to the features present in the architecture used to run it: Memory latencies, the ability to perform memory operations in memory, the presence of a shared bus or a crossbar, the number of memory banks and so on.

We use Cyclops-64 (C64) to explain our ideas about throughput and to show how to do a throughput-oriented design of an algorithm. C64 was chosen because it has a large number of thread units per chip, its architectural features are relatively easy to control and predict, there is no virtualization or preemption that introduces noise, the user can directly access the hardware, and it has features such as in-memory atomic operations.

3.1. A Simple Example: Throughput of a Test and Set Lock

Consider a program δ composed of only two operations: (1) obtain a global lock using the test-and-set algorithm, and (2) release the global lock.

The intrinsic throughput of program δ is the number of processors that can *complete* program δ per unit of time. Note that the intrinsic throughput does not talk about the time taken by individual processors to complete the program. It talks about the number of completions per unit of time.

Possession of the lock is the bottleneck for the program. The number of programs that can complete per unit time depends on the number of times that the lock can be obtained and released per unit time.

Figure 1 shows that, from the point of view of the memory, acquiring the lock only takes *half a roundtrip*, because the lock is free (or owned by another processor) during

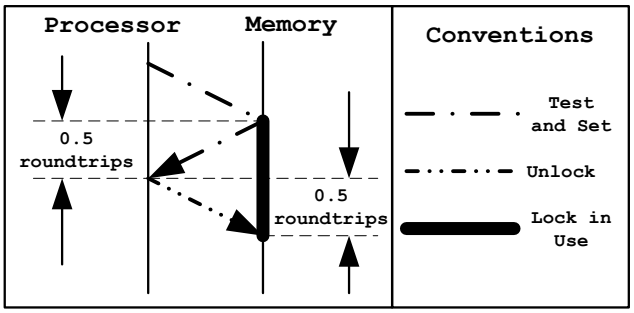

Fig. 1. Time for ownership of lock

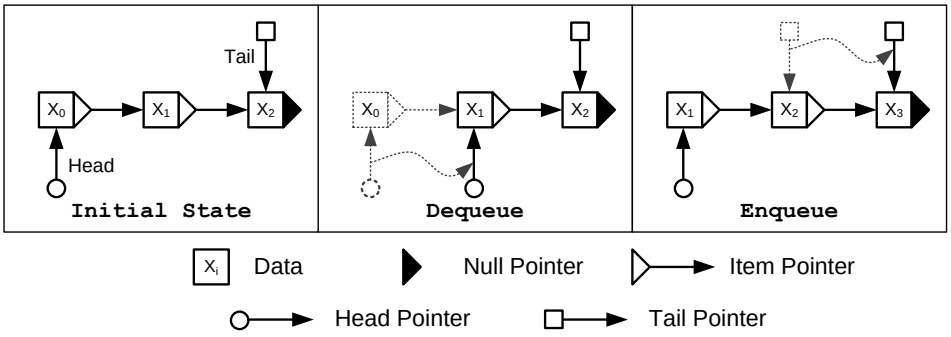

Fig. 2. Traditional view of a queue

the first half roundtrip of the test and set operation. Likewise, releasing the lock only costs half a roundtrip.

Thus the throughput of program δ is $\mu_\delta = 1/m$ (m is defined as a memory roundtrip in Section 2.3).

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 2 shows the data structure commonly used to implement this 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:

$$\mu = \frac{1}{2m} \tag{1}$$

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 MS-Queue is a popular algorithm used in many commercial implementations, including the Java Concurrent Class. Its algorithm is described in detail in Michael and Scott's 1996 work [Michael and Scott 1996].

The MS-Queue algorithm uses a data structure similar to that of Figure 2. 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 2 memory roundtrips (half a memory roundtrip to read the initial pointer, one memory roundtrip to dereference the pointer, and half a 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 total throughput is the throughput for enqueues plus the throughput for dequeues, and it is given by Eq. 2. 2 queue operations can be executed every 2 memory roundtrips.

$$\mu = \frac{2}{2m} = \frac{1}{m} \tag{2}$$

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. MC-Queue

The MC-Queue increases throughput by distributing queue requests over multiple traditional queues. The bottleneck is either a sequence of operations on a shared variable that keeps track of the element count or the aggregated throughput of all the traditional queues in the implementation. A enqueue-dequeue pair performs 2 atomic operations and one read on the shared variable limiting the throughput to 2 operations every $2k + z$ cycles. Enqueues and dequeues can complete in each individual queue after 3 roundtrips to memory limiting the throughput to 2 operations every $3m$ cycles. Eq. 3 presents the intrinsic throughput, the $\min$ is simplified under the assumption that the number of queues (G) is large enough.

$$\mu = \min\left(\frac{2}{2k+z}, \frac{2}{3m}G\right) = \frac{2}{2k+z} \tag{3}$$

3.5. Limitations on existing queues in many-core architectures

Many-core architectures are particularly sensitive to low queue throughput. Any queue, even when highly optimized, can perform, at most, 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:

$$\lambda^{-1} = \mu^{-1}P \tag{4}$$

Consider the case of C64 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 4800 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: Our experiments show examples of two applications where each processor uses the queue, in average, every 10000 cycles. The limitation is given specifically by the product $\mu^{-1}P$. Eq. 4 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, high-throughput, highly concurrent FIFO queue with enqueue and dequeue operations for many core architectures? If so, how can we implement these kind of queue algorithms? What are the trade-offs of the algorithms in terms of their properties?

4. SOLUTION METHOD

The *"inquire-then-update"* approach is one of the main throughput limitations in current queue implementations: 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 implementations using Compare and Swap (See Section 3.3).

A surprising result from Sections 3.2 and 3.3 is that nonblocking implementations and lock-based implementations of queues in non-preemptive environments have throughputs that are in the same order of magnitude.

This paper seeks a significantly stronger stance: 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 approach, because changes to the queue happen during the time the memory controller serves the memory operation *in memory* as opposed to happening over the course of several memory roundtrips.

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. Processors performing enqueue or dequeue operations can claim positions in the array using a single atomic increment without exclusive access to the queue during a certain number of memory roundtrips.

Two versions of the algorithm are presented here. Section 4.1 shows the CB-Queue algorithm: A simple queue algorithm used to demonstrate the main idea. The CB-Queue algorithm allows dequeues on non-empty queues and enqueues on non-full queues, waiting if necessary until the queue becomes nonempty or not full. Section 4.2 extends the CB-Queue to allow enqueue and dequeue operations regardless of the previous state of the queue. Finally, Section 4.3 details additional considerations made on the theoretical analysis that can affect the behavior of the implementations.

4.1. CB-Queue

The CB-Queue (Figures 3 and 4) allows fast, concurrent, queue operations.

The main idea behind the CB-Queue is that an *atomic increment* can be used to claim a position in the queue. Enqueues can atomically increment a variable (`WriterTicket`) to obtain a position in the array where the user data and acknowledgement flag reside. In the same way, dequeues can atomically increment a variable (`ReaderTicket`) to obtain an array position where to read. A modulo operation is used to map the value

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

/* Global Variables */

  // Array
  QueueItem_t gQueue[QUEUESIZE];

  // Writer Counter
  int64_t WriterTicket;

  // Reader Counter
  int64_t ReaderTicket;

/* Macros */
  #define upper_bits( x ) ( x / QUEUESIZE )
  #define lower_bits( x ) ( x % QUEUESIZE )
```

```
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;
}

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 );
}
```

Fig. 3. A circular buffer implementation

obtained to an offset in the array. The modulo operation can be simplified to a simple lower and upper bit extraction if the array length is a power of 2. A *turn* (Figure 4) is a flag that indicates the status of the last operation at a particular location. The turn for a particular array position is increased every time that an operation is completed at a particular array position. The value of the turn represents the number of times that the queue has been accessed, and it allows synchronization between processors that attempt to use the same array location. A position claimed by an atomic increment (for both enqueues and dequeues) can be easily matched to a required turn (See Figure 3) to ensure correct result and ordering.

Figure 4 provides a visual example of a series of enqueues and dequeues on a CB-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 4, each array element has been used 0 times in the past, so enqueues wait until turn=0 (given by the initial state), write the user data, and finally set the turn to 1. Dequeues proceed similarly, incrementing `ReaderTicket`, waiting for the correct turn (turn=1 in the third frame of Fig. 4), reading the data, and incrementing the turn.

The throughput of the CB-Queue is only limited by the time required to execute an atomic increment *at the memory controller* (this time is k) where the variables reside. Then, one enqueue and one dequeue operation can be completed every k cycles, making the throughput:

$$\mu = \frac{2}{k} \tag{5}$$

The throughput of the CB-Queue is considerably larger than the throughput of other implementations where the throughput is on the order of $1/m$ queue operations per cycle (*e.g.* in C64, $k = 3$ and $m = 30$. We don't see a reason for the ratio between k and m to change in the future). Note that atomic increments are executed at the memory controller (Memory controllers can execute some operations. See Section 2.1).

The maximum number of elements in a CB-Queue is limited by its array size. However, in many applications this can be statically determined. *e.g.* In an operating sys-

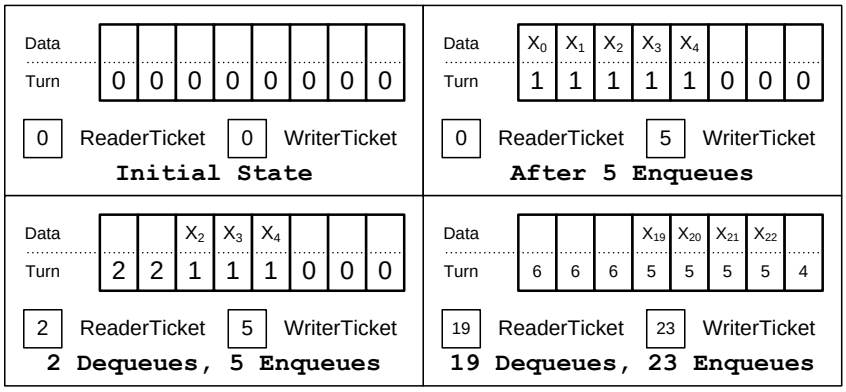

Fig. 4. CB-Queue

tem, the maximum number of active threads is limited by the available stack space and it is typically less than 65536. In general, a CB-Queue of size Q can be used to arbitrate concurrent access to Q units of a particular resource (like active threads).

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

4.2. The HT-Queue: An extension of the CB-Queue

The HT-Queue is a fast queue that overcomes the limitations of the CB-Queue. The main algorithm of the HT-Queue (Figures 5 to 7) is based on the CB-Queue algorithm.

The limitations of the CB-Queue are overcome with the addition of new features (1) to support an unbounded number of elements, (2) to allow inquiring the status of the queue and (3) to avoid the presence of dangling pointers (*i.e.* pointers that are obsolete because the memory has been freed by another processor).

An unbounded number of elements is supported by the HT-Queue because the HT-Queue is constructed as a linked list of *nodes*. A node (Figure 5) is a data structure composed of (1) several queue items, each with a reader/writer synchronization flag, (2) pointers for the linked list and (3) an integer that counts how many reads to the node have been made. Among other things, the node structure amortizes the overhead of memory allocation because it holds several queue elements.

Inquiring about the status (*e.g.* empty) of the queue is supported by the HT-Queue algorithm with the addition of an element counter and a free space counter. Additionally, the *turn* variables associated with each queue element in the CB-Queue have been replaced with flags in the HT-Queue.

Dangling pointers are avoided in the HT-Queue because pointers are only dereferenced when it is guaranteed that the processor will successfully complete the operation for which the pointer is required. A simple idea is used to accomplish this: Obtaining a pointer and knowing whether or not the pointer is valid should be a single, atomic operation. For the HT-Queue, this is achieved by placing the (reader, writer) position counter and the (head, tail) pointer 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

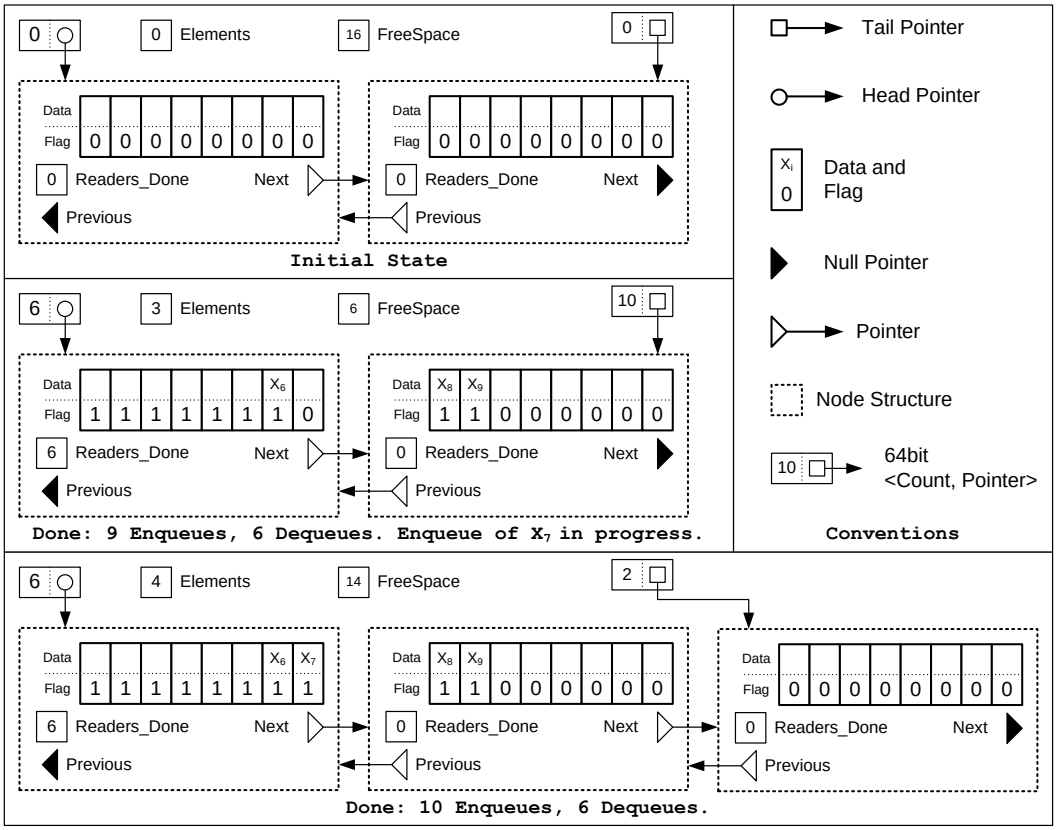

Fig. 5. The HT-Queue

distinction that avoids the possibility of memory access exceptions, caused by a slow processor reading a pointer to a queue node that is about to be deallocated. Our use of a single 64 bit word to have a pointer and an integer should not be confused to other techniques such as solutions to the ABA problem [Michael and Scott 1996]. Section 6 discusses the differences in detail.

The HT-Queue uses 64 bits to represent the head and the tail of the HT-Queue (Figure 5). Each 64 bit value contains a (head, tail) pointer in the lower 32 bits and a (reader, writer) 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 5. Each node structure contains an array of size Q (A sample queue with $Q = 8$ is shown in Figure 5).

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

```
void Enqueue( value ) {
Enqueue1:
 Space = Atomic_Addition( &FreeSpace, -1);
 if ( Space < 0 ) {
  Atomic_Addition( &FreeSpace, 1 );
  while ( FreeSpace <= 0 ) {;}
  goto Enqueue1;
 }

 LocalTail =
   Atomic_Addition( &TailStruct, <1,0> );

 LocalCounter = LocalTail.Count;
 if( LocalCounter%QUEUESIZE == (QUEUESIZE-1))
 {
  Node = NewNode();
  if ( LocalCounter == QUEUESIZE -1 )
  {  EndNode = LocalTail.Pointer }
  else  {
Enqueue2:
   EndNode = LocalTail.Pointer->Next;
   if ( EndNode == NULL ) { goto Enqueue2; }
  }
  EndNode->Next = Node;
  Node->Previous = EndNode;

  Atomic_Addition(
    &TailStruct, <-QUEUESIZE, Node-EndNode>);
  Atomic_Addition(&FreeSpace, QUEUESIZE);
 }

 if ( LocalCounter < QUEUESIZE ) {
   Tail_Pointer =
     LocalTail.Pointer->Previous;
 }
 else {
  Tail_Pointer = LocalTail.Pointer;
  LocalCounter = LocalCounter - QUEUESIZE;
 }

 pItem = &Tail_Pointer->Array[ LocalCounter ];
 pItem->Value = value;
 pItem->Flag = 1;

 Atomic_Addition( &Elements, 1 );
}
```

Fig. 6. Enqueue (HT-Queue)

```
int Dequeue( int *value )
{
 if( Elements <= 0 )
 { return( QUEUE_IS_EMPTY ); }
 LocalElements = Atomic_Addition( &Elements, -1 );
 if ( LocalElements <= 0 )
 {
  Atomic_Addition( &Elements, 1 );
  return( QUEUE_IS_EMPTY );
 }

Dequeue1:
 // Claim item in queue
 while( LocalHead.Count > QUEUESIZE )
 { LocalHead=Atomic_Addition(&HeadStruct,<1,0>); }

 // Move Head if necessary
 if ( LocalHead.Count == QUEUESIZE ){
  HeadStruct = < 0, LocalHead.Pointer->Next >;
  Head_Pointer = LocalHead.Pointer;
  pRD = &LocalHead.Pointer->Readers_Done;

  // Free if necessary
  Readers_Done = Atomic_Addition( pRD, 1 );
  if ( Readers_Done == QUEUESIZE - 1 )
  { Free( Head_Pointer ); }
  goto Dequeue1;
 }

 LocalCounter = LocalHead.Count;
 Head_Pointer = LocalHead.Pointer;
 pRD = &LocalHead.Pointer->Readers_Done;

 // Make sure enqueue has completed
 while(Head_Pointer->Array[LocalCounter.Flag]==0) {;}

 // Read value
 *value = Head_Pointer->Array[ LocalCounter ];

 // Free if necessary
 Readers_Done = Atomic_Addition( pRD, 1 );
 if ( Readers_Done == QUEUESIZE - 1 )
 { Free( Head_Pointer ); }
 return ( SUCCESS );
}
```

Fig. 7. Dequeue (HT-Queue)

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 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 HT-Queue aim to achieve a high throughput in a concurrent environment. The high throughput of the HT-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 HT-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 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 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:

$$\mu = \min\left(\frac{2}{2k+z}, \frac{1}{k+m/Q} + \frac{1}{k+k/Q}\right) \tag{6}$$

In many-core architectures, it is reasonable to expect that k and z are a small number of cycles, (For example in C64 $k = 3$ and $z = 1$). In addition, it is reasonable to assume that $k < m$ and that Q is made reasonable large ($m << Q$). Simplifying Eq. 6 we obtain Eq. 7.

$$\mu = \frac{2}{2k+z} \tag{7}$$

4.3. Additional considerations

This section presents a discussion of the assumptions used during the design of the algorithms of Sections 4.1 and 4.2.

Memory model: By design, our HT-Queue interacts with shared variables through atomic operations. The memory model of the system must ensure that atomic operations appear to complete in the same order from the point of view of each processor. In particular, weak consistency [Dubois et al. 1998] guarantees that all accesses to synchronization variables are seen by all processes in the same order, which is the condition required. The CB-Queue, MC-Queue and MS-Queue rely on a combination of atomic operations and particular sequences of memory operations, requiring sequential consistency. The Spinlock implementation uses locks, allowing implementations that use significantly weaker memory models.

The work of Zhang et al [Zhang et al. 2005] has shown that C64 is sequentially consistent, allowing C64 to correctly support all the queue algorithms presented here. The HT-Queue and Spinlock queue can be implemented in C64 because the weaker memory models that they require are a subset of Sequential Consistency.

In architectures where weak consistency or sequential consistency is not available, correct results for the queue algorithms discussed in this paper (CB-Queue, HT-Queue, MS-Queue, MC-Queue and Spinlock) can be obtained by using memory barriers (1) around atomic operations and (2) around memory operations to variables that participate in synchronization (such as the `turn` variable in the CB-Queue).

Memory allocation: The throughput analysis of the previous sections considers the effects of the algorithms themselves. However, the throughput of an algorithm can be affected by external factors. For example, the throughput of algorithms that allocate memory can vary greatly depending on the implementation of the memory allocator. In situations where the memory allocation is done through a centralized lock, the throughput of the algorithm may be significantly reduced. On the other hand, the throughput of a processor will not be affected at all if each processor has its own memory pool where memory can be requested without interference from other processors.

To avoid the problems of memory allocation, the examples and experiments of this paper use a distributed memory allocator where little or no contention between processors exist.

Effects caused by the Operating System (OS): The OS can potentially limit the throughput of an algorithm. Any call that is serialized by the OS can become a bottleneck that limits the throughput of a parallel algorithm. For example, parallel algorithms making system calls will see their throughput limited if those calls are implemented so that they are serviced by a single, particular processor.

To isolate our examples and experimental data from the interference of particular implementations of the OS, we have implemented our algorithms without system calls.

Use of other architectures: This section has presented a methodology that produces a high throughput algorithm given a particular architecture. The particular algorithm that produces the highest throughput may change when the architecture is changed due to the details in the way memory operations are served, atomic operations are executed, and the availability and quantity of shared resources of each type.

The usability and throughput of the algorithms presented here, if used in another architecture, will depend on the characteristics of the architecture where they are used: whether or not particular atomic operations are available, whether or not those operations can be executed in-memory, the number of memory banks and so on.

For example, the effect of the memory requirements of our algorithms will depend on many factors, including how many bits are required for the flags associated with each element, the number of pointers used by the algorithms, the size of the caches and the size of individual cache lines. The methodology followed in our paper may be used to determine whether or not the use of one flag per element causes negative effects for architectures with caches.

Memory banks used by multiple variables: The throughput expressions obtained assume that memory banks serve particular variables exclusively. Although it is possible to place the critical variables in different memory banks, other variables, including the queues themselves, may access all memory banks. The large number of memory banks in C64 (96 total) make these accesses infrequent resulting in a slight reduction of the actual throughput experienced.

5. EXPERIMENTS

This section evaluates the performance of our queue algorithms. First, we use small benchmarks that show the latency of individual queue operations and the total throughput of each queue implementation. Then, we show the effectiveness of each queue implementation to support the execution of two applications.

We used the C64 architecture in our experiments due to its large number of hardware threads. We have used the highly accurate ET International's C64 simulator [del

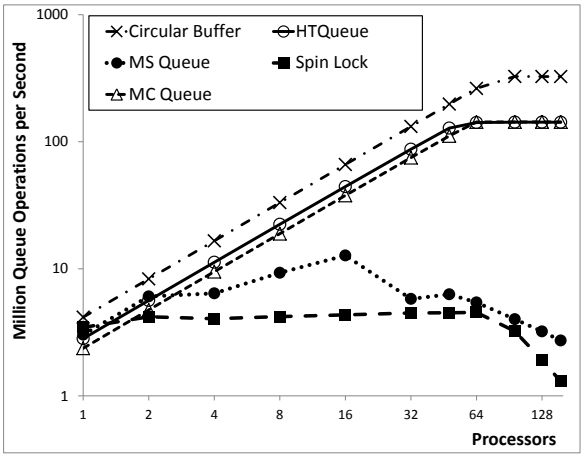

Fig. 8. Queue Throughput

Cuvillo et al. 2005] to gather all the data presented in this paper because (1) despite the existence of real C64 chips, all of them are currently held by the U.S. Government and they have not been released to the public and (2) some of the other queue techniques used as comparison, such as the MS-Queue [Michael and Scott 1996] and the MC-Queue [Mellor-Crummey 1987] require the use of Compare and Swap, which is not available on the C64 chip produced. The C64 simulator was modified to include a CAS native instruction in its ISA. To make our comparison fair, the CAS instruction has the same implementation in the simulator as all other atomic operations: It has the same latency, it uses the same resources, it has its own opcode in the ISA, it is also executed in-memory and it generates the same contention at the memory controller. The compiler was modified accordingly to support the new opcode. The microbenchmarks were written in assembly while the applications were written in C.

5.1. Microbenchmarks

Throughput (Figure 8) and latency (Figure 9) were measured using experiments where each processor performs a sequence of 75000 enqueue-dequeue pairs. During the interval measured, the processors do not execute anything other than the enqueue-dequeue pairs, and there is no waiting. Indirect effects such as system calls to `malloc` or `free` that may affect throughput were avoided. All memory is allocated before the experiment is run and deallocated after the experiment completes.

The throughput is defined as the total (aggregated) number of operations completed by all processors per unit of time. The latency reported is an average over all the individual queue latencies.

Table I shows that our theoretical predictions on throughput are confirmed by the experiments. The theoretical throughput for the MC-Queue, the CB-Queue and the HT-Queue matches very well the throughput measured. The reason is that the expression for the theoretical throughput is a function of the memory controller parameters (k and z) which are constant for C64. The theoretical throughput for the MS-Queue and the Single Lock queue does not match the throughput observed because the expressions for the theoretical throughput depend on the latency of individual memory operations, which is not a constant, and increases with the load of the system. This degradation

Table I. Comparison of theoretical and experimental results.

Queue	μ (Theoretical)	μ (Experimental)
Single Lock	8.33	4.54
MS-Queue	16.7	12.7
MC-Queue	142.8	142.8
CB-Queue	333	326.0
HT-Queue	142.8	142.5

Units: Million Queue Operations per Second.

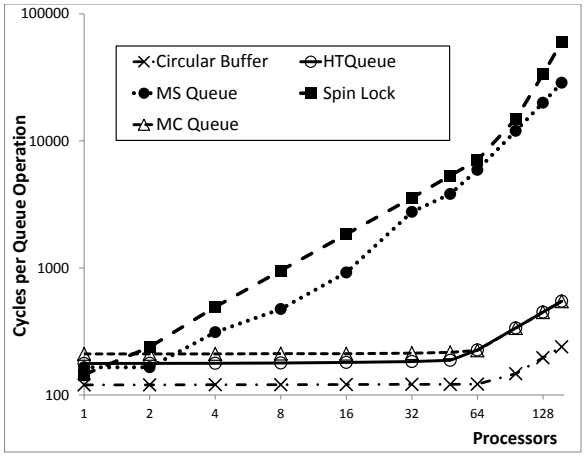

Fig. 9. Latency for a single queue operation

Table II. Maximum request rate (Million Operations per Second) before queue latency increases 5%.

Queue	Single Lock	MS-Queue	MC-Queue	CB-Queue	HT-Queue
Max. Request Rate	3.46	6.06	142.5	262.68	128.00

of throughput can be seen in Figure 8: When the system is heavily loaded, the latency for individual memory operations increases, lowering the total system throughput.

The CB-Queue and the HT-Queue show significantly better intrinsic (maximum) throughput than the MS-Queue and the Single Lock implementation, and equal (in the case of the HT-Queue) or greater (CB-Queue) throughput than the MC-Queue.

The behavior of latency with respect to the utilization factor ρ of the queue is shown in Figure 10. ρ has been calculated as the ratio of requests to theoretical throughput (Section 2.3) for each queue. The latency of our implementations when the queue is not saturated ($\rho < 1$) is better than the latency of all other implementations tested. When the queue becomes saturated ($\rho = 1$) the latencies of all queue implementations increase. Due to its high throughput, our CB-Queue can handle a large request rate before the latency increases due to saturation. Our HT-Queue and the MC-Queue saturate similarly. Experimental values for maximum request rates before the latency increases 5% are shown in Table II.

5.2. Applications

The impact of our queueing algorithms in larger applications was explored by using them in ET International's parallel runtime system [ET International] where queues

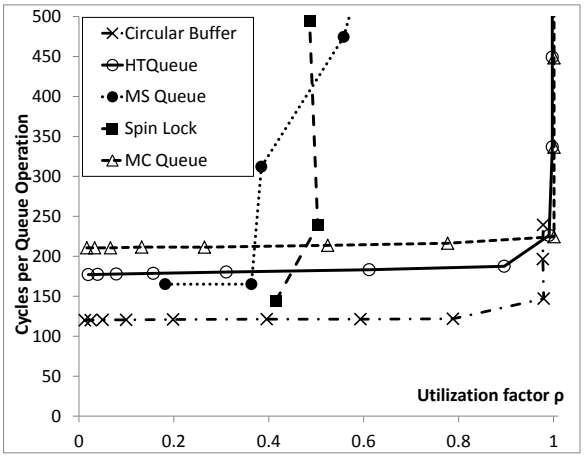

Fig. 10. Latency vs. Utilization Factor

are used as the central mechanism for scheduling in a way similar to that of the Cilk runtime [Blumofe et al. 1995] and the EARTH runtime [Theobald 1999] systems.

Two applications were used to test the impact of each queue implementation: A tiled version of a 3-Dimensional Reverse Time Migration (RTM) [Baysal et al. 1983] used for oil exploration (8000 C code lines) with input size of $276 \times 276 \times 276$ and Blocked Matrix Multiply (MM) [Garcia et al. 2010] (3000 C code lines) of size 5760×5760.

RTM consists of repeated point-wise multiplication and convolution of a set of 3-Dimensional input samples with a 3-Dimensional kernel. Good performance is achieved though multiple code transformations [Nguyen et al. 2010] that improve the locality during execution, resulting in moderate parallelism and abundant synchronization between tasks. The input samples reside in DRAM and they are executed as tiles that fit in on-chip memory. Overlapping of communication (between DRAM and SRAM) and computation is done by having two tiles in on-chip memory. In the experiments conducted, four arrays of $276 \times 276 \times 276$ single precision numbers are used (approx 370MB in DRAM) with a kernel of size $13 \times 13 \times 13$. Tasks either compute a convolution between a data set of size $6 \times 1 \times 1$ and the kernel of size $13 \times 13 \times 13$ or they do memory movement between DRAM and SRAM.

For MM, the three Double Precision matrices are in off-chip DRAM memory (approx. 800MB), the nature of the software-managed memory hierarchy of C64 requires a double buffering strategy where some threads move blocks of 192×192 between off-chip DRAM memory and on-chip SRAM memory while the other threads make computations of 6×6 tiles allocated in registers from the blocks in SRAM. The optimum ratio between data movement threads and computation threads, the optimum sizes of blocks and register tiles and other optimizations applied to this benchmark have been detailed in previous publications for C64 [Garcia et al. 2010]

The impact of the queue choice in the overall application is shown in Figures 11 and 12. As seen in the figures, the choice of queue implementation does not play a critical role when few processors compete for access to the queue. When the number of processors is increased, however, the throughput available at the queue becomes a dominant factor in performance.

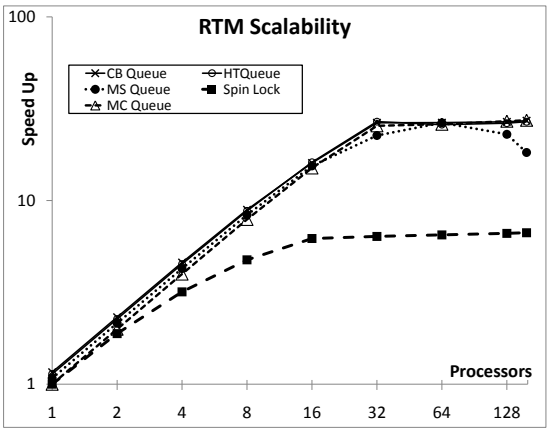

Fig. 11. Reverse Time Migration Results

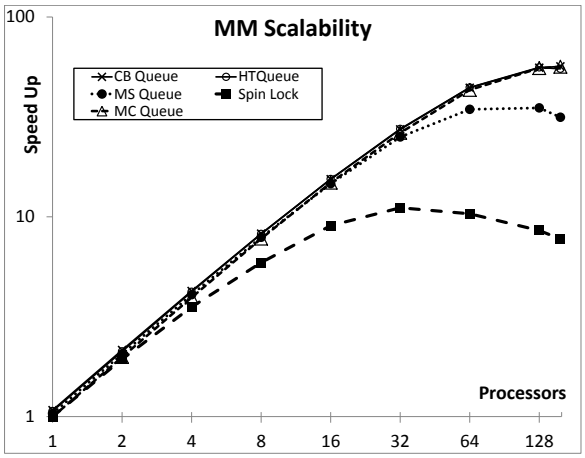

Fig. 12. Matrix Multiply Results

The MC-Queue and the HT-Queue have a high throughput in themselves, but they require calls to malloc and free, which could become the bottleneck. To avoid this, a high performance, distributed memory allocator (described in Section 4.3) was used instead of the standard C64 memory allocator.

Results of MM show that the proposed HT-Queue and the MC-Queue reach a similar maximum speed up: 55.9 for HT-Queue and 56.3 for MC-Queue given their similar theoretical throughput. CB-Queue performance is always slightly better than HT-Queue and MC-Queue, and its maximum speed up is 56.6. RTM shows similar results where the maximum speedup obtained is related to the throughput of the queue used.

6. DISCUSSION

The CB-Queue and the HT-Queue algorithms are fast, they have a very high throughput, they have a very low latency and in general, they are excellent choices to imple-

ment algorithms using queues: They have a much greater throughput than the MS-Queue and the Spin Lock queue and they have a lower latency than the MC-Queue.

The theory developed in this paper ultimately leads to the conclusion that both the MC-Queue and the HT-Queue algorithm have the same intrinsic throughput. Experiments with the microbenchmarks and the applications also confirm this conclusion. However, the HT-Queue algorithm has advantages over the MC-Queue algorithm other than its intrinsic throughput. For example, Figures 9 and 10 show that the HT-Queue has lower latency than the MC-Queue. Other reasons may also be influential: The HT-Queue does not require a Compare and Swap operation while the MC-Queue does. The availability of particular atomic operations such as CAS or atomic addition may restrict the ability to use the MC-Queue or the HT-Queue. Finally, it is a convenience for developers that the HT-Queue performs less frequent calls to memory allocation than the MC-Queue, resulting in better throughput when naive memory allocators are used. Of course, if a good memory allocator is available, the throughput of both will be the same.

The main idea driving the design of high-throughput algorithms was the use of a single atomic operation to make changes as opposed to an inquire-then-update approach. In the particular case of C64, it was straightforward to put a counter and a pointer into a 64 bit word because C64 uses a 32-bit address space. However, the ideas presented here are not limited to a 32-bit address space or to the availability of 64-bit atomic operations because, in general, a pointer can be represented by an integer that uses less bits than the address space requires. Pointers of 64-bits can be represented as integers of 32 bits or less in a variety of ways: as offsets in a large pool of allocated memory, as indexes in a table of pointers, or as indexes in a structure of preallocated elements. Such techniques enable implementations of the queues proposed here in architectures with a 64-bit address space.

The throughput and performance behavior of the algorithms presented are a result of both the algorithms themselves and the architecture where they run. In C64, throughput was supported by in-memory atomic operations. It is possible that algorithm-specific architectural support (such as in-memory enqueue/dequeue operations) can increase the throughput of particular algorithms. However, providing such algorithm-specific hardware is infeasible on general-purpose many-core architectures such as C64 due to their limited usability in general-purpose situations and the increased design complexity and die area required to implement them.

Our use of the same word for a pointer and an integer is different (and it is not related) to solutions to the ABA problem [Michael and Scott 1996]. Solutions to the ABA problem typically place a pointer and an integer timestamp in the same 64 bit word. Our technique is not related to the ABA problem solution because (1) the ABA problem is endemic to algorithms with inquire-then-update approaches which we do not use, and (2) the integer in our technique is used as an offset to the pointer, it is not a timestamp and it is not compared to any other timestamps.

7. CONCLUSIONS AND FUTURE WORK

This paper developed the concept of *intrinsic throughput* of algorithms, which we use to develop the CB-Queue and the HT-Queue. Both algorithms have a large throughput and low latency. Dataflow runtime systems and operating systems can benefit from the CB-Queue implementation. The HT-Queue serves as a viable replacement for traditional queues because it matches their functionality and it exhibits excellent throughput and low latency.

The CB-Queue and the HT-Queue have been shown to have exceptional performance due to their very high throughput and very low latency. High throughput is a result of executing the critical parts of the queue operations *in memory* through the use of

atomic instructions as opposed to attempting inquire-then-update operations that are common to 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.

Space constraints have forced us to limit the number of implementations that we could compare. Nevertheless, we were able to present a theoretical analysis and experiments for 5 different implementations including our own, the one with the highest throughput that we could find (MC-Queue), the nonblocking implementation most used in the industry (MS-Queue) and the traditional Single Lock Queue.

Future work will focus on correctly classifying our algorithms in terms of their properties and in expanding our analysis of intrinsic throughput to a broader range of algorithms.

8. ACKNOWLEDGMENTS

The authors thank Robert Pavel for his valuable suggestions and for his help throughout the production of this paper.

This research was made possible by the generous support of the NSF through grants CCF-0833122, CCF-0925863, CCF-0937907, CNS-0720531, and OCI-0904534.

REFERENCES

BAYSAL, E., KOSLOFF, D. D., AND SHERWOOD, J. W. C. 1983. Reverse time migration. *Geophysics 48*.

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1995. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPOPP '95. ACM, New York, NY, USA, 207–216.

DEL CUVILLO, J., ZHU, W., HU, Z., AND GAO, G. 2005. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. *CAPSL Technical Memo 062*.

DEL CUVILLO, J., ZHU, W., HU, Z., AND GAO, G. R. 2006. Toward a software infrastructure for the cyclops-64 cellular architecture. In *High-Performance Computing in an Advanced Collaborative Environment, 2006*. 9.

DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. 1998. Memory access buffering in multiprocessors. In *25 years of the international symposia on Computer architecture (selected papers)*. ISCA '98. ACM, New York, NY, USA, 320–328.

ET International. http://www.etinternational.com.

GARCIA, E., KHAN, R., LIVINGSTON, K., VENETIS, I., AND GAO, G. 2010. Dynamic percolation - mapping dense matrix multiplication on a many-core architecture. *CAPSL Technical Memo 098*.

GARCIA, E., VENETIS, I. E., KHAN, R., AND GAO, G. 2010. Optimized Dense Matrix Multiplication on a Many-Core Architecture. In *Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010), Part II*. Lecture Notes in Computer Science Series, vol. 6272. Springer, Ischia, Italy, 316–327.

KLEINROCK, L. 1975. *Queueing Systems. Volume 1: Theory*.

MELLOR-CRUMMEY, J. 1987. Concurrent queues: Practical fetch and phi algorithms. *Tech. Rep. 229, Dep. of CS, University of Rochester*.

MICHAEL, M. M. AND SCOTT, M. L. 1996. 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. ACM, New York, NY, USA, 267–275.

MOIR, M., NUSSBAUM, D., SHALEV, O., AND SHAVIT, N. 2005. Using elimination to implement scalable and lock-free fifo queues. In *Proc. of 17th ACM Symp. on Parallelism in Algorithms and Architectures*. SPAA '05. ACM, New York, NY, USA, 253–262.

NGUYEN, A., SATISH, N., CHHUGANI, J., KIM, C., AND DUBEY, P. 2010. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. IEEE Computer Society, Washington, DC, USA, 1–13.

THEOBALD, K. 1999. Earth: An efficient architecture for running threads. Ph.D. thesis.

TSIGAS, P. AND ZHANG, Y. 2001. 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. ACM, New York, NY, USA, 134–143.

ZHANG, Y., ZHU, W., CHEN, F., HU, Z., AND GAO, G. R. 2005. Sequential Consistency Revisit: The Sufficient Condition and Method to Reason the Consistency Model of a Multiprocessor-on-a-Chip Architecture. In *Parallel and Distributed Computing and Networks*. 13–19.