A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems*

Philippas Tsigas
Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg,Sweden
tsigas@cs.chalmers.se

Yi Zhang
Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg,Sweden
yzhang@cs.chalmers.se

ABSTRACT

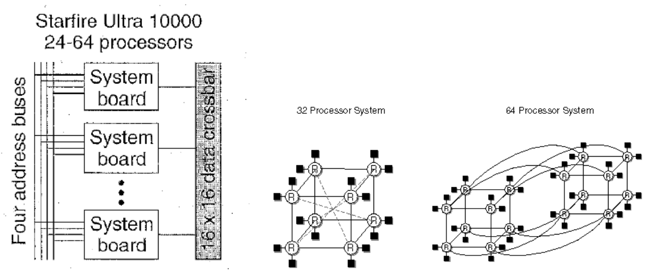
A non-blocking FIFO queue algorithm for multiprocessor shared memory systems is presented in this paper. The algorithm is very simple, fast and scales very well in both symmetric and non-symmetric multiprocessor shared memory systems. Experiments on a 64-node SUN Enterprise 10000 – a symmetric multiprocessor system – and on a 64-node SGI Origin 2000 – a cache coherent non uniform memory access multiprocessor system – indicate that our algorithm considerably outperforms the best of the known alternatives in both multiprocessors in any level of multiprogramming. This work introduces two new, simple algorithmic mechanisms. The first lowers the contention to key variables used by the concurrent enqueue and/or dequeue operations which consequently results in the good performance of the algorithm, the second deals with the pointer recycling problem, an inconsistency problem that all non-blocking algorithms based on the `compare-and-swap` synchronisation primitive have to address. In our construction we selected to use `compare-and-swap` since `compare-and-swap` is an atomic primitive that scales well under contention and either is supported by modern multiprocessors or can be implemented efficiently on them.

1. INTRODUCTION

Concurrent FIFO queue data structures are fundamental data structures used in many applications, algorithms and operating systems for multiprocessor systems. To protect the integrity of the shared queue, concurrent operations that have been created either by a parallel application or by the operating system have to be synchronised. Typically, algorithms for concurrent data structures, including FIFO queues, use some form of mutual exclusion (locking) to synchronise concurrent operations. Mutual exclusion protects the consistency of the concurrent data structure by allowing only one process (the holder of the lock of the data structure) at a time to access the data structure and by blocking all the other processes that try to access the concurrent data structure at the same time. Mutual exclusion and, in general, solutions that introduce blocking are penalised by locking that introduces priority inversion, deadlock scenarios and performance bottlenecks. The time that a process can spend blocked while waiting to get access to the critical section can form a substantial part of the algorithm execution time [5, 9, 10, 14]. There are two main reasons that locking is so expensive. The first reason is the convoying effect that blocking synchronisation suffers from: if a process holding the lock is preempted, any other process waiting for the lock is unable to perform any useful work until the process that hold the locks is scheduled. When we taking into account that the multiprocessor running our program is used in a multiprogramming environment, convoying effects can become serious. The second is that locking tends to produce a large amount of memory and interconnection network contention, locks become hot memory spots. Researchers in the field first designed different lock implementations that lower the contention when the system is in a high congestion situation, and they give different execution times under different contention instances. But on the other hand the overhead due to blocking remained. To address the problems that arise from blocking researchers have proposed non-blocking implementations of shared data structures. Non-blocking implementation of shared data objects is a new alternative approach to the problem of designing scalable shared data objects for multiprocessor systems. Non-blocking implementations allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Since in non-blocking implementations of shared data structures one process is not allowed to block another process, non-blocking shared data structures have the following significant advantages over lock-based ones:

1. they avoid lock convoys and contention points (locks).
2. they provide high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios, where two or more tasks are waiting for locks held by the other.
3. they do not give priority inversion scenarios.

*This work is partially supported by: i) the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council for Engineering Sciences.



(a) The architecture of the SUN Enterprise 10000

(b) The architecture of the Origin 2000

Figure 1: Architectures

Among all the innovative architectures for multiprocessor systems that have been proposed the last forty years shared memory multiprocessor architectures are gaining a central place in high performance computing. Over the last decade many shared memory multiprocessors have been built and almost all major computer vendors develop and offer shared memory multiprocessor systems nowadays. There are two main classes of shared memory multiprocessors: the Cache-Coherent Nonuniform Memory Access multiprocessors (ccNUMA) and the symmetric or Uniform Memory Access (UMA) multiprocessors, their differences coming from the architectural philosophy they are based on. In symmetric shared memory multiprocessors every processor has its own cache, and all the processors and memory modules attach to the same interconnect, which is a shared bus. ccNUMA is a relatively new system topology that is the foundation for next-generation shared memory multiprocessor systems. As in UMA systems, ccNUMA systems maintain a unified, global coherent memory and all resources are managed by a single copy of the operating system. A hardware-based cache coherency scheme ensures that data held in memory is consistent on a system-wide basis. In contrast to symmetric shared memory multiprocessor systems in which all memory accesses are equal in latency, in ccNUMA systems, memory latencies are not all equal, or uniform (hence, the name - Non-Uniform Memory Access). Accesses to memory addresses located on "far" modules take longer than those made to "local" memory.

This paper addresses the problem of designing scalable, practical FIFO queues for shared memory multiprocessor systems. First we present a non-blocking FIFO queue algorithm. The algorithm is very simple, it algorithmically implements the FIFO queue as a circular array and introduces two new algorithmic mechanisms that we believe can be of general use in the design of efficient non-blocking algorithms for multiprocessor systems. The first mechanism restricts contention to key variables generated by concurrent enqueue and/or dequeue operations in low levels; contention to shared variables degrades performance not only in memory tanks where the variables are located but also in the processor-memory interconnection network. The second algorithmic mechanism that this paper introduces is a mechanism that deals with the pointer recycling (also known as ABA) problem, a problem that all non-blocking algorithms based on the `compare-and-swap` primitive have to address. The performance improvements are due to these two mechanisms and to its simplicity that comes from the simplicity and richness of the structure of circular arrays. We have selected to use the `compare-and-swap` primitive since it scales well under contention and either is supported by modern multiprocessors or can be implemented efficiently on them. Last, we evaluate the performance of our algorithm on a 64-node SUN Enterprise 10000 multiprocessor and a 64-node SGI Origin 2000. The SUN system is a Uniform Memory Access (UMA) multiprocessor system while the SGI system is a Cache-Coherent Nonuniform Memory Access (ccNUMA) one; SUN Enterprise 10000 supports the `compare-and-swap` while SGI Origin 2000 does not. The experiments clearly indicate that our algorithm considerably outperforms the best of the known alternatives in both UMA and ccNUMA machines with respect to both dedicated and multiprogramming workloads. Second, the experimental results also give a better insight into the performance and scalability of non-blocking algorithms in both UMA and ccNUMA large scale multiprocessors with respect to dedicated and multiprogramming workloads, and they confirm that non-blocking algorithms can perform better than blocking on both UMA and ccNUMA large scale multiprocessors, and that their performance and scalability increases as multiprogramming increases.

Concurrent FIFO queue data structures are fundamental data structures used in many multiprocessor programs and algorithms and, as can be expected, many researchers have proposed non-blocking implementations for them. Lamport [6] introduced a wait-free queue that does not allow more than one enqueue operation or dequeue operation at a time. Herlihy and Wing in [4] presented an algorithm for a non-blocking linear FIFO queue which requires an infinite array. Prakash, Lee and Johnson in [11] presented a non-blocking and linearisable queue algorithm based on a singly-linked list. Stone describes a non-blocking algorithm based on a circular queue. Massalin and Pu [8] present a non-blocking array-based queue which requires the `double-compare-and-swap` atomic primitive that is available only on some members of the Motorola 68000 family of processors. Valois in [12] presents a non-blocking queue algorithm together with several other non-blocking data structures, his queue is an array-based one. Michael and Scott in [10] presented a non-blocking queue based on a singly-link list, which is the most efficient and scalable non-blocking algorithm compared with the other algorithms mentioned above.

The remainder of the paper is organised as follows. In Section 2 we give a brief introduction to shared memory multiprocessors. Section 3 presents our algorithm together with a proof sketch. In Section 4, the performance evaluation of our algorithm is presented. The paper concludes with Section 5.

2. SHARED MEMORY MULTIPROCESSORS: ARCHITECTURE AND SYNCHRONIZATION

There are two main classes of shared memory multiprocessors; the Cache-Coherent Nonuniform Memory Access (ccNUMA) multiprocessors and the symmetric multiprocessors. The most familiar design for shared memory multiprocessor systems is the "fixed bus" or shared-bus multiprocessor system. The bus is a path, shared by all processors, but usable only by one at a time to handle transfers from CPU to/from memory. By communicating on the bus, all CPUs share all memory requests, and can synchronise their local cache memories. Such systems include the Silicon Graphics Challenge/Onyx systems, OCTANE, Sun's Enterprise (300-6000), Digital's 8400, and many others - most server vendors offer such systems.

Central Crossbar Mainframes and supercomputers have often used a crossbar "switch" to build shared multiprocessor systems with higher bandwidth than feasible with busses, where the switch supports multiple concurrent paths to be active at once. Such systems include most mainframes, the CRAY T90, and Sun's new Enterprise 10000; Figure 1(a) graphically describes the architecture of the new SUN Enterprise 10000. Shared-bus and central crossbar systems are usually called UMAs, or Uniform Memory Access systems, that is, any CPU is equally distant in time from all memory locations. Uniform memory access shared memory multiprocessors dominate the server market and are becoming more common on the desktop. The price of these systems rise quite fast as the number of processors increases.

ccNUMA is a relatively new system topology that is the foundation for many next-generation shared memory multiprocessor systems. Based on "commodity" processing modules and a distributed, but unified, coherent memory, ccNUMA extends the power and performance of shared memory multiprocessor systems while preserving the shared memory programming model. As in UMA systems, ccNUMA systems maintain a unified, global coherent memory and all resources are managed by a single copy of the operating system. A hardware-based cache coherency scheme ensures that data held in memory is consistent on a system-wide basis. I/O and memory scale linearly as processing modules are added, and there is no single backplane bus. The nodes are connected by an interconnect, whose speed and nature varies widely. Normally, the memory "near" a CPU can be accessed faster than memory locations that are "further away". This attribute leads to the "Non" in Non-Uniform. ccNUMA systems include the Convex Exemplar, Sequent NUMA-Q, Silicon Graphics/CRAY S2MP (Origin and Onyx2). In the Silicon Graphics Origin 2000 system a dual-processor node is connected to a router. The routers are connected with a fat hypercube interconnect, Figure 1(b) graphically describes the architecture.

```
LL(p_i, O)
{
Pset(O) := Pset(O) ∪ {p_i}
return value(O)
}

SC(p_i, v, O)
{
if p_i ∈ Pset(O)
      value(O):= v
      Pset(O):=  ∅
      return true
else
      return false
}
```

Figure 2: The `load-linked/store-conditional` primitive

ccNUMA systems are expected to become the dominant systems on large high performance systems over the next few years. The reasons are: i) they scale up to as many processors as needed. b) they support the cache-coherent globally addressable memory model c) their entry level and incremental costs are relatively low.

A widely available hardware synchronisation primitive that can be found on many common architectures is `compare-and-swap`. The `compare-and-swap` primitive takes as arguments the pointer to a memory location, and old and new values. As it can be seen from Figure 3 that describes the specification of the `compare-and-swap` primitive, it automatically checks the contents of the memory location that the pointer points to, and if it is equal to the old value, updates the pointer to the new value. In either case, it returns a boolean value that indicates whether it has succeeded or not. The IBM System 370 was the first computer system that introduced `compare-and-swap`. SUN Enterprise 10000 is one of the systems that support this hardware primitive. Some newer architectures, SGI Origin 2000 included, introduce the `load-linked/store-conditional` instruction which can be implemented by the `compare-and-swap` primitive. The `load-linked/store-conditional` is comprised by two simpler operations, the `load-linked` and the `store-conditional` one. The `load-linked` loads a word from the memory to a register. The matching `store-conditional` stores back possibly a new value into the memory word, unless the value at the memory word has been modified in the meantime by another process. If the word has not been modified, the store succeeds and a 1 is returned. Otherwise the, `store-conditional` fails, the memory is not modified, and a 0 is returned. The specification of this operation is shown in Figure 2. For more information on the SGI Origin 2000 and the SUN ENTERPRISE the reader is referred to [7, 2] and [1], respectively.

```
Compare-and-Swap(int *mem, register old, new)
{
   temp = *mem;
   if (temp == old) {
       *mem = new;
       new = old;
   } else
       new = *mem
}
```

Figure 3: The `Compare-and-Swap` primitive

The `compare-and-swap` primitive though gives rise to the pointer recycling (also known as ABA) problem. The ABA problem arises when a process p reads the value A from a shared memory location, computes a new value based on A, and using `compare-and-swap` updates the same memory location after checking that the value in this memory location is still A and mistakenly concluding that there was no operation that changed the value to this memory location in the meantime. But between the read and the `compare-and-swap` operation, other processes may have changed the context of the memory location from A to B and then back to A again. In this scenario the `compare-and-swap` primitive fails to detect the existence of operations that changed the value of the memory location; in many non-blocking implementations of shared data structures this is something that we would like to be able to detect without having to use the `read_modify_write` operation that has very high latency and creates high contention. A common solution to the ABA problem is to split the shared memory location into two parts: a part for a modification counter and a part for the data. In this way when a process updates the memory location, it also increments the counter in the same atomic operation. There are several drawbacks of such a solution. The first is that the real word-length decreases as the counter now occupies part of the word. The second is that when the counter rounds there is a possibility for the ABA scenario to occur, especially in systems with many, and with fast processors such as the systems that we are studying. In this paper we present a new, very simple efficient technique to overcome the ABA problem; the technique is described in the next section together with the algorithm.

```
Compare-and-Swap(int *mem, register old, new)
{
   do
   {
    temp = LL(mem);
    if  (temp != old)  return FALSE;
   }while(!SC(mem,new));
   return TRUE;
}
```

Figure 4: Emulating `compare-and-swap` from `load-linked/store-conditional`

3. THE ALGORITHM

During the design phase of any efficient non-blocking data structure, a large effort is spent on guaranteeing the consistency of the data structure without generating many interconnection transactions. The reason for this is that the performance of any synchronisation protocol for multiprocessor systems heavily depends on the interconnection transactions that they generate. A high number of transactions causes a degradation in the performance of memory banks and the processor/memory interconnection network.

As a first step, when designing the algorithm presented here, we tried to use simple synchronisation instructions (primitives), with low latency, that do not generate a lot of coherent traffic but are still powerful enough to support the high-level synchronisation needed for the non-blocking implementation of a FIFO queue. In the construction described in this paper we have selected to use the `compare-and-swap` atomic primitive since it meets the three important goals that we were looking for. First, it is a quite powerful primitive and when used together with simple read and write registers is sufficient for building any non-blocking implementation of any "interesting" shared data-structure [3]. Second, it is either supported by modern multiprocessors or can be implemented efficiently on them. Finally, it does not generate a lot of coherent traffic. The only problem with the `compare-and-swap` primitive is that, it gives rise to the pointer recycling (also known as ABA) problem. As a second step, we have tried when designing the algorithm presented here to use the `compare-and-swap` operation as little as possible. The `compare-and-swap` operation is an efficient synchronisation operation and its latency increases linearly with the number of processors that use it concurrently, but still it is a transactional one that generates coherent traffic. On the other hand `read` or `update` operations require a single message in the interconnection network and do not generate coherent traffic. As a third step we propose a simple new solution that overcomes the ABA problem that does not generate a lot of coherent traffic and does not restrict the size of the queue.

Figure 6 and Figure 7 present commented pseudo-code for the new non-blocking queue algorithm. The algorithm is simple and practical, and we were surprised not to find it in the literature. The non-blocking queue is algorithmically implemented as a circular array with a *head* and a *tail* pointer and a special shared variable called the *vnull*. The *vnull* shared variable has been introduced in order to help us to avoid the ABA problem as we are going to see at the end of this section. During the design phase of the algorithm we realised that: i) we could use the structural properties of a circular array to reduce the number of `compare-and-swap` operations that our algorithm uses as well as to overcome more efficiently the ABA problem and ii) all previous non-blocking implementations were trying to guarantee that the *tail* and the *head* pointers always show the real head and tail of the queue but by allowing the *tail* and *head* pointers to lag behind we could even further reduce the number of `compare-and-swap` asymptotically close to optimal. We assume that enqueue operations inserts data at the tail of the queue and dequeue operations remove data from the head of the queue if the queue is not empty. In the algorithm presented here we allow the head and the tail pointers to lag at most m behind the actual head and tail of the queue, in this way only one every m operations has to consistently adjust the tail or head pointer by performing a `compare-and-swap` operation. Since we implement the queue as a circular array, each queue operation that successfully enqueues or dequeues data knows the index of the array where the data have been placed, or have been taken from, respectively; if this index can be divided by m, then the operation will try to update the head/tail of the queue, otherwise it will skip the step of updating the head/tail and let the head/tail lag behind the actual head/tail. In this way, the amortised number of `compare-and-swap` operations for an enqueue or dequeue operation is only $1 + 1/m$, 1 `compare-and-swap` operation per enqueue/dequeue operation is necessary. The drawback that such a technique introduces is that each operation on average will need $m/2$ more read operations to find the actual head or tail of the queue; but if we fix m so that the latency of $(m-1)/m$ `compare-and-swap` operations is larger than the latency of $m/2$ read operations, there will be a performance gain from the algorithm, and these performance gains will increase as the number of processes increases.

It is definitely true that array-based queues are inferior to link-based queues, because they require inflexible maximum queue size. But, on the other hand, they do not require memory management schemes that link-based queue implementations need and they benefit from spatial locality significantly more than link-based queues. Taking these into account and having a a simple, fast and practical implementation in mind we decided to use a cyclical-array in our construction.

We have used the `compare-and-swap` primitive to atomically swing the head and tail pointers from their current value to a new one. For the SGI Origin 2000 system we had to emulate the `compare-and-swap` atomic primitive with the `load-linked/store-conditional` instruction; this implementation is shown in Figure 4. However, using `compare-and-swap` in this manner is susceptible to the ABA problem. In the past researchers have proposed to attach a counter to each pointer, reducing in this way the size of the memory that these pointers can point at efficiently. In this paper we observe that the circular array itself works like a counter mod l where l is the length of the cyclical array, and we can fix l to be arbitrary large. In this way by designing the queue as a circular array we overcome the ABA problem the same way the counters do but without having to attach expensive counters to the pointers, that restrict the pointer size. Henceforth, when an enqueue operation takes place, the tail changes in one direction and goes back to zero when it reaches the end of the array. Henceforth, the tail will change back to the same old value after the shared object finishes l enqueue operations and not after two successive operations (exactly as when using a counter mod l). The same also holds for the dequeue operations.

The atomic operations on the array are other potential places where the ABA problem can take place giving rise to the following scenario:

1. the array location x is the actual tail of the queue and its content is $Null$[1]
2. processes a and b find the actual tail

[1] the cell is empty

```
# MAXNUM is l, the length of the cyclical
structure Queue
  {head: unsigned integer,
   nodes: array[0..MAXNUM+1] of pointer,
   tail: unsigned integer,
   vnull: unsigned integer}
newQueue(): pointer to Queue
  Queue *temp;
  temp = (Queue *) malloc( sizeof(Queue));
  temp->head = 0;
  temp->tail = 1;
  #we define another NULL
  temp->vnull = NULL(1);
  for (i=0;i<=MAX_NODES;i++)
      #NULL means empty
      temp->nodes[i]=NULL(0);
  temp->nodes[0] = NULL(1);
  return temp;
```

Figure 5: Initialisation

3. process a enqueues data and updates the content of location x with the use of `compare-and-swap`. Since the contents of x is $Null$, a succeeds
4. process c dequeues data and updates the content of location x to $Null$, changing also the pointer head
5. process b enqueues data and updates the contents of location x with the use of `compare-and-swap`. Since the content of x is $Null$, b incorrectly succeeds to enqueue a non active cell in the queue.

In order to overcome this specific ABA instance instead of using a counter with all the negative side-effects, we introduce a new simple mechanism that we were surprised not to find in the literature. The idea is very simple; instead of using one value to describe that an entry in the array is empty we use two, $NULL(0)$ and $NULL(1)$. When a processor dequeues an item, it will swap into the cell one of the two $NULLs$ in such a way that two consecutive dequeue operations on the same cell give different $NULL$ values to the cell. Returning to the ABA scenario described above, the scenario would now look like this:

1. array location x is the actual tail and it's content is $NULL(0)$
2. processes a and b find the actual tail, ie. location x
3. process a enqueues data and updates the content of location x with a `compare-and-swap` operation. Since $x's$ content is $NULL(0)$, a succeeds
4. process c dequeues data and updates the content of location x to $NULL(1)$
5. process b enqueues data and updates the content of location x with `compare-and-swap`. As the content is $NULL(1)$, b fails in this turn.

```
Enqueue(t: pointer to Queue, newnode:
            pointer to data type):Boolean
loop
    te = t->tail;          #read the tail
    ate = te;
    tt = t->nodes[ate];
    #the next slot of the tail
    temp = (ate + 1) & MAXNUM;
    #we want to find the actual tail
    while (tt<>NULL(0) AND tt<>NULL(1)) do
        #check tail's consistency
        if (te != t->tail) break;
#if tail meet head,
# it is possible that Queue is full
        if (temp == t->head) break;
        #now check the next cell
        tt = t->nodes[temp];
        ate = temp;
        temp = (ate + 1) & MAXNUM;
    end while
    #check the tail's consistency
    if (te != t->tail) continue;
    #check whether Queue is full
    if (temp == t->head)
        ate = (temp + 1) & MAXNUM;
        tt = t->nodes[ate];
        #the cell after head is OCCUPIED
        if (tt<>NULL(0) AND tt<>NULL(1))
            return FAILURE;       #Queue Full
        #if head rewind try update null
        if (!ate)
            t->vnull = tt;
        #help the dequeue to update head
        cas(&t->head,temp,ate);
        #try enqueue again
        continue;
    end if
    #check the tail consistency
    if (te != t->tail) continue;
    #get the actual tail and try enqueue data
    if (cas(&(t->nodes[ate]),tt,newnode))
       if (temp%2==0)     #enqueue has succed
          cas(&(t->tail),te,temp);
       return SUCCESS;
    end if
endloop
```

Figure 6: The enqueue operation

A variable, *vnull* is used to help the dequeue operations to determine which $NULL$ they should use any time.

With this mechanism the ABA scenario that was taking place before, when a process was preempted by only one other process, now changes to an $ABA'BA$ scenario. The $ABA'B'A$ scenario is still a pointer recycling problem, but in order to take place l dequeue operations are needed to take the system from A to B and subsequently to A', after that l more dequeue operations are needed in order to take the system from A' to B' and then to A. Moreover, all these operations have to take place while the process that will experience the pointer recycling is preempted. Taking into account that l is an arbitrary large number, the probability that the above $ABA'B'A$ scenario can happen can go as close to 0 as we want[2].

The above sketches a proof of the following theorem:

THEOREM 1. *The algorithm does not give rise to the pointer recycling problem, if an enque or dequeue operation can not be preempted by more than l operations, l is an arbitrary large number.*

For the rest of these paper we assume that we have selected l to be large enough not to give rise to the pointer recycling problem in our system.

The accessing of the shared object is modelled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, an Enqueue operation or a Dequeue operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $<_h$ are called *overlapping*. A complete history h is linearisable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_h$ that respects the specification of the object [4].

Any possible history, produced by our implementation, can be mapped to a history where operations use an auxiliary array that is not bounded on the right side. In order to simplify the proof we will use this new auxiliary array. Our algorithm guarantees that enqueue operations enqueue data at consecutive array entries from left to right on this array, and dequeue operations dequeue items also from left to right. In this way it makes sure that the operations are dequeued in the order they have been enqueued. From the previous theorem we also have that when an $Enqueue(x)$ operation finishes after writing x to some entry e of the array, the head pointer of our implementation, that guides the dequeue operations, will not over-pass this entry e, thus making sure that no enqueued item is going to be lost. The above sketches a proof for the next theorem:

THEOREM 2. *In a complete history such that $Enqueue(x) \rightarrow Enqueue(y)$, either $Dequeue(x) \rightarrow Dequeue(y)$ or $Dequeue(y)$ and $Dequeue(x)$ overlap.*

[2] We should point out that the technique of using 2 different $NULL$ values can be extended to k different values requiring more than $k * l$ dequeue operations to preempt an operation inorder to cause the pointer recycling problem. We think that the scheme with $2 * NULL$ values is simple enough and sufficient for the systems that we are looking at.

The dequeue operation that dequeues x is the only one that succeeds to read and "empty" the array entry where x was written, because of the atomic `compare-and-swap` operation; making in this way sure that no other operation dequeues the same item. Moreover, since the array entry was written by an enqueue operation, the dequeue operations will always dequeue items that have been "really" enqueued. The above sketches the proof of the following theorem:

THEOREM 3. *If x has been dequeued, then it was enqueued, and $Enqueue(x) \rightarrow Dequeue(x)$*

The last 2 theorems guarantee the linearizable behaviour of our FIFO queue implementation [4]. Due to space constraints, we only sketched the proof of these theorems.

4. PERFORMANCE EVALUATION

We implemented our algorithm and conducted our experiments on a SUN Enterprise 10000 with 64 250MHz UltraSPARC processors and on a SGI Origin 2000 with 64 195MHz MIPS R10000 processors. The SUN multiprocessor is a symmetric multiprocessor while the SGI multiprocessor is a ccNUMA one. To ensure accuracy of the results, we had exclusive access to the multiprocessors while conducting the experiments. For the tests we compared the performance of our algorithm (new) with the performance of the algorithm by Michael and Scott (MS) [10] because their algorithm appears to be the best non-blocking FIFO queue algorithm. In our experiments, we also included a solution based on locks (ordinary lock) to demonstrate the superiority of non-blocking solutions over blocking ones.

4.1 Experiments on SUN Enterprise 10000

We have conducted 3 experiments on the SUN multiprocessor, in all of them we had exclusive use. In the first experiment we measured the average time taken by all processors to perform one million pairs of enqueue/dequeue operations. In this experiment (Figure 8a) a process enqueues an item and then dequeues an item and then it repeats. In the second experiment (Figure 8b) processes stay idle for some random time between each two consecutive queue operations. In the third experiment we used parallel quick-sort, that uses a queue data structure, to evaluate the performance of the three queue implementations. Parallel quick-sort had to sort 10 million randomly generated keys. The results of this experiment are shown in Figure 8c. The horizontal axis in the figures represent the number of processors, while the vertical one represents execution time normalised to that of Michael and Scott algorithm.

The first two experiments (on 58 processors), show that the new algorithm outperforms the MS algorithm by more than 30% and the spin-lock algorithm by more than 50%. The third experiment shows that the new queue implementation offers 40% better response time to the sorting algorithm.

4.2 Experiments on the SGI multiprocessor

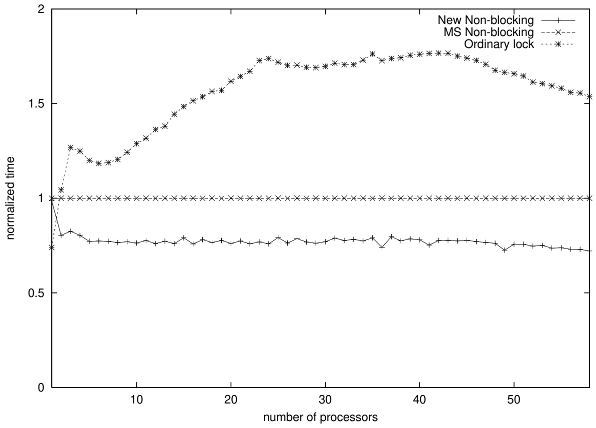
On the SGI machine, the first three experiments were basically the same experiments that we performed on the SUN multiprocessor. The only difference is that on the SGI machine we could select to use the system as a dedicated system (multiprogramming level one) or as a multiprogrammed

```
Dequeue(t: pointer to Queue, oldnode:
         pointer of pointer to data type)
loop
  th = t->head;      #read the head
  #here is the one we want to dequeue
  temp = (th + 1) & MAXNUM;
  tt = t->nodes[temp];
  # find the actual head after this loop
  while (tt==NULL(0) OR tt==NULL(1)) do
      #check the head's consistency
      if (th != t->head) break;
      #two consecutive NULL means EMPTY return
      if (temp == t->tail) return 1;
      temp = (temp + 1) & MAXNUM;  #next cell
      tt = t->nodes[temp];
  end while
  #check the head's consistenicy
  if (th != t->head) continue;
  #check whether the Queue is empty
  if (temp == t->tail)
    #help the enqueue to update end
    cas(&t->tail,temp,(temp+1) & MAXNUM);
    continue;      #try dequeue again
  end if
  #if dequeue rewind to 0,
  #switching NULL to avoid ABA
  if (temp)
    if (temp < th)
      tnull = t->nodes[0];
    else
      tnull = t->vnull;
  else
    tnull = (void *) (((unsigned ) t->vnull) ^ 1);
  #check the head's consistency
  if (th != t->head) continue;
#Get the actual head, null value means empty
  if (cas(&(t->nodes[temp]),tt,tnull))
     #if dequeue rewind to 0,
     #switch NULLs to avoid ABA
     if(!temp) t->vnull = tnull;
     if ((temp%2)==0) cas(&(t->head),th,temp);
     *oldnode = tt;       #return the value
     return 0;
  end if
endloop
```

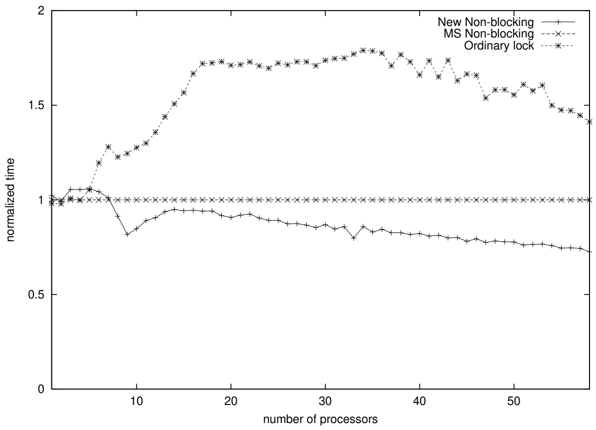
Figure 7: The dequeue operation

system with two and three processes per processor (multiprogramming level two and three respectively). For the SUN multiprocessor this was not possible. Figures 9, 10 and 11a show graphically the performance results. What is very interesting is that our algorithm gives almost the same performance improvements on both machines.

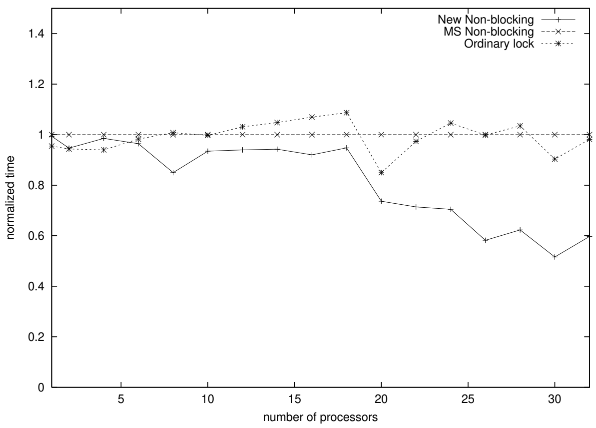
On the SGI multiprocessor, it was possible to use the radiosity from SPLASH-2 shared-address-space parallel applications[13]. Figure 11b shows the performance improvement compared with the original SPLASH-2 implementation. The vertical axis represents execution time normalised to that of the SPLASH-2 implementation.



(a) on the SUN Starfire with full contention

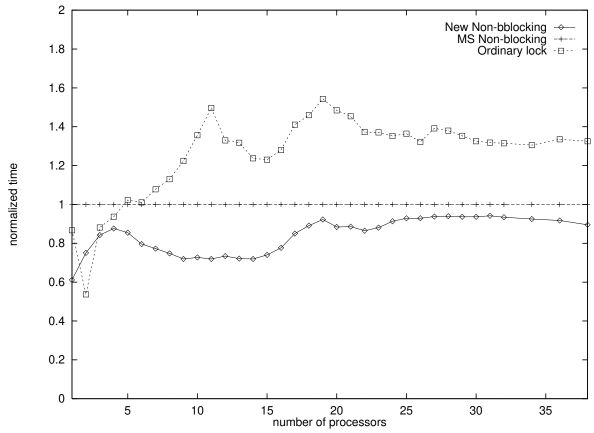


(b) on the SUN Starfire with random waiting contention

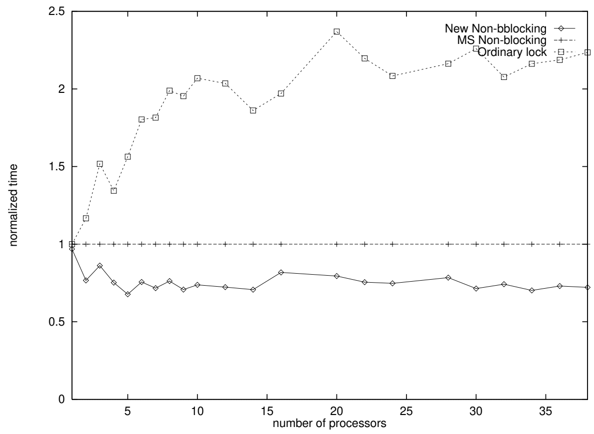


(c) Quick-sort on SUN Starfire

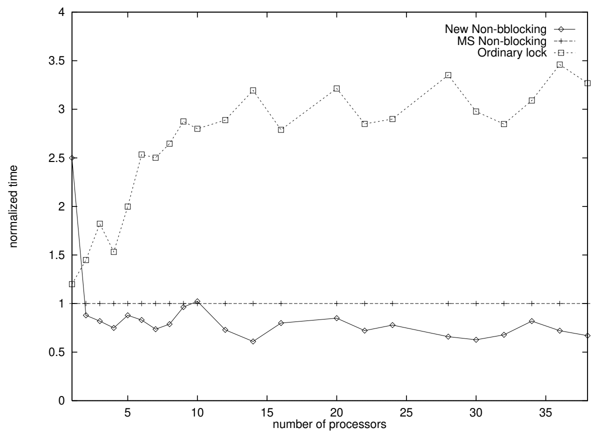
Figure 8: Results on the SUN multiprocessor



(a) Level one

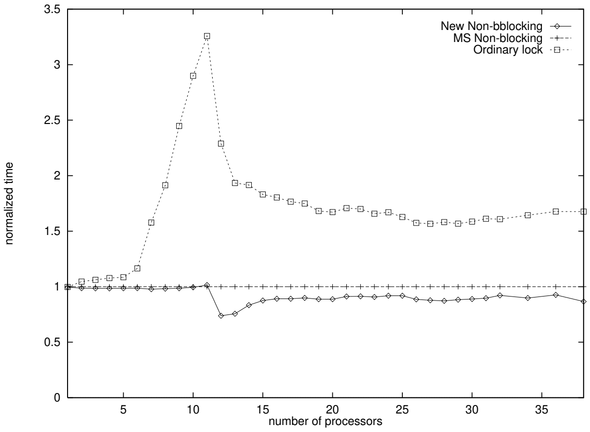


(b) Level 2

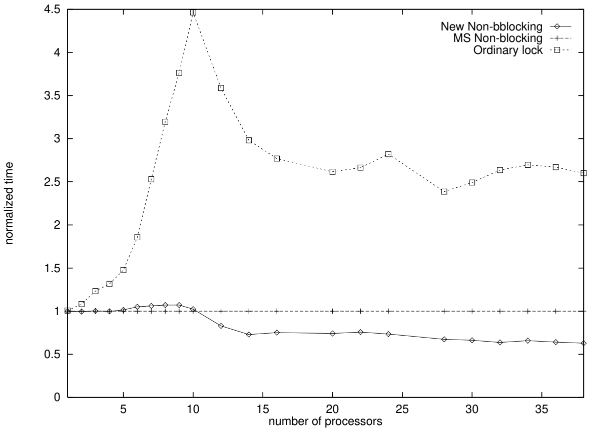


(c) Level 3

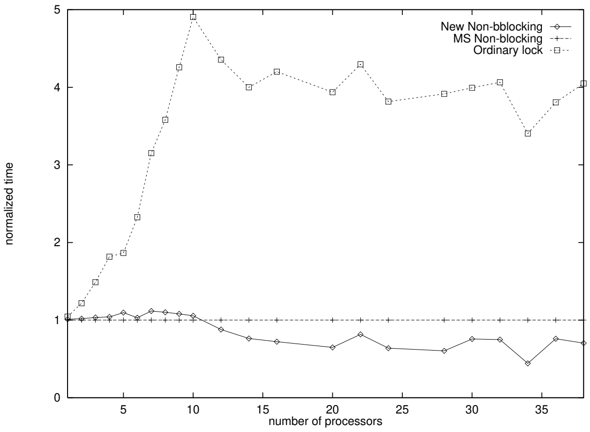
Figure 9: Results on the SGI multiprocessor with different multiprogramming levels under full contention



(a) Level 1

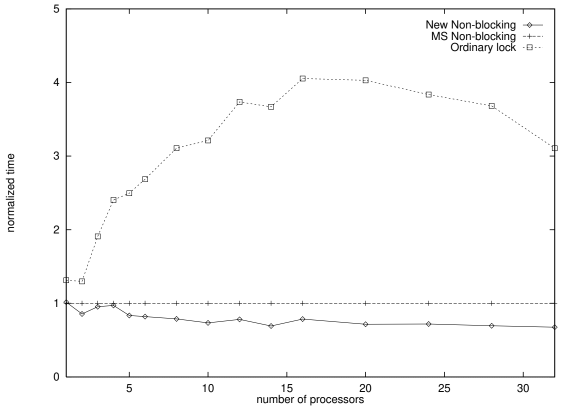


(b) Level 2

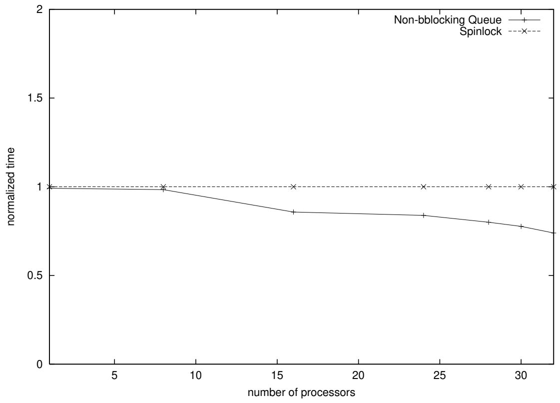


(c) Level 3

Figure 10: Results on the SGI multiprocessor with different multiprogramming levels under random waiting contention



(a) Quick-sort



(b) Radiosity

Figure 11: Applications on SGI

5. CONCLUSIONS

In this paper we presented a new bounded non-blocking concurrent FIFO queue algorithm for shared memory multiprocessor systems. The algorithm is simple and introduces two new simple algorithmic mechanisms that can be of general use in the design of efficient non-blocking algorithms. The experiments clearly indicate that our algorithm considerably outperforms the best of the known alternatives in both UMA and ccNUMA machines with respect to both dedicated and multiprogramming workloads. The experimental results also give a better insight into the performance and scalability of non-blocking algorithms in both UMA and ccNUMA large scale multiprocessors with respect to dedicated and multiprogramming workloads, and they confirm that non-blocking algorithms can perform better than blocking on both UMA and ccNUMA large scale multiprocessors.

Acknowledgements

We would like to thank David Rutter for his great help during the writing phase of this paper. We are grateful to Carl Hallen, Andy Polyakov and Paul Waserbrot, they made the impossible possible and at the end we could have exclusive access to our heavily (thanks to our physics department) loaded parallel machines.

6. REFERENCES

[1] A. Charlesworth. Starfire — extending the SMP envelope. *IEEE Micro*, 18(1):39–49, 1998.

[2] D. Cortesi. Origin 2000 and onyx2 performance tuning and optimization guide. http://techpubs.sgi.com/library/, SGI Inc., 1998.

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

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

[5] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles Operating Systems Review (13th SOSP 1991)*, pages 41–55, Pacific Grove, CA, Oct. 1991.

[6] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, Apr. 1983.

[7] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New YOrk, June 2–4 1997. ACM Press.

[8] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

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

[10] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 25 May 1998.

[11] S. Prakash, Y. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43:548–559, May 1994.

[12] J. D. Valois. *Lock-Free Data Structures.* PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

[14] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel processors. *IEEE Transactions on Parallel and Distributed Systems*, PDS-2(2):180–198, Apr. 1991.