Non-Blocking Concurrent FIFO Queues With Single Word Synchronization Primitives

Claude Evequoz
University of Applied Sciences Western Switzerland
1400 Yverdon-les-Bains, Switzerland
Claude.Evequoz@heig-vd.ch

Abstract

We present 2 efficient and practical non-blocking implementations of a concurrent array-based FIFO queue that are suitable for both multiprocessor as well as preemptive multithreaded systems. It is well known that concurrent FIFO queues relying on mutual exclusion cause blocking, which have several drawbacks and degrade overall system performance. Link-based non-blocking queue algorithms have a memory management problem whereby a removed node from the queue can neither be freed nor reused because other threads may still be accessing the node. Existing solutions to this problem introduce a fair amount of overhead and, when the number of threads that can access the FIFO queue is moderate to high, are shown to be less efficient compared to array-based algorithms, which inherently do not suffer from this problem. In addition to being independent on advance knowledge of the number of threads that can access the queue, our new algorithms improve on previously proposed algorithms in that they do not require any special instruction other than a load-linked/store-conditional or a compare-and-swap atomic instruction both operating on pointer-wide number of bits. Our new algorithms are thus portable to a broader range of architectures.

Keywords: Concurrent queue, lock-free, compare-and-swap (CAS), load-linked/store-conditional (LL/SC).

1. Introduction

Lock-free data structures have received a large amount of interest as a mechanism that ensures that the shared data is always accessible to all threads and a temporarily or permanently inactive thread cannot render the data structure inaccessible. A concurrent data structure implementation is *non-blocking* (or *lock-free*) if it guarantees that at least one thread is guaranteed to finish its operation on the shared objects in a finite number of steps, even if there are other halted or delayed threads currently accessing the shared data structure. By definition, non-blocking implementations have no critical sections in which preemption can occur. These data structures also do not require any communication with the kernel and have been repeatedly reported to perform better than their counterparts implemented with critical sections [10].

First-in-first-out (FIFO) queues are an important abstract data structure lying at the heart of most operating systems and application software. They are needed for resource management, message buffering and event handling. As a result, the design of efficient implementations of FIFO queues has been widely researched. A FIFO queue supports 2 operations: an *enqueue* operation inserts a new item at the *tail* of the queue, and a *dequeue* operation removes an item from the *head* of the queue if the queue is not empty.

This paper addresses the problem of designing practical non-blocking FIFO queues based on a bounded circular array using only widely available pointer-wide atomic instructions. We begin by reviewing previous work done on FIFO queues and the memory reclamation problem inherent in link-based algorithms. Of particular interest in that section are algorithms that can adapt to a varying number of threads; these algorithms are called *population-oblivious* [5]. Section 3 presents the problems that must to be dealt with when designing non-blocking circular array FIFO queues. Section 4 introduces our first algorithm, which is population-oblivious and has a space consumption depending only on the number of items in the queue. Our second algorithm, presented in Section 5, applies to a broader range of architectures but unlike our first algorithm, the space consumption also depends on the maximum number of threads that ac-

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.82

cessed the queue at any given time. This algorithm is based on the popular compare-and-swap (CAS) instruction, which takes 3 parameters: the address of a memory location, an expected value, and a new value. The new value is written into the memory location if and only if the location holds the expected value and the returned value is a Boolean indicating whether the write occurred. Performance evaluations of our algorithms are conducted in Section 6. The paper concludes in Section 7.

2. Related Work

Practical algorithms of non-blocking FIFO queues fall into two categories. The first category consists of algorithms based on finite or infinite arrays. Herlihy and Wing [3] gave a non-blocking FIFO queue algorithm requiring an infinite array. Wing and Gong [16] later removed the requirement of an infinite array. In their implementation, the running time of the dequeue operation is proportional to the number of completed enqueue operations since the creation of the queue. Treiber [13] also proposed a similar algorithm that does not use an infinite array. Although the enqueue operation requires only a single step, the running time needed for the dequeue operation is proportional to the number of items in the queue. These last two algorithms are inefficient for large queue lengths and many dequeue attempts. Valois [15] also presented an algorithm based on a bounded circular array. However, both enqueue and dequeue operations require that two array locations which may not be adjacent be simultaneously updated with a CAS primitive. Unfortunately this primitive is not available on modern processors. Shann *et al*. [12] present an efficient FIFO queue based on a circular array where each array element stores 2 fields: a data field and a reference counter field that prevents the so-called ABA problem (see section 3). Their algorithm is useful for processors that offer atomic instructions that can manipulate an array element as a whole. Because certain 32-bit architectures (e.g., PowerPC and Pentiums) support 32- and 64-bit atomic instructions, the data field may also represent a pointer to a record when there is a need to expand the data field size. Current and emerging 64-bit architectures do not provide atomic access to more than 64-bit quantities, thus it is no longer possible to pack a large reference counter along with pointer-wide values in 64-bit applications. When application software exploiting these 64-bit capabilities becomes widespread [2], their algorithm will be of limited use. Tsigas and Zhang [14] proposed the first practical non-blocking FIFO queue based on a circular array using single word synchronization primitives found on all modern architectures and suitable for 64-bit applications. Their algorithm applies only to queued items that are pointers to data and they show that it outperforms link-based FIFO queues. However, for queueing operations to appear as FIFO (*linearizability* property [3]), the algorithm assumes that an enqueue or a dequeue operation cannot be preempted by more than s similar operations, where s is the array size. Their algorithm is therefore not population-oblivious.

The second category of FIFO queues is implemented by a linked list of queued nodes. Michael and Scott [9] proposed an implementation based on a single-linked list where an enqueue operation requires 2 successful CAS operations and a dequeue operations needs a single successful CAS. More recently, Ladan-Mozes and Shavit [6] presented an algorithm based on a doubly-linked list requiring one successful atomic synchronization instruction per queue operation. Although there are more pointers to update, these are preformed by simple reads and writes. They show that their algorithm consistently performs better than the single-linked list suggested in [9].

Although the advantage of linked-based FIFO queues over array-based implementations is that the size of the queue and the number of nodes it holds may vary dynamically, these queues are subject to a memory management problem. A dequeued node can be freed and made available for reuse only when the dequeuer is the only thread accessing the node. The easiest approach to deal with this problem is to ignore it and assume the presence of a garbage collector. However not all systems and languages provide garbage collector support. Another approach is to never free the node and to store it in a free pool for subsequent reuse once it is dequeued. When a new node is required, the node is obtained from the free pool. An important drawback of this approach is that the actual size of FIFO is equal to the maximum queue size since its initialization and is not really dynamically sized. Valois [15] presented an approach that actually frees a dequeued node. The mechanism associates a reference counter field with each node. Each time a thread accesses a node, it increments the node's reference counter; when the thread no longer accesses the node, it decrements the counter. A node can be freed only if the value of its reference counter drops to zero. Although the scheme is simple, a basic problem arises making this scheme impractical. The scheme proposed in [8,15] involves 3 steps: (1) a pointer is set to the node that is to be accessed, (2) the reference counter of the possibly reclaimed node is then incremented, and (3) the pointer is verified that it still points to the correct node. Should the verification step (3) fail, the reference counter is decremented and all three steps repeated. Note that the reference counter of a node can be accessed and modified even after it has been freed.

None of the reclaimed node can thus be definitely released to the memory allocator and reused for arbitrary purposes without possibly corrupting memory locations; all must again be stored in some free pool. Detlefs *et al*. [1] alleviate the above problem by performing steps (1) and (2) atomically. But because the reference to a node and its associated reference counter are not contiguous in memory, the needed primitive requires the atomic update of two arbitrary memory locations that is not supported in hardware by any modern processor.

Michael [10] presented a lock-free memory management technique that allows safe memory reclamation. Whenever a thread is about to reference a node, it publishes the address of the node in a global memory location. When the dequeuer removes a node from the queue, it scans the published accesses of the other threads. If a match is not found, the node may be safely freed. Otherwise the node is stored until a subsequent scan. A similar scheme, but not using pointer-wide instructions, was also independently proposed by Herlihy *et al*. [4].

Doherty *et al*. [2] present the first link-based FIFO queue that is population-oblivious and has a space consumption depending only on the number of queued items and the number of threads currently accessing the queue. However, their algorithm introduces significant overheads and trades memory space for computational time.

3. ABA Problem

The ABA problem is a well-known problem in the context of data structures implemented by CAS statements. The desired effect of a CAS operation is that it should succeed only if the value of a location does not change since the previous reading of the contents of the location. A thread may read value *A* from a shared location and then attempt to modify the contents of the location from *A* to some other value. However it is possible that between the read and the CAS other threads change the contents of the location from *A* to *B* and then back to *A* again. The CAS therefore succeeds when it should fail. The semantics of read and CAS prevent them from detecting that a shared variable has not been written after the initial read.

In a circular list based on a finite array, there are 3 different ABA problem sources. First, the `Head` and `Tail` indices are each prone to the ABA problem, which we call the *index*-ABA problem. Next, each slot in the array holds 2 different value types: a queued data item and a null item that indicates that the slot is empty and available for reuse. Each of these 2 data values gives rise to an ABA problem that may be solved differently. In order to distinguish them, we refer to them as the *data*-ABA and the *null*-ABA problem respectively. In the following we illustrate an instance of how each identified ABA problem manifests itself and the means to elude the problem.

The enqueue and dequeue operations increment their respective indices once they have inserted or removed an item in or from the array. If these operations are delayed immediately prior to the increment but after modifying the contents of the array, other threads may successfully complete $s-1$ identical operations and leave the index concerned by the delayed operation in the same state, where s is the size of the circular array. When the delayed operation resumes, it wrongly adjusts its index. Figure 1 illustrates such a scenario.

The index-ABA problem can easily be dealt with if we let each counter occupy a word and only increment these counters. The enqueue and dequeue functions can then map the counter into a valid index before accessing an array slot with a modulo operator. Although this solution does not guarantee that the ABA problem will not occur, its likelihood is extremely remote.

A simple example of the data-ABA problem can be given for an array having 2 slots. Assume that the array initially contains a single item *A*. Since a dequeue operation must first read the contents of the slot before removing it, a dequeuer may read item *A* and then be preempted before it gets a chance to remove *A* from the array. During its preemption, another thread may dequeue item *A* and then successively enqueue items *B* and *A*. The array is now full and when the preempted dequeue operation resumes, it wrongly removes item *A* instead of *B*. The implementation proposed in [14] circumvents this difficulty by assuming that the duration of preemption cannot be greater than the time for the indices to rewind themselves. This assumption may result into an exceedingly oversized array or be impossible to meet when the upper bound on the number of threads is unknown.

If we assume for the ease of explanation that the array is infinite, the array can be divided into 3 consecutive intervals: a possibly empty succession of empty slots that held removed items, a possibly empty series of enqueued items, and finally a series of empty slots that never held any item. A null-ABA problem occurs when an enqueuer mistakenly inserts an item into a free slot that belongs to the first interval. An enqueuer reads the contents of the first slot in the 3rd interval, notices that it is empty but gets preempted before inserting its item in the slot. Another thread may then insert an item and dequeue all the items in the array. When the enqueuer resumes, it incorrectly inserts its item in the first interval of the array. This flaw is corrected in [14] by cleverly having 2 empty indicators. Initially the array slots are set to $null_0$ (3rd interval) and once items are removed from the array the slots are

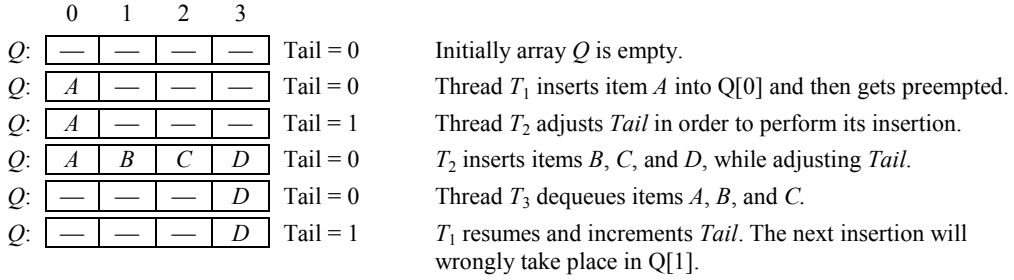

Figure 1. Scenario with 3 threads illustrating the index-ABA problem

marked with $null_1$ to become part of the 1st interval. When the head index rewinds to 0, the interpretations of the null values are switched from their corresponding intervals. The most common solution to this and other ABA problems is to split each shared memory location into 2 parts that are accessed simultaneously: a part for a version number or counter and a part for the shared data item; when a thread updates the memory location it also increments the counter in the same atomic operation. Because the version number is not unbounded, this technique does not guarantee the ABA scenario will not occur but it makes it extremely unlikely. Shann *et al*. [12] rely on this technique to solve the data-ABA and null-ABA problem for their FIFO queue implemented by a circular array. Algorithm designers using this technique usually assume that the version number and the data item each occupy a single word and that the architecture supports double-word atomic operations. In practice, this assumption may be valid for some 32-bit architectures, but it is invalid for the emerging 64-bit architectures.

4. First Algorithm

Modern instruction-set architectures, such as ARMv6 and above, DEC Alpha, MIPS II and PowerPC processors, do not provide atomic primitives that read and update a memory location in a single step because it is much more complex than a typical RISC operation and it is difficult to pipeline. Instead these processors provide an alternative mechanism based on two atomic instructions, *load-linked* (LL, also called load-lock or load reserve) and *store-conditional* (SC). The usual semantics of LL and SC assumed by most algorithm designers are given in Figure 2. A shared variable X accessed by these instructions can be regarded as a variable that has an associated shared set of thread identifiers $valid_X$, which is initially empty. Instruction LL returns the value stored in the shared location X and includes the calling thread identifier in set $valid_X$. Instruction SC checks if the calling thread's identifier is in $valid_X$, and if so, clears $valid_X$ and updates the location before returning success; otherwise the instruction returns failure.

LL(X) ≡ $valid_X \cup \{threadID\}$; **return** X
SC(X,Y) ≡ **if** $threadID \in valid_X$ **then** $valid_X \leftarrow \emptyset$;
$X \leftarrow Y$; **return** *true*
else return *false*
end if

Figure 2. Equivalent atomic statements specifying the theoretical semantics of LL/SC

Based on the semantics of the LL and SC instructions, we can design a FIFO queue that is immune to ABA problems. Figure 3 shows our algorithm in a self-explanatory pidgin C notation. For clarity, various type casts are missing but pointer dereferencing obey strict C notation. Following standard usage, all global variables have identifiers beginning with an uppercase letter and variables completely in lowercase are local to an operation.

Our FIFO queue is implemented as a circular list by an array of `Q_LENGTH` slots named `Q`, along with two indices named `Head` and `Tail`. An array slot contains either a pointer to a data item or the value *null* to indicate that it is currently free. `Head` refers to the first slot of the queue that may hold an item. `Tail` designates the next free slot where a new item can be inserted. The queue is empty when `Head` is equal to `Tail`, and it is full when `Head + Q_LENGTH` is equal to `Tail`. We assume that `Q_LENGTH` is a power of 2 so that `Head` and `Tail` can wraparound without skipping array slots. Finally, the array slots are initialized to *null* and the indices are set to 0 prior to calling an enqueue or a dequeue operation.

To add a data item, the enqueue operation first reads the current `Tail` value (line E5) and reserves the slot it intends to insert the item into (line E9). The test on line E10 verifies that the reserved slot still corresponds to the one designated by `Tail`, and its purpose is to avoid the null-ABA problem. After successfully reserving an array slot, the enqueuer checks the contents of the slot. If it is empty, the enqueuer tries to insert its item into it (line E15). On the other hand, if the reserved slot is not empty, another concurrent thread has successfully inserted an item but was preempted before it had the chance to update `Tail` and the `Tail` value read on line E5 is lagging behind. In this case, the en-

```
     Q: array[0..Q_LENGTH-1] of *NODE;   // Circular list initialized with null
     unsigned int Head, Tail;            // Extraction and insertion indices

E1:  BOOL Enqueue(NODE *node) {
E2:    unsigned int t, tail;
E3:    NODE *slot;
E4:    while (true) {
E5:       t = Tail;
E6:       if (t == Head + Q_LENGTH)
E7:          return FULL_QUEUE;
E8:       tail = t % Q_LENGTH;
E9:       slot = LL(&Q[tail]);
E10:      if (t == Tail)
E11:         if (slot != null) {
E12:            if (LL(&Tail) == t)
E13:               SC(&Tail,t+1);
E14:         }
E15:         else if (SC(&Q[tail],node)) {
E16:            if (LL(&Tail) == t)
E17:               SC(&Tail,t+1);
E18:            return OK;
E19:         }
E20:   }
E21: } /* end of Enqueue */

D1:  NODE *Dequeue(void) {
D2:    unsigned int h, head;
D3:    NODE *slot;
D4:    while (true) {
D5:       h = Head;
D6:       if (h == Tail)
D7:          return null;
D8:       head = h % Q_LENGTH;
D9:       slot = LL(&Q[head]);
D10:      if (h == Head)
D11:         if (slot == null) {
D12:            if (LL(&Head) == h)
D13:               SC(&Head,h+1);
D14:         }
D15:         else if (SC(&Q[head],null)) {
D16:            if (LL(&Head) == h)
D17:               SC(&Head,h+1);
D18:            return slot;
D19:         }
D20:   }
D21: } /* end of Dequeue */
```

Figure 3. ABA problem-free implementation of a FIFO queue

queuer helps the delayed thread and advances the `Tail` index (line E13) on its behalf before restarting its loop.

The dequeue operation removes the oldest item in the queue and returns it to its caller. The first 10 lines of the dequeue operation are identical to those of the enqueue operation: A dequeuer reads and reserves the array slot identified by the current `Head` index value. The test on line D10 confirms that the marked slot is indeed the oldest item that can be removed. Without this check, a dequeuer can read the current `Head` index, say *h*, and be preempted anywhere between lines D5 and D10. During this preemption, other threads can enqueue and dequeue items wrapping the array as they do so. When the preempted dequeuer resumes its execution and carries out the LL operation on line D9, Q[*h*] may hold an item that is not the oldest. Figure 4 illustrates such a scenario for a queue with 5 slots.

After the test on line D10, the slot may be in one of two states. If it is empty, the dequeuer can infer that the `Head` index is falling behind since the dequeuer checked that the queue was not empty (line D6) before attempting to remove an item. In this case, the dequeuer tries to update `Head` before restarting from the beginning. On the other hand, if the slot is not empty, the dequeuer attempts to substitute the slot for a *null* and then, if it succeeds, tries to update `Head` before returning the dequeued item.

5. Second Algorithm

Unfortunately, all architectures that support LL/SC instructions do not support the full theoretical semantics that we have described in Figure 2. These architectures have one or several of the following limitations [11]:

1. There can be neither nesting nor interleaving of LL/SC pairs.
2. There can be no memory access between a LL/SC pair.
3. The cache coherence mechanism may allow the SC instruction to fail spuriously if a cached word is selected for replacement by the cache protocol or when the executing thread is preempted.
4. The set of thread identifiers (see Figure 2) is replaced by a single bit per processor.
5. The reservation bit typically may also be associated to a set of memory locations and a normal write to an address close to the one that was read by a LL can clear the bit. Algorithms performing such a write between LL/SC pairs can give rise to various starvation and livelock problems.

More importantly, some architectures do not support the LL/SC instructions at all, but have a CAS instruction instead. We extend the applicability of our LL/SC based algorithm to these architectures. The basic idea behind our CAS-based implementation is to replace an accessed shared location with a specific thread-owned tag. An update of the shared location is then allowed only if its content matches the executing thread's tag. In the following we elaborate on this scheme and present our modified FIFO queue algorithm shown in Figure 5.

Assume for the moment that each thread holds a specific variable that can be accessed by all other participating threads. This variable is passed as parameter to our simulated LL instruction, which reads and returns the content of a shared variable. The shared location is read on line L5. If it holds an application related data, the shared location is atomically replaced by the

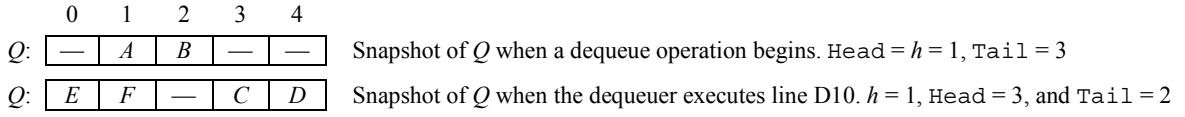

Figure 4: Possible snapshots experienced by a dequeuer immediately prior to and following its preemption

address of the specific thread-owned variable, which acts as a reservation marker. On the other hand, if the shared location contains the address of another thread's owned variable, the shared location is already reserved; the application data is read by means of this variable (line L8) before atomically substituting the location with the address of the caller's owned variable. To distinguish between application data and thread-owned variables, we use the fact that modern 32- and 64-bit architectures allocate memory blocks at addresses that are evenly dividable by 2; therefore, the least significant bit of a valid address is always 0. As our FIFO array contains addresses to application nodes, the least significant bit of an address is in excess and can be used as an indicator. The atomic primitives operating on these addresses need only be pointer-wide primitives, and thus meet the requirement for the emerging 64-bit architectures. In the LL function of Figure 5, odd valued addresses indicate thread-owned variables. The hat symbol (^) in Figure 5 refers to C's bitwise xor operator.

A similar scheme was proposed in [7], however, theirs uses a thread identifier augmented by a version tag rather than the address of an owned variable to mark a reservation. The thread identifier is used as an index into an array that serves as a placeholder for the substituted value. The purpose of their version tag is to avoid ABA problems associated with the replacement. The resulting scheme uses only pointer-wide atomic primitives but it is not population-oblivious because of the fixed-sized array. Their scheme may however be transformed into a population-oblivious one like the one we expose below.

We next explain how threads acquire their owned variable. The register and deregister operations that are part of our algorithm are a simplification of those proposed in [5], and which were designed to solve the collect problem. Each thread that calls an enqueue or dequeue operation requires a global variable that other threads may consult in the simulated LL operation. This global variable is acquired by a registration operation. Once a thread no longer enqueues nor dequeues data, its variable is no longer required but cannot be freed because other threads may still access it (lines L5, L7, L8 and L14). Consequently, allocated variables are kept permanently in a list but other threads may recycle them. If a thread needs to perform more than a single operation on the array, its owned variable may be reused only if other threads are not currently accessing it. Assume that this is not the case, a thread, say *A*, may substitute an array slot by its owned variable and be preempted by another thread, *B*. *B* can read the owned variable of *A* (line L5) and be preempted by *A* before the CAS operation on line L12. *A* may then finish its operation on the array and latter, after many enqueues and dequeues by threads other than *B*, reinsert its owned variable into the same array slot currently referenced by *B*. If *B* now resumes, it will wrongly succeed the CAS operation on line L12 and insert its owned variable with erroneous information regarding the actual content of the array slot. To circumvent this ABA problem, we simply use a reference counter that is incremented on line L7 and decremented when an owned variable ceases to be accessed (line L14). These operations are done by an atomic FetchAndAdd instruction.

The data type used for the simulated LL/SC operations is called *LLSCvar* and contains a placeholder for a FIFO slot, a reference counter indicating how many threads are currently accessing it, and a link to the next *LLSCvar* variable in the list. To acquire a *LLSCvar* variable, a thread first traverses the `First` list and tries to reclaim an unowned variable by setting its reference counter to 1 (line R4). If the CAS succeeds, an available *LLSCvar* variable is found and returned (line R5). If the thread reaches the end of the list, it assumes there is no *LLSCvar* variable that can be recycled. The thread then allocates a new *LLSCvar* variable and adds it to the list following a simple LIFO policy—a FIFO policy would require an extra variable. A simple retry-loop with a CAS operation is used to add the new owned variable to the list. The register operation takes time and space that is a function of the maximum number of threads that accessed the queue at any given time. A thread must call `ReRegister` to check that no other thread is accessing its owned variable between any two consecutive operations on the array. This operation either returns the same *LLSCvar* variable or another one after deregistering the currently held *LLSCvar* variable. The deregister operation removes the owner's reference to the *LLSCvar* variable (line DR2) so that it may be reclaimed by future register operations, and takes constant time. If a thread fails after its register operation but before its corresponding

```
     typedef struct LLSCvar {
        NODE *node;
        unsigned int r;
        struct LLSCvar *next;
     } LLSCvar;

     LLSCvar *First = null;

L1:  void *LL(void *addr, LLSCvar *var) {
L2:    NODE *slot;
L3:    BOOL restart = true;
L4:    while (restart) {
L5:       slot = *addr;
L6:       if (slot % 2 != 0) {
L7:          FetchAndAdd(&(slot^1)->r,1);
L8:          var->node = (slot^1)->node;
L9:       }
L10:      else
L11:         var->node = slot;
L12:      restart = CAS(addr,slot,var^1);
L13:      if (slot % 2 != 0)
L14:         FetchAndAdd(&(slot^1)->r,-1);
L15:   }
L16:   return var->node;
L17: } /* end of LL */

BOOL Enqueue(NODE *node, LLSCvar *var) {
  unsigned int t, tail;
  NODE *slot;
  while (true) {
     t = Tail;
     if (t == Head + Q_LENGTH)
        return FULL_QUEUE;
     tail = t % Q_LENGTH;
     slot = LL(&Q[tail],var);
     if (t == Tail) {
        if (slot != null) {
           CAS(&Q[tail],var^1,slot);
           CAS(&Tail,t,t+1);
        }
        else if (CAS(&Q[tail],var^1,node)) {
           CAS(&Tail,t,t+1);
           return OK;
        }
     }
     else
        CAS(&Q[tail],var^1,slot);
  }
} /* end of Enqueue */
```

```
R1:  LLSCvar *Register(void) {
R2:    LLSCvar *var = First;
R3:    while (var != null) {
R4:      if (var->r == 0 && CAS(&var->r,0,1))
R5:         return var;
R6:      else
R7:         var = var->next;
R8:    }
R9:    var = malloc(sizeof(LLSCvar));
R10:   var->r = 1;
R11:   while (true) {
R12:      var->next = First;
R13:      if (CAS(&First,var->next,var))
R14:         return var;
R15:   }
R16: } /* end of Register */

RR1: LLSCvar *ReRegister(LLSCvar *var) {
RR2:   if (var->r == 1) return var;
RR3:   FetchAndAdd(&var->r,-1);
RR4:   return Register();
RR5: } /* end of ReRegister */

DR1: void Deregister(LLSCvar *var) {
DR2:   FetchAndAdd(&var->ref,-1);
DR3: } /* end of Deregister */

NODE *Dequeue(LLSCvar *var) {
  unsigned int h, head;
  NODE *slot;
  while (true) {
     h = Head;
     if (h == Tail)
        return null;
     head = h % Q_LENGTH;
     slot = LL(&Q[head],var);
     if (h == Head) {
        if (slot == null) {
           CAS(&Q[head],var^1,slot);
           CAS(&Head,h,h+1);
        }
        else if (CAS(&Q[head],var^1,null)) {
           CAS(&Head,h,h+1);
           return slot;
        }
     }
     else
        CAS(&Q[head],var^1,slot);
  }
} /* end of Dequeue */
```

Figure 5. Pointer-wide CAS implementation of a FIFO queue

deregister operation, its *LLSCvar* variable is never reclaimed and results into a memory leak.

We now explain the modifications brought to the enqueue and dequeue operations. These operations read the content of an array slot into variable `slot`. Then depending on `slot`, they either substitute the array slot with a new value or release their reservation. All substitutions are done by means of a CAS instruction with expected value being the address of the caller's owned variable having its least significant bit set. Restoring the original content of the slot undoes any reservation for the slot. Finally, observe that any eventual ABA problem is avoided by verifying that the `Head` or `Tail` index remains unchanged from the moment it is first read until the moment the slot is substituted by the thread-owned variable.

6. Experimental Results

We evaluated the performance of our FIFO queue algorithms relative to other known algorithms by running a set of synthetic benchmarks written in C using *pthreads* for multithreading. In all our experiments, each thread performs 100000 iterations consisting of a series of 5 enqueue operations followed by 5 dequeue operations. A node allocation immediately precedes each enqueue operation, and each dequeued node is freed. We synchronized the threads so that none can begin its iterations before all others finished their initialization phase. We report the average of 50 runs

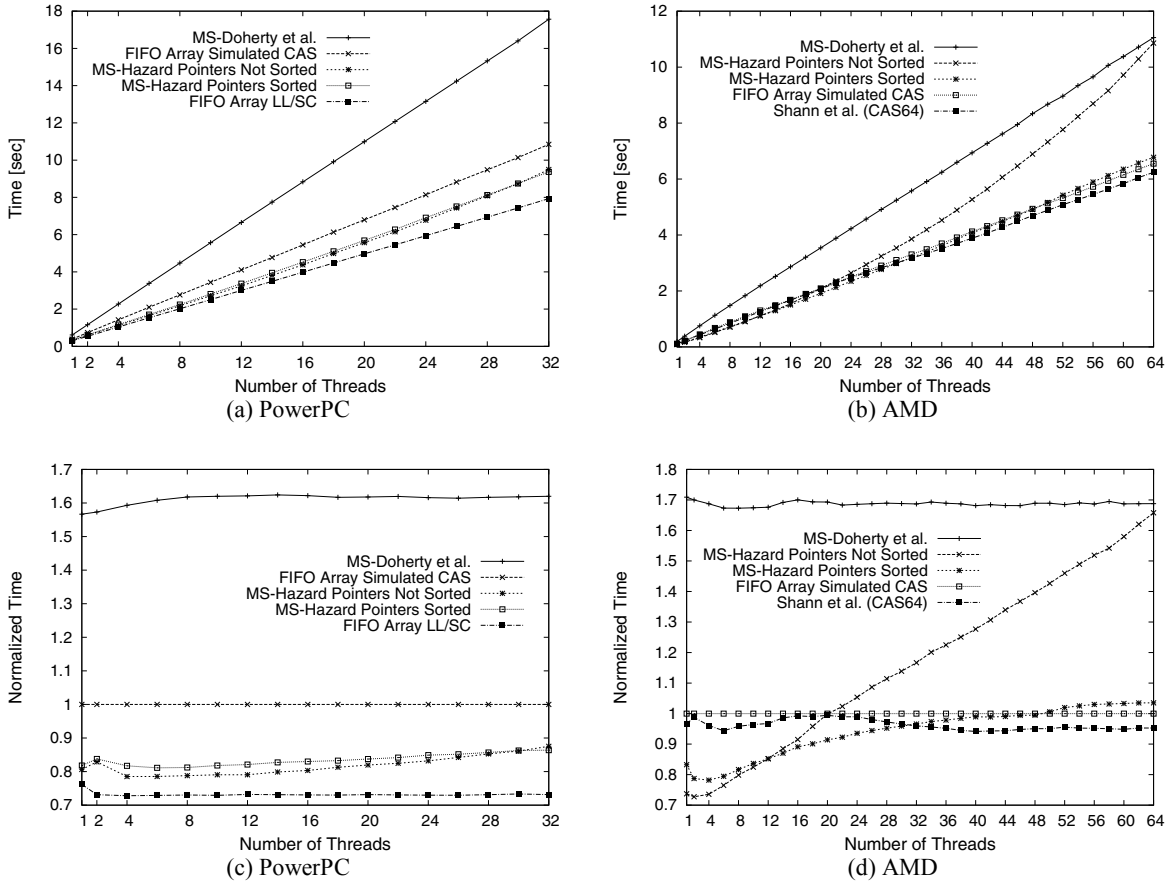

Figure 6. Actual and normalized running time as a function of the number of threads on specific architectures

where each run is the mean time needed to complete the thread's iterations.

We conducted experiments for two different systems. The first is a PowerPC G4 1.5 GHz running on Darwin 8.8.0, which only has pointer-wide LL/SC instructions that can be accessed in C by implementing them as functions written in assembler. 32-bit CAS operations are provided by *libkern*. For this system, we were able to compare our algorithms with 2 different implementations of Michael and Scott's link-based FIFO algorithm [9] that allow safe memory reclamation. The first uses hazard pointers [10] (MS-Hazard Pointers) and the second a CAS-based simulation of LL/SC instructions [2] (MS-Doherty *et al.*). Both algorithms require only pointer-wide instructions.

The second system is a Linux version 2.6.18 running on an AMD Sempron 3000+ 1.6 GHz, which has 32- and 64-bit CAS instructions with 32-bit wide pointers. For this system, we were able to include Shann et al.'s array-based FIFO [12] in our comparisons.

Figure 6 shows the actual and normalized running times of the selected algorithms as a function of the number of threads. The basis of normalization was chosen to be our CAS-based implementation because this algorithm is common to both experiments and allows for easy comparisons.

Our LL/SC-based implementation is the fastest and it is approximately 27% faster than our CAS-based implementation. We also conducted an experiment with a single thread accessing the FIFO array in absence of contention and without any synchronization in order to evaluate the overhead imposed by our implementations. Our LL/SC and CAS-based implementations are respectively 12% and 50% slower on the PowerPC, and the CAS-based implementation is 90% slower on the AMD.

Compared to Shann *et al.*'s implementation, which uses a 32- and a 64-bit CAS operation to enqueue or dequeue a node, our CAS-based implementation requires three 32-bit CAS and two FetchAndAdd operations, and it is roughly only 5% slower because a 64-bit CAS roughly takes 4.5 more time than its 32-bit counterpart on the AMD. As can be seen from all the graphs, the MS hazard pointer FIFO queue implementation is a better algorithm when the number of threads

is moderate. In this algorithm, nodes that are currently accessed by a thread have their addresses stored in a per-thread global variable and a node can be reclaimed only when its address doesn't appear in any of these variables. In our experiments, a thread attempts to free all the nodes it dequeued when the number of freed nodes it holds is equal to 4 times the number of threads. Even though this results in a huge waste of memory, the cost to reclaim the nodes becomes fairly low. As the number of threads increases, so does the time to traverse all these variables, and hence the benefit of sorting them when the number of threads is moderate to high. Although the algorithm uses a single successful CAS to dequeue and 2 successful CASs to enqueue, making it the algorithm with the least number of synchronization instructions, the overhead associated to free the nodes gets the upper hand when the number of threads is high. It is interesting to observe that the MS hazard pointer implementation is more efficient on the PowerPC than on the AMD because of the relatively cheaper cost of a CAS operation. The slowest of the measured FIFO implementations is unquestionably the Doherty *et al.* because it requires 7 successful CAS instructions per queueing operations.

7. Conclusions

We have presented 2 space-adaptive non-blocking implementations of a concurrent FIFO queue based on a bounded circular array. Our first implementation uses load-linked/store conditional atomic instructions and the second is based on the popular CAS atomic instruction. Compared to concurrent non-blocking link-based FIFO queues, we showed that array-based implementations are valuable alternatives when the number of threads accessing the queue is high or when memory usage and management issues are the main concern. Compared to other non-blocking FIFO queue implementations, our new algorithms improve on previous ones by using only pointer-wide atomic instructions, as well as reducing space requirements and the need for advance knowledge of the number of threads that will access the queue.

We believe that our new algorithms are of highly practical interest for multithreaded applications because they are based on atomic primitives that are available in today's processors and microcontrollers.

8. References

[1] D.L. Detlefs, P.A. Martin, M. Moir, and G.L. Steele Jr., "Lock-free reference counting", *Proc. of the 20th Ann. ACM Symp. on Principles of Distributed Computing* (PODC 2001), pp. 190-199, Aug. 2001.

[2] S. Doherty, M. Herlihy, V. Luchangco, M. Moir, "Bringing Practical Lock-Free Synchronization to 64-bit Applications", *Proc. of the 23rd Ann. ACM Symp. on Principles of Distributed Computing* (PODC 2004), pp. 31-39, July 2004.

[3] M.P. Herlihy, and J.M. Wing, "Linearizability: A correctness condition for concurrent objects", *ACM TOPLAS*, 12:3, pp. 463-492, July 1990.

[4] M. Herlihy, V. Luchangco, and M. Moir, "The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized, Lock-free Data Structures", *Proc. of the 16th International Symposium on Distributed Computing* (DISC 2002), pp. 339-353, Oct. 2002.

[5] M. Herlihy, V. Luchangco, and M. Moir, "Space- and Time-adaptive Nonblocking Algorithms", *CATS'03 Proc. of Computing: The Australasian Theory Symposium*, pp. 260-280, April 2003.

[6] E. Ladan-Mozes, and N. Shavit, "An Optimistic Approach to Lock-Free FIFO Queues", *Proc. of the 18th International Conference on Distributed Computing* (DISC 2004), pp. 117-131, October 2004.

[7] V. Luchangco, M. Moir, and N. Shavit, "Nonblocking k-Compare-And-Swap", *Proc. of the 15th Ann. ACM Symp. on Parallel Algorithms and Architectures* (SPAA'03), pp. 314-323, June 2003.

[8] M.M. Michael, and M.L. Scott, "Correction of a memory management method for lock-free data structures", Computer Science Department, University of Rochester, Technical Report, 1995.

[9] M.M. Michael, and M.L. Scott, "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors", *J. Parallel Distrib. Comput.*, Vol. 51, No 1, pp. 1-26, May 1998.

[10] M.M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 15, No. 6, pp. 491-504, June 2004.

[11] M. Moir, "Practical Implementations of Non-Blocking Synchronization Primitives", *Proc. of the 16th Ann. ACM Symp. on Principles of Distributed Computing* (PODC'97), pp. 219-228, Aug. 1997.

[12] C.-H. Shann, T.-L. Huang, and C. Chen, "A Practical Nonblocking Queue Algorithm Using Compare-And-Swap", *Proc. of the 7th International Conf. on Parallel and Distributed Systems* (ICPADS 2000), pp. 470-475, July 2000.

[13] R. Treiber, "Systems Programming: Coping With Parallelism", Technical Report RJ5118, IBM Almaden Research Center, April 1986.

[14] P. Tsigas, and Y. Zhang, "A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems", *Proc. of the 13th Ann. ACM Symp. on Parallel Algorithms and Architectures* (SPAA'01), pp. 134-143, July 2001.

[15] J.D. Valois, "Lock-free linked lists using compare-and-swap", *Proc. of the 14th ACM Symposium on Principles of Distributed Computing* (PODC'95), pp. 214-222, Aug. 1995.

[16] J.M. Wing, and C. Gong, "Testing and Verifying Concurrent Objects", *J. Parallel Distrib. Comput.*, Vol. 17, Nos. 1-2, pp. 164-182, 1993.