Efficient Pure-buffer Algorithms for Real-time Systems [1]

James H. Anderson and Philip Holman
University of North Carolina, Chapel Hill, NC 27599-3175

Abstract

We present wait-free algorithms for implementing multi-writer read/write pure-buffers in multiprocessor real-time systems. Such buffers are commonly used when existing data is overwritten as newly-produced data becomes available. Pure-buffer algorithms share several buffers between client processes and use handshaking mechanisms to ensure the safety of concurrent read and write operations. We present algorithms optimized for both quantum- and priority-scheduled systems. When used to implement a B-word buffer shared across a constant number of processors, the time complexity for reading and writing in each of our algorithms is $O(B)$, and the space complexity is $\Theta(B)$.

1 Introduction

Shared read/write buffers are commonly used in real-time applications to exchange data values between producer and consumer processes. Such a buffer is defined by its size and the number of writer and reader processes. A write operation completely overwrites the buffer's previous contents, while a read operation returns the most recently-written value. Read/write buffers are appropriate to use if more-recently-produced data is always of greater value than older data, which is often the case when data values are time-sensitive.

In real-time systems, read/write buffer operations are usually implemented using locks. When locks are used, kernel support is needed to limit the impact of priority inversions. A priority inversion occurs when a process is forced to block on a process of lower priority. Conventional mechanisms for bounding priority inversions [7, 14, 16] rely on the kernel to dynamically raise the priority of the lock-holding process. This adds complexity to the kernel and complicates dynamic process creation and removal. In addition, the blocking-time estimates used to account for priority inversions in the scheduling analysis of multiprocessor systems can be prohibitively large.

In recent years, several researchers have investigated the use of wait-free shared-object algorithms as an alternative to lock-based mechanisms in object-based real-time systems [3, 4, 5, 6, 9, 10, 15]. In a *wait-free* object implementation, operations must be implemented using bounded, sequential code fragments, with no blocking synchronization constructs. Thus, a process never blocks while accessing a wait-free object, and hence priority inversions cannot arise due to object accesses. In this paper, we present new wait-free implementations of read/write buffers that are highly optimized for use in real-time systems.

Related work. There has been a long history of work on wait-free buffer algorithms. For historical reasons, these buffers are usually referred to as *atomic registers* in the wait-free algorithms literature. It has been shown that multi-writer, multi-reader, multi-bit atomic registers can be implemented in a wait-free manner from single-writer, single-reader, single-bit atomic registers[2] (see, for example, [8, 12, 13, 17]). In principle, these constructions could be used to implement read/write buffers in a real-time system. However, actual systems provide much stronger synchronization primitives. By using available primitives, much simpler and more efficient algorithms can be derived.

Chen and Burns recently showed that, by using *compare-and-swap* and *test-and-set*,[3] it is possible to efficiently implement a one-writer wait-free buffer [9, 10]. Their algorithm can be seen as a variant of several previous algorithms that do not use strong synchronization primitives [8]. In the wait-free algorithms literature, these algorithms are known as "pure-buffer" algorithms. In a pure-buffer algorithm, several buffers are shared between the writer and reader processes, and a handshaking mechanism is employed that ensures that a writer never writes into a buffer that is concurrently being read by some reader. When used to implement a B-word buffer that may be read by R processes, Chen and Burns' algorithm requires $R+2$ buffers, and hence its space complexity is $\Theta(RB)$. $\Theta(B)$ time is required to read the implemented buffer, and $\Theta(R+B)$ time is required to write it. These complexity figures are listed

[1] Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, and ITR 0082866.

[2] In fact, multi-writer, multi-reader, multi-bit atomic registers can be implemented in a wait-free manner from *nonatomic* single-writer, single-reader, single-bit registers.

[3] Their algorithm is actually based on a consensus object and test-and-set. However, in most systems, the consensus object would be implemented using compare-and-swap.

Algorithm	**Processors/ Writers**	**System Model**	**Read Complexity**	**Write Complexity**	**Space Complexity**
Chen & Burns	P/1	Asynchronous	$\Theta(B)$	$\Theta(R+B)$	$\Theta(RB)$
Algorithm 1	P/W	Priority-based	O(B)	$\Theta(P+B)$	$\Theta(PB)$
Algorithm 2	P/W	Quantum-based	$\Theta(B)$	$\Theta(P+B)$	$\Theta(PB)$

Table 1: Wait-free read/write buffer algorithms.

in Table 1.

Other recent research on pure-buffer constructions includes a nonblocking algorithm presented at RTCSA '99 by Tsigas and Zhang [18]. However, their algorithm is not wait-free and thus is of less relevance to our work (in a nonblocking algorithm, operations can be unboundedly retried; such retries are not allowed in a wait-free algorithm). Moreover, their algorithm is limited to systems in which there is at most one writer on each processor, and each writer has the highest priority of any process on its processor.

Recent research at the University of North Carolina has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems [2, 3, 15]. In particular, if processes are scheduled by priority, then object calls by high-priority processes *automatically* appear to be atomic to lower-priority processes executing on the same processor. In a quantum-scheduled system, if an object call crosses a quantum boundary, then when it resumes, it will execute nonpreemptively, assuming that it cannot cross multiple quantum boundaries (which would almost certainly be the case, since most object calls are short in duration relative to the size of a scheduling quantum). These facts can be exploited to obtain algorithms that have complexities that are a function of the number of process*ors* in the system, not the number of process*es*.

Most prior work on optimizing wait-free object implementations for use in real-time systems has focused on the development of algorithmic techniques that can be generally applied to implement any object. While it is important to have general-purpose object-sharing mechanisms, it is our belief that, in most real-time applications, a small number of shared objects predominate; these include read/write buffers, queues, priority queues, and perhaps linked lists. Thus, it would benefit the real-time community to have highly-optimized wait-free implementations of these particular objects.

Contributions of this paper. In this paper, we present new wait-free algorithms for efficiently implementing multi-writer read/write buffers in priority- and quantum-scheduled multiprocessor real-time systems. These algorithms are listed in Table 1. In the full version of this paper [1], simplified versions of these algorithms are also included for more restricted cases, such as single-writer buffers and buffers for uniprocessor systems. These other versions, as well as formal correctness proofs, are omitted here due to space constraints. All of our algorithms are pure-buffer algorithms based on compare-and-swap.

In Table 1, P denotes the number of processors sharing the buffer. In most applications, one would expect P to be quite small. R and W denote the number of reader and writer processes (respectively), and B denotes the number of words in the buffer. In all of our algorithms, the time complexity for reading is comparable to Chen and Burns' algorithm, and the time complexity for writing is better. In addition, each of our algorithms has better space complexity than their algorithm. (As explained later, the actual space complexity of Algorithm 1 is $\Theta(PB + RB + WB)$, but the $\Theta(RB + WB)$ term represents extra space that is common to *all* buffers in the system, so it is not listed in Table 1. In other words, the space required to implement M buffers is only $\Theta(MPB + RB + WB)$. Algorithm 2 also has extra space complexity terms of this nature.) If P is viewed as a constant, which is reasonable for most systems, then the time complexity for reading and writing in each of our algorithms is $O(B)$, and the space complexity is $\Theta(B)$; these complexity figures are obviously asymptotically optimal.

2 Preliminaries

Each of our buffer algorithms is defined by specifying a procedure that is invoked to read the buffer, and one that is invoked to write the buffer. Each invocation of the read procedure (respectively, write procedure) is called a *read operation* (respectively, *write operation*). The processes in the system are partitioned into a set of *reader processes* and a set of *writer processes*. For our purposes, it suffices to view each reader (writer) process as consisting of an infinite loop that repeatedly invokes the read (write) procedure. With this assumption, we are simply abstracting away from the activities of these processes outside of buffer accesses. Each of our algorithms is designed for use in either a priority- or quantum-scheduled system. We make the following assumptions regarding the manner in which processes are scheduled for execution on a processor.

Axiom 1: (*Priority-based Scheduling*) A process's priority does not change during a read or write operation. $\square$

Axiom 2: (*Quantum-based Scheduling*) The quantum is large enough to ensure that each process can be preempted at most once within one read or write operation. $\square$

In many of our algorithms, single-word variables are used that have counter fields, which are used to distinguish recently-written data from older data. We assume that the range of each counter is sufficient to ensure that it does not cycle during any read or write operation. Each counter ranges over $\{0, \ldots, 2k+1\}$ for some $k \in \mathbf{N}$ and is assumed to wrap around to zero when incremented beyond its range. Such variables are declared using the following template.

```
template tagged(T):
        record tag: bounded integer; val: T
```

For example, a variable of type *tagged*(1..W+P+2) has a *tag* field that is a bounded integer, and a *val* field that ranges over $\{1, \ldots, W+P+2\}$.

Our algorithms also use compare-and-swap (`CAS`) operations. Such operations are denoted `CAS`(*adr*, *old*, *new*), where *adr* is the address of a shared variable, *old* is a value to which this variable is compared, and *new* is a new value to assign to the variable if the comparison succeeds. The `CAS` operation returns *true* if and only if the comparison succeeds.

Notational Conventions: We let R, W, B, and P be defined as in Table 1. Unless stated otherwise, we let p, q, and r denote reader processes, and v and w denote writer processes. Each of p, q, and r ranges over $\{1, \ldots, R\}$, and each of v and w ranges over $\{1, \ldots, W\}$.

We assume that each labeled statement in each algorithm is atomic. We also assume that all private variables of a process retain their values between operations on the implemented buffer by that process. The notation $x.y$ will be used to refer to the value of process x's private variable y.

3 Priority-based Algorithm

Our multi-writer algorithm for priority-based multiprocessors is shown in Figure 1. This algorithm implements a shared buffer using $P+2$ pure buffers, which we will call "slots." In contrast, Chen and Burn's algorithm uses $R+2$ slots (and is also limited to only one writer). We reduce the number of required slots by ensuring that there is only one active reader on any processor at any time. Thus, each writer only needs to coordinate with at most P active readers at any time. To ensure that there is only one active reader per processor, each reader process is required to *help* complete any read operation that it preempts.

A handshaking mechanism is used to ensure that a writer never writes into a slot that is being read by some reader. This mechanism requires a total of $P+2$ slots. This is because, due to preemptions, there may be P active readers that are in the process of reading P distinct values that were written previously by some writer. Each of these values may differ from the last value written by the writer. The last-written value cannot be immediately overwritten because this would temporarily leave the buffer in a state in which the most-recently-written value is unavailable. Thus, $P+2$ slots are needed.

So that readers may help one another, the buffer into which each reader saves the value that it reads is shared, rather than private. Thus, R shared buffers are needed for helping, but these buffers can be used across *all* shared buffers in the system. In other words, these R buffers are part of the *system*'s overhead rather than the *buffer*'s overhead. We also assume that each writer stores the value it wants to write in an input buffer that can then be swapped with one of the slots of the implemented buffer. Thus, W input buffers are needed. Once again, however, these same W buffers can be used across all shared buffers in the system, so we do not consider them as per-buffer overhead. In the rest of this section, we give a detailed description of the algorithm, followed by a summary of the techniques used to manage concurrent buffer accesses.

Detailed description. We begin our detailed description of the algorithm by describing the shared variables that are used. The $P+2$ slots along with each writer's input buffer are stored in the *In* array. We assume that each slot consists of B words. The *Bufptr* array indicates which $P+2$ of the slots in the *In* array are currently part of the implemented buffer. The variable *Latest* indicates the slot that holds the most-recently written value. Each reader r has an output buffer *Out*$[r]$. Each time r reads the implemented buffer, the value it reads is stored in *Out*$[r]$. If reader p helps reader r, then to ensure that p does not repeat steps already performed by r or other processes, we maintain a count of the words already copied to *Out*$[r]$. This count is stored in the shared variable *Wdcnt*$[r]$. *Reader*$[k]$ is used to indicate the currently-active reader (if any) on processor k. *Reading*$[k]$ indicates the last slot read by

```
shared var
    $In$: array$[1..W+P+2][1..B]$ of $wordtype$;
    $Bufptr$: array$[1..P+2]$ of $tagged(1..W+P+2)$;
    $Latest$: $tagged(1..P+2)$ initially $(0,1)$;
    $Out$: array$[1..R][1..B]$ of $wordtype$;
    $Wdcnt$: array$[1..R]$ of $0..B$ initially $0$;
    $Reader$: array$[1..P]$ of $0..R$ initially $0$;
    $Reading$: array$[1..P]$ of $tagged(0..P+2)$ initially $(0,1)$

private var
    $bf$, $cbf$: $1..W+P+2$;   $nbf$: $tagged(1..W+P+2)$;
    $next$, $n$, $val$: $1..P+2$;   $\ell$: $tagged(1..P+2)$;
    $bp$: $0..P+2$;   $rb$: $tagged(0..P+2)$;
    $inuse$: array$[0..P+2]$ of boolean;
    $wc$: $0..B$;   $rd$: $0..R$;
    $wd$: $wordtype$;   $succ$: boolean
```

initially $In[1] = initial\ value \wedge (\forall y\colon 1 \le y \le P+2\colon Bufptr[y] = (0, y)) \wedge (\forall w\colon 1 \le w \le W\colon w.cbf := P + 2 + w)$

```
procedure Read($rid$,$myproc$)
        returns array$[1..B]$ of $wordtype$
1:  $rd := Reader[myproc]$;
2:  if $rd \ne 0$ then Help-Read($rd$,$myproc$) fi;
3:  UpdateReading($myproc$);
4:  $Wdcnt[rid] := 1$;
5:  $Reader[myproc] := rid$;
6:  Help-Read($rid$,$myproc$);
7:  return $Out[rid]$

procedure Help-Read($rd$,$myproc$)
8:  $bp := Reading[myproc].val$;
9:  $bf := Bufptr[bp].val$;
10: $wc := Wdcnt[rd]$;
11: while $Reader[myproc] = rd \wedge wc > 0$ do
12:     $wd := In[bf][wc]$;
13:     if $Reader[myproc] = rd$ then
14:         $Out[rd][wc] := wd$
        fi;
15:     $Wdcnt[rd] := (wc + 1) \bmod (B + 1)$;
16:     $wc := Wdcnt[rd]$
    od;
17: $Reader[myproc] := 0$

procedure UpdateReading($myproc$)
18: $rb := Reading[myproc]$;
19: $succ :=$ CAS($\&Reading[myproc], rb, (rb.tag + 1, 0)$);
20: if $\neg succ$ then
21:     $rb := Reading[myproc]$;
22:     $succ :=$ CAS($\&Reading[myproc], rb, (rb.tag + 1, 0)$)
    fi;
23: if $succ$ then
24:     $\ell := Latest$;
25:     CAS($\&Reading[myproc]$, $(rb.tag+1,0)$, $(rb.tag+2,\ell.val)$)
    fi

procedure Write($wid$)
26: $\ell := Latest$;
27: $bp :=$ FindNext();
28: $nbf := Bufptr[bp]$;
29: if $\ell = Latest$ then
30:     if CAS($\&Bufptr[bp]$, $nbf$, $(nbf.tag+1,cbf)$) then
31:         $cbf := nbf.val$
        fi;
32:     CAS($\&Latest$, $\ell$, $(\ell.tag+1,bp)$)
    fi

procedure FindNext() returns $1..P+2$
33: for $n := 1$ to $P$ do
34:     $rb := Reading[n]$;
35:     $val := Latest.val$;
36:     if $rb.val = 0$ then
37:         CAS($\&Reading[n]$, $rb$, $(rb.tag+1,val)$)
        fi
    od;
38: for $n := 1$ to $P+2$ do
39:     $inuse[n] := false$
    od;
40: $inuse[Latest.val] := true$;
41: for $n := 1$ to $P$ do
42:     $inuse[Reading[n].val] := true$
    od;
43: $next := 1$;
44: while $inuse[next] \wedge next < P+2$ do
45:     $next := next + 1$
    od;
46: return $next$
```

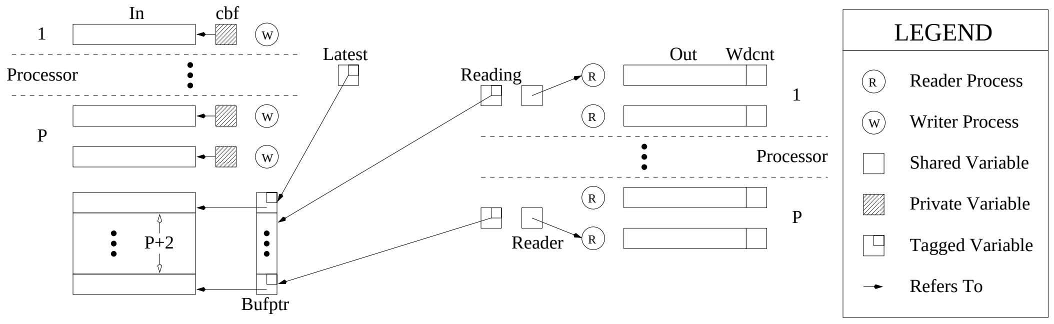

Figure 1: Algorithm 1: Multi-writer buffer for priority-based multiprocessors.

a reader on processor k. A reader r on processor k performs a read operation by invoking the `Read` procedure. Within `Read`, r first checks to see if there is a preempted read operation on processor k (statements 1-2). If there is a preempted read, then `Help-Read` is called (statement 2). This routine is described below. After helping any preempted read, r calls `UpdateReading` (statement 3), which attempts to copy the value of *Latest* to *Reading*$[k]$. r's call to `UpdateReading` can fail to update *Reading*$[k]$ only if either a writer updates *Reading*$[k]$ (see statement 37) or if r is preempted by another reader on processor k. In either case, *Reading*$[k]$ points to a slot written "sufficiently recent" by the time r returns from `UpdateReading`. After invoking `UpdateReading`, r updates *Wdcnt*$[r]$ and *Reader*$[k]$ (statements 4-5) to indicate that it is now the active reader on processor k. Note that statement 5 effectively "announces" r's read on processor k — if r is preempted by another reader after this point, then it will be helped. After updating *Reader*$[k]$, r performs its own operation by calling `Help-Read` (statement 6).

We now describe what happens when `Help-Read` is invoked by r. First, the state of the read being helped is determined (statements 8-10). Let p be the reader r is helping (note that p could be r). r helps p by updating *Out*$[p]$ one word at a time (statement 14). If r finds *Reader*$[k] \neq r.rd$ at either statement 11 or 13, then r must have been preempted by a higher-priority reader. In this case, by the time r resumes execution, p's operation has been completed, so r can discontinue helping. Note that it is possible for r to be preempted by a higher-priority reader q between its execution of statements 13 and 14, in which case its execution of statement 14 will overwrite a word of *Out*$[p]$ already written by q. In this case, the value written to this word by r must be the same as its current value. (This property is proved in the full paper.) Thus, this "late write" causes no harm. The time complexity of a read operation is clearly dominated by the calls to `Help-Read`, which take $O(B)$ time.

A writer w performs a write operation by invoking the `Write` procedure. It is assumed that w has already copied the words it intends to write into *In*$[w.cbf]$ before invoking the `Write` procedure. (Also, recall that, by assumption, $w.cbf$ retains its value between write operations of w.) w's write operation is performed in three steps. First, `FindNext` is called to locate an unused slot (statement 27). Then, w attempts to swap *Bufptr*$[w.bp]$ and $w.cbf$ (statements 30 and 31), which has the effect of swapping w's input buffer with the free slot l returned by `FindNext`. If `CAS` at statement 30 fails, then another writer must have swapped its own input value into slot l before w's attempt. Finally, *Latest* is updated to indicate that slot l holds the latest value written to the buffer (statement 32). Note that if the value of *Latest* changes between statements 26 and 29, then w's operation has been "overwritten" by a concurrent write and thus there is no need to swap in slot l. A concurrent write operation can also cause the `CAS` at statement 32 fail.

Within `FindNext`, w first reads *Latest* and then completes any stalled updates of *Reading* variables (statements 34-37). Using a `CAS` at statement 37 ensures that w cannot write an out-of-date value into some *Reading* variable in the event that it is itself preempted. In the rest of the `FindNext` procedure, w simply chooses a slot index that differs from the current value of *Latest* and any *Reading* variable. Since `FindNext` has $\Theta(P)$ time complexity, the time complexity of a write operation is $\Theta(P + B)$.

Mechanisms. In our algorithm, three distinct mechanisms are used. The first mechanism, helping, is used to manage local reader-reader interactions. This mechanism requires a local reader to acquire a "token" prior to performing the read operation. (*Reader* records which process currently holds the token.) To avoid starvation, other local readers are not permitted to take the token from the current reader. However, these processes can force the existing reader out of the announce position once its operation is completed. Though a similar mechanism could be applied to local writer-writer interactions, it is more costly to do so in terms of time and space. In addition, this approach would favor low-priority writers over higher-priority writers.

The second mechanism, pointer swapping, handles all writer-writer interactions. This approach is derived from shadow copying, which uses a base pointer to reference the current value of an object. To update the object, a process must duplicate the current value, apply the changes to its copy, and then attempt to atomically swap the base pointer to reference the modified copy. In a pure-buffer algorithm, there is no need to perform the initial duplication since the entire buffer is being overwritten. Instead, a writer possesses a dedicated buffer that already contains the new value prior to calling `Write`. The pointer swapping mechanism adds a layer of indirection (the *Bufptr* array) in the data referencing.

The third mechanism, the reader-writer handshake, prevents writers from swapping out a buffer that is being read. This is accomplished by allowing writers to detect and complete any stalled read slot selections at statements 33-37 prior to selecting a swapping slot at statements 38-46.

In the full version of this paper, we show that the algorithm can be simplified considerably if there is only one writer or if the buffer is shared across only one processor. In essence, these simplifications consist of removing one or more of the above mechanisms. For example, pointer swapping is clearly unnecessary in a single-writer system.

The proof of correctness for this algorithm is given in the full version of this paper [1]. From this proof we have the following theorem.

Theorem 1: *An R-reader, W-writer, B-word read/write buffer can be implemented in a wait-free manner on a P-processor priority-scheduled system with $\Theta(B)$ time complexity for reading, $\Theta(P+B)$ time complexity for writing, and $\Theta(PB)$ per-buffer space complexity.* □

4 Quantum-based Algorithm

Our multi-writer algorithm for quantum-based multiprocessors is shown in Figure 2. Unlike the previous priority-based algorithm, our quantum-based algorithm allows a write operation to write into a slot being read by a reader on its processor. Such read operations are retried. In essence, retries take the place of helping in our priority-based algorithm. By Axiom 2, each operation will have to be retried at most once. The handshaking mechanism of Algorithm 1 is used here to prevent write operations from interfering with remote readers. Because there are $P-1$ remote processors, $P-1+2=P+1$ slots are needed, which is one less than before. As before, we present a detailed description of the algorithm followed by a summary of the algorithm's key mechanisms.

Detailed description. The shared variables used in Algorithm 2 are analogous to those used in Algorithm 1 except that we now have $P+1$ slots instead of $P+2$. (Note also that some of the shared variables used in Algorithm 1 are no longer needed. This is mainly because there is no helping in Algorithm 2.) We also have the same procedures in both algorithms.

Suppose that a reader r on processor k invokes the `Read` procedure. At statement 1, r invokes `UpdateReading`, which attempts to copy the value of *Latest* to $Reading[k]$. `UpdateReading` is exactly the same as in Algorithm 1. After invoking `UpdateReading`, $Reading[k]$ points to a slot written "sufficiently recently," so its value can be copied to $r.out$ and returned. (Note that, because there is no helping, output buffers are now private variables.) This is done in statements 4-6. At statement 7, r checks to see if the value of $Reading[k]$ has changed. If not, then r can safely return. If $Reading[k]$ *has* changed, then r has been preempted by another process on its processor. In this case, the previous steps are tried again (statements 8-12). We explain below why one retry suffices. It is easy to see that read operations complete in $\Theta(B)$ time.

The `Write` procedure is identical to that used in Algorithm 1. The `FindNext` procedure is also the same, except that `UpdateReading` is invoked to update the *Reading* variable for the local processor. This is merely an optimization that allows us to use $P+1$ slots instead of $P+2$. In particular, all readers and writers on the same processor now update the local *Reading* variable. With $P+2$ slots, we could use the same `FindNext` procedure as in Algorithm 1. As before, the time complexity of a write operation is $\Theta(P+B)$.

Note that in Algorithm 1, helping is used to prevent the interference of readers by other readers. Moreover, the code that is executed to update the *Reading* variables prevents interferences by writers. In Algorithm 5, this code is the same as before, except that local writers update the local *Reading* variable just like local readers. Thus, a reader could potentially return an incorrect result only if repeatedly interfered with by local readers and writers. However, by Axiom 2, there can be at most one such interference. This is why one retry suffices.

Mechanisms. The quantum-based algorithm also uses three mechanisms for coordination. However, only one of the three mechanisms differs significantly from its priority-based counterpart.

In a quantum-based system, we can permit interference from a local reader as long as the interference is both safe and detectable. To allow a reader to detect interference, the read slot pointer for the processor is tagged with a counter that is incremented with each pointer change. If this counter changes while the copy is being made, the reader pessimistically assumes that its copy is invalid. However, by Axiom 2, the copying procedure can be restarted and completed before the next quantum boundary. Since a read operation does not change the state of the shared buffer, detection implies safety here.

Since the pointer swapping mechanism does not rely on a particular scheduling model, it appears unmodified in the quantum version of the algorithm. The reader-writer handshake mechanism, on the other hand, can be slightly optimized in quantum-based systems by allowing a writer to update the local *Reading* pointer (statement 35) rather than simply avoiding that slot. Since the reader design already detects and recovers from this form of interference, this behavior is safe. In addition, this reduces the number of shared buffers needed by one.

A proof of correctness for this algorithm is given in the full version of this paper [1]. From that proof, we have the following theorem.

```
shared var
    $In$: array $[1..W+P+1][1..B]$ of $wordtype$;
    $Bufptr$: array $[1..P+1]$ of $tagged(1..W+P+1)$;
    $Reading$: array $[1..P]$ of $tagged(0..P+1)$ initially $(0,1)$;
    $Latest$: $tagged(1..P+1)$ initially $(0,1)$
```

```
private var
    $out$: array $[1..B]$ of $wordtype$;   $succ$: boolean;
    $next$, $bp$, $val$: $1..P+1$;   $\ell$: $tagged(1..P+1)$;
    $cbf$, $rb$: $1..W+P+1$;   $nbf$: $tagged(1..W+P+1)$;
    $rbp$: $tagged(0..P+1)$;   $n$: $1..\mathbf{max}(B, P+1)$;
    $inuse$: array $[0..P+1]$ of boolean
```

initially $In[1] = initial\ value \ \wedge\ (\forall y\colon 1 \le y \le P+2\colon Bufptr[y] = (0, y)) \ \wedge\ (\forall w\colon 1 \le w \le W\colon w.cbf := P + 1 + w)$

```
procedure Read($out$, $myproc$)
        returns array $[1..B]$ of $wordtype$
1:  UpdateReading($myproc$);
2:  $rbp := Reading[myproc]$;
3:  if $rbp.val \neq 0$ then
4:      $rb := Bufptr[rbp.val].val$;
5:      for $n := 1$ to $B$ do
6:          $out[n] := In[rb][n]$
        od
    fi;
7:  if $rbp \neq Reading[myproc] \vee rbp.val = 0$ then
8:      UpdateReading($myproc$);
9:      $rbp := Reading[myproc]$;
10:     $rb := Bufptr[rbp.val].val$;
11:     for $n := 1$ to $B$ do
12:         $out[n] := In[rb][n]$
        od
    fi;
13: return $out$

procedure UpdateReading($myproc$)
14: $rb := Reading[myproc]$;
15: $succ :=$ CAS($\&Reading[myproc], rb, (rb.tag + 1, 0)$);
16: if $\neg succ$ then
17:     $rb := Reading[myproc]$;
18:     $succ :=$ CAS($\&Reading[myproc], rb, (rb.tag + 1, 0)$)
    fi;
19: if $succ$ then
20:     $\ell := Latest$;
21:     CAS($\&Reading[myproc]$, $(rb.tag+1,0)$, $(rb.tag+2,\ell.val)$)
    fi
```

```
procedure Write($myproc$)
22: $\ell := Latest$;
23: $bp :=$ FindNext();
24: $nbf := Bufptr[bp]$;
25: if $\ell = Latest$ then
26:     if CAS($\&Bufptr[bp]$, $nbf$, $(nbf.tag+1, cbf)$) then
27:         $cbf := nbf.val$
        fi;
28:     CAS($\&Latest$, $\ell$, $(\ell.tag+1, bp)$)
    fi

procedure FindNext($myproc$) returns $1..P+1$
29: for $n := 1$ to $P$ do
30:     if $n \neq myproc$ then
31:         $rbp := Reading[n]$;
32:         $val := Latest.val$;
33:         if $rbp.val = 0$ then
34:             CAS($\&Reading[n]$, $rbp$, $(rbp.tag+1, val)$)
            fi
35:     else UpdateReading($myproc$)
        fi
    od;
36: for $n := 1$ to $P+1$ do
37:     $inuse[n] := false$
    od;
38: for $n := 1$ to $P$ do
39:     $inuse[Reading[n].val] := true$
    od;
40: $next := 1$;
41: while $inuse[next] \wedge next < P+1$ do
42:     $next := next + 1$
    od;
43: return $next$
```

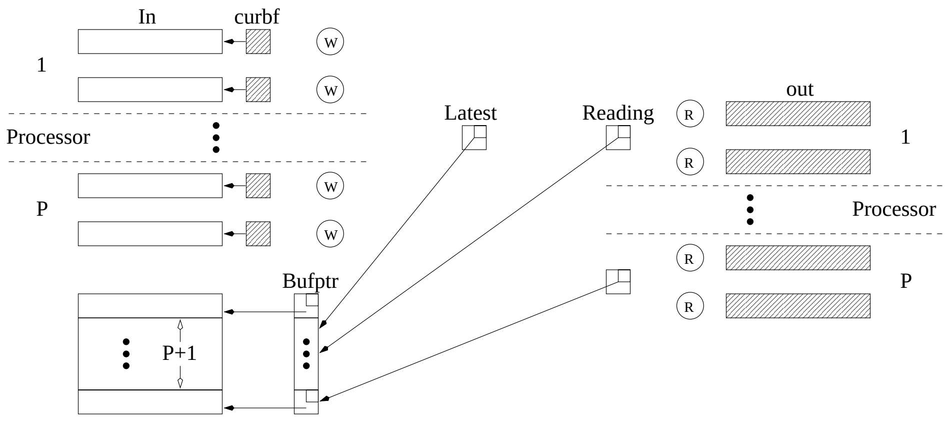

Figure 2: Algorithm 2: Multi-writer buffer for quantum-based multiprocessors.

Theorem 2: *An R-reader, W-writer, B-word read/write buffer can be implemented in a wait-free manner on a P-processor quantum-scheduled system with $\Theta(B)$ time complexity for reading, $\Theta(P+B)$ time complexity for writing, and $\Theta(PB)$ per-buffer space complexity.* □

5 Concluding Remarks

We have shown that characteristics of real-time systems can be exploited to implement highly-optimized wait-free shared buffers. Our work has been driven by the observation that, in most real-time applications, a small set of shared objects predominates. Such common objects include read/write buffers, queues, priority queues, and perhaps linked lists. We believe designers of real-time applications would benefit from having highly-optimized wait-free implementations of objects such as these. This is particularly true for multiprocessor applications. The alternative for such applications is to use priority-ceiling mechanisms. Unfortunately, the conservatism of such mechanisms makes them inefficient. In future work, we hope to consider some of the other objects listed above. Our goal is to produce a library of such implementations, along with formal correctness proofs. Such a library would allow real-time system designers to more easily incorporate wait-free objects in their applications.

References

[1] J. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems (full version). `http://www.cs.unc.edu/~anderson/papers.html`.

[2] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 346–355. Dec. 1998.

[3] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 111–122. Dec. 1997.

[4] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 92–105. Dec. 1996.

[5] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects in priority-based systems. *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pp. 229–238. Aug. 1997.

[6] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free objects. *ACM Transactions on Computer Systems*, 15(6):388–395, May 1997.

[7] T. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, Mar. 1991.

[8] J. Burns and G. Peterson. Pure buffers for concurrent reading while writing. Technical Report GIT-ICS-87/17, School of Information and Computer Science, Georgia Institute of Technology, 1987.

[9] J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, Department of Computer Science, University of York, 1997.

[10] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus.hard real-time scheduling: The deadline monotonic approach. *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pp. 2–9, June 1998.

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

[12] L. Lamport. On interprocess communication, parts 1 and 2. *Distributed Computing*, 1:77–101, 1986.

[13] G. Peterson and J. Burns. Concurrent reading while writing ii: The multi-writer case. *Proceedings of the 28th Annual ACM Symposium on Foundation of Computer Science*. 1987.

[14] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings of the International Conference on Distributed Computing Systems*, pp. 116–123, 1990.

[15] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal support. *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pp. 233–242. May 1996.

[16] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[17] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register, revisited. *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 206–221. Aug. 1987.

[18] P. Tsigas and Y. Zhang. Non-blocking data sharing in multiprocessor real-time systems. *Proceedings of the Sixth International Conference on Real-time Computing Systems and Applications*, pp. 247–254, 1999.