Efficient Pure-buffer Algorithms for Real-time Systems*

James H. Anderson and Philip Holman

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
Phone: (919) 962-1757
Fax: (919) 962-1799
E-mail: {anderson,holman}@cs.unc.edu

July 2000

Abstract

We present several wait-free algorithms for implementing read/write buffers in real-time systems. Such buffers are commonly used in situations where newly-produced data values take precedence over older data, and hence older data can be overwritten. Each of our algorithms is a "pure-buffer" algorithm. 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. Each of our algorithms is optimized by taking characteristics of quantum- and priority-based schedulers into account. When used to implement a B-word buffer that is 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 $\Theta(B)$. These complexity figures are obviously asymptotically optimal and are independent of the number of writer and reader processes. In contrast, all previously-published pure-buffer algorithms are limited to one writer process and have time and space complexity that is at least linear in the number of readers.

Keywords: Interprocess communication, priority scheduling, quantum scheduling, read/write buffers, wait-free synchronization.

*Work supported by NSF grants CCR 9732916 and CCR 9972211.

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, the number of processes that can write into it, and the number of processes that can read from it. A write operation on a read/write buffer 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, operations on read/write buffers are usually implemented using locks. When locks are used, kernel support is needed to limit the impact of priority inversions. A priority inversion is said to occur when a process is forced to block on a process of lower priority. Conventional mechanisms for dealing with priority inversions [8, 22, 23, 24, 26] rely on the kernel to dynamically raise the priority of a lock-holding process so that the duration of any priority inversion is bounded. This adds complexity to the kernel and complicates the job of supporting dynamic process creation and removal. In addition, in multiprocessor systems, the blocking-time estimates used to account for priority inversions in scheduling analysis 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, 7, 12, 13, 25]. In a *wait-free* object implementation [20], 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 several 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. In a series of papers, it was 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[1] [1, 9, 10, 11, 15, 16, 17, 18, 19, 20, 21, 27, 28, 30]. Directly implementing registers of the former type using registers of the latter type is quite hard, so most of these papers focused only on a single dimension, such as showing that a *multi*-reader register could be implemented from *single*-reader ones. In principle, these atomic register constructions could be used to implement read/write buffers in a real-time system. However, actual systems provide synchronization primitives that are much stronger than single-writer, single-reader, single-bit atomic registers. 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*,[2] it is possible to efficiently implement a one-writer wait-free buffer [12, 13]. Their algorithm can be seen as a variant of several

[1] 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.

[2] 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.

previous algorithms that do not use strong synchronization primitives [1, 11, 28, 30]. 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 in Table 1.

Other recent research on pure-buffer constructions includes a nonblocking algorithm presented at RTCSA '99 by Tsigas and Zhang [29]. 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, 25]. 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 been directed towards 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 several new wait-free algorithms for efficiently implementing read/write buffers in real-time systems. These algorithms are listed in Table 1. We present algorithms for implementing buffers in both priority- and quantum-scheduled systems. In addition, while only single-writer buffers have been considered in most previous work, we consider both single- and multi-writer buffers. All of our algorithms are pure-buffer algorithms based on compare-and-swap.

In Table 1, P denotes the number of processors across which the implemented buffer is shared. In most applications, one would expect P to be quite small. R and W denote the number of processes that may read and write (respectively) the implemented buffer, 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

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/1	Priority-based	$O(B)$	$\Theta(P+B)$	$\Theta(PB)$
Algorithm 3	1/W	Priority-based	$O(B)$	$\Theta(B)$	$\Theta(B)$
Algorithm 4	1/1	Priority-based	$O(B)$	$\Theta(B)$	$\Theta(B)$
Algorithm 5	P/W	Quantum-based	$\Theta(B)$	$\Theta(P+B)$	$\Theta(PB)$
Algorithm 6	P/1	Quantum-based	$\Theta(B)$	$\Theta(P+B)$	$\Theta(PB)$
Algorithm 7	1/W	Quantum-based	$\Theta(B)$	$\Theta(B)$	$\Theta(B)$
Algorithm 8	1/1	Quantum-based	$\Theta(B)$	$\Theta(B)$	$\Theta(B)$

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

complexity for writing is better. In addition, all of our algorithms have 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 $\Theta(MPB + RB + WB)$. Algorithms 2 through 5 also have extra space complexity terms that are common to all buffers.) 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.

The rest of this paper is organized as follows. In Section 2, we present definitions and notation that will be used in the remainder of the paper. Our algorithms for priority-scheduled systems are then given in Section 3, and our algorithms for quantum-scheduled systems in Section 4. We conclude in Section 5.

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. □

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. □

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.

$$\textbf{template } \textit{tagged}(T)\text{: } \textbf{record } \textit{tag}\text{: } \textbf{bounded integer}\text{; } \textit{val}\text{: } T$$

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.

The following notational conventions will be adhered to in the remainder of the paper.

Notational Conventions: R denotes the number of reader processes, W the number of writer processes, and B the number of words in the implemented buffer. 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 use subscripts to denote operations of reader and writer processes. For example, r_i denotes the i^{th} read operation of reader r.

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. Let S be a subset of the statement labels in process p. Then, $p@\{S\}$ holds if and only if the program counter for process p equals some value in S. (Note that if s is a statement label, then $p@\{s\}$ means that process p is *enabled* to execute statement s, i.e., it hasn't executed statement s yet.)

We use $s.p$ to denote the statement of process p with label s, and $p.v$ to represent p's private variable v. We use $s.p_i$ to denote the execution of statement $s.p$ in the i^{th} operation of process p. If s is a statement within a **for** loop, then we denote its k^{th} execution by operation p_i as $s[k].p_i$. (For example, for Algorithm 1 in Figure 1, $36[k].w_i$ refers to the execution of statement 36 by w_i with $w.n = k$.) In our correctness proofs, we often consider the relative ordering of various statement executions. If statement execution $s.p_i$ precedes statement execution $s'.q_j$, then we write $s.p_i \prec s'.q_j$. $\Box$

3 Priority-based Algorithms

In this section, we present several pure-buffer algorithms for priority-based systems. We begin by presenting a multi-writer algorithm that can be used in a multiprocessor system. We then show that this algorithm can be simplified considerably if there is only one writer or if used in a single-processor system.

3.1 Algorithm 1: Multi-writer Buffer for Priority-based Multiprocessors

Our multi-writer algorithm for priority-based multiprocessors is shown in Figure 1. In this algorithm, a shared buffer is implemented using $P+2$ pure buffers, which we will call "slots" to avoid confusion. 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.

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 from by 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

```
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;   ℓ: 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 ∧ (∀y: 1 ≤ y ≤ P+2: Bufptr[y] = (0, y)) ∧ (∀w: 1 ≤ w ≤ W: w.cbf := P + 2 + w)

procedure Read(rid,myproc)
        returns array[1..B] of wordtype
    rd := Reader[myproc];
    if rd ≠ 0 then Help-Read(rd,myproc) fi;
    UpdateReading(myproc);
    Wdcnt[rid] := 1;
    Reader[myproc] := rid;
    Help-Read(rid,myproc);
    return Out[rid]

procedure Help-Read(rd,myproc)
    bp := Reading[myproc].val;
    bf := Bufptr[bp].val;
    wc := Wdcnt[rd];
    while Reader[myproc] = rd ∧ wc > 0 do
        wd := In[bf][wc];
        if Reader[myproc] = rd then
            Out[rd][wc] := wd
        fi;
        Wdcnt[rd] := (wc + 1) mod (B + 1);
        wc := Wdcnt[rd]
    od;
    Reader[myproc] := 0

procedure UpdateReading(myproc)
    rb := Reading[myproc];
    succ := CAS(&Reading[myproc], rb, (rb.tag + 1, 0));
    if ¬succ then
        rb := Reading[myproc];
        succ := CAS(&Reading[myproc], rb, (rb.tag + 1, 0))
    fi;
    if succ then
        ℓ := Latest;
        CAS(&Reading[myproc], (rb.tag+1,0), (rb.tag+2,ℓ.val))
    fi

procedure Write(wid)
    ℓ := Latest;
    bp := FindNext();
    nbf := Bufptr[bp];
    if ℓ = Latest then
        if CAS(&Bufptr[bp], nbf, (nbf.tag+1,cbf)) then
            cbf:= nbf.val
        fi;
        CAS(&Latest, ℓ, (ℓ.tag+1,bp))
    fi

procedure FindNext() returns 1..P+2
    for n := 1 to P do
        rb := Reading[n];
        val := Latest.val;
        if rb.val = 0 then
            CAS(&Reading[n], rb, (rb.tag+1,val))
        fi
    od;
    for n := 1 to P+2 do
        inuse[n] := false
    od;
    inuse[Latest.val] := true;
    for n := 1 to P do
        inuse[Reading[n].val] := true
    od;
    next := 1;
    while inuse[next] ∧ next < P+2 do
        next := next + 1
    od;
    return next
```

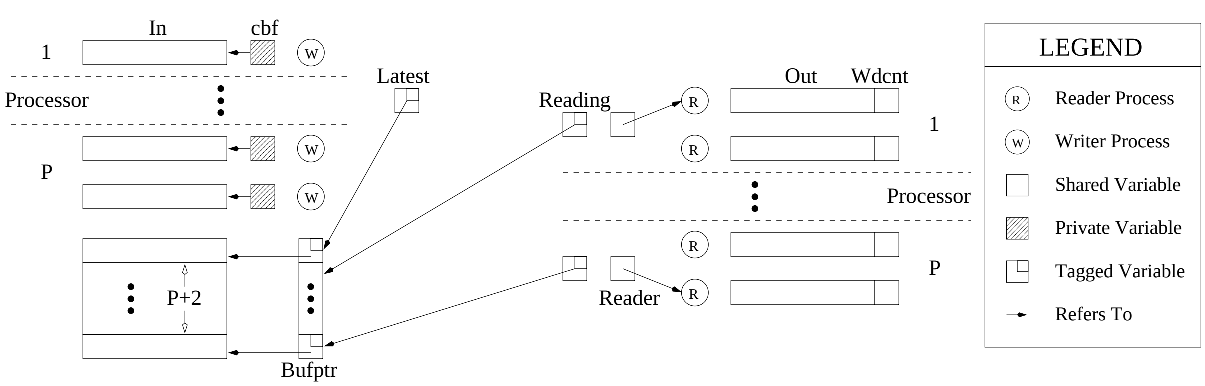

Figure 1: Algorithm 1: Multi-writer buffer for priority-based multiprocessors.

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 invoking `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. As the proof below shows (see invariant (I2)), the value written to this word by r must be the same as its current value. 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 to write to (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 $\Theta(P + B)$.

Correctness proof. We show that the algorithm is correct by proving two invariants, which are used to prove that each operation is linearizable [14]. The first invariant shows that a writer process cannot write into a slot being read by some reader. The second shows that readers cannot interfere with each other while helping.

invariant $w@\{32\} \wedge Latest = w.\ell \Rightarrow (\forall k : 1 \leq k \leq P : w.bp \neq Reading[k].val)$ (I1)

Proof: Suppose, to the contrary, that

$$w@\{32\} \wedge Latest = w.\ell \wedge w.bp = Reading[k].val \tag{1}$$

holds at some state t. Let w_j be the current operation of w at t. If no process updates $Reading[k]$ between $34[k].w_j$ and state t, then because each writer chooses a slot that differs from $Reading[k].val$, we have $w.bp \neq Reading[k].val$ at t, which contradicts (1).

Otherwise, $Reading[k]$ *is* updated between $34[k].w_j$ and state t, and the last statement to do so is

- $19.r_i$, $22.r_i$, or $25.r_i$, where r_i is an operation of some reader r on processor k, or
- $37[k].v_l$, where v_l is an operation of some writer v.

However, if $Reading[k]$ is last updated by $19.r_i$ or $22.r_i$, then because these statements establish $Reading[k].val = 0$, and because $w.bp$ ranges over $\{1, \ldots, P+2\}$, we have $w.bp \neq Reading[k].val$ at state t, which contradicts (1).

The remaining statements to consider are $25.r_i$ and $37[k].v_l$. The reasoning is the same for each of these statements, so we consider only $25.r_i$. In this case, we have $26.w_j \prec 34[k].w_j \prec 25.r_i \prec 32.w_j$. Consider the relative ordering of $34[k].w_j$ and $24.r_i$. If $34[k].w_j \prec 24.r_i$, then we have $26.w_j \prec 34[k].w_j \prec 24.r_i$. Because $Latest = w.\ell$ holds at t, the value of $Latest$ does not change in the interval between $26.w_i$ and state t. Let L denote the value of $Latest$ in this interval. Then, $25.r_i$ clearly establishes $Reading[k].val = L.val$, which also holds at t. Because each writer chooses a slot that differs from the previous value of $Latest$, $w.bp \neq L.val$ also holds at t. However, this contradicts (1).

The remaining possibility is that $24.r_i \prec 34[k].w_j$ holds. Because r_i executed statement 24, the `CAS` at either $19.r_i$ or $22.r_i$ succeeded. Without loss of generality, assume the one at $22.r_i$ succeeded. Because $24.r_i \prec 34[k].w_j$, we have $22.r_i \prec 34[k].w_j \prec 25.r_i$. Now, either w_j updates $Reading[k]$ by performing a successful `CAS` at $37[k].w_j$, or some other process updates $Reading[k]$ between $22.r_i$ and $37[k].w_j$. In either case, the value of $Reading[k]$ is changed after $22.r_i$ and before state t. Because $Reading[k]$ is last updated by $25.r_i$, it must be the case the $Reading[k]$ is updated between $22.r_i$ and $25.r_i$. Because $Reading[k]$ is updated only by `CAS` operations that increment its *tag* field, this implies that the `CAS` operation at $25.r_i$ fails, which is a contradiction. □

invariant $q@\{14\} \wedge r@\{14\} \wedge q.rd = r.rd \wedge q.wc = r.wc \Rightarrow q.w = r.w$ (I2)

Proof: Suppose, to the contrary, that

$$q@\{14\} \wedge r@\{14\} \wedge q.rd = r.rd \wedge q.wc = r.wc \wedge q.w \neq r.w$$

holds at some state t. Let q_i and r_j be the current operations of processes q and r, respectively, at state t. Without loss of generality, assume that r_j has higher priority than q_i.

Because $q.rd = r.rd$ holds at t, q_i and r_j are attempting to help the same operation, say p_l, and hence they are executing on the same processor, say processor k. Let b (respectively, c) denote the value of $q.buf$ (respectively, $q.wc$) at state t. Extending our notation for statement executions within **for** loops, let $12[b,c].q_i$ denote the execution of statement 12 by q_i with $q.buf = b$ and $q.wc = c$ (and similarly for $12[b,c].r_j$). Then, because $q.wc = r.wc$ holds at t, and because q_i and r_j are helping the same operation, $12[b,c].q_i$ (respectively,

$12[b,c].r_j$) is the last execution of statement 12 by q_i (respectively, r_j) before state t. Moreover, because r_j has higher priority than q_i, we have $12[b,c].q_i \prec 1.r_j \prec \cdots \prec 12[b,c].r_j$. We claim that *Reading*$[k]$ is not updated between $12[b,c].q_i$ and $12[b,c].r_j$. This is because each read operation helps any pending read operation before invoking `UpdateReading` (and operation p_l is pending within this interval), and because each writer can update *Reading*$[k]$ only if *Reading*$[k]$.*val* is zero (and it is nonzero while p_l is pending).

Since $q.w \neq r.w$ holds at t, $In[b][c]$ must be updated by some process between $12[b,c].q_i$ and $12[b,c].r_j$. However, *Reading*$[k]$ is not modified between these two statement executions, so by (I2), this is impossible. $\Box$

To be linearizable [14], it must be possible to define a "linearization point" for each operation, which is some statement execution during its execution. These linearization points define a total order on operations. It is required that each read operation return the most-recently written value according to this total ordering.

Consider a write operation w_j. By (I1), w_j cannot reuse any buffer that is being read from by some reader. It follows from this that the value that w_j writes becomes available (atomically) if its `CAS` at statement 32 succeeds. If this `CAS` fails, then some other write operation v_l must have succeeded in updating *Latest* during w_j's execution. Such an operation v_l may actually swap in w_j's input value (this could happen if v_l's `CAS` at statement 30 fails). In this case, w_j can be linearized to the statement execution of v_l that updates *Latest* (i.e., $32.v_l$). If no overlapping write that updates *Latest* swaps in w_j's input value, then w_j can be linearized to occur immediately before some overlapping write.

Now, consider a read operation r_i on processor k. Note that the read operations on processor k are applied in the order in which they are announced (statement 5). (I1) and (I2) ensure that all return values are correctly determined. Prior to announcing its operation, r_i first calls `UpdateReading` (statement $3.r$). While r_i executes `UpdateReading`, *Reading*$[k]$ must be updated at least once (by r, or by another reader on processor k executing statement 19, 22, or 25, or by a writer executing statement $37[k]$). Because r_i attempts to set *Reading*$[k]$.*val* to zero a second time in `UpdateReading` if its first attempt fails, it can be shown that the last update of *Reading*$[k]$ prior to the execution of statement $5.r_i$ (when r_i announces its operation) is preceded by a read of *Latest* (statement 24 or 35) that occurs during r_i's execution. It is relatively easy to see that r_i linearizes to this read of *Latest*. From the results of this subsection, 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. $\Box$

3.2 Variations

If there is only one writer, Algorithm 1 can be simplified. The resulting algorithm, Algorithm 2, is shown in Figure 2. In Algorithm 2, the `Write` procedure has been shortened considerably. In particular, the comparison at statement 29 in Algorithm 1 is no longer needed because in a single-writer system, it can never fail. Similarly, the `CAS` operations at statements 30 and 32 can be reduced to simple assignments. However, even with these

```
shared var
    Buffer: array[1..P+2][1..B] of wordtype;
    Latest: 1..P+2 initially 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 0..P+2 initially 1

private var
    next, bf, n: 1..P+2;   ℓ: 1..P+2;
    inuse: array[0..P+2] of boolean;
    wc: 0..B;   rd: 0..R;   wd: wordtype;
    in: array[1..B] of wordtype

initially Buffer[1] = initial value

procedure Read(rid, myproc)
        returns array[1..B] of wordtype
1:  rd := Reader[myproc];
2:  if rd ≠ 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:  bf := Reading[myproc];
9:  wc := Wdcnt[rd];
10: while Reader[myproc] = rd ∧ wc > 0 do
11:     wd := Buffer[bf][wc];
12:     if Reader[myproc] = rd then
13:         Out[rd][wc] := wd
        fi;
14:     Wdcnt[rd] := (wc + 1) mod (B + 1);
15:     wc := Wdcnt[rd]
    od;
16: Reader[myproc] := 0

procedure UpdateReading(myproc)
17: Reading[myproc] := 0;
18: ℓ := Latest;
19: CAS(&Reading[myproc], 0, ℓ)

procedure Write(wid, in)
20: bf := FindNext();
21: for n := 1 to B do
22:     Buffer[bf][n] := in[n]
    od;
23: Latest := bf

procedure FindNext() returns 1..P+2
24: ℓ := Latest;
25: for n := 1 to P do
26:     CAS(&Reading[n], 0, ℓ)
    od;
27: for n := 1 to P+2 do
28:     inuse[n] := false
    od;
29: inuse[ℓ] := true;
30: for n := 1 to P do
31:     inuse[Reading[n]] := true
    od;
32: next := 1;
33: while inuse[next] ∧ next < P+2 do
34:     next := next + 1
    od;
35: return next
```

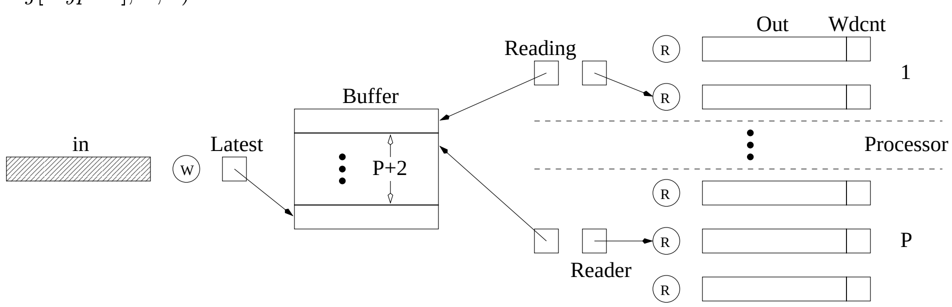

Figure 2: Algorithm 2: Single-writer buffer for priority-based multiprocessors.

optimizations, the writer is still doing more work than is necessary due to the copy-and-swap approach used in Algorithm 1. Because there is no threat of interference by writers, the writer can simply copy its new value directly into the unused object slot (see statements 20-23 of Algorithm 2). The elimination of swapping also allows the slots to be directly referenced, which eliminates the need for the *Bufptr* array. In addition, with only one writer, it becomes much easier to update the local *Reading* variable, so `UpdateReading` can be simplified considerably (see statements 17-19 of Algorithm 2).

Algorithm 1 can also be optimized to implement a multi-writer buffer in a uniprocessor system. The

```
shared var
    In: array[1..W+3][1..B] of wordtype;
    Bufptr: array[1..3] of tagged(1..W+3);
    Latest: tagged(1..3) initially (0,1);
    Out: array[1..R][1..B] of wordtype;
    Wdcnt: array[1..R] of 0..B initially 0;
    Reader: 0..R initially 0;
    Reading: 0..3 initially 1

private var
    bf, cbf: 1..W+3;   nbf: tagged(1..W+3);   n: 1..B;
    ℓ: tagged(1..3);   bp: 0..3;   wc: 0..B;   rd: 0..R;
    wd: wordtype;
                                                        [ 2, 3, 2 ]
    next: array[1..3][1..3] of 1..3 initially           [ 3, 3, 1 ]
                                                        [ 2, 1, 1 ]

initially In[1] = initial value ∧ (∀y: 1 ≤ y ≤ 3: Bufptr[y] = (0, y)) ∧ (∀w: 1 ≤ w ≤ W: w.cbf := 3 + w)

procedure Read(rid)
        returns array[1..B] of wordtype
    rd := Reader;
    if rd ≠ 0 then Help-Read(rd) fi;
    UpdateReading();
    Wdcnt[rid] := 1;
    Reader := rid;
    Help-Read(rid);
    return Out[rid]

procedure Help-Read(rd)
    bp := Reading;
    bf := Bufptr[bp].val;
    wc := Wdcnt[rd];
    while Reader = rd ∧ wc > 0 do
        wd := In[bf][wc];
        if Reader = rd then
            Out[rd][wc] := wd
        fi;
        Wdcnt[rd] := (wc + 1) mod (B + 1);
        wc := Wdcnt[rd]
    od;
    Reader := 0

procedure UpdateReading()
    Reading := 0;
    ℓ := Latest;
    CAS(&Reading, 0, ℓ.val)

procedure Write(wid)
    ℓ := Latest;
    bp := FindNext();
    nbf := Bufptr[bp];
    if ℓ = Latest then
        if CAS(&Bufptr[bp], nbf, (nbf.tag+1, cbf)) then
            cbf := nbf.val
        fi;
        CAS(&Latest, ℓ, (ℓ.tag+1, bp))
    fi

procedure FindNext() returns 1..3
    ℓ := Latest;
    CAS(&Reading, 0, ℓ.val);
    return next[Reading][ℓ.val]
```

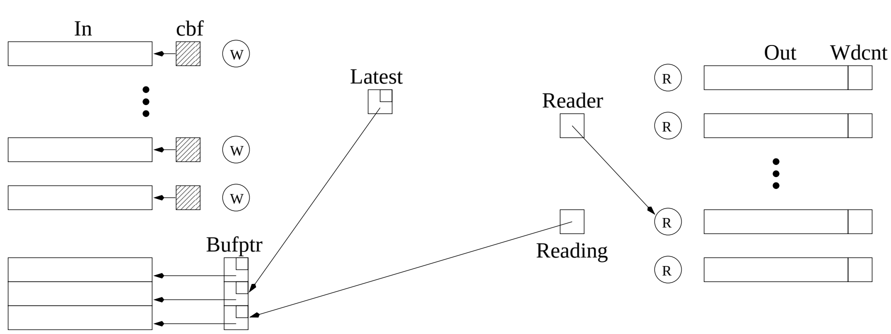

Figure 3: Algorithm 3: Multi-writer buffer for priority-based uniprocessors.

```
shared var
    Buffer: array[1..3][1..B] of wordtype;
    Latest: 1..3 initially 1;
    Out: array[1..R][1..B] of wordtype;
    Wdcnt: array[1..R] of 0..B initially 0;
    Reader: 0..R initially 0;
    Reading: 0..3 initially 1

private var
    bf, ℓ: 1..3;   n: 1..B;
    wc: 0..B;   rd: 0..R;   wd: wordtype;
    in: array[1..B] of wordtype;
    next: array[1..3][1..3] of 1..3 initially [2,3,2; 3,3,1; 2,1,1]

initially Buffer[1] = initial value

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

procedure Help-Read(rd)
8:  bf := Reading;
9:  wc := Wdcnt[rd];
10: while Reader = rd ∧ wc > 0 do
11:     wd := Buffer[bf][wc];
12:     if Reader = rd then
13:         Out[rd][wc] := wd
        fi;
14:     Wdcnt[rd] := (wc + 1) mod (B + 1);
15:     wc := Wdcnt[rd]
    od;
16: Reader := 0

procedure UpdateReading()
17: Reading := 0;
18: ℓ := Latest;
19: CAS(&Reading, 0, ℓ)

procedure Write(wid, in)
20: bf := FindNext();
21: for n := 1 to B do
22:     Buffer[bf][n] := in[n]
    od;
23: Latest := bf

procedure FindNext() returns 1..3
24: ℓ := Latest;
25: if Reading = 0 then
26:     Reading := ℓ
    fi;
27: return next[Reading][ℓ]
```

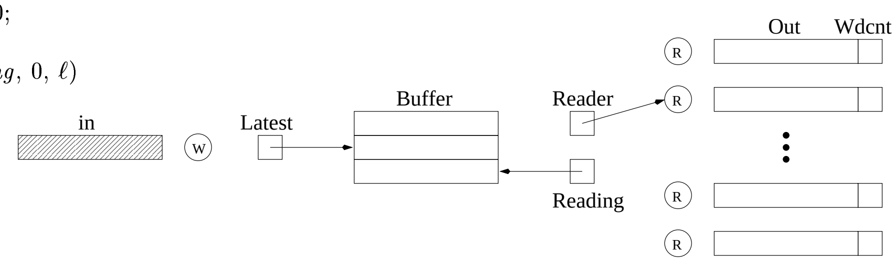

Figure 4: Algorithm 4: Single-writer buffer for priority-based uniprocessors.

resulting algorithm, Algorithm 3, is shown in Figure 3. One obvious change here is the elimination of all processor references. In addition, on a uniprocessor, only three slots are needed. Thus, in `FindNext`, a free slot can be found in constant time using a simple table lookup. Such a lookup mechanism was also used by Chen and Burns. Because readers are scheduled by priority, `UpdateReading` can be simplified as before.

Algorithm 3 can be further simplified if there is only one writer. The resulting algorithm, Algorithm 4, is shown in Figure 4. In Algorithm 4, the `CAS` operation in `FindNext` has been replaced by a simple assignment. This is possible because the writer can only detect stalled updates of *Reading* initiated by lower-priority readers at statement 25 of Algorithm 4. Such readers cannot resume execution until the writer completes. Also, direct copying has again replaced the swapping approach.

4 Quantum-based Algorithms

In this section, we present several pure-buffer algorithms for quantum-based systems. As before, we begin by presenting a multi-writer algorithm that can be used in a multiprocessor system. We then show that this algorithm can be simplified if there is only one writer or if used in a single-processor system.

4.1 Algorithm 5: Multi-writer Buffer for Quantum-based Multiprocessors

Our multi-writer algorithm for quantum-based multiprocessors is shown in Figure 5. Unlike the previous priority-based algorithms, our quantum-based algorithms allow 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 algorithms. 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.

Detailed description. The shared variables used in Algorithm 5 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 5.) 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 $\mathit{Reading}[k]$. `UpdateReading` is exactly the same as in Algorithm 1. After invoking `UpdateReading`, $\mathit{Reading}[k]$ 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 $\mathit{Reading}[k]$ has changed. If not, then r can safely return. If $\mathit{Reading}[k]$ *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 in exactly the same way. 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.

```
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;   ℓ: tagged(1..P+1);
    cbf, rb: 1..W+P+1;   nbf: tagged(1..W+P+1);
    rbp: tagged(0..P+1);   n: 1..max(B, P+1);
    inuse: array [0..P+1] of boolean

initially In[1] = initial value ∧ (∀y: 1 ≤ y ≤ P+2: Bufptr[y] = (0, y)) ∧ (∀w: 1 ≤ w ≤ W: 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 ≠ 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 ≠ Reading[myproc] ∨ 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 ¬succ then
17:     rb := Reading[myproc];
18:     succ := CAS(&Reading[myproc], rb, (rb.tag + 1, 0))
    fi;
19: if succ then
20:     ℓ := Latest;
21:     CAS(&Reading[myproc], (rb.tag+1,0), (rb.tag+2,ℓ.val))
    fi

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

procedure FindNext(myproc) returns 1..P+1
29: for n := 1 to P do
30:     if n ≠ 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] ∧ next < P+1 do
42:     next := next + 1
    od;
43: return next
```

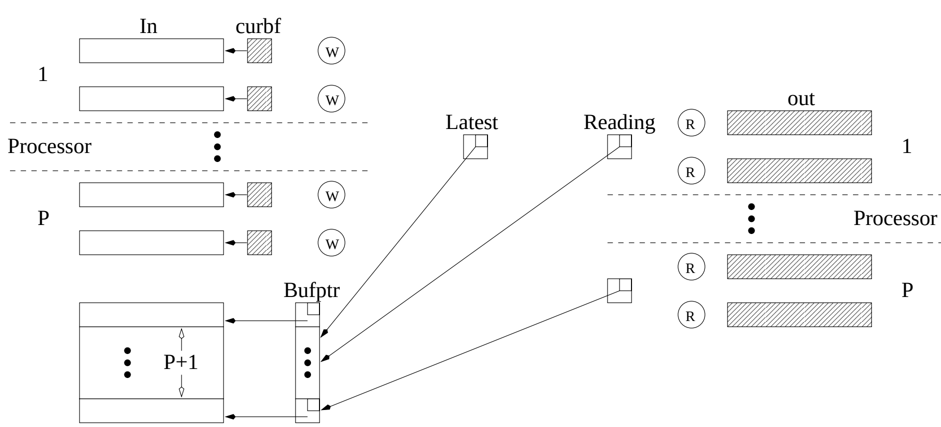

Figure 5: Algorithm 5: Multi-writer buffer for quantum-based multiprocessors.

Correctness proof. The following invariant is analogous to (I1) of Algorithm 1.

invariant $w@\{28\} \wedge \mathit{Latest} = w.\ell \Rightarrow (\forall k : 1 \leq k \leq P : w.bp \neq \mathit{Reading}[k].val)$ (I3)

Proof: Suppose, to the contrary, that

$$w@\{28\} \wedge \mathit{Latest} = w.\ell \wedge w.bp = \mathit{Reading}[k].val \quad (2)$$

holds at some state t. Let w_j be the current operation of w at t. We begin by considering the case in which $k \neq w.myproc$. In this case, w_j executes statements $31[k]$ through $34[k]$. If no process updates $\mathit{Reading}[k]$ between $31[k].w_j$ and state t, then because each writer chooses a slot that differs from $\mathit{Reading}[k].val$, we have $w.bp \neq \mathit{Reading}[k].val$ at t, which contradicts (2).

Otherwise, $\mathit{Reading}[k]$ *is* updated between $31[k].w_j$ and state t, and the last statement to do so is

- $15.u_i$, $18.u_i$, or $21.u_i$, where u_i is an operation of some reader or writer u on processor k, or
- $34[k].v_l$, where v_l is an operation of some writer v.

However, if $\mathit{Reading}[k]$ is last updated by $15.u_i$ or $18.u_i$, then because these statements establish $\mathit{Reading}[k].val = 0$, and because $w.bp$ ranges over $\{1, \ldots, P+1\}$, we have $w.bp \neq \mathit{Reading}[k].val$ at state t, which contradicts (2).

The remaining statements to consider are $21.u_i$ and $34[k].v_l$. Similar reasoning applies to both statements, so we consider only $21.u_i$. In this case, we have $22.w_j \prec 31[k].w_j \prec 21.u_i \prec 28.w_j$. Consider the relative ordering of $31[k].w_j$ and $20.u_i$. If $31[k].w_j \prec 20.u_i$, then we have $22.w_j \prec 31[k].w_j \prec 20.u_i$. Because $\mathit{Latest} = w.\ell$ holds at t, the value of *Latest* does not change in the interval between $22.w_i$ and state t. Let L denote the value of *Latest* in this interval. Then, $21.u_i$ clearly establishes $\mathit{Reading}[k].val = L.val$, which also holds at t. By updating its local *Reading* variable, a write ensures that the previous value of *Latest* is avoided when choosing a new slot. Therefore, $w.bp \neq L.val$ also holds at t. However, this contradicts (2).

The remaining possibility is that $20.u_i \prec 31[k].w_j$ holds. Because u_i executed statement 20, the `CAS` at either $15.u_i$ or $18.u_i$ succeeded. Without loss of generality, assume the one at $18.u_i$ succeeded. Because $20.u_i \prec 31[k].w_j$, we have $18.u_i \prec 31[k].w_j \prec 21.u_i$. Now, either w_j updates $\mathit{Reading}[k]$ by performing a successful `CAS` at $34[k].w_j$, or some other process updates $\mathit{Reading}[k]$ between $18.u_i$ and $34[k].w_j$. In either case, the value of $\mathit{Reading}[k]$ is changed after $18.u_i$ and before state t. Because $\mathit{Reading}[k]$ is last updated by $21.u_i$, it must be the case the $\mathit{Reading}[k]$ is updated between $18.u_i$ and $21.u_i$. Because $\mathit{Reading}[k]$ is updated only by `CAS` operations that increment its *tag* field, this implies that the `CAS` operation at $21.u_i$ fails, which is a contradiction.

In the remainder of the proof, we consider the case in which $k = w.myproc$. Reasoning as above, we have that $\mathit{Latest} = L$ holds throughout the interval after $22.w_i$ and before state t. Because $k = w.myproc$, w_i invokes `UpdateReading` in this interval (statement $35.w_i$). By reasoning as above, if $\mathit{Latest} = L$ holds throughout an interval that includes a call to `UpdateReading`(k), then $\mathit{Reading}[k].val = L.val$ holds after the invocation of

`UpdateReading` and continues to hold while *Latest* $= L$. Because each writer selects a slot that differs from *Latest*, this implies that $w.bp \neq \mathit{Reading}[k].\mathit{val}$ at state t, which contradicts (2). □

We now argue that each operation is linearizable. Consider a write operation w_j. By (I3), w_j cannot reuse any buffer that is being read from by some reader. It follows from this that the value that w_j writes becomes available (atomically) if its `CAS` at statement 28 succeeds. If this `CAS` fails, then some other write v_l must have succeeded in updating *Latest* during w_j's execution. In this case, like in Algorithm 1, w_j can be linearized to occur either immediately before or after a write to *Latest* by some overlapping write operation.

By (I3) and Axiom 2, each read operation r_i returns a value that was written by some write operation. Arguing as in Algorithm 1, we can show that there must exist a state during r_i's execution at which *Latest* points to a slot whose value equals that returned by r_i. (The argument here is nearly identical to that given for Algorithm 1.) This implies that reads can be correctly linearized. From the results of this subsection, we have the following theorem.

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. □

4.2 Variations

When applied to implement a single-writer buffer in a multiprocessor system, most of the optimizations used to obtain Algorithm 2 from Algorithm 1 can be applied to Algorithm 5. The resulting algorithm, Algorithm 6, is shown in Figure 6. As before, a direct copying approach is used instead of swapping. However, the optimizations of `UpdateReading` in Algorithm 2 cannot be applied in a quantum-scheduled system, so it remains unchanged.

When applied to implement a mutli-writer buffer in a uniprocessor system, Algorithm 5 can be reduced to a surprisingly simple algorithm. The resulting algorithm, Algorithm 7, is shown in Figure 7. On a uniprocessor, the call to `UpdateReading` at statement 35 in Algorithm 5 will *always* return the slot indicated by *Latest*. Therefore, statements 36-42 in Algorithm 5 will simply choose a new slot that differs from *Latest*. Thus, in Algorithm 7, the writer can simply alternate between two slots in successive operations. In addition, because the order of the writes is fixed in this manner, the *Reading* and *Latest* variables can be merged into a single variable, which is called *Ver* in Algorithm 7. When taken modulo-two, this merged variable gives the index of the most-recently written slot. Hence, `FindNext` can be replaced by statements 10-13 and 18-20 in Algorithm 7. Note that statements 18-20 are executed only if there is a preemption by another writer within statements 10-13. By Axiom 2, there can be at most one such preemption, so `CAS` operations are not needed in statements 10-13.

Algorithm 7 can be further simplified to implement a single-writer buffer. The resulting algorithm, Algorithm 8, is shown in Figure 8. With just one writer, write operations cannot preempt each other. Thus, statements 10-18 in Algorithm 7 are not necessary.

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

initially Buffer[1] = initial value

procedure Read(out, myproc)
        returns array [1..B] of wordtype
    UpdateReading(myproc);
    rb := Reading[myproc];
    if rb.val ≠ 0 then
        for n := 1 to B do
            out[n] := Buffer[rb.val][n]
        od
    fi;
    if rb ≠ Reading[myproc] ∨ rb.val = 0 then
        UpdateReading(myproc);
        rb := Reading[myproc];
        for n := 1 to B do
            out[n] := Buffer[rb.val][n]
        od
    fi;
    return out

procedure UpdateReading(myproc)
    rb := Reading[myproc];
    succ := CAS(&Reading[myproc], rb, (rb.tag + 1, 0));
    if ¬succ then
        rb := Reading[myproc];
        succ := CAS(&Reading[myproc], rb, (rb.tag + 1, 0))
    fi;
    if succ then
        ℓ := Latest;
        CAS(&Reading[myproc], (rb.tag+1,0), (rb.tag+2,ℓ))
    fi
```

```
private var
    out: array [1..B] of wordtype;
    in: array [1..B] of wordtype;
    next, ℓ, bf: 1..P+1;   succ: boolean;
    rb: tagged(0..P+1);   n: 1..max(B, P+1);
    inuse: array [0..P+1] of boolean

procedure Write(in, myproc)
    bf := FindNext();
    for n := 1 to B do
        Buffer[bf][n] := in[n]
    od;
    Latest := bf

procedure FindNext(myproc) returns 1..P+1
    ℓ := Latest;
    for n := 1 to P do
        if n ≠ myproc then
            rb := Reading[n];
            if rb.val = 0 then
                CAS(&Reading[n], rb, (rb.tag+1,ℓ))
            fi
        else UpdateReading(myproc)
        fi
    od;
    for n := 1 to P+1 do
        inuse[n] := false
    od;
    for n := 1 to P do
        inuse[Reading[n].val] := true
    od;
    next := 1;
    while inuse[next] ∧ next < P+1 do
        next := next + 1
    od;
    return next
```

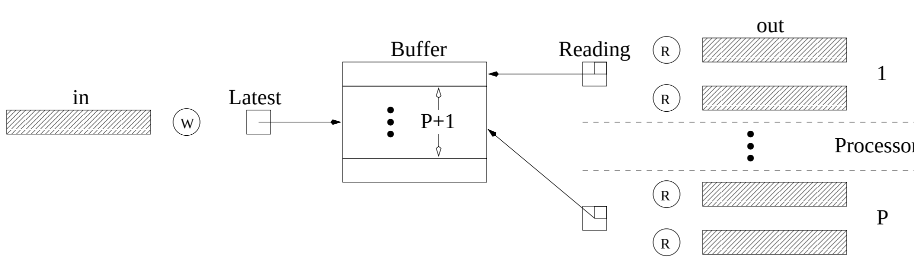

Figure 6: Algorithm 6: Single-writer buffer for quantum-based multiprocessors.

5 Concluding Remarks

We have shown that characteristics of real-time systems can be exploited to implement highly-optimized wait-free shared buffers. Moreover, we have presented the first pure-buffer algorithms for the multi-writer case. When viewing the number of processors P as a constant, our algorithms have optimal space and time complexity.

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

shared var
      Buffer: array [1..2][1..B] of wordtype;
      Ver: tagged(1..W) initially (0,1)

initially
      Buffer[1] = initial value

procedure Read(out)
            returns array [1..B] of wordtype
      v := Ver;
      bf := (v.tag mod 2)+1;
      for n := 1 to B do
          out[n] := Buffer[bf][n]
      od;
      if v.tag ≠ Ver.tag then
          bf := (Ver.tag mod 2)+1;
          for n := 1 to B do
              out[n] := Buffer[bf][n]
          od
      fi;
      return out

private var
      out: array [1..B] of wordtype;   pm: boolean;
      in: array [1..B] of wordtype;   bf: 1..2;
      n: 1..B;   v: tagged(1..W)

procedure Write(in,wid)
      pm := false;
      v := Ver;
      if CAS(&Ver, v, (v.tag,wid)) then
          bf := ((v.tag+1) mod 2)+1;
          for n := 1 to B do
              Buffer[bf][n] := in[n]
          od;
          pm := ¬CAS(&Ver, (v.tag,wid), (v.tag+1,wid))
      else pm := true
      fi;
      if pm then
          v := Ver;
          bf := ((v.tag+1) mod 2)+1;
          for n := 1 to B do
              Buffer[bf][n] := in[n]
          od;
          Ver := (v.tag+1,wid)
      fi
```

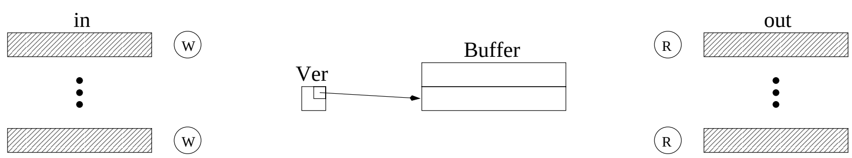

Figure 7: Algorithm 7: Multi-writer buffer for quantum-based uniprocessors.

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 M. Gouda. A criterion for atomicity. *Formal Aspects of Computing: The International Journal of Formal Methods*, 4(3):273–298, 1992.

[2] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. *Proceed-*

```
shared var
    Buffer: array [1..2][1..B] of wordtype;
    Ver: integer initially 0

initially Buffer[1] = initial value

private var
    out: array [1..B] of wordtype;
    in: array [1..B] of wordtype;
    v: integer;   bf: 1..2;   n: 1..B

procedure Read(out)
        returns array [1..B] of wordtype
    v := Ver;
    bf := (v mod 2) + 1;
    for n := 1 to B do
        out[n] := Buffer[bf][n]
    od
    if v ≠ Ver then
        bf := (Ver mod 2) + 1;
        for n := 1 to B do
            out[n] := Buffer[bf][n]
        od
    fi;

    return out

procedure Write(in)
    v := Ver;
    bf := ((v + 1) mod 2) + 1;
    for n := 1 to B do
        Buffer[bf][n] := in[n]
    od;
    Ver := v + 1
```

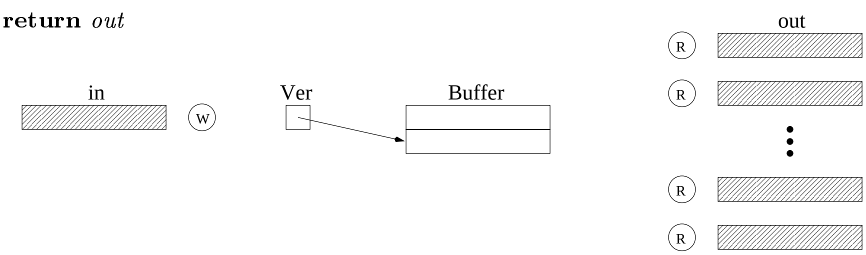

Figure 8: Algorithm 8: Single-writer buffer for quantum-based uniprocessors.

ings 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] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. *Real-Time Databases: Issues and Applications.* Kluwer Academic Publishers, Amsterdam, 1997.

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

[9] B. Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computer Systems*, 37(12):1506–1514, Dec. 1988.

[10] J. Burns and G. Peterson. Constructing multi-reader atomic values from non-atomic values. *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pp. 222–231, Aug. 1987.

[11] 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.

[12] 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.

[13] 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.

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

[15] A. Israeli and M. Li. Bounded time-stamps. *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pp. 371–382, 1987.

[16] L. Kirousis, E. Kranakis, and P. Vitanyi. Atomic multireader register. *Proceedings of the Second International Workshop on Distributed Algorithms*, pp. 278–296, October 1987.

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

[18] M. Li, J. Tromp, and P. Vitanyi. How to construct wait-free variables. *Proceedings of International Colloquium on Automata, Languages, and Programming*, pp. 288–505, 1989.

[19] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, pp. 232–248, 1987.

[20] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[21] 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.

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

[23] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, Boston, 1991.

[24] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proceedings of the Ninth IEEE Real-Time Systems Symposium*, pp. 259–269. 1988.

[25] 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.

[26] 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.

[27] 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.

[28] J. Tromp. How to construct an atomic variable. *Proceedings of the Third International Workshop on Distributed Algorithms*, pp. 292–302. Lecture Notes in Computer Science 392, Springer-Verlag, 1989.

[29] 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.

[30] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science*, pp. 233–243, 1986.