

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 a wait-free algorithm for implementing multi-writer read/write pure-buffers in priority-based multiprocessor real-time systems. For a B -word buffer shared across a constant number of processors, the time complexity for reading and writing in our algorithm 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.

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. Conventional mechanisms for bounding priority inversions (see, for example, [11]) 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, 6, 7, 10]. 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 a new wait-free implementation of read/write buffers that is highly optimized for use in priority-based real-time systems.

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 (see, for example, [8, 9, 12]). Though these constructions could be used to implement read/write buffers in a real-time system, commonly available and stronger synchronization primitives can be used to produce simpler and more efficient implementations.

Chen and Burns recently showed that, by using *compare-and-swap* and *test-and-set*,² it is possible to efficiently implement a single-writer wait-free buffer [6, 7]. Their algorithm is a "pure-buffer" algorithm [5]. 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.

Recent research at the University of North Carolina has shown that algorithms for real-time systems can be simplified by exploiting the way that processes are scheduled for execution [2, 3, 10]. 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. This fact can be exploited to obtain algorithms that have complexities that are a function of the number of processors in the system, not the number of processes.

The algorithm presented in this paper exploits priority-scheduled systems in this way to improve the algorithm's performance. Complexity figures for our algorithm are listed in Table 1. In the full version of this paper [1], simplified versions of this algorithm

¹ Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, and ITR 0082866. Longer version of this paper to be presented at the Seventh International Conference on Real-Time Systems and Applications (RTCSA 2000).

²Their 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	$\Theta(B)$	$\Theta(R + B)$	$\Theta(RB)$
Our Algorithm	P/W	Priority-based	$O(B)$	$\Theta(P + B)$	$\Theta(PB)$

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

are also included for more restricted cases, such as single-writer buffers and buffers for uniprocessor systems, along with algorithms for quantum-scheduled 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 our algorithm, the time complexity for reading is comparable to Chen and Burns’ algorithm, and the time complexity for writing is better. In addition, our algorithm has better space complexity than their algorithm. If P is viewed as a constant, which is reasonable for most systems, then the time complexity for reading and writing in our algorithm is $O(B)$, and the space complexity is $\Theta(B)$; these complexity figures are obviously asymptotically optimal.

2 Preliminaries

Our buffer algorithm 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. 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

In our algorithm, 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 $\{0, \dots, 2k + 1\}$ for some $k \in \mathbb{N}$ 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 $\text{tagged}(1..W+P+2)$ has a *tag* field that is a bounded integer, and a *val* field that ranges over $\{1, \dots, W + P + 2\}$.

Our algorithm also uses compare-and-swap (CAS) operations. Such operations are denoted $\text{CAS}(\text{adr}, \text{old}, \text{new})$, 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. We assume that each labeled statement in each algorithm is atomic and that all private variables of a process retain their values between operations on the implemented buffer by that process.

3 Priority-based Algorithm

Our multi-writer algorithm for priority-based multiprocessors is shown in Figure 1. We reduce the $R + 2$ pure buffers of Chen and Burns’ algorithm to $P + 2$ by ensuring that there is at most one active reader on each 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. In the worst case, each of the P readers is reading a distinct value other than the most recently written value when the writer is invoked. Therefore, $P + 2$ pure buffers are necessary to ensure the availability of a pure buffer to write to.

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, making them 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 a dedicated input buffer ($\text{In}[cbf]$) that can then be swapped with one of the pure buffers. 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

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)

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

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: *bp* := *Reading*[*myproc*].*val*;
9: *bf* := *Bufptr*[*bp*].*val*;
10: *wc* := *Wdcnt*[*rd*];
11: while *Reader*[*myproc*] = *rd* ∧ *wc* > 0 do
12: *wd* := *In*[*bf*][*wc*];
13: if *Reader*[*myproc*] = *rd* then
14: *Out*[*rd*][*wc*] := *wd*
15: fi;
16: *Wdcnt*[*rd*] := (*wc* + 1) mod (*B* + 1);
17: *wc* := *Wdcnt*[*rd*]
18: od;
19: *Reader*[*myproc*] := 0

procedure UpdateReading(*myproc*)

20: *rb* := *Reading*[*myproc*];
21: *succ* := CAS(&*Reading*[*myproc*], *rb*, (*rb.tag* + 1, 0));
22: if ¬*succ* then
23: *rb* := *Reading*[*myproc*];
24: *succ* := CAS(&*Reading*[*myproc*], *rb*, (*rb.tag* + 1, 0))
25: fi;
26: if *succ* then
27: *l* := *Latest*;
28: CAS(&*Reading*[*myproc*], (*rb.tag*+1,0), (*rb.tag*+2,*l.val*))
29: fi

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

procedure Write(*wid*)

26: *l* := *Latest*;
27: *bp* := FindNext();
28: *nbf* := *Bufptr*[*bp*];
29: if *l* = *Latest* then
30: if CAS(&*Bufptr*[*bp*], *nbf*, (*nbf.tag*+1, *cbf*)) then
31: *cbf* := *nbf.val*
32: fi;
33: CAS(&*Latest*, *l*, (*l.tag*+1, *bp*))
34: fi

procedure FindNext() returns 1..P+2

35: for *n* := 1 to *P* do
36: *rb* := *Reading*[*n*];
37: *val* := *Latest.val*;
38: if *rb.val* = 0 then
39: CAS(&*Reading*[*n*], *rb*, (*rb.tag*+1, *val*))
40: fi
41: od;
42: for *n* := 1 to *P*+2 do
43: *inuse*[*n*] := false
44: od;
45: *inuse*[*Latest.val*] := true;
46: for *n* := 1 to *P* do
47: *inuse*[*Reading*[*n*].*val*] := true
48: od;
49: *next* := 1;
50: while *inuse*[*next*] ∧ *next* < *P*+2 do
51: *next* := *next* + 1
52: od;
53: return *next*

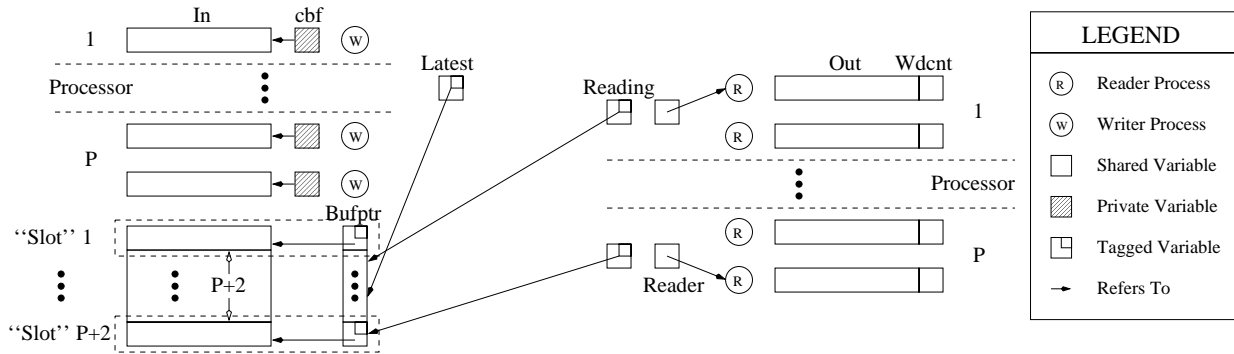


Figure 1: Algorithm 1: Multi-writer buffer for priority-based multiprocessors.

per-buffer overhead.

Mechanisms. Our algorithm uses three distinct mechanisms. The first mechanism, helping, is used to manage reader-reader interactions on the same processor. Before a reader on processor p may begin its operation, it must “announce” it by updating $Reader[p]$ (statement 5). However, before announcing its operation, a reader must first help complete any unfinished previously-announced (lower-priority) read operation on its processor (statements 1-3). As mentioned previously, this mechanism ensures that there is at most one active reader per processor at any time.

The second mechanism, indirect referencing, handles all writer-writer interactions. (Such interactions do not arise in Chen and Burns’ algorithm, since they assume only a single writer.) To avoid confusion, we henceforth refer to the $P + 2$ pure buffers used in the implementation as “slots.” The i^{th} slot is defined by the pair $(Bufptr[i], In[Bufptr[i]])$. The shared variable $Latest$ “points” to the “current” slot, i.e., the slot that gives the implemented buffer’s current contents. To perform a write operation, a writer first finds a “free” slot (statement 27), writes its value into it (statement 30), and then makes that slot the current slot by updating $Latest$ (statement 32). Because different writers may choose the same free slot, slots are updated using indirect referencing. In particular, a writer atomically updates slot i by swapping its cbf pointer with $Bufptr[i]$ (statement 30). If either this swap or the update of $Latest$ fails, then the effect is as if the writer’s value was immediately overwritten by another writer. If the swap of cbf and $Bufptr[i]$ is successful, then the “swapped out” buffer becomes the dedicated write buffer for the writer’s next operation (statement 31).

The third mechanism, the reader-writer handshake, ensures that the “free” slot selected by a writer differs from any that any reader is either currently reading or “about to” read from. This is accomplished by allowing writers to detect and complete any stalled slot selections by readers at statements 33-37. This handshake, along with the check at statement 29, ensures that any update of a “free” slot is safe.

The detailed description and proof of correctness for this algorithm are given in the full version of this paper [1]. From this proof, we have the following.

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. \square*

4 Concluding Remarks

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. Our goal is to produce a library of such implementations that will facilitate the use of wait-free algorithms in real-time applications.

References

- [1] J. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems (full version). Available at <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, 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] L. Lamport. On interprocess communication, parts 1 and 2. *Distributed Computing*, 1:77–101, 1986.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.