Multiword Atomic Read/Write Registers on Multiprocessor Systems

ANDREAS LARSSON
Chalmers University of Technology
ANDERS GIDENSTAM
Max Planck Institute for Computer Science
PHUONG H. HA
University of Tromsø
and
MARINA PAPATRIANTAFILOU and PHILIPPAS TSIGAS
Chalmers University of Technology

Modern multiprocessor systems offer advanced synchronization primitives, built in hardware, to support the development of efficient parallel algorithms. In this article, we develop a simple and efficient algorithm, the ReadersField algorithm, for atomic registers (variables) of arbitrary length. The simplicity and better complexity of the algorithm is achieved via the utilization of two such common synchronization primitives. In this article, we also experimentally evaluate the performance of our algorithm, together with lock-based approaches and a practical, previously known algorithm that is based only on read and write primitives. The experimental evaluation is performed on three well-known parallel architectures. This evaluation clearly shows that both algorithms are practical and that as the size of the register increases the ReadersField algorithm performs better, following its analytical evaluation.

Categories and Subject Descriptors: E.1 [**Data structures**]; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Algorithms, Theory, Performance

Additional Key Words and Phrases: Atomic register, synchronization, wait-free

1.7

Authors' address: Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.
This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC) and is an extended version of the paper that appeared in the Proceedings of ESA 2004, Bergen, Norway, 14–17 September, 2004.

ACM Reference Format:
Larsson, A., Gidenstam, A., Ha, P.H., Papatriantafilou, M., and Tsigas, O. 2008. Multiword atomic read/write registers on multiprocessor systems. ACM J. Exp. Algor. 13, Article 1.7 (November 2008), 30 pages. DOI = 10.1145/1412228.1455262 http://doi.acm.org/10.1145/1412228.1455262

1. INTRODUCTION

Cooperating processes in multiprocessor and multiprocessing systems may share data via shared data objects. In this article, we are interested in designing and evaluating the performance of shared data objects for cooperative tasks in multiprocessor systems. More specifically, we are interested in designing a practical wait-free algorithm for implementing registers (or memory words) of arbitrary length that could be read and written atomically. (Typical modern multiprocessor systems support words of 64-bit size.)

The most commonly required consistency guarantee for shared data objects is *atomicity*, also known as *linearizability* [Herlihy and Wing 1990]. An implementation of a shared object is *atomic* or *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant that lies in its respective time duration, in a way that the effect of each operation is in agreement with the object's sequential specification. The latter means that if we speak of, for example, read/write objects, the value returned by each read equals the value written by the most recent write according to the sequence of "shrunk" operations in the time axis.

The classical well-known and simplest solution for maintaining consistency of shared data objects enforces mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at time to access it. However, mutual exclusion causes large performance degradations, especially in multiprocessor systems [Silberschatz et al. 2001; Sundell 2004] and suffers from potential priority inversion, in which a high-priority task can be blocked for unbounded time by a lower-priority task [Sha et al. 1990]. Several synchronization protocols have been introduced to solve the priority inversion problem for uniprocessor [Sha et al. 1990] and multiprocessor [Rajkumar 1990] systems. The solution presented in [Sha et al. 1990] solves the problem for the uniprocessor case, at the cost of limiting the schedulability of task sets and also making the scheduling analysis of real-time systems hard. The situation is much worse in a multiprocessor real-time system, where a task may be blocked by another task running on a different processor [Rajkumar 1990].

Nonblocking implementation of shared data objects is an alternative approach for the problem of inter-task communication. Nonblocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. They offer significant advantages over lock-based schemes because (1) they do not suffer from priority inversion; (2) they avoid lock convoys; (3) they provide high tolerance to processor failures (process or processor stop failures will never corrupt shared data objects); and (4) they eliminate deadlock scenarios, involving two or more tasks both waiting for locks held by the other. On the other hand, nonblocking protocols have to use

more delicate strategies to guarantee data consistency than the simple enforcement of mutual exclusion between the readers and the writers of the data object.

Nonblocking algorithms can be lock-free or wait-free. Lock-free [Barnes 1993; Herlihy 1993] algorithms guarantee that regardless of the contention and the interleaving of concurrent operations, at least one operation will always make progress. However, there is a risk that the progress of other operations might cause one specific operation to take unbounded time to finish. In a wait-free [Herlihy 1991] algorithm, every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Nonblocking algorithms have been shown to be of good practical importance [Tsigas and Zhang 2001, 2002], and in the recent years, NOBLE, which is a library of non-blocking interprocess communication objects, has been introduced [Sundell and Tsigas 2002].

From a historic perspective, research on nonblocking algorithms stems from the *readers/writers problem*. In this problem, a number of concurrent processes are interested in reading from or writing to a shared data object (here also called a *register*). A read operation as well as a write operation should take effect atomically and return or update the entire state of the shared data object. When the shared data object is larger than a single (atomic) word (of the word size supported by the multiprocessor system at hand) an algorithm is needed to coordinate the readers/writers so that access to the multiword data is atomic. The classical solution is to use mutual exclusion to enforce that either (1) no read or write operations overlap each other, or (2) no write operations overlap each other or any read operation. These methods, normally implemented using a *mutual exclusion lock* or a *readers-writers lock*, respectively, suffer from the drawbacks of mutual exclusion mentioned above. Lamport [1977] introduces a lock-free solution to the readers/writers problem with one writer. Lamport's algorithm is actually wait-free for the writer but lock-free for the readers, since the writer can force a slow reader to retry indefinitely. This algorithm, followed by the first wait-free algorithm by Peterson [1983], marked the start of long running research efforts to construct wait-free solutions to the readers/writers problem.

This problem, also known as the problem of multiword wait-free read/write registers, has become one of the well-studied problems in the area of nonblocking synchronization, with numerous results for the construction of, for example, (1) single-writer single-reader registers [Lamport 1986; Simpson 1990; Chen and Burns 1997]; (2) single-writer n-reader registers [Peterson 1983; Burns and Peterson 1987; Kirousis et al. 1987; Newman-Wolfe 1987; Kopetz and Reisinge 1993; Singh et al. 1994; Haldar and Vidyasankar 1995; Larsson et al. 2004]; (3) 2-writer n-reader registers [Bloom 1988]; and (4) m-writer n-reader registers [Vitányi and Awerbuch 1986; Peterson and Burns 1987; Israeli and Shaham 1992; Li and Vitányi 1992; Li et al. 1996; Haldar and Vidyasankar 1996].

The main goal of the algorithms in the above results is to construct wait-free multi-word read/write registers using single-word read/write registers and not any other synchronization primitives that may be provided by the hardware in a system. This has been very significant, providing fundamental results in the area of wait-free synchronization, especially when we consider the nowadays well-known and well-studied hierarchy of shared data objects and their

```
int swap(int *mem, int new)
{
        tmp := *mem;
        *mem := new;
        return tmp;
}
```

```
int fetch_and_or(int *mem, int arg)
{
        tmp := *mem;
        *mem := tmp | arg;
        return tmp;
}
```

Fig. 1. The specifications of the *swap* and *fetch_and_or* atomic suboperations.

synchronization power [Herlihy 1991]. Many of these solutions also involve elegant and symmetric ideas. Moreover, they have formed the basis for further results in the area of nonblocking synchronization.

Our motivation for further studying this problem is as follows: As the aforementioned solutions were using only read/write registers as components, they necessarily have each write operation on the multiword register write the new value in several copies (roughly speaking, as many copies as we have readers in the system), which may be costly. However, modern computer architectures provide hardware synchronization primitives stronger than atomic read/write registers. Some of them are even accessible at a constant cost-factor away from read/write accesses. We consider it a useful task to investigate how to use this power, to the benefit of designing economical solutions for the same problem, which can lead to structures that are more suitable for being adopted in practice. Moreover, to the best of our knowledge, none of the previous solutions have been implemented and evaluated on real systems.

In this article, we present the ReadersField algorithm, a simple, efficient wait-free algorithm for implementing multi-word n-reader/single-writer registers of arbitrary word length. In the ReadersField algorithm, each multiword write operation only needs to write the new value in one copy, thus having significantly less overhead. The algorithm uses synchronization primitives called *fetch_and_or* and *swap* (c.f. Figure 1 and [Silberschatz et al. 2001]), which are available in several modern processor architectures, to synchronize n readers and a writer accessing the register concurrently. Since the ReadersField algorithm is wait-free, it provides high parallelism for the accesses to the multiword register and thus significantly improves performance. We study the ReadersField algorithm together with the wait-free one in [Peterson 1983], which is also practical and simple, and two lock-based algorithms, one using a single spin-lock and one using a readers-writer spin-lock. We design benchmarks and study these solutions on three multiprocessor architectures [Tanenbaum 2001]: UMA (Uniform Memory Access) Sun-Fire-880 with six processors, ccNUMA (Non-Uniform Memory Access) SGI Origin 2000 with 29 processors and ccNUMA SGI Origin 3800 with 128 processors.

The rest of this article is organized as follows. Section 2 describes the formal requirements of the problem and the related algorithms that we are using in the evaluation study. Section 3 presents our protocol. Section 4 gives the proof of correctness and the complexity of the new protocol. Section 5 is devoted to the experimental study comparing our wait-free protocol with previous work, both nonblocking and lock-based. The article concludes with Section 6, with a discussion on the contributed results and further research issues.

2. BACKGROUND

2.1 System and Problem Model

A *shared register* of arbitrary length [Lamport 1977; Peterson 1983] is an abstract data structure that is shared by a number of concurrent processes, which perform read or write operations on the shared register. In this article, we make no assumption about the relative speed of the processes (i.e., the processes are asynchronous). One of the processes, the *writer*, executes write operations and all other processes, the *readers*, execute read operations on the shared register. The order of execution of steps by each process depends on the consistency model of the underlying system. Consistency models, in general, and the consistency model assumptions of the algorithms in this article are discussed in the next subsection.

An implementation of a register consists of (1) *protocols* for executing the operations (read and write), (2) a data structure consisting of shared *subregisters*, and (3) a set of initial values for these. The protocols for the operations consist of a sequence of operations on the subregisters, called *suboperations*. These suboperations are reads, writes or other atomic operations, such as *fetch_and_or* or *swap* (cf. Figure 1), which either are available directly by the hardware or can be implemented from other, commonly available, hardware synchronization primitives, such as *compare_and_swap* or *load_linked/store_conditional* [Herlihy 1991]. Furthermore, matching the capabilities of modern multiprocessor systems, the subregisters are assumed to be atomic and to support multiple processes.

A register implementation is *wait-free* [Herlihy 1991] if it guarantees that any process will complete each operation in a finite number of steps (suboperations) regardless of the execution speeds of the other processes.

For each operation O, there exists a time interval $[s_O, f_O]$ called its *duration*, where s_O and f_O are the starting and ending times, respectively. There is a precedence relation on the operations that form a strict partial order (denoted '$\rightarrow$'). For two operations a and b, $a \rightarrow b$ means that operation a ended before operation b started. If two operations are incomparable under $\rightarrow$, they are said to *overlap*.

A *reading function* π for a register is a function that assigns a high-level write operation w to each high-level read operation r such that the value returned by r is the value that was written by w (i.e., $\pi(r)$ is the write operation that wrote the value that the read operation r read and returned).

CRITERION 2.1. *A shared register is* atomic *iff the following three conditions hold for all possible executions:*

1. NO-IRRELEVANT. *There exists no read r such that $r \rightarrow \pi(r)$.*
2. NO-PAST. *There exists no read r and write w such that $\pi(r) \rightarrow w \rightarrow r$.*
3. NO-NEW-OLD-INVERSION. *There exist no reads r_1 and r_2 such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$.*

Besides atomicity, there also exist other useful consistency guarantees that have been defined in the literature (e.g., regularity). An implementation of a

shared object is *regular* [Lamport 1986] if it merely guarantees that a read of the object that does not overlap with any write returns the latest written value of the object and that a read that overlaps with a write either returns that written value or the old one. The criterion for proving that a register implementation is regular is given below:

CRITERION 2.2. *A shared register is* regular *iff the following two conditions hold for all possible executions:*

1. NO-IRRELEVANT. *There exists no read r such that $r \rightarrow \pi(r)$.*
2. NO-PAST. *There exists no read r and write w such that $\pi(r) \rightarrow w \rightarrow r$.*

2.2 Memory Consistency Models

There are a number of different memory consistency models in the literature and in existing systems. The system supports a given model if operations on memory follow specific rules.

In the *sequential consistency model*, "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [Lamport 1979]. This implies that assuming the sequential consistency model, we can assume that operations performed by the same process execute sequentially and also that all reads and writes to subregisters are observed in the same order by all the processes/threads. The sequential consistency model is supported by the the SGI Origin [Silicon Graphics 1993; Culler et al. 1998] platforms.

In some consistency, models there are explicit synchronization actions that are treated differently from ordinary memory accesses. Such actions usually involve variables called *synchronizing variables*.

Definition 2.3 (*Synchronizing Variable*). [Dubois et al. 1986] Synchronizing variables are the shared variables that are used to control the concurrency between several processes.

To allow the system to distinguish between accesses to synchronizing variables and to other shared variables at run time, synchronizing variables are required to be accessed via synchronization instructions like atomic read-modify-write or special load/store instructions [Dubois et al. 1986]. Therefore, accesses to such hardware-recognized synchronizing variables are implicitly assumed to use synchronization instructions supported by the system.

One consistency model that distinguishes such variables and is common in multiprocessor systems, such as the SPARC architectures [CORPORATE SPARC International 1994], is the weak consistency model [Dubois et al. 1986]:

Definition 2.4 (*Weak Consistency*). A multiprocessor is weak consistent if:

1. Accesses to global synchronizing variables are strongly ordered (i.e., sequentially consistent).
2. No access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed.

Variable	Type	Description
WFLAG	Boolean	Indicates that the writer is writing in BUF1.
SWITCH	Boolean	To check if the writer has written in BUF1.
READING	Array of n Boolean	Used together with WRITING to handle reader-writer handshaking.
WRITING	Array of n Boolean	Used together with READING to handle reader-writer handshaking.
BUF1	Buffer	Main buffer.
BUF2	Buffer	Backup buffer.
COPYBUF	Array of n buffers	An individual backup buffer for each reader.

Fig. 2. The shared variables used by Peterson's algorithm. The number of readers is n. BUF1 holds the initial register value. All other variables are initialized to 0 or false.

3. No access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

Peterson's algorithm and the lock-based alogorithms studied in this article require sequential consistency. The memory consistency model assumed by the READERSFIELD algorithm need not satisfy sequential consistency; instead, it only needs to satisfy the following properties: (for ease of reference later in the article, we call them the *Cache Weak Consistency* properties).

Definition 2.5 (*Cache Weak Consistency (CWC)*). A multiprocessor is cache weak consistent if:

1. Accesses to global synchronizing variables are *Cache Consistent* [Goodman 1989]. The cache consistency requires that accesses to *the same memory location* be sequentially consistent.
2. No access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed.
3. No access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

The cache weak consistency model is even weaker than the weak consistency model, in particular with respect to the first restriction.

2.3 Peterson's Shared Multiword Register

Peterson [1983] describes an implementation of an atomic shared multiword register for one writer and many (n) readers. The protocol does not use any other atomic suboperations than reads and writes and is described below.

The idea is to use $n + 2$ shared buffers, each of which can hold a value of the register, together with a set of shared handshake variables to make sure that the writer does not overwrite all buffers that are being read by some reader and that each reader chooses a stable but up-to-date buffer to read from. The shared variables are shown in Figure 2 and the protocols for the read and write operations are shown in Algorithm 1.

Algorithm 1. Peterson's algorithm. Lower-case variables are local variables.

```
Read operation by reader r:
PR1    READING[r] := !WRITING[r];
PR2    flag1 := WFLAG;
PR3    sw1 := SWITCH;
PR4    read BUF1;
PR5    flag2 := WFLAG;
PR6    sw2 := SWITCH;
PR7    read BUF2;
PR8    if (READING[r] == WRITING[r])
PR9        return the value in COPYBUF[r];
PR10   else if ((sw1 != sw2) || flag1 || flag2)
PR11       return the value read from BUF2;
PR12   else
PR13       return the value read from BUF1;

Write operation:
PW1    WFLAG := true;
PW2    write to BUF1;
PW3    SWITCH := !SWITCH;
PW4    WFLAG := false;
PW5    for (each reader r)
PW6        if (READING[r] != WRITING[r])
PW7            write to COPYBUF[r];
PW8            WRITING[r] := READING[r];
PW9    write to BUF2;
```

In the algorithm, each reader r has three choices regarding where to get the current register value, namely BUF1, BUF2, and COPYBUF[r]. The primary alternative for a reader is BUF1, which it reads at line PR4. However, if there is a concurrent write, the writer might write to BUF1 (line PW2) while reader r is reading the same buffer. The flags WFLAG and SWITCH are used to detect most of these conflicts. A reader reads WFLAG and SWITCH both before (lines PR2 and PR3) and after (lines PR5 and PR6) it reads BUF1. The writer sets WFLAG to true (line PW1) before it begins to write to BUF1 (line PW2) and flips SWITCH (line PW3) and resets WFLAG to false (line PW4) after it has updated BUF1. Through this handshake mechanism, reader r can detect if (1) the writer was writing to BUF1 when it started to read it, (2) the writer was writing to BUF1 when it finished reading that buffer, or (3) the writer did a complete write of BUF1 (i.e., lines PW1 to PW4) while r was reading that buffer (in this case, SWITCH has been flipped). In all these cases r cannot assume it managed to read BUF1 correctly.

However, by using WFLAG and SWITCH alone, a reader cannot detect the case where the writer managed to complete an even number of writes to BUF1, since in those cases, SWITCH will be back to its original value. The per-reader flags READING[r] and WRITING[r] are used to handle these cases. A reader r makes READING[r] opposite to the writer's corresponding flag WRITING[r] at the beginning of each read operation (line PR1). This is a signal to the writer that it should write the register value also to COPYBUF[r] (lines PW6 and

PW7). When the writer has written COPYBUF[r], it sets WRITING[r] equal to READING[r] (line PW8) to signal the reader that COPYBUF[r] now contains a valid register value. If the reader detects this signal (line PR8) (i.e., when the writer performed lines PW6 to PW8 after the reader performed line PR1 but before it reached line PR8), then the reader can safely return the value in COPYBUF[r] as the current register value.

If the reader did not detect this signal, then it can examine (line PR10) the values it read from WFLAG and SWITCH to determine whether it read a correct value from BUF1 or not. This test (line PR10) is now safe, since to fool it, the writer must have flipped SWITCH (line PW3) at least twice, since the reader was at line PR3, and in that case, the writer should have written to COPYBUF[r] and signaled the reader (at line PR8) as described above.

If the test at line PR10 indicates that the reader did not read BUF1 correctly, then the reader can safely return the value it read from BUF2 (line PR7) instead. The value read from BUF2 is safe because, for the writer to interfere with both the reading of BUF1 (line PR4) and the reading of BUF2 (line PR7), the writer must execute both its write of BUF1 (line PW1) and its write of BUF2 (line PW9) before the reader reached line PR8, and in that case, it should have written to COPYBUF[r] and signaled the reader (at line PR8) as described above.

Peterson's algorithm is designed for sequentially consistent shared memory. On shared memory multiprocessors that have a weaker memory consistency model, memory barriers need to be inserted at appropriate places in the algorithm. A memory barrier ensures that all preceding memory accesses by a processor have completed before any subsequent one takes effect. To ensure sufficient memory consistency, memory barriers are needed (1) before line PR1, (2) after each line from line PR1 until line PR8, (3) before line PW1, and (4) after each line from line PW1 until line PW9 except after line PW5. Note that memory barriers are not needed during the bulk reads or writes of a register value, only before and after such reads or writes.

Peterson's implementation is simple and efficient in most cases, however, a high-level write operation potentially has to write $n+2$ copies of the new value and all high-level reads read at least two copies of the value, which can be quite expensive when the register is large. Although it is unlikely that one can do better using only read/write subregisters, it may be possible to come up with more efficient solutions if we consider also using other atomic suboperations. Many are available on most modern systems, such as those discussed in the previous subsection. Our new register implementation uses such additional suboperations to implement high-level multiword read and write operations that only need to read or write one copy of the register value.

We have decided to comparatively study our method with Peterson's algorithm because (1) they are both designed for the 1-writer n-reader shared register problem, and (2) compared to other more general solutions based on weaker subregisters (which are much weaker than what common multiprocessor machines provide) Peterson's algorithm involves the least communication overhead among the processes, without requiring unbounded timestamps or methods to bound the unbounded version [Haldar and Vitányi 2002; Israeli and Li 1993].

Algorithm 2. A spin-lock algorithm with exponential back-off.

```
spin_lock(int *lock)
    backoff := 1;
    while (swap(lock, 1))
        backoff := 2 * backoff;
        spin for backoff iterations;

spin_unlock(int *lock)
    *lock := 0;
```

Algorithm 3. A readers-writers spin-lock algorithm with exponential back-off.

```
struct rwlock_t
    int lock
    int rlock
    int readers

writer_lock(rwlock_t *lock)
    spin_lock(&lock->lock);

writer_unlock(rwlock_t *lock)
    spin_unlock(&lock->lock);

reader_lock(rwlock_t *lock)
    spin_lock(&lock->rlock);
    lock->readers := lock->readers + 1;
    if (lock->readers == 1)
        spin_lock(&lock->lock);
    spin_unlock(&lock->rlock);

reader_unlock(rwlock_t *lock)
    spin_lock(&lock->rlock);
    lock->readers := lock->readers - 1
    if (lock->readers == 0)
        spin_unlock(&lock->lock);
    spin_unlock(&lock->rlock);
```

2.4 Mutual-Exclusion Based Solutions

For giving an even broader perspective in the experimental evaluation, we also study the performance of two mutual-exclusion-based register implementations, one that uses a single *spin-lock* with exponential back-off (see Algorithm 2) and another that uses a *readers-writers spin-lock* (see Algorithm 3 and Silberschatz et al. [2001]) with exponential back-off to protect the shared register. Both assume sequential consistency. On machines without sequential consistency, memory barriers need to be inserted last in *spin_lock* and first in *spin_unlock*. This ensures that all reads and writes inside a critical section will be performed after the lock is acquired and will be completed before the lock is released.

The readers-writers spin-lock is similar to the spin-lock but allows readers to access the register concurrently with other readers. The memory barriers in the spin-lock operations are sufficient for correctness on machines without sequential consistency.

How the locks are used to implement an atomic register algorithm can be seen in Algorithm 4. The prefix "r" (resp. "w") before "lock" and "unlock" is to be replaced with "spin" or "reader" (resp. "writer") to use the spinlock or the readers-writers lock, respectively.

3. THE READERSFIELD ALGORITHM

The idea of the READERSFIELD algorithm is to remove the need for reading or writing several buffers during a read and write operation, by utilizing the atomic

Constants	Description
PTRFIELDLEN	The number of bits used for the pointer field indexing the most recently update buffer.
PTRFIELD	A bitmask containing 1's in the PTRFIELDLEN least significant bits.

Variable	Type	Description
BUF$[n+2]$	Array of $n+2$ buffers	The buffers for the register value.
SYNC	Unsigned word	Consists of two fields.
bit 0 .. PTRFIELDLEN-1 of SYNC	Integer	Index of the buffer with the most recent register value. Initialized to $\perp$
bit PTRFIELDLEN $+ r$ of SYNC	Bit	The reading bit for reader r.

Fig. 3. The constants and shared variables used by the READERSFIELD algorithm. The number of readers is n. Initially BUF[0] holds the register value and SYNC points to this buffer while all reader-bits are 0.

Algorithm 4. A multiword register algorithm based on spin-locks or the readers-writers spin-lock.

Read operation by reader r:		**Write operation:**	
LR1	r_lock(lock);	LW1	w_lock(lock);
LR2	**read** BUF;	LW2	**write** BUF;
LR3	r_unlock(lock);	LW3	w_unlock(lock);
"r" is exchanged for "spin" or "reader"		"w" is exchanged for "spin" or "writer"	

synchronization primitives available on modern multiprocessor systems. These primitives are used for the communication between the readers and the writer, to help the former find the most recently written values and the latter find the "clean" space to write the new value.

The READERSFIELD algorithm uses $n + 2$ shared buffers that each can hold a value of the register. The number of buffers is the same as for Peterson's algorithm, which matches the lower bound on the required number of buffers. An informal argument is that the number of buffers cannot be less for any wait-free implementation because each of the n readers may be reading from one buffer concurrently with a write, and the write should not overwrite the last written value (since one of the readers might start to read again before the new value is completely written).

The shared variables used by the algorithm are presented in Figure 3. The shared buffers are in the $(n+2)$-element array BUF. The atomic variable SYNC is used to synchronize the readers and the writer. This variable consists of two fields: (1) the *pointer field*, which contains the index of the buffer in BUF that contains the most recent value written to the register, and (2) the *reading-bit field*, which holds a *handshake* bit for each reader. Each reading-bit is set when the corresponding reader has read the value presently contained in the pointer field.

A reader (Algorithm 5) uses *fetch_and_or* to atomically read the value of SYNC and set its reading-bit. Then it reads the value from the buffer pointed to by the pointer field.

Algorithm 5. The read and write operations of the READERSFIELD algorithm. The trace-array and oldwptr are static, i.e. stay intact between write operations. They are both initialized to zero.

```
Read operation by reader r:
R1    readerbit := 1 << (r + PTRFIELDLEN);
R2    rsync := fetch_and_or(&SYNC, readerbit);
R3    rptr := rsync & PTRFIELD;
R4    read BUF[rptr]

Write operation:
W1    choose newwptr such that newwptr != oldwptr and
          newwptr != trace[r] for all r; /* oldwptr initialized to ⊥*/
W2    write BUF[newwptr];
W3    wsync := swap(&SYNC, 0 | newwptr); /* Clears all reading bits */
W4    oldwptr := newwptr;
      usedwptr := wsync & PTRFIELD;
W5    for each reader r
W6        if (wsync & (1 << (r + PTRFIELDLEN)))
W7            trace[r] := usedwptr;
```

The writer (Algorithm 5) needs to keep track of the buffers that are available for use. To do this, it stores the index of the buffer where it last saw each reader, in a n-element array trace, in persistent local memory. At the beginning of each write, the writer selects a buffer index to write to. This buffer should be (1) different from the last one it used and (2) with no reader possibly using it. The writer writes the new value to that buffer and then uses the suboperation *swap* to atomically read SYNC and update it with the new buffer index and cleared reading-bits. The old value read from SYNC is then used to update the trace array for those readers whose reading-bit was set.

The maximum number of readers is given by the size of the words that the two atomic primitives used can handle. If we use 64-bit words, we can support 58 readers, as 6 bits are needed for the pointer field to be able to distinguish between 58 + 2 buffers. The number of readers can be extended beyond the limit of the wordsize by using a multiword *fetch_and_or* and multiword *swap*. That could either be available in hardware or be implemented in software and need only work for consecutive words in memory.

4. ANALYSIS

4.1 Correctness Proof

In this section, we show that the READERSFIELD algorithm satisfies the atomicity criteria given earlier in the article, by assuming only the cache weak consistency model. The fact that an algorithm that assumes a certain memory consistency model works correctly on stronger models, implies that the READERSFIELD algorithm works correctly on systems like the SGI Origin and SPARC architectures which support the sequential consistency and the weak consistency

models, respectively [Silicon Graphics 1993; Culler et al. 1998; CORPORATE SPARC International 1994].

First of all, we prove a couple of important properties of the READERSFIELD algorithm. Note that the SYNCH variable in the READERSFIELD algorithm is a synchronizing variable. It is only accessed through two synchronization instructions: *fetch_and_or* and *swap*.

LEMMA 4.1. *Let w be a write-operation that wrote the buffer index ptr read by a read-operation r to SYNC. There exists no write-operation w' issued after w (by the unique writer) that can write to the buffer $BUF[ptr]$ before r finishes reading the buffer.*

PROOF. We prove the lemma by contradiction. Assume there exists such a write-operation w'.

If there exist write-operations w'' so that r's R2 $\rightarrow$ w'''s W3 $\rightarrow$ w'''s W1, let w^* be the earliest of the set of w''. Since (1) r atomically sets its reading bit in *SYNC* while reading *SYNC* (line R2), and (2) its reading bit in *SYNC* is unchanged until the next write-operation w^* (line W3), the buffer index *ptr* that r is using will be recorded in the writer's *local* variable $trace[r]$ by w^* (lines W3-W7). Since w^*'s W3 $\rightarrow$ w'''s W1 and lines W4-W7 operate on the unique writer's *local* variables, w' will choose a buffer-to-write different from $BUF[trace[r]]$ (or $BUF[ptr]$) until r finishes reading $BUF[trace[r]]$ and the corresponding reader issues another read-operation. Note that both w^* and w' are issued by the same process—the unique writer, and that two read/write suboperations that belong to the same process and have *local* data or control dependence between them will be executed in program order [Adve and Gharachorloo 1996].

Otherwise, the writer's *local* variable *oldwptr* will be unchanged, since it is set to the buffer index *ptr* by w (line W4), which is also the the buffer index that r is using. Note that w's W3 has been completed due to the hypothesis. Since w' is issued after w by the same process (the unique writer), w' will choose a buffer-to-write different from $BUF[oldwptr]$ (or $BUF[ptr]$) (line W1).

That means w' never uses the buffer $BUF[ptr]$ before r finishes reading this buffer, a contradiction. □

The following lemma proves that the read-operation only returns the value that either is the initial value $BUF[\perp]$ or has been completely written by a write operation.

LEMMA 4.2. *For each complete read r, there exists a corresponding write $\pi(r)$.*

PROOF. We prove the lemma by contradiction. Assume there is a read r returning a value v that neither is the initial value $BUF[\perp]$ nor has completely written by a write operation. Since $v \neq BUF[\perp]$, r read a buffer index $ptr \neq \perp$ from the global synchronizing variable *SYNC* (line R2 in Figure 5) that had been written by a write operation w (line W3). Since accesses to *SYNC* is sequentially consistent due to the first requirement of the cache weak consistency model (CWC), w's *swap* operation (line W3) precedes r's *fetch_and_or* (line R2) (i.e. w's W3 $\rightarrow$ r's R2). Since (1) w's *write* suboperation to the global variable $BUF[ptr]$ (line W2) had been performed before w's *swap* (line W3) was issued due to the

CWC second requirement (i.e., w's W2 $\rightarrow$ w's W3), and (2) r's $fetch\_and\_or$ (line R2) had been performed before r's $read$ suboperation to $BUF[ptr]$ (line R4) was issued due to the CWC third requirement (i.e., r's R2 $\rightarrow$ r's R4), w's W2 $\rightarrow$ r's R4 or, in the other words, w's $write$ to $BUF[ptr]$ precedes r's $read$ from $BUF[ptr]$.

Therefore, r returns the value v that has not completely written by a write operation only if there exists a write operation w' that was issued after w (due to only one writer) and was using the buffer $BUF[ptr]$ that r was reading. From Lemma 4.1, there exists no such write operation w', a contradiction. □

In the following lemmas, we use Lemma 4.2 implicitly.

LEMMA 4.3. *For each read operation r it holds that $\pi(r)$'s W3 $\rightarrow$ r's R2.*

PROOF. We prove the lemma by contradiction. Assume that r's R2 $\rightarrow$ $\pi(r)$'s W3. Note that r's R2 and $\pi(r)$'s W3 are sequentially consistent due the CWC first requirement. Let w^* be the write operation that writes the buffer index ptr read by r to $SYNC$. We have w^*'s W3 $\rightarrow$ r's R2 (due to the CWC first requirement) and $w^* \neq \pi(r)$. From the CWC second requirement, w^* has completely written to $BUF[ptr]$ (line W2) before writing ptr to $SYNC$ (line W3). From the CWC third requirement, only after r has successfully read ptr from $SYNC$, it starts to read $BUF[ptr]$. Therefore, r can read the value written by $\pi(r)$ only if $\pi(r)$ has overwritten $BUF[ptr]$ before r reads $BUF[ptr]$.

However, since w^*'s W3 $\rightarrow$ (r's R2 $\rightarrow$) $\pi(r)$'s W3 and there is only one writer, $\pi(r)$ is issued after w^*. From Lemma 4.1, $\pi(r)$ cannot write to $BUF[ptr]$ before r finishes reading the buffer, a contradiction. □

LEMMA 4.4. *For each read operation r it holds that here exists no write w such that $\pi(r)$'s W3 $\rightarrow$ w's W3 $\rightarrow$ r's R2.*

PROOF. We prove the lemma by contradiction. Assume there exists such a write w: $\pi(r)$'s W3 $\rightarrow$ w's W3 $\rightarrow$ r's R2. Following the CWC first requirement, the buffer index written to $SYNC$ by $\pi(r)$ has been overwritten by w before r reads $SYNC$. Let w^* be the write operation writing to $SYNC$ the buffer index ptr that r reads (w^* may be w). Since accesses to $SYNC$ are sequentially consistent, $\pi(r)$'s W3 $\rightarrow$ w^*'s W3, i.e. $w^* \neq \pi(r)$.

—If $\pi(r)$ chose a buffer index ptr' different from ptr, $\pi(r)$ had completely written to $BUF[ptr'] \neq BUF[ptr]$ (line W2) before writing ptr' to $SYNC$ due to the CWC second requirement. On the other hand, only after successfully reading the buffer index ptr from $SYNC$ (line R2), r starts to read the buffer $BUF[ptr]$ (line R4) due to the CWC third requirement. That means r reads a buffer different from the buffer to which $\pi(r)$ wrote, or r reads a value that was not written by $\pi(r)$.

—If $\pi(r)$ chose the same buffer-index ptr as w^* does, the value $\pi(r)$ wrote to $BUF[ptr]$ (line W2) is overwritten by w^*. Indeed, since there is only one writer, $\pi(r)$'s W3 was issued (but may not be completed) before w^*'s W2 is issued. Following the CWC second requirement, $\pi(r)$ had completely written to $BUF[ptr]$ before $\pi(r)$'s W3 was issued. Following the CWC third

requirement, w^* started to write to $BUF[ptr]$ after $\pi(r)$'s W3 had completed. That means w^* started to write to $BUF[ptr]$ after $\pi(r)$ had completely written to $BUF[ptr]$. Since w^* wrote the buffer index ptr to $SYNC$ (line W3) (due to the definition of w^*), w^* completely overwrote the value that $\pi(r)$ had written to $BUF[ptr]$ (line W2) due to the CWC second requirement. On the order hand, only after successfully reading the buffer index ptr from $SYNC$ (line R2), r starts to read the buffer $BUF[ptr]$ (line R4) due to the CWC third requirement. That means r cannot read the value written by $\pi(r)$.

In all the cases, r cannot read the value written by $\pi(r)$, a contradiction to the definition of $\pi(r)$. □

We now prove that the READERSFIELD algorithm satisfies the conditions in Lamport's criterion [Lamport 1986] (see Criterion 2.1 in Section 2) on the cache weak consistency model, which guarantee atomicity.

LEMMA 4.5. *The* READERSFIELD *algorithm satisfies condition* "No-irrelevant".

PROOF. From Lemma 4.3, we have $\pi(r)$'s W3 $\rightarrow$ r's R2. Since the starting time-point of $\pi(r)$ is before $\pi(r)$'s W3 and r's R2 is before the ending time-point of r, the starting time-point of $\pi(r)$ is before the ending time-point of r, or $r \not\rightarrow \pi(r)$. □

LEMMA 4.6. *The* READERSFIELD *algorithm satisfies condition* "No-past".

PROOF. We prove the lemma by contradiction. Assume there are a read r and a write w such that $\pi(r) \rightarrow w \rightarrow r$. Since $\pi(r) \rightarrow w \rightarrow r$, $\pi(r)$'s W3 $\rightarrow$ w's W3 $\rightarrow$ r's R2. From Lemma 4.4, such a write w does not exist, a contradiction. □

LEMMA 4.7. *The* READERSFIELD *algorithm satisfies condition* "No N-O inversion".

PROOF. We prove the lemma by contradiction. Assume there are reads r_1 and r_2 such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$. We have r_1's R2 $\rightarrow$ r_2's R2 and $\pi(r_2)$'s W3 $\rightarrow$ $\pi(r_1)$'s W3. From Lemma 4.3, we have $\pi(r_1)$'s W3 $\rightarrow$ r_1's R2. Therefore, $\pi(r_2)$'s W3 $\rightarrow$ $\pi(r_1)$'s W3 $\rightarrow$ (r_1's R2 $\rightarrow$) r_2's R2. From Lemma 4.4, there exists no write w such that $\pi(r_2)$'s W3 $\rightarrow$ w's W3 $\rightarrow$ r_2's R2, a contradiction. □

With the correctness of the proposed wait-free algorithm established, let us focus on its complexity, also in comparison with Peterson's wait-free algorithm.

4.2 Complexity

The complexity of a read operation is of order $O(m)$, where m is the size of the register, for both the READERSFIELD algorithm and Peterson's algorithm [Peterson 1983]. However, in Peterson's algorithm the reader may have to read the value up to 3 times, while in the READERSFIELD algorithm the reader will only read the value once but has to use the *fetch_and_or* suboperation once. A write operation in the READERSFIELD algorithm writes one value of size m and then traces the n readers. The complexity of the write operation is therefore of order $O(n+m)$. For Peterson's algorithm, however, the writer must, in the worst case, write to $n + 2$ buffers of size m, thus its complexity is of order $O(n \cdot m)$.

As the size of registers and the number of threads increase, the READERSFIELD algorithm is expected to perform significantly better than Peterson's algorithm with respect to the writer. With respect to the readers, the handshake mechanism used in the READERSFIELD algorithm can be more expensive compared to the one used by Peterson's, but on the other hand, the READERSFIELD algorithm only needs to read one m-word buffer, whereas Peterson's need to read at least two and sometimes three buffers.

Using the above, we have the following theorems:

THEOREM 4.8. *The* READERSFIELD *algorithm constructs a multireader, single-writer, m-word sized register using* $n + 2$ *buffers of size m words each.*

THEOREM 4.9. *The complexity of a read operation for the* READERSFIELD *algorithm is* $O(m)$ *memory operations. The complexity of a write operation is* $O(n+m)$ *memory operations.*

5. PERFORMANCE STUDY

In our experimental study the performance of the proposed algorithm was evaluated in comparison with: (1) Peterson's algorithm [Peterson 1983], (2) a spinlock-based implementation with exponential backoff and (3) a readers-writers spinlock with an exponential backoff (see Section 2 for descriptions of the respective algorithms).

5.1 Method

We measured the number of successful read and write operations during a fixed period of time: the higher this number (throughput) the better the performance. In each test one thread is the writer and the rest of the threads are the readers. Two sets of experiments have been done: (1) one set with *low contention* and (2) one set with *high contention*. During the high-contention experiments, each thread reads or writes continuously with no delay between successive accesses to the multi-word register. During the low-contention experiments, each thread waits for a time-interval between successive accesses to the multiword register. This time interval is much longer than the time used by one write or read. Experiments have been performed for different number of threads and for different sizes of the register. The runs for the different register types were interleaved for each setting of number of threads and size of the register. The caches of the processors are flushed before each run. On one of the platforms (in particular the NUMA Origin 3800), the experiment was one among many concurrent processes. This could raise a concern that competing workloads might affect the experiment (e.g., by causing high loads on the interconnect or forcing our benchmark to use widely distributed processors and memory banks in the machine). However, in the data from our experiments, there are very few indications of such interference and, moreover, as the runs for the different registers are interleaved, such interference would be able to systematically favor or disfavor one type of register *only if* its intensity and effect could closely follow the phases of the experiment (i.e., only if the systems' load was deliberately designed to play the role of an adversary for the experiment, which can hardly be the case).

5.2 Systems

The performance of the READERSFIELD algorithm has been measured on both UMA (Uniform Memory Architecture) and NUMA (Non Uniform Memory Architecture) multiprocessor systems [Tanenbaum 2001]. The difference between UMA and NUMA is how the memory is organized. In a UMA system, all processors have the same latency and bandwidth to the memory. In a NUMA system, processors are placed in nodes and each node has some of the memory directly attached to it. The processors of one node have fast access to the memory attached to that node, but accesses to memory on another node are made over the interconnect network and are therefore significantly slower.

The three different systems we used are:

—An UMA Sun SunFire 880 with 6 900MHz UltraSPARC III+ (8MB L2 cache) processors running Solaris 9.
—A ccNUMA SGI Origin 2000 with 29 250MHz MIPS R10000 (4MB L2 cache) processors running IRIX 6.5.
—A ccNUMA SGI Origin 3800 with 128 500MHz MIPS R14000 (8MB L2 cache) processors running IRIX 6.5.

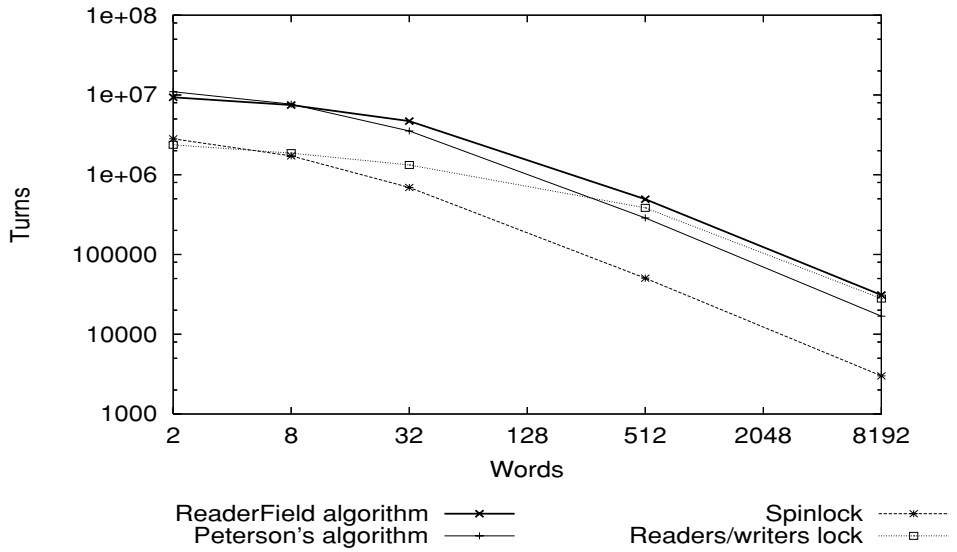
The systems were used nonexclusively, but for the SGI systems, the batch-system guarantees that the required number of CPUs was available. The *swap* and *fetch_and_or* suboperations were implemented by the `swap` hardware instruction [Weaver and Germond 2000] and a lock-free subroutine using the `compare_and_swap` hardware instruction on the SunFire machine. As the SunFire does not ensure sequential consistency, memory barriers were inserted in Peterson's algorithm and the lock-based algorithms as described in Section 2. Regarding the SGI Origin machines, *swap* and *fetch_and_or* were implemented by the system-provided [Cortesi 2004] synchronization primitives `__lock_test_and_set` and `__fetch_and_or`, respectively. The SGI Origin platforms ensure sequential consistency [Culler et al. 1998], so no memory barriers were needed there.

5.3 Results

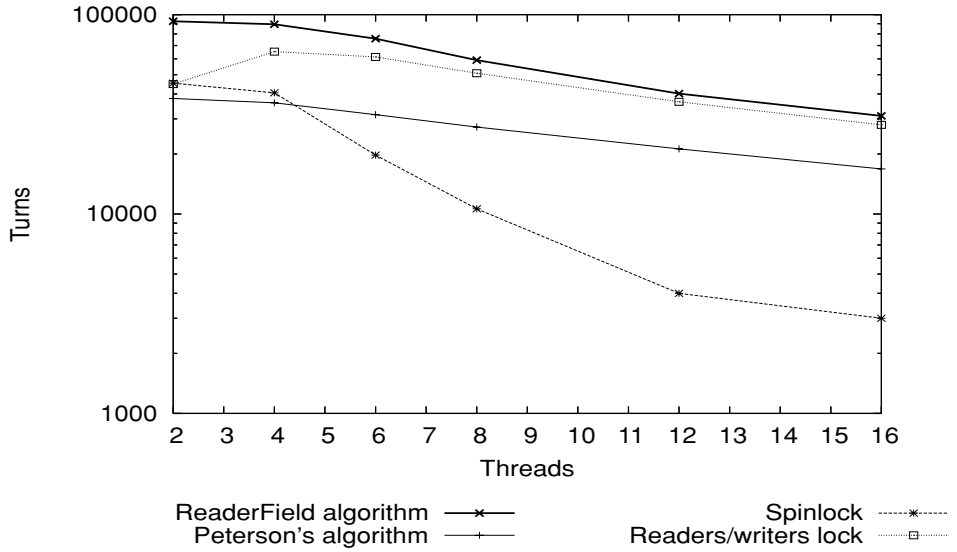
Following the analysis and the diagrams presenting the experiments' outcome, it is clear that the performance of the lock-based solutions is not even near the figures of the wait-free algorithms unless the number of threads is minimal (two) *and* the size of the register is small. Moreover, as expected following the analysis, the READERSFIELD algorithm performs at least as well and, in the large-size register cases, better than Peterson's wait-free solution.

In the following paragraphs, we provide a more detailed discussion of the experimental results illustrated in the figures. Note that the *high contention* results are presented on an *exponential scale*, whereas the *low contention results* use a *linear scale* in the figures.

More specifically, on the UMA SunFire the READERSFIELD algorithm outperforms the others for large registers under both low and high contention (see Figures 4 and 5). The worst performer under high contention is the spinlock,



(a) 16 threads, high contention.

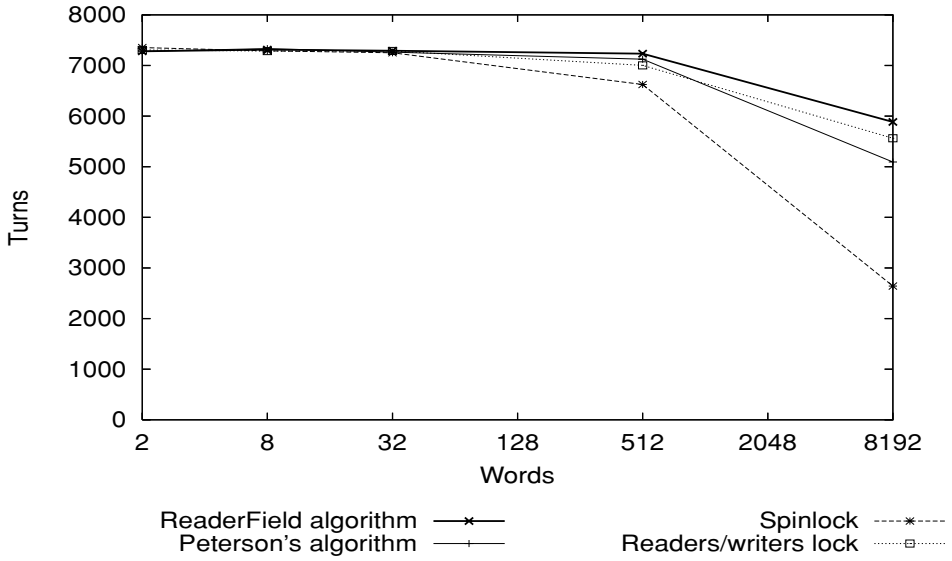


(b) 8192 word register, high contention.

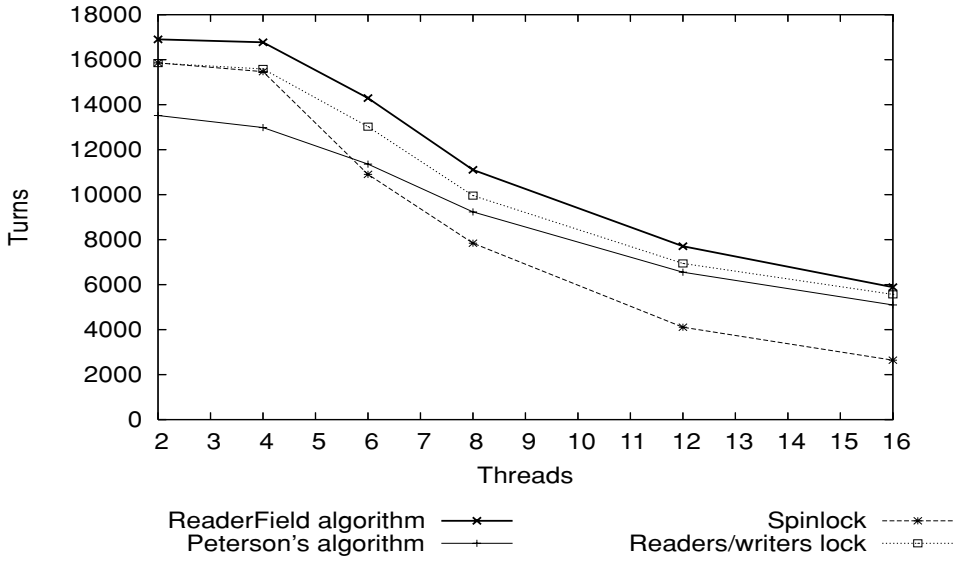
Fig. 4. Average number of reads or writes per thread on the UMA SunFire 880 at high contention.

which is particularly vulnerable to threads being preempted inside the critical section, something that is increasingly likely as the number of threads exceed the number of processors. Under low contention the differences are, as expected, much less pronounced.

On the NUMA Origin 2000 platform (Figures 6(a)–8(b)) and on the NUMA Origin 3800 platform (Figures 9(a)–10(b)), we observe the effect of the particular architecture, namely that the possibility to cause high contention on a



(a) 16 threads, low contention.

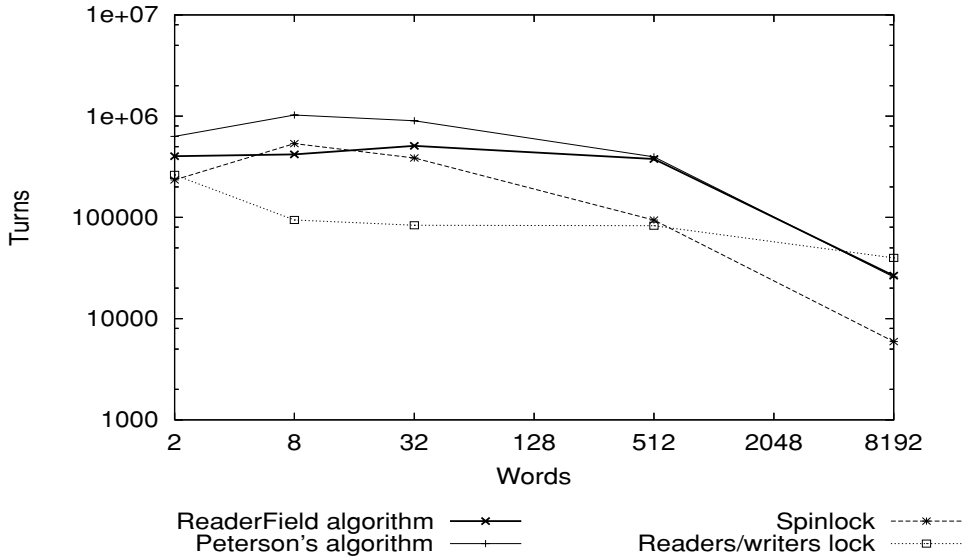


(b) 8192 word register, low contention.

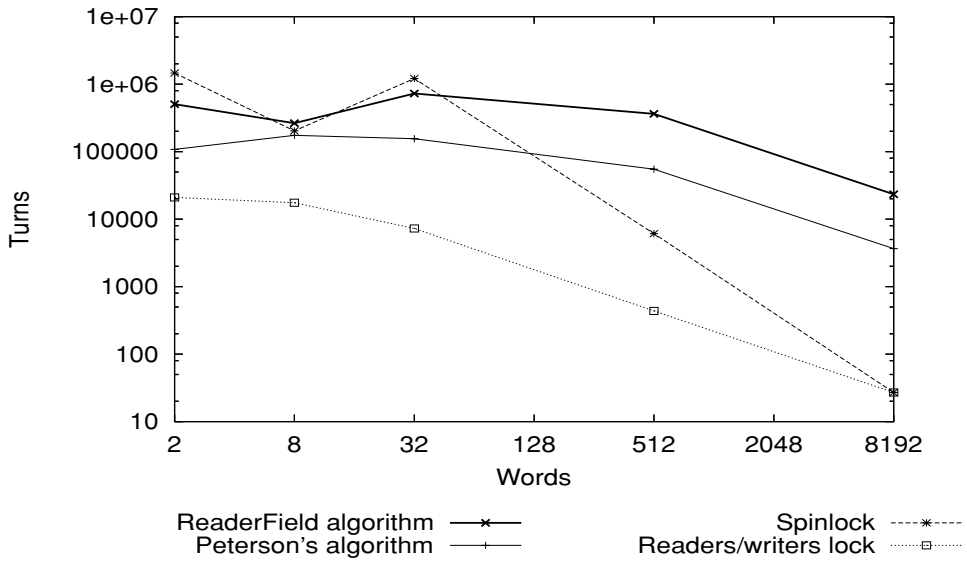
Fig. 5. Average number of reads or writes per thread on the UMA SunFire 880 at low contention.

synchronization variable significantly affects the performance of the solutions. By observing the performance diagrams for this case, we still see the writer in the READERSFIELD algorithm performs significantly better than the writer in Peterson's algorithm in both the low- and high-contention scenarios.

Recall that the writer, following Peterson's algorithm, may have to write to more buffers as the number of readers grow. The writer of



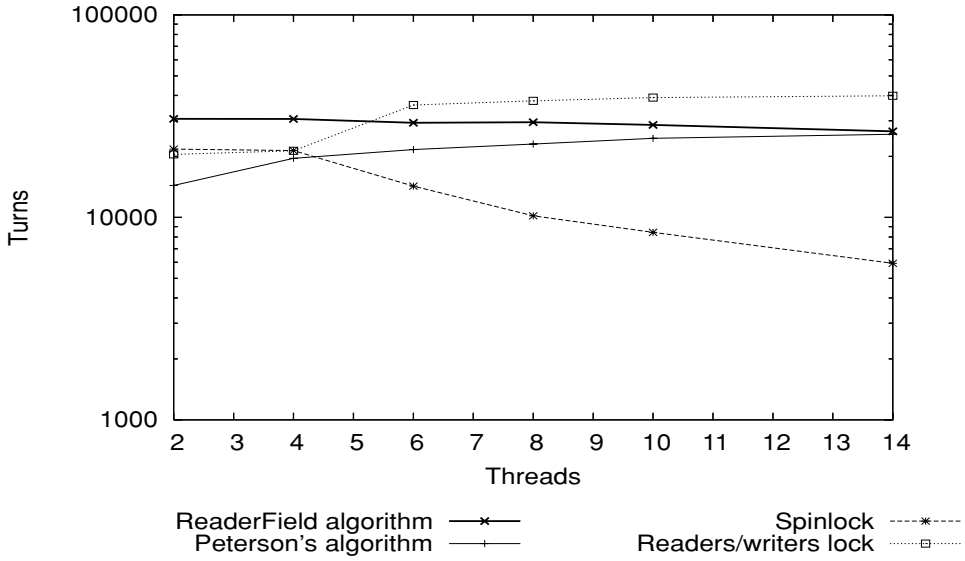
(a) All operations (readers and writer).



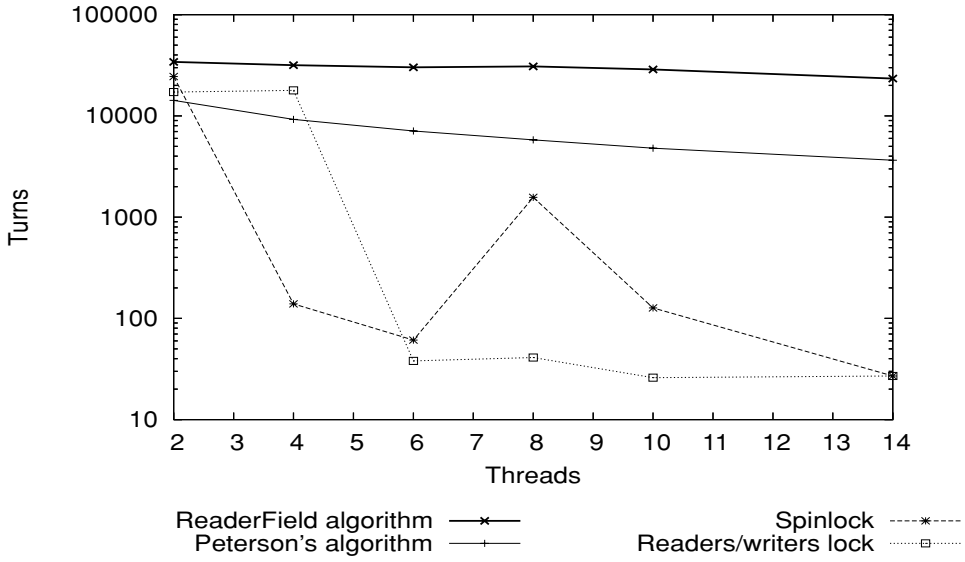
(b) Operations by the writer.

Fig. 6. Average number of operations per thread with 14 threads and high contention on NUMA Origin 2000.

the READERSFIELD algorithm has no such problems. This phenomenon, though, has a seemingly positive side-effect in Peterson's algorithm; namely, as the writer becomes slower, the chances that the readers have to read their individual buffers (apart from the two buffers used by all readers), become smaller. On the other hand the readers will read old data they might already have



(a) All operations (readers and writer).

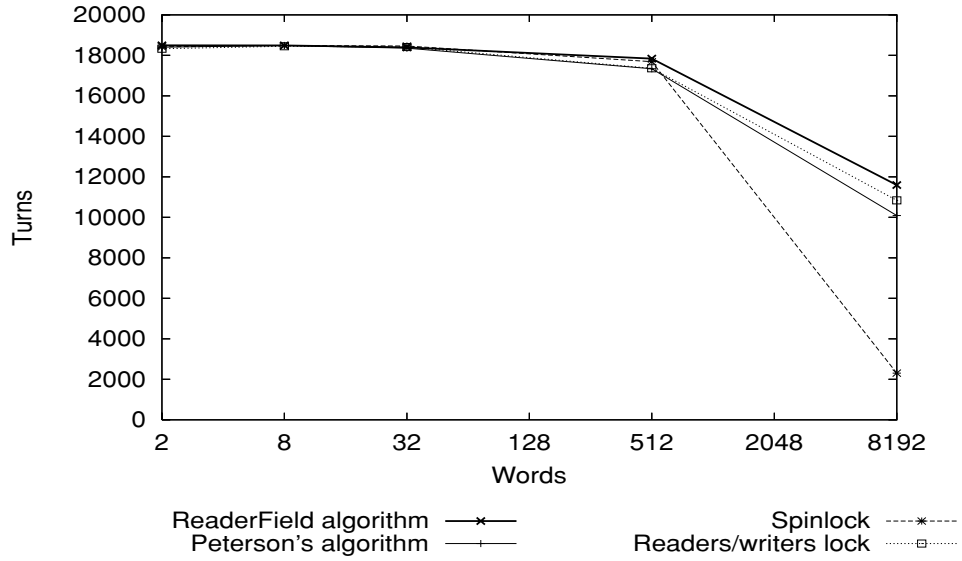


(b) Operations by the writer.

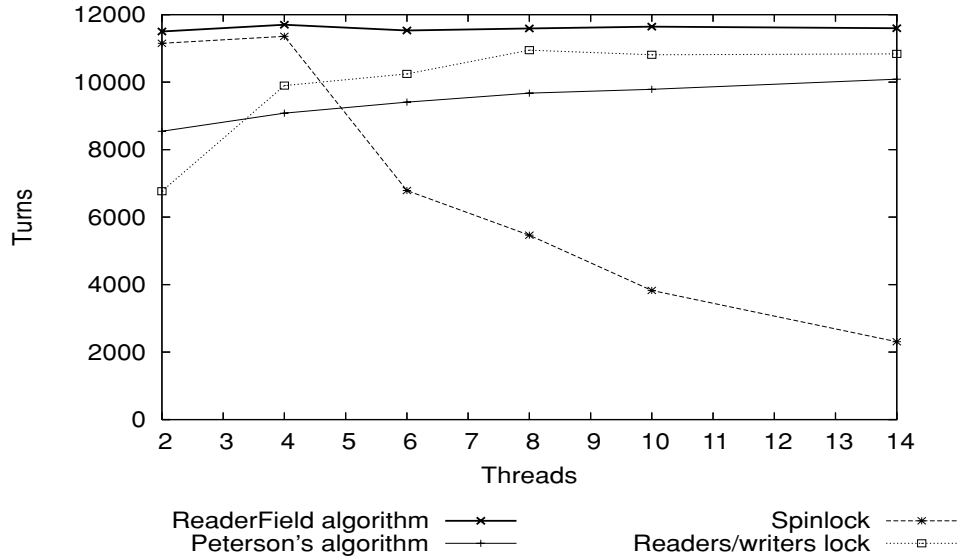
Fig. 7. Average number of operations per thread with a register size of 8192 words and high contention on NUMA Origin 2000.

read. Hence, the difference in the readers' performance for the two wait-free algorithms under high contention becomes smaller (Figures 6(a) and 7(a)).

The difference in behaviour between readers and writers under high contention and a varying number of threads can be studied in detail in Figure 7(a) and Figure 7(b). Some observations based on the results are (1) the



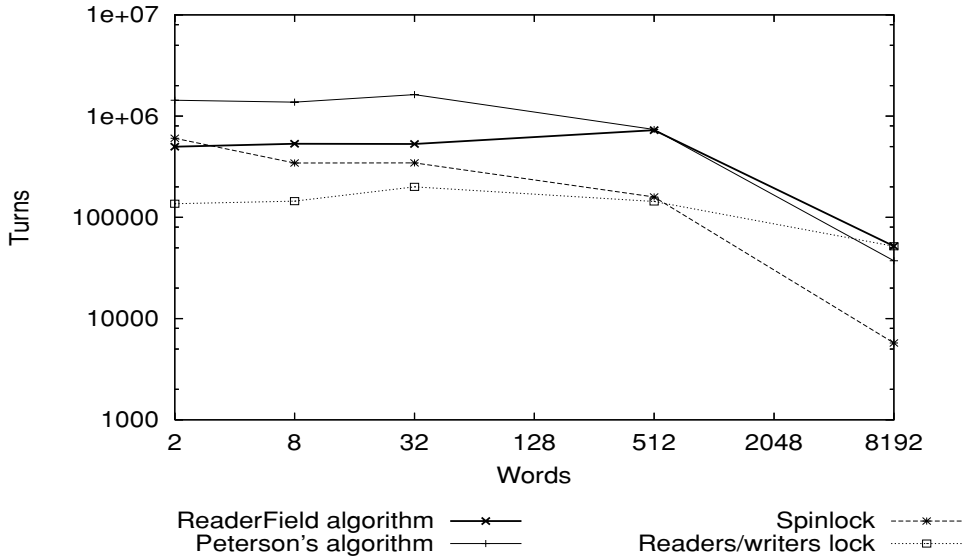
(a) Run with 14 threads



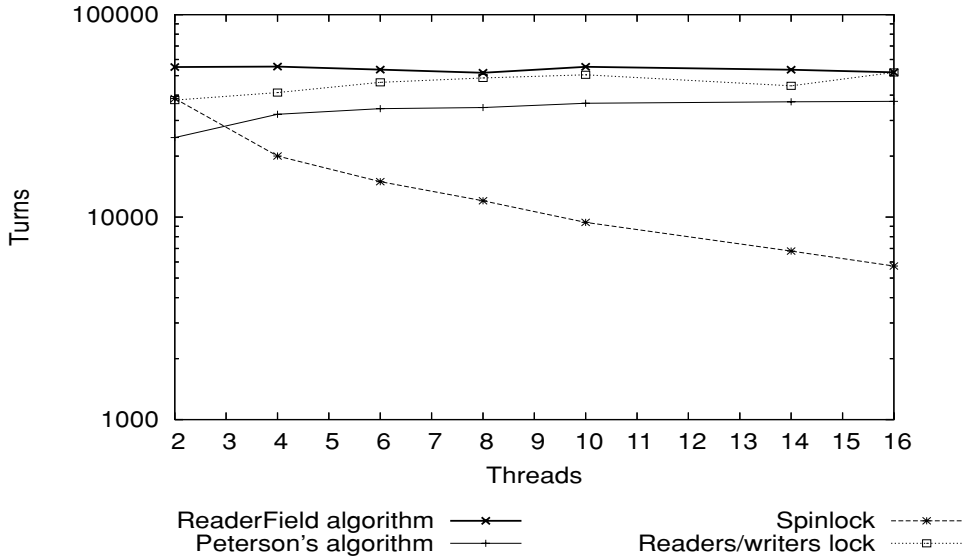
(b) Run with 8192 word register

Fig. 8. Average number of operations per thread with low contention on NUMA Origin 2000.

readers/writers lock does well in terms of read operations while the writer suffers significantly since a write can be locked out by preceding and concurrent overlapping reads, and (2) the READERSFIELD algorithm's advantage over Peterson's for the readers decreases when the number of threads increases, while it increases for the writer. In the lock-based algorithms, threads can easily be starved. Regarding the readers/writers lock, the writer will never get



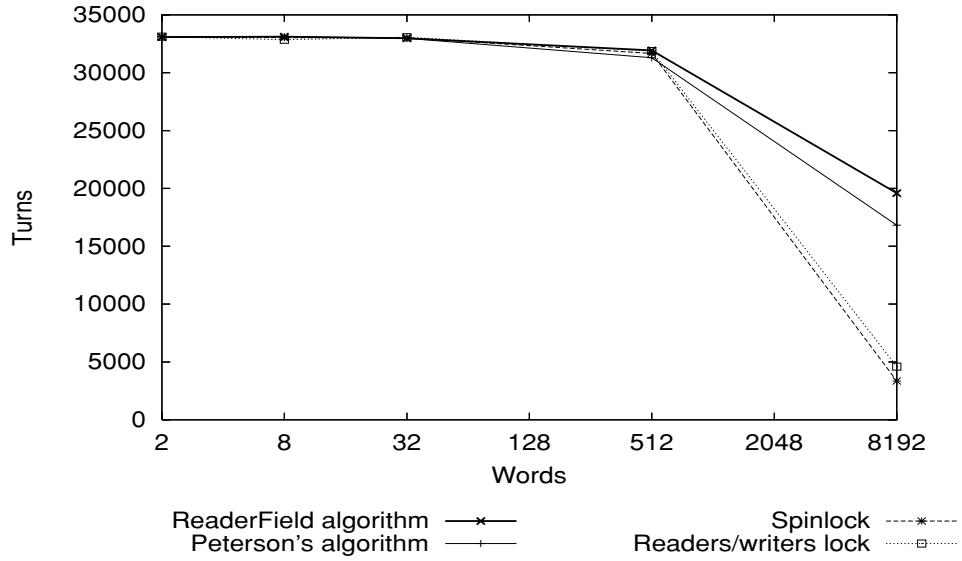
(a) 16 threads, high contention.



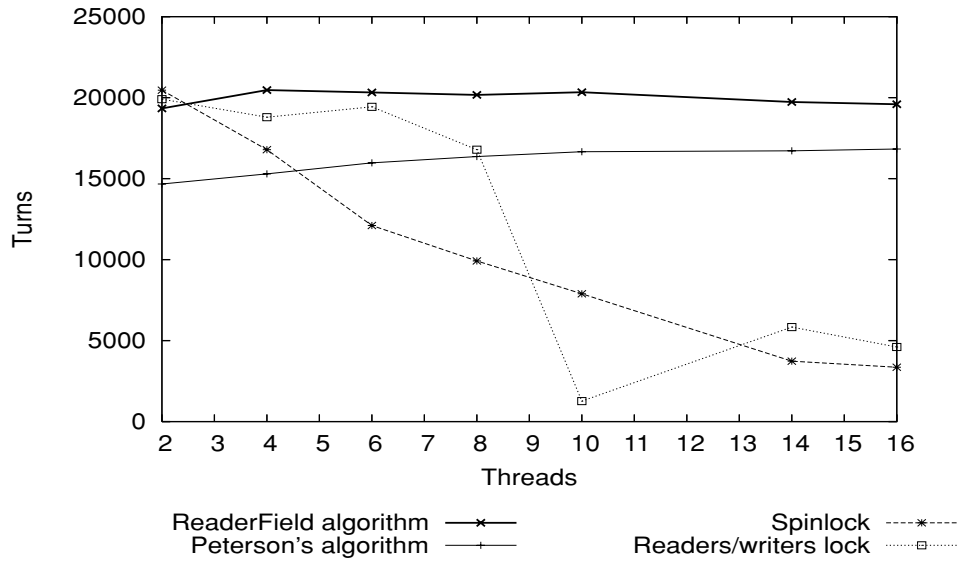
(b) 8192 word register, high contention.

Fig. 9. Average number of reads or writes per thread on NUMA Origin 3800 at high contention.

access to the buffer while at least one reader is reading. So in the worst case, if readers take overlapping turns reading the buffer, the writer can be shut out indefinitely. For the spinlock, there is no such difference between readers and the writer, so any thread can get an unfair share of accesses to the register. On NUMA architectures, some threads might have the lock in their local memory. They will have much higher chances to get the lock again, compared to other



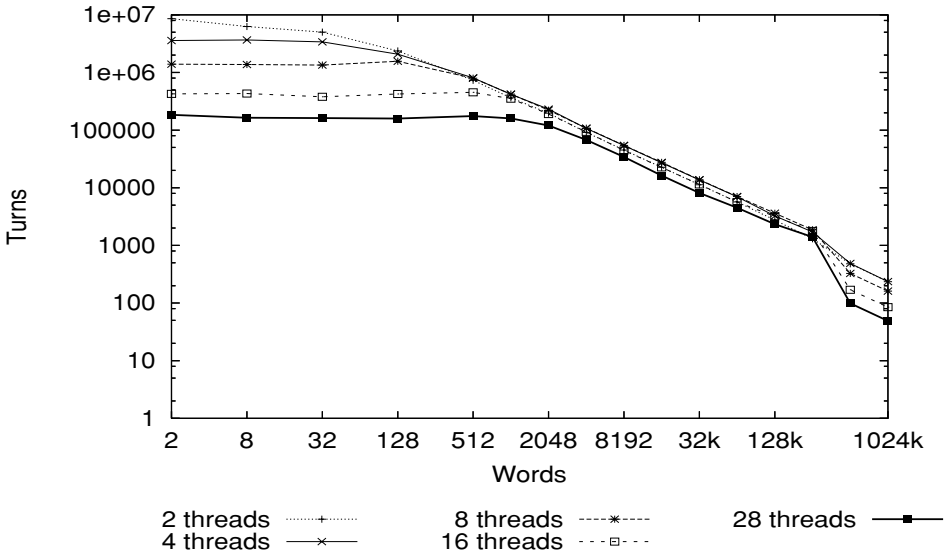
(a) 16 threads, low contention.



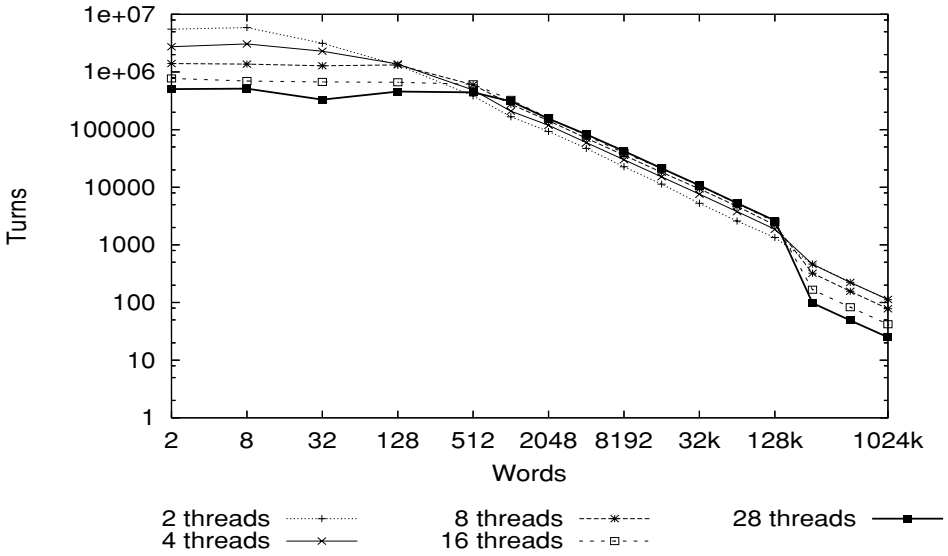
(b) 8192 word register, low contention.

Fig. 10. Average number of reads or writes per thread on NUMA Origin 3800 at low contention.

threads that have to go over the interconnect network to access it. The effect is most visible for a single thread, which is the reason we see irregular behaviour for the lock based registers in Figure 7(b). When looking at a single thread like this, the luck of the writer plays a large role in the result. As we can see in Figure 7(a), the mean taken over all threads is much more consistent over different runs.



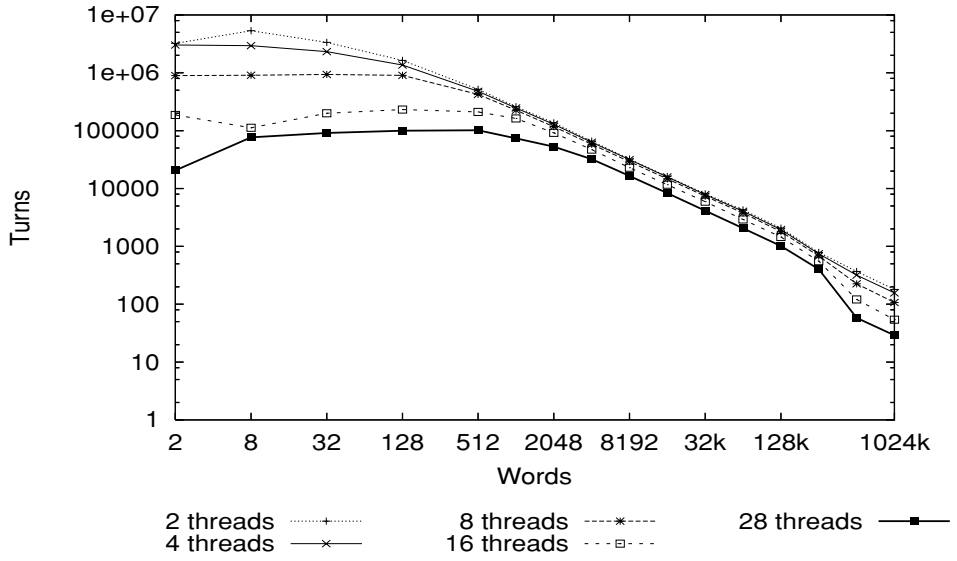
(a) The READERSFIELD algorithm, all operations



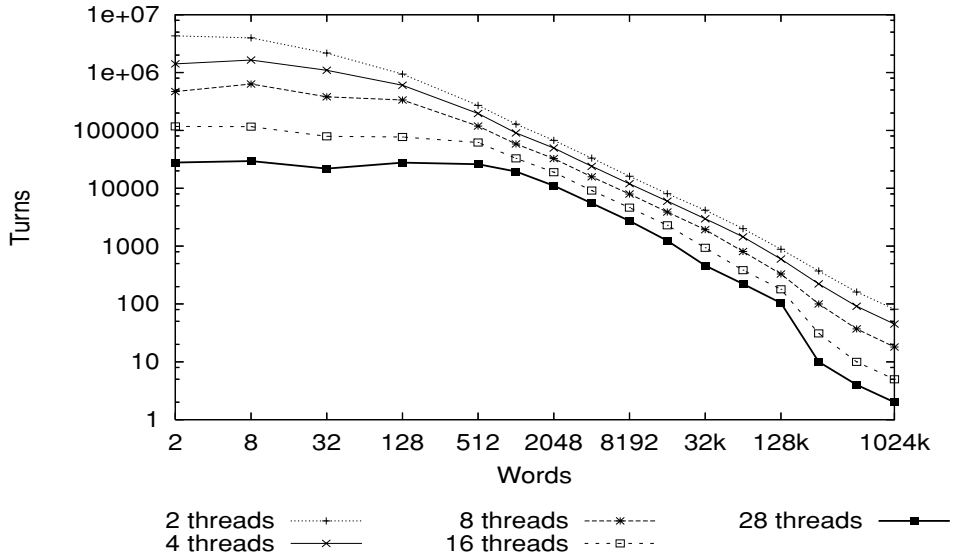
(b) Peterson's algorithm, all operations

Fig. 11. The average number of reads or writes per thread for the READERSFIELD algorithm and Peterson's algorithm compared with themselves for different number of threads under high contention. Run on NUMA Origin 2000.

Under low contention, the two NUMA Origin platforms (Figures 8 and 10) show similar behaviour. In the experiments with varying register size (Figures 8(a) and 10(a)) there is a break in the number of operations per thread at a certain register size which is attributed to the increased amount of interconnect traffic needed at the larger sizes. The two wait-free algorithms show



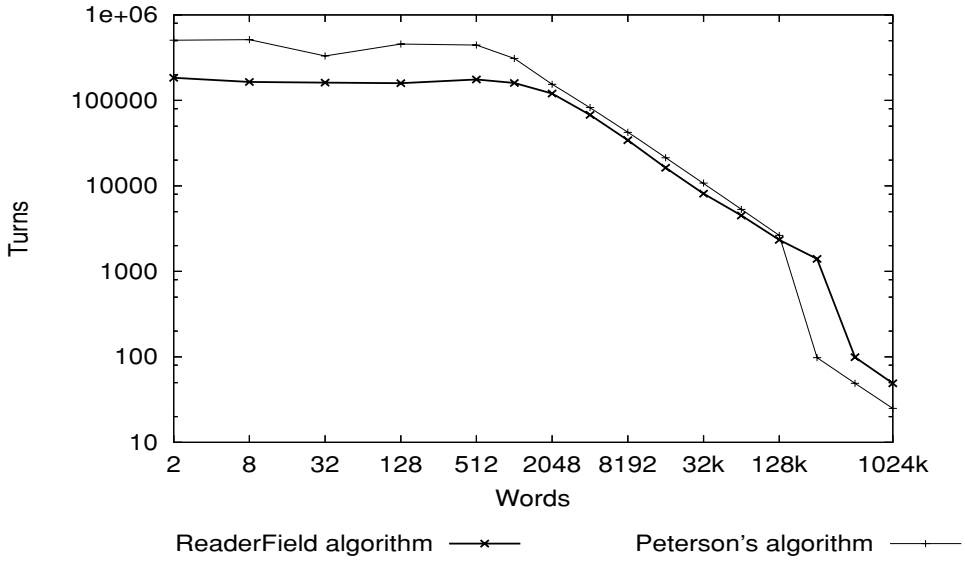
(a) The READERSFIELD algorithm, number of writes



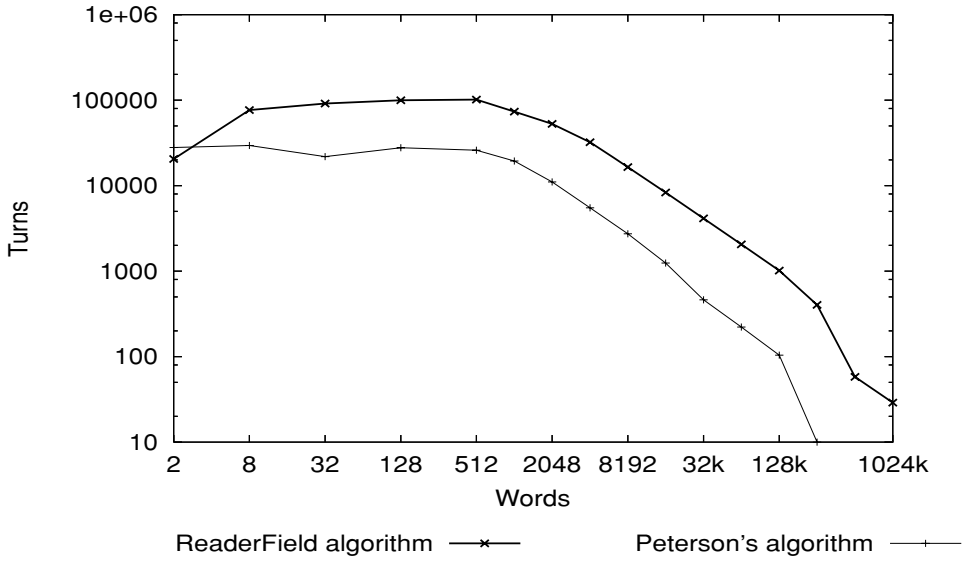
(b) Peterson's algorithm, number of writes

Fig. 12. The number of writes per thread for the READERSFIELD algorithm and Peterson's algorithm compared with themselves for different number of threads under high contention. Run on NUMA Origin 2000.

good scalability behaviour across different number of threads (Figures 8 (b) and 10 (b)) with the READERSFIELD algorithm having an advantage over Peterson's. The average number of operations per thread for the readers/writers lock is also good, although as discussed above the writer is likely to be locked out most of the time. The large drop at 10 threads for the readers/writers lock in



(a) All operations.



(b) Write operations.

Fig. 13. Average number of operations per thread at high contention and 28 threads on NUMA Origin 2000.

Figure 10(b) was seen in all our runs and can be attributed to the NUMA Origin 3800's nonuniform memory interconnect architecture, following similar argumentation as in the previous paragraph.

In Figures 11 and 12, we can see how the performance of the READERSFIELD algorithm and Peterson's algorithm changes both with the number of words in

the register and the number of threads. The READERSFIELD algorithm is more sensitive to the addition of threads than Peterson's algorithm on this platform, due to increased contention for the trace variables. For larger words, the difference becomes smaller and smaller and eventually the READERSFIELD algorithm outperforms Peterson's. On the writer's side, we see that Peterson's algorithm loses performance to a higher degree with increasing number of threads than the READERSFIELD algorithm algorithm does. The similar and characteristic shape of the performance curves for both algorithms are due to the properties of the particular multiprocessor hardware (e.g., cache-line and page size and the interconnect bandwidth). Figure 13 directly compares the performance of the READERSFIELD algorithm and Peterson's algorithm for 28 threads from the experiments on the NUMA Origin 2000 system above. Here we can see that although Peterson's algorithm maintains an advantage in the number of read operations until a large register size, the READERSFIELD algorithm has an increasing advantage in the number of write operations.

6. CONCLUSIONS AND DISCUSSION

This article presents the READERSFIELD algorithm, a simple and efficient algorithm for atomic registers (memory words) of multiple-word length. The simplicity and the good time complexity of the algorithm are achieved via the use of two common synchronization primitives. The correctness of the READERSFIELD algorithm is shown assuming only a weak consistency underlying memory access model. The paper also presents a performance evaluation of (i) the READERSFIELD algorithm; (ii) a previously known practical algorithm that is based only on read and write operations; and (iii) two mutual-exclusion-based registers. The evaluation is performed on three different well-known multiprocessor systems.

Further, it is worth noticing that the READERSFIELD algorithm can easily be converted to a regular register for an arbitrary number of readers, by using a set of SYNC variables. Instead of a single one-word swap, the writer performs a sequence of swaps on an array of words. Each reader has a bit in one of the words in the array. The criterion for a regular register holds with similar proofs as for the atomic register as the proofs for *No-Irrelevant* and *No-Past* only involves buffers and the SYNC variable for one reader. Only in the proof for *No-New-Old-inversion* are different readers involved.

Shared objects are commonly used in parallel/multithreaded applications. Results like this and further research along this line are, therefore, significant towards providing better support for efficient synchronization and communication.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their helpful and thorough comments on the earlier version of this article.

REFERENCES

ADVE, S. V. AND GHARACHORLOO, K. 1996. Shared memory consistency models: A tutorial. *IEEE Computer 29*, 12, 66–76.

BARNES, G. 1993. Method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, 261–270.

BLOOM, B. 1988. Constructing two-writer atomic registers. *IEEE Transactions on Computers 37*, 12 (Dec.), 1506–1514.

BURNS, J. E. AND PETERSON, G. L. 1987. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 222–231.

CHEN, J. AND BURNS, A. 1997. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Tech. Rep. YCS-288, Department of Computer Science, University of York. May.

CORPORATE SPARC INTERNATIONAL, I. 1994. *The SPARC Architecture Manual: Version 9*. Prentice-Hall, Inc.

CORTESI, D. 2004. *Topics in IRIXA Programming*. Silicon Graphics, Inc. (doc #:007-2478-009).

CULLER, D., SINGH, J. P., AND GUPTA, A. 1998. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann.

DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. 1986. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA'86)*. 434–442.

GOODMAN, J. R. 1989. Cache consistency and sequential consistency. Tech. Rep. 61, IEEE Scalable Coherent Interface (SCI) Working Group.

HALDAR, S. AND VIDYASANKAR, K. 1995. Constructing 1-writer multireader multivalued atomic variable from regular variables. *J. ACM 42*, 1 (Jan), 186–203.

HALDAR, S. AND VIDYASANKAR, K. 1996. Simple extensions of 1-writer atomic variable constructions to multiwriter ones. *Acta Informatica 33*, 2, 177–202.

HALDAR, S. AND VITÁNYI, P. 2002. Bounded concurrent timestamp systems using vector clocks. *J. ACM 49*, 1 (Jan), 101–126.

HERLIHY, M. 1991. Wait-free synchronization. *ACM Transaction on Programming and Systems 11*, 1 (Jan), 124–149.

HERLIHY, M. 1993. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems 15*, 5 (Nov), 745–770.

HERLIHY, M. P. AND WING, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July), 463–492.

ISRAELI, A. AND LI, M. 1993. Bounded time-stamps. *Distributed Computing 6*, 4, 205–209.

ISRAELI, A. AND SHAHAM, A. 1992. Optimal multi-writer multi-reader atomic register. In *Proceedings of the 11th Annual Symposium on Principles of Distributed Computing*. ACM Press, 71–82.

KIROUSIS, L. M., KRANAKIS, E., AND VITÁNYI, P. M. B. 1987. Atomic multireader register. In *Distributed Algorithms, 2nd International Workshop*. LNCS, vol. 312. Springer, 1988, 278–296.

KOPETZ, H. AND REISINGE, J. 1993. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society Press, 131–137.

LAMPORT, L. 1977. Concurrent reading and writing. *Communications of the ACM 20*, 11 (Nov), 806–811.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers 28*, 9, 690–691.

LAMPORT, L. 1986. On interprocess communication. *Distributed Computing 1*, 2, 77–101.

LARSSON, A., GIDENSTAM, A., HA, P. H., PAPATRIANTAFILOU, M., AND TSIGAS, P. 2004. Multi-word atomic read/write registers on multiprocessor systems. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA'04) LNCS 3221*. Springer-Verlag, 736–748.

LI, M., TROMP, J., AND VITÁNYI, P. M. B. 1996. How to share concurrent wait-free variables. *J. ACM 43*, 4 (July), 723–746.

LI, M. AND VITÁNYI, P. M. B. 1992. Optimality of wait-free atomic multiwriter variables. *Information Processing Letters 43*, 2 (Aug), 107–112.

NEWMAN-WOLFE, R. 1987. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 232–248.

PETERSON, G. L. 1983. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems 5*, 1 (Jan), 46–55.

PETERSON, G. L. AND BURNS, J. E. 1987. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE, 383–392.

RAJKUMAR, R. 1990. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS '90)*. IEEE Computer Society Press, 116–123.

SHA, L., RAJKUMAR, R., AND LOJOCZKY, J. 1990. Priority inheritence protocols: An approach to real-time syncronization. *IEEE Transactions on Computers 39*, 9 (Sept), 1175–1185.

SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. 2001. *Operating Systems Concepts*. Addison-Wesley.

SILICON GRAPHICS, I. 1993. *IRIX 6.5: Man Pages*. SGI Techpubs Library.

SIMPSON, H. R. 1990. Four-slot fully asynchronous communication mechanism. *IEE Proc., Computers and Digital Techniques 137*, 1 (Jan.), 17–30.

SINGH, A. K., ANDERSON, J. H., AND GOUDA, M. G. 1994. The elusive atomic register. *J. ACM 41*, 2 (Mar.), 311–339.

SUNDELL, H. 2004. Efficient and practical non-blocking data structures. Ph.D. thesis, Chalmers University of Technology.

SUNDELL, H. AND TSIGAS, P. 2002. NOBLE: A non-blocking inter-process communication library. In *Proceedings of the 6*th *Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*. LNCS. Springer Verlag.

TANENBAUM, A. S. 2001. *Modern Operating Systems*, 2nd ed. Prentice Hall.

TSIGAS, P. AND ZHANG, Y. 2001. Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In *Proceedings of the ACM SIGMETRICS 2001/Performance 2001*. ACM Press, 320–321.

TSIGAS, P. AND ZHANG, Y. 2002. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP'02)*. ACM Press, 55–67.

VITÁNYI, P. M. B. AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science*. IEEE, 233–243.

WEAVER, D. L. AND GERMOND, T., Eds. 2000. *The SPARC Architecture Manual*. Prentice Hall. Version 9.

Received November 2006; revised September 2007; accepted April 2008