Multi-word Atomic Read/Write Registers on Multiprocessor Systems *

Andreas Larsson, Anders Gidenstam, Phuong H Ha, Marina Papatriantafilou, and Philippas Tsigas

Department of Comp. Science, Chalmers University of Technology, SE-412 96 Göteborg, Sweden

Abstract. Modern multiprocessor systems offer advanced synchronization primitives, build in hardware, to support the development of efficient parallel algorithms. In this paper we develop a simple and efficient algorithm of 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 paper we also evaluate the performance of our algorithm and the performance of a practical previously know algorithm that is based only on read and write primitives. The evaluation is performed on 3 well-known, parallel architectures. This evaluation clearly shows that both algorithms are practical and that as the size of the register increases our algorithm performs better accordingly to its complexity behavior.

1 Introduction

In multiprocessing systems cooperating processes may share data via shared data objects. In this paper we are interested in designing and evaluating the performance of efficient 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*, aka *linearizability*. A shared object (its implementation) is *atomic* or *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect in an atomic time instant which 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 e.g. 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 most simple solution to maintaining the consistency of a shared data object enforces mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at time to access

* This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC).

the data. Mutual exclusion causes large performance degradation especially in multiprocessor systems [1] and suffers from potential priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [2].

Non-blocking implementation of shared data objects is an alternative approach for the problem of inter-task communication. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this.

Non-blocking protocols on the other hand 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.

Non-blocking algorithms can be lock-free or wait-free. *Lock-free* implementations guarantee that regardless of the contention caused by concurrent operations and the interleaving of their steps, always at least one operation will 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* algorithm [3], every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [4, 5], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [6].

The problem of multi-word wait-free read/write registers is one of the well-studied problems in the area of non-blocking synchronization [7–9, 9–21]. The main goal of the algorithms in these results is to construct wait-free multiword read/write registers using single-word read/write registers and not other synchronization primitives wich 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 synchronization power [3]. A lot of these solutions also involve elegant and symmetric ideas and have formed the basis for further results in the area of non-blocking 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 multi-word register write the new value in several copies (roughly speaking, as many copies as we have readers in the system), which may be very costly. However, modern architectures provide hardware synchronization primitives stronger than atomic read/write registers, some of them even accessible at a constant cost-factor away from read/write accesses. We consider it a very useful task to investigate how to use this power, to the benefit of designing economical solutions for the same problem, which can lead to usable structures in practice. To the best of our knowledge, none of the previous solutions have been implemented and evaluated on real systems.

In this paper we present a simple, efficient wait-free algorithm for implementing multi-word n-readers/single writer registers of arbitrary word length. In the new algorithm each multi-word write operation needs to write the new value only in one copy,

thus having significant gain in overhead. To achieve this, the algorithm uses synchronization primitives called *fetch-and-or* and *swap* (Fig. 10 in the Appendix), which are available in several modern processor architectures, to synchronize n readers and a writer accessing the register concurrently. Since the new algorithm is wait-free, it provides high parallelism for the accesses to the multi-word register and thus significantly improves performance. We compare the new algorithm with the wait-free one in [22], 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 to test them on three different architectures: UMA Sun-Fire-880 with 6 processors, ccNUMA SGI Origin 2000 with 29 processors and ccNUMA SGI Origin 3800 with 128 processors.

The rest of this paper is organized as follows. In Section 2 we describe the formal requirements that any solution to the problem that we are addressing must guarantee. In the same section, we also describe the related algorithms that we are using in the evaluation study. Section 3 presents our protocol. In Section 4, we give the proof of correctness of the new protocol. Section 5 is devoted to the performance evaluation. The paper concludes with Section 6, with a discussion on the contributed results and further research issues.

2 Background

2.1 System and problem model

In this paper we are interested in atomic multi-word shared registers [22, 23]. A *shared register* of arbitrary length is an abstract data-structure that is shared by a number of concurrent processes which perform either read or write operations on the shared register. In this paper 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. Operations performed by the same process are assumed to execute sequentially.

An implementation of a register consists of: (i) protocols for executing the operations (read and write); (ii) a data structure consisting of shared subregisters and (iii) a set of initial values for these. The protocols for the operations consist of a sequence of operations on the subregisters, called *suboperations*. The sub-operations on subregisters are reads, writes, *fetch-and-or* and *swap* (see Fig. 10 in the Appendix), which are either available directly on modern multiprocessor systems or can be implemented from other available synchronization primitives. 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*[3] if it guarantees that any process will complete each operation in a finite number of steps (sub-operations) 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. We assume that there is an 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.

Criterion 1. *[15] Lamport 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 N-O inversion**. *There exist no reads r_1 and r_2 such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$.*

2.2 Peterson's shared multi-word register

In [22] Peterson describes an implementation of an atomic shared multi-word register for one writer and many readers. The protocol does not use any other atomic sub-operations than reads and writes and is described below.

We have decided to compare our method with this algorithm because: (i) they are both designed for the 1-writer n-reader shared register problem; (ii) compared to other more general solutions based on weaker subregisters (which are much weaker than what common multiprocessor machines provide) this one involves the least communication overhead among the processes, without requiring unbounded timestamps or methods to bound the unbounded version (cf. [24, 25] for examples of such methods).

The idea of Peterson's register implementation is to use $n + 2$ shared buffers, which each can hold a value of the register, together with a set of shared handshake variables to make sure that the writer does not overwrite a buffer that is 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 Fig. 1 and the protocols for the read and write operations are shown in Fig. 2.

The writer first informs the readers that it is going to write to `BUF1` by setting `WFLAG`, then writes there and then unsets `WFLAG`. This way a reader may be able to detect if the writer has written in the buffer during the time the reader has read there. However, if the writer both starts and ends the write while that reader is reading, the reader will not notice this from a single flag. Therefore the writer has another variable `SWITCH` that it changes after each write. By checking `WFLAG` and `SWITCH` before and after reading the buffer, a reader can see if the writer has written that buffer concurrently with the read.

By this scheme a reader is guaranteed to know whether it has read a consistent value or not, but in case it has not it must be able to find one without waiting. To ensure that there will always be such a value, the reader always informs the writer when it starts a read by setting `READING[r]` to the opposite of `WRITING[r]`. The writer, in turn, will then write the new value to `COPYBUF[r]` of each reader that has announced that is has started to read since the last write.

Since the writer makes `WRITING[r]` and `READING[r]` equal after it has written the new value to `COPYBUF[r]`, a read that finds `WRITING[r]` and `READING[r]` equal knows that the value in `COPYBUF[r]` is valid. If `COPYBUF[r]` has not been written and there has been a concurrent write to `BUF1`, then the value read from `BUF2` is correct, since the writer writes `BUF2` after it writes `COPYBUF[r]`.

Peterson's register 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 value is large. While it is unlikely that one can do better using subregisters that support only reads and writes, most modern multiprocessor systems supports additional sub-operations, such as *swamp*. Our new register implementation uses these additional sub-operations to implement high-level read and write operations that only need to read or write one copy of the register value.

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 concurrent reads.
`WRITING`	Array of n Boolean	Used together with `READING` to handle concurrent reads.
`BUF1`	Buffer	Main buffer.
`BUF2`	Buffer	Backup buffer.
`COPYBUF`	Array of n buffers	An individual buffer copy for each reader.

Fig. 1. The shared variables used by Peterson's algorithm. The number of readers is n. BUF1 holds the initialized register value. All other variables are initialized to 0 or false.

2.3 Mutual-exclusion based solutions

For comparison we also evaluate the performance of two mutual-exclusion-based register implementations, one that uses a simple *spin-lock* with exponential back-off and the other a *readers-writers spin-lock* with exponential back-off to protect the shared register. The readers-writers spin-lock is similar to the simple spin-lock but allows readers to access the register concurrently with other readers.

3 The new algorithm

The idea behind the new algorithm is to remove the need for reading and writing several buffers during read and write operations by utilizing the atomic synchronization

```
READING[r] = !WRITING[r];
flag1   = WFLAG;
switch1 = SWITCH;
read BUF1;
flag2   = WFLAG;
switch2 = SWITCH;
read BUF1;
bad1 = switch1 != switch2 || flag1 || flag2;
if(READING[r] == WRITING[r]) {
   return the value in COPYBUF[r];
} else if(bad1) {
   return the value read from BUF2;
} else {
   return the value read from BUF1;
}
```

```
WFLAG = true;
write to BUF1;
SWITCH = !switch;
WFLAG = false;
for(each reader r) {
   if(READING[r] != WRITING[r]) {
      write to COPYBUF[r];
      WRITING[r] = READING[r];
   }
}
write to BUF2;
```

Fig. 2. Peterson's algorithm. The read operation on the left and the write operation on the right. Each reader has a unique index denoted by the variable r, which is used to index its set of variables. Lowercase variables are local variables.

primitives available on modern multiprocessor systems. These primitives are used for the communication between the readers and the writer. The new 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.

The shared variables used by the algorithm are presented in Fig. 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: The first field, the *pointer field*, contains the index of the buffer in `BUF` that contains the most recent value written to the register. The second field, the *reading-bit field*, holds a *handshake* bit for each reader that is set when the corresponding reader has followed the value contained in the pointer field.

A reader (Fig. 4) 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.

The writer (Fig. 5) needs to keep track of the buffers that are available for use. To do this it stores the index of the last written value `wptr` and a n-element array `trace`. In this array the writer records the buffer index where it last saw each reader, in persistent local memory. At the beginning of each write the writer selects a buffer index to write to. This buffer should be different from the last one it used and with no reader intending to use it. Then the writer writes the new value to that buffer and then it uses the suboperation *swap* to atomically read `SYNC` and update it with the new buffer index and clear the 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 limited by the size of the words that the two atomic primitives used can handle. If we are limited to 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.

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

Variable	Type	Description
SYNC	Unsigned word	Consists of two fields.
SYNC & PTRFIELD		Index of the buffer with the most recent register value.
bit PTRFIELD_LEN + r of SYNC		The reading bit for reader r.
BUF	Array of n+2 buffers	The buffers for the register value.

Fig. 3. The constants and shared variables used by the new 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.

```
R1  readerbit = 1 << (r + PTRFIELD_LEN)
R2  rsync = fetch_and_or(&SYNC,readerbit)
R3  rptr = rsync & PTRFIELD
R4  read(BUF[rptr])
```

Fig. 4. The read operation of the new algorithm. The variable r is a unique index for each reader to index their bit in SYNC.

```
W1  choose newwptr such that newwptr != wptr and newptr != trace[r] for all r;
W2  write(BUF[newwptr]);
W3  wsync = swap(&SYNC, 0 | newwptr); /* Clears all reading bits. */
W4  oldwptr = wsync & PTRFIELD;
W5  for each reader r {
W6     if (wsync & (1 << (r + PTRFIELD_LEN))) {
W7        trace[p] = oldwptr;
W8     }
W9  }
```

Fig. 5. The write operation of the new algorithm. The trace-array and oldwptr are static, i.e. stay intact between write operations. They are all initialized to zero.

4 Analysis

We first prove that the new algorithm satisfies the conditions in Lamport's criterion [15] (cf. Criterion 1 in section 2), which guarantee atomicity.

Lemma 1. *The new algorithm satisfies condition* "No-irrelevant"

Proof. Because a read r reads the value that is written by write $\pi(r)$, r's $read(BUF[j])$ operation (line R4 in Fig. 4) starts after $\pi(r)$'s $write(BUF[j])$ operation (line W2 in Fig. 5) starts. On the other hand, the starting time-point of the $read(BUF[j])$ operation is before the ending time-point of r and the starting time-point of the $write(BUF[j])$ operation is after the starting time-point of $\pi(r)$, so the ending time-point of r must be after the starting time-point of $\pi(r)$, or $r \not\to \pi(r)$. □

Lemma 2. *The new algorithm satisfies condition* "No-past"

Proof. We prove the lemma by contradiction. Assume there are a read r and a write w such that $\pi(r) \to w \to r$, which means i) write $\pi(r)$ ends before write w starts and

write w ends before read r starts and ii) r reads the value written by $\pi(r)$. Because $\pi(r) \rightarrow w \rightarrow r$, the value of $SYNC$ that r reads (line R2 in Fig. 4) is written by a write w' using the $swap$ primitive (line W3 in Fig. 5), where $w' = w$ or $w \rightarrow w' \rightarrow r$, i.e. $w' \neq \pi(r)$ because $\pi(r) \rightarrow w$. On the other hand, because r reads the buffer that is pointed by $SYNC$ (lines R2-R4), r would read the buffer that has been written completely by $w' \neq \pi(r)$. That means r does not read the value written by $\pi(r)$, a contradiction. □

Lemma 3. *The new 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 $\pi(r_2) \rightarrow \pi(r_1)$. Because i) $\pi(r_1)$ always keeps track of which buffers are used by readers in the array $trace[]$ (lines W4-W7 in Fig. 5) and ii) the buffer $\pi(r_1)$ chooses to write to is different from those recorded in $trace[]$ as well as the last buffer the writer has written, $wptr$ (lines W1-W2), r_1 reads the value written by $\pi(r_1)$ only if r_1 has read the correct value of $SYNC$ (line R2 in Fig. 4) that has been written by $\pi(r_1)$ (line W3 in Fig. 5).

On the other hand, because $r_1 \rightarrow r_2$, the value of $SYNC$ that r_2 reads must be written by a write w_k where $w_k = \pi(r_1)$ or $\pi(r_1) \rightarrow w_k$, i.e. $w_k \neq \pi(r_2)$ because $\pi(r_2) \rightarrow \pi(r_1)$. Moreover, because r_2 reads the buffer that is pointed by $SYNC$ (lines R2-R4), r_2 would read the buffer that has been written completely by $w_k \neq \pi(r_2)$ (lines W2-W3). That means r_2 reads the value that has not been written by $\pi(r_2)$, a contradiction. □

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

The complexity of a read operation is of order $O(m)$, where m is the size of the register, for both the new algorithm and Peterson's algorithm. However, in Peterson's algorithm the reader may have to read the value up to 3 times, while in our algorithm the reader will only read the value once and has to use the *fetch-and-or* suboperation. A write operation in the new 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 register algorithm however, the writer must in the worst case write to $n+2$ buffers of size m. The complexity of a write operation of Peterson's algorithm is thus of order $O(n \cdot m)$. As the size of registers and the number of threads increase, the new 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 new algorithm can be more expensive compared to the one used by Peterson's, but on the other hand the new 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 theorem

Theorem 1. *A multi-reader, single-writer, m-word sized register can be constructed using $n + 2$ buffers of size m each. The complexity of a read operation is $O(m)$. The complexity of a write operation is $O(n + m)$.*

5 Performance evaluation

As also mentioned in earlier sections, the performance of the proposed algorithm was tested against: i) Peterson's practical register algorithm, ii) a spinlock-based implementation with an exponential backoff and iii) a readers writers spinlock with an exponential backoff (for the latter cf. description in Section 2).

Method We measured the number of successful read and write operations during a fixed period of time. The higher this number the better the performance. In each test one thread is the writer and the rest of the threads are the readers. Two sets of tests have been done – one set where the contention is low and one set where the contention is high. During the high contention tests each thread reads or writes continuously with no delay between successive accesses to the multi-word register. During the low contention tests each thread waits for a time-interval between successive accesses to the multi-word register. This time interval is much longer than the time used by one write or read. Tests has been performed for different number of threads and for different sizes of the register.

Systems The performance of the new algorithm has been measured on both UMA and NUMA architectures. The three different systems 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.

There are two popular families among multiprocessor architectures – UMA (Uniform Memory Architecture) and NUMA (Non Uniform Memory Architecture). The difference between them is how the memory is accessed. 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 the memory attached to that node, but accesses to memory on another node has to be made over a network and is therefore significantly slower.

The systems were used non-exclusively, 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 [26] and a lock-free subroutine using the `compare_and_swap` hardware instruction on the SunFire machine. On the SGI Origin machines *swap fetch-and-or* and were implemented by the `__lock_test_and_set` and the `__fetch_and_or` synchronization primitives (provided by the system) [27], respectively.

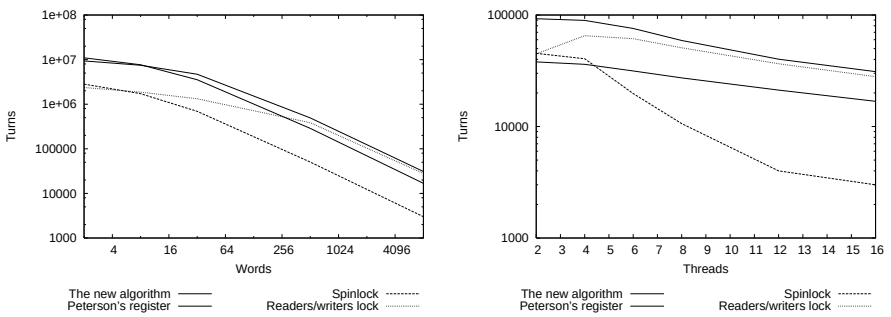
Results Following the analysis and the diagrams presenting the experiments' outcome, it is clear that the performance of the lock-based solutions is not even near the figures of the wait-free algorithms unless the number of threads is minimal (2) *and* the size of the register is small.

Moreover, as expected following the analysis results, the algorithm proposed in this paper performs at least as well and in the large-size register cases better than the wait-free solution by Peterson.

More specifically, on the UMA SunFire the new algorithm outperforms the others for large registers under both low and high contention (cf. fig: 6(a)-7(b)).

On the NUMA Origin 2000 platform (cf. fig. 11(a)-13(b) in the Appendix), we observe the effect of the particular architecture, namely the possibility for creating contention on the synchronization variable may become high, affecting the performance of the solutions. By observing the performance diagrams for this case, we still see the writer in the new 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 has to write to more buffers as the number of readers grow. The writer of the algorithm proposed here has no such problems. The latter phenomenon, though, has a 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 regular two buffers, become smaller. Hence, the readers' performance difference of the two wait-free algorithms under high contention becomes smaller.

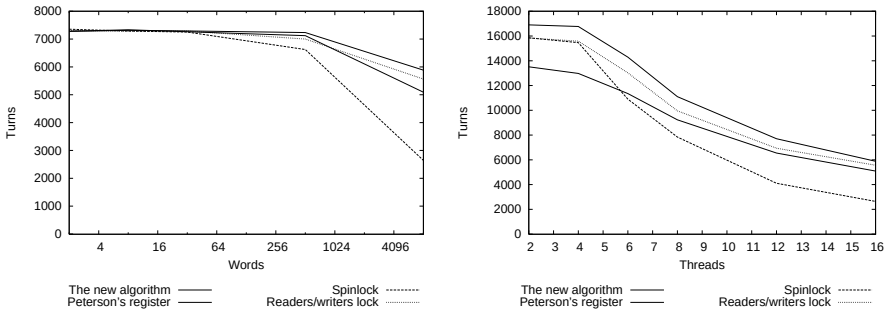
The results for the NUMA Origin 3800 (cf. fig. 8(a)-9(b)) are similar to the ones for the NUMA Origin 2000 . The problem with the writers, for the readers/writers lock solutions and Peterson's algorithm, is the same.



(a) Run with 16 threads

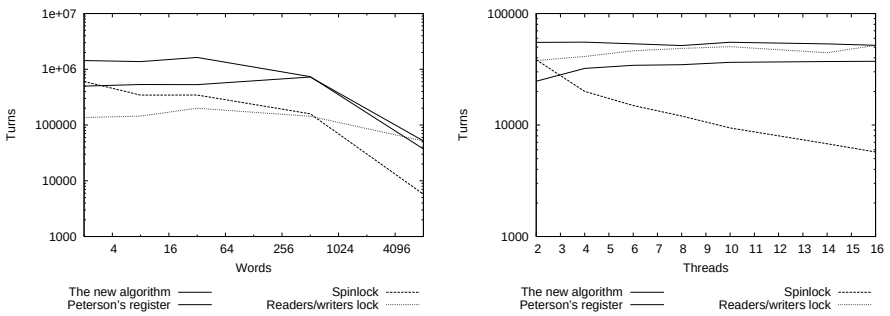
(b) Run with 8192 words

Fig. 6. Mean value from high contention test on the UMA SunFire 880



(a) Run with 16 threads (b) Run with 8192 words

Fig. 7. Mean value from low contention test on the UMA SunFire 880



(a) Run with 16 threads (b) Run with 8192 words

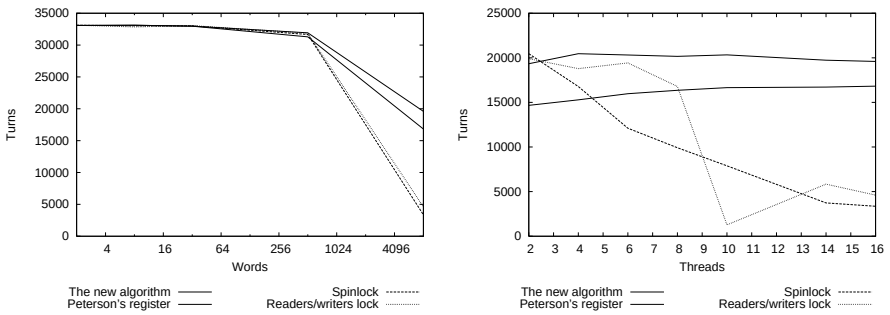
Fig. 8. Mean value from high contention test on NUMA Origin 3800

6 Conclusions

First, a simple and efficient algorithm of atomic registers (memory words) of arbitrary size is presented in this paper. The simplicity and the good time complexity of the algorithm are achieved via the utilization of two synchronization primitives commonly supported in hardware by modern multiprocessor systems.

Besides, the paper also presents a performance evaluation of i) the new algorithm; ii) a previously known practical algorithm that is based only on read and write operations; and iii) two mutual-exclusion-based register implementations. This evaluation is performed on three different well-known multiprocessor systems.

Since shared objects are very commonly used in parallel/multithreaded applications, such results and further research along this line, on shared objects implementa-



(a) Run with 16 threads

(b) Run with 8192 words

Fig. 9. Mean value from low contention test on NUMA Origin 3800

tions, is significant towards providing better support for efficient synchronization and communication for these applications.

References

1. Silberschatz, A., Galvin, P.B.: Operating Systems Concepts. 4 edn. Addison-Wesley, Reading (1994)
2. Sha, L., Rajkumar, R., Lojoczky, J.: Priority inheritence protocols: An approach to real-time syncronization. IEEE Transactions on Computers **39** (1990) 1175–1185
3. Herlihy, M.: Wait-free synchronization. ACM Transaction on Programming and Systems **11** (1991) 124–149
4. Tsigas, P., Zhang, Y.: Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In: Proceedings of the ACM SIGMETRICS 2001/Performance 2001, ACM press (2001) 320–321
5. Tsigas, P., Zhang, Y.: 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 (2002) 55–67
6. Sundell, H., Tsigas, P.: NOBLE: A non-blocking inter-process communication library. In: Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers. Lecture Notes in Computer Science, Springer Verlag (2002)
7. Bloom, B.: Constructing two-writer atomic registers. IEEE Transactions on Computers **37** (1988) 1506–1514
8. Burns, J.E., Peterson, G.L.: Constructing multi-reader atomic values from non-atomic values. In Schneider, F.B., ed.: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, Vancouver, BC, Canada, ACM Press (1987) 222–231
9. Haldar, S., Vidyasankar, K.: Counterexamples to a one writer multireader atomic variable construction of burns and peterson. Operating Systems Review **26** (1992) 78–87
10. Haldar, S., Vidyasankar, K.: Constructing 1-writer multireader multivalued atomic variable from regular variables. Journal of the ACM **42** (1995) 186–203
11. Haldar, S., Vidyasankar, K.: Buffer-optimal constructions of 1-writer multireader multivalued atomic shared variables. Journal of Parallel and Distributed Computing **31** (1995) 174–180
12. Haldar, S., Vidyasankar, K.: Simple extensions of 1-writer atomic variable constructions to multi-writer ones. Acta Informatica **33** (1996) 177–202

13. Israeli, A., Shaham, A.: Optimal multi-writer multi-reader atomic register. In Herlihy, M., ed.: Proceedings of the 11th Annual Symposium on Principles of Distributed Computing, Vancouver, BC, Canada, ACM Press (1992) 71–82
14. Kirousis, L.M., Kranakis, E., Vitányi, P.M.B.: Atomic multireader register. In van Leeuwen, J., ed.: Distributed Algorithms, 2nd International Workshop. Volume 312 of Lecture Notes in Computer Science., Amsterdam, The Netherlands, Springer, 1988 (1987) 278–296
15. Lamport, L.: On interprocess communication. Distributed Computing **1** (1986) 77–101
16. Li, M., Vitányi, P.M.B.: Optimality of wait-free atomic multiwriter variables. Information Processing Letters **43** (1992) 107–112
17. Li, M., Tromp, J., Vitányi, P.M.B.: How to share concurrent wait-free variables. Journal of the ACM **43** (1996) 723–746
18. Peterson, G.L., Burns, J.E.: Concurrent reading while writing II: The multi-writer case. In: 28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, IEEE (1987) 383–392
19. Singh, A.K., Anderson, J.H., Gouda, M.G.: The elusive atomic register. Journal of the ACM **41** (1994) 311–339
20. Newman-Wolfe, R.: A protocol for wait-free, atomic, multi-reader shared variables. In Schneider, F.B., ed.: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, Vancouver, BC, Canada, ACM Press (1987) 232–248
21. Vitányi, P.M.B., Awerbuch, B.: Atomic shared register access by asynchronous hardware (detailed abstract). In: 27th Annual Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, IEEE (1986) 233–243
22. Peterson, G.L.: Concurrent reading while writing. ACM Transactions on Programming Languages and Systems **5** (1983) 46–55
23. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems **15** (1993) 745–770
24. Haldar, S., Vitányi, P.: Bounded concurrent timestamp systems using vector clocks. Journal of the ACM **49** (2002) 101–126
25. Israeli, A., Li, M.: Bounded time-stamps. Distributed Computing **6** (1993) 205–209
26. Weaver, D.L., Germond, T., eds.: The SPARC Architecture Manual. Pretice Hall (2000) Version 9.
27. Cortesi, D.: Topics in IRIX Programming. Silicon Graphics, Inc. (2004) (doc #:007-2478-009).

A Appendix

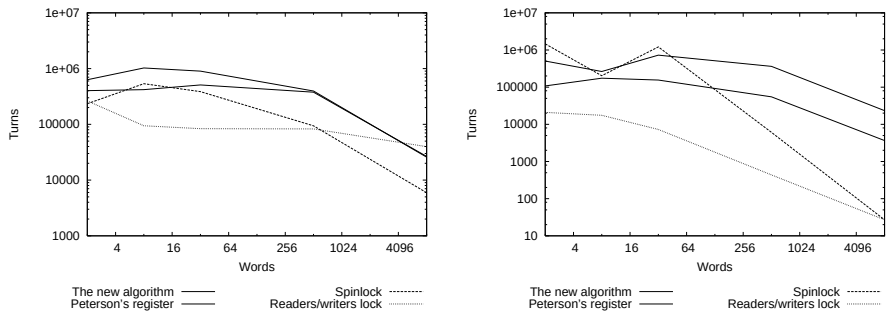
A.1 Extra figures

```
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. 10. The specifications of the *swap* and *fetch-and-or* suboperations.

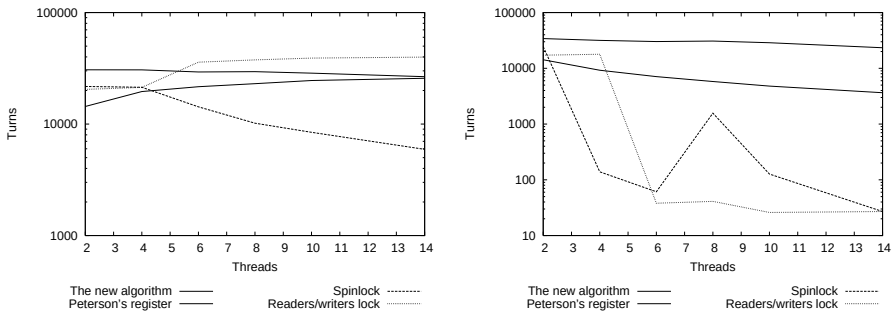
A.2 Results from Origin 2000



(a) Mean value

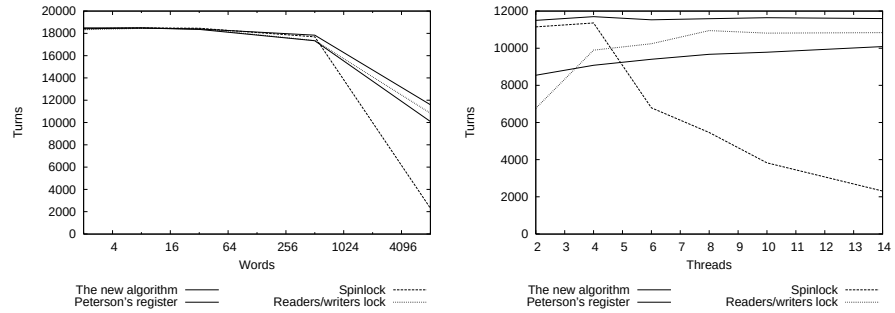
(b) Writers value

Fig. 11. High contention test with 14 threads on NUMA Origin 2000



(a) Mean value (b) Writers value

Fig. 12. High contention test with 8192 words on NUMA Origin 2000



(a) Run with 14 threads (b) Run with 8192 words

Fig. 13. Mean value from low contention test on NUMA Origin 2000