Optimizing a FIFO, Scalable Spin Lock
Using Consistent Memory*

Injong Rhee
Department of Math and Computer Science
Emory University
Atlanta, GA 30322
email: rhee@mathcs.emory.edu
Phone: (404) 727-5605, Fax: (404) 727-5611

April 1996

Abstract

This paper presents a FIFO queue-based spin lock that (1) uses only one atomic swap operation; (2) is scalable as it requires a constant amount of communication; (3) runs without a coherent cache support; and (4) provides a timing guarantee required for real-time applications. The algorithm is optimal in terms of the number of atomic operations required to solve a scalable mutual exclusion problem in NUMA architectures, improving on Craig's spin lock [5] that uses four atomic swap operations. It minimizes the number of atomic operations by replacing them with non-atomic read and write operations. This optimization can benefit greatly from modern multiprocessors where non-atomic memory operations are much more optimized than atomic operations. The algorithm runs correctly in various weakly consistent memories, providing a potentially significant speed-up over the algorithms with more atomic operations.

1 Introduction

The use of spin locks in shared-memory multiprocessor systems is ubiquitous as they provide efficient mutual exclusion for small critical regions. Especially, recently proposed FIFO queue-based spin locks (e.g., [1, 9, 12]) are also suitable for real-time systems as they guarantee an upper bound on the time for a task to busy-wait (i.e., spin) to acquire a lock. This upper bound can be used for determining the bound of the worst case execution time of the tasks that use the spin locks, and efficient synchronization provided by the spin locks.

*This research is supported in part by NSF grant ASC-9527186.
Table 1: Atomic operations supported by various microprocessors.

<table>
<thead>
<tr>
<th>Atomic Operation</th>
<th>Microprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic swap</td>
<td>Intel 80486, Intel Pentium, Motorola 88000, SPARC</td>
</tr>
<tr>
<td>test-and-set</td>
<td>Motorola 68040</td>
</tr>
<tr>
<td>compare-and-swap</td>
<td>Motorola 68040</td>
</tr>
</tbody>
</table>

Figure 1: Atomic operations supported by various microprocessors.

can benefit real-time thread systems or real-time micro kernel systems (e.g., [11]) where spin locks are one of main building blocks. Unfortunately, most of these spin lock algorithms require either strong atomic read-modify-write operations (e.g., compare-and-swap [12] and fetch-and-increment [1]) that are not supported by many modern microprocessors, or a coherent cache support [1, 9] which may not be supported in large-scale multiprocessor systems for its non-scalability (e.g., NUMA architectures).

Craig [5] proposed the first FIFO, scalable spin lock algorithm that uses only atomic swaps for its atomic operations, and runs without a coherent cache support. That it uses only atomic swaps makes the algorithm more portable as atomic swap is the most commonly supported atomic instruction in modern microprocessors (see Table 1). This algorithm also provides an example that atomic swaps can be more powerful than compare-and-swaps as Cypher [6] proved that no algorithm can solve the scalable mutual exclusion problem using only reads and writes, or compare-and-swaps, or both. The scalable mutual exclusion problem is a mutual exclusion problem of which solution should incur only a constant number of communications.

In this paper, we present another spin lock algorithm that provides the same merits as Craig’s, but only more efficiently: the algorithm uses one atomic swap operation – an improvement over Craig’s NUMA architecture-based algorithm which requires four atomic swap operations. Our algorithm utilizes memory consistency provided by the underlying systems on (non-atomic) reads and writes in solving potential data-race problems among processes. This is in contrast with the previous solutions that use atomic operations to deal with the race problems. Our algorithm is optimal in terms of the number of atomic operations used to solve the scalable mutual exclusion problem, followed from [6].

The algorithm runs correctly in various weakly consistent memories such as total store ordering memory [13] or processor consistency memory [8, 2, 7]. Since these weakly consistent memories allow many compiler and hardware optimizations to reduce the latency of non-atomic operations, the algorithm can provide a significant speed-up over Craig’s algorithm in these memory systems.

One disadvantage of our algorithm is that the memory space allocated for a process to acquire a lock cannot be reused for another lock. (Note that this is unlike array-based spin locks such as Graumke and Thakker’s [9] or Anderson’s [1] that require memory spaces linear in the number of processors. In our algorithm, the memory space is allocated on a per-process basis.) However, as long as a process uses the same lock, the process can reuse the space to acquire the lock again. We believe that this is not a serious drawback as most applications use a small set of locks repeatedly.
We implemented our algorithm and Craig’s on two different architectures (KSR1 and SGI-Challenge) that support sequential consistency and processor consistency respectively. In KSR1, our algorithm runs up to 10% faster, and in SGI-Challenge, up to 35% faster than Craig’s. The better performance under SGI Challenge is because the read and write operations in SGI Challenge are more optimized than in KSR1. Note that processor consistency is weaker than sequential consistency, and thus allows more hardware and compiler optimizations [4]. As future microprocessors (e.g., SPARC V9, Alpha and PowerPC) support even weaker and more optimized memory operations, we expect a further speed-up of our algorithm over Craig’s in those architectures.

Section 2 describes the memory models used in the paper. Section 3 presents our algorithm and Section 4 shows the experimental result.

2 The memory model

In this section, we provide an informal description of sequential consistency and processor consistency memory models. More precise and formal definition of these consistency models is available in the literature [10, 4, 2, 8, 7, 3].

A multiprocessor system is *sequentially consistent* if the result 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 [10]. As it does not allow a reordering of memory operations, sequential consistency limits possible hardware and compiler optimizations.

A little relaxation of the consistency requirement can provide a significant room for hardware and compiler optimizations. This relaxation can be best described by a canonical hardware view in Figure 2 (This figure is modified from the one given in [4].)
Initially Flag1 = Flag2 = 0

<table>
<thead>
<tr>
<th>Processor1</th>
<th>Processor2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag1 = 1</td>
<td>Flag2 = 1</td>
</tr>
<tr>
<td>if (Flag2 = 0)</td>
<td>if (Flag1 = 0)</td>
</tr>
</tbody>
</table>

critical region  critical region

Figure 3: Difference between sequential consistency and processor consistency. With sequential consistency, it is not possible that both processes enter the critical section at the same time, but it is with processor consistency.

**Processor consistency** allows a reordering of read operation followed by write operation. Write operations of a processor is buffered in the FIFO order in the write buffer of that processor before it is written to the memory (i.e., visible to the other processors). If there is no write in the buffer to the same location as a read followed by a write, the read can bypass the buffer, and may return the value either from the memory or other processor’s buffer. If a read is for the same memory location as a write in its write buffer, the read always returns the value from the buffer. Therefore, the memory operations of a process appear to the other processes to be executed in a different order than its program order although they are to that process (e.g., a read of a process can be executed even before the results of its previous writes are visible to the other processes). Additionally, all the write operations done by a process i must appear to be executed in its program order to all the other processes. However, the interleaving order among writes of any two processes i and j may be perceived differently by different processes.

The example given in Figure 3 illustrates the difference between sequential consistency and processor consistency. In the example, it is not possible that both Flag1 and Flag2 can be read 0 with sequential consistency, but it is with processor consistency. Processor consistency is weaker than sequential consistency in that any program runs correctly with processor consistency also runs with sequential consistency, but not vice versa. KSR1 supports sequential consistency, and MIPS 4400, used in SGI-Challenge, supports processor consistency.

Modern multiprocessor systems also support several atomic read-modify-write instructions (e.g., atomic swap and compare-and-swap) to provide faster synchronization. Atomic read-modify-write instructions appear to all the processes to be executed instantaneously. For example, an atomic swap of X with Y returns the value of Y and then writes the value of X to Y atomically so that no other operation can be performed to Y while this operation is in progress.

In addition, many multiprocessor systems support a *fence* operation that forces all the memory operations of a process invoked before the fence operation to be completed before any memory operations after the fence operation are invoked. For instance, SPARC V9 contains MEMBAR instructions, MIPS 4400 and PowerPC
Figure 4: This shows the state of the linked list after $P_5$ swaps the address of its spin variable with the content of Lock, which is the address of the spin variable of $P_2$. Now $P_5$ knows the address of the spin variable of $P_2$ and can connect the next of $P_2$ to its spin variable.

have SYNC instruction, and Alpha has MB. Note that atomic read-modify-write operations implicitly execute a fence operation before their invocation.

3 A spin lock algorithm

3.1 Informal Description

Omitting certain important details, the basic idea of the algorithm is the following. The more precise description of the algorithm is given in Figure 9. Each process joins a linked list (i.e. a queue) of processes that attempt to acquire a lock. Each process in the queue is aware of the address of a local variable of its immediate successor where its successor is spinning (we call the spin variable of the successor). The process at the head of the queue enters the critical region. After finishing the critical region, the process in the critical region sets the spin variable of its immediate successor, which effectively frees the successor from spinning and lets it enter the critical region.

The linked list is constructed using an atomic swap operation. A process trying to acquire a lock performs an atomic swap to a global variable, called Lock, with the address of its spin variable. This swap returns the value of Lock, which is the spin variable address of the previous process that performed the swap operation to the Lock. Note that because of the atomicity of the swap, each process has a unique predecessor. To form a linked list, each process makes the next field of the spin variable of its predecessor point to its spin variable (see Figures 4 and 5). Each spin variable contains the next field that points to its successor. It also contains some other data fields including the one on which its process spins.

The process at the head of the list that has not entered the critical region recognizes that it is at the head and enters the critical region. This is done in cooperation with its predecessor, if it has any, that releases the lock (i.e. leaves the critical region). The responsibilities of the process releasing the lock are to ensure that (1) if it has a successor, the successor gets the lock, and (2) if it doesn't have a successor, then whoever joins after the releaser acquires the lock without spinning.

There is a potential data-race between the releaser and its successor because it is possible that the releaser
Initially flag1 = flag2 = 0

P1                                    P2
1: flag1 = 1                        4: flag2 = 1
2: fence                              5: fence
3: if (flag2 == 0)                6: if (flag1 == 0)
   critical region                   critical region

(a)                                    (b)

Figure 5: The linked list after the spin variable of P5 is connected to P2. The process at the head of queue enters the critical region. After finishing the critical region, it writes to the spin variable area where the next process busy-waits, in order to notify that it has finished.

Initially free_successor = releaser_was_here = successor_was_here = false

Figure 6: Race breakers.

might be in the transition of finishing step (2) while its successor joins the list and start spinning. Then, the successor ends up spinning forever – a deadlock.

The race breaker described in Figure 6(a) can provide a solution. Note that this race breaker does not use any atomic read-modify-write operations. The fence operation at Line 2 (and 5) ensures that the value written at Line 1 (and 4) is visible to all other processors before Line 2 (and 5) is invoked. Thus, there is no situation that both processes are in the critical region in Figure 6(a). (Note that in a sequentially consistent memory, no fence operations are needed.) Likewise, in Figure 6(b), the fence operations ensure that there is no such situation where the releaser finds that successor_was_here is false while the successor also finds that releaser_was_here is false. The part of the algorithm that corresponds to this race breaker is shown from Line 17 to Line 23 for the successor and from Line 25 to 30 for the releaser in Figure 9.

Two subtle problems still remain. One problem arises if the releaser tries to acquire the lock and reinitializes its spin variable (i.e., sets releaser_was_here to false) before the successor sees it to be true. Then, its successor finds releaser_was_here false and might spin forever. The problem is solved by a data structure switching. The idea is to employ another spin variable for each process. (Each process now has two spin variables for each lock that it needs to acquire.) Every time a process acquires a lock, it alternatingly
Figure 7: P1 reuses spin\_variable1 only after P3 enters the critical region because at the second time P1 joins the linked list, it joins with a different spin variable.

uses the other spin variable. From Figure 7, when the releaser (P1) joins the linked list again after leaving the critical region, it uses a different spin variable (spin\_variable2). Now P1 does not enter the critical region until its “previous” successor (P3) does. After P1 enters the critical region again, it can reuse its “previous” spin variable (spin\_variable1 in the example) safely because its successor already left the linked list and no longer reads the variable. Thus, it is safe for the releaser to reuse that spin variable.

The other problem arises because the releaser may try to free the successor even if the successor is not spinning. This happens when the successor finds out that releaser\_was\_here is true at line 7, and the releaser is just about to set free\_successor to true. The successor enters and leaves the critical region. Then, the successor can try to enter the critical section even before the releaser finishes line 7. Figure 8 shows the situation. In 8 (a), P3 enters the critical region before P1 writes to P3. After some time later, in (b), P3 joins the queue with the same spin variable that P1 is still pointing to, and P1 has not finished writing to P3 yet. Then, P3 is waiting for P2 to set it free from spinning. In the meantime, P1 writes to P3 setting P3 free, violating mutual exclusion. In order to remedy the problem, we ensure that the releaser always sets free\_successor to true only when it is certain that the successor is spinning for it. Thus, P1 would not write to P3 unless P3 is spinning for P1. Lines 28 and 29 of the codes in Figure 9 ensure this.

Pseudo-code of lock acquire and release procedures appears in Figure 9 and an example program that shows how to call these procedures appears in Figure 10.

### 3.2 Correctness Proofs

We define a few terms before we prove that the algorithm ensures mutual exclusion, starvation freedom and the FIFO entrance to the critical region.

We say that a write is visible to a process only if any read of that process to the same location returns the value written by the write or any subsequent write. We say that a process is in the trying region if it is executing within Line 14 and Line 23 inclusively. We also denote a local variable of a process by the variable name subscripted with its process id (e.g., \texttt{pred}_i).

The following lemma is true because of the property of the atomic swap operation at Line 14.

**Lemma 3.1** At any time, there is a total ordering \(\prec\) among all processes in the trying and critical regions such that for two processes \(i\) and \(j\), \(i \prec j\), if and only if \texttt{pred}_i is equal to \texttt{spinitem}_j.
Figure 8: In (a), $P_3$ enters the critical region before $P_1$ writes to $P_3$. After some time later, in (b), $P_3$ joins the queue with the same spin variable that $P_1$ is still pointing to, and $P_1$ has not finished writing to $P_3$ yet. Then, $P_3$ is spinning for $P_2$ to set it free from spinning. In the meantime, $P_1$ writes to $P_3$ setting $P_3$ free, violating mutual exclusion.

**Lemma 3.2** For a process $i$ in the trying region, if a process $k$, such that $i \sim k$, is in the trying or critical region, then process $i$ is not in the critical region.

**Proof:** If $k$ is in the trying or critical region, $\text{spinitem}_k \to \text{locked}$ is true as the write operation at Line 11 should be visible to process $i$ before Line 16 is performed (because of the implicit fence operation of the atomic swap at Line 14). While $k$ is in the trying or critical region, it hasn’t executed Line 25. Therefore, if statement at Line 19 will be satisfied, and $i$ will be spinning at Line 21. Since $k$ is the only process whose $\text{spinitem}_k$ is equal to $\text{pred}_i$, $i$ is in the trying region unless $k$ executes Line 25 or 30.

**Theorem 3.3** (Mutual exclusion) No two processes are in the critical region at the same time.

**Proof:** For a process $i$ in the trying region, if $\text{pred}_i$ is null, then no other process in the trying region has its $\text{pred}$ equal to null. This is because Line 16 is the only statement that changes the value of $\text{Lock}$, no process has its $\text{spinitem}$ equal to null, and the swap operation ensures that only one process has the initial value of $\text{Lock}$ which is null. Because of Line 15, $i$ enters the critical region, and no other process is in the critical region at that time. If $\text{pred}_i$ is not null, then, by Lemmas 3.1 and 3.2, it is trivially true that when $i$ enters the critical region, there is no other process in the critical region.

The following lemma is also true because the pointer to $\text{spinitem}$ is alternated between the dual structures in $\text{spinStruct}$ (see Figure 10). We say that a spin variable is initialized if a process executes Line 11 with that spin variable.

**Lemma 3.4** After $i$ enters the trying region, the $\text{spinitem}$ pointed to by $\text{pred}_i$ is not initialized before $i$ enters the critical region.
type spin_variable = record
  next : spin_variable \Initially NULL
  locked : Boolean \Initially FALSE
  status : spin_variable \Initially NULL
  succ_status: (PENDING, SPINNING, DONE)
end

type spin_variable_struct = record
end

turn : int \Initially 0

Figure 9: A FIFO, Scalable Spin Lock.

Theorem 3.5 A process $i$ in the trying region eventually enters the critical region.

Proof: If $\text{pred}_i$ is null, then $i$ enters the critical region by Line 15. If $\text{pred}_i$ is not null, let $k$ be the process such that $i \sim k$. 

9
Lock: lock \ lock and a global variable

Procedure get\_spinitem(spinStruct: spin\_variable\_struct) := spin\_variable
spinStruct.turn := spinStruct.turn ^ 1 // this exclusive OR flips turn between 1 and 0.
return "spinStruct.spinitems[spinStruct.turn]"
\ // this returns the address of the current spinitem

proc()
SpinStruct: spin\_variable\_struct
spinitem: ^spin\_variable = get\_spinitem(spinStruct)
acquire\_lock(Lock, spinitem)
critical\_region()
release\_lock(Lock, spinitem)

Figure 10: An example program

We only need to prove that if process $i$ is at Line 21, spinitem$_i$ \rightarrow status becomes equal to pred$_i$ (= spinitem$_k$) eventually. When pred$_i$ \rightarrow locked is true, then $k$ hasn’t executed Line 25 – true because of Lemma 3.4 and the code, and because of the fence operation at Line 26, it hasn’t executed Line 27. If $i$ is at Line 21, then it must have executed Lines 17 and 20 and the write at Line 17 much be visible to process $k$ because of the fence operation at Line 18. Thus, when $k$ executes Line 27, spinitem$_k$ \rightarrow next is not null. Then $k$ will see spinitem$_k$ \rightarrow succ\_status is equal to SPINNING, and execute Line 30. In addition, no other process but $k$ can write to spinitem$_i$ \rightarrow status while $i$ is spinning at Line 21. Thus, $i$ eventually enters the critical region.

The following theorem combined with Theorem 3.5 proves starvation freedom.

**Theorem 3.6** A process $k$ eventually finishes executing Procedure release after leaving the critical region.

The following corollary is true because of Lemma 3.1 and Theorem 3.5, and proves that processes enter the critical region in the FIFO ordering imposed by the total ordering ‘\rightarrow’.

**Corollary 3.7** Processes in the trying region enter the critical region in the total ordering ‘\rightarrow’.

4 Experiment

We implemented our algorithm and Craig’s algorithm in two multiprocessor architectures and compared their mean response times. The response time is defined to be the time from when a process attempts to
acquire a lock to when the process releases the lock subsequently. In our experiment, each process makes 10^5 loops of lock acquisition and release with an empty critical region, and the mean response time of a process is obtained by dividing the total execution time of the process by 10^5. In the experiment, each process is assigned to a different processor.

KSR1 (32 processors), and SGI Challenge (11 processors) are used for the experiment. KSR1 supports sequential consistency; and SGI Challenge supports processor consistency. As KSR1 and SGI Challenge do not have atomic swap instructions, we used a cache line locking mechanism in KSR1, and atomic \textit{load-linked} and \textit{store-conditional} instructions of MIPS 4400 microprocessor in SGI Challenge to implement an atomic swap operation. In SGI Challenge, \texttt{SYNC} is used for memory fence operations, and in KSR1, no fence operation is needed as it supports sequential consistency.

Figures 11 and 12 show the mean response times of the two algorithms over the various numbers of participating processes in KSR1 and SGI-Challenge respectively. In KSR1, our algorithm gives about 10 % faster response time than Craig’s algorithm. This is rather surprising because we expected that the performance would be much better as it doesn’t require any fence operation. We suspect that the slow performance in KSR1 is mostly due to its slow implementation of sequential consistency. KSR1 supports all cache concept where each local memory acts as a cache so that all the data accessed can be duplicated to many caches. Thus, each write operation, before its invocation, generates a write invalidate message and is initiated only after the invalidation is acknowledged, which slows down the write operation. However, we expect much more speedup of our algorithm on future microprocessors (e.g., MIPS R10000, Intel P6) that implement sequential consistency more efficiently.

In SGI Challenge, our algorithm runs up to 35 % faster than Craig’s algorithm. The better performance under SGI-Challenge is because the read and write operations in SGI-Challenge are much more optimized than those in KSR1. Note that processor consistency is weaker than sequential consistency, and thus allows
more hardware and compiler optimizations [4].

Since we implemented atomic swap in software in the experiment, the experimental result may not show the true performance of the two algorithms. However, we focus on the trend of the result that in more optimized architectures, our algorithm can show more (possibly significant) improvement over other algorithms that use more atomic operations. Our experiment confirms this intuition because the implementation of atomic swap in SGI Challenge is, in fact, more efficient than in KSR1. As future microprocessors (e.g., SPARC V9, Alpha and PowerPC) support even weaker and more optimized memory operations, we expect a further speed-up of our algorithm.

5 Conclusion and future work

A FIFO, scalable spin lock algorithm is presented for real-time systems. The algorithm uses only one atomic operation, improving Craig’s NUMA architecture based algorithm that uses four atomic operations. The algorithm replaces some of atomic operations required to eliminate data-race conditions with non-atomic read and write operations. This optimization benefits greatly from memory operation optimizations performed by compilers and hardware on reducing the latency of non-atomic reads and writes operations. The algorithm runs correctly on various multiprocessors with weakly consistent memories, and shows a significant speed-up over Craig’s algorithm. In the future multiprocessors with even more weakly consistent memories, the algorithm is expected to show a further improvement.

Our performance result confirms our initial conjecture that by replacing the atomic operations with cheaper non-atomic operations combined with some fence operations can improve the performance of spin locks substantially. However, it is yet inconclusive how much this improvement affects the real performance (or the worst case time) of an application program. More experiments involving real applications answer this question, and we leave them for future work.

The algorithm presented here is designed only for non-preemptive systems where at most one process is running at each processor. Unfortunately, queue-based spin locks does not perform very well under a preemptive environment. This is because a preempted process can be granted the lock, so that all the processes behind the preempted process in the queue also have to wait for the preempted process to be rescheduled. These preemptions also cause a severe variation in the times taken to acquire the lock, which could be intolerable, especially in real-time systems.

The preemption handling techniques that either prevent or recover from “inopportune” preemptions are being actively developed [16, 15, 14]. Most of these techniques are designed by adding a few more atomic operations on top of existing spin lock algorithms. Thus, non-preemptive spin lock algorithms such as the one presented here can also be used to design preemptive spin lock algorithms. It will be interesting to see how our algorithm can be combined with those preemption handling techniques to develop an efficient spin lock that is immune to preemption.
Software Availability: The software implementation of our algorithm and Craig's is available in ftp.machcs.emory.edu:/pub/rhee/{ksr,sgi}.tar.gz.

Acknowledgement: The author would like to thank Graham Riley, Tim Robinson and the Center for Novel Computing at Manchester University for the permission to use their KSR1 machine, and Michael Scott and Leonidas Kontothanassiss for the arrangement to use their SGI Challenge machine. Special thanks go to Travis Craig who found a technical error in an earlier version of the spin lock algorithm presented in this paper, and anonymous reviewers whose comments helped improve the presentation of the paper greatly.

References


