Fairlocks - A High Performance Fair Locking Scheme

Swaminathan Sivasubramanian, Iowa State University**,** *swamis@iastate.edu*
John Stultz, IBM Corporation**,** *jstultz@us.ibm.com*
Jack F. Vogel, IBM Corporation, *jfv@us.ibm.com*
Paul McKenney, IBM Corporation, *Paul.McKenney@us.ibm.com*

Abstract

Over the past several decades, much research has been done in the area of modeling, simulating, and measuring the performance of locking primitives under conditions of low and high contention and with attention to memory locality of the locking data structures. Most of the existing locking primitives are not fair with respect to lock grants and can cause *lock starvation* among CPUs during high contention. Locking primitives proposed to eliminate lock starvation employ complex schemes to achieve fairness, resulting in poor performance under low contention. In this paper, we propose a new locking scheme, called *fairlocks,* which, on many architectures, is as fast as test-and-set locks during low contention, and maintains both fairness and data locality for lock grants.

Keywords: locking synchronization performance, lock starvation

1. Introduction

In order for parallel shared-memory multiprocessors to scale well, low lock contention levels must be maintained. However, existing code that experiences high lock contention must often be used as is until a redesigned version of the code becomes available. In addition, it may not be worthwhile to optimize code that executes infrequently (for example, handlers for rare error conditions). Primitives that increase data locality, while providing some fairness guarantees, can improve the performance of such code while simultaneously preventing unfairness and lock starvation. Moreover, the increase in instruction execution rate outstrips the reductions in global latencies among large-scale multiprocessors, as shown in Figure 1. The figure shows that memory accesses were less expensive than instructions in the early 80s; however, the Moore's-law-driven increases in CPU core performance have outstripped those of memory, so that now literally hundreds of instructions can execute in the time required to complete a single memory access. This motivates the need for locking primitives that preserve memory locality or for some solution from the underlying processor architecture. Thus, some new architectures provide a solution to this problem by providing closely bound groups of CPUs, called "nodes," with lower latencies within nodes than between nodes. Examples of such architectures include cache-coherent non-uniform memory-access (CC-NUMA) architectures [DASH, FLASH, NUMA-Q], shared-cache architectures [CMP, NUMA-Q], and systems with hierarchical system busses.

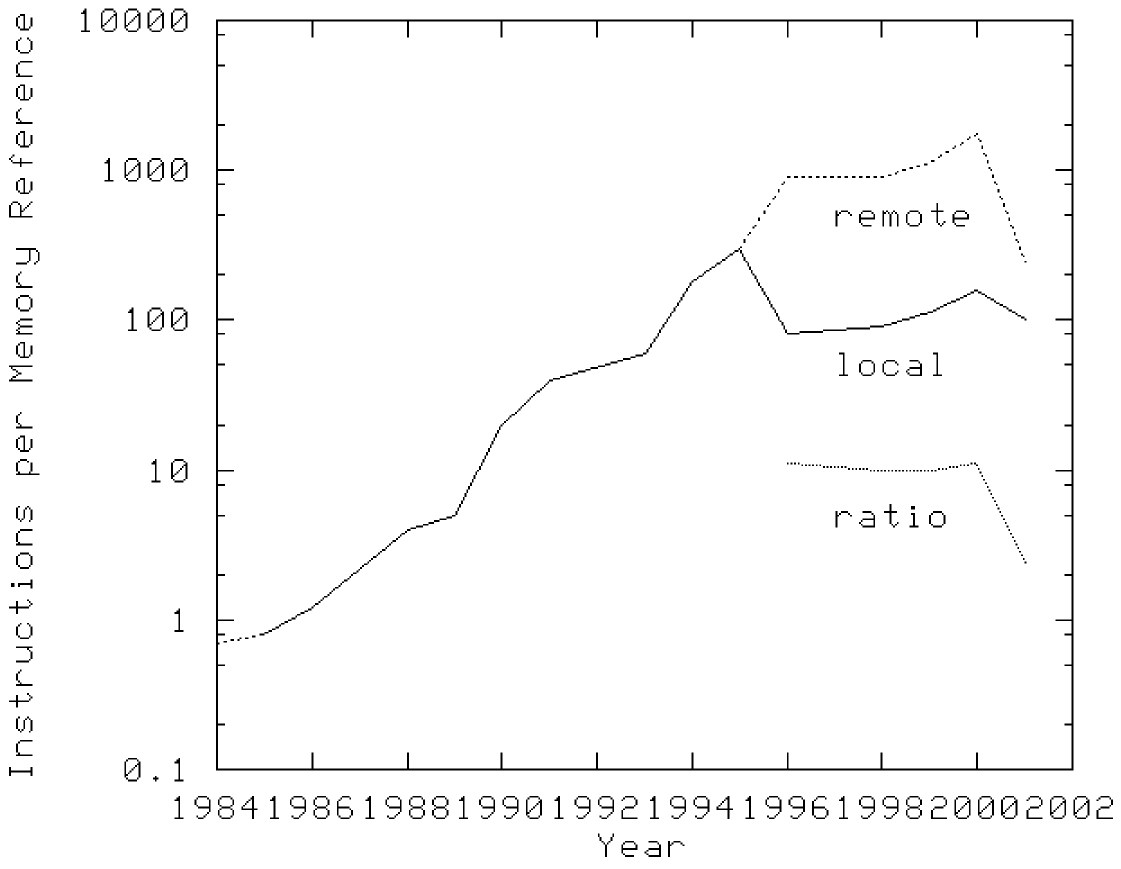

Figure 1: Memory Latency Trend

In these architectures, the memory latency for CPUs within a node, *intra-node latency*, is much lower than memory latency for access across nodes, *inter-node latency*. These architectures can cause lock starvation during high lock contention periods, as the CPUs on the same node as the one releasing the lock have an unfair advantage over the other CPUs (due to their ability to access the memory faster). Hence, locks are always kept within a node and starve CPUs on the other nodes. To prevent this starvation, the locking primitives should maintain some kind of fairness among CPUs. In this paper, we analyze the existing locking schemes and evaluate their performance based on aspects of complexity and fairness. Later, we propose a new locking which, with little complexity, eliminates the problem of lock starvation in the high contention case and minimally impacts performance in the low contention case.

The remainder of this paper is organized as follows: Section 2 discusses the existing solutions and evaluates them based on aspects of their complexity and fairness. Section 3 describes our proposed locking scheme, *fairlocks*. Section 4 states the system conditions and assumptions made in the paper. Section 5 contains the pseudocode for the proposed locking scheme. Section 6 presents the performance results of the locking scheme in comparison with the existing locking schemes. Section 7 concludes the paper.

2. Existing Solutions

This section outlines several representative locking primitives that have been produced over the past few decades and studies their performance with respect to throughput and fairness.

Simple test-and-set spinlocks have been used for decades and are often the primitive of choice for low contention regions because of their extremely short code-path lengths. However, they perform poorly under high contention because of lock starvation caused by the disparity in memory latency for intra-node and inter-node access. In CC-NUMA systems, simple spinlock can also experience unfairness and even starvation—the CPUs on the same node as the CPU releasing the lock have an unfair advantage in acquiring the lock. In one case in our study, the CPUs on one node were granted the lock over 2000 times more frequently than the CPUs on another node, where both nodes were running identical workloads (as shown in Section 6). This poor performance and unfair access has motivated researchers to propose a large number of alternative locking primitives.

Ticket locks [*ticket, olock*], queued locks [*ticket, olock*] and adaptive queued locks [*beng-hong lim*] solve the unfairness problem and avoid the excessive memory contention problem that simple spinlock faces at high contention [*MCS*]. However, queued locks implement a blind first-come-first-served discipline. This implementation results in lower levels of performance on CC-NUMA systems than do simple spinlocks with respect to the average throughput, because of its complex scheme of maintaining queues for locking.

Reader-writer spinlocks [*rw-spin*] allow reading processes to proceed concurrently, without lock contention or memory contention. Specially constructed implementations [*Hsieh & Weihl*] are also free of memory contention. However, not all algorithms can be designed to use primarily read-side locking. For write-intensive workloads, reader-writer spinlocks perform even worse than simple spinlock [*pdcs'99*]. Read-copy update [*pdcs'98*], like reader-writer spinlocks, allows reading processes to proceed concurrently, with neither memory contention nor lock contention. But again, not all algorithms can be designed to use read-copy update.

J-Lock [*jlock*] provides an effective solution for the problem of lock starvation by using a global bitmask and per-CPU spin pointers. Although this solution provides fairness, it is a complex scheme and requires disabling interrupts on each acquisition and release in order to avoid race conditions. The added overhead imposed by disabling interrupts makes jlock uncompetitive at low levels of contention.

The algorithms discussed above either do not provide fairness at high lock contentions at all or provide fairness among CPUs with complicated schemes, making them expensive to use under low lock contention periods. In contrast, the algorithm described in this paper is simple and, on many architectures, performs as well as test-and-set-bit locks at low contention levels while providing fairness among CPUs at high contention levels. Thus, the algorithm performs well under low lock contention and high lock contention periods and maintains absolute fairness under both cases.

3. Fairlocks Overview

The fairlocks algorithm performs as well as test-and-set-bit locks during low contention and provides fairness among CPUs during high contention. The algorithm uses a simple one-word per-lock bitmask. The bitmask is divided, as shown in Figure 2. The rightmost bit is called the LOCK_BIT, which indicates whether a CPU is holding the lock. The other bits in the bitmask correspond to the spin bit of the CPUs and mean that the CPUs want the lock and are spinning on their respective bit.

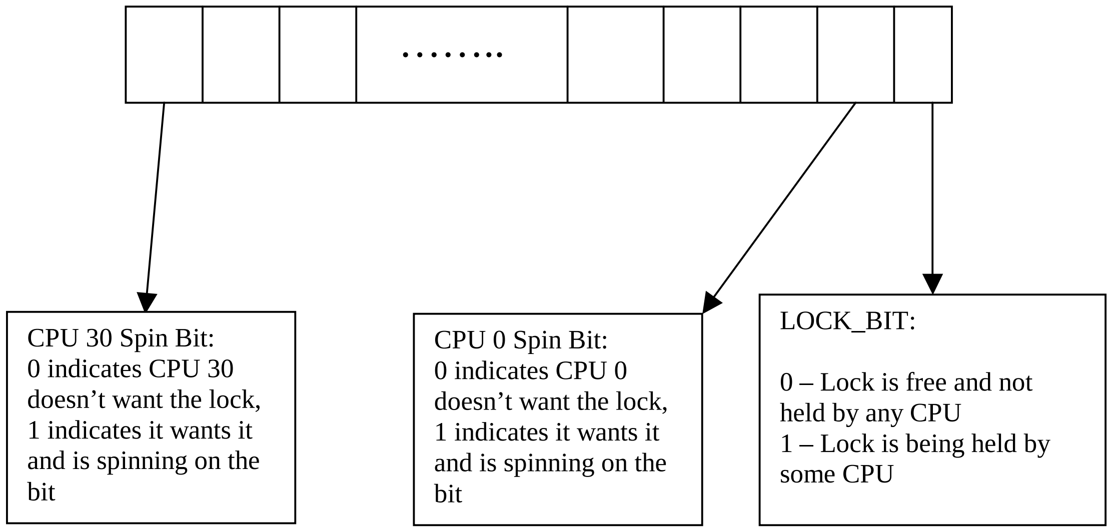
……..
CPU 30 Spin Bit: 0 indicates CPU 30 doesn’t want the lock, 1 indicates it wants it and is spinning on the bit
CPU 0 Spin Bit: 0 indicates CPU 0 doesn’t want the lock, 1 indicates it wants it and is spinning on the bit
LOCK_BIT:
0 – Lock is free and not held by any CPU
1 – Lock is being held by some CPU

Figure 2: Fairlock Structure

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1

Figure 3: Example of the Fairlock Bitmask

An example of a typical/possible bitmask is shown in Figure 3. The figure shows that the LOCK_BIT is being set, indicating that the lock is being held by one of the CPUs (other than those whose CPU spin bits are being set). The figure further shows that the CPU spin bits (with bits numbered from 0 to 31 from left to right) 1, 4, and 8 are set, indicating that CPUs 0, 3 and 7 require the lock is spinning for them. The locking scheme works as follows:

A CPU acquiring a lock checks the LOCK_BIT to see if it is set. If it is not set (if it is equal to 0), then it sets the LOCK_BIT and holds the lock. Otherwise, it sets its corresponding spin bit and spins on the particular bit, waiting for the bit to be changed to

0. If the bit becomes 0, then the CPU holds the lock. In the example in Figure 3, CPUs 0, 3 and 7 have their spin bits set and spin on their respective bit, waiting for it to be changed to 0.

A CPU releasing a lock checks to see if some other CPU wants the lock (by checking the spin bits), and if no bits are set, clears the LOCK_BIT in order to fully release the lock. If even one of the CPU spin bits are set, then it finds the next CPU to whom the lock must be handed off, in a round-robin fashion (starting from its spin bit), and clears the bit corresponding to the selected CPU. For example, in a scenario like Figure 3, if CPU 2 is releasing the lock, it finds that there are CPUs spinning for the lock. It therefore selects the next CPU that wants the lock, scanning from its bit position in a round-robin fashion, which is CPU 3. CPU 2 would therefore clear CPU 3’s bit (bit 4), thus handing the lock off to CPU 3.

The primary advantage of our proposed locking scheme is its good performance during both low and high lock contention phases. During low contention phases, the locks perform nearly as well as test-and-set bit locks, as the fast path (the path used under low contention) of fairlocks is extremely simple and involves only testing and setting of LOCK_BIT. During high lock contention phases, it eliminates lock starvation among CPUs, and the overhead introduced in the slow path (the path executed during high contentions) for maintaining this fairness is less. Thus, the proposed fairlocks can maintain high system performance under all contention levels.

4. Conditions and Assumptions

The locking algorithm presented in this paper is designed for the following conditions:

1. The system provides a shared-memory model.
2. Performance is critical at low levels of contention, since well designed algorithms and systems maintain low lock contention.
3. Fairness is critical at high levels of contention, because starvation of a portion of the system often has the same effect as a *system hang or crash*.
4. CPUs have equal priority with respect to the lock. Although the algorithms presented could easily be adapted to consider priorities, these more complex schemes are outside the scope of this paper.
5. The algorithms must support conditional-acquisition primitives, which either immediately acquire the lock, or immediately return a failure indication.

5. Fairlocks Details

This section discusses the implementation details of the fairlocks algorithm. We present the pseudocode for the scheme. We have eliminated issues such as debug assist code, compiler optimizations, and cache alignment padding for brevity.

Primitives Required

The pseudocode for the algorithm makes use of the following primitives:

- **cmpxchg(ptr,old,new):** atomically compares the value pointed to by "ptr" with "old" and if the two values are identical, stores "new" through "ptr" and returns old if successful.
- **clear_bit (bitnum,ptr):** atomically clears the "bitnum" bit in the value pointed by "ptr."
- **test_and_set_bit(bitnum, ptr):** atomically sets the "bitnum" bit in the value pointed by "ptr" and returns the old bit value.

Data Structures

Fairlocks use a single word bitmask, which allows it to be competitive with test-and-set-bit locks at low contention levels and to maintain high efficiency and fairness at high contention levels.

Pseudocode

numa_trylock(lock): (Conditional acquisition of a lock)

```
return(!test_and_set_bit(LOCK_BIT, lock));
```

Figure 4: Pseudocode for Conditional Locking

In numa_trylock, the algorithm does a test_and_set_bit and checks to see whether the LOCK_BIT (see Figure 2) was set. If it was not set, then the lock has been acquired and it returns TRUE; otherwise some other CPU already holds the lock and it returns false.

numa_lock(lock):

The unconditional acquisition of the lock is as follows:

```
reacquire:
if(test_and_set_bit(LOCK_BIT, &lock)) {
   mymask = 1U<<(smp_processor_id()+1);
   mylock = lock;
   if(mylock&1) goto reacquire;
   actual = cmpxchg(&lock, mylock, mylock|mymask);
   if(actual != mylock) goto reacquire;
   while(lock & mymask) {/*spin*/}
}
```

Figure 5: Pseudocode for Unconditional Locking

In the unconditional locking routine described in Figure 5, line 2 does a check on the LOCK_BIT and sets it. If LOCK_BIT was not set previously, then it has the lock and returns. Otherwise, it sets its spin bit (line 3 and 6) and spins on it (line 8). Lines 5 and 7 are present to eliminate race conditions.

numa_unlock(lock):

The unconditional release of the lock is as follows:

```
1  if(cmpxchg(&lock, 1U, 0U) != 1U)
2  {
3     mask = (1U<<(smp_processor_id+2))-1;
4     cpus = lock & ~mask & LOCK_BIT_MASK;
5     if(!cpus)
6        cpus = lock & mask & LOCK_BIT_MASK;
7     nextcpu =ffs(cpus) –1;
8     clear_bit(nextcpu,lock);
9  }
```

Figure 6: Pseudocode for Unlocking

In the unlock routine pseudocode described in Figure 6, line 1 checks to see if any spin-bits are set; if not it clears the LOCK_BIT and returns. If some spin bits are set, it finds the next CPU to give the lock (lines 3-7) and clears the selected spin bit (in line 8).

6. Performance Analysis

In this section, we evaluate the performance of fairlocks in comparison with spinlocks inside the Linux® kernel with respect to the following metrics: fairness, lock overhead, and overall system performance impact. Note that currently fairlocks is implemented in C to simplify portability between many architectures supported by Linux, whereas spinlocks are implemented in optimized assembly for each architecture. Future optimizations to fairlocks will have assembly language optimizations to further improve performance.

6.1 *Lock Overhead: Fairness*

We created a user-space test application to consider lock overhead, where a single thread acquires and releases a spinlock, then a fairlock, a specified number of times. We measured the amount of time it took to acquire and release each type of lock and compared the two locking mechanisms. We used this test to measure the overhead of fairlocks compared to that of spinlocks. We further modified the test to measure the number of times the lock was acquired by each thread. This allowed us to see exactly how the memory latency between nodes affected lock throughput on each quad. Additionally, we changed the code to flexibly pin threads to CPUs. The result of running four threads (for 10 seconds), with two on each quad, is shown in Table 1.

Thread #	Spinlock – Acquisitions	Fairlocks – Acquisitions
Thread 0 (Quad 0)	780297	310915
Thread 1 (Quad 0)	801152	316622
Thread 2 (Quad 1)	26213	238295
Thread 3 (Quad 1)	26558	234268

Table 1: Number of Lock Acquisitions for 4 Threads

Table 1 shows that spinlocks favor Quad 0 30 times over Quad 1, illustrating a fairly bad case of lock starvation. However, fairlocks distributes the locks much more fairly, as shown in Table 2. Further, the problem persists with an increase in contention and the result of running eight threads, with four on each quad, running for 10 seconds, also shown Table 2.

Quad #	Spinlock Acquisitions	Fairlock Acquisitions
Quad 0 threads	1293874	539797
Quad 1 threads	312856	539696

Table 2: Number of Lock Acquisitions for 8 Threads

Spinlock starvation can be observed again in Table 2, whereas the fairlocks are extremely fair in the same scenario. Spinlock starvation observed in this case is lower than the previous case, but could be due to the fact that the 8 threads use all the CPUs in the machine, causing a thread to be frequently scheduled out to allow other system processes to run.

6.2 *Lock Overhead: Throughput*

The above test was also used to measure the overhead of both the locks. The results of these measurement are shown in Table 3.

	# of lock/unlock calls	Time taken (in micro-seconds)
Spinlocks	100000000	2937928
Fairlocks	100000000	5910851

Table 3: Throughput of Locks

Table 3 shows that fairlocks have about twice the overhead of spinlocks in the no-contention case. Again, it must be noted that the throughput results measured are for assembly optimized spinlock code, and the fairlock code is written in C.

We created another user space test to measure the lock throughput of both the implementations under different levels of contention. Additionally, we studied the lock acquisition frequency of different CPUs to study the effects of lock starvation, if any. We implemented the tests by creating a master process that spawned off a specified number of threads, pinning the nth thread to the nth processor in the system. After all the threads were created and ready, the controlling thread signaled them to start and went to sleep for a specified amount of time. While the master thread slept, the children threads entered a loop where they all tried to grab and release a common lock, incrementing an internal counter each time. After the master thread woke up, it signaled the children to stop, then summed up the total number of locks acquired and printed the results.

Figure 7 shows the results we obtained running the test on a 2-quad, 8-way IBM® NUMA-Q® system. The figure shows that fairlocks performs reasonably well when compared with spinlocks, especially at higher contention when the increased memory latency of the second quad begins to show up (threads 5-8). Further, Figure 8 shows the fairlocks/spinlocks performance ratio measured under different levels of contention. Figure 8 shows that, on average, fairlocks has 64% of spinlock throughput.

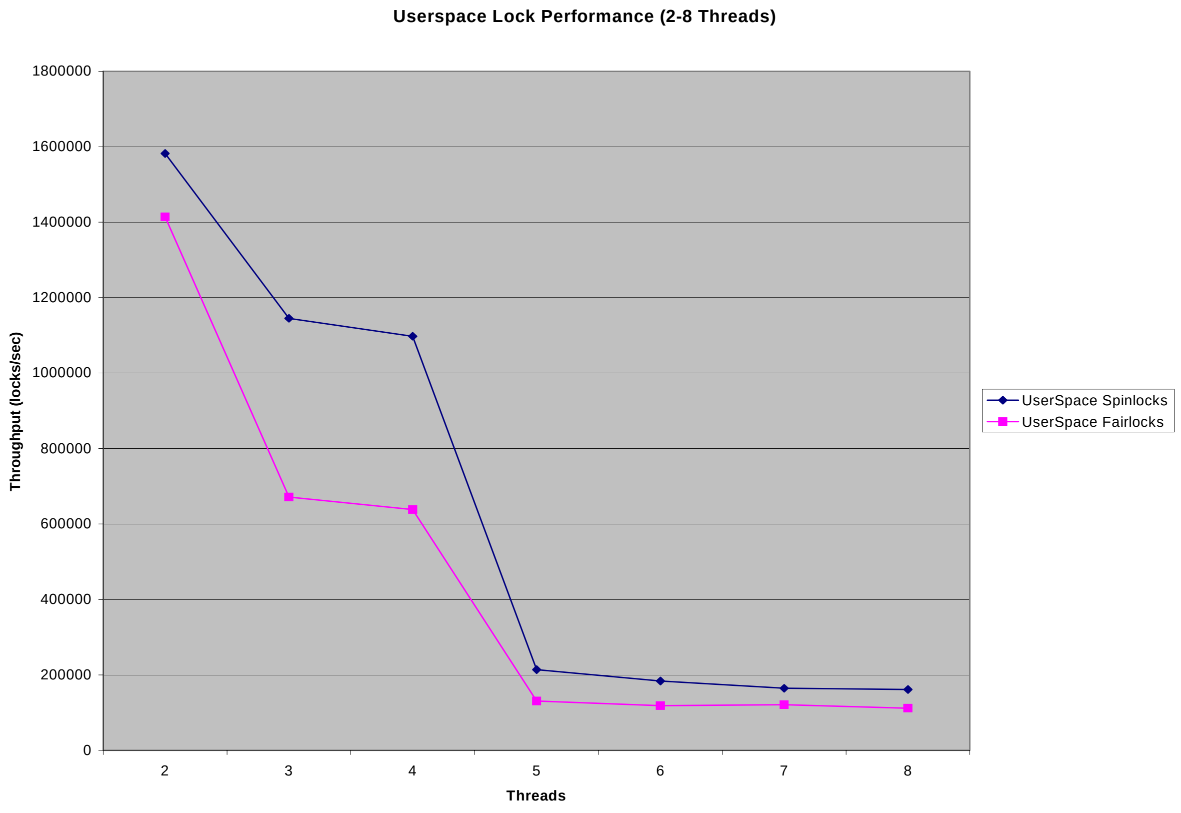

Figure 7: UserLock Performance

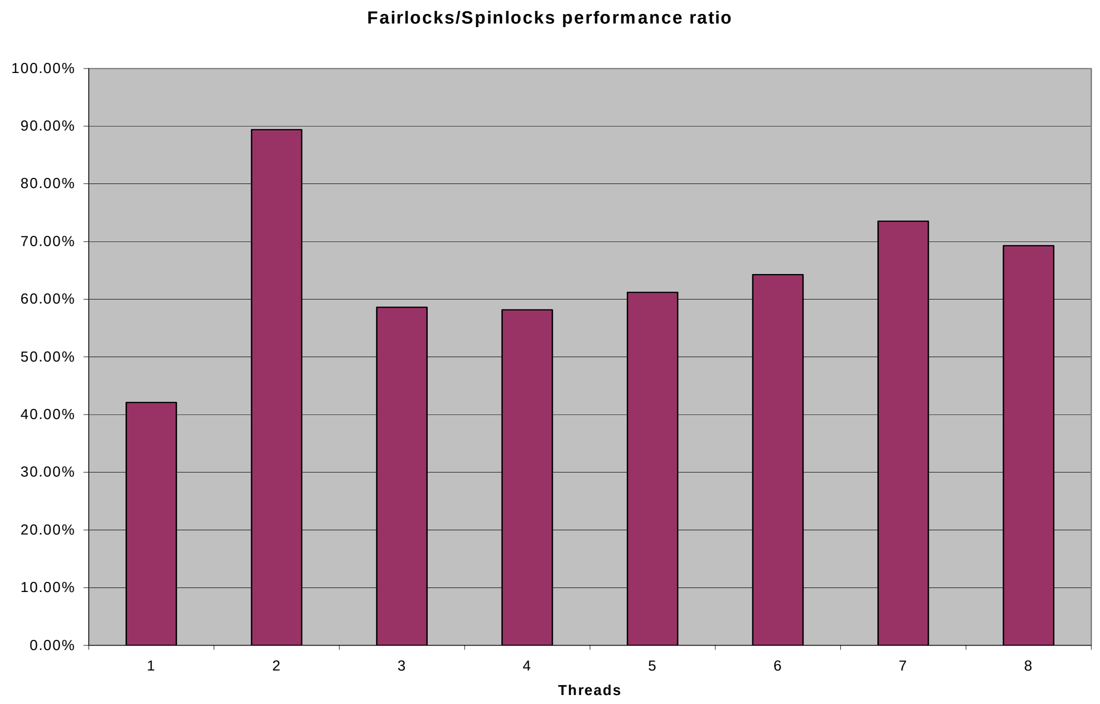

Figure 8: Fairlocks/Spinlocks Performance Ratio

6.3 *System Performance:*

In order to measure the real world impact of fairlocks with respect to *throughput and fairness*, we made the *runqueue_lock* spinlock in the Linux 2.4.12 kernel use fairlocks. Then, we used the Reflex benchmark to compare the performance of fairlocks with spinlocks in a live Linux kernel. Reflex is a micro-benchmark designed to exercise the *schedule()* and *reschedule()* functions in a controlled manner. The program creates a number of threads, which are grouped into active sets. All threads in one active set pass a token around using blocking reads and writes on message pipes. After receiving the token, the program performs several rounds of computation before passing the token to its neighbor in the active set and blocks on a read for the same token. The explicit yield invokes schedule(), whereas the blocking yield invokes the reschedule() idle function. Further, as the number of active threads increases, so does the contention for the global runqueue_lock. For the simulation, we used zero computation rounds, as these rounds precisely measure the overhead of scheduler and the contention details over the runqueue_lock of the scheduler. We did the performance evaluation by running Reflex on a two quad, 8-way IA32 IBM NUMA-Q system and the locks (spinlock, fairlocks) were tested for its throughput and fairness (using a lock instrumentation tool called Lockmeter [*lockmeter*]). The throughput and fairness results of Reflex are as follows:

Reflex – Fairness:

In this test, we observed the fairness of both the locks (such as lock mean wait time, maximum waiting time, contention, spin time, etc.) while running the Reflex benchmark.

This test also provides insight into how lock usage is affected by fairlocks. We monitored the spinlock contention details, such as maximum waiting time, average utilization and number of lock acquisitions, using lockmeter [Lockmeter] during the re-run of Reflex benchmarks. We made minor modifications to the lockmeter for it to work with fairlocks to measure the same parameters. We ran the Reflex benchmark with the lockmeter-instrumented kernels. Table 4 shows the lock usage information for the running of the Reflex benchmark with a lockmeter-instrumented kernel.

	Utilization	Contention	Mean Hold	Max Hold	Mean Wait	Max Wait	Total	Nowait	Spin	Reject
spinlocks	31.70%	64.70%	66	2720	372	9287000	3206567	35.30%	64.70%	0%
fairlocks	27.40%	54.60%	66	1289	261	4515	2773485	45.40%	54.60%	0%

Table 4: runqueue_lock Usage for Spinlocks and Fairlocks

Table 4 shows that for the allotted time of the benchmark, the lock was acquired equally for both the locks. *However, the maximum wait time of spinlock was two thousand times larger than that of fairlocks.* Thus, it is evident that spinlocks causes lock starvation among CPUs in a real world scenario, as maximum wait time is much larger than the maximum hold time. The starvation is eliminated by fairlocks, where the maximum wait time of a CPU on the lock was a little more than three times the maximum hold time. This is a very clear illustration of how fairlocks is able to avoid starvation with very little overhead.

Reflex – Throughput:

Figure 9 summarizes the performance of both the locks with respect to throughput. The figure shows that the results when fairlocks is used are statistically equivalent, if not occasionally better, than those with spinlocks. Though fairlocks has more overhead than spinlocks, the comparable performance can be due to the starvation caused by the memory latency between quads, which can hamper the performance of spinlocks.

7. Conclusion

In this paper, we studied the problem of lock starvation among CPUs in some architectures with variable memory latencies. We also studied the currently available locking schemes, observed their shortcomings, and proposed a new locking scheme, *fairlocks*, which eliminates lock starvation among CPUs with minimal overhead under low contention. The fairness of the algorithm is substantiated both theoretically and through our experimental studies. Thus, in cases where reducing lock contention is impractical, our proposed locking scheme can be used as a locking solution because it eliminates lock starvation and performs better than the currently available fair locking schemes.

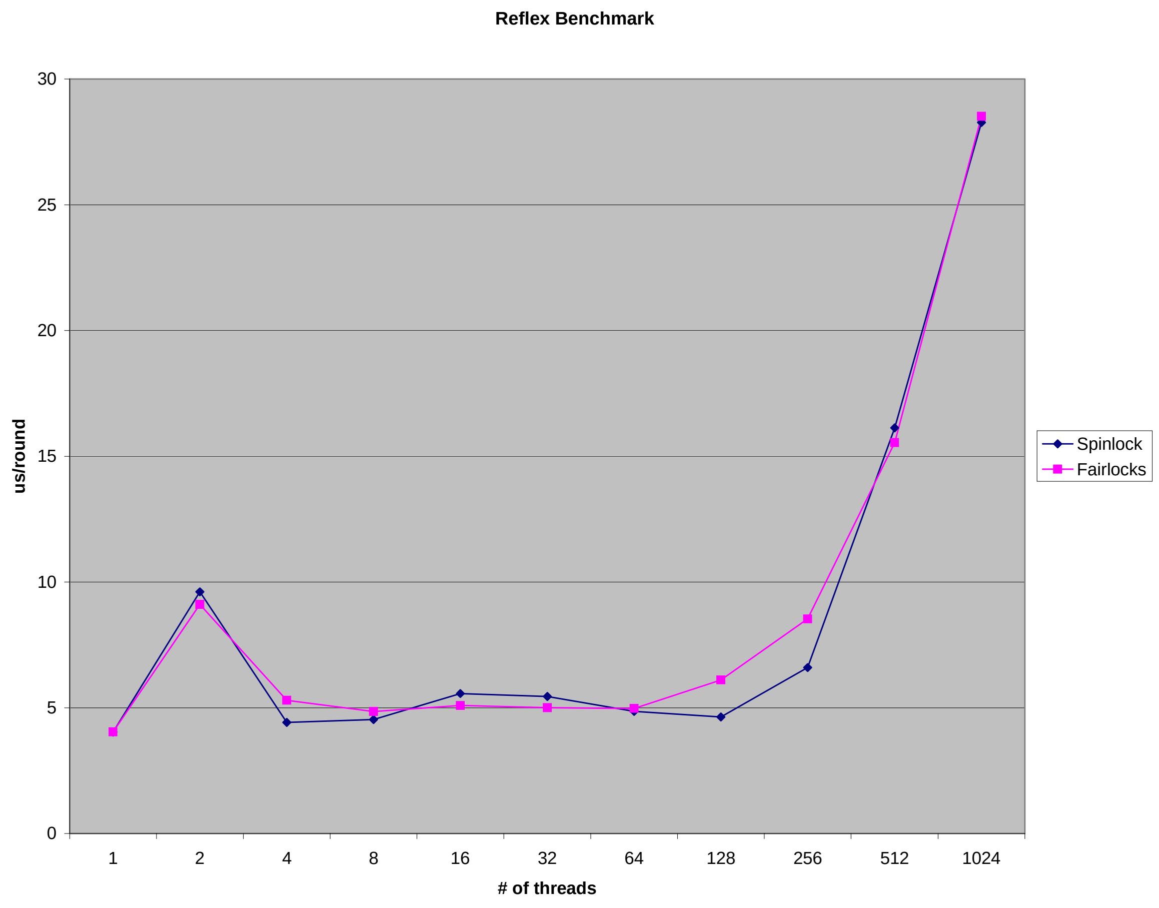

Figure 9: Lock Performance Measured with Reflex

References

[NUMA-Q] T. Lovett and R. Clapp. StiNG. "A CC-NUMA computer system for the commercial marketplace." In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996, pages 308-317.

[CMP] K. Onlukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. "The case for a single-chip multiprocessor." In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* 1996, pages 2-11.

[DASH] D. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. "The Stanford DASH multiprocessor." In *IEEE Computer*, March 1992, pp 63-79.

[FLASH] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo,D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, October 1994.

[RW-Spin] Peter S. Magnusson, Anders Landin, and Erik Hagersten. "Queue Locks on Cache Coherent Multiprocessors." In *8th International Parallel Processing Symposium (IPPS)*, 1994.

[TICKET,QLOCK] J. Mellor-Crummey, M. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." *TOCS* 9(1): 21-65 (1991)

[beng-hong lim] Beng-Hong Lim, Anant Agarwal. "Reactive Synchronization Algorithms for Multiprocessors." *ASPLOS* 1994: 25-35

[MCS] J. M. Mellor-Crummey and M. L. Scott. "Scalable reader-writer synchronization for shared-memory multiprocessors." *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (3rd PPOPP'91), SIGPLAN Notices.* April 1991, pp 106-113

[Hseih & Weihl] Wilson C. Hsieh and William E. Weihl. “Scalable reader-writer locks for parallel systems.” *MIT Laboratory for Computer Science technical report MIT/LCS/TR-521*, Cambridge, MA. 1991.
[PDCS’98 Paper] P. E. McKenney and J. D. Slingwine. “Read-copy update: using execution history to solve concurrency problems.” *Parallel and Distributed Computing and Systems*, October 1998.
[PDCS’99 Paper] P. E. McKenney. “Practical performance estimation on shared-memory multiprocessors.” *Parallel and Distributed Computing and Systems*, November 1999.
[JLOCK] – IBM Technical Report
[Lockmeter] Kernel Spinlock Metering for Linux, *URL:* http://oss.sgi.com/projects/lockmeter/

Trademarks