RH Lock: A Scalable Hierarchical Spin Lock

Zoran Radović and Erik Hagersten

Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
E-mail: {zoranr,eh}@it.uu.se

Abstract

Scalable architectures with non-uniform memory access time (NUMAs) have gained increased popularity in recent years. The increased scalability have increased the demand for scalable lock implementations, such as the queue-based locks of Mellor-Crummey and Scott (MCS lock), and of Craig, Landin and Hagersten (CLH lock).

This paper demonstrates that the first-come first-served nature of queue-based locks make them less suitable for non-uniform communication architectures (NUCAs), for example NUMAs built from a few large nodes. In contrast, the simpler test-and-set locks gives a unfair advantage to neighboring processors when a lock is released, which will create a fast lock handover time as well as create more locality for the data accessed in the critical region.

We also propose the new RH lock that explores the NUCA architectures by creating a controlled unfairness in combination with a much reduced traffic compared with the test-and-set locks. A critical section guarded by the RH lock is shown to take less than half the time to execute compared with the same critical section guarded by any other lock. We also investigate the effectiveness of our new lock on a set of real SPLASH-2 applications. For example, execution time for Raytrace with 30 processors can be improved between 1.83 and 5.70 times by using the RH locks instead of any other tested locks.

1 Introduction

There are plenty of examples in academia and industry of shared-memory architectures with a non-uniform memory access time to the shared memory (NUMA). Most of the NUMA architectures, but not all, also have a non-uniform communication architectures (NUCA), i.e., the access time from a processor to the other processor's caches varies greatly depending on their placement. In particular, node-based NUCAs, where a group of processors have a much shorter access time to each other's caches than to the other caches, are common.

Recently, technology trends have made it attractive to run more than one thread on a chip, using either the Chip Multiprocessor (CMP) and/or the Simultaneous Multithreading (SMT) approach. Larger servers, built from several of those chips, can therefore be expected to be NUCA architectures, since collocated threads will most likely share an on-chip cache at some level [3]. In our opinion, there are strong indications that many important architectures in the future will have a non-uniform access time to each other's caches, as well as to the shared memory.

NUMA optimizations have attracted much attention in the past. The migration and replication of data in NUMA systems have demonstrated a great performance improvement in many applications [11], [20]. However, many of today's applications show a large fraction of cache-to-cache misses [4], which is why attention should also be given to the NUCA nature of the system.

The scalability of a shared-memory application is often limited by contention for some critical section, often accessing some shared data, guarded by mutual exclusion locks. The simpler, and most widely used, test-and-set locks implementations perform worse at high contention, i.e., the more important the critical section gets, the worse the lock algorithm performs. This is mostly due to the vast amount of traffic generated at the lock handover.

An application can often be rewritten to decrease the contention. This could, however, be a complicated task. More advanced queue-based locks have been proposed that have a slightly worse performance at light lock contention, but a much better performance at high lock contention because less traffic is generated [19], [6], [17]. Furthermore, the queue-based locks maintain a first-come first-served order between the contenders. While queue-based locks have shown low traffic and great scalability on many architectures, their first-come first-served property is less desirable on a NUCA architecture.

Three properties determine the average time between two threads entering the contended critical section: lock handover time, traffic generated by the lock, and the data locality created by the lock algorithm.

We have noticed that the test-and-set locks give an unfair advantage to processors in the NUCA node where the lock last was held. This will create more node locality and will partly make up for the more traffic generated by the test-and-set locks. The increased node locality will improve on the lock handover time, but also on the locality of the work in the critical section.

The goal of this work is to create a lock that minimizes the global traffic generated at lock handover, and maximizes the node locality of NUCA architectures.

The remainder of this paper is organized as follows. Section 2 gives an introduction to several machines with NUCA architectures. Background and related work is presented in section 3. The key idea behind the RH lock is given in section 4, and section 5 presents the RH lock algorithm. In section 6 we present performance results obtained on a 32-processor Sun WildFire machine (not to be confused with the Compaq product of the same name). Finally, we conclude in section 7.

2 Non-Uniform Communication Architectures

Many large-scale shared-memory architectures have non-uniform access time to the shared memory (NUMA). In order to make a key difference, the non-uniformity should be substantial, let's say at least a factor two between best case and worst unloaded case. Most of the NUMA architectures also have a substantial difference in latency for cache-to-cache transfer—a *Non-Uniform Communication Architecture* (NUCA). A NUCA is an architecture where the unloaded latency for a processor accessing data recently modified another processor differs at least a factor two depending on where that CPU is located.

DASH was the first NUCA architecture [14]. Each DASH node consists of four processors connected by a snooping bus. A cache-to-cache transfer from a cache in a remote node is 4.5 times slower than a transfer from a cache in the same node. We call this the *NUCA ratio*. Sequent's NUMA-Q has a similar topology, but its NUCA ratio is closer to ten [16]. Both DASH and NUMA-Q have a remote access cache (RAC) located in each node that simplifies the implementation of the node-local cache-to-cache transfer.

Sun's WildFire system can have up to four nodes with 28 processors each, totaling 112 processors [11]. Parts of each node's memory can be turned into a RAC a using technique called Coherent Memory Replication (CMR). Accesses to data allocated in a CMR cache have a NUCA ratio of about six, while accesses to other data only have a minor latency difference between node-local and remote cache-to-cache transfers.

Compaq's DS-320 (which was also code-named WildFire) can connect up to four nodes, each with four processors sharing a common DTAG and directory controller [8]. Its NUCA ratio is roughly 3.5.

The IBM Regatta system is built from the new Power4 chips [25]. Each Power4 contains two processor cores sharing one common L2 cache. Four such chips are grouped into a "supernode" with eight processor cores sharing one L3 cache. A fully configured system consists of four such supernodes, i.e., up to 32 processor cores. There is sparse information about Regatta's different latencies, but its NUCA ratio can be expected to be very high.

Future microprocessors can be expected to run many more threads on a chip by a combination of CMP and SMT technology. This can already be seen in the Pentium 4's Hyper-threading and the IBM Power4's dual CMP processors on a chip. The Piranha CMP proposal expects 8 CMP threads to run on each chip [3]. Larger systems, built from many such CMPs, are expected to have a NUCA ration of between six and ten depending on the technology chosen.

Not all architectures are NUMAs nor NUCAs. The recent SunFire 15k architecture can have up to 18 nodes, each with four processors, memory and directory controllers [5]. The nodes are connected by a fast backplane. It has a flavor of both NUMA and NUCA. However, both its NUMA ratio and NUCA ratio is well below two. The SGI Origin 2000 is a NUMA architecture with a NUMA ratio of around three for reasonable sized systems [13]. However, it does not efficiently support cache-to-cache transfers between adjacent processors and has a NUCA ratio below two.

3 Background and Related Work

Ideally, synchronization primitives should provide high performance under both high and low contention without requiring substantial programmer effort. Mutual exclusion (lock-unlock) operations can be implemented in a variety of different ways, including: (1) atomic memory primitives (e.g., `test_and_set` and `fetch_and_add`), (2) non-atomic memory primitives (e.g., load-linked/store-conditional), and (3) explicit hardware lock-unlock primitives (e.g., CRAY Xmp lock registers, DASH's lock-unlock operations on directory entries, or queue-on-lock-bit, QOLB). We will concentrate on implementing locks with only atomic primitives. Explicit hardware primitives are currently not popular on modern bus-based machines.

The five synchronization primitives we discuss and directly compare in this paper are test-and-`test_and_set` (abbreviated TATAS), test-and-`test_and_set` with exponential backoff (abbreviated TATAS_EXP), MCS locks, CLH locks, and RH locks (our new hierarchical spin lock). We also present a short introduction to alternative synchronization approaches; reactive synchronization and an aggressive queue-on-lock-bit (QOLB) hardware scheme.

3.1 Atomic Primitives

In this paper we make reference to three atomic operations. `Test_and_set` (*address*) atomically writes a nonzero value to the *address* memory location and returns its original contents. A nonzero value for the lock represents the locked condition, while a zero value means that the lock is free. `Swap` (*address*, *value*) atomically writes a *value* to the *address* memory location and returns its original contents. `Compare_and_swap` (*address*, *expected_value*, *new_value*) atomically checks the contents of a memory location *address* to see if it matches an *expected_value* and, if so, replaces it with a *new_value*.

`Compare_and_swap` first appeared in the IBM 370 instruction set. `Test_and_set`, `swap` and `compare_and_swap` are provided by Sparc V9 — our target architecture for this paper.

3.2 Simple Lock Algorithms

Traditionally, the simple synchronization algorithms tend to be fast when there is little or no contention for the lock, while more sophisticated algorithms usually have a higher cost for low-contention case, but, on the other hand, they handle contention much better. In this section we describe two still very commonly used *busy-wait* algorithms: TATAS and TATAS_EXP.

It was Rudolph and Segall who first proposed an extension to ordinary `test_and_set` (this was the sole synchronization primitive available on numerous early systems, such as the IBM 360 series) that performs a read of the lock before attempting the actual atomic `test_and_set` operation [21]. A typical TATAS algorithm is shown below:

```
    typedef unsigned long bool;
    typedef volatile bool tatas_lock;

 1: void tatas_acquire(tatas_lock *L)
 2: {
 3:   if (tas(L)) {
 4:     do {
 5:       if (*L)
 6:         continue;
 7:     } while (tas(L));
 8:   }
 9: }
10:
11: void tatas_release(tatas_lock *L)
12: {
13:   *L = 0;
14: }
```

This is the most basic busy-wait algorithm in which a process (or thread) repeatedly attempts to change a lock-value *L* from `true` to `false`, using an atomic hardware primitive (for example, in the Sparc V9 instruction set, this is typically done with load-store unsigned byte (LDSTUB) instruction). Traditional `test_and_set`-based spin locks are vulnerable to memory and interconnect contention, and do not scale well to large machines. This contention can be reduced by polling (busy-wait code) with ordinary load operations to avoid generating expensive stores to potentially shared location (lines 4–6 in the code above). Furthermore, the burst of refill traffic whenever lock is released can be reduced by using the Ethernet-style exponential backoff algorithm in which after a failure to obtain the lock a requester waits for successively longer periods of time before trying to issue another lock operation [2], [19]. This is the idea behind the TATAS_EXP lock. The `acquire` function of one typical TATAS_EXP implementation is shown below:

```
 1: void tatas_exp_acquire(tatas_lock *L)
 2: {
 3:   int b = BACKOFF_BASE, i;
 4:
 5:   if (tas(L)) {
 6:     do {
 7:       for (i = b; i; i--) ; // delay
 8:       b = min(b * BACKOFF_FACTOR, BACKOFF_CAP);
 9:       if (*L)
10:         continue;
11:     } while (tas(L));
12:   }
13: }
```

Type definitions and release code are the same as in the TATAS example. Parameters `BACKOFF_BASE`, `BACKOFF_FACTOR`, and `BACKOFF_CAP` must be tuned by trial and error for each individual architecture. We use the following settings in our experiments, which are identical to the settings used by Scott and Scherer on the same platform [22]:

`BACKOFF_BASE`	625
`BACKOFF_FACTOR`	2
`BACKOFF_CAP`	2,500

3.3 Queue-Based Locks

Even with exponential backoff, TATAS locks still induce significant contention (performance results using backoff with a real `test_and_set` instruction on older machines can be found in the literature [10], [19]). On a standard symmetric multiprocessor (SMP) with uniform access times between all processor's caches in the node, queue-based locks may eliminate these problems by letting each process spin on a different local memory location. The first proposal for a distributed, queue-based locking scheme in hardware was made by Goodman, Vernon, and Woest [9] (see section 3.4 for more details). Several researchers have independently proposed locking primitives that incorporate both local spinning and queue-based locking in software [1], [10], [19]. The locking primitive called MCS is one of the first software queue-based lock implementations, originally inspired by the QOSB [9] hardware primitive proposed for the cache controllers of the Wisconsin Multicube in the late 1980s. The MSC lock was developed by Mellor-Crummey and Scott [19]. During the acquire request, the MSC lock inserts requesters for a held lock into a software queue using the atomic operations such as `swap` and `compare_and_swap`. Mellor-Crummey and Scott also describe another version of the MSC lock which only requires the `swap` operation. Fairness in that case is no longer guaranteed, and the implementation is slightly more complex.

Magnusson, Landin, and Hagersten proposed two software queue-based locking primitives about three years after MCS, namely LH and M [17] (Craig independently developed a lock identical to LH [6], in this paper we will refer to this lock as the CLH lock). The CLH lock requires one fewer remote access to transfer a lock than does MCS, and will usually outperform MCS when high lock contention exists [22]. The CLH lock achieves this behavior at the expense of increased latency to acquire an uncontested lock. The M lock achieves the more efficient lock transfer without increased uncontested lock access latency, at the expense of significant additional complexity in the lock algorithm.

3.4 Alternative Approaches

The fact that some synchronization algorithms perform well under low-contention periods and other under high-contention periods is the basic idea behind "reactive synchronization" presented by Lim and Agarwal a couple of years after first proposals for queue-based locks [15]. Reactive synchronization algorithm will dynamically swith among several software lock implementations. Typically, spin-locks (e.g., TATAS_EXP) are used during the low-contention phase, and queue-based locks (e.g., MCS) are used during

the high-contention phase [12]. The goal of reactive synchronization is to achieve both low latency lock access and efficient lock-handoff at low cost.

Very aggressive hardware support for locks have been proposed by Goodman, Vernon, and Woest [9]. They introduce the queue-on-lock-bit primitive (QOLB, originally called QOSB), which was the first proposal for a distributed, queue-based locking scheme. In this scheme, a distributed linked list of nodes waiting on a lock is maintained entirely in hardware, where the releaser grants the lock to the first waiting node without affecting others. Furthermore, QOLB prevents unnecessary network traffic or interference with the lock holder by letting the waiting processors spin locally on a "shadow" copy of the lock address. Effective collocation is possible because processors that are requesting a lock spin on the same address as that of the lock, without evicting or downgrading the lock holder's copy. Thus, this hardware scheme may reduce the lock handover time as well as the interference of lock traffic with data access and coherence traffic.

Unfortunately, QOLB requires additional hardware support. Most synchronization primitives that we discussed in previous sections can be implemented entirely in software, requiring only an atomic memory operation available in the majority of modern processors. Detailed evaluation of all hardware requirements for QOLB is presented by Kägi, Burger, and Goodman [12].

4 Key Idea Behind RH Lock

The goal for the RH lock is to create a lock that minimizes the global traffic generated at lock-handover, and maximizes the node locality of NUCA architectures. Queue-based locks implement a first-come first-served fairness, which is less desirable on a NUCA machine because of the potentially huge percentage of lock node-handoffs, in other words, there is a risk that a contended lock might "jump" back and forth between the nodes, creating enormous amount of traffic.

In our scheme, every node contains a copy of the lock. Our total lock storage cost is thus $N \times$ `sizeof`(lock), where N is the number of nodes in the system. Initially, we can for example decide to logically place a lock in node 0 (mark the lock-value as `FREE`, meaning that both threads from the local node or from another nodes are allowed to acquire the lock). At most one node may have this local copy of the lock in state `FREE`. In that case, all other nodes (node 1, node 2, ..., node N - 1) are going to "see" a `REMOTE` tag if they try to acquire their local copy of the lock.

One way to increase locality is to handover the lock to another thread running in the same node (we will later refer to this operation as marking the lock-value with `L_FREE` tag). This will not only cut down on the lock handover time, but will also create locality in the critical section work, since its data structures will already reside in the node.

One way to cut down on lock traffic is to make sure that only one thread per node will try to retrieve a lock which is currently not owned by a thread in the node.

Even if the first-come first-served policy may be a too strong requirement for a lock, it must guarantee some fairness, i.e., make sure that other nodes will eventually get the lock even if there are always local requests for the lock.

5 The RH Lock

During the design phase of our RH lock we paid attention to several general performance goals for locks, given by Culler et al. [7], page 343:

- *Low latency*. If a lock is free and no other processors are trying to acquire it at the same time, processor should be able to acquire it with low latency.
- *Low traffic*. If many or all processors try to acquire a lock at the same time, they should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible.
- *Scalability*. Neither latency nor traffic should scale quickly with the number of processors used.
- *Low storage cost*. The information needed for a lock should be small and should not scale quickly with the number of processors.
- *Fairness*. Ideally, processors should acquire a lock in the same order as their requests are issued. At the least, starvation or substantial unfairness should be avoided. Since starvation is usually unlikely, the importance of fairness must be traded off with its impact on performance.

Well, in fact, we paid attention to only first four goals and ignored the last one (with the exception of goals for starvation). We also paid attention to the data locality created by our lock algorithm, in other words, our additional goal is to maximize the node locality of NUCA architectures.

Our hierarchical spin lock algorithm (called the RH lock, after our initials) is shown in Figure 1 and Figure 2.[1] We use the following settings and definitions:

`BACKOFF_BASE`	625
`BACKOFF_FACTOR`	2
`BACKOFF_CAP`	2,500
`REMOTE_BACKOFF_BASE`	625
`REMOTE_BACKOFF_CAP`	20,000
`FREE`	max number of threads
`REMOTE`	`FREE` + 1
`L_FREE`	`FREE` + 2

The following atomic operations are used in our current implementation: `test_and_set`, `swap` and `compare_and_swap`, which are all available in the Sparc V9 instruction set.

`my_tid` is the thread identification number (0, 1, 2, ..., max number of threads – 1), and `my_node_id` is the node number in which thread is placed (0, 1, 2, ..., max number of nodes – 1). Both `my_tid` and `my_node_id` must

[1] The RH lock algorithm is currently optimized for small amount of nodes or chip multiprocessors.

```
   typedef volatile unsigned long rh_lock;

 1: void rh_acquire(rh_lock *L)
 2: {
 3:   unsigned long tmp;
 4:
 5:   tmp = swap(L, my_tid);
 6:   if (tmp == L_FREE || tmp == FREE)
 7:     return;
 8:   if (tmp == REMOTE) {
 9:     rh_acquire_remote_lock(L);
10:     return;
11:   }
12:   rh_acquire_slowpath(L);
13: }

 1: void rh_acquire_slowpath(rh_lock *L)
 2: {
 3:   unsigned long tmp;
 4:   int b = BACKOFF_BASE, i;
 5:
 6:   if ((random() % FAIR_FACTOR) == 0)
 7:     be_fare = TRUE;
 8:   else
 9:     be_fare = FALSE;
10:
11:   while (1) {
12:     for (i = b; i; i--) ; // delay
13:     b = min(b * BACKOFF_FACTOR, BACKOFF_CAP);
14:     if (*L < FREE)
15:       continue;
16:     tmp = swap(L, my_tid);
17:     if (tmp == L_FREE || tmp == FREE)
18:       break;
19:     if (tmp == REMOTE) {
20:       rh_acquire_remote_lock(L);
21:       break;
22:     }
23:   }
24: }

 1: void rh_acquire_remote_lock(rh_lock *L)
 2: {
 3:   int b = REMOTE_BACKOFF_BASE, i;
 4:
 5:   L = get_remote_lock_addr(L, my_node_id);
 6:
 7:   while (1) {
 8:     if (cas(L, FREE, REMOTE) == FREE)
 9:       break;
10:     for (i = b; i; i--) ; // delay
11:     b = min(b * BACKOFF_FACTOR, REMOTE_BACKOFF_CAP);
12:   }
13: }
```

Figure 1: RH lock acquire code.

```
 1: void rh_release(rh_lock *L)
 2: {
 3:   if (be_fare)
 4:     *L = FREE;
 5:   else {
 6:     if (cas(L, my_tid, FREE) != my_tid)
 7:       *L = L_FREE;
 8:   }
 9: }
```

Figure 2: RH lock release code.

be thread-private values, and preferably efficiently accessible from `rh_acquire` and `rh_release` functions.

To achieve controlled unfairness we use a thread-private `be_fare` variable that initially is `TRUE`. The `random` function (line 6 in the `rh_acquire_slowpath` function) uses a nonlinear feedback random-number generator. It returns pseudo-random numbers in the range from 0 to $2^{31} - 1$. If `FAIR_FACTOR` is equal to one, the RH lock will behave "as much fair as it can," i.e., it will be comparable to the ordinary TATAS_EXP lock because `rh_release` function will always mark lock-data with a `FREE` tag.

6 Performance Evaluation

Most experiments in this paper are performed on a Sun Enterprise E6000 SMP [24]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (*lmbench* latency [18]) and a total bandwidth of 2.7 Gbyte/s. Each processor has a 16 kbyte on-chip instruction cache, a 16 kbyte on-chip data cache, and a 4 Mbyte second-level off-chip data cache.

The hardware DSM numbers have been measured on a 2-node Sun WildFire built from two E6000 nodes connected through a hardware-coherent interface with a raw bandwidth of 800 Mbyte/s in each direction [11], [20]. The Sun WildFire access time to local memory is the same as above, 330 ns, while accessing data located in the other E6000 node takes about 1700 ns (lmbench latency). Accesses to data allocated in a CMR cache have a NUCA ratio of about six, while accesses to other data only have a minor latency difference between node-local and remote cache-to-cache transfers. The E6000 and the WildFire DSM system are both running a slightly modified version of the Solaris 2.6 operating system.

We have implemented the traditional TATAS lock and the RH lock using the `test_and_set`, `swap` and `compare_and_swap` operations available in the Sparc V9 instruction set. The source code for TATAS_EXP, CLH, and MCS lock is written by Scott and Scherer [22], and is available for download:

`ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/PPoPP_01_trylocks.tar.gz`

The entire experimentation framework is compiled with GNU's gcc-3.0.4, optimization level -O3. The TATAS_EXP lock was previously tuned for a Sun Enterprise E6000 machine by Scott and Scherer [22]. We use identical values in our experiments.

6.1 Traditional Microbenchmark

The traditional microbenchmark we use in this paper is a slightly modified code used by Scott and Scherer in [22] on the same architecture—Sun WildFire prototype SMP cluster. The code consists of a tight loop containing a single aquire/release lock operations, plus some minimal critical work in form of code for gathering statistics. Also, after release operation, we make sure that the same thread will not acquire the lock immediately. Thus, our lock-handoff val-

ues are nearly equal to 100% for all tested lock algorithms. Machines used for tests were otherwise unloaded.

6.1.1 Single-Processor Results

By running the traditional microbenchmark on a single processor and then subtracting the loop overhead, we can obtain an estimate of lock overhead in the absence of contention. Results on our Sun Enterprise E6000 SMP are as follows:

pthread_mutex	267 *ns*
TATAS	97 *ns*
TATAS_EXP	97 *ns*
MCS	202 *ns*
CLH	137 *ns*
RH	121 *ns*

Unsurprisingly, TATAS and TATAS_EXP are fastest acquire/release operations for this simple test without contention. We observe that our *low latency* design goal for RH lock is within the reasonable magnitudes. In this experiment, we also report numbers for the native implementation of *pthread_mutex* locks. The fast paths of all other locks are in-lined with the `static __inline__` GCC-construct, only pthread_mutex is a "real" function call, and that is why that number is somewhat higher than others.

6.1.2 Parallel Performance

In this section we present the parallel performance results for both single-node Sun Enterprise E6000 and for a 2-node Sun WildFire system. Given numbers are derived from the worst case execution time runs.

Single-node Sun Enterprise E6000

The microbenchmark iteration time for parallel execution on Sun Enterprise E6000 is shown in Figure 3. As expected, the RH lock performs similarly to TATAS_EXP. If we look at the queue-based locks, the CLH lock has slightly better performance than the MCS lock, but both of them are outperformed by TATAS_EXP and the RH lock for almost all cases. The CLH and the RH lock demonstrates stable and predictable performance.

2-node Sun WildFire

The microbenchmark iteration time for parallel execution on a 2-node Sun WildFire is shown in Figure 4. In this study, we simultaneously bind first half of the running threads to cabinet number one, and the other half to cabinet number two. `FAIR_FACTOR` is equal to one (we are as much fair as we can be). The RH lock outperforms all other tested locks for all runs with more than two threads, and is comparable to other locks for one (no contention) and two threads. In the two-thread case the node-handoff is 100%, i.e., all `rh_acquire` accesses will result in the `rh_acquire_remote` calls. A critical section guarded by the RH lock takes less than half the time to execute compared with the same critical section guarded by any other lock.

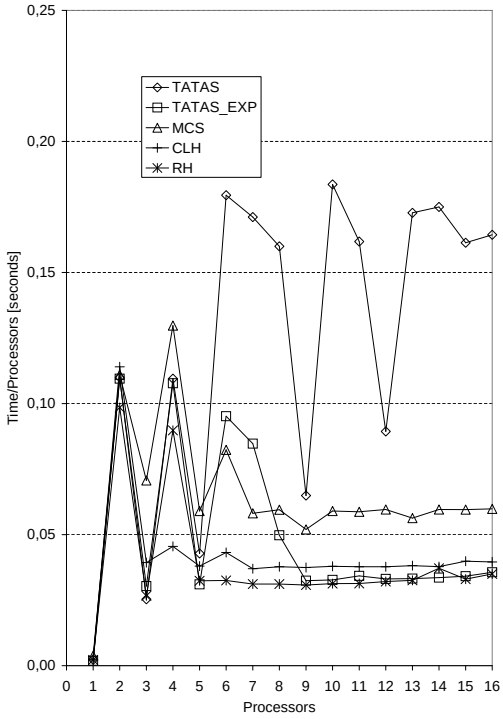

Figure 3: Traditional microbenchmark iteration time for a single-node Sun Enterprise E6000.

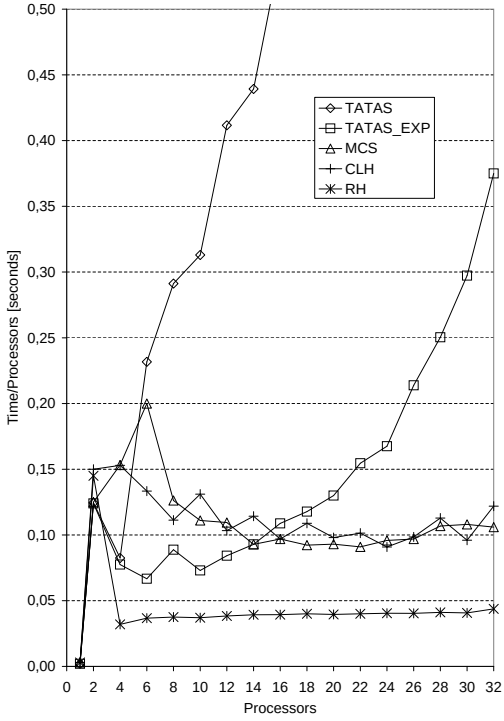

Figure 4: Traditional microbenchmark iteration time for a 2-node Sun WildFire system.

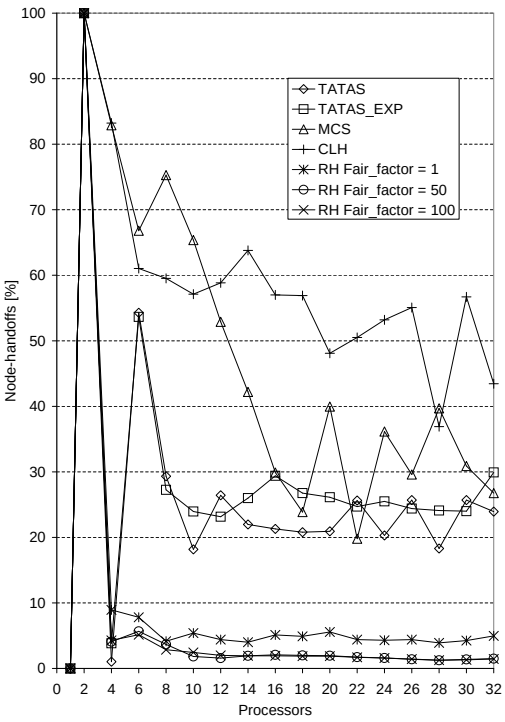

Figure 5: Traditional microbenchmark locality study for a 2-node Sun WildFire system.

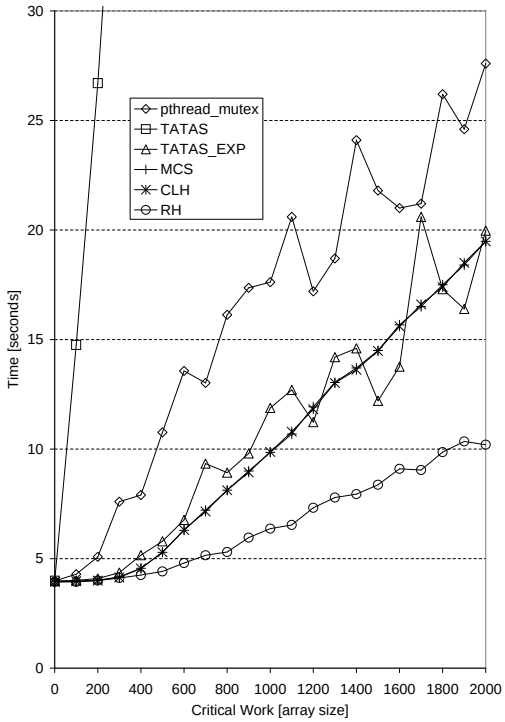

Figure 6: New microbenchmark iteration time for a 2-node Sun WildFire system, 28-processor runs.

Figure 5 shows the ratio of node-handoff for each lock type, i.e., how likely is a lock to migrate between nodes each time it is acquired. The RH lock consistently show low node-handoff numbers for all the three settings of the FAIR_FACTOR. The simple spin locks also show a fairly low node-handoff, which could be expected since local processors can aquire a released lock much faster than a remote processors. However, the queue-based locks shows an unnatural behavior. It can be expected that the fair queue locks should show a node-handoff equal to $(N/2)/(N-1)$, since $N/2$ of the processors reside in the other node and we do not allow the same processor to aquire the lock twice in a row. However, the simplistic microbenchmark that we, as well as most other lock studies use, make processors in the same node more likely to queue up after each other, why the lock ratio is substantially lower than expected. This is especially true for MCS which is taking unfair advantage of the test setup.

6.2 New Microbenchmark

The unnatural node handover behavior of the traditional lock benchmark led us to a new lock benchmark that we think reflects the expected behavior of a real application better. In the new microbenchmark, the number of processors is kept constant. They each perform some amount of non-critical work between trying to aquire the lock, which consists of one static delay and one random delay of similar sizes. Initially, the length of the non-critical work is chosen such that there is little contention for the critical section and all lock algorithms perform the same. More contention is modeled by increasing the number of elements of a shared vector that are modified before the lock is released. The pseudocode of the new benchmark is shown below:

```
shared int cs_work[MAX_CRITICAL_WORK];
shared int iterations;

 1: for (i = 0; i < iterations; i++) {
 2:   ACQUIRE(lock);
 3:   {
 4:     int i;
 5:     for (i = 0; i < critical_work; i++)
 6:       cs_work[i]++;
 7:   }
 8:   RELEASE(lock);
 9:   {
10:     int non_cs_work[MAX_NON_CRITICAL_WORK];
11:     int i, j;
12:     j = random() % non_critical_work;
13:     for (i = 0; i < non_critical_work; i++)
14:       non_cs_work[i]++;
15:     for (i = 0; i < j; i++)
16:       non_cs_work[i]++;
17:   }
18: }
```

Figure 6 shows that the two queue-based locks perform almost identical for the new benchmark and Figure 7 show their node handover to be close to the expected values of 50%. The simple spin locks still perform unpredictable which is tied to their unpredictable node-handover. The RH lock performs better the more contention there is, which can be explained by its decreasing amount of node handover. This is exactly the behavior we want in a lock: the more contention there is, the better it should perform.

Program	pthread_mutex	TATAS	TATAS_EXP	MCS	CLH	RH
Barnes	1.44 (0.048)	1.54 (0.068)	1.35 (0.075)	1.28 (0.064)	1.44 (0.108)	1.44 (0.009)
Cholesky	2.11 (0.019)	2.40 (0.033)	2.44 (0.044)	2.18 (0.047)	2.26 (0.044)	2.38 (0.013)
FMM	3.55 (0.028)	4.47 (0.166)	3.70 (0.082)	–	–	3.59 (0.066)
Radiosity	6.38 (2.264)	1.26 (0.018)	1.58 (0.127)	–	–	1.48 (0.049)
Raytrace	2.86 (0.128)	2.27 (0.097)	1.56 (0.079)	1.14 (0.207)	1.66 (0.522)	0.71 (0.009)
Volrend	1.45 (0.026)	1.39 (0.032)	1.40 (0.106)	1.42 (0.143)	1.79 (0.168)	1.30 (0.034)
Water-Nsq	1.99 (0.038)	1.90 (0.009)	2.08 (0.092)	2.15 (0.053)	1.97 (0.056)	1.94 (0.023)
Average variance	(0.365)	(0.060)	(0.086)	(0.103)	(0.180)	(0.029)

Table 2: Application performance for six different synchronization algorithms for 28-processor runs, 14 threads per WildFire node. Execution time is given in seconds and the variance is shown in parentheses.

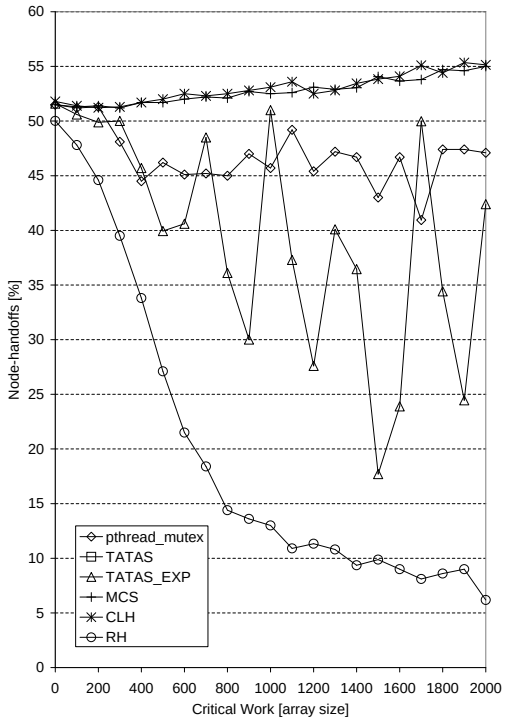

Figure 7: New microbenchmark locality study for a 2-node Sun WildFire system, 28-processor runs. Note that the TATAS numbers are not shown in this diagram.

Program	Problem Size	Total Locks	Lock Calls
Barnes	29k particles	130	69 193
Cholesky	tk29.O	67	74 284
FFT	1M points	1	32
FMM	32k particles	2 052	80 528
LU-c	1024×1024 matrix, 16×16 blocks	1	32
LU-nc	1024×1024 matrix, 16×16 blocks	1	32
Ocean-c	514×514	6	6 304
Ocean-nc	258×258	6	6 656
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10	3 975	295 627
Radix	4M integers, radix 1024	1	32
Raytrace	car	35	366 450
Volrend	head	67	38 456
Water-Nsq	2197 molecules	2 206	112 415
Water-Sp	2197 molecules	222	510

Table 1: The SPLASH-2 programs. Only emphasized programs are studied further. Lock-statistics are obtained for 32-processor runs.

6.3 Application Performance

In this section we evaluate the effectiveness of our new locking mechanism using the real SPLASH-2 applications [26]. Table 1 shows SPLASH-2 applications with the corresponding problem sizes and lock-statistics (*Total Locks* is the number of allocated locks, and the *Lock Calls* is the total number of acquire/release lock operations during the execution). We chose to further examine only applications with more than 10,000 lock calls, i.e., Barnes, Cholesky, FMM, Radiosity, Raytrace, Volrend, and Water-Nsq. For each application, we vary the synchronization algorithm used and measure the execution time on a 2-node Sun WildFire machine. Programs are compiled with GNU's gcc-3.0.4 (optimization level -O3). Table 2 presents the execution times in seconds for 28-processor runs for six different locking

schemes: native Solaris implementation of pthread_mutex-locks, TATAS, TATAS_EXP, MCS, CLH, and our RH lock (`FAIR_FACTOR` is equal to one in this experiment).[2] Variance is given in parentheses in the same table.

We chose to further investigate only Raytrace. This application renders a three-dimensional scene using ray tracing, and is one of the most unpredictable SPLASH-2 programs [7]. Detailed analysis of Raytrace is out of scope for this paper (see [23], [26], or [7] for more details). In this application, locks are used to protect task queues and for some global variables that track statistics for the program. The work between synchronization points is usually quite large. Execution time given in seconds for six different synchronization algorithms for 30- and 32-processor runs is shown below (variance is presented in parentheses):

Lock type	30 processors	32 processors
pthread_mutex	3.77 (0.840)	3.12 (0.166)
TATAS	3.93 (2.550)	3.86 (0.921)
TATAS_EXP	1.90 (0.101)	1.72 (0.125)
MCS	1.26 (0.265)	> 250 *s*
CLH	1.44 (0.255)	> 250 *s*
RH	0.69 (0.005)	0.70 (0.005)

The RH lock outperforms all other locks with a factor between 1.83 and 5.70 for 30-processor runs. Our lock also demonstrates the lowest measurement variance, only 0.005, compared to the second best value of 0.101 for TATAS_EXP. In the table above, we also demonstrate that MCS and CLH locks are practically unusable for a 32-processor runs. They seem to be extremely sensitive for small disturbances produced by the operating system itself. Speedup for Raytrace is shown in Figure 8.

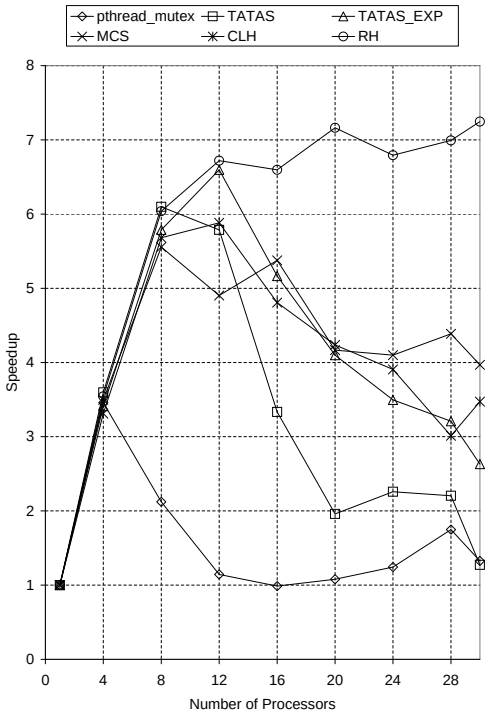

Figure 8: Speedup for Raytrace.

7 Conclusions

Three properties determine the average time between two processes/threads entering the contended critical section: lock handover time, traffic generated by the lock, and the data locality created by the lock algorithm. This paper demonstrates that the first-come first-served nature of queue-based locks make them less suitable for architectures with a non-uniform cache access time (NUCA), for example NUMAs built from a few large nodes or chip multiprocessors. In contrast, the simpler test-and-set locks gives a unfair advantage to neighboring processors when a lock is released, which will create a fast lock handover time as well as create more locality for the data accessed in the critical region.

We propose the new RH lock, that explores the NUCA architectures by creating a controlled unfairness in combination with a much reduced traffic compared with the test-and-set locks. The RH lock algorithm minimizes the global traffic generated at lock handover by making sure that only one thread per node will try to retrieve a lock which is currently not owned by a thread in the node. Also, the RH lock maximizes the node locality of NUCA architectures by handing over the lock to another process/thread in the same node. This will not only cut down on the lock handover time, but will also create locality in the critical section work, since its data structures will already reside in the node. A critical section guarded by the RH lock is shown to take less than half the time to execute compared with the same critical section guarded by any other lock.

Finally, we investigate the effectiveness of our new lock on a set of real SPLASH-2 applications. For example, execution time for Raytrace with 30 processors can be improved between 1.83 and 5.70 times by using the RH locks instead of any other tested locks.

[2] Unmodified versions of FMM and Radiosity will not execute correctly with queue-based locks, i.e., with the MCS and/or CLH locks. We did not investigated this any further.

Acknowledgments

We thank Michael L. Scott and William N. Scherer III, Department of Computer Systems, University of Rochester, for providing us with the source code for many of the tested locks, and for the entire microbenchmark experimentation framework. We would also like to thank Bengt Eliasson, Sverker Holmgren, Henrik Löf and the Department of Scientific Computing at Uppsala University for the use of their Sun WildFire machine. This work is supported in part by the Sun Microsystems, Inc., and the Parallel and Scientific Computing Institute (PSCI — ψ), Sweden.

References

[1] T. E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II Software, pages 170–174, August 1989.

[2] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, pages 282–293, June 2000.

[4] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*, pages 3–14, June 1998.

[5] A. E. Charlesworth. The Sun Fireplane System Interconnect. In *Proceedings of Supercomputing 2001*, November 2001.

[6] T. S. Craig. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, February 1993.

[7] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1999.

[8] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 13–24, November 2000.

[9] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Shared-Memory Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 64–75, April 1989.

[10] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.

[11] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, February 1999.

[12] A. Kägi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 170–180, June 1997.

[13] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, June 1997.

[14] D. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[15] B-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 25–35, October 1994.

[16] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 308–317, May 1996.

[17] P. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 165–171, Cancun, Mexico, April 1994. Extended version available as "Efficient Software Synchronization on Large Cache Coherent Multiprocessors," SICS Research Report T94:07, Swedish Institute of Computer Science, February 1994.

[18] L. W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 279–294, January 1996.

[19] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[20] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of Supercomputing '99*, November 1999.

[21] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA'84)*, pages 340–347, June 1984.

[22] M. L. Scott and W. N. Scherer. Scalable Queue-Based Spin Locks with Timeout. In *PPOPP'01*, Snowbird, Utah, USA, June 2001.

[23] J. P. Singh, A. Gupta, and M. Levoy. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer*, 27(7):45–55, July 1994.

[24] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of IEEE Hot Interconnects IV*, pages 41–52, August 1996.

[25] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.