

# Efficient Synchronization for Nonuniform Communication Architectures\*

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 parallel computers are often nonuniform communication architectures (NUCAs), where the access time to other processor's caches vary with their physical location. Still, few attempts of exploring cache-to-cache communication locality have been made. This paper introduces a new kind of synchronization primitives (lock-unlock) that favor neighboring processors when a lock is released. This improves the lock handover time as well as access time to the shared data of the critical region.*

*A critical section guarded by our new RH lock takes less than half the time to execute compared with the same critical section guarded by any other lock on our NUCA hardware. The execution time for Raytrace with 28 processors was improved 2.23–4.68 times, while global traffic was dramatically decreased compared with all the other locks. The average execution time was improved 7–24% while the global traffic was decreased 8–28% for an average over the seven applications studied.*

## 1 Introduction

There are plenty of examples in academia and industry of shared-memory architectures with a nonuniform memory access time to the shared memory (NUMA). Most of the NUMA architectures, but not all, also have *nonuniform communication architectures* (NUCA). This means the access time from a processor to the other processor's caches varies greatly depending on their placement. In particular, node-based NUCAs, in which 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 [BGM<sup>+</sup>00]. In our opinion, there are

strong indications that many important architectures in the future will have a nonuniform 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 [HK99, NvdP99]. However, many of today's applications show a large fraction of cache-to-cache misses [BGB98], 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&set* lock 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 [MCS91, Cra93, MLH94]. 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 contested 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&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&set* locks. The increased node locality will improve 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

\*SC2002 November 2002, Baltimore, Maryland, USA

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. Finally, we conclude in section 7.

## 2 Nonuniform Communication Architectures

Many large-scale shared-memory architectures have nonuniform access time to the shared memory (NUMA). In order to make a key difference, the nonuniformity 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 nonuniform communication architecture (NUCA). A NUCA is an architecture where the unloaded latency for a processor accessing data recently modified by another processor differs at least a factor of two depending on where that processor is located.

DASH was the first NUCA architecture [LLG<sup>+</sup>92]. 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 10 [LC96]. 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.

NUCA architecture	NUCA ratio
Stanford DASH	~ 4.5
Sequent NUMA-Q	~ 10
Sun WildFire	~ 6
Compaq DS-320	~ 3.5
Future: CMP & SMT	~ 6–10

Sun’s WildFire system can have up to four nodes with up to 28 processors each, totaling 112 processors [HK99]. Parts of each node’s memory can be turned into a RAC using a 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 [GSSD00]. Its NUCA ratio is roughly 3.5.

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 Hyperthreading and the IBM Power4’s dual CMP processors on a chip. The Piranha CMP proposal expects 8 CMP threads to run on each chip [BGM<sup>+</sup>00]. Larger systems, built from many such CMPs, are expected to have a NUCA ratio of between six and ten depending on the technology chosen.

Not all architectures are NUMAs or NUCAs. The recent SunFire 15k architecture can have up to 18 nodes, each with four processors, memory and directory controllers [Cha01]. The nodes are connected by a fast backplane. It has a flavor of both NUMA and NUCA. However, both its NUMA and NUCA ratios are well below two. The SGI Origin 2000 is a NUMA architecture with a NUMA ratio of around three for reasonably sized systems [LL97]. 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: atomic memory primitives; nonatomic memory primitives (load-linked/store-conditional), and explicit hardware lock-unlock primitives (CRAY’s 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 not currently popular on modern machines.

The five synchronization primitives we discuss and directly compare in this paper are: *test&test&set* (abbreviated TATAS), *test&test&set* with exponential backoff (abbreviated TATAS\_EXP), queue-based locks of Mellor-Crummey and Scott (abbreviated MCS), queue-based locks of Craig, Landin, and Hagersten (abbreviated CLH), and RH lock (our new NUCA-aware 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: *tas(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; *cas(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*.

The IBM 370 instruction set introduced *cas*. Sparc V9 provides *tas*, *swap*, and *cas* and is 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 cases. 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.

Rudolph and Segall first proposed an extension to ordinary *test&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 *tas* operation [RS84]. 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 false/zero to true/nonzero, using an atomic hardware primitive. In the Sparc V9 instruction set, this is typically done with load-store unsigned byte (LDSTUB) instruction. Traditional *test&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 [And90, MCS91]. The delay between *tas* attempts should not be too long; otherwise, processors might remain idle even when the lock becomes free. 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 [SS01]:

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 *tas* instruction on older machines can be found in the literature [GT90, MCS91]. On a standard symmetric multi-processor (SMP) with uniform access times between all of the 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 [GVW89] (see section 3.4 for more details). Several researchers have proposed locking primitives that incorporate both local spinning and queue-based locking in software [And89, GT90, MCS91].

The acquire function of the software-based queue locks perform three basic phases: 1) a `flag` variable in a shared address space is initialized to the value `BUSY`; 2) the content at the lock location in memory is swapped with the address value pointing to the `flag`; 3) the thread spins until the `prev_flag` memory location, a pointer which was returned by the swap, contains the value `FREE`. The release function of the queue-based locks writes a `FREE` value to the `flag` location.

The locking primitive called MCS is one of the first software queue-based lock implementations, originally inspired by the QOSB [GVW89] 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 [MCS91]. During the acquire request, the MSC lock inserts requesters for a held lock into a software queue using atomic operations such as `swap` and `cas`. 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 [MLH94]. Craig independently developed a lock identical to LH [Cra93]. In this paper we will refer to this lock as the CLH lock. The CLH lock requires one fewer remote accesses to transfer a lock than does MCS, and will usually outperform MCS when high lock contention exists [MLH94, SS01]. 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 the “reactive synchronization” presented by Lim and Agarwal a couple of years after the first proposals for queue-based locks [LA94]. Reactive synchronization algorithms will dynamically switch among several software lock implementations. Typically, spin locks (TATAS\_EXP) are used during the low-contention phase, and queue-based locks (MCS) are used during the high-contention phase [KBG97]. 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 [GVW89]. 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, and 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. Effective collocation is possible. 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 [KBG97].

## 4 Key Idea Behind RH Lock

Queue-based locks implement a first come, first served fairness, which is less desirable on a NUMA machine because of the potentially huge percentage of the lock’s node handoffs. In other words, there is a risk that a contested lock might “jump” back and forth between the nodes, creating an enormous amount of traffic. The goal of the RH lock is to create a lock that minimizes the global traffic generated at lock handover and maximizes the node locality of NUMA architectures. To make this possible in our first proposal of a NUMA-aware lock, it is necessary to have the information about what thread is performing the acquire-release operation and in which node that thread is running.

*Preliminaries.* Every node contains a copy of the lock. The total lock storage for a 2-node case is thus  $2 \times \text{sizeof}(\text{lock})$ . Initially, we could decide to logically place a lock in node 0 (mark the lock value as FREE in that node, meaning that both threads from the local node or from another node can acquire the lock). The copy of the lock that is allocated and placed in node 1 is then marked with a REMOTE value, meaning that the “global” lock is in another node (node 0). At most one node may have this local copy of the lock in state FREE. Thus, the threads from node 1 would “see” a REMOTE tag if they tried to acquire the local copy of the lock for the first time. The first thread that gets back the REMOTE value is the “node winner” and is allowed to con-

tinue to spin remotely with a larger backoff until the global lock is obtained. Other threads will spin locally (on their local copy) until the lock is fetched and released by the node winner.

*Minimizing global traffic.* One way to cut down on lock traffic is to make sure that only one thread per node (the node winner) tries to retrieve a lock which is currently not owned by a thread in the node.

*Maximizing the node locality of NUMAs.* One way to increase locality is to hand over 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 not only cuts down on the lock-handover time, but creates locality in the critical section work, since its data structures already reside 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 and make sure that other nodes eventually get the lock even if there are always local requests for the lock.

## 5 The RH Lock

During the design phase of the RH lock we paid attention to several general performance goals for locks, as given by Culler et al. [CSG99], page 343:

- *Low latency.* If a lock is free and no other processors are trying to acquire it at the same time, a 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 locks in the order 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.

In fact, we paid attention only to the 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 was to maximize the node locality of NUMA architectures.

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

Our NUMA-aware lock algorithm is shown in Figures 1 and 2. The RH lock algorithm supports only two nodes. `my_tid` is the thread identification number (0, 1, 2, ..., maximum number of threads - 1), and `my_node_id` is the node

```

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_fair = TRUE;
8:   else
9:     be_fair = 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_fair)
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.

number in which thread is placed (0 or 1). Both `my_tid` and `my_node_id` must be thread-private values, and preferably efficiently accessible from `rh_acquire` and `rh_release` functions. In our implementation, we reserve one of the Sparc’s thread-private global registers (%g2) for that particular task which is the most efficient way to obtain `my_tid` and `my_node_id`. During the `rh_acquire` function, every thread will swap its own thread identification number into the node-local copy of the lock. If it happens that the lock is already in the node and is in the state `L_FREE` or `FREE` (lines 6–7 in the `rh_acquire` function), the acquire operation finishes and the thread can proceed with its critical section. If the lock value is in the `REMOTE` state (lines 8–11), the function `rh_acquire_remote_lock` is called. Otherwise, the lock is in the node and some other neighbor thread performs the critical task. In that case, the current thread calls the `rh_acquire_slowpath` function and spins locally until it succeeds with its own acquire operation. Of course, there is one rare special case when another node is lucky enough to obtain the lock before the current thread (line 19 in the `rh_acquire_slowpath` function). Once again, in that case, the function `rh_acquire_remote_lock` is called.

To achieve controlled unfairness we use a thread-private `be_fair` 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 fairly as it can.” During the `rh_release` operation, the thread first checks if the `be_fair` variable is `TRUE` or not (line 3). If thread-private `be_fair` is `TRUE`, the lock will be released by writing the `FREE` value into the lock’s place. Otherwise, the lock can be released only to local/neighbor threads if interest was shown by them (line 7). Or, the lock can be released to the “world” by an atomic `cas` operation if no other from the same node showed any interest to acquire the same lock (line 6).

We use the following settings and definitions:

BACKOFF_BASE	625
BACKOFF_FACTOR	2
BACKOFF_CAP	2,500
REMOTE_BACKOFF_BASE	2,500
REMOTE_BACKOFF_CAP	10,000
FREE	max. number of threads
REMOTE	FREE + 1
L_FREE	FREE + 2

## 6 Performance Evaluation

Most experiments in this paper are performed on a Sun Enterprise E6000 SMP [SBC<sup>+</sup>96]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (*lmbench* latency [MS96]) 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 [HK99, NvdP99].<sup>1</sup> 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 `tas`, `swap`, and `cas` operations available in the Sparc V9 instruction set. The source code for TATAS\_EXP, CLH, and MCS lock is written by Scott and Scherer [SS01]. The entire experimentation framework is compiled with GNU’s `gcc-3.0.4`, optimization level `-O1`. The TATAS\_EXP lock was previously tuned for a Sun Enterprise E6000 machine by Scott and Scherer [SS01]. We use identical values in our experiments.

## 6.1 Uncontested Performance

One important performance goal for locks is *low latency* [CSG99]. In other words, if a lock is free and no other processors are trying to acquire it at the same time, the processor should be able to acquire it as quickly as possible. This is especially important for applications with little or no contention for the locks, which is a quite common case.

In this section we obtain an estimate of lock overhead in the absence of contention for three common scenarios. We evaluate the cost of the (1) acquire-release operation if the same processor is the owner of the lock (lock is in its cache); (2) if the lock is in the same node but the previous owner is not the current processor (lock is in the neighbor’s cache), and (3) if the lock was owned by a remote node (lock is in the cache of a processor that is in another node). The pseudocode for this NUCA-aware microbenchmark is shown below.

```

1: acquire and release all locks;
2: BARRIER

// case 1: previous owner: same processor
3: if (my_tid == 0) {
4:   acquire and release all locks;
5:   start timer;
6:   acquire and release all locks;
7:   stop timer;
8: }
9: BARRIER

// case 2: previous owner: same node
10: if (my_tid == 1) {
11:   start timer;
12:   acquire and release all locks;
13:   stop timer;
14: }
15: BARRIER

```

<sup>1</sup>Currently, our system has 30 processors, 16 plus 14, and therefore we perform our experiments mainly on a 14 plus 14 configuration.

```

// case 3: previous owner: remote node
16: if (my_tid == 2) {
17:   start timer;
18:   acquire and release all locks;
19:   stop timer;
20: }

```

For this experiment we need three threads. Threads with thread identification numbers (`tid`) zero and one are executing inside cabinet 1 and thread two is running inside cabinet 2. The total number of allocated locks is 2,000. The first two lines of the microbenchmark are used to warm the TLBs and are executed by all threads. Results are presented in Table 1.

Lock type	Previous Owner		
	<i>same processor</i>	<i>same node</i>	<i>remote node</i>
TATAS	135 ns	614 ns	2081 ns
TATAS_EXP	139 ns	668 ns	2014 ns
MCS	250 ns	722 ns	2192 ns
CLH	230 ns	827 ns	2623 ns
RH	178 ns	663 ns	4497 ns

Table 1: Uncontested performance for a single acquire-release operation for different synchronization algorithms.

Unsurprisingly, TATAS and TATAS\_EXP are the fastest acquire-release operations for this simple test without contention. We observe that our low latency design goal for the RH lock is within the reasonable magnitudes for the *same processor* and the *same node* case. For the *remote node* case RH lock performs much more poorly compared to other locks. The reason for this is two fold: 1) the `rh_acquire` function executed by the third thread will always acquire the local copy of the lock before the “global” lock which is in another cabinet (the `rh_acquire_remote_lock` function is always called), and 2) the `rh_acquire` function generates some additional remote coherence traffic at line 5.

## 6.2 Traditional Microbenchmark

The traditional microbenchmark that is used by many researchers consists of a tight loop containing a single acquire-release operation. The iteration time for a single-node Sun Enterprise E6000 is shown in Figure 3. The number of iterations performed by every thread in this microbenchmark is 10,000. The `FAIR_FACTOR` for the RH lock is equal to one. From the Figure 3 it appears that TATAS\_EXP and RH performs much better than the other locks. Both these locks have an exponential backoff, why waiting threads will “sample” the lock variable less often. This increases the probability that the thread releasing a lock will manage to acquire it again right away. The iteration time for these two locks indeed look similar to that of their uncontested performance if the previous owner is the *same processor* (see Table 1). TATAS has no backoff for the waiting threads, that is why a handoff to the same processor is less likely to occur, especially at high processor counts. Queue-based locks on the

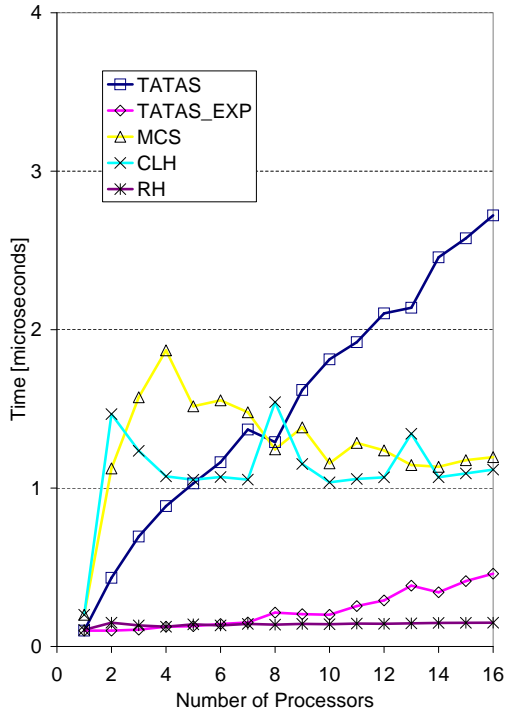


Figure 3: Traditional microbenchmark iteration time for a single-node Sun Enterprise E6000 (empty critical section).

other hand will rarely allow the releasing processor to directly acquire the lock again.

To overcome this problem we altered the microbenchmark to initialize a global variable `last_owner` inside the critical section, and force the thread to observe a new owner before it is allowed to compete for the lock again. A single remaining thread will be excluded from this requirement in order to run until completion. Slightly modified traditional microbenchmark is shown below.

```

shared int iterations, total_threads;
shared volatile int last_owner = -1;
shared volatile int total_finished = 0;

1: for (i = 0; i < iterations; i++) {
2:   ACQUIRE(L);
3:   if (my_tid != last_owner) last_owner = my_tid;
4:   // some more statistics goes here
5:   RELEASE(L);
6:   while ((last_owner == my_tid) &&
7:         (total_finished < total_threads - 1))
8:     ; // spin
9: }
10: atomic_increas(total_finished);

```

Figure 4 shows the result from the modified microbenchmark (number of iterations is still equal to 10,000 and RH's FAIR\_FACTOR is one). The iteration time here is an offset by the time it takes to perform the extra work in the critical section. The queue-based CLH and MCS perform the best. This is because only their third phase of the lock-acquire function (see section 3.3) is performed at lock-handover time. The releasing thread will perform a single store upgrade cache miss to its `flag` (its cache already contains a

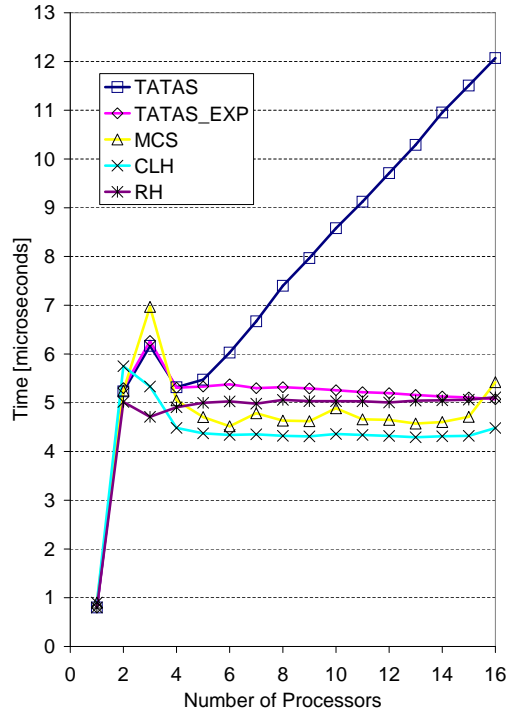


Figure 4: Slightly modified traditional microbenchmark iteration time for a single-node Sun Enterprise E6000.

valid copy in shared state) and the next thread will need a single load cache miss for its `prev_flag`, i.e., the same memory location as the releasing thread's `flag`. For all the other locks the releasing thread will need to perform a store miss to `L` (its cache does not contain a valid copy) and the next thread will perform a load cache miss and a store upgrade miss to `L`. TATAS is the only lock showing a distinct degradation caused by increased traffic as more processors are added.

The iteration time for the modified benchmark on a 2-node Sun WildFire is shown in Figure 5. In this study, we use round-robin scheduling for thread binding to different cabinets. 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 percent, and all `rh_acquire` accesses will result in the `rh_acquire_remote` calls.

Figure 6 shows the ratio of node handoffs for each lock type, reflecting how likely it is for a lock to migrate between nodes each time it is acquired. We ignore the TATAS values for more than 16 processors. The graph clearly shows the key advantage of the RH lock. The RH lock consistently shows low node handoff numbers for all the three settings of FAIR\_FACTOR.

The simple spin locks also show a node handoff ratio below 50 percent, which could be expected since local processors can acquire a released lock much faster than can remote

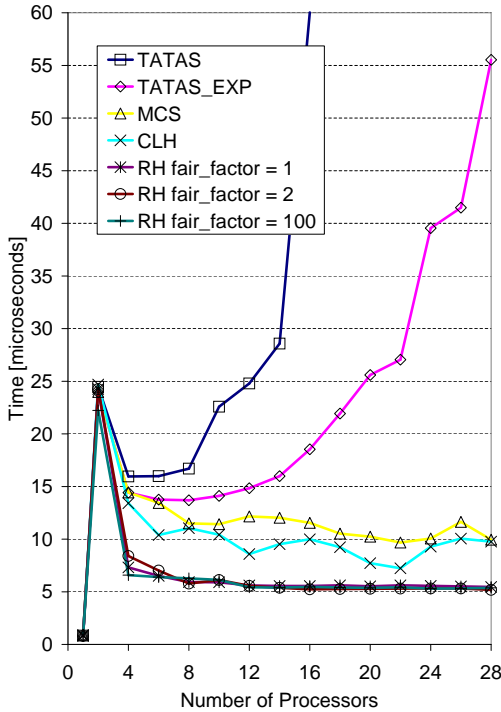


Figure 5: Slightly modified traditional microbenchmark iteration time for a 2-node Sun WildFire system.

processors. The queue locks are expected to show a node handoffs 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 acquire the lock twice in a row. However, the queue-based locks show unnatural behavior with large variation in the node handoff ratio. Our only explanation for this is pure luck. At 22 processors the CLH shows a ratio of 23 percent. This also explains the varied performance in Figure 5, for example the good CLH performance at 12 and 22 processors. At 8–10 processors, the node handoff ratio is fairly normal for both queue-based locks. Here we can see that 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.

It appears that the simplistic regular microbenchmark that we, as well as most other lock studies, use sometimes makes processors in the same node more likely to queue up after each other making the lock ratio substantially lower than expected. We also suspect that RH’s job of creating locality is greatly simplified by this highly regular benchmark.

### 6.3 New Microbenchmark

No real applications have a fixed number of processors pounding on a lock. Instead, they have a fixed number of processors spending most of their time on noncritical work, including accesses to uncontested locks. They rarely enter the “hot” critical section. The degree of contention is affected by

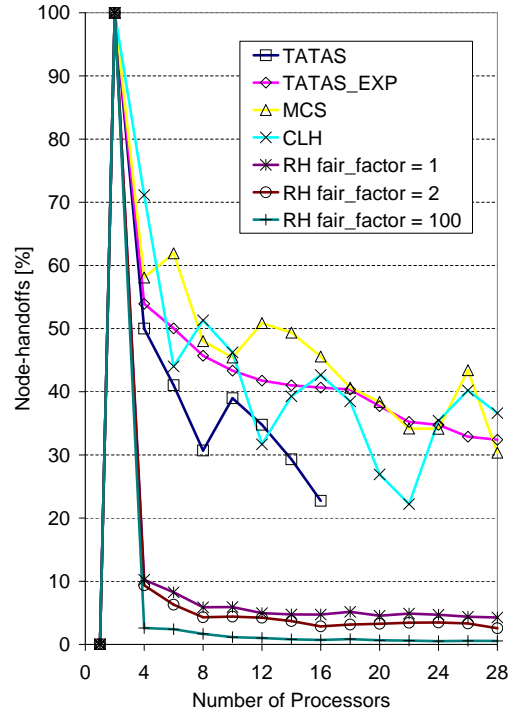


Figure 6: Slightly modified traditional microbenchmark locality study for a 2-node Sun WildFire system.

the ratio of noncritical work to critical work. The unnatural node handover behavior of the traditional lock benchmark led us to this new benchmark that we think reflects the expected behavior of a real application better. 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(L);
3:   {
4:     int j;
5:     for (j = 0; j < critical_work; j++)
6:       cs_work[j]++;
7:   }
8:   RELEASE(L);
9:   {
10:    int private_work[MAX_NONCRITICAL_WORK];
11:    int j, random_delay;
12:    for (j = 0; j < noncritical_work; j++)
13:      private_work[j]++;
14:    random_delay = random() % noncritical_work;
15:    for (j = 0; j < random_delay; j++)
16:      private_work[j]++;
17:   }
18: }

```

In the new microbenchmark, the number of processors is kept constant. They each perform some amount of noncritical work between trying to acquire the lock, consisting of one static delay (lines 12–13) and one random delay (lines 14–16) of similar sizes. Initially, the length of the noncritical work is chosen such that there is insignificant contention for the critical section (lines 3–7) and all lock algorithms perform almost identically. More contention is modeled by in-

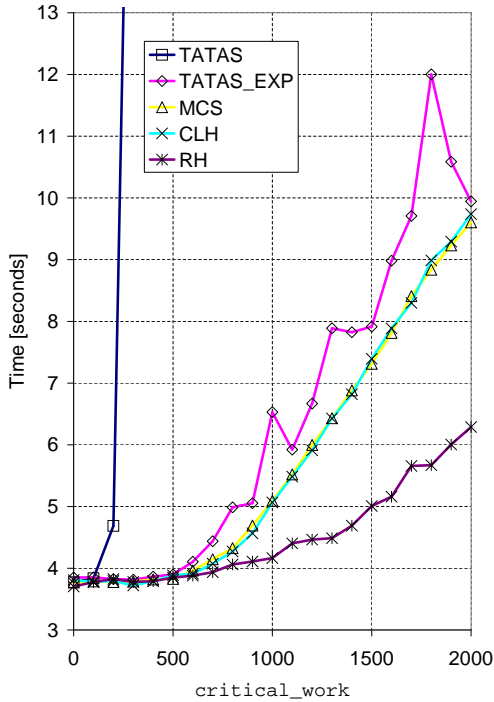


Figure 7: New microbenchmark iteration time for a 2-node Sun WildFire system, 28-processor runs.

creasing the number of elements of a shared vector that are modified before the lock is released.

Figure 7 shows that the two queue-based locks perform almost identically for the new benchmark and Figure 8 shows their node handoffs to be close to the expected values of 50 percent. The number of iterations is 1,000 in this experiment, and the `noncritical_work` is equal to 80,000. The `FAIR_FACTOR` for the RH lock is equal to one. As the amount of critical work is increased, the time to perform the critical work gets longer and contention for the lock is intensified. The TATAS poor contested performance will further add to the time period the lock is held for each iteration. This results in even more contention—very much like the feedback loop of an instable control system. This clearly demonstrates the danger of using TATAS in the application with some contention.

The TATAS values are measured for a `critical_work` between 0 and 400 because its performance is extremely poor as soon some contention is present. The simple spin locks still perform unpredictably 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.

In Table 2 we also present the numbers for the traffic that is generated on the machine for our new microbenchmark. The numbers are normalized against the TATAS\_EXP

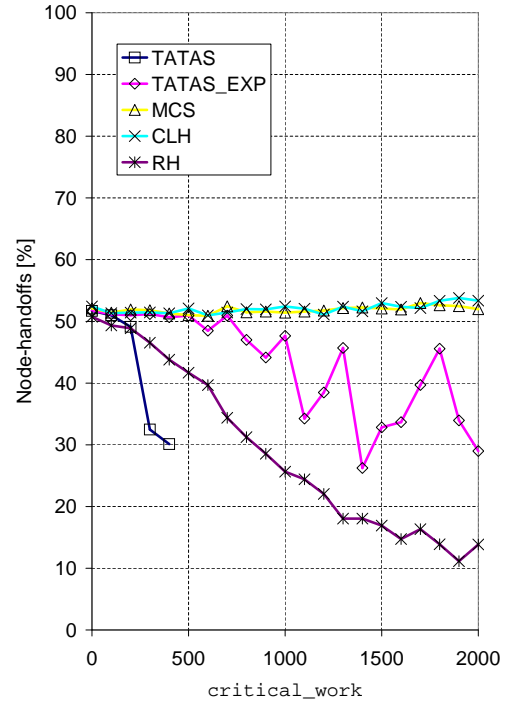


Figure 8: New microbenchmark locality study for a 2-node Sun WildFire system, 28-processor runs.

which is generating 15.145 millions local transactions and 8.878 millions global/remote transactions. The queue-based locks performs almost the same, MCS is generating slightly more transactions than CLH. The RH lock performs best, it generates about the same amount of local transactions as queue-based locks, but it generates only 2.777 millions global transactions, which is more than three times better than the TATAS\_EXP. For this setup, the execution time is improved 1.46–1.58 times, while the global traffic is significantly decreased.

Lock type	Local transactions	Global transactions
TATAS_EXP	1.00	1.00
MCS	0.49	0.47
CLH	0.48	0.46
RH	0.48	0.31

Table 2: Local and global/remote traffic generated for the new microbenchmark (`critical_work` = 1500, 28 processors). The performance for TATAS is extremely poor for this setting (see Figure 7), and is excluded from the table.

Program	TATAS	TATAS_EXP	MCS	CLH	RH
Barnes	1.54 (0.052)	1.43 (0.010)	1.83 (0.153)	1.54 (0.099)	1.54 (0.137)
Cholesky	2.31 (0.072)	2.04 (0.043)	2.09 (0.027)	2.25 (0.107)	2.23 (0.061)
FMM	4.84 (0.333)	4.19 (0.193)	4.33 (0.057)	4.46 (0.067)	4.27 (0.134)
Radiosity	1.66 (0.059)	1.75 (0.067)	N/A	N/A	1.44 (0.068)
Raytrace	2.90 (0.914)	1.71 (0.183)	1.41 (0.284)	1.38 (0.319)	0.62 (0.011)
Volrend	1.70 (0.031)	1.57 (0.096)	1.48 (0.278)	1.75 (0.157)	1.61 (0.088)
Water-Nsq	2.37 (0.028)	2.25 (0.057)	2.20 (0.035)	2.45 (0.031)	2.21 (0.011)
<b>Average</b>	<b>2.47 (0.212)</b>	<b>2.13 (0.093)</b>	<b>2.22 (0.139)</b>	<b>2.31 (0.130)</b>	<b>1.99 (0.073)</b>

Table 4: Application performance for five 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.

Program	Problem size	Total locks	Lock calls
<i>Barnes</i>	29k particles	130	69,193
<i>Cholesky</i>	tk29.O	67	74,284
FFT	1M points	1	32
<i>FMM</i>	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
<i>Radiosity</i>	room, -ae 5000.0 -en 0.050 -bf 0.10	3,975	295,627
Radix	4M integers, radix 1024	1	32
<i>Raytrace</i>	car	35	366,450
<i>Volrend</i>	head	67	38,456
<i>Water-Nsq</i>	2197 molecules	2,206	112,415
Water-Sp	2197 molecules	222	510

Table 3: The SPLASH-2 programs. Only emphasized programs are studied further. Lock statistics are obtained for 32-processor runs.

## 6.4 Application Performance

In this section we evaluate the effectiveness of our new locking mechanism using the real SPLASH-2 applications [WOT<sup>+</sup>95]. Table 3 shows SPLASH-2 applications with the corresponding problem sizes and lock statistics. *Total locks* is the number of allocated locks, and *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, which were the cases of 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

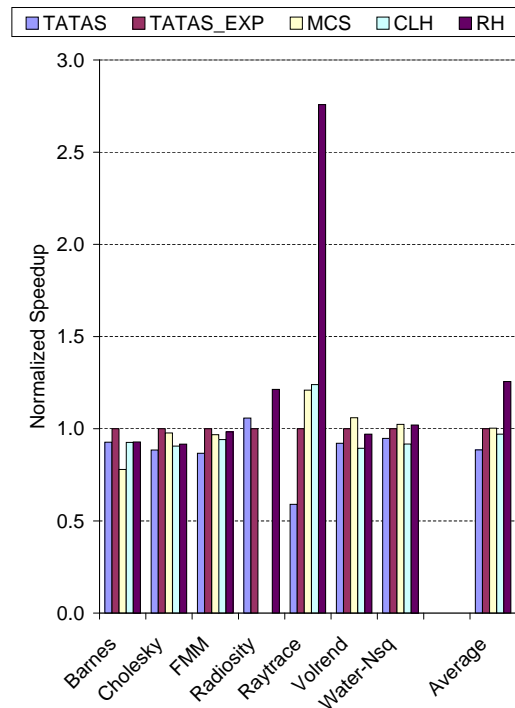


Figure 9: Normalized speedup for 28-processor runs on a 2-node Sun WildFire.

GNU’s gcc-3.0.4 (optimization level -O1). Table 4 presents the execution times in seconds for 28-processor runs for five different locking schemes: TATAS, TATAS\_EXP, MCS, CLH, and our RH lock.<sup>2</sup> (FAIR\_FACTOR is equal to one in this experiment.) Variance is given in parentheses in the same table. On the average, the execution time is improved 7–24 percent with the RH locks instead of other locks.

Normalized speedup for all lock algorithms is shown in Figure 9. For Barnes, the MCS lock is much worse than the

<sup>2</sup>Unmodified version of Radiosity will not execute correctly with queue-based locks. We did not investigate this any further.

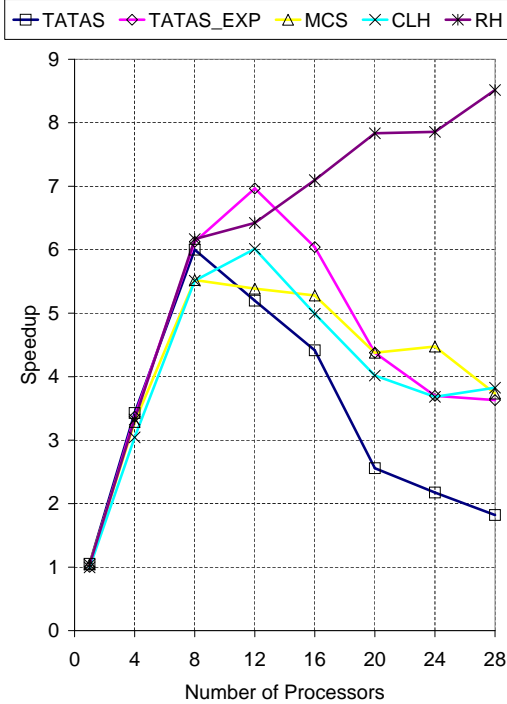


Figure 10: Speedup for Raytrace.

ordinary TATAS\_EXP, and for Volrend and Water-Nsq that is also the case for the CLH lock. On the average, queue-based locks perform about the same as the TATAS with exponential backoff. The RH lock demonstrates quite stable performance for both uncontested and contested applications.

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 [CSG99]. Detailed analysis of Raytrace is out of the scope of this paper (see [SGL94, WOT<sup>+</sup>95, CSG99] 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 five different synchronization algorithms, for single-, 28-, and 30-processor runs, is shown below (variance is presented in parentheses).

Lock type	1 CPU	28 CPUs	30 CPUs
TATAS	5.02	2.90 (0.914)	2.70 (0.445)
TATAS_EXP	5.26	1.71 (0.183)	2.05 (0.257)
MCS	5.05	1.41 (0.284)	> 200 s
CLH	5.30	1.38 (0.319)	> 200 s
RH	5.08	0.62 (0.011)	0.68 (0.002)

Speedup for Raytrace is shown in Figure 10. There is a decrease in performance for all other locks above 12 processors, while the RH lock continue to scale all the way up to 28 processors. The RH lock outperforms all other

Program	Local transactions	Global transactions
Barnes	2.339	1.779
Cholesky	14.323	4.553
FMM	6.771	3.227
Radiosity	4.142	1.876
Raytrace	7.751	2.882
Volrend	5.172	1.298
Water-Nsq	2.840	1.157
<b>Total</b>	<b>43.338</b>	<b>16.772</b>

Table 5: Local and global/remote traffic for TATAS\_EXP on a 2-node Sun WildFire, 28-processor runs. The numbers are given in the millions of transactions.

locks by a factor of 2.23–4.68 for 28-processor runs. Also, our NUCA-aware lock demonstrates the lowest measurement variance, only 0.011, compared to the second best value of 0.183 for TATAS\_EXP. In the table above, we also demonstrate that MCS and CLH locks are practically unusable for a 30-processor runs. They are extremely sensitive for small disturbances produced by the operating system itself. This unwanted behavior of the queue-based lock has been studied further by Scott on the same architecture and on the Sun Enterprise 10000 multiprocessor [SS01, Sco02].

Table 5 shows the generated traffic by all applications for the TATAS\_EXP synchronization algorithm. The normalized traffic numbers for all other synchronization algorithms are shown in Table 6. On the average, the global traffic is decreased 8–28 percent by RH locks for an average over the seven applications studied.

## 7 Conclusions

Three properties determine the average time between two processes/threads entering the contested 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 makes them less suitable for architectures with a nonuniform cache access time (NUCA), such as NUMAs built from a few large nodes or chip multiprocessors. In contrast, the simpler *test&set* locks gives an unfair advantage to neighboring processors when a lock is released, which will create a fast lock handover time as well as more locality for the data accessed in the critical region.

We also propose the new RH lock, which explores the NUCA architectures by creating controlled unfairness and much reduced traffic compared with the simple *test&set* locks. The RH lock algorithm minimizes the global traffic generated at lock handover by making sure that only one thread per node tries to retrieve a lock which is currently not owned by the same 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

Program	TATAS	TATAS_EXP	MCS	CLH	RH
Barnes	1.01 / 0.67	1.00 / 1.00	1.01 / 0.66	1.14 / 0.78	1.02 / 0.60
Cholesky	0.99 / 1.00	1.00 / 1.00	0.96 / 0.87	0.97 / 0.90	0.95 / 0.87
FMM	1.09 / 1.17	1.00 / 1.00	0.99 / 0.83	0.97 / 0.80	1.00 / 0.83
Radiosity	1.06 / 1.08	1.00 / 1.00	N/A	N/A	1.00 / 0.85
Raytrace	1.15 / 1.24	1.00 / 1.00	0.91 / 0.84	1.04 / 0.78	0.86 / 0.49
Volrend	1.02 / 1.07	1.00 / 1.00	1.02 / 1.05	1.04 / 1.17	1.01 / 1.03
Water-Nsq	1.01 / 1.03	1.00 / 1.00	1.00 / 1.04	1.07 / 1.10	1.03 / 1.02
<b>Average</b>	1.05 / 1.04	1.00 / 1.00	0.98 / 0.88	1.04 / 0.92	0.98 / 0.81

Table 6: Normalized traffic (local / global) for all synchronization algorithms for 28-processor runs, 14 threads per node.

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. We also demonstrate that one of the most commonly used *test&set* locks shows extremely unstable performance for a certain microbenchmark. We highly recommend to avoid this type of synchronization algorithms in a large-scale parallel applications, even though the risk of lock contention is minimal.

Finally, we investigate the effectiveness of our new lock on a set of real SPLASH-2 applications. For example, execution time for Raytrace with 28 processors was improved between 2.23 and 4.68 times, while the global traffic was dramatically decreased by using the RH locks instead of any other tested locks. The average execution time was improved 7–24 percent while the global traffic was decreased 8–28 percent for an average over the seven applications studied.

## 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. We would also like to thank Bengt Eliasson, Sverker Holmgren, Anders Landin, Henrik Löf, Fredrik Strömberg and the Department of Scientific Computing at Uppsala University for the use of their Sun WildFire machine. We are grateful to Karin Hagersten for her careful review of the manuscript. Finally, we would like to thank anonymous reviewers for their comments. This work is supported in part by Sun Microsystems, Inc., and the Parallel and Scientific Computing Institute (PSCI —  $\psi$ ), Sweden.

## References

- [And89] 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.
- [And90] 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.
- [BGB98] 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.
- [BGM<sup>+</sup>00] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, 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.
- [Cha01] A. E. Charlesworth. The Sun Fireplane System Interconnect. In *Proceedings of Supercomputing 2001*, November 2001.
- [Cra93] 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.
- [CSG99] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1999.
- [GSSD00] 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.
- [GT90] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.
- [GVW89] 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.
- [HK99] 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.

- [KBG97] 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.
- [LA94] 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.
- [LC96] 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.
- [LL97] J. Laudon and D. Lenoski. The SGI Origin: A cc-NUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, June 1997.
- [LLG<sup>+</sup>92] 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.
- [MCS91] 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.
- [MLH94] 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.
- [MS96] L. W. McVoy and Carl Staelin. Imbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 279–294, January 1996.
- [NvdP99] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun’s Wildfire Prototype. In *Proceedings of Supercomputing '99*, November 1999.
- [RS84] 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.
- [SBC<sup>+</sup>96] 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.
- [Sco02] M. L. Scott. Non-Blocking Timeout in Scalable Queue-Based Spin Locks. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, July 2002. Expanded version available as TR 773, Computer Science Dept., University of Rochester, February 2002.
- [SGL94] J. P. Singh, A. Gupta, and M. Levoy. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer*, 27(7):45–55, July 1994.
- [SS01] M. L. Scott and W. N. Scherer. Scalable Queue-Based Spin Locks with Timeout. In *PPOPP'01*, Snowbird, Utah, USA, June 2001. Source code is available for download from [ftp://ftp.cs.rochester.edu/pub/packages/scalable\\_synch/](ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/), filename: PPOPP\_01\_trylocks.tar.gz.
- [WOT<sup>+</sup>95] 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.