Hierarchical Backoff Locks for Nonuniform Communication Architectures

Zoran Radović and Erik Hagersten
Department of Information Technology, Uppsala University
P.O. Box 337, SE-751 05, Uppsala, Sweden
{zoran.radovic, erik.hagersten}@it.uu.se

Abstract

This paper identifies node affinity as an important property for scalable general-purpose locks. Nonuniform communication architectures (NUCAs), for example CC-NUMAs built from a few large nodes or from chip multiprocessors (CMPs), have a lower penalty for reading data from a neighbor's cache than from a remote cache. Lock implementations that encourages handing over locks to neighbors will improve the lock handover time, as well as the access to the critical data guarded by the lock, but will also be vulnerable to starvation.

We propose a set of simple software-based hierarchical backoff locks (HBO) that create node affinity in NUCAs. A solution for lowering the risk of starvation is also suggested. The HBO locks are compared with other software-based lock implementations using simple benchmarks, and are shown to be very competitive for uncontested locks while being more than twice as fast for contended locks. An application study also demonstrates superior performance for applications with high lock contention and competitive performance for other programs.

1. Introduction

The scalability of a shared-memory application is often limited by contention for some critical section, such as modification of shared data guarded by mutual exclusion locks. The simplest, and most widely used, test-and-*test&set* lock implementation suffers from poor performance at high contention: the more contested the critical section gets, the lower is the rate at which new threads can enter it. This very inappropriate behavior is mostly due to the vast amount of traffic generated at each lock handover event.

Some alternative lock implementations' traffic is less dependent on the number of contenders, which is why their lock hand-over rates do not decrease as significantly at high contention. The simplest way to limit the contention traffic is to apply some backoff strategy that causes the threads to access the common lock variable less frequently the longer they have waited. The more advanced queue-based locks instead maintain a first come, first served order between the contending threads. Each contender will only spin on the dedicated flag set at its predecessor's release of the lock, and contenders ordered after it will not be affected [5, 16, 18]. However, the complicated software queuing locks are less efficient for uncontested locks, which have led to the creation of even more complicated adaptive hybrid proposals in the quest for a general-purpose solution [14].

Shared-memory architectures with a nonuniform memory access time to the shared memory (CC-NUMAs) are gaining popularity. Most systems that form NUMA architectures also have the characteristic of a *nonuniform communication architecture* (NUCA), in which the access time from a processor to other processors' caches varies greatly depending on their placement. In node-based NUMA systems; in particular, processors have much shorter access times to other caches in their group than to the rest of the caches. Recently, technology trends have made it attractive to run more than one thread per chip, using either the chip multiprocessor (CMP) and/or the simultaneous multithreading (SMT) approach. Large servers, built from several such chips, can therefore be expected to form NUCAs, since collated threads will most likely share an on-chip cache at some level [2].

Due to the popularity of NUMA systems, optimizations directed to such architectures have attracted much attention in the past. For example, optimizations involving the migration and replication of data in NUMAs have demonstrated a great performance improvement in many applications [10, 12, 19]. In addition, since many of today's applications exhibit a large fraction of cache-to-cache misses [3], optimizations which consider the NUCA nature of a system may also lead to significant performance enhancements.

On a NUCA, it is attractive from a performance point of view to hand the lock to a waiting neighbor, rather than the thread which has waited the longest time. Favoring the neighbor will improve the lock handover time, as well as the access to the critical data that most likely reside in its cache. However, in such a scheme attention must also be given to starvation avoidance. We have noticed that the existing test-and-*test&set* locks already give some unfair advantage to the processor neighbors 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*-based locks.

The goal of this work is to create a new set of locks that efficiently exploit communication locality in a NUCA while

minimizing the potential risk of starvation. In order to be generally usable, such a lock should scale to a large number of nodes, handle contended locks well, have reasonable memory space requirements, and introduce minimal overhead for the uncontested locks.

The remainder of this paper is organized as follows. Section 2 gives an introduction to several NUCAs. Background and related work is presented in section 3. Our new NUCA-aware locks are presented in section 4. In section 5 we present performance results obtained on a 2-node Sun WildFire machine. A simple fairness and sensitivity study is performed in section 6, and we conclude in section 7.

2. Nonuniform Communication Architectures

Many large-scale shared-memory architectures have nonuniform access time to the shared memory. In order to make a key difference, the nonuniformity should be substantial—let's say at least a factor of two between best and worst unloaded latency. Most NUMA architectures also have a substantial difference in latency for cache-to-cache transfer—a nonuniform communication architecture. A NUCA is an architecture in which the unloaded latency for a processor accessing data recently modified by another processor differs at least by a factor of two, depending on where that processor is located.

DASH was the first NUCA machine [13]. 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 [15]. 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 Example	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 [10]. Parts of each node's memory can be turned into an RAC using a technique called coherent memory replication (CMR). Accesses to data allocated to a CMR cache have a NUCA ratio of about six, while accesses to other data only have a NUCA ratio of less than two.

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 [7]. Its NUCA ratio is roughly 3.5.

Future microprocessors can be expected to run many more threads on a chip through 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 eight CMP threads to run on each chip [2]. Larger systems, built from many such CMPs, are expected to have a NUCA ratio of between six and ten depending on the technology chosen.

It is possible that several levels of non-uniformity will be present in future large-scale servers. A simple example of this would be one of today's NUMA architectures populated with CMP processors instead of traditional single-threaded processors. This would create a hierarchical NUMA and NUCA property of the system.

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 [4]. The nodes are connected by a fast backplane. It has a flavor of both NUMA and NUCA. However, both of its NUMA and NUCA ratios are below two. The SGI Origin 2000 is a NUMA architecture with a NUMA ratio of around three for reasonably sized systems [12]. However, it does not efficiently support cache-to-cache transfers between adjacent processors and also has a NUCA ratio below two.

3. Background and Related Work

Ideally, synchronization primitives should provide good 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 Goodman's queue-on-lock-bit). In this paper, we will concentrate on implementing locks entirely in software using the atomic memory primitives, that are available in the majority of modern processors. The software-only locking primitives we directly compare are the following:

1. **TATAS**: traditional test-and-*test&set* lock
2. **TATAS_EXP**: TATAS with exponential backoff
3. **MCS**: queue-based locks of Mellor-Crummey and Scott [18]
4. **CLH**: queue-based locks of Craig, Landin, and Hagersten [5, 16]
5. **RH**: our proof-of-concept NUCA-aware lock [20]
6. **HBO**: our new NUCA-aware spin lock with hierarchical backoff (see section 4.1)
7. **HBO_GT**: HBO with global traffic throttling (see section 4.2)
8. **HBO_GT_SD**: HBO_GT with starvation detection (see section 4.3)

We also present a short introduction to alternative synchronization approaches; reactive synchronization and several hardware locking schemes.

Atomic Primitives. In this paper we make reference to three atomic operations: (1) `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; (2) `swap`(*address*, *value*) atomically writes a *value* to the *address* memory location and returns its original contents; (3) `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, returns its original contents and replaces it with a *new_value*.

Simple Locking Algorithms. Two very commonly used *busy-wait* algorithms are TATAS and TATAS_EXP. The contention produced by the traditional *test&set*-based spin locks can be reduced by polling (busy-wait code) with ordinary load operations to avoid generating expensive stores to potentially shared locations (TATAS algorithm). Furthermore, the burst of refill traffic whenever a 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 [1, 18]. 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, and one typical implementation is shown below.

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

```
01 void tatas_exp_acquire(tatas_lock *L)
02 { if (tas(L)) tatas_exp_acquire_slowpath(L); }
```

```
03 void tatas_exp_acquire_slowpath(tatas_lock *L)
04 {
05   int b = BACKOFF_BASE, i;
06   do {
07     for (i = b; i; i--) ; // delay
08     b = min(b * BACKOFF_FACTOR, CAP);
09     if (*L)
10       continue;
11   } while (tas(L));
12 }
```

```
13 void tatas_exp_release(tatas_lock *L)
14 { *L = 0; }
```

In many implementations, *acquire* and *release* functions are *in-lined*, while the *acquire_slowpath* routine is linked to the binary code. Backoff parameters must be tuned by trial and error for each individual architecture. The storage cost for TATAS locks is low and does not increase with the number of processors.

Software Queuing Locks. Even with exponential backoff, TATAS locks still induce significant traffic. Software queuing locks may eliminate this problem by letting each process spin on a different local-memory location. Many of the software queuing locks are inspired by the first proposal for a distributed, hardware queue-based locking scheme proposed for the cache controllers of the Wisconsin Multicube in the late 1980s [8].

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. Numerous variations of software queuing lock implementations are known [1, 5, 9, 16, 18, 22].

A unique feature of software queuing locks found in many implementations is an explicit starvation avoidance and maximal fairness; in other words, first come, first served order of lock-acquire requests is guaranteed. In addition, software queue-based locks provide reasonable latency in the absence of contention, provide good scalability for high-contended locks on many architectures, and can easily be completely in-lined into the application code.

The RH Lock. The RH lock is our proof-of-concept NUCA-aware spin lock that supports two nodes [20]. The goal for the RH lock was to create a lock that minimizes the global traffic generated at lock handover and maximizes the node locality of NUCAs. In the RH scheme, every node contains a copy of a lock. Thus, the lock storage cost is twice that of simple locking algorithms. Allocating and physically placing memory in different nodes may be difficult or even impossible task for many machines. That is why the RH lock code is not particularly portable.

Initially, the lock is logically placed in node 0 (the lock value is marked as `FREE`, meaning that both threads from the local or remote node are allowed to acquire the lock). The other node (node 1) will observe a `REMOTE` value if it acquires its local copy of the lock for the first time. The first thread that observes the `REMOTE` tag is the "node winner" and will continue to spin remotely with a larger backoff until the global lock is obtained. The RH scheme can exclusively hand over the lock to another thread in the same node by marking the lock value with "local free" tag `L_FREE`. This will not only cut down on the lock handover time, but will also create more locality in the critical section work, since its data structures are probably already in the node. The `swap` and `cas` primitives are used, and the implementation is vulnerable to starvation.

Alternative Approaches. The fact that some synchronization algorithms perform well under low-contention periods and others under high-contention periods is the basic idea behind the *reactive synchronization* presented by Lim and Agarwal [14]. Reactive 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 [11]. Reactive algorithms demonstrate modest performance gains.

The first to propose a distributed, queue-based locking scheme in hardware were Goodman, Vernon, and Woest [8]. They introduced the queue-on-lock-bit primitive (QOLB, originally called QOSB). In this scheme, a distributed, linked list of nodes waiting on a lock is maintained entirely in hardware (pointers in the processor caches are used to maintain the list of the waiting processors), and the releaser grants the lock to the first waiting node without affecting others. Effective *collocation* (allocation of the protected data in the same cache line as the lock) is possible; thus, this hardware scheme may reduce the lock hand-over time

as well as the interference of lock traffic with data access and coherence traffic. The original QOLB proposal demonstrates good performance [11], but it requires complex protocol support, new instructions, and recompilation of applications. Another purely hardware-based mechanism called *Implicit* QOLB uses speculation and delays to transparently convert software locks to provide a hardware queued-lock behavior without requiring any software support or new instructions [21]. The load-linked/store-conditional instructions are used to demonstrate a possible implementation.

Stanford DASH uses directories to indicate which processors are spinning on the lock [13]. When the lock is released, one of the waiting nodes is chosen at random and is granted the lock. The grant request invalidates only that node's caches and allows one processor in that node to acquire the lock with a local operation. This scheme lowers both the traffic and the latency involved in releasing a processor waiting on a lock. A time-out mechanism on the lock grant allows the grant to be sent to another node if the spinning process has been swapped out or migrated.

4. Hierarchical Backoff Locks

In this section we describe a set of a new NUCA-aware spin locks with hierarchical backoffs (HBO and HBO_GT) that exploit communication locality and reduce global traffic for contended locks while adding less overhead for uncontested locks than any of the software queue-based lock implementations. We also suggest one solution that lowers the risk of starvation (HBO_GT_SD). The storage cost for all proposed locks is low (a single variable allocated in just any of the NUCA nodes suffices) and does not increase with the number of processors. HBO_GT and HBO_GT_SD also use one extra variable per NUCA node.

What do we need to make this possible? All three proposals only use one common atomic operation: `cas`. In addition, per-thread/process `node_id` information is needed.

4.1. The HBO Lock

The goal of the HBO lock is similar to that of the RH lock, which is that the algorithm should exploit communication locality and reduce global traffic for contended locks. In addition, in this algorithm, we pay attention to adding as little overhead as possible for uncontested locks. Ideally, at low contention or in the absence of contention the algorithm should not add any overhead, it should simply perform an atomic operation directly on the lock variable.

The idea behind the HBO lock is really simple: when a lock is acquired, the `node_id` of the thread/process is `cas`-ed into the lock location. In other words, if the lock-value is in the `FREE` state, it is atomically changed into the `node_id`, otherwise it remains the same. If a busy lock is held by someone in the same node, the `cas` will return the thread's own `node_id`, and the thread will start spinning with a small backoff constant (the same, for example, as the typical TATAS_EXP configuration). If the `cas` returns a different `node_id`, the thread will use a larger backoff constant. In this manner, a thread that is executing in the same node in which a lock has already been obtained will be more likely to subsequently acquire the lock when it is freed in relation to other contending threads executing in other nodes. Decreased migration of the lock (and the shared critical-section data structures) from node to node is obtained, and the overall performance is enhanced. This scheme can be expanded in a hierarchical way, using more than two sets of constants, for a hierarchical NUCA. Figure 1 shows code for the HBO lock. Emphasized lines are related to the HBO_GT lock and should be ignored for the HBO proposal.

Note that the "critical path" of the HBO lock (lines 6–9) does not add any significant overhead compared with the simple spin locks (assuming that the `node_id` information is easily accessible, e.g., it is stored in a thread-private register). That is important for the performance of the lock in the absence of contention.

As in the TATAS_EXP implementation (see section 3), the *acquire* and *release* functions of HBO locks can be inlined, while the *acquire_slowpath* routines are linked to the binary code. Backoff parameters must also be tuned by trial and error for each individual architecture.

4.2. The HBO_GT Lock

When multiple processors in the same node are executing in the slow spin loop (HBO lock, lines 37–52, Figure 1), the global coherence traffic through network is created by `cas` operations of each of the spinning processors. The purpose of the HBO_GT (global traffic throttling) lock is to limit the number of processors that are spinning in the same node and attempting to gain a lock currently owned by another node, thereby reduce global traffic on the network.

Before acquiring a lock, the thread reads the per-node memory location (not necessarily allocated in the local memory) `is_spinning`, compares its content with the lock address, and keeps spinning for as long they are equal (line 5 and 56, Figure 1). Then, the thread performs the atomic `cas` operation (line 6 and 57). If the `cas` returns a `node_id` different from the thread's own id, the thread will store the lock address *L* in the node's `is_spinning` and start to spin for the lock with a fairly large backoff constant. This operation may thus prevent others in the same node from performing lock-acquisition transactions to the lock address (the `cas` operations) that might otherwise create global coherence traffic on the network. As soon as the thread has acquired the lock, it writes the "dummy value" to the node's `is_spinning`, which allows any waiting neighbor to start spinning. By using this algorithm, there is usually only one thread per node (or a small number of threads) that is performing remote spinning.

4.3. The HBO_GT_SD Lock

Many of the queue-based locks guarantee starvation freedom. Even though starvation is usually unlikely [6], with multiple threads competing for a lock, it is possible that

```
typedef volatile unsigned long hbo_lock;
```

```
01 void hbo_acquire(hbo_lock *L)
02 {
03   unsigned long tmp;
04
05   while (L == is_spinning[my_node_id]) ; // spin
06   tmp = cas(L, FREE, my_node_id);
07   if (tmp == FREE)
08     return; // lock was free, and is now locked
09   hbo_acquire_slowpath(L, tmp);
10 }
```

```
11 void backoff(int *b, int cap)
12 {
13   int i;
14   for (i = *b; i; i--) ; // delay
15   *b = min(*b * BACKOFF_FACTOR, cap);
16 }
```

```
17 void hbo_acquire_slowpath(hbo_lock *L,
18                       unsigned long tmp)
19 {
20   int b;
21
22  start:
23
24   if (tmp == my_node_id) {      // local lock
25     b = BACKOFF_BASE;
26     while (1) {
27       backoff(&b, BACKOFF_CAP);
28       tmp = cas(L, FREE, my_node_id);
29       if (tmp == FREE)
30         return;
31       if (tmp != my_node_id) {
32         backoff(&b, BACKOFF_CAP);
33         goto restart;
34       }
35     }
36   }
37   else {                        // remote lock
38     b = REMOTE_BACKOFF_BASE;
39     is_spinning[my_node_id] = L;
40     while (1) {
41       backoff(&b, REMOTE_BACKOFF_CAP);
42       tmp = cas(L, FREE, my_node_id);
43       if (tmp == FREE) {
44         is_spinning[my_node_id] = DUMMY;
45         return;
46       }
47       if (tmp == my_node_id) {
48         is_spinning[my_node_id] = DUMMY;
49         goto restart;
50       }
51     }
52   }
53
54  restart:
55
56   while (L == is_spinning[my_node_id]) ; // spin
57   tmp = cas(L, FREE, my_node_id);
58   if (tmp == FREE)
59     return;
60   goto start;
61 }
```

```
62 void hbo_release(hbo_lock *L)
63 {
64   *L = FREE;
65 }
```

Figure 1. Acquire and release code for HBO and HBO_GT locks. Emphasized lines are related to the HBO_GT implementation; they should be ignored for the HBO lock.

some threads may be granted the lock repeatedly while others may not and may become starved. Since both HBO and HBO_GT (and all other simple spin-locking algorithms) are vulnerable to starvation, we need a solution that at least lowers the risk of potential starvation, which is especially important for HBO locks because of their "nonuniform" nature in the locking algorithm. For example, a count can be maintained of the number of times a lock-request has been denied and, after a certain threshold, a thread's priority may be increased (a thread can start spinning without any backoff until the lock is obtained). In addition to this simple "thread-centric" solution, HBO_GT_SD provides a "node-centric" mechanism, described in further detail below, which lowers the risk of node starvation. Explicit thread-centric starvation avoidance can be implemented with alternative approaches (reactive synchronization [14], see section 3), at the expense of additional complexity in the lock algorithm.

The HBO_GT_SD lock is based on the HBO_GT proposal. The idea behind the node-centric algorithm, which lowers the risk of node starvation, is the following: after a "winning" thread (or a small number of threads), has tried several times to acquire a remote lock owned by another node, it gets "angry." An angry node will take two measures to get the lock more quickly: (1) it will spin more frequently, and (2) it will set the `is_spinning` for the other nodes to the lock address and thus prevent more threads in those nodes from trying to acquire the lock.

```
43 if (tmp == FREE) {
44   // release the threads from our node
45   is_spinning[my_node_id] = DUMMY;
46   // release the threads from stopped nodes, if any
47   if (stopped_nodes > 0)
48     is_spinning[for each stopped_node] = DUMMY;
49   return;
50 }
51 if (tmp == my_node_id) {
52   is_spinning[my_node_id] = DUMMY;
53   if (stopped_nodes > 0)
54     is_spinning[for each stopped_node] = DUMMY;
55   goto restart;
56 }
57 if (tmp != my_node_id) {
58   // lock is still in some remote node
59   get_angry++;
60   if (get_angry == GET_ANGRY_LIMIT) {
61     stopped_node_id[stopped_nodes++] = tmp;
62     is_spinning[tmp] = L;
63   }
64 }
```

Figure 2. Part of the HBO_GT_SD lock's acquire function. Lines 43–50 from the HBO_GT algorithm are replaced with the code in this figure.

The details of this algorithm are shown in Figure 2 (lines 43–50 from Figure 1 are replaced with the code from Figure 2). Note that the initialization of variables `get_angry` and `stopped_nodes` is excluded from the code example.

5. Performance Evaluation

Most of the experiments in this paper are performed on a Sun Enterprise E6000 SMP [25]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (*lmbench* latency [17]) 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 [10, 19].[1] 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 both run 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. All HBO locks are implemented with only a `cas` operation. The code for TATAS_EXP, CLH, and MCS lock is written by Scott and Scherer [23], and the entire experimentation framework is compiled with GNU's gcc-3.2, optimization level -O3. The TATAS_EXP lock was previously tuned for a Sun Enterprise E6000 machine by Scott and Scherer [23]. We use identical values in our experiments. By using the gcc's `static __inline__` construct, we explicitly inline TATAS, CLH, and MCS locks. All other locks have a "slowpath" routine called from the corresponding in-line part of the acquire function. Release functions for all locks are in-lined. Machines used for tests were otherwise unloaded.

5.1. Uncontested Performance

One important design goal for locks is a *low latency* acquisition of a free lock [6]. 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 fortunately 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 acquire-release operation (1) if the *same processor* as the previous owner 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). More details about this NUCA-aware microbenchmark are given in [20]. Results are presented in Table 1. We observe that our low latency design goal for the HBO locks is fulfilled, and performance is almost identical with the simplest locks: TATAS and TATAS_EXP.

[1] Currently, our system has 30 processors, 16 plus 14, and therefore we perform our experiments mainly on a 14 plus 14 configuration.

Lock Type	Previous Owner		
	Same Processor	*Same Node*	*Remote Node*
TATAS	150 ns	660 ns	2050 ns
TATAS_EXP	143 ns	613 ns	2070 ns
MCS	210 ns	732 ns	2120 ns
CLH	234 ns	806 ns	2630 ns
RH	198 ns	672 ns	4480 ns
HBO	152 ns	652 ns	2010 ns
HBO_GT	152 ns	643 ns	2010 ns
HBO_GT_SD	149 ns	638 ns	2010 ns

Table 1. Uncontested performance for a single acquire-release operation.

5.2. Traditional Microbenchmark

The traditional microbenchmark we use in this paper is a slightly modified code used by Scott and Scherer in [23] on the same architecture—Sun WildFire prototype SMP cluster. The code consists of a tight loop containing a single acquire-release lock operation, plus some critical section work for gathering statistics. In addition, we initialize `last_owner`, a global variable inside the critical section, and force the thread to observe a new owner before it is allowed to contend for a lock again. The last remaining thread is excluded from this requirement in order to run to completion (see [20] for more details).

The microbenchmark iteration time for parallel execution on a 2-node Sun WildFire is shown in Figure 3 (left diagram). In this study, we use round-robin scheduling for thread binding to different cabinets. Figure 3 (right diagram) shows also 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. As expected, NUCA-aware locks consistently demonstrate low node-handoff numbers. The simple spin locks (especially TATAS) also show fairly low node handoffs, which can be expected since local processors acquire a released lock much faster than remote processors do. The queue-based locks are expected to show 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 exhibit unnatural behavior in the node-handoff ratio. The simplistic microbenchmark we use is used in most other lock studies. It makes processors in the same node more likely to "queue up" after each

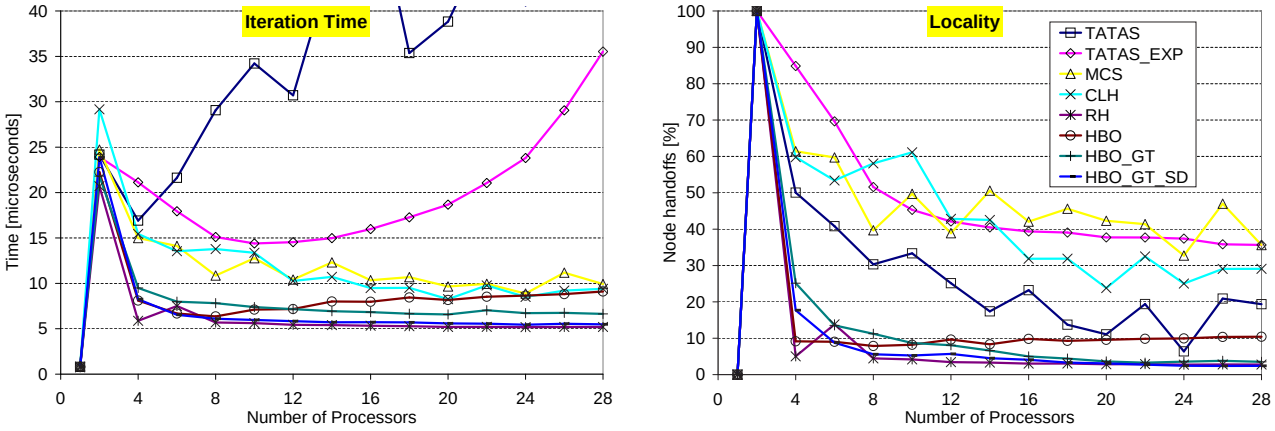

Figure 3. Slightly modified traditional microbenchmark on a 2-node Sun WildFire system.

other, and node handoffs are substantially lower than expected. This is especially true for CLH, which takes unfair advantage of the test setup. Our only explanation for this is pure luck. At 20 processors, the CLH shows a ratio of about 24/100. This also explains the varied performance in Figure 3, such as the good CLH performance at 20 and 24 processors. At 8–10 processors, the node-handoff numbers are fairly normal for both queue-based locks. Here we can see that a critical section guarded by NUCA-aware locks (especially by RH, HBO_GT, and HBO_GT_SD) takes about half the time to execute compared with the same critical section guarded by any other software-based lock.

5.3. 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 the ratio of noncritical work to critical work. The unnatural node-handover behavior of the traditional lock benchmark led us to a new lock benchmark, which we feel better reflects the expected behavior of a real application. In the new microbenchmark, the number of processors is kept constant. Each performs some amount of noncritical work, which consist of one static delay and one random delay of similar sizes, between attempts to acquire the lock. Initially, the length of the noncritical 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 in Figure 4.

Figure 5 (left diagram) shows that the two queue-based locks perform almost identically for the new benchmark. Figure 5 (right diagram) shows also their node handover to be close to the expected values. The simple spin locks still perform unpredictably. This is tied to their unpredictable

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

```
01 for (i = 0; i < iterations; i++) {
02   ACQUIRE(L);
03   {
04     int j;
05     for (j = 0; j < critical_work; j++)
06       cs_work[j]++;
07   }
08   RELEASE(L);
09   {
10     int non_cs_work[MAX_PRIVATE_WORK];
11     int j, random_delay;
12     for (j = 0; j < private_work; j++)
13       non_cs_work[j]++;
14     random_delay = random() % private_work;
15     for (j = 0; j < random_delay; j++)
16       non_cs_work[j]++;
17   }
18 }
```

Figure 4. New microbenchmark.

node handover. (TATAS values are measured for a `critical_work` of 0–1300 because its performance is poor for higher levels of contention.) The NUCA-aware locks perform better the more contention there is, which can be explained by its decreasing amount of node hand-over. This is exactly the behavior we want in a lock: the more contention there is, the better it should perform.

In Table 2 we present the numbers for the traffic that is generated by our new microbenchmark. The numbers are normalized to the TATAS_EXP which generates 15.1 million local and 8.9 million global transactions. Once again, software queuing locks performs almost identically. We can also observe that NUCA-aware locks generate less than half the amount of global transactions than any of the software-based locks on this NUCA machine. The global traffic is reduced by a factor of 15 compared to the traditional TATAS locks.

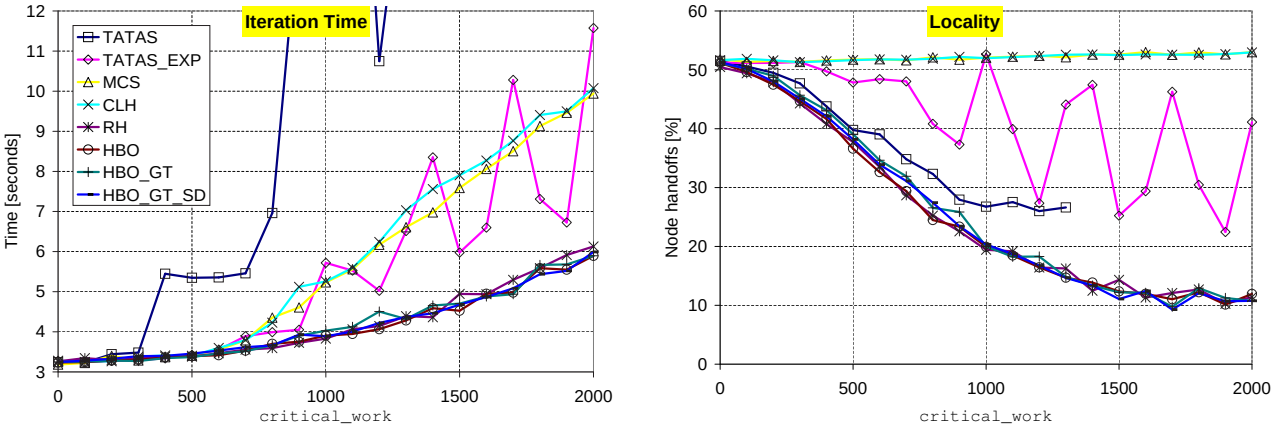

Figure 5. New microbenchmark on a 2-node Sun WildFire system, 28-processor runs.

Lock Type	Local Transactions	Global Transactions
TATAS	4.41	4.70
TATAS_EXP	1.00	1.00
MCS	0.53	0.65
CLH	0.54	0.63
RH	0.54	0.28
HBO	0.60	0.30
HBO_GT	0.60	0.30
HBO_GT_SD	0.61	0.29

Table 2. Normalized local and global traffic generated for the new microbenchmark (`critical_work` = 1500, 28 processors).

5.4. Application Performance

In this section we evaluate the effectiveness of our new locking mechanisms using explicitly parallel programs from the SPLASH-2 suite [26]. 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). Problem size is a very important issue in this context. Generally, the larger the problem size, the lower the frequency of synchronization relative to computation. On the one hand, using large problem sizes will make synchronization operations seem less important. On the other hand, small problem sizes might lead to very low speedup of the application, rendering it uninteresting on a machine of this scale, even though we chose the fairly standard problem sizes found in many other related investigations. We also chose to further examine only applications with more than 10,000 lock calls, as in the cases of

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 matrices, 16×16 blocks	1	32
LU-nc	1024×1024 matrices, 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 3. The SPLASH-2 programs. Only emphasized programs (marked with ▶) are studied further. Lock statistics are obtained for 32-processor runs.

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 -O1.[2] Table 5 presents the execution times in seconds for 28-processor runs for all eight studied locking schemes.[3] Variance is given in parentheses in the same table. Normalized speedup for TATAS, TATAS_EXP, MCS, CLH, and HBO_GT_SD is shown in Figure 6. For Barnes, the MCS lock is much worse than the ordinary TATAS_EXP, and for Volrend and Water-Nsq that is also the case for the CLH lock. As expected from our new microbenchmark study, on average, queue-based locks perform about the same as the TATAS with exponential backoff.

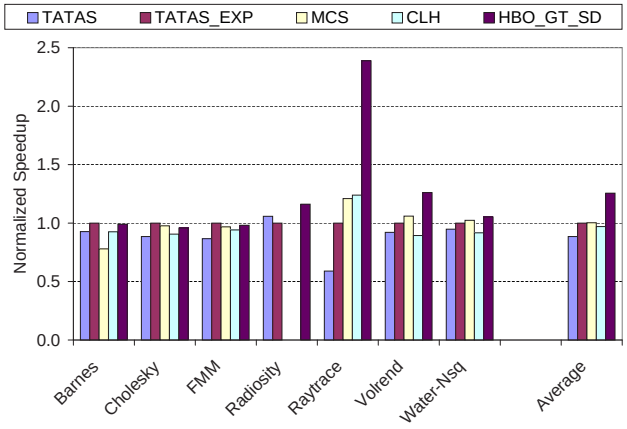

Figure 6. Normalized speedup for 28-processor runs on a 2-node Sun WildFire.

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 [6]. Detailed analysis of Raytrace is beyond the scope of this paper (see [6, 24, 26] for more details). In this application, locks are used to protect task queues; locks are also used for some global variables that track statistics for the program. A large amount of work is usually done between synchronization points. Execution time in seconds for all eight synchronization algorithms for single-, 28-, and 30-processor runs is shown in Table 4.

NUCA-aware locks demonstrate very low measurement variance for both 28- and 30-processor runs. In the same table, we also demonstrate that MCS and CLH locks are practically unusable for 30-processor runs. They are extremely sensitive for small disturbances produced by the operating system itself. Traditionally, original software queuing locks

[2]This is the highest level of optimization that does not break the correctness of execution for all applications and lock implementations.

[3]An unmodified version of Radiosity will not execute correctly with software queuing locks on any optimization level. We did not investigated this any further.

Lock Type	1 CPU	28 CPUs	30 CPUs
TATAS	5.02	2.90 (0.91)	2.70 (0.45)
TATAS_EXP	5.26	1.71 (0.18)	2.05 (0.26)
MCS	5.05	1.41 (0.28)	> 200 s
CLH	5.30	1.38 (0.32)	> 200 s
RH	5.08	0.62 (0.01)	0.68 (0.00)
HBO	5.00	0.77 (0.01)	0.78 (0.01)
HBO_GT	5.02	0.70 (0.01)	0.75 (0.00)
HBO_GT_SD	5.02	0.72 (0.01)	0.80 (0.02)

Table 4. Raytrace performance. Execution time is given in seconds and the variance is presented in parentheses.

exhibit poor behavior in the presence of multiprogramming, because a process near the end of the queue, in addition to having to wait for any process that is preempted during its critical section, must also wait for any preempted process ahead of it in the queue. This unwanted behavior of the queue-based locks has been studied further by Scott on the same architecture and on the Sun Enterprise 10000 multiprocessor [22, 23].

Speedup for Raytrace is shown in Figure 7. There is a significant decrease in performance for all other locks above 12 processors, while the NUCA-aware locks continue to moderately scale all the way up to 28 processors.

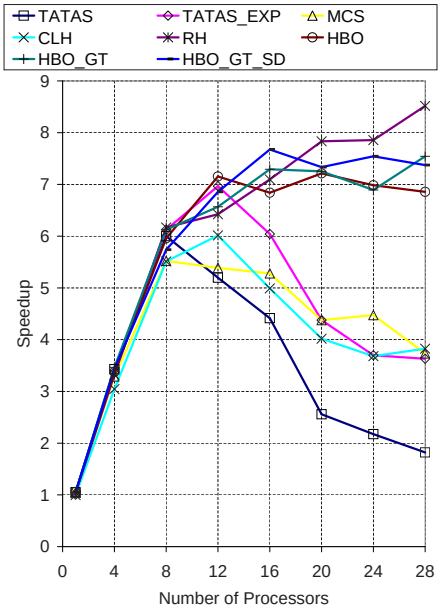

Figure 7. Speedup for Raytrace.

We also present the normalized traffic numbers for all synchronization algorithms in Table 6.

Program	TATAS	TATAS_EXP	MCS	CLH	RH	HBO	HBO_GT	HBO_GT_SD
Barnes	1.54 (0.05)	1.43 (0.01)	1.83 (0.15)	1.54 (0.10)	1.54 (0.14)	1.50 (0.04)	1.69 (0.06)	1.44 (0.10)
Cholesky	2.31 (0.07)	2.04 (0.04)	2.09 (0.03)	2.25 (0.11)	2.23 (0.06)	2.06 (0.09)	2.34 (0.03)	2.13 (0.11)
FMM	4.84 (0.33)	4.19 (0.19)	4.33 (0.06)	4.46 (0.07)	4.27 (0.13)	4.37 (0.09)	4.59 (0.27)	4.27 (0.09)
Radiosity	1.66 (0.06)	1.75 (0.07)	N/A	N/A	1.44 (0.07)	1.45 (0.09)	1.68 (0.04)	1.51 (0.03)
Raytrace	2.90 (0.91)	1.71 (0.18)	1.41 (0.28)	1.38 (0.32)	0.62 (0.01)	0.77 (0.01)	0.70 (0.01)	0.72 (0.01)
Volrend	1.70 (0.03)	1.57 (0.10)	1.48 (0.28)	1.75 (0.16)	1.61 (0.09)	1.68 (0.14)	1.33 (0.10)	1.24 (0.03)
Water-Nsq	2.37 (0.03)	2.25 (0.06)	2.20 (0.04)	2.45 (0.03)	2.21 (0.01)	2.14 (0.03)	2.09 (0.02)	2.14 (0.01)
Average	2.47 (0.21)	2.13 (0.09)	2.22 (0.14)	2.31 (0.13)	1.99 (0.07)	2.00 (0.07)	2.06 (0.08)	1.92 (0.05)

Table 5. Application performance for eight locking algorithms for 28-processor runs, 14 threads per WildFire node. Execution time is given in seconds and the variance is shown in parentheses.

6. Fairness and Sensitivity

The task of the lock-unlock synchronization primitives is to create a serialization schedule for each critical region such that simultaneous attempt to enter the region will be ordered in some serial way. Any serial order will result in a correct execution, as long as starvation is avoided. Fairness is often considered a desirable property, since it can create an even distribution of work between threads. Without fairness, the threads may arrive unevenly at a barrier, even though they have performed the same amount of work. This will force the early arriving threads to wait for the last arriving thread while performing no useful work. However, the importance of fairness must be traded off with its impact on performance [6].

It is not clear that fairness always results in the fastest execution. Assume that all threads are expected to enter the same critical region exactly once between two barriers. All threads arrive roughly at the same time to the critical section. Here, the serialization scheme with the shortest time between two contenders entering the critical section would be preferable over the fairness scheme that strictly scheduled the threads according to their arrival time.

The queue-based locks implement a first come, first served order between simultaneous attempts to enter a critical region and guarantee both fairness and starvation avoidance. The TATAS locks rely on the coherence implementation for its serialization, and can not make such guarantees. They are dependent on such guarantees made by the underlying coherence mechanism. HBO algorithms are based on the TATAS proposals. In addition, they maximize node affinity of the NUCAs, and improve the lock handover time by handing over the lock to a waiting neighbor from the same NUCA node, rather than the thread which has waited the longest time in the system.

A very simple fairness study is performed on a new microbenchmark by measuring the finish times of all individual threads in the benchmark. The results for all lock algorithms are shown in Figure 8. As expected, we can see that the queue-based locks are the fairest locks in this experiment; the percentage difference in completion time between the first and the last processor is only 2.1 percent. TATAS_EXP seems to be most "unfair" lock for this benchmark with the same percentage difference of 28.9 percent. NUCA-aware locks perform about the same, for example, the percentage difference for HBO_GT_SD is 5.6 percent, which is quite close to the difference of software queuing lock implementations.

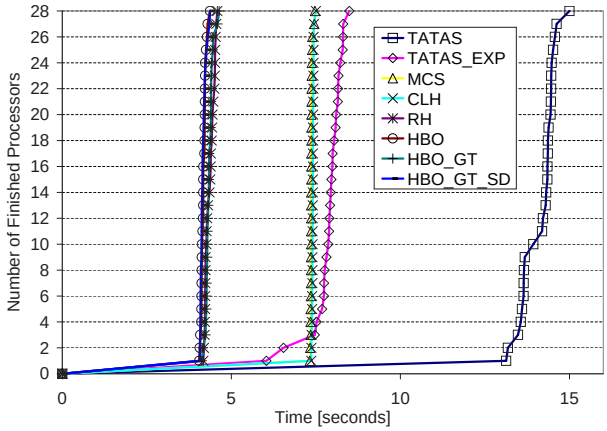

Figure 8. Fairness study.

The sensitivity of the HBO_GT_SD algorithm is studied by running the new microbenchmark algorithm on a 2-node Sun WildFire machine. Results for 26-processor runs are shown in Figure 9 and 10. In Figure 9 we perform the study by varying the `REMOTE_BACKOFF_CAP` parameter (see Figure 1), and in Figure 10, the `GET_ANGRY_LIMIT` parameter is varied (see Figure 2). The MCS and the HBO_GT algorithms are used for comparison.

7. Conclusions

Efficient and scalable general-purpose synchronization primitives should perform well both at high and low contention. Most existing synchronization proposals, such

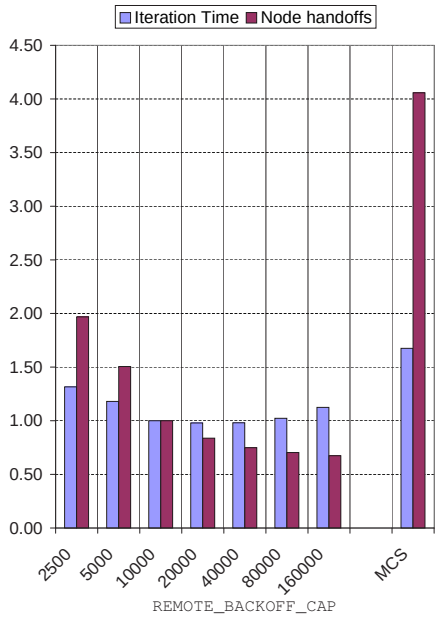

Figure 9. Sensitivity study of the HBO_GT_SD algorithm. The REMOTE_BACKOFF_CAP parameter is examined. The values are normalized, and the MCS is used for comparison.

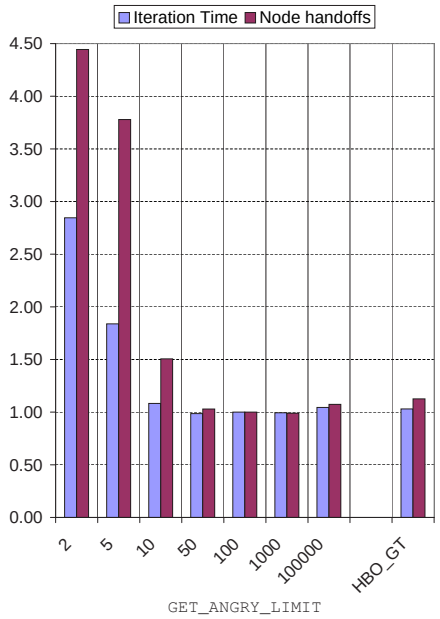

Figure 10. Sensitivity study of the HBO_GT_SD algorithm. The GET_ANGRY_LIMIT parameter is examined. The values are normalized, and the HBO_GT is used for comparison.

as queue-based locks and locks based on various backoff strategies, reduce the traffic generated at high contention while adding a reasonable overhead at low contention.

The node affinity is identified as yet another important property for scalable general-purpose locks. NUCAs, for example CC-NUMAs built from a few large nodes or from CMPs, have a lower penalty for reading data from a neighbor's cache than from a remote cache. Lock implementations that encourage handing over locks to neighbors will improve the lock handover time, as well as the access to the critical data guarded by the lock, but will also be vulnerable to starvation.

In this paper we propose a set of new HBO locks that exploit communication locality and reduce global traffic for contended locks while adding less overhead for uncontested locks than any of the software queue-based lock implementations. We also suggest one simple solution for detecting starvation and lowering the risk of starvation.

A critical section guarded by the HBO locks is shown to take about half the time to execute compared with the same critical section guarded by any other software-based lock. We also investigate the effectiveness and stability of our new locks on a set of real SPLASH-2 applications. The global traffic in the system is significantly reduced for several microbenchmarks and applications.

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 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. This work is supported in part by Sun Microsystems, Inc., and the Parallel and Scientific Computing Institute (PSCI — ψ), Sweden.

References

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

[2] 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.

[3] 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.

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

[5] 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, Feb. 1993.

Program	TATAS	TATAS_EXP	MCS	CLH	RH	HBO	HBO_GT	HBO_GT_SD
Barnes	1.01 / 0.67	**(2.3) 1.00 / 1.00 (1.8)**	1.01 / 0.66	1.14 / 0.78	1.02 / 0.60	0.92 / 0.61	0.92 / 0.62	0.97 / 0.62
Cholesky	0.99 / 1.00	**(14.3) 1.00 / 1.00 (4.6)**	0.96 / 0.87	0.97 / 0.90	0.95 / 0.87	0.96 / 0.90	0.96 / 0.90	0.97 / 0.91
FMM	1.09 / 1.17	**(6.8) 1.00 / 1.00 (3.2)**	0.99 / 0.83	0.97 / 0.80	1.00 / 0.83	0.96 / 0.84	0.99 / 0.89	1.03 / 0.98
Radiosity	1.06 / 1.08	**(4.1) 1.00 / 1.00 (1.9)**	N/A	N/A	1.00 / 0.85	1.01 / 0.89	0.92 / 0.82	0.99 / 0.98
Raytrace	1.15 / 1.24	**(7.8) 1.00 / 1.00 (2.9)**	0.91 / 0.84	1.04 / 0.78	0.86 / 0.49	0.83 / 0.58	0.82 / 0.58	0.81 / 0.64
Volrend	1.02 / 1.07	**(5.2) 1.00 / 1.00 (1.3)**	1.02 / 1.05	1.04 / 1.17	1.01 / 1.03	1.01 / 0.87	1.02 / 0.87	1.01 / 0.86
Water-Nsq	1.01 / 1.03	**(2.8) 1.00 / 1.00 (1.2)**	1.00 / 1.04	1.07 / 1.10	1.03 / 1.02	0.98 / 0.97	0.96 / 0.98	0.99 / 0.98
Average	1.05 / 1.04	**1.00 / 1.00**	0.98 / 0.88	1.04 / 0.92	0.98 / 0.81	0.95 / 0.81	0.94 / 0.81	0.97 / 0.85

Table 6. Normalized traffic (local/global) for all locking algorithms for 28-processor runs. Number of transactions in millions for TATAS_EXP is shown in parentheses.

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

[7] K. Gharachorloo, M. Sharma, S. Steely, and S. V. 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, Nov. 2000.

[8] 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, Apr. 1989.

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

[10] 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, Feb. 1999.

[11] 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.

[12] 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.

[13] 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, Mar. 1992.

[14] 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, Oct. 1994.

[15] 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.

[16] 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, Apr. 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.

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

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

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

[20] Z. Radović and E. Hagersten. Efficient Synchronization for Nonuniform Communication Architectures. In *Proceedings of Supercomputing 2002*, Baltimore, Maryland, USA, Nov. 2002.

[21] R. Rajwar, A. Kägi, and J. R. Goodman. Improving the Throughput of Synchronization by Insertion of Delays. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 168–179, Jan. 2000.

[22] 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.

[23] M. L. Scott and W. N. Scherer. Scalable Queue-Based Spin Locks with Timeout. In *Proceedings of the ACM SIGPLAN 2001 Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, Snowbird, Utah, USA, June 2001. Source code is available for download from `ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/`.

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

[25] 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, Aug. 1996.

[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.