A Hierarchical CLH Queue Lock

Victor Luchangco, Dan Nussbaum, and Nir Shavit

Sun Microsystems Laboratories

Abstract. Modern multiprocessor architectures such as CC-NUMA machines or CMPs have nonuniform communication architectures that render programs sensitive to memory access locality. A recent paper by Radović and Hagersten shows that performance gains can be obtained by developing general-purpose mutual-exclusion locks that encourage threads with high mutual memory locality to acquire the lock consecutively, thus reducing the overall cost due to cache misses. Radović and Hagersten present the first such *hierarchical* locks. Unfortunately, their locks are backoff locks, which are known to incur higher cache miss rates than queue-based locks, suffer from various fundamental fairness issues, and are hard to tune so as to maximize locality of lock accesses.

Extending queue-locking algorithms to be hierarchical requires that requests from threads with high mutual memory locality be consecutive in the queue. Until now, it was not clear that one could design such locks because collecting requests locally and moving them into a global queue seemingly requires a level of coordination whose cost would defeat the very purpose of hierarchical locking.

This paper presents a hierarchical version of the Craig, Landin, and Hagersten CLH queue lock, which we call the *HCLH queue lock*. In this algorithm, threads build implicit local queues of waiting threads, splicing them into a global queue at the cost of only a single CAS operation.

In a set of microbenchmarks run on a large scale multiprocessor machine and a state-of-the-art multi-threaded multi-core chip, the HLCH algorithm exhibits better performance and significantly better fairness than the hierarchical backoff locks of Radović and Hagersten.

1 Introduction

It is well accepted that on small scale multiprocessor machines, queue locks [1,2,3,4] minimize overall invalidation traffic by allowing threads to spin on separate memory locations while waiting until they are at the head of the queue. Their advantage over backoff locks [5] is not only in performance, but also in the high level of fairness they provide in accessing a lock.

Large scale modern multiprocessor architectures such as cache-coherent nonuniform memory-access (CC-NUMA) machines, have nonuniform communication architectures that render programs sensitive to memory-access locality. Such architectures include clusters of processors with shared local memory, communicating with each other via a slower communication medium. Access by a processor to the local memory of its cluster can be two or more times faster than access to the remote memory in another cluster [6]. Such machines also have large per-cluster caches, further reducing the cost of communication between processors on the same cluster. A recent paper by Radović and Hagersten [6] shows that performance gains can be obtained by developing *hierarchical locks*: general-purpose mutual-exclusion locks that encourage threads with high mutual memory locality to acquire the lock consecutively, thus reducing the overall level of cache misses when executing instructions in the critical section.

Radović and Hagersten's locks are simple backoff locks: *test-and-test-and-set* locks, augmented with a *backoff scheme* to reduce contention on the lock variable. Their hierarchical backoff mechanism allows the backoff delay to be tuned dynamically so that when a thread notices that another thread from its own local cluster owns the lock, it can reduce its delay and increase its chances of acquiring the lock. The dynamic shortening of backoff times in Radović and Hagersten's lock introduces significant fairness issues: it becomes likely that two or more threads from the same cluster will repeatedly acquire a lock while threads from other clusters starve. Moreover, because the locks are test-and-test-and-set locks, they incur invalidation traffic on every modification of the shared lock variable, which is especially costly on CC-NUMA machines, where the cost of updating remote caches is higher.

We therefore set out to design a hierarchical algorithm based on the more advantageous queue-locking paradigm. A queue lock uses a FIFO queue to reduce contention on the lock variable and provide fairness: if the lock is held by some thread when another thread attempts to acquire it, the second thread adds itself to the queue, and does not attempt to acquire the lock again until it is at the head of the queue (threads remove themselves from the queue when they execute their critical section). Thus, once a thread has added itself to the queue, another thread cannot acquire the lock twice before the first thread acquires it. Several researchers have devised queue locks [1,2,3,4] that minimize overall invalidation traffic by allowing threads to spin on separate memory locations while waiting to check whether they are at the head of the queue. However, making a queue-lock hierarchical implies that requests from threads with high mutual memory locality be consecutive in the queue. To do so one would have to somehow collect local requests within a cluster, integrating each cluster's requests into a global queue, a process which naïvely would require a high level of synchronization and coordination among remote clusters. The cost of this coordination would seemingly defeat the very purpose of hierarchical locking.

This paper presents a hierarchical version of what is considered the most efficient queue lock for cache-coherent machines: the *CLH queue lock* of Craig, Landin, and Hagersten [2,4]. Our new *hierarchical CLH queue lock* (HCLH) has many of the desirable performance properties of CLH locks and overcomes the fairness issues of backoff-based locks. Though it does not provide *global FIFO ordering* as in CLH locks—that is, FIFO among the requests of all threads—it does provide what we call *localized FIFO ordering*: lock-acquisitions of threads from any given cluster are FIFO ordered with respect to each other, but globally, there is a preference to letting threads from the same cluster follow one another (at the expense of global FIFO ordering) in order to enhance locality.

The key algorithmic breakthrough in our work is a novel way for threads to build implicit local queues of waiting threads, and splice them to form a global queue at the cost of only a single *compare-and-swap* (CAS) operation.

In a bit more detail, our algorithm maintains a *local queue* for each cluster, and a single *global queue*. A thread can enter its critical section when it is at the head of the global queue. When a thread wants to acquire the lock, it adds itself to the local queue of its cluster. Thus, threads are spinning on their local predecessors. At some point, the thread at the head of the local queue attempts to splice the entire local queue onto the global queue, so that several threads from the same cluster appear consecutively in the global queue, improving memory-access locality. This splicing into the global queue requires only a single CAS operation, and happens without the spliced threads knowing they have been added to the global queue (except, of course, for the thread doing the splicing): they continue spinning on their local predecessors. The structure of our lock maintains other desirable properties of the original CLH queue lock: It avoids extra pointer manipulations by maintaining only an implicit list; each thread points to its predecessor through a thread-local variable. It also uses a CLH-like recycling scheme that allows the reuse of lock records so that, as in the original CLH algorithm, L locks accessed by N threads require only $O(N + L)$ memory.

We evaluated the performance of our new HCLH algorithm on two nonuniform multiprocessors: a large-scale Sun Fire$^{\text{TM}}$ E25K[7] SMP (*E25K*) and a Sun Fire$^{\text{TM}}$ T2000, which contains a UltraSPARC® T1[8] 32-thread 8-core multithreaded multiprocessor (*T2000*). In a set of microbenchmarks, including one devised by Radović and Hagersten [6] to expose the effects of locality, the new HCLH algorithm shows various performance benefits: it has improved throughput, better locality, and significantly improved fairness.

In Section 2, we describe our algorithm in detail, in Section 3, we present and discuss the experimental results and we conclude and touch on future work in Section 4.

2 The HCLH Algorithm

In this section, we explain our new queue-lock algorithm in detail. We assume that the system is organized into clusters of processors, each of which has a large cache that is shared among the processors local to that cluster, so that intercluster communication is significantly more expensive than intracluster communication. We also assume that each cluster has a unique *cluster id*, and that every thread knows the `cluster_id` of the cluster on which it is running (threads do not migrate to different clusters). An HCLH lock consists of a collection of `local_queues`, one per cluster, and a single `global_queue`.

As in the original CLH queue lock [2,4], our algorithm represents a queue by an *implicit* linked list of elements of type `qnode`, as follows: A queue is represented by a pointer to a qnode, which is the tail of the queue (unless the queue is empty—see below for how to determine whether the queue is empty). Each thread has two local variables, `my_qnode` and `my_pred`, which are both pointers to qnodes. We say that a thread *owns* the qnode pointed to by its `my_qnode` variable, and we maintain the invariant that at any time, all but one qnode is owned by exactly one thread; we say that one qnode is owned by the lock. For any qnode in a queue (other than the head of the queue), its predecessor in the queue is the qnode pointed to by the `my_pred` variable of its owner. This is well-defined because we also maintain the invariant that the qnode owned by the lock is either not in any queue (while some thread is in the critical section) or is at the head of the global queue. A qnode consists of a single word containing three fields: the `cluster_id` of the processor on which its current owner (or most recent owner, if it is owned by the lock) is running, and two boolean fields, `successor_must_wait` and `tail_when_spliced`. The `successor_must_wait` field is the same as in the original CLH algorithm: it is set to `true` before being enqueued, and it is set to `false` by the qnode's owner upon exit from the critical section, signaling the successor (if any) that the lock is available. Thus, if a thread is waiting to acquire the lock, it may do so when the `successor_must_wait` field of the predecessor of its qnode is `false`. We explain the interpretation of `tail_when_spliced` below.

Threads call the procedure `acquire_HCLH_lock()` when they wish to acquire the lock. Briefly, this procedure first adds the thread's qnode to the local queue, and then waits until either the thread can enter the critical section or its qnode is at the head of the local queue. In the latter case, we say the thread is the *cluster master*, and it is responsible for splicing the local queue onto the global queue. We describe the algorithm in more detail below. Pseudocode appears in Figure 1.

A thread wishing to acquire the lock first initializes its qnode (i.e., the qnode it owns), setting `successor_must_wait` to `true`, `tail_when_spliced` to `false`, and the `cluster_id` field appropriately. The thread then adds its qnode to the end (tail) of its local cluster's queue by using CAS to change the tail to point to its qnode. Upon success, the thread sets its `my_pred` variable to point to the qnode it replaced as the tail. We call this qnode the *predecessor qnode*, or simply the *predecessor*.

Then `wait_for_grant_or_cluster_master()` (not shown) is called, which causes the thread to spin until one of the following conditions is true:

1. the predecessor is from the same cluster, the boolean flag `tail_when_spliced` is `false`, and the boolean flag `successor_must_wait` is `false`, or
2. the predecessor is not from the same cluster or the predecessor's boolean flag `tail_when_spliced` is `true`.

In the first case, the thread's qnode is at the head of the global queue, signifying that it owns the lock and can therfore enter the critical section. In the second case, as we argue below, the thread's qnode is at the head of the local queue, so the thread is the cluster master, making it responsible for splicing the local queue onto the global queue. (If there is no predecessor—that is, if the local queue's tail pointer is null—then the thread becomes the cluster master immediately.) This spinning is mostly in cache and hence incurs almost no communication cost. The procedure `wait_for_grant_or_cluster_master()` (not shown) returns a boolean indicating whether the running thread now owns the lock (if not, the thread is

```
qnode* acquire_HCLH_lock(local_q* lq, global_q* gq, qnode*
my_qnode) {
  // Splice my_qnode into local queue.
  do {
    my_pred = *lq;
  } while (!CAS(lq, my_pred, my_qnode));
  if (my_pred != NULL) {
    bool i_own_lock = wait_for_grant_or_cluster_master(my_pred);
    if (i_own_lock) {
      // I have the lock. Return qnode just released by previous owner.
      return my_pred;
    }
  }

  // At this point, I'm cluster master.  Give others time to show up.
  combining_delay();

  // Splice local queue into global queue.
  do {
    my_pred = *gq;
    local_tail = *lq;
  } while (!CAS(gq, my_pred, local_tail));

  // Inform successor that it is new master.
  local_tail->tail_when_spliced = true;

  // Wait for predecessor to release lock.
  while (my_pred->successor_must_wait);

  // I have the lock. Return qnode just released by previous owner.
  return my_pred;
}

void release_HCLH_lock(qnode* my_qnode) {
  my_qnode->successor_must_wait = false;
}
```

Fig. 1. Procedures for acquiring and releasing a hierarchical CLH lock. The `acquire_HCLH_lock()` procedure returns a `qnode` to be used for next lock acquisition attempt.

the cluster master). It is at this point that our algorithm departs from the original CLH algorithm, whose nodes do not have `cluster_id` or `tail_when_spliced` fields, in which only the first case is possible because there is only one queue. The chief difficulty in our algorithm is in moving qnodes from a local queue to the global queue in such a way that maintains the desirable properties of CLH queue locks. The key to achieving this is the `tail_when_spliced` flag, which is raised (i.e., set to `true`) by the cluster master on the last qnode it splices onto the global queue (i.e., the qnode that the cluster master sets the tail pointer of the global queue to point to).

If the thread's qnode is at the head of the global queue, then, as in the original CLH algorithm, the thread owns the lock and can enter the critical section. Upon exiting the critical section, the thread releases the lock by calling `release_HCLH_lock()`, which sets `successor_must_wait` to `false`, passing ownership of the lock to the next thread, allowing it to enter the critical section. The thread also swaps its qnode for its predecessor (which was owned by the lock) by setting its `my_qnode` variable.

Otherwise, either the predecessor's `cluster_id` is different from mine or the `tail_when_spliced` flag of the predecessor is raised (i.e., `true`). If the predecessor has a different `cluster_id`, then it cannot be in the local queue of this thread's cluster because every thread sets the `cluster_id` to that of its cluster before adding its qnode to the local queue. Thus, the predecessor must have already been moved to the global queue and recycled to a thread in a different cluster. On the other hand, if the `tail_when_spliced` flag of the predecessor is raised, then the predecessor was the last node moved to the global queue, and thus, the thread's qnode is now at the head of the local queue. It cannot have been moved to the global queue because only the cluster master, the thread whose qnode is at the head of the local queue, moves qnodes onto the global queue.

As cluster master, a thread's role is to splice the qnodes accumulated in the local queue onto the global queue. The threads in the local queue are all spinning, each on its predecessor's qnode. The cluster master reads the tail of the local queue and then uses a CAS operation to change the tail of the global queue to point to the qnode it saw at the tail of its local queue, and sets its `my_pred` variable to point to the tail of the global queue that it replaced. It then raises the `tail_when_spliced` flag of the last qnode it spliced onto the global queue, signaling to the (local) successor of that qnode that it is now the head of the local queue. This has the effect of inserting all the local nodes up to the one pointed to by the local tail into the CLH-style global queue in the same order they were in in the local queue.[1] To increase the length of the combined sequence of nodes that is moved into the global queue, the cluster master waits a certain amount of time for threads to show up in the local queue before splicing into the global queue. We call this time the *combining delay*. With no combining delay, we achieved little or no combining at all, since the time between becoming cluster master and successfully splicing the local queue into the global queue was generally so small. By adding a simple adaptive scheme (using exponential backoff) to adjust the combining delay to current conditions, we saw combining rise to the level we hoped for.

Once in the global queue, the cluster master acts as though it were in an ordinary CLH queue, entering the critical section when the `successor_must_wait` field of its (new) predecessor is `false`. The threads of the other qnodes that were spliced in do not know they moved to the global queue, so they continue spinning as before, and each will enter the critical section when the `successor_must_wait` field of its predecessor is `false`. And as in the case above, and in the original CLH algorithm, a thread simply sets its qnode's `successor_must_wait` field to `false` when it exits the critical section.

[1] Note that in the interval between setting the global tail pointer and raising the `tail_when_spliced` flag of the last spliced qnode, the qnodes spliced onto the global queue are in both local and global queues. This is okay because the cluster master will not enter the critical section until after it raises the `tail_when_spliced` flag of the last spliced qnode, and no other thread from that cluster can enter the critical section before the cluster master, since all other threads from that cluster are ordered after the cluster master's in the global queue.

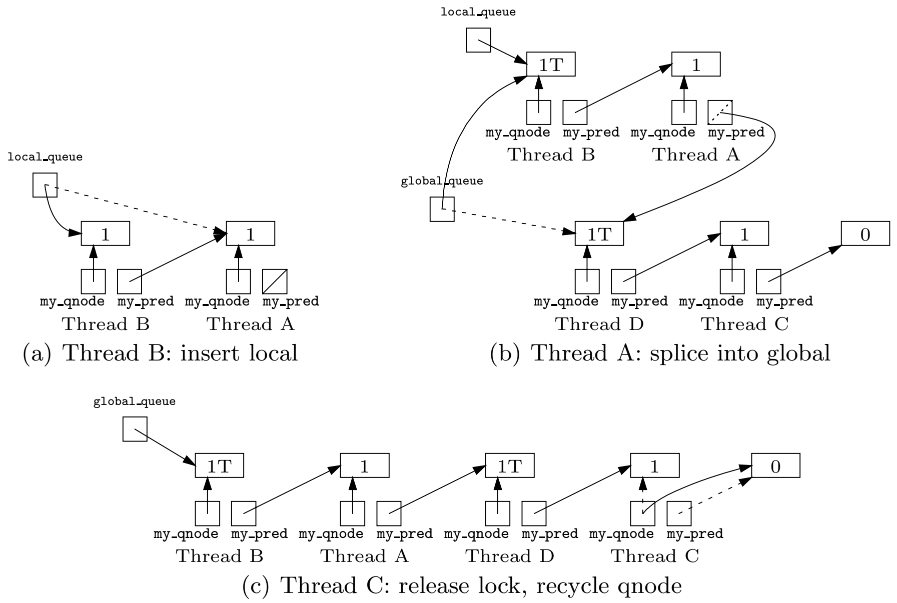
local_queue
1
1
my_qnode my_pred my_qnode my_pred
Thread B Thread A
(a) Thread B: insert local
local_queue
1T
1
my_qnode my_pred my_qnode my_pred
Thread B Thread A
global_queue
1T
1
0
my_qnode my_pred my_qnode my_pred
Thread D Thread C
(b) Thread A: splice into global
global_queue
1T
1
1T
1
0
my_qnode my_pred my_qnode my_pred my_qnode my_pred my_qnode my_pred
Thread B Thread A Thread D Thread C
(c) Thread C: release lock, recycle qnode

Fig. 2. Lock acquisition and release in a hierarchical CLH lock

Figure 2 illustrates a lock acquisition and release in a hierarchical CLH lock. The `successor_must_wait` flag is denoted by 0 (for `false`) or 1 (for `true`) and the raised `tail_when_spliced` flag by a T. We denote the thread's predecessor, local or global (they can be implemented using the same variable), as `my_pred`. The local queue already contains a qnode for a thread A that is the local cluster master since its `my_pred` is null. In part (a), thread B inserts its qnode into the local queue by performing a CAS operation on the local queue's tail pointer. In part (b), thread A splices the local queue consisting of the qnodes of threads A and B onto the global queue, which already contains the qnodes of threads C and D, spliced at an earlier time. It does so by reading the local queue pointer, and using CAS to change the global queue's tail pointer to the same qnode it read in the local queue's tail pointer, and then raising the `tail_when_spliced` flag of this qnode (marked by a T). Note that in the meantime other qnodes could have been added to the local queue but the first among them will simply be waiting until B's `tail_when_spliced` flag is raised (marked by T). In part (c), thread C releases the lock by lowering the `successor_must_wait` flag of its qnode, and then setting `my_qnode` to the predecessor qnode. Note that even though thread D's qnode has its `tail_when_spliced` flag raised, and it could be a node from the same cluster as A, A was already spliced into the global queue and is no longer checking this flag, only the `successor_must_wait` flag.

As can be seen, the structure of the HCLH algorithm favors sequences of local threads, one waiting for the other, within the waiting list in the global queue. As with the CLH lock, additional efficiency follows from the use of implicit

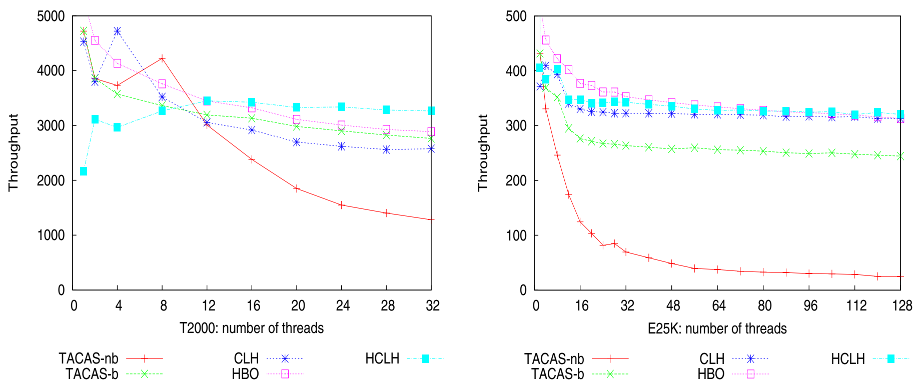

Fig. 3. Traditional microbenchmark performance, measured for each lock type. Results for the T2000 are on the left; those for the E25K are on the right. Throughput is measured in thousands of lock acquire/release pairs per second.

pointers which minimizes cache misses, and from the fact that threads spin on local cached copies of their successor's qnode state.

3 Performance

In this section, we present throughput figures using the same two microbenchmarks suggested by Radović and Hagersten[6]. For lack of space, we present only a subset of the relevant results. A full version of the results can be found in `http://research.sun.com/scalable/pubs/hclh-main.pdf`. In particular, we omit locality data and fairness data (both of which show our algorithm in a good light), along with uncontested-performance data (in which area our algorithm suffers as compared to the others).

These experiments were conducted on two machines: a 144-processor Sun Fire^{TM} E25K[7] SMP (*E25K*) with 4 processor chips per cluster (two cores per chip), and a prototype Sun Fire^{TM} T2000 UltraSPARC® T1[8]-based single-chip multiprocessor (*T2000*) with 8 cores and 4 multiplexed threads per core. We compared the following locking primitives:

TACAS-nb: The traditional test-and-compare-and-swap lock, without backoff.
TACAS-b: The traditional test-and-compare-and-swap lock, with exponential backoff.
CLH: The queue-based lock of Craig, Landin, and Hagersten[2,4].
HBO: The hierarchical backoff lock of Radović and Hagersten[6]. The HBO backoff mechanism allows the backoff parameters to be tuned dynamically so that when a thread that notices that another thread from its own cluster owns the lock, it can reduce the delay between attempts to acquire the lock, thus increasing its chances of acquiring the lock.

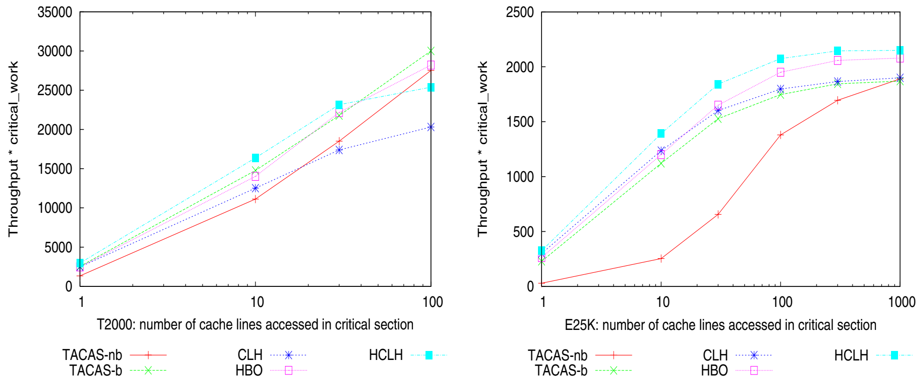

Fig. 4. New microbenchmark performance, measured for each lock type. Results for the T2000 are on the left; those for the E25K are on the right. The Y measures of work per unit time, as given by thousands of lock acquire/release pairs per second multiplied by `critical_work` performed in the critical section. `critical_work` varies along the x-axis. T2000 tests were run with 32 threads; E25K tests were run with 128 threads.

HCLH: Our hierarchical CLH lock. We choose cluster sizes of 8 for the E25K and 4 for the T2000, which make the most sense for the respective architectures.

Results omitted due to lack of space show that HBO's locality is considerably better than random, and HCLH's is considerably better than HBO's for large numbers of (hardware and software) threads. Performance results for the traditional microbenchmark are presented in Figure 3 as run on each platform. This is a variant of the simple loop of lock acquisition and release used by Radović and Hagersten [6] and by Scott and Scherer [9]. As one might expect given its locality advantage, HCLH outperforms all other candidates on both platforms. On the T2000, this superiority asserts itself on tests of 12 or more threads; on the E25K, the effect of improved locality doesn't really assert itself until around 80 threads, and even from there on up, the separation between HCLH, HBO and (somewhat surprisingly) CLH is minimal.

Performance results for our version of Radović and Hagersten's new microbenchmark are presented in Figure 4. In this microbenchmark, each software thread acquires the lock and modifies `critical_work` cache-line-sized blocks of shared data. After exiting from the critical section, each thread performs a random amount of noncritical work. On the E25K, HCLH outperforms the others along the entire range, with HBO close behind, and CLH and TACAS-b not far back. (In fact, CLH slightly outperforms HBO for small `critical_work` values.)

4 Conclusions

Hierarchical mutual-exclusion locks can encourage threads with high mutual memory locality to acquire the lock consecutively, thus reducing the overall level of cache misses when executing instructions in the critical section. We present the HCLH lock—a hierarchical version of Craig, Landin, and Hagersten's queue lock—with that goal in mind.

We model our work after Radović and Hagersten's hierarchical backoff lock, which was developed with the same ends in mind. We demonstrate that HCLH produces better locality and better overall performance on large machines than HBO does when running two simple microbenchmarks.

Compared with the other locks tested (including HBO), the HCLH lock's uncontested performance leaves something to be desired. We have achieved some preliminary success in investigating the possibility of bypassing the local queue in low-contention situations, thus cutting this cost to be near to that of CLH, which is only slightly worse than that of HBO and the others. This is a topic for future work.

Acknowledgements

We wish to thank Brian Whitney for getting us access to the E25K machine and Rick Hetherington for providing us architectural details of the T2000 machine.

References

1. Anderson, T.: The performance implications of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel and Distributed Systems **1**(1) (1990) 6–16
2. Craig, T.: Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Dept of Computer Science (1993)
3. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Computer Systems **9**(1) (1991) 21–65
4. Magnussen, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Proc. 8th International Symposium on Parallel Processing (IPPS). (1994) 165–171
5. Agarwal, A., Cherian, M.: Adaptive backoff synchronization techniques. In: Proc. 16th International Symposium on Computer Architecture. (1989) 396–406
6. Radović, Z., Hagersten, E.: Hierarchical Backoff Locks for Nonuniform Communication Architectures. In: HPCA-9, Anaheim, California, USA (2003) 241–252
7. Sun Microsystems: Sun Fire E25K/E20K Systems Overview. Technical Report 817-4136-12, Sun Microsystems (2005)
8. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: A 32-way multithreaded sparc processor. IEEE Micro **25**(2) (2005) 21–29
9. Scott, M., Scherer, W.: Scalable queue-based spin locks with timeout. In: Proc. 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming. (2001) 44–52