Experiences with Locking in a NUMA Multiprocessor Operating System Kernel

Ronald C. Unrau* Orran Krieger Benjamin Gamsa Michael Stumm
Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto
Email: `unrau@eecg.toronto.edu`

Abstract

We describe the locking architecture of a new operating system, HURRICANE, designed for large scale shared-memory multiprocessors. Many papers already describe kernel locking techniques, and some of the techniques we use have been previously described by others. However, our work is novel in the particular combination of techniques used, as well as several of the individual techniques themselves. Moreover, it is the way the techniques work together that is the source of our performance advantages and scalability. Briefly, we use:

- a hybrid coarse-grain/fine-grain locking strategy that has the low latency and space overhead of a coarse-grain locking strategy while having the high concurrency of a fine-grain locking strategy;
- replication of data structures to increase access bandwidth and improve concurrency;
- a clustered kernel that bounds the number of processors that can compete for a lock so as to reduce second order effects such as memory and interconnect contention;
- Distributed Locks to further reduce second order effects, with modifications that reduce the uncontended latency of these locks to close to that of spin locks.

1 Introduction

The question of how to structure locks within an operating system is important, because it directly affects both the available concurrency and the latency of operating system services. The correct choice of locking strategy for a particular data structure or subsystem depends on the expected access pattern and the overall system workload. In a shared-memory multiprocessor environment, we need to efficiently support a workload consisting of either parallel applications or multiple sequential applications or both. These workloads result in four types of access behaviors for operating system data structures: 1) non-concurrent accesses, 2) concurrent accesses to independent data structures, 3) concurrent, read-shared accesses, and 4) concurrent, write-shared accesses.

A Unix application workload consisting of many sequential applications will primarily induce the first two types of access behaviors. Much of the existing work on operating system locking issues has focused on these types of workloads. Parallel applications, on the other hand, primarily induce the second and third type of behavior. The third and fourth types of behaviors are in some ways the most important, however, as these can induce worst-case behavior in the operating system.

In this paper, we describe a locking architecture that addresses all four types of access behaviors. It uses a *hybrid approach*, which combines properties of both coarse-grained and fine-grained locks. The coarse-grained locks minimize the number of atomic operations needed in the critical path of non-concurrent operations. Minimizing the latency of uncontended locks in the critical path is important, because it can constitute a significant portion of the overall response time of an operation. In our system, for example, the measured time for a simple page fault is 160 μsec, of which 40 μsec is attributable to lock overhead. Fine-grained locks, on the other hand, provide the high degree of concurrency needed for concurrent, independent operations. Further, we employ a technique called hierarchical clustering, which replicates data that is primarily read-shared so as to increase overall lock bandwidth, and bounds the contention on shared structures by constraining the number of processors that can access the structure. Finally, we make extensive use of Distributed Locks proposed by Mellor-Crummey and Scott [19], in order to reduce second-order effects for those cases where contention cannot be otherwise avoided. We have improved on the basic algorithm and optimized Distributed Locks for use in a kernel environment.

The design of a locking architecture is heavily dependent on the parameters of the system environment for which it is targeted. In our case, the design is influenced by

*currently at IBM Canada

the fact that 1) atomic operations are expensive relative to normal memory accesses, 2) `swap` is the only atomic operation supported, 3) our operating system is an exception-based micro-kernel (as opposed to process-based [6]), and 4) many of the kernel data structures are left uncached because our hardware does not support cache-coherence. Nevertheless, we believe that elements of our architecture are relevant to a wide variety of system architectures. It should also be noted that many of the techniques we use have been proposed previously. However, the strength of our approach lies in the particular combination of techniques used to efficiently support the four access patterns described above.

Section 2 of this paper describes our general locking architecture. Section 3 describes our improvements to, and experiences with, Distributed Locks. In Section 4 we present performance results from our system. This is followed in Section 5 by a discussion of how our approach to locking might generalize to other systems, and a discussion of our ongoing work. Finally, we conclude in Section 6 with a summary of this paper.

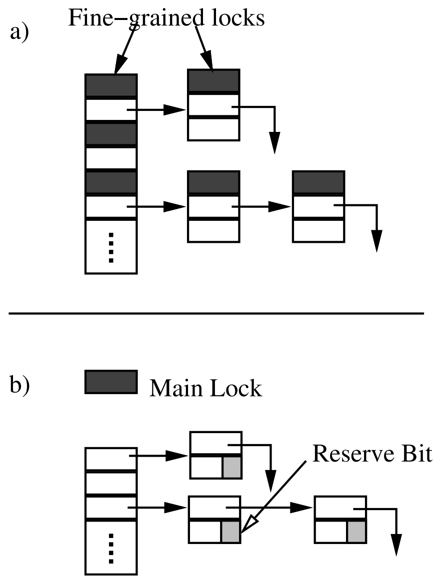

Figure 1: A chained hash table in a) a system with fine-grained locks, and b) the hybrid locking of HURRICANE. The boxes marked with the dark shading indicate locks that are set with atomic operations. The boxes marked with light shading are reserve bits.

2 Locking Architecture

This section describes the locking architecture of the HURRICANE [8, 13, 25, 26] operating system. Three key features distinguish this architecture: a mix of coarse and fine grained locks are used to achieve low latency while still supporting high concurrency for independent operations; hierarchical clustering is used to limit contention by replicating data structures and constraining the number of processors that can directly access a particular data structure; and an optimistic deadlock avoidance protocol is used to reduce common case latency.

2.1 A Hybrid Approach

Our Hybrid approach uses coarse-grained locks, where a single lock may be used to protect several data structures but may only be held for short periods of time, and it uses "light-weight" fine-grained locks to protect data for longer periods of time but at a much finer granularity.

Consider the chained hash table of Figure 1. In a system using fine-grained locks (Figure 1a), each bin would have its own lock to serialize updates to the hash chains, and each hash entry would have one or more locks to protect the data therein. With our hybrid approach, the entire hash table might be protected by a single coarse-grained lock. Using coarse-grained locks in this way has both advantages and disadvantages with respect to space and time. Clearly the number of locks required to access a data structure is minimized, but that alone does not minimize the locking time, except in the case of no contention. The challenge is thus to keep locking time low as concurrency is increased, while still minimizing the number of locks held. The remainder of this section considers the trade-offs involved.

One disadvantage of a coarse-grained lock is that concurrent accesses to different elements are protected by the same lock, causing unnecessary contention for independent operations. Our first step to resolve this issue was to use Distributed Locks. Distributed Locks allow processors to spin locally while waiting for a lock, thereby removing the second order contention effects caused by spinning over the inter-connection network. The additional traffic on the network and memory caused by remote spinning not only slows down other non-contending processors, but also slows the processor that is holding the lock, extending the length of its critical section and exacerbating the contention problem. Although a Distributed Lock requires more space than a spin lock (an additional two words per actively spinning processor), a fine-grained approach would require one spin lock per hash element, a much higher total cost.

Figure 1b) shows how we allow increased concurrency by holding the main lock only long enough to search the hash table and set a *reserve* bit in the required element, after which the coarse-grained lock is released.[1] Other processors waiting for the reserved element spin on the reserve bit (with exponential back-off). When the bit is released, the waiting processors re-acquire the coarse-grained lock and search again.[2]

[1] Our hybrid locking approach is in some ways similar to the locking strategy used by Peacock, et al, for locking cache elements in a multiprocessor version of System V Release 4 [12, 21].

[2] Currently in our kernel, memory used for an object is always reused for objects of the same type. Hence, there is no danger that a process could

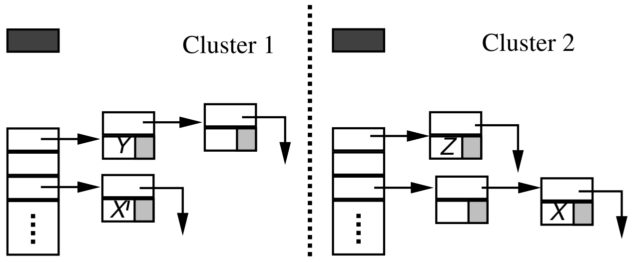

Figure 2: A chained hash table in a system with 2 clusters. Each cluster has a separate instantiation of the hash table and the locks that protect it. Deadlock must be avoided when a cluster requires a lock in two clusters simultaneously, for example, to instantiate a new copy of element Z on cluster 1.

The reserve bit is in essence a fine-grained spin lock, except that 1) it requires only one bit of storage instead of a whole word and is typically co-located with other needed status information, 2) multiple reserve bits can be acquired while the main lock is held, and atomic operations are not required to do so, and 3) it plays a special role in our deadlock avoidance strategy (described in Section 2.3). The fine-grained spin locks, however, can be subject to the contention effects that result from the bursty accesses we expect in a multiprocessor environment. We counteract this problem by controlling the number of processors that can simultaneously access the data structure and by replicating the data structure, as described next.

2.2 Hierarchical Clustering

To control the concurrency demands on the kernel data structures, we use a technique called hierarchical clustering. Briefly, hierarchical clustering is a framework for managing locality in a scalable shared-memory multiprocessor [25]. Instead of having one set of system data structures shared by all processors in the system, the processors are grouped into *clusters* and a complete set of system data structures are instantiated within each cluster. With this framework, the read-mostly data is replicated onto each cluster, and write-shared data exists on only one cluster, possibly being migrated from one cluster to another.

Figure 2 shows how the hash table of Figure 1 might appear in a two cluster system. A separate instance of the hash table, each with its own lock, exists in each cluster. The hash tables are used to hold both replicated entries (if they are primarily read-only) and non-replicated entries. In Figure 2, X is replicated to cluster 1 and cluster 2, while there is only a single copy of Y and Z in the system. Only processors in the cluster may access the cluster-local hash table. To access entries of a hash table in another cluster, a remote procedure call (RPC) must be made to that cluster.[3]

Replicas of primarily read-shared data are typically made on demand. Hierarchical clustering supports efficient replication management by organizing the system into a tree. In the simplest case, the tree has three levels, with processors at the leaves, clusters forming intermediary nodes, and a logical top of tree for the entire system. The act of replication is made more efficient by combining multiple simultaneous requests from the processors in one cluster into a single request to the target cluster. This is accomplished by always first creating a local (reserved) instance before performing the replication. This local instance acts as a place-holder until the real data is obtained, preventing redundant replication requests from being issued. Combining is important because it is common that the many processes of a large SPMD program make concurrent demands that require the same (read shared) data.

The hierarchical clustering tree is also used to efficiently support the broadcast of modifications or invalidations when a replicated data structure is changed. For most data structures, the change is simply sent in parallel to each cluster. For data structures that are replicated per-processor, such as page tables in our system, the change is further broadcast within each cluster to each processor affected.

From the point of view of locking, hierarchical clustering provides two major benefits. First, it bounds contention for both coarse-grained locks and reserve bits, since RPCs are used instead of shared memory to access remote entries (Chaves et al discuss some of the trade-offs between using RPC and shared-memory [3]). Second, it increases lock bandwidth by i) instantiating per-cluster system data structures (such as the hash table in Figure 2), each with its own coarse-grained lock, and by ii) replicating read-shared objects (such as X in Figure 2), each with its own reserve bit.

Although hierarchical clustering bounds lock contention and increases lock bandwidth, it does complicate locking protocols [26]. For example, a needed data structure may not be present in the local cluster, requiring a remote operation to get it. If the data is replicated, then it must be kept consistent, which also requires remote operations. If the local kernel is holding any locks when initiating a remote operation, then some protocol must be established so that deadlock does not occur.

2.3 Deadlock Management Protocols

This section describes the protocol used to prevent deadlock across the clusters of a hierarchically clustered system. The protocol applies only to locks within the same class (e.g., the

spin indefinitely because the memory has been reallocated for another type of object that happens to have the same bit permanently set. If memory was arbitrarily recycled, then it would be necessary to either (i) periodically check, while spinning, that the data structure is still the one being sought, or (ii) use reference counts [1] to ensure that objects are not deallocated while a process is spinning on them.

[3] In theory, any processor of the target cluster may be used to execute the RPC. In our implementation, RPCs from the ith processor in the source cluster are always directed to the ith processor in the target cluster so as to roughly balance the RPC load.

class of process descriptor locks); we use a lock hierarchy across classes.

Consider again the clustered hash table of Figure 2. Assume a local copy of element Z is required by cluster 1. A search of the local hash table reveals that it is not present, at which point a data specific location resolution technique [25] indicates that the item resides in cluster 2. The kernel on cluster 1 then allocates a local instance of the element, Z', and initiates an RPC call to cluster 2 to retrieve the data. The local instance is created before the remote access is started so that other processors within the cluster that try to access Z while the remote operation is in progress will not also initiate (redundant) remote requests.

Because of the symmetric relationship between clusters, deadlock can result at this point if the RPC to retrieve Z is started without releasing the locally held locks. Our initial pessimistic deadlock prevention protocol required the initiator to release all locks before initiating the RPC, and then to re-acquire them once the remote operation had completed. This added considerable overhead to the code. Because the data structures were unprotected and could be modified or even removed while the local locks were released, the kernel had to search the hash table again to re-establish the continued existence of element Z after the RPC had completed. In addition, the kernel had to be prepared to handle the case where the data was no longer present. In the common case this re-establishment of state was unnecessary, since the probability of the data having been migrated or destroyed was very small.

To avoid the overhead of re-establishing state for the common case, we have implemented an optimistic algorithm that avoids deadlock and is similar to that described by Paciorek [20]. Before releasing the local locks, a reserve bit is set in any structure that might be needed after completing the call. The reserve bit may act as an exclusive lock or a reader-writer lock, depending on the data it protects. The local locks are then released and the RPC is initiated. If a reserve bit is encountered during processing in the remote cluster, the RPC operation fails and returns with an indication that a potential deadlock situation exists. The local reserve bits set earlier are then released and the remote operation is retried until it succeeds. Note that the local state must be re-established only if a retry is necessary.

Our optimistic approach saves us from having to re-establish state in the common case, although not for every case. In practice, we have found that retries are seldom needed, and the need to restart an operation because of modified state occurs even less frequently. Our approach does have two disadvantages, however. First, we have found that the deadlock avoidance algorithm required us to have two versions of code in many cases: one version has release/retry handling (used when executing as the target of a RPC); and one allows spinning without release/retry handling. Second, by requiring a remote request to continuously retry the RPC until it succeeds, the probability of being able to acquire a remote reserve bit is lower than for requesters that are in the target cluster and hence can spin directly on the reserve bit. Remote processors therefore have a greater potential of being starved for resources that are over-committed.

Remote operations must be retried even when the local locks and reserve bits have all been released, because the processor itself is effectively a locked resource (that could participate in a deadlock cycle) in an exception-based kernel. This is particularly apparent in our use of RPCs. Consider once again the hash table in Figure 2, and suppose processor P_1 in cluster 1 would like to modify X globally. Consider the situation where P_1 has dropped its local locks and reserve bits in order to retry the request, but the processor it directs the RPC to, say processor P_2, has the reserve bit for X set and is currently performing an RPC to another cluster. If the RPC from processor P_1 now executing on processor P_2 were to spin on X's reserve bit on processor P_2, deadlock would result. The deadlock cycle is caused by processor P_1's RPC holding the processor as it spins waiting for the reserve bit, while processor P_2 holds the reserve bit while waiting for the processor to be released.

Although our algorithm has been presented in the context of a clustered system, it is important to note that the same protocol could be applied to any system that requires multiple locks simultaneously. Also, we chose not to use the more common global ordering approach [22] within a particular class of locks, because the only ordering that makes sense in a clustered system is by cluster number. Since remote cluster operations are in general uniformly distributed across clusters, we would still have to release locks to preserve the correct ordering, and we would still require the ability to roll-back and restart.

We have observed two situations where modifications occur often to data required for the completion of the operation while their locks are released: copy-on-write page faults and program destruction. Both situations occur infrequently relative to other kernel operations, and of course are only a concern with large applications that span multiple clusters. Nevertheless, they will perform less efficiently on average because of the overhead of the retries.

2.4 Advantages of our Approach

The four classes of workload an operating system must handle are: 1) non-concurrent requests, 2) concurrent independent requests, 3) concurrent requests to read-shared operating system resources, and 4) concurrent requests to write-shared operating system resources. We have found that our hybrid locking approach, in conjunction with hierarchical clustering, allows us to effectively address all four workload classes.

Non-concurrent requests: The only important goal for this workload is to minimize latency. With our hybrid locking strategy, many kernel requests require only a single atomic operation. Hence, HURRICANE is able to achieve uncontended response times comparable to uniprocessor systems [14].

Concurrent independent requests: The important goal for this workload is to maximize concurrency, so an optimal strategy would be to use fine grain locks for these requests. With our hybrid locking strategy, the reserve bits serve as fine grain locks to maximize concurrency and minimize the time that the coarse-grained locks are held. By using hierarchical clustering, the number of processes concurrently contending for a coarse-grained lock is bounded, so, for this workload, HURRICANE is able to achieve performance comparable to systems structured using only fine grain locks. This is demonstrated in the results section.

Concurrent requests to read-shared resources: Hierarchical clustering allows us to instantiate multiple instances of read-mostly data structures to increase the access bandwidth to the data. Since requests from SPMD applications to shared resources can be bursty, it is important that the replication be done efficiently; hierarchical clustering creates a combining tree to reduce the demand on the source data structure, should many processors wish to make copies of the data structure simultaneously.

Concurrent requests to write-shared resources: Although kernel resources are seldom actively write-shared in our system, it is still important to minimize second order effects for those cases where write-sharing does occur. Since accesses to shared data in remote clusters typically occur via RPC calls, the number of processors competing for a write-shared data structure is bounded by the number of processors in the cluster.

2.5 Experiences using our approach

In the previous sections, we discussed our general locking methodology from an architectural perspective. Naturally, when applying a general methodology to a particular situation, one often makes adaptations and optimizations to accommodate particular uses. In this section we describe the more interesting lessons we learned from applying our general techniques to a full operating system implementation.

Pessimistic vs. Optimistic

In a number of cases we found it advantageous to use a pessimistic (i.e., release all locks, including reserve bits, prior to making a remote request) rather than an optimistic locking strategy, primarily for reasons of simplicity. For example, although we use an optimistic strategy for many data structures when creating local replicas, we typically use a pessimistic strategy for global updates. The optimistic approach is preferable for the former case, since it allows us to use the combining tree approach discussed in Section 2.2. The pessimistic approach is generally preferable for updates that may be broadcast to many clusters: if a processor in one cluster asks a processor in another cluster to broadcast modifications to data for which it also has a copy, then it is obviously better to have the local copy unlocked from the start.

Hybrid Compromises

Although we generally use the hybrid coarse-grain/fine-grain locking strategy as described, we do not follow it religiously. Our kernel has some data structures that are protected by coarse-grained locks and have no fine-grained locks, and in some cases, we have found it advantageous to split coarse-grained locks to achieve somewhat greater concurrency.

Retries

The rationale for using an optimistic approach is to trade off performance under contention (possibly requiring retries) for performance under light load (allowing locks to be safely held during remote operations). We found not only that retries are rarely required, but in those cases where they are required they would still have been required using a pessimistic approach.

For example, with SPMD programs, simultaneous faults to copy-on-write pages raise a number of potential deadlock situations that require retries with the optimistic approach. However, because a copy-on-write fault requires instantiating a new private page to replace the current shared page, the pessimistic approach would likely find that its copy of the page had disappeared by the time it completed its remote operation, requiring it to re-search its data structures and re-issue the request.

A second example can be found in the destruction of parallel programs containing many processes. Hurricane maintains a family tree of processes in the system, where the links of the tree run through the process descriptors. When a process in the application is to be destroyed, multiple process descriptors in different clusters must be updated to remove the process from the tree. Since all processes of an application are destroyed at approximately the same time, retries are common, independent of the strategy chosen.

Data structure design

One lesson we learned from the case of program destruction was that combining two structures with different locking characteristics into a single entity can lead to many concurrency control problems. In this particular case, the problem came from the fact that program destruction can involve up to three process descriptors and has a natural lock ordering that follows the structure of the tree, while process descriptors are also used to implement message passing which always involves two arbitrarily related processes, with no natural ordering. Had the family tree been implemented as a separate data structure, it would have been possible to exploit the hierarchy of the tree to enforce a lock ordering that would have allowed us to avoid the RPC retries described above.

3 Using Distributed Locks

Distributed Locks [19] are used in our system primarily for per-cluster coarse-grained locks, since cross-cluster interactions most often occur through RPCs. Distributed Locks are particularly well-suited for NUMA shared-memory multiprocessors and can substantially reduce the second-order effects stemming from the memory and inter-connection network contention that occurs when processors spin on remote memory. Distributed Locks build a queue of processors waiting to acquire a lock. Second-order contention effects are reduced because waiting processors spin on their local queue elements, instead of across the interconnection network. The queue also has the benefit that accesses to the lock are distributed fairly, since processors are queued in order of arrival. The remainder of this section describes several interesting lessons we have learned from using Distributed Locks.

3.1 Latency in the uncontended case

The high uncontended latency of Distributed Locks relative to spin locks was originally a concern to us, since other researchers had found that it could be as much as twice as high as that of simple spin locks [15]. One way to address this problem is to use an adaptive technique, where the locks switch between spin and distributed locks, depending on the amount of contention observed [2, 15]. We instead found that two simple modifications to the original distributed locking algorithm could improve the uncontended latency to make it competitive with that of simple spin locks (on our system), while preserving the advantages of distributed locks in the contended case.

The original and modified distributed locking algorithms are shown in Figures 3a and 3b, respectively. The first modification removes the code that initializes the per-processor local structure from the critical path of the uncontended case (i.e., the first dashed box in Figure 3a). This was done by requiring the per-processor queue structure to be initialized prior to the first request to the lock, and by re-initializing the structure when it is modified, which occurs only when there is contention for the lock. The code added is highlighted in Figure 3b.

The second modification to the Distributed Lock algorithm removes the condition in `release_lock`, which determines whether another processor has since added itself to the queue, and which is executed just prior the execution of the `compare_and_swap` to release the lock in the uncontended case (i.e., the second dashed box in Figure 3a). The check was there as an optimization for the contended case, assuming local memory accesses are much cheaper than remote accesses. However, this check degrades the performance of the common case where the lock is uncontended. Removing the check does not affect the scalability of the algorithm, since it adds only a constant overhead to the case where there is contention.

With these two modifications the uncontended latency on HECTOR improved from 5.40 μsec to 3.69 μsec — an improvement of 32%. The optimized time now compares favorably to the uncontended spin lock time of 3.65 μsec, the algorithm of which is shown in Figure 3c. These results are described in more detail in Section 4.1.1.

3.2 TryLock

As described by Mellor-Crummey and Scott, Distributed Locks do not support a *TryLock* operation. TryLock makes a single attempt to obtain a lock, and returns either with the lock held, or with a failure code if the lock is not free. In operating system kernels, TryLocks are typically used by the interrupt handlers, which cannot wait for a lock in case it is held by the pre-empted process. In our system, interrupts are used not only for devices, but also for invoking RPCs. In the case of an RPC, if a TryLock fails then the invoking processor is returned an error and retries the operation.

Our first attempt to extend the basic Distributed Locking algorithm to support TryLock took advantage of the fact that the local queue structures could be pre-allocated on a per-processor basis, one for each coarse-grained lock. The interrupt handler checks whether the pre-allocated local queue element is in use before it enqueues itself; if the queue element is free, then it is certain it did not interrupt a current holder of the lock and can therefore safely wait for the lock to be released. While this does not implement a true TryLock (because the interrupt handler will enqueue itself and wait rather than returning immediately if the lock is held), it does prevent deadlock and has the advantage of allowing the interrupt handler to acquire the lock under all conditions except when it clearly cannot, namely when it has interrupted the lock holder. Unfortunately, this implementation of TryLock required a flag in the local queue structure that had to be modified both when acquiring and releasing the lock, and hence had a negative impact on the base performance of our distributed locks in the uncontended case.

We developed a second variant of the Distributed Locking algorithm, which also supported TryLock (this time a true TryLock) but which only added overhead to `release_lock` in the contended case. The new algorithm is similar to `acquire_lock` of Figure 3b, except that it uses a separate local queue structure just for interrupt handlers. If an interrupt handler discovers that the lock is already held after

```
type qnode = record
  next : ^qnode
  locked : Boolean
type lock = ^qnode

procedure acquire_lock( L: ^lock, I : ^qnode )
  I−>next := nil
  predecessor : ^qnode := fetch_and_store( L, I )
  if predecessor != nil
    I−>locked := true
    predecessor−>next := I
    while I−>locked do <nothing>

procedure release_lock( L: ^lock, I: ^qnode )
  if I−>next = nil
    if compare_and_swap( L, I, nil )
      return
    while I−>next = nil do <nothing>
  I−>next−>locked := false
```

a) MCS distributed locks

```
type qnode = record
  next : ^qnode
  locked : Boolean
type lock = ^qnode

procedure init_qnode( I : ^qnode )
  I−>next := nil

procedure acquire_lock( L: ^lock, I : ^qnode )
  predecessor : ^qnode := fetch_and_store( L, I )
  if predecessor != nil
    I−>locked := true
    predecessor−>next := I
    while I−>locked  do <nothing>


procedure release_lock( L: ^lock, I: ^qnode )
  if compare_and_swap( L, I, nil )
    return
  while I−>next = nil do <nothing>
  I−>next−>locked := false
  I−>next := nil
```

b) modified distributed locks

```
type lock = (unlocked, locked)

procedure acquire_lock(L : ^lock)
  while test_and_set( L ) = locked
    delay : integer := 1
    while delay < MAX_DELAY
      Delay( delay )
      if test_and_set( L ) != locked
        return
      delay := delay * 2

procedure release_lock(L : ^lock)
  lock^ := unlocked
```

c) exponential backoff locks

Figure 3: Locking algorithms used by HURRICANE

having enqueued itself, then it returns with an error code (rather than spinning), leaving its local queue structure still in the queue. The queue structures from failed TryLock requests are garbage collected by `Release_lock` operations. This implementation of TryLock is similar to the timeout mechanism for the queueing lock, developed independently by Craig [5].

Unfortunately, we found that this second variant of TryLock discriminated against RPC operations and favored local operations. In hindsight, we realized that this use of TryLock was fundamentally incompatible with Distributed Locks, since Distributed Locks are inherently fair, while retry-based locking is only probabilistically fair. That is, if a lock is saturated, then a Distributed Lock's `release_lock` operation will always hand-off the lock to some local processor that is waiting in the queue, keeping the lock permanently held; remote requests using TryLock will never see the lock free.

An alternative to using TryLock for RPCs is to disable interrupts while the lock is held, thus preventing RPCs from getting through. This way, the RPC interrupt handler can be sure it cannot deadlock with the processor it interrupted. Unfortunately, our hardware only provides the ability to enable and disable all interrupts, and for a number of reasons the HURRICANE kernel always runs with interrupts on. We therefore adapted a strategy first suggested by Stodolsky et al [23].

Inter-processor interrupts are treated as a separate interrupt class that can be logically masked. A per-processor flag is set whenever a lock is about to be acquired that could cause deadlock with an interrupt handler. An interrupt handler always first checks the flag, and if clear, can safely queue for the lock. If, on the other hand, the flag is set, then the interrupt handler enqueues a record of the work to be done on a per-processor work queue. When the flag is cleared, the queue is checked and any pending work is immediately completed. Because the flag and the queue are accessed strictly locally, they can be cached effectively.

The per-processor flag acts as a lock for the processor, placed at the top of the lock hierarchy: it must be acquired before any other lock can be acquired. For RPCs, it allows fair access to the processor, because work is enqueued for later execution whenever the interrupt handler finds the processor locked in a manner similar to the way processors enqueue themselves on Distributed Locks.

In retrospect, it may have been better to combine the work queue with our second TryLock variant, rather than adding the additional per-processor flag to the top of the lock hierarchy.

4 Experimental Results

In this section, we use synthetic stress tests to demonstrate the performance of our locking architecture. The experiments were run on a fully configured version of HURRICANE with all servers active, but with no other applications running at the time. The operating system was running on a 16 processor HECTOR prototype with 16 MHz MC88100 processors [27]. The particular hardware configuration used in our experiments consists of 4 processor-memory modules per station (a shared bus) and 4 stations connected by a ring. This causes access times to vary from 10 cycles for a local (on-board) access, to 19 cycles for an on-station access, and 23 cycles for a cross-ring access.

4.1 Basic locking performance

We first present performance results for the three locking algorithms of Figure 3 in the absence of contention, and then show their performance as the locks become con-

	Atomic	Mem.	Reg.	Br.
MCS	2	2	3	5
H1-MCS	2	1	3	5
H2-MCS	2	0	3	4
Spin	2	0	1	3

Figure 4: Instruction counts required to execute a lock/unlock pair for the various routines in the absence of contention. MCS is the unmodified Mellor-Crummey and Scott Distributed Lock algorithm; H1-MCS is the MCS algorithm with our first modification, that removes the initialization code; H2-MCS is the H1-MCS algorithm with the conditional test in the unlock removed; Spin is the exponential backoff spin lock algorithm. *Atomic* are atomic `read_modify_write` instructions (swap instructions in our case); *Mem* are loads or stores to memory; *Reg* are single-cycle register-to-register instructions; *Br* are branch instructions (including return).

tended. Our processors only support `fetch_and_store` instructions (and not `compare_and_swap`). Therefore, we use Mellor-Crummey and Scott's `fetch_and_store` variant of their Distributed Lock algorithm in these experiments. Using this variant of the algorithm only impacts the performance of the contended case, as described in Section 4.1.2.

4.1.1 Uncontended performance

We measured the performance of the three locking algorithms by measuring the average time to acquire and release a lock 10^6 times. The uncontended latency of exponential backoff spin locks (Figure 3c) varies between 3.65 μsec and 4.63 μsec, depending on the distance between the process requesting the lock and the lock variable. The latency of the unmodified Distributed Locks (Figure 3a) varies between 5.40 μsec and 6.02 μsec. With our first modification that eliminates the initialization code, latency improves to between 4.56 μsec and 5.33 μsec, and with our second modification that also removes the condition code, the uncontended latency further improves to between 3.69 μsec and 4.63 μsec.

The instruction counts for the three locking algorithms, obtained by inspecting the assembly code, are shown in Figure 4.[4] While the modified Distributed Lock algorithm (H2-MCS) has the same number of atomic operations and memory accesses as the spin lock algorithm, it should have five additional cycles of latency due to branch instructions and register to register instructions. This expected latency is not reflected in the measured performance results, because the execution of these instructions is overlapped with the `store` part of the `fetch_and_store` instructions (the MC88100 processor can proceed as soon as the `fetch` portion of the `fetch_and_store` has completed). Hence, our modified Distributed Lock algorithm performs almost as well as the spin lock algorithm on our system.

[4] On our system, all stores to a variable that might be modified with a `fetch_and_store` instruction must also occur using a `fetch_and_store` instruction. For this reason, the unlock operation for a spin lock releases the lock using a `fetch_and_store` rather than a store instruction.

4.1.2 Performance under contention

Figure 5 compares the response times of the different locking algorithms under contention, when p processors continuously acquire and release the same lock. Figure 5a and 5b show the performance for the case where the lock is held for 0 μsec and 25 μsec, respectively.

Because we use the `fetch_and_store` variant of Distributed Locks, it is possible that a *nil* will be stored to the lock variable in `release_lock`, even if there is some successor waiting for the lock. In this case, a performance penalty is incurred to repair the queue. From Figure 5a and 5b, we can see that the first modification we made to Distributed Locks does not degrade performance in the case of contention, while the second modification adds a constant overhead to `release_lock`, which is shown by that fact that the latency increases linearly with the number of contending processors. The extra latency for the second variant is a result of not checking for successors in the unlock operation, requiring the queue to always be repaired if there is a successor. If the lock is held for zero time, then this degradation has a significant effect on performance (Figure 5a), but if the lock is held for as little as 25 μsec, then the extra latency is much less significant (Figure 5b). Note that if `compare_and_swap` were available, then the performance differential would be significantly lower, although it would not be eliminated.

The Distributed Locks are compared against two variants of the exponential backoff spin locks in Figure 5, one where the maximum backoff is 35 μsec and the other where the maximum backoff is 2 msec. The former value is intended for lightly contended locks to reduce the latency in the case where the lock could not be acquired immediately, and is the value used internal to our operating system (for a cluster size of 4). The latter value was chosen because it yields optimal results for the experiments presented. With a maximum backoff of 2 msec, the performance of the spin locks is competitive with that of the Distributed Locks, since the memory contention becomes negligible. However, using this value makes the lock highly susceptible to starvation: with 16 processors contending for the lock and a lock hold time of 25 μsec, it took over 2 msec to acquire the lock in over 13% of the acquisition attempts.

4.2 General locking results

We use two synthetic page fault tests to demonstrate the effects of our locking architecture. In particular, the tests use soft page faults (i.e., faults to pages already in core), since such faults are fairly common in our system, both for mapping in cached files, and to support page-level cache coherence, page migration, and page replication. The tests model particular phases of real applications, stressing the

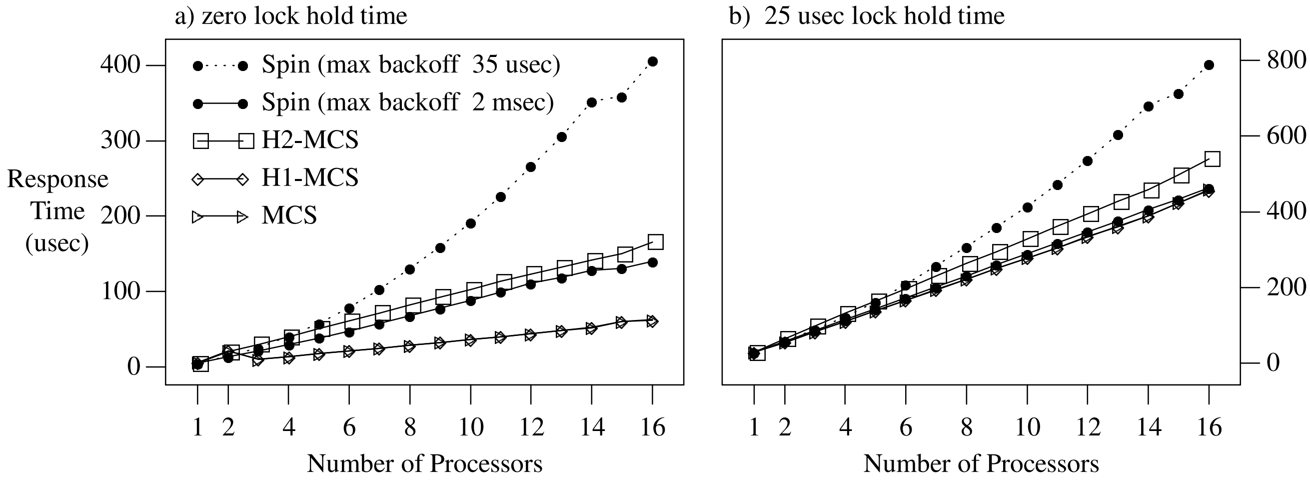

Figure 5: The average response times of a lock/unlock pair when p processes repeatedly access a critical section. The curves show the original Distributed Locking algorithm (MCS), the original algorithm with the initialization code removed (H1-MCS), the second algorithm with the conditional test also removed (H2-MCS), the exponential backoff spin lock with a maximum backoff of 35 μsec, and the exponential backoff spin lock with a maximum backoff of 2 msec.

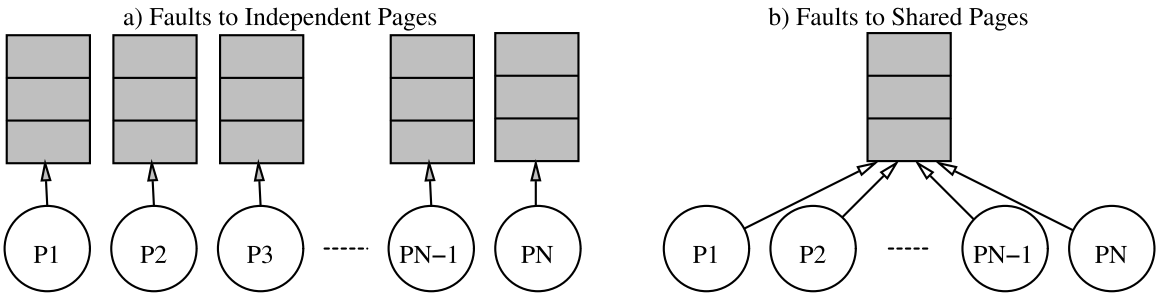

Figure 6: A schematic depiction of the programs that stress the memory management sub-system. (a) p processes repeatedly fault on a region of local memory. (b) p processes simultaneously write to the same small number of shared pages.

boundary cases. Using synthetic tests instead of real applications has the advantage that it allows us to focus our attention on results of interest to this paper. Application results on our system are presented in [25].

The two synthetic tests used are:

Independent faults (Figure 6a): p processes repeatedly fault on a per-process private region of local memory. Because the faults are to different physical resources (i.e., different pages) the only lock contention in this experiment is due to unnecessary locking conflicts in the kernel.

Shared faults (Figure 6b): p processes repeatedly 1) write to the same small number of shared pages, 2) barrier, and 3) unmap the pages from the processes' page tables. Because the faults from the different processes are all to the same shared pages, lock contention is implicit in the application demands.

Figures 7a and 7b show the response time of a page fault for the two tests on a single cluster of 16 processors, as p is varied from 1 to 16 processors. The different curves represent performance when either Distributed Locks or exponential backoff spin locks are used.

For the independent fault test (Figure 7a), there is little difference between the performance of Distributed and exponential backoff spin locks if the number of contending processors is between 1 and 4. However, if p is increased beyond four, then the use of spin locks degrades performance substantially, indicating that the coarse grain locks are a source of contention. With 16 processors faulting concurrently, the latency to handle a page fault is over twice as high when spin locks are used instead of Distributed Locks. These results demonstrate the dramatic impact that second order effects can have on the performance of kernel operations, since the latency increases are due almost entirely to contention at the memory and interconnection network.

For the shared fault test (Figure 7b), the difference in latency between Distributed Locks and spin locks is much smaller. This is because processes contend more for reserve bits, and less for the coarse grain locks. However,

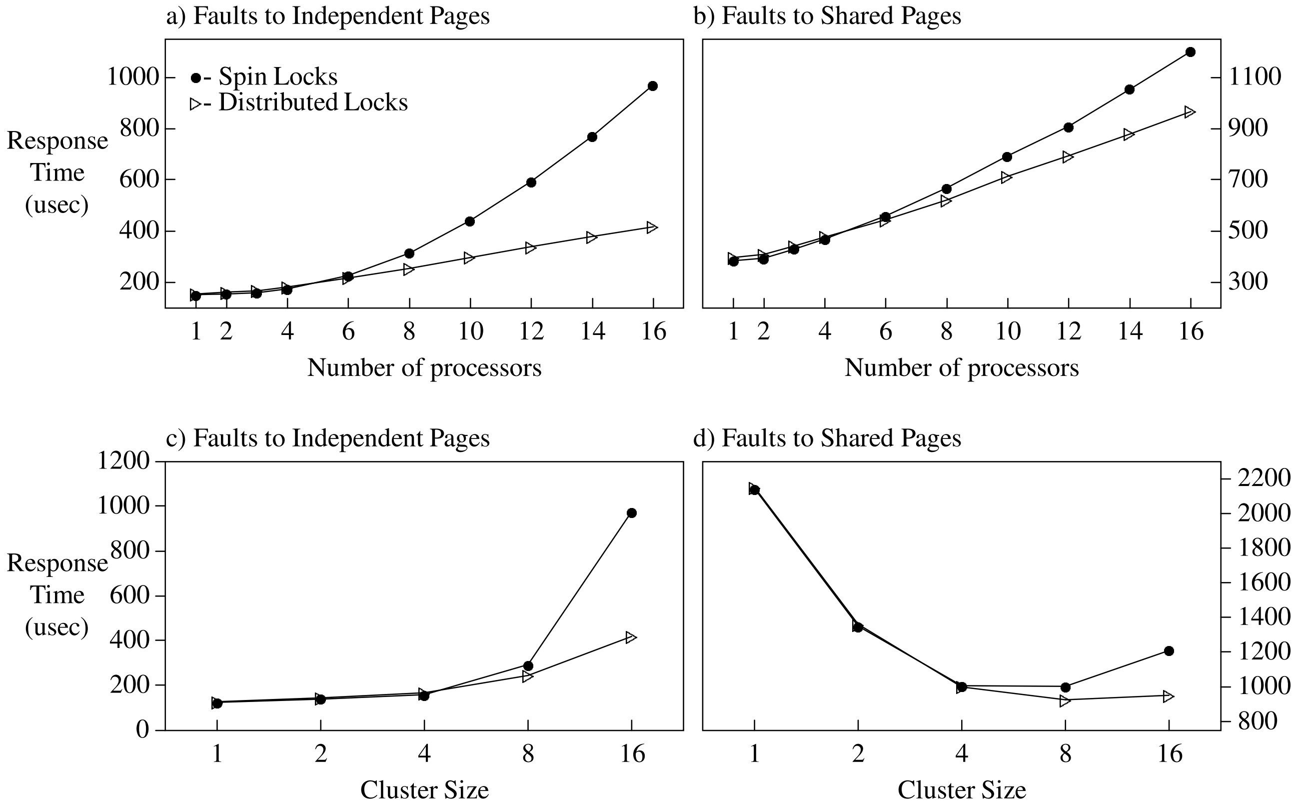

Figure 7: Page fault response times for the two synthetic page fault tests. Figures (a) and (b) demonstrate the effect of different amounts of contention in a fixed cluster size of 16 processors. Figures (c) and (d) demonstrate the impact of different cluster sizes given a fixed degree of contention caused by 16 processors.

it is apparent that there is at least some contention for the coarse grain locks as the number of contending processors increases. This stems from the fact that multiple processors simultaneously attempt to acquire the coarse grain lock that protects the reserve bit they are waiting on whenever a reserve bit is cleared, so there are bursts of heavy demand on the coarse grain locks.

Figures 7c and 7d show the response time with 16 processors as a function of the cluster size, which is varied between 1 and 16. For the independent page fault test, we expect small cluster sizes to result in the best performance. For the shared fault test, however, the situation is not so clear. We expect the sharing in this test to give larger cluster sizes an advantage. On the other hand, having a cluster size smaller than the system size means that the page descriptors are replicated to each cluster, increasing the lock bandwidth and bounding the contention on each page descriptor to the number of processors in the cluster.

Figure 7c shows that smaller cluster sizes do indeed lead to better performance for independent requests. For this experiment, performance does not degrade under contention if the cluster size is 4 or less.[5] Hence, it is clear that the coarse-grained locks that protect the reserve bits do not constrain concurrency; i.e., they do as well as a fine grained locking strategy. Since this is a stress test that exerts demands on the kernel that are more extreme than that of any application, one can expect cluster sizes larger than 4 to perform as well with our hybrid locking strategy as with a fine grained locking strategy, assuming the requests are independent.

By comparing the results of Figure 7c to those of Figure 7a, we can see that the performance of 16 processes, faulting independently in 4 clusters of 4 processors each, is as good as the performance of 4 processes faulting in a single 16 processor cluster. From this we can conclude that hierarchical clustering is effective in localizing requests, allowing independent requests on different clusters to proceed concurrently without interference.

Figure 7d shows that moderate cluster sizes yield the best performance in the case of the shared fault test. For very small cluster sizes, the overhead of inter-cluster operations dominates performance.[6] As the cluster size increases, the cost of obtaining a local copy of the page descriptor with

[5] The difference between the performance of the 2 processor cluster and 4 processor cluster is due to NUMA affects rather than contention.

[6] A null Remote Procedure Call (RPC) requires 27 μsec, while the cost to perform a cluster-wide page lookup and replicate a page descriptor is approximately 88 μsec.

an RPC is amortized over more processors, since only one processor in each cluster must do so; the other processors in the same cluster can then service the fault using the local replica of the page descriptor. This is an example of increasing the lock bandwidth through replication, and merging requests using a combining tree. For very large cluster sizes, the page descriptors become shared by too many processors, and lock contention within the cluster becomes a problem.

A workload with a mix of real applications can be expected to have both independent and non-independent faults. The performance behavior can therefore be expected to be a combination of the behavior of both tests. From this, we conclude that a cluster size somewhere in the range of 4 to 16 processors could be optimal for our system. It is also interesting to note that results similar to those presented above were obtained for stress tests that exercised other portions of the kernel, such as the message passing subsystem.

5 Generalization of Experiences

Our experiences relate to one particular system. In this section we attempt to generalize our results to other hardware and software system architectures.

5.1 Other Operating System Designs

Process-based operating systems

Because our operating system is exception-based, as opposed to process-based [6], non-blocking locks are the most natural approach to locking, since there is no process in the kernel to block. While our approach can be applied directly to process-based kernels in those places where they use spin locks, process-based kernels can also use blocking locks which open up new opportunities to improve on our techniques. For example, by creating a process to handle interrupts and letting them block on locks, it is possible to remove the processor from the lock cycle and to provide greater fairness for remote requests, eliminating the problem described in Section 3.2.

Monolithic operating systems

Another question concerns how our locking strategy might apply to monolithic operating systems. We have applied the techniques described in this paper to several of our system servers, in particular the file system [13], and have found the benefits of reduced latency and increased concurrency that stem from the use of both hierarchical clustering and our hybrid locks apply. This gives us some confidence that our approach would be just as effective when applied to monolithic kernels.

5.2 Other Multiprocessor Designs

The system we used to test our ideas has only 16 relatively slow processors, does not support hardware cache coherence, and provides locking support that is relatively slow (requiring two remote memory accesses for a single read-modify-write sequence) and inflexible (supporting only atomic-swap). It is natural, therefore, to question how our results might apply to more modern systems, with faster processors and interconnection networks, cache-coherence or COMA (Cache-Only Memory Access [7, 9]) support, and more powerful cache-based atomic primitives. Although such architectural changes will shift the tradeoff points, we believe that many of the techniques described in this paper would still apply to these systems. We address each of these issues in turn.

Larger and faster systems

Recent progress in processor design and manufacturing have allowed CPU performance to advance at a significantly greater rate than other components in multiprocessor systems. Although higher bandwidth can be achieved in other components by widening the data paths, the latency to access memory or transfer data between processor caches is a difficult problem to overcome. If one additionally considers the effect of larger systems, it becomes clear that contention for shared resources and the locks that protect them will only get worse. Therefore, techniques such as hierarchical clustering to bound the number of contending processors and lock replication to increase lock bandwidth, RPCs to increase locality, and distributed locks to reduce second-order effects, will become all the more important for such systems.

Cache-coherent systems

One of the more significant differences between our system and many others is our lack of hardware cache-coherence. As a general effect, the lack of cache-coherence makes physical locality more important in our kernel than in others, since we run with most of the kernel data uncached and hence cannot take advantage of temporal locality.

Cache-coherent NUMA and COMA systems have higher data access bandwidth and lower access latency when data is read shared, which suggests that our software techniques for increasing data bandwidth and reducing access latency would be less important in such systems. However, for many kernel data structures we do not believe this to be the case. For example, the HURRICANE memory manager modifies the reference count on page descriptors when a page is mapped into an application address space, and hence page descriptors cannot be replicated efficiently using hardware cache coherence; with hierarchical clustering, a separate local reference count is maintained for each instance of the page descriptor replicated by software. Also, it is still important to limit the number of processors that can contend for a lock, and distribute kernel data structures so that they are near requesting processors. Hence we believe that hierarchical clustering can help, even for systems with hardware cache coherence.

(As an aside, we believe that one might want to run with some portions of the kernel data uncached, even in a hardware cache-coherent system. Running uncached eliminates cache-line-based false sharing, and with it the cache-line ping-pong effects that often occur when data with different access patterns share cache lines. Also, temporal and spatial locality in current operating systems is often quite poor [4, 24], leading to very low cache hit rates, reducing the benefits of cache-coherence.)

Advanced atomic primitives

The atomic-swap operation supported by our processor requires two main memory accesses and is hence relatively slow compared to cache-based atomic operations which permit a lock to be acquired without going to memory (provided the cache-line is currently held in the exclusive state). Newer systems also provide more powerful atomic primitives, such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC), which allow a number of additional optimizations that were not available to us in our system.

The benefits of our hybrid locking strategy come primarily from the reduced number of atomic primitives required, their reduced space overhead, and their natural support for performing multiple simple atomic operations under a single lock. However, cache-based atomic primitives can reduce the cost of atomic operations to close to that of regular memory accesses, bringing into question our focus on reducing the number of atomic operations required for locking. The trade-off between atomic and regular memory accesses depends, however, on subtle implementation issues that can change from year to year. We believe that reducing the number of atomic operations will likely remain beneficial, although not at the expense of larger numbers of regular loads and stores, and hence this benefit of hybrid locking should still apply.

Both CAS and LL/SC instructions allow single bit locks to be implemented that can share a word with other data, thus eliminating another advantage of the hybrid locks. However, the hybrid-lock technique of using coarse-grained locks to protect large data structures has the advantage that multiple operations (such as dequeuing and locking an element) can be performed atomically under a single lock. Hence hybrid locks, with Distributed Locks as the coarse-grain locks, would remain a good choice.

Distributed Locks are affected by cache-based locks in a number of ways. The trade-off between regular spin locks, our version of MCS Distributed Locks, and newer cache-based queueing locks which are optimized for the contended case [5, 17] depends on three primary factors: 1) the degree of sharing of the locks (and thus its hit rate); 2) the amount of steady-state contention expected; and 3) the probability of bursts of very high contention. For low sharing, low steady-state contention, and low burstiness, spin locks would be the better choice, since they have the lowest latency. With higher degrees of sharing the savings from using spin locks are likely to be minimal, since the cost of the cache misses will swamp any savings. In addition, if occasional burstiness is a problem, spin locks must also be ruled out because of the second-order effects from cache-coherence traffic. If the steady-state contention is expected to be low, our modified MCS locks have the advantage, since they have lower latency than other queueing locks. Finally, if high contention is common, the cache-based queueing locks would be the better choice, since their contended-case performance is better than the MCS locks.

Finally, LL/SC or CAS instructions, whether cached or not, can be used to implement lock-free operations, which can remove the need for locks entirely [18]. Lock-free data structures have a number of benefits, both in terms of performance (by removing the extra space and time cost of locks) and in terms of functionality (they eliminate deadlock), but also have a number of disadvantages. Because only a single word (or double word) can be updated atomically, modifications often become more complex: either an entire data structure is copied, changes made to the copy, and a pointer to the copy atomically swapped in (provided the previous pointer still points to the original copy [11]); or the changes can be performed as a series of atomic operations on single words, but only if each change leaves the full data structures in a valid, consistent state [18]. The first approach can be very expensive if the data structure is large, while the second approach requires finding safe states for each atomic change, which can be difficult and error prone. Even when atomic modifications can be done with a small number of atomic primitives, it may still be more efficient to use regular locks depending on the true relative cost of the atomic primitives compared to regular loads and stores. Finally one must be careful about the possibility of starvation using the lock-free approach.[7]

5.3 Current Directions

We are currently in the process of redesigning our locking strategy for our next operating system, TORNADO, targeting a new, T5-based multiprocessor called NUMAchine. This multiprocessor will have an order of magnitude faster processors, cache-coherence support in hardware, cache-based LL/SC instructions, and network caches. Our initial design considerations include:

- Operating systems have traditionally had poor caching behavior [4, 24]. However, we believe this is primarily because caching and multiprocessor cache-coherence

[7] An alternative is to use a wait-free approach, but this is generally much more expensive [10].

effects have been largely ignored in the design of operating systems. Today's processor speeds relative to memory speeds make it imperative to seriously consider the caching effects. We believe it is possible to design the data structures and the locking architecture of an operating system to be cache friendly. Since 10 to 20 lock operations can be performed in the processor's primary cache in the time it takes to service a single cache miss, improving locality and reducing the sharing of locks is likely to be more important than reducing the number of locks.

- We are considering using lock-free data structures for simple leaf locks, particularly for data structures that are required by interrupt handlers and if the data to be modified is contained in a single word.
- Clustering to bound contention and increase lock bandwidth is a clear necessity and should prove to be even more beneficial in our new, larger and faster system.
- Although some of the benefits of our hybrid locks observed in our current system will no longer apply to our new system, their ability to reduce the number of critical sections and to simplify atomic operations involving multiple data structures is still valuable.
- We are currently investigating alternative deadlock management schemes such as the timestamp based approach used in the OSF/1 UFS implementation [16], to be used in conjunction with hierarchical clustering. We hope to be able to preserve the simplicity of the pessimistic approach, and the performance of the optimistic approach.
- Finally, we are starting with a more process-oriented kernel, in part to remove some of the complications of clustering and deadlock, and in part because we believe dynamic process creation can be made to be very fast [8]. We will be reducing our reliance on spin locks, choosing instead to use either lock-free data structures or spin-then-block locks, depending on the situation. As such, the benefits of distributed spin locks will likely be reduced, although it should be possible to support process blocking under distributed locks by building on some of the techniques described in Section 3.2 for handling TryLock.

6 Concluding Remarks

In this paper, we have described a new locking architecture designed for large-scale shared-memory multiprocessors. This architecture consists of a number of components that together provide high performance and scalability. First, a hybrid coarse-grain/fine-grain locking strategy is used that has the low latency and space overhead of a coarse-grained locking strategy, while having the high concurrency of a fine-grained locking strategy. The coarse-grained locks protect large amounts of data, but may only be held for short periods of time. The fine-grained locks must be set under the protection of coarse-grained locks, but can protect individual objects, require only a single bit of storage, and may be held for longer periods of time.

Second, Hierarchical Clustering extends the effectiveness of the hybrid locking strategy to large systems. It organizes the processors into clusters, with separate instances of data structures and the locks that protect them on each cluster. Primarily read-shared data is replicated as needed to accessing clusters, increasing concurrency. Because only processors local to the cluster may access the data in the cluster (requiring an RPC to access remote data), the number of processors contending for a lock is bounded, limiting second-order contention effects on the fine-grained locks.

Finally, Distributed Locks are used to further reduce the second-order effects of lock contention. Our modifications to Distributed Locks bring their uncontended cost close to that of spin locks.

The results of our performance experiments clearly demonstrate the effectiveness of our strategy, at least for our current hardware base. The independent fault test showed little contention for the coarse-grained locks up to 4 processors, suggesting that this aspect of the hybrid locking strategy is appropriate for clusters with up to 8 or even 16 processors under more realistic workloads. However, for non-independent faults, which require greater cross-processor interactions, cluster sizes larger than 4 provided the best performance. Taken together these results suggest that with a mix of real applications having both independent and non-independent demands, a cluster size somewhere in the range of 4 to 16 processors would be optimal for our system.

Overall, we have found that the design of a locking architecture is largely an exercise in global optimization, as one tries to balance the strengths and weaknesses of both the techniques and the underlying hardware. However, we believe that many of the techniques presented in this paper will also apply to other systems.

References

[1] David L. Black, Avadis Tevanian Jr. , David B. Golub, and Michael W. Young. Locking and reference counting in the mach kernel. In *Proc. 1991 ICPP*, volume II, Software, pages II–167–II–173, Boca Raton, FL, August 1991. CRC Press.

[2] H.H.Y. Chang and B. Rosenburg. Experience porting mach to the RP3 large-scale shared-memory multiprocessor. *Future Generation Computer Systems*, 7(2–3):259–267, April 1992.

[3] E. Chaves, P.C Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-

memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, May 1993.

[4] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proc. 14th ACM SOSP*, pages 120–133, 1993.

[5] Travis S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, 02 1993. (ftp tr/1993/02/UW-CSE-93-02-02.PS.Z from cs.washington.edu).

[6] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. 13th ACM SOSP*, page 122, Pacific Grove, CA, October 1991.

[7] S. Frank, J. Rothnie, and H. Burkhardt. The KSR1: Bridging the gap between shared memory and MPPs. In *IEEE Compcon 1993 Digest of Papers*, pages 285–294, 1993.

[8] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proc. 1994 ICPP*, pages 208–211, Boca Raton, FL, August 1994. CRC Press.

[9] Erik Hagersten, Anders Landin, and Seif Haridi. "DDM – A Cache-Only Memory Architecture". *IEEE Computer*, pages 44–54, September 1992.

[10] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.

[11] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM TOPLAS*, 15(5):745–770, November 1993.

[12] J. Kent Peacock, S. Saxena, D. Thomas, F. Yang, and W. Yu. Experiences from multithreading system V release 4. In *SEDMS III*, pages 77–91. Usenix Assoc, March 1992.

[13] Orran Krieger. *HFS: A flexible file system for shared memory multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, 1994.

[14] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. Technical Report CSRI-275, Computer Systems Research Institute, University of Toronto, Toronto, Canada, M5S 1A1, October 1992.

[15] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *ASPLOS-VI*, 1994. To appear.

[16] Susan LoVerso, Noemi Paciorek, Alan Langerman, and George Feinberg. The OSF/1 UNIX filesystem (UFS). In *USENIX Conference Proceedings*, pages 207–218, Dallas, TX, January 21-25 1991. USENIX.

[17] Peter Magnussen, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *8th IPPS*, pages 26–29, 1994.

[18] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, February 1991.

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

[20] Noemi Paciorek, Susan Lo Verso, and Alan Langerman. Debugging multiprocessor operating system kernels. In *SEDMS II*, pages 185–202. USENIX, Atlanta GA, March 21 - 22 1991.

[21] J. Kent Peacock. File system multithreading in system V release 4 MP. In *USENIX Conference Proceedings*, pages 19–30, San Antonio, TX, Summer 1992. USENIX.

[22] Abraham Silberschatz, James L. Peterson, and Peter Galvin. *Operating Systems Concepts*. Addison-Wesley, third edition edition, 1991.

[23] Daniel Stodolsky, J. Bradley Chen, and Brian N. Bershad. Fast interrupt priority management in operating system kernels. In *USENIX Microkernels Workshop*. USENIX, 1993.

[24] Josep Torrellas, Anoop Gupta, and John L. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *ASPLOS-IV Proceedings*, pages 162–174, Boston, Massachusetts, 1992.

[25] R. Unrau, M. Stumm, O. Krieger, and B. Gamsa. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*. To appear. Also available as technical report CSRI-268 from ftp.csri.toronto.edu.

[26] Ronald C. Unrau. *Scalable Memory Management through Hierarchical Symmetric Multiprocessing*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, January 1993.

[27] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. "The Hector Multiprocessor". *Computer*, 24(1), January 1991.