Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design

Ron Unrau, Orran Krieger, Benjamin Gamsa, Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 1A4

Abstract

We introduce the concept of Hierarchical Clustering as a way to structure shared memory multiprocessor operating systems for scalability. As the name implies, the concept is based on clustering and hierarchical system design. Hierarchical Clustering leads to a modular system, composed of easy-to-design and efficient building blocks. The resulting structure is scalable because it i) maximizes locality, which is key to good performance in NUMA systems, and ii) provides for concurrency that increases linearly with the number of processors. At the same time, there is tight coupling within a cluster, so the system performs well for local interactions which are expected to constitute the common case. A clustered system can easily be adapted to different hardware configurations and architectures by changing the size of the clusters.

We show how this structuring technique is applied to the design of a microkernel-based operating system called HURRICANE. This prototype system is the first complete and running implementation of its kind, and demonstrates the feasibility of a hierarchically clustered system. We present performance results based on the prototype, demonstrating the characteristics and behavior of a clustered system. In particular, we show how clustering trades off the efficiencies of tight coupling for the advantages of replication, increased locality, and decreased lock contention. We describe some of the lessons we learned from our implementation efforts and close with a discussion of our future work.

1 Introduction

Considerable attention has been directed towards designing "scalable" shared-memory multiprocessor hardware, capable of accommodating a large number of processors. These efforts have been successful to the extent that an increasing number of such systems exist [1, 8, 19, 28, 30, 32]. However, scalable hardware can only be fully exploited and cost effective for general purpose use if there exists an operating system that is as scalable as the hardware.

An operating system targeting large-scale multiprocessors must consider both concurrency and locality. However, existing multiprocessor operating systems have been scaled to accommodate many processors only in an *ad hoc* manner, by repeatedly identifying and then removing the most contended bottlenecks, thus addressing concurrency issues but not locality issues. Bottlenecks are removed either by splitting existing locks, or by replacing existing data structures with more elaborate, but concurrent ones. The process can be long and tedious, and results in systems that 1) have a large number of locks that need to be held for common operations, with correspondingly large overhead, 2) exhibit little locality, and 3) are not scalable in a generic sense, but only until the next bottleneck is encountered [4, 5, 11, 12, 23]. Porting an existing system designed for networked distributed systems is also unsatisfactory, because of the large communication requirements

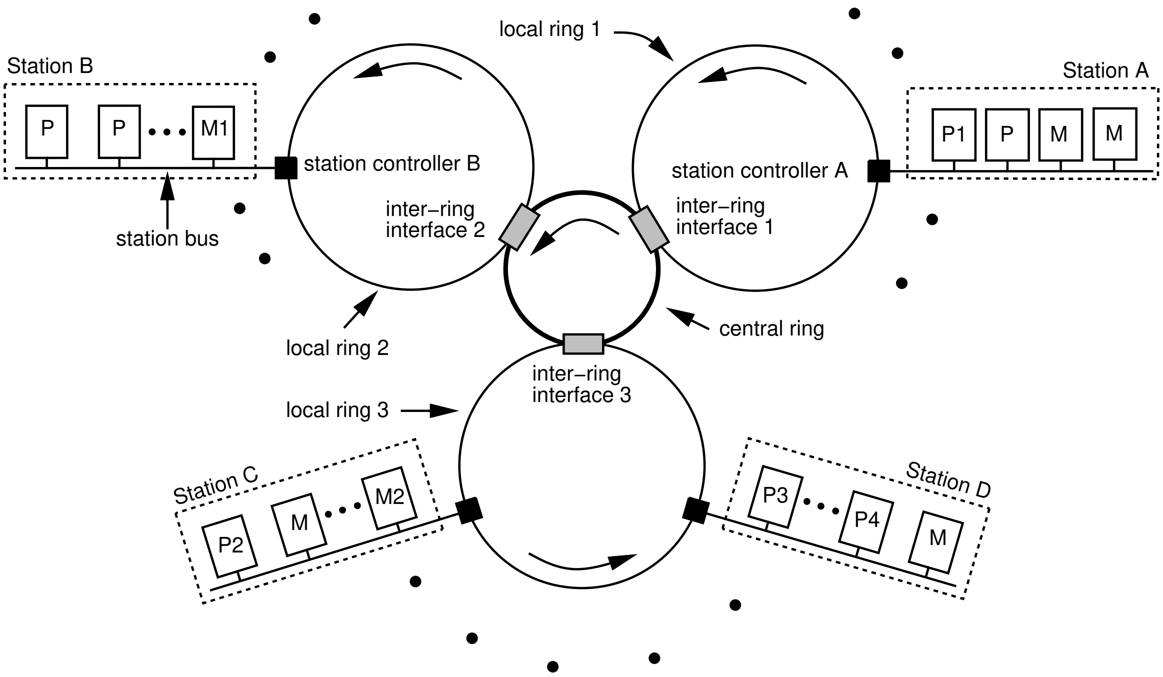

Figure 1: A high-level representation of the HECTOR multiprocessor. Each processor box in the system includes a CPU and cache, local memory, and I/O interface.

caused by replicating code and data and because of the overhead and lack of locality in inter-processor interactions.

We believe that a more structured approach for designing scalable operating systems is essential and propose a new, simple structuring technique based on clustering and hierarchical system design called Hierarchical Clustering. This structuring technique leads to a modular system, composed from easy-to-design and efficient building blocks. It is applicable to a wide variety of large-scale shared-memory architectures and configurations. The structure maximizes locality, which is key to good performance in Non-Uniform Memory Access (NUMA) systems, and provides for concurrency that increases linearly with the number of processors.

In this paper, we describe Hierarchical Clustering and its advantages. We show how it can be applied in implementing a micro-kernel based operating system, and present the results of performance measurements obtained from a running prototype. We describe some of the lessons we learned from our implementation efforts and close with a discussion of our future work.

2 Operating System Targets

The design of an operating system is influenced both by the hardware platform it is responsible for managing and the demands of the applications it expects to serve. The internal structure of the operating system and the policies it enacts must accommodate these targets if it is expected to perform well. In this section, we introduce the particular hardware and workload targets we assume for this paper and briefly describe how the operating system can accommodate them.

2.1 The Target Hardware

Our hardware target is the class of "scalable" shared memory multiprocessors. Most current scalable shared memory multiprocessor are based on segmented architectures, which allow the aggregate communication bandwidth to increase as new segments are added to the system. In these systems, the memory is distributed across the segments, but remains globally accessible. Because the cost of accessing memory is a function of the distance between the accessing processor and the memory being accessed, they exhibit non-uniform memory access (NUMA) times. Examples of systems with segmented architectures include HECTOR [37], DASH [30], and the KSR1 [19]. The HECTOR system is depicted in Figure 1 and was used for the experiments we discuss in Section 5.

Although the aggregate bandwidth of these scalable shared-memory machines increases with the size of the system, the bisectional bandwidth (from any one segment to the rest of the system) is typically much lower and limited. Consequently, it is important to organize the location of data so that it is kept close to the accessing processors. In some cases, locality can also be improved by replicating data or migrating it closer to the accessing processor, depending on how the data is being accessed. It should be noted that locality is important even for COMA-based architectures [19, 22] and systems with hardware-based cache coherence [1, 30], because operating system data is typically accessed with little temporal locality, yet shared at a fine granularity [14], leading to generally low cache hit rates.

2.2 The Target Workload

The workload of an operating system stems primarily from requests that come from application programs and will include, among other things, page fault handling, I/O, interprocess communication, process management, and synchronization. The operating system is thus largely demand driven. The relative timing of the requests and the way they interact (by accessing common data structures) in the operating system depends on the type of applications currently running. The applications we expect to see running at any given time include a mix of sequential interactive programs, small-scale parallel applications, and very large parallel applications.

Each of these application classes places different demands on the operating system. The requests of an interactive, sequential application will be largely independent from the requests of the other applications, both with respect to the time they are issued and the operating system data they access. For example, the probability of two independent applications simultaneously accessing the same file is small. However, we can expect the load from these requests to grow linearly with the size of the system. To satisfy this load without creating a bottleneck, the number of operating system service centers must increase with the size of the system. Moreover, it is important to limit contention for locks that regulate access to shared data and to limit contention at physical resources, such as disks.

In contrast, the threads of a parallel application may well make simultaneous requests that cause the same data to be accessed. This might happen, for example, after a program phase change when each thread starts to access the same data file. When servicing these non-independent requests, the accesses to the shared resources must be fair, and the latency of accessing these resources must degrade linearly with the number of concurrent requesters.

The operating system can help exploit the locality inherent in an application, by scheduling communicating processes to neighboring processors, by directing I/O to nearby disks, and by placing application data close to where it will be accessed. Policies such as data replication or migration can also be used for both memory and secondary storage management to increase access bandwidth, decrease latency, or both.

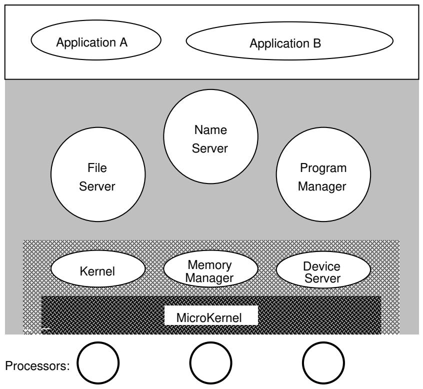

Figure 2: A single clustered system

3 Hierarchical Clustering

Hierarchical Clustering is a structuring technique based on clustering and hierarchical system design that we believe addresses most of the issues described in the previous section. Hierarchy is a well known organizational framework, occurs frequently in nature [34], and is used often in business, government, and the military. In these systems, the purpose of the higher levels of the hierarchy is to manage or control resources in a coarse, strategic, and global fashion, while lower levels in the hierarchy control resources in a finer, more localized way. Communication occurs primarily between entities close to each other, and interaction between distant entities typically occurs through higher levels of the hierarchy. If the operations at the lowest levels are indeed localized, then there can be much concurrency in the system. It is both the locality and concurrency aspects of theses systems that allow them to scale to relatively large sizes.

A Hierarchically Clustered system is scalable for the same reason: it provides for locality and it provides for concurrency that increases with the number of processors. It avoids concurrency constraints imposed at the hardware level due to bus and network bandwidth limitations and imposed at the operating system level due to locks by replicating data structures, and by exporting appropriate policies. Clusters that provide for tight coupling and good performance for neighboring objects that interact frequently with one another. Hierarchical Clustering also has a number of pragmatic advantages, for example by simplifying lock tuning and by being adaptable to different architectures and configurations.

In this section, we first describe the basic structure and then argue why this structure is scalable. This is followed by a discussion of factors that affect the best choice of cluster sizes.

3.1 The Basic Structure

In a system based on Hierarchical Clustering, the basic unit of structuring is a *cluster*. A cluster provides the functionality of an efficient, small-scale symmetric multiprocessing operating system targeting a small number of processors. Kernel data and control structures are shared by all processors within the cluster.[1] In our implementation, a cluster consists of a symmetric micro-kernel, memory and device management subsystems, and a number of user-level system servers such as a scheduler and file servers (Figure 2).

On a larger system, multiple clusters are instantiated such that each cluster manages a unique group of "neighboring" processing modules (including processors, memory, and disks). All major system services are

[1] Note, clustering does not preclude the management of some resources on a per-processor basis within a cluster. For example, the performance of task dispatching can be improved if processes are maintained on per-processor rather than per-cluster ready queues.

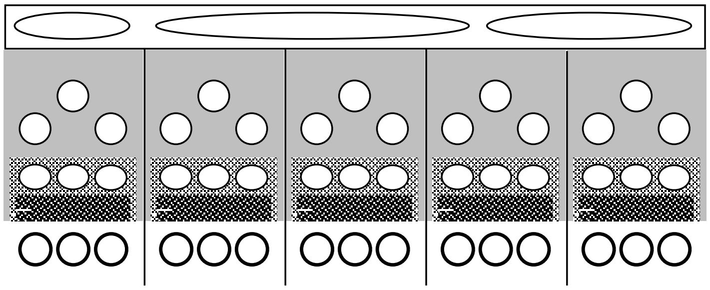

Figure 3: A multiple clustered system

replicated to each cluster so that independent requests can be handled locally (Figure 3). Clusters cooperate and communicate to give users and applications an integrated and consistent view of a single large system.

The basic idea behind using clusters for structuring is to use multiple easy-to-design and hence efficient computing components to form a complete system. A single level of clusters can be expected to support moderately large systems (in the, say, 64–256 processor range). However, for larger systems, additional levels in the hierarchy will be necessary. In particular, while each cluster is structured to maximize locality, there is no locality in cross-cluster communication with only a single level of clusters. The logical next step is to group clusters into super clusters, and then to group super clusters into super-super clusters, where each level in the clustering hierarchy involves looser coupling than the levels below it.

Hierarchical Clustering incorporates structuring principles from both tightly-coupled and distributed systems, and attempts to exploit the advantages of both. By using the structuring principles of distributed systems, the services and data are replicated and migrated so as to: a) distribute the demand, b) avoid centralized bottlenecks, c) increase concurrency, and d) increase locality. Hence, many design issues in a clustered system are similar to those encountered in distributed systems. For example, shared objects can be distributed and migrated across clusters to increase locality, but then the search to locate these objects becomes more complex. Other objects may be replicated to increase locality and decrease contention, but then consistency becomes a critical issue.

On the other hand, there is tight coupling within a cluster, so the system is expected to perform well for the common case where interactions occur primarily between objects located in the same cluster. For this reason, there are important differences between a clustered system and a distributed system that lead to completely different design trade-offs. In a distributed system, for example, the hosts do not share physical memory, so the cost for communication between hosts is far greater than the cost of accessing local memory. In a (shared memory) clustered system, it is possible to directly access memory physically located in other clusters, and the costs for remote accesses are often not much higher than for local accesses. Additionally, demands on the system are different in the multiprocessor case, because of tighter coupling assumed by the applications. Finally, fault tolerance is a more important issue in distributed systems where the failure of a single node should not crash the entire system. In a shared-memory multiprocessor, this type of fault tolerance is not (yet) a major issue (although clustering should provide a good foundation on which to address operating system design for fault containment when suitable hardware becomes available).

3.2 Scalability

Intuitively, scalability of an operating system refers to its capacity to provide increased service as the hardware resources (eg. processors, memory, or disks) are increased. In queuing theoretic terms, an operating system can only scale if there are no resources within the system (eg. queues, locks, etc.) that saturate as the system size is increased.

In structuring an operating system for scalability, it is important that application requests to independent physical resources be managed by independent operating system resources. For example, if two applications simultaneously request data from two different disks, concurrency in servicing the requests is maximized if the operating system structures that implement the service are completely independent.

More generally, a scalable operating system will possess the following properties [36]:

- *The parallelism afforded by the applications is preserved.*

The operating system workload can be expected to increase with the size of the system. Because most of the parallelism in an operating system comes from requests that are concurrently issued by multiple application threads, the number of service centers must increase with the size of the system, as must the concurrency available in accessing data structures.

With Hierarchical Clustering, each cluster provides its own service centers and has its own local data structures. Application requests are always directed to the local cluster. Hence, the number of service points to which an application's requests are directed grows proportional to the number of clusters spanned by the application. As long as the requests are to independent physical resources, they can be serviced in parallel.

- *The overhead for independent service requests is bounded by a constant* [6].

If the overhead of a service call increases with the number of processors, then the system will ultimately saturate. For this reason, the demand on any single operating system resource cannot increase with the number of processors. With Hierarchical Clustering,, there are no system-wide ordered queues, and more generally, no queues whose length increases with the size of the system. Because the number of processors in a cluster is fixed, fixed-sized data structures can be used within a cluster to bound the overhead of accessing these structures.

- *The locality of the applications is preserved*

Hierarchical Clustering increases locality by co-locating objects that need to interact frequently in the same cluster (or super-cluster if the interactions cross cluster boundaries) and by always first directing requests from the application to the local cluster. For example, descriptors of processes that interact frequently are forced to be close to each other, and memory mapping information is located close to the processors that need it for page fault handling. Hierarchical Clustering also provides a framework for enacting policies that increase locality in the applications' memory accesses and system requests. For example, policies can attempt to run the processes of a single application on processors close to each other, place memory pages in proximity to the processes accessing them, and attempt to direct file I/O to nearby devices.

3.3 Cluster Size Trade-offs

The above requirements for scalability can be largely met by using a distributed systems approach, where system services and structures are replicated onto each individual processing module; that is, using a cluster size of one. The many large-scale distributed (i.e. networked) systems in existence today clearly demonstrate this. However, using a cluster size of one on a shared memory multiprocessor will for several reasons result in greater overhead than using larger cluster sizes. First, the smaller the cluster size, the more inter-cluster operations are incurred, and inter-cluster operations are more costly than intra-cluster operations. Second, there is no locality in inter-cluster communication, which further increases its cost. Third, it is necessary to keep the state on each cluster consistent with that on the other clusters. Finally, a cluster size of one results in greater memory requirements due to the replication of much of the operating system's code and data.

A cluster size larger than one reduces the consistency and space overhead and reduces the number of inter-cluster operations required. However, the size of a cluster cannot become arbitrarily large without

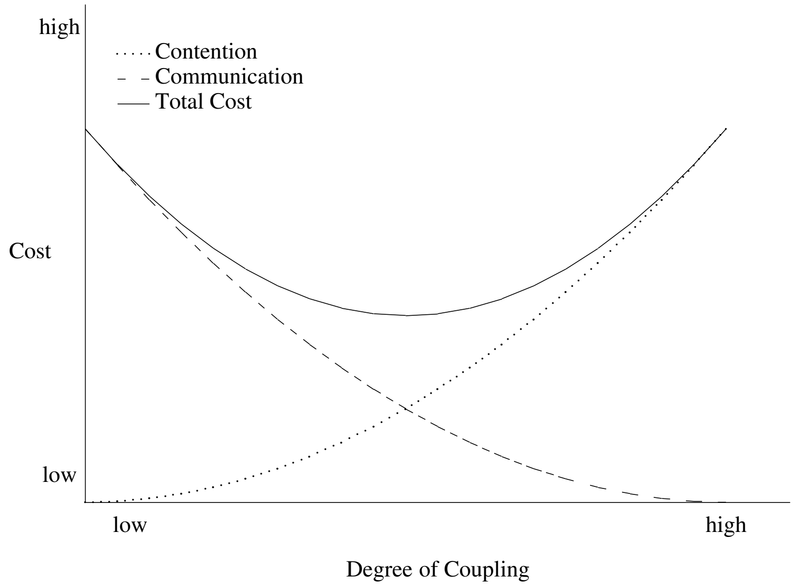

Figure 4: Cost versus coupling in a large-scale system

reorganizing the internal cluster structure, because the probability of contention for shared data structures will increase with the number of processors in the cluster, ultimately resulting in saturation.

These trade-offs in the choice of cluster size are depicted abstractly by the graph in Figure 4. The graph plots cost (including overhead) against the degree of coupling or cluster size for some fixed number of processors. The dotted curve in the figure represents access overhead and contention, which increases as the cluster size becomes larger because the probability of active resource sharing increases, while locality within the cluster deceases. The dashed curve represents the portion of overhead due to remote, inter-cluster communication; in general, fewer inter-cluster operations are necessary when the size of the cluster is larger. The solid curve is the sum of the contention and communication costs, and suggests that there is some degree of coupling for which the overall cost is minimized. In Section 5 we show that this cost behavior can actually be observed in real systems.

One of the benefits of Hierarchical Clustering is that system performance can be tuned to particular architectures, configurations, and workloads, by adjusting the size of the clusters. The architectural issues that affect the appropriate cluster size include the machine configuration, the local-remote memory access ratio, the hardware cache size and coherence support, the topology of the interconnection backplane, etc. On hierarchical systems such as Cedar [28], Dash [30], or Hector [37], the lowest level operating system cluster might correspond to a hardware cluster. On a local-remote memory architecture, such as the BBN Butterfly [7], a smaller cluster size (perhaps even a cluster per processor), may be more appropriate; in this case, clustering can be viewed as an extension of the fully distributed structuring sometimes used on these machines.

Choosing the correct locking granularity for a parallel system is generally difficult. A large granularity restricts concurrency, but a fine granularity increases the length of the critical path. Hierarchical Clustering allows the locking granularity within a cluster to be optimized for the size and concurrency requirements of the cluster [35]. For example, the message passing code in our implementation employs only a small number of large grain locks, adding little overhead to the critical path but providing sufficient concurrency for the expected demand given our commonly used cluster size. Fine-grained locking is used when a high degree of concurrency is required or if only a small number of locks need be acquired in the critical path.

4 Implementation

We have implemented a micro-kernel based operating system called HURRICANE, structured according to the principles of Hierarchical Clustering. The prototype does not as yet support the super-clusters we envisage are necessary for large systems. Instead, we have focused initial efforts on structuring within and across first level clusters. At present, the system is fully functional, and in this section we show how clustering has affected the structure of its primary components.

In order to make a clustered operating system successful, it is important to keep the following issues in mind. Applications should be shielded from the fact that the operating system is partitioned into clusters; to users and applications the system should, by default, appear as a single, large, integrated system. Moreover, the complexity introduced by separating the system into clusters must be kept reasonably low, since clusters were introduced as a structuring mechanism primarily to reduce overall complexity, not increase it. Finally, inter-cluster communication must be efficient; while we expect most interactions to be local to a cluster, those interactions that must access remote clusters should not be overly penalized.

The main design issue in implementing a clustered system (as opposed to a traditionally structured system) is the choice in strategy for managing the data and the choice in mechanism for communication. For each type of data, one must decide whether data of that type should be replicated or whether there should be only one copy. If there is only one copy, there is a choice of placing it statically, or allowing it to dynamically migrate to where it is being accessed. If data is replicated then one must decide how to keep it consistent. The above choices will be influenced primarily by the expected access behavior. For example, one would not want to replicate actively write-shared data.

Inter-cluster communication is needed i) to access remote data that is not replicated, ii) to replicate and migrate data, and iii) to maintain data consistency. Three different mechanisms can be used for inter-cluster communication [13]. First, shared memory access is always a viable option on a shared-memory multiprocessor, particularly for light-weight operations that make only a few references. For instance, our implementation uses shared memory for table lookup to locate objects such as page descriptors. However, we prefer to minimize cross-cluster shared-memory accesses because of the lack of locality in these accesses (and hence the increased probability of contention if used indiscriminately). There are also other problems with accessing remote resources using shared memory. With spin locks, secondary effects such as memory and interconnect contention can result in greater than linear increase in cost as the number of contending processors grows [3, 31]. When only processors belonging to the same cluster contend for a spin lock, these secondary effects can be bounded.

A second possibility is to use Remote Procedure Calls (RPCs). In our system, RPCs are implemented using remote interrupts, and are used for kernel-kernel communication. The interrupted cluster obtains the call information and arguments through shared memory directly, after which data accesses required to service the request can be local to the interrupted cluster. The savings gained from the local data accesses must amortize the cost of the interrupt (ie., the cycles that are stolen and the register saving and restoring).

Finally, HURRICANE supports message passing that allows user-level processes to send messages to other processes regardless of their location in the system. This provides a simpler form of cross-cluster communication for higher-level servers that are replicated across the clusters of the system.

4.1 The HURRICANE Kernel

The HURRICANE kernel is responsible for process management and communication, and is similar in structure to the V kernel [15]. Most operations on processes involve the queuing and dequeuing of process descriptors. For example, when a process sends a message to another process, its descriptor (containing the message) is added to the Message Queue of the target descriptor. Other queues include a Ready Queue for processes that are ready to run, and a Delay Queue for processes waiting for a time-out. Note that the number of queues increases with the number of clusters, so that by adding more clusters as the system grows, the average demand on each queue should remain constant. The process descriptor table and kernel queues are local to each cluster, preserving the locality for the common case where the request can be handled entirely within the cluster. In addition, migrated processes are represented by process descriptors that are local to the new cluster, so that operations on them do not require remote accesses.

A home encoding scheme is used to allow a process to be located anywhere in the system with a constant number of operations, independent of the number of processors. When a process is created, it is assigned a process identifier within which the id of the *home* cluster is encoded. Usually, a process remains within its home cluster, but if it migrates, then a record of its new location (a *ghost* descriptor) is maintained at the home cluster. Operations that require access to the migrated process look first in the home cluster, find the ghost, and use the locator information in the ghost to determine the current cluster in which the process resides.

Inter-cluster message passing is implemented using remote procedure calls that interrupt the target cluster to 1) allocate a local message retainer (similar to a process descriptor) in the target cluster, 2) copy the message and other information into the retainer by directly accessing the process descriptor in the source cluster, and 3) add the message retainer to the message queue of the target process descriptor. Note that by using local message retainers in the target cluster (as opposed to a global pointer to the process descriptor in the source cluster), the message queue will always be an entirely local data structure.

In the case of the HURRICANE kernel, having the cluster size smaller than the size of the system has three key advantages:

1. it localizes the kernel data structures that need to be accessed;
2. it reduces the number of process descriptors that must be managed within a cluster, thus reducing the average length of cluster-wide queues; and
3. system-wide, it increases the number of queues and tables (since each cluster has local instances), reducing demand on the resources and increasing concurrency.

On the other hand, having a cluster span more than one processor reduces the amount of inter-cluster communication, and makes it easier to balance the load of the processors.

4.2 Protected Procedure Calls

With a micro-kernel multiprocessor operating system, the majority of IPC operations are required solely to cross address space boundaries (ie., for client-server interactions) and not to cross processor boundaries. The HURRICANE message passing facility described above is optimized for inter-processor communication, while another facility, called Protected Procedure Calls (PPCs) is optimized for inter-address space communication.

The PPC facility employs the same structuring principles as Hierarchical Clustering but applied on a per-processor basis. As a result, it accesses only local data, requires no access to shared data, and requires no locking, so end-to-end performance is comparable to the fastest uniprocessor IPC times. Yet, because of the concurrency provided, this performance can be sustained independent of the number of concurrent PPC operations, even if all such operations are directed to the same server [20]. Moreover, the PPC facility ensures that local resources are made available to the target server so that any locality in client-server interactions can be exploited by the server.

In our implementation of Protected Procedure Calls (PPCs), separate worker processes in the server are created as needed to service client calls on the same processor as the requesting client. On each call, they are (re)initialized to the server's call handling code, affecting an up-call directly into the service routine. This results in a number of benefits: i) client requests are always handled on their local processor; ii) clients and servers share the processor in a manner similar to hand-off scheduling; and iii) there are as many threads of control in the server as client requests.

The key benefits of our approach all result from the fact that resources needed to handle a PPC are accessed exclusively by the local processor. By using only local resources during a call, remote memory accesses are eliminated. More importantly, since there is no sharing of data, cache coherence traffic can be eliminated. Since the resources are exclusively owned and accessed by the local processor, no locking is required (apart from disabling interrupts, which is a natural part of system traps).

Although call handling is managed at the local processor level, creation and destruction of communication endpoints are managed at the cluster level (in order to synchronize these events across the cluster), requiring cross-cluster communication only for those servers that span multiple clusters. Such operations are sufficiently rare in our system that the performance impact is negligible.

4.3 Memory Management

Each HURRICANE virtual address space defines the application accessible memory resources as a set of non-overlapping regions, each of which describes the attributes of a contiguous sequence of memory pages. Each region is directly associated to a corresponding file region, so that accesses to the memory region behave as though they were accesses to the file region directly. This relationship is known as a single-level store, or mapped-file, memory abstraction [36]. Within a cluster, address spaces are represented by an address space descriptor and an associated balanced binary tree of region descriptors. The region descriptors maintain the virtual limits and mapping attributes of each address space region.

For programs that span multiple clusters, the address space and region descriptors are replicated to the clusters as needed. To maintain the integrity of these replicated structures, all modifications are directed through the *home cluster*, which serves as a point of synchronization. Currently, the home cluster is designated as the cluster on which the program was created. Since the address space and region descriptors are accessed for reading far more often than they are modified, replication of these structures improves locality of access in the common case.

In large systems, applications that span many clusters could cause the memory manager on the home cluster to become contended, because it must both replicate the data structures and distribute modifications to them. Additional levels in the hierarchy could be used to improve both cases. On first access, a source replica for the data structure could be obtained from the local super- or super-super-cluster rather than the home cluster. A software combining tree could then be used to distribute modifications to all the replicas.

Each physical page in a cluster has a page descriptor associated with it. This structure identifies the physical address of the page, its state, and the file block stored there. All pages that contain valid file data are kept in a *page cache*, which is a hash table indexed by file block, and is used to determine whether a file block is already resident in the cluster. Unreferenced pages are placed on a free list, but remain in the page cache in case the file block is referenced again in the near future. These structures are always local to a cluster and need not be replicated or migrated, since they reeepresent local physical resources. To reduce the amount of cross-cluster communication, representative page descriptors are used to identify pages that are accessed remotely. This keeps all the data structures local to the cluster, and allows the results of a remote page search to be subsequently used by other processors within the cluster.

The HURRICANE memory manager supports page replication and migration across clusters. This can reduce access latency and contention for some applications [9, 17, 29], however, the overhead of these policies must be amortized to realize a net gain in performance. Hierarchical Clustering provides a framework for enacting paging policies, where pages are shared within a cluster to reduce overhead, but are replicated and migrated across clusters to reduce access latency and memory contention.

A simple directory mechanism is used to maintain page consistency between clusters. There is one directory entry for each valid file block currently resident in memory; it identifies which clusters have copies of the page. The directory is distributed across the processors of the system (thus spanning all clusters), allowing concurrent searches and balanced access demand across the system. Pages that are actively write-shared are not replicated, but instead are accessed remotely.

4.4 The File System

File I/O in HURRICANE is provided largely outside of the kernel; only the device drivers are in the kernel. The HURRICANE File System (HFS) implementation makes use of three user-level system servers [25, 26]. The *Name Server* manages the HURRICANE name space. The *Open File Server* (OFS), maintains the file system state for each open file, and is (largely) responsible for authenticating application requests to lower level servers. The *Block File Server* (BFS) controls the disks, determines the target for each I/O request, and directs the request to a corresponding device driver.

The *Alloc Stream Facility* (ASF) is a file system library in the application address space [24, 27, 25] that translates application file `read` and `write` requests into accesses to a mapped file region. On the first access to a page in the region, the process will page fault. If the data is not available in the file cache, then the fault is translated by the memory manager into a `read_page` request to the BFS.

There is a separate instance of all file system servers on each cluster. Client/server interactions within a cluster are directed to servers local to the client. For example, the memory manager directs all page fault requests to its local BFS. Only requests between peer HFS servers (eg., BFSs in different clusters) may cross cluster boundaries. Cross cluster communication between file system servers occurs through explicit message passing, not through shared memory.

When an application spans multiple clusters, or when programs on different clusters access the same files, then the file system state for shared files is replicated to the cluster servers as they are needed. When state is first required by a server, it locates some other server that has the state (currently by directing a request to the server in the cluster identified by the persistent token of the file), to obtain a local copy of the state. To maintain the integrity of these replicated structures, all modifications are directed through the *home cluster*, which serves as a point of synchronization. In our current design, the home cluster is the cluster where the file was first opened. Depending on the particular type of state being accessed, modifications result in either update or invalidate messages being sent to the servers on the clusters that have a copy.

In addition to providing a structured way to achieve scalability for the file system servers, clustering provides a framework for invoking policies that localize application I/O requests and balance the load across the disks of the system. Files are created on disks that are in clusters near the requesting applications. If a file is replicated to multiple disks, the file system can use Hierarchical Clustering to determine the nearest replica of the file and direct read requests to that replica. Similarly, for some kinds of distributed files, the file system can distribute write requests to disks in clusters near the page frame that contains the dirty page. In the case of load balancing, HFS maintains both long and short term statistics on the aggregate disk load of the clusters and super clusters. This information can be used in order to direct I/O requests to the least loaded clusters. This information can also be used when a new file is created so that the file is created on disks in lightly loaded clusters.

4.5 Scheduling

One goal of the scheduling subsystem is to keep the load on the processors (and possibly other resources, such as memory) well balanced. The scheduling decisions in HURRICANE are divided into two levels. Within a cluster, the load on the processors can be balanced at a fine granularity through the dispatcher (in the micro-kernel). This fixed scope limits the load on the dispatcher itself and allows local placement decisions on different clusters to be made concurrently. Cross-cluster scheduling is handled by higher-level scheduling servers, which balance the load by assigning newly created processes to specific clusters, and by migrating existing processes to other clusters. The coarser granularity of these movements permits a low rate of inter-cluster communication between schedulers.

A second goal in a Hierarchical Clustering system is to enact process placement policies that maximize locality for applications. For parallel programs with a small number of processes, all of the processes should run in the same cluster. For larger-scale parallel programs that span multiple clusters, the number of clusters spanned should be minimized. These policies are motivated by studies [10, 39, 18, 2] which have shown that clustering can noticeably improve overall application performance.

5 Experimental Results

An important aspect of our implementation is that the cluster size can be specified at boot time. This allows us to explore the trade-offs between contention and communication overhead, and to quantitatively evaluate our locality management policies. This section reports the results of several experiments designed to evaluate Hierarchical Clustering for both synthetic stress tests and real applications. All experiments were run on a fully configured version of HURRICANE with all servers running, but with no other applications running at the time.

The experiments were run on a 16 processor prototype of a hierarchical NUMA architecture, HECTOR (see Figure 1). Although 16 processors is a relatively small system, the impact of contention and the NUMAness of the system are clearly evident, and will only get worse in larger systems with faster processors. Hence, we believe that the general trends exhibited by the results of our experiments can be extrapolated to larger systems. Also, our results show the basic costs for a clustered system of inter- and intra-cluster operations, which are largely independent of the system size.

The prototype system used for experimentation uses Motorola 88100 processors running at 16.67 MHz, with 16KB data and instruction caches (88200), and a local portion of the globally addressable memory. At the lowest level, processors are connected by a bus. These lowest level units, called stations, are then connected together by a high-speed ring, with multiple rings connected together by higher-level rings, and so on. The particular hardware configuration used in our experiments consists of 4 processing modules per station and 4 stations connected by a ring. By comparison with other NUMA systems [1, 8, 19, 28, 30, 32] the cost of accessing remote memory on HECTOR grows relatively slowly as the system is scaled. The time to load a cache line grows from 19 cycles for local memory, to 29 cycles for a different memory module in the same station, up to 34 cycles when the memory module is in a different station.[2]

5.1 Basic Operations

In this section we present some of the basic costs of the system and compare intra-cluster versus inter-cluster times for some common operations.

Protected Procedure Calls (PPC) are only supported for local (on processor) communication, and hence their cost is independent of clustering. A null-PPC from a user application to a user-level server is 32.4 μsec, and drops to 19.2 μsec when directed to a kernel-level server (both are round-trip times). The performance of PPCs remains constant irrespective of the number of processors making concurrent requests.

Remote procedure calls are used for cross-cluster communication within the operating system kernel. The cost of a cross-cluster null-RPC in our implementation is 27 μsec, which includes the time to interrupt the target processor and the time to save and subsequently restore its registers.

[2] The HECTOR prototype on which we ran these experiments does not support hardware cache coherence, but page-based cache coherence is supported by the HURRICANE memory manager.

CSize-1	16 clusters with 1 processor each
CSize-2	8 clusters with 2 processors each
CSize-4	4 clusters with 4 processors each
CSize-8	2 clusters with 8 processors each
CSize-16	1 cluster with 16 processors

Table 1: The cluster configurations used for the synthetic stress tests and application benchmarks. **CSize-16** corresponds to a tightly-coupled system, **CSize-1** corresponds to a fully distributed system.

The cost to handle a read page-fault for an in-core page increases from 128 μsec within a cluster to 243 μsec across clusters. The additional overhead in the latter case is primarily due to the overhead of replicating the page descriptor and the extra directory lookup needed to locate the remote copy of the page. Subsequent faults by other processors on the remote cluster incur only the in-cluster cost.

The cost of a 32 byte send-response message transaction between two user-level processes increases from 328 μsec within a cluster to 403 μsec across clusters. Here the extra overhead is due to two RPCs needed to communicate between clusters and the allocation, initialization, and subsequent deallocation of message retainers at the target cluster.

5.2 Synthetic Stress Tests

To study the effects of clustering on throughput and response time, we designed several synthetic tests to stress the concurrency limits of the system. A message-passing test is used because message-passing is a key means of communication between the servers of a micro-kernel operating system. A test of the performance of page faults to in-core pages (soft page faults) is used, since such soft page faults are fairly common, as they required to support page-level cache coherence and page migration and replication, both important for performance in a NUMA multiprocessor. There are two groups of tests, both containing the message-passing and page fault test. The first group exercises the system for simultaneous operations to *independent* resources; the second group shows the performance for concurrent operations to *shared* resources.

The tests were run on each of the cluster configurations[3] of Table 1. The cluster configurations correspond to different degrees of coupling, and therefore allow us to explore the communication/contention trade-offs alluded to in Figure 4. We expect **CSize-1**, where each processor has its own set of operating system data structures, to perform well for coarse-grained parallel or independent applications, but to suffer from high communication costs for tightly-coupled applications. At the other end of the spectrum, **CSize-16** has one set of kernel data structures that are shared by all processors in the system. For this configuration, we expect contention for the shared data structures to be a significant portion of the response times.

5.2.1 Independent Operations

Figure 5 is a schematic depiction of the programs that stress simultaneous operations on independent resources. In the message passing test, neighboring pairs of processes repeatedly exchange 32-byte messages. In the page fault test, p processes repeatedly fault on a separate region of local memory. Figure 6 shows the response times and normalized throughput of the tests of Figure 5. Normalized throughput is calculated as $X(p) = T(1)/T(p) \cdot W(p)/W(1)$, where $T(p)$ is the response time for p processes, and $W(p)/W(1)$ reflects the amount of work done relative to the single operation case. In these tests, the p processes are assigned to separate processors in a way that minimizes the number of clusters spanned. That is, the processes

[3] Note that the hardware configuration does not change and is always 4 processors to a station, and the clusters are assigned to the hardware stations in a natural way.

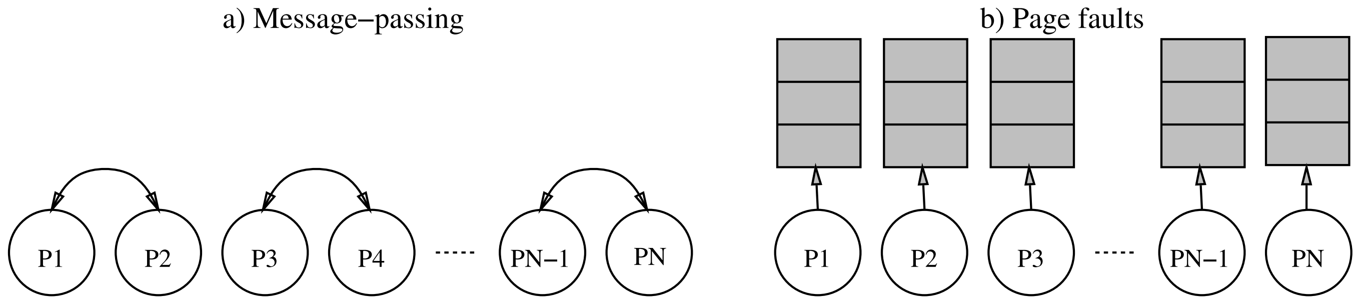

Figure 5: A schematic depiction of the programs that stress simultaneous operations on independent resources. In the message passing test (a), neighboring pairs of processes repeatedly exchange 32-byte messages. In the page fault test (b), p processes repeatedly fault on a separate region of local memory.

are first added to one cluster until each processor in the cluster has a process, before adding processes to the next cluster. This corresponds to a scheduling policy that attempts to assign related processes to the same cluster whenever reasonable.

Figure 6 (a) and (b) shows that for both tests, the response time increases until p is equal to the number of processors in a cluster, at which point lock contention within the cluster is maximized. For example, in the page fault experiments the address space structure and the top levels of the balanced binary tree of regions become the contended resources. As the processes “spill over” to the next cluster, the response times drop because the new processes experience no contention and thus lower the over-all average response times.

Observe that the response times for single operations (the lower left corner of the figure) increase slightly as the cluster size increases. This is due to NUMA effects in the larger clusters. Since there is only one operation in progress, there is no lock contention, but the probability that a particular data structure is remote increases with cluster size, since the shared kernel resources are distributed across all the processors of the cluster. Thus, the latency of kernel data accesses increases with cluster size.

Not surprisingly, the tests reveal that the smallest cluster size for each test has the best performance. Locality is maximized in this case and since the operations are independent, no cross-cluster operations are required. For large cluster sizes, the locks for common data structures suffer increasing contention, and hence, response times increase correspondingly.

In summary, the results show that when clusters are large, contention can severely affect performance, and even when there is not much concurrency, the loss of locality will adversely affect performance to some degree. While these results were obtained from a small system, it is clear that both affects would be stronger in a larger system. Also, since HECTOR is only mildly NUMA, we would expect these results to be stronger on systems with a greater degree of NUMAness (eg., [1, 8, 19, 30]).

5.2.2 Operations to Shared Resources

Figure 7 gives a schematic depiction of the programs that stress simultaneous access to shared resources. In the message-passing test, p children repeatedly send 32-byte messages to a single parent. In the page fault test, p children simultaneously write to the same small set of shared pages. Figures 8 shows the response times for the tests of Figure 7. This time, the response times are plotted as a function of cluster size, and the different curves in each figure represent different numbers of active children.

Figure 8 shows that an intermediate degree of coupling has the best performance for both tests. Also, the minimum response time for p processes occurs when the cluster contains p processors, which suggests that it may be beneficial to match the cluster size to the application communication patterns. When the cluster size is smaller, the cost of cross-cluster operations is incurred, while if the cluster size is larger, locality suffers.

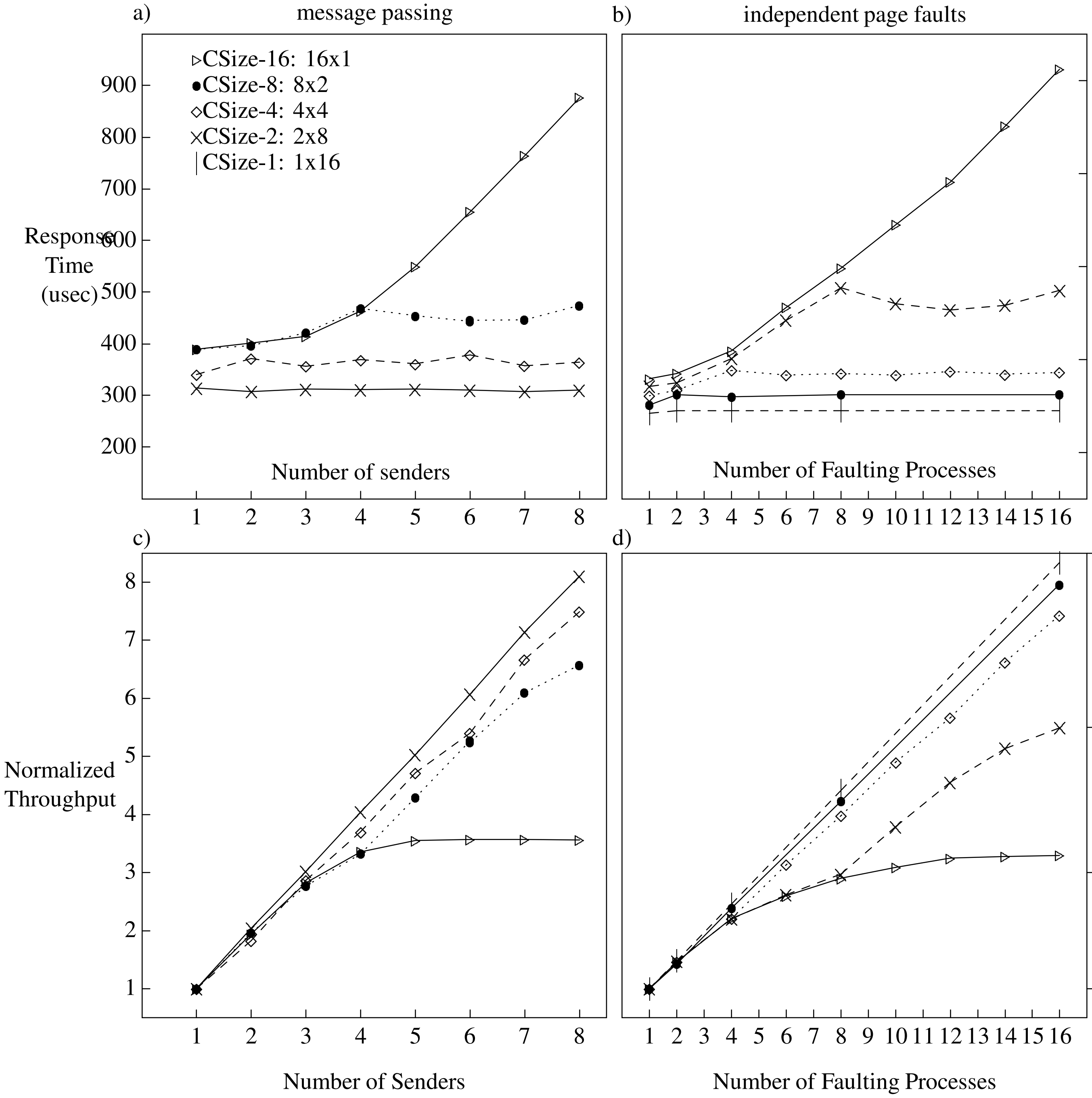

Figure 6: The response times and normalized throughput for simultaneous independent operations: (a) and (c) message passing; (b) and (d) page faults.

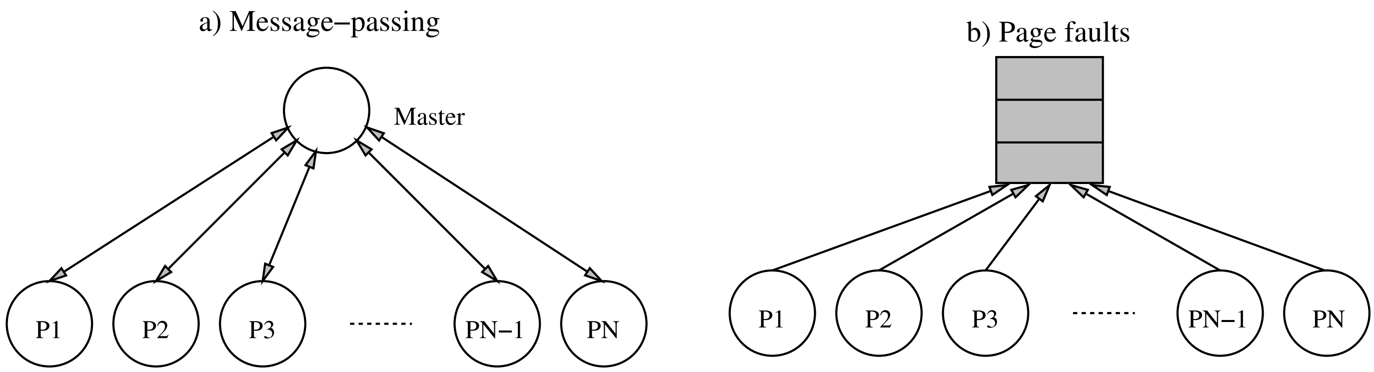

Figure 7: A schematic depiction of the programs that stress simultaneous access to shared resources. In the message-passing test (a), p children repeatedly send 32-byte messages to a single parent. In the page fault test (b), p children simultaneously write to the same small set of shared pages.

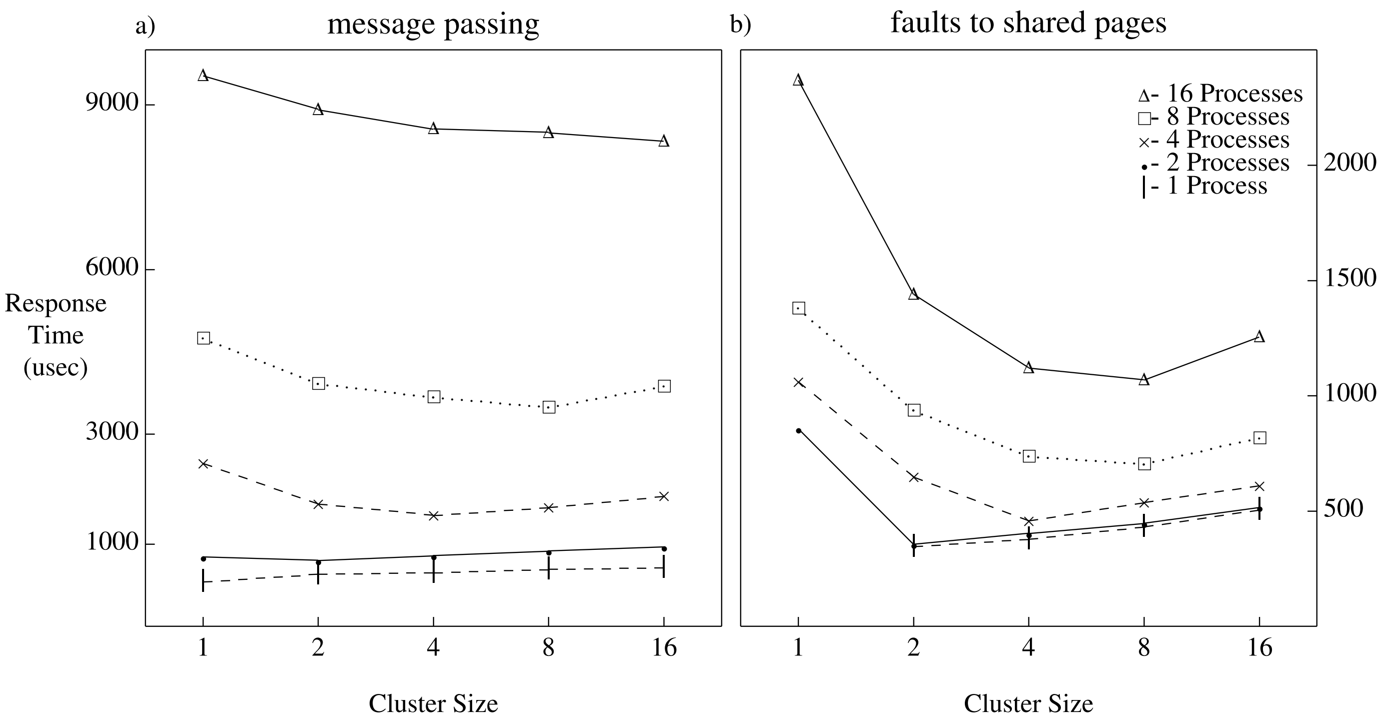

Figure 8: The response times for simultaneous operations to shared resources: (a) message passing; (b) page faults.

The exception to the matching observation occurs for the 16 process page fault test, where **CSize-8** achieves the best performance. This is due to contention for shared resources, which dominates for the large cluster sizes. For small cluster sizes, the cost of the inter-cluster communication is more dominant.

Depending on the expected workload, there are two ways these results might be extrapolated to systems larger than the prototype. Consider first the case of a workload consisting primarily of large numbers of small-scale parallel applications (16 processes or under) running on a significantly larger system (several hundred processors). The results in this section show that as the cluster size is increased beyond the size of the application, performance degrades due to a loss of locality in the operating system. This trend is likely to continue in future systems where we expect even higher remote memory access costs, suggesting that clusters should remain relatively small (but larger than the average parallelism of the applications) in order to efficiently support this type of workload.

The second case to consider is a workload of large parallel applications. Here the results are not as clear cut. Extrapolating the message passing test would lead to the conclusion that the cluster size should grow with the applications. However, the page fault test demonstrates the danger of increasing the cluster size too far, even for tightly coupled applications. In particular, for the 16 process case, the optimal cluster size is 8, with a sharp rise in cost for cluster size 16. On the other hand, a cluster size significantly smaller than the application also results in poor performance. We expect from these results that increasing the cluster size will result in a diminishing improvement in performance until ultimately an increase in cluster size will decrease performance, suggesting an intermediate cluster size should be chosen even for tightly coupled large scale applications.

The page-fault experiment of Figure 7 can be used to illustrate how additional levels of hierarchy can help performance on larger systems. To support coherence of read-shared pages at the time of a write, processors that are accessing the page must be notified. This is done using a spanning tree, where a single processor in each cluster is notified, and that processor is then responsible for notifying all other processors in its cluster. On larger systems with super-clusters, this spanning tree can be extended to further improve concurrency.

5.3 Applications

The synthetic stress tests of the previous section were designed to exercise a single aspect of the system, namely simultaneous independent operations and simultaneous operations to shared resources. However, real applications seldom incur only one or the other, and in fact exhibit both types.

This section examines the performance of three parallel applications as a function of cluster size. The applications are: SOR, a partial differential equation solver; Matrix Multiply; and 2D-FFT, which calculates the fourier transform of a two-dimensional array of data. All three programs are of the data-parallel, or SPMD, class of applications. However, the data access patterns of the applications are very different, so that each stresses a different aspect of the system.

Figure 9 shows the page fault behavior of the 3 applications when 4 worker processes are used. For each of the applications, the computation is partitioned along equal strips of the result matrix. The page fault behavior is inherent in the algorithm, and does not change[4] as the operating system configuration is varied. However, the cost of handling the page faults changes with the configuration, depending on whether the fault is to a shared page or not.

The SOR application exhibits concurrent faults to primarily independent pages, since the only shared data is along the shared edges of the strips of the matrix. Figure 9 shows that there is little paging activity after the initial iteration of the algorithm, so we expect this application to perform well for small cluster sizes.

[4] Time dependent accesses will change the page fault behavior.

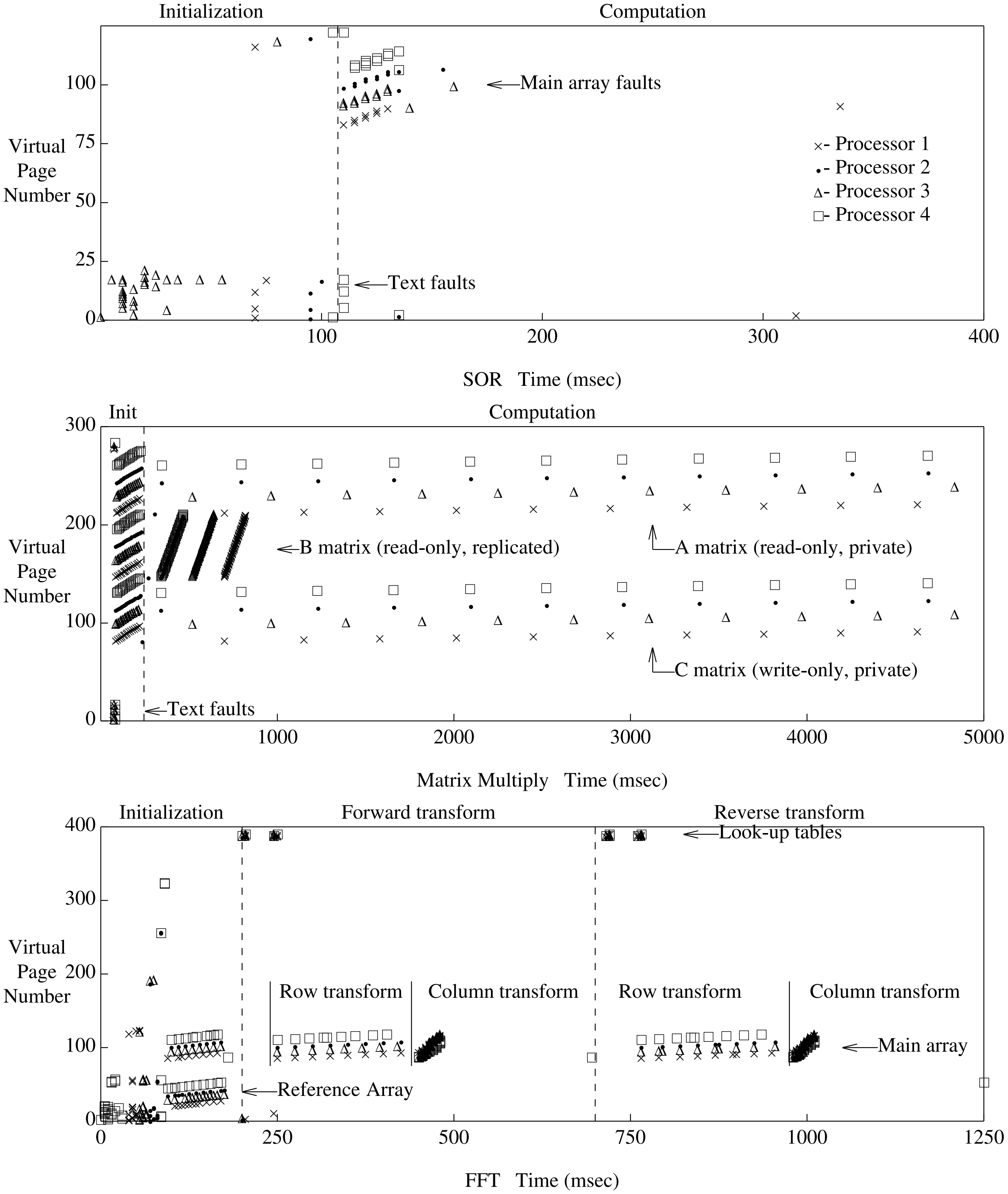

Figure 9: The page access and coherence faults of the three application programs (using four workers). The Y axis indicates the virtual page number (virtual address divided by the page size). The X axis gives the time in milliseconds. Each "point" represents a page-fault event for the corresponding process. Page fault events are caused either by the process mapping the page into its address space on its processor, or by writes trapped in order to support page-level coherence.

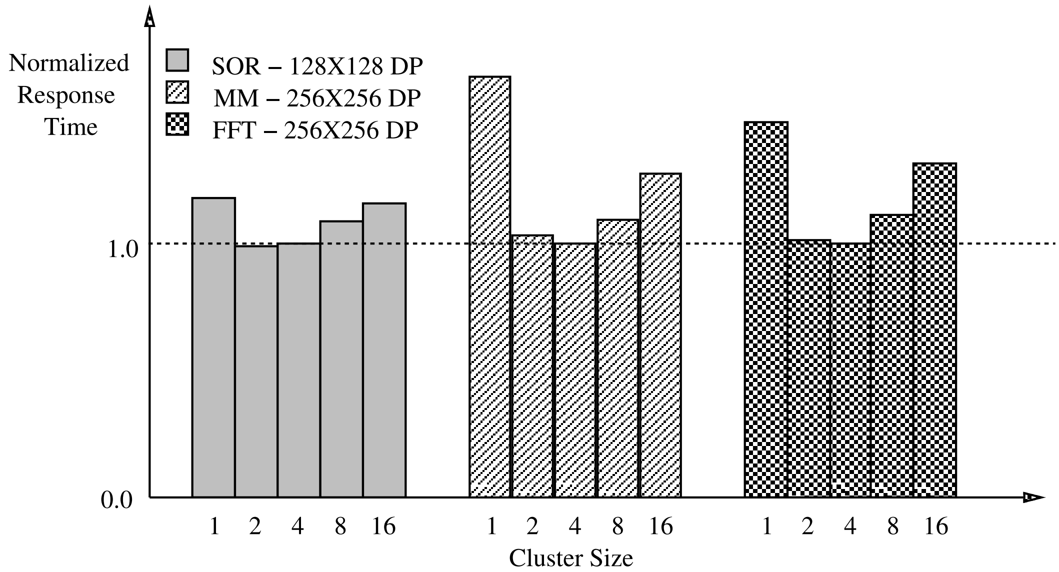

Figure 10: The normalized response times of the application suite for different data set sizes and cluster configurations.

In contrast, the Matrix Multiply and 2D-FFT programs have phases of intense concurrent accesses to both independent and shared pages. For Matrix Multiply (C = A $\times$ B), the program starts with a period of concurrent accesses to independent pages, as the workers initialize the matrices in parallel. The computation starts at time 200 msec and proceeds along the rows of C (and A). The accesses to these matrices are seen as the slowly increasing diagonals of fault instances in Figure 9. To compute a single row of C, each worker must read all the columns of B, which is seen as the sharp diagonals of faults in the figure. This access behavior stresses the memory manager's ability to handle simultaneous faults to shared pages.

The 2D-FFT application also begins with a flurry of faults to independent pages as the matrix (and a reference matrix to check the answer) are initialized by the workers in parallel. The forward transform begins along the rows at time 200 msec. At time 400 msec there is a period of intense coherence faults as the rows that had been accessed by a single worker are now accessed by all the workers to perform the column transform. The row-column fault pattern is repeated for the reverse transform, which begins at time 700 msec.

Figure 10 shows the response times for the three applications (on larger data sets than those shown in Figure 9) as the cluster size is varied from 1 to 16. The response times are plotted normalized to the response time of **CSize-4**. We are interested primarily in the general trends represented by each application, and hence the results for each application should be interpreted qualitatively and independently from each other. All the programs were run with 16 processes on 16 processors; only the cluster configurations were varied. The response times reported in this section are for the entire program, which includes the time to create, schedule, and destroy the worker processes, as well as the time to initialize any data structures. As a result, the times reported include the overhead due to both process and memory management. The algorithms (and problem sizes) used for these applications were chosen to stress the system and not necessarily because they were the best algorithms possible. By way of reference, the measured speed-ups for **CSize-4** were 13, 11, and 8 for SOR, Matrix-Multiply, and 2D-FFT, respectively.

The figure shows that all 3 programs perform best when the cluster size is 4. The performance of all 3 programs suffer when the cluster size is much larger or smaller than 4. The amount of degradation reflects application-specific system service demands, due in part to the differing page fault behavior.

At **CSize-1**, we see that the SOR program degrades only a small amount. This is because the workers

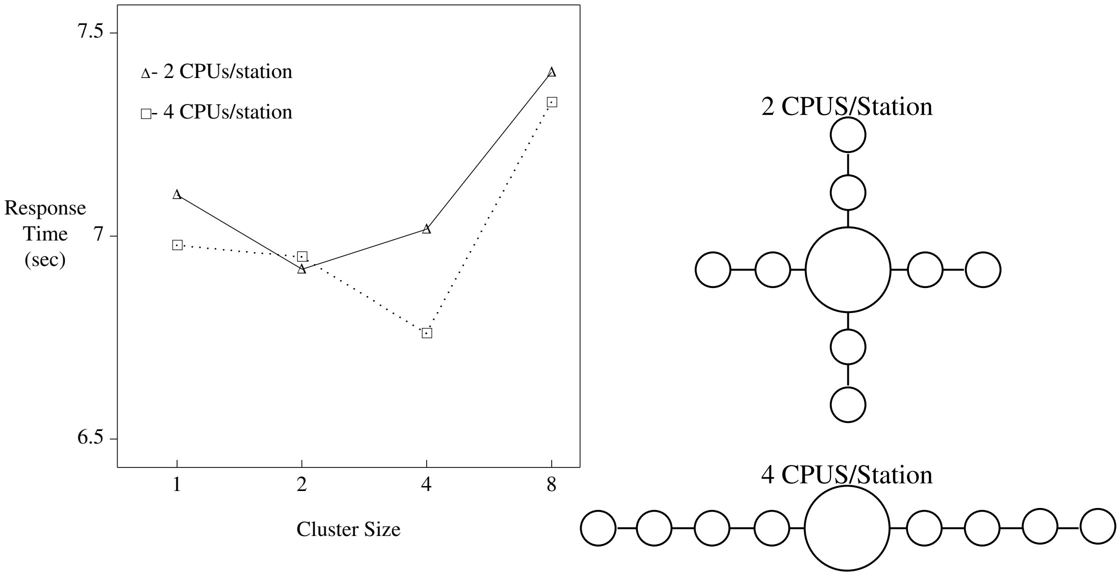

Figure 11: Matching the cluster size to the architecture for Matrix Multiply on 8 processors.

have primarily independent working sets, and so there is little shared data that must be kept consistent. The increase in response time for SOR over the response time of **CSize-4** is mostly due to the increased cost of migrating and destroying the worker processes on remote clusters during the initialization and termination phases.

Matrix Multiply is severely affected by too small a cluster size. This is actually a policy effect, since the default policy is to replicate read-only shared pages across clusters. Although this policy improves access times once the page is local, the cost of making 15 copies of the B matrix for **CSize-1** overshadows any gains in locality. Also, this degree of replication can cause the per-cluster working set to exceed the amount of available physical memory if the problem size is large. Using an intermediate cluster size thus helps to balance the trade-off between locality and space utilization.

At the other end of the coupling spectrum (**CSize-16**), 2D-FFT suffers greatly because of the high number of consistency faults that occur when the workers change phases from rows to columns. Matrix Multiply is also affected by contention, because of the high number of accesses to the shared pages of the B array (see Figure 9).

Again, given the significant impact visible for a relatively small system, the benefits of clustering are likely to be more pronounced on larger systems.

5.4 Effects of Hardware Configuration

An interesting question is whether the cluster size should match the application communication pattern or the architectural configuration. So far, we have seen examples where both were appropriate: the synthetic stress tests of concurrent accesses to shared resources showed the cluster size should match the application; the application performance curves suggested that the cluster size should match the architecture. To explore this question further, we ran the Matrix Multiply test on an 8 processor multiprocessor configured as 4 hardware stations of 2 processors each, and configured as 2 hardware stations of 4 processors each. For both configurations, the cluster size was varied from 1 to 8. Figure 11 shows the response times obtained for this experiment.

For both hardware configurations, the figure shows that the response time is minimized when the cluster size matches the architecture. Also note that 4 processors per station has generally better response time than 2 processors per station because there is less inter-station communication. The exception occurs at cluster size 2, where the 4 processors per station configuration suffers from having two clusters sharing a common bus for independent accesses.

6 Lessons learned

Our experiences with HURRICANE have demonstrated that Hierarchical Clustering can exploit the advantages of both small scale tightly coupled systems and large scale distributed systems. The tight coupling within a cluster results in performance comparable to small scale shared memory systems for the common case, where interactions occur primarily between objects located in the same cluster. On the other hand, because system services and data are replicated and migrated to different clusters in the system, there is much locality and less sharing and hence fewer bottlenecks for independent operations. Hence we expect a system based on Hierarchical Clustering to scale well.

In the course of our work on HURRICANE, we have learned a number of lessons which suggest changes that could improve the performance and reduce the complexity of an operating system implemented using Hierarchical Clustering.

Locking protocols

The locking protocols for within and across clusters were developed independently, adding much complexity to the implementation. Within a cluster, we enforce an ordering on the acquisition of locks to avoid deadlock. Because lock contention is often expected to be low with reasonable cluster sizes, many locks are implemented as (MCS [31] distributed) spin locks, where a process requesting a lock does not release the processor it is spinning on (essentially holding a lock on the processor). For inter-cluster operations, we release all local locks (including the processor) before going remote in order to avoid deadlock.

We found that releasing locks for remote operations results in an excessively large discrepancy between the cost of local and remote operations and adds much complexity to the code. As an example of the complexity, many operations in the kernel have three versions: one that implements the local case, another for the client side of the RPC, and a third for the corresponding server side. Releasing locks for remote operations also makes it more difficult to ensure fairness, since local requests have a higher probability of acquiring a contended resource. To address the cost and fairness problems that arise from releasing locally held locks for remote operations, we modified the implementation to release locally held locks only if the locks required at the remote site are not free. This optimization resulted in even greater code complexity.

We now believe that the locking protocol should have been developed independent of the clustering. We are currently investigating the merits of imposing a system wide ordering on the acquisition of all locks in the hopes of reducing code complexity. For this approach to work, some of the locking structures must be changed, increasing the number of locks that must be acquired for some critical paths. We intend to study the trade-off between simpler code and the increased overhead.

Relationship between clustering and policies

In our original implementation, the cluster size determined both the structures used to manage resources and a framework for invoking policies on those resources. We have since experimented with de-coupling the cluster size and the policies. For example, it is now possible to request that pages of a memory region be replicated or migrated to the local processor, even if there is already a copy of the page on another processor in the local cluster [21]. For some applications, we have found that this approach results in substantial performance improvements.

Fixed cluster sizes

When HURRICANE is booted, a fixed sized cluster is established for all resources. We believe that more flexibility could result in performance improvements; in particular, we believe that each class of physical (eg., pages, processors, disks) and virtual (eg., address spaces, regions, files) resource could be clustered independently with little increase space requirements or complexity. As an example, the HURRICANE PPC facility manages resources more naturally on a per-processor rather than a per-cluster basis.

Other experiences

In our current implementation, the RPC service load is balanced by having processor i of each cluster direct its RPC requests to processor i of the target cluster. We would like to investigate more dynamic load balancing approaches; an example would be to direct an RPC request to any idle processor in the target cluster.

Some operating system services can affect all clusters on which that application is running. For example, when the state of a page used by all processes is changed, all processors must be notified. To support these operations efficiently, we have implemented an asynchronous RPC facility that allows a particular cluster to simultaneously invoke code on a number of (or all) other clusters. For larger scale systems, we expect to use the clustering hierarchy to implement a spanning tree broadcast that will allow not only the servicing but also the distribution of the asynchronous RPC operations to be done concurrently.

As discussed in the implementation section, local structures (representatives) are used to represent remote resources, such as processes or physical pages. In the worst case, the number of remote resources accessed by a cluster will grow proportionally to the number of clusters in the system. If we allowed the number of representatives to grow at this rate, then clearly our approach would not be scalable. However, for well behaved applications, we expect that the average number of cross-cluster interactions that each cluster will have to support will remain constant as the system size is scaled. Therefore we believe we can obtain good performance with a fixed number of representatives per cluster, paying a somewhat higher cost to support the case where the fixed set is depleted. A proper study of the impact of this decision is still required.

7 Concluding Remarks

We have introduced the concept of Hierarchical Clustering as a way to structure operating systems for scalability, and described how we applied this structuring technique to the HURRICANE operating system. We presented performance results that demonstrate the characteristics and behavior of the clustered system. In particular, we showed how clustering trades off the efficiencies of tight coupling for the advantages of replication, increased locality, and reduced lock contention, but at the cost of cross-cluster overhead.

Concepts similar to Hierarchical Clustering have been proposed by a number of other researchers [2, 16, 18, 39]. One of the major contributions of our work is that we have implemented a complete operating system based on this structuring methodology. To the best of our knowledge, HURRICANE is the only complete and running implementation of such a system. Our work gives a degree of confidence that systems structured in such a fashion can be reasonably implemented, a confidence that cannot be obtained from a paper design. The lessons learned from this implementation were not obvious (except perhaps in hindsight) and should influence future systems designed with similar structures.

We are now embarking on the design of a new operating system, TORNADO, for a new shared memory multiprocessor. TORNADO will incorporate the lessons we learned from our experiences with Hierarchical Clustering on HURRICANE. In particular: 1) the locking protocol will be independent of the clustering, 2) each virtual and physical resource class will use its own cluster size, and 3) the cluster size and the policies will be largely de-coupled.

Our experiences with scientific applications on HURRICANE/HECTOR has inspired two new research directions for TORNADO. First, we are investigating a new methodology called *Physical Resource-based Virtual Machines* whose goal is to guarantee predictable performance, eliminating interference from other concurrently running but independent applications. This will allow compilers and applications to make optimizations that depend on predictable physical resource allocation. Second, we are investigating operating system policies and structures that will work well for applications with large data sets (eg., data sets that do not fit in main memory).

Acknowledgments

We wish to thank David Blythe, Yonatan Hanna, and Songnian Zhou for their significant contribution to the design and implementation of HURRICANE. We also gratefully acknowledge the help of Tim Brecht, Jan Medved, Fernando Nasser, and Umakanthan Shunmuganathan.

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[2] I. Ahmad and A. Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Transactions on Software Engineering*, 17(10):987–1004, October 1991.

[3] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Par. and Dist. Sys.*, 1(1):6–16, Jan 1990.

[4] Ramesh Balan and Kurt Gollhardt. "A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines". In *Summer '92 USENIX*, pages 107–115, San Antonio, TX, June 1992.

[5] David Barach, Robert Wells, and Thomas Uban. Design of parallel virtual memory management on the TC2000. Technical Report 7296, BBN Advanced Computers Inc., 10 Moulton Street, Cambridge, Massachusetts, 02138, June 1990.

[6] A. Barak and Y. Kornatzky. Design principles of operating systems for large scale multicomputers. Technical Report RC 13220 (#59114), IBM T.J. Watson Research Center, 10 1987.

[7] BBN Advanced Computers, Inc. *Overview of the Butterfly GP1000*, 1988.

[8] BBN Advanced Computers, Inc. *TC2000 Technical Product Summary*, 1989.

[9] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for NUMA memory management. In *Proc. 12th ACM Symp. on Operating System Principles*, pages 19–31, 1989.

[10] T.B. Brecht. On the importance of parallel application placement in NUMA multiprocessors. In *SEDMS IV. Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 1–18. Usenix Assoc, September 1993.

[11] M. Campbell, R. Holt, and J. Slice. Lock granularity tuning mechanisms in SVR4/MP. In *SEDMS II. Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 221–228. Usenix Assoc, March 1991.

[12] 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.

[13] E. Chaves, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. In *Proc. Second Symposium on Distributed and Multiprocessor Systems*, pages 105–116, Atlanta, Georgia, March 1991. Usenix.

[14] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 120–133, 1993.

[15] David R. Cheriton. "The V Distributed System". *Communications of the ACM*, 31(3):314–333, March 1988.

[16] D.R. Cheriton, H. Goosen, and P. Boyle. ParaDiGM: A highly scalable shared-memory multi-computer architecture. *IEEE Computer*, 24(2), February 1991.

[17] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proc. 12th ACM Symp. on Operating System Principles*, pages 32–44, 1989.

[18] D.G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–81, May 1990.

[19] 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.

[20] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. Technical Report 294, CSRI, University of Toronto, 1993.

[21] Benjamin Gamsa. Region-oriented main memory management in shared-memory NUMA multiprocessors. Master's thesis, Department of Computer Science, University of Toronto, Toronto, September 1992.

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

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

[24] O. Krieger, M. Stumm, and R. Unrau. Exploiting the advantages of mapped files for stream I/O. In *Proc. 1992 Winter USENIX Conf.*, 1992.

[25] 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.

[26] Orran Krieger and Michael Stumm. HFS: a flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[27] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer*, 27(3):75–83, March 1994.

[28] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. Parallel supercomputing today and the Cedar approach. *Science*, pages 967–974, 1986.

[29] R.P. LaRowe Jr., C.S. Ellis, and L.S. Kaplan. Tuning NUMA memory management for applications and architectures. In *Proc. 13th ACM Symp. on Operating System Principles*, October 1991.

[30] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th Intl. Symp. on Computer Architecture*, 1990.

[31] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Comp. Sys.*, 9(1), Feb. 1991.

[32] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. "The IBM Research Parallel Processor Prototype". In *Proc. 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[33] M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos, and N. G. Smithline. Implementation issues for the psyche multiprocessor operating system. *Computing Systems*, 3(1):101–137, Winter 1990. psyche WEBDMS.

[34] H. A. Simon. *The Sciences of the Artificial*. MIT Press, 2nd edition, 1985.

[35] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Experiences with locking in a numa multiprocessor operating system kernel. In *OSDI Symposium*, Nov 1994. accepted for publication.

[36] 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.

[37] Z.G. Vranesic, M. Stumm, D. Lewis, and R. White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1), January 1991.

[38] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, j. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An OSF/1 UNIX for massively parallel multicomputers. In *USENIX Winter Conference*, pages 449–468. Usenix, January 1993.

[39] S. Zhou and T. Brecht. Processor pool-based scheduling for large-scale NUMA multiprocessors. In *Proc. ACM Sigmetrics Conference*, September 1990.