Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design

Ron Unrau, Michael Stumm, and Orran Krieger

Technical Report CSRI-268
March, 1992

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design

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

Abstract

We propose a simple structuring technique based on *clustering* for designing scalable shared memory multiprocessor operating systems. Clustering has a number of advantages. First distributed systems principles are applied by distributing and replicating system services and data objects to increase locality, increase concurrency, and to avoid centralized bottlenecks, thus making the system scalable. However, since there is tight coupling within a cluster, the system performs well for local interactions. Second, a clustered system can easily be adapted to different hardware configurations and architectures by changing the size of the clusters. Third, clustering maximizes locality which is key to good performance in large NUMA systems. Finally, clustering leads to a modular system composed from easy-to-design and hence efficient building blocks.

In this paper, we describe clustering and how it is applied to a micro-kernel-based operating system. We present measurements of performance, as obtained from a working prototype implementation.

1 Introduction

Considerable attention has been directed towards designing "scalable" multiprocessor hardware, capable of accommodating a large number of processors. These efforts have been successful to the extent that a number of such systems exist. However, scalable multiprocessor hardware can only be cost effective for general purpose usage if the operating system is as scalable as the hardware. This paper addresses scalability in operating system design for such scalable shared memory multiprocessors as the Stanford Dash [15], the Kendall Square Research KSR-1 [6], the University of Toronto Hector [18], the BBN TC2000 [4], and the IBM RP3 [17].

Typically, existing multiprocessor operating systems have been scaled to accommodate a large number of processors in an ad hoc manner, by repeatedly identifying and then removing the most contended bottlenecks. This is done 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 a system that 1) is fine-tuned for a specific architecture and hence is not easily portable to other hardware bases with respect to scalability; 2) is not scalable in a generic sense, but only until the next bottleneck is reached; and 3) has a large number of locks that need to be held for common operations, with correspondingly large overhead.

We believe that a more structured approach to design scalable operating systems is necessary, and propose a new structuring technique based on clustering and hierarchical system design. In particular, it addresses the following set of goals:

1. *Scalability*: The operating system should run efficiently on a large number of processors, and make the potential of the hardware base available to application programs. However, the system should not penalize small-scale parallel or sequential applications.

2. *Adaptability*: At minimum, the operating system should be easily adaptable (that is, without having to rewrite code) to run efficiently on different configurations of a given architecture. More ideally, the operating system should be adaptable to run efficiently on different (shared memory) architectures.

3. *Modularity and Simplicity*: The design of the operating system should be amenable to modern software engineering principles: it should be structured using modular components or basic building blocks that together form the entire system.

Clustering incorporates structuring principles from both tightly-coupled and distributed systems, and attempts to exploit the advantages of both. On the one hand, scalability is obtained by using the structuring principles of distributed systems, where services and data are replicated: a) to distribute the demand, b) to avoid centralized bottlenecks, c) to increase concurrency, and d) to increase locality. 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.

Clustering is described in more detail in Section 3, together with its advantages. In Section 4 we argue that a clustered system can be implemented with little increase in complexity by briefly describing an implementation we have developed. We then present the results of performance measurements obtained from our implementation, and we conclude in Section 6. First, however, we consider the meaning of scalability in the context of operating systems and the implications it has.

2 Requirements for Scalability

It is surprisingly difficult to give a formal definition that characterizes scalability in a general sense, especially for operating systems. For example, one of the better known formal definitions is by Nussbaum and Agarwal [16][1], but is not applicable to operating systems for several reasons. First, they exclude the operating system from their considerations by treating it as an extension of the hardware. Second, the definition is based on asymptotic limits and thus provides no hints as to how to design or build a scalable (operating) system. Finally, the definition is based on speedup, and hence the response time, which we do not believe to be appropriate for operating systems. For parallel systems, we intuitively do not expect the response time of an operating system call to speed up as more processors are added to the system; rather, we are content if the system call does not take longer to complete as additional processors are added. We believe that throughput and concurrency are the dominant issues in scalable operating system design. Since one can expect the demand on the operating system to increase proportionally with the size of the system, the throughput must also increase proportionally, which is possible only if the operating system is sufficiently concurrent.

[1] Nussbaum and Agarwal's definition is: *The scalability of a machine for a given algorithm is the ratio of the asymptotic speedup on the real machine and the ideal realization of an EREW PRAM.*

Instead of attempting to develop a definition of our own, we identify several necessary requirements for an operating system to be scalable by considering throughput and utilization formulas from elementary queueing theory [14]. If c is a type of OS service request, then $\lambda_c(p)$, the arrival rate of requests for this service, can be expected to grow linearly with the number of processors in the worst case. Each service request c will require $v_{c,k}(p)$ visits at service center k, where k represents an operating system resource, and each such visit will require a service time of $s_{c,k}(p)$. Note that in general, both $v_{c,k}(p)$ and $s_{c,k}(p)$ are functions of p.

The utilization of resource k devoted to servicing requests for c is then:

$$U_{c,k} \;=\; X_{c,k} \cdot s_{c,k} \;=\; \lambda_c(p) \cdot v_{c,k}(p) \cdot s_{c,k}(p)$$

Because a system cannot scale if any of its resources are saturated, overall utilization of resource k cannot be larger than one:

$$1 \geq U_k \;=\; \sum_c U_{c,k} \;=\; \sum_c \lambda_c(p) \cdot v_{c,k}(p) \cdot s_{c,k}(p)$$

Therefore, since λ_c is expected to increase with p, the following set of requirements must hold:

1) The time spent at resource k to service a request c, $s_{c,k}(p)$, must be bounded by a constant, for otherwise, resource k will begin to saturate. Hence, there must be a high degree of locality in the data accesses when servicing c at resource k, and the time to access data structures when servicing c must be bounded and independent of p.

2) The total number of visits to any operating system resource k ($\sum_c v_{c,k}(p)$) must be bounded by a constant. This is only possible if the number of service centers increases proportionally with p.

Additionally, since in the worst case λ_c may increase proportionally with p, and since the number of service centers can increase at most proportionally with p:

3) The total number of visits to all service centers for request c ($\sum_k v_{c,k}(p)$) must not increase with p (or otherwise, $\sum_k s_{c,k}$ must decrease by a corresponding amount).

The above requirements can be translated into the following design guidelines:

Preserving parallelism: *The operating system must preserve the parallelism afforded by the applications.* This follows directly from requirement 2 above. If several threads of an executing application (or of independent applications running at the same time) request independent operating system services in parallel, then they must be serviced in parallel. Otherwise, the operating system becomes a bottleneck, limiting scalability and application speedup. Because we do not expect parallelism in servicing a single operating system request, and because an operating system is primarily demand driven, parallelism can only come from application demand. Therefore, the number of operating system service points must increase with the size of the system and the concurrency available in accessing the data structures must grow with the size of the system to make it possible for the overall throughput to increase proportionally.

Bounded overhead: *The overhead for each independent operating system service call must be bounded by a constant, independent of the number of processors* [2]. This follows directly from requirements 1 and 3. If the overhead of each service call increases with the number of processors, the system will ultimately saturate, so the demand on any single resource cannot increase with the number of processors. For this reason, system wide ordered queues cannot be used and objects cannot be located by linear searches if the queue lengths or search lengths increase with the size of the system. Broadcasts cannot be used for the same reason.

Preserving locality: *The operating system must preserve the locality of the applications.* It is important to consider the memory access locality in large-scale systems, because for example, many large-scale shared memory multiprocessors have non-uniform memory access (NUMA) times, where the cost of accessing memory is a function of the distance between accessing processor and the target memory, and because cache consistency incurs more overhead in a large systems. Locality can be increased a) by properly choosing and placing data structures within the operating system, b) by directing requests from the application to nearby service points, and c) by enacting policies that increase locality in the applications' memory accesses. For example, policies should 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 direct file I/O to devices close by. Within the operating system, descriptors of processes that interact frequently should lie close together, and memory mapping information should lie close to the processors which must access them to handle page faults.

3 Clustering

We believe that an operating system designed using the structuring principles based on clustering and described in this section will largely meet the above requirements for scalability, and is conducive for enacting appropriate policies that enhance locality (although we do not intend to address policy issues in this paper).

The basic unit of structuring for the operating system is a *cluster*, which provides the functionality of a complete, efficient, small-scale symmetric multiprocessing operating system. Kernel data and control structures are expected to be shared by all processors within the cluster, giving good performance for fine-grained communication. 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.

On larger systems, multiple clusters are instantiated so that each cluster manages a unique group of "neighboring" processing modules (including processors, memory and disks), where neighboring implies that in the absence of contention memory accesses within a cluster are never more expensive (and usually cheaper) than memory accesses to another cluster. 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. Clustering incorporates structuring principles from both tightly-coupled and distributed systems. By using the structuring principles of distributed systems, services are naturally replicated to distribute the demand, to avoid centralized bottlenecks, and to increase locality. The structuring principles of small-scale multiprocessors allow fine-grained data sharing for the common case where communication is local.

In a clustered system, many important design issues are similar to those encountered in distributed systems. For example, shared objects can be distributed and migrated across clusters to increase locality and decrease contention, but then it must also be possible to find them. Other objects may be replicated to increase locality and decrease contention, but then consistency becomes an important issue.

Despite the similarities between a clustered system and a distributed system, there are important differences that lead to completely different design tradeoffs. For example, in a distributed system 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. Moreover, 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 large issue.

Clustering leads to a number of direct advantages. First, it provides a framework for managing locality. For the case of small-scale parallel or sequential programs, all processes are scheduled onto the same cluster. This enhances performance as all interactions are local. Large-scale applications are scheduled across multiple clusters, and can benefit from the concurrency afforded through replicated system services; the components of the large-scale application typically request service from local servers.

Second, clustering allows performance tuning to different configurations and architectures by allowing the size of the clusters to be adjusted. The appropriate cluster size for a system is affected by several factors, including 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 [12], Dash [15], KSR-1 [6], or Hector [18], a cluster might correspond to a hardware station. On a local-remote memory architecture, such as the Butterfly [3], 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.

Finally, clustering simplifies lock structuring issues, and hence reduces code complexity, which can lead to improved performance and scalability. For example, Chaves reports that the fine-grained locking used in an unclustered system significantly increases the length of the critical path, even when there is no lock contention [7]. As well, deadlock can be a problem when several fine-grained locks must be held simultaneously. Because contention for a lock is primarily limited to the number of processors in a cluster, clustering allows for coarser grained locking, as we show in Section 5.

Using clusters to structure an operating system for scalable systems also presents several challenges. One challenge is to completely hide from the applications 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.[2]

Another challenge is to keep the complexity introduced by separating the system into clusters within reasonable bounds; clusters were introduced as a structuring mechanism primarily to reduce overall complexity. Therefore, the performance of each cluster should be similar to that of a small-scale multiprocessor operating system. For those applications with a small degree of parallelism (or sequential applications) we expect the vast majority of all system interactions to be intra-cluster, and even for larger applications that span multiple clusters, we expect most of the system interactions to be intra-cluster, because of the replication of system services. Nevertheless, the overhead of inter-cluster communication should be minimized in order not to overly penalize those cases where inter-cluster communication is necessary.

4 Implementation

In this section, we show how clustering affects the structure of the primary operating system components. While the principles of clustering are applicable to many OS philosophies, the discussion is presented using examples from the *Hurricane* operating system, a prototype clustered system we have implemented for the Hector multiprocessor.

The Hurricane kernel, which is similar in structure to the V kernel [8], provides for 1) address spaces, 2) processes, and 3) message passing. The address spaces are (initially empty) containers in which an arbitrary number of processes can run. Through the memory manager (and indirectly through various file servers), regions of the address space are bound to file regions before they can be used. Once a file is bound to a virtual address space, all references to memory are effectively references to the corresponding locations in the file, thus providing for a single-level store abstraction, where main memory is considered a cache of secondary store.

Multiple processes can run within an address space. Processes within an address space typically commu-

[2] Clusters can be made visible to sophisticated applications to allow performance optimizations.

nicate through shared memory, but messages are used to communicate between processes in different address spaces, and in particular between application and server processes.

Services not provided for by the kernel are provided for by servers. Some of these servers run in kernel space for protection (and sometimes for performance) reasons; examples are the device server that has to service interrupts, and the memory server that manages address spaces. The other servers run as normal user-level processes; they include the file server, the program manager, the pipe server, and the internet server.

4.1 The Hurricane Kernel

The kernels of each cluster in the system communicate and cooperate in order to provide the processes and users a consistent view of the system. While shared memory is used as the primary mode of communication within a cluster, three different mechanisms can be used for the kernels to communicate across clusters.

First, shared memory access remains a viable option for cross cluster communication, particularly for light-weight operations that make only a few references. For example, shared memory is used to locate the current position of a process descriptor. 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 is maintained at the home cluster. As a process migrates from cluster to cluster, its location information at the home cluster is updated each time. Remote shared memory access to this data structure is appropriate, because this information is maintained in a hash table so that only a small number of references are needed for lookup.

Remote procedure calling is a second mechanism for cross-cluster communication. It is implemented using remote interrupts; the interrupted cluster obtains the call information and arguments through shared memory directly. The use of remote procedure calls allow data accesses to be local to the interrupted cluster, but the cost of the interrupt (i.e. the cycles that are stolen and the register saving and restoring) must be amortized.

Finally, message passing can be used to communicate between processes on different clusters. For message passing within a cluster, the message is copied into the descriptor of the sending process, which in turn is queued in the message queue of the target process descriptor. For message passing across clusters, remote procedure calls are used to interrupt the target cluster to 1) allocate a message retainer (similar to a process descriptor), 2) copy the relevant information including the message by directly accessing the source processor descriptor on the source cluster, and 3) add the message retainer to the message queue of the target descriptor. The actions between the source and the target cluster are similar to the actions taken by the V kernels on different hosts, except for the fact that data is passed through shared memory directly instead of through a network packet. Other message passing primitives, such as Reply and Forward, are implemented in a similar fashion. Note that by using local message retainers as opposed to the source process descriptors which are remote, the message queue will always be an entirely local data structure.

Having the size of the clusters be 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. it limits the amount of searching for ready processes when dispatching.

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, leading to better system throughput and reduced application response time.

4.2 Memory Management

The memory management sub-system is responsible for virtual resources: the address spaces and their corresponding memory regions; and for the management of physical resources: the physical pages of memory and hardware support. Virtual resources are managed by servers on each cluster, which accept application requests to, for example, allocate and deallocate address spaces and regions. All memory-related operations at the physical level are on a per page basis, and are primarily demand-driven, which means they are initiated by the hardware as the result of translation or protection exceptions.

Clustering increases locality of accesses to the data structures of the memory manager, since each cluster maintains its own set of data structures to manage both the virtual and physical resources of local processes. This improves performance, since the structures that manage private data (such as process stacks) are local to the cluster on which the process executes. When applications share resources across clusters, the data structures that manage these resources can typically be replicated and cached locally, preventing bottlenecks and increasing concurrency. Virtual resources are kept consistent by directing all modifications through the home cluster, which thus serves as a point of synchronization. Since address space modifications are infrequent relative to the number of accesses due to page faults, this synchronization point has so far not been a bottleneck. Hurricane uses a simple directory mechanism to maintain the consistency of physical pages across clusters. There is one directory entry for each valid file block currently resident in memory; the entry identifies which clusters cache the page, and serves as the synchronization point for cross-cluster page operations. The directory entries are distributed across the processors of the system, allowing concurrent searches and balancing the accesses evenly across the system.

Clustering also provides a framework for enacting paging policies. At this time, the default policy is to share physical pages within a cluster, but to replicate and migrate pages across clusters. Several research groups have shown that page-level replication and migration policies can reduce access latency and contention for some applications [5, 9, 13]. However, the overhead of these policies must be amortized to realize a net gain in performance. On machines where the local-remote access ratio is high, relatively few local accesses are sufficient to recoup these costs. However, technology advances have permitted this gap to narrow, and have allowed increases in hardware cache size. Both these trends mean that more local accesses are required to justify a page movement, which argues for less aggressive placement/replication policies. Moreover, replication lowers the effective utilization of memory by increasing the resident set size of an application, which could lead to increased disk paging when the level of multi-programming is high. The current default policy therefore limits the amount of replication or migration, reducing overhead, but still allows for the reduced latency and increased concurrency of localized accesses to replicated data.

4.3 The File System

File system responsibilities can be divided into three levels: name space management; open file state management; and the handling of I/O. In Hurricane[3], these services are provided on a per cluster basis, and all application requests are directed to local servers.

Since changes to the name space are relatively localized, file names and directories are replicated across clusters. Consistency of replicated entries is maintained through an updating mechanism (instead of invalidating). This approach localizes name space searches and allows them to proceed in parallel across clusters.

Open files are seldom shared between programs, so open file state is generally maintained on the cluster where the file was opened. In two cases the open state may become used at other clusters, namely: 1) the process that opened the file passes the file handle as a capability to another program, or 2) several processes of a program spanning multiple clusters are accessing the file. In these cases the open file state is replicated on demand from the home cluster (the id of which is encoded in the file handle), which thus serves as the point of synchronization for state updates.

For those operations that require I/O, we believe clustering can provide a framework for balancing the load across the disks of the system. Although this part of the file system has not yet been implemented we believe it would be beneficial if the blocks of individual files could be distributed and replicated across a number of disks. All requests for disk I/O from a particular cluster would still be directed to the local cluster, but the request is forwarded to the appropriate server on another cluster if it cannot be handled locally.

4.4 Scheduling

The primary purpose of the scheduling subsystem is to keep the loads on the processors (and possibly other resources, such as memory) well balanced, in order to decrease the average response time of the applications. In a clustered system, the processes of an application are scheduled to run in a single cluster, unless there are performance advantages for a job to span multiple clusters. Hence, for parallel programs with a small number of processes, all of the processes will run on the same cluster. For larger-scale parallel programs that span multiple clusters, the number of clusters spanned is minimized. These policies are motivated by simulation studies[19], which have shown that clustering can noticeably improve overall performance[4].

The scheduling decisions are divided into two levels. Within a cluster, the load between the processors is balanced at a fine granularity through the dis-

[3] A more complete description of the Hurricane file system can be found in [11]

[4] A similar structuring mechanism for scheduling has been proposed by Feitelson and Rudolph [10], and by Ahmad and Ghafoor [1].

patcher (in the micro-kernel). This fixed scope limits the load on the dispatcher itself and allows local placement decisions to be made in parallel. 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.

4.5 Super Clusters

A single level of clusters can be expected to effectively support moderately large systems (in the, say, 100–200 processor range). However, for larger systems, additional levels in the hierarchy will probably be necessary. In particular, while each cluster is structured to maximize locality, there is no locality in cross-cluster communication. Examples of this are the home cluster concept for address spaces and the directory for locating pages. The logical next step is to group clusters into super clusters. Processor load balancing is an obvious candidate for this hierarchical clustering. A high-level process manager schedules processes between super clusters, while lower-level managers schedule processes within a super cluster.

The introduction of super clusters should not affect the lower levels of the system. For example, the micro-kernel requires no changes. Super clusters can also be applied to manage memory and I/O. For example, in the memory management, the single-level directories could be replaced with a hierarchical one, and multiple copies of the home address descriptor may be necessary; say one per super cluster.

5 Experimental Results

In this section, we present the results of simple experiments and performance measurements. All our experiments were conducted on a 16 processor Hector shared-memory multiprocessor running the Hurricane operating system. Hector is a hierarchical multiprocessor designed for scalability [18], where a number of processing modules are connected by bus to form *stations*, which in turn are connected by a hierarchy of rings. The processing modules of the current implementation contain a Motorola 88000 processor, up to 128 Kbytes instruction and 128 Kbytes data cache and 16 Mbytes RAM. The particular configuration used in our experiments consists of 4 processing modules per station and 4 stations connected by a ring, and runs at 20 MHz.

Hector is a NUMA system in that the memory access times differ depending on the distance between accessing processor and target memory, although the difference on Hector is typically less pronounced than on other NUMA systems, such as the BBN TC-2000. In our configuration, it takes 20 cycles to fill a 16 byte cache line from local (on-board) memory, 26 cycles if filled from off-board but on-station memory, and 30 cycles if filled from the memory of another station (in the absence of contention). The differences in memory access times can be more pronounced if there is memory contention, so the non-uniform memory access times in Hector can still affect performance in a significant way. For example, we can compare the execution time of a 512×512 matrix multiply in a system configured as 4 clusters containing 4 processors each, where the clusters are mapped onto the hardware in different ways. If the clusters correspond to the hardware stations, then the matrix multiplication takes 6% less time to complete than when each processor of a cluster is assigned to a different station; and it takes 2.3% less time to complete when 2 processors of a cluster share a station.

All experiments were run on a fully configured version of Hurricane with all servers running, but no other applications were running at the time. We have spent considerable efforts to optimize the memory management for improved performance in general, and maximum concurrency in particular. In contrast, the rest of the kernel has not been optimized and is rather crude; for example, only a single lock is available in the kernel to control concurrency (effectively serializing kernel operations). To reduce caching effects, we disabled the caching of all data within the kernel except for stack data; instructions were cachable.

5.1 Basic Operations

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

The cost to handle a read-page-fault increases from 128 microseconds within a cluster to 243 microseconds across clusters. The additional overhead in the latter case is due to the extra directory lookup needed to locate an existing copy of the page (because it is in another cluster), and the overhead of replicating a page descriptor.

The cost of a 32 byte send-response message transaction between two user-level processes increases from 328 microseconds within a cluster to 403 microseconds across clusters. Here the extra overhead is due to two RPCs 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 ran several tests with different cluster sizes:[5]

Config-1:	1 cluster of 16 processors
Config-2:	2 clusters of 8 processors
Config-4:	4 clusters of 4 processors
Config-8:	8 clusters of 2 processors
Config-16:	16 clusters of 1 processor

In these tests, p processes are run on *different* processors, where p is varied from 1 to 16 to adjust the load on the system. Figure 1 shows the behavior of the system as a function of the load for three basic system functions, namely page fault handling, kernel call handling and message passing. In the page fault test (Figure 1.a), the page faults of each process are to pages in different regions, so that the faults on different processors are independent of each other, except for the fact that they are all in the same address space. Moreover, the pages are found in a local page cache, so no I/O is required; i.e. the page only needs to be mapped in. In the kernel call test (Figure 1.c), a user-level process sends a message to a local process server (running in the kernel address space) to request the id of its creator. Finally, in the message passing test (Figure 1.d), a user-level sender process sends a 32 byte message to a user-level receiver process running on a different processor; the message is received by the receiver and a 32 byte response is sent back to the sender.

In these tests, the p processes are assigned to separate processors, but in a way that minimizes the number of clusters spanned. That is, the processes are first added to one cluster until each processor in the cluster has a process, before adding them to the next cluster. This corresponds to a scheduling policy that attempts to assign related processes to the same cluster whenever reasonable. Figure 1 shows that for all three 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. Because *Config-1* has only one cluster, the response time is maximized at 16 processors with a response time of approximately 1500 microseconds.

[5] 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.

For the other configurations, however, the *average* response time decreases sharply when p is further increased by one, because the last process added runs in a cluster by itself and therefore has a low response time, pulling down the average. As p is further increased, the average response time will begin to increase, until p is a multiple of the cluster size again. For this reason, we find peaks at $p = 8$ and $p = 16$ in the 2 cluster configuration, and we find peaks at 4, 8, 12, and 16 in the 4 cluster configuration. Moreover, we note that at the peaks, the response time is the same; that is, at equal levels of cluster loading, we observe the same response time, indicating that the speedup is linear in the number of clusters regardless of size (as far as lock contention for independent operations is concerned).

The figure shows that all three tests behave in the same way; this is because the overhead for lock contention dominates in each case. Although the page-fault test primarily exercises the memory management subsystem which has been optimized for increased concurrency, it still behaves similar to the less optimized components of the system (although a higher load is needed before lock contention takes effect).[6] The figure also clearly shows that by increasing the number of clusters (and consequently decreasing cluster size) additional concurrency made available, substantially reducing response time when locks are highly contended. Also note that when there is not much contention, the response time gets marginally better as we go to smaller clusters because of the higher degree of locality and the smaller data structures. (See, for example, the case where $p = 1$.)

Figure 2 shows the results of the same page-fault and message passing test, but where the p processes are spread evenly across the clusters. In this case, the response times stay constant as p is increased until p is equal to the number of clusters, because no cross-cluster interaction is necessary at this point. But when more processors are added, the response time begins to increase (because of lock contention).

The above tests show that for independent operations the response time and concurrency improve as the cluster size is reduced. The next set of tests consider operations that interact with one another, causing a significant amount of cross-cluster interaction. Figure 3.a shows the results of a page fault test, where one process first initializes a number of pages, after which all participating p processes start to access the same pages, causing the processes to fault. In these tests, processes are again assigned to the same cluster until each processor in the cluster is assigned a pro-

[6] In the first test, contention is caused by a common address space descriptor that must be accessed for all processes.

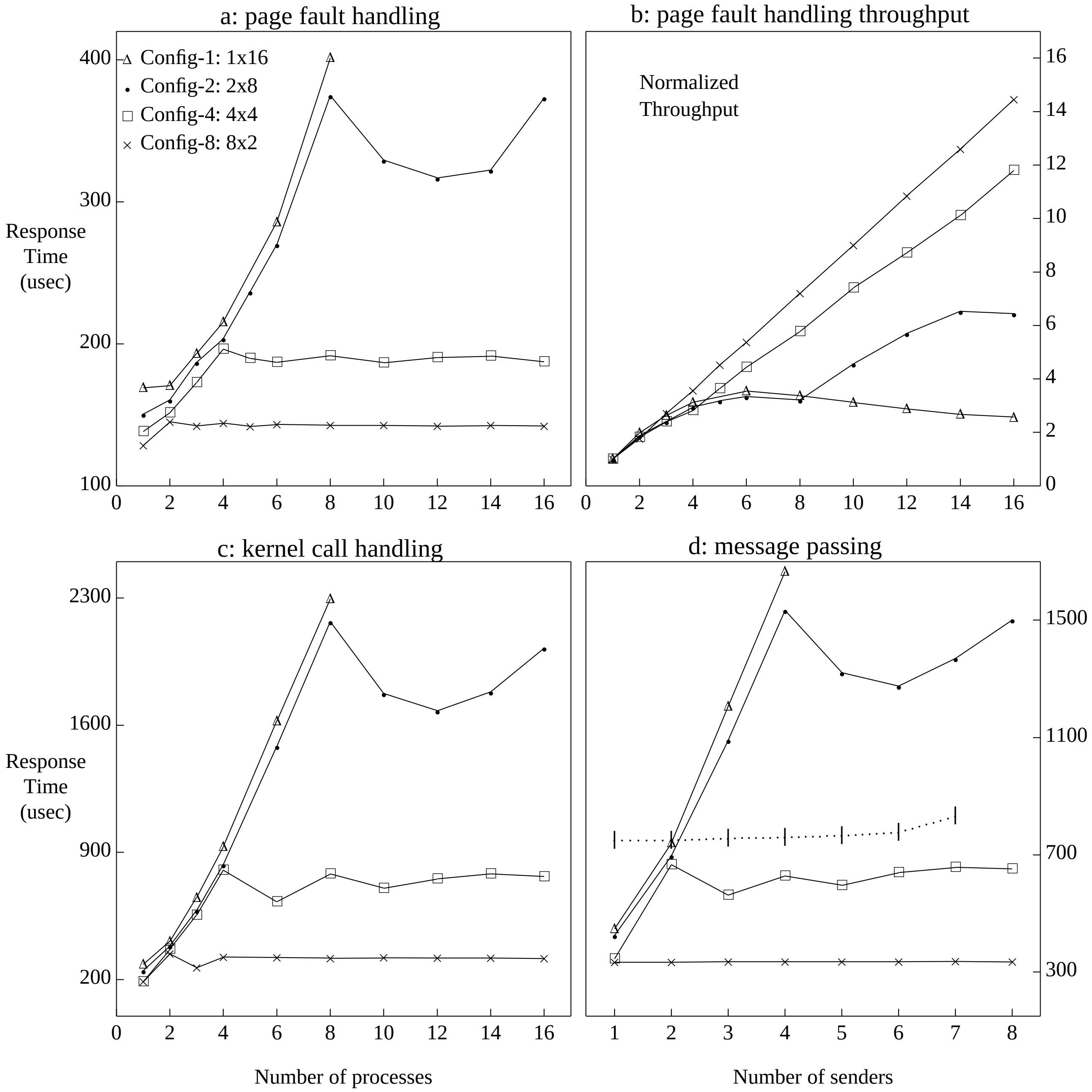

Figure 1: The response times for page fault handling, kernel request servicing, and message passing for different cluster sizes.

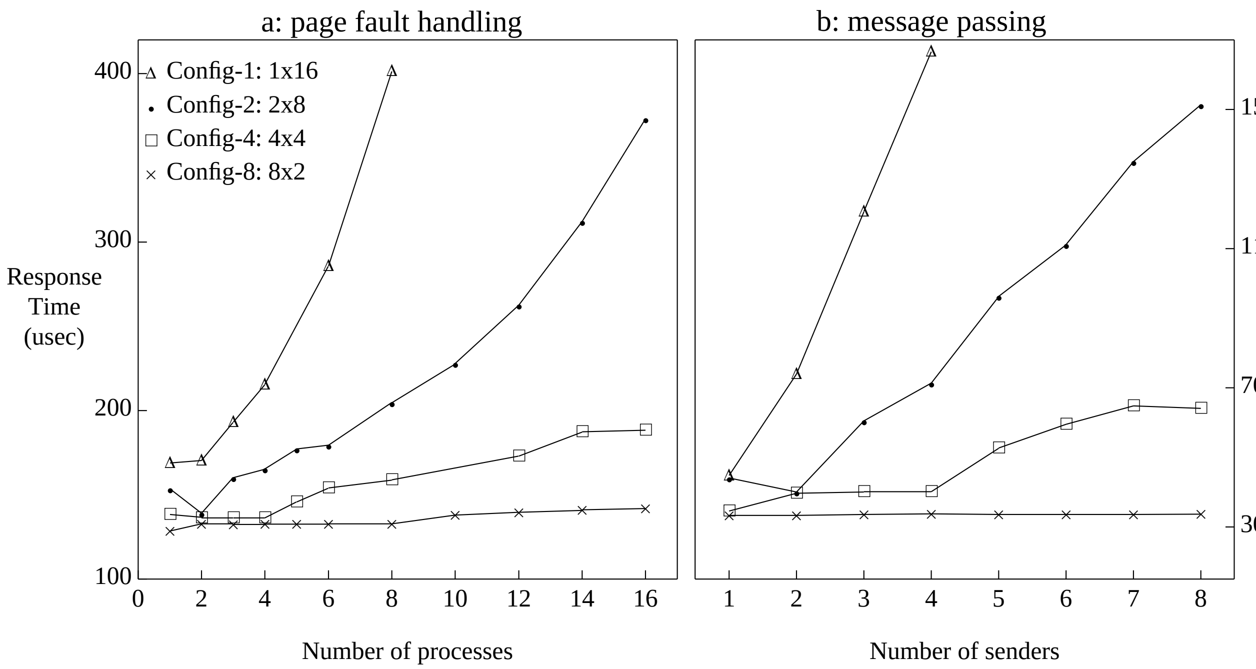

Figure 2: The response times for page fault handling, and message passing. These are the same tests as in the previous figure, except that the processes here are spread as evenly across the clusters as possible.

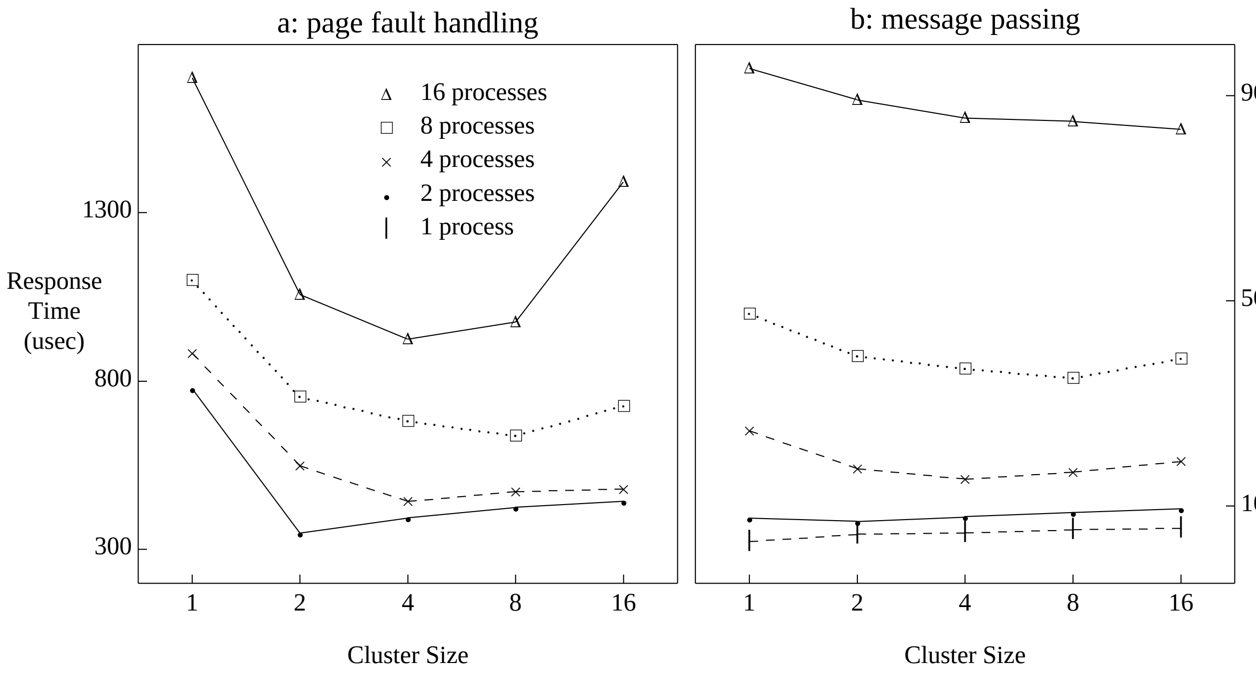

Figure 3: The response times for (a) page fault handling, and (b) message passing requiring cross-cluster communication, for varying cluster sizes. The different curves represent the tests performed with different numbers of processes.

cess, before starting to assign processes to the next cluster. The figure shows that the response times to handle a page fault for cluster sizes of 1, 2, 4, 8 and 16 processors. When p is 2, 4 or 8, the ideal cluster size is 2, 4 and 8, respectively. In each case, the efficiency of staying within a cluster outweighs the overhead of cross-cluster communication. However, when p is 16, a cluster size of 4 yields the best response time; in this case, the contention within the cluster dominates at large cluster sizes, so the increased overhead for cross-cluster communication is amortized by the increase in concurrency available when smaller cluster sizes are used.

Figure 3.b shows the response time for a message passing test, where one process iteratively receives a message from p processes after which it replies to each (thus implementing a barrier of sorts). The behavior of this test is similar to the behavior of the page-fault test. The only difference occurs when $p = 16$. In the message passing test, a cluster size of 16 is the optimal, because there is little lock contention, in contrast to the page fault test where lock contention is significant.

6 Concluding Remarks

We have introduced the concept of clustering as a way to structure operating systems for scalability, and described how we applied this structuring technique in our experimental operating system. We presented performance results from our prototype 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.

The system we have designed and implemented can easily be adapted to different machine configurations and sizes, by changing the cluster size. Moreover, our design meets a number of requirements necessary for scalability. First, we showed that independent operations directed to different clusters can be serviced completely in parallel; there are no central servers, or system-wide locks. The system can therefore preserve the parallelism afforded by the applications.

Second, the number of queues and the number of service points in the system increases with the size of the system. The overhead for each independent operation is therefore independent of the size of the system.

Finally, the system preserves the locality of the applications. Service requests are directed to local service points, and state is distributed and replicated in order to localize access whenever possible. With only one exception, the data structures of only those clusters directly involved in a service request are accessed when servicing the request. The exception involves lookup tables accessed by hash functions. These tables are used to locate objects (such as process descriptors, or cached file pages), can be accessed concurrently, and span the system to distribute the load.

Our work on clustering and Hurricane is still in its initial stages. We intend to pursue further experiments to determine how cluster size affects performance, and how the workload and the attributes of the hardware affect the ideal cluster size. We also wish to study whether each service (i.e., kernel, memory management, file service, etc.) is best served by the same cluster size, and whether one should entertain the possibility of having non-uniform cluster sizes, where the larger clusters are used to run applications with a higher degree of parallelism. In the long term, we intend to study policy issues for the various components of the system, including paging, scheduling and I/O subsystems.

Acknowledgements

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, Ben Gamsa, Jan Medved, Fernando Nasser, and Umakanthan Shunmuganathan.

References

[1] 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.

[2] 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.

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

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

[5] 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.

[6] H. Burkhardt. *KSR1 computer system.* Comp.arch Netnews article, Kendall Square Research, February 1992.

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

[14] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance.* Prentice Hall, 1984.

[15] 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.

[16] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, March 1991.

[17] G.F. Pister, W.C. Brantley, and George D.A. The IBM parallel research processor prototype. In *Proc. Intl. Conf. on Parallel Processing*, pages 764–769, 1985.

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

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