HFS: A Flexible File System for large-scale Multiprocessors

Orran Krieger and Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada, M5S 1A4
phone: (416) 978-5036, email: okrieg@eecg.toronto.edu

Abstract

The Hurricane File System (HFS) is a new file system being developed for large-scale shared memory multiprocessors with distributed disks. The main goal of this file system is scalability; that is, the file system is designed to handle demands that are expected to grow linearly with the number of processors in the system. To achieve this goal, HFS is designed using a new structuring technique called Hierarchical Clustering. HFS is also designed to be flexible in supporting a variety of policies for managing file data and for managing file system state. This flexibility is necessary to support in a scalable fashion the diverse workloads we expect for a multiprocessor file system.

1 Introduction

The Hurricane File System (HFS) is a new file system being developed for large-scale shared memory multiprocessors. In this paper the goals and basic architecture of this file system are introduced. The main goal of this file system is scalability; we expect the file system load to grow linearly with the number of processors in the system. A secondary goal is to be flexible in supporting a variety of policies for managing file data and for managing the internal file system state.

Many current large scale multiprocessors achieve their scalability by using segmented architectures [5, 19, 29]. As new segments are added, there is an increase in: 1) the number of processors, 2) the network bandwidth, and 3) the number of memory banks and hence the memory bandwidth. The memory banks are distributed across the machine so that the cost for a processor to access nearby memory is less than the cost of accessing more distant memory. In these systems, the application and operating system must cooperate to maximize the locality of a processor's memory references in order to minimize the access time and network bandwidth required by that processor.

If a system is to scale in its I/O capabilities, then the number of disks should also increase with the size of the system and these disks must also be distributed across the different segments [8, 13, 22]. As with memory, distributing the disks across the system has the advantage that some disks can be made more local to a processor, reducing the cost of accessing those disks for some processors. Even if the time to transfer a disk block across the interconnection backplane is insignificant compared to the cost of getting the block from disk, the interconnection backplane will become contended if a large number of disks are concurrently transferring data over the backplane [12]. Hence, it is important that most of a processor's I/O requests be directed to nearby devices. This rules out simple disk striping as a means for distributing file data across the disks, where blocks of a file are uniformly striped across all disks in the system.

Designing a file system for a large scale multiprocessor with distributed disks is challenging for a number of reasons. First, a file system has non-negligible cpu and I/O requirements (*e.g.*, for file meta-data) of its own and we expect the demands on the file system to grow linearly with the number of processors. This means that the file system for a multiprocessor is itself a complicated parallel program whose processing and I/O requirements must be spread across the machine.

Second, while having many disks results in a large aggregate bandwidth to the disks, the file system must aid applications to exploit this bandwidth. To do this, the file system must spread data across the disks in a way that allows accesses to be serviced in parallel. Also, we believe that many applications will be able to exploit this bandwidth only if the file system effectively employs latency hiding techniques (such as pre-fetching and caching data).

Third, the file system must try to manage locality so that a processor's I/O requests are primarily directed to nearby devices. In practice (especially when an application accesses persistent data) this may be difficult to achieve in an application independent manner. That is, we believe that to manage locality the file system must invoke policies that take advantage of applications specific knowledge of how files are to be accessed.

Finally, since large-scale multiprocessors have only recently become available, application I/O requirements for this hardware base are not yet well understood and the application-level I/O interfaces are not yet mature. Hence, the file system for such a machine must be flexible so that it can be extended to support new I/O interfaces and so that it can adapt to applications with different types of I/O requirements. Even in the most recent draft language specification for High Performance Fortran [9], no consensus was achieved on a set of parallel I/O extensions to the language.

In this paper the architecture of the HURRICANE File System is described. Section 2 describes our view of scalability and a new structuring technique for operating systems that aids in achieving scalability. Section 3 describes the architecture of the HURRICANE File System based on this structuring technique. Section 4 describes a number of different policies for improving application I/O performance, and how this file system invokes these policies. Section 5 describes the organization of file data and meta-data on the disks. We conclude with a description of the current state of our implementation effort.

2 Scalability and Hierarchical Clustering

It is difficult to design software that is scalable. Existing operating systems have typically been scaled to accommodate a large number of processors in an ad hoc manner, by repeatedly identifying and then removing the most contended bottlenecks [2, 18]. 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 systems that 1) are fine-tuned for a specific architecture/workload and hence not easily portable to other environments with respect to scalability; 2) are not scalable in a generic sense, but only until the next bottleneck is reached; and 3) have a large number of locks that need to be held for common operations, with correspondingly large overhead.

We have developed a more structured way of designing scalable operating systems called Hierarchical Clustering. This structuring technique was developed from a set of guidelines for designing scalable demand driven systems [28] (of which operating systems are an example).

2.1 Scalability Guidelines

For a demand-driven system to be scalable, it must satisfy each of the following guidelines:

Preserving parallelism: *A demand driven system must preserve the parallelism afforded by the applications.* The potential parallelism in the file system comes from application demand and from the distribution of data on the disks. Hence, if several threads of a parallel application (or of simultaneously executing but independent applications) request independent services in parallel, then they should be serviced in parallel. This demand for parallel service can only be met if the number of service centers increases with the size of the system, and if the concurrency available in the data structures of the file system also grows with the size of the system. This also means that the file system should enact policies that balance the I/O demand across the system disks.

Bounded overhead: *The overhead for each independent system service request must be bounded by a constant* [3]. If the overhead of each service call increases with the number of processors, then 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 must not be located by linear searches if the queue lengths or search times increase with the size of the system.

The principle of bounded overhead also applies to the space costs of the internal data structures of the system. While the data structures must grow at a rate proportional to the physical resources of the hardware [28, 1], the principle of bounded space cost restricts growth to be no more than linear. For example, the meta data maintained by the file system cannot grow more than linearly with the disk capacity and the in-core state maintained by the file system cannot grow more than linearly with the amount of physical memory.

Preserving locality: *A demand driven system must preserve the locality of the applications.* It is important to consider locality in large-scale systems in order to reduce the average access time (since remote resources are more expensive to access) and in order to minimize the use of interconnection bandwidth. Locality can be increased by; a) properly choosing and placing data structures internal to the system, b) directing requests from the application to nearby service points, and c) enacting policies that increase locality in the applications' disk accesses and system requests. For example, per-application data should be stored near the processors where the application is running, and meta data should lie on disks close to the files that contain the data.

2.2 Hierarchical Clustering

The HURRICANE File System is part of a larger effort to investigate operating system design for large scale multiprocessors. Our operating system structure is called Hierarchical Clustering. The goal of Hierarchical Clustering is to support large-scale applications without penalizing the performance of small-scale applications [27].

The basic unit of structuring within Hierarchical Clustering is the *cluster*, which provides the full functionality of a tightly coupled small-scale symmetric multiprocessor operating system. On a large system, multiple clusters are instantiated such that each cluster manages a unique group of "neighboring" processors, where neighboring implies that memory accesses within a cluster are generally less expensive than accesses to another cluster. All system services are replicated to each cluster so that independent requests can be handled locally. Clusters cooperate and communicate in a loosely coupled fashion to give applications an integrated and consistent view of the system. For very large systems, extra levels in the logical hierarchy can be created (*i.e.*, *super clusters*, *super super clusters*, etc.) where each level in the clustering hierarchy involves looser coupling than the levels below it. Requests that are not independent are resolved by directing requests to servers (and data structures) located at the higher levels in the clustering hierarchy.

Hierarchical Clustering incorporates structuring principles from both tightly-coupled and distributed systems, and attempts to exploit the advantages of both. Small-scale multiprocessor operating systems achieve good performance through tightly coupled sharing and fine-grain communication; however, it is difficult to make these systems scale to handle a large number of processors. Distributed operating systems appear to scale well by replicating services to the different sites in the distributed system; however, the high communication costs in these environments dictates course-grain sharing. We believe that on large-scale multiprocessors, the relatively low cost of accessing remote memory should encourage fine-grain sharing between small groups of processors, while for the system as a whole replicating services seems imperative for locality and scalability.

If all the processes of an application can fit on a single cluster, and if all I/O can be directed to local devices, then Hierarchical Clustering results in an operating system that for that application has the same performance as a tightly coupled small-scale multiprocessor operating system. Hierarchical Clustering also provides a good structure for addressing the design guidelines presented in the previous section:

Preserving parallelism: Process requests are always directed to their 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 independent, they can be serviced by the clusters in parallel. Similarly, independent requests by different applications are serviced in parallel.

Bounded overhead: The number of processors in a cluster is fixed, therefore, fixed sized data structures can be used within a cluster to bound the overhead of independent requests. Inter-cluster communication is required only when requests are not independent.

Preserving locality: The hierarchical structure of Hierarchical Clustering lends itself to managing locality. First, requests are always directed to the local cluster (and if this fails the local super cluster etc.). Second, data structures needed to service independent requests are located on the local cluster, hence they are located near the requesting process. Third, it provides a natural framework for invoking policies that increase the locality of the applications. For example, the scheduler attempts to run the processes of a single application on the same cluster (and if that fails on the same super cluster ...), place memory pages in the same cluster as the processors accessing them, and direct file I/O to devices on that cluster.

Hierarchical Clustering allows the performance of the operating system to degrade gracefully when applications make requests that are not independent. These requests are handled at the lowest possible layer of the clustering hierarchy (*i.e.*, if possible a request is handled entirely within a cluster, and if not possible, then at the super cluster, etc.).[1] Therefore, the cost of non-independent requests depends on the degree of sharing (or contention) of the applications making those requests and not on the size of the multiprocessor.

3 The File System Architecture

The HURRICANE File System is being developed as part of the HURRICANE operating system [26]. In this section we describe the characteristics of the HURRICANE operating system that have influenced the design of the HURRICANE File System, describe the servers that make up the file system, and describe the state maintained by the different servers.

3.1 The HURRICANE operating system

There are a number of characteristics of HURRICANE that have had an impact on the design of the file system. First,

[1] For example, if a page is replicated to a number of clusters in a particular super cluster, but is not accessed outside of that super cluster, then the data structure describing that page is local to a memory module in that super cluster.

HURRICANE is structured according to Hierarchical Clustering. Therefore, the file system on each cluster provides the functionality of a full small scale file system. In order to minimize cross-cluster communication, file system state is cached whenever possible on the clusters where it is being accessed, as it would be in a distributed system. The per-cluster file system servers communicate with each other to keep this state consistent and to distribute disk I/O across the different clusters.

Second, HURRICANE is a micro-kernel operating system, where most of the operating system services are provided by user-level servers. The micro-kernel provides for basic interprocess communication and memory management. As described in the next section, most of the file system exists outside of the HURRICANE micro-kernel.

Third, HURRICANE is a single level store operating system that supports mapped file I/O. With mapped file I/O, the application can map regions of a file into its address space and access the file by referencing memory in the mapped regions. This has a number of advantages. For example, it allows all of main memory to be used as a cache of the file system. Also, mapped file I/O results in less overhead for accessing data in the file cache than for example Unix I/O[17]. However, mapped file I/O presents a challenge to the file system, since it must handle demands that are implicit due to accesses to memory rather than explicit due to calls to the file system. Moreover, since the memory manager and not the file system controls the file cache, the two must cooperate to manage the data being cached.

Finally, HURRICANE supports a new facility, called local server invocations (LSI) that; 1) allows fast, local, cross-address space invocation of server code and data, and 2) on demand results in new workers being created in the server address space. Since LSI requests are very fast[2], all file system state that can be accessed by multiple applications is maintained in file system servers (rather than cached in the application address space). The speed of LSI requests also allows the file system to be partitioned into different address spaces without a major impact on performance. The LSI facility also impacts the file system architecture in that it simplifies deadlock avoidance. Since with LSI new worker processes are created on demand, different servers can make LSI requests to each other without fear of running out of workers that can satisfy the requests.

3.2 The per-cluster file system

The structure of the HURRICANE File System is illustrated in Figure 1. There are three user level system servers in the file system, namely: the *Name Server*, *Open File Server* (OFS), and *Block File Server* (BFS).

[2]With our current implementation LSI requests are close to an order of magnitude faster than the more traditional Send/Receive/Reply interface also supported by HURRICANE.

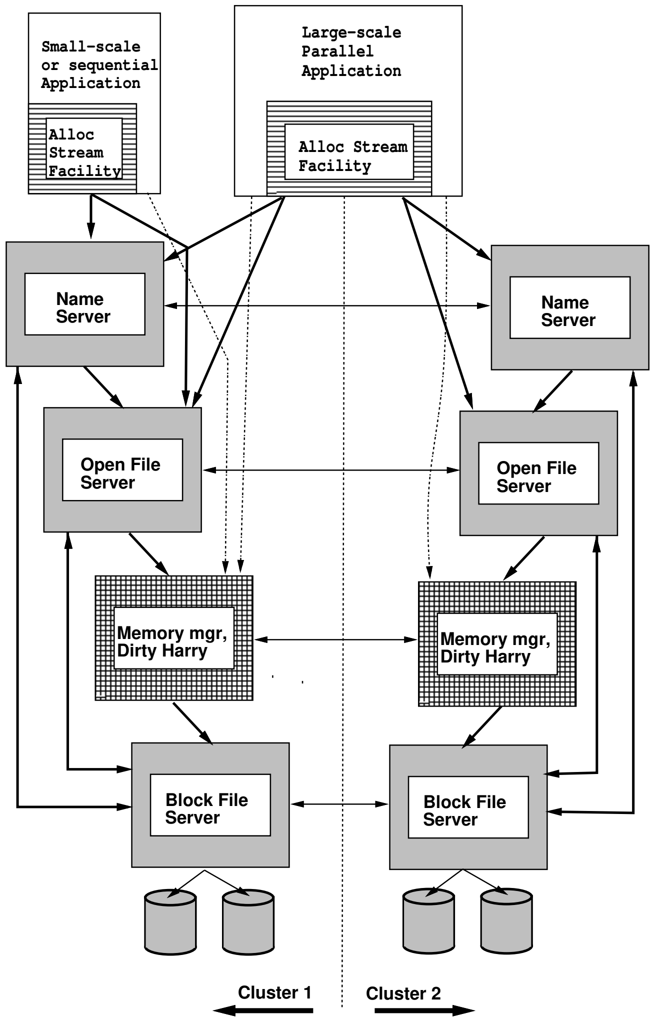

Figure 1: *The HURRICANE File System. The Name Server, Open File Server and Block File Server are user level file system servers. The Alloc Stream facility is a file system library in the application address space. The Memory Manager and Dirty Harry Layer are part of the HURRICANE micro-kernel. Dark solid lines indicate local server invocations (LSI). Light solid lines indicate cross cluster IPC operations. Dotted lines indicate page faults.*

The name server manages the HURRICANE name space and is responsible for authenticating requests to open files. The OFS maintains the file system state kept for each open file, and is responsible for authenticating requests to map files into the application address space. The BFS controls the system disks, it is responsible for determining to which disk an operation is destined, and directing the operation to the corresponding device driver. *Dirty Harry* (DH) is the only kernel level file system server. It collects dirty pages from the memory manager and makes requests to the BFS to write the pages to disk.

The *Alloc Stream Facility* [16, 17] (ASF) is a user level library that maps files into the application's address space and translates `read` and `write` operations into accesses to the mapped regions. It supports a variety of I/O interfaces including Unix I/O (*i.e.*, read/write), Stdio, and the *Alloc Stream Interface* [17].

The file system servers (*i.e.*, name server, OFS, and BFS) each maintain different state. The name server maintains logical directory state (*e.g.*, directory size and access permissions) and directory contents (*i.e.*, the mapping between file names and the identifiers used by the BFS). The OFS maintain logical file information (*e.g.*, file length, access permissions, and modification time) and per-open instance state (*e.g.*, file offset). The BFS maintains for each file the block map, that is, the mapping between logical blocks of a file and the corresponding disks blocks. Since the state maintained by the different servers is independent, there is no need for the different servers within a cluster to communicate in order to keep the state consistent.

A simple example best illustrates the interactions between the ASF library and the servers on each cluster. When an application opens a file, the open is sent by the library to the name server, the name server translates the character string name of the file into its *token*; a number that uniquely and permanently identifies the file to the system, checks if the application is allowed access to the file and forwards the request to the BFS. The BFS uses the token to locate the logical file state for that file and forwards this information to the OFS. Finally, the OFS records the logical state and the file token and returns to the application a *file handle*; a capability used on subsequent accesses to the file.

When the application makes a read or write request to an open file, the library sends a request to the OFS to map the file into the application address space. The OFS checks that the access is valid, translates the file handle into the file *token* and sends the request to the memory manager. Once a file (or portion thereof) has been mapped into the application address space, the library services the application I/O request by copying data to or from the mapped region.[3] On the first access to a page in the region, the process will page fault. If the data is available in the file cache, the memory manager, as part of handling the page fault, modifies the application page tables to reference the physical page and returns control to the faulting process. Otherwise, the fault is translated by the memory manager into a `block-read` operation to the BFS. In this case, the BFS determines which disk contains the requested block and initiates the operation to the corresponding device driver (possibly by communicating with a BFS on another cluster). Control is returned to the faulting process when the I/O operation to the disk has completed. If a mapped page is modified by the application, then Dirty Harry will eventually send a `block-write` request to the BFS (at the latest when the memory manager wishes to reassign that physical memory).

While we are still experimenting with this structure, the division of the file system into four layers (application, OFS, name server and BFS) appears to have a number of advantages. The ASF library contains much of the code that is more commonly implemented by system servers. It is easier to modify and extend this library to handle new application demands or new I/O interfaces. Also, having the code in the application library reduces the demand on possibly contended system servers. The fast LSI facility allows us to maintain state in the OFS that we would otherwise have to cache in the application level (as is done by Mach 3.0 [7]). This greatly simplifies our code for handling application errors and is probably also more robust. The name server allows locally accessed portions of the name space to be cached on each cluster. Finally, the portion of the file system that cannot be paged out is minimized (*i.e.*, the BFS layer).

3.3 The cross-cluster file system

On a large system, an instance of each file system server described in the previous section exists on each cluster. A number of decisions must be made for each type of state maintained by these replicated servers:

- How to locate state the first time it is accessed. While broadcast is disallowed by our scalability guidelines, there are a variety of other alternatives available. For example, each level of the clustering hierarchy can be consulted in turn (*e.g.*, if the state is not available in the cluster, then the super cluster is searched, etc.). Another alternative is to assign (on creation) a *home* cluster to each file that is identified by the file token. The BFS on the *home* cluster identifies the set of servers (if any) that cache the requested state as well as the on-disk location of this state.

[3] When necessary, the ASF makes requests to the OFS to modify the file length, modification time, and file offset.

- The policy for maintaining the state. For example, the state could be statically assigned to a fixed location, migrated to the locations where it is accessed, or replicated to the locations where it is accessed. Each of these alternatives has advantages depending on the access pattern. For example, the block map of a file contained entirely on the disks of a single cluster should be cached only by the BFS on that cluster, since all read and write operations must be sent to that BFS. On the other hand, if a file is distributed across disks on a number of clusters, then to avoid one BFS from becoming a bottleneck, it may be worthwhile to cache the block map at the clusters where the file is accessed (as is done by CFS [22]).

- If state is replicated, then it must be kept consistent. Different protocols could be used for this purpose, such as, invalidate or update. Similarly, message passing or shared memory can be used for communication. Also, updates or invalidates can be disseminated by logically organizing the servers caching the state, for example, into a linked list, a balanced binary tree, or a tree as defined by the hierarchy of Hierarchical Clustering.

As implied by this discussion, the possible design space is huge. Moreover, we believe that no particular set of decisions will always yield the best performance for all file types. Therefore, we plan to implement a variety of different policies for managing each type of file system state, so that a particular policy can be chosen at runtime on a file by file basis. This is discussed in more detail in the next section.

4 Policy issues and storage objects

For a file system to be scalable, it must invoke policies that; 1) minimize the contention on the disks and interconnection backplane and, 2) aid applications to exploit the bandwidth of the distributed disks. In this section we first describe some possible policies and then present our strategy for implementing them.

4.1 Policy issues

There are two key areas where policies can have a large impact on performance:

File block distribution: There have been numerous proposals for policies that distribute file data across multiple disks. Several systems stripe file blocks across all disks in the system [8, 21, 22]. Crockett suggests six different distributions according to how processes of parallel applications generate I/O requests [6]. Jensen analyzed a series of distribution functions, and found that the performance of these functions depends on the total I/O load, the access pattern of the applications, the concurrency in the applications, and the number of files being accessed [12].[4] Also, several file block distribution policies attempt to maximize locality. In particular, paging I/O can be directed to disks near the memory being paged out, and, if a file is to be repeatedly accessed, it (or individual file blocks) can be migrated to disks close to the requesting process. Finally, if I/O requests to a file are primarily reads, then replicating the file can be used both to obtain better performance [4, 20, 24] and to manage locality (since the more local copy of a disk block can be used to satisfy a read request).

Latency hiding: Two strategies that can be used to reduce the latency of I/O are caching data in memory and prefetching data that will soon be required from disk. For caching, different policies can be used to determine which blocks should be replaced in the file cache (*e.g.*, LRU, MRU) and to determine when dirty blocks should be written to disk [14]. For prefetching, Kotz and Ellis have proposed algorithms that track file accesses in order to determine candidate blocks for prefetching [15]. The ELFS file system prefetches by taking advantage of application specific knowledge of both the logical file structure (*e.g.*, a two dimensional matrix) and the application access pattern [10, 11].

To choose the best policy at runtime, the system can take into account: 1) the nature of the file, 2) the nature of the applications accessing that file, 3) the observed use of the data (*i.e.*, statistics gathered during run time), and 4) contention for shared resources (*e.g.*, the load on the disks due to other applications).

4.2 Storage objects

We believe that it is important that a large scale multiprocessor file system be flexible in supporting a multitude of policies for managing file data and file system state. Because of the immaturity of the current multiprocessor applications, it is also important that a file system be easily extendible to support new policies.

To achieve these goals, HFS uses an object oriented approach. Logically, a file, or more generally a *storage object*, encapsulates the meta-data, the data, and various methods (*i.e.*, code) for accessing that storage object. Storage objects can have diverse organizations of meta data (for example, to allow different organizations

[4] This same study also suggests the possibility of custom mappings for application-specific access patterns.

of data on disk) and methods that implement object specific policies.

Since each level of the file system manages different state, and since different policies are appropriate at each level, every storage object is made up of three components, one for the BFS, one for the OFS or (in the case of a directory) the name server, and one for the application library (*i.e.*, ASF). Each storage object component has its own type field (and other meta data). When a storage object is first accessed at a particular level of the file system, the corresponding type field is used to decide what code should be executed for operations on that storage object (and how to interpret the corresponding meta data).

For code reuse, storage objects can be derived from other storage objects. For example, at the BFS layer, storage objects that involve different policies for prefetching data can be derived from a single storage object that involves a particular block distribution algorithm. As another example, at the application layer, storage objects that compress and decompress data can be derived from a single simple storage object.[5]

To allow for even greater flexibility, storage objects can refer to other storage objects in their meta data. For example, the BFS level supports *primitive* storage objects, where both the file data and meta data exist on a single disk, and *complex* storage objects, where the file data and meta data can be distributed across a number of disks. The meta data of a complex storage object indicates which local storage objects contain a particular file block, and the local storage objects indicate the actual position on disk of these file blocks. Having complex storage objects constructed from primitive storage objects has a number of advantages. First, it allows the disk block position to be determined at the site controlling the disk. This allows, for example, the disk head positions to determine where a file block should be written next. Second, it allows the data on each disk to be re-organized without requiring any central communication. Finally, it allows for error recovery in the case of a system crash to be done largely independently on each disk.

It is possible to extend the hierarchy further. Complex storage objects can be constructed from other complex storage objects. For example, a replicated file where each replica is distributed across a number of disks would be implemented as a replicated storage object whose meta data refers to two distributed storage objects. As another example, if a distributed file spans a super super cluster, then it is possible to build the distributed storage object using a number of other distributed storage objects, each of which spans a single super cluster. This allows changes to the storage object on each super cluster to be made local to that super cluster. As a final example, Jensen has proposed a number of different ways of distributing data across disks [12], each of which has advantages depending on how the data is being accessed. If read operations are dominant, then replicated storage objects could be constructed from multiple classes of distributed storage objects, each optimized for a particular type of read access.

A storage object can implement multiple policies and keep statistics (possibly stored in the meta data) to determine which policy is most effective for that data. A similar approach is used by Staelin *et al* for reorganizing data on disk [25], and by Kotz to prefetch data from disk [15].

Finally, an application can derive a new storage object from a file system storage object at run time. The ELFS file system [10, 11] uses such an approach both to invoke application specific policies and to close the semantic gap between the application programer and the file system by deriving a new storage object that matches the application view of the data in the file.

5 On disk data organization

The main goal of our research is to study performance issues pertaining to large scale multiprocessors. We ignore fault tolerance at the level of disk failures because using RAID like approaches (*i.e.*, parity or mirroring disks) it is possible to design a system that can have any required degree of redundancy. However, we cannot ignore the case of system crashes leaving the system in a corrupt state. We have not optimized per-disk file system performance because it is not crucial from a functional point to view. However, for a proper performance evaluation, the per-disk file system performance cannot be totally neglected. For example, if no attempt is made to exploit locality on a single disk then making use of multiple disks appears to be more advantageous than it would otherwise.

To address both the file system recovery and the performance issues we decided to base our per-disk file system on the idea of log-structured file systems [23]. A log-structured file system allows for fast crash recovery on a single disk. Moreover, as long as the files are read and written in the same way, a log-structured file system results in good performance.

Our file system differs from other log structured file systems in two important respects. First, it allows for fast crash recovery not only for a single disk, but also for files that span multiple disks. In brief, this is accomplished by having the per-disk file system maintain mul-

[5] Having compression and decompression performed at the application level is useful, since the pages in the file cache are in a compressed format. Having the file type and hence the compression algorithm associated with the file allows the library to hide from the application that compression is being done. This is important because standard utilities (*e.g.*, a compiler) can be executed on the file without knowledge of the compression algorithm.

tiple versions of each *primitive* storage object and having the *complex* storage objects refer to particular versions of the primitive storage objects.

Second, our file system allows storage objects to override the standard logging policy in an object specific fashion. While a log-structured file system provides close to optimal performance for files read and written in their entirety, previous log structured file systems implementations do not allow the default policy to be overridden. We do not feel this allows sufficient flexibility for exploiting information that may be available to some applications. For example, data base applications may wish to control explicitly the organization of blocks on the disk.

6 Concluding remarks

In this paper we have outlined the design of a new file system for a large-scale shared memory multiprocessor. The three key attributes of this design are: 1) a file system architecture based on Hierarchical Clustering, 2) a log-structured organization of data on disk, and 3) the use of objects for flexibility in supporting a variety of policies for managing both file data and the file system state.

Designing software for a new type of a machine with an unknown workload is an iterative process. We have implemented an initial version of the per-cluster HURRICANE File System. Our experiences with this initial implementation inspired many of the ideas described in this paper. We are currently in the process of extending this file system to handle multiple clusters and support a larger set of storage objects. The Alloc Stream Facility (*i.e.*, the application level library) is quite mature and has been ported to a number of different systems [17]. The greatest handicap we are facing with this research is the lack of availability of (public domain) parallel applications that do real I/O.

References

[1] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. "Generic Virtual Memory Management for Operating System Kernels". In *Proc. 12th ACM Symposium on Operating System Principles*, pages 123–136, Litchfield Park, Arizona, December 1989.

[2] 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.

[3] Amnon Barak and Yoram Kornatzky. Design principles of operating systems for large scale multicomputers. Computer Science RC 13220 (#59114), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, October 1987.

[4] D. Bitton and J. Gray. Disk shadowing. In *14th International Conference on Very Large Data Bases*, pages 331–338, 1988.

[5] Henry Burkhardt III, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendell Square Research, Boston, February 1992.

[6] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[7] Randall W. Dean and Francois Armand. Data Movement in Kernelized Systems. In *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 243–262, Seattle, Wa., April 1992.

[8] P. C. Dibble and M. L. Scott. The Parallel Interleaved File System: A solution to the multiprocessor I/O bottleneck. *IEEE Transactions on Parallel and Distributed Systems*, 1992. To appear.

[9] High Performance Fortran Forum. DRAFT High Performance Fortran Language Specification. Technical report, Rice University, 1992.

[10] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 177–179, 1991.

[11] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. Technical Report TR-91-14, Univ. of Virginia Computer Science Department, July 1991.

[12] David Wayne Jensen. *Disk I/O in High-Performance Computing Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[13] David Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. PhD thesis, Duke University, April 1991. Available as technical report CS-1991-016.

[14] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[15] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[16] O. Krieger, M. Stumm, and R. Unrau. Exploiting the Advantages of Mapped Files for Stream I/O. In *Winter USENIX*, pages 27–42, 1992.

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

[18] Alan Langerman, Joseph Boykin, and Susan LoVerso. A Highly-Parallelized Mach-based Vnode Filesystem. In *Winter USENIX*, 1990.

[19] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. "The Stanford Dash Multiprocessor". *Computer*, 25(3):63–79, March 1992.

[20] Raymond Lo and Norman Matloff. A probabilistic limit on the virtual size of replicated file systems. Technical report, Department of EE and CS, UC Davis, 1989.

[21] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[22] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[23] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Symposium on Operating System Principles*, 1990.

[24] John A. Solworth and Cyril U. Orji. Distorted mirrors. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 10–17, 1991.

[25] C. Staelin and H. Garcia-Molina. Smart Filesystems. In *Winter USENIX*, 1991.

[26] Michael Stumm, Ron Unrau, and Orran Krieger. "Designing a Scalable Operating System for Shared Memory Multiprocessors". In *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 285–303, Seattle, Wa., April 1992.

[27] Ron Unrau, Michael Stumm, and Orran Krieger. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. Technical report, Computer Systems Research Institute, University of Toronto, 93.

[28] 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.

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