HFS: A flexible file system for shared-memory multiprocessors

by

Orran Krieger

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
Computer Engineering Group
University of Toronto

Abstract

The HURRICANE File System (HFS) is designed for large-scale, shared-memory multiprocessors. Its architecture is based on the principle that a file system must support a wide variety of file structures, file system policies and I/O interfaces to maximize performance for a wide variety of applications. HFS uses a novel, object-oriented building-block approach to provide the flexibility needed to support this variety of file structures, policies, and I/O interfaces. File structures can be defined in HFS that optimize for sequential or random access, read-only, write-only or read/write access, sparse or dense data, large or small file sizes, and different degrees of application concurrency. Policies that can be defined on a per-file or per-open instance basis include locking policies, prefetching policies, compression/decompression policies and file cache management policies. In contrast, most existing file systems have been designed to support a single file structure and a small set of policies.

We have implemented large portions of HFS as part of the HURRICANE operating system running on the HECTOR shared-memory multiprocessor. We demonstrate that the flexibility of HFS comes with little processing or I/O overhead. Also, we show that HFS is able to deliver the full I/O bandwidth of the disks on our system to the applications.

Acknowledgements

First, I would like to thank Dr. Michael Stumm for his supervision and guidance throughout. His support was invaluable as I learned the ropes of experimental research, in particular, learning to temper wild inspiration with more practical experience and intuition.

The Hurricane File System is a small part of the HURRICANE/HECTOR project. I would like to thank all the people involved in this project for their support and for the incredibly synergetic environment that I am grateful to have been able to participate in. Special thanks must go to Ron Unrau and Ben Gamsa, without whose help this work would not have been possible. Ron designed and implemented most of the infrastructure on which I depended. Ben designed and implemented an entirely new IPC facility when the existing one proved to be insufficient.

Over the last 7 years the Computer Group has provided the stimulating environment so necessary for research. I would like to thank the many colleagues and friends, both inside and outside of the Computer Group, who have made these years of graduate school such a great experience.

My parents' support throughout has been enormous. They have at least as much emotional investment in this degree as I do.

Finally, I would like to thank the Natural Sciences and Engineering Council, and the University of Toronto for their financial support.

Contents

Chapter 1

Introduction

This dissertation describes a new file system, called the Hurricane File System (HFS), whose primary goal is to maximize I/O performance for a large set of parallel applications. There are several reasons why I/O in parallel computers has become an important problem. First, advances in processor architecture and VLSI technology have resulted in much faster processors, while corresponding improvements in secondary storage technology have been slower. Second, as application writers learn how to better parallelize the CPU-intensive parts of their applications, a growing number of even traditional supercomputer applications are becoming I/O-bound. Third, an increasing number of I/O bound applications are being developed for parallel supercomputers. For example, many of the *Grand Challenges* problems have extremely high I/O requirements [33]. As another example, parallel supercomputers are starting to be used extensively for business applications such as data bases. Fourth, because the increasing demands on parallel supercomputers requires them to be multi-programmed, the file systems must be able to handle paging and/or swapping demands, while minimizing the effect of this activity on application performance.

The HURRICANE File System (HFS) is designed for large-scale shared-memory multiprocessors with disks distributed across the multiprocessor. The three most important features of HFS are:

Flexibility: HFS can support a wide variety of file structures and file system policies.

Customizability: The structure of a file and the policies invoked by the file system can be controlled by the application.

Efficiency: The flexibility of HFS comes with little I/O and CPU overhead.

Section 1.1 describes the need for flexibility in a file system for parallel supercomputers. Section 1.2 argues why we feel it necessary for the file system to allow the application or compiler to determine file structure and file system policies. Section 1.3 describes the approach we use to achieve the required flexibility with low processing and I/O overhead. Finally, Section 1.4 describes the major contributions of our work and outlines the remainder of this dissertation.

1.1 The need for file system flexibility

A file system for a large scale multiprocessor must meet several challenging requirements. First, the file system must allow applications to exploit the aggregate bandwidth of the large number of disks attached to the system. This means that the file system must support multiple data distribution and latency-hiding policies, so that the policies employed when a file is being accessed can be chosen to match the access pattern. The data distribution policy should match the application access pattern to avoid concurrent requests for different data being directed to a single disk [27, 56, 135]. Similarly, the latency-hiding policy (e.g., prefetching and poststoring) should match the application access pattern to prevent the application from seeing the large latency of disk I/O [49, 69, 71, 103, 121] and to avoid unnecessary disk requests.

Second, the file system must 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), locality may be difficult to achieve in an application-independent manner. For example, without *a-priori* knowledge of which processes of an application will access what data, the file system cannot ensure that a file is structured so as to enable each process's requests to be satisfied from nearby disks.

Third, the file system must be scalable. A file system has non-negligible CPU and I/O requirements of its own (e.g., for file meta-data) and must be able to handle demands that grow linearly with the number of processors and disks. Hence, the file system of a large-scale multiprocessor is itself a complicated parallel program whose processing and I/O requirements must be balanced across the machine. This means that file system state must be cached at the sites where it is used, and the communication and synchronization strategies used to keep cached state consistent must adapt to application demands.

Fourth, the file system must be able to recover from component failures in a timely fashion. Because parallel supercomputers are made up of a large number of components, including processors, disks, memory banks and network segments, the failure of individual components can be expected to occur relatively frequently. This means that the file system must maintain redundant information to allow important files and file system state to be recovered in the event of a failure. The need for flexibility arises from the need to minimize the cost of maintaining redundancy by having the technique used (e.g., none, parity, replication) conform to both the reliability requirements of the data and the expected access pattern of the applications.

It is clear that to meet the above requirements a file system for a parallel supercomputer must have at least some flexibility. It must provide a variety of policies for data distribution, latency hiding and locality management. Also, it should support a variety of synchronization and communication strategies to deliver scalable performance to a large set of applications. Finally, it should support a variety of techniques for maintaining redundancy for fault tolerance.

An open question is how much flexibility is required. HFS has been designed with the goal of making it simple to extend the file system to support any file structure, file system policy, or I/O interface. Since large-scale multiprocessors have only recently become available, application I/O requirements for this hardware base are not yet well understood and even the I/O interfaces are not yet mature.[1] While we do not know how much of the flexibility of HFS will be necessary once the requirements of I/O intensive parallel applications are better understood, we believe that providing such flexibility is crucial today so that application developers are not constrained by file system limitations.

1.2 The need for customizability

We contend that the file system flexibility must be exported to the application level, and that the application, compiler, and file system must cooperate in order to be able to fully exploit the I/O capabilities of a parallel supercomputer.

When vector supercomputers first became available, a common misconception was that new compiler, operating system, and hardware technology would some day make it possible for *dusty deck* programs to exploit the full power of these machines. Eventually it was recognized that application programmers must modify their programs in order to be able to fully exploit the power of these machines [59]. Similarly, all parallel supercomputer vendors have recognized that they must support new programming languages (e.g., HPF) that allow applications to provide extra information to simplify the task of exploiting the processing power of their machines.

Harnessing the I/O capabilities of a parallel supercomputer is at least as difficult as harnessing the processing capability of such a machine. As with the latter, the former requires cooperation between application programmers, compilers, and the operating/file system. For example, the correct choice of data distribution, latency hiding, and locality management policy is often specific to the application and cannot be extracted (at reasonable overhead) from the I/O requests made by the application. Hence, we believe the choice of policies used by the file system can be greatly simplified if the application can identify its expected demands in advance.

Most researchers studying file system issues for parallel supercomputers have recognized the need for cooperation between the application and the file system [26, 34, 49, 68, 83, 103]. Our approach differs from others in that HFS gives the application the ability to explicitly customize the implementation of a file to conform to its specific requirements. We believe that this extra level of control is important, given the current lack of understanding of the requirements of parallel applications.

[1] Even the most recent draft language specification for High Performance Fortran (HPF) does not contain a set of parallel I/O extensions to Fortran [41].

1.3 The HFS file system

HFS uses an object-oriented building-block approach, where files are implemented by combining together a (possibly large) number of simple building blocks called *storage objects*. Each storage object defines a portion of a file's structure or implements a simple set of policies. For example, storage objects belonging to different classes[2] may: store file data on a single disk, distribute file data to other storage objects, replicate file data to other storage objects, store file data with redundancy (for fault tolerance), prefetch file data into main memory, enforce security, manage locks, or interact with the memory manager to manage the cache of file data. A major part of our research has been to develop interfaces that allow storage objects to be conveniently combined together in useful ways and minimize the overhead when a file is implemented using multiple layers of storage objects.

We have found this building-block approach to be ideally suited for achieving the flexibility requirements outlined earlier. New policies can typically be implemented by combining together existing storage object classes in new ways. We have found that a surprisingly small set of storage object classes allows a large set of policies to be implemented. Also, it is easy to add new storage object classes to the file system to provide new functionality, adapt to new machine architectures, or extend the file system to support the semantics of new I/O interfaces.

It seems clear that constructing a file from a large number of storage objects entails some overhead. Our experience, however, indicates that in many cases our approach results in less overhead than more traditional approaches. The functions and data of a storage object are specific to the (typically simple) policies it implements, so it is often possible to avoid executing conditional statements (e.g., to determine the policy to use) that a more traditional approach would require. Also, the data is optimized to provide exactly the information required, and hence can be represented more compactly, improving the hit rate in the main memory cache. If the use of many storage objects happens to result in poor performance for some important class of files, then it is straightforward to define and add a new storage object class to specifically handle this class of files.

There are three ways an application or compiler can cooperate with HFS to optimize performance. First, when creating a file, the application can explicitly choose which storage objects are to be used and hence the policies that are to be applied. Second, for some storage object classes, the policies applied can be changed at run time. For example, the storage object class used for replicated files provides a function to change the policy for future read requests to: 1) use a particular replica, 2) use the replica that will require the minimum amount of time to handle the request, or 3) use the replica closest to the location of the memory buffer into which the data will be read. Finally, an application can attach temporary storage objects to a file at run time to invoke policies that are different from those established at file creation time. This allows, for example, an application to attach a prefetching storage object to an existing file that is specific to how the application expects to access the file data.

1.4 Contributions and dissertation outline

The main contribution of this work is the design and validation of a file system architecture that possesses the following properties:

Flexibility: HFS can support the wide variety of file structures and file system policies needed to deliver the the full I/O capability of a parallel supercomputer to a large set of applications.

Customizability: The structure of a file and the policies invoked by the file system when the file is accessed can be specified by the application. This allows the file structure and policies to be tailored to the requirement of a particular application.

Efficiency: The flexibility of HFS comes with little I/O and CPU overhead.

The unique nature of the environment in which this research took place, and the lack of existing infra structure, caused us to take a holistic view of the file system and its infra structure.[3] Few other file systems have been designed from scratch for: a NUMA multiprocessor, a micro-kernel operating system, an operating system that supports

[2] In the terminology of object-oriented programming, the definition of an object type is called the object's *class*.

[3] The file system architecture was guided by experience with the entire surrounding infra structure. While pursuing this research, new hardware for connecting disks to HECTOR was implemented, a device driver architecture for the HURRICANE operating system was designed, an NFS service was implemented, application-level I/O libraries were developed, a naming facility that directs requests to system services was developed, a scalable reader-writer locking algorithm was developed [72], and the basic IPC mechanism of HURRICANE was re-designed [45].

mapped files, or an operating system designed for scalability. Our file system architecture was designed as a part of a new experimental operating system (HURRICANE) on a new experimental hardware platform (HECTOR). The holistic approach we have taken has resulted in a number of secondary research contributions.

First, we have investigated the use of mapped files to a greater extent than other file system designers. As we will show later, much of the cost of I/O for many applications is the overhead that stems from copying data from one memory buffer to another and the cost of making calls to the operating system. Our file system makes extensive use of mapped files to reduce this overhead and in doing so improves I/O performance.

Second, a large portion of HFS is implemented within the address space of the applications that use it. This application-level facility, called the *Alloc Stream Facility* (ASF) [73, 74, 75], has several key advantages:

1. The structure of ASF makes it easy to adapt the facility to any operating system and hardware platform; taking advantage of features specific to each platform to improve performance.

2. ASF is designed for multi-threaded applications running on multiprocessors and allows for a high degree of concurrency.

3. ASF can support a variety of I/O interfaces, including stdio, emulated Unix I/O, ASI, and C++ streams, in a way that allows applications to freely intermix calls to the different interfaces, resulting in improved code re-usability.

ASF has been implemented for HURRICANE [129, 128], *SunOS* [63], *IRIX* [118], *AIX* [91], and *HP-UX* [133]. Also ASF has been adopted by TERA computer [2] as the I/O library for their parallel supercomputer. On most systems, improved performance can be achieved when applications use ASF rather than the facilities provided by the system.

As a third secondary research contribution, we have made extensive use of object-oriented technology not only to improve performance, but also for the more common goal of code re-use. For example, the NFS, naming, and disk file system services all share a great deal of code. This has allowed us to implement new services on HURRICANE with less programming effort than would be required on other systems.

Overview of dissertation

In Chapter 2 we define terminology used in this dissertation, describe some background necessary to understand it, present motivation for the key design choices we have made, and describe related research.

In Chapter 3 we provide an overview of the HFS architecture. In particular, we describe the object-oriented building-block approach, discuss the different logical layers of the file system, and provide examples of the storage objects defined by each layer. The layers are defined to (i) maximize the amount of the file system that can be implemented by an application-level library, and (ii) facilitate mapped-file I/O.

In Chapters 4, 5, and 6 we describe the details of the application-level library and system servers that make up our implementation of HFS. Chapter 4 discusses the interactions between the library, the different file system servers, and the HURRICANE micro-kernel. From a performance perspective, the most important file system server is the *Block File Server*, the server that controls the disks and distributes `read` and `write` requests across them. The architecture and implementation of this server is described in detail in Chapter 5. Much of our work has focused on the *Alloc Stream Facility*, the application-level library for HFS. The architecture and implementation of this application-level library is described in Chapter 6.

While we believe that fault recovery and scalability are important issues, the limited size of the prototype hardware used for our implementation made them irrelevant to application performance in our experimental environment. For this reason, we have only partially implemented the portions of the HFS architecture that relate to these issues. Our proposals for addressing these issues are described in Chapters 7 and 8.

In Chapter 9 we present results obtained from our implementation. These results demonstrate that (i) HFS can indeed support a variety of policies, (ii) the correct choice of policy can have a major impact on performance, (iii) HFS has low processing and I/O overhead and is able to export to applications the full I/O bandwidth of our hardware, and (iv) substantial performance advantages arise both from the choice to base the file system on mapped-file I/O and from the choice to implement substantial portions of the file system in an application-level library. The performance of the application-level library component of HFS is compared to existing libraries on several Unix platforms. The performance of HFS as a whole is examined on HURRICANE/HECTOR.

Chapter 2

Background and Motivation

The purpose of this chapter is to define common terms, give an overview of related work, and present motivation for several of the key design choices made for the HURRICANE file system. Section 2.1 describes the basic concepts and terminology of object-oriented programming. Section 2.2 describes some of the interfaces developed for sequential and parallel I/O. Section 2.3 discusses the basic methodology used by the Unix *Fast File System* (FFS) and related file systems. Section 2.4 describes our assumptions about the target environment for the HURRICANE file system. Finally, Section 2.5 gives an overview of other file system research.

2.1 Object-oriented programming

The HURRICANE file system exploits an object-oriented methodology to achieve much of its flexibility. In this section we informally introduce some of the concepts and terminology associated with object-oriented programming.

We use C++ terminology throughout this dissertation [124]. An *object* encapsulates both data and functions that operate on that data. The data and functions are referred to as *member data* and *member functions*. *Public members* define the object's external interface (both functions and data). An object's *private members* can only be used by the member functions of the object. The *class* of an object is its type; the class defines the organization of the data (i.e., the data structures) and the member functions. An object is said to be an *instance* of the class that defines it. Two special kinds of member functions are *constructors*, which are invoked when an object is instantiated to initialize the object, and *destructors*, which are invoked when an object is being terminated to run any clean up code.

Object classes can be *derived* from other object classes, where the *derived* class may redefine or extend the members it *inherits* from its *base* class. An object that is an instance of a derived class can be referred to either as belonging to the derived class or the base class. The base class determines whether a member function is *virtual* (i.e., polymorphic). When a derived class redefines a virtual member function, any calls to that member function invokes the derived (and not the base) class's definition of the function, even when the object is referenced as being an instance of the base class. A hierarchy of derived classes can be defined, where a derived class can itself be the base class from which another class can be derived. Also, an object class can be derived from multiple classes, in which case *multiple inheritance* occurs.

Object-oriented software design has several software engineering advantages. First, an object controls how its data is accessed, hence it insulates other objects from the details of both the data and the code for manipulating its data. Second, through the use of inheritance, the programming effort placed into designing an object class can be re-used by other classes with common characteristics. Finally, polymorphism (i.e., the use of virtual member functions) allows an object's member functions to be called without detailed knowledge of that object's class, that is, the object can be referred to as being an instance of some base class.

2.2 I/O interfaces

Much of our work has been motivated by inadequacies in current interfaces for accessing disk files and in the common implementations of these interfaces. In this section we describe the three main classes of these interfaces, namely:

`read/write` system-call interfaces, *mapped-file I/O* system-call interfaces, and application-level I/O interfaces. We give examples of particular instances of each of these interfaces, and describe the advantages and disadvantages of how they are commonly implemented. We conclude with a short description of parallel-I/O interfaces.

2.2.1 Read/Write system-call interfaces

The best known `read/write` system-call interface is the *Unix I/O* interface. The primary Unix I/O abstraction is a sequential byte stream, and is provided by an interface consisting of the system calls `open`, `read`, `write`, `lseek`, `close`, `fcntl` and `ioctl` (See Figure 2.1). The Unix I/O facility is simple, easy to use, and has proven to be versatile in that it can be applied in a uniform way to a large variety of I/O services, including disk files, terminals, pipes, networking interfaces and other low-level devices. We refer to an interface that can be used in this fashion as a *uniform I/O interface*. Uniform I/O interfaces allow programs to be independent of the type of data sources and sinks with which they communicate [21].

open(filename, mode) : opens a stream

read(stream, buffer, nbytes) : reads `nbytes` from the current stream offset into `buffer` and increments the stream offset by `nbytes`

write(stream, buffer, nbytes) : writes `nbytes` from buffer to the current stream offset and increments the stream offset by `nbytes`

lseek(stream, offset, whence) : moves the stream offset to `offset` according to `whence`

fcntl(stream, cmd, arg) : manipulates or locks open stream

ioctl(stream, request, arg) : special control functions

close(stream) : closes `stream`

Figure 2.1: The Unix I/O system-call interface.

With Unix I/O, when calling `read` or `write`, the application supplies a private buffer into which data should be read, or from which data should be written. The operating system copies data between system buffers and application buffers. For most operating systems, `read` is a system call that blocks until the data has been transferred to the application buffer. `Write` operations block until the data has been transferred to the file cache. `Read` and `write` operations are defined to be atomic with respect to all other I/O operations.

Accessing cached file data

The Unix I/O interface was developed in the early 70's, when a PDP-11 system with 64 KBytes of main memory was the state of the art. Since that time, hardware advances have led to systems with hundreds of Megabytes of main memory. Once accessed, files will often remain cached in memory, so that the operating system I/O calls no longer involve accesses to I/O devices. Many files in fact are created and deleted without ever being written to secondary storage [99]. For this reason, an important component of the cost of Unix I/O is the overhead that stems from copying data from one memory buffer to another and from the cost of making calls to the operating system.

Also, with new technology, the overhead of copying and of system calls has increased relative to processor speeds. In recent years processor speeds have improved much more dramatically than main memory speeds have, requiring the presence of large memory caches. This increases the cost of buffer copying (required for Unix I/O) relative to processor speeds, because copying either occurs through the cache, destroying the cache contents, or the copying occurs directly to and from (the relatively slow) memory. Moreover, in many systems (e.g., multiprocessors), memory bandwidth may be a contended resource.

The system-call overhead of Unix I/O has also been increasing relative to processor speeds [96]. This is again partially due to the effects of slower relative memory speeds and due to the increased number of registers some modern processors need to save and restore on context switches. In some cases, it is also due to the increased use of microkernel operating systems, where system calls are directed to servers rather than being handled directly by the kernel. With distributed and multiprocessor systems, the cost of a system call may be even more expensive if the operation requires service at a remote processor.

It is possible to implement most of Unix I/O in the application address space to reduce the number of system calls. For example, the OSF/1 implementation of Unix I/O uses an emulation library in the application address space that maps (Section 2.2.2) large file regions into the application address space and copies data to or from the mapped regions, hence reducing the number of system calls [30].

Accessing uncached file data

Unix I/O also entails large overhead when data is being transferred to or from disks. The copying between buffers (described in the previous section) increases contention on the memory. For example, researchers who have dramatically improved file I/O bandwidth (by introducing disk arrays [102]) have found that Unix I/O copying overhead makes it difficult to exploit this bandwidth [23]. Since the application can specify an arbitrary number of bytes for I/O (and use a buffer with an arbitrary alignment), and since most devices have some fixed block size, the operating system must buffer data, and hence it is difficult to avoid copying when Unix I/O is used.

Another disadvantage of Unix I/O is that (since `read` operations are synchronous) the requesting application is blocked until data is transferred from the disks. Forcing applications to see the high latency of disk operations can limit performance. Most Unix implementations attempt to alleviate this problem by detecting when an application is accessing a file sequentially and, if so, prefetching data into the file cache.

Supporting multithreaded applications

Multithreaded programs are becoming more common, both because of the increasing availability of multiprocessor systems and because of their suitability as a structuring mechanism for some applications (e.g., event-driven systems such as windowing systems). Unix I/O is grossly inadequate for multithreaded programs. An obvious inadequacy is the way in which errors are reported to applications: a single global variable *errno* is set by the I/O system whenever an I/O error is encountered [57]. If multiple concurrent I/O operations incur errors, then having a single *errno* variable makes it impossible to properly identify the errors.

A second inadequacy of Unix I/O for multithreaded applications is the way random-access I/O is supported. For random-access file I/O, *lseek* operations are needed to position the *file offset* for subsequent reads or writes to begin from that point. To prevent concurrent *lseek* operations by multiple threads from interfering with each other, it is necessary for a thread to hold a lock on the stream from the time of the *lseek* operation until the corresponding I/O operation has completed, effectively serializing I/O operations.

Other `read/write` interfaces

Other `read/write` interfaces overcome some of the limitations of Unix I/O. For example, Cheriton's UIO interface [21] (i) allows different threads to make random I/O requests without interference, (ii) takes into account the block size of the requested service, and (iii) returns thread-specific errors. Also, many Unix systems (e.g., IRIX version 5.1 and SunOS) provide asynchronous-I/O facilities that allow applications to request that data be read or written without blocking. Asynchronous-I/O facilities based on a `read/write` interface are, however, difficult to use [103], since on reading data, the application must check to see if the request has completed before it can use the data.

Extensions to Unix I/O allow some systems to deliver the full disk bandwidth to the applications. For example, IRIX 5.1 supports *direct I/O*, where the file system schedules the disk requests directly to or from the application buffer. A disadvantage of direct Unix I/O is that the file cache is bypassed, and hence an application should only use direct I/O if it knows that the data is not in the file cache (and cannot be effectively cached). Also, direct Unix I/O may require that the application use buffers that are a multiple of (and aligned to) the disk block size. These disadvantages may make direct I/O difficult for some applications to use efficiently.

2.2.2 Mapped file system-call interfaces

Most modern operating systems now support mapped-file I/O, where a contiguous memory region of an application's address space can be mapped to a contiguous file region on secondary store. Once a mapping is established, accesses to the memory region behave as if they were accesses to the corresponding file region. The `mmap/munmap` [58] interface is the standard Unix mapped-file interface. `Mmap` takes as parameters the file number, protection and control flags, the length of the region to be mapped, an offset into a file, and optionally a virtual address indicating where the file should be mapped. It returns a pointer to the mapped region. `Munmap` accepts as parameters the address and length of the region to be unmapped.

Although support for mapped files is becoming a common feature on many operating systems, few applications have yet been written to use it. The most important reason for this is that mapped files cannot provide a uniform interface for all I/O. Unix I/O allows applications to use the same operations whether the I/O is directed to a file, terminal or network connection. Mapped file I/O can only be used for I/O directed to a random-access block-oriented I/O device that can be used to handle page faults, such as a disk.

Accessing cached file data

When data in the file cache is being accessed, the use of mapped-file I/O can reduce both copying and system-call overhead. With mapped-file I/O, rather than requesting a private copy of file data, the application is given direct access to the data in the file cache. Hence, the use of mapped-file I/O eliminates both the processing and the memory bandwidth costs incurred to copy data.

With mapped-file I/O the system-call overhead is reduced (relative to Unix I/O) because applications tend to map large file regions into their address space; if it turns out that the application accesses a small amount of the data, only the pages actually accessed will be read from the file system. In contrast, the application must be pessimistic about the amount of data it requests when Unix I/O is used, since `read` incurs any I/O cost when invoked. The reduction in the number of system calls when mapped-file I/O is used may be offset by an increase in the number of page faults. However, some systems (e.g., AIX) do not incur any page faults when pages in the file cache are accessed. Also, as we will show in Appendix A, the cost of a page fault is substantially less than the cost of a read system call on many systems.

Two cases where mapped-file I/O can result in performance worse than Unix I/O are (i) modifications of entire pages when the data is not in the file cache, and (ii) writing a large amount of data past the end-of-file (EOF). In the former case, mapped-file I/O will cause the page fault handler to first read from disk the old version of the file data. This is not necessary for `Unix I/O`, since the system is aware that the entire page is being modified. Similarly, mapped-file I/O will cause the page fault handler to zero-fill each new page accessed past EOF. Again, this does not have to be the case with Unix I/O.

In practice, the cost to zero-fill a new page may be very small. In fact, on some systems zero-filling a page that will be modified in its entirety may actually improve performance. Most modern processors are capable of zero filling cache lines without having to first load them from memory. Thus, with such hardware support, zero-filling the page saves the cost otherwise incurred to load from memory the data being modified.

Accessing uncached file data

In many current systems, achieving high I/O rates when reading data from disk is difficult if mapped-file I/O is used. The basic problem is that disk-read requests are for individual pages because they result from page faults. Unix I/O `reads`, on the other hand, can request a large number of disk blocks in a single request. Making large requests amortizes per-request costs (e.g., the cost to cross address spaces in a micro-kernel OS). Also, large requests make it easier to exploit either locality in the disk blocks allocated to the file (e.g., cylinder clustering) or concurrency in the disk system (e.g., if a file is striped across multiple disks).

Some operating systems give the application limited control over how file data is to be cached. For example, the SunOS `madvise` system call informs the operating system how a mapped region will be accessed (e.g., sequential or random access). This allows the operating system to prefetch data and to eject file pages with some knowledge about the application access pattern to the data. Similarly, the SunOS `msync` call can be used to either synchronously or asynchronously flush file data to disk, and the `mlock` call can be used to lock pages of a file in memory.

Mapped-file I/O may result in less I/O activity than Unix I/O if the amount of main memory is limited. When an application uses *Unix I/O*, there are often multiple copies of the same data resident in memory, since data is copied from the file cache to application specific buffers. This can result in an increase in paging or swapping activity. If mapped-file I/O is used, no extra copies of the data are made, so the system memory is used more effectively. Moreover, if data in the file cache is not modified it does not need to be paged out, since the data is already on disk, whereas with Unix I/O, the copy of the file data read in the application address space is considered dirty and hence must be paged out to disk.

As we will describe later, HURRICANE (which provides a mapped-file interface) provides a `prefetch` operation that allows the application to asynchronously read a (potentially) large number of pages from disk. This operation has the same advantages as large Unix I/O read requests (i.e., amortizing per-request overhead and sending multi-block requests to the disks), but is not synchronous. The asynchronous nature of `prefetch` requests is similar to asynchronous Unix I/O `read` requests, but applications using `prefetch` do not need to check if data is valid before accessing it. If a page is accessed that has not yet been read from disk then a page fault will occur and the faulting process will be blocked until the data becomes available.

Other mapped-file interfaces

Other interfaces for mapped-file I/O include the HURRICANE `BindRegion/UnbindRegion` interface (inspired by V [21]) and the AIX `shmget` interface [91]. The `BindRegion` interface is similar to the `mmap` interface, except that the management of bound regions is based on regions of an application's address space rather than individual pages. The AIX `shmget` interface maps entire files into regions of the application address space. AIX exploits the segments of the RS/6000 architecture so that all programs accessing the same file share the same segment. If a page in the file cache has previously been accessed by a processor, new accesses to that page (even by a process that has not previously accessed it) incur no page fault cost.

2.2.3 Application-level interfaces

Application programs typically do not invoke system-call interfaces directly, but instead use higher-level facilities implemented either by the programming language (e.g., Pascal, Fortran, and Ada) or by application-level libraries associated with the language (e.g., Modula, C and C++). I/O facilities at the application level offer several advantages. The interfaces can be made to match the programming languages, both in terms of syntax and semantics, and they can provide functionality lacking at the system level. They increase application portability because the I/O facility can be ported to run under a variety of operating systems. Application-level I/O facilities can also significantly improve application performance, primarily by buffering the input and output to translate multiple fine-grained application-level I/O operations into individual coarser-grained system-level operations. For example, if an application inputs one character at a time, each request can be serviced from an application-level buffer without a system call; a system-level read must be issued only when the buffer is empty.

The standard I/O library for the C programming language, stdio, is an example of a well known application-level I/O facility that is available on numerous operating systems running on almost all current hardware bases [106]. The stdio interface (Fig. 2.2) provides functions that correspond to the Unix I/O interface (`fopen`, `fread`, `fwrite`, `fseek`, `fclose`), functions for character-based I/O (`getc`, `putc`), and functions for formatted I/O (`fprintf`, `fscanf`). The function `ungetc` pushes a byte that was previously read back onto the input stream so that in can be reread later, and is an example of functionality not directly available at the system level.

As is the case with most application-level I/O facilities, the interface and implementation of stdio have remained largely unchanged since the mid-70s [123]. For this reason, the stdio interface has many of the same problems as the Unix I/O interface. For example, the basic I/O operations, `fread` and `fwrite`, require that data be copied to or from application-specified buffers. The stdio interface is also inappropriate for multi-threaded applications.

Some of the problems with stdio have been addressed by sfio, an application-level library developed by Korn and Vo [65]. Sfio is a replacement library for stdio that, in addition to the stdio interface, provides a more consistent, powerful and natural interface than stdio and uses algorithms that are more efficient than those used by typical stdio implementations. The sfio implementation exploits mapped-file I/O whenever possible. Also, sfio provides an operation called `sfpeek` that allows the application access to the library buffers, getting rid of copying between the application and library buffers in some cases. However, `sfpeek` data is only available until the next I/O operation, which is not suitable for multi-threaded applications.

General I/O:

fopen(filename, mode) : opens a stream

fread(buf, size, nitems, stream) : reads `nitem` items of size `size` from the current stream offset into `buffer` and increments the stream offset

fwrite(buf, size, nitems, stream) : writes `nitem` items of size `size` from the `buffer` to the current stream offset and increments the stream offset

fseek(stream, offset, whence) : moves the stream offset to `offset` according to `whence`

fflush(stream) : flush buffers

fclose(stream) : close `stream`

String I/O:

fgets(string, nbytes, stream) : reads a line of at most `nbytes` into `string`

fputs(stream, string) : writes `string`

Character I/O:

getc(stream) : reads next character

ungetc(character, stream) : returns last character

putc(stream, character) : writes a character

Formatted I/O:

fprintf(stream, format, args...) : formats `args` according to `format`, writes to `stream`

fscanf(stream, format, args...) : reads from `stream`, formats `args` according to `format`

Figure 2.2: A subset of the stdio interface

Since application-level facilities buffer data, it is difficult for an application to concurrently use more than one application-level facility, or even both an application-level and a system-level facility. For example, the result of intermixed calls to both the Unix I/O and the stdio interfaces depends on the particular implementation of the library; a `read` operation that follows a `getchar` operation may receive data that is thousands of characters later in the stream than that returned by `getchar`. Since most languages provide their own I/O facilities, it becomes difficult to implement an application with individual parts written in different languages.

The *Alloc Stream Facility* (ASF) that we have developed (described in Chapter 6) provides an interface that overcomes the limitations of the stdio interface, has better support for multi-threaded applications than *sfio*, and can support multiple I/O interfaces efficiently. Requests to the different interfaces provided by ASF can be interleaved with predictable results.

2.2.4 Parallel I/O interfaces

As with sequential I/O interfaces, parallel I/O can be supported with either system-call interfaces or application-level interfaces.

Parallel system-call interfaces

Kotz provides an overview of different parallel I/O interfaces in [67], and proposes a set of extensions to the *Unix I/O* interface to handle parallel I/O. These extensions include (i) a choice of per-application or per-process file stream pointers, (ii) `readp/writep` operations that return the file offset used to handle the request, (iii) record oriented I/O,

(iv) the ability to access a file as a collection of subfiles, and (v) a flexible mapping function between the logical file offset of the application and the position in the file. The latter two extensions allow each process of an application to maintain its own view of the file data, thus allowing sequential requests by each process to be distributed across the disks used for the file in a fashion specific to the application.

In the Bridge (also called PIFS) file system [35, 36], Dibble defines three interfaces: 1) a *standard* interface that is similar to Unix I/O except that it is record oriented, 2) a *parallel open* interface, where a process issues `read` and `write` requests that, in a SIMD fashion, transfer data concurrently to a pre-specified set of workers, and 3) a *tool* interface, where applications have access to the low-level details of the file system. The tool interface allows applications to take into account the disk/memory locality by executing code on the I/O nodes that store the applications' data.

The Vesta file system for the SP2 multicomputer [25, 26] provides a record oriented interface optimized for matrix operations. On creating a file, the application specifies the number of "physical partitions" (normally per-disk sub-files) that the data will be distributed across and the record size. When opening a file, the application specifies a "view" of the file that determines a mapping between the records in the different partitions and the order that the application will see the records (for that open instance). This view allows read and write requests to be directed to, for example, a single physical partition or a stripe of data across a number of (or all) physical partitions. More importantly, it allows common matrix access patterns to be specified directly. The Vesta file system also provides asynchronous requests and `RECKLESS` requests that have relaxed the Unix I/O atomicity constraints.

The nCUBE file system [31, 32] provides a mapping interface that is similar (but more constrained) than that provided by Vesta.

According to Kotz [67], most file systems of commercial multiprocessors provide the standard Unix I/O interface, typically with extensions for asynchronous I/O. The KSR provides for mapped-file I/O [11]. Some file systems (e.g., the CFS file system for the iPSC/2 and iPSC/860 [105]) implement the Unix I/O interface in a user-level library on top of a lower-level internal system-call interface. The OSF/1 file system for the Paragon implements the Unix I/O interface in an emulation library on top of a mapped-file interface [111]. Both mapped-file I/O and Unix I/O interfaces are exported to the application on the Paragon.

Parallel application-level I/O interfaces

Recently, there have been several proposals for developing application-level parallel I/O libraries [9, 25, 36, 44, 49, 119]. By providing an interface that matches common application requirements, these libraries can simplify the job of the application programmer. For example, they can provide operations specific to accessing a 2-D matrix. Also, the library can exploit both system-specific and application-specific information to optimize I/O performance.[1] Finally, these libraries can help make I/O intensive applications portable, because they can provide a consistent interface regardless of the system specific file system interface.

A major difficulty in designing the interface of an application-level library is that the information needed for optimizing I/O is typically architecture specific. For example, for some architectures there is no need take locality into account, while for other architectures it is necessary to take into account both the disk/memory and the memory/processor locality. As another example, if sufficient memory is available, an entire matrix can be read into memory (with the optimal transfer method dependent only on how the data is distributed across the disks), while in other cases the access pattern to the file data must be specified so that the library can overlap the computation of the application and the I/O (e.g., by prefetching).

Only recently have parallel systems with large-scale I/O become available. Hence, application I/O requirements for this hardware base are not yet well understood and the application-level I/O interfaces are not yet mature. All of the proposals for application-level parallel I/O facilities contain at least some suggestion that the interface to their facility should be expected to evolve in the future.

The functionality a system-call layer should have to efficiently support a sophisticated application-level facility is not clear. A strong argument could be made for having the lower-level facility only provide per-disk files, with the application-level facility responsible for providing applications with the abstraction of files distributed across multiple disks. The attraction of this approach is that it centralizes all decision making (e.g., synchronization, distribution, cache

[1] An example of system-specific information that might by specified to an application-level library is that a file is distributed across 10 disks in a striped fashion. An example of application-specific information is that a file contains a two dimensional matrix that should be distributed to the processors in a block cyclic fashion. Note, while application-level facilities could accept such information to optimize I/O performance, most current application-level facilities are written to be independent of the underlying system. For example, Scott describes how it is necessary to bypass the Fortran run time library to get good I/O performance on an IPSC/860 [115].

management) in the application-level facility, simplifying the lower levels of the file system and avoiding contradictory policies (e.g., different prefetching policies) being concurrently invoked at different levels. However, such an approach has disadvantages in a multi-programmed environment. In particular, it is difficult for an application-level facility to properly handle and optimize for (i) synchronization between different applications accessing the same (distributed) file, (ii) the caching of file data in memory, especially when the amount of main memory is constrained, and (iii) concurrent use of the same sets of disks by multiple applications. Also, if the file system is not aware of the distribution of data across (sub) files, then to handle a read request for data distributed across multiple sub-files, the application-level facility must make independent requests to each of the sub files. If the file system is aware of the distribution, then the cost of a system call (e.g., the cost to cross address spaces) can be amortized over a larger number of blocks.

The application-level interface we have developed for HFS is an object-oriented interface similar to that developed by Grimshaw and Loyot [49, 50] for the ELFS file system. An object-oriented approach has the advantage that the interface can be easily extended by adding new object classes (through inheritance, re-using the code of other classes with common characteristics). The information provided by the application to the file system in order to optimize I/O is divided into two phases, namely: 1) the choice of the object class and the information provided on instantiating an object of that class, and 2) subsequent requests to an instantiated object. If all system-specific information can be specified in the first phase, then (through the use of virtual functions) the majority of the application code can be written to be independent of the system. This is similar in philosophy to the pragmas used by HPF [41] that specify data distribution across memory banks, where only the pragmas and not the code that accesses memory has to be changed to optimize performance when a HPF program is ported to a new platform.

2.3 The Unix Fast File System

Most Unix file systems, including parallel ones, are based on the the Unix Fast File System (FFS) [87]. In this section we describe how the FFS (together with the Unix operating system) implements files, and why the approach taken by this file system to implement files is inherently inflexible.

In addition to the file data, all file systems maintain state for each file that describes the file to the file system and to the application. This state is often referred to as *meta-data*, and can either be persistent and stored on disk with the file data (e.g., meta-data describing the disk blocks used to hold the file data) or transitory existing only when a file is being actively accessed (e.g., an offset for an open file). Some meta-data is visible to the application, such as the file length, while other meta-data is hidden from the application, such as the disk-layout information.

Much of the inflexibility of the FFS is due to the way it maintains and interprets file meta-data. The FFS persistent meta-data is maintained in four types of data structures (Fig. 2.3a), namely: 1) direct blocks that contain pointers to disk blocks holding file data, 2) indirect blocks that contain pointers to direct blocks, 3) doubly indirect blocks that contain pointers to indirect blocks, and 4) inodes that contain pointers to a fixed number of direct blocks, indirect blocks, and doubly indirect blocks. The persistent visible meta-data (e.g., file length, access times, access permissions) is stored as part of the inode. The file system identifies the type of a data structure, and hence the code to use to manipulate it, only by its context (i.e., the path used to reach it). For example, the file system can recognize that a particular block is an indirect block only by the fact that a particular inode pointer references it.

FFS inodes are located at fixed areas on each disk partition, where the disk space used for these inodes is allocated when the disk partition is created[2]. Unix identifies a particular file on a disk partition by an *inode number* which is used as a index into the array of inodes. The disk blocks used for the other persistent data structures (i.e., direct blocks, indirect blocks, and doubly indirect blocks) are fixed when they are allocated.

For transitory meta-data, the Unix kernel maintains extended in-memory inode structures and per-open data structures. In-memory inodes include (or reference) meta-data used by file system policies such as prefetching (i.e., the last block accessed). Per-open data structures maintain information specific to an open instance (e.g., the file offset). If an application is using an application-level library (like stdio), then additional per-open state is kept by the library in the application address space.

The FFS has no flexibility in supporting alternative file structures — the structure of a file is determined entirely by its file length. The way files are structured by the FFS is a compromise between the varying requirements of different files (e.g., write mostly, read mostly, large, small, etc.) and hence is optimal for none of them.

[2] The disk space consumed by inodes is generally 6.25% of the total.

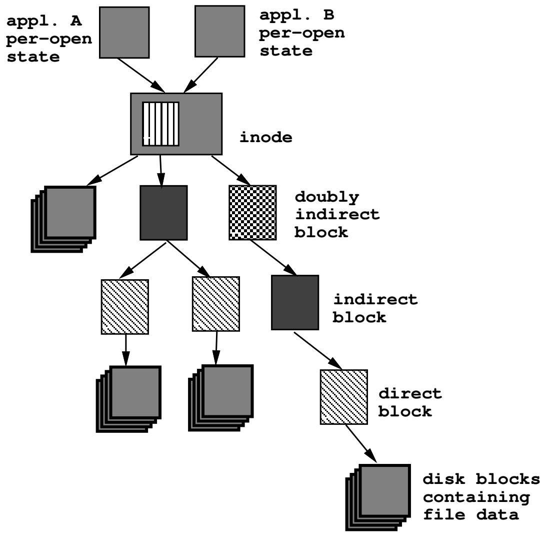

Figure 2.3: Unix/FFS implementation of an open file.

The policies that can be invoked on a FFS file are limited, in part because there is no way to associate policy-related persistent meta-data with the file. Such meta-data is important so that policies can be invoked automatically when a file is opened. Also, as will be discussed in Chapter 3, such meta-data is crucial for many policies such as prefetching, locking and compression that can depend on characteristics of the file data.

Another reason why the policies that can be invoked on a FFS file are limited is that they are typically implemented internal to the Unix kernel [52]. Since it is difficult to modify the Unix kernel, it is hard to develop new file system policies, especially if the new policies require input from the application (or application-level library). For example, the only prefetching policy supported by most Unix systems is sequential prefetching; a policy that can be invoked by the kernel when it detects that a file is being accessed sequentially. Existing Unix systems provide no way for applications to explicitly invoke prefetching policies specific to other access patterns.

Most parallel file systems implemented to date have provided for parallel files by extending the approach used by the Unix FFS. Typically, they support a single file type, where files are striped across a number of disks, or at most a small number of file types. The structure and policies used for a file are typically determined by the file type and (possibly) a small amount of additional persistent state such as the file length, the stripe length, and the number of disks used.

We used a similar approach in our first attempt to create a parallel file system, but found that we had to create a new file type every time we wanted a new file structure and any time we wanted to have a new policy invoked automatically when a file was opened. The proliferation of file types resulted in excessive complexity and code duplication. Moreover, every time a new file type was added to the system, a substantial part of the system (i.e., different libraries and servers) had to be modified to deal with the new file type. The complexity of adding new file types limited the file structures and policies we could support, and caused us to discard this first file system and develop a new file system based on a different way of representing files.

2.4 The target environment

A realistic evaluation of a file system architecture can only be obtained by targeting a specific class of machine architecture and implementing the file system on a real machine as part of a real operating system. In this section we first describe the target architecture of our file system, and then the characteristics of the HURRICANE operating system that have most influenced the design of HFS.

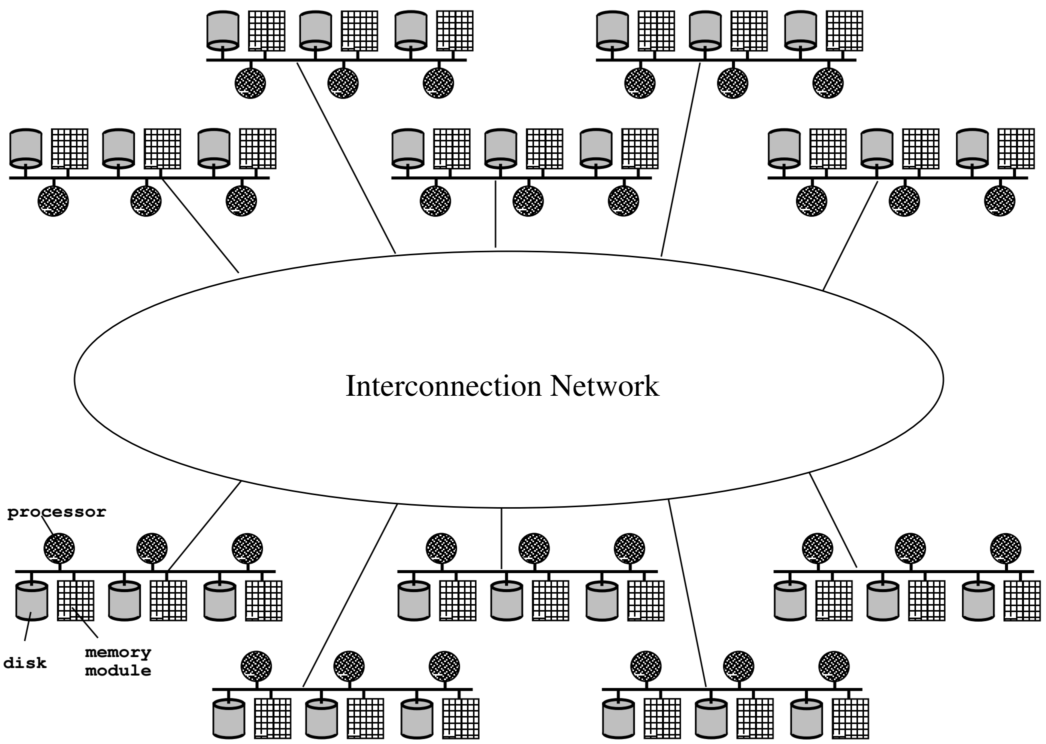

Figure 2.4: NUMA multiprocessor with distributed disks.

2.4.1 The target architecture

The class of machine architecture we considered is the class of large-scale NUMA multiprocessors with disks distributed across the multiprocessor (Figure 2.4).

NUMA multiprocessors achieve their scalability by using segmented architectures [12, 79, 131]. 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 to minimize the access time and network bandwidth required by that processor.

If such a system is to scale in its I/O capabilities, then the number of disks should also increase with the size of the system. For our file system, we assume that these disks are spread across the segments of the NUMA multiprocessor and that the distribution of data to the disks is under the control of the file system software[3].

With such an architecture, the file system must be concerned with distributing file data across the disks to balance the load on these disks. Also, the file system must be aware of the locality between the memory banks and the disks that are the sources or sinks of file data. It may appear that locality is not important, since the time to transfer a disk block across the interconnection network is typically insignificant compared to the cost of getting the block from disk. However, the interconnection network will become congested if a large number of disks are concurrently transferring data over the network [56].[4] Hence, it is important that most of a processor's I/O requests be directed to nearby devices.

An alternative to distributing the disks across the multiprocessor is to put all disks in a single large disk array [7, 19, 20, 23, 47, 62, 101, 102, 107], where the distribution of data to the disks is handled by a hardware controller associated

[3] In the context of HFS, when we refer to the file system software we are referring to all components of the file system including the servers, application-level library, and device drivers.

[4] This is especially the case with architectures like HECTOR and NUMAchine, where the hardware is designed under the assumption that the applications exploit locality, and hence the bisection bandwidth is quite low.

with the array. Using a disk array is attractive, because it greatly simplifies the file system issues. For example, it is not necessary to consider processor/disk locality since all disks are equally distant. The disk controller manages the distribution of the data across the disks, further simplifying the task of the file system. Finally, redundancy can be built into the array, allowing the file system to be designed assuming full reliability in the underlying medium.

However, disk arrays have disadvantages, especially in the case of a shared-memory multiprocessor. The disk striping techniques used by the controllers of such disk arrays, where data is distributed in a simple round-robin fashion across the array, are very restrictive. For parallel applications it is crucial that the file data distribution across the disks matches the access patterns of the application [27, 56, 135]. Also, it is expensive to build a network connection to the disk array that does not become contended when a large number of disks concurrently transfer data. (The latter problem has been addressed in part by the TickerTAIP parallel RAID architecture and by the I/O architecture of the CM-5, both of which provide multiple paths to the disk array. However, these architectures still use simple disk striping for distributing data across the disks [16, 127].)

We assume in our file system that the disks, processors, and memories are all connected using the same network. An alternative would be to use a separate network for the disks, which can have the advantage that data can be distributed to the disks without regard for locality to the memory modules [46]. However, we believe that for many applications memory/disk locality can be effectively exploited by file system software, and hence the additional cost of a parallel network is wasted for these applications.

The majority of the current large-scale systems (including the Vulcan, SP2, iPSC/2, iPSC/860, Paragon, CM-2 and CM-5 [26, 39, 83]) have separate I/O and compute nodes. This has the advantage that: 1) the resources of the nodes (e.g., memory for caching, or special hardware accelerators for computing parity or compression) can be tuned to their function, 2) I/O requests do not interfere with another application's computation, and 3) it may be possible to reduce expense by using a single channel to control a larger number of disks. However, in systems where there is a strict division between compute nodes and I/O nodes, the processing power of the I/O nodes is wasted during periods of heavy computation and the compute nodes are idle during high I/O periods [90]. Also, we believe that if I/O resources are provided to all processors, the locality between the processor and its local disks can be exploited by the file system.

Our choice of target machine architecture was motivated largely by the availability of such a machine (HECTOR). However, much of our work is also relevant to multiprocessors with disk arrays, multiprocessors with separate I/O networks, multicomputers (such as the Paragon [55] or CM5 [127]), and even vector or SIMD computers with a large number of disks.

2.4.2 The target operating system

The HURRICANE File System is being developed as part of the HURRICANE operating system [129, 125, 128]. There are a number of characteristics of HURRICANE that have had an impact on the design of the file system. First, 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 Chapter 4, most of the file system exists outside of the HURRICANE micro-kernel.

Second, HURRICANE is a single-level-store operating system that supports mapped-file I/O. Most file I/O in HFS therefore occurs through mapped-file I/O. Mapped-file I/O has the advantages that copying overhead is reduced and that all of main memory is used as a cache of the file system. 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.

Third, HURRICANE supports a new interprocess communication facility, called *Protected Procedure Calls* (PPC) that 1) allows fast, local, cross-address space invocation of server code and data, and 2) results in new workers being created in the server address space as they are required to handle requests [45]. Since PPC requests are fast[5], the HURRICANE file system can be partitioned into separate address spaces without a major impact on performance. The PPC facility also affects the file system architecture in that it simplifies deadlock prevention. Since PPCs create new worker processes on demand, servers can make PPC requests to each other without fear of running out of workers that can satisfy the requests.

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

Finally, HURRICANE is structured for scalability using a technique called Hierarchical Clustering [129, 130]. With this structuring technique, the operating system is constructed from a hierarchy of clusters, where each cluster manages a unique group of "neighboring" processors. All system services are replicated to the different clusters. The clusters cooperate and communicate in a loosely coupled fashion to give applications an integrated and consistent view of the system. Scalability has been a secondary goal with the HFS and we currently do not support clustering in the implementation of the HFS. Hierarchical Clustering and our future plans for clustering the file system are described in Chapter 7.

2.5 Related work

In this section, we describe some of the related file system research. We start by describing studies of the I/O demand of parallel or supercomputer applications, and the workload models others have used to evaluate or characterize file system performance. Subsequent sections describe file system policies for optimizing performance. We conclude with a brief review of the salient characteristics of other file system architectures.

2.5.1 Workload studies and models

Most current parallel machines have poor support for high-performance I/O [33, 56]. As a result, the parallel processing community has mainly studied applications that have small I/O requirements. In this section, we provide motivation for our research by describing studies that suggest that many important applications have high I/O requirements. Also, we describe work by others that attempt to characterize the types of I/O requirements supercomputer applications have.

Several examples of scientific applications requiring high performance I/O are described by Intel [54]. Many of these applications require: 1) the ability to handle working sets that are larger than any possible amount of physical memory, 2) the ability to checkpoint the state of programs that can run for many days, and 3) the ability to read and write data sets composed of many Gigabytes of data, both in a random-access and a sequential fashion. Many of the applications require that the persistent data be in tertiary storage; the file system is used as a staging area for data actively being accessed. Scott describes one of these applications, out of memory solving of large systems of linear equations, in more detail [115]. The example provided is that of an airplane modeled with 150,000 degrees of freedom, requiring 90 Gigabytes of storage to hold the matrix that is accessed in full on every iteration.

del Rosario and Choudhary give a summary of the overall I/O requirements for a number of Grand Challenges applications [33]. These problems have similar I/O requirements to the applications described above. A particular example of a Grand Challenges problem is the genome-sequence pattern matching problem [3]. The sequence comparison is "embarrassingly parallel" since a single input sequence has to be compared against a large number of independent sequences in the data base. These sequences can be very large and are read sequentially. (While currently the data base can fit in the memory of large supercomputers, this problem is expected to eventually become I/O bound.) An application with I/O characteristics similar to genome sequence pattern matching is a full text retrieval program described by Lin and Zhou [81].

NHT-1 is an I/O benchmark developed at NASA to evaluate the I/O systems of parallel machines [17, 40]. This benchmark: 1) evaluates the capability of systems to handle problems that iterate for many steps, where at each step solution files are written out concurrently with the calculation of the next step, and 2) evaluates how long the file system takes to write to and read from secondary storage 80% of the system memory.

Miller and Katz traced and analyzed the I/O behaviour of a number of supercomputer applications on a Cray Y-MP [89] (mostly computational fluid dynamics problems). The I/O accesses of these programs are: 1) a relatively short burst of large I/O requests to input and output application data, 2) periodic short burst of (for most applications) large I/O requests to checkpoint application state, and 3) staging I/O required in each iteration because the memory is insufficiently large to hold the entire problem. The third category dominated the I/O requests in the applications considered.[6] Miller and Katz observed that the staging activity for all of the applications was bursty, but with a very regular cycle corresponding to the stage in the iterations. For several applications, the reference pattern of the I/O

[6] As Miller and Katz state, applications on the Cray Y-MP are charged by the scheduler for CPU cycles and memory used, but the I/O requirements are ignored. Since the Cray has a fast solid state disk, some of the applications take advantage of this idiosyncrasy, maximizing the time they are granted the processor by minimizing their in-memory working set. Because the Cray does not have virtual memory, this staging I/O is done under the direction of the applications (rather than as a result of paging activity by the memory manager). Miller and Katz argue that the applications can control staging activity in an application dependent fashion more effectively than the operating system could if paging were available.

was essentially identical for each cycle. Another study of I/O intensive supercomputing applications by Pasquale and Polyzos [100] supports Miller and Katz's results. Galbreath, Gropp and Levine [44] investigated the I/O pattern of various applications running at Argonne National Laboratory. Their characterization of the application demands is similar to Miller and Katz's.

Kotz and Ellis [70, 71] used a synthetic workload to analyze different cache management and prefetching policies. They characterize parallel application access patterns as: local whole file, local fixed-length portions, local random portions, segmented, global whole file, or global fixed portions. The local access patterns are those where each process independently accesses portions of the file sequentially. The global access patterns are those where the access pattern described is the merged access pattern of the different processes in the application. Crockett [28] describes six different access patterns: sequential, sequential partitioned, sequential interleaved, sequential self-scheduled, global random and partitioned random.

Reddy and Banerjee [108] studied the implicit and explicit file I/O activity of five applications from the Perfect benchmark suite running on a simulator. These programs were originally written: 1) without trying to avoid false page-level sharing, 2) to run on machines with small amounts of memory, and 3) to do little I/O. To make the applications I/O bound, Reddy and Banerjee extrapolated the I/O traffic for the case where processors are sped up by a factor of 10.[7] When considering implicit I/O for their applications, they found: 1) that the majority of pages are shared, and hence there is little advantage in trying to exploit processor/disk locality, and 2) the working sets of the applications are very small, and hence implicit I/O is not a concern for these applications. They found that there was substantial variance in the size of the explicit I/O requests of the different applications, and that the explicit I/O requests were entirely sequential.

The success of the world championship calibre Chinook checkers program is largely due to its endgame databases [114]. The current databases represent over 150 billion positions, requiring 30 Gigabytes of storage when uncompressed. Using application-specific techniques, the databases are compressed into 5.22 Gigabytes of disk storage. Commonly accessed portions of the database are pre-loaded into memory and have a greater than 99% hit rate with a 500 megabyte cache. Cache misses result in random-access I/O. Even with this large cache, the sequential version of the program is I/O bound. A parallel searching version of Chinook further increases the I/O rate [84]. (Computing the database is even more I/O intensive than running a match.)

2.5.2 Latency hiding

Even if a system has a large disk bandwidth (i.e., if there are a large number of disks) it may be difficult for I/O bound parallel applications to exploit this bandwidth, because of the high latency of operations that go to disk. Hence, it is important that a large portion of applications' I/O requests be serviced from the file cache. The three issues that researchers have concentrated most on are: 1) when to eject data from the file cache, 2) when to write back dirty data, and 3) when to pre-read or *prefetch* data into the file cache.

The majority of research on latency hiding has been for uniprocessor systems. A well-known trace study of uniprocessor Unix systems by Ousterhout *et al.* found that files are typically read and written sequentially in their entirety, files are repeatedly accessed, many files are temporary and, if they are effectively cached, will be deleted before ever being written to disk [99]. From the preceding section, it is clear that, while sequential access may be the norm for many parallel applications, the remainder of the conditions are less likely to hold. A good overview of different latency-hiding techniques is given by Kotz [66].

Kotz and Ellis have studied automatic cache management and prefetching techniques for shared-memory multiprocessors [69, 70, 71], where automatic means that the system invokes policies dependent on detected application access patterns (rather than in response to application directives). For cache replacement, Kotz and Ellis propose a variant of LRU that ensures that currently accessed blocks cannot be ejected.[8] Assuming that blocks are typically written in their entirety, but that multiple processes may conspire to write a block, Kotz and Ellis propose a *write full* policy, that delays writing a block in the file cache until all data in the block has been written. For prefetching, Kotz and Ellis propose two automatic prefetching predictors. IOPORT is a local (i.e., per process) predictor that repeatedly tries each of: one-block look-ahead, infinite-block look-ahead, and portion recognition (that detects when processes each read sequential portions for a regular length and skip). RGAPS is a global predictor that assumes sequential

[7] It is difficult to say if such an approach yields useful results, since (presumably) the programs were initially written under the assumption that they were not I/O bound.

[8] Their policy maintains a reference count that indicates the number of processes actively accessing a block. This reference count is used to avoid the block being ejected while it is the most recently used one for some process.

access until it detects random accesses. Kotz and Ellis compare their policies against simple and optimal policies on the RAPID-Transit testbed [66], which is a file system implemented on a shared-memory multiprocessor that uses simulated disks, and found that for the synthetic workload used (described earlier) their relatively simple automatic prefetching techniques perform close to optimal in many cases.

Kotz and Ellis did not study the impact of a multiprogrammed environment, where multiple applications contend for the same file cache and disks. Also, they did not study bursty access patterns, a characteristic common to many of the applications described in the previous section. Finally, Kotz and Ellis did not consider implicit I/O, where the I/O results either from mapped-file I/O or because of insufficient main memory. Some of the policies they developed may be difficult to use with implicit I/O; for example, their write full policy cannot be used with mapped-file I/O since file blocks are mapped in their entirety into the application's address space. An important observation in their work is that, even given optimistic assumptions about the workload, automatic prefetching can in some cases result in substantially worse performance than if no prefetching is done.

An automatic policy for page prefetching due to implicit I/O was proposed by Song and Cho [121]. This approach uses a history of previous page faults to predict subsequent page faults when an application repeatedly accesses pages in the same order.

While automatic cache management may be the only feasible alternative on uniprocessor systems, where applications expect to run on different (binary compatible) systems, it is not clear that this is the best choice for multiprocessors, where programs are at the least re-compiled for different platforms. Recent research has shown that regular memory access patterns can be detected by compilers, and the compiler can use this information to insert instructions into the code that prefetch data into the processor cache [93, 94]. It seems feasible that the compiler could provide similar information to the file system to manage the file cache. Some steps have been taken in this direction. For example, Malkawi and Patel describe how memory directives can be inserted by the compiler to manage the memory being used for an application in a multiprogrammed environment [86]. Patterson, Gibson and Satyanarayanan propose having the compiler provide the file system with high level *hints* that will allow intelligent prefetching [103].[9] The ELFS file system prefetches file data by taking advantage of application-specific knowledge of both the logical file structure (e.g., a two dimensional matrix) and the application access pattern [49, 50]. (An ELFS application specifies its access pattern by its choice of object.)

While it is typically not under software control, it is important to keep in mind that most current disks support disk caches into which data is prefetched sequentially according to the position of the data on the disk. It is also important to consider where in memory data should be cached. The Intel CFS prefetches data sequentially into memory on the I/O nodes. On the other hand, the Intel Paragon prefetches data into the memory of the compute nodes [111]. With a shared-memory multiprocessor there is a greater range of alternatives. For example, data can be read into a remote memory [78], and, rather than ejecting data from the file cache, it can be migrated to remote memories.

2.5.3 Disk block distribution

The majority of file systems for parallel machines use policies that distribute data in a simple round-robin fashion across a number of disks [11, 31, 36, 66, 105], possibly with the stripe size and number of disks to stripe across determined by the application. However, a number of studies have indicated that it is important that data be distributed to the disks in a mapping that matches the application access pattern [9, 26, 27, 28, 56, 135].

Crockett [28] describes six application access patterns and suggests different distributions that match these patterns. He suggests that the best distribution patterns (given the access patterns) are striped or partitioned. Crockett also discusses having *specialized* parallel files, where there need not be a meaningful global view of the file (e.g., a sequential stream of bytes) since they will only be accessed in the context of a particular parallel application. These specialized files are convenient in that they provide the same functionality as having a large number of independent files, but without the management overhead.

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 [56]. An important consideration in the results is the effect of the on-disk cache, where mapping functions

[9] The authors argue that the application should provide the file system with "hints" disclosing the application access pattern rather than specific advice on what resources should be reserved for the application. Their argument is that only the system can make intelligent decisions based on global resource usage, and that the application can only aid this by providing more information. The format for specifying hints appears to be in an early stage in its development.

that result in good disk cache hit rates generally perform better. This same study also suggests the possibility of custom mappings for application-specific access patterns.

Womble *et al.* observe that programmers using multicomputers have to explicitly arrange for the data they access to be in local memory [135]. Given this, they suggest that it is natural for the I/O system to maintain the same view of local secondary storage, where each processor has its own logical disk. They assume that the file system is being used primarily as a staging area for actively used data stored in tertiary storage.

The Rama file system acts as a cache for tertiary storage [90]. Disk space is broken into "lines" of a few Megabytes, where each line has a descriptor identifying the file blocks it holds and the status of these blocks. The location of a file block in the system is determined by hashing on the *file id* and offset to determine the disk and line number on that disk where the data may be contained.

The file system for the nCUBE distributes file data across the disks in a simple striped fashion with the stripe size determined by the system [31, 32]. However, when an application opens a file it specifies a mapping between the data in the files and on the processors. On read requests, the file system permutes the data between the two, collecting the corresponding data from the different disks.

In terms of distribution, the most flexible among existing parallel file systems is the Vesta file system for the SP2 multicomputer [25, 26]. It has separate concepts for physical partitions (effectively the number of disks used) and logical partitions. Logical partitions allow applications to specify a mapping between file data and an application's view of the data. The mapping takes four parameters that specify the partitioning scheme (i.e., vertical group size, vertical interleave, horizontal group size, and horizontal interleave) and a parameter that specifies the position of the mapping in the partitioning scheme. With this approach, the application can specify a very wide variety of regular distribution strategies. For example, it can request that all I/O operations be directed to a single physical partition, striped across all physical partitions, or striped across some subset of the physical partitions.

All the distributions discussed above are static. It is also possible to support dynamic distributions that optimize for locality or for the load on the disks. For example, it is possible to have a distribution that directs paging I/O to disks near the memory being paged out. If data is repeatedly accessed by a particular processor, then the corresponding blocks of the file can be migrated to disks near the requesting processor. If the load on disks varies, then it may be a good idea to direct write operations to the least loaded disks, especially in a multiprogrammed environment. Replicating a file can be used both to get better performance [8, 120] and to manage locality (since the more local copy of a disk block can be used to satisfy a read request). To the best of our knowledge, while migration and replication have been studied for distributed systems, these dynamic schemes have not been studied in the context of a tightly coupled parallel machine.

2.5.4 File system architectures and implementations

In previous sections we outlined different proposed file system policies and interfaces. In this section we discuss related work in file system architectures and implementations.

In the last decade, most file system research has been directed towards developing file systems for loosely coupled networks of workstations [6, 22, 53, 98, 132]. A good overview of research in this area is given by Levy and Sibershatz [80]. Key issues on these systems are scalability, fault tolerance, preserving traditional uniprocessor semantics in a distributed environment, and achieving good performance through caching, replication, and migration of file data and meta-data.

There have been a number of file systems designed for large-scale multicomputers, including CFS [105] for the Intel iPSC, *sfs* [83] for the CM-5, Vesta [25, 26] for the SP2, the OSF/1 file system [136], the file system for the nCUBE [31], and the Rama file system [90]. To achieve scalability, most of the file systems incorporate to some extent the strategies of distributed file systems. For example, the Vesta file system hashes file names across all I/O nodes in order to balance the load on directory operations. A major difference between multicomputers and distributed systems is that there is a greater expectation that many processes running on different nodes will concurrently access the same file on multicomputers. To handle such tight coupling, the Vesta file system allows the application to specify that I/O should be RECKLESS, in which case the file system does not enforce the atomicity of `read/write` requests. The *sfs* file system differs from the other file systems in that it is designed to perform well only for a special purpose workload, namely SPMD applications where a single read or write operation causes massive amounts of data to be moved between all processes in the application and the disks.

The majority of file system work on shared-memory multiprocessors targets small scale systems. In particular, there have been a number of projects that introduced fine grain locks into existing file system code to allow for greater concurrency [60, 77, 82, 104].

The most innovative file system development in recent years is that of log-structured file systems [97, 110]. A log-structured file system treats the disk as a segmented append-only log. Write operations to the disk are coalesced and written to the end of the log in large I/O operations.[10] The log contains indexing information so that files can be read back with the same performance as other file systems. Because information is always written to a log, it is easy to support fast crash recovery. A log-structured file system requires that a *segment cleaner* run periodically to compresses the live information from fragmented segments.

The log-structured file system has been extended in a number of ways. For example, Burrows *et al.* describe a method for adding automatic compression that benefits from the large writes [13]. The sequential nature and large size of the writes also makes log-structured file systems attractive for automatic migration of data to tertiary storage [64]. Finally, the Zebra file system combines ideas from log-structured file system and RAID-5 to maintain the file data for a distributed system in a fault tolerant fashion [51].[11]

There has been a great deal of work in recent years to improve the flexibility of file systems in order to make it easier to add new features such as compression and encryption. Two examples that use an object-oriented approach are the Spring file system [61] and the Choices file system [15]. The iPcress file system by Staelin and Garcia-Molina [122] also uses (in a more limited fashion) an object-oriented approach, where each file is a *file object* that maintains statistics on the usage of the file to determine how data should be cached and how the file should be stored on disk. Stackable file systems are another approach for achieving the goal of flexibility, where the file system is composed of layer "building blocks" [52]. The layers can be developed by independent parties, each building on the functionality of the layers below it.

Finally, extent-based file systems, which structure files into sets of consecutive file blocks stored in contiguous disk blocks, and clustered file systems, that employ cylinder clustering techniques and always read large numbers of consecutive disk blocks, try to minimize the overhead of I/O by increasing the granularity of requests that go to disk. Before Sun's UFS file system was modified to adopt such a strategy, it took about half of a 12 MIPS CPU to get just half the disk bandwidth of a 1.5 MB/second disk [88].

[10] These large write operations are especially important if a RAID-5 disk array is being used, since these disk arrays perform poorly for small write operations.

[11] Multiple small files are combined together into a log that is distributed across a large number of disks. Parity information is calculated for every n blocks of the log and is also distributed across the disks.

Chapter 3

Architectural Overview

The goals of our file system architecture are the following:

Flexibility: It should be able to support a wide variety of file structures and file system policies.

Customizability: It should allow the application to specify the structure of the file and the policies that are to be invoked by the file system when the file is accessed.

Efficiency: The flexibility should come with little I/O or CPU overhead.

In this chapter we give an overview of the HURRICANE File System (HFS) architecture, describing the key features of this architecture that make the above goals possible.

The flexibility of HFS is achieved through of a novel, object-oriented building-block approach, where files are implemented by combining together a (possibly large) number of simple building blocks. The building blocks, called *storage objects*, each define a portion of a file's structure or implement a simple set of policies. The flexibility results both from the ability of the file system to support a large number of storage-object classes and from its ability to combine together storage objects of diverse classes.

HFS allows applications control over the storage objects used to implement their files. This allows an application to define the internal structure of its files, as well as the policies that should be invoked when the files are accessed. For a persistent file that is accessed by different applications over time, the storage objects used can be modified (in part) at run time to conform to specific application requirements.

Three features of HFS contribute to its low overhead. First, to minimize the overhead incurred by crossing address spaces, much of the functionality of the file system is implemented by an application-level library. Second, to minimize copying overhead, HFS is structured to facilitate the use of mapped-file I/O. Finally, the storage object interfaces that we have developed minimize copying overhead due to our building-block approach.

We designed HFS to support an unlimited set of file structures and policies. For example, file structures can be defined in HFS that optimize for small files, read-only files, or sparse files, and file structures can be defined to support files that span multiple disks or files replicated onto a number of disks. Policies that can be defined on a per-file (or in some cases on a per-open instance) basis include locking policies, prefetching policies, and cache management policies. In stark contrast, most existing file systems have been designed to support a single file structure and a small set of policies.

This chapter is organized as follows. Section 3.1 describes the object-oriented building-block approach. Section 3.2 describes the storage object interfaces we have developed. Section 3.3 describes the logical layers of the file system, provides examples of the storage objects defined at each layer, and discusses how this layering is designed to: 1) maximize the amount of the file system that can be implemented by an application-level library, and 2) facilitate mapped file I/O. Section 3.4 summarizes the main features of the HFS architecture.

3.1 Storage objects

HFS uses an object-oriented building-block approach, where files are implemented by combining together a (possibly large) number of simple building blocks called *storage objects*. Each storage object defines a portion of a file's structure

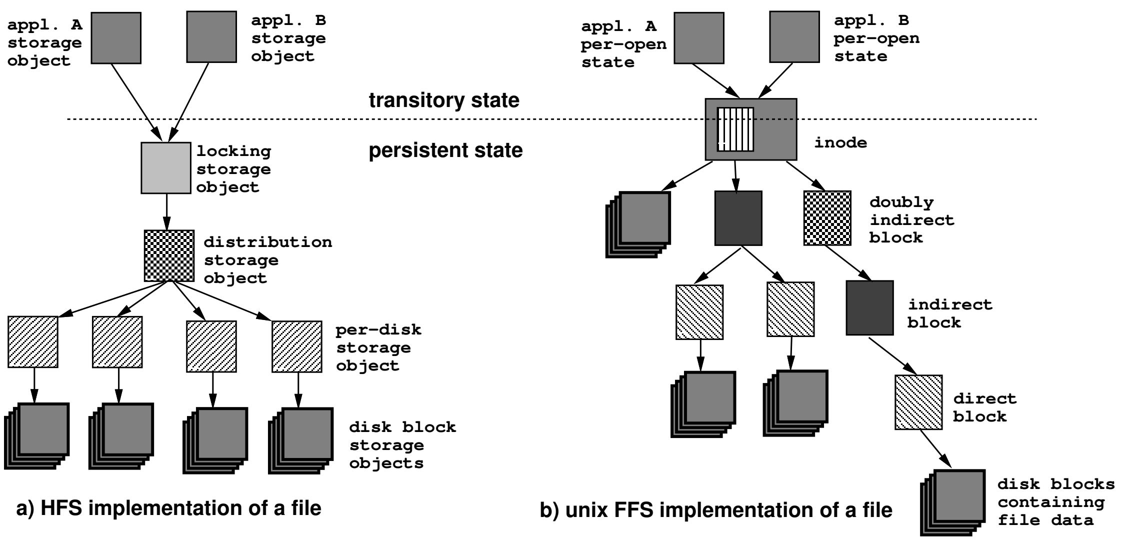

Figure 3.1: Unix and HFS implementation of an open file. The persistent state is state that is stored on disk. The transitory state is state that only exists while a file is being actively accessed.

or implements a simple set of policies. In an object-oriented fashion, each storage object encapsulates *member data* and *member functions* that operate on the member data. For example, the simplest class of storage objects are disk blocks, where the member data is the contents of the disk block and the member functions are `read` and `write`.

With the exception of disk blocks, storage objects contain, among other state, pointers to other storage objects. We refer to the targets of a storage object's pointers as *sub-objects* of the storage object. To allow storage objects to be combined together in diverse ways, storage object classes are designed so that the sub-objects of a storage object can belong to any of a large set of classes.[1]

We say that file data is stored by a storage object if that object handles requests to read and write that data. A storage object may handle these requests by forwarding them to its sub-objects, in which case the sub-objects also store all or part of the file data.

Storage objects belonging to different classes may: forward read and write requests to disk block sub-objects on a single disk, distribute file data across sub-objects spread across multiple disks, replicate file data to each sub-object, store file data with redundancy (for fault tolerance), implement a policy for directing prefetch requests to sub-objects, enforce security, manage locks, or interact with the memory manager to manage the cache of file data.

A simple example illustrates how a file is implemented using our object-oriented building-block approach. Consider the distributed file shown in Figure 3.1(a). (Distributed, here, means that the blocks of the file are distributed across multiple disks.) For each of two applications accessing the file, there is a separate per-open storage object that maintains the file offset. These per-open storage objects share a single sub-object that implements some locking policy. The sub-object of the locking storage object is a distribution storage object that implements a policy for distributing the file across multiple disks. The sub-objects of the distribution storage object are per-disk storage objects that implement a policy for distributing the file across disk block storage objects on a single disk.

In Figure 3.1(a), application "A" directs its requests for file data to the storage object specific to its open instance of the file. The per-open storage object translates stream requests (i.e., requests for the "next" portion of the file) into random-access requests that identify a file offset, and forwards each request to its locking sub-object. The locking storage object ensures that the data requested has been locked by the application, and forwards the request to its distribution sub-object. The distribution storage object determines the disk that should be used to service the request, and forwards the request to the corresponding per-disk sub-object. Finally, the per-disk storage object determines the

[1] We have defined only three base storage object classes from which all other storage object classes are derived, and designed most storage object classes so that the sub-objects belong to one of the three base classes. Hence, a sub-object can belong to any of the classes derived from the expected base class. This is described in more detail in the next section.

disk block storage object that stores the requested data, forwards the request to that object, and the disk block storage object handles the request.[2]

The ability to mix and match storage objects belonging to different classes is the key to the flexibility of HFS. For example, we can define a file that has the same distribution policy but a different locking policy to that shown in Figure 3.1(a), by using a different class for the locking storage object, while keeping the other storage object classes the same. Similarly, for a file that we don't want to distribute across multiple disks, the sub-object of the locking storage object can be a per-disk rather than a distribution storage object. The per-open storage objects used can be tuned to the characteristics of the application that has the file open. For example, if an application is accessing a file sequentially, a per-open storage object that prefetches data sequentially can be used.

Let us contrast our approach with that used by parallel file systems based on the Unix FFS. As shown in Figure 3.1(b), these file systems are similar to HFS in that files are implemented using building blocks, such as inodes, direct blocks, indirect blocks, and doubly indirect blocks. However, these building blocks differ from those used by HFS in that they are passive data structures rather than objects. This means that the way the FFS building blocks are accessed, and the way they are connected together, is fixed by the type of the file. As described in Section 2.3, it is difficult with such an approach to support more than a small number of file types, resulting in limited flexibility with respect to file structure and policy.

HFS storage objects can be either persistent or transitory, where persistent storage objects are stored on disk with the file data and transitory storage objects exist only when a file is being actively accessed. In Figure 3.1(a), the per-open storage objects are transitory and all other storage objects are persistent.[3] In the remainder of this section we describe the purpose of transitory and persistent storage objects and describe how they are instantiated.

3.1.1 Transitory storage objects

Transitory storage objects are used to implement all file system functionality that is not specific to the file structure and that does not have to be fixed at file creation time (say for security). For example, we have defined transitory storage objects that implement latency-hiding policies, compression/decompression policies, and advisory locking policies (for multi-threaded applications). Also, we have developed transitory storage objects that allow applications to have customized views of file data (as was discussed in Section 2.2.4) to simplify file access.

Transitory storage objects are specific to an open instance of a file[4], and we have found it useful to define a large number of transitory storage object classes so that the objects used by an application can be chosen to match the expected behaviour of the application. For example, transitory object classes are typically specific to the access mode (i.e., read, write, and read/write), the particular access pattern (e.g., sequential, or random), and the expected degree of concurrency.

It is important to be able to associate a set of policies with a file that are invoked by default when the file is opened, but still have the flexibility to modify these policies at run time to conform to application-specific requirements. Transitory storage object classes are an important part of our strategy for achieving this goal. Information associated with each file specifies a default set of transitory storage objects that are automatically instantiated by the file system when the file is opened. To conform to application-specific requirements, an application can request that new transitory storage objects be attached to a file at run time. These new objects either use the existing objects as sub-objects or replace existing transitory objects.

3.1.2 Persistent storage objects

Persistent storage objects are used to define the structure of a file, to store file data and other file system state on disk, and to implement policies that are either specific to that structure or, because of security or performance reasons, are fixed at file creation time. Examples of structure-specific *storage objects* include distribution storage objects that distribute file data across sub-objects, and replication storage objects that store a full replica of the file data in each sub-object. Examples of structure-specific *policies* include the policy used by a distribution storage object to distribute write requests and the policy used by the replication storage objects to distribute read requests. Persistent storage

[2] For simplicity, we have assumed in this example that the requested data is stored by a single disk block.

[3] Pointers to transitory storage objects differ from pointers to persistent storage objects. The former are memory pointers that directly reference the target object. The latter are handles that can be used to locate the object either on disk or in memory.

[4] While it would be possible to define transitory storage objects that are shared by multiple applications that have a file open, we have not found the need to do so.

objects used to authenticate open requests are an example of objects that implement file system policy that because of security cannot be changed at run time.

The persistent storage objects that are fixed at file creation time, such as those that determine the file structure, are instantiated (optionally under application control) in a bottom-up fashion when the file is created. For example, the per-disk storage objects of a distributed file are instantiated before the distribution storage object is instantiated; the parameters passed to the constructor[5] of the distribution storage object include references to its sub-objects — i.e., the per-disk storage objects. In some cases, persistent storage objects will instantiate other storage objects at run time. For example, the per-disk storage objects will instantiate disk block storage objects as they are required when the file grows.

The only persistent storage objects that are stored directly on disk are the disk block objects. All other persistent objects are stored by other objects, and ultimately stored on disk by disk block storage objects.[6] This means that the policies applied to store file data can be applied in the same way to store object member data such as file system meta-data. This allows us, for example, to take advantage of the disk head position when writing object data to disk in the same way this can be done when writing file data to disk. Also, as we will show later, the object member data of many objects can be stored by a single object, which allows the co-location of related objects on disk, reduces disk fragmentation, and allows object data to be efficiently stored redundantly for fault tolerance. In contrast, the location of the building blocks used by the FFS is fixed when they are created, and, except for inodes that are written to reserved portions of the disk, each building block occupies an entire disk block.

Unlike transitory storage objects, persistent storage objects associated with a file cannot be changed at run time. Therefore, to handle application-specific requirements, some persistent storage objects must implement a number of policies. The policy used can then be changed at run time. For example, some distribution storage objects implement multiple policies for distributing write requests to their sub-objects (e.g., to the least loaded disk, to the disk nearest the memory, etc.).

3.2 The Storage Object interfaces

If storage objects are to be combined in diverse ways, the object classes must share common interfaces. There are three classes from which all other storage-object classes are derived (one for each of the three layers of HFS described in the next section), and the virtual member functions of these three base classes define the common interfaces. As described in Section 2.1, even when an object is referenced as being an instance of a base class, calls to virtual member functions invoke the functions of the derived class and not the base class. Hence, a storage object can make calls to its sub-objects without knowing their specific class (as long as the sub-objects' class is derived from the expected base class).

A key feature of all three interfaces is that the *target* storage object chooses the buffer that is the source or destination of the I/O requests. On a read access, the storage object returns a pointer to a buffer containing the requested data, and on a write access, the object returns a pointer to a buffer into which the data should be placed. This allows objects to be combined together in a building-block fashion without requiring data to be copied every time a storage object makes requests to its sub-objects. This contrasts strongly with traditional read/write interfaces (e.g., stdio and Unix I/O) where the *requester* supplies a buffer into which or from which data must be copied on each request. In Chapter 9 we show that the cost of copying can be a major source of I/O overhead.

While having common interfaces is important, for extensibility it is also important that storage objects be able to export additional class specific functions, such as functions that query an object's state, change a policy used by an object, or *migrate* a file block from one sub-object to another. A problem with our building-block approach is how to allow the application to invoke a class-specific function when the target object is a sub-object of other storage objects. For example, consider a locking storage object that implements some locking policy and stores file data indirectly in a sub-object. The locking storage object does not directly support a `migrate` function, but its sub-object may support such a function. In this case, the locking storage object must be able to pass on to its sub-object all requests that the sub-object can handle, so that objects can be layered without restricting the requests that can be made to the lower level objects. In effect, we want a storage object to be able to inherit the extended interface of its sub-objects.

To allow objects to inherit the interface of their sub-objects, each of the three common interfaces provide a

[5] A constructor in C++ terminology is the function of a class that instantiates an object of the class.

[6] The object used to store a persistent storage object is specified as a parameter when it is instantiated.

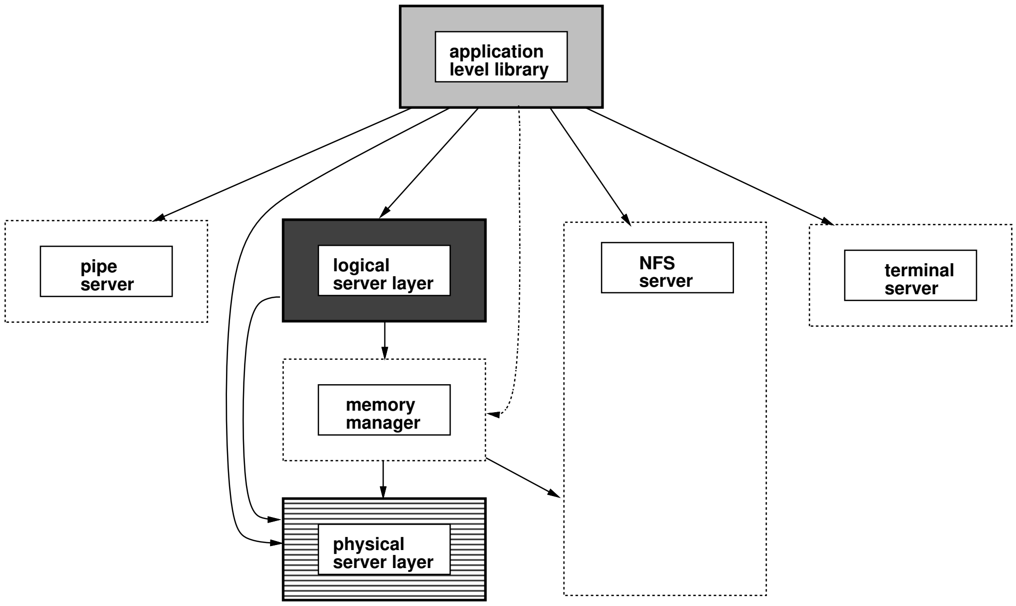

Figure 3.2: The HURRICANE File System architecture. Shaded boxes show components of HFS. Dotted boxes show other system servers with which HFS must cooperate. The application-level library, physical server layer and logical server layer form the three layers of the file system. Most file I/O occurs through mapped files, so the memory manager is involved in most requests to read or write file data. All explicit inter-layer requests are shown as solid lines. Page faults that occur when the application layer accesses a mapped-file region are shown as a dotted line.

"`special`" member function whose arguments identify the kind of operation requested and include a variable length buffer to hold any arguments specific to that operation.[7] In many cases, after determining that the operation is not one that can be handled directly, an object will forward a `special` request to a sub-object without trying to interpret it. This allows us to use new storage-object classes (that handle new types of requests) without modifying the object classes that may use objects of the new class as sub-objects. Only the application making the request and the object handling the request need to be able to interpret it.

3.3 File system layers

HFS is logically divided into three layers: an application-level library, the physical server layer, and the logical-server layer. (Figure 3.2).

The application-level library, in addition to being a layer of HFS, is the library for all other I/O servers provided by HURRICANE. It attempts to service application requests for I/O within the application-level wherever possible. If the target file is mapped, then the application-level library translates `read` and `write` requests into accesses to a mapped-file region. If the application-level library cannot service the request directly, then it forwards the request to a server that provides the required I/O service. These requests might be directed to a terminal, socket, HFS, NFS, or pipe server.

The physical server layer directs requests for file data to the disks. It maintains the mapping between file blocks and the disk blocks that store them. Consecutive blocks of a file may be spread over a number of disks.

The logical server layer implements all file system functionality that does not have to be implemented at the physical server layer and, for security or performance reasons, should not be implemented in the application-level

[7] The UCLA stackable layers interface adopts a very similar solution to the same problem, where their `default` function is equivalent to our `special` function [52]. Also, `special` is similar to `ioctl` provided by Unix, where it is used for exceptional requests, that are unique to particular types of storage objects, and under conditions where the cost to marshal the arguments into the buffer is not important.

library. For example, this layer provides naming (i.e., directories), authentication, and locking services. There are benefits in implementing naming in a system server, in that cached naming information can be shared by multiple applications. In order for locks on file data to be enforced (rather than just advisory) they must be implemented by a system server that can prevent applications from accessing data that has not been locked.

Each layer of HFS implements a different set of storage-object classes. In the remainder of this dissertation, we will differentiate between storage objects at each layer by the following naming convention: a storage object is called an *Application-level Storage Object* (ASO), a *Logical-level Storage Object* (LSO), or a *Physical-level Storage Object* (PSO), depending on whether it is implemented in the application, logical, or physical layer of the file system.

3.3.1 The physical server layer

The physical server layer is responsible for file structure and all policies related to disk block placement, including load balancing, locality management, and cylinder clustering.

Studies of parallel applications (described in Section 2.5.1) suggest that files are often accessed sequentially. To facilitate mapped-file I/O for this kind of access, we designed HFS with the physical server layer below the layer containing the memory manager. This allows an application to map a large portion of a file into its address space, even if the disk blocks are distributed across a number of disks. Also, an application can make a single request to the memory manager to prefetch or eject a large number of logically contiguous file pages from the file cache maintained by the memory manager. As will be shown in Chapter 9, this is important to amortize per-request overhead.

All *Physical layer Storage Objects* (PSOs) are persistent. They store file data and the data of other persistent storage objects — i.e., LSOs, and some PSOs. The simplest class of PSO are disk blocks. Besides disk blocks, there are three other types of PSO classes, namely: (i) *basic* PSO classes, (ii) *composite* PSO classes, and (iii) *read/write* PSO classes. Basic PSO classes define objects whose sub-objects are (largely) disk blocks contained on a single disk. Composite PSO classes define objects whose sub-objects are not disk blocks, but other PSOs that may be spread across a number of disks. Read/write PSO classes define objects whose data, for performance or security reasons, is accessed using a read/write rather than a mapped-file interface. We now briefly describe these classes in more detail.

Basic PSO classes

Basic PSOs provide a mapping between file blocks and the disk blocks on a particular disk. Each basic PSO class is optimized for a particular access pattern. For example, we provide a PSO class for sparse files; a PSO class for dense files that are mostly accessed sequentially; a PSO class for dense files that are mostly accessed in a random-access fashion. In contrast, the Unix FFS supports only one file type, suitable for all files and usages, but optimized for none of them.

There are three reasons why basic PSOs are restricted to refer to disk blocks on a single disk, as opposed to blocks distributed across multiple disks. First, it makes it easier to re-organize the blocks of each disk independently (e.g., for de-fragmentation). Second, it is necessary for the mechanisms for fast fault recovery used by HFS (as described in Chapter 8). Finally, it is necessary for the scalability mechanisms used by HFS (as described in Chapter 7).

Composite PSO classes

A composite PSO services read and write requests by directing them to appropriate sub-objects. For example, a file distributed across a number of disks can be implemented by using a distribution PSO whose sub-objects are basic PSOs that store the file data assigned to a particular disk.

Composite PSOs can be used in a hierarchical fashion, where a composite PSO can have sub-objects that are themselves composite PSOs. For example, a replicated file where each replica is distributed across a number of disks would be implemented using a replication PSO with two distribution PSO sub-objects.

The PSO classes we have implemented either provide a single static policy or provide a small number of dynamic policies for directing requests to sub-objects. A distribution PSO class that distributes consecutive file blocks in a round robin fashion across sub-objects is an example of class with a static policy. Our replication PSO class supports a number of policies that define which replica should be used on reads (i.e., the replica on the disks with the least load, the closest replica, and a particular replica) and the policy used can be changed at run time.

Read/write PSO classes

Read/write PSO classes are so named because the data they store can only be accessed by an application using a read/write interface rather than a mapped-file interface. An example of a read/write PSO class is the class used to store the data of small files. In this case, a read/write interface that copies a small amount of data can be less expensive than the cost of a page fault that is incurred if mapped-file I/O were used. Also, if a read/write interface rather than a mapped-file interface is used, it is not necessary to allocate a full page frame to hold the data of a small file.

The *fixed sized record store* PSO class is a read/write PSO class used when fixed sized records are stored in a file. If locks are to be enforced (by an LSO) at a record granularity, then the application cannot be given access to the file using mapped-file I/O because of its page-oriented nature. One of the major users of *fixed sized record store* PSOs is the file system itself. For most storage object classes, object member data consists of a small amount of data, and is stored as a record by a *fixed sized record store* PSO.

Read/write PSOs are similar to composite PSOs in that they contain pointers to sub-objects that store the file data. For example, a read/write PSO that stores fixed sized records can have as a sub-object a basic PSO (in which case all the file data is on a single disk) or a composite PSO (in which case the file data may be distributed across a number of disks).

PSO Instantiation

The structure of a file is defined when the file is created by instantiating a number of PSOs in a bottom-up fashion. For example, the basic PSOs of a distributed file are instantiated before the composite distribution PSO is instantiated; the parameters passed to the constructor of the distribution PSO include references to its sub-objects, i.e., the basic PSOs. Applications can control how the PSOs are instantiated, allowing sophisticated application programmers to construct complicated files optimized for their own purposes. However, for other (naive) users we provide an application-level library of functions that can be used to create most popular file structures, which hide the process of constructing the file from the application.

The storage objects that define the file structure are fixed at file creation time. However, if necessary, an application can change the distribution of a file manually by creating a new file with a different distribution, copying the file data between the two files, making a call to the physical server layer to swap the identities of the two files (i.e., the integer number that uniquely identifies the files to the system) and deleting the first file. We provide library routines to simplify this task.

Some PSOs are also instantiated as a file is being accessed. For example, basic PSOs will instantiate disk block PSOs as the file grows.

PSO member data must also be stored on disk. The only PSOs that are stored directly on disk are disk blocks. The PSOs from all other classes are themselves stored on disk by other PSOs, and ultimately stored on disk by disk block PSOs. Typically, a single PSO is used to store the member data of a large number of other PSOs. Whenever a PSO other than a disk block is instantiated, an argument specifies which other PSO will be used to store its member data.[8] It is interesting to compare our approach for storing meta-data with that of the Unix FFS. The disk block position of a FFS inode is fixed when the file is created and the position of all other FFS data structures (e.g., indirect blocks) is fixed when they are created.

The approach used by HFS has a number of advantages over that used by the FFS:

1. The disk blocks used for storage-object data are not fixed. Hence, it is possible to place object data near other related data. For example, the member data of a PSO can be placed on disk near the member data of its sub-objects. Also, by selecting an appropriate PSO to store it, it is simple to take advantage of the disk head position when writing object data to disk.

2. The disk space wasted for small files is minimized. The read/write PSO class used by HFS for small files stores the file data directly in the member data of the PSO, and any number of these objects can be packed into a single disk block, depending on the PSO used to store them. This means that HFS can use very large disk blocks to reduce management overhead and obtain improved performance for large files while not wasting disk space for small files.

[8] On each disk, there are a small number of PSOs whose location on disk is recorded in a well known location.

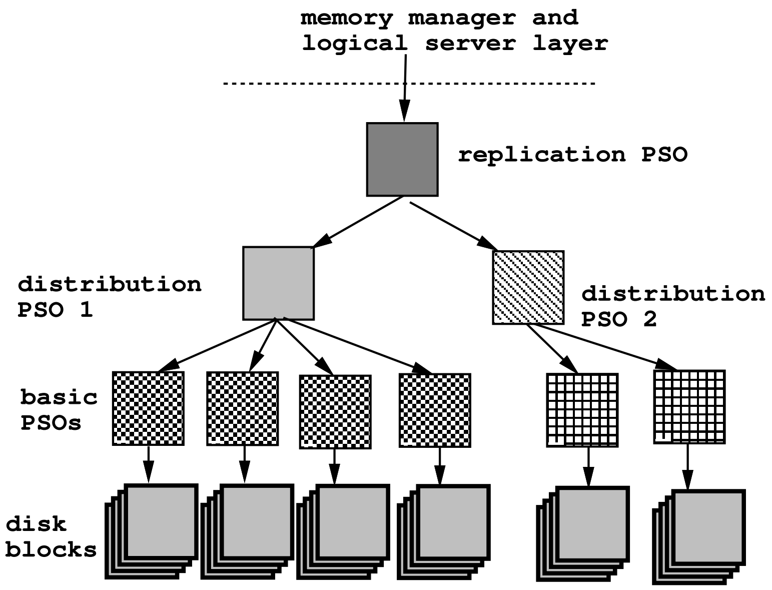

Figure 3.3: A replicated file where each replica is distributed across disks differently. The first replica is distributed across four disks and the second across two disks. Each replica is implemented using different distribution PSO classes and different basic (per-disk) classes. Hence, for each of the replicas a different policy is used both for distributing file blocks across the disks and for organizing file data on each disk.

3. PSO data can be stored redundantly for fault tolerance by selecting an appropriate PSO for storage. Since the member data of many PSOs is typically stored by a single PSO, the cost of maintaining this redundancy can be amortized over a large amount of data. This is important in the case of small files, like directories, where the cost of maintaining redundancy would otherwise be very high. This is discussed in more detail in Chapter 8.

An example

Figure 3.3 illustrates how simple PSOs can be combined together in a building-block fashion to create complex file structures. In this example, a replication PSO replicates data to two distribution PSOs, that in turn distribute data across a number of basic PSOs. The distribution and basic PSOs used for each replica belong to different classes, and hence implement different policies for distributing the data on and across the disks.

Such a file structure could be useful for a read-mostly file, where data is accessed in different ways at different times. For example, consider a file where the first replica uses a distribution PSO that stripes file data across a large number of disks with a small stripe size and basic PSOs that are optimized for random-access I/O. This replica would result in good performance for a multi-threaded application with threads that make many small random-access I/O requests, because interference that occurs when multiple threads access data on the same disks would be minimized. The second replica might use a distribution PSO that stripes file data across a small number of disks with a large stripe size and basic PSOs that allocate disk block extents of contiguous disk blocks on each disk. This replica would result in good performance when the file is accessed sequentially in its entirety (making effective use of the on-disk caches and large I/O requests).

With the type of file structure described in this example, the application (or application-level library) should explicitly select at run time the replica that is to be used for all subsequent requests so as to match the expected access pattern.[9] If multiple applications make conflicting requests to select a replica, then the last request will override all previous ones.

[9] The application specifies the replica to be used by making a `special` request to the replication PSO.

A general problem exhibited by this example is what to do when multiple applications make conflicting policy requests to a PSO. It is possible to concurrently support different policies for each application. Using this example, if two applications request that different replicas be used, the replication PSO could service each application's requests from the replica it selected. To support this functionality, we would have to make the application "ID" a parameter to the read/write member functions of our PSO classes, and the replication PSO would have to maintain policy related state for each application. We chose not to do this because the added complexity is quite large, and because we do not believe that it is common for multiple applications to concurrently access the same file in different ways. However, it is still an open question if this workload assumption is in fact true.

3.3.2 Logical server layer

The logical server layer implements all file system functionality that does not have to be implemented at the physical server layer and, for security or performance reasons, should not be implemented by the application-level library. LSO classes exist for naming, handling requests for file data, locking and authentication:

Naming LSO classes: Naming LSOs maintain the name space necessary to refer to and locate files; they are similar to Unix directories. All file systems cache naming information in memory in order to minimize the number of disk I/O operations required for name lookup. On large systems, file systems may replicate naming information to improve performance [80]. In HFS, the particular naming LSO class determines the policy for caching and replicating the naming information of each directory. Allowing these policies to be determined on a per-directory basis is important, since different naming objects have different types of access patterns. For example, "/bin" is primarily accessed read-only, suggesting replicated management, while "/tmp" and user home directories are modified frequently, and replication would hurt performance.

Access-specific LSO classes: An access-specific LSO class handles application requests to transfer data to or from a file. So far we have implemented only two such LSO classes, one which first authenticates read and write requests and then forwards them to the appropriate (read/write PSO) sub-object, and the other which first authenticates requests to map file regions into the application address space and then forwards them to the memory manager.

Locking LSO classes: These classes implement locking policies and ensure that the required locks have been acquired before forwarding file access requests to a sub-object. Different classes of this type may lock data at the granularity of the entire file, file block, memory manager page, or record. Also, they may implement class specific policies for acquiring locks, such as reader/writer and exclusive, and class-specific strategies for detecting and recovering from deadlock conditions.

Open authentication LSO classes: An open authentication LSO authenticates requests to open a file. Currently we have implemented only one such class which uses the same kind of permission information as Unix (i.e., read/write/execute for user/group/other). New LSO classes could provide alternative kinds of authentication, for example, using an access list.

LSO instantiation

All LSOs associated with a file are persistent and are instantiated when a file is created in a bottom-up fashion. The last persistent storage object instantiated when creating a file (i.e., the one for which all others are directly or indirectly sub-objects) must be an open authentication LSO. Once this LSO is instantiated, a `bind` call is made to a naming LSO to associate the file with some string name in the file system name space.

We chose to make all LSOs persistent, because it was simpler to do so in our implementation, and because we have not found the need to define any transitory LSO classes. However, if such a need does arise, then the changes required to support transitory LSO classes would be minimal.

An example

As an example of the kind of functionality provided in the logical server layer and of how this layer interacts with the other layers, consider a file composed of a large number of fixed sized records. A file of this type could be implemented as shown in Figure 3.4. On an open request, the logical server layer uses the pathname specified by the application to

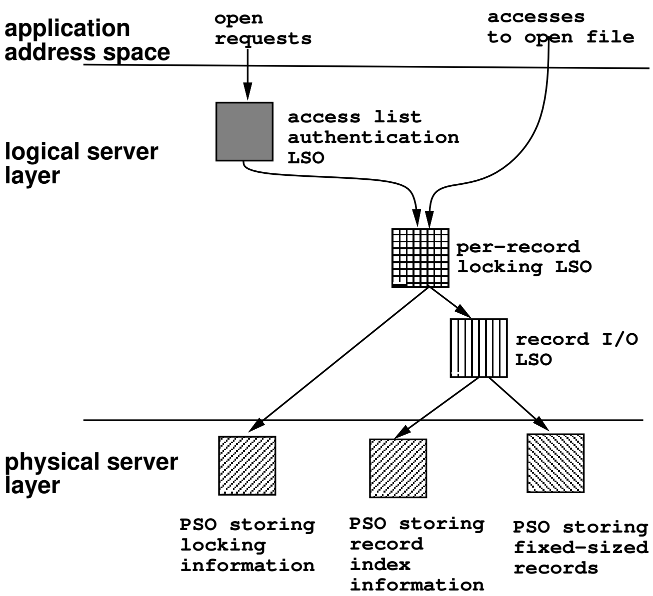

Figure 3.4: A file with an *open authentication* LSO that authenticates open requests, a *locking* LSO that handles per-record locking, and an *access specific* LSO that handles reads and writes.

locate the open authentication LSO via the naming LSOs that maintain the HURRICANE name space. The request to open the file is then directed to the LSO that implements an access list based policy for authenticating open requests. When the file is opened, this LSO returns to the application a handle to its sub-object, in this case an LSO that handles lock requests on a per-record basis. The locking LSO stores locking information in a PSO sub-object. After ensuring that the corresponding locks have been acquired, the locking LSO directs read and write requests to an access-specific LSO sub-object that understands how to handle unlocked read and write requests for a file composed of fixed sized records. This record I/O LSO stores indexing information in one PSO sub-object and the file data in another PSO sub-object. After determining which record is requested, the record I/O LSO forwards the request to the (read/write) fixed-sized record PSO. Finally, this PSO will respond to the requesting application.

3.3.3 The application-level library

One of the most original features of HFS is the amount of file system functionality implemented by the application-level library. The main advantage of having functionality provided by an application-level library, instead of by a system server, is that it improves performance by reducing the number of requests to system servers that incur the cost to cross address space boundaries. There are also a number of secondary advantages. For example, the interface to the library can be made to match the syntax and semantics of the application programming language. Moreover, some file system policies are specific to an application, and in our experience it is simpler and more efficient if these policies are implemented in the application address space. Finally, it allows an expert user to modify or add to the functionality provided by the library without requiring the privileges needed to modify a system server.

Our application-level library is called the *Alloc Stream Facility* (ASF). In addition to being the application-level library for the HFS file system, it is the library for all other I/O services provided on HURRICANE. Also, ASF has been ported (independent of the rest of the file system) to other operating systems, including *SunOS*, *IRIX*, *AIX*, and *HP-UX* [73, 74, 75].

The common interface provided by all *Application-level Storage Objects* (ASOs) is called the *Alloc Stream Interface* (ASI). While having the characteristics described in Section 3.2, ASI differs from the other storage-object interfaces (provided by PSOs and LSOs) in that it is a *uniform interface*. That is, it is an interface that can be used for any kind of I/O, including I/O not provided by HFS servers (e.g., pipes, terminals, network I/O) (see Section 2.2). Similar to

Unix I/O and stdio, the primary abstraction provided by ASI is a sequential byte stream, and we will often refer to an open file as a *stream*.

All ASO are transitory and are specific to an open instance of a file. We have defined ASO classes that handle requests for different types of I/O services, that implement file system policies, and that simplify the task of the programmer by exposing the semantic structure of a file to the application.

Service-specific ASO classes

The requests that come through the ASI interface will, if they cannot be handled at the application level, be passed to an I/O server. Because each I/O server may have a different interface, ASF must translate between ASI and the service-specific interface provided by the I/O servers. Exporting a uniform interface to the applications is important so that the programmers are able to write applications independent of the sources and sinks of their data. For good performance, on the other hand, we have found it is important that data be transferred to and from the application address space in a service-specific fashion. For example, large files are most efficiently accessed using mapped-file I/O, while terminal I/O is best provided using a read/write interface.

We have found it advantageous to have a large number of specific ASO classes, instead of a smaller number of general ones. At least one service-specific ASO class exists for each I/O service. Also, for each I/O service, a separate ASO class exists for read only, write only, and read/write access. For uni-directional streams less checking is necessary when read-only and write-only ASOs are used (rather than read/write ASOs), leading to improved performance.

Policy ASO classes

Some ASO classes are used to implement file system policies. We have defined ASO classes that prefetch, compress, decompress, and enforce advisory locks. Implementing policies at the application level rather than in a system server reduces the number of calls to system servers, and can have other secondary advantages. For example, having compression and decompression performed at the application-level may be useful, since it allows the pages in the file cache to be in a compressed format, possibly reducing the number of pages required for the file.

Our building-block approach makes it easy to add new policies to the library. Policy ASO classes are generally written to be independent of the stream type, that is, they direct all stream requests to a sub-object. For example, our locking ASO classes have no knowledge of how to interact with the file system server; instead they use a service-specific ASO sub-object to handle these interactions.[10]

A user can easily add new policy ASO classes to optimize for her applications. For example, it is easy to add new application specific compression algorithms, since the ASO classes that do the decompression need only be shared by applications accessing files in that format. That is, the user does not require the permissions needed to add the policy to a system wide library or server.

Matching application semantics

Many files have some underlying semantic structure. For example, the data in a file may be a two or three dimensional matrix of fixed sized elements. It is possible to define ASO classes that match a particular semantic structure. These objects may export additional methods that allow the application to directly access the structure stored in the file. For example, if a file contains a two dimensional matrix, an ASO could export member functions such as `read_row` and `read_column`, in addition to the standard ASI interface. Since the logical structure of the file data is understood by the ASO class, the task of invoking policies is also simplified, allowing the ASO in this example to prefetch data either by row or by column. Also, if the matrix is predominantly accessed either by row or by column, then an ASO that optimizes for that type of access behavior can be used, which would save the matrix in row major or column major fashion, whichever is more appropriate. As described in Chapter 2, the ELFS file system [49, 50] provides for similar functionality.

Special ASO classes can also be defined to provide for functionality like that of the Vesta file system [25, 26], where an application can request a particular "view" of a file. The "view" determines a mapping between the bytes in the file and the order that the application will see these bytes. This simplifies file access for those applications that access files in a regular, but non-sequential fashion.

[10] In addition to simplifying the policy ASO classes, having the policy ASO classes independent of the stream type makes it easier to port ASF to new platforms, since only the service specific ASOs have to change.

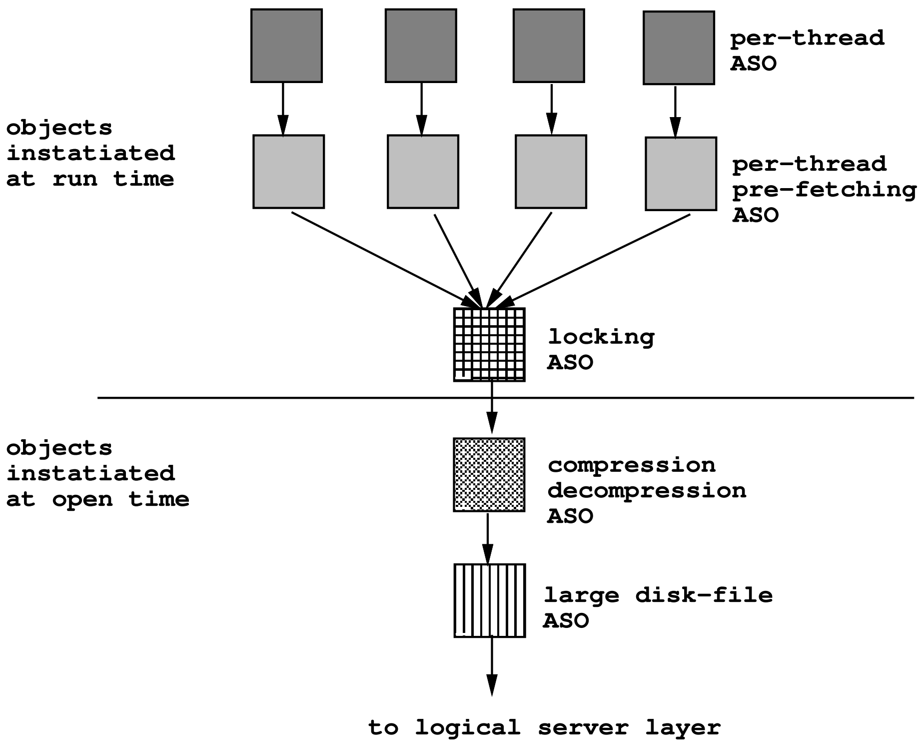

Figure 3.5: An example of ASOs used by a multi-threaded application to access a file stored on disk in a compressed format.

Instantiating ASOs

ASOs are transitory and specific to a open file instance, so that for every open file instance there is an independent set of file-specific ASOs. When a file is created, information specifying a default set of ASOs is associated with each file. ASF uses this information to automatically instantiate the ASOs when the file is opened. Having ASOs instantiated by default is important in order to provide functionality to the application transparently. For example, if compression/decompression ASOs are instantiated automatically, then the library can hide from the application the fact that compression is being done. This allows standard utilities, such as an editor, to process a compressed file without knowledge of this fact.

A stream's ASOs can be modified after a file has been opened.[11] It is important to be able to do this, since the applications that access a particular file, or even the phases of a single application, will have different access patterns and hence can benefit from access-specific ASOs. For example, an application can instantiate at run time a prefetching ASO specific to how it expects to access the data if one does not already exist. Or, if a prefetching ASO was automatically instantiated when the file was opened, the application can replace that ASO with another one, or remove it entirely.

We allow a single ASO to be the sub-object of more than one ASO. This allows the threads of a multi-threaded application to each have their own thread-specific ASOs, each sharing a common sub-object, so that only a single open request is made to the LSO, rather than one for each thread. The thread-specific ASOs would all share a common service-specific ASO, confining the overhead to instantiate the many objects to the application layer. As shown in Figure 3.5, the shared sub-object of the per-thread ASOs can be a locking ASO to ensure that multiple threads do not access the same file data concurrently.

[11] This is in contrast to the PSOs and LSOs that are fixed at file creation time.

An example

Figure 3.5 illustrates how ASOs are can be used in a building-block fashion. It shows four threads of an application, each sequentially accessing a different part of a single file. The file is stored on disk in a compressed format. Each application thread has a per-thread ASO that maintains the file offset for that thread. Since each thread accesses data sequentially, but the file as a whole is not being accessed sequentially, each per-thread ASO has an independent prefetching sub-object.[12] The prefetching ASOs have a common sub-object that implements some policy for (advisory) locks. The sub-object of the locking ASO decompresses data read from the file and compresses data being written to the file. Finally, the compression/decompression ASO makes requests to access file blocks to a service-specific ASO optimized for large files.

The service-specific and the compression/decompression ASOs are instantiated automatically by ASF when the file is opened. Hence, the application need not be aware that the file is stored in a compressed format. On the other hand, the number of threads and the way threads access the file are specific to the application, so the thread-specific ASOs and the locking ASO are instantiated by the application at run time.

Interface modules

ASF was designed so that (i) it can efficiently support multiple I/O interfaces, so that (ii) it can be easily extended to support new I/O interfaces, and so that (iii) an application can intermix requests to the different I/O interfaces. Each interface is implemented by ASF in an *interface module*, which translates between application requests to that interface and the ASI interface provided by the ASOs. Four features of ASI facilitate the implementation of I/O interfaces using it:

1. ASI is implemented by ASOs in the application address space rather than by a system server; hence, there is no system-call overhead to make an ASI request.
2. ASI is designed to avoid copying by having the target ASO provide the buffer for an I/O request,
3. ASI supports random-access requests, where the file offset for a request is specified by a parameter to the request.
4. ASI supports stream requests, where the file offset for a request is determined by the request that preceded it.

These features make it possible to implement I/O interfaces using ASI very efficiently. In Chapter 9 we show that the *Unix I/O* and stdio interfaces provided by ASF interface modules perform better on some Unix platforms than the native implementations on these systems.

New I/O interfaces can be easily added to ASF. This is important because the existing parallel I/O interfaces are immature and are likely to change in the future. Since ASI supports both random-access and stream requests, it is simple to translate from other interfaces to ASI. In our limited experience, the code required to implement ASF interface modules is trivial, especially in comparison with traditional implementations of I/O interfaces above the Unix I/O system-call interface. The key reason for this is that buffer management in ASF is handled by ASOs, above which the interface modules are built, and not in the interface modules. In contrast, with traditional implementations of stdio more of the code is related to buffer management than to the stdio interface.

Requests to different interfaces provided by ASF can be intermixed because the interface modules do not buffer data; only the ASOs that are shared by the interface modules buffer data. For example, the application can use stdio `fread` to read the first ten bytes of a file and then Unix I/O `read` to read the next five bytes. This allows an application to use a library implemented with, for example, stdio even if the rest of the application was written to use Unix I/O, improving code reusability. More importantly, it also allows the application programmer to exploit the performance advantages of ASI by rewriting just the I/O intensive parts of the application to use that interface. Because requests to the different interfaces can be intermixed, ASI appears to the programmer as an extension to the other interfaces.

[12] We have defined a prefetching ASO class that fetches data sequentially in a bounded region, ensuring that the prefetching ASO will not request (and lock) more data than the thread will use.

3.4 Summary of HFS architecture

In this chapter we provided an overview of the HURRICANE file system architecture. In particular, we have described (i) the object-oriented building-block approach used by the file system to represent files, (ii) the main features of the storage-object interfaces, and (iii) the functionality provided by the three logical layers of our file system.

There are three main goals for HFS:

1. to support a wide variety of file structures, file system policies, and I/O interfaces,
2. to allow applications to actively cooperate with the file to optimize I/O performance, and
3. to provide these structures, policies and interfaces at low processing and I/O overhead.

These goals are important in making it possible for a wide variety of applications to exploit most of the I/O bandwidth of a parallel supercomputer. In this section, we review the main features of the HFS architecture, and discuss how they contribute towards meeting these three goals.

3.4.1 Flexibility in file structure, file system policy, and interface

The object-oriented building-block approach we use provides much of the needed flexibility. Files are implemented by combining together a potentially large number of simple storage objects. Because each storage object defines its own member functions and because the object classes at each layer of the system have common interfaces, we are able to combine together objects of different classes in a multitude of ways. This allows us to support many different file structures and file system policies.

The Alloc Stream Interface provided by ASOs is designed so that other interfaces can be easily and efficiently implemented using it. This allows HFS to be extended to support new I/O interfaces. Also, the design allows applications to intermix requests to the different I/O interfaces with predictable results. This is important, because it allows each part of an application to use a different I/O interface, which means that a programmer can take advantage of a new I/O interface by modifying just the I/O intensive part of her application. Also, it increases code reusability, because an application can be linked to libraries that use a different I/O interface.

Another aspect of the storage-object interfaces that contributes to the flexibility of HFS is the fact that an object's member functions can be invoked by an application even if the function is not part of the common interface and the object is not directly referred to by the application. This allows us to define new storage-object classes that handle new types of requests without having to modify all object classes that directly or indirectly may have an object of the new class as a sub-object.

HFS treats file system meta-data in the same way that it treats file data, where a single PSO can be used to store the member data of a large number of other storage objects. This has the advantage that the policies used for managing file data can also be used for object member data. It allows HFS to, for example, co-locate related objects on the disk, reduce disk fragmentation, and store object data with redundancy (for fault tolerance) in an efficient manner. To our knowledge, no other existing file system allows this much flexibility with respect to the file system meta-data.

The HFS application-level library design has advantages that extend beyond HFS and our operating systems. ASF can easily be ported to new platforms; only the set of service-specific ASO classes must be changed. We have ported ASF to *SunOS* [63], *IRIX* [118], *AIX* [91], and *HP-UX* [133], where it runs independent of any server or special kernel support. Also, ASF can easily be extended to support new types of I/O services that a particular platform might provide. Because the ASO classes can be optimized to take advantage of operating system or hardware specific features (such as different variants of mapped files, and different types of communication facilities), ASF in many cases performs better than the native facilities of these platforms, as will be shown in Chapter 9.

3.4.2 Application–file system cooperation

We believe that the application, compiler, and file system must all cooperate to optimize the I/O performance of a parallel supercomputer. For the file structure to be tuned to the application(s) that will access it, the file structure must be specified by the creating application, since it is difficult to modify file structure at run time. Also, we believe that for many applications the best choice of file system policy is specific to the application and cannot (at reasonable overhead) be extracted by analysis of the I/O requests made by the application.

When creating a HFS file, the application can specify (i) which persistent storage objects, i.e., PSOs and LSOs, should be used to store the file data, (ii) which persistent storage objects should be used to store the meta-data, and (iii) which transitory storage objects, i.e., ASOs, should be instantiated when a file is opened. The application can therefore request any file structure or policy that can be composed from the storage objects defined by the file system. To maximize this flexibility, we have developed a large set of simple object classes rather than a smaller set of more complex ones. Also, expert users can add new policies to the file system by adding new ASO classes.

For a persistent file that is accessed by different applications over time, HFS provides two mechanisms that allow the default policies associated with the file to be modified to conform to a specific application's requirements. First, the set of ASOs used for an open file can be changed at run time by adding, removing, or replacing ASOs. Second, some PSOs accept requests to modify their policy at run time.

3.4.3 Low overhead

It seems intuitive that accesses to a file constructed from a large number of storage objects will incur some extra overhead. Our experience, however, indicates that our approach typically results in less overhead than the more traditional approaches. This low overhead is in part a result of the storage-object interfaces we developed, which do not require that data be copied as control passes from one storage object to another. The low overhead is also a result of the large number of storage objects that are provided, allowing the application to use storage objects that are tuned for its exact requirements. This makes it possible to avoid executing conditional statements, for example, to determine which policy to use, and to optimize the storage-object member data so that it can be represented compactly and effectively cached in main memory. Should the use of a large number of storage objects result in poor performance for some important file type, then it is always possible to add a new storage-object class to HFS to specifically handle that file type.

The layered structure of HFS, with the physical server layer below the memory manager layer, also contributes to a reduction of overhead, in that it allows mapped-file I/O to be used efficiently for many parallel applications. As discussed in the previous chapter, mapped-file I/O is an important way to reduce I/O overhead. Studies of parallel applications (described in Section 2.5.1) suggest that files are often accessed sequentially. Because the physical server layer that implements logical to physical mappings is below the memory manager, an application can map a large portion of a file into its address space, even if the disk blocks are distributed across a number of disks. Also, an application can make a single request to the memory manager to prefetch or eject a large number of logically contiguous file pages from the file cache maintained by the memory manager. As shown in Chapter 9, this is important in order to amortize per-request overhead.

Finally, implementing as much of the file system as possible at the application level (instead of in a system server) also reduces file system overhead, because it reduces the number of times it is necessary to make calls to system servers and hence cross address space boundaries.

The following chapter describes key implementation details that affect the overhead of HFS requests that cross file system layers. Succeeding chapters describe the internal implementation of the application and physical layers of HFS, the two layers that have the greatest impact on performance. We show that our object-oriented building-block approach can in fact be implemented so that the cost of implementing a file from a large number of objects can be low.

Chapter 4

HFS Servers

In the previous chapter we described the architecture of the HURRICANE File System (HFS). We have implemented substantial portions of this file system as a part of the HURRICANE operating system running on the HECTOR shared-memory multiprocessor. This implementation is a proof of concept prototype that demonstrates that the HFS architecture can be implemented on a real system and that a file system based on this architecture can deliver high performance I/O.

This and the next two chapters discuss the key HFS implementation issues. This chapter provides an overview of the servers that implement the three architectural layers, and discusses how these servers interact. The next two chapters describe in detail implementation issues related to the physical-server layer and the application level library layers of HFS.

4.1 Overview

Our implementation makes use of three user-level (i.e., not in the kernel) system servers. 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, it determines which disk an I/O request should target, and directs the request to a 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* (ASF) is a file system library in the application address space. For many disk files, the ASF maps files into the application's address space and translates `read` and `write` operations into accesses to the mapped regions. The servers, library, and their interactions are depicted in Figure 4.1. The BFS is described in detail in Chapter 5, and the ASF is described in detail in Chapter 6.

The BFS maintains *Physical-level Storage Objects* (PSOs), and is our implementation of the HFS physical server layer. The Name server maintains Naming *Logical-level Storage Objects* (LSOs), and the OFS maintains all other LSOs. Together they implement the logical server layer of HFS. The ASF implements the application level (library) layer of HFS. For each application, the ASF maintains the *Application-level Storage Objects* (ASOs) for the files accessed by that application.

Ninety percent of the code in all layers of HFS is specific to storage object classes. The remaining ten percent is glue, primarily to locate objects, to manage shared resources such as caches of object member data, and to handle interactions between the different servers. Each request to an HFS server from the library, memory manager, or another server, contains an argument that identifies the target storage object that will handle the request. In our implementation, we use three kinds of storage object identifiers:

tokens: A token is a number that uniquely identifies a persistent storage object (i.e., a LSO or PSO) to its server. Tokens are used internal to each server and are exported to trusted servers. For example, when the memory manager makes a read request to the BFS, it identifies the target file by using the token of the top level PSO. Similarly, LSOs and PSOs reference their sub-objects using tokens.

file handles: A file handle is a capability that gives an untrusted client access to a file. For example, when a file is opened, the OFS returns to the library a file handle that (indirectly) identifies the LSO to which subsequent requests will be made.

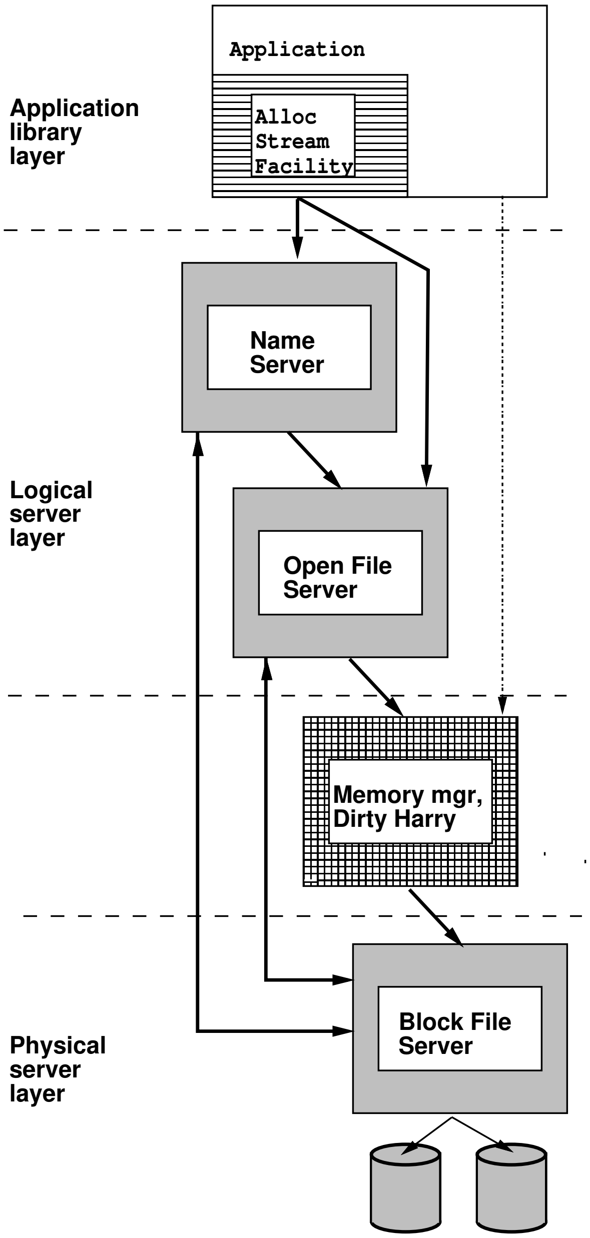

Figure 4.1: The HURRICANE File System. The Name Server, Open File Server and Block File Server are user level file system servers. The Alloc Stream facility is a file system library in the application address space. The Memory Manager and Dirty Harry Layer are part of the HURRICANE micro-kernel. Solid lines indicate cross-server requests. Dotted lines indicate page faults.

pointers: Within the application level, ASOs are identified directly using memory pointers. For example, when an application makes a read request to an open file, it identifies the target (open) file using a pointer to the top level ASO. ASOs reference ASO sub-objects using pointers.

When an application opens a file, the server allocates a per-open data structure that contains the token of a persistent storage object and any state needed to authenticate requests to the open file[1], and returns to the application a handle that can be used to locate this data structure. On each subsequent request to the open file, the server uses the handle to locate the per-open data structure, obtains from the data structure the token of the target persistent storage object, and passes to the target object both the arguments of the request and a pointer to the per-open data structure that contains the information needed for the persistent object to authenticate the request. We place both the token of the object and the authentication state in the per-open data structure, even though the authentication state is interpreted by the object, so that all per-open state can be localized to a single data structure.[2]

A simple example illustrates the interactions between ASF and the system servers. When an application opens a file, the open request is sent by the library to the name server. The name server translates the character string name of the file into the token for the associated *open authentication* LSO, and forwards the request to the OFS. The OFS uses the token to locate the LSO (reading it in through the BFS if it is not cached in memory), and invokes that LSO's `open` member function. The LSO determines whether the application is allowed to open the file, and if so, obtains the token of its LSO sub-object (e.g., an *access-specific* LSO) to which subsequent requests will be made, makes a request to that LSO to obtain authentication state for subsequent requests, and allocates a per-open data structure in which it stores the token and the authentication state. The OFS then returns to the ASF a file handle that identifies the per-open data structure. Finally, the ASF obtains from the OFS information identifying the ASOs that should be instantiated, instantiates them, and returns to the application a pointer to the highest-level ASO instantiated.

Let us assume a file is being accessed for which mapped-file I/O is to be used and that the data being accessed is not currently mapped in the application address space. When the application makes a read or write request to an open file, the *service-specific* ASO will make a request to the OFS to map the file into the application address space. The OFS invokes the `map region` member function of the corresponding LSO, located using the file handle. The LSO uses the state in the per-open data structure to ascertain that the access is valid and forwards the request to the memory manager, identifying the target file by using the token of the top level PSO (i.e., its sub-object). Once a file (or portion thereof) has been mapped into the application address space, the library (more specifically, the *service-specific* ASO), 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, then the memory manager, as part of handling the page fault, will modify 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 `read_page` request to the BFS. In this case, the BFS invokes the corresponding member function of the PSO identified by the token argument of the request. This function determines which disk contains the requested block and initiates the operation to the corresponding device driver. Control is returned to the faulting process when the I/O operation to the disk has completed and the memory manager has updated the application's page tables. If a mapped page is modified by the application, then Dirty Harry will eventually make a `write_page` request to the BFS (at the latest when the memory manager wishes to reassign that physical page frame).

4.2 Implementation details

We have implemented HFS as a part of the HURRICANE operating system running on the HECTOR multiprocessor. We placed the name server, OFS, and BFS in the same address space for convenience. Since our research has focused on the application level library and the physical server layer, we minimized our efforts in implementing the name server and OFS, so both of these servers are quite basic.

[1] We also store in the per-open data structure additional state used for garbage collecting these data structures for the case when the applications do not properly close the file.

[2] By placing both the token and authentication state in the per-open data structure, and by passing a pointer to this data structure to the persistent object, we allow the persistent object to modify the token field of the per-open data structure so that subsequent requests will be directed to some other persistent storage object. This is used by the LSO that maintains the state needed to instantiate the per-open ASOs. After handling a request to obtain the ASO instantiation information, the LSO changes the token in the per-open data structure so that all subsequent requests for file data will be directed to the LSO that services these requests.

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

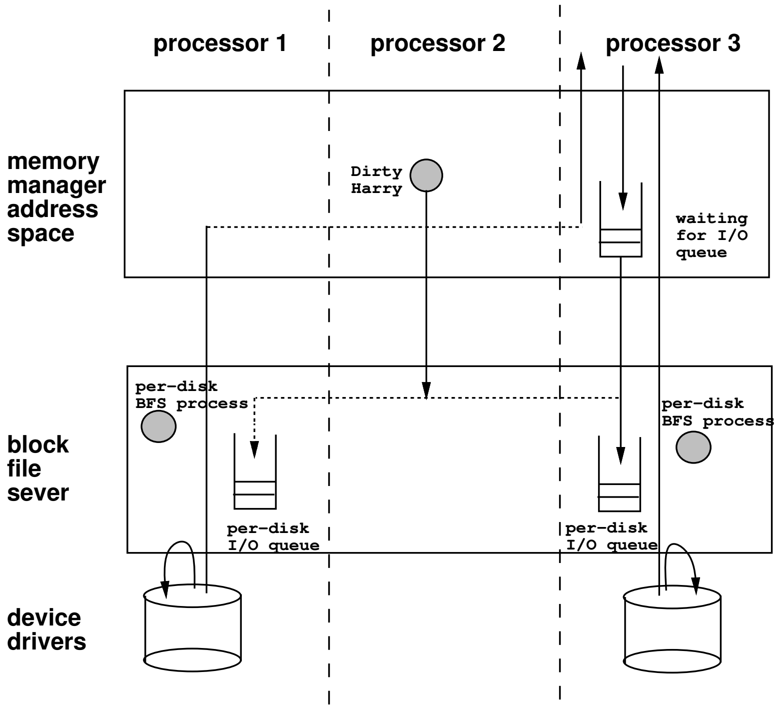

Figure 4.2: Interactions between the memory manager, block file server, and device driver. All interactions that cross address spaces are PPC operations. Dotted lines are used to indicate remote accesses to shared data structures. Solid lines indicate page faults and page fault completions.

All of the explicit cross-address-space communication is performed using HURRICANE PPC operations [45]. We chose to use PPC operations because they are very fast (on the order of 30 μsec), they have semantics similar to that of procedure calls, and they require no locks in the micro-kernel. PPC requests are executed on the local processor, which allows for efficient implementation (bypassing the scheduler and avoiding locking) and makes it possible to take advantage of temporal locality when a process repeatedly makes requests to a server. On each PPC request, a worker process is created in the server address space to service the request, so the number of workers handling requests in the server address space is always equal to the number of client requests it is servicing. This also simplifies deadlock avoidance, since different servers can make PPC requests to each other without fear of running out of workers that can satisfy the requests. For example, the BFS and memory manager can both make requests to each other.[4] A PPC variant, called anonymous PPC, is used when the PPC is invoked by a micro-kernel page-fault or interrupt routine (i.e., when it is not a HURRICANE process invoking the request). In this case, the invoker is not blocked and any response to the PPC is discarded.

In the prototype implementation of HECTOR, all disks are connected to specific processors. The disks can DMA to or from any processor in the system, but disk requests can only be initiated from the local processor[5], and all interrupts from the disk are received by the local processor.

A *per-disk BFS process* executes on each processor to which a disk is attached. Except when initiating a new request to the disk, the per-disk BFS process is blocked. In our implementation, the per-disk BFS process needs to be awakened only if the disk is idle when a new request arrives.

Figure 4.2 depicts the interactions between the memory manager, BFS, and device driver. The memory manager interacts with the BFS either to read data (as a result of a page fault or a prefetch operation) or through Dirty Harry to write data. When servicing a page fault, the memory manager places the faulting process's descriptor into a *waiting*

[4] The BFS makes requests to the memory manager to, for example, allocate physical memory for growing the cache of file system meta-data.

[5] Only the processor local to the disk can access the registers of the disk controller.

for I/O queue and translates the page fault into an anonymous PPC to the BFS. The BFS, by invoking the `read_page` method of the corresponding PSO, determines the disk and disk block to be used to service the request, and enqueues the operation on the per-disk I/O queue (using shared memory if the disk is remote). If the disk is idle, the per-disk BFS process is awakened to initiate the disk request.

When an interrupt occurs, indicating that a disk operation has completed, the device driver preempts the currently running process and does an anonymous PPC to the BFS. The BFS determines whether additional requests for the disk are pending, and if there are, initiates a new request to the disk. It then informs the memory manager that one of its requests has completed so that the memory manager can unblock the process waiting for the I/O.

To write out a page, Dirty Harry unmaps it from the address spaces that reference it and issues a `write_page` request to the BFS. After determining the target disk and disk block, the BFS initiates the request in the same way as for a read fault and then replies to Dirty Harry. Dirty Harry is only blocked for the amount of time taken to enqueue the disk request; hence, it can have multiple requests outstanding at a time. As an optimization, the memory manager can request that the BFS remove a `write_page` request from the per-disk I/O queue to allow a process faulting on that page to proceed without blocking.

The interaction between the memory manager, BFS and device driver is optimized for the case where disks are busy. In that case, it is not necessary to wake up the per-disk BFS process to initiate a new request to disk, because the request will be serviced when a previous request is completed. To lower the turn around time between the completion of a disk request and the initiation of the next request, the new request is issued to the disk even before the memory manager is informed of the completion of the previous request.

The time between an interrupt and the initiation of the next disk request could be further reduced by maintaining the queue of outstanding requests at the device driver instead of at the BFS. We may do this in the future, but have so far not done so because keeping the list of waiting requests in the BFS makes it easier: 1) for the memory manager to reclaim pages that are being actively accessed, and 2) to take into account the priority of the requests (e.g., for fault recovery some write operations must complete before other write operations) when scheduling operations to the disk.

Chapter 5

Physical-server layer implementation: The BFS

This chapter describes the Block File Server (BFS) that implements the physical server layer of the HURRICANE file system. Section 5.1 describes the per-disk data structures. Section 5.2 describes the common interface exported by *Physical Server Object* (PSO) classes. The key feature of this interface is that it minimizes copying overhead. The PSO classes that have been developed are described in Section 5.3, and examples of the data distributions that can be achieved using these PSOs is presented in Section 5.4. Section 5.5 describes how the application-level library interacts with the BFS to instantiate PSOs. Finally, Section 5.6 describes some additional implementation details of the BFS.

5.1 Per-disk data structures

Figure 5.1 depicts the per-disk data structures maintained by the BFS. The primary purpose of the *superblock* is to indicate the position of all other per-disk data structures on disk. It also contains general information describing the disk such as the disk block size and the number of disk blocks. The superblock is the only per-disk data structure whose location on the disk is fixed.

The *block map* identifies which disk blocks are free and which are being used. It is organized as a simple vector of bits, where bit i indicates whether block i on that disk has been allocated. We chose a bit map because simple masks can be used to allocate or deallocate groups of contiguous blocks, which allows a file system to employ cylinder clustering techniques to reduce the number of seek operations required.

Each PSO has an entry in the *object map*. The entry identifies the class of the object and the location of the object's

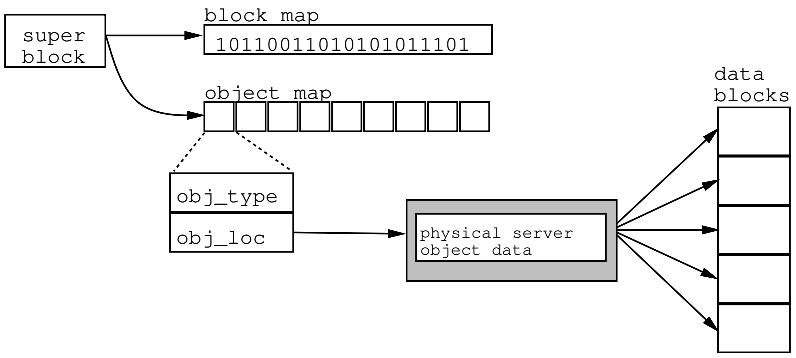

Figure 5.1: Per-disk data structures.

member data on the disk. In our current implementation, the class of the PSO is specified by the 8 bit `obj_type` field. The location of the PSO's member data is specified by the 24 bit `obj_loc` field. The interpretation of the object location is specific to the PSO class. For some PSO classes, the `obj_loc` field identifies a disk block directly, while for other PSO classes it identifies the PSO that stores the member data.

Each PSO is identified to the BFS by a *token*. The token contains up to three values: 1) an *omap* number that identifies an *object map*, 2) an *object number* which is used as an index into the *object map*, and 3) a `obj_type` field, which, if it exists, identifies the class of the PSO. PSOs reference their sub-objects using tokens that contain a type field, so that a type lookup becomes unnecessary when a PSO makes requests to its sub-objects. However, tokens exported by the BFS do not include the type field, so that it is possible to change the object type without affecting other servers.

It is necessary for some PSOs to identify on which disks their sub-objects are stored so that load balancing and locality management become possible. The *omap* field of the token, which identifies the per-disk object map (and hence implicitly a disk), is used for this purpose. In the case of basic per-disk PSOs, all data stored by the PSO is on the disk identified by the *omap*. For the other PSOs, the *omap* number identifies one of the disks that stores the data, and is used as a hint for locality management.

Figure 5.1 is a logical representation of the actual per-disk data structures. In reality, the object and block maps are spread over a number of disk blocks. The superblock contains two arrays of disk block pointers, one for the object map blocks and one for the block map blocks. The number of elements in these arrays is determined when the disk is formatted.

The object map provides HFS with a level of indirection. The token identifies an object map entry and the object map entry identifies the location of the PSO data. This indirection has several advantages over the Unix FFS, where the inode number determines the disk location of the inode. First, PSO member data can be located anywhere on disk, and the location can be changed over time, for example, to take advantage of the current disk head position when writing the PSO to disk. Second, there is no need to reserve disk space for PSOs that do not exist, and hence: 1) space is not wasted in the case of a small number of very large files, and 2) there is no pre-set limit to the number of small files.[1] Finally, as will be described in Chapter 8, the object map allows a low cost fault recovery mechanism to be incorporated into the file system.

The main disadvantage of the object map is that extra disk operations will be required to read and write object map data. However, we expect accesses to the object map to only rarely result in disk operations, since each object map entry is very small (a single 32 bit word) and therefore can effectively be cached in main memory. Also, the location of each block of the object map is determined by the superblock, and is not fixed on disk, so HFS can take into account the current disk head position in order to minimize the cost of writing out object map blocks.

5.2 The PSO interface

All PSO classes are derived from a single class whose virtual functions defines the common PSO interface. The key operations of this interface are shown (in slightly simplified form) in Figure 5.2.

`Read_page` and `write_page` are used to read or write file data to or from physical page frames when data stored by a PSO is accessed using mapped-file I/O. The arguments to `read_page` and `write_page` include a file offset that identifies the target data, a physical address that identifies the physical page frames, a length which specifies the number of pages being requested, and information returned to the memory manager to identify the request when the I/O request has completed.

The `acquire` and `release` member functions are used by the BFS to access the data stored by PSOs when mapped-file I/O is not being used. The `acquire` operations obtain a read or write lock on requested data and return a pointer to a buffer containing the data. If the data is not available in memory, an `acquire` operation will read the data from disk. The `release` operations release locks held on previously acquired data.

An important feature of both the read/write and acquire/release operations is that data is not copied as control passes through multiple layers of objects. This is important so that a file can be constructed with many object layers without excessive overhead.

[1] The Unix FFS inodes consume 6.25% of the disk space, whether these inodes are used or not. HFS uses almost no disk space for files that do not exist (1 word for every 32000 possible files).

read_page(token, off, paddr, len, mminfo) : read into physical page frames
write_page(token, off, paddr, len, mminfo) : write from physical page frames
acquire_read(token, off, len) : acquire read access to data
release_read(token, off) : release read access
acquire_write(token, off, len) : acquire write access to data
release_write(token, off) : release write access
flush(token) : flush state from meta-data cache
freeBlocks(token, from, to) : free storage in specified range
special(token, op, buf) : execute special function

Figure 5.2: The minimal common interface for PSO classes.

Some PSO classes extend the interface inherited from the base class with operations specific to the policies or structures they implement. For example, most PSO classes provide additional functions that return the type of the object and the PSO member data, allowing applications to discover how a file is structured. Also, some PSO classes provide functions that allow the application to set the policy to be used for subsequent read and write requests.

The requests a PSO can service depends not only on its class, but also on its sub-objects. For example, if a PSO's class does not directly support a migrate function, but its sub-objects do, then it may be able to forward such a request to a sub-object. Rather than trying to exhaustively define all possible requests as part of the common PSO interface, the `special` function handles this case. The arguments identify the operation requested and refer to a variable length buffer that holds the arguments specific to the operation.

5.3 Physical Server Objects

We have developed the PSO classes shown in Table 5.1. PSO classes are divided into three categories, namely: basic, read/write, and composite. Basic PSO classes define objects that directly refer to disk blocks. Read/write PSO classes define objects which can only be accessed using read/write operations (rather than mapped-file I/O). Composite PSO classes define objects that store file data indirectly by referencing sub-objects.

In the next three sub-sections we briefly describe these PSO classes. For the sake of simplicity, we will assume throughout this discussion that disk blocks, physical page frames, and file blocks are all 4096 bytes. To simplify the caching of PSO member data in memory, we have defined PSO classes to (with one exception) have member data that is either 64 or 4096 bytes. The need to have the amount of object member data fit into one of these two sizes is the cause of constraints like the maximum number of sub-objects a PSO can have.

5.3.1 Basic PSO classes

All disk blocks referenced by *basic* PSOs must be on a single disk.

random-access PSO: These objects maintain a vector of one thousand pointers to disk blocks. The i^{th} entry in the vector identifies the disk block that holds the file data for the i th block of the file stored by the PSO. A random-access PSO can directly reference up to 4000 KBytes of file data.

sparse-data PSO: These objects maintain a vector of 512 entries, where each entry identifies a file block and the disk block which stores it. A simple hash function, based on the file block number, is used to locate the entry for a file block. Hence, these objects are best suited for sparse files, where large portions of the file are not populated. An instantiation parameter identifies which bits of the file block number are used by the hash function.

extent-based PSO: This object class is intended for files typically accessed using requests for large numbers of sequential file blocks. Each such PSO can represent up to 14 extents, where an extent is a set of up to 256

Basic PSO classes			
object description	data length (bytes)	max sub-objects	`obj_type`
random-access PSO	4096	1	5
sparse-data PSO	4096	1	6
extent-based PSO	64	1	7
Read/write PSO classes			
object description	data length (bytes)	max sub-objects	`obj_type`
small-data PSO	variable	0	8
fixed-sized record-store PSO	64	1	9
Composite PSO classes			
object description	data length (bytes)	max sub-objects	`obj_type`
striped-data PSO	64	15	10
replicated-data PSO	64	15	11
distribution PSO	64	15	12
checkpoint PSO	64	15	13
parity PSO	64	2	14
appl. specific distribution PSO	4096	256	15

Table 5.1: PSO classes.

consecutive file blocks stored in contiguous disk blocks. Each extent is identified by a 24 bit number indicating a starting disk block number and a 8 bit number indicating the extent size. The maximum extent size is determined by an instantiation parameter. The size of an extent may depend on the number of contiguous disk blocks available when the extent was created.

The file data blocks of *random-access* and *sparse-data* PSOs are always written to disk at a location as close as possible to the current disk head so as to maximize disk throughput. If the file block being written has previously been written to disk, the previously used disk block is freed. In contrast, the disk blocks of an *extent-based* PSO are reused.[2]

With *extent-based* and *sparse-data* PSOs, the amount of data that can be directly stored by a PSO will vary from PSO to PSO. The size of each extent of an *extent-based* PSO, for example, can differ depending on how fragmented the disk is when the extent is allocated.[3] Hence, each basic PSO includes a sub-object pointer to allow objects of the same class to be chained together. A chain of basic PSOs can thus reference an arbitrary number of file blocks.

When an *extent-based* or *random-access* PSO forwards a request along its chain, it adjusts the file offset being referenced by the amount of data it directly references. For example, if a *random access* PSO receives a request to write file block 1001, it instantiates a new *random-access* PSO, saves the token of the new object as its sub-object, and directs the write operation to that sub-object (as a write to file block 1 for that object). Any subsequent requests to read or write file blocks past 1000 are directed to the sub-object. When a *sparse-data* PSO forwards requests along its chain it does not adjust the file offset.

It is obvious that it is inefficient if data access must occur indirectly through a long chain. Our solution to this problem involves the use of composite PSOs and is discussed in Section 5.4.

[2] A new extent is allocated on the first write past a previous extent. Our application-level library releases any excess at the earliest opportunity.

[3] With our implementation of the sparse-data PSO class, the number of blocks referenced directly depends on how the hash function matches the file blocks used. The sparse-data PSO attempts to place a new entry either in the entry indicated by the hash function or the two succeeding entries, and, if all three entries are used for other file blocks, forwards the request to the sub-object PSO.

5.3.2 Read/write PSO classes

Read/write PSO classes are used for those files where it is not appropriate to use a page-sized block-oriented interface to access the file. If an application accesses such a file, it must use a read/write interface rather than a mapped-file interface. Read/write PSOs are primarily used by the file system itself to store meta-data. We have implemented two read/write PSO classes.

small-data PSO: These object can store less than 2 KBytes of data. They only accept read and write requests for the entire file data. The maximum file size is specified with an instantiation parameter. The file data is contained directly as part of the PSO's member data.

fixed-sized record-store PSO: These objects store small, fixed sized records, where the record size is an instantiation parameter. A bit map 30 Kbits large identifies the free and occupied record locations. A *fixed-sized record-store* PSO has a single sub-object that it uses to store the file data. Read and write requests are always for entire records, and the offset of a read/write request is interpreted as a record number rather than a character offset.

Internally, HFS uses *fixed-sized record-store* PSOs to store all PSOs whose member data requires less than a disk block. For example, while *small-data* PSOs are used for small files, those PSOs are, in turn, stored as records of a *fixed-sized record-store* PSO. A *fixed-sized record-store* PSO's record size can be chosen so as to reduce the disk space wasted; in contrast, the Unix FFS uses a minimum fragment size of 1 KByte.

On each disk the superblock maintains an array of a small number of tokens of *fixed-sized record-store* PSOs, where the upper 8 bits of a PSO's `obj_loc` (from the object map entry) is used as an index into this array to obtain the token of the *fixed sized record store* PSO that stores the object member data, and the lower 16 bits is used to identify the particular record number of the record used.

5.3.3 Composite PSO classes

A composite PSO handles requests for file data by directing the requests to its sub-objects, where the sub-objects of a composite PSO can be *basic* PSOs or other *composite* PSOs. Generally, a composite PSO has a number of sub-objects, where each of the sub-objects is used to store some part of the file data. For example, one of the classes distributes file blocks in a round robin fashion across its sub-objects. Unless otherwise stated, when a composite PSO forwards a request to read or write file data to a sub-object, it adjust the file offset so that the sub-objects can be dense files. That is, a request made to a sub-object for block n is for the n^{th} block stored by the sub-object and not for the n^{th} block of the file as a whole.

striped-data PSO: These objects distribute data in a round robin fashion across up to 15 sub-objects. Four instantiation parameters define the striping policy: 1) the number of sub-objects, 2) the *striping unit*, that is, the number of consecutive file blocks written to a sub-object, 3) the *stripe length*, that is, the number of consecutive striping units written to consecutive sub-objects, and 4) the *skip unit*, that is the number of sub-objects skipped after writing each stripe length. (As described in Section 5.4, the stripe length and skip unit make it possible to emulate RAID-5 disk arrays, where parity blocks are distributed across the disks so that they are interleaved with data blocks.)

replicated-data PSO: These objects replicate file data across up to 15 sub-objects. Three policies for reading file data are defined: 1) reads directed to the sub-object located closest to the memory module containing the target physical page frame, 2) reads directed to the sub-object located on the disk with the shortest queue of pending requests, or 3) reads directed to a particular sub-object. A default policy is chosen at PSO instantiation time and can be modified at run time.

distribution PSO: These objects distribute file data across up to 15 sub-objects. A *sub-object size* is defined by an instantiation parameter. The first sub-object is filled before the following one is used. The first sub-object then stores the first *sub-object size* blocks of file data, the next sub-object stores the next *sub-object size* blocks of file data, etc.

checkpoint PSO: These objects are used for checkpointing, where the data is written once and very seldom read. File data is distributed across up to 15 sub-objects. Two policies for writing file data are defined: 1) writes directed to the sub-object located closest to the memory module containing the physical page frame, 2) writes directed to the sub-object located on the disk with the shortest queue of pending requests. A default policy is chosen at file creation time, and can be modified at run time.

A checkpoint PSO does not keep track of which sub-object stores what file data. A read request is forwarded to each sub-object until the read is successful. This class differs from the other classes described so far in that the offset passed to the sub-object is not altered before the request is forwarded, which means that the sub-objects of a checkpointing PSO will typically be sparse even if the checkpointed data itself is not sparse. It is not possible for the checkpointing PSO to adjust the offset, since it has no way to identify *a priori* what data will be stored by each sub-object.

parity PSO: These objects have two sub-objects, one that stores the file data, and one that stores parity information for the file data. The parity information is used for fault tolerance as described in Chapter 8. A *blocking factor*, specified at object instantiation time, determines the number of consecutive file data blocks used in calculating a parity block. That is, with blocking factor b, word w in the n^{th} parity block is equal to the exclusive-or of word w in data blocks $(n \times b)$ to $((n + 1) \times b)$.

application-specific-distribution PSO: These objects distribute data across up to 256 sub-objects in an application determined way. They maintain a vector of three thousand numbers, where entry *i* in the vector identifies the sub-object that stores file block *i*. For example, if entry 15 in the vector has the value 100, then the 15th file block is stored by 100th sub-object. The maximum amount of file data that can be stored is 12 MBytes. The vector is downloaded by the application when the PSO is instantiated. The application can at run time request that a file block be migrated from one sub-object to another.[4] This class is similar to the *checkpoint* PSO class in that the offset passed to sub-objects must be the same as the offset received by the *application-specific-distribution* PSO, since there is no way to determine *a priori* what file data will be stored by which sub-object. Hence, the sub-objects of an *application-specific-distribution* PSO store sparse data.

We do not claim that the set of composite PSO classes listed here is complete. However, we developed this particular set for three reasons. First, they each implement different policies for data distribution, and hence form a reasonable set of PSO classes to test our design. Second, while each of them is quite simple, they can be combined together to implement a wide variety of policies for distributing data to different disks, as we will show in the next section. Finally, these object classes all have characteristics described in Chapters 7 and 8 that simplify fault recovery and scalability.

5.4 Data distribution examples

This section illustrates how the simple PSO classes described in the previous section can be combined together in arbitrary ways to achieve tremendous flexibility.

Example: Large per-disk files

An object of any of the basic PSO classes can handle arbitrary amounts of data by chaining multiple objects of the same type together. However, the length of these chains can become long, and hence expensive to traverse, for large files.

A *distribution* PSO can efficiently implement large per-disk files by referring to up to 15 basic PSO sub-objects on a single disk. If the basic PSOs are *random-access* PSOs, then the *sub-object size* of the *distributed* PSO should be set to 4 MBytes, which is the amount of data that a *random-access* PSO can reference directly. In this case, the file can be used to store up to 60 MBytes of file data.

For even larger per-disk files, the sub-objects of the *distribution* PSO can themselves be *distribution* PSOs whose sub-objects are basic PSOs. In this case, if the basic PSOs are *random-access* PSOs, the *sub-object size* of the top level PSO should be set to 60 MBytes, and the file as a whole can store 900 MBytes.

In all subsequent examples, whenever we refer to a per-disk file, we are referring to a file implemented in any of the ways described here.

[4] On receiving a request to migrate a file block, the PSO reads the file block from the sub-object that currently stores it, makes a request to that that sub-object to delete that block, writes the block to the sub-object that will now store it, and modifies the corresponding entry in the vector to identify the new sub-object for that block.

Example: Striped files

A storage object that is striped across 15 disks can be constructed from a *striped-data* PSO that refers to 15 per-disk files. To have successive file blocks written to successive disks, the striping unit should be set to 1 and the skip unit should be set to 0.

To stripe across a larger number of disks, *striped-data* PSOs can recursively refer to other *striped-data* PSOs. For example, a file can be striped across 3375 disks using three levels of *striped data* PSOs. To have successive file blocks written to successive disks, the top *striped-data* PSO should have a striping unit of 225 and the *striped-data objects* in the second level should have a striping unit of 15. With this approach a file can be striped across any number of disks that is equal to the product of factors that are each less than or equal to 15.

When data is striped across a large number of disks, it many be important to store data redundantly so that disk failures can be tolerated. A file that emulates RAID level 4[5] can be implemented using a *parity* PSO with the data sub-object pointer referencing a striped file and the parity sub-object pointer referencing a per-disk file. RAID level 5 files can be implemented using a *parity* PSO referring to two files striped across the same sets of disks. The stripe length and skip unit of the two sub-objects would be set so that the parity blocks are properly interleaved with the data blocks.

Example: A checkpoint file

Periodically checkpointing application state allows the application to recover from the checkpointed state if the system crashes. A checkpoint file distributed across 225 disks can be constructed from a *checkpoint* PSO that has 15 *checkpoint* PSO sub-objects that each in turn have 15 *sparse-data* PSO sub-objects. To take into account memory/disk locality, the writing policy for the top *checkpoint* PSO should be to direct writes to the sub-object nearest to where the memory is located. To balance the load across the disks (near the memory containing the page frame), the policy for the second level of *checkpoint* PSOs should be set to write data to the least loaded disk.

Example: A read only large file

Consider a file that is sometimes accessed using requests for large numbers of sequential file blocks, and is at other times accessed by many different threads (or applications) using small random-access requests. To optimize for the first case, we might want to distribute the file across a small number of disks in order to exploit the on-disk caches. To optimize for the second case, we might want to distribute the file across a large number of disks in order to minimize interference between the different threads. Each of these distributions would perform poorly for the type of access it is not optimized for.

For a read-only file, it makes sense to replicate the file using a *replication* PSO with each replica optimized for a different type of access. In this case, the application (or application-level library) selects which replica is used for read operations at run time, depending on the expected access pattern.

Application-specific distributions

Some applications want explicit control over how data is to be distributed across the disks. For example, data bases can maintain statistics on the load on different parts of the data base, and hence can exploit this information to balance the load on the disks. A file implemented using *application-specific-distribution* PSOs gives the application this kind of control. The application can make `migrate` requests at run time to move file blocks between the disks. Files larger than 12 MBytes, can be implementing using *distribution* PSOs with *application-specific-distribution* sub-objects. (We cannot conveniently distribute file data across more than 256 disks, with our implementation of these PSO classes.)

We could always use *application-specific-distribution* PSOs instead of *striped-data* or *distribution* PSOs. However, the member data of *striped-data* and *distribution* PSOs is more compact, allowing it to be cached in main memory more efficiently. Also, with *striped-data* PSOs and *distribution* PSOs, any of the three basic PSO classes can be used for storing the file data on disk, while with *application-specific-distribution* PSOs only *sparse-data* PSOs can be used to store the file data on disk.

[5] RAID level 4 stripes data across some number of disks and parity information is written to a single other disk. RAID level 5 stripes both data blocks and parity blocks across the same set of disks.

PSO class	input parameters
random-access	disk
sparse-data	disk, bit_mask
extent-based	ssoid, disk
small-data	disk, max length
fixed-sized record-store	ssoid, disk, el_size, token
striped-data	ssoid, disk, numobj, str_unit, str_len, skip_unit, tokens 1-15
replicated-data	ssoid, disk, policy
distributed	ssoid, disk, numobj, obj_len, tokens 1-15
checkpoint	ssoid, disk, numobj, policy, tokens 1-15
parity	ssoid, disk, token1, token2
appl. specific distribution	disk, token1-256, dist_pattern

Table 5.2: Constructor parameters for the PSO classes. A `token` parameter identifies a token that will be used as a sub-object. A `ssoid` parameter identifies some *fixed-sized record-store* PSO that will store the object member data. All PSO class constructors return a status and a token for the object created.

5.5 Object instantiation

When creating a file, the application (or application-level library) makes requests to the BFS to instantiate the PSOs that will implement the file.[6] In doing so, the application passes to the BFS a buffer containing the instantiation parameters. The parameters for the object classes we have described are shown in Table 5.2. The `ssoid` parameter identifies a particular *fixed-sized record-store* PSO that will store the member data of the PSO being instantiated. With our current implementation, the application can choose from *fixed-sized record-store* PSOs that store object member data: 1) on a particular disk, 2) striped across all system disks, and 3) striped across all system disks with redundancy (for fault tolerance).

It is clear that it is important to match the PSOs used to the expected file access behavior in order to obtain good performance. While we have found writing application-level code to create files to be relatively easy, we do not in general expect application programmers to write this code. Instead we believe that a number of approaches will allow even novice users to exploit the flexibility of our building-block approach:

1. For many parallel applications a large part of the I/O is a result of paging activity due to insufficient physical memory to hold the entire application image in memory at the same time. This is entirely transparent to the application. The files used to page different parts of the application memory image will be created by the compiler, and we expect that future compilers will be able to take the requirements of the application and the capabilities of the I/O system (i.e., the number and latency of the disks, and the capabilities of the file system) into account to select appropriate PSOs. This is no different than an HPF compiler having to take into account the characteristics of the application and system in order to distribute data across the memory modules of the system.

2. The application-level library can create files using reasonable defaults if no information is available from the application. Initially, it assumes that a file will be small, and uses a *small data* PSO to store the file data. If this turns out to be incorrect (when the application writes more data than can be held by a *small-data* PSO), the application-level library assumes that the file will be medium sized and should be distributed across a small number of disks. In this case, the library will create a new file, copy the data from the original to the new file, use a rename operation to switch the contents of the object map entries for the two files, and then deletes the original file.[7] If a file grows to be very large, then the library will: 1) instantiate a new top level distribution

[6] Applications typically communicate with the BFS indirectly through the OFS or the memory manager, which are trusted servers. Object instantiation and some `special` requests are exceptions, where the application communicates directly with the BFS. The authentication mechanism used for these requests is described in Section 5.6.3.

[7] As an optimization, the application-level library need not instantiate any PSOs until either the application has written more data (initially buffered in the application address space) than can be handled by a small-data PSO, or the application has closed the file.

PSO, 2) use the existing file as the first sub-object of the new PSO, and ultimately, 3) instantiate new PSOs for the other sub-objects on demand. With this scheme, new, higher layers are added as the file grows in size.

3. We have constructed a variety of library routines that implement the most common types of data distribution, including the different example distributions described in the previous section. These routines hide the process of instantiating PSOs from the application programmer. We expect that the choice of routine, together with the parameters specified on calling it, will provide sufficient flexibility for many applications.

4. In many cases, files are created by system utilities, such as Unix `cp` or `cat`. These utilities cannot know how the file will subsequently be accessed. To allow a utility to be used to create a file with a specific distribution, we plan to associate *templates* with directories, where the template specifies the application library routine and its parameters that should be used to create the file. We expect to provide a number of directories that have default templates, so that the user can create files in these directories and then move the file into a target directory owned by the user. We also expect to provide utilities to convert a file from one structure to another.

5. While we have not yet done so, in the future we would like to implement objects that keep statistics on how the file is being accessed, periodically re-evaluate the choice of file structure, and restructure if necessary.

5.6 Implementation details

The internal structure of the Block File Server is shown in Figure 5.3. We have implemented the BFS in C++, and all blocks shown in this figure are C++ objects. Each PSO class is implemented by a C++ class, and derivation is used to allow code to be re-used in a natural way. Ninety percent of the code of the BFS is specific to the PSO classes. The remaining ten percent is glue, to locate objects, to manage shared resources such as the cache of file system meta-data, and to handle interactions with the device drivers.

While conceptually a PSO encapsulates both member data and member functions for operating on that data, in our implementation the member functions of a PSO are not stored on disk with the member data, but are instead implemented by a C++ object that is always memory resident. Since the member functions are common to all objects of the same class, a single C++ object is used per-PSO class to implement the member functions for all PSOs of that class.[8] The separation between member functions and member data in our implementation allows PSO member data to be stored on disk in a class-specific fashion, where a member function of the PSO can be invoked in order to read the member data of the PSO from disk.

Each request to the BFS includes a token parameter that identifies the target PSO. The request is received by the *object manager* which looks up the class of the PSO from the `obj_type` in the object map entry of the PSO. The object manager then invokes the corresponding class-specific code by forwarding the request to the corresponding C++ object.

All file system meta-data is cached in main memory in the *meta-data cache*, which is described in detail in Section 5.6.1. It caches blocks of object map information, superblocks, blocks of block map information, and the PSO member data. All file system meta-data is cached in one common cache, instead of separate caches, so that the memory can be used most efficiently. While the meta-data cache is responsible for caching PSO member data, since the member data is stored on disk in a class-specific fashion, the meta-data cache does not read or write the data to disk.

The *PSO class array* is an array of C++ objects, where each C++ object corresponds to a particular PSO class, implementing its member functions. When handling a request to a PSO, the `obj_type`, obtained either from the token or from the object map entry, is used to index into this array to locate the appropriate C++ object and the corresponding member function. The C++ object then uses the PSO's member data in the meta-data cache to handle the request. If the PSO's member data is not available in the cache, the C++ object loads the required data into the cache.

A *disk object* controls the I/O to a particular disk. It handles all interactions with the device driver and maintains a queue of outstanding read and write requests for the disk (as described in Section 4.2).

[8] Our implementation is similar to the way C++ objects are commonly implemented, where a pointer maintained with the object data identifies an array of virtual function pointers, and the array of virtual function pointers is shared by all objects of the same class. For comparison, the `obj_type` of the PSO corresponds to the pointer to the array of virtual function pointers, and the C++ object that implements the member functions of a PSO class corresponds to the shared array of virtual function pointers.

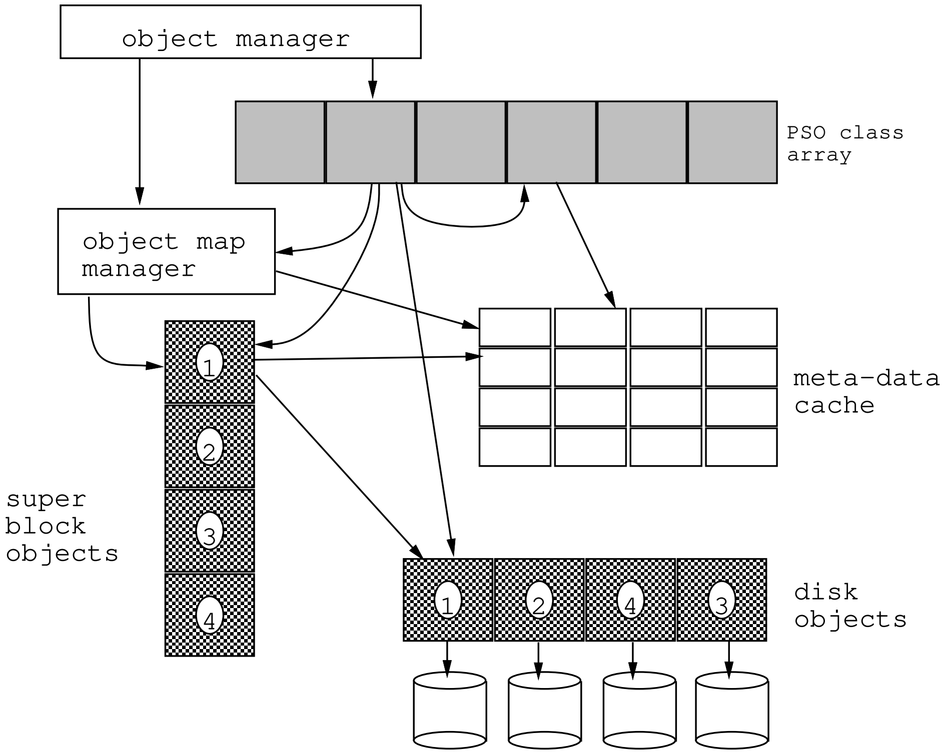

Figure 5.3: BFS internals. The PSO class array is an array of C++ objects that each implement the member functions of a PSO class. All file system meta-data, including PSO data, superblocks, block map blocks, and object map blocks, is cached in the meta-data cache.

A *superblock object* maintains the superblock, block map, and object map of a particular disk.[9] It is responsible for reading and writing these structures (via the disk objects) and controls the caching of this information in the meta-data cache. Also, each superblock object controls the allocation of disk blocks on that disk.

All requests to read or modify object map entries are directed to the *object map manager*. The object map manager uses the *omap* field in the PSO's *token* to determine which superblock object is to read or write the object map. The object map manager uses the object number in the PSO's *token* to determine the corresponding the object map entry.

5.6.1 The meta-data cache

The meta-data cache is used by all components of the BFS to cache file system state. In particular, it is used to cache object map blocks, PSO member data, file blocks, superblocks, and block map blocks. In this section, we describe the internal structure of the meta-data cache and the interface the meta-data cache exports. We will refer to particular state cached by the meta-data cache as a cached element.

The internal structure of the meta-data cache is shown in Figure 5.4. A simple hash function, based on type, id, and token, is used to locate a particular cached element. Token identifies the PSO to which the element belongs.[10] `Type` identifies the type of the data being cached (e.g., file data or object map data).[11] `Id` identifies the instance of data of that type cached (e.g., block 10).

[9] Both the disk and superblock objects are organized in arrays, so that the *omap* field in the token can be used as an index into this array for fast access.

[10] Tokens with object numbers less than 5 are reserved to identify cached superblock, block map, and object map blocks.

[11] For a PSO that stores the member data of other PSOs, both the object member data and the file blocks stored by this PSO may cached in the meta-data cache, and the `type` field is used to differentiate between these two kinds of information.

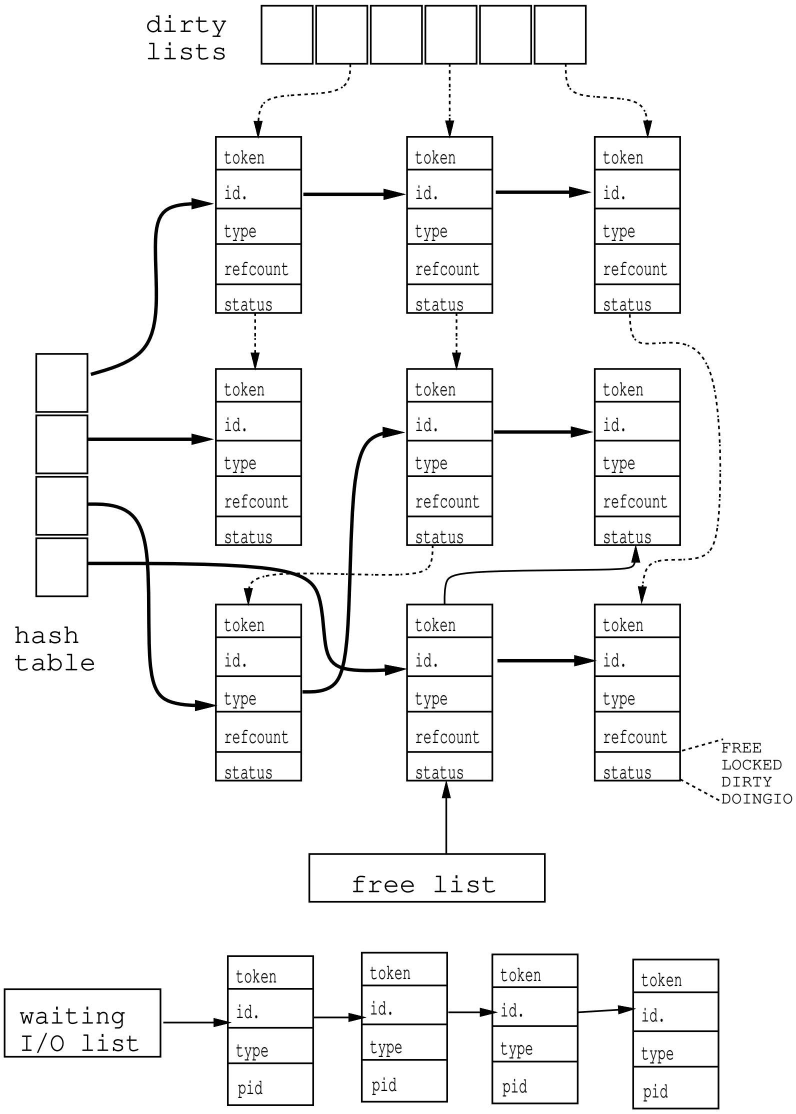

Figure 5.4: Organizations of BFS meta-data cache.

acquire_rd(token, id, type, el) returns pointer to element `el` and status: locks `el` for read access

acquire_wt(token, id, type, el) : returns pointer to element `el` and status: locks `el` for write access

release_rd(el) : releases read lock

release_wt(el) : releases write lock

mark_dirty(el, list) : marks `el` as dirty

clear_dirty(el) : marks `el` as clean

mark_doingIO(el) : marks `el` as doing I/O

clear_doingIO(el) : clears doing I/O

downgrade(el) : downgrade access from exclusive to inclusive

getdirty() returns token: get a token of a file with something dirty in the cache

getdirtyel(token) returns cache element: return a dirty element that from the specified file

invalidate(el) : marks `el` invalid

invalidate(token) : marks all elements from a file invalid

Figure 5.5: The meta-data cache interface

The interface exported by the meta-data cache is shown in Figure 5.5. `Acquire_rd` and `acquire_wt` locate and lock a cached element and return a pointer to that element. A reader-writer lock is used, allowing either multiple readers to inspect shared data or a single writer exclusive access for modifying the data. `Release_rd` and `release_wt` unlock a cached element.

On a cache miss, the meta-data cache allocates a cache element from the free list, exclusively locks the element, and returns a pointer to the newly allocated cache element together with status information that indicates that the element returned contains invalid data. The requester is then responsible for obtaining the valid data, either from disk or from some other PSO, and placing it into the acquired cache element. All other processes requesting the same element will block on the cache element until a valid copy of the data becomes available.

Whether processes should spin or block when waiting for a lock depends on how long the element is expected to remain locked. If a lock on a cache element is being held for a short period of time, then spinning is more appropriate. However, if the lock is being held for a long period of time, it is better for the process to be blocked (instead of spinning) in order to release the processor for more productive use. The DOINGIO flag in the status field of the cache element is used to determine which strategy should be used; a set flag indicates that the lock will be held for a long time. A PSO can set and unset the DOINGIO flag explicitly using the `mark_doingIO` and `clear_doingIO` operations. Also, the DOINGIO flag is set by the meta-data cache when a miss occurs and a new cache element is allocated.

Whenever a process is blocked, its identity and the identity of the target element are enqueued on the *waiting for I/O* list. When the lock is released, then all processes waiting for the element are awakened.

A single free list runs through the elements of the meta-data cache. The free list contains items that are neither locked nor dirty. Dirty elements are not kept on the free list, because it simplifies the allocation of new elements, since it is not necessary to check if the element removed from the free list is dirty or not. New elements are allocated from the front of the list, and freed elements are appended to the end, implementing an LRU policy for re-allocating the memory of the meta-data cache.

The meta-data cache is virtually addressed (by `token`, `type`, `id`) rather than physically addressed (by disk address), because it allows the position of the data on disk to be determined when it is being written instead of when it is allocated, making it is easier to take advantage of the current disk head position when allocating the disk frame for the data. Moreover, if a file exists for only a short period of time (and for a large number of files this is the case), then no disk blocks have to be allocated for the file if it is created and then deleted without ever having been written to disk.

The BFS periodically flushes dirty meta-data to disk to make the on-disk copy consistent with the cached copy of the meta-data. The writing out of cache elements is always under the control of a PSO. The BFS calls `getdirty` to retrieve the token of a PSO with dirty elements in the cache, and then makes a request to that PSO to flush its dirty state from the cache. The PSO then repeatedly calls `getdirtyel` to retrieve one dirty cache element (specific to that PSO), writes that cached data element to disk, and calls `clear_dirty` to mark the cache element as clean.

There are multiple dirty lists, as shown in Figure 5.4. They are required to make sure that dirty elements are written to disk in the proper order. For example, PSO member data should be written to disk before the object map blocks, since the object map entry will change to reflect the (new) position of the object member data once it has been written to disk. Similarly, superblocks should be written only after the object map blocks have been written to disk. The routines `getdirtyel` and `getdirty` search each dirty list in turn, from lowest to highest priority, when searching for an element to be cleaned.

While logically there is only a single meta-data cache, we have implemented them as two caches, one for elements that are 64 bytes large, and one for elements the size of a disk block; most of the PSOs we have implemented come in one of these two sizes. Having multiple caches, each tuned for a different element size, results in better performance than having a single physical cache with multiple element sizes.

Before a PSO forwards a request to a sub-object, the PSO either releases (i.e., unlocks) any meta-data cache elements it holds, or sets the DOINGIO flag for each of the acquired elements. It is preferable to release cache elements rather than set the DOINGIO flag, since it allows greater concurrency, and since setting and un-setting the DOINGIO flag can be expensive relative to the simple code executed by many PSOs. If a element cannot be released, then it is important to set the DOINGIO flag, since it is possible that the sub-object will require I/O, and setting the DOINGIO flag will cause other processes requesting locked data to block and release the processor rather than spin during the entire time required for the I/O. We have found that by properly designing the PSO classes, it is usually possible to release all cache elements before forwarding the request. For example, when a read operation is passed from a PSO that implements a striped file to a per-disk PSO, the meta-data cache element for the striped file PSO member data is released (and never re-acquired for that request).[12]

5.6.2 A simple example

To illustrate how the data structures and objects described in the previous sections are used, consider a `read_page` request from the memory manager to a file implemented by a *random-access* PSO. Let us assume that neither the object map block that contains the object map entry nor the PSO member data are in the meta-data cache, but that the superblock is in the meta-data cache. This is an extreme example, since before the `read_page` request can be handled, both the object map block and the PSO member data will have to be read from disk.

The `read_page` request received by the object manager includes as parameters 1) the token of the top level (in this case *random access*) PSO, 2) the address of the physical page frame into which the data should be read, and 3) an offset identifying the required file block.

To determine which object-specific code should handle the request, the object manager requests from the object map manager the `obj_type`. The object map manager uses the `omap` and `object number` embedded in the PSO's token to determine the object map block that contains the required object map entry, and attempts to acquire that block from the meta-data cache. Because the object map block is not found in the cache, the object manager acquires a new meta-data cache entry from the meta-data cache, who locks the entry as a side effect. (All subsequent requests for that object map block will block until it has been read from disk and released by the object manager.) The object map manager then makes a request to the superblock object to read the object map block into the acquired meta-data cache entry. The superblock object determines the location of the data on the disk by accessing the superblock data, which it acquires from the meta-data cache and then releases. It then makes a request to read the corresponding disk block. Once the read operation has completed, the object map manager retrieves the `obj_type` from the object map entry, releases the cached block, and returns the `obj_type` to the object manager.

The object manager uses the `obj_type` returned by the object map manager to determine which C++ object implements the PSO's member functions and makes the read request to that object. The C++ object then attempts to acquire the PSO's member data from the meta-data cache. Because it is not found there, the meta-data cache allocates a new locked cache element for this data. The C++ object then makes a request to the object map manager to obtain the `obj_loc`, uses this information to determine the disk block that contains the PSO data, and then makes a request to a disk object to read that data into the acquired meta-data cache element. Once the PSO data has been read from disk, the C++ object uses this data to determine the location on the disk of the file data requested by the memory manager

[12] The only cases where we have found it necessary to hold on to cache elements is for non-performance critical operations, such as when a file is being deleted.

and initiates a read request to a disk object for that data. Finally, once the file data has been read into the physical page frame, the BFS responds to the memory manager as described in the previous chapter.

We have illustrated how the in-memory data structures are used for a relatively simple case. More complicated cases include PSOs whose member data is stored by other PSO and composite PSOs. While a detailed description of these cases may seem complicated, our object-oriented building-block approach makes each of the objects relatively simple.

5.6.3 Authentication

Most requests to the BFS come from trusted servers, such as the memory manager or the OFS. No authentication needs to be done on these requests. However, the application level interacts directly with the BFS when a file is created and for some `special` requests, so the BFS must ensure that these requests and their parameters are valid.

Since the arguments to instantiate an object and the `special` arguments are class specific, the target PSO class must be responsible for authenticating these requests. To facilitate object-specific authentication, the BFS maintains a simple hash table. Each entry in the hash table contains a `token`, a `pid` and a `flags` field. An object can place an entry in the hash table to indicate that a particular process is allowed access to a particular PSO identified by the token. One bit in `flags` indicates whether the process is allowed *bind* privileges, that is the ability to instantiate new PSOs that references the PSO identified by the token as a sub-object. The other bits in the `flags` are interpreted in a class-specific fashion.

On instantiation of a PSO, the class constructor ascertains that the requesting process has *bind* privileges for the sub-objects specified, instantiates the new PSO, removes its sub-objects from the hash table, creates a new token that references the new object, grants the process permission to bind to that PSO, and finally returns a status and the new token to the application. When the application binds a token into a directory (i.e., a Naming LSO), then the name server makes a request to the BFS to determine whether the process has bind permission on that object, and if so removes that permission and does the binding.

Chapter 6

Application-level library implementation: The ASF

This chapter describes the application-level library portion of the HURRICANE File System (HFS), the *Alloc Stream Facility* (ASF). One of the most original features of HFS is the amount of file system functionality implemented at the application-level by ASF. The main advantage to having functionality provided by an application-level library, instead of by a system server, is that it improves performance by reducing the number of times it is necessary to make calls to system servers and hence cross address-space boundaries.

In addition to being a part of HFS, ASF serves as the library component for all other I/O services provided on HURRICANE. That is, it handles all interactions between applications requesting I/O and the HURRICANE servers that provide I/O services, including the HFS servers, NFS servers, pipe server and TCP/IP servers.

As with the other layers of HFS, we use an object-oriented building-block approach to implement the functionality provided by ASF. The storage objects implemented by ASF are called *application-level Storage Objects* (ASOs). For each type of I/O service (and each type of disk file) there is at least one ASO class specific to that service. We have also defined a large number of ASO classes to provide file system policies, such as prefetching and advisory locking[1].

All ASO classes provide a minimum common interface, called the *Alloc Stream Interface* (ASI), that allows applications to make I/O requests to an ASO without knowledge of the type of I/O service that object provides. ASI minimizes copying overhead, and hence allows ASOs to be efficiently combined together in a building-block fashion.

We believe that the application-level library of a parallel file system must support a variety of I/O interfaces, both for compatibility with existing interfaces, and to allow for new parallel I/O interfaces currently being developed. ASF is unique in its ability to support a variety of I/O interfaces. In our current implementation we provide, in addition to ASI, the stdio and Unix I/O interfaces.

Section 6.1 describes the Alloc Stream Interface. Section 6.2 describes some ASO classes we have developed. Section 6.3 describes how ASOs are instantiated. Section 6.4 describes some of the implementation details of ASF. Finally, Section 6.4.2 reviews the advantages of ASF, both in general, and more specifically as a part of the HURRICANE file system.

6.1 The Alloc Stream Interface

The *Alloc Stream Interface* (ASI) is the common interface provided by all ASO classes. This interface is similar to the Unix I/O or stdio interfaces (described in Chapter 2) in that it is an interface to a character stream that can be used uniformly for all I/O services (e.g., disk files, terminals, pipes). The key difference between the ASI, and the Unix I/O and stdio interfaces, is that the ASI allows applications to directly access the internal buffers of the ASO instead of having the application specify a buffer into which or from which the I/O data is copied. We describe in this section the C variant of this interface, where the first parameter to each ASI function identifies the target ASO.[2]

[1] As discussed in the next chapter, many of the policy ASO classes have not yet been implemented.

[2] The C++ variant of ASI differs only in that there is no need to explicitly specify the target object in the arguments of member functions.

Memory allocation operations :

malloc(length) returns pointer : allocates `length` bytes of memory

free(ptr) returns error code : frees previously allocated region

ASI operations :

salloc(stream, length) returns pointer : allocates `length` bytes from `stream`

sfree(stream, ptr) returns error code : frees previously allocated region

Figure 6.1: Comparing The Standard C Memory Allocation Interface to ASI

ASI is modeled after the standard C memory allocation interface. The two most important ASI operations are `salloc`, which allocates a region of an I/O buffer, and `sfree`, which releases a previously allocated region. They correspond to the two memory allocation operations: `malloc`, which allocates a memory region and returns a pointer to that region, and `free`, which releases a previously allocated region (Fig. 6.1). The arguments to `salloc` and `sfree` differ from the corresponding memory allocation operations in that the ASI operations require a stream handle to identify the target ASO.[3]

`Salloc` is used together with `sfree` for both input and output. In the case of input, `salloc` returns a pointer to a region of memory that contains the requested data, and advances the stream offset by the specified length. `Sfree` tells the ASO when the application has finished accessing the data, at which point the ASO can discard any state associated with it. In the case of output, `salloc` returns a pointer to a region of memory where the data should be placed and advances the stream offset. The application can then use this region as a buffer in which to place data to be written to the stream. `Sfree` tells the ASO that the modifications to the buffer are complete.

For a disk file in read mode, only data already in the file can be allocated. If an application attempts to allocate past the end of file, the length parameter is set to the amount of remaining data and only that data is allocated. The application should not modify data obtained by `salloc` in read mode.[4]

For a disk file in write mode, if data is allocated past the end of the file, then the file length (from the perspective of the application) is automatically extended on `sfree`. This occurs on `sfree` rather than on `salloc`, since the data is in a non-deterministic state until the `sfree` has been performed and therefore should not be visible to other threads. The order of writes are guaranteed to be in the same order as the corresponding `salloc` operations even if the `sfree` operations occur in a different order.[5]

The full ASI interface is shown in Figure 6.2. An ASI stream is always in one of two modes: read or write. If the stream is opened for read-only access, then the mode of the stream defaults to read mode; otherwise it defaults to write mode. The mode of the stream can be changed using the ASI routine `set_stream_mode`. The ASO pointer (i.e., stream handle) returned by `set_stream_mode` identifies the ASO to which subsequent requests should be directed.[6]

`SallocAt` allows the application to allocate data from a specified location in a stream. The `whence` field indicates whether the `offset` should be interpreted as absolute, relative to the current offset, or relative to the end of file.

`Srealloc` allows the application to shrink or grow a previously allocated region. Library buffers are a convenient location to prepare data for writing and `srealloc` is used when the application does not know *a priori* the amount of data to be output: the application can `salloc` a large data space and then shrink the region back to the actual amount

[3] In the case of `salloc`, the length parameter is passed by reference and on return either indicates the amount of allocated data or holds a negative error code (in which case the pointer returned by `salloc` is NULL).

[4] The type of error that occurs if the application attempts to modify data obtained by `salloc` in read mode depends on the particular stream type. For streams not implemented using mapped files, the modification will have no effect. For mapped file streams, modifications to the buffer results in a memory access protection violation that returns control to the library.

[5] The one exception is unbuffered read/write streams, where data is written in the order of `sfree` rather than `salloc` operations.

[6] As we describe in Section 6.4.2, some read/write streams are implemented as an ASO with a read-only and a write-only sub-object. For these kinds of streams, `set_stream_mode` returns a stream handle for the sub-object that corresponds to the requested mode. For other read/write streams, `set_stream_mode` changes the mode of the stream and returns a handle to the ASO to which the request was made.

Basic operations:

- **salloc(stream, length)** returns pointer: allocates `length` bytes from `stream`
- **sfree(stream, ptr)** returns error code: frees previously allocated region
- **srealloc(stream, ptr, newlen)** returns pointer: changes length of previously allocated region
- **sallocAt(stream, length, offset, whence)** returns pointer: moves the stream offset to `offset` according to `whence` and allocates `length` bytes
- **sflush(stream)** returns error code: flushes buffers
- **set_stream_mode(stream, mode, error)** returns stream handle: changes mode of stream to `mode` handle

Mode variable operations: These operations differ from the corresponding basic operations in that a `mode` argument is used to explicitly specify that the operation is for reading or writing

- **Salloc(stream, mode, length)**
- **SallocAt(stream, mode, length, offset, whence)**

Unlocked operations: These operations have the same arguments as the corresponding basic and mode variable operations, but differ in that they do not lock the stream.

- **u_salloc(stream, length)**
- **u_sfree(stream, ptr)**
- **u_srealloc(stream, ptr, newlen)**
- **u_sallocAt(stream, length, offset, whence)**
- **u_sflush(stream)**
- **u_SallocAt(stream, mode, len, offst, whce)**
- **u_Salloc(stream, mode, length)**
- **u_set_stream_mode(stream, mode, error)**
- **LockAsfStream(stream):** locks stream
- **UnlockAsfStream(stream):** unlocks stream

Figure 6.2: The Alloc Stream Interface

of data prepared. If the region being reallocated is the most recently allocated region, then `srealloc` repositions the offset in the stream. For example, if `salloc` allocated bytes 1-200 of a file, and `realloc` shrinks the region to bytes 1-20, then the stream offset for the next access will be at byte 21 of the file.

In addition to the basic ASI operations described above, there are also low level operations that allow applications greater flexibility at the cost of additional program complexity. For example, the operations `u_salloc`, `u_sfree`, `u_srealloc` and `u_sallocAt` have the same parameters as the basic operations, but differ in that they do not lock the stream. The (capitalized) `Salloc`, `Srealloc` and `SallocAt` operations are mode variable variants of the corresponding uncapitalized basic operations in that the (read/write) mode is specified on each call, instead of switching between stream modes with `set_stream_mode`.

6.1.1 Advantages of ASI

The most important advantage of ASI over other interfaces is that the buffer used for I/O is chosen by the ASO and not the application. This results in less overhead than read/write interfaces (like Unix I/O) where data must be copied between application and library buffers.

The low overhead of ASI has two major implications for ASF. First, it allows us to combine together ASOs in a building-block fashion. If we had instead used a read/write interface, then n levels of ASOs would have resulted in the data being copied $n-1$ times. The second implication is that it allows other I/O interfaces to be efficiently implemented in a layer above the ASOs. For example, we have implemented the stdio interface in a library layer that uses ASI and have found that our implementation has better performance than most vendor implementations of stdio.

The idea of having the I/O facility choose the location of the I/O data is in itself not new. For record I/O, Cobol and PL/I `read` both provide a *current record*, the location of which is determined by the language implementation [95]. Also, the `sfpeek` operation provided by the sfio library [65] allows applications access to the libraries internal buffers. However, the data of a Cobol or PL/I `read` or a sfio `sfpeek` is only available until the next I/O operation, which is not suitable for multi-threaded applications. In contrast, ASI data is available until it is explicitly freed.

ASI was designed from the start to support multi-threaded applications. All ASI operations return an error code directly so that it is always clear which thread incurred an error. Also, ASI provides the `sallocAt` operation which atomically moves the stream offset to a particular location and performs an `salloc` at that location. Because this is done atomically, there will be no interference from other concurrently executing threads. (As described in Chapter 2, a major problem with the Unix I/O and stdio interfaces is that they provide no such function.) Finally, ASI allows a high degree of concurrency when accessing a stream. Since data is not copied to or from user buffers, the stream only needs to be locked while the library's internal data structures are being modified, so the stream is locked only for a short period of time.

A final advantage of ASI over other interfaces is that it is optimized for uni-directional accesses, that is, a read-only or write-only stream. The majority of streams are used uni-directionally [70, 99]. Also, even for streams open for read/write access, most applications do not frequently interleave read and write accesses. ASF (and other application-level libraries like stdio) must execute special code whenever the application switches between reading from and writing to a stream.[7] Hence, interfaces like stdio or Unix I/O, where the mode of the access is specified by the function, (i.e., `read` is for read access, and `write` is for write access), require the library to check whether the previous operation was of the same mode. In contrast, if an application uses the basic ASI operations (rather than the mode variable operations), then it is agreeing to tell the library explicitly when it is changing the mode of a stream, and hence no checking is necessary.

6.2 ASO classes

It is useful to have a single storage object in the lower layers of the file system implement multiple policies, because the LSOs and PSOs used for a file are fixed, and hence policies can be changed at run time only if the storage objects provide several of them. While having multiple policies results in extra overhead, such as an extra conditional statements or memory accesses, this overhead is typically not significant relative to the incurred cost of crossing address spaces.

In contrast to the other levels of HFS, we have found it advantageous at the application level to have a large number of very specific ASO classes instead of a smaller number of more general ones. One of the major functions of ASF

[7] For example, depending on the type of the stream, the library may have to modify the current offset in the buffer, invalidate some buffers, acquire a different set of locks, etc.

is to reduce the number of interactions with the file system servers, for example, by buffering data in the application address space. Hence, we expect many more requests to ASOs than to PSOs or LSOs. Also, some applications make many fine grain I/O requests (e.g., for a single character) to an ASO. Therefore, overhead at this layer of the system is critical, and an extra conditional statement or memory access may impact application performance substantially.

6.2.1 Service-specific ASO classes

One of the ASF's tasks is to translate between the uniform interface provided to the application (i.e., ASI) and the service specific interfaces provided by the I/O servers. Providing the applications with a uniform interface is important so that the programmers are able to write applications independent of the sources and sinks of their data. On the other hand, we have found that for good performance it is important for the servers to be able to transfer data to and from the application address space in a service specific way. For example, large files are most efficiently accessed using mapped-file I/O, while terminal I/O is best provided using a read/write interface.

At least one service-specific ASO class is defined for each I/O service. Also, for each type of I/O service, we typically define a separate ASO class for read-only, write-only, and read/write access. Many of these ASO classes are quite similar, so for brevity we describe different types of ASO classes rather than the specific classes.

read-only ASO classes: These classes are used for read-only streams, where the source of the stream is a server that exports a read/write interface. We have implemented three ASO classes of this type: one for non-buffered streams, one for servers that export a random-access read interface, and one for servers that only allow for sequential access. For a buffering ASO, an instantiation parameter specifies how large read requests to the server should be. For a non-buffering ASO, each `salloc` operation results in a read request to the server.

write-only ASO classes: These classes are used for write-only streams, where the sink of the stream is a server that exports a read/write interface. We have implemented three ASO classes of this type: one for non-buffered streams, one for servers that export a random-access write interface, and one for servers that only allow for sequential access. For a buffering ASO, an instantiation parameter specifies how large write requests to the server should be. For a non-buffering ASO, each `sfree` operation results in a write request to the server.

read/write ASO classes: These classes are used for read/write streams to servers that export a read/write interface. ASOs belonging to one of these classes have two sub-objects, a read-only ASO and a write-only ASO, where each belongs to one of the classes described above. Depending on the mode of the stream (i.e., read or write), requests are forwarded by the read/write ASO to the appropriate sub-object. We have implemented five read/write ASO classes; the major difference between them is the action taken when the mode is changed.

mapped ASO classes: These classes are used for servers that export a mapped file interface. We have implemented three ASO classes of this type: one for read-only streams, one for write-only streams, and one for read/write streams. These ASO classes cache the file length in the application address space. If data is appended to the file, then the cached file length is changed, and the logical server layer is eventually informed of this change. For a writable stream, an instantiation parameter specifies how often the ASO should inform the logical server that the file length has changed (e.g., on every `sfree`, on every 4 KBytes of new data, etc.). When a read request is made for data past the cached file length, a request is made to the logical server level to determine if the file length has changed.

`Salloc` operations are atomic, or from an implementation perspective, the ASO data is accessed under the protection of a lock. Hence, multiple threads concurrently invoking `salloc` will each obtain different data. Also, the buffer returned by `salloc` will remain valid until either the ASO is destroyed or a `sfree` request for that buffer is made. However, the data returned by `salloc` is not locked. Hence, it is possible for an application to use `sallocAt` to obtain multiple pointers to the same buffer.[8]

6.2.2 Per-thread ASO classes

There is only one ASO class that we have developed specifically to be used on a per-thread basis:

[8] Also, applications in different address spaces may be concurrently accessing the same data. To avoid this, locking LSOs should be used as described in Chapter 3.

basic per-thread ASO class: These objects are used when multiple threads are each accessing a different part of a file. Per-thread ASOs share a common service-specific ASO sub-object. An instantiation parameter specifies how large the `salloc` requests to the sub-object should be[9]. The larger the requests, the less often the per-thread ASO must make requests to its sub-object. On the other hand, it may be necessary that requests to the sub-object be of the same length as requests to the *basic per-thread ASO*, especially if the sub-object is a locking ASO (as described in the next section).

Performance optimizations are possible when an ASO is being used exclusively by a single thread. For example, an ASO accessed exclusively by a single thread does not have to be locked and its member data can be allocated from memory close to the thread using it. An instantiation parameter for all ASO classes indicates whether the ASO will be used by a single or multiple threads.

6.2.3 Locking ASO classes

The Unix I/O read/write interface transfers data to or from an application buffer atomically. With such an interface, a thread that performs a read operation is guaranteed that it has exclusive access to its private buffer, but has no guarantee that other threads are not accessing a different buffer holding the same file data. If such a guarantee is required, then locks must be acquired independently of the read and write operations used to access the file data.

In contrast, the ASI `salloc` and `sfree` operations identify both the data an application is accessing and when it has finished accessing that data. Hence, it seems natural to incorporate locking as part of ASF, where the requested data is locked on `salloc` and released on `sfree`. Possible locking ASO classes include:

local locking ASO classes: With these ASO classes, data is implicitly locked on a `salloc` operation at some fixed granularity and alignment (and released on a `sfree`). For example, data may be locked at a page or record granularity. An instantiation parameter specifies the locking granularity. The locks are only visible to threads in the same address space using the same ASO. Possible ASO classes of this type include: exclusive blocking locks, exclusive spin locks, fair reader/writer locks (blocking or spin), reader preference reader/writer locks, etc.

global locking ASO classes: These classes are similar to the local locking ASO classes, but store locking information in a separate file that is visible to other interested applications.

variable length locking ASO classes: These classes are similar to the ones described above, but they maintain exact information about what data is locked, instead of recording locking state at some fixed granularity. They are used to avoid deadlock that may result from a locking granularity that does not match the amount of data allocated.

sequential locking ASO classes: These classes are used when data is being accessed primarily in a sequential fashion. Instead of keeping detailed information about the locks acquired, `sallocAt` operations are blocked until an `sfree` has been executed for all outstanding `salloc` operations, guaranteeing that there is only one outstanding `salloc` for any data.

It is important to realize that the locking ASO classes described above are advisory only, and do not enforce locking. Hence, it is still necessary to provide locking policies at the logical-server level.

6.2.4 Prefetching ASO classes

In order to tolerate large disk latencies, read requests that go to disk should be issued well in advance of the time that the data is required. Other researchers have proposed techniques for the file system to automatically recognize how data is being accessed so that prefetching requests can be issued by the file system on the applications' behalf [69]. In general, techniques that prefetch file data can only be effective for applications that access file data in a very regular fashion (e.g., sequentially). Recent research has shown that regular memory access patterns can be detected by compilers, and this information can be used to prefetch cache lines into the processor cache [93, 94]. We believe that compilers could also detect file access patterns, and hence, rather than having the file system attempt to recognize the file access

[9] A per-thread ASO can make a single large-grain `salloc` to its sub-object and service multiple fine-grain `sallocs` from the buffer returned. In this case, the fine-grain requests return pointers to different parts of the same large buffer, that is, no copying is done.

pattern, we provide mechanisms for the compiler (or programmer) to inform the file system *a-priori* of the expected file access pattern.[10]

We have developed a number of prefetching ASO classes:

sequential prefetching ASO class: These objects prefetch file data assuming that a file is being accessed sequentially. The amount in advance of the current file offset prefetching is to occur is adjusted at run time to conform to the rate the file data is being accessed and to how long it takes for the system to respond to the requests.

bounded sequential prefetching ASO class: This class is similar to the *sequential prefetching ASO class*, but is used when the application is sequentially accessing some bounded portion of the file data. It can be important to constrain the region being prefetched, since prefetching past the region that will be accessed by a thread will result in overhead and may interfere with other threads. Also, if a locking ASO is used as a sub-object, then bounding the region for prefetching ensures that the prefetching ASO will not request, and hence implicitly lock, more data than the thread will access. An ASO of this class will accept requests to modify the region being prefetched at run time.

general prefetching ASO class: These objects expect from the application (or compiler) a list of file regions (i.e., file offset; region length pairs) that will be accessed in order. For example, a thread that is sequentially accessing every n^{th} row of a matrix can specify this explicitly. The list of file regions can be changed at run time.

ASF allows the threads of an application to use a single prefetching ASO, a prefetching ASO per-thread, or some solution in-between. Hence, it was only necessary to define three ASO classes to handle seven of the eight "representative parallel file access patterns" described by Kotz and Ellis [69].

6.2.5 Matching application semantics

Many files have some underlying semantic structure. For example, the data in a file may be a two or three dimensional matrix of fixed-sized elements. We have defined several ASO classes that each match a particular type of logical file structure and export an interface that matches the application's view of the data in the file. These ASO classes can simplify the programmer task, by closing the semantic gap between the programmers view of a file and the abstraction provided by the file system.

matrix ASO classes: These classes are used by applications that view a file as containing a matrix of fixed-sized elements. Classes of this type could, for example, support matrices of different dimensions, matrices stored column major or row major, or matrices with different sized elements. ASOs of these classes export class specific member functions like `read_row` and `read_column`, in addition to the standard ASI functions.

string ASO classes: These classes treat files as containing strings of text. Successive `salloc` operations return pointers to successive text strings. Classes of this type could, for example, return pointers to the string data as it appears in the file or ensure that strings are always null terminated (simplifying C programs that manipulate text strings).

line ASO classes: These classes are similar to *string ASO classes*, but are used for applications that access file data as text terminated by carriage return or line feed characters.

directory ASO classes: These classes interpret HFS directory information. Successive `salloc` operations return successive directory entries. Classes of this type could, for example, sort entries alphabetically, sort entries by modification time, or provide specific information such as access permissions or file length.

record locking ASO classes: These classes treat a file as a vector of fixed sized records and locks, where the locks and records are interleaved in the file. An instantiation parameter specifies the length of the records. The offset parameter to `sallocAt` is used as a record index rather than as a character offset. On `Salloc`, after acquiring the lock for the requested data, the ASO returns to the application a pointer to just the requested record without the lock.

[10] An alternative to having the compiler inform the file system of the file access pattern is to have the compiler insert prefetch operations into the object code of the application. However, other researchers have shown that prefetching that only takes into account the application and not the state of the system (e.g., the file cache and disk utilization) can result in poor performance [103].

mapping ASO classes: These classes allow applications to specify a mapping function between the bytes in the file and the order in which the application will obtain these bytes. Such mapping functions can simplify applications, like blocked matrix multiply [26].

In some cases, it may be important that data be aligned on particular memory boundaries. For example, some languages always align records on word boundaries. As another example, compilers can produce more efficient code if a buffer returned by an ASO is guaranteed to be aligned on a word or double word boundary. We are currently exploring the possibility of providing ASO classes that guarantee particular memory alignments.

6.2.6 Other ASO classes

In earlier sections we describe ASO classes that were inspired by our implementation of a parallel file system with a general purpose workload. A flexible file system like HFS can provide additional functionality:

compression/decompression ASO classes: These classes compress and decompress file data to reduce disk space, increase the effective disk bandwidth, and/or allow more file data to be cached in memory. ASO classes of this type might use the Lempel-Ziv compression algorithm, designed for whole file access [134], the algorithms described by Burrows *et al*, that allow random file access [13], or application-specific algorithms, such as that used to compress the data base for the Chinook checkers program [114].

encryption/decryption ASO classes: These classes provide for additional security by encrypting and decrypting files as they are accessed. Possible ASO classes of this type query the user for the encryption/decryption key, obtain the key from a secure file, or hard code the key in the definition of the ASO class.

New ASO classes that implement application-specific compression or encryption algorithms can be easily added to ASF, since the ASO classes that do the decompression or decryption need only be shared by applications accessing files in that format. That is, the user does not require the permissions needed to add the algorithm to a system wide library or server.

6.3 ASO management

ASOs are specific to a particular open instance of the file. This is in contrast to the PSOs and LSOs associated with a file that are persistent and global, so every application accessing a file uses the same PSOs and LSOs. A default set of ASOs, specified at file-creation time, is instantiated when a file is opened. The application can also instantiate new ASOs and associate them with the open file instance.

In this section, we describe how the ASOs associated with an open file instance can be modified, and then describe how the default set of ASOs (instantiated when the file is opened) is specified at file-creation time.

6.3.1 Modifying the ASO set

Each ASO class provides a constructor whose parameters include: 1) information required for the policy implemented by the ASO (e.g., with locking ASOs, the granularity of locking) and 2) a list of sub-objects. An application can associate a new ASO to an open file instance by instantiating the new ASO and passing the constructor a pointer to an existing ASO to be used as a sub-object. For example, an application can instantiate a prefetching ASO at run time that is specific to the expected access behaviour, in which case the previously top ASO becomes a sub-object of the prefetching ASO. Also, a multi-threaded application can instantiate an ASO for each thread and pass to each the pointer to the common service-specific ASO sub-object.

When an ASO is instantiated, it makes a `connect` call to each of its sub-objects. This allows the sub-objects to maintain a count of the number of ASOs that reference them.

Each ASO class provides a destructor that decrements its reference count, and, if the reference count is zero, destroys itself and returns the list of its sub-objects. The list of sub-objects allows the library to recursively destroy all ASOs used for that file.

The list of sub-objects returned by a destructor can also used by applications when modifying the ASOs associated with a open file instance. For example, if a prefetching ASO is instantiated by default that is not appropriate, then it is possible for the application to destroy the ASO and instantiate a new one with a different prefetching policy, passing it the sub-objects of the original ASO.

6.3.2 Creating and opening a file

For most streams, ASF chooses a single default service-specific ASO when the file is opened. The choice is based on the type of the stream opened (obtained by making a query to the server) and if the stream is opened for read-only, write-only, or read/write access. When creating a disk file, the application can (optionally) specify up to three default sets of ASOs, one for each of read-only, write-only, and read/write access. These default ASOs can be organized as a tree[11] that is specified using an infix notation. The set is stored in this notation by the logical server layer of HFS.[12]

On file creation, after instantiating the PSOs and LSOs for the file, the creating application also temporarily instantiates the ASOs it wishes to associate with the file, and then calls the ASF function `save_ASOs` to associate these ASOs with the file. The `save_ASOs` function traverses the tree of ASOs in a depth-first fashion and for each ASO calls an ASO-specific implementation of a `get_params` function to obtain the information needed to re-create it. The application can repeat this step three times to associate the three sets of default ASOs with the file.

The class of each ASO in the default set is identified by a integer number, where ASF uses these numbers to index into an array of constructor function pointers. A user can define her own ASO class, and make an ASO of that class part of the default set, by defining a `addAppASOclasses` that ASF will call on program startup to add to the array of constructor function pointers any constructors defined by the application.

6.4 Implementation Issues

We implemented ASF in C with the three layers shown in Figure 6.3. At the top layer, *interface modules* implement particular I/O interfaces. At the bottom layer, *stream modules* provide code specific to particular ASO classes. The middle, *backplane* layer, provides code common to the different ASO classes and maintains the member data of the ASOs.

The simplest way to implement ASOs would be to use an object-oriented language, such as C++, where each ASO would be defined by a language level object. Unfortunately, such an approach would result in excessive overhead for applications that make fine-grain I/O requests (e.g., on a per-character basis), because each ASI operation would be implemented as virtual member function so that each request would incur the overhead of a function call through a pointer indirection.

Our implementation of ASOs in two layers is based on two observations. First, the most-used ASO classes buffer data. Second, code that is independent of the specific ASO class can be used for all performance-critical streams[13] to access buffered data. Hence, we divided our implementation of the ASOs into *stream modules* that implement class-specific code and optionally export a buffer to the *backplane*, and the *backplane* that implements class-independent code and exploits the buffers exported by the *stream modules* to minimize calls to the stream modules.

The interface exported by the *stream modules* consists of the unlocked ASI functions together with the `set_stream_mode` function, and possibly other class-specific functions. Each stream module can also provide the backplane with access to a single buffer, called the *current buffer*, that is shared by the stream module and backplane using a simple producer/consumer protocol.

The interface provided by the *backplane* to the interface modules is the full ASI interface. The majority of backplane code is implemented as C macros[14] that, whenever possible, use the current buffer provided by the stream modules to minimize the number of function calls to the stream modules.

In the remainder of this section we describe our implementation of the backplane and describe some of the interface and stream modules we have implemented.

---

[11] A tree is required (rather than just a list) because some ASOs require multiple sub-objects, such as the global locking ASO classes, where a single ASO has as sub-objects the file that contains the data and a separate file that contains locking information.

[12] An LSO class exists for the purpose of maintaining these trees.

[13] In this context, "performance-critical" means streams that can have high performance (e.g., file I/O and not terminal I/O).

[14] With a macro, the code is inserted by a compiler pre-processor wherever the macro is called. This avoids the overhead of a procedure call and provides opportunities for compiler optimizations.

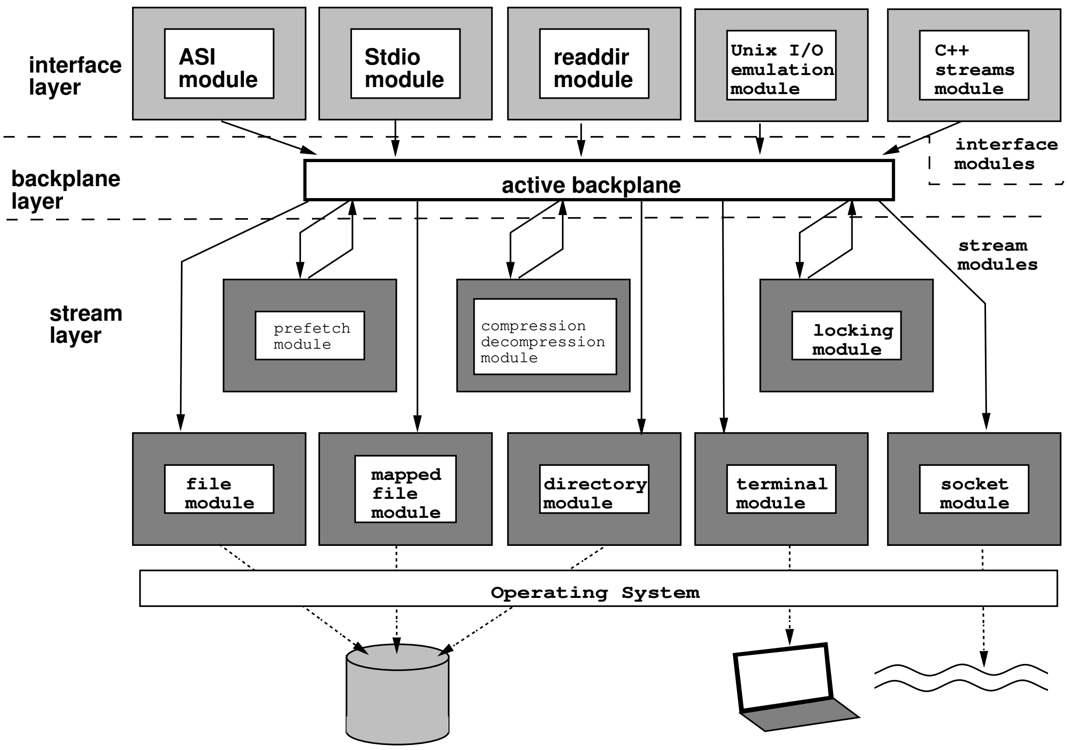

Figure 6.3: Structure of the Alloc Stream Facility

6.4.1 The Backplane

The ASF backplane establishes and manages the communication between the top layer interface modules and the bottom layer stream modules. A *Client I/O State* (CIOS) data structure (Fig. 6.4) and a *current buffer* are maintained for each open stream, and are shared by the backplane and the stream modules. In addition to a lock variable, the CIOS structure contains: 1) function pointers to class specific implementations of the unlocked ASI operations, 2) state used to manage the *current buffer*, and 3) a pointer to class-specific data used only by the stream modules.

To minimize the size of the CIOS structure, only the most common ASI operations (i.e., `salloc`, `sfree`, `srealloc` and `sallocAt`) have a function pointer; other, less common operations, such as `sflush` and `sclose`, are implemented indirectly through the `special` function pointer. The parameters to `special` are: 1) an integer identifying the request being made and, 2) a pointer to a variable length buffer containing request-specific arguments. In addition to a standard set of request types supported by all stream specific modules, `special` can also be used to support class-specific requests, which are not part of the standard interface.

The state used to manage the current buffer contains the following fields: 1) the mode of the stream (i.e., read or write), 2) the number of outstanding `salloc` operations to that buffer, 3) the amount of data (or space) remaining in the buffer, 4) a pointer to the buffer, and 5) a pointer to the data to be used for the next `salloc` operation. For those stream specific modules that do not provide a *current buffer* these fields are set to zero.

The backplane typically implements all ASI operations as macros. The code executed by `salloc` and `sfree` are shown in Figure 6.5. In the common case, when the request can be satisfied by the *current buffer*, `salloc`: 1) acquires the lock for the CIOS structure, 2) checks that there is sufficient data (or space) in the buffer, 3) increments the number of references to that buffer, 4) decreases the amount of data (or space) remaining in the buffer, 5) advances the pointer for the next I/O operation, 6) releases the lock, and 7) returns a pointer to the allocated data. `Sfree`, in addition to acquiring/releasing the lock, simply decrements the reference count to this buffer. If `salloc` cannot be satisfied by the current buffer, then the stream-specific `salloc` function is called, a pointer to which is contained in the CIOS structure.

```
struct CIOS {
    slock    lock ;       /* lock for cios entry               */

    /* class-specific function pointers */
    void * (*u_salloc)   () ; /* unlocked salloc             */
    int    (*u_sfree)    () ; /* unlocked sfree              */
    void * (*u_srealloc) () ; /* unlocked srealloc           */
    void * (*u_sallocAt) () ; /* unlocked sallocAt           */
    void * (special)     () ; /* other class-specific funcs  */

    void *sdata ;   /* handle to class-specific data */

    /* state of current buffer */
    int   mode ;          /* 0 - read mode, 1 - write mode    */
    int   refcount ;      /* outstanding sallocs to buffer    */
    int   bufcnt ;        /* characters/space in buffer       */
    char *bufbase ;       /* pointer to current buffer        */
    char *bufptr ;        /* pointer for next I/O op          */
  } ;
```

Figure 6.4: The Client I/O State structure

In addition to satisfying the request, the stream specific function will typically make a new current buffer available for subsequent `salloc` operations.

As can be seen from Figure 6.5, the amount of code executed for `salloc` and `sfree` is very small in the common case. Despite the fact that they provide functionality equivalent to the stdio `fread` or `fwrite` functions, they have the complexity of the stdio `putc` and `getc` macros (discounting the acquisition and release of the lock). Other *basic* ASI operations are implemented in the backplane in a similar fashion. They use the current buffer when possible and call the corresponding function provided by the stream module otherwise. The unlocked ASI operations (e.g., `u_salloc`) differ only in that they do not acquire locks. The mode variable operations (e.g., `Salloc`) call `set_stream_mode` to set the stream to the required mode and then execute the same code as the *basic* ASI routines.

A problem in dividing the implementation of ASO classes between the stream modules and the backplane is that, while we want the locking of ASO data to be class-specific, the backplane has to lock the CIOS structure, and we would prefer if locks could be acquired and released in the common case without the overhead of a function call. Locking should be class-specific, because only the class-specific code knows if an operation will take a short or long amount of time to determine whether waiting processes should spin or block. Our solution to this problem was to develop the backplane macros shown in Figure 6.6. The important features of these macros are: 1) if the lock is not held, then the lock is acquired and released without making a function call to the stream module, 2) locking can be disabled for an ASO, in which case the cost to determine this is only a single memory load and two conditional branches,[15] and 3) if the lock is held then subsequent lock requests (as well unlock if there are any subsequent lock requests) are handled by the stream module.

Interface modules

The simplest interface module is, of course, the ASI module. It simply exports the same interface as the backplane so that application programs can use it directly. It also exports functions that open and close files and that instantiate ASOs.

Two other interfaces supported by our implementation are the stdio interface and the *emulated Unix I/O* interface.[16] Figure 6.7 shows a slightly simplified version of the algorithm used to implement an *emulated Unix I/O* `read`. `Read`

[15] It is important that on a lock request there be a fast way to determine whether locking is disabled or not, because the interfaces like Unix I/O are implemented using the locked ASI operations, and hence will attempt to acquire locks on each operation.

[16] The emulated Unix I/O interface has the same operations as the Unix I/O interface, but read and write operations are not atomic.

```
void *salloc( FILE *iop, int *lenptr )
  {
    void *ptr ;
    LockAsfStream( iop ) ;
    ptr = iop->bufptr ;
    if( iop->bufcnt >= *lenptr )
      {
        iop->refcnt++ ;
        iop->bufcnt -= *lenptr ;
        iop->bufptr += *lenptr ;
      }
    else
      ptr = iop->u_salloc( iop, lenptr ) ;
    UnlockAsfStream( iop ) ;
    return ptr ;
  }

int sfree( FILE *iop, void *ptr )
  {
    int rc = 0 ;
    LockAsfStream( iop ) ;
    if( (ptr <= iop->bufptr ) && (ptr >= iop->bufbase) )
      iop->refcnt-- ;
    else
      rc = iop->u_sfree( iop, ptr ) ;
    UnlockAsfStream( iop ) ;
    return rc ;
  }
```

Figure 6.5: Code for salloc and sfree.

```
void LockAsfStream( FILE *iop )
  {
    if( iop->lock == NOT_LOCKED )
      {
        if( SWAP( &iop->lock, IS_LOCKED ) == IS_LOCKED )
          acquire_lock( iop ) ;
      }
     else if( iop->lock != LOCKING_DISABLED )
       acquire_lock( iop ) ;
  }

void UnlockAsfStream( FILE *iop )
  {
    if( (iop->lock != IS_LOCKED) &&
      (iop->lock != LOCKING_DISABLED) )
       release_lock( iop ) ;
    else if( iop->lock != LOCKING_DISABLED )
       SWAP( &iop->lock, NOT_LOCKED )
  }
```

Figure 6.6: Backplane routines for acquiring and releasing locks on a CIOS structure. The lock variable indicates whether: 1) the lock is free, 2) the lock has been acquired, 3) locking is disabled, or 4) a class-specific function should be called for both acquiring and releasing the lock. The acquire_lock and release_lock routines invoke class-specific locking functions. SWAP is an atomic operation which sets the lock variable pointed to by its first parameter to the value specified by its second parameter and returns the previous contents of the lock variable.

first calls `salloc` to allocate the data from the stream, then copies the data from the allocated region to the user specified buffer, frees the allocated region, and finally returns to the application the amount of data read.

The code of Figure 6.7 can be executed by different threads concurrently. Both the `salloc` and `sfree` operations acquire (and release) a lock to ensure that the CIOS structure is updated atomically. Also, because the stream offset is advanced by `salloc`, other threads calling `salloc` for the same stream will be given different areas of the buffer. This illustrates a major advantage of using ASI as the interface to the backplane: copying data from the library to the application buffer is performed with the stream unlocked, allowing for a greater degree of concurrency than if the stream had to be locked for the entire read operation.

To translate between Unix I/O stream handles (i.e., integer numbers) and ASI stream handles (i.e., pointers to CIOS structures), the backplane maintains an array of pointers to CIOS structures, where the Unix I/O stream handle is used as an index into this array. We provide functions that translate CIOS pointers to array indexes and vice versa.

The implementation of stdio is very similar to that of *emulated Unix I/O*. The `stdio.h` file that is included with the library defines the `_iobuf` structure as a `CIOS` structure, so that a stdio `FILE` pointer is actually a pointer to the corresponding `CIOS` structure and no conversion between ASI and stdio streams is necessary. Our implementation of the stdio interface is fully source code compatible with other implementations. However, the application writer needs to take care when using stdio operations that constrain buffering such as `setbuf`. These operations are typically used to increase I/O performance, but may actually hurt performance with our implementation, because it increases the amount of copying.

The interface modules are *interoperable* because they do not buffer data; only the stream modules buffer data. Applications can therefore freely intermix operations from different interfaces. For example, an application can use the stdio operation `fread` to read the first ten bytes of a file and then the *emulated Unix I/O* operation `read` to read

```
int read( int fd, char *buf, int length )
 {
   int error ;
   FILE *stream = streamptr( fd ) ;
   if( ptr = Salloc( stream, SA_READ, &length ) )
     {
        bcopy( ptr, buf, length ) ;
        if( !(error = sfree( stream ) ) )
           return length ;
     }
   else error = length ;
   RETURN_ERR( error ) ;
 }
```

Figure 6.7: Read implemented using ASI

the next five bytes. This allows an application to use a library implemented with, for example, stdio even if the rest of the application was written to use Unix I/O, improving code re-usability. More importantly, it also allows the application programmer to exploit the performance advantages of the *Alloc Stream Interface* by re-writing just the I/O intensive parts of the application to use ASI. Because the different interfaces are interoperable, the *Alloc Stream Interface* appears to the programmer as an extension to the other (already existing) interfaces.

6.4.2 Stream modules

In this section we describe some of the more interesting issues in the design of stream modules that implement service-specific ASO classes.

Read/write based stream modules

In this section we briefly describe the main features of the read-only, write-only and read/write stream modules that are used when the source of the stream is a server that exports a read/write interfaces.

Read-only and write-only stream modules that communicate with servers using a read/write interface are implemented similarly to traditional stdio implementations, where a single buffer (the current buffer) is used for the next access by the application to the stream. In addition to this buffer, the stream module keeps a list of (old current) buffers for which there are still outstanding `salloc` operations. It associates with each buffer a file offset and a reference count of outstanding `sallocs`.

For read-only streams, the stream-specific implementation of `salloc`: 1) allocates a new buffer that is, at minimum, large enough to satisfy the request, 2) copies the portion of un-accessed data from the (old) current buffer to the new buffer, 3) performs a read operation to the operating system for the remainder of the data, and 4) updates the CIOS state accordingly. If the reference count for the (old) current buffer is zero then it is deallocated.[17] The stream specific implementation of `sfree` locates the buffer that contains the data being freed and decrements the reference count of that buffer. If the reference count is zero then the buffer is deallocated.

For write-only streams, the stream specific implementations of `salloc` and `sfree` differ from the read-only implementations in that no data is read into a buffer and that a buffer must first be written to a file before it is discarded. In order to ensure that modifications appear in the file in the same order as `salloc` operations, a buffer is not written to the file until all previous buffers have also been written.

[17] As an optimization, if the current buffer is large enough, it is used as the new current buffer instead of being deallocated.

Conceptually, read/write ASOs have two sub-objects, a read-only ASO and a write-only ASO, and all I/O requests are directed to one or the other. This is implemented by having the read/write stream module maintain in its object-specific state (referenced by the CIOS structure) handles to read-only and write-only streams. Depending on the particular read/write stream, different actions are taken when the stream changes mode in order to keep the read-only and write-only streams consistent. For a disk file, the read/write stream module makes `sallocAt` requests (for 0 bytes) to the read-only and write-only streams to make the offset maintained by the two streams consistent. Also, when the mode is changed from write to read, the read/write stream makes a `special` request to the read-only stream to invalidate any read buffers that may contain a copy of file data that has been modified in the file by the write-only stream.[18] For terminals and sockets, no special action needs to be taken by the read/write stream when the mode is changed.

Mapped-file-based stream modules

The mapped-file stream module maps file regions into the application address space where they are treated in the same way as the file buffers described in the previous section. However, since data can be mapped into the application address space read/write (rather than buffered), the read/write stream module based on mapped file I/O is simpler than those based on a read/write interface. Also, writable mapped file-based stream modules do not need to inform the operating system that a page has been modified, since the memory management system will detect this on its own in a system dependent way.

On requesting data from an I/O server, mapped file-based modules typically request much larger amounts of data than modules based on a read/write interface, because with mapped-file I/O no cost for I/O is incurred until the data is accessed. Hence, mapped-file I/O not only reduces copying overhead (as described in Chapter 2) but also reduces the number of server requests[19] and the number of function calls to the stream module (since the *current buffer* provided to the backplane is larger).

Locking a read/write stream

A read/write disk file stream must use a single lock for both reading and writing, since the same file offset is used for both reading and writing. On the other hand, a read/write socket or terminal stream must use separate locks for reading and writing.[20]

The ASI operation `set_stream_mode`, which changes the mode of a stream, returns a stream handle (i.e., an CIOS pointer) that identifies the stream to which subsequent requests should be directed, and the mode-variable ASI operations make a request to `set_stream_mode` before locking the stream. For a disk-file stream, `set_stream_mode` changes the mode of the stream and returns a handle to the read/write stream (i.e., the stream to which the request was directed). Hence, a single lock will be used for reading and writing. For a read/write socket stream, `set_stream_mode` returns a handle to either a read-only socket stream or a write-only socket stream, and hence separate locks will be used for reading and writing.

File I/O on Unix platforms

We have ported ASF to a variety of Unix platforms, and found that the only changes required between the different platforms were in the stream specific modules.

Depending on the platform, file stream modules can be implemented in a variety of ways. For example, we have three different Unix specific implementations: one for systems where the `mmap` mapped file facility[21] is the fastest way of accessing a file, a second for systems that either do not support mapped files or where the `read` and `write` system calls are faster than accesses to mapped files, and a third for `AIX` systems where the `shmget` mapped file facility performs better than `mmap`. This variety in the implementation of stream types illustrates an important aspect of our

[18] In the case of outstanding `salloc` operations (i.e., the read buffer has a reference count greater than zero), the buffer is made unavailable for subsequent `salloc` requests, but is not discarded until the corresponding `sfree` operations are called.

[19] On the other hand, most implementations of mapped files result in the cost of a page fault being incurred every time a new page is accessed. However, it is possible to implement mapped files in such a way that page faults to not occur when the file data is already cached in main memory. The AIX operating system supports mapped files in such a fashion.

[20] We originally did not do this and found that some Unix applications deadlocked because they needed to have outstanding both read and write operations to the same read/write socket stream.

[21] This facility is described in Chapter 2.

facility: a new stream module can easily be added to take advantage of specific characteristics of a particular system without requiring changes to other parts of the facility.

A difficulty with the implementation of the mapped file module on current Unix systems is that when a page past the end of file (EOF) is accessed, the system increases the file length to include the entire new page, thus incorrectly identifying the amount of valid data in the file.[22] To correctly handle files accessed concurrently by multiple applications, ASF can be configured to use Unix I/O writes to write data past EOF. Note, however, that the EOF problem is an artifact of the current mapped file interfaces, and is not a generic problem with mapped files per se. For example, the file length and the number of pages in the file are independent in HURRICANE, so the file length need not be extended when the application uses `salloc` past the end of the file. Instead, the file length is extended (via a request to the LSO) after the application has completed the modification and has called `sfree`.

Advantages of ASF

In this chapter we have described the *Alloc Stream Facility*, the application-level I/O facility for the HURRICANE file system. As with the rest of HFS, we use an object-oriented building-block approach which results in a great deal of flexibility. We have shown how storage objects can be used to prefetch file data, lock file data, and ease the programmer burden by supporting application-specific views of file data.

We wanted to allow fine grain I/O requests to be serviced at low overhead at the application level, which motivated the development of a new I/O interface called the *Alloc Stream Interface* (ASI). ASI has substantial performance advantages over other interfaces. Also, in contrast to most other I/O interfaces, it is suitable for multi-threaded applications.

ASF is used not only as a portion of HFS, but also as the application level library for all I/O services provided on HURRICANE. The object-oriented building-block approach we use allows each I/O service to transfer data into the application address space in a manner most suited for that service. This allows us to minimize I/O overhead for each type of I/O service by minimizing the number of system calls, server interactions, and the number of times data has to be copied from one memory buffer to another.

The flexibility of our approach makes ASF readily portable to other platforms. In addition to running on HURRICANE as a part of HFS, ASF runs independent of any server or kernel support on *SunOS* [63], *IRIX* [118], *AIX* [91], and *HP-UX* [133]. Although each of these systems support some variant of *Unix*, we have found that large improvements in performance can be obtained by adapting the facility to the particular characteristics of each system. For example, the performance improvements obtained on the systems we tested were largely due to the fact that our facility could use the most appropriate mapped file variant on each system.

The ASI interface is designed so that other interfaces can be easily and efficiently implemented using it. Being able to easily add new interfaces is important, since current parallel I/O interfaces are immature, and new interfaces are still being developed.

An application can freely intermix requests to any of the interfaces provided by ASF. This is important, because it allows each part of an application to use a different I/O interface, which means that a programmer can take advantage of a new I/O interface by modifying just the I/O intensive part of her application. Also, it increases code re-usability, because an application can be linked to libraries that use a different I/O interface.

We have concentrated in this chapter (and in this dissertation in general) on-disk files. We believe that with the development of new service-specific ASO classes, ASF can deliver improved performance for other I/O services such as pipes or network facilities. Related work by Maeda and Bershad [85] demonstrates that substantial performance advantages result from moving critical portions of network protocol processing into the application address space and by modifying the networking interface to avoid unnecessary copying. Also, we could easily develop new stream modules that exploit specialized facilities for transferring data between address spaces, such as Govidan and Anderson's memory-mapped stream facility [48] and Druschel and Peterson's *Fbufs* facility [38].

There is currently an effort under way to develop a new implementation of pipes for ASF that exploits shared memory.[23] This shared memory is managed as a circular buffer, with a single producer using ASI operations to

[22] Both AIX and IRIX support options that causes the file to grow every time a page past the current EOF is modified. On other systems (e.g., SunOS), if a page past EOF is accessed, a segment fault results. On these systems, the mapped file module uses `ftruncate` to change the file length before a `salloc` past EOF can proceed.

[23] Our implementation depends on the applications on both ends of the pipe being linked to our facility. Hence it is an open question whether it can be made compatible with existing binaries on Unix systems.

allocate portions of the buffer into which data can be placed, and a single consumer using *ASI* operations to allocate portions of the buffer containing valid data. We expect that the performance of this implementation will improve relative to implementations where the operating system buffers the pipe data, because less data copying and fewer systems calls are required. Because pipes implemented this way do not cause page faults, we expect these advantages to be even greater than those achieved by using mapped-file I/O for disk file accesses.

Chapter 7

HFS Scalability

Our current implementation of the HURRICANE file system runs on a prototype of the HECTOR multiprocessor that has only sixteen processors and seven disks. As such, issues related to scalability were not important for our implementation. However, one of our goals was to design a file system that could scale to large numbers of processors, and all the major design decisions were made with this goal in mind. We believe that HFS can be made to scale to large systems by extending the current implementation, without having to modify the existing library and server code in any substantial way.

The HURRICANE File System is part of a larger effort to investigate operating system design for large-scale multiprocessors. Hierarchical Clustering is an operating system structuring technique developed for scalability [129, 130]. Section 7.1 describes Hierarchical Clustering. Section 7.2 describes how we apply Hierarchical Clustering to the HFS.

7.1 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 [4, 5, 15, 18, 60]. 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, of which operating systems are an example [130].

7.1.1 Scalability guidelines

Unrau used fundamental equations of quantitative performance evaluation to derive a set of necessary and sufficient conditions for the scalability of demand-driven systems [130]. These conditions were then used to derive the following set of guidelines for designing scalable demand-driven systems[1]:

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

[1] Scalability is impossible in a general sense. No matter how a system is designed, there will always be applications that can place demands on the system that will lead to saturation. The guidelines for designing a scalable demand-driven system therefore must take into account the applications making demands on that system.

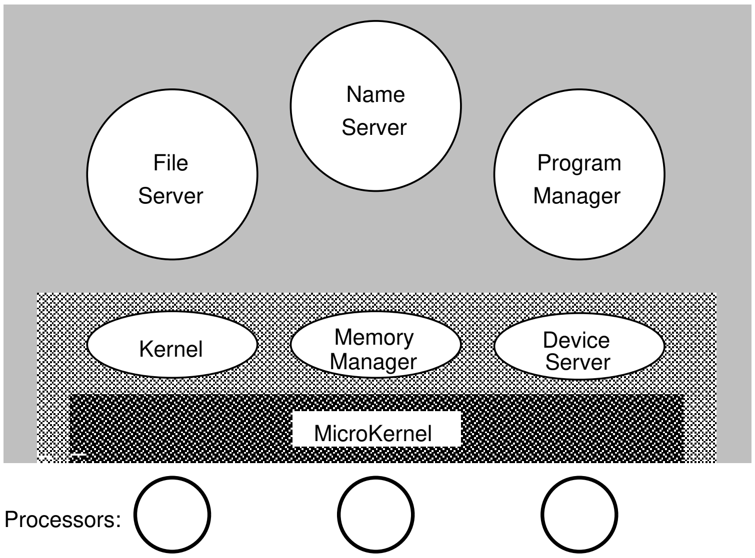

Figure 7.1: A single HURRICANE cluster.

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* [6]. If the overhead of each service call increases with the size of the system, then the system will ultimately saturate. Hence, the demand on any single resource cannot increase with the size of the system. 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 [1, 130], 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-memory 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 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' system requests and disk accesses. 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.

In the mathematical analysis which led to the above design guidelines, scalability was considered asymptotically, that is, as the number of processors approached infinity. At the asymptotic limit only independent requests can be considered, since any dependence between requests will cause an infinitely large machine to saturate. In practical systems, which are finite, the performance of requests that are not independent is also important. Also, in the asymptotic analysis the size of the constants is irrelevant, while with real systems these constants will often dominate performance, and are hence important.

In the next section we describe a new structuring technique for designing operating systems. This technique aids in meeting the design guidelines presented above. Moreover, it attempts to minimize the size of the constants for requests that are independent, while allowing performance to degrade gracefully for requests that are not independent.

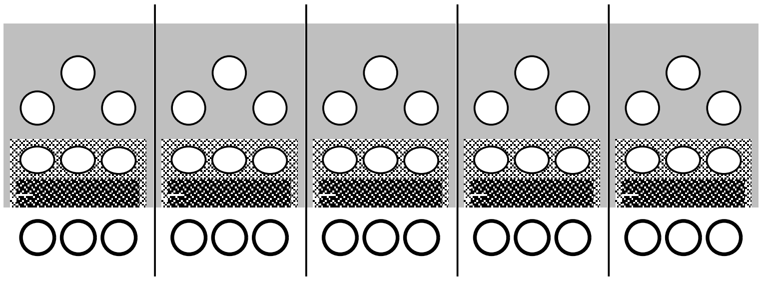

Figure 7.2: Multiple HURRICANE clusters.

7.1.2 Hierarchical Clustering

The basic unit of structuring within Hierarchical Clustering is the *cluster* (Figure 7.1), 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 not more 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 large 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, and are hence located near the requesting process. Third, the hierarchy provides a natural framework for invoking policies that increase the locality of the applications. For example, the system will attempt to run the processes of a single application on the same cluster, and if that fails on the same super cluster. The system will also attempt to 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: if possible a request is handled entirely within a cluster, and if this is not possible, then at the super cluster.[2] Therefore, the cost of non-independent requests does not depend on the size of the multiprocessor, but only on the degree of sharing of the resources required to handle the applications requests.

7.2 Applying Hierarchical Clustering to HFS

In this section we describe how HFS can be scaled to large systems. Section 7.2.1 describes how the structure of HFS can be extended to use Hierarchical Clustering. Section 7.2.2 reviews some of the issues that arise when storage objects are used by servers on multiple clusters and describes how we expect to address these issues. Section 7.2.3 shows how Hierarchical Clustering can be used by the file system on a large system for locality management and load balancing.

7.2.1 Structure of a multi-clustered HFS

The structure of a multi-clustered HFS is shown in Figure 7.3. There is a separate instance of each file system server on each cluster. All client/server requests that cross HFS layers are in the form of PPC requests that are directed to servers local to the client. All requests that cross cluster boundaries are between servers in the same layer of HFS. Cross cluster communication between file system servers is through explicit message passing, so no memory is shared between the servers on different clusters.

Our experiences with the HURRICANE micro-kernel have taught us that the data structures, synchronization strategies, and code structure of per-cluster services must be designed with clustering in mind. For example, on a clustered system it may be necessary for deadlock avoidance to drop locks and later re-acquire them, and the per-cluster code must be structured to facilitate this. HFS was designed with clustering in mind from the start. Hence, we believe that only minor changes to the HFS servers will be required for a multi-clustered system. Each per-cluster server has a private cache of the storage objects it accesses, and most of the server code is unaware that a storage object may also be cached by servers on other clusters. In the next section, we describe how cross-cluster interactions are hidden from the servers by a new type of storage object class defined for this purpose. We also describe how the PSO classes were designed to simplify cache coherence issues when multiple clusters access the same objects.

We expect to make no changes to ASF for a multi-clustered system. In contrast to the HFS servers that cache PSOs and LSOs, the ASF maintains a single application wide instance of each ASO. It would not be useful to replicate ASOs, since they export a stream interface, where the offset for each request depends on the requests that precede it. As described in the previous chapter, a large-scale application should instantiate per-thread ASOs to get scalable performance.

When a thread makes a request to an ASO, any resulting PPC requests to a HFS server will always be directed to a local server, even if the ASO is shared by threads on other clusters. Aliasing techniques are used to have a single PPC handle refer to different, local, servers on each cluster.

7.2.2 Caching storage objects

In a clustered system, several design decisions regarding storage objects accessed by multiple servers are important:

Object placement: Object member data can be statically cached by a single server, migrated to the servers that access it, or replicated to the servers where it is accessed. These policies can be invoked on the entire object member data, or separately on portions of the object member data.

Consistency: If object member data is replicated, then it must be kept consistent with a policy such as invalidate or update. A compromise policy is also possible. For example, replicas in the same super cluster could be updated, while replicas in remote super clusters are invalidated.

[2] For example, if the replicas of a page all lie within a super cluster, then the data structure describing that page will be local to a memory module in that super cluster.

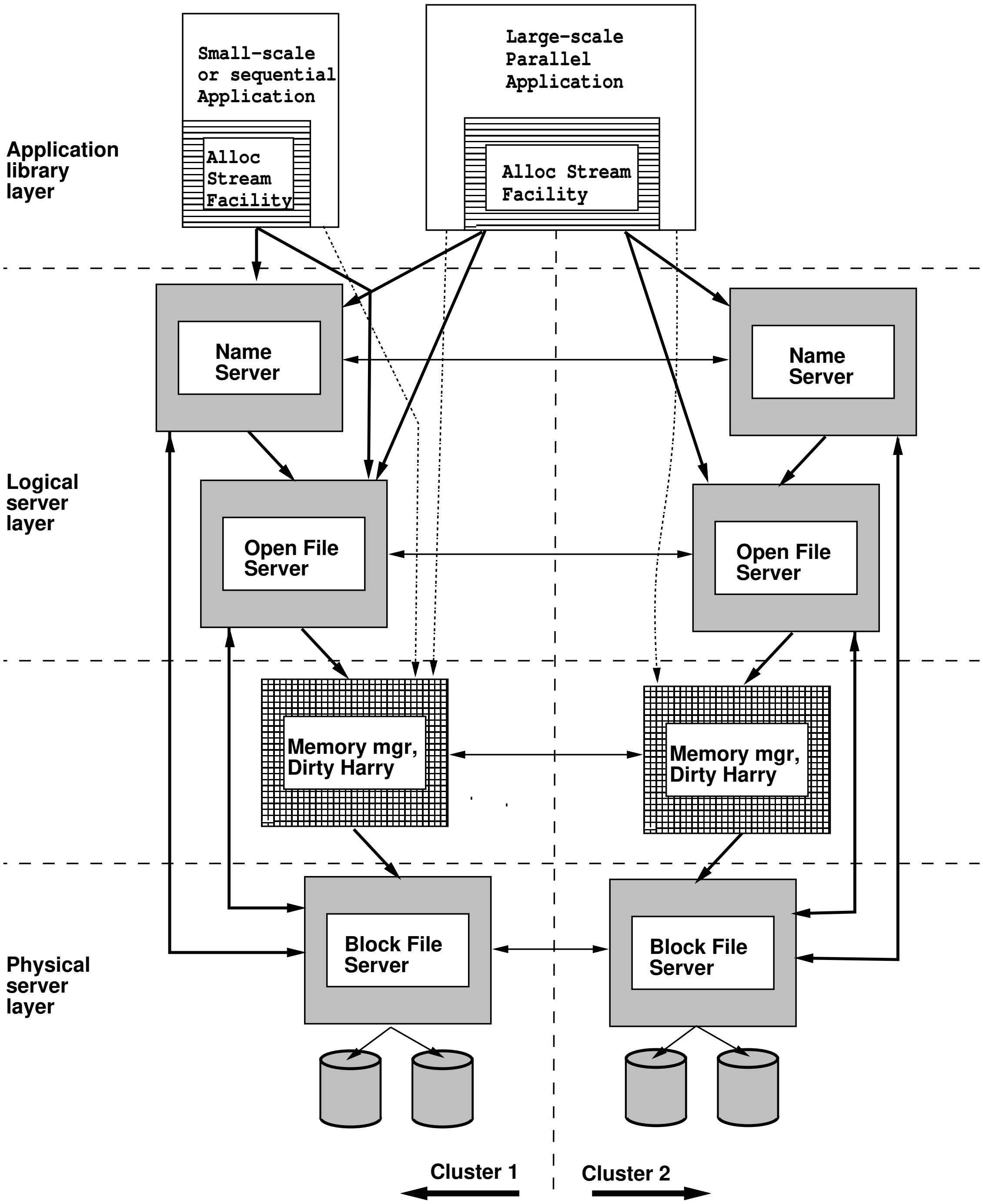

Figure 7.3: The HURRICANE File System. The Name Server, Open File Server and Block File Server are user-level file system servers. The Alloc Stream Facility is a file system library in the application address space. The Memory Manager and Dirty Harry Layer are part of the HURRICANE micro-kernel. Dark solid lines indicate cross-server requests. Dotted lines indicate page faults. Solid lines indicate cross-cluster requests.

Communication: Update and invalidate messages can be communicated by broadcasting them to all servers, or directories can be maintained that limit the number of servers to which a message need be sent.

Clearly, the possible design space is huge, and no particular choice of policies will always yield the best performance. Our solution is based on adding *representative* storage objects to the file system, where a representative is a per-cluster storage object instantiated at run time to act on the behalf of the persistent storage object it represents. When multiple servers access the same persistent storage object, each instantiates a local representative for that object, and directs all subsequent requests to this local representative. The representatives of a persistent storage object cache the member data of the object, implement policies to keep this data consistent, and maintain the state needed to invoke these policies. In effect, representatives hide clustering from both the HFS servers and from the persistent storage objects.

All persistent storage objects have a "home" cluster that is identified by the storage object's token.[3] The representative at the home cluster is different from other representatives in that it is responsible for interactions with the persistent storage object.

To instantiate a local representative of a persistent storage object, a server that has either a representative or the persistent storage object itself is located, and a request to that server is made to obtain the instantiation information. For a system with a single layer of clusters, all such requests are directed to the persistent storage object's home cluster. For larger systems, to avoid the server at a single cluster becoming a hot spot, each super cluster maintains a hint table which identifies the site of some representative local to that super cluster.

Example representative classes

We designed the PSO classes described in Chapter 5 to simplify the representative storage object classes and to minimize the overhead needed to keep the PSO member data cached by representatives on different clusters consistent.

Basic PSOs are restricted to refer to disk blocks on a single disk (and hence a single cluster) and the basic PSO member data is used only for disk scheduling purposes, so only the local BFS will require access to the member data of the basic PSO. Remote representatives therefore only direct requests to the home cluster BFS. Consistency becomes an issue only when a basic PSO is destroyed.

The composite PSO classes are designed so that the member data is infrequently modified and can hence be cached efficiently. For example, all of the member data of a *distribution* PSO is fixed at object instantiation time. For those composite PSO classes where `special` requests can change the object data (e.g., policy-change requests to a *replication* PSO), we expect the changes to be infrequent. The representatives of a composite PSO cache all object member data, and modifications to the object member data result in update messages to all representatives.[4]

PSO member data is not always fully replicated by each representative or exclusively modified at the home cluster. For example, representatives for *fixed sized record store* PSOs (described in Chapter 5) replicate the token of the sub-object, but do not replicate the bit map portion of the object data, since we expect requests to allocate and de-allocate records to be frequent. The remote representatives each maintain a local list of free records, and only make requests to the home cluster representative when the local list is exhausted or overflows.

The persistent storage object class does not necessarily define what the representative classes will be. An example is the *Naming* LSO class used to maintain HURRICANE directories. All directories are stored on disk in a common format, and hence a common LSO class is used for all of the directories. However, the policy used to cache *naming* data should be determined on a per-directory basis, since different directories have different access patterns. Hence, the class of the representative storage objects should be specific to the LSO and not the LSO class. In this case, the representative storage object class to be used is specified by an instantiation parameter to the Naming LSO.

7.2.3 Policies based on Hierarchical Clustering

Part of the attraction of Hierarchical Clustering is that it provides a convenient framework for invoking file system policies. We intend to use this framework for both disk load balancing and locality-management policies.

[3] The home cluster of a storage object is the cluster that contains the home disk of the object map that references it, where, as described in Chapter 4, the home disk is identified using the `omap` field of the token.

[4] For composite PSOs, the remote representative class is derived from the PSO class, and only the code to distribute update requests is new to the representative class.

In the case of load balancing, HFS will maintain both long and short term statistics on the aggregate disk load of the clusters and super clusters.[5] This information can be used by *replication* and *checkpointing* PSOs 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 persistent storage objects of the file are allocated from lightly loaded clusters.

Disk-load information may also be useful to other system services. For example, the system scheduler may use this information when scheduling applications with large data sets that do not fit in memory. If such an application is scheduled to a cluster with low disk utilization, then the files that back the application's memory would be allocated from local disks, and the I/O demands of the application can be met from local rather than remote disks.

Clustering also provides a framework for invoking policies that manage locality. For example, when the policy being used by a *checkpointing* PSO is to direct writes to the most local sub-object, the clustering hierarchy can be used to make a simple definition of "local" (i.e., in same cluster, in same super cluster, etc.).

[5] We expect to maintain these statistics in memory that can be mapped into applications address spaces, so that the application-level library can at low cost exploit this information.

Chapter 8

Fault tolerance

On a very large system, the file system must be able to tolerate and/or recover from component failures in a timely fashion. A parallel supercomputer, made up of many processors, disks, memory banks and network segments, can expect the relatively frequent failure of individual components. With most current systems, the failure of any single component results in the system crashing. Much of the state of a file system persists across a system crash, and must be restored to a consistent state after a system crash. Also, to allow files that contain important information to survive disk failures, the file system must be able to maintain the files, and all file system state required to access them, with redundancy.

This chapter considers two issues: First, how to maintain the state that is needed to recover from disk failures; Second, how to quickly restore the file system to some consistent state after a system crash.

8.1 Maintaining redundant information

HFS maintains redundancy on a per-file basis, where the degree of redundancy and the technique used to maintain the redundant state can vary for different files. This is in contrast to the most commonly used hardware technique, namely RAID-5, where redundancy is maintained on a system wide basis.

There are several advantages of the approach used by HFS over RAID-5. First, we can define a HFS file with no redundancy, to reduce overhead. This is important because for many supercomputer applications I/O is primarily due to insufficient main memory to contain the application's working set (Section 2.5.1) and it is typically not necessary for the files used for this I/O to be fault tolerant. Second, the policy used to maintain redundancy for an HFS file can be tuned to the expected access pattern of the applications that will use it. For example, the number of data blocks used to calculate a parity block can be chosen to match the expected size of write operations. Also, replication can be used if read operations dominate, or if there is excess disk bandwidth. Finally, a RAID-5 disk array stripes data in a fixed fashion across the disks, while an HFS file can be distributed across the disks in a wide variety of ways.

If the file system stores a file with redundancy, then it must also store the meta-data of that file redundantly to be able to recover the file on a disk failure. The relevant meta-data is the member data of the persistent storage objects that implement the file, and the object map that maintains the type and location of the persistent storage objects. Most persistent storage objects are stored on disk by other persistent storage objects, and a single persistent storage object can store the object data of a large number of other objects. Hence, the policies used for storing file data, including policies for maintaining redundancy, can also be used for object member data. When creating the file, the application can select the persistent object that will store the objects that implement the file, and hence the policy (if any) used to maintain this information redundantly. In this case, it is important that many objects can be stored by a single object, because it means that the cost of maintaining redundancy can be amortized over a large amount of data and meta-data.[1]

We plan to provide a per-cluster file for logging each modification to the object map. This file will be striped with distributed parity across the disks connected to the cluster. A similar approach is used by the Zebra file system [51], which combines ideas from log-structured file systems and RAID-5 to maintain file data and meta-data for a distributed system in a fault tolerant fashion.

[1] Because the data of small files is stored directly in the member data of a *small-data* PSO, and the *small-data* PSO is in turn stored by another PSO, small files can also be efficiently stored redundantly by HFS.

Our prototype environment has a small number of disks. Hence, while we have implemented much of the functionality needed to maintain files redundantly, we have not implemented the functionality needed to recover from disk failure given this redundancy.

8.2 Fast crash recovery

When a system crashes, the file system must restore its persistent information to some consistent state, where the state is consistent if all files referenced by directories exist, all files are referred to by some directory, the disk blocks referenced by a file are not marked free, etc. Most file systems do this by performing the time consuming task of examining all the file system state and repairing it if necessary. One of our goals with HFS was to make the task of recovering from a system crash very fast, so that we could restore the file system to some consistent state within seconds of a reboot. Fast crash recovery is important for shared-memory multiprocessors, because to our knowledge no current system is designed to survive component failures. Also, from a pragmatic perspective, our file system is part of an experimental operating system running on experimental hardware, so fast reboots are important.

The log structured file system [97, 110] can recover from crashes quickly. It appends all file system state to the end of a log. Whenever data is flushed to disk, the log structured file system first appends file data, then file meta-data, and finally the superblock to the log. Each superblock refers to a fully consistent snapshot of the file system. If the file system crashes before the superblock is written, then the file system can recover the most recent consistent version of the file system by retrieving the previous superblock. To allow recovery from crashes while writing a superblock, two copies of the superblock are kept on disk, each with a different timestamp. On a system crash, the superblock with the most recent timestamp is the one recovered.

There are disadvantages to the approach taken by the log structured file system. All file data and meta-data are appended in a system specific fashion to the end of a log, allowing for no file specific policies for writing file data or meta-data. Also, it requires that a *segment cleaner* run periodically to de-fragment the disk by collecting disk data from several fragmented segments into a single non-fragmented segment. Even if a large portion of the disk space is unused, it may be inaccessible until the segment cleaner runs.

Our strategy for incorporating fast crash recovery in HFS is based on the realization that the *log* part of the log structured file system was not necessary for fault recovery, and that the log is the cause of the disadvantages stated above. For fault recovery, it is only necessary to avoid overwriting file system meta-data until it is certain that it will no longer be needed, that is, until the superblock has been flushed to disk. This means that the file system must store file system meta-data to a free location (irrespective of its previous location on disk), and ensure that the previous copy of the meta-data not be freed until it is no longer needed.

Therefore, HFS ultimately stores all storage-object member data and object-map data on disk using *random access* PSOs that have the attribute that data is always written to a new location. Also, HFS ensures that freed blocks are not made available for re-allocation until the superblock has been written to disk.

All object data and object-map data is flushed to disk before the superblock is flushed to disk. This ordering ensures that the superblock refers to a consistent file system. As an optimization, HFS goes through two phases during a flush operation. In the first phase, it flushes all meta-data, but services new application requests that might modify the cached meta-data. In the second phase, it flushes meta-data again, during which time any application requests that modify meta-data are not serviced. Once the second phase is complete, HFS writes out all superblock information.

In our current implementation, two copies of superblock information are maintained on each disk. Because HFS spans multiple disks, when recovering from a system crash the system must restore consistent superblocks on each disk. Hence, associated with each superblock is a system wide version number. On crash recovery, the system examines the version numbers on all disks, and the latest version common to all disks is selected as the version of the system to restore. Note, that this means that a new superblock cannot be written to a disk until all other disks have written the previous superblock.

We would like to allow the file state to be flushed to disk independently on each cluster of a large system, so that it can be done when the disks on a cluster are not heavily used. Also, we would like the file system to be able to recover partial file system state, that is, recover any files whose state is complete. Distributed flushing and partial recovery are topics for future research.

Chapter 9

Evaluation

The best way to evaluate a file system whose goal is performance is to implement the file system and measure its performance. Section 9.1 describes the goals and status of our implementation of the HURRICANE File System (HFS). Section 9.2 compares the performance of the *Alloc Stream Facility* (ASF), the application-level library of HFS, to native application-level I/O facilities on a number of Unix platforms. Section 9.3 uses synthetic benchmarks to evaluate the performance of HFS on HURRICANE/HECTOR.

9.1 Implementation goals and status

The goals of our implementation are to demonstrate: 1) that the full HFS architecture can be implemented, 2) that the HFS architecture provides flexibility at low overhead, and 3) that flexibility, low overhead, and cooperation with the application are all important to deliver to a wide variety of applications the full I/O bandwidth of a parallel supercomputer. Time did not permit us to implement all of HFS, in part because the file system was designed to have more functionality than any existing file system, and in part because it was necessary to first implement all of the surrounding infra structure (e.g., disk controller hardware, device drivers, other I/O services, etc.) before we could begin implementing the file system. We first describe the current status of our implementation and then argue why our existing implementation is sufficient to meet the above goals.

9.1.1 Implementation status

Of the three HFS layers described in Chapter 3, we have concentrated on the physical-server layer and the application-level library. Our implementation of the logical-server layer, which is not in the critical path of most read and write requests, is incomplete.

For the physical-server layer, we have implemented all the functionality described in Chapter 5 that is not specific to *Physical-level Storage Object* (PSO) classes, except for the authentication hash table described in Section 5.6.3. We have complete implementations of the three basic PSO classes, the *fixed-sized-record-store* read/write PSO class, and the *distribution* and *striped-data* composite PSO classes.

For the application-level library, we have implemented all the functionality described in Chapter 6 that is not specific to *Application-level Storage Object* (ASO) classes, except for automatic instantiation of default ASOs. We have implemented many of stream-specific ASO classes, a *sequential-locking* ASO class, and an ASO class for line buffered streams.

Our implementation of the application-level library of HFS, called the *Alloc Stream Facility* (ASF), has been ported to a number of Unix systems, including *SunOS* [63], *IRIX* [118], *AIX* [91], and *HP-UX* [133]. In addition to the *Alloc Stream Interface* (ASI), our implementation exports the Unix I/O and stdio I/O interfaces. We have had requests for ASF from many researchers and from dozens of companies, and ASF has been adopted by TERA computer [2] as the I/O library for their parallel supercomputer.

Our implementation of HFS uses mapped-file I/O to minimize copying costs, and hence the HURRICANE memory manager is involved in most requests to read and write file data. We extended the HURRICANE memory manager to support prefetch operations that allow ASF to request a (potentially) large number of pages to be asynchronously fetched from disk. Limitations of the HURRICANE memory manager that we have not addressed are the facts that: 1) flushing dirty pages to the file system is inefficient, and 2) there is no way to obtain contiguous physical page frames so that contiguous disk blocks can be read from disk using a single disk request. Fixing these limitations would involve modifying fundamental data structures of the memory manager.

We have not yet implemented the extensions for clustering HFS described in Chapter 7, but have implemented the fast crash recovery technique described in Chapter 8. Since incorporating this feature into the file system, it has never been necessary to reformat the disks because of the system crashing. Also, this feature was useful for file system debugging. A debug mode was added to the file system which gives the user control over when the superblock is written to disk. With this feature, it is possible to ensure that the file system is recoverable to a previous state even if a bug in the file system (being debugged) writes corrupt file system state to the disks.

9.1.2 Meeting implementation goals

We believe that our existing implementation demonstrates that the full HFS architecture is implementable. For the physical-server layer and application-level library, we have gained sufficient experience to be confident that all storage object classes described in the previous chapters can be implemented without requiring major changes to either the common storage object interfaces or the code for these layers that is not class-specific. While we have less experience with the logical-server layer of HFS, we are also less concerned about this layer, since it is not heavily involved when reading and writing file data, and hence is not performance-critical.

Our existing implementation is sufficient to demonstrate how the architectural choices we have made affect file-system overhead. The three most important of these choices are: 1) implementing files using a possibly large number of simple building blocks, 2) maximizing the functionality of the file system implemented in the application address space, and 3) relying on the use of mapped-file I/O for transferring file data into the application address space. In the later sections of this chapter, we show that there is little overhead when a file is implemented using multiple layers of storage objects, that implementing file system functionality in the application address space can substantially improve performance, and that mapped-file I/O can be used to efficiently deliver file data to the application address space, both when the data is available in the file cache and when it has to be read from disk. We show that for several access patterns, HFS delivers the full disk bandwidth to the application, implying that the file system imposes negligible I/O overhead of its own for these access patterns. Moreover, we show that the processing overhead required by the file system to deliver this performance is very low compared to the other costs required for I/O, such as the memory management overhead and the memory bandwidth consumed by the I/O.

The final goal for our implementation was to demonstrate that flexibility, low overhead, and cooperation with the application are all important to deliver the full I/O bandwidth of a parallel supercomputer to a wide variety of applications. We will show that the flexibility currently provided is important for some access patterns. However, we believe that the only true way to demonstrate that the flexibility of HFS can be usefully exploited by applications is to have real applications use the file system. While we have no results for applications written to exploit HFS, our implementation is now stable enough for application development.[1]

9.2 ASF under Unix

One of the major difficulties in trying to evaluate a file system (or any other component of an operating system) is that it is seldom possible to make direct comparisons between competing alternatives, since the alternatives are typically implemented as a part of a different operating system and on a different hardware platform.

We ported ASF to a variety of Unix platforms, in part to illustrate our claim that the flexibility of our approach makes the file system readily portable, and in part because it allows a portion of the file system to be directly compared to alternatives. We compare the performance of the Alloc Stream Facility against the original stdio and Unix I/O facilities already available on three systems: an IBM R6000/350 system with 32 MBytes of main memory running AIX Version 3.2, a Sun 4/40 with 12 MBytes of main memory running SunOS version 4.1.1, and a SGI Iris 4D/240S with 64 MBytes of main memory running IRIX System V release 3.3.1. To make this comparison, we measured the time to execute:

[1] While the file system as a whole has only recently become stable enough for application development, ASF has been the only I/O library on HURRICANE for the last four years and is in daily use. HFS is currently being used by two other researchers working on HURRICANE/HECTOR.

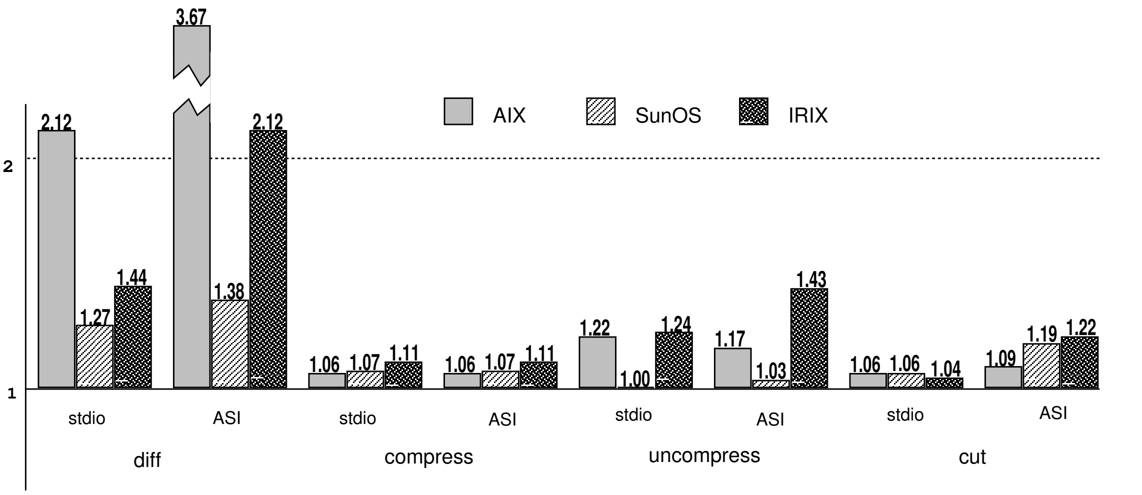

Figure 9.1: Speedup of stdio applications that are 1) linked to the *Alloc Stream Facility* and 2) modified to use ASI. `Diff` compares the contents of two files (identical in our experiments); `compress` and `uncompress` use adaptive Lempel-Ziv coding to respectively compress and uncompress files; `cut` is a Unix filter that removes selected fields from the input file and writes the result to an output file.

1. several programs that use stdio and Unix I/O linked to each systems installed facilities;
2. the same programs, with no source code modifications, linked to ASF (so that they use the stdio and emulated Unix I/O interfaces); and
3. the same programs, with source code modified to use the *ASI* interface directly.[2]

Each system on which we performed these experiments had its own *standard* configuration of ASF. The standard configuration on the *AIX* system uses mapped files based on `shmget` for both input and output.[3] The standard configuration on the *SunOS* system uses mapped files based on `mmap` for input and output. The standard configuration on the *IRIX* system uses mapped files based on `mmap` for input and uses Unix I/O for output. For comparison purposes, we also present the performance of each experiment when the facility is configured to use Unix I/O system calls for both input and output.

Results for each program are given in terms of speedup relative to the same program linked to the machine's installed facilities — that is, the time to run the program linked to the installed facilities divided by the time to run the program using ASF. In these experiments all input and output was directed to files. The numbers we present indicate the expected speedup on an idle system with all file data available in the main memory file cache. To obtain these numbers, we ran each experiment many times. To subtract away the impact of any disk I/O and other applications, we present the smallest numbers measured. If a small number of outliers are ignored, then the average of the many runs was within 10% of the minimum number we use.

stdio applications

Figure 9.1 shows the speedup of several standard Unix programs that use stdio. The programs linked to ASF perform at least as well as those linked to the installed system libraries, and in some cases significantly better. For example, `diff` runs in less then half the time when linked to the *Alloc Stream Facility* on the AIX system. The major reason for the improved performance is the use of mapped files.

[2] Only minor changes to the programs were necessary to adapt them to ASI, typically affecting fewer than 10 lines of code.

[3] In the case of a file opened for write-only access, we first attempt to open it for read/write access to allow the file to be mapped into the application address space, and if this fails we open it for write-only access and use Unix I/O write operations to modify the file.

appl.	AIX		SunOS		IRIX	
	Unix I/O	ASI	Unix I/O	ASI	Unix I/O	ASI
diff	1.00	1.53	0.96	1.31	0.99	1.26
compress	1.00	1.00	1.01	1.01	1.06	1.07
uncomp.	0.99	0.99	0.99	1.00	1.23	1.42
cut	0.87	0.91	0.90	1.03	0.92	1.07

Table 9.1: Speedup for stdio applications when ASF uses Unix I/O operations for both input and output.

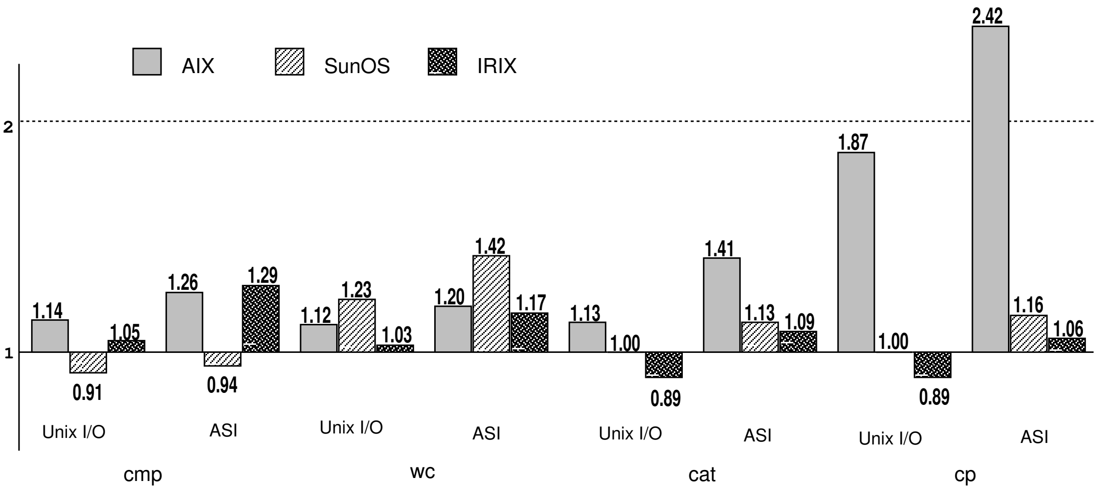

Figure 9.2: Speedup of Unix I/O applications that are 1) linked to the *Alloc Stream Facility* and 2) modified to use ASI. `Cmp` compares the contents of two files; `wc` counts the number of characters, words and lines in a file; `cat` copies the input file to the standard output; `cp` copies one file to another.

Many stdio-based programs show further improvements when they are modified to use ASI directly. For example, `diff` improves by a further 40% from the unmodified program linked to our facility on the AIX system, to be 3.67 times as fast as the original program linked to the installed stdio. This performance gain is due to the fact that data need not be copied to or from application buffers.

Table 9.1 shows the speedup of stdio applications when ASF is configured to use Unix I/O for both input and output. In this case, unmodified stdio applications linked to our facility do not, in general, perform better than when linked to the installed facility. This behavior is expected, because the ASF stdio implementation using Unix I/O is similar to the implementation of native stdio facilities. However, if the applications are modified to use ASI, then they again perform better in several cases.

9.2.1 Unix I/O applications

The Unix I/O interface is specific to the Unix operating system, and for portability reasons, its direct use is generally considered poor programming practice. However, I/O-intensive Unix programs that do large-grain I/O often use the Unix I/O interface for performance reasons, because stdio entails an extra level of copying.

Figure 9.2 shows the performance of four standard Unix programs that use Unix I/O. Surprisingly, these programs unmodified and linked in to ASF so that they use emulated Unix I/O often perform better than if they use Unix I/O directly. For example, `cp` is almost twice as fast when linked to ASF on the AIX system. Thus, on some systems our

appl.	AIX Unix I/O	AIX ASI	SunOS Unix I/O	SunOS ASI	IRIX Unix I/O	IRIX ASI
cmp	0.88	0.97	0.80	0.97	0.84	0.99
wc	0.92	0.97	0.87	1.00	0.89	0.99
cat	0.85	1.00	1.08	1.13	0.73	0.85
cp	0.83	0.95	0.97	0.99	0.71	0.83

Table 9.2: Speedup of Unix I/O applications when ASF is configured to use Unix I/O for both input and output.

application-level library implements the Unix I/O interface more efficiently than the operating system. Moreover, the performance of these Unix I/O applications improves further when modified to use the Alloc Stream Interface. For example, `cp` modified to use ASI runs 2.4 times faster than the original Unix-I/O based version on the AIX system.

As expected, unmodified Unix I/O programs that use the *emulated Unix I/O* interface of ASF perform significantly worse when the facility is configured to use Unix I/O. This is shown in Table 9.2. This behavior is expected, because the *emulated Unix I/O* interface implemented by the application-level library introduces extra overhead. However, we found it surprising that in most cases the original Unix I/O applications could be modified to use ASI (a high level interface) with almost no performance penalty.

9.2.2 Caveats

Locking was disabled for all the experiments performed on Unix systems. This was fair on the AIX and SunOS systems, where the installed versions of stdio did not do locking. However, the comparison of ASF to the installed stdio on IRIX, where locking was performed, may not be fair. An important example of where the comparison to the IRIX system was not fair is the compress and uncompress programs, where the overhead of locking in the installed version of the `putc` and `getc` macros is probably quite high [57].

We have generally made every effort to ensure that the comparisons to installed facilities are fair, but it is difficult to be certain they are. For example, we do not know what size buffer was used by the installed versions of stdio that we compare against. The default buffer size used by ASF for accessing a file was the buffer size returned by `lstat` if available, or otherwise, the BUFSIZ specified in the installed `stdio.h` file.

9.3 HFS under HURRICANE

We have implemented HFS as a part of the HURRICANE operating system running on the HECTOR shared-memory multiprocessor. Section 9.3.1 shows some of the basic file system, disk, and memory management overheads for I/O on this system. Section 9.3.2 demonstrates the performance of the file system for a simple non-I/O stress test. The performance of the file system for two different I/O bound workloads is demonstrated in Sections 9.3.3 and 9.3.4. Finally, Section 9.3.5 shows the file system overhead when many storage objects were used to implement a file.

9.3.1 Basic timings

We present basic performance measurements of the disks on our system, describe the various system software costs of I/O, and show the basic performance of the file system.

I/O hardware

Seven Conner CP3200 disks are connected to the HECTOR prototype. The specifications for these disks are given in Table 9.3 [24]. The on-disk controller sequentially prefetches disk blocks into the on-disk cache. If we assume data is transferred from the disk at the maximum rate for 8 tracks (given 8 heads), and take the rotational latency and the

characteristic	value
track to track seek time	5 msec
rotational speed	3485 RPM
average rotational latency	8.61 msec
disk block size	512 bytes
blocks per track	38
buffer size	64 KBytes
transfer rate from media	1.5 MBytes/sec
transfer rate from cache	4 MBytes/sec
controller overhead	1 msec
number of heads	8

Table 9.3: Connors CP3200 disk drive specifications.

track-to-track seek time of the disk into account (ignoring any cache and controller overhead), then we can expect about 1 MBytes/sec transfer rate for sequential accesses. In practice, we expect cache and controller overhead to reduce this transfer rate further.

On HECTOR, disks are connected directly to processors. Requests to a disk must be initiated by the local processor and all disk interrupts are serviced by that processor. Disks can DMA data to or from any memory module in the system. When a disk reads or writes memory (local or remote) the local processor is blocked from making any memory accesses.

Figure 9.3 shows the maximum throughput of a disk on our system in KBytes/sec and msecs/disk block for different block sizes. To avoid interrupt overhead, we obtained these measurements by polling a register on the SCSI controller that indicates whether an I/O operation has completed. While this experiment avoided the overhead of interrupts, polling the register interfered with the DMA memory accesses by the disk. Best performance was obtained by spinning on a processor register for about 20 μsec between successive reads of the controller register.

Figure 9.3(a) shows that the maximum performance obtained from the disk was about 720KBytes/sec. This maximum performance was obtained by making requests larger than 5KBytes and is independent of the memory to which the request is directed. Unfortunately, the performance was worse when 4KByte disk blocks (the memory manager page size on HURRICANE) were used. Moreover, the performance for 4KByte blocks depends on the target memory; the cost varies from 640 KBytes per-second, to 670 KBytes per-second and 712 KBytes per-second, depending on whether the request was to remote, on-station, or local memory.

Figure 9.3(c) shows that the location of the target memory affected performance when data in the on-disk cache is being accessed. The maximum rate to local, on-station and off-station memory was 1.89, 1.8, and 1.63 MBytes/sec, respectively. For a 4KByte block, the rates were 824, 795 and 780 KBytes/sec, respectively. From this figure, we can calculate that the basic overhead to initiate a disk request was slightly larger that 1.5 msec, which means that when a disk was transferring a disk block to nearby memory the transfer rate was over 2MBytes/sec. Since 2 MBytes/sec is a major portion of the memory bandwidth of a HECTOR memory module, and since the memory modules do not buffer requests,[4] we can expect that when a disk is transferring a disk block to a memory module any processor accessing that memory module will be significantly delayed.

In the remainder of this section, we present measurements of the software overhead for various file system, memory management and micro-kernel operations required for I/O. The numbers were obtained either using a microsecond timer with 10-cycle overhead or by measuring the time to repeat many similar requests. All reported times are for the uncontended case; in the case of memory or lock contention, performance can get arbitrarily worse. In some cases, performance can vary substantially depending on the locality of the data used by the file system or memory manager to service the request.

[4] If a memory access is to a busy memory module, the requesting processor retries the request with, for several retries, a retry time that doubles every time the request fails.

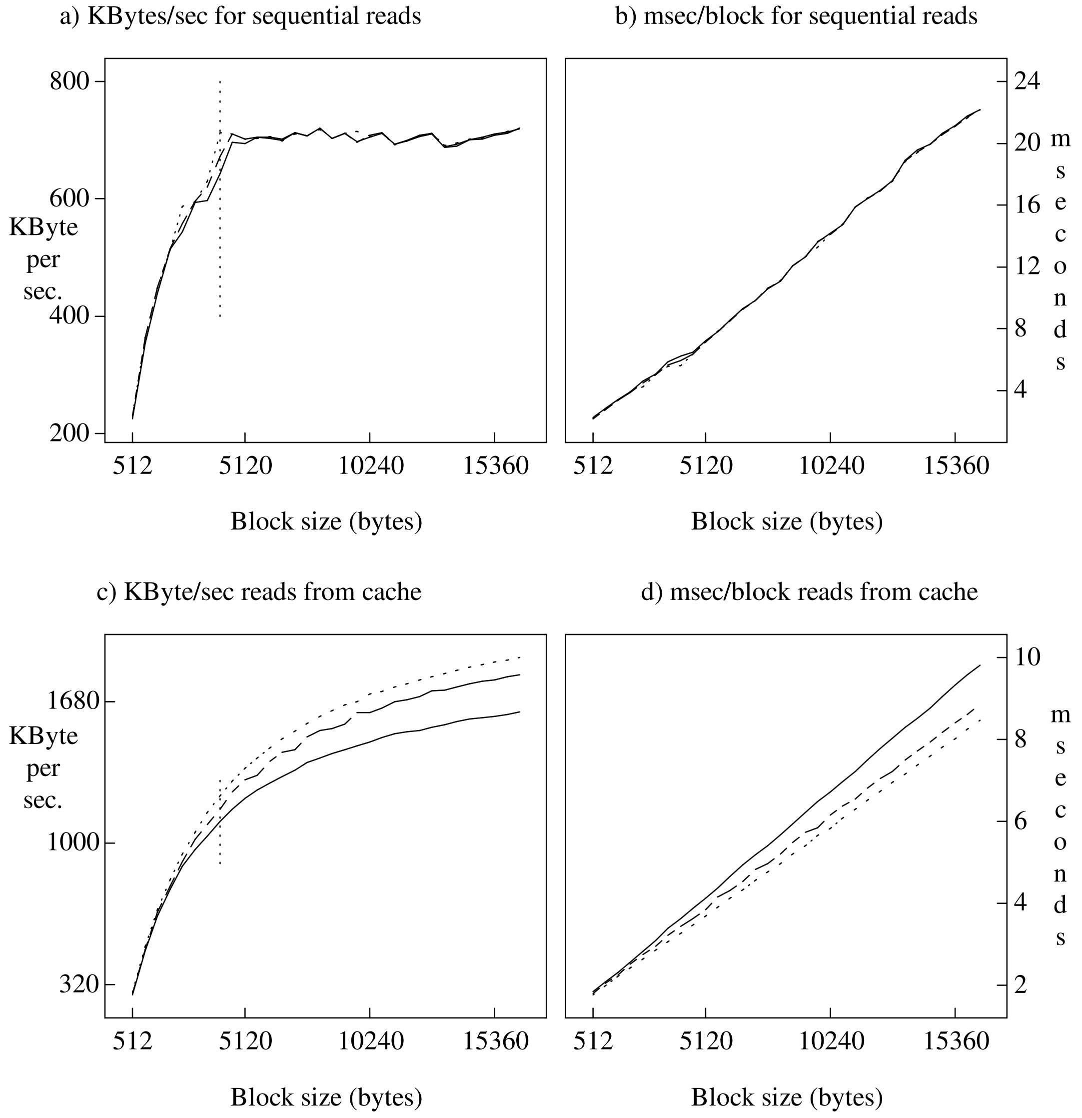

Figure 9.3: Basic performance of Connors CP3200 disk on HECTOR for different block sizes. Figures (a) and (b) show the performance for sequential requests. Figures (c) and (d) show the disk performance when the same block is repeatedly read, and therefore can be expected to be found in the on-disk cache. Solid lines show performance for requests that are directed to off-station memory. Dashed lines show performance for on-station requests, dotted lines show performance for on-processor module requests. The vertical dotted lines indicate the performance for 4096byte disk block requests, corresponding to the HURRICANE page size.

PPC overhead

The cost on HURRICANE to make a PPC request between two application address spaces varies between 35 and 70 μsec depending on how much of the state required for the PPC is in the processor cache. The 88100 TLB has a single supervisor and a single application context. Hence, an application-to-application PPC requires the flushing of the application context from the TLB, with a cost dependent on the amount of application context that must be re-loaded when the application continues. Repeated application PPC requests to the kernel, since they do not require flushing of the TLB, are about 10 μsec less expensive than application-to-application requests. PPC forward takes between 30 and 70 μsecs.

File system overhead

The time for the file system to enqueue and respond to a `read_page` request from the memory manager is 80 μsec in the best case. This best case occurs when the disk is busy and when the file is implemented by a single basic storage object that directly identifies the disk block containing the additional data. The per-object overhead is on average about 20 μsec for most types of storage objects if they are in the meta-data cache. Roughly half of this cost is due to the overhead of locating and locking the entry in the meta-data cache and the remainder is object-specific. If multiple entries hash to the same entry in the hash table, then the cost to access an entry increases depending on the length of the hash chain. For some storage objects, the per-object overhead depends on the block being accessed. For example, for a *basic extent-based* storage object the overhead can grow to 40 μsec if the block is part of the last extent referenced by the object.

When requests are queued for the disk, the file system, on the handling of a disk interrupt (corresponding to the last request), takes about 50 μsec to initiate the next request to the disk. When no requests are queued, the file system takes about 50 μsec to initiate a message to the memory manager indicating that the request has completed on an interrupt from the disk.

When a disk is idle, the per-disk BFS process for that disk must be awakened to initiate a new disk request (Section 4.2). The time to make the wakeup request to the HURRICANE micro-kernel varies between 200 μsec and 600 μsec, depending on the position of the process in the delay queue. The time from when the wakeup request completes to when the per-disk BFS process begins executing is 40 μsec on average if the processor is idle. If another process is running on that processor, then (assuming the file system has a higher priority than the running process) the file system process will be scheduled to run at the next timer interrupt, which occurs every 10 msec.

Memory management overhead

The memory-management overhead to handle a read-page fault when the page is in the file cache is on average 200 μsec on a 16 processor system. If the data is not in the file cache, the memory-manager overhead to initiate a request to the file system is about 800 μsec. The memory management overhead for a `prefetch` request is 60 μsec per request and an additional 200 μsec per page if the page is not in memory, 30 μsec per page if the page is in memory but not in the local page table, and 10 μsec per page if the page is already in the local page table.

When an application makes a prefetch request to the memory manager, it specifies a region of virtual memory, and a single PPC operation can request an unlimited number of file blocks. When the memory manager makes a prefetch request to the file system, it specifies the offset of the first file block in the file, and for each file block the address of the target physical page frame. In our implementation, the maximum number of file blocks the memory manager can request from the file system using a single PPC operation is limited to five.

Basic file system performance

To measure how much of a disk's bandwidth the file system can deliver to a simple application, we created a file implemented by a single *extent-based* PSO on a single disk. We then measured the performance when a single process reads data into memory local to the disk, on-station memory, and off-station memory. The data rate delivered to the application varies between 515 and 540 KBytes/second. There are two reasons for this poor performance. First, whenever a request is sent to the disk, the disk is idle, so the file system incurs the overhead to wake up the per-disk BFS process to initiate the I/O request to the disk. Second, since there is no request waiting for the disk when a request completes, there is a large latency between the time a request completes and the next request is sent to the disk.

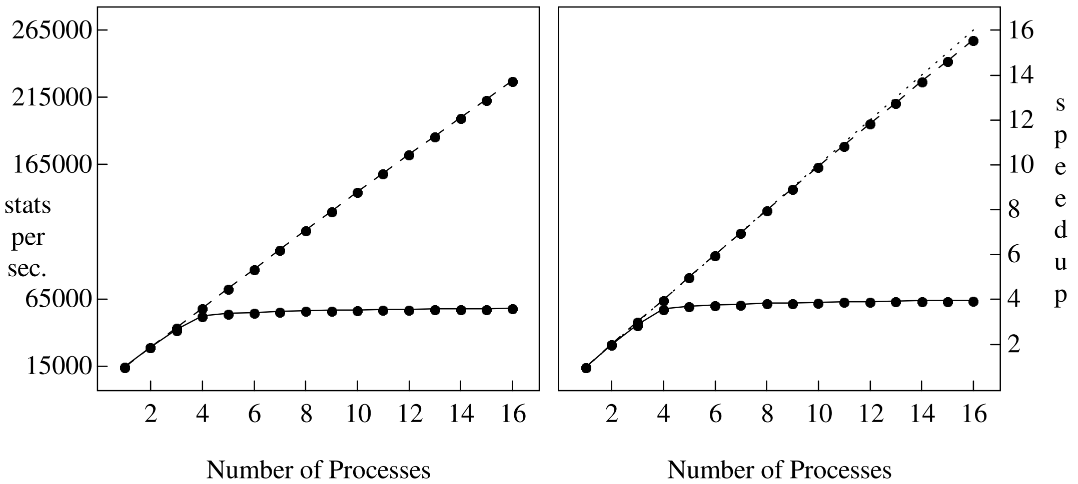

Figure 9.4: Throughput and speedup of concurrent requests to obtain the length of a file. The solid line shows the performance when all requests are directed to a single file. The dashed line shows the performance when each requester is directing requests to a different file.

We ran the same experiment as above but with the reading process always issuing a `prefetch` request to the page succeeding the page it is about to access. In this case, the file system delivers to the application between 640 and 712 KBytes/sec, depending on the target memory. This corresponds to 100% of the available disk bandwidth given the HURRICANE page size. Even with prefetching, there is some delay between the time a disk request completes and the next request arrives. This delay is about 200 μsecs and includes the time to preempt the running process, the file system overhead, and the PPC costs. The reason this delay does not degrade performance is that the disk prefetches data into the on-disk cache. Hence, the disk can tolerate some delay between a request completing and a new request being initiated without loss of performance.

9.3.2 Logical file system operations

Figure 9.4 shows the throughput and speedup when n processes make repeated requests to the file system to obtain the length of an open file. In the uncontended case this operation takes 72.5 μsec on average. Depending on the locality of the requesting process, the process description of the requesting process, the memory containing the hash chain being accessed, and the memory containing the meta-data cache entry, this operation can take as little as 67 μsec or as much as 73 μsec. About 35 μsec of this cost is attributable to the PPC request and the remainder is due to the file system.

To ensure that all processes were making requests concurrently, the numbers shown in Figure 9.4 were obtained by having the processes execute 90000 repetitions and measuring the time between repetition 30000 and 60000. If all requests are to the same file, then the file system saturates at about four processes. If each process accesses a different file then the file system delivers nearly linear speedup until around eleven processes. For sixteen processes the speedup is 15.59.

When multiple processes query the lengths of different files there are no shared locks unless the objects happen to hash to the same entry in the meta-data cache hash table (which does not occur in this example). Linear speedup is not entirely achieved because of memory contention, which is made up of three parts. First, some of the uncached file-system state is located in a single memory module.[5] Second, some of the locks that need to be acquired to service different requests are on the same memory modules.[6] Finally, while file system state is distributed round-robin across the memory modules to balance the load of the file system on the memory modules, the file system must access several

[5] Most of this uncached file state is cacheable, but may not be in the cache because of cache line conflicts.

[6] Having multiple locks on the same memory module can cause a performance problem on HECTOR. When a remote processor executes an atomic SWAP operation to a memory module, all other SWAP operations are excluded from accessing any part of the memory module until the first processor has completed the two remote memory accesses needed for the SWAP.

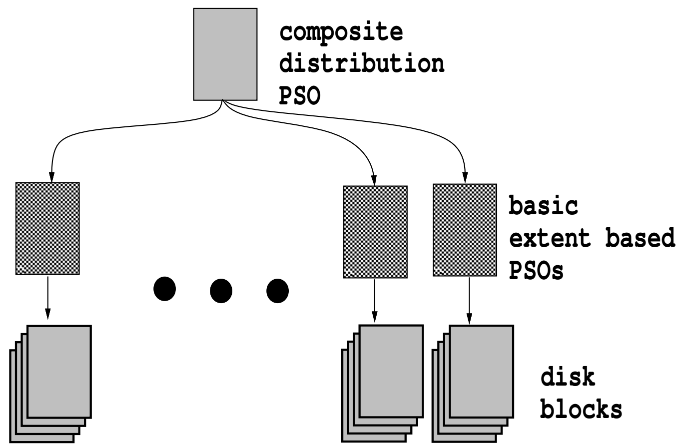

Figure 9.5: Structure of a file distributed across seven disks.

data structures to satisfy the file length requests, and hence requests by different processes access some of the same memory modules, leading to some contention.

We believe that if HECTOR had hardware-supported cache coherence then the file system would perform better with a single process repeatedly making file length requests and would exhibit perfect speedup for multiple processes making these requests. Since each process repeatedly makes request to the same file, there is temporal locality between the processors and the file system state needed to satisfy the processes requests. With cache coherence this temporal locality would automatically be exploited by the hardware.

9.3.3 Parallel file access

This section shows the performance of HFS for when up to 7 processes of an application concurrently access the file shown in Figure 9.5. The file is implemented using a composite *distribution* PSO with seven sub-objects, where each of the sub-objects is a basic *extent-based* PSO on a different disk. Each of the processes of the application access one of the per-disk portions of the file sequentially. The data is read into pages distributed round-robin across the memory modules.

The file system delivers 522 KBytes/sec of data to a single application process with no prefetching. With seven processes the speedup is 6.89 (Figure 9.6). We believe that the degradation is primarily due to delay queue management in the HURRICANE micro-kernel. For this experiment, each disk request is made to an idle disk, and hence the per-disk BFS process must be awakened to initiate each request to the disk.

With prefetching, the file system delivers 646.6 KBytes/sec of data to a single application process. In this case, the speedup for seven processes is perfect. The perfect speedup does not imply that there is no lock or memory contention, but that any increase in overhead is entirely hidden by the cost of accessing the disk. As stated earlier, the disk can handle some latency between requests without any performance degradation, since it uses that time to prefetch disk blocks into its on-disk cache.

9.3.4 Sequential file access of a striped file

In this section, we evaluate the performance of HFS when an application accesses a file whose file blocks are striped round-robin across the disks. The file is implemented using a composite *striped-data* PSO with seven sub-objects. The sub-objects are basic *extent-based* PSOs, each local to a different disk (Fig. 9.7).

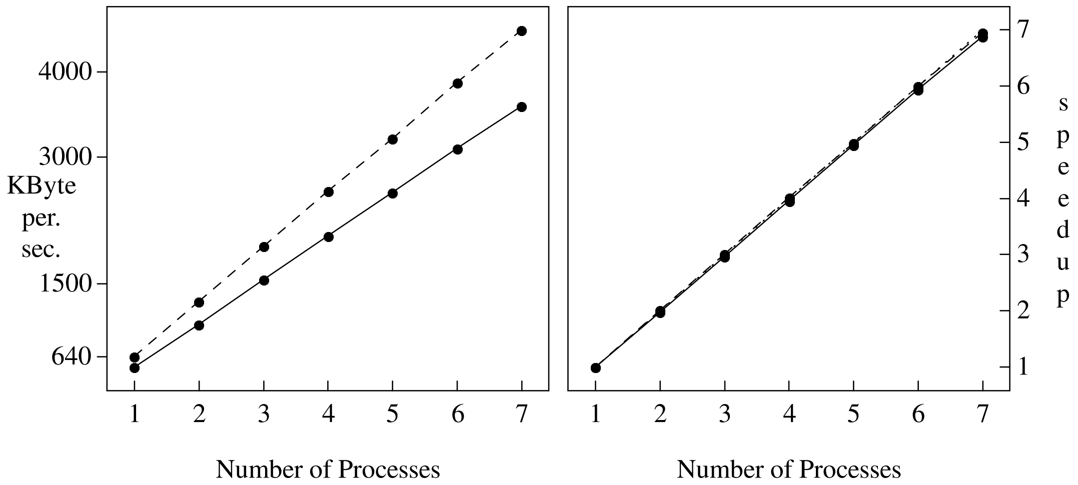

Figure 9.6: Throughput and speedup for concurrent reads of a distributed extent-based file. The solid line shows the performance with no prefetching. The dashed line shows the performance with prefetching of a single block.

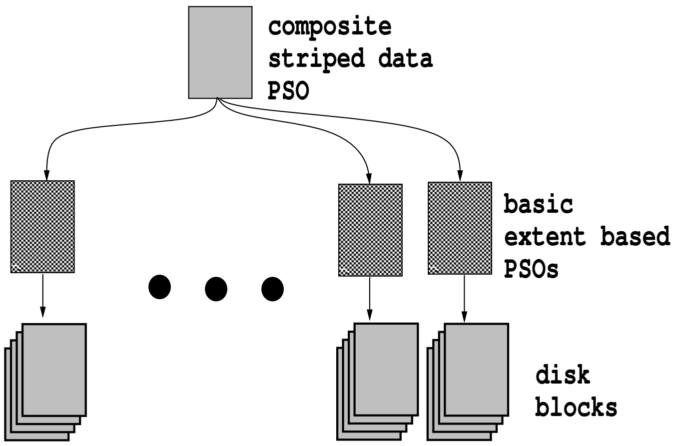

Figure 9.7: Structure of a file striped across seven disks.

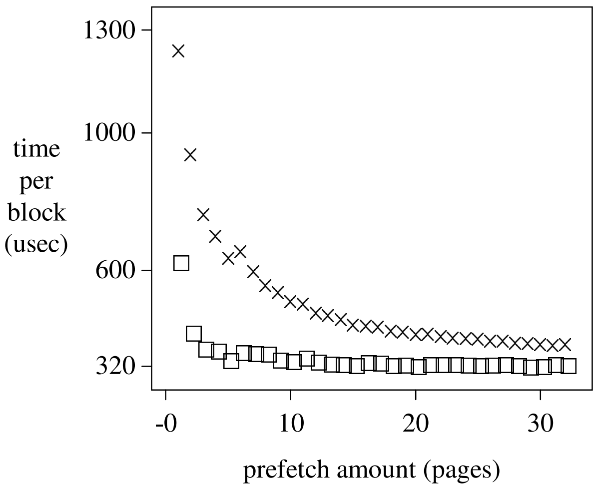

Figure 9.8: The per-block processing overhead to execute prefetch requests for sequential file blocks. The file is striped across seven disks. The bottom line indicates the cost when the disks are busy, while the top line shows the cost for a cold start.

Performance without prefetching

With no prefetching, the maximum total transfer rate from all seven disks is 640 KBytes/sec, which is better than when a single process accesses a file stored on a single disk. We would have expected the performance to be the same, since there is never more than one request outstanding at a time. However, because requests are made to each disk in turn, each disk has a long time to prefetch data into its on-disk cache between successive requests to that disk.

Overhead of prefetching

The software overhead of prefetching can have a substantial impact on performance. Figure 9.8 shows the per-block processing overhead to execute `prefetch` requests — i.e., the system response time for an application `prefetch` request divided by the number of blocks requested. Recall that `prefetch` requests are for sequential file blocks, and in this case a `prefetch` request of *n* blocks will be for file blocks on $n \bmod 7$ disks.

When a disk is idle, the time to execute a single block `prefetch` request is on average 1243 μsec, where about 620 μsec is attributable to delay queue management (to wake up the per-disk BFS process), 260 μsec to the memory manager, 120 μsec to the file system, and about 140 μsec to PPC operations. For comparison, the time to read a 4 KByte disk block is about 3900 μsec when disk blocks are being sequentially read from the disk. Hence, for single block `prefetch` requests to an idle disk, the processing overhead of `prefetch` requests is almost 1/3 the cost of reading a disk block.

If the application issues larger `prefetch` requests, then the per-block overhead drops. This drop is in part because the overhead of the `prefetch` request and the cost of crossing address spaces is amortized over a larger number of blocks. Another contributing factor for the drop in overhead is that for large requests the disks are still busy when the last set of blocks requested are enqueued to the disks, and hence there is no need to wake up the per-disk BFS process. The per-block overhead drops to 387 μsec for a `prefetch` request of 30 blocks. Local minimums occur at multiples of 5 blocks, since the memory manager makes requests to the file system in PPC requests for up to 5 blocks. The time per-block for a 5 bock `prefetch` request is 640 μsec.

If the disks are busy when a prefetch request is initiated, then the time for a single block `prefetch` request is on average 621 μsec. The per-block time for a 5 block `prefetch` request to busy disks is 337 μsec, and for a 30 block `prefetch` request the time decreases slightly to just under 320 μsec.

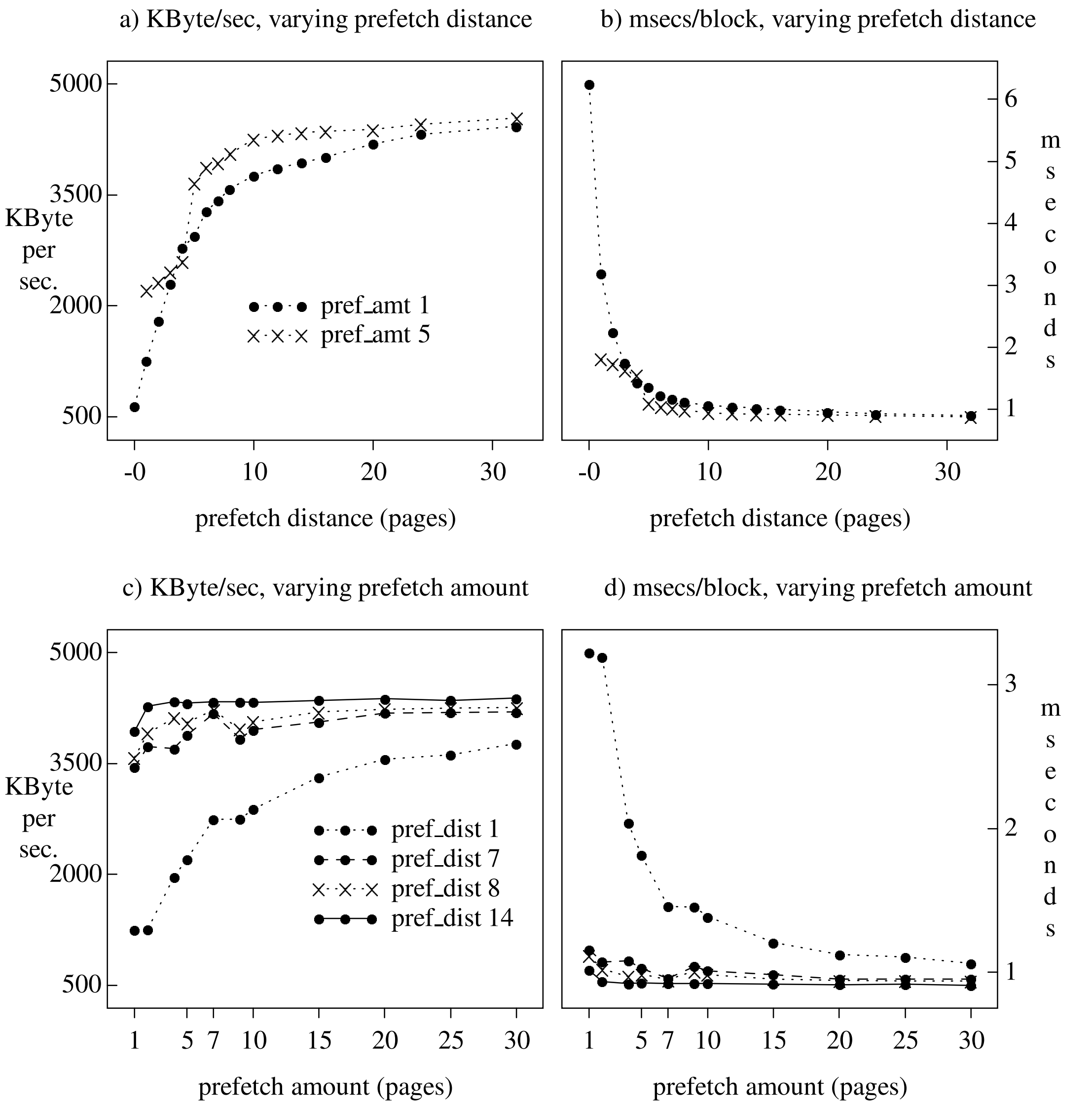

Figure 9.9: Throughput in KBytes/sec and msec/block with prefetching for a file striped across seven disks. Figures a and b show the throughput when the prefetch amount is set to 1 or 5 and the prefetch distance is varied from 0 to 30. Figures c and d show performance when the prefetch distance is fixed to 1, 7, 8, or 14 and the prefetch amount is varied from 0 to 30.

Performance with prefetching

Figure 9.9 shows the I/O performance for a test program that reads a file striped across seven disks with different types of prefetching. The prefetching is determined by two parameters: `pref_amt` is the number of pages (i.e., file system blocks) that the application requests on each `prefetch`, and `pref_dist` is how far (in pages) in advance of the current offset that `prefetch` requests will be made. In these experiments, the application issues a single `prefetch` request for `pref_amt` plus `pref_dist` pages before the first access to a page.

Figure 9.9(a) and 9.9(b) show the throughput in KBytes/sec and msecs/page as seen by the application for different values of `pref_dist` when `pref_amt` is set to 1 or 5. With `pref_amt` set to 1, as `pref_dist` is increased from 0 to 6 performance initially improves linearly and then improves at a slower rate. The reason performance does not increase linearly for this entire range is that the time between successive requests to each disk decreases, and hence each disk has less time to prefetch data into its on-disk cache between successive requests. After 7, further increases in the `pref_dist` continues to result in a gradual improvement in performance, because it is more common for a disk to be busy when a new request to that disk arrives, and hence it is less common to have to wake up the per-disk BFS process.

With `pref_amt` set to five, there is a discontinuity in the performance when `pref_dist` changes from 4 to 5. In fact, for `pref_dist` of 4, performance is worse when `pref_amt` is 5 than when `pref_amt` is 1. For small values of `pref_dist` and `pref_amt`, increasing `pref_amt` does not necessarily improve performance, since the application touches a larger number of pages before issuing a new `prefetch` request, and hence disks may be idle for longer periods of time. However, for large values of `pref_dist` having a larger `pref_amt` is advantageous, since the disks are always busy and hence increasing `pref_amt` reduces processing overhead. For a `pref_dist` set to 30 and `pref_amt` set to 5, the full bandwidth of the seven disks is delivered to the application.

Figures 9.9(c) and 9.9(d) show the throughput represented as KBytes/sec and msecs/page for different values of `pref_amt` with `pref_dist` set to 1, 7, 8, or 14. The figures show discontinuities similar to those of Figures 9.9(a) and 9.9(b). For example, with `pref_dist` set to 1, the same throughput is achieved when the `pref_amt` is set to 1 or 2.

These experiments show the importance of amortizing the per-request overhead of prefetch requests by having the application make requests for large amounts of file data, and in turn justifies the way HFS is layered to facilitate this. The discontinuities in Figures 9.9c and 9.9d that occur with large values of `pref_dist` when `pref_amt` is changed from 1 to 2 are because it is difficult to keep all the disks busy with a `pref_amt` of 1. In this case, the uncontended per-page processing time for `prefetch` is only slightly less than the average per-page response time of the disks when seven disks are kept busy handling sequential reads. Hence, if there is any further degradation due to memory contention, then the system would not be able to keep all disks busy. While with our system a `pref_amt` of 2 is sufficient to keep all disks busy, with more disks the minimum `pref_amt` needed to keep them busy will increase.

In this and the previous section we demonstrated that the file system can deliver the full disk bandwidth for two different file access patterns when the structure of the file corresponds to the application requests. In this section we showed how a single process can obtain the full disk bandwidth when reading data striped across 7 disks. In the previous section we showed how multiple processes each accessing different regions of a file can obtain the full disk bandwidth when each file region is stored on a different disk. To show how important it is to match the disk-block distribution across the disks to the access pattern of the application, we ran the parallel application described in the previous section (seven processes, each accessing a different region of the file) on the striped file used earlier in this section. The performance obtained for this experiment was in the best case less than 30% of the maximum disk bandwidth with seven disks. Similarly, when the distributed file of the previous section (consisting of seven regions, each stored on a different disk) is accessed sequentially by a single process, the maximum performance obtained is that of a single disk regardless of the `pref_dist` or `pref_amt`.

9.3.5 Storage object overhead

To investigate the cost of implementing a file with many layers of storage objects, we constructed a distributed file similar to that described in Section 9.3.3 (consisting of seven regions, each stored on a different disk), except that we added 10 levels of composite *distribution* PSOs between the top level PSO and the basic extent based PSOs (Fig. 9.10). Each of the *distribution* PSOs has only a single *distribution* PSO sub-object, except for the bottom one, which has seven *extent-based* PSO sub-objects. The extra *distribution* PSOs are distributed round-robin across the disks. Figure 9.11 shows the file system performance with and without prefetching when up to seven processes of an application each access a different region of this file sequentially.

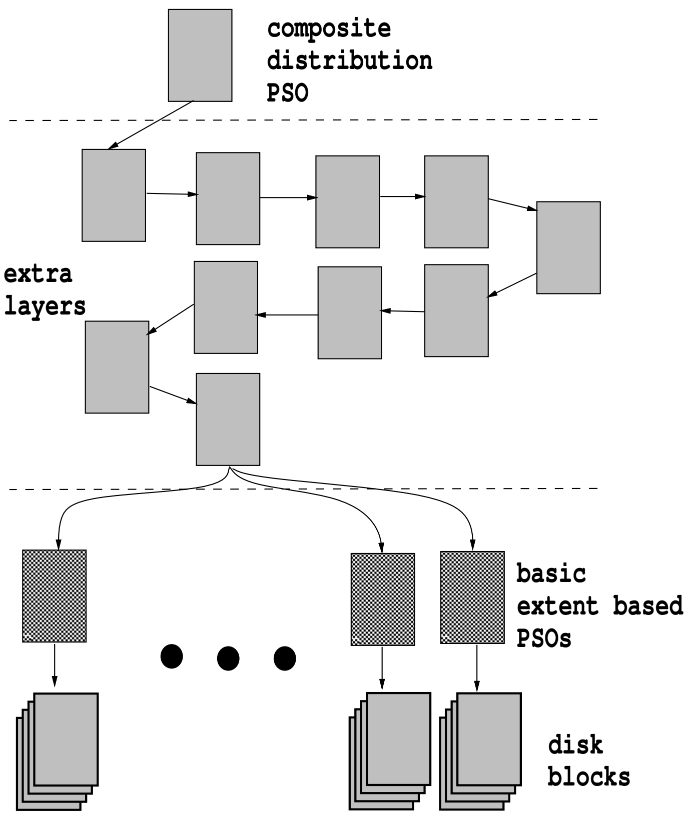

Figure 9.10: Structure of a file distributed across seven disks with 10 extra layers of storage objects

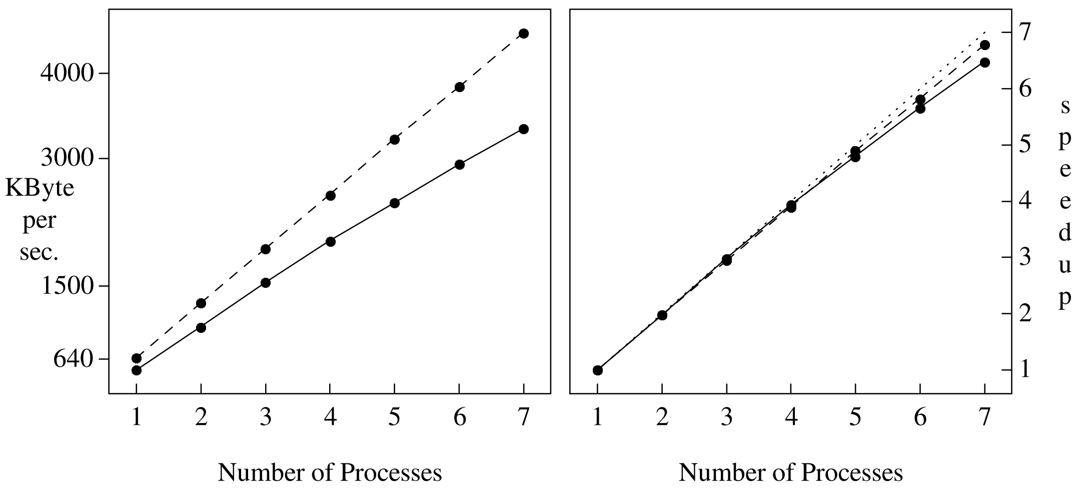

Figure 9.11: Performance for multiple readers of a single distributed file, with 10 extra levels of objects. The solid line shows the performance with no prefetching. The dashed line shows the performance with prefetching of a single block.

The file system delivers 516 KBytes/sec of data to a single application process with no prefetching. This is about 1.2% worse than when no extra objects are used. This degradation is due to the file system overhead to traverse the extra objects, and amounts to about 300 μsecs for each disk block request. The speedup with seven processes is 6.49 compared to 6.89 when no extra objects are used. It is difficult to determine the source of this degradation. One possibility is increased memory contention, since the extra objects end up being distributed across the memory modules when cached in the meta-data cache, and hence there is an increased probability the file system will need to access memory to which a disk is actively transferring data to handle a read request.

The results with prefetching were more surprising and even more difficult to explain than the case without prefetching. With a single process, the file system delivers about 659 KBytes/sec, which is better than the case with no extra objects.[7] We do not have a good explanation for this result, but it is possible that some controller overhead is avoided if new requests arrive at the disk at particularly fortunate times. While the speedup with seven processes is worse than in the case with no extra objects (6.79 compared to 6.96), the absolute performance with seven processes is about the same in both cases. Hence, we are not sure if some fortunate timing is less likely, or if memory and lock contention is causing the degradation.

We conclude that the impact of implementing a file using many layers of storage objects is small. The case considered is extreme, in that the number of layers of *distribution* PSOs used could support a file distributed across more than eight trillion disks, and in that each of the intermediate layers was implemented by a single *distribution* PSO that is accessed for every I/O request. While several layers of PSOs might be required for other purposes, the 12 layers used in this example is unlikely. In any case, while some impact in performance is observable, we believe that this impact is largely due to characteristics of the HECTOR multiprocessor, including the inefficient memory system, lack of cache coherence, and mechanism used to implement atomic operations.

9.4 Summary of results and implementation experiences

In this chapter we describe the goals and current status of our implementation of HFS and presented performance results obtained from this implementation. On Unix platforms we demonstrated that the ASF application-level library can result in significant performance improvements for I/O bound applications. On HURRICANE/HECTOR we have demonstrated that HFS has small processing overhead compared to the other costs associated with I/O, such as the overhead for memory management, crossing protection boundaries, manipulating micro-kernel data structures, and the overhead due to the memory bandwidth consumed by disks. Also, we have demonstrated that a file can be implemented using many layers of storage objects without having a large impact on performance.

We have shown that at least some flexibility in how a file system distributes data across disks is necessary to exploit the disk bandwidth of multiple disks for parallel applications. In particular, for two access patterns we have shown that each requires a different data distribution.

HFS was designed to maximize the amount of file system functionality implemented by its application-level library. Our results on Unix systems demonstrate that implementing functionality in the application address space can result in performance improvements, where the Unix I/O interface implemented by ASF has better performance than the system-call interface implemented by the underlying operating system.

We also designed HFS to facilitate the use of mapped-file I/O. On Unix systems we have shown that mapped-file I/O can result in performance improvements over Unix I/O when accessing data in the file cache. (Further results that demonstrate this are provided in Appendix A.) Our results on HURRICANE/HECTOR shows that our design of HFS allows mapped-file I/O to be used to deliver the full disk bandwidth of multiple disks to the application address space. To the best of our knowledge, HFS is the only file system for which such results have been obtained. In section 9.3.4 we showed how important it is that the physical-layer of HFS be below the memory manager, allowing the application to make requests to prefetch a large number of contiguous file pages into memory so that the per-request overhead is amortized over a large number of disk blocks.

We believe that our current implementation is sufficient to show that the entire HFS architecture can be implemented. Also, the existence of ASF on both HURRICANE and a variety of Unix platforms demonstrates that the flexibility of our architecture makes it easy to port at least one part of the file system to a variety of platforms.

[7] This rate is actually better than what we expect to see from the disks given the results in Section 9.3.1.

Chapter 10

Concluding Remarks

In this dissertation we have described a new file system for shared-memory multiprocessors called the HURRICANE File System (HFS), whose primary goal is to maximize I/O performance for a large set of applications. The design of HFS is based on three principles. First, a file system must support a wide variety of file structures, file-system policies and I/O interfaces to maximize performance for a wide variety of applications. Second, an application should be able to take an active role in determining the structure of its files and the policies that should be applied when its files are accessed. Third, flexibility must come with low processing and I/O overhead.

The flexibility of HFS is achieved by the use of a novel, object-oriented building-block approach, where files are implemented by combining together a (possibly large) number of simple building blocks. The building blocks, called *storage objects*, each define a portion of a file's structure or implement a simple set of policies. The flexibility results both from the ability of the file system to support a large number of storage-object classes and from its ability to combine together storage objects of diverse classes.

HFS enables applications to cooperate with the file system by giving them control over the storage objects used to implement their files. This allows an application to define the internal structure of its files, as well as the policies that should be invoked when the files are accessed. For a persistent file that is accessed by different applications over time, the storage objects used can be modified at run time to conform to specific application requirements.

Three features of HFS contribute to its low overhead. First, to minimize the overhead incurred by crossing address spaces, much of the functionality of the file system is implemented at the application level. Second, to minimize copying overhead, HFS is structured to facilitate the use of mapped-file I/O. Finally, the storage-object interfaces that we have developed minimize the overhead of our building block approach by not requiring that data be copied as control passes from one storage object to another.

Two important issues for a large-scale system are scalability and fault recovery. HFS was designed from the start to be scalable, and we believe that the policy issues specific to a large scale system can be addressed without modification to our basic architecture or implementation. HFS uses a fast crash recovery technique that allows the file system to be restored to a consistent state within seconds of a reboot.

The only way to fully evaluate a file system whose goal is performance is to implement it and measure its performance. We have implemented large portions of HFS as part of the HURRICANE operating system running on the HECTOR shared-memory multiprocessor. Also, the application-level library of HFS has been ported to a number of Unix platforms. Using this implementation, we have demonstrated that the flexibility of HFS comes with low processing and I/O overhead, and showed that the flexibility of HFS allows the full I/O bandwidth of the disks in our system to be delivered to the application address space for two different file access patterns.

While we have been able to demonstrate that our file system architecture is flexible, can give the application control over file structure and policy, and has low overhead, we have not shown that these attributes and the way we provide them are sufficient to achieve our primary goal of maximizing I/O performance for a large set of applications. We will only be able to claim that our file system meets its primary goal when a large body of applications has been written to exploit the novel features of HFS. For this reason, we placed more effort into our implementation than that required to run the simple synthetic stress tests shown in Chapter 9. HFS is now stable enough for application development.

In the remainder of this chapter, we first contrast HFS to other file systems, then describe some of the lessons we have learned from this work, and finally describe our plans for future research.

10.1 Comparison to other file systems

HFS is more flexible than other existing and proposed parallel file system of which we are aware, including CFS [105] for the Intel iPSC, *sfs* [83] for the CM-5, Vesta [25, 26] for the SP2, the OSF/1 file system [136], the nCUBE file system [31], the Bridge file system [35], and the Rama file system [90]. Our current implementation supports or can easily be extended to support all of the policies used by these file systems to distribute file data across the disks. In addition, HFS allows for flexibility in how file data is stored on the disks, allows for flexibility in how file system meta-data that describes a file is stored, and provides for fast crash recovery. The full implementation of HFS will have additional capabilities:

dynamic distributions: HFS is designed so that dynamic policies can be used to distribute requests across the disks, where the load on the disks and the location of the requesting process can be used to determine the target disk for a read or write request. All other existing parallel file systems distribute file data across the disks using static policies, where the offset of the data in the file and the file structure (or some mapping function specified in part by the application) determine the target disk.

latency tolerance: An application will be able to select a prefetching policy on a per-open-file instance and per-thread basis. All other existing file systems either do not prefetch file data or have a single prefetching policy invoked automatically by the file system on a per file or per open file basis.

maintaining redundancy: An application will be able to request on a per file basis the kind of redundancy (for fault tolerance) that should be maintained for its file's data and meta-data. All other existing parallel file systems either do not provide for any redundancy or have a single policy that is applied uniformly to the data and meta-data of all files.

scalability: An application will be able to control how its file's meta-data is cached on the nodes of a large system, and how this cached state is kept consistent. All other existing parallel file systems cache meta-data using a common policy for all files.

The full implementation of HFS will also support a variety of file system interfaces, advisory and enforced locking policies, and compression/decompression policies.

HFS differs from most other parallel file systems in that it has been designed for a shared-memory multiprocessor as opposed to a distributed-memory multicomputer. We believe that the HFS architecture would apply equally well to multicomputers, although substantial modifications to our implementation of HFS would be necessary. One open question in porting HFS to a multicomputer would be whether mapped-file I/O should be used. The fact that the OSF/1 file system for the Paragon multicomputer uses mapped-file I/O [30] suggests that it would not be a problem. However, it is not clear how the mapping facility provided by the Vesta file system [25] (where an application can impose a mapping between the bytes in the file and the order that the file system will provide these bytes to the application) can be efficiently supported for the case where the mapping requires a small amount of data to be transferred from each of a large number of pages.

Conceptually, HFS has more in common with flexible file systems designed for uniprocessors than it does with other parallel file systems. For example, stackable file systems [52] implement a file using "layer" building blocks in the same way that HFS uses storage objects. However, the goals of stackable file systems are very different from those of HFS, and hence their architectures have little in common with that of HFS. The primary goal of stackable file systems is to allow layers to be developed by independent vendors and "stacked" by a system administrator; the layers are potentially available only in binary form. Hence, a single layer is used for a large number of files, the relationship between the layers is determined for all files when the layers are *mounted*, and all interactions between layers must pass through the operating system kernel. In contrast, HFS storage objects are specific to a single file, the relationship between the storage objects is determined on a per-file (or per-open-file instance, or per-thread) basis, and most interactions between storage objects are between objects implemented in the same address space.

10.2 Lessons learned

In this dissertation, we justified the major architectural and implementation choices by taking into account system overhead and software engineering costs. These costs may appear obvious, but in reality they were so only in hindsight.

We recognized them by designing and implementing everything incorrectly at least once. This experience mirrors the advice given by Butler Lampson in his often cited paper *"Hints for Computer System Design"* [76]:

> Plan to throw one away: you will anyhow [10]. If there is anything new about the function of a system, the first implementation will have to be redone completely to achieve a satisfactory (i.e., acceptably small, fast, and maintainable) result. It costs a lot less if you plan to have a prototype. Unfortunately, sometimes two prototypes are needed, especially if there is a lot of innovation.

The final HFS implementation is the product of the lessons learned by: 1) implementing and measuring a prior file system that we eventually discarded, and 2) at least two re-implementations of each part of HFS. These lessons taught us that it is crucial to consider:

copying overhead: Our first file system implementation primarily used the traditional read/write I/O interfaces and made little use of mapped-file I/O. We only realized how important it is to minimize copying costs when we found that the majority of the time spent accessing file data in the file cache was spent copying that data, and that it was impossible to exploit the full disk bandwidth of our platform because of the processor time spent copying data and because of the memory bandwidth consumed copying data.

function call overhead: Our first implementation of ASF did not have the backplane layer that exploits the *current buffer* provided by most ASO classes to reduce the required number of function calls to ASOs. Each ASI request resulted in a function call (through a pointer indirection) to an ASO. This function call overhead made fine grain requests, such as requests for a single character, very expensive.

overhead of crossing address space: We originally thought that the cost to cross address spaces would be negligible compared to the overhead of a disk access. However, when the file system organizes file data on disk so that on-disk caches have high hit rates, then many disk accesses actually have low overhead. The HURRICANE PPC facility was developed when we found that it was impossible to exploit the full disk bandwidth of our system with the original messaging facility provided by HURRICANE. Even with the low overhead of PPC requests, we found that it was important to amortize the cost of crossing address spaces over several blocks by having the application make multi-block prefetch requests to the memory manager and the memory manager make multi-block prefetch requests to the BFS.

meta-data cache overhead: Our implementation first used a single meta-data cache, and we found that 80% of the time spent making a request to a cached persistent storage object was spent (i) locating the object member data in the cache, (ii) enqueuing and dequeuing it from various queues (e.g., the free list of the cache), and (iii) acquiring and releasing various locks. We now have multiple meta-data caches, where the caches have different locking and replacement policies, and the cache to be used for a particular storage object class is chosen to match the demands on objects of that class. For most requests and storage object classes, the time spent executing cache management code is now less than 50% of the total time taken to handle a request.

effectiveness of processor cache: We found that the file system cannot depend on its state remaining in the processor cache between successive requests. This increases the overhead of many operations. For example, it causes PPC operations that cross address spaces to be twice as expensive as when measured using stress tests. A large part of our implementation effort was spent developing techniques to distribute and replicate file system state across multiple memory modules to avoid memory contention; this work would have been unnecessary if the state was typically available in the processor cache. The poor cache hit rate we observed may have been due to the small caches and lack of hardware cache coherence on our experimental platform. However, it may also be the case that applications will always make use of the full processor cache and that the file system can always expect a poor cache hit rate.

memory-bandwidth considerations: We originally thought that memory bandwidth would not be a consideration in our implementation on a small-scale system, but found that (i) the locality of the memory to the disk has an impact on the transfer rate from the disks, and (ii) a single disk transferring data to a memory module results in significant delays to processor requests to that memory module. While disk to memory locality will always be important on large systems, more modern systems support block transfers that allow a processor to obtain the full disk transfer rate from any nearby disk. While the poor memory bandwidth of the HECTOR memory modules (and the backoff algorithm used in the case of contention) is the cause of disk transfers slowing down processor accesses to memory, it is possible that the same effect will occur in other systems. Disks are now available that can transfer data at 20 MBytes/sec from the on-disk cache, and this rate in constantly increasing.

code re-use: A parallel file system is a complicated piece of software, and code re-use is crucial for maintainability. Our first file system was similar to most other file systems in that it had code specific to each type of file (e.g., a replicated file, a distributed file, etc.) and had separate code for storing each type of file system meta-data. We found that the programming effort to add a new type of file was very high with this approach. HFS was less complicated to implement than our first file system, even though it offers better performance and a great deal more flexibility.

on-disk caches: Much research on parallel file systems does not consider how data is stored on the disk, but only how the data is distributed across disks. We found that it is important to consider the tradeoff between having the data for a request spread across multiple disks to exploit concurrency, and having large amounts of data written to sequential disk blocks on a single disk to make effective use of the on-disk caches. Also, surprising results can occur if one ignores the on-disk caches. For example, when we first implemented HFS (and did not employ cylinder clustering techniques), we spent several weeks analyzing the software to discover why the system exhibited poor speedup, and finally discovered that the reason was the prefetching policy used by the on-disk caches. When a single process accessed a striped file, the prefetch requests of the on-disk caches were overlapped in time with the requests by the file system to other disks. However, when multiple processes accessed the same file, the prefetch requests by the on-disk caches delayed the requests by the file system.

interface compatibility: ASF allows the application to interleave requests to different interfaces, even if the requests are directed to a single stream. This allows us, for example, to exploit the performance advantages of ASI by modifying just the I/O intensive parts of an application. No other facility provides similar functionality, and we never considered it important until we developed a new I/O interface and were faced with the task of re-writing applications to use this new interface.

scalability: We originally thought that Hierarchical Clustering would allow us to design the per-cluster system without regard to clustering. Our experience with the HURRICANE micro-kernel taught us, however, that the data structures, synchronization strategies, and code structure of per cluster services must be designed with clustering in mind. For example, on a clustered system it may be necessary for deadlock avoidance to drop locks and later re-acquire them. If this is not considered from the start, then much code will have to be rewritten when clustering is introduced.

10.3 Future work

While we are satisfied with most of the current architecture and implementation, there is still much that requires additional work. First, we are not yet comfortable with any single alternative for having storage objects with multiple sub-objects determine the sub-object to which a `special` request should be forwarded. Second, the logical-server layer of HFS is poorly defined. Third, we need to better define how applications can obtain information describing the structure of the files they access. Fourth, we need to investigate the different techniques described in Section 5.5 to simplify the task of applications creating files. Fifth, we still have to implement the hash table for authentication described in Section 5.6.3, and the technique for associating default ASOs with a file described in Section 6.3.

In the near term, we plan to evaluate our current implementation of HFS using a larger set of file access patterns, such as write mostly, read/write and non-sequential access patterns. Also, we plan on spending more time tuning some features of HURRICANE, like delay queue management, that could be improved to obtain better file system performance. In the longer term, to better demonstrate the flexibility of HFS, we plan to implement a larger number of storage object classes.

More work is needed on the fault tolerance and scalability aspects of HFS. For scalability, we have not implemented any of the *representative* storage object classes described in Chapter 7. For fault tolerance, although we have developed storage objects that maintain redundancy, we have not investigated how the file system can take advantage of this information to recover from a disk failure. Finally, for a large-scale system, we need to extend on our fast crash recovery technique so that the servers on different clusters can save their state independently, and so that the partial state can be recovered on a system crash.

We plan to extend on our current research by studying the I/O demands of parallel applications and the policies that can best handle these demands. Most current file systems constrain how applications request I/O if they are to obtain good performance [33]. Hence, it is difficult to use results obtained from these systems to characterize the "real" requirements of the applications. We hope that the flexibility of HFS will encourage application developers to exploit its features, and plan to extend the monitoring software provided by the file system so that statistics and trace information can be readily obtained. We have worked with Kotz and other researchers to establish a common format for capturing trace information for parallel file systems.

We are currently involved in a project at the University of Toronto to design a new operating system called Tornado for a new shared-memory multiprocessor called NUMAchine, and HFS will be the file system for Tornado/NUMAchine. We plan on studying how the architectural features of NUMAchine impact the file system architecture and implementation. Tornado is being designed under the premise that if a large-scale multi-programmed system is to support high performance I/O, the application, compiler, scheduler, memory manager, and file system must be all be designed with this goal in mind.

Appendix A

The overhead of copying and of mapped-file I/O

We layered HFS to maximize the use of mapped-file I/O and designed the storage object interfaces of HFS to minimize copying. In this appendix we examine basic costs on several systems to determine the overhead of copying and mapped-file I/O on real systems. The four systems we consider are: an IBM R6000/350 running AIX, a Sun 4/40 running SunOS, an SGI Iris 4D/240S running *IRIX*, and the HECTOR prototype running HURRICANE. The AIX system has a 4-way set-associative physical write-back cache with a cache line size of 64 bytes. The Sun system has a direct-mapped virtual write-through cache with a 16-byte cache line. The SGI system has a direct-mapped physical write back cache with a 16 byte cache line[1]. HECTOR has a 4-way set associative physical cache with a 16-byte cache line. On HECTOR, the write-through or write-back policy can be specified on a per-page basis.

A.1 Copying overhead

It has been argued by others that an important goal of HFS, avoiding copying, is of little value. The proponents of this view believe that any advantages obtained, for example, by using mapped-file I/O are attributable to poor implementations of Unix `read` and `write` rather than from avoiding the cost of copying. They believe that with large fast caches, if data is being copied to or from relatively small buffers, the cost of copying the data in the cache is irrelevant compared to the time to transfer the data to or from the memory.

To examine the cost of copying data, we ran four simple experiments that copy data from one buffer to another (Figure A.1). The `copy` test copies data between two four-megabyte memory regions. The `copy to buf` test copies data from a four-megabyte memory region, in page-size chunks, to a single page-size buffer. This second test emulates the copying overhead of page-size read operations that transfer data from the file cache to an application buffer. The `copy from buf` test repeatedly copies data from a page-size buffer to a four megabyte memory region (in chunks of a page size). This experiment emulates the copying overhead of `write` operations that transfer from an application buffer to the file cache. The `copy buf` test repeatedly copies data between two page-sized buffers. This test emulates the copying that occurs between library and application buffers as would occur, for example, in stdio.

To avoid any memory management overhead, we pre-initialized the buffers and memory regions used in the experiments. On all systems, we did various optimizations (e.g., hand unrolling the copy routine) to try to optimize performance, hence the numbers presented can be viewed as upper bounds on performance for (non-hardware-assisted) copying on these systems. The `copy from buf` and `copy to buf` tests were run with the buffers aligned to cache-line or page boundaries. On HECTOR, we ran the copy tests with: 1) caching enabled for both the memory region and the buffer (both write through and write back), 2) caching enabled for the buffer but not the memory region, and 3) caching disabled for both. In all cases, the large memory regions were distributed (at page boundaries) round-robin across 8 memory modules, and the buffer was allocated from local memory.

[1] This specification, although obtained from the SGI technical support group in Toronto, is surprising given the results obtained later. The results would suggest a higher degree of associativity of the cache.

experiment (alignment)	AIX	SunOS	IRIX	HURRICANE cached wr b	HURRICANE cached wr t	HURRICANE buf cached wr b	HURRICANE buf cached wr t	HURRICANE unca-ched
copy	0.086	0.57	0.43	2.31	1.77	—	—	2.04
copy from buf (pg)	0.069	0.48	0.28	1.83	1.29	0.86	—	1.49
copy from buf (cl)	0.069	0.43	0.28	1.83	1.29	0.86	—	1.49
copy to buf (pg)	0.070	0.59	0.27	0.59	1.21	1.25	1.87	1.87
copy to buf (cl)	0.070	0.56	0.27	0.59	1.21	1.25	1.87	1.87
copy buf	0.053	0.42	0.16	0.14				

Table A.1: Measurements of the cost of copying data in milliseconds per page. These three tests repeatedly copy four-megabytes of data. With the `copy` test, the data is copied between two four megabyte memory regions. With the `copy to buf` test, the data is copied from a four megabyte region, in page-size chunks, to a single page-size buffer. With the `copy from buf` test, the data is copied from a page size buffer to a four megabyte memory region (in chunks of a page size). We ran the `copy to buf` and `copy from buf` with the buffer aligned to a page (pg) and to a cache line (cl). The `copy buf` experiment repeatedly copies data between two page-sized buffers. On HECTOR results are obtained using both write-back and write-through caching.

We can see from these experiments that the relative costs of different types of copy operations is system dependent. For example, the alignment of the data being copied affects performance on some systems and not on others. For some of these experiments write-back caching performs best, while for others it is better to either not cache the data or to use a write-through policy.

The most important comparison to made from Table A.1 is the relative performance of the `copy buf` test and the different `copy to buf` and `copy from buf` tests. This comparison shows that on the systems evaluated a substantial portion of the time spent copying data is attributable to the processor (and cache) rather than the memory. For example, on the AIX system the performance of the `copy buf` experiment is only 25% better than the `copy from buf` experiment. The main reason for this is that the memory bandwidth on this machine is high and the cache lines are long. Hence, even though processor cycles are shorter than memory cycles, the processor can be the bottleneck since it transfers data on a word by word basis rather than on a cache line basis. Even on a system like HECTOR, where remote memory accesses can be very slow, the `copy buf` experiment takes nearly 25% of the time of the `copy to buf` experiment.

We do not believe that copying overhead will become irrelevant in future systems, even with the growing gap between processor speeds and memory latency. With relatively small on-chip first-level caches, copying data through the cache will destroy important cache context. Moreover, direct-mapped second-level caches can perform poorly if the alignment is unfortunate (e.g., if a page is copied to a second page where both are mapped to the same cache lines). Finally, copying results in a large amount of data being accessed with no processing time between the memory accesses. If the application accesses data without copying it, than it may be possible to overlap the time to process the data with the time to transfer it to or from the memory (e.g., if prefetching is used for input data [93, 94]).

It is interesting to note that the SPARCserver 400 series machines implement hardware accelerators to reduce the kernel cost of copying data. This accelerator performs memory to memory copies at the full memory bandwidth, bypassing the cache. This indicates that the designers of this machine do not believe caches will make copying costs irrelevant. Since memory bandwidth is a crucial resource in current machines (especially with shared-memory multiprocessors, or systems with parallel disks and high I/O requirements), we feel that software solutions that can avoid copying are better than a hardware solution that reduces the processor cost but does nothing to reduce the memory bandwidth used.

	AIX		SunOS	IRIX	HURRICANE
	shmget	mmap	mmap	mmap	BindRegion
page touch (r)	0.00	0.04	0.56	0.14	0.16
read file 4meg	0.137		1.19	0.53	
read file 2meg	0.137		0.67	0.53	
read page	0.138		0.55	0.43	
read byte	0.084		0.17	0.27	
read 10 pages	0.973		5.39	3.23	
lseek	0.011		0.035	0.038	

Table A.2: Measurements of input using mapped files and Unix I/O in milliseconds per page. The `page touch (r)` test repeatedly maps in a file, touches each page in the file, and then un-maps the file. The `read file` test repeatedly uses `lseek` to move the file offset to the beginning of the file, and then reads each page of the file into a single page aligned buffer. The `read page`, `read byte`, and `read 10 pages` tests repeatedly read the first portion of the file and then use `lseek` to reposition the file offset to the beginning of the file. The `lseek` test, measured in milliseconds per-request, shows the cost of Unix I/O `lseek` operations.

A.2 Reading data in the file cache

Table A.2 compares the cost of accessing a page of a mapped file relative to the cost of Unix I/O `read` and `lseek` operations. On the AIX system, both `shmget` and `mmap` are used to map the file into the application address space. On the other Unix systems `mmap` is used. For comparison purposes, the performance of mapped-file I/O on HURRICANE is shown, in which case the HURRICANE `BindRegion` call is used. (We do not show `read` and `write` times on HURRICANE because these operations are emulated by the ASF library rather than implemented by an operating system server or kernel.)

We consider the time for `lseek` to be a lower bound for a non-trivial system call. The three Unix systems used for evaluation are monolithic systems. Therefore, the basic cost of a system call can be expected to be small (in contrast to some micro-kernel operating systems). This is demonstrated by the low cost of the `lseek` operations on the three systems.

When mapped-file I/O is used (assuming all data is in the file cache), the greatest cost incurred is that to handle page faults. On the three Unix systems, the cost to handle a read-page fault relative to a `lseek` operation varies dramatically. On the AIX system, the page fault time is quite close to the `lseek` time, while with the SunOS and IRIX systems the difference between the two is larger. Without knowing the details of the operating systems and hardware, it is difficult to explain this. However, we believe that it may be possible to improve the performance of read-page faults on these systems, since better performance is achieved under HURRICANE, despite the fact that it runs on slower hardware and does not cache its data.

On all three Unix systems it is more efficient to access large files using mapped-file I/O rather than using `read` operations. On both the AIX and IRIX systems, the cost of reading a page is over 3.75 times the cost of touching a page of a file mapped with `mmap`. The ratio on SunOS (2.32) is also quite high. When `shmget` is used on the AIX system, the cost of accessing a mapped page is zero. AIX exploits the segments of the RS6000 architecture, so that all processes accessing the same file (mapped using `shmget`) share the same segment and data in the file cache can be accessed with no page faults.

A surprising result in Table A.2 is that on SunOS much better performance is obtained if `read` operations are used to (repeatedly) access a 2 Megabyte file rather than a 4 Megabyte file (even though the amount of main memory is sufficiently large to in both cases to keep all data in the file cache). SunOS does better for small files than large files because only a subset of the file cache is mapped into the kernel address space at a given time. If a page in this subset is accessed, then the kernel can service the read request quickly; otherwise, the kernel incurs the cost of a page fault to service the request [117].

We have compared the cost of a page `read` to the cost of a page fault (i.e., touching a mapped page) because

	AIX		SunOS	IRIX	HURRICANE
	shmget	mmap	mmap	mmap	BindRegion
page touch (w)	0.00	0.04	0.54	0.19	0.53
write page	2.78		3.31	0.46	

Table A.3: Measurements of modifying a mapped file versus writing that file in milliseconds per page. The `page touch (w)` test repeatedly maps a four megabyte file, modifies a single character in each page, and then un-maps the file. The `write page` test repeatedly uses `lseek` to move the file offset to the beginning of the file, and then writes each page of the file from a single page aligned buffer.

each brings a page into the application address space. They are not entirely equivalent, however, because the data (in the page) probably resides in the processor cache after the `read` but not if the page is only touched. It is difficult to determine how great an impact this has on application performance, since it depends both on the particular machine architecture (e.g., the size of the cache) and the data access pattern of the application. However, on all three Unix systems, the cost of a page `read` is greater than the combined cost of touching a mapped page and copying a page sized buffer (i.e., the `copy to buf` experiment in Table A.1). As a result, it is always more efficient to use mapped files for accessing large files in read-only mode on these systems.

If data is read in large blocks (e.g., the `read 10 pages` experiment) on the AIX and IRIX systems, or if a small file is repeatedly read on SunOS, then (considering the `copy to buf` experiment in Table A.1) copying overhead is more than 70% of the cost of read operations on all systems. However, with single-page reads on AIX and IRIX, and with reads to a large file on SunOS, copying overhead makes up less than half the cost of the `read` operations.

If buffers are page aligned, then an alternative to copying data on a read operation is for the operating system to map the page copy-on-write. However, this could lead to worse performance if the buffer is later modified. Moreover, we believe that requiring the application to ensure the alignment of its buffers (rather than having the operating system choose the alignment as is done with mapped-file I/O) entails an excessive increase in complexity for many user applications.

A.3 Modifying data in the file cache

Tables A.3 and A.4 show the cost of modifying data in the file cache. The `page touch (w)` test repeatedly maps a four megabyte file, modifies a single character in each page, and then un-maps the file. The `write page` test repeatedly uses `lseek` to move the file offset to the beginning of the file, and then writes each page of the file from a single page aligned buffer. Table A.3 shows the performance of these experiments when the data being modified is already in the file cache, and Table A.4 shows the performance when new pages are being added to the file.

Table A.3 shows that the cost of modifying a page using mapped files is substantially less than the cost of a `write` operation for all Unix systems. In fact, on SunOS and AIX, using mapped files for modifying data already in a file improves performance even more than it did when using mapped files for input.

Table A.4 shows that on the AIX system there is a significant advantage in using mapped-file I/O to append data to a file. However, on the IRIX system using mapped files past the EOF is much more expensive than using `write` operations.

Adding a new page to a file can be expensive, since the first fault to a new page often incurs the cost of zero filling a page of memory. However, depending on the hardware, it is possible that this cost can be very low. For example, with the RS6000 (as well as the MIPS 4400) cache controllers allow the processor to request that a cache line be zero filled. In this case, zero filling a page that is being modified may actually improve performance, since zeroing the cache lines may be less expensive than the cost that is otherwise incurred to load them from memory. As another example, SPARCserver 400 series machines implement hardware accelerators to zero memory.[2]

[2] These accelerators zero the memory rather than the cache, and hence consume memory bandwidth.

	AIX		SunOS	IRIX	HURRICANE
	shmget	mmap	mmap	mmap	BindRegion
page touch (w)	0.21	0.27	1.49	0.73	3.33
write page	2.93		3.00	0.68	

Table A.4: Measurements of adding new pages to a file using mapped files and using `write` operations in milliseconds per page. The `page touch (w)` and `write page` tests are similar to the previous experiment, except that before each iteration `ftruncate` is used to first change the length of the file to 0 (freeing all pages in the file) and then (except on the AIX system where the file grows automatically every time a page past the EOF is touched) to change the file length to four megabytes.

Bibliography

[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] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, pages 1–6, 1990.

[3] James W. Arendt. Parallel genome sequence comparison using a concurrent file system. Technical Report UIUCDCS-R-91-1674, University of Illinois at Urbana-Champaign, 1991.

[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] 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.

[7] Azer Bestavros. IDA-based redundant arrays of inexpensive disks. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 2–9, 1991.

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

[9] R. Bordawekar, J. del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings Supercomputing*, pages 452–461. IEEE Comput. Soc. Press, 1993.

[10] F. P. Brooks. *The Mythical Man Month — Essays on Software Engineering*. Addison Wesley, Reading, MA, 1975.

[11] Edmund Burke. An overview of system software for the KSR1. In *Proceedings of Compcom*, pages 295–299, Feb. 1993.

[12] 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.

[13] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–9, Oct 1992. ACM SIGPLAN Notices.

[14] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4), Fall 1991.

[15] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. Choices, frameworks and refinement. In *1991 Workshop on Object Orientation in Operating Systems*, pages 9–15, 1991.

[16] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 52–63, 1993.

[17] Russel Carter, Bob Ciotti, Sam Fineberg, and Bill Nitzberg. NHT-1 I/O benchmarks. Technical Report RND-92-061, NASA Ames Research Center, Moffett Field, CA 94035-1000, November 1992.

[18] 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.

[19] Peter Chen, Garth Gibson, Randy Katz, and David Patterson. An evaluation of redundant arrays of disks using an Amdahl 5890. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1990.

[20] Peter M. Chen, Edward K. Lee, Ann L. Drapeau, Ken Lutz, Ethan L. Miller, Srinivasan Seshan, Ken Shirriff, David A. Patterson, and Randy H. Katz. Performance and design evaluation of the RAID-II storage server. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 110–120, 1993.

[21] D. Cheriton. UIO: A Uniform I/O system interface for distributed systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[22] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[23] Ann L. Chervenak and Randy H. Katz. Performance of a disk array prototype. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, 1991.

[24] Conner Peripherals, Inc, San Jose, CA. *CP3200 product manual*, v.2 edition, June 1990.

[25] P. F. Corbett, D. G. Feitelson, S. J. Baylor, and J. Prost. Parallel access to files in the Vesta file system. In *Proceeding of Supercomputing*. IEEE Computer Society Press, November 1993.

[26] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993.

[27] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised from Dartmouth PCS-TR93-188.

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

[29] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 15–28, 1993.

[30] R. Dean and F. Armand. Data movement in kernelized systems. In *Usenix Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[31] Erik P. DeBenedictis and Juan Miguel del Rosario. Modular scalable I/O. *Journal of Parallel and Distributed Computing*, 17(1–2):122–128, January and February 1993.

[32] Erik P. DeBenedictis and Stephen C. Johnson. Extending Unix for scalable computing. *IEEE Computer*, 26(11):43–54, November 1993.

[33] J. M. del Rosario and A. Choudhary. High performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[34] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[35] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[36] Peter C. Dibble. *A Parallel Interleaved File System*. PhD thesis, University of Rochester, March 1990.

[37] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum. A comparison of two distributed systems: Amoeba and Sprite. *Computing Systems*, 4(4):353–384, 1991.

[38] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 189–202, 1993.

[39] T. H. Dunigan. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. *Parallel Computing*, 17:1285–1302, 1991.

[40] Samuel A. Fineberg. Implementing the NHT-1 application I/O benchmark. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 37–55, 1993.

[41] High Performance Fortran Forum. DRAFT High Performance Fortran language specification. Technical report, Rice University, 1992.

[42] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[43] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1–2):115–121, January and February 1993.

[44] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings Supercomputing*, pages 388–395. IEEE Comput. Soc. Press, 1993.

[45] Benjamin Gamsa, Orran Krieger, and Michael Stumm. Optimizing IPC performance for shared-memory multiprocessors. Technical Report 294, CSRI, University of Toronto, 1993.

[46] Joydeep Ghosh, Kelvin D. Goveas, and Jeffrey T. Draper. Performance evaluation of a parallel I/O subsystem for hypercube multiprocessors. *Journal of Parallel and Distributed Computing*, 17(1–2):90–106, January and February 1993.

[47] Garth A. Gibson. Designing disk arrays for high data reliability. *Journal of Parallel and Distributed Computing*, 17(1–2):4–27, January/February 1993.

[48] R. Govidan and D.P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. 13th ACM Symp. on Operating System Principles*, pages 68–109, 1991.

[49] 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.

[50] 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.

[51] John H. Hartman and John K. Ousterhout. Zebra: A striped network file system. In *Proceedings of the Usenix File Systems Workshop*, pages 71–78, May 1992.

[52] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM transactions on Computer Systems*, 12(1):58–89, Feb. 1994.

[53] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions of Computer Systems*, 6(1):51–81, February 1988.

[54] Concurrent I/O application examples. Intel Corporation Background Information, 1989.

[55] Paragon XP/S product overview. Intel Corporation, 1991.

[56] David Wayne Jensen. *Disk I/O In High-Performance Computing Systems*. PhD thesis, Univ. Illinois, Urbana-Champagne, 1993.

[57] M. Jones. Bringing the C libraries with us into a multi-threaded future. In *USENIX-Winter 91*, pages 81–91, 1991.

[58] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher. 4.2BSD system manual. 1983.

[59] K. Kennedy. Keynote address. International Conference on Parallel Processing, 1993.

[60] 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.

[61] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14, 1993.

[62] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[63] S. Kleiman and D. Williams. SunOS on SPARC. *Sun Technology*, Summer 1988.

[64] J. Kohl, C. Staelin, and M. Stonebraker. Highlight: Using a log-structured file system for tertiary storage management. In *USENIX Winter Conference*, pages 435–447. USENIX Association, Jan 1993.

[65] D. Korn and K.-Phong Vo. Sfio: Safe/fast string/file I/O. In *USENIX-Summer'91*, 1991.

[66] 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.

[67] David Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[68] David Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994.

[69] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[70] 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.

[71] 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.

[72] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–201–II–204, Boca Raton, FL, August 1993. CRC Press.

[73] 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.

[74] Orran Krieger, Michael Stumm, and Ronald Unrau. Exploiting the advantages of mapped files for stream I/O. In *1992 Winter USENIX*, 1992.

[75] 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.

[76] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–31, January 1984.

[77] A. Langerman, J. Boykin, S. LoVerso, and S. Mangalat. A highly-parallelized Mach-based vnode filesystem. In *Proceedings of the Winter USENIX*, pages 297–312. USENIX Association, 1990.

[78] Richard P. JR. LaRowe and Carla Schlatter Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, 11(2):112–129, Feb 1991.

[79] 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.

[80] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):323–374, December 1990.

[81] Zheng Lin and Songnian Zhou. Parallelizing I/O intensive applications on a workstation cluster: a case study. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, 1993.

[82] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg. The OSF/1 Unix filesystem (UFS). In *Proceedings of the Winter 1991 USENIX*, pages 207–218. USENIX Association, Jan 1991.

[83] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer Usenix Conference*, pages 291–305, 1993.

[84] P. Lu. Parallel search of narrow game trees. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, 1993.

[85] C. Maeda and B.N. Bershad. Protocol service decomposition for high-performance networking. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 244–255, 1993.

[86] M. Malkawi and J. Patel. Compiler directed memory management policy for numerical programs. In *Proc. of the Tenth ACM Symp. on Oper. Syst. Princ*, pages 97–106, 1985.

[87] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3):181–193, August 1984.

[88] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *USENIX-Winter 91*, pages 1–11, 1991.

[89] E.L. Miller and R.H. Katz. Input/output behavior of supercomputing applications. In *Proceedings Supercomputing*, pages 567–76. IEEE Comput. Soc. Press, Nov 1991.

[90] Ethan L. Miller and Randy H. Katz. RAMA: A file system for massively-parallel computers. In *Proceedings of the Twelfth IEEE Symposium on Mass Storage Systems*, pages 163–168, 1993.

[91] M. Misra, editor. *IBM RISC System/6000 Technology*, volume SA23-2619. IBM, 1990.

[92] James Morris, Mahadev Satyanarayanan, Michael Conner, John Howard, David Rosenthal, and F. Donelson Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[93] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Systems Laboratory, Stanford, CA 94305, March 1994.

[94] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[95] J. Nicholls. *The Structure and Design of Programming Languages*, chapter 11, pages 443–446. Addison-Wesley, 1975.

[96] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. of the Summer USENIX Conference*, pages 247–256, June 1990.

[97] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM press, Operating Systems Review*, 23(1):11–28, 1989.

[98] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[99] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings 10th ACM Symposium on Operating System Principles*, 1985.

[100] B. K. Pasquale and G. C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings Supercomputing*. IEEE Comput. Soc. Press, 1993.

[101] David Patterson, Peter Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). In *Proceedings of IEEE Compcon*, pages 112–117, Spring 1989.

[102] 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.

[103] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. Informed prefetching: Converting high throughput to low latency. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 41–55, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[104] J.K. Peacock. File system multithreading in system V release 4 MP. In *Proceedings of the Summer 1992 USENIX*, pages 19–29. USENIX Association, June 1992.

[105] 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.

[106] P.J. Plauger. *The Standard C Library*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[107] A. Reddy and P. Banerjee. Evaluation of multiple-disk I/O systems. *IEEE Transactions on Computers*, 38:1680–1690, December 1989.

[108] A. L. Narasimha Reddy and Prithviraj Banerjee. A study of I/O behavior of Perfect benchmarks on a multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 312–321, 1990.

[109] Mendel Rosenblum and John Ousterhout. The LFS storage manager. In *Summer 1990 USENIX Conference*, pages 315–324, Anaheim, CA, June 1990. USENIX Association.

[110] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15. Association for Computing Machinery SIGOPS, October 1991.

[111] P. J. Roy. UNIX file access and caching in a multicomputer environment. In *USENIX Mach III symposium proceedings*, pages 21–37, April 1993.

[112] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[113] M. Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, 18(1):1–8, Jan 1992.

[114] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[115] David S. Scott. Parallel I/O and solving out of core systems of linear equations. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 123–130, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[116] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a Log-structured file system for UNIX. In *USENIX Winter Conference*. USENIX Association, Jan 1993.

[117] Bill Shannon. Implementation information for Unix I/O on SunOS. personal communication, 1992.

[118] Silicon Graphics, Inc., Mountain View, California. *IRIX Programmer's Reference Manual*.

[119] Tarvinder Pal Singh and Alok Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, June 1994.

[120] John A. Solworth and Cyril U. Orji. Distorted mirrors. *Journal of Distributed and Parallel Databases*, 1(1):81–102, January 1993.

[121] I. Song and Y. Cho. Page prefetching based on fault history. In *USENIX Mach III symposium proceedings*, pages 203–213, April 1993.

[122] C. Staelin and H. Garcia-Molina. Smart filesystems. In *Winter USENIX*, pages 45–51, 1991.

[123] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1992.

[124] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition edition, 1991.

[125] 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.

[126] Andrew Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory Sharp, Sape Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[127] *The Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, October 1991.

[128] 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.

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

[130] 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.

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

[132] B. Walker, G. Popek, R. English, C. Kline, and G. Theil. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 49–70. Operating Systems Review, October 1983.

[133] S. Wang and J. Lindberg. HP-UX: Implementation of Unix on the HP 9000 series 500 computer systems. *Hewlett-Packard Journal*, 35(3):7–15, March 1984.

[134] Terry A. Welch. A technique for high performance data compression. *Computer*, 17(6):8–19, June 1984.

[135] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[136] 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.