Appeared in ACM Trans. On Computer Systems, 15(3), August 1997, pp. 286-321.

HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions

Orran Krieger
IBM T.J. Watson Research Center
and
Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto

The Hurricane File System (HFS) is designed for (potentially large-scale) shared memory multiprocessors. Its architecture is based on the principle that, in order to maximize performance for applications with diverse requirements, a file system must support a wide variety of file structures, file system policies and I/O interfaces. Files in HFS are implemented using simple building blocks composed in potentially complex ways. This approach yields great flexibility, allowing an application to customize the structure and policies of a file to exactly meet its requirements. As an extreme example, HFS allows a file's structure to be optimized for concurrent random-access write-only operations by ten threads, something no other file system can do. Similarly, the prefetching, locking, and file cache management policies can all be chosen to match an application's access pattern. In contrast, most parallel file systems support a single file structure and a small set of policies.

We have implemented 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. We also show that for a number of file access patterns HFS is able to deliver to the applications the full I/O bandwidth of the disks on our system.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*Access methods*; D.4.3 [**Operating Systems**]: File Systems Management—*File organization*; D.4.8 [**Operating Systems**]: Performance—*Measurements*; E.5 [**Files**]: Organization; E.5 [**Files**]: Optimization

General Terms: Design, Performance, Flexibility, Customization

Additional Key Words and Phrases: Data partitioning, data replication, parallel computing, parallel file system

---

This work was done while the authors were at the University of Toronto, and supported by a grant from the Canadian Natural Sciences and Engineering Research Council. Name: O. Krieger Affiliation: IBM T.J. Watson Research Center Address: P.O.Box 218, Yorktown Heights, NY 10598; email: okrieg@watson.ibm.com Name: M. Stumm Affiliation: Department of Electrical and Computer Engineering, University of Toronto Address: 10 King's College Road, Toronto, Canada M5S 3G4; email: stumm@eecg.toronto.edu

1. INTRODUCTION

The Hurricane File System (HFS) supports a large, expandable set of file structures, interfaces, and policies, and allows an application to customize the files it uses to match its I/O requirements. This high degree of flexibility is achieved by having many fine-grained building blocks, each of which defines a portion of a file's structure or implements a simple set of policies. The application can control the composition of the building blocks used to implement files on a per file and per open file instance basis. This allows the file's implementation to be customized to best meet the application's I/O requirements.

A parallel file system differs from uniprocessor file systems both because of the hardware resources it must manage and because of the requirements of the applications it must support. HFS is designed for (potentially large-scale) shared-memory multiprocessors. Such systems have a large number of disks and memory modules distributed across the system. To optimize I/O performance, a file system must properly exploit these resources. For example, data should be distributed across the disks to improve I/O bandwidth, cached in memory to reduce the demand on the system disks, and prefetched from disk in order to hide the latency of disk I/O from applications.

Many important supercomputer applications have massive I/O requirements [del Rosario and Choudhary 1994; Galbreath et al. 1993; Intel 1989; Lin and Zhou 1993; Miller and Katz 1991; Poole 1994; Scott 1993]. These parallel applications have needs that are different than those of the typical sequential Unix applications for which many file systems have been optimized. First, for many supercomputer applications, most of the I/O bandwidth goes to accessing temporary files, say for data sets that cannot fit in main memory.[1] Because the files are created specifically for the application, the file structure can be optimized for the particular access pattern of the application that will use it. Second, such applications are much less likely to access an entire file sequentially, although they may still have predictable access patterns that can be used to optimize I/O performance. Finally, the interface requirements of such applications differ from traditional Unix applications. Unix read/write requests both implicitly impose synchronization that unnecessarily constrains performance and results in poor performance when requests are not to sequential data.

Optimizing a file system for I/O-intensive parallel applications is complicated by the fact that the requirements of these applications are poorly understood, and hence there is currently little understanding of the file structures, policies, and I/O interfaces required in order to best satisfy these requirements. This poor understanding exists, in part, because most current parallel machines have poor support for high performance I/O, and as a result the parallel processing community has mainly studied applications that have small I/O requirements [Crandall et al. 1995; del Rosario and Choudhary 1994].

The flexibility of HFS allows a file to be customized to match the needs of a particular application on a particular hardware platform, and the application can

[1] Another reason why many supercomputer applications access temporary files is that the massive files used may require storage on tertiary rather than secondary storage, and the files are only transfered to disk when an application will actively be using them.

directly control this customization. An HFS application can choose how file data is to be distributed across the disks, how the data is to be distributed on each of the disks, whether to store redundant copies for fault tolerance, how to hide latencies, what advisory or enforced synchronization policies to use, and whether to transparently invoke compression/decompression algorithms on the data. We contend that this level of flexibility is necessary in order to be able to accommodate the diverse I/O requirements of applications executing on parallel hardware.

While we have focused on I/O intensive parallel scientific applications, HFS has been designed to be a general file system that can efficiently support the needs of a wide variety of applications. As as result, HFS also efficiently supports the interface standards used by sequential applications. This is important for the file system of a shared-memory multiprocessor, since these machines are expected to (concurrently) support applications that range from sequential interactive jobs to very large parallel applications, and from scientific applications to transactional data base systems [Frank et al. 1993; Kuskin et al. 1994; Lenoski et al. 1992].

Overall, the flexibility of HFS is important for a number of reasons. First, it is crucial in allowing a large class of I/O-intensive supercomputer applications to exploit the full I/O bandwidth of the disks attached to the system. Second, given the poorly understood requirements of I/O-intensive parallel applications, flexibility is necessary to ensure that the file system doesn't constrain application developers. Third, the flexibility allows HFS to easily adapt to new I/O interfaces, new operating systems, new demands on the file system functionality, and new hardware platforms. Fourth, the flexibility allows the requirements of other workloads, such as sequential Unix applications, to be met without imposing additional overheads. Finally, the flexibility is important for experimentation, making it easy to explore different policies and file structures in the context of a common system environment.

For parallel processing, flexibility and customizability is of little use if the associated overhead results in poor performance. We demonstrate that the flexibility of HFS comes with negligible overhead when compared to the other costs associated with I/O. Moreover, we argue that in many cases the ability to customize an implementation of a file to its expected demands can result in lower overhead than a more generic implementation.

In the next section we describe the building-block composition technique. The following section presents the architecture of HFS and shows how building-block compositions are implemented by our file system. Subsequent sections present performance results obtained from our implementation of HFS as a part of the Hurricane operating system [Unrau et al. 1995] on the Hector shared-memory multiprocessor [Vranesic et al. 1991]. Our experimental results show that (1) it is feasible to implement a file system based on building-block compositions, (2) the performance overhead is low, and (3) the basic goal of flexibility is indeed important.

2. BUILDING-BLOCK COMPOSITION

Each HFS file is implemented by combining together a set of *building blocks*. A building block can define a portion of a file's structure or implement a simple set of policies. For example, different types of building blocks exist to store file data on a single disk, distribute file data to other building blocks, replicate file data to other building blocks, store file data with redundancy for fault tolerance, pre-fetch

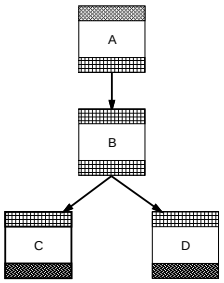

Fig. 1. This HFS building-block composition implements a file. Building blocks C and D may each store data on a single disk, building block B might be a distribution building block that distributes the file data to C and D, and building block A might be a compression/decompression building block that decompresses data read from B and compresses data being written to B.

file data into main memory, enforce security, manage locks, and interact with the memory manager to cache file data. Building blocks are implemented as objects and thus contain state and a set of operations that manipulate that state.

A building-block object exports an interface that specifies the operations that can be invoked by other building blocks. It may also import (one or more) interfaces that are exported by other building blocks. Two building blocks are said to be *connected* if one of them can invoke operations of the other, and the building block is then also said to *reference* the other. Two building blocks may be connected only if the exported interface of the one is imported by the other.

The particular composition of building blocks that implements a file (i.e., the set of building blocks and the way they are connected) determines the behavior and performance of the file. As a simple example, Figure 1 shows four building blocks and how they are connected. Building block B contains references to C and D, and in turn is referenced by A. Building blocks C and D may each store data on a different disk, B might be a distribution building block that distributes the file data to C and D, and A might be a compression/decompression building block that de-compresses data read from B and compresses data being written to B. The imported and exported interfaces are indicated by the pattern at the top and bottom of each building block. If two building blocks are connected then the corresponding imported and exported interfaces must match.

It is important to note that *each* file and *each* open file instance will (at least in part) have a different building-block composition. For example, two open file instances (even of the same file) will be implemented by a different set of building blocks, possibly with a different topology, making it possible to offer highly customized services.

2.1 Flexibility

In our building-block framework, flexibility can be achieved in a number of ways. First, given a particular composition, it is possible to exchange one building block for another as long as the imported and exported interfaces of the two are the same. For example, in Figure 1, building block B could be replaced by another building block B' that implements a different distribution. Thus for each type of building

block, multiple implementations may exist, each supporting a different policy or optimized for a different application behavior. In practice this is achieved by having multiple subclasses provide separate implementations with identical interfaces inherited from a common superclass. Even with only a few subclasses for each building block, the combinatorial effect on the behavior of an entire composition can be huge.

Second, new building blocks can be added to an existing structure if the connecting interfaces match, thus modifying the topology. This can be used to add new functionality. For example, a new building block E (that, say, implements prefetching of some sort) can be inserted between A and B, as long as both the imported and the exported interface of E are the same as that exported by B. Building blocks that import the same interface they export can be arbitrarily stacked. As another example, to implement a replicated file, one can imagine just adding a replication building block F between A and B that is connected to both B and a second subtree rooted by another distribution building block similar to B.

Finally, it is possible to support new interfaces to applications by introducing new building blocks that export these interfaces, but import existing interfaces so that they can be connected to existing structures.

In general, the finer the granularity of the building blocks used in a composition (and thus the larger the number of building blocks in the composition) the larger the degree of flexibility. In our implementation, we have found that we tend to use many fine-grained building blocks in a composition, as opposed to using a few large ones. For example, the distribution building blocks B above might execute only 2–3 lines of C code in a typical flow of control through the building block. Similarly, the larger the number of building-block types with identical interfaces is, the more flexibility exists in defining compositions. This is particularly true for building blocks that export the same interface they import.

2.2 Operation

In our model, control is passed from one building block to another by having the first invoke an exported operation of the other. The flow through building-block compositions is initiated either by an application invoking an operation on one of the building blocks, or as a result of a page fault or disk interrupt. In the case of a building block that resides in a different address space[2], an RPC-like facility is used to pass control. In our implementation, the building block code is executed by a thread that is created in the target address space as a by-product of the Hurricane protected procedure call (PPC) facility [Gamsa et al. 1994]. The degree of concurrency in servicing I/O requests is thus equal to the number of requests issued.

A crucial aspect of our work is that we allow applications to specify the initial composition of the files they create, and we allow applications to modify (in part) the existing building-block compositions of the files they are accessing. Arguments to a constructor specify how that building block is to be connected to the other already existing building blocks. Because the applications that specify compositions

[2]As will be seen in the next section, the file system is partitioned across a number of address spaces.

may be untrusted, it is necessary to validate the safety of these compositions. Hence, once the building blocks have been instantiated, they verify that each referenced building block is of the correct type (i.e., the right interface) and that any other required constraints are met. For example, if some building block requires that a particular file block size be supported, it verifies that all building blocks it references can in fact support that block size. In system servers, only hierarchical compositions are allowed (i.e., no loops), and only unidirectional links between building blocks are allowed. This simplifies validating the safety of the compositions.

Some building blocks, such as those that define file structure, are persistent and exist (on disk) as long as the file exists. In our current implementation these building blocks cannot be changed once created. Other building blocks, such as those that represent state for open file instances, are transitory and exist only when a file is open. These building blocks can be changed at any time, say to conform to a specific application's access pattern.

2.3 Performance considerations

Since performance is the ultimate goal of our work, the overhead of our building-block framework is an important concern. If a file is implemented using many layers of simple, fine-grain building blocks, then one might suspect that the overhead of traversing these layers would be a problem. In practice, we have found the overhead to be small for the following five reasons.

(1) Even if a file is implemented with many layers of building blocks, requests are often serviced by traversing only a small number of layers. For example, many building blocks cache data, so read and write requests can often be satisfied from a cache managed by a building block that is close to the client.

(2) Our approach naturally reduces the amount of cross-address space communication. Building-block composition differs from other approaches for customizing system software by not requiring the system to redirect requests to the application address space or to a separate server [Druschel 1993; Heidemann and Popek 1994; Khalidi and Nelson 1993]. With our approach, the customization occurs in the server providing the service, thus naturally avoiding cross-address space communication.

(3) We have found it possible to define building-block interfaces that have extremely low overhead. For example, the interfaces we have developed for our building blocks typically allow control to pass through multiple building blocks without requiring data to be copied.

(4) Since the functions and data of a building block are specific to the policies it implements, it is often possible to avoid executing conditional statements (e.g., to determine the policy to use) that a more traditional approach would require. Avoiding such conditional statements can reduce system overhead [Massalin and Pu 1989]. Also, since the data is optimized to provide exactly the information required, it can be represented more compactly, reducing the main memory and caching overhead of the system.

(5) If the use of many building blocks happens to result in excessive overhead for some important new workload, then it is straightforward for a systems programmer to define and add a new type of building block to specifically

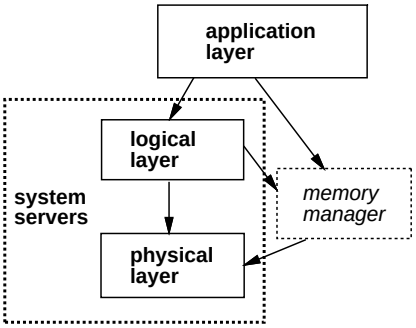

Fig. 2. This figure illustrates the relationship between the different components of the Hurricane file system. The Alloc Stream Facility (ASF) is an application-level I/O library. The physical layer of the HFS file system implements files and controls the system disks. The logical layer server provides file system authentication services. Most file I/O occurs through mapped files, so the memory manager is involved in most requests to read or write file data.

handle the demands of this workload, possibly by combining a set of simple building blocks into a single more complex building block.

3. ARCHITECTURE AND IMPLEMENTATION

The Hurricane File System (HFS) implements files using building-block compositions. HFS is logically divided into three layers: the application layer, the physical layer, and a logical layer between the two (Figure 2). The Alloc Stream Facility (ASF) is an I/O library that makes up the application layer of HFS [Krieger et al. 1994]. We provide as much functionality as possible in this layer in order to minimize communication with servers in other address spaces. The HFS physical layer, which implements files and controls the system disks, and the HFS logical layer, which provides file system authentication services, are system servers of the Hurricane operating system. The memory manager is involved in most requests to read or write file data, because ASF ensures that most file I/O occurs through mapped files. We describe each of the three layers of HFS in turn.

3.1 The Alloc Stream Facility

Application requests for I/O are often handled by an application-level I/O library, such as stdio, that acts as an intermediary between the application and the I/O servers of the system. The Alloc Stream Facility (ASF) is another such library, which we developed using our building-block approach. It handles all I/O on Hurricane (e.g., terminal, socket, disk, and pipe I/O), is suitable for multiprocessor operation, and exports a number of different I/O interfaces (such as the Unix read/write and stdio interfaces) to applications [Krieger et al. 1994]. AFS has been ported to a number of uniprocessor and multiprocessor systems, including SunOS, IRIX, AIX, and HP-UX.

The ASF I/O package is an example where we have clearly found it advantageous to define a large number of simple building blocks, instead of a smaller number of more general ones. Most of the building blocks import and export the same interface, so there is much flexibility in the compositions. This common interface,

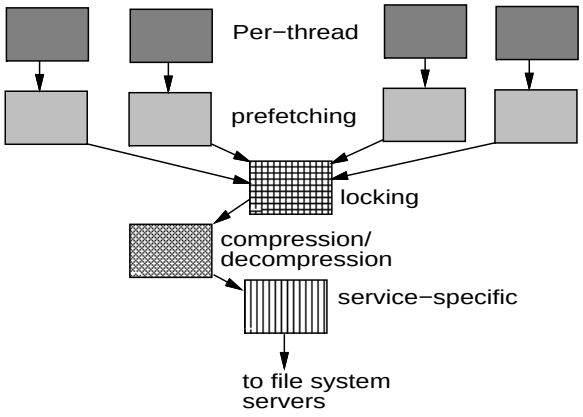

Fig. 3. Example of an ASF building-block composition used by a multi-threaded application to access a file stored on disk in a compressed format. All building blocks other than the service-specific one import and export the ASI interface. The application is assumed to use ASI directly, so no interface building blocks are shown.

the *Alloc Stream Interface* (ASI) was carefully designed to minimize the amount of data copying required as control is passed from one building block to another.[3]

The implementation of ASF supports three types of building-blocks: (1) ASI building blocks, (2) service-specific building blocks, and (3) interface building blocks. ASI building blocks import and export the ASI interface, and can hence be connected in arbitrary ways. We have defined ASI building blocks that implement policies for latency hiding, compression/decompression and advisory locking (for multi-threaded applications). To simplify the task of the application programmer, we have defined ASI building blocks that allow applications to have customized views of file data.[4]

Service-specific building blocks import the interface of a particular I/O service (e.g., terminal, file, network connection) and export the ASI interface. For good performance, we have found it important that data be transferred to and from server address spaces using service-specific interfaces. For example, on Hurricane large files are most efficiently accessed using mapped-file I/O, while terminal I/O is best accessed using a read/write interface. We also define a separate building block for read-only, write-only, and read/write access for each I/O service, respectively. Separate, direction-specific building blocks are more efficient than more general

[3]While having a common interface is important, it is also important that building blocks be able to export additional building block-specific functions for extensibility. For example, if a file system supports prefetching, the service-specific building blocks for that file system should export an interface that allows for prefetching. ASI supports this by providing a `special` function (similar to the `default` function Heidemann and Popek [Heidemann and Popek 1994] define for their stackable file system) that handles any class-specific requests. When a building block cannot directly interpret a special request, it re-directs it to the building blocks it references.

[4]This is similar to the functionality provided by the ELFS file system [Grimshaw and Loyot 1991] that allow the semantic structure of a file (e.g., a file containing a two dimensional matrix) to be taken into account. Similarly, we have defined other ASI building blocks that implement the mapping functions defined by the Vesta file system [Corbett and Feitelson 1996].

ones, since less checking is necessary when they are specialized.

Interface building blocks import the ASI interface but export a standard I/O interface to the application, such as the (emulated) standard Unix I/O system call interface and the stdio interface. (Applications can also use the ASI interface directly.) Language interfaces, such as the C++ iostream, could also be supported, although we have not yet implemented them.

Figure 3 illustrates how ASF building blocks might be used in a building-block composition for a file being accessed by a multi-threaded application where each thread sequentially accesses a different part of a single file stored on disk in a compressed format. Each application thread has a *per thread* building block that maintains the file offset for that thread. Since each thread accesses data sequentially, but the file as a whole is not accessed sequentially, each per-thread building block references a different prefetching building block. The prefetching building blocks all reference a single building block that implements some advisory locking policy.[5] This building block, in turn, references a compression/decompression building block, which decompresses data read from the file and compresses data being written to the file. Finally, the compression/decompression building block makes requests to access file blocks to a service-specific building block, which maps the file into the application address space, and translates I/O requests for file data into accesses to the mapped region.

An application that creates a file can specify which ASF building blocks should be instantiated by default when the file is opened.[6] Once a file has been opened, the accessing application can customize the file by modifying the set of building blocks used. For example, the number of threads and the way threads access the file are specific to the application, so the per-thread prefetching and locking building blocks shown in Figure 3 are instantiated explicitly by the application at run time. These ASF building blocks can be subsequently changed to adapt to the different access patterns an application might have in different phases of its execution. For example, an application traversing a matrix first by row and then by column would benefit from changing the ASF building blocks used when its access pattern to the data changes.

3.2 Physical-layer file service

The physical layer of HFS implements files and controls the disks on the system. It handles disk block placement and is thus responsible for cylinder clustering, load balancing, and locality management. Logically, the physical layer of HFS is situated below the memory manager, as shown in Figure 2, so that mapped file I/O

[5] Any locking policy implemented in the application address space can only be advisory (rather than enforced), since there is no way to ensure that the application cannot bypass the library and call the file system servers directly. For scalability, building blocks through which all control must pass, such as the locking building block shown, must be implemented using sophisticated techniques in order to properly handle concurrency, caching and NUMA effects [Parsons et al. 1995].

[6] Having application-layer building blocks instantiated automatically is important in order to provide functionality transparent to the application. For example, if the compression/decompression building block of Figure 3 is instantiated automatically, then ASF can hide the fact that compression is being done from applications. This allows standard utilities, such as editors, to process compressed files without knowledge of this fact.

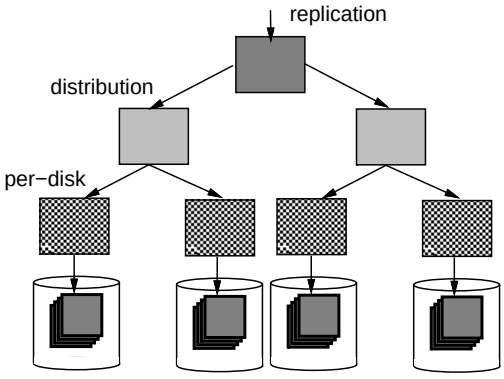

Fig. 4. Example building-block composition in the physical layer of the HFS file system.

can exploit the facilities of the physical layer. 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.

All building blocks used in the physical layer of HFS import and export the same interface, except for those that interact directly with the disks. Again, this allows the building blocks to be composed in a wide variety of ways, and hence allows for much flexibility in the policies implemented by this layer of the file system. The interface used here is similar to the ASI interface used in the application-level library, where the key operations to read and write file blocks involve no copying as control passes through the building blocks.

We have implemented three types of *per-disk* building blocks that organize file data on the disk: *extent based* building blocks store file data using contiguous extents of up to 256 disk blocks [Sweeney et al. 1996], *random access* building blocks always write file blocks to a (new) location near the current disk head position [Rosenblum and Ousterhout 1991], and *sparse data* building blocks are optimized for files with large un-populated areas.

Physical-layer building blocks that import and export the same interface include:

Striped data. stripes data in a round-robin fashion across the referenced building blocks,

Distribution. partitions a file into contiguous regions, each stored by a different referenced building block,

Write-mostly. takes into account disk proximity or load to optimize write performance,

Replication. replicates data to each referenced building block,

Parity. computes and stores parity information for fault recovery [Patterson et al. 1988],

Application specific distribution. maintains an application specified table that defines how data is distributed to referenced building blocks, and

Small data. stores the data of a small file together with the file meta-data.[7]

[7]Multiple small data building blocks can be packed into a single disk block to minimize disk

These building blocks are for the most part simple and are described (together with their interfaces) in more detail in [Krieger 1994].

Figure 4 illustrates how the simple building blocks of the physical layer can be combined together to create very complex file structures. In this example, a replication building block replicates data to two distribution building blocks, each of which distribute data across a number of per-disk building blocks. Such a file structure would allow file data to be replicated to different parts of a large scale multiprocessor, allowing requests for file data to be serviced from a nearby replica.

The structure of a file is determined by how the initial building blocks are instantiated and connected, and this is controlled directly by the application. The structure is defined in a bottom up fashion. As the file grows, the file system may instantiate new building blocks to store the additional data, but does so while retaining the basic file structure defined by the application.

Unlike application-layer building blocks, the physical-layer building blocks associated with a file are persistent and cannot be changed at run time.[8] Therefore, some building blocks at this layer must implement more than a single policy to be able to handle different application requirements. For example, the *replication* building block might implement two policies for directing read requests: (1) to the object on the least loaded disk, or (2) to the object on the disk closest to the target memory. Which policy to use can then be specified (and changed) at run time.

Persistent building blocks, such as those described in this section, are stored on disk with the file data, and only cached in memory when they are needed. One novel feature of our system is that the persistent building blocks themselves are stored in a regular file that is itself stored on disk (by other persistent building blocks).[9] When an application instantiates a persistent building block, it can specify which file should be used to store that building block. This means that all the same policies used for storing file data can be used to store the building blocks themselves. This flexibility makes it possible, for example, to co-locate building blocks near other related data, to take advantage of the disk head position when writing out building blocks, or to store building blocks redundantly for fault tolerance.

3.3 Logical-layer file service

The logical layer of HFS implements all file system functionality that does not have to be implemented at the physical layer and, for security or performance reasons, should not be implemented in the application layer. In our implementation, this layer provides naming (i.e., directories), authentication, and locking services. 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 been locked by others.

The logical layer of HFS differs from the other layers in that there is less flexibility in how building blocks can be composed. Building blocks in this layer fit into four basic types. Naming building blocks maintain the name space of HFS and the

fragmentation.

[8] We are currently exploring techniques to ease this restriction so that file structures can be modified dynamically without requiring a copy.

[9] On each disk, there are a small number of persistent building blocks whose location on disk is recorded in a well known location.

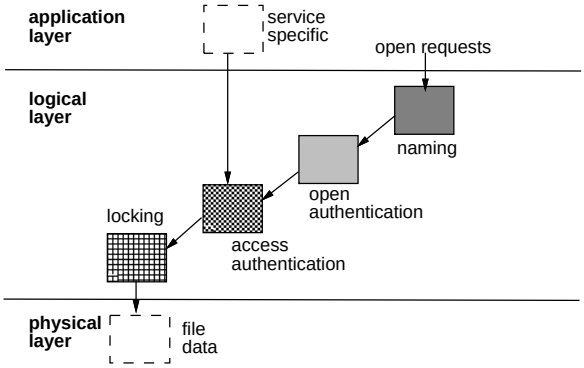

Fig. 5. An example of a logical-layer building-block composition.

particular type of building block used for a directory determines how the directory is to be cached in main memory. Open-authentication building blocks authenticate requests to open a file. Access-authentication building blocks authenticate requests to access (i.e., read, write or map) file data. Locking building blocks enforce locking policies.

Figure 5 illustrates a typical composition of logical-layer building blocks and shows how these building blocks interact with the building blocks in the application and physical layer. Open requests are resolved by naming building blocks that use the pathname specified by the application to locate the open-authentication building block for the file. After authenticating the request, for example by using an access list based policy, these building blocks return to ASF information which identifies both the application-layer building blocks to be instantiated (by default) and a handle of the access-authentication building block to which subsequent requests are to be directed. In the application layer, service-specific ASF building blocks direct requests to read, write, or map the file data to the access-authentication building block returned by an open request. After authenticating the request, the access-authentication building block directs the request to the locking building block which enforces locking constraints. In the case of mapped file I/O, the request is then forwarded to the memory manager. Otherwise, read and write requests are satisfied by corresponding requests made to the physical layer.

As with physical layer building blocks, the logical layer building blocks are persistent and instantiated by the application when the file (or directory) is created.

We have put in less work developing the logical layer of HFS than we have developing the other layers, primarily because the logical layer has very little effect on performance in our system. Except for small files, we have found that mapped file I/O results in the best performance, and hence most accesses to file data on our system do not go through the logical layer. Nevertheless, we believe that the flexibility of the building block approach could be usefully exploited even for this layer, making it easy to experiment with new mechanisms for caching directory information, authenticating open requests (e.g., using access lists or Unix style permissions) and enforcing locking protocols.

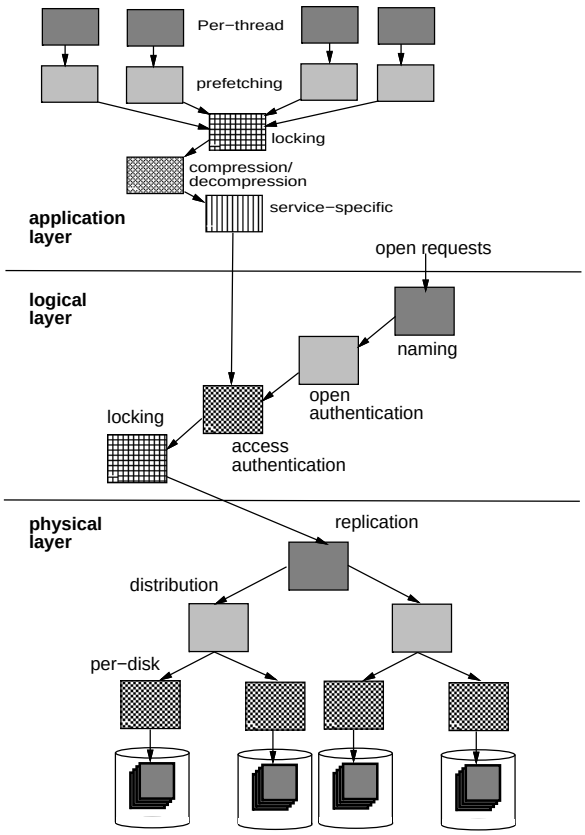

Fig. 6. A full example of an HFS file.

3.4 Putting it all together

Figure 6 shows how the different layers of HFS discussed in the previous sections work together when an application is accessing a file. The building blocks implement a file where (1) there are two replicas of the file data, (2) each of the replicas is distributed across a different set of disks, (3) the data is stored in the file in a compressed format, (4) the file is being accessed by a multi-threaded application, and (5) each thread is sequentially accessing a different part of the file data.

The figure demonstrates the strength of the building-block approach employed by HFS. The application is able to customize a file and open-file instance to match its specific access pattern by customizing both the file structure and the file system policies implemented by all three layers of the file system. Not only does each of the layers provide its functionality in a flexible fashion, but a combinatorial effect arises because compositions span all layers of the system. While file systems such as Vesta [Corbett and Feitelson 1996] support quite complex file structures, and systems such as ELFS [Grimshaw and Loyot 1991] allow for customized policies at the application level, HFS is unique in having a single consistent technique for customized functionality at all levels of the system.

It is clear from the above discussion that the development of HFS differed from most other file systems in that we took a holistic view of the file system, considering all the system servers and layers that affect I/O performance together. Looking at the system in this fashion resulted in numerous software engineering and performance advantages. For example, the physical layer of the file system could rely on ASF to guarantee that all accesses to large files use mapped file I/O, making it unnecessary to support multiple techniques for accessing the same file with the attendant required effort to provide consistency. As another example, ASF can rely on the logical layer to tell it what building blocks should be instantiated by default, allowing the compression implemented by ASF to be transparent to the application accessing a file.

4. MINIMIZING APPLICATION COMPLEXITY

There is a tradeoff between flexibility and the complexity an application programmer is faced with when developing applications to use the system. While we have found writing application-level code to create HFS files to be straightforward, we do not, in general, expect application programmers to write such code themselves. Instead, we believe that a number of approaches will allow even novice users to exploit the flexibility of our file system.

ASF defaults. The application-level library can create files using reasonable defaults if no information is available from the application.

Library functions. We have constructed a variety of library routines that implement the most common types of data distribution, including all examples described in the previous section. These routines hide the process of instantiating building blocks from the application programmer.

Compiler support. A compiler has been developed that for a class of applications hides all I/O operations from the application programmer, while exploiting the flexibility of HFS to achieve good performance [Mowry et al. 1996].

5. EVALUATION

In this section we describe our experiences with, and show performance results from, our implementation of HFS. Our goal is to demonstrate that (1) it is feasible to implement a file system based on building-block compositions, (2) the performance overhead is low, and (3) the basic goal of flexibility is indeed important. While it is difficult to quantify flexibility, we will argue in Section 7 that HFS is significantly more flexible than other existing and proposed multiprocessor file systems.

We have implemented HFS as part of the Hurricane operating system [Unrau et al. 1995] on the Hector shared-memory multiprocessor [Vranesic et al. 1991]. The ASF application-level library has also been ported to a number of other systems. The first section describes our qualitative experiences in implementing HFS and summarizes results on ASF that have previously been published [Krieger et al. 1994]. The remaining sections show performance results on Hector/Hurricane. Sections 5.2 and 5.3 provide measurements of the experimental environment and of some basic file system operations. Section 5.4 presents the performance of the file system for two file access patterns, and demonstrates that the flexibility of HFS is important in obtaining good performance for these access patterns. Finally, using

an extreme stress test, Section 5.5 shows that the software overhead of our approach is negligible.

5.1 Experiences

As a structuring technique for developing flexible system software, we found that building-block composition has a software engineering advantage, resulting in greatly reduced code complexity. We had developed an earlier file system prior to HFS that, like most other file systems, had code specific to each type of file (e.g., replicated file, distributed file, etc.) and specific to the meta-data of each type of file. With this earlier system, the programming effort to add a new type of file was high. In contrast, we found that HFS, using building-block compositions, was less complicated to implement, even though better performance and a great deal more flexibility was achieved.

One measure of flexibility is how easy it is to adapt to new environments. We ported the ASF application-level library[10] to a number of systems, including SunOS, IRIX, AIX, HP-UX, and Tornado [Auslander et al. 1997], and found the port to be relatively easy; it was only necessary to change some of the service-specific building blocks to adapt to a new platform.

The flexibility of ASF allowed us to easily modify the library to exploit the best features of each host operating system while bypassing any poorly performing ones. As reported in previous work [Krieger et al. 1994], this resulted in substantial performance advantages. For instance, a Unix `diff` application comparing two identical files runs nearly four times faster on an AIX system when using ASF instead of the AIX native stdio library. Performance improvements like these, although representing extreme cases, indicate the advantages of developing a flexible system that can be easily modified to adapt to a particular platform. In the particular example cited, we were able to exploit a mapped file I/O facility that is specific to the AIX system.

Another measure of flexibility of the system is how easy it is to handle requirements not foreseen by its developers. We initially had no intention of making crash recovery fast, but recovering from a system crash involved re-formatting the disks with our initial implementation, making the system essentially unusable in our experimental OS/hardware environment. Adding efficient crash recovery to HFS required only minor modifications to the building blocks that store persistent building blocks on disk (Section 3.2) and to the definition of the per-partition superblock. As a result, it now only takes seconds to recover from a typical system crash.

Experimental research in system software always entails compromises. In developing HFS, the choice of the Hurricane/Hector platform involved both advantages and disadvantages. The key advantages were that the operating system and hardware could be modified for our experimentation, and that the platform was available for system development. The disadvantages were (1) much of the basic operating system and hardware infrastructure required for this research had to be developed from scratch; and (2) the small size of the system limited the experiments we could

[10]Since HFS requires low-level control over the disks, it would have required great effort to port the entire file system to a different platform.

perform. For this reason, we have confined our implementation goals to producing a proof of concept prototype. We have implemented the entire file system infrastructure (e.g., device drivers, code for marshalling and de-marshalling requests to the file system, code for caching persistent building blocks in main memory, etc.) and have implemented many but not all of the building blocks described in the previous sections.[11] Nevertheless, our prototype is sufficiently complete to show:

(1) The full HFS architecture is implementable. We are confident that all building blocks of the physical and application layers described in the previous sections are implementable without requiring major changes to either the common interfaces or the non-building-block-specific code. While we have less experience with the logical-layer of HFS, we are also less concerned about this layer, since it is not heavily involved for reading and writing file data and hence is not performance-critical.

(2) The overhead of building-block composition is low and, in fact, negligible. Even with a composition that uses many layers of building blocks, the processing overhead of HFS is low compared to other processing overhead involved in I/O. We will show that HFS is capable of delivering the full I/O bandwidth of the disks on our system.

(3) The flexibility available in HFS (but not other systems) is useful. In particular, we use two synthetic stress tests with very different access patterns to show that the flexibility of HFS is needed to be able to deliver 100% of the disk bandwidth to the application with these access patterns.

5.2 Infrastructure

We first describe our hardware infrastructure and the experimental setup, and then describe the key performance characteristics of the operating system infrastructure.

5.2.1 *Hardware infrastructure.* Our experiments were performed on the Hector multiprocessor [Vranesic et al. 1991], which is constructed from processing modules (PMs) each with a 16.67 MHz Motorola 88100 processor, 16KB data and 16KB instruction caches (88200), and a local portion of the globally-addressable memory. A small number of PMs are connected by a bus to form a station, and several stations are connected by a high-speed ring, with multiple rings connected together by higher-level rings, and so on (Figure 7). The particular hardware configuration used in our experiments consisted of 4 PMs per station and 4 stations connected by a ring. Hector does not support hardware cache coherence, so many of the internal data structures used in HFS are not cached. Seven Conner CP3200 disks are connected to the Hector prototype, each directly connected to a different PM. Requests to a disk must be initiated by its local processor and all disk interrupts are serviced by that processor. Disks can DMA data to or from any memory module

[11] Our current implementation includes about 40K lines of C and C++ source code. This includes about 2K lines of code used to cache path names (and shared by a number of system servers), 5K lines of code for handling I/O requests (and shared by our NFS and disk file system), 6K lines of infrastructure code specific to the disk file system, 7K lines of device driver code, 5K lines of physical-layer building-block code, 1K lines of logical-layer building-block code, and 8K lines of application-level mostly building-block code.

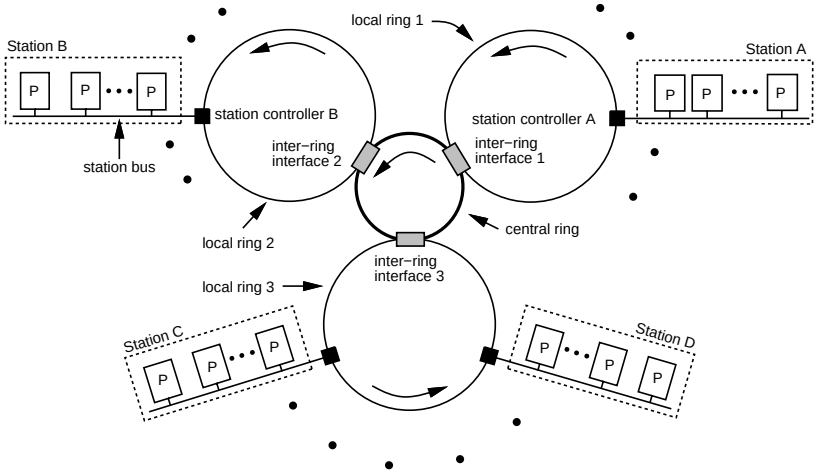

Fig. 7. A high-level representation of the Hector multiprocessor. Each processor box in the system includes a CPU and cache, local memory, and I/O interface.

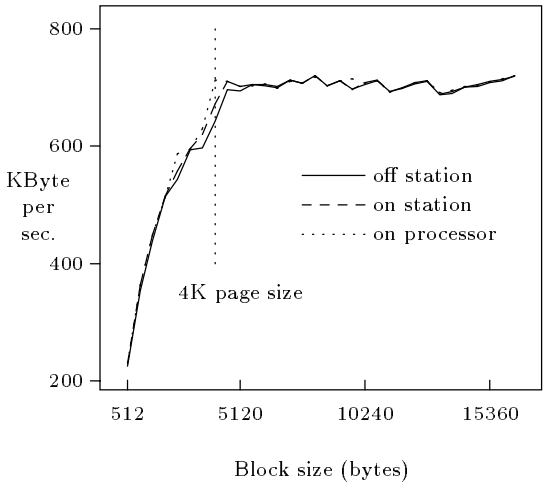

Fig. 8. Basic per-disk I/O throughput on Hector as a function of block size. Solid lines show performance for requests that are directed to off-station memory. Dashed lines show performance for on-station requests, dotted lines show performance for on-processor module requests. The vertical dotted line indicates the performance for 4 KByte disk block requests, corresponding to the Hurricane page size.

in the system.[12]

Figure 8 shows maximum disk throughput on our system in KBytes/sec. for dif-

[12]During a load or store by the disk to memory (local or remote), the processor co-located with the disk is blocked from making any memory accesses, which can have a substantial impact on performance.

ferent block sizes.[13] To avoid interrupt overhead, we obtained these measurements by instrumenting the device driver to poll waiting for I/O operations to complete. The maximum performance obtained from the disk is about 720 KBytes/sec. This maximum performance was obtained by making requests larger than 5 KBytes and is independent of the memory to which the request is directed. Unfortunately, the performance is slightly worse with 4 KByte disk blocks (the memory manager page size on Hurricane) and depends on the target memory, varying from 640 KBytes/sec. to between 670 and 712 KBytes/sec.

5.2.2 *Operating system infrastructure.* HFS was developed as part of the Hurricane operating system [Unrau et al. 1994; Unrau et al. 1995]. In Hurricane, a micro-kernel provides for basic process and thread management, interprocess communication and memory management, while most of the operating system services are provided by user-level servers.

The interprocess communication facility of Hurricane, *Protected Procedure Calls* (PPC), has three important characteristics:

(1) It is fast:[14] a null PPC request between two application address spaces varies between 35 and 70 μsec. on our hardware, depending on how much of the state required for the PPC is in the processor cache.

(2) In the common case, no locks need to be acquired and only local data is accessed, making the IPC system fully concurrent.

(3) New threads are created dynamically (as required) to handle requests in the server address space [Gamsa et al. 1994]. This allows the concurrency in the file system to match the demands being made on it by application threads.

The speed and concurrency of the PPC facility minimizes the impact of splitting the file system into multiple servers and ensures that IPC is not a bottleneck on file system requests.

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. However, a limitation of the memory manager we have not addressed yet is that it cannot allocate contiguous physical page frames so that contiguous disk blocks cannot be read from disk using a single disk request. Hence all requests by the memory manager to the file system are for individual 4 KByte pages. On the other hand, the memory manager supports prefetch and poststore operations that allow ASF to request pages to be asynchronously fetched from or stored to disk; a single PPC operation can request the prefetch or poststore of multiple of file blocks.[15]

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

[13] The numbers we present in this section were obtained either using a microsecond timer with 10-cycle access overhead or by measuring the total 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.

[14] The performance of our multiprocessor PPC calls (including the creation of a thread in the target address space) is competitive with the fastest known uniprocessor IPC implementations [Liedtke 1993].

[15] In our implementation, the memory manager can request from the file system at most five file blocks with a single PPC operation.

in the file cache, then the memory-manager overhead to initiate a request to the file system is about 800 μsec.[16] 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.

5.3 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 on a single disk implemented with an extent-based building block. We then measured the performance of a single thread reading data into memory on the same PM, the same station, and on a different station as the disk, respectively. The data rate delivered to the application varies from 515 to 540 KBytes/sec. There are two reasons for this relatively poor performance. First, whenever a request is sent to the disk, the disk is idle, so the file system incurs the overhead of waking up a thread local to the target disk to initiate the I/O request.[17] 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 time the next request is sent to the disk.

We ran the same experiment but with the reading thread 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 from 640 to 712 KBytes/sec., depending on the target memory. This corresponds to 100% of the available disk bandwidth, given the Hurricane page size.

To demonstrate the basic software overhead and concurrency of the file system and the PPC facility, we ran a simple experiment where n threads 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, varying between 67 μsec. and 73 μsec. depending on the locality of the requesting thread and the data being accessed by the file system and kernel. About 35 μsec. of this cost is attributable to the PPC request, 10 μsec. to HFS for locating the building blocks in its main memory cache of persistent building blocks, and 10 μsec. to the building block. The rest is file system overhead to authenticate the request and to marshal and de-marshal the arguments of the request.

Figure 9 shows the throughput when multiple threads run the same experiment. If all requests are to the same file, then the file system saturates at about four threads. Speedup in this case is limited primarily by contention on the locks in the file's building blocks and on the structure used to locate the building blocks in HFS's cache of persistent building blocks.[18] If each thread requests the length of a different file, then the speedup is 15.59. In this case there is no lock contention

[16] The memory manager cost of a page fault (that results in I/O) is larger than it should be, but we have not yet optimized this part of the system. However, because in our experiments most I/O is due to prefetch requests, this cost has little impact on the results shown.

[17] New disk requests are placed on a shared queue, and when a disk request completes, the file system code that handles the interrupt de-queues any pending request from this shared queue. Hence, as long as the disk does not become idle, it is not necessary to wake up the per-disk thread.

[18] A hash table is used by HFS to locate cached building blocks, where each hash chain is locked independently.

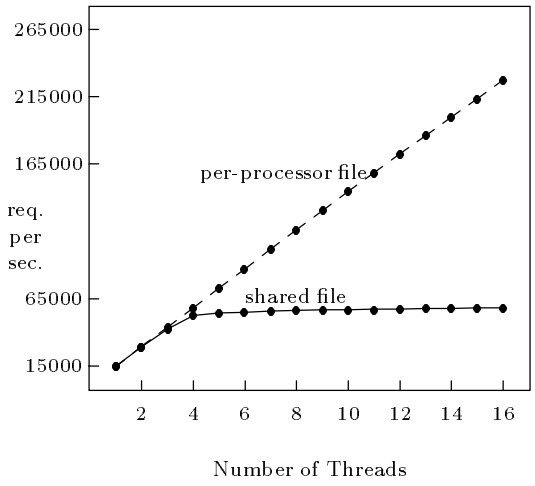

Fig. 9. Throughput of concurrent requests to obtain the length of a file. The solid line shows the performance when all requests are directed to a single file. The dashed line shows the performance when each requester is directing requests to a different file.

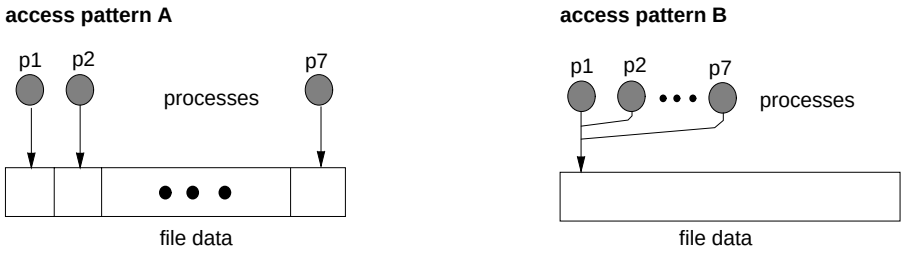

Fig. 10. Two file access patterns. With access pattern A, up to seven threads of a parallel application concurrently read a different region sequentially. Each thread has its own file pointer. With access pattern B, the threads of a parallel application cooperate to read the file sequentially. The threads share a common file pointer.

because different building blocks and hash chains are being locked. Linear speedup is not entirely achieved because of memory contention, i.e., concurrent accesses to different data structures in the same memory modules.

We can deduce from this experiment that is is unlikely that locks in the file system will be contended during regular file accesses. The time per-building block locks are held in this example is typical of all our building blocks. Even in the case where all threads are making requests to the same building block (i.e., the case that saturates at four threads), the file system can handle over 50000 requests per second before lock contention becomes a problem. This rate of requests is an order of magnitude higher than the maximum rate of requests our disks can sustain. The only locks not exercised in this experiment are those associated with the per-disk request queues, and these locks are obviously less of a bottleneck than the disks they protect.

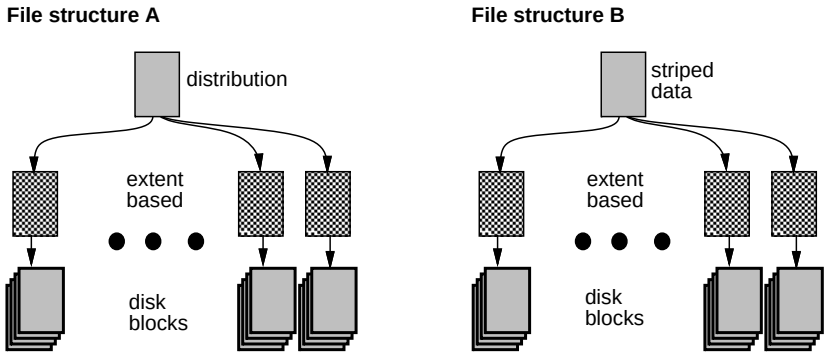

Fig. 11. Files distributed across multiple disks. File structure A implements a file using a distribution building block that partitions the file into seven contiguous regions, each stored by a different extent-based building block. File structure B implements a file using a striped-data building block that stripes file blocks round robin across seven extent-based building blocks.

5.4 Parallel I/O

In this section we consider the two file access patterns depicted in Figure 10 and show the importance of matching application requirements. With access pattern A, seven threads of a parallel application concurrently read a file; each thread sequentially reads from a different region of the file. For many parallel applications this is a natural way to partition file access [Crockett 1989]. With access pattern B, seven threads cooperate to read a file sequentially; each thread reads a small amount of data at a time. Such an access pattern is natural for algorithms where threads proceed at their own rate obtaining the next available unit of work for processing [Crockett 1989].

5.4.1 *File structure.* Clearly, the best way to support access pattern A is to have the file partitioned into 7 regions, each stored on a different disk. This balances the load on the disks, ensures that the requests of one thread don't interfere with those of other threads, and allows the sequence of requests of a thread to be mapped to consecutive disk blocks (minimizing seek operations). We call this file structure A.

The physical-layer HFS building blocks used to implement file structure A are shown in Figure 11. A distribution building block partitions the file into seven contiguous regions and directs requests for each region to a different extent-based building block. The extent-based building blocks store file blocks on disk using contiguous extents of up to 256 disk blocks, hence minimizing seek operations for sequential access patterns.[19] The size of the regions are specified when the file is created.

The best way to support access pattern B is to stripe the file across all disks in

[19]Each of the extent-based building blocks is stored on the same disk as the disk blocks it controls to ensure that file system requests for meta-data do not interfere with the file data requests by application threads. The only common building block for the seven regions is the distribution block, and since the operations performed by this building block are simple and hold a lock for just a short period of time, it is not a source of contention.

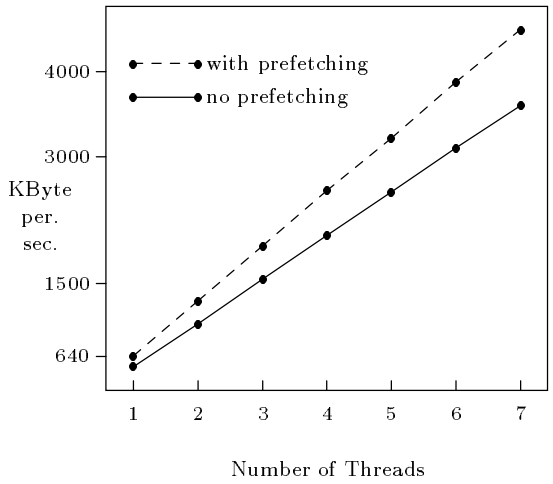

Fig. 12. Throughput for access pattern A run on file structure A. The solid line shows the performance with no prefetching (as discussed in Section 5.4.1). The dashed line shows the performance with prefetching of a single block (as discussed in Section 5.4.3).

the system to balance the requests across the disks so as to give each disk as much time as possible to pre-fetch into its on disk cache before it must process the next request. The HFS building blocks used to implement file structure B are shown in Figure 11. The striped-data building block stripes file blocks round robin across seven extent-based building blocks.[20]

The similarity of file structures A and B illustrates the expressive power of building-block composition. The two compositions differ only in a single building block, yet implement completely different file structures that, as we will show, result in very different performance characteristics for the two access patterns.

5.4.2 *File access with no prefetching.* We measured the performance of the file system with access patterns A and B on both file structures A and B, and the results are summarized in Table 1. In all cases the data is read into application pages distributed round-robin across the memory modules of the system. The seven threads are run on seven consecutive processors, and, to compensate for NUMA effects, the experiment was run repeatedly until every sequence of seven processors had been used. All disk requests are initiated by page faults, because ASF building blocks are used that map the file into the application address space. For access pattern A, each thread uses an independent ASF building block to maintain a separate file offset. For access pattern B, a single shared ASF building block is used.

When access pattern A is run on file structure B, the application receives about one quarter the bandwidth compared to when file structure A is used. The reason

[20] The striped-data building block adjusts the file offset of requests made to the referenced building blocks so that data can be stored on disk in a dense fashion. That is, a request made to an extent-based building block for block n is for the n^{th} block stored by the extent-based building block and not for the n^{th} block of the file as a whole.

Table 1. Bandwidth achievable for file access patterns A and B

		access pattern A	access pattern B
without prefetching	file structure A	3597	520
	file structure B	1000	640
with prefetching	file structure A	4526	646
	file structure B	1212	4526

for the poor performance is that requests by the different threads arrive at disks in an interleaved fashion, resulting in many seek operations. In fact, our results for this case are optimistic, in that the extents used by the different regions are likely to be close to each other on disk given that the disk was otherwise empty.

For access pattern B, the performance difference between the two file structures is not as large (although we will show that the difference becomes significant if prefetching is used). When access pattern B is run on file structure B, the file system cannot keep multiple disks busy because only one request is outstanding at a time. Nevertheless, performance is somewhat better than if the file is stored on a single disk, since each disk has time to prefetch data into its on-disk cache. When access pattern B is run on file structure A, then the file system obtains the same performance as if the file were stored on a single disk.

For file access pattern A on file structure A, we also varied the number of requesting threads between one and seven and measured the speedup in terms of KBytes/sec. to determine the amount of concurrency available in the file structure. As shown in Figure 12, the speedup with seven threads is 6.89. This does not correspond to perfect speedup, but it turns out that none of the file system locks are heavily contended, and hence we do not believe that the degradation is due to the file system. We believe that the degradation is due to (1) memory contention as the disks transfer data to memory, (2) pre-emption of the file system and application threads due to disk interrupts, and (3) contention for kernel data structures such as the ready queues.

5.4.3 *File access with prefetching.* If the full bandwidth of the disks on our system is to be delivered to the applications, then prefetching is necessary; the file system must generate sufficient requests to keep the disks continuously busy. In HFS, this is accomplished by using ASF building blocks that direct prefetching requests to the memory manager.

For access pattern A on file structure A, prefetching should be done on a per-thread basis, since each thread is accessing a different portion of the file. Hence, a separate prefetching ASF building block should be used for each of the threads. The simplest possible prefetching algorithm is to, on the first request to a block, generate a prefetch request for the next block. With this prefetching policy, we obtain 4526 KBytes/sec. of disk bandwidth for the application, which corresponds to 100% of the disk bandwidth available on our system, or 646.6 KBytes/sec. per-disk. In this case, speedup is perfect (Figure 12).[21] When the same prefetch policy

[21] This 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

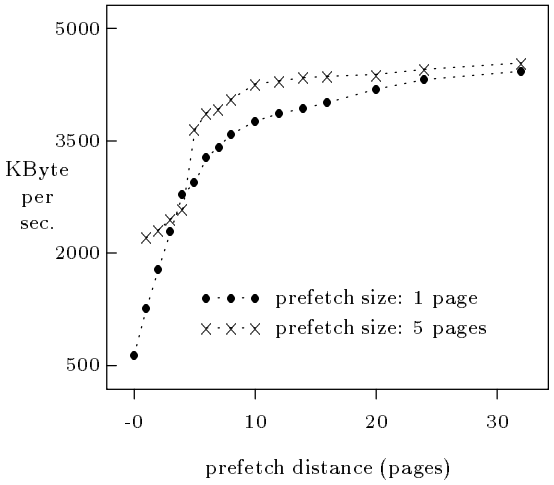

Fig. 13. Throughput in KBytes/sec. for access pattern B on file structure B. Prefetch distance is how far (in pages) in advance of the current offset that prefetch requests will be made. The prefetch amount is the number of pages that are requested on each prefetch.

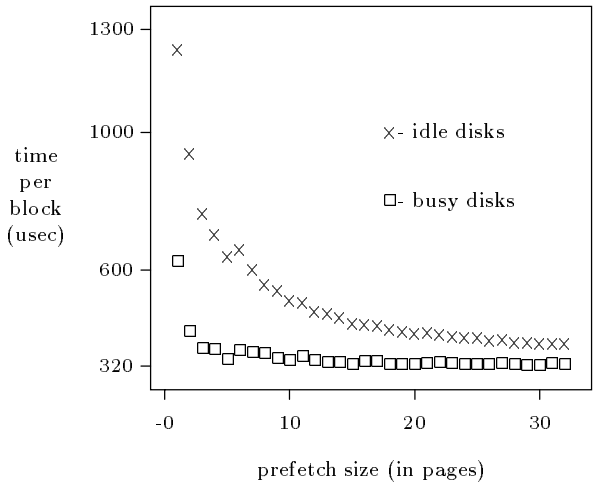

Fig. 14. The per-block processing overhead to execute prefetch requests for sequential file blocks with access pattern B run on file structure B. The top line shows performance for idle disks, while the bottom line shows performance for busy disks.

is used for access pattern A on file structure B, the application again only receives about one quarter of the disk bandwidth. This is again due to the large number of seek operations required. Performance is, however, slightly improved relative to the no-prefetching case, since sometimes a disk can service sequential requests by a thread before having to perform a seek to handle the requests of other threads.

For access pattern B on file structure B, prefetching should be done on a per-application rather than on a per-thread basis, since the entire file is being accessed sequentially. Also, in order to have a request outstanding for each disk, ASF should generate a prefetch request for the block at least seven blocks in advance of the one currently being accessed. Figure 13 shows the I/O performance as we vary the prefetch distance with prefetch request sizes of 1 and 5 pages. From the figure, we can see that while it is possible to exploit the full disk bandwidth of our system using such a policy, it is necessary to be aggressive, using a prefetch distance of thirty and a prefetch request size of five.

To understand why access pattern B requires such an aggressive prefetching policy, we measured the basic per-block software overhead of prefetching — that is, the system response time for an application prefetch request divided by the number of blocks requested. This is shown in Figure 14. When a disk is idle (the top curve), the time to execute a single block prefetch request is on average 1243 μsec. This high overhead comes from the cost to wake up a per-disk thread to issue the request to disk, the cost to cross address spaces boundaries to the memory manager, file system, device driver, and back to the application, as well as from the basic memory manager and file system overheads described earlier. If the application issues larger prefetch requests, then the per-block overhead drops quickly because the cost of crossing address spaces is amortized over more data. Also, if the disks are busy when a prefetch request is initiated (the bottom curve) then the cost is reduced because it is not necessary to wake up the per-disk thread.

When the same prefetch policy is used on file structure A for access pattern B, the application receives only about one seventh of the available disk bandwidth. As the threads sequentially access each file region, their requests are typically all directed to one disk, and hence the bandwidth of the other disks is not exploited.

In summary, when the access patterns are run on their matching file structure, the best prefetching policies differ in that (1) one is per-thread and the other is per-application; and (2) one is conservative, making requests for single blocks one block in advance, while the other is aggressive. If a per-thread prefetching policy were used for access pattern B, then many useless prefetch requests would be issued. If a per-application prefetching policy was used for access pattern A, then it would be entirely ineffective. While it would be possible to use an aggressive prefetching policy for access pattern A as well, it is both unnecessary and could be counterproductive if prefetched pages displace other pages that are still needed.

5.4.4 *Summary of parallel I/O results.* In this section, we have shown that it is crucial for the file structure and policy to match the application access pattern. For both access pattern A and B, HFS is able to deliver to the application address

---

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.

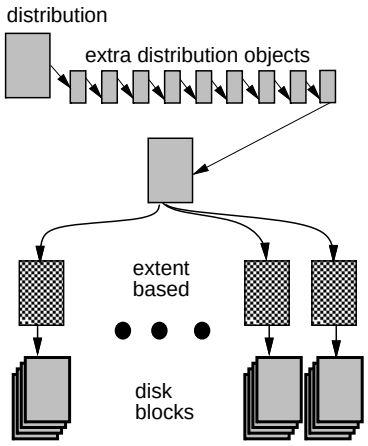

Fig. 15. Structure of a file distributed across seven disks with 10 extra layers of building blocks.

space 100% of the disk bandwidth of our system *if the file structures and policies match the application access pattern*. By using an appropriate structure, we are able to ensure that all disk requests are sequential and avoid seek operations. By using an appropriate prefetching policy, we are able to exploit the full concurrency of our disks while minimizing the memory and processing cost of prefetching.

5.5 Building-block overhead

One potential concern in using building blocks is the amount of overhead they might introduce. In a previous paper [Krieger et al. 1994], we showed that the overhead of ASF building blocks is negligible relative to other overhead. To show that the overhead at the physical layer is also low, we constructed a distributed file that implements file structure A (of the previous section), but added 10 levels of *distribution* building blocks between the top level building block and the per-disk building block (Figure 15). Figure 16 shows the file system performance with and without prefetching when access pattern A is used on this file.

In this case, HFS delivers 516 KBytes/sec. of data to a single application thread with no prefetching, about 1.2% worse than with no extra building blocks. This degradation is due to the file system overhead to traverse the extra building blocks, and amounts to about 300 μsec. for each disk block request, or about 30 μsec. per building block. The speedup with seven threads is 6.49 compared to 6.89 for the case with only one distribution building block. It is difficult to determine the source of this degradation. We suspect that increased memory contention is the culprit; the increased number of memory accesses to file structure data (building blocks and hash tables in this case) compete in the memory with the DMA accesses from the disks.

With prefetching, the file system delivers the full disk bandwidth of our system to the application threads with perfect speedup as the number of processors is varied from one to seven. Again, this speedup does not imply that there is no extra overhead, just that the overhead is masked by other time consuming operations.

We can conclude that the impact of implementing a file using many layers of

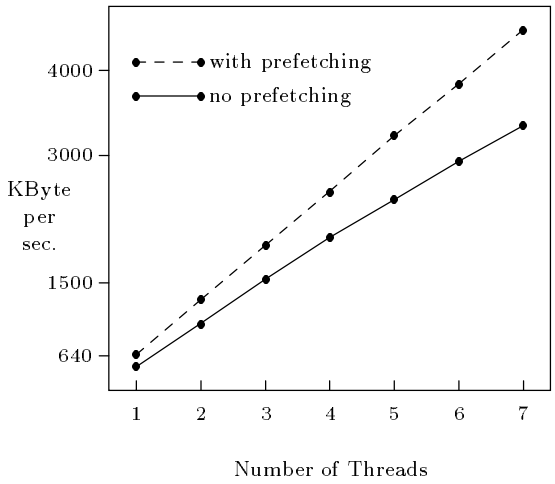

Fig. 16. Throughput for multiple readers of a single distributed file, with 10 extra levels of building blocks. The solid line shows the performance with no prefetching. The dashed line shows the performance with prefetching of a single block.

building blocks is small. The case considered is extreme in that eleven distribution building blocks suffice to create a file distributed across more than eight trillion disks. Moreover, while some impact in performance is observable, we believe that this impact is largely due to the particular characteristics of the Hector multiprocessor (i.e., slow memory system, lack of cache coherence, and the mechanism used to implement atomic operations).

6. LESSONS LEARNED

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 and memory bandwidth consumed copying data.

Overhead of crossing address spaces. 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 disk accesses frequently have low overhead. The Hurricane PPC facility [Gamsa et al. 1994] 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 physical layer of HFS.

Persistent building-block cache overhead. Our implementation first used a single cache of persistent building blocks, and we found that 80% of the time spent making a request to a cached persistent building block was spent (1) locating the building block in the cache, (2) enqueuing and dequeuing it from various queues (e.g., the free list of the cache), and (3) acquiring and releasing various locks. We now have multiple caches, each with different locking and replacement policies. The cache to be used for each type of building block is chosen to match the demands on that type of building block.[22] For most requests and types of building blocks, the time spent executing cache management code is now less than 50% of the total time taken to handle a request.

State distribution and replication. For good performance, we had to spend a large part of our implementation effort on developing techniques to distribute and replicate file system state across multiple memory modules to increase locality and avoid memory contention. This may have partly been an artifact of our ("mature") hardware base that was not cache coherent. However, we believe that similar efforts are necessary on more modern NUMA multiprocessors, both because file system state often does not remain in the cache between successive requests and because of such issues as false sharing [Gamsa et al. 1995; Parsons et al. 1995].

On-disk caches. We have found it important to take the behavior of disk caches into account. In particular, there is an interesting tradeoff between having the data for a request spread across multiple disks to increase concurrency and having data on consecutive disk blocks on a single disk to make effective use of the on-disk caches.

Interface compatibility. ASF allows the application to interleave requests to different interfaces, even if the requests are directed to a single stream.[23] 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 user-level I/O facility provides similar functionality, and we did not consider 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.

7. COMPARISON TO OTHER SYSTEMS

The HFS structure, based on building-block compositions, is unique. It provides file system support with unprecedented flexibility. HFS is more flexible than all other

[22] To illustrate how different policies are appropriate for different building blocks, consider distribution building blocks and per-disk random-access building blocks. The latter have a large amount of data and may be accessed for a long time. Hence, we maintain strict information about which was used most recently for replacement, and lock them using a reader/writer lock. The former are small and accessed for a short time. Hence, we can cache many and avoid keeping exact state about when each was used. Also, we can use an exclusive lock on distribution building blocks, since the lock will be held for just a short period of time.

[23] A 5 byte request to stdio, followed by a 20 byte Unix I/O request and a 5 byte ASI request, all to the same file, return the expected data.

existing and proposed parallel file systems we are aware of, including CFS [Pierce 1989] for the Intel iPSC, *sfs* [LoVerso et al. 1993] for the CM-5, PIOFS and Vesta [Corbett and Feitelson 1996; Corbett et al. 1995] for the SP2, XFS [Sweeney et al. 1996] for SGI multiprocessors, the OSF/1 file system [Zajcew et al. 1993], the nCUBE file system [DeBenedictis and del Rosario 1993], the Bridge file system [Dibble et al. 1988], the RAMA file system [Miller and Katz 1993], and the Galley file system [Nieuwejaar and Kotz 1997]. 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 HFS architecture is designed to support the following capabilities typically not available on other systems (although not all have been implemented).

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 thread can be used to determine the target disk for a read or write request. It is important to consider disk load in multiprogrammed environments, where other applications may be competing for the same disks. Locality is important in large scale NUMA systems, where the bandwidth available can depend on the distance between the processor and the disk. All other existing parallel file systems distribute file data across the disks using some static policy, 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. Applications can specify the prefetching policy on a per-open-file instance and per-thread basis. As we have seen in Section 5.4.3, this capability is crucial to allow applications to exploit the full disk bandwidth of our system. 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 can specify on a per-file basis the kind of redundancy (for fault tolerance) that should be maintained for its file's data and meta-data. Redundancy imposes a performance cost, but choosing a policy to match the target application's access pattern can reduce this overhead. All other existing parallel file systems we are aware of 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.

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

Parallel file I/O research is attempting to address the portability problem for I/O-intensive parallel applications by establishing standards for I/O interfaces [Corbett et al. 1995] and by developing facilities that can easily be ported above any native file system to provide a common application interface [Huber et al. 1995; Moyer and Sunderam 1994]. Also, there have been several projects to develop libraries and servers specifically targeted to meet the needs of scientific applications [Seamons et al. 1995; Thakur et al. 1994; Vengroff and Vitter 1995]. The flexibility of HFS allows it to be easily extended to support new interface standards as they are

developed. Moreover, because HFS supports common I/O interfaces, we expect it to be a simple target for libraries and servers that use native file systems for their I/O. Once a facility has been ported, it can then be incrementally optimized to take advantage of HFS's unique properties.

Most of the system development efforts discussed above have concentrated on particular aspects of I/O performance instead of taking a holistic approach as we did with HFS. These systems depend on other layers of system software to independently handle other concerns. For example, the XFS [Sweeney et al. 1996] file system for IRIX addresses scalability and concurrency issues in file system data structures, but assumes that lower layers of software efficiently manage the system disks. HFS spans all layers of the system, allowing an application to customize all aspects of I/O performance to match its specific requirements in a consistent fashion. For example, an HFS application can control everything from the way its threads accessing file data synchronize (in ASF building blocks) to how the file blocks are organized on the individual system disks (in physical-layer building blocks).

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 [Heidemann and Popek 1994] implement a file using "layer" building blocks in the same way that HFS uses building blocks. However, the goals of stackable file systems are very different from those of HFS, and hence the architectures have little in common. 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 building blocks are specific to a single file, the relationship between the building blocks is determined on a per-file (or per-open-file, or per-thread) basis, and most interactions between building blocks are between building blocks implemented in the same address space.

8. CONCLUDING REMARKS

We have developed building-block composition as a technique for structuring flexible file systems, and described the Hurricane File System that is based on this technique. We showed that even with a small number of simple building blocks, the ability to compose them gives the application tremendous flexibility in defining a large number of different file structures and file system policies. The performance results obtained from the HFS implementation on Hector/Hurricane demonstrate that (1) it is practical to implement a file system based on building-block compositions, (2) the overhead of this approach can be made very low, and (3) the flexibility is important for good performance.

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. Hence, it is reasonable to question whether the architecture of HFS is appropriate for multicomputer systems, and whether the techniques others have developed to optimize I/O performance can be adapted to mapped file I/O using building-block composition. While a multicomputer implementation will be quite

different from HFS, we believe that the building-block composition approach and the use of mapped file I/O applies equally well to multicomputer systems. Also, it appears to us that the techniques proposed by others to optimize performance can also be implemented in our environment; for example, we are studying adopting Kotz's disk-directed I/O [Kotz 1994] in our system.

HFS was developed in conjunction with Hurricane, a microkernel-based research operating system. The basic techniques of HFS, however, also fit well with systems with more monolithic operating systems, because customization occurs in the address space providing the service and does not require redirecting requests to other processes. Moreover, we believe the techniques of HFS are also applicable in current and future commercial systems, because flexibility and customizability is provided to applications by letting them compose *trusted* building blocks, and the compositions can easily be validated to be safe. This is in contrast to other more aggressive techniques for customizability that allow untrusted programmers to extend operating system functionality with new code [Bershad et al. 1995; Engler et al. 1995; Seltzer et al. 1996]. Also, it is already well accepted for I/O system software to employ object oriented technology for flexibility [Peterson et al. 1990; Ritchie 1984; Rosenthal 1990].

In this paper we have focused on validating the HFS design. Synthetic stress tests have been used to measure basic file system performance and demonstrate the benefits of flexibility. Since this paper was submitted, we have experimented with out-of-core scientific applications and demonstrated that HFS is able to achieve very good performance for these applications [Mowry et al. 1996]. These applications have working sets many times the size of the physical memory available on our system, and hence provided an extreme test of our system. We found that HFS could handle this new workload with only trivial modifications.

One of the most novel aspects of the more recent work ([Mowry et al. 1996]) was that the good performance was accomplished in a fully automated fashion; the code to exploit the features of HFS was generated by a compiler developed for this purpose. We believe that in the future such compiler technology will become more effective for a wider class of applications [Bordawekar et al. 1996; Cormen and Colvin 1994; Mowry et al. 1996]. This will simplify the introduction of new non-standard features such as those we propose, since one only has to change a compiler to exploit those features, and not a large set of applications.

The building-block composition technique we developed for HFS is now being employed by the Tornado multiprocessor operating system from the University of Toronto, and the Kitchawan multiprocessor operating system from IBM research [Auslander et al. 1997]. In this context, the HFS work is being extended in several ways. First, building-block compositions will be supported by all components of the new operating systems, including the memory manager. This will address one of the main limitations in HFS, namely the lack of flexibility in the memory manager, a crucial component for I/O performance. Second, HFS will be incorporated into these systems and will hence be ported to new hardware platforms. This will allow experiments to be performed on systems that are more modern and larger scale. Finally, HFS will be available from a very early stage on the new systems providing us with a much better insight into the advantages and limitations in using building-block compositions for file I/O.

ACKNOWLEDGMENTS

Ronald Unrau and Benjamin Gamsa developed much of the Hurricane infrastructure we depended on for this research and also contributed substantially to the design and implementation of the Hurricane File System. Ron White and Jan Medved implemented and maintained the hardware. We also gratefully acknowledge the help of Angela Demke, Karen Reid, Paul Lu, and Eric Parsons.

REFERENCES

Auslander, M., Franke, H., Gamsa, B., Krieger, O., and Stumm, M. 1997. Customization lite. In *Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS- VI)* (May 1997), pp. 43–48.

Bershad, B., Savage, S., Pardyak, P., Sirer, E., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th Symp. on Operating Systems Principles* (1995), pp. 267–284.

Bordawekar, R., Choudhary, A., and Ramanujam, J. 1996. Compilation and communication strategies for out-of-core programs on distributed-memory machines. *Journal of Parallel and Distributed Computing 38*, 2 (November), 277–288.

Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., Snir, M., Traversat, B., and Wong, P. 1995. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems* (April 1995), pp. 1–15.

Corbett, P. F. and Feitelson, D. G. 1996. The Vesta parallel file system. *ACM Transactions on Computer Systems 14*, 3 (August), 225–264.

Corbett, P. F., Feitelson, D. G., Prost, J.-P., Almasi, G. S., Baylor, S. J., Bolmarcich, A. S., Hsu, Y., Satran, J., Snir, M., Colao, R., Herr, B., Kavaky, J., Morgan, T. R., and Zlotek, A. 1995. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 222–248.

Cormen, T. H. and Colvin, A. 1994. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243 (November), Dept. of Computer Science, Dartmouth College.

Crandall, P. E., Aydt, R. A., Chien, A. A., and Reed, D. A. 1995. Input/output characteristics of scalable parallel applications. In *Proc. of Supercomputing '95* (Dec. 1995).

Crockett, T. W. 1989. File concepts for parallel I/O. In *Proceedings of Supercomputing '89* (1989), pp. 574–579.

DeBenedictis, E. P. and del Rosario, J. M. 1993. Modular scalable I/O. *Journal of Parallel and Distributed Computing 17*, 1–2 (Jan/Feb), 122–128.

del Rosario, J. M. and Choudhary, A. 1994. High performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer 27*, 3 (March), 59–68.

Dibble, P., Scott, M., and Ellis, C. 1988. Bridge: A high-performance file system for parallel processors. In *Proc. of the Eighth International Conference on Distributed Computer Systems* (June 1988), pp. 154–161.

Druschel, P. 1993. Efficient support for incremental customization of OS services. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems* (Asheville, NC, Dec. 1993), pp. 186–190.

Engler, D., Kaashoek, F., and Jr, J. O. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symp. on Operating Systems Principles* (1995), pp. 251–267.

Feitelson, D. G., Corbett, P. F., Baylor, S. J., and Hsu, Y. 1995. Parallel I/O subsystems in massively parallel supercomputers. *IEEE Parallel and Distributed Technology 3*, 3, 33–49.

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

Galbreath, N., Gropp, W., and Levine, D. 1993. Application-driven parallel I/O. In *Proc. Supercomputing* (1993), pp. 388–395. IEEE Comput. Soc. Press.

Gamsa, B., Krieger, O., Parsons, E. W., and Stumm, M. 1995. Performance issues for multiprocessor operating systems. Technical Report CSRI-339 (November), Computer Systems Research Institute, University of Toronto.

Gamsa, B., Krieger, O., and Stumm, M. 1994. Optimizing IPC performance for shared-memory multiprocessors. In *Proc. 1994 Intl. Conf. on Parallel Processing (ICPP)* (Boca Raton, FL, Aug. 1994), pp. 208–211. CRC Press.

Grimshaw, A. S. and Loyot, E. C., Jr. 1991. ELFS: Object-oriented extensible file systems. In *Proc. of the First International Conference on Parallel and Distributed Information Systems* (1991), pp. 177–179.

Heidemann, J. S. and Popek, G. J. 1994. File-system development with stackable layers. *ACM Transactions on Computer Systems 12*, 1 (Feb.), 58–89.

Huber, J., Elford, C. L., Reed, D. A., Chien, A. A., and Blumenthal, D. S. 1995. PPFS: A high performance portable parallel file system. In *Proc. of the 9th ACM International Conference on Supercomputing* (Barcelona, July 1995), pp. 385–394.

Intel. 1989. Concurrent I/O application examples. Intel Corporation Background Information.

Khalidi, Y. A. and Nelson, M. N. 1993. Extensible file systems in Spring. In *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles* (1993), pp. 1–14.

Kotz, D. 1994. Disk-directed I/O for MIMD multiprocessors. In *Proc. of the 1994 Symposium on Operating Systems Design and Implementation* (Nov. 1994), pp. 61–74.

Krieger, O. 1994. *HFS: A flexible file system for shared memory multiprocessors.* Ph. D. thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada.

Krieger, O., Stumm, M., and Unrau, R. 1994. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer 27*, 3 (March), 75–82.

Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J. 1994. The Stanford FLASH multiprocessor. In *Proc. of the 21st International Symposium on Computer Architecture* (Chicago, IL, April 1994), pp. 302–313. ACM.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W. D., Gupta, A., Henessy, J., Horowitz, M., and Lam, M. S. 1992. The Stanford DASH multiprocessor. *IEEE Computer 25*, 3 (March), 63–79.

Liedtke, J. 1993. Improving IPC by kernel design. In *Proc. of the Fourteenth ACM Symposium on Operating System Principles* (North Carolina, Dec. 1993), pp. 175–188.

Lin, Z. and Zhou, S. 1993. Parallelizing I/O intensive applications on a workstation cluster: a case study. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems* (1993), pp. 17–36.

LoVerso, S. J., Isman, M., Nanopoulos, A., Nesheim, W., Milne, E. D., and Wheeler, R. 1993. *sfs*: A parallel file system for the CM-5. In *Proc. of the 1993 Summer Usenix Conference* (1993), pp. 291–305.

Massalin, H. and Pu, C. 1989. Threads and input/output in the Synthesis kernel. In *Proc. of the Twelfth Symposium on Operating Systems Principles* (Arizona, Dec. 1989), pp. 191–201.

Miller, E. and Katz, R. 1991. Input/output behavior of supercomputing applications. In *Proc. Supercomputing* (Nov. 1991), pp. 567–76. IEEE Comput. Soc. Press.

Miller, E. L. and Katz, R. H. 1993. RAMA: A file system for massively-parallel computers. In *Proc. of the Twelfth IEEE Symposium on Mass Storage Systems* (1993), pp. 163–168.

Mowry, T. C., Demke, A. K., and Krieger, O. 1996. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (October 1996), pp. 3–17. USENIX Association.

Moyer, S. A. and Sunderam, V. S. 1994. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proc. of the Scalable High-Performance Computing Conference* (1994), pp. 71–78.

Nieuwejaar, N. and Kotz, D. 1997. The Galley parallel file system. *Parallel Computing 23*, 4 (June), 447–?

Parsons, E., Gamsa, B., Krieger, O., and Stumm, M. 1995. (De-)clustering objects for multiprocessor system software. In *Proc. 4th Intl. Workshop on Object Orientation in Operating Systems 95 (IWOOOS'95)* (1995), pp. 72–81.

Patterson, D., Gibson, G., and Katz, R. 1988. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference* (June 1988), pp. 109–116.

Peterson, L., Hutchinson, N., O'Malley, S., and Rao, H. 1990. The x-kernel: A platform for accessing internet resources. *IEEE Computer 23*, 5 (May), 23–33.

Pierce, P. 1989. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications* (1989), pp. 155–160.

Poole, J. T. 1994. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech.

Ritchie, D. 1984. A stream input-output system. *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct.), 1897–1910.

Rosenblum, M. and Ousterhout, J. K. 1991. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 1–15. Association for Computing Machinery SIGOPS.

Rosenthal, D. S. H. 1990. Evolving the vnode interface. In *USENIX Conference Proc.* (Anaheim, CA, Summer 1990), pp. 107–118. USENIX.

Scott, D. S. 1993. Parallel I/O and solving out of core systems of linear equations. In *Proc. of the 1993 DAGS/PC Symposium* (Hanover, NH, June 1993), pp. 123–130. Dartmouth Institute for Advanced Graduate Studies.

Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., and Winslett, M. 1995. Server-directed collective I/O in Panda. In *Proc. of Supercomputing '95* (Dec. 1995).

Seltzer, M., Endo, Y., Small, C., and Smith, K. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd Symp. on Operating Systems Design and Implementation* (1996), pp. 213–228.

Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., , and Peck, G. 1996. Scalability in the XFS file system. In *USENIX Technical Conference* (Jan. 1996), pp. 1–14. Usenix.

Thakur, R., Bordawekar, R., Choudhary, A., Ponnusamy, R., and Singh, T. 1994. PASSION runtime library for parallel I/O. In *Proc. of the Scalable Parallel Libraries Conference* (Oct. 1994), pp. 119–128.

Unrau, R. C., Krieger, O., Gamsa, B., and Stumm, M. 1994. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Operating System Design and Implementation* (Nov. 1994), pp. 139–152.

Unrau, R. C., Krieger, O., Gamsa, B., and Stumm, M. 1995. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing 9*, 1/2, 105–134.

Vengroff, D. E. and Vitter, J. S. 1995. I/O-efficient scientific computation using TPIE. In *Proc. of the 1995 IEEE Symposium on Parallel and Distributed Processing* (Oct. 1995), pp. 74–77.

Vranesic, Z. G., Stumm, M., White, R., and Lewis, D. 1991. The Hector Multiprocessor. *IEEE Computer 24*, 1 (Jan.), 72–80.

Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., and Netterwala, D. 1993. An OSF/1 UNIX for massively parallel multicomputers. In *USENIX Winter Conference* (Jan. 1993), pp. 449–468. Usenix.