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

Orran Krieger and Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto

## Abstract

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. For example, a file's structure can be optimized for concurrent random-access write-only operations by ten processes. Similarly, the prefetching, locking, and file cache management policies can all be chosen to match an application's access pattern. In contrast, most existing 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.

## 1 Introduction

Many important supercomputer applications have massive I/O requirements [26, 5, 8, 19, 21]. However, 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 [5, 3]. For this reason, there is little understanding of the requirements of I/O-intensive parallel applications, and hence little understanding of the file structures, policies, and I/O interfaces a parallel file system should provide in order to best satisfy these requirements.

In this paper, we describe the architecture of the Hurricane File System (HFS), designed for (potentially large-scale) shared-memory multiprocessors with many disks distributed across the system. The key characteristic of HFS that sets it apart from other file systems is its flexibility in being able to support a large, expandable set of file structures, policies, and I/O interfaces, and to do so with low processing and I/O overhead. The flexibility of HFS allows it to accommodate new uses, new systems, new operating system architectures, and new I/O interfaces. Moreover, we show that this flexibility is crucial in allowing a large class of I/O-intensive applications to exploit the full I/O bandwidth of the disks attached to the system.

HFS allows an application to customize the file structure and policies to match its I/O requirements. For example, the application can choose how data is to be distributed across the disks, how the data is to be distributed on each of the disks, the policies (if any) to be used to store the file data with redundancy (for fault tolerance or read performance), the latency hiding policies, advisory and enforced synchronization policies, and the compression/decompression policies that are to be invoked (transparently) by the file system when the file is opened. HFS is more flexible than any other existing and proposed parallel file system we are aware of, including CFS [25] for the Intel iPSC, *sfs* [20] for the CM-5, Vesta [1] for the SP1, the OSF/1 file system [34], the nCUBE file system [4], the Bridge file system [6], and the RAMA file system [22].

Flexibility in our system is achieved by having a large number of fine-grained building blocks that can be composed by the user to create a file with the desired properties. In the next section we describe the technique used. 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 part of the Hurricane operating system [31] on the Hector shared-memory multiprocessor [33].

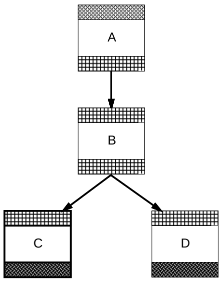


Figure 1: *This HFS building block composition implements a file. Objects* C *and* D *may each store data on a single disk, object* B *might be a distribution object that distributes the file data to* C *and* D*, and object* A *might be a compression/decompression object that de-compresses data read from* B *and compresses data being written to* B.

# 2 Building-block composition

Each HFS file is implemented by combining together a set of what we call *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 file data into main memory, enforce security, manage locks, and interact with the memory manager to manage the cache of file data. Building blocks are implemented as objects and thus contain state and a set of operations that manipulate the state.

A building-block object exports an interface that specifies the operations that can be invoked by other objects. It may also import (one or more) interfaces that are exported by other building-block objects. Two building blocks are said to be *connected* if one of them might invoke operations of the other, and the object 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 objects and the way they are connected) determines the behavior and performance of the file. As a simple example, Figure 1 shows four building-block objects 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 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 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, object $B$ could be replaced by another object $B'$ 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 object $F$ between $A$ and $B$ that is connected to both $B$ and a second subtree rooted by another distribution object similar to $B$.

Finally, it is possible to support new interfaces to applications by introducing new building-block objects 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 object $B$ above might execute only 2–3 lines of C code in a typical flow of control through the object. Similarly, the larger the number of building-block classes with identical interfaces, the more flexibility in (re-)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 along connections by invoking an exported operation. The flow of control is thus functional and synchronous. 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[1], the building block code is executed in the context of a process that is created as a by-product of the Hurricane protected procedure call (PPC) facility [9].

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 (in part) modify the existing building-block composition of files they access. The arguments to an object's constructor specifies how that object is connected to other existing building blocks.

In some cases it is problematic to change the composition of a file after it has been created. For example, it is difficult to dynamically change the objects that control how a file is distributed across the system disks. However, objects that control, for example, prefetching can be changed at any time 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 reasons. First, even if a file is implemented with many layers of objects, requests are often serviced by traversing only a small number of layers. For example, many building blocks cache data so that read and write requests can often be satisfied from a cache managed by an object that is close to the client.

Second, our approach naturally minimizes the amount of cross-address space communication. Building-block composition differs from other approaches for customizing system software in that the customization does not occur by having the system redirect requests to the application address space or to a separate server [7, 11, 14]. With our approach, the customization occurs in the server providing the service, thus naturally avoiding cross-address space communication.

Third, we have found it possible to define object interfaces that have extremely low overhead. For example, the interfaces provided by most building blocks we have defined allow control to pass through multiple building blocks without requiring data to be copied.

Finally, if the use of many building blocks happens to result in poor performance for some important new workload, then it is straightforward for a systems programmer to define and add a new object class to specifically handle the demands of this workload, possibly by combining a set of simple objects into a single more complex object.

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

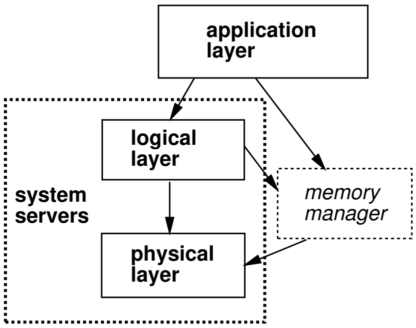


Figure 2: *This figure illustrates the relationship between the different components of the Hurricane file system. The Alloc Stream Facility (ASF) is an application-level I/O library. The physical layer of the HFS file system implements files and controls the system disks. The logical layer server provides 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.*

# 3 Architecture and implementation

The Hurricane File System (HFS) implements files using building-block compositions. HFS is logically divided into three layers, namely: the application layer, the physical layer, and between these two layers, the logical layer (Figure 2). The Alloc Stream Facility (ASF) is an I/O library that makes up the application layer of HFS [17]. We provide as much functionality as possible in this layer in order to minimize communication with servers in different address spaces. The physical and logical layers of HFS are system servers of the Hurricane operating system. The physical layer implements files and controls the system disks. The logical layer provides file system authentication services. ASF ensures that most file I/O occurs through mapped files, so the memory manager is involved in most requests to read or write file data. We describe each of the three layers of HFS in turn.

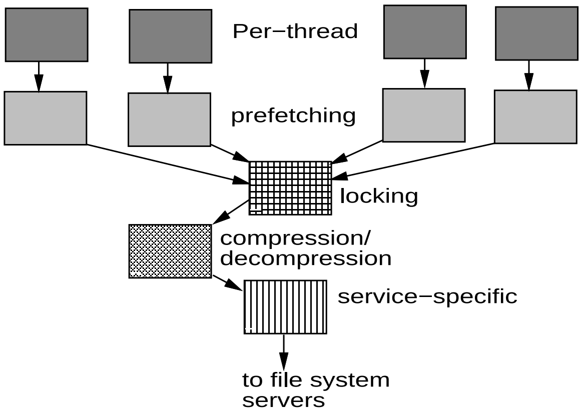


Figure 3: *Example of an ASF building-block composition used by a multi-threaded application to access a file stored on disk in a compressed format. All building blocks other than the service-specific one import and export the ASI interface. The application is assumed to use ASI directly, so no interface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

## 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 to applications a number of different I/O interfaces (e.g., Unix read/write and stdio) [17]. It 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, 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.[2]

The implementation of ASF supports three types of building-block classes: $i$) ASI classes, $ii$) service-specific classes, and $iii$) interface classes. ASI classes import and export the ASI interface. We have defined ASI classes 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 also defined ASI classes that allow applications to have customized views of file data, such as classes similar to those defined by the ELFS file system [10] that take into account the semantic structure of a file (e.g., a file containing a two dimensional matrix), and classes that implement the mapping functions defined by the Vesta file system [1]. Because ASI building blocks import the same interface they export, they can be connected in arbitrary ways.

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, 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, for each I/O service, a separate class for read-only, write-only, and read/write access, respectively. Using separate, direction-specific building blocks makes them more efficient, 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. We have implemented building blocks that export 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 implemented them.

Figure 3 illustrates how ASF objects 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 building block that maintains the file offset for that thread. Since each thread accesses data sequentially, but the file as a whole is not being accessed sequentially, each per-thread object references an independent prefetching building block. The prefetching building blocks all reference a single building block that implements some (advisory) locking policy.[3] This building block, in turn, references a compression/decompression object, which decompresses data read from the file and compresses data being written to the file. Finally, the compression/decompression

[2] While having a common interface is important, it is also important that objects be able to export additional class specific functions for extensibility. For example, if a file system supports prefetching, the service-specific objects 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 of Heidemann and Popek [11]) that handles any class specific requests. When an object cannot directly interpret a special request, it re-directs it to the objects it references.

[3] 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.

building block makes requests to access file blocks to a service-specific object, which maps the file into the application address space, and translates I/O requests for file data into accesses to the mapped region.

The flexibility achievable with these building-block compositions is entirely under the control of the application. The application that creates a file can specify which ASF objects should be instantiated automatically when the file is opened. 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 from the application the fact that compression is being done. This allows standard utilities, such as editors, to process compressed files without knowledge of this fact.

Once a file has been opened and all default application-layer objects have been instantiated, the 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. It is also important to be able to change a composition dynamically to adapt to the different access patterns an application might have in different phases, and hence benefit from building blocks specific to each particular access pattern. For example, an application traversing a matrix first by row and then by column would benefit dramatically from changing the ASF objects 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 is responsible for disk block placement; that is, 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 can exploit the facilities of the physical layer. This allows, for example, 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 interface directly with the disks. Again, this allows building blocks to be composed in a wide variety of ways, and hence allows for a great deal of 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-block layers.

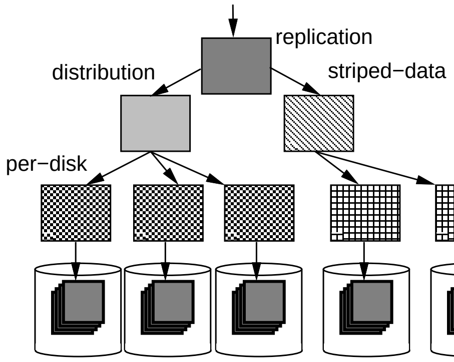


Figure 4: *Example building-block composition in the physical layer of the HFS file system.*

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; *random access* building blocks always write file blocks to a (new) location near the current disk head position; 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 objects,

**distribution:** partitions a file into contiguous regions, each stored by a different referenced object,

**write-mostly:** takes into account disk proximity or load to optimize write performance,

**replication:** replicates data to each referenced object,

**parity:** computes and stores parity information for fault recovery,

**application specific distribution:** maintains an application specified table that defines how data is distributed to referenced objects, and

**small data:** stores the data of a small file together with the file meta-data.[4]

These building blocks are for the most part very simple. (See [16] for a more detailed description of these building blocks.)

Figure 4 illustrates how the simple building blocks of the physical layer can be combined together in a composition to create very complex file structures that traditional file systems cannot provide. In this example, a replication building block replicates data to a distribution building block and a striped data building block. Both, in turn, distribute data across a number of per-disk building blocks. Such a file structure could be useful for a read-mostly

[4] An arbitrary number of small data building blocks can be packed into a single disk block to minimize disk fragmentation.

file, where data is accessed in different ways at different times. The replica based on the distribution object would be useful for a multi-threaded application where each thread accesses a disjoint region of the file, while the replica based on the striped-data building block would be useful for a multi-threaded application with threads that make many small random-access I/O requests.

The structure of a file is specified by the application at creation time by instantiating the building blocks in a bottom up fashion. In Figure 4 the per-disk building blocks for each of the two replicas would be instantiated first, followed by the distribution and striped-data building blocks, and finally the replication building block. Requests to instantiate a building block pass references to the lower-level building blocks to which the new block is to be connected. As the file grows, the file system may instantiate new objects to store the additional data, but does so while retaining the basic file structure defined by the application.

Unlike application-layer objects, the physical-layer objects associated with a file are persistent and cannot be changed at run time.[5] 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.

## 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. For example, 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 particular class of object 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.

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

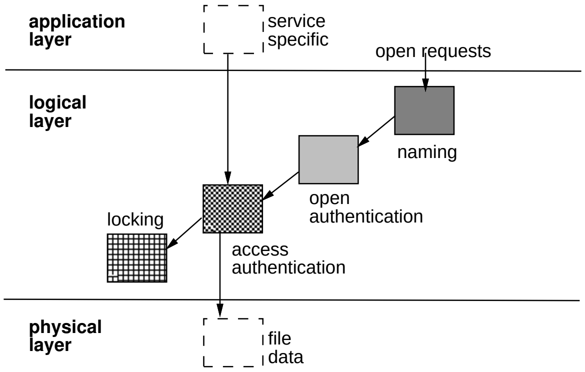


Figure 5: An example of a logical-layer building block composition.

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 objects that use the pathname specified by the application to locate the open-authentication building block for the file. After authenticating the request, say by using an access list based policy, these objects return to ASF information which identifies both the application-layer objects to be instantiated (by default) and a handle of the access-authentication object 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 object 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 access-authentication building block forwards the mapping request to the memory manager. Otherwise, the access-authentication building block satisfies read and write requests by making corresponding requests to the physical layer.

As with physical layer objects, the logical layer objects are persistent and instantiated by the application when the file (or directory) is created. Except for small files, we have found that mapped file I/O results in the best performance, and hence files in our system are typically accessed in this manner. In contrast to the other layers we have described, the logical layer has little impact on performance when mapped files are used, and as a result we have done much less work in developing the logical layer of HFS than the other layers. However, 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 request (e.g., using access lists or Unix style permissions) and enforcing locking protocols.

# 4 Minimizing application complexity

One of the challenges in providing a flexible system is to minimize the complexity an application programmer is confronted with. While we have found writing application-level code to create HFS files to be relatively easy, we do not in general expect application programmers to write this code. Instead we believe that a number of approaches will allow even novice users to exploit the flexibility of our file system.

**ASF defaults:** The application-level library can create files using reasonable defaults if no information is available from the application. Initially, it assumes that a file will be small, and uses a *small data* building block to store the file data. If this turns out to be incorrect (when the application writes more data than can be held by a *small-data* building block), the application-level library assumes that the file will be medium sized and should be distributed across a small number of disks. In this case, the library will create a new file, and copy the data from the original to the new file.

**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. We expect that the choice of routine, together with the parameters specified on calling it, will provide sufficient flexibility for many applications.

**Compiler support:** For many parallel applications, much of the I/O is a result of paging activity due to insufficient physical memory to hold the entire application image in memory at the same time. This is entirely transparent to the application. We expect that the files used to page different parts of the application memory image will be created by the compiler and that future compilers will be able to take the requirements of the application and the capabilities of the I/O system (i.e., the number and latency of the disks, and the capabilities of the file system) into account to select appropriate building blocks. This is similar to the task an HPF compiler performs in taking into account the characteristics of the application and system when distributing data across the memory modules of the system.

**Automatic restructuring:** Finally, it should be possible to implement objects that keep statistics on how the file is being accessed, periodically re-evaluate the choice of file structure, and restructure if necessary.

# 5 Evaluation

We have developed building-block composition as a technique for providing flexibility in file structure and policy and the Hurricane File System (HFS) employs this technique. It is difficult to quantitatively measure flexibility, but we will argue in Section 7 that HFS is significantly more flexible than other multiprocessor file systems. In this section we describe our experiences with, and show performance results from, our implementation of HFS. The purpose 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.

We have implemented HFS as part of the Hurricane operating system on the Hector shared memory multiprocessor. Also, the ASF application-level library has 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 [17]. The remaining sections show performance results on Hector/Hurricane. Section 5.2 provides measurements of the experimental environment and of some basic file system operations. The next section 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, we show that the software overhead of our approach is negligible.

## 5.1 Experiences

Building block composition has a number of software engineering advantages. Prior to developing HFS, we developed an earlier file system for Hurricane that did not support building-block compositions. Similar to most other file systems, it had code specific to each type of file (e.g., a replicated file, a distributed file, etc.) and had separate code for storing each type of file system meta-data. With this earlier system, the programming effort to add a new type of file was high. Employing building-block compositions, HFS was less complicated to implement, even though better performance and a great deal more flexibility was achieved.

One of the ways to evaluate flexibility is to see how easy it is to adapt the system to a different environment. Since HFS requires low-level control over the disks, it would have been difficult to port the entire file system to a different platform. However, we did port the ASF application-level library to a number of systems, including

SunOS, IRIX, AIX, HP-UX, and Tornado. We found that the library was easy to port; it was only necessary to change some of the service-specific building blocks to adapt to a new platform. ASF is also being used by Tera Computer Company as an I/O library for their parallel supercomputer.

Another measure of flexibility is how easy is it for the system to handle requirements not foreseen by its developers. We initially had no intention of making crash recovery fast, and in our initial implementation, recovering from a system crash involved re-formatting the disks; the time this required made the file system essentially unusable in our environment (with an experimental operating system running on experimental hardware). 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.

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 largely 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 that could be performed.

Due to the disadvantages of the Hurricane/Hector platform, we have confined our implementation goals to producing a proof of concept prototype. In addition to the building blocks, implementing HFS required a great deal of infrastructure to, for example, cache persistent building blocks in main memory, marshal and demarshal arguments on IPC requests, and maintain per-open state. We have implemented all of this infrastructure and many, but not all, of the building blocks described in the previous sections. For the physical layer, we have complete implementations of the three per-disk, the distribution, and the striped-data building blocks. For the application layer, we have complete implementations of a number of stream-specific building blocks, three ASI building blocks (locking, prefetching and line buffering), and three interface building blocks.

The prototype is sufficiently complete to answer several important questions about the HFS architecture. First, we believe that the prototype demonstrates that the full HFS architecture is implementable. For the physical and application layers, we have gained sufficient experience to be confident that all building blocks described in the previous sections can be implemented without requiring major changes to either the common interfaces or the code for these layers that is not class-specific. 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.

Second, the prototype clearly shows that the overhead of building-block compositions is low, and in fact negligible. We will show that HFS is capable of delivering the full I/O bandwidth of the disks on our system to the application. Moreover, even with a composition that uses many layers of building blocks, the processing overhead of the file system is low compared to other processing overhead involved in I/O.

Finally, we demonstrate that flexibility not available in other systems is useful. In particular, using two synthetic stress tests with very different access patterns, we show that the flexibility of HFS makes it possible to deliver 100% of the disk bandwidth of our system to the application address space for both access patterns. The need for flexibility demonstrated by these tests is clearly insufficient to warrant all the features of HFS. Demonstrating this in a realistic way would require extensive work with real applications, which is not feasible in the context of our prototype environment. On the other hand, the advantages shown, coupled with the fact that there appear to be no major disadvantages to our architecture from either an overhead or complexity perspective, suggest that further development is warranted. As we will describe later, we will be porting HFS to a more modern larger-scale platform and expect to investigate it extensively using real I/O-intensive applications.

## 5.2 Infrastructure

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

#### Hardware infrastructure

The hardware platform used for our experiments is a prototype of the Hector multiprocessor [33] depicted in Figure 6. The system uses Motorola 88100 processors running at 16.67 MHz, with 16KB data and instruction caches (88200), and a local portion of the globally-addressable memory. At the lowest level, processors are connected by a bus. These lowest level units, called stations, are then connected together by a high-speed ring, with multiple rings connected together by higher-level rings, and so on. The particular hardware configuration used in our experiments consists of 4 processing modules per station and 4 stations connected by a ring. Hector does not support cache coherence, and for this reason many of the internal data structures used in HFS are not cached.

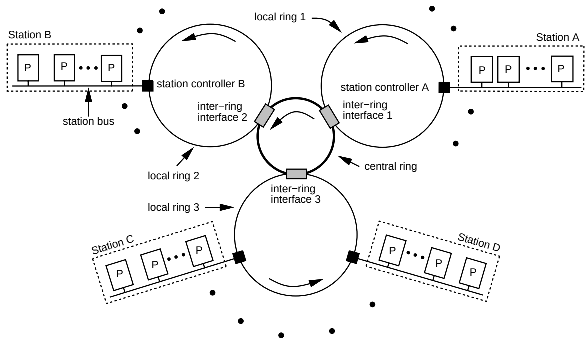


Figure 6: A high-level representation of the Hector multiprocessor. Each processor box in the system includes a CPU and cache, local memory, and I/O interface.

Seven Conner CP3200 disks are connected to the Hector prototype, each directly connected to a different processing module. 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 in the system.[6] In all the experiments shown, data read from disk is distributed round robin across the different memory modules in the system. The maximum per-disk bandwidth available on our system, measured at the lowest level by instrumenting the device driver, is 646 KBytes/sec.

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. In some cases, performance can vary substantially depending on the locality of the data used by the file system or memory manager to service the request.

**Operating system infrastructure**

The Hurricane File System was developed as part of the Hurricane operating system [31, 30]. Hurricane is a micro-kernel operating system, where most of the operating system services are provided by user-level servers. The micro-kernel provides for basic process management, interprocess communication and memory management.

The interprocess communication facility of Hurricane, *Protected Procedure Calls* (PPC), has three important characteristics: 1) it is fast, 2) in the common case no locks are acquired and only local data is accessed, and 3) new workers are dynamically created (as required) to handle requests in the server address space [9]. The speed and concurrency of the facility are obviously important, minimizing the impact of splitting the file system into multiple servers and ensuring that IPC is not a bottleneck on file system requests.[7] The automatic creation of new workers is important, since it allows the concurrency in the file system to match the demands being made on it by application processes. The cost to make a PPC request between two application address spaces varies between 35 and 70 $\mu$sec depending on how much of the state required for the PPC is in the processor cache.

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. The Hurricane memory manager supports prefetch and poststore operations that allow ASF to request that a (potentially) large number of pages be asynchronously fetched from or stored to disk. A limitation of the memory manager that we have not yet addressed is that it cannot allocate contiguous physical page frames so that contiguous disk blocks can be read from disk using a single disk request.

The memory-management overhead to handle a read-page fault when the page is in the file cache is on average 200 $\mu$sec on a 16 processor system. If the data is not in the file cache, the memory-manager overhead to initiate a request to the file system is about 800 $\mu$sec.[8] The memory management overhead for a prefetch request is 60 $\mu$sec per request and an additional 200 $\mu$sec per page if the page is not in memory, 30 $\mu$sec per page if the page is in memory but not in the local page table, and 10 $\mu$sec per page if the page is already in the local page table.

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

## 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 implemented by an extent-based building block on a single disk.

[6] 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.

[7] The performance of our multiprocessor PPC calls (including the creation of a process in the target address space) is competitive with the fastest known uniprocessor IPC implementations [18].

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

We then measured the performance when a single process reads data into memory on the same processing module as the disk, into memory on the same station as the disk, and into off-station memory. The data rate delivered to the application varies between 515 and 540 KBytes/second. 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 process local to that disk to initiate the I/O request.[9] Second, since there is no request waiting for the disk when a request completes, there is a large latency between the time a request completes and the next request is sent to the disk.

We ran the same experiment but with the reading process always issuing a prefetch request to the page succeeding the page it is about to access. In this case, the file system delivers to the application 646 KBytes/sec, which corresponds to 100% of the available bandwidth of a single disk on our system. Even with prefetching, there is some delay between the time a disk request completes and the next request arrives. This delay is about 200 $\mu$secs and includes the time to preempt the running process, the file system overhead, and the PPC costs. The reason this delay does not degrade performance is that the disk prefetches data into the on-disk cache when it is not servicing requests from the file system. Hence, the disk can tolerate some delay between the end of one request and the start of the next request without loss of performance.

## 5.4 Parallel I/O

We have argued that a file system should have enough flexibility so that the file structures and policies can match application requirements. In this section we look at two simple file access patterns, and show the importance of matching the applications' requirements. The two access patterns are depicted in Figure 7. With access pattern A, seven processes of a parallel application concurrently read a file, with each process sequentially reading a different region of the file. With access pattern B, the processes of a parallel application cooperate to read the file sequentially, each reading a small amount of data at a time.

### 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 different disks, ensures that the requests of one process don't delay requests from other processes, and allows the sequential requests of each process to be mapped to sequential 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 8. 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 size of the regions are specified when the file is first created. 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.[10]

The best way to support access pattern B is to stripe the file across all disks in the system, thus balancing the load uniformly across the disks and giving each disk the maximum time to pre-fetch into its on disk cache before the next request arrives from the application. The HFS building blocks used to implement file structure B are shown in Figure 8. The striped-data building block stripes file blocks round robin across seven extent-based building blocks.[11] As with file structure A, the blocks on each disk are stored by extent-based building blocks.

The similarities between the two file structures shown in Figure 8 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.

We measured the performance of the file system with access patterns A and B on both file structures A and B. In all cases the data is read into pages distributed round-robin across the memory modules. The seven processes are scheduled round robin to the 16 processors on our system, and the experiment run repeatedly until every sequence of seven processors had been used. The processes access the file using ASF objects that map the file into the application address space, hence all requests are initiated by page faults. For access pattern A, each process uses an independent ASF object to maintain a separate file offset. For access pattern B, a single shared ASF object is used.

When access pattern A is run on file structure A, the file system delivers to the application about 3597 KBytes/sec of disk bandwidth. To find out how much concurrency

[9] When a disk request completes, the file system code that handles the interrupt de-queues any pending requests from a shared queue. As long as the disk does not become idle, it is not necessary to wake up the per-disk process; new requests are placed on the shared queue to be handled when the current request completes.

[10] Each of the extent-based building blocks is stored on the same disk as the disk blocks it controls, ensuring that file system requests for meta-data do not interfere with the file data requests by application processes. The only common building block for the seven regions is the distribution block, and since the operations performed by this building block are very simple (and hence entail holding a lock for just a short period of time), it is not expected to be a source of contention.

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

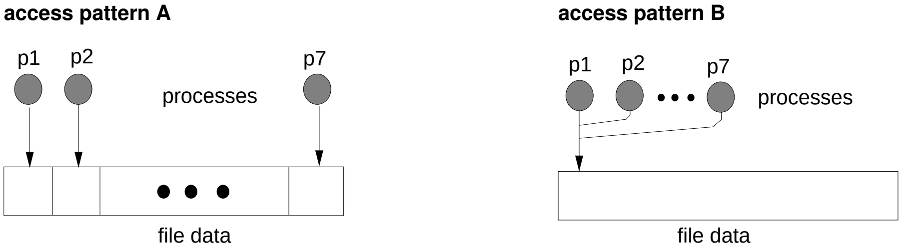


Figure 7: Two file access patterns. With access pattern A, seven processes of a parallel application concurrently read a file, where each reads a different region sequentially. With access pattern B, the processes of a parallel application cooperate to read the file sequentially, each reading a small amount of data at a time.

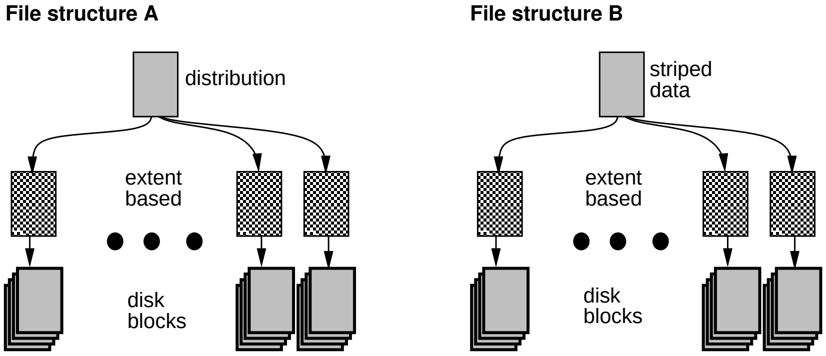


Figure 8: Files distributed across multiple disks. File structure A implements a file using a distribution building block that partitions the file into seven contiguous regions, each stored by a different extent-based building block. File structure B implements a file using a striped-data building block that stripes file blocks round robin across seven extent-based building blocks.

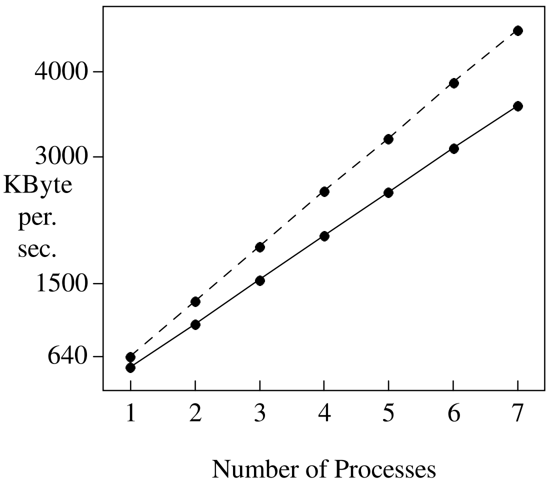


Figure 9: Throughput for concurrent reads of a distributed extent-based file. The solid line shows the performance with no prefetching. The dashed line shows the performance with prefetching of a single block.

there is handling these requests, we varied the number of requesting processes between one and seven and measured the speedup in terms of KBytes/sec. As shown in Figure 9 the speedup with seven processes is 6.89. While we did not obtain perfect speedup, it turns out that none of the file system locks are heavily contented, 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) preemption of the file system and application processes due to disk interrupts, and 3) contention for kernel data structures such as the ready queues.

When access pattern B is run on file structure B, the application receives 640 KBytes/sec of file data. Since there is only one request outstanding at a time, the file system cannot keep multiple disks busy. However, performance is somewhat better than when a single process accesses a file stored on a single disk; since requests are made to each disk in turn, each disk has a long time to prefetch data into its on-disk cache between successive requests to that disk.

When access pattern B is run on file structure A, the file system delivers 520 KBytes/sec, the same performance as if the file were stored on a single disk. When access pattern A is run on file structure B, the application receives less than 100 KBytes/sec, with a large variability in the time taken to service individual requests. The reason for the poor performance is that requests by the different processes arrive at disk in an interleaved fashion, resulting in many seek operations. As we will see in the next section, the importance of choosing the correct file structure becomes even more important when prefetching is used.

### 5.4.2 Prefetching policy

It is clear that in order to deliver to the applications the full bandwidth of the disks on our system, it is necessary to perform prefetching. That is, the file system must generate sufficient requests to keep the disks continuously busy. In HFS, this can be accomplished by using ASF objects that direct prefetching requests to the memory manager.

For access pattern A, since each process is accessing a different portion of the file, prefetching should be done on a per-process basis. Hence, a separate prefetching ASF object should be used for each of the processes. The simplest possible prefetching algorithm is to, on the first request to a block, generate a prefetch request for the next block. When employing this prefetching policy for access pattern A (on its matching file structure) we obtain 4526 KBytes/sec of disk bandwidth for the application, or a per-disk bandwidth of 646.6 KBytes/sec, which corresponds to 100% of the per-disk bandwidth available on our system. In this case, the speedup is perfect as the number of processors accessing the file is varied from one to seven (Figure 9).[12]

For access pattern B, since the entire file is being accessed sequentially, prefetching should be done on a per-application rather than a per-process basis. 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. It is indeed possible to exploit the full disk bandwidth of our system (i.e., 4526 KBytes/sec) using such a prefetching policy, however, we found that to do so it is necessary to be very aggressive: prefetching blocks 30 blocks in advance of the current one, and making prefetch requests for five blocks at a time. This aggressive policy is necessary in order to reduce the software overhead by: 1) amortizing the cost of PPC requests over multiple blocks; and 2) ensuring that disks never become idle so as to avoid having to wake up per-disk processes when initiating new requests.

The two prefetching policies differ in that: 1) one is per-process 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-process prefetching policy were used for access pattern B, then many useless prefetch requests would be issued. If a per-application prefetching policy were used for access pattern A, it would be ineffective. While it would be possible to use an aggressive prefetching policy for both access patterns, for access pattern A it is both unnecessary and could be counterproductive by requiring more memory for prefetched pages, possibly resulting in paging activity that would remove the benefits of prefetching.

[12] 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 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.

### 5.4.3 Summary of parallel I/O results

In this section, we have seen how crucial it is for the file structure and policy to match the application access pattern. For both access pattern A and B, we have shown that HFS is able to deliver to the application address space 100% of the disk bandwidth of our system. This is accomplished by employing HFS building blocks that match the application access pattern. From a file structure perspective, by using the appropriate building blocks, we are able to ensure that all disk requests are sequential and hence avoid any seek operations. From a policy perspective, by using the appropriate building blocks we are able to exploit the full concurrency of our disks while minimizing the memory and processing cost of prefetching.

Let us consider access pattern A in more detail. For this access pattern, the overall accesses to the file (i.e., the interleaved requests by the different processes) is non-sequential, even though each process is accessing the file sequentially.[13] Most parallel file systems employ only simple striping techniques for distributing file data across the disks of the system. As we have seen, for access pattern pattern A such a file structure results in non-sequential disk accesses and hence very poor performance. Even with prefetching, the large number of seek operations that result allows only a small fraction of the maximum disk bandwidth to be exploited. By allowing each part of the file to be distributed across a different set of disks, HFS enables the overall non-sequential access pattern to be turned into a sequential access pattern by each of the processes.

We chose to use access patterns that are (or can be made) sequential to evaluate the file system because disks perform best for such access patterns, and hence the stress on the file system software (as opposed to the disks) is maximized for such access patterns. It can be argued that file system flexibility is just as important for intrinsically non-sequential access patterns, since the large latency of I/O necessitates very aggressive prefetching policies. However, such access patterns make few demands on the file system software and hence provide less information about

[13] This illustrates a common problem, where the addition of concurrency to an application may change the application access pattern from sequential to non-sequential.

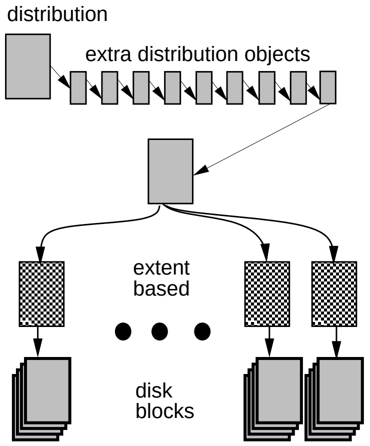


Figure 10: Structure of a file distributed across seven disks with 10 extra layers of building blocks.

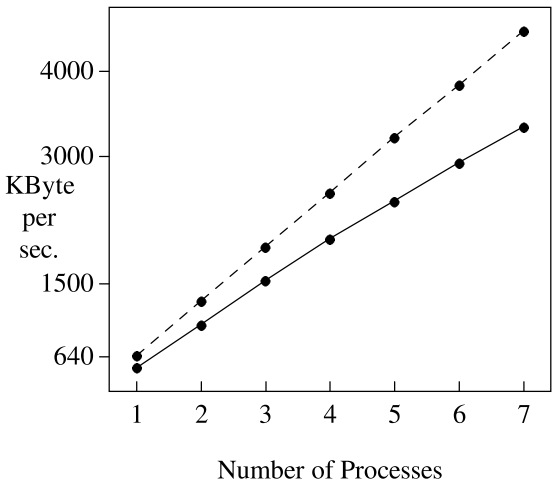


Figure 11: Throughput for multiple readers of a single distributed file, with 10 extra levels of building blocks. The solid line shows the performance with no prefetching. The dashed line shows the performance with prefetching of a single block.

the performance and overhead of a particular file system architecture.

### 5.5 Building-block overhead

One of the key issues in using building blocks is the amount of overhead they introduce. In a previous paper [17], we showed that the overhead of ASF building blocks is negligible relative to other overhead. To show that the overhead of the physical-layer building blocks is 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 object and the per-disk objects (Figure 10). Figure 11 shows the file system performance with and without prefetching when access pattern A is used on this file.

The file system delivers 516 KBytes/sec of data to a single application process with no prefetching. This is about 1.2% worse than when no extra building blocks are used. This degradation is due to the file system overhead to traverse the extra objects, and amounts to about 300 $\mu$secs for each disk block request, or about 30 $\mu$secs per building block. The speedup with seven processes 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. One possibility is increased memory contention, since the extra objects end up being distributed across the memory modules when cached by HFS in main memory, and hence there is an increased probability the file system will need to access memory to which a disk is actively transferring data.

With prefetching, the file system delivers the full disk bandwidth of our system to the application processes 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 hidden.

We can conclude that the impact of implementing a file using many layers of building blocks is small. The case considered is extreme, in that the number of layers of distribution building blocks used could support a file distributed across more than eight trillion disks, and in that each of the intermediate layers was implemented by a single object that is accessed for every I/O request (potentially resulting in lock and memory contention). While several layers might be required for other purposes, the 12 layers used in this example is unlikely. In any case, while some impact in performance is observable, we believe that this impact is largely due to characteristics of the Hector multiprocessor, including the inefficient memory system, the 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 spent copying data and because of the 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 [9] 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.

**effectiveness of processor caches:** We found that the file system cannot depend on its state remaining in the processor cache between successive requests. This increases the overhead of many operations. For example, when running real applications it causes PPC operations that cross address spaces to be twice as expensive as when measured using stress tests. A large part of our implementation effort was spent developing techniques to distribute and replicate file system state across multiple memory modules to avoid memory contention; this work would have been unnecessary if the state was typically available in the processor cache.

**memory-bandwidth considerations:** We originally believed that memory bandwidth would not be a consideration in our implementation on a small-scale system, but found that (i) the locality of the memory to the disk has an impact on the transfer rate from the disks, and (ii) a single disk transferring data to a memory module slows down the servicing of processor memory accesses. While disk to memory locality will always be important on large systems, more modern systems support block transfers that allow a processor to obtain the full disk transfer rate from any nearby disk. While the poor memory bandwidth of the Hector memory modules (and the backoff algorithm used in the case of contention) is the cause of disk transfers slowing down processor accesses to memory, it is possible that the same effect will occur in other systems. Disks are now available that can transfer data at 40 MBytes/sec from the on-disk cache, and this rate in constantly increasing.

**on-disk caches:** Parallel file system researchers often do not consider how data is stored on the disks, but only how the data is distributed across disks. We found that it is important to consider the tradeoff between having the data for a request spread across multiple disks to exploit concurrency, and having large amounts of data written to sequential disk blocks on a single disk to make effective use of the on-disk caches. Also, surprising results can occur if one ignores the on-disk caches. For example, when we first implemented HFS without cylinder clustering, we spent several weeks of performance debugging to discover that the system exhibited poor speedup due to the prefetching policy used by the on-disk caches. When multiple processes access a striped file, the prefetch requests by the on-disk caches interfere with the requests by the file system; when a single process accesses the striped file, the prefetch requests to the on-disk cache tend to complete before new requests by the file system arrive.

# 7 Related work

HFS is more flexible than all other existing and proposed parallel file system which we are aware of, including CFS [25] for the Intel iPSC, *sfs* [20] for the CM-5, PIOFS and Vesta [1] for the SP2, the OSF/1 file system [34], the nCUBE file system [4], the Bridge file system [6], and the RAMA file system [22]. 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 (although not all have been implemented) typically not available on other systems.

**dynamic distributions:** HFS is designed so that dynamic policies can be used to distribute requests across the disks, where the load on the disks and the location of the requesting process can be used to determine the target disk for a read or write request. All other existing parallel file systems distribute file data across the disks using 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:** The prefetching policy can be determined (by the application) on a per-open-file instance and per-thread basis. All other existing file systems either do not prefetch file data or have a single prefetching policy invoked automatically by the file system on a per-file or per-open-file basis.

**maintaining redundancy:** An application 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. All other existing parallel file systems either do not provide for any redundancy or have a single policy that is applied uniformly to the data and meta-data of all files.

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.

Recent parallel file I/O research has attempted to address the portability problem for I/O-intensive parallel applications by establishing standards for I/O interfaces [2] and by developing facilities that can easily be ported above any native file system to provide a common application interface [12, 24]. Also, there have been several projects to develop libraries and servers specifically targeted to meet the needs of scientific applications [28, 29, 32]. 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.

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 [11] implement a file using "layer" building blocks in the same way that HFS uses objects. 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 objects are specific to a single file, the relationship between the objects is determined on a per-file (or per-open-file, or per-thread) basis, and most interactions between objects are between objects 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, with even 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 to maximize 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 none of the techniques proposed by others to optimize performance present any special problems in our environment, and we are studying adopting, for example, Kotz's disk-directed I/O [15] in our system.

We believe that compiler support could be very useful to simplify the task of application programmers. In many cases, a compiler should be able to analyze the I/O bandwidth and latency requirements of an application, and automatically generate building-block compositions that will satisfy these requirements. Moreover, an intelligent compiler should be able to insert prefetch and poststore operations into an application to aid the application in tolerating the large latencies of disks. As a first step, we recently ported to our system a compiler that automatically generates prefetch instructions for cache lines [23] and modified it to generate prefetch requests to HFS files. Initial performance results appear very promising.

We are currently developing a new operating system for a more modern shared memory multiprocessor. 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 system, including the memory manager. This will address one of the main limitations in HFS, the lack of flexibility in the memory manager, a crucial component for I/O performance. Second, HFS will be ported to the new multiprocessor, allowing experiments to be performed on a more modern system with a large number of disks (128). Finally, HFS will be available from a very early stage on the new system, encouraging application developers to take advantage of its unique properties, and providing us with a much better insight into the advantages and limitations in using building-block compositions for file I/O.

We have found it challenging to adapt building-block compositions to other system software components. For example, in HFS we were able to design the system such that all interactions between building blocks where unidirectional. In contrast, for the memory manager it is necessary for building blocks to interact in a bi-directional fashion. This has resulted in an increase in the complexity and overhead of our locking protocols. As another example, all HFS compositions in system servers are hierarchical, and we were able to exploit this in developing algorithms to ensure that a composition created by the application is valid. In other portions of the system we have found that the structure of useful compositions are more complex, and we still have to develop algorithms that ensure that a composition is valid.

### 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

[1] P. F. Corbett, D. G. Feitelson, S. J. Baylor, and J. Prost. Parallel access to files in the Vesta file system. In *Proc. of Supercomputing*. IEEE Computer Society Press, November 1993.

[2] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[3] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proc. of Supercomputing '95*, December 1995.

[4] Erik P. DeBenedictis and Juan Miguel del Rosario. Modular scalable I/O. *Journal of Parallel and Distributed Computing*, 17(1–2):122–128, Jan/Feb 1993.

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

[6] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proc. of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[7] P. Druschel. Efficient support for incremental customization of OS services. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 186–190, Asheville, NC, December 1993.

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

[9] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proc. 1994 ICPP*, pages 208–211, Boca Raton, FL, August 1994. CRC Press.

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

[11] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[12] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proc. of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[13] Concurrent I/O application examples. Intel Corporation Background Information, 1989.

[14] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14, 1993.

[15] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994.

[16] Orran Krieger. *HFS: A flexible file system for shared memory multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, 1994.

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

[18] Jochen Liedtke. Improving IPC by kernel design. In *Proc. of the Fourteenth ACM Symposium on Operating System Principles*, pages 175–188, North Carolina, December 1993.

[19] Zheng Lin and Songnian Zhou. Parallelizing I/O intensive applications on a workstation cluster: a case study. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, 1993.

[20] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proc. of the 1993 Summer Usenix Conference*, pages 291–305, 1993.

[21] E.L. Miller and R.H. Katz. Input/output behavior of supercomputing applications. In *Proc. Supercomputing*, pages 567–76. IEEE Comput. Soc. Press, November 1991.

[22] Ethan L. Miller and Randy H. Katz. RAMA: A file system for massively-parallel computers. In *Proc. of the Twelfth IEEE Symposium on Mass Storage Systems*, pages 163–168, 1993.

[23] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 62–73, October 1992.

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

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

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

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

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

[29] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION runtime library for parallel I/O. In *Proc. of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[30] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Operating System Design and Implementation*, pages 139–152, November 1994.

[31] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[32] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-efficient scientific computation using TPIE. In *Proc. of the 1995 IEEE Symposium on Parallel and Distributed Processing*, October 1995.

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

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