Extensible File Systems in Spring

Yousef A. Khalidi and Michael N. Nelson

Sun Microsystems, Inc.
Mountain View, CA 94043 USA

Abstract

In this paper we describe an architecture for extensible file systems. The architecture enables the extension of file system functionality by composing (or stacking) new file systems on top of existing file systems. A file system that is stacked on top of an existing file system can access the existing file system's files via a well-defined naming interface and can share the same underlying file data in a coherent manner. We describe extending file systems in the context of the Spring operating system. Composing file systems in Spring is facilitated by basic Spring features such as its virtual memory architecture, its strongly-typed well-defined interfaces, its location-independent object invocation mechanism, and its flexible naming architecture. File systems in Spring can reside in the kernel, in user-mode, or on remote machines, and composing them can be done in a very flexible manner.

1 Introduction

File systems are an important part of operating systems. Typically, an operating system provides one or two types of file systems that are not extensible. In current systems such as the UNIX® operating system, the file system provides a storage mechanism (in addition to a naming facility) that manages stable storage media and cooperates with the virtual memory system to cache data in memory [1]. To add new file system functionality requires either modifying the existing file systems or adding a new file system. Examples of new functionality that may need to be added include compression, replication, encryption, distribution, and extended file attributes.

SIGOPS '93/12/93/N.C., USA

There are several architectures that allow for extending the functionality of the file system in one way or another. The Virtual File System (VFS) architecture was originally designed to accommodate multiple file systems within a UNIX kernel [2]. More recently, a proposal was made to evolve VFS to support the implementation of new file systems in terms of pre-existing ones [3]. The 4.4BSD UNIX system includes a stackable file system module that is based on the Ficus work described in [4]. Systems based on an external pager interface [5, 6] potentially have the ability to provide for file system extensibility. Other systems such as the Apollo extensible IO system [7] and *watchdogs* [8] allow for extending the file system in a limited manner. Such systems demonstrate along with other systems the need for extending the file system. However, we believe that the goals of these systems are too limited.

In this paper we describe an architecture for extensible file systems. We also describe an implementation of the architecture and report on our experience using it in extending the base Spring system. The architecture enables the extension of file system functionality by stacking (or composing) new file systems on top of existing file systems. The implementor of a new layer has the option of keeping the files exported by the new layer coherent with files of the underlying layer, as well as the option of sharing the same cached memory with the files of the underlying layer. A flexible framework is also provided for arranging the file systems' name spaces. Composing new layers on top of existing ones can be done statically (at compile/configuration time) or dynamically (at boot/run time). In addition, the file system layers can reside in the same address space or in different address spaces, and be implemented locally or remotely.

This paper is organized as follows. Section 2 presents the goals and requirements of the architecture. Section 3 gives an overview of the Spring system, emphasizing those aspects of Spring that are relevant to this paper. The general stacking architecture is described in Section 4, while the special case of interposing on a per-file basis is presented in Section 5. The implementation of the system and our experience in using it are described in Section 6. Section 7 compares our architecture to other related work. Conclusions

and future work are discussed in Section 8. Finally, the Appendices list some of the important interfaces used in the paper.

2 Requirements

There are four broad requirements that we believe are necessary for a flexible extensible file system architecture:

1. **Leveraging existing file systems.** In order to add new file system functionality to a system, one has two basic options: either to build new file systems from scratch, or somehow to leverage the existing file systems. Clearly, it is preferable to be able to easily leverage existing file systems when introducing new file system functionality. This should be achievable without affecting the clients of the existing file systems.
2. **Caching.** For performance reasons, the extensible file system architecture must define means for caching file data and attributes.
3. **Coherency.** Due to caching, to distribution, and to multiple clients accessing the same data from different points of view, the architecture must define a framework for keeping file data and attributes coherent. However, the coherency policies should be left to the implementation of the file systems.
4. **Dynamic addition of functionality.** For ease of administering and configuring the system, it should be possible to add new functionality to a running system, and to dynamically extend the functionality of files. In addition, new file systems should be able to reside in the kernel or in user mode.

These requirements lead us to the following design decisions:

- The implementation of a new file system leverages existing file systems by being stacked on top of existing file systems. The files of the underlying file system can be directly accessible if desired. Whether a file is accessible or not is an administrative decision.
- The architecture provides the building blocks for caching file data and attributes and for keeping them coherent with the files of the underlying file systems.
- A particular file system decides whether or not to keep its files coherent with the files of the underlying file systems, and a file system is free to implement the coherency algorithm it desires.
- A file system that does not modify the data of its underlying file system may use the same memory used to cache the underlying file(s) to avoid caching the "same" file data twice.
- File systems can be implemented in the kernel or in user processes, and in either case can be implemented locally or on a remote machine. In addition, file systems can be added to or removed from the system statically or dynamically.
- The architecture allows for interposing on individual files or entire collections of files.

3 Spring

In this section we give a brief overview of the Spring system and its naming architecture, followed by a more detailed description of the virtual memory system, an aspect of Spring that is particularly relevant to this paper.

3.1 The Spring Operating System

Spring is a distributed, multi-threaded operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. The description of an object and its operations is an *interface* that is specified in an *interface definition language*. The interface is a strongly-typed contract between the *server* (implementor) and the *client* of the object.

A Spring domain is an *address space* with a collection of *threads*. A given domain may act as the server of some objects and the client of other objects. The server and the client can be in the same domain or in a different domain. In the latter case, the representation of the object includes an unforgeable nucleus *handle* that identifies the server domain.

Since Spring is object-oriented, it supports the notion of *interface* inheritance[1]. An interface that accepts an object of type *foo* will also accept a subclass of foo.

The Spring kernel supports basic cross domain invocations, supports threads, and provides basic virtual memory support for memory mapping and physical memory management [9, 10].

A typical Spring node runs several servers in addition to the kernel as shown in Figure 1. Support for running UNIX binaries is also provided [11]. All services are exported via objects defined using the interface definition language. In general, any collection of servers may reside in the same or in different domains. The decision on where to run a particular server is made for administrative, security, and perfor-

1. Note the emphasis on interface inheritance as opposed to implementation inheritance. From the operating system point of view, what matters are the interfaces of the different servers and programs. Specific implementations are free to use implementation inheritance and to reuse code.

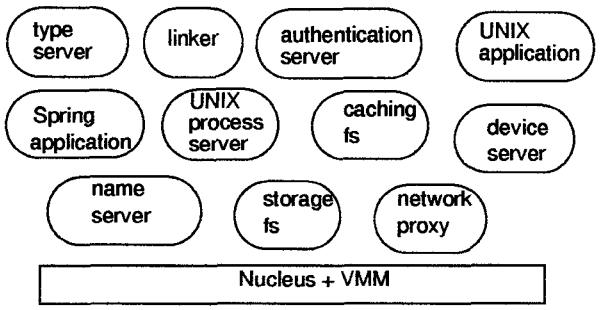

FIGURE 1. Major system components of a Spring node

mance reasons, and is independent of the interface of the service.

3.2 Naming

The Spring naming service allows any object to be associated with any name. A name-to-object association is called a *name binding*. A *context* is an object that contains a set of name bindings in which each name is unique. An example of a context is a UNIX file directory. An object can be bound to several different names in possibly several different contexts at the same time. Since a context is like any other object, it can also be bound to a name in some context. Our naming system is based on the architecture described in [12].

There are two aspects of Spring naming that are very useful for our extensible file system architecture:

- Any domain may implement a naming context and, if the domain is appropriately authenticated, can bind the context in any other context.
- Each Spring domain has a context object that implements a per-domain name space. All domains have part of their name space in common, but they can also customize their name space as appropriate.

3.3 Virtual Memory

3.3.1 Overview

A per-node virtual memory manager (VMM) is responsible for handling mapping, sharing, and caching of local memory. The VMM depends on external pagers for accessing backing store and maintaining inter-machine coherency.

Most clients of the virtual memory system only deal with *address space* and *memory* objects. An address space object represents the virtual address space of a Spring domain while a memory object is an abstraction of store (memory) that can be mapped into address spaces. An example of a memory object is a file object (the file interface in Spring inherits from the memory object interface). Address space objects are implemented by the VMM.

	MACH	Spring
Memory Object	Memory mapped & encapsulates access rights	Memory mapped & encapsulates access rights
	Init/terminate ops	Bind operation
	Paging operations	*No* paging operations
File Object	Can be same port as memory object	Inherits from memory object
	May provide file operations using paging operations	Provides file read/write operation
		No paging ops

TABLE 1. Memory object in MACH and Spring

A memory object has operations to set and query the length, and an operation to *bind* to the object (see below). There are no page-in/out or read/write operations on memory objects (which is in contrast to systems such as Mach [5]). The Spring file interface provides file read/write operations (but not page-in/page-out operations). Separating the memory abstraction from the interface that provides the paging operations is a feature of the Spring virtual memory system that we found very useful in implementing our file system [13]. This separation enables the implementor of the memory object to be in a different location from the implementor of the *pager* object which provides the contents of the memory object. Table 1 shows the differences between our memory object and the memory object of a traditional external pager-based system such as MACH. We will show uses of this feature in Sections 4 and 6.2.

3.3.2 Cache and Pager Objects

In order to allow data to be coherently cached by more than one VMM, there needs to be a two-way connection between the VMM and the provider of the data (e.g., a file server). The VMM needs a connection to the data provider to allow the VMM to obtain and write out data, and the data provider needs a connection to the VMM to allow the provider to perform coherency actions (e.g., invalidate data cached by the VMM). In our system we represent this two-way connection as two objects. The VMM obtains data by invoking on a *pager* object implemented by a data provider, and a data provider performs coherency actions by invoking on a *cache* object implemented by a VMM. Throughout the rest of this paper we call data providers "pagers".

When a VMM is asked to map a memory object into an address space, the VMM must be able to obtain a pager object to allow it to manipulate the objects's data. Associated with this pager object must be a cache object that the pager can use for coherency. In addition, a VMM wants to

make sure that if two equivalent memory objects (i.e. two memory objects that refer to the same file on disk) are mapped that they can share the same data cached by the VMM. Therefore the VMM contacts the pager that implements the memory object by invoking the *bind* operation on the memory object. The result of the bind operation is a *cache_rights* object implemented by the VMM. If two equivalent memory objects are mapped, then the same cache_rights object will be returned. The VMM uses the returned object to find a pager-cache object connection to use, and to find any pages cached for the memory object.

When a pager receives a bind operation from a VMM, it must determine if there is already a pager-cache object connection for the memory object at the given VMM. If there is no connection, the pager contacts the VMM, and the VMM and the pager exchange pager, cache, and cache_rights objects. Once the connection is set up the pager returns the appropriate cache_rights object to the VMM.

Appendices A and B list the operations of the cache and pager objects, respectively. Typically, there are many pager-cache object channels between a given pager and a VMM (see Figure 2 for an example).

3.3.3 Maintaining Data Coherency

The task of maintaining data coherency between different VMMs that are caching a memory object is the responsibility of the pager for the memory object. The coherency protocol is not specified by the architecture—pagers are free to implement whatever coherency protocol they wish. The cache and pager object interfaces provide basic building blocks for constructing the coherency protocol. Our current pager implementations use a single-writer/multiple-reader per-block coherency protocol (Section 6.2).

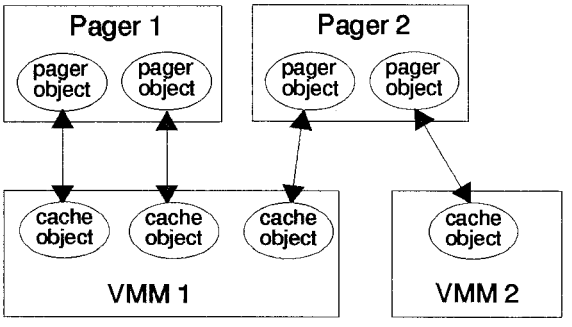

FIGURE 2. Pager-cache object example

A VMM and a pager have a two-way pager-cache object connection. In this example, Pager 1 is the pager for two distinct memory objects cached by VMM 1, so there are two pager-cache object connections, one for each memory object. Pager 2 is the pager for a single memory object cached at both VMM 1 and VMM 2, so there is a pager-cache object connection between Pager 2 and each of the VMMs.

4 Spring File Stacking Architecture

4.1 Overview

The Spring file system stacking architecture enables new file systems to be added that extend the functionality of and build on existing file system implementations. This is achieved by adding new file system *layers* to the system. It is important to note that as long as the *interface* of the new layer conforms to the interface of a file system, clients will view the new layer as a file system, regardless of how it is implemented. The *implementation* decides how the new layer utilizes the underlying file systems, which in turn also must conform to the file system interface. *Administrative* decisions are used to choose which file systems to stack on top of other file systems (or individual files), and to arrange the name space appropriately.

There need not be a one-to-one correspondence between the files exported by a given layer and its underlying layers. A file system may even export files that do not actually exist. Again, the implementation of a given file system makes such decisions.

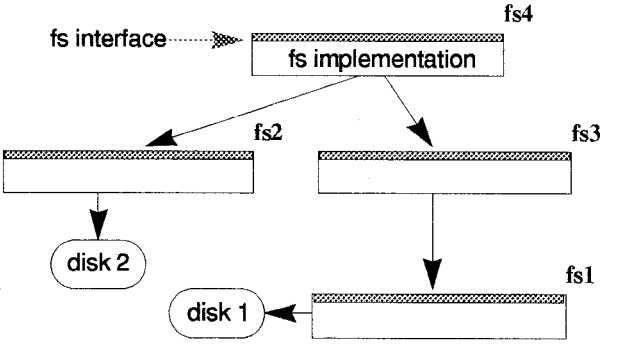

FIGURE 3. Implementation vs. administrative decisions

Figure 3 illustrates a configuration of stacked file systems. In the figure, each box represents a layer[2] that exports a file system interface. At the bottom of the stack are *base* file systems fs1 and fs2 that build directly on top of storage devices (e.g., UFS [14]). The implementation of fs3 uses one underlying file system, while the implementation of fs4 uses two underlying file systems to implement its function (e.g. fs3 is a compression file system, and fs4 is a mirroring

2. As with other Spring servers, the layers of a given configuration of stacked file systems may be implemented by one or more domains, on one or more machines.

file system). The choice of which file systems to use as the underlying file systems for fs3 and fs4 is an administrative decision. It is also an administrative decision whether (and to whom) to expose the files exported by the various file systems. Note that the decision of which disk drives to use for the base file systems fs1 and fs2 is similar to the current practice of mounting disk partitions.

There are three components to the extensible file system architecture:

1. A stackable pager interface for caching data and keeping it coherent.
2. A stackable attribute interface for caching file attributes and keeping them coherent.
3. A stackable file system interface that is used with a flexible naming architecture to compose file systems and to arrange the file name space.

We will discuss each of these components in turn.

4.2 Stackable Pager Interface

In Section 3.3 we described the cache and pager object interfaces and how they can be used by external pagers to keep memory shared by more than one VMM coherent. In general, anybody can implement cache objects. A VMM is one such cache manager (i.e. it is an implementor of cache objects); pagers can also act as cache managers to other pagers. Therefore, a pager may have its data cached at several cache managers, some of which may be virtual memory managers. As far as the pager is concerned, it uses the same algorithm to maintain the coherency of its data regardless of whether a particular cache manager is a VMM or not.

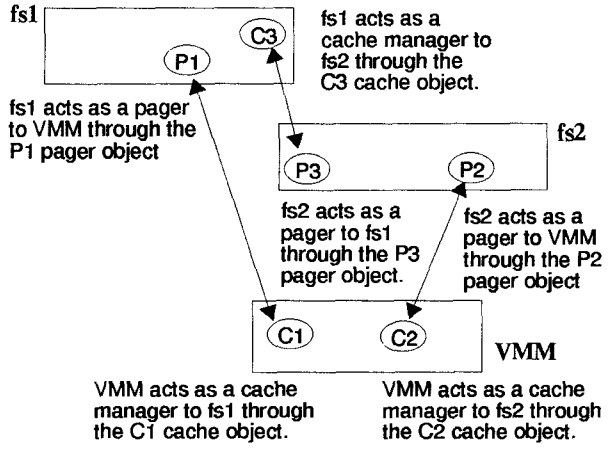

FIGURE 4. File systems as pagers & cache managers

Figure 4 shows how a file server may act as a pager and a cache manager at the same time. In particular, in this figure, fs1 acts as a pager to the VMM through the P1 pager object,

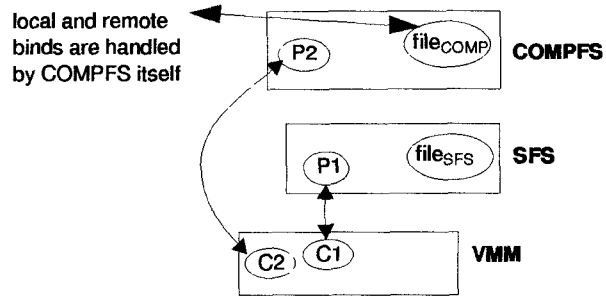

FIGURE 5. Stacking COMPFS on top of SFS (case 1)

- Local and remote binds to COMPFS are handled by COMPFS itself. Binds from the local VMM result in using the P2-C2 connection.
- COMPFS accesses the contents of file$_{SFS}$ by either mapping it or issuing read/write calls to it.
- Clients can access the underlying SFS file (file$_{SFS}$) as usual.
- Mappings of file$_{SFS}$ and file$_{COMP}$ are *not* coherent with respect to each other.

and fs1 acts as a cache manager to fs2 through the C3 object. Note that the pager and cache object interfaces are the same as those described in Section 3.3.

There are two possible design decisions that the implementor of a file system layer has regarding data caching:

- Whether to keep the layer's files coherent with the files of the underlying file system. A file system can maintain coherency with the underlying files by acting as a cache manager for those files.
- Whether to use the same cached pages for the layer's files and for the files of the underlying file system. A file system can use the same cached memory by forwarding bind operations from local cache managers to the underlying file system which has the effect of using the same pager-cache object connection as the one used for the underlying file.

We illustrate these two design points with examples.

4.2.1 Stacking COMPFS on top of SFS

Suppose we would like to implement a compression file system (COMPFS). We can use COMPFS to save disk space by compressing all data before writing it out and by uncompressing all data read from the disk. Since we are not interested in rewriting an on-disk file system, we can implement COMPFS as a layer on top of a base file system (SFS).

Figure 5 illustrates the example of stacking COMPFS on SFS. A request to COMPFS to create a new file$_{COMP}$ results in COMPFS creating a new underlying file$_{SFS}$.

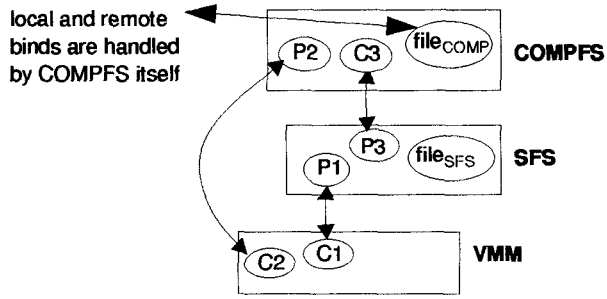

FIGURE 6. Stacking COMPFS on top of SFS (case 2)

Same as Figure 5 except:

- COMPFS acts as a cache manager to SFS by establishing a P3-C3 connection. Remote page-in/page-out and read/write requests on $file_{COMP}$ result in requests made by COMPFS to SFS through the P3-C3 connection.
- Mappings of $file_{SFS}$ and $file_{COMP}$ *are* coherent with respect to each other.

As with other files in Spring, a client that holds $file_{COMP}$ may access its contents either by memory mapping the file or by calling the file read/write operations. COMPFS implements the read/write operations the same way as other Spring file systems: it maps the $file_{COMP}$ into its address space and reads/writes the mapped memory. Therefore, in either case, the cache object used by the VMM to service the file map request is C2, and COMPFS receives page-ins/page-outs from the VMM on its P2 object.

When a page-in is received on P2 from the VMM, COMPFS has to read the compressed version of the data stored in $file_{SFS}$, uncompress the data, and return it as a result of the page-in request. Similarly for page-outs, COMPFS has to compress the data and write it to $file_{SFS}$.

Note if the name space is configured appropriately (see Section 4.4), the underlying SFS files can also be accessed by other clients in the system. A client opening $file_{SFS}$ can access this file as usual, reading and writing its *compressed* data.

COMPFS can access $file_{SFS}$ by reading/writing the file or by memory mapping it. Such a scheme works and is simple, but has one drawback: if $file_{COMP}$ and $file_{SFS}$ are both memory mapped, the setup shown in Figure 5 will *not* keep accesses to both files coherent. For example, if a client writes directly into $file_{COMP}$ the corresponding changes may not be reflected into $file_{SFS}$ until some time later, or they may be clobbered by direct writes to $file_{SFS}$.

Keeping concurrent accesses to $file_{COMP}$ and $file_{SFS}$ coherent may or may not be important. In the case of COMPFS, one may argue that it is not important, and a quick solution is to disallow direct accesses to $file_{SFS}$. For the sake of discussion, let's assume that it is important to keep all mappings to $file_{COMP}$ and $file_{SFS}$ coherent.

When a file system is stacked on top of another file system, it can act as a cache manager to the underlying pager. A new setup is shown in Figure 6. Note that the only difference from Figure 5 is the addition of the C3-P3 connection. To keep all mappings to $file_{COMP}$ and $file_{SFS}$ coherent, COMPFS now acts as a cache manager for $file_{SFS}$. Therefore, instead of accessing $file_{SFS}$ through the file interface, COMPFS establishes itself as a cache manager for $file_{SFS}$ by issuing a bind operation on $file_{SFS}$ and establishing a C3-P3 connection.

The advantage of the second approach is that COMPFS can now keep $file_{COMP}$ coherent with $file_{SFS}$. When COMPFS is a cache manager for $file_{SFS}$, SFS (acting as a pager) engages COMPFS in its coherency actions regarding $file_{SFS}$. For example, if the VMM requests a block of $file_{SFS}$ in read-write mode through the P1-C1 connection, the SFS acting as a pager may first need to flush the block from COMPFS through the C3-P3 connection before to the VMM's request.

Each pager is responsible for keeping its own files coherent. The exact algorithm implemented by a given file system is not dictated by the architecture. This implies that a given layer can only guarantee that its files will be coherent with the files' underlying state. An underlying file system may choose not to keep its state coherent with *its* underlying file system. In Section 6.3 we describe how, utilizing a generic coherency layer, one can construct a stack of file systems where all file accesses are coherent, even though each individual layer implementation does not maintain coherency.

4.2.2 Stacking DFS on top of SFS

Another design decision that a file system implementor may want to make is whether to use the *same* pager-cache object connection used by the underlying file system for local accesses to the file. This is possible if the layer does not change the data obtained from the underlying file system.

Figure 7 illustrates a network-coherent distributed file system (DFS) layered on top of SFS. The job of DFS is to export SFS files to other machines in a coherent fashion through some existing protocol (e.g., AFS [15]). For each underlying $file_{SFS}$, DFS exports a $file_{DFS}$. $File_{DFS}$ may be accessed on the local machine through the normal Spring mechanisms, or it may be accessed remotely through the DFS protocol.

DFS arranges to act as a cache manager for $file_{SFS}$ to handle remote DFS operations. Therefore, all accesses to $file_{DFS}$ will be coherent with $file_{SFS}$. But since the data of $file_{DFS}$ is

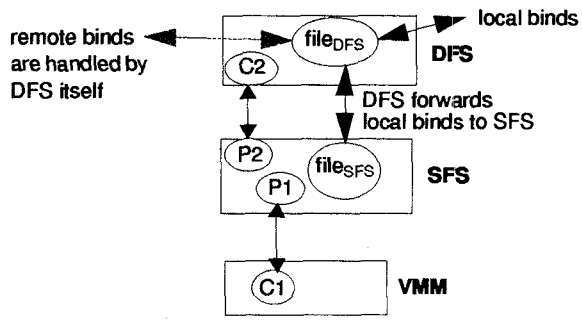

FIGURE 7. Stacking DFS on top of SFS

- Local binds to file$_{DFS}$ are forwarded to the corresponding file$_{SFS}$. Thus, local clients of file$_{DFS}$ use the same cache object (C1) as clients of file$_{SFS}$, and DFS is not involved in local page-in/page-out requests for file$_{DFS}$.
- Remote binds to DFS are handled by DFS itself. DFS acts as a cache manager to SFS by establishing a pager object-cache object (P2-C2) pair. Remote page-in/page-out requests to DFS result in requests made by DFS to SFS through P2-C2 connection.
- DFS handles read/write requests on file$_{DFS}$ by mapping file$_{SFS}$ in its address space and reading/writing the data directly in memory.

identical to that of file$_{SFS}$, DFS arranges to use locally cached memory of file$_{SFS}$ to handle *local* accesses to file$_{DFS}$. This is achieved by forwarding bind operations from local cache managers on file$_{DFS}$ to the bind operation on file$_{SFS}$. The pager object returned from the bind operation on file$_{SFS}$ is P1. (In other words, when the VMM binds to a locally managed DFS file, DFS reroutes the VMM to the SFS, so that the VMM ends up dealing with SFS directly.) Therefore, local accesses to file$_{DFS}$ use the same cached memory as file$_{SFS}$.

Note that since DFS is acting as a cache manager for file$_{SFS}$, changes to the data in C1 (through file$_{SFS}$ and local mappings of file$_{DFS}$) that affect pages cached by remote DFS clients will be communicated to DFS by SFS. Likewise, any coherency actions taken by DFS through its private network protocol will be communicated to SFS through the P2-C2 channel.

There are cases, of course, when sharing the same underlying data cache is not feasible. In our first example (Figure 5), COMPFS *computes* its data based on SFS files. In this case, file$_{COMP}$ data is different from file$_{SFS}$ data, and no sharing is possible. Note that these decisions are made by the implementation of COMPFS—not SFS or any other file system in the stacking hierarchy.

4.3 Stackable File Attributes Interface

The previous section described how we can use the cache and pager object interfaces as the building blocks for accessing and caching data, and keeping it coherent. The cache and pager object interfaces, however, are insufficient for stacking file systems. In addition to data, files contain attributes such as access and modified times, and file length. Other attributes that may be associated with files include access control lists and generalized attribute lists.

One approach to handling file attributes would be to add more operations to the cache and pager object interfaces. Such an approach suffers from two problems. First, it is not possible to decide on all operations that may be needed by possible future file system extensions. Second, adding file system-specific operations to a data movement interface complicates the implementation of non-file system clients of that interface.

Instead of burdening the cache and pager object interfaces with file-specific operations, we *subclass* the cache and pager object interfaces into fs_cache and fs_pager interfaces, respectively. These two interfaces add some file attribute operations that we believe are a good starting point for handling file systems (basically, operations for caching and keeping coherent the access and modified times and file length). Future systems are free to subclass these interfaces further to add more operations.

Since fs_cache and fs_pager objects are subclasses of cache and pager objects, they can be passed wherever cache and pager objects are expected. Referring back to Figure 7, the P2-C2 connection is established when DFS, acting as a cache manager, invokes the bind operation on file$_{SFS}$. Recall that as a result of the bind operation a pager-cache channel is established (or reused). When the channel is first established, DFS sends an fs_cache object instead of a cache object, and similarly SFS sends an fs_pager object instead of a pager object. DFS attempts to narrow the pager object it receives to an fs_pager object. If it succeeds, it knows that it is talking to a file system. Otherwise, DFS assumes that it is talking to a simple storage pager that only provides the pager object functionality. Similarly, SFS attempts to narrow the cache object it receives to an fs_cache object. If the narrow succeeds, SFS knows that it is talking to a file system, and therefore engages it in the file attributes coherency protocol. Otherwise, SFS assumes that it is talking to a simple cache manager (e.g., a VMM).

Note that the fs_cache and fs_pager interfaces can be subclassed further to add more file system functionality. A particular file system implementation may attempt to narrow these objects to other subtypes.

4.4 Configuring file systems

In this section, we discuss two related issues: how to configure file system stacks, and how to configure the resulting file name space.

A mechanism is needed to construct file systems layers. In all the examples presented so far, a layer (e.g., DFS in Figure 7) is really an *instance* of a DFS file system, since there can be other DFS layers stacked on other file systems. We therefore define an interface, *stackable_fs_creator*, that is used to create instances of stackable file systems. This interface provides one operation, *create*, which returns instances of file systems of type *stackable_fs*. The stackable_fs interface inherits from the *fs* and *naming_context* interfaces as shown in Figure 8.

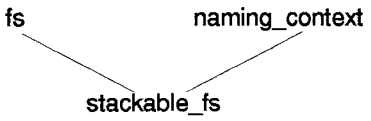

FIGURE 8. File system interface hierarchy

At boot-time or during run-time, the file system creator for each file system type (e.g., DFS and COMPFS) is created. When a file system creator is started, it registers itself in a well-known place e.g., /fs_creators/dfs_creator.

The method to configure a new file system is as follows:

1. A file system creator object is looked up from the well-known place using a normal naming resolve operation (e.g., "dfs_creator" is looked up in /fs_creators, returning the object dfs_creator_obj, which is of type *stackable_fs_creator*).
2. The file system creator object returned by Step 1 is used to create an instance of the file system, e.g.,
   stackable_fs dfs = dfs_creator_obj ->create();
3. The dfs instance is given an object of type stackable_fs as the underlying file system, e.g.,
   dfs->stackon(fs_2);
   Note that since fs_2 is of type stackable_fs it is also a naming_context. The stack_on operation can be called more than once to stack on more than one underlying file system—the maximum number of file systems a particular layer may be stacked on is implementation dependent.
4. The new file system instance is bound somewhere in the name space to expose its files to user programs, e.g.,
   some_name_server->bind(cxt, dfs);
   Note that dfs is also a naming_context.

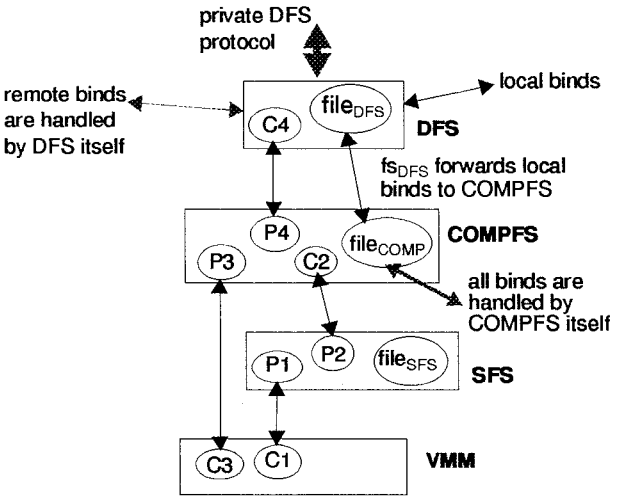

FIGURE 9. Stacking DFS on COMPFS on SFS

4.5 Putting everything together

In this section, we present a walk-through of the extended example shown in Figure 9. In this setup, DFS is stacked on COMPFS, which is in turn stacked on SFS. The stack is created using the following steps:

- A dfs_creator_obj and a compfs_creator_obj are looked up from a well-known location (e.g., /fs_creators).
- An SFS object (which is of type stackable_fs) is located (e.g., looked up from /fs/SFS0a).
- An instance of COMPFS is created:
  stackable_fs compfs = compfs_creator_obj->create()
- COMPFS is stacked on top of SFS:
  compfs->stackon(sfs)
- Similarly for DFS:
  stackable_fs dfs = dfs_creator_obj->create()
  dfs->stackon(compfs)
- A decision is made whether or not to export SFS, COMPFS, and DFS files (and to whom). A file system is exported by binding its stackable_fs object in a context somewhere. Let's assume in this example that all file systems are exported.

Now suppose a name lookup arrives through the private DFS protocol:

- DFS resolves the file in its underlying file system.
- COMPFS in turn resolves the file in SFS.
- SFS returns $file_{SFS}$ to COMPFS.
- COMPFS invokes the bind operation on $file_{SFS}$ to setup (or reuse) the P2-C2 connection and returns a locally implemented $file_{COMP}$ to DFS.

- DFS invokes the bind operation on $file_{COMP}$ to setup (or reuse) the C4 - P4 connection.

A remote read request arriving through the private DFS protocol results in:

- DFS issuing a read-only page-in on P4.
- COMPFS issuing one or more read-only page-ins on P2.
- SFS reading the data from disk.
- COMPFS uncompressing the data and returning it to DFS.
- DFS sending the data to its DFS client through the private DFS protocol.

Note that at any point the underlying data of $file_{DFS}$ may be accessed through $file_{COMP}$ or (uncompressed) through $file_{SFS}$. All such accesses will be coherent with each other *and* with remote DFS clients.

5 Interposing on a Per-file Basis

In Section 4, we described how a new file system can be stacked on top of another. In this section, we describe how we can use the same stacking architecture to change the semantics of individual files or even individual file operations (functionality similar to *watchdogs* described in [8]).

Spring provides a general mechanism for object *interposition*. An object O_1 can be substituted for another object O_2 of type *foo* as long as O_1 is also of type foo. The implementation of O_1 decides on a per-operation basis whether to invoke the corresponding operation on O_2, or whether to implement the functionality itself. We describe in Section 6.2 a caching file system that interposes on remote files.

Another way to interpose on files is at name-resolution time. To interpose on one or more files, an interposer resolves the name of the context where the file object(s) is bound, unbinds the context from the name space, and binds in its place a naming context implemented by the interposer itself. As a result, all naming operations through this context will go to the interposer.[3] The interposer can then selectively intercept some name resolutions while passing the rest to the original context.

When the interposer intercepts a name resolution, it returns a file implemented by itself. All calls on the new file are handled by the interposer, which may implement the operation itself, or may forward the call to the original file object. A more sophisticated interposer may act as a cache manager to the original file.

3. Of course, the interposer has to be appropriately authenticated to be able to manipulate the name space. Note that naming contexts are associated with access control lists (ACLs) as explained in [12].

6 Implementation

6.1 Spring Operating System

The Spring operating system is implemented, and is now stable. We are currently on a push to make it our daily production system. All system servers including the kernel are multi-threaded, and the system runs on several uniprocessor and symmetric multiprocessor Sun SPARCstation models. Virtually all of the system is implemented in C++.

The system interfaces are stable. Some of the interfaces, especially the cache and pager object interfaces, underwent several revisions as we gained more experience with using and extending the system during the last three years.

6.2 File System Implementations

We have implemented and experimented with the extensible file system architecture described in this paper. Our "production" system has several layered file system implementations:

Spring DFS. The Spring distributed file system [13] is implemented as a *coherency* layer. The coherency layer implements a per-block multiple-readers/single-writer coherency protocol. Among other things, the implementation keeps track of the state of each file block (read-only *vs.* read-write) and of each cache object that holds the block at any point in time. Coherency actions are triggered depending on the state and the current request using a single-writer/multiple-reader per-block coherency algorithm. The coherency layer also caches file attributes using the operations provided by the fs_cache and fs_pager interfaces.

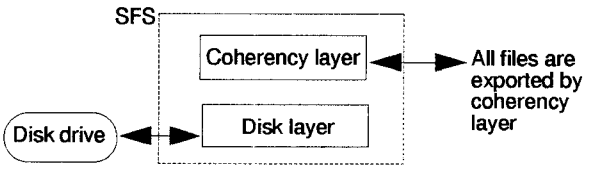

FIGURE 10. Spring SFS

Spring SFS. The Spring storage file system is actually implemented using two layers as shown in Figure 10. The base *disk* layer implements an on-disk UFS [14] compatible file system. It does not, however, implement a coherency algorithm. Instead, an instance of the coherency layer is

stacked on the disk layer, and all files are exported via the coherency layer.

We structure the SFS in this manner for two reasons:

1. By stacking the coherency layer on top of the disk layer, we leverage the existing coherency implementation.
2. By placing each layer in its own address space, we can lock the disk layer implementation in physical memory, while keeping the coherency layer in pageable memory. Note that the system needs a "default" pager that is locked-down in memory and is responsible for paging other pagers. The state maintained by the disk layer implementation is small (basically an *i*-node cache). However, the state maintained by the coherency layer is larger, and is proportional to the number of clients of the system.

It is interesting to note that when we initially implemented the Spring distributed file system, we planned to extract the coherency code into a regular C++ library that any pager implementation could use. We soon realized that by structuring the coherency code as a file system layer we were able to stack this layer on top of non-coherent layers with comparable performance to a library approach.

Spring CFS. CFS is an attribute-caching file system. Its main function is to interpose on remote files when they are passed to the local machine as described in [13]. Once interposed on, all calls to remote files end up being forwarded to the local CFS.

An interesting aspect of CFS is the manner in which it *dynamically* interposes on individual remote DFS files. When CFS is asked to interpose on a file, it becomes a cache manager for the remote file by invoking the bind operation on the file as described in Section 4.2.

When a remote file is mapped locally, the VMM invokes the bind operation on the file. Since the file is interposed on by CFS, CFS receives the bind request. CFS proceeds by returning to the VMM a pager-cache object channel to the remote DFS. Therefore, all page-ins and page-outs from the VMM go directly to the remote DFS.

CFS caches file attributes using the fs_pager and fs_cache objects described in Section 4.3. CFS also services read/write requests by mapping the file into its address space and reading/writing the data from/to its memory (thus utilizing the local VMM for caching the data).

Note that CFS is optional. If it is not running, remote files will not be interposed on, and all file operations go to the remote DFS.

6.3 Constructing coherent stacks

Using the coherency layer, we can construct coherent file system stacks out of non-coherent layers. As in the case of Spring SFS, starting from a non-coherent base layer, we stack a coherency layer on the non-coherent layer and export all files through the coherency layer. In the resulting file system stack, any exported file will be coherent with its underlying file(s).

6.4 Performance

In this section we look at the performance of a file system that was built using our extensible file system architecture. We focus our performance measurements on determining any additional overhead introduced by using our layering approach to extensibility.

The additional overhead from our layering approach is mainly due to crossing interface boundaries. Two file system layers communicate by invoking operations on file, stackable_fs, pager, and cache objects. When two layers are in different domains, each object invocation requires a cross-domain call. However, if the two layers are in the same domain, then the object invocation consists of only two local procedure calls. Whether one layer is in the same domain or is in a different domain from another layer is transparent to the implementation. Our object invocation stub technology automatically chooses the optimal path (procedure calls or cross-domain calls).

The file system that we measured is the SFS described in the previous section. This file system contains two layers. We provide measurements of stacking overhead by measuring three implementations of the SFS:

- One that does not use stacking—this is the case with no stacking overhead.
- One that uses stacking, but both file system layers are in the same domain.
- One that uses stacking where the two file system layers are in different domains.

The benchmarks we use are opening, reading, writing, and getting the attributes of a file stored on a machine's local disk. All measurements were taken on Spring running on a 40 Mhz SPARCstation 10 with 64 Mbytes of memory and a 424 Mbyte 4400 RPM disk. Each data point is the average of 5 runs of 10000 invocations of the given operation. Variance between runs was less than 8 percent.

Table 2 compares measurements of the three implementations of the SFS. These measurements show several interesting results. First, there is no significant overhead from stacking if the layers are in the same domain. The highest overhead is 39 percent on the open operation, and is due to

Operation	Cached by Coherency Layer?	Not stacked	Stacked one domain	Stacked two domains
open	No	0.98 100%	1.36 139%	1.97 201%
4KB read	Yes	0.17 100%	0.17 100%	0.17 100%
4KB read	No	13.7 100%	13.7 100%	13.7 100%
4KB write	Yes	0.16 100%	0.16 100%	0.16 100%
4KB write	No	13.7 100%	13.7 100%	13.7 100%
fstat	Yes	0.13 100%	0.11 85%	0.11 85%
fstat	No	0.13 100%	0.16 123%	0.22 169%

TABLE 2. Spring Performance Measurements

The table shows measurements of simple operations with and without caching by the coherency layer. The disk layer maintains its own cache to handle open and stat operations without requiring disk I/Os, but reads and writes to the disk layer do require disk I/Os. The first row is the cost of opening a file using a single-component path name. The second and third rows are the cost of reading 4 Kbytes from a file. The fourth and fifth rows are the cost of writing 4 Kbytes to a file. The sixth and seventh rows are the cost of getting an open file's attributes. The top line of each row is the time in milliseconds and the bottom line is normalized relative to the non-stacked implementation.

maintaining state about open files in two layers instead of one. There is no measurable overhead on read, write, and stat operations, since these merely require two extra procedure calls across the layer.

The second interesting result is that when the two layers are in different domains, there is a fairly significant overhead for the open operation (101 percent). This overhead is basically due to maintaining extra state in the top layer and to the cross-domain call overhead when the top layer calls the lower layer to perform the open. Although the open overhead appears to be significant, it will only be important to applications that are dominated by the cost of file opens. Based on the estimates of name lookup overhead on the macro-benchmarks in [16], we believe that the open overhead when two layers are in different domains will not be significant for real applications.

If the open overhead caused by splitting file system layers across domains turns out to be significant for some applications, name caching can be used to eliminate the overhead. We are currently implementing name caching in Spring in order to eliminate the network overhead of remote name resolutions. However, this same implementation can be used, if necessary, to eliminate the domain crossing overhead as well.

The third interesting result from Table 2 is that when the coherency layer caches the results of read, write, and stat calls, there is no overhead from stacking since there are no calls from the coherency layer to the lower layer. Thus, with data and attribute caching, the data movement overhead of putting layers in separate domains can be made insignificant.

The fourth interesting result is that when there is no data caching by either the non-stacked implementation or the coherency layer, the stacking overhead is insignificant. This is the case because, without caching by the coherency layer, all read and write calls go to the lower layer, which accesses the disk directly, and the disk overhead is much higher than the cross domain call overhead.

Table 3 shows the cost of open, read, write, and stat operations on SunOS 4.1.3 running on the same hardware used for the Spring measurements. The measurements show that Spring is from 2 to 7 times slower than SunOS. This is not surprising since SunOS is a production system and Spring is an untuned research prototype. Since there is no stacking overhead when reads, writes, and attributes are cached, the speed differential between SunOS and Spring for these operations is irrelevant with respect to stacking overhead. However, the open stacking overheads are very significant when compared to the much faster SunOS open operation. This just amplifies our earlier remark that if the open stacking cost adversely affects system performance, then name caching will be necessary to eliminate the stacking overhead.

Operation	Time in microseconds
open	127
4KB read	82
4KB write	86
fstat	28

TABLE 3. SunOS 4.1.3 Performance

The Spring performance measurements show that good performance is attainable with our extensible file system architecture. There are three basic ways of configuring stacked file system layers that will provide performance equivalent to non-stacked implementations:

- The layers can reside in the same domain. However, without name caching there will still be open overhead.
- The layers can be in different domains but data and attribute caching (and possibly name caching) can be used by the top layer to eliminate stacking overhead on cache hits.
- The bottom layer can be attached to a slow device such as a disk such that the overheads of higher layers are insignificant.

7 Comparisons with Related Work

In this section we compare our architecture to several systems that have recognized the need for extending the file system.

7.1 VFS

The VFS architecture provides an interface between a file system implementation and the kernel. The original notion of an *i*-node [14] was abstracted into a virtual node (vnode). The vnode interface has proven useful in abstracting file system dependencies and in supporting several file system types within the UNIX kernel [2]. In [3] a prototype implementation is described where the functionality of a file system is extended by stacking a new vnode on top of another vnode. The work described in [3] inspired the Ficus layering mechanism [4], as well as further prototyping work within the SunOS system [17]. The Ficus file system [4, 18] makes further improvements to the vnode interface to provide a layering mechanism. In particular, the Ficus system contains a delegation mechanism that allows a file system layer to dynamically extend its interface. A layer may forward an operation to a lower layer if it does not implement this operation. Our system does not provide such a mechanism.

There are several limitations to VFS-related systems:

- No general support for data or attribute coherency. The issues of page identity and coherency are muddled. There is one cache manager (the VM system), and pages are identified by an internal (vnode address, offset) pair. Yet different vnodes may want to share and maintain the coherency of the "same" data. Although Ficus has limited support for centralized page ownership that effectively implements a simple single-owner coherency protocol between the different vnodes, it is not clear how useful or general this mechanism is.
- No general transport mechanism, strong interfaces, or distributed or user-level implementations.
- New file systems are added statically, not dynamically. In addition, arcane knowledge of the UNIX kernel in general, and vnodes in particular, is needed to introduce new functionality in the system (see, for example, the discussion of inter-vnode locking in [17]).
- Naming and file mounting issues are closely intertwined with vnode composition. There is no naming system as such and no per-process view of the naming space.

We believe that introducing the notion of more than one cache manager in the system, and identifying the roles of the pagers and cache managers, are the most important differences between our approach and the approach taken by VFS-based systems. In our architecture, interfaces and protocols are defined for each of the producer (pager) and the consumer (cache manager) of the data. Vnodes are traditionally producers of data, yet when they are stacked they also become consumers of data. However, their role as data consumers is not well-defined with the end result that there is no general support for data or attribute coherency.

7.2 External Pagers

Systems such as MACH [5], Chorus [6], and V++ [19] support external pager interfaces. Our file system architecture utilizes several aspects of the Spring operating system, some of which have counterparts in Mach and Chorus. Other aspects of Spring, such as the naming architecture, the separation of the memory object from the pager object, and the stub technology, can be added to these systems. Therefore, we believe that it is possible to extend systems such as Mach and Chorus to support the architecture described in this paper. However, to our knowledge none of these systems implements such a framework.

7.3 Other Related Work

There are other systems that attempt to extend the functionality of the file system in one way or another. For example, *watchdogs* [8] provide extensions to the 4.3UFS UNIX file system that allow users to define and implement their own semantics for files. The Apollo extensible IO system [7] uses libraries linked with the application to define a framework for accessing base system I/O objects which include files. The *x*-kernel [20] supports access to multiple file systems by providing a flexible directory service that maps file names to a location where the file can be found. The *Choices* system [21] provides a framework for extensible file systems, but does not address issues of caching, coherency, and separate address spaces.

8 Conclusions and Future Work

Spring's extensible file system architecture provides a powerful mechanism for extending file system functionality by structuring file systems as a set of dynamically configurable layers. The architecture provides the ability to keep file data and attributes coherent between layers and provides the flexibility necessary to build extensible file systems without sacrificing performance.

Spring's extensible file system architecture benefits from several Spring features:

- The basic Spring object mechanism provides a flexible location-independent transport mechanism that enables the different layers to reside in either different address spaces or the same address space.

- Interface inheritance provides a clean way to extend the functionality of a file system without the need to resort to untyped interfaces (e.g., ioctl in UNIX).
- Object interposition provides a natural mechanism for interposing on files.
- The Spring naming system, with its support for flexible manipulation of the name space, enables the naming system to be largely orthogonal to the file system.
- The Spring virtual memory system provides the basic building blocks for data caching, coherency, and movement.

Although we believe it may be possible to build an extensible file system framework in other systems, Spring's many desirable features made our job much easier.

An interesting open problem is how to implement optimizations such as read-ahead and *clustering* [22] in a system that utilizes external pagers. This problem is complicated further by file system stacking. We are experimenting with extensions to the pager object interface to address this issue. One approach we are currently investigating allows a cache manager to convey to the pager the maximum and minimum amount of data required during a page-in. The pager is then given the opportunity to return more data than strictly needed.

We are also continuing our work with our extensible file system architecture. Our current work includes implementing a compression file system layer, implementing name caching, implementing an efficient bulk data transfer mechanism, and designing the proper extensible file system configuration tools. We hope that other users of Spring will take advantage of our architecture, and add important system functionality by adding new file system layers to Spring.

Acknowledgments

We would like to thank Steve Kleiman for the very helpful early discussions of this work. We would like also to acknowledge Arup Mukherjee for his help in implementing and testing the coherency layer, and Peter Madany for his help in obtaining the performance figures.

References

[1] Robert A. Gingell, Joseph P. Moran, and William A. Shannon, "Virtual Memory Architecture in SunOS," *Proceedings of 1987 Summer USENIX Conference*, June 1987.

[2] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of 1986 Summer USENIX Conference*, pp. 238-247, June 1986.

[3] David S. H. Rosenthal, "Evolving the Vnode Interface," *Proceedings of 1990 Summer USENIX Conference*, pp. 107-117, June 1990.

[4] John S. Heidemann and Gerald J. Popek, "A Layered Approach to File System Development," UCLA Computer Science Department, Technical Report CSD-910007, March 1991.

[5] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*, 37(8), pp. 896-908, August 1988.

[6] V. Abrosimov, F. Armand, and M. I. Ortega, "A Distributed Consistency Server for the CHORUS System," *Proceedings of the 3rd USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pp. 129-148, March 1992.

[7] Jim Rees, Paul H. Levine, Nathaniel Mishkin, and Paul J. Leach, "An Extensible I/O System," *Proceedings of '86 Summer USENIX Conference*, pp. 114-125, June 1986.

[8] Brian N. Bershad and C. Brian Pinkerton, "Watchdogs: Extending the UNIX File System," *Proceedings of 1988 Winter USENIX Conference*, pp. 267-275, February 1988.

[9] Yousef A. Khalidi and Michael N. Nelson, "A Flexible External Paging Interface," *Proceedings of the 2nd Microkernels and Other Kernel Architectures*, September 1993.

[10] Graham Hamilton and Panos Kougiouris, "The Spring Nucleus: A Microkernel for Objects," *Proceedings of the 1993 Summer USENIX Conference*, pp. 147-160, June 1993.

[11] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of the 1993 Winter USENIX Conference*, pp. 469-480, January 1993.

[12] Sanjay R. Radia, Michael N. Nelson, and Michael L. Powell, "The Spring Name Service," Sun Microsystems Laboratories, Technical Report SMLI 93-16, September 1993.

[13] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "Experience Building a File System on a Highly Modular Operating System," *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEMDS IV)*, September 1993.

[14] Kirk McKusic *et al.*, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, 2(3), pp. 181-197, August 1984.

[15] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer*, vol. 23(5), pp. 9-21, May 1990.

[16] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computing Systems*, 6(1), pp. 134-154, February 1988.

[17] Glenn C. Skinner and Thomas K. Wong, "Stacking Vnodes: A Progress Report", *Proceedings of Summer 1993 USENIX Conference*, pp. 161-174, June 1993.

[18] Richard G. Guy, *et al.*, "Implementation of the Ficus Replicated File System," *Proceedings of Summer 1990 USENIX Conference*, pp. 63-71, June 1990.

[19] Kieran Harty and David R. Cheriton, "Application-Controlled Physical Memory Using External Page-Cache Management," *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 187-197, September 1992.

[20] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao, "The *x*-kernel: A Platform for Accessing Internet Resources," *IEEE Computer*, vol. 23(5), pp. 23-33, May 1990.

[21] Roy H. Campbell, Nayeem Islam, and Peter Madany, "Choices, Frameworks, and Refinement," *Computing Systems*, vol 5(3), pp. 217-257, Summer 1992.

[22] L. W. McVoy and S. R. Kleiman, "Extent-like Performance from a UNIX File System," *Proceedings of Winter 1991 USENIX Conference*, January 1991.

A Cache object interface definition

We list in this appendix the interface of the cache object exported by cache managers (the VMM and pagers acting as cache managers). In Appendix B we list the interfaces of the objects exported by the pagers. The code below specifies for each parameter a passing mode: a Spring object passed *copy* remains accessible to the caller and callee after the call is made, while a *consumed* object is deleted from the calling domain as a side effect of the call. *Borrow* is an in-out passing mode, while *produce* is an out mode. Due to space considerations we elide some methods, most comments, and all type declarations. Most methods raise exceptions when errors are encountered; we elide the description of the exceptions as well.

```
// Cache objects are implemented by cache managers and
// are invoked by pagers.
interface cache_object {
 // Remove data from the cache and send modified blocks to
 // the pager.
 void flush_back(copy offset_t offset, copy offset_t size,
  produce data memory);
 // Downgrade read-write blocks to read-only and return
 // modified blocks to the pager.
 void deny_writes(same parameters as flush_back());
 // Return modified blocks to the pager. Data is retained
 // in the cache in the same mode as before the call.
 void write_back(same parameters as flush_back());
 // Remove data from the cache—no data is returned.
 void delete_range(copy offset_t cache_offset,
  copy offset_t size);
 // Indicate that a particular range of cache is zero-filled.
 void zero_fill(same parameters as delete_range());
 // Introduce data into the cache.
 void populate(copy offset_t cache_offset,
  copy offset_t size, copy rights access_rights,
  copy data memory);
 void destroy_cache();
}; // cache_object interface
```

B Pager objects interface definitions

```
interface memory_object {
 // Return a cache_rights object that the caller can use
 // to locate a pager-cache object connection. The name
 // passed in is used to identify the cache manager making
 // the call.
 void bind(copy name caller, copy rights requested_access,
  copy offset_t mem_obj_offset, borrow offset_t length,
  produce cache_rights rights, produce offset_t offset);
 void get_length(produce offset_t length);
 void set_length(copy offset_t length);
}; // memory_object interface

// Pager objects are implemented by pagers and are
// invoked by cache managers.
interface pager_object {
 // Request data be brought from the pager object in
 // read-only or read-write mode.
 void page_in(copy offset_t offset, copy offset_t size,
  copy rights requested_access, produce data memory );
 // Write data to pager. Data is no longer retained by
 // the caller.
 void page_out(copy offset_t offset, copy offset_t size,
  copy data memory);
 // Write data to pager. Data is retained in read-only mode
 // by the caller.
 void write_out(same parameters as page_out());
 // Write data to pager. Data is retained in same mode
 // by the caller.
 void sync(same parameters as page_out());
 // done_with_pager_object is called by cache
 // manager when it closes its end of this connection.
 void done_with_pager_object();
}; // pager_object
```