The Spring File System

Michael N. Nelson
Yousef A. Khalidi
Peter W. Madany

SMLI TR-93-10 February 1993

Abstract:

The Spring File System supports cache coherent file data and attributes. It uses the virtual memory system to provide data caching and uses the operations provided by the virtual memory system to keep the data coherent. The file system consists of two types of file servers: ones that provide coherent access to files they export and ones that run on each Spring machine and provide caching of data for read and write operations, and has its own private protocol with the remote file servers to cache file attributes.

Categories and Subject Descriptors:

D.4 Software:
[**Operating Systems**]: D.4.2 Storage management
D.4.3 File systems management
D.4.7 Distributed systems

Additional Keywords and Phrases: Caching, Distributed Shared Memory, Object-oriented Operating System, External Pagers

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email address:
yak@eng.sun.com
mnn@eng.sun.com
madany@eng.sun.com

The Spring File System

Michael N. Nelson, Yousef A. Khalidi, Peter W. Madany

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Network file systems that utilize caching to provide good performance have existed for many years. Examples of these file systems include Sprite [1] and Andrew [2]. However, until recently all of these file systems were implemented as part of a monolithic operating system. With the recent advent of microkernel systems (e.g., Mach [9] and CHORUS [8]) file systems are now being implemented outside the kernel in user level servers. These new file systems must solve a new set of problems if they are to provide performance equal to the file systems of the past.

This paper describes the Spring File System, which is built on top of Spring – a highly modular, distributed, object-oriented operating system. Spring has several properties that provide unique opportunities and challenges when building a file system, including:

- A powerful virtual memory (VM) system with support for external pagers and operations that allow the construction of distributed shared memory systems.
- A naming system that allows objects of all types, including files, to be bound into the namespace.
- A capability-based security model.
- An object model that allows objects to be passed freely between domains on the same or different machines.

The Spring File System was designed to take advantage of these and other unique Spring properties to build a powerful distributed file system. More specifically the goals of the Spring File System were:

- Provide access to files on the local disk as well as remote UNIX files.
- Allow per-machine caching of data and attributes to eliminate network accesses for remote files.
- Provide coherent distributed shared memory and coherent file attributes for shared files.
- Build the system within the Spring model including using the Spring security and naming models and utilizing the Spring virtual memory system [3].

The Spring File System that we implemented accomplishes all of these goals. The file system consists of two types of file servers: ones that provide access to files that they implement and ones that cache accesses to files implemented by remote file servers. File servers of the first type are responsible for providing access control and providing coherent access to file data and attributes. These file servers keep the data coherent by using the primitives provided by the virtual memory system and they keep the attributes coherent by using a private protocol with caching file servers. There are two file servers of this first type: one that runs on each Spring machine and provides access to files on the local disk (this is called Local-FS or LFS) and one that runs on SOUP[1] and provides access to UNIX

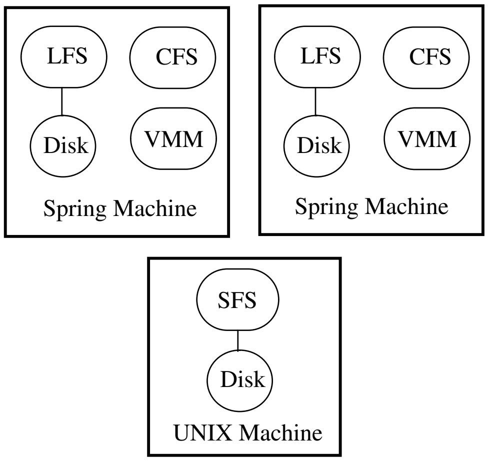

FIGURE 1. Spring File System Components

files (this is called SOUP-FS or SFS). Except for the implementation of file storage, these two file systems use exactly the same code. Throughout this paper we will refer to these two file system implementations collectively as the *storage file servers.*

The second type of file server, ones that cache accesses to remote files, is responsible for making access to remote data and attributes efficient. One of these file servers runs on each Spring machine that desires to have file caching (this file server is called Caching-FS or CFS); access to remote files will work fine if there is no CFS, access will just be slower. The CFS utilizes the virtual memory system to provide caching of data for read and write operations and has its own private protocol with the remote file servers to cache file attributes.

Figure 1 shows the various components of the Spring File System. The LFS and SFS export files that they implement. The CFS caches file attributes for files implemented by the LFS and SFS. The Virtual Memory Manager (VMM) caches data for LFS and SFS files. For example, when a client invokes a read or write operation on a file object the operation goes to the CFS. The CFS handles the operation by mapping the file into its address space thus utilizing the VMM to cache file data. The VMM gets the file data by communicating with the appropriate storage file server.

1. SOUP is the Spring on UNIX prototype. It allows Spring applications to run on UNIX and communicate with other SOUP systems and native Spring systems.

The Spring File System utilizes a unique dynamic caching protocol to allow file objects to be cached by the CFS. Under this protocol the CFS is contacted to cache file objects when they first appear in a client domain. The result is that file objects are always cached by the CFS on the same machine as the client that possesses the file object.

Besides the file servers there is a name server that resolves names to files. Under the current implementation the storage file servers implement name lookup for the file objects that they implement. However, files that they implement can also be bound into other name servers and retrieved through them.

The resulting file system provides excellent performance. Once files are cached on the local machine, no remote operations are required to perform any operation on the file. Preliminary measurements show that caching allows basic file operations such as read, write, and map to be executed at least five times faster than without caching.

The rest of this paper is organized as follows: Section 2 provides an overview of the Spring Operating System; Section 3 discusses the file interface; Section 4 discusses the fs and fs_context object interfaces and discusses how clients obtain file objects; Section 5 describes the implementation of coherent files by the storage file servers; Section 6 describes the implementation of a swap server object by the storage file servers; Section 7 describes the implementation of the CFS and discusses the coherency protocol used by it and the storage file servers; Section 8 describes how we use name caching to improve caching performance; Section 9 discusses cache reclamation issues; Section 10 provides details about the LFS and SFS implementations; Section 11 discusses performance; Section 12 discusses related work; Section 13 describes the current status of the implementation and future work; and Section 14 offers some conclusions.

2 Spring Operating System

Spring is a distributed, multi-threaded, extensible operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of methods to manipulate that state. The description of the object and its methods is an *interface* that is specified in an *interface definition language*. The interface is a strongly-typed contract between the implementor (*server*) 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 clients of other objects. The implementor and the client can be in the same domain or in different domains.

Since Spring is object-oriented it supports the notion of *interface* inheritance. Spring supports both notions of *single* and *multiple* interface inheritance. Interface inheritance is an important factor in making Spring extensible. An interface that accepts an object of type *foo* will also accept an instance of a subclass of foo. For example, the *address space* object has a method that takes a *memory object* and maps it in the address space. The same method will also accept *file* and *frame buffer* objects as long as they inherit from the memory_object interface.

The Spring kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a *network proxy* server. In addition, the virtual memory system depends on external pagers to handle storage and network coherency (see Section 2.1).

A typical Spring node runs several servers in addition to the kernel (Figure 2). These include the domain manager and the virtual memory manager (VMM); a name server; the file servers described in this paper; a linker domain that is responsible for managing and caching dynamically linked libraries; a network proxy that handles remote invocations; and a tty server that provides basic terminal handling as well as frame-buffer and mouse support.

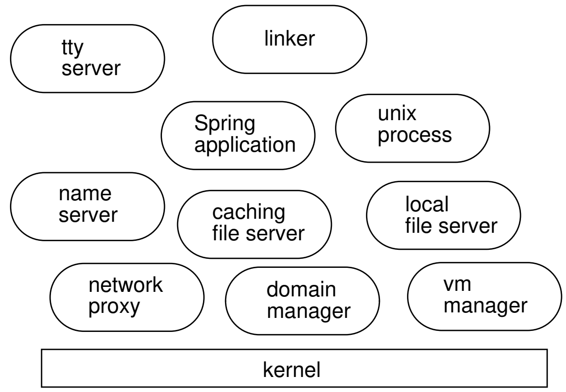

FIGURE 2. Major Spring system components

2.1 Spring Security

If the implementor and the client are in a different domain, the representation of the object includes an unforgeable nucleus *door* (or *handle*) that identifies the server domain. These unforgeable handles have many of the security properties of capabilities in traditional operating systems. If a server determines that a client is entitled to specific access rights to a given piece of state (e.g., a file), it can give the client an unforgeable handle X. Encapsulated in the server side state for handle X will be the granted access rights and possibly the principal name of the client. Whenever a call arrives quoting handle X, then the server can permit the given access to the underlying state without further checks.

Servers determine if a client is allowed access to a piece of state by consulting an access control list (ACL) that is associated with the state. Each ACL entry contains a principal name and a list of access rights. A server will only believe that a client is a given principal if that client has first been authenticated to be that principal. Once a client has been authenticated to the server as a given principal P, then the server will be willing to return objects that grant specific rights for P as determined by the ACL.

Authenticated Spring objects can be freely passed between domains. Once an object has been issued that contains a set of encapsulated rights, anyone to whom an object is passed will be allowed the same rights.

2.2 Spring Naming

The Spring naming service allows any object to be associated with any name. A name-to-object association is called a *name binding*. Each name binding is stored in a context. 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. By binding contexts we can create a *naming graph*—a directed graph with nodes and labelled edges where the nodes with outgoing edges are contexts. The UNIX file system is a naming graph that is frequently restricted to a tree.

Objects are retrieved from a context by invoking the *resolve* method, bound by invoking the *bind* method, and removed by invoking the *unbind* method. In addition there are methods to obtain a list of all of the bindings in a context, to create contexts, to retrieve multiple objects at once, and to get the statistics about a context.

Spring contexts provide support for the Spring security model. When an object is bound, an ACL can be given that specifies which principals are allowed which *modes* on an object. Modes are a superset of rights. For example, read and write modes correspond directly to read and write access rights; however, file_append mode implies write access but also indicates the "mode" with which the file should be accessed when writes occur. When an object is resolved, a set of desired modes can be specified and an object with the desired modes will be returned if the client doing the resolve is allowed the corresponding rights.

2.3 Spring VM

Spring provides a powerful virtual memory architecture that utilizes an external pager interface for paging memory. The virtual memory system plays an important part in our file system architecture. The rest of this section presents an overview of the Spring virtual memory system; more details can be found in [3].

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

Most clients of the virtual memory system only deal with two types of objects: *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 read-only file object obtained from the Spring File System (the file interface in Spring inherits from memory object). The same memory object can be mapped into more than one address space at the same time on more than one machine. An address space may be sparse and can be operated on by any client that holds its address space object. Note that the memory object interface does *not* provide page-in and page-out or read/write methods; a holder of a memory object can either map it into an address space or pass it to another client.

The address space object provides a rich set of operations, including the ability to map and unmap memory objects into the address space, to change protections of an address range, and to handle faults on the address space. A memory object has methods to set and query the length, and methods to *bind* to the object (see below). There are no page-in/out or read/write methods on memory objects. The Spring file interface which inherits from memory object provides file read and write methods (but not page-in and page-out methods).

There are two types of servers that co-operate to provide the implementation of address spaces and memory objects: the per-node VMM and *pagers*. The VMM is the implementor (object manager) of address space objects, while memory objects are implemented by pagers. Pagers supply (page-in) and store (page-out) the actual contents (bits) of the memory objects. Any domain can act as a pager.

2.3.1 Binding a Memory Object to a Cache Object

A VMM presented with a request to map a memory object into an address space must be able to obtain the actual data represented by the memory object, since the memory object itself does not provide methods for obtaining this data. Therefore, the VMM contacts the pager that implements the memory object by invoking the *bind* method on the memory object. The objective of the bind operation is to give the VMM a local data cache that provides the contents of the memory object.

During a bind operation the VMM and the pager exchange two objects: a *pager* object and a *cache* object. The pager object, which is implemented by a pager domain, provides methods to page-in and page-out memory blocks and is used by the VMM to populate a local cache. The cache object is implemented by the VMM and is used by the pager to affect the state of a cache. A given pager object—cache object pair constitutes a two-way communication channel between a pager and a VMM. Typically, there are

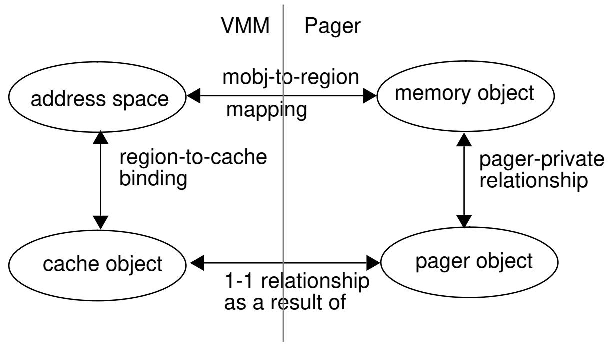

FIGURE 3. Relationship among basic VM objects

many such channels between a given pager domain and a VMM. Figure 3 summarizes the relationship among the various objects.

As far as the VMM is concerned, each memory object is unique; the VMM relies on the memory object's pager to point it to a data cache implemented by the VMM from which the VMM obtains the contents of the memory object. This extra level of indirection has two advantages. First, it allows different memory objects that share the same pages (but perhaps encapsulate different access rights) to have their pages cached in the VMM instead of flushing the same pages back and forth between the VMM and the pager. Second, it allows the memory object and the pager object to be implemented by two different domains. An application of this feature is described in Section 7.

Pager domains acquire cache objects by asking the VMM to create them as the side effect of a bind (see Figure 4).

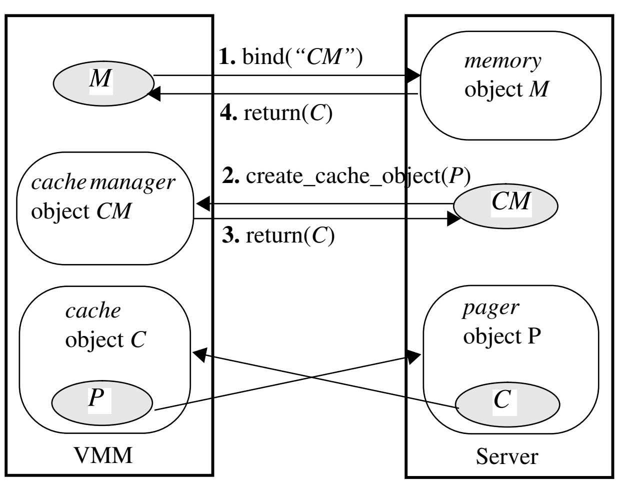

FIGURE 4. Bind protocol

When a VMM issues a bind call on a memory object *M* to a pager (step **1** in Figure 4), the VMM includes a name *CM* that can be resolved to a *cache manager* object implemented by the VMM. When a bind call arrives at a pager, the pager goes through the following steps:

- The pager checks if a VMM named *CM* is caching *M*.
- If not, the pager resolves *CM* to a cache manager object using the Spring naming service.
- The pager then invokes the *create_cache_object* method on the cache manager object (step **2** in Figure 4) with a pager object *P* implemented by the pager as a parameter. The VMM responds to this call by returning a new cache object *C* (step **3**).
- The pager domain takes *C* and *CM* and enters this information into its data structures so that future binds for the memory object *M* from the given VMM will use the same cache.
- Once the pager has a cache object *C* for the given VMM it returns *C* as the result of the bind call (step **4** in Figure 4).

The pager domain does not actually return a cache object as the result of a *bind* operation. Instead, for security reasons described below, it returns a *cache rights* object. This object is implemented by the VMM and encapsulates a set of rights to an associated cache. The VMM can use this cache rights object to find the associated cache and discover what rights the user doing the map is allowed on the cache; the cache rights object is useless to anyone else. Thus the *create_cache_object* call actually returns an array of <cache rights object, encapsulated access> pairs in addition to the cache object. The pager domain responds to bind requests by returning a cache rights object that encapsulates the rights that are encapsulated in the file object on which the bind was invoked. Since the pager gave the VMM a pager object when it created the cache object, the VMM can use the returned cache rights object to discover not only the allowed rights and associated cache, but the associated pager object as well. See Figure 5 for the state after a successful bind has occurred.

The list of cache rights objects returned from the *create_cache_object* call need not contain objects that encapsulate all possible access rights. A cache rights object supports the *create_restricted_sibling* method that takes a desired set of rights and returns a new cache rights object that encapsulates the given rights; the desired rights must be lower than the actual encapsulated rights in the

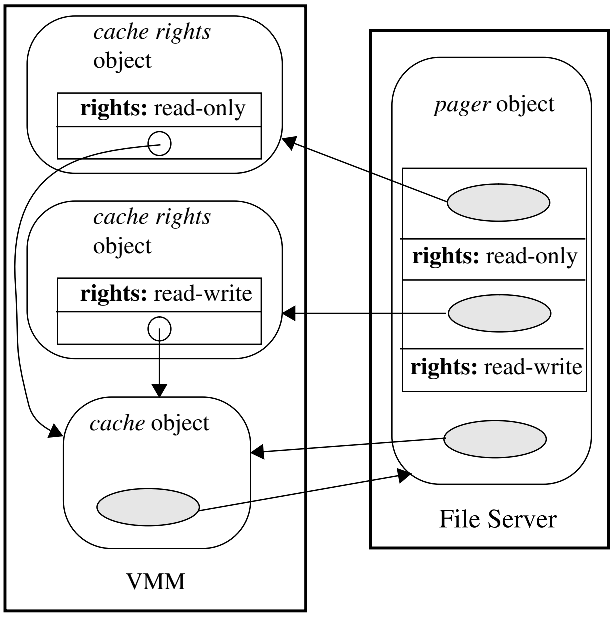

FIGURE 5. State after a successful bind.

invoked cache rights object. Thus as long as the VMM returns at least one cache rights object that encapsulates the highest possible access rights, the pager can create a cache rights object with any needed set of encapsulated rights.

The reason a cache rights object is returned instead of a cache object is the following. A pager uses the cache manager name to determine if a VMM is already caching a file object. This cache manager name is forgeable. Thus any random client could invoke the *bind* call on a memory object and specify the name of some VMM. If the pager were just to return a cache object from the bind it could inadvertently give a read-write cache object to a client that is only allowed read access to the file. By using the cache rights object we can guarantee the following after a successful bind:

- The only domains that have access to cache objects are the VMM that created the cache object and the pager that invoked the *create_cache_object* method that created the cache.
- The rights granted to the client that is mapping the file cannot be forged since the access rights are encapsulated in an unforgeable cache rights object.

2.3.2 Cache Object - Pager Object Interaction

The VMM issues several requests to the pager object. For each request, the VMM specifies the starting offset in the cache and a length (both aligned to the block size of the cache), whether the request is for read-only or read-write data, and the current cache ownership state. Data is transmitted using the normal Spring bulk data passing mechanism [3]. As an optimization, if the data is null (all zeroes) an indication is sent instead of the actual data. The requests issues by the VMM to the pager object are:

- page_in: used to bring data into the cache.
- page_out: used to move data out of the cache.
- write_out: used to change the state of read-write data to read-only. Data is retained in read-only mode in the cache. Only called for read-write data.
- sync: used to copy data back to the pager. Data is retained in same mode.

For clean (i.e., non-modified) blocks and for cache objects that are not bound read-write (i.e., no read-write mappings exist to these caches), the VMM will never transmit the actual data in page_out, write_out, or sync calls. The pager is at liberty to cease communicating with a misbehaving VMM that is violating the protocol. In addition, the system is designed such that the VMM can always reuse a clean page without the need to inform the pager.

The pager issues the following requests to the cache object:

- flush_back: removes data from the cache and sends it back to the pager.
- deny_writes: “downgrades” read-write blocks to read-only. Any modified blocks are returned. A copy of the data is retained in the cache in read-only mode.
- write_back: requests a copy of all modified blocks. Data is retained in the cache in the same mode as before the call.
- delete_range: removes data from the cache. No data is returned.
- zero_fill: indicates that a particular range in the cache is zero-filled. The data blocks indicated by this range are held by the VMM in read-write mode.
- populate: introduces data blocks into the cache.

3 The File Interface

Spring files contain data and attributes and support authentication and file locking. The interface provides access to the file's data through two mechanisms. One way is through *read* and *write* methods; these methods are inherited from the Spring *io* interface. The other way is by mapping the file object into an address space: This works because a file object inherits the *memory* object interface.

The file interface provides a *write_back* method to force modified data safely to stable storage. This method takes a range of bytes and will not return until all bytes in the range are safely on stable storage.

Spring files have three attributes—length, access time, and modify time. The file interface provides *get_length* and *set_length* methods to retrieve and change the file length; these methods are inherited from the memory object interface. All three of the attributes can be retrieved via the *stat* method; there is no direct way to modify the access or modify time.

Spring files also support file locking. Files support single writer, multiple reader semantics. The *read_lock* method will succeed if the file is not write locked. Once a read lock is granted all writes will fail. The *write_lock* method will succeed if the file is open for writing and no one has either a read or a write lock on the file. Once a write lock is granted, all reads and writes by other equivalent[2] file objects will fail. The *unlock* method will release any lock previously granted by the *read_lock* and *write_lock* methods.

The Spring file interface does not have a *close* method. When file objects are no longer needed they are consumed just like any other Spring object. This will cause the file server to be called with an unreferenced call[3] on the file object when the last reference to the file object goes away.

Spring files support Spring authentication by inheriting the *authenticated* interface. The authenticated class provides support for access control lists, encapsulated rights and principals, and allows new file objects to be created that reference the same underlying file state as the current file, yet contain different encapsulated rights.

In summary, the file objects implemented by the Spring File System inherit from the io, memory object, and authenticated interfaces. In addition the files support methods for locking, getting and setting attributes, and forcing a file's data to disk.

4 Obtaining File Objects

File objects can be obtained in two ways: looking them up in a name server or by invoking a method on a *fs* object. This section discusses the implementation of the file system naming context, the file and context duplication services that allow files and contexts to be stored in generic name servers, and the fs object.

4.1 The File System Context

The storage file servers implement naming contexts that allow users to use the normal Spring naming operations to access files implemented by the storage file servers. The context implemented by the storage file servers is a subclass of context called *fs_context*. The fs_context class adds the *create_file* method. The methods in the context class support resolving, binding, and unbinding files, but they don't support file creation. The *create_file* method allows a file to be created in a given context and bound to a name.

The fs_context implementation in the storage file servers support all of the methods in the *context* class. However, there are some limitations:

- Only file objects and symbolic link objects can be bound.
- Non-empty contexts cannot be unbound.

The fs_context class inherits from the authenticated interface. Thus the fs_context objects implemented by the file servers contain an encapsulated principal, encapsulated rights, and an access control list. The encapsulated rights are used to perform access checks on naming operations. When a resolve occurs on an fs_context object, the file servers ensure that the encapsulated rights of the object include read access. Likewise, if a bind occurs the file

2. Two file objects are equivalent if they reference the same underlying file state (i.e., the same file on disk).

3. Spring guarantees that when the last reference to an object goes away the object manager will receive an unreferenced call on the object.

servers ensure the encapsulated rights contain write access.

The ACL on an fs_context object is used when a client wishes to upgrade their access to the fs_context object. This will involve an authentication exchange. When a file server is asked to upgrade access on an fs_context object for a given principal *P* it will check the ACL to see if *P* is allowed the desired access. If the access is allowed then the file server will deposit the object with the desired rights into an authentication manager and return a ticket to the client. If the client doing the upgrade can authenticate itself to the authentication manager as principal *P,* then the authentication manager will return the newly upgraded context when presented with the proper ticket.

In addition to the ACL on each fs_context object, there is also an ACL on each name to object binding in the context. This ACL is used to determine which principals are allowed which access rights to the bound object.

The encapsulated principal in an fs_context object is used during naming operations. To explain how encapsulated principal is used during name resolution, denote the encapsulated principal of the fs_context object on which the operation is being performed *P* and denote the context FCO_0. A naming operation on a name with *N* components can be viewed as a two step process: resolving the $N - 1$ components to an fs_context FCO_{N-1} and then performing the operation on FCO_{N-1} with the last component:

$$FCO_0\text{->operation}(C_1/C_2/ \ldots /C_N) =$$
$$FCO_0\text{->resolve}(C_1/C_2/ \ldots /C_{N-1})\text{->}operation(C_N)$$

The encapsulated principal *P* is first used when resolving the first $N - 1$ components. Each of these $N - 1$ components C_i must be bound to an fs_context object FCO_i. At each of the resolution steps for i equals 1 to $N - 2$, *P* is checked against the ACL on the binding for component C_i to see if the given principal is allowed read access. When the $N - 1$ component is resolved, *P* is checked against the ACL on the binding for component C_{N-1} to see if the given principal is allowed the access needed to perform the naming operation (read access for operations such as resolve and write access for operations such as bind).

If the operation being performed is *resolve* then an object must be returned. The mode given to the *resolve* method is checked against the ACL on the binding for C_N to see if *P* is allowed to return an object with the rights that correspond to the mode. If the access is allowed, then an object is returned that encapsulates principal *P* and the desired mode.

4.2 Naming Separate From FS

File and fs_context objects are just like any other objects as far as the naming service is considered: they can be bound, unbound, and retrieved. The only complication is when file or fs_context objects with particular modes need to be returned. When the resolution goes directly to a storage file server as described in Section 4.1 it can easily return an object with the desired mode. However, when a file object is retrieved from a context that isn't implemented by one of the storage file servers, a Spring duplication service must be used.

The purpose of a duplication service is to take an object and return an object with the desired modes. A standard Spring naming server does not know how to change the modes of an object. Thus any object manager that wishes to allow their objects to be stored in name servers and allow the modes to be changed, must implement a Spring duplication service object. This object supports two methods:

- *dup* takes an object and a principal name and returns a copy of the object that encapsulates that principal. The copy has the same modes as the original object.
- *dup_with_mode* takes an object, a principal, and a desired set of modes and returns a copy of the object that encapsulates the given principal and modes.

The file servers implement two duplication services: one for files and one for contexts. A file server will only duplicate objects that it implements. If a file server is given an object to duplicate that it does not implement, then the file server will raise an exception.

4.2.1 File Duplication Service

The file duplication service is not interested in the principal name that is passed to the *dup* methods because file objects don't need an encapsulated principal (see Section 5.1). Thus the implementation of the *dup* method merely makes a copy of the passed in object. The copy has the same modes and shares the same underlying file state as the original object.

The file duplication service is interested in the modes passed to the *dup_with_mode* method. When a *dup_with_mode* request comes in, the file server checks to see if the rights that correspond to the modes are less than or equal to the encapsulated rights. If so it creates a new object that references the same file as the original object but with the new encapsulated rights. A file server will always be willing to return a file with less rights than those encapsulated in the object that the server is passed because whoever called the duplication service already possessed a file with greater rights. However, a file server will not increase the encapsulated rights unless it trusts the invoker.

4.2.2 The Context Duplication Service

The context duplication service is interested in both encapsulated principals and encapsulated rights. The encapsulated principal in contexts is important for the reasons given in Section 4.1. The context duplication service will not allow objects to be duplicated with new principals unless it trusts the invoker of the duplicator object. Clearly letting anyone arbitrarily change encapsulated principals on context objects would be insecure.

Assuming that the context duplication service trusts the invoker of the duplicator object, it will allow fs_context objects to be duplicated if the ACL on the fs_context object allows the needed mode to the encapsulated principal. In the case of the *dup* method the given principal must be allowed the current encapsulated rights on the object. With *dup_with_mode* the given principal must be allowed the desired mode.

4.3 The FS Object

The *fs* object allows file servers to create unnamed files. The fs object supports one method, *get_file,* that returns a new unnamed file object. In order for this new file object to be bound to a name it must be bound into some context.

5 File Implementation

Files are implemented by the storage file servers. In this section we will discuss the file implementation. We will focus in particular on the implementation of distributed shared memory. In addition we will help motivate the need for the CFS by pointing out where local caching can eliminate network accesses.

5.1 Security

The file objects implemented by the storage file servers are authenticated objects. Therefore they have both an encapsulated principal and encapsulated rights. The encapsulated rights are set when a file object is created and the rights are checked on each access to the object. The encapsulated principal is not used for file objects because file objects are capabilities: once a file object that encapsulates a set of rights is given to someone those rights will be granted regardless of the fact that the principal that acquired the object may no longer be allowed the rights.

File objects support the authenticated object methods to create new *sibling* objects. A sibling of an object is an object that shares the same underlying state as the original object yet encapsulates different access rights. These methods are implemented by using the Spring library authentication package.

5.2 Mapping Files

As we described in Section 2.3, Spring files can be mapped into address spaces because the Spring file class inherits the memory object class. When a client maps a file object into its address space, the virtual memory system and the file system follow the bind protocol described in Section 2.3.1. The result is that the cache – pager object connection between the VMM and the file system is set up. Figure 6 gives the state of the system after a file object is bound into a client domain's address space.

Note that each time a client domain maps a file into its address space, a bind call must go to the file server. If the file is remote, this requires a network access on each map call. If the result of binds could be cached locally, then many of these network accesses could be eliminated.

5.3 Keeping Files Coherent

There is a potential coherency problem when a particular file is mapped into multiple client's address spaces on several machines at the same time. For example, if two clients on different machines have the same page of a file mapped into their address spaces both readable and writable, then some action must be taken to ensure that both clients see a coherent view of the page. One of the goals when building the file system was to give clients a coherent view of files. As a result one of the primary jobs of the file system is to keep files coherent.

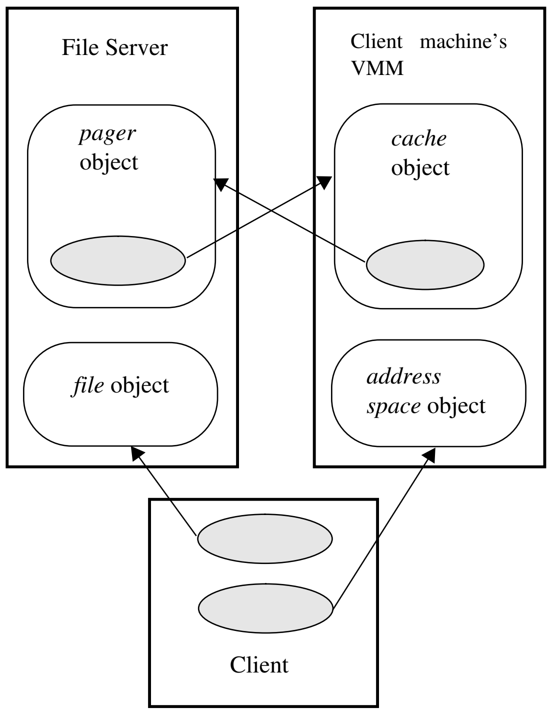

FIGURE 6. State after a file has been mapped.

The file server implements the file object. When the file is mapped into the client's address space, a pager object is created at the file server and a cache object is created at the client's VMM.

Although providing coherent files adds complexity to the file system, we believe that it is necessary if we intend to have a network transparent system. By keeping shared files coherent, we are providing distributed shared memory. Thus clients can communicate transparently via shared files; they don't have to know whether other domains that they are communicating with are local or remote. This network transparency can be exploited to dramatically improve performance by distributing applications such as **make** across multiple machines.

Since files can be accessed a page at a time, coherency is done on a per page level. The file system ensures that if a page is writable, then the page will only be in a single cache. If a page is only being read, then copies of the page may be present in any number of caches. The file system can guarantee this because it gets all page-in requests. Each request indicates whether the page is desired in read-only or read-write mode.

A page-in request can cause a page to go from read-only to read-write or from read-write to read-only. When this transition happens the file server must ensure that coherency is maintained before returning a page to the requesting client. It does this by invoking methods on the cache objects that are associated with each memory object. If a page transitions from read-only to read-write, then the file server invokes the *flush_back* method on all cache objects that currently possess the page; if the client that requests the page read-write already has the page readable, then the page is not flushed from the requesting client's cache. If a page transitions from read-write to read-only, then the file system invokes the *deny_writes* method on the cache object that possesses the page; *deny_writes* will force a cache to write the page back if it is dirty and then make the page read-only.

The VMM can perform an optimization if it knows that it is the only client caching a file's data. If a VMM knows no one else is caching data for a file and the page being faulted on is mapped read-write, then the VMM can fetch the page read-write even if the page fault is read-only. This will avoid a subsequent page fault if the same page is modified. This should only be done if no one else is caching a file's data because it can cause extra coherency traffic if shared pages are fetched read-write unnecessarily. The file servers allow the VMM to perform this optimization by informing the VMM on each page-in or page-out operation whether or not the file is cached by multiple VMMs.

5.4 Periodic Data Write Back

Since data can be cached by a VMM, data can be lost if the VMM's machine crashes. In order to reduce the amount of data lost in a machine crash, the storage file servers write back all of the modified data for their files in the VMM cache every 30 seconds. The file servers do this by invoking the *write_back* method on the cache object.

The file servers don't actually write back all files every 30 seconds. They don't write back the anonymous file objects mentioned in Section 6 and they also don't write back files that are in an fs_context named "tmp". These files aren't written back because it is assumed that the data need not persist across system crashes.

5.5 Getting and Setting File Length

Other operations that can be performed on files are getting and setting the file length. Getting the file length just involves invoking the *get_length* method on the file object; since there are no side-effects of the *get_length* method, it has no impact on file data coherency. However, if the file is remote, *get_length* will require a network access. This is another place where local caching can be effective.

Setting the length involves invoking the *set_length* method on a memory object. There are two interesting cases with *set_length*: the length is decreased and the length is increased.

If the length is decreased, then the pages at the end of the file need to be eliminated from the file and from all caches of the file. If the pages aren't removed from the caches, then clients won't see a consistent view of the file because some clients may be able to access parts of the file that no longer exist. Pages are deleted from caches by invoking the *delete_range* method with the appropriate data range on all cache objects that possess deleted pages.

If a file's length is increased, then nothing has to be done in order to ensure coherency. However, there is an opportunity for an optimization that can best be done by the caching file server. We will discuss this optimization in Section 7.8.

5.6 Reading and Writing Files

File objects support *read* and *write* methods to access data. When these operations are invoked on a file object, the request goes to the file server. If the most recent data for a page that is being accessed is being cached by some remote virtual memory cache object, then the *deny_writes* or *flush_back* method on the cache object must be invoked to get the most recent data to the file server.

Note that all reads and writes must go to the file server. Thus if the file server is executing on a remote machine, all reads and writes will require a network access. This is one of the areas where local caching has been used in the past and has proven to be very effective [1].

5.7 Getting File Attributes

All of a file's attributes are returned via the *stat* call. In current UNIX systems, *stat* calls are very frequent. This is yet another area where caching can be effective.

6 The File System as a Swap Server

The virtual memory system needs to acquire memory objects that can be used for stacks and heap memory. These objects are called *anonymous* memory objects and are acquired from *swap_pager* objects. The storage file servers implement swap_pager objects.

There is only one method on a swap_pager object: *get_anon_mobj*. This method takes a list of cache objects implemented by a given virtual memory system and returns a list of memory objects and associated pager objects. The file servers return file objects when the *get_anon_mobj* call is invoked. However, these are not normal file objects—the file servers allocate internal data structures for these file objects, but they don't create any on-disk structures until absolutely necessary.

One of the decisions that we had to make when implementing the swap_pager objects was whether to pre-allocate swap space on disk or not. The virtual memory system invokes the *set_length* method on the anonymous memory objects returned by the file system in order to inform the file server how much swap space is needed. However, we chose not to allocate disk blocks for the swap space until a page-out occurs that requires new disk space. In fact we don't even open a file until the first page-out occurs. This is more efficient in time when allocating anonymous memory objects and in space because disk space is only allocated when it is actually used. However, this has the disadvantage that page-outs of anonymous memory could fail due to lack of disk space. We will revisit our decision to not pre-allocate swap space if this turns out to be a problem.

7 The Caching File Server

We previously described the implementation of the storage file servers. During this discussion we mentioned the many opportunities where local caching of data and attributes can eliminate network accesses. In this section we describe the implementation of the Caching File Server (CFS) which exploits all of the caching opportunities that we mentioned, including:

- Attribute caching to eliminate remote *get_length* and *stat* calls.
- Data caching to eliminate remote *read* and *write* calls.
- VM cache object caching to eliminate remote *bind* calls and allow an additional optimization that eliminates most zero-fill page faults.

7.1 Basic Structure

In order to allow local file caching to be implemented, the file objects used by client domains must be implemented by the CFS. In addition the CFS must have a special communication channel for caching with the file servers whose data and attributes its caching and a copy of the VMM cache object for the file.

The other component of the caching structure is the virtual memory system. The virtual memory system uses the cache and pager objects described in Section 2.3. In order to make page-ins and page-outs as efficient as possible, the virtual memory manager should be able to communicate directly with the file server that stores the data; in other words, the pager object should be implemented by the file server, not the file cacher. The desired structure for data caching involving the CFS, the storage file server, and the VMM is given in Figure 7.

7.2 The Caching Subcontract

When client domains receive objects from a remote file server, the CFS must somehow be able to interpose on these objects so that caching can occur. This is done through the use of the *caching* subcontract.

Every Spring object has an associated subcontract. Subcontract is responsible for many things including marshaling, unmarshaling, and invoking methods on the object. Subcontract also defines the representation for each object that appears in a client domain's address space. The standard Spring subcontract is called *singleton.* The representation of a singleton object includes a nucleus *door* (or *handle*) that identifies the server domain. When a client invokes on an object that uses singleton, this handle is used to send the invocation to the server domain.

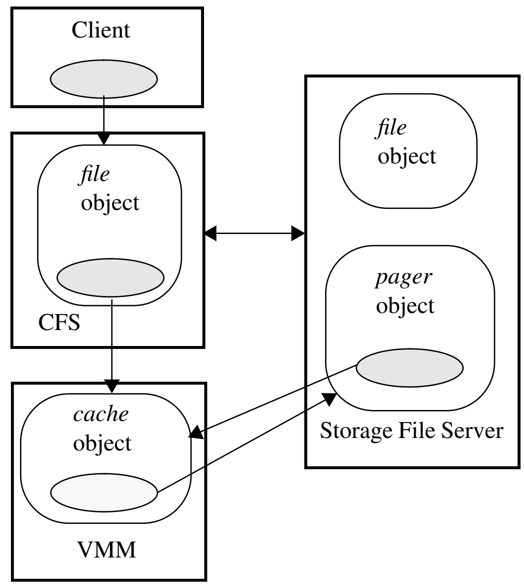

FIGURE 7. Desired caching structure

The client has a file object that is implemented by the CFS. The CFS has a private caching communication channel with the storage file server. If the contents of the file object is cached by the VMM, then the VMM has a pager object implemented by the storage file server and the CFS has a copy of the VMM cache object.

File objects use a different subcontract called the *caching* subcontract. The representation for an object that uses the caching subcontract contains:

- A handle that identifies the server domain (this is the same handle that is in the singleton representation).
- An object, called the *cached_object*, that is implemented by a domain that caches the original object.
- A name, called *cacher_name*, that names the cacher to use.

Figure 8 shows the configuration after an object with the caching subcontract is cached by a local cacher domain.

The *cached_object* in the caching subcontract representation is used when an invocation occurs on an object that uses the caching subcontract. If the *cached_object* is non-null, then the invocation is done on the *cached_object*; if the *cached_object* is null, then the invocation is done on the server's handle. The *cached_object* will be null if there is no cacher or the server is on the local machine.

The *cached_object* is obtained when an object is unmarshaled into a client domain; this is done using the *cacher_name*. Each cacher domain (such as the CFS) implements a *cacher* object. This object provides the method *get_cached_obj* that takes an object implemented by a remote server and returns an object implemented by the cacher domain. This cacher object is bound in the local machine's name space under a name that must be agreed upon by the implementor of the cacheable service and the implementors of cacher domains for the service. This is the name that is stored as the *cacher_name* in the subcontract representation. This name is put there by the server domain that created the cacheable object.

When an object is unmarshaled into a client domain the unmarshaling code resolves the *cacher_name* to a cacher object implemented by a cacher domain. The unmarshaling code then invokes the *get_cached_obj* method on the cacher object passing it in a copy of the object. When the cacher domain receives the object, it creates a new object that it implements and returns this new object to the client domain. The object returned from the cacher is stored as the *cached_object* in the subcontract representation.

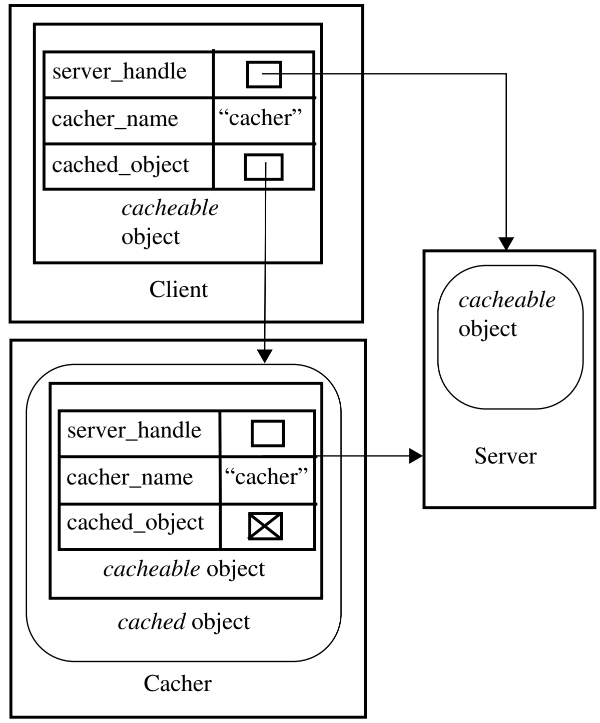

FIGURE 8. State after object is cached.

The client has a cacheable object that uses the caching subcontract. Its representation contains a handle that points to the server, the name of the cacher, and a *cached_object* that is implemented by the cacher. The cacher has a copy of the cacheable object but the *cached_object* is null since this is the cacher.

7.3 The CFS Cacher Object

The CFS implements a cacher object that it exports in a public place under a known name. Whenever a storage file server creates a file object, it sets the cached name in the file object's representation to be the name of the CFS. Thus when a file object is unmarshaled, the CFS's cacher object will be found and the *get_cached_obj* method will be invoked on the cacher object. The CFS will then return a file object that it implements.

When the CFS receives a *get_cached_obj* call it must determine two things. First, it has to determine if it implements the *cached_object;* if so it just returns it. Second, it has to find the internal cache state for the file; this is done by using the same bind protocol that is used by VMM to set up the cache object – pager object connection. The details of the bind between the CFS and the storage file servers is given in the next section.

7.4 CFS to Remote File Server Connection

The CFS and remote file servers need a connection similar to the connection between the VMM and pagers. The CFS needs to be able to get cached information for files and the remote file server needs to perform call-backs for cache coherency. This connection consists of two objects: an *fs_cache* object and an *fs_pager* object. The fs_cache object is a subclass of the VM cache object and is implemented by the CFS. The fs_pager object is a subclass of the VM pager object and is implemented by the storage file servers. These objects are subclasses of the VM objects for two reasons:

- They allow the normal bind method on a file object to be used to set up the connection and discover whether a file is already cached.

- They allow the storage file servers to keep data coherent while being ignorant of whether they are dealing with a VM system or a CFS – the file servers just use the VM cache object methods for data coherency.

The CFS–remote file server connection is set up using the same bind protocol described in Section 2.3.1—it just involves different objects. This protocol requires that the CFS implements a *cache manager* object that it exports under some name in the Spring name space.

The details of the *bind* operation between the CFS and a remote file server are the following:

- When the CFS is asked to cache a file object, it invokes the *bind* method on the file object; the CFS specifies the name that it bound its cache_manager object under as the name to the *bind* method.
- The remote file server checks to see if the CFS already caches the file.
- If not, the remote file server resolves the name given to bind to the CFS's cache_manager object.
- The file server then invokes the *create_cache_object* method on the cache_manager object passing in an fs_pager object.
- The CFS responds to the *create_cache_object* call by returning an fs_cache object and a list of <cache_rights object, encapsulated access> pairs to the remote file server.
- Once the remote file server has a list of cache_rights objects from the given CFS for the given file, the file server returns a cache_rights object that encapsulates the rights allowed on the file object.

Note that each time that the CFS is asked to cache a file it must invoke a bind on the file object. In the case where the CFS is not already caching the file, the remote file server will be forced to ask the CFS to create a cache. Thus on a CFS cache hit there is one network call and on a cache miss there are two calls. In Section 8 we describe a mechanism for eliminating this one network call on each cache hit.

7.5 Caching Binds

One of the important jobs of the CFS is to cache the results of VM binds since they occur on every map call. The CFS implements binds in the following manner. When the *bind* method is invoked on one of the CFS's file objects, the CFS checks to see if the client has the requested rights. If so, the CFS then checks to see if it already has a VM cache object with the requested rights cached for the file object. If it does, then it can merely return an appropriate cache_rights object to the VMM thereby eliminating a network access.

When the CFS finds that it doesn't have an appropriate cache object to return to the VMM, it must get one. Before it can do this, it must tell the remote file server that the VMM is caching file data; the remote file server needs to know that a VMM is caching file data so it can tell whether a file is shared for the reasons discussed in Section 5.3 and because of the VM cache reclamation described in Section 9.1. The CFS talks to the file server by invoking the *cached_bind* method on the fs_pager object. The file server will perform an access check and then return the attributes for the file such as length and modify time, and flags indicating whether these attributes can be cached; this is done so that subsequent remote calls to get attributes can be avoided.

Once the CFS has contacted the remote file server, it calls the local VMM with the fs_pager object implemented by the storage file server to create a VM cache object. Once the CFS has a cache object, it keeps a copy of it and returns the appropriate cache_rights object to the caller of *bind.*

The configuration that results after a successful cached bind operation is shown in Figure 9. Note that the VMM has a direct pager connection to the remote file server and the remote file server's cache object is actually implemented by the CFS. Thus all cache coherency operations on the cache object will indirect through the CFS. This shouldn't significantly degrade performance since we are just adding one extra local call to two remote calls (the coherency call and the page-out operation) and all of the data is being transferred using the direct pager object connection.

7.6 Caching Read and Write

The CFS caches data for reads and writes on files by mapping the file that is being read or written into the CFS's own address space. Once the file is mapped, then the data can be copied into or out of the mapped region as appropriate. Since file mapping is used, all of the issues of data

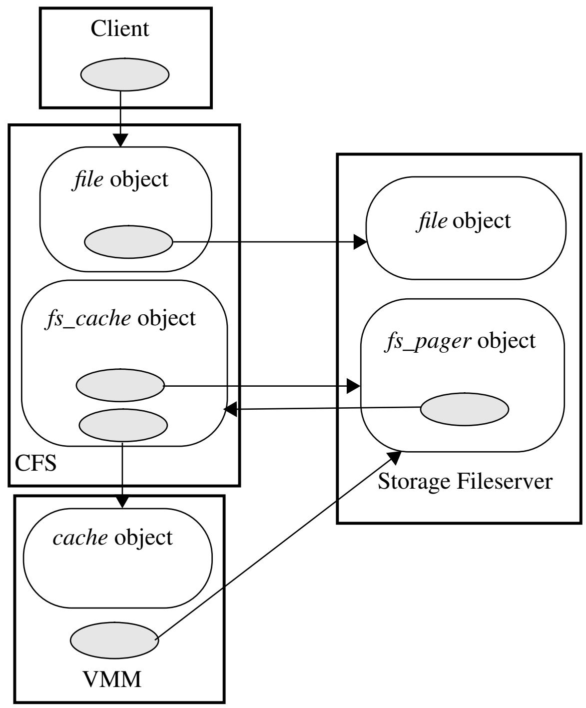

FIGURE 9. State after a cached bind

The client has a file object that is implemented by the CFS. The private caching channel between the CFS and the storage file server shown in Figure 5 is actually the *fs_cache* object–*fs_pager* object pair plus the *file* object. The storage file server's pager object actually has a *fs_cache* object implemented by the CFS instead of the VM cache object shown in Figure 7. However, the VMM still has a direct pager connection to the storage fileserver.

caching are handled by the virtual memory system and the remote file servers.

When the file being read or written is mapped by the CFS, the file object implemented by the CFS is used rather than the file object implemented by the remote file server. This is to take advantage of the bind caching that was described in the last section. This is important because the CFS only maps in a fixed-size region of the file so that the entire address space of the CFS is not occupied by a couple of big files. If a file is larger than this region, then it requires multiple map operations, and hence multiple bind operations, to access the entire file.

In order to implement read and write, the file length must be available locally. In particular, for writes that append data to a file, the CFS must have the ability to modify the length locally. Therefore the ability to cache the file length is essential to providing good read and write performance.

7.7 Caching Length

As we have mentioned, the file length is cached by the CFS. Caching the length is important for two reasons:

- It allows *get_length* and *set_length* operations to happen locally.
- It allows *read* and *write* operations to happen locally.

In order to let *set_length* and *write* operations happen locally, CFSs must have the ability to modify the length locally. As a result a length coherency algorithm is necessary. This coherency algorithm is similar to the page coherency algorithm that was mentioned earlier. The storage file servers allow multiple read-only cached copies of the length, but only one read-write copy. If a CFS requests a copy of the length and it is being cached read-write, then the remote file server retrieves the length from the caching client before giving a copy of the length to the requesting client. If a CFS requests permission to modify the length of a file, the remote file server invalidates the length from all caches after retrieving any cached read-write copy.

The file length also has to be retrieved by the storage file servers on page faults. Whenever a fault happens on a file object, the file server must know the current length of the file to determine if the page fault is legal. Thus, if on a page fault the length is being cached read-write, the file server will fetch the length back from the CFS and revoke write permission.

The storage file servers keep the length coherent by using the fs_cache object that is provided by the CFS. This object has operations to get the cached length and to invalidate the length.

Having the length cached read-write allows a CFS only to increase the length without informing the storage file server. A CFS still has to call through to the file server when a file is truncated for two reasons. First, the file server needs to take coherency action and flush the truncated data from all other caches; this reason isn't valid if the file is only cached by the CFS that is doing the truncation. Second, the file server will get confused if it isn't told about truncations. This confusion is best illustrated by an example. Assume that a CFS is caching a file which originally has length N and the CFS is allowed to modify the length. Now assume that the CFS truncates the file to length 0 without telling the remote file server and then sets the length back to N. If a page fault now occurs on the file, the remote file server will erroneously assume that the data on disk is valid since it doesn't know that the file has been truncated.

The length is retrieved by a CFS by invoking the *cached_-stat* method on the fs_pager object. The file server will return all of the file's attributes including an indication of whether the length and the time can be cached read-write. This is done so that subsequent remote calls to get attributes can be avoided.

7.8 Zero-filling Cache Objects

When a file is lengthened, all of the new pages between the old length and the new length will be read as zeros until the pages are modified. Instead of the remote file server zero-filling these pages on page faults, it would be much more efficient if the virtual memory system could zero-fill these pages itself thus avoiding a cross-machine call and a data transfer. This optimization is implemented by the CFS. If the CFS has the length cached exclusively and the length is increased, the CFS invokes the *zero_fill* method on the VM cache object. If the file object hasn't been bound yet, then the CFS will do the zero-fill after the object gets bound.

The storage file servers have to keep track of pages that are being zero-filled by virtual memory managers. Whenever a storage file server discovers that the length of the file has been extended by a CFS, it assumes that all new pages between the old length and the new length are being zero-filled by the VMM on the CFS's machine. A storage file server can discover that a CFS has lengthened a file in three ways:

- The length is retrieved for coherency purposes.
- The CFS gives the length back because it no longer is caching it (see Section 7.11).
- A page-out past the end-of-file occurs from a machine that has the length cached read-write. In this case the length is set to contain the last byte of the paged-out page.

In principle, this zero-filling optimization for file objects could be implemented by the storage file servers as well; that is, they could call the *zero_fill* method on the appropriate cache object. The only problem is that the storage file server may not know which cache object to do the zero filling on since a file object could be mapped by multiple VM systems. Of course in the normal case where a file object isn't being shared, then the storage file servers will know which VM to talk to. We haven't bothered to implement this on the storage file servers because most clients get their files through the CFS.

7.9 Caching Time

In addition to file length, we also allow both the access time and the modify time to be cached by a CFS. In addition we allow the modify time to be cached read-write under certain conditions. However, we only employ a coherency algorithm for the modify time – we don't bother to keep the access time coherent.

We don't bother to keep the access time coherent for two reasons. First, we don't believe that applications need a strongly coherent view of the access time; in fact we know of no application that requires a strongly coherent access time. Second, it is impossible to keep a cached access time coherent if the file is mapped in multiple caches since the data could be accessed at any time. Thus if we insisted on a coherent access time, it would require that *stat* operations on all shared mapped files, even read-only ones, are remote.

There are applications that would like at least some level of coherency for access time. One example is a file migration system where files that are not accessed after some time period are migrated to optical disk or tape. Right now we do not even implement weak access time coherency. However, it would be fairly simple to have the storage file servers periodically get the most recent access time from all cachers and flush all the cached notions of the access time. It is unclear how often this would need to be done.

The modify time is kept coherent. This is required so that programs such as **make** can function properly. A CFS is allowed to cache the modify time for a file under two circumstances:

- No one has the file cached read-write.

- The CFS is the only one that has the file cached read-write.

In the second case, the CFS is allowed to change the modify time. The modify time is kept coherent using the *dont_cache_time* method on the fs_cache object. This method is used when a storage file server determines that a file whose modify time is being cached can be modified by more than one VMM or CFS. In this case the storage file server invokes the *dont_cache_time* method on the fs_cache object of the CFS that is currently caching the modify time.

A CFS retrieves the time using the *cached_stat* method on an fs_pager object. When this method is invoked, one of the return parameters is an indication of whether the modify time can be cached and if so if it can be cached read-write.

7.10 Caching Other Attributes

We cache all attributes that are retrieved via the *stat* call. In fact when we do a bind or fetch the length or the time, we automatically return the other attributes as well. In the best case when attributes are not being modified on other nodes, the *stat* call can be handled with at most one network call to initialize the cache.

The one file attribute that isn't returned from the *stat* call is the access control list. This list is currently not cached by the CFS. If it turns out that this list is frequently accessed, we could allow ACL's to be cached and we could implement an ACL coherency algorithm using the fs_cache object/fs_pager object pair.

7.11 Releasing Cached Attributes

When a CFS decides to discard its cached attributes, it needs to write any modified ones back to the storage file server. This is done by invoking the *release_cache_info* call on the fs_pager object.

7.12 Data and Length Write Back Policy

Modified data is cached by the VMM for files that are cached by the CFS. If the machine that the data is cached on crashes, this data will be lost. The storage file servers employ a 30 second write back policy for writing back this cached data. In order to make the data even more secure, the CFS employs its own write back policy: when the last reference to a cached file object is gone, the CFS will write back all modified data for the file. Data is not written back for temporary or anonymous files.

Writing back the data is not sufficient—the length must be written back as well. As we discussed in Section 7.8, the storage file servers implicitly lengthen the file when page-outs past the end-of-file occur. Since page-outs are in page-size quantities, the file length is set to include the whole page. Thus the length has to be written back after the data is written back so the file server can know the true length of the file. When a storage file server gets the length from a CFS that is caching the length read-write, it will truncate the file to that length.

The data and the length are actually written back by a set of write-back daemons. When the last reference to a cached file goes away, the CFS will append a write-back request to the queue if the file is writable. If the length is also writable, then the write-back request will indicate that the length should also be written back after the data is written back.

7.13 Security

The CFS file server is trusted by client domains to cache their files. The CFS needs to ensure that it doesn't accidently allow some client to attain greater access to some cached file than the client is allowed. This is guaranteed by using the access rights obtained from the secure bind protocol described in Section 7.4. These access rights are checked on every operation on the file object implemented by the CFS to ensure that the client is allowed the desired access.

7.14 A Caching Swap Server

The CFS caches anonymous files. In the most straightforward implementation the CFS would rely on the caching subcontract to allow the files to be cached. Unfortunately the straightforward solution is not a good one for the following reasons:

- The VMM which gets the anonymous file objects from the storage file servers can't use the Spring naming system to resolve the cacher; there is no notion of naming in the kernel.

- The storage file server has the wrong cache object—it has the one implemented by the VMM instead of the fs_cache object implemented by the CFS.

For these reasons the CFS implements its own VM swap pager. This swap pager implementation uses a special swap_pager interface with the storage file servers. These file servers actually implement a subclass of the VM swap_pager class called *cacheable_swap_pager*. This class adds the additional method *get_anon_files* that allows the CFS and the storage file servers to set up the fs_cache object – fs_pager object connection for each anonymous memory object.

The CFS only registers its swap pager object with the VMM if there is no LFS. If there is an LFS, all swapping will happen to the LFS, so there is no need for the CFS to register its swap pager.

8 Using the Name Cacher

The implementation of caching that we have described so far requires a network access on each cache hit. Fortunately by allowing name caching to help us, we can eliminate this network access on name cache hits. Before we explain how name caching can help the CFS we have to explain some details of naming.

Objects that are stored in name servers are not unmarshaled by the name server for two reasons. First, a name server cannot expect to have the correct unmarshaling code for all objects. Second, the name server may not be willing to trust arbitrary unmarshaling code.

To allow objects to be passed to a name server and not be unmarshaled, the objects are enclosed in wrapper objects. The object inside a wrapper is not unmarshaled until the wrapper is returned to the client that did the resolve. In addition to an object, the wrapper also contains the object's type-id.

The Spring naming system uses name caching to provide good naming performance. In the best case, name resolutions can be handled without any network accesses. Name caching is done by *name cacher* domains. Name cachers, for the same reason as name servers, store only wrapped objects.

The implication that wrapped objects have for the CFS is the following. If a file object is resolved and stored in a name cache, the name cacher will not unmarshal the object; the name cacher will merely make a copy of the wrapped object and return it to the resolver. As a result the file object will not be cached by the CFS until it is unmarshaled by a client domain. Unfortunately each time that a client domain gets a file object from the name cacher, the client will have to contact the CFS, which will have to go through the bind protocol. Thus even on a name cache hit, one network access will be required to cache the object.

In order to allow cacher domains to avoid the bind call, the cacher object has a method in addition to the *get_cached_obj* method. The new method is *get_named_cached_obj* and is used in the following manner. When a name cacher gets an object to cache, it gets the object type-id out of the wrapper and tries to find a cacher object stored under the type-id. If it finds one, it passes in the wrapped object to the *get_named_cached_obj* method. This method returns a new wrapped object that the name cacher will return from cache hits.

The CFS uses the *get_named_cached_obj* method to avoid the bind call. The CFS stores a cacher object under the cacheable file object's type-id in the machine name server. When the CFS receives a wrapped object in the *get_named_cached_obj* call, it unwraps the object to get the file object stored inside. It then does the same thing as the implementation of the *get_cached_obj* described in Section 7.3. However, instead of just returning a file object it does the following:

- It marks the returned file object as inactive.
- It puts the object into a wrapper.
- It returns the wrapped object.

When the name cache returns an object to the client, the client will unmarshal the file object and invoke the *get_cached_obj* method on the cacher object implemented by the CFS. When the CFS gets the file object it will notice that it implements it and it is inactive. The CFS will then mark the object as active and return it to the client.

The notion of an object being active or inactive allows the CFS to know when the last active reference to a file is gone: file objects stored in the name cache are considered inactive and all other objects are considered active. The CFS needs to know when there are no more active references so that it can write back the file data and file length for the reasons described in Section 7.12. If we didn't consider objects in the name cache inactive, then name cached files would never be written back until the file was deleted from the name cache.

9 Cache Reclamation

9.1 VM Cache Reclamation

Spring VMMs maintain many data caches, including some that are not currently attached to any domain. At some point the VMM may determine that it has too many unattached caches and it wants to reclaim some of these caches. Since the main user of VM cache objects is the Spring File System, it is involved in this VM cache reclamation.

VM cache reclamation is complex because of a race condition: there is a race between a file server returning a cache object from a bind call and a VMM reclaiming the same cache before the bind call returns. We had to design a special protocol to deal with this situation. This protocol is described in [3] and all three file servers implement their part of the protocol.

When a VMM reclaims a cache it invokes the *done_with_pager_object* method on the pager object. This will go to a storage file server even if the file is cached by a CFS on the VMM's machine. When a storage file server gets a *done_with_pager_object* method invocation on a file that is cached by a CFS, it invokes the *delete_cache* method on the fs_cache object implemented by the CFS. This allows the CFS to destroy its cached copy of the VM cache object that is being reclaimed.

9.2 CFS Cache Reclamation

The CFS caches information for many files, including some that aren't actively being used by any client. When the CFS determines that it is caching too many inactive files, a background thread will begin reclaiming unused files until the number is acceptable. It notifies the storage file server that a cache is being reclaimed by invoking the *release_cache_info* method on the *fs_pager* object. Since the CFS uses the same bind protocol used by the VMM, the CFS cache reclamation protocol has the same race condition that VM cache reclamation has. We use the same solution that VM uses.

Files that are stored in a local name cache will be considered active for the purposes of cache reclamation. We have not had much experience with name caching so we don't know how many files this will force us to cache. If the file cache gets too large because of name caching, then we will need to implement a mechanism for reclaiming files that are stored in the name cache.

9.3 Storage File Server Cache Reclamation

The storage file servers also cache information for many files including some that aren't being actively used by clients. The storage file servers keep one cache entry for every file that has ever been opened by the file server and has not been removed. This cache is currently unbounded. Clearly a mechanism should be implemented that bounds the storage file server file caches.

10 Storage File Server Implementation

The Local and SOUP file servers are almost identical. The only major difference is in how the files on disk are accessed. We have already described most of the common implementation of these two file systems. In this section we will go into the details of the two implementations. We will first discuss the differences between the two file access mechanisms, then we will describe the details of the common implementation, and finally we will discuss the peculiarities of each implementation.

10.1 File Storage

The file format used by both file systems is identical: it is the SunOS UFS file format. The SOUP File Server (SFS) accesses files stored on disk via the UNIX file system: it just uses UNIX system calls. The Local File Server (LFS) accesses the files on disk directly. Thus the LFS has to understand the file system format and the SFS does not.

10.2 Common Implementation

10.2.1 File Object Type

As we described in Section 3, the file objects implemented by the storage file servers inherit the *memory object, authenticated,* and *io* interfaces. Since not all file objects

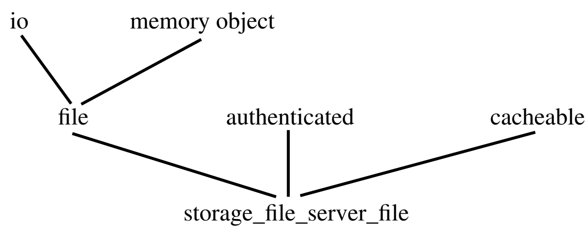

FIGURE 10. File type hierarchy

need to be authenticated objects, the standard Spring file interface does not inherit from *authenticated.* However, the file objects implemented by the storage file servers do inherit from the *authenticated* interface (see Figure 10 for the type inheritance graph for the file objects implemented by the storage file servers).

The file objects implemented by storage file servers also inherit from the *cacheable* interface and use the caching subcontract. Inheriting from the *cacheable* interface is needed because the caching subcontract assumes that it is caching objects of type cacheable for implementation reasons.

10.2.2 Context Object Type

As we described in Section 4.1, the fs_context objects implemented by the storage file servers inherit from the *context* and *authenticated* interfaces. Since not all fs_context objects need to be authenticated objects, the standard Spring fs_context interface does not inherit from the *authenticated* interface. However, the fs_context objects implemented by storage file servers do inherit from the *authenticated* interface. The fs_context objects implemented by storage file servers also inherit from the *cacheable* interface and use the caching subcontract so that these contexts can support name caching (see Figure 11 for the type inheritance graph for the fs_context objects implemented by the storage file servers).

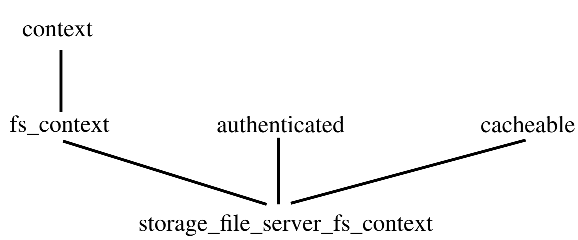

FIGURE 11. Context type hierarchy

10.2.3 UNIX System Call Interface

In order to allow the two file servers to share code, they need to have a common interface to perform operations such as read, write, and open. This common interface is the *unix_system_call* class. This class provides methods that correspond to each interesting UNIX system call. The SFS implements this class by mapping the calls directly onto the corresponding UNIX system call. The LFS implements the class by mapping each call onto internal routines that do the real work. Since the LFS uses the same file system format as UNIX, the mapping of the methods in the *unix_system_call* class to the actual implementation on Spring is straightforward.

10.2.4 Naming

The contexts exported by the file servers correspond to UNIX directories. Names are resolved relative to contexts exported by the file servers by invoking the open call on the *unix_system_call* object with the absolute pathname of the context plus the name being resolved. If the opened object is a directory, then an fs_context object is returned, otherwise a file object is returned.

The *create_file* method of the fs_context class maps onto the UNIX *open* system call with the create flag. Additional arguments to the *create_file* call include an ACL, a desired set of rights, and a set of flags. The ACL is currently ignored since UNIX files don't have ACLs, although we could try to map the ACL into the UNIX owner, group, and world access fields. The rights are checked and used to create the returned file object. The flags are used to get a read or write lock, to raise an error if the file already exists, and to indicate that the file should be truncated. The last flag maps directly onto the truncate flag to UNIX *open*.

The file servers only support binding of files and symbolic links; contexts are not supported because UNIX will not let anyone but superuser make a hard link to a directory. The file servers have the UNIX limitation that hard links cannot span file systems. Thus a bind of a file may fail if it would cause a link that spans file systems.

The *unbind* operation does not map directly onto the UNIX *unlink* or *rmdir* calls. In Spring, even after an object is unbound, it can be rebound under a different name. Therefore the file servers use the following rules for unbinding files:

- If a file isn't cached, then it is unlinked.

- Otherwise if a file isn't referenced by some client or the file has more than one link, then the file is unlinked.
- Otherwise, the file is renamed to the name *%SFS#* in the same directory; the # is an internal SFS identifier for the file.

Once a file that is unbound is no longer referenced by any client, the *%SFS#* file is unlinked. When a file that is unbound is bound to a name, the *%SFS#* file is renamed to the new bound name.

The file servers use the following rules for unbinding contexts:

- If the UNIX directory is empty, then the *rmdir* system call will be used.
- Otherwise, if the directory only contains %SFS# files, then these files are moved to the parent directory and the directory is removed.
- Otherwise, the directory will not be unbound.

The *get_all_bindings* method on a context maps onto the UNIX *getdents* call. The file servers will do a getdents and then create the appropriate binding list to return. This will include getting the attributes for each entry and creating an ACL from the UNIX owner, group, and world fields.

10.2.5 Access Control Lists

ACLs on file and fs_context objects are created using the UNIX attributes returned from the UNIX *stat* call. These attributes contain the UNIX owner, the UNIX group, and a set of modes bits that define what access the owner, the group, and all others are allowed on the object. This information is used to construct an ACL. Each object returned from the file system has the appropriate ACL attached to it. In addition these ACLs are returned when information about name to object bindings is returned from the file system.

10.2.6 Anonymous Memory Objects

Each anonymous memory object is stored in a different file. However, these files are not created until the first page-out occurs to the anonymous memory object.

10.3 Common Data Structures

The LFS and the SFS use the same data structures to keep track of open files and cached data and attributes. This section describes these data structures.

10.3.1 File System State

Each file system has a single instance of a per-file system object (see Figure 12). This object implements both the Spring *fs* object methods and internal methods to locate per-file and per-context state when names are resolved to files or contexts. This object is of type *fs_body*.

The fs_body class maintains four hash tables:

- One that maps from file ids to the per-file structures described in Section 10.3.2.
- One that maps from context ids to the per-context structures described in Section 10.3.5.
- One that maps from file instance values and cache manager names to the per-cache structures described in Section 10.3.3.
- One that maps from file instance values and page numbers to the per-page structures described in Section 10.3.4.

When a name is resolved in a storage file server, the name is looked up in the UNIX directory structure. The result is a <dev, inode> pair. This is used as a unique ID for files and contexts. This ID is the key into the per-file and per-context hash tables (the other two hash tables are described in Section 10.3.2). Thus when a name is resolved the resulting ID is hashed in the appropriate hash table depending on the type of the object (file or context) and the result of the hash is a pointer to a per-file structure or a per-context structure. If there is no entry in the hash table, then a new data structure is created and entered into the table.

10.3.2 File State

Each file being cached by a storage file server is represented by one instance of a per-file data structure of type *cacheable_file_body* (see Figure 13). The file system hash table contains a reference to this data structure and each file object implemented by the storage file server contains a reference to the appropriate cacheable_file_body instance.

Each cacheable_file_body structure contains several fields including:

- A list of *cache_state* structures (see Section 10.3.3) with one entry for each VMM or CFS that is caching the file.

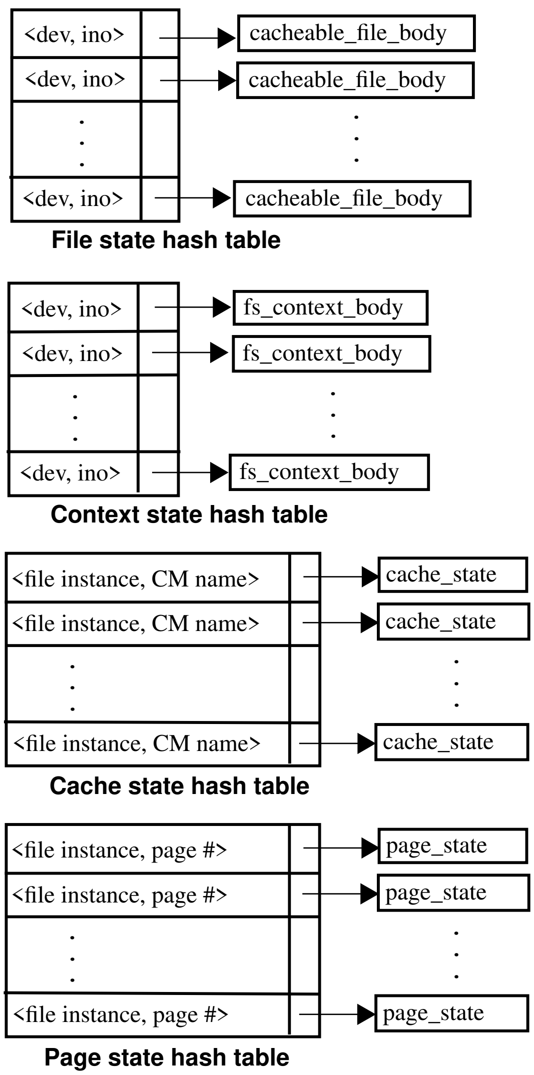

FIGURE 12. fs_body structure

Each file system maintains four hash tables. One maps from <dev, inode> pairs to cacheable_file_body structures. Another maps from <dev, inode> pairs to fs_context_body structures. Another maps from <cacheable_file_body instance value, cache manager name> pairs to cache_state structures. Finally one maps from <cacheable_file_body instance value, page number> pairs to page_state structures.

- A list of *page_state* structures (see Section 10.3.4) with one entry for each page that is cached by some VMM.
- Two bitmaps of in-use pages that are used instead of the page_state structures in the case when a file isn't shared.

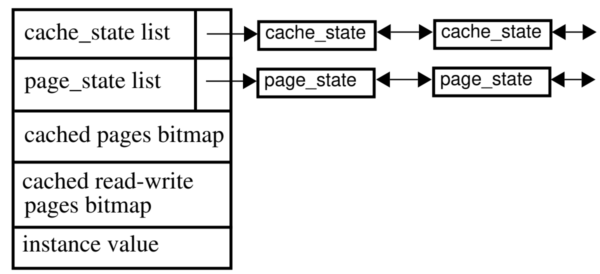

FIGURE 13. cacheable_file_body structure

Each cacheable_file_body structure contains a list of cache_state structures for caches that cache the file, a list of page_state structures for all of the cached pages, two bitmaps to keep track of cached pages, and a unique instance value for hashing purposes.

- A unique instance value for this cacheable_file_body structure that can be used for hashing.

In the normal case when a file is only being cached by one VMM, the two bitmaps are used instead of the page_state structures. Each bitmap has one bit for each page in the file. One bitmap keeps track of which pages are being cached by the VMM and the other keeps track of which pages are cached read-write by the VMM. The bitmaps are used when possible because they are more compact than the page_state structures.

The implementation of the cacheable_file_body class contains code to handle bind calls. This code uses the per-cache hash table that is maintained by the fs_body class. When a bind occurs the cache_manager name and the cacheable_file_body instance value is used as a key into the cache hash table. The instance value is required because this hash table is shared by all cacheable_file_body instances.

The cacheable_file_body implementation also uses the per-page hash table maintained by the fs_body class. This hash table is used to quickly locate information about cached pages. The key into this hash table is the page number and the cacheable_file_body instance number. The instance number is used because the hash table is shared by all cacheable_file_body instances.

10.3.3 Cache State

There is one instance of the *cache_state* structure for each VMM or CFS cache for a file (see Figure 14). Each fs_pager object implemented by the storage file server

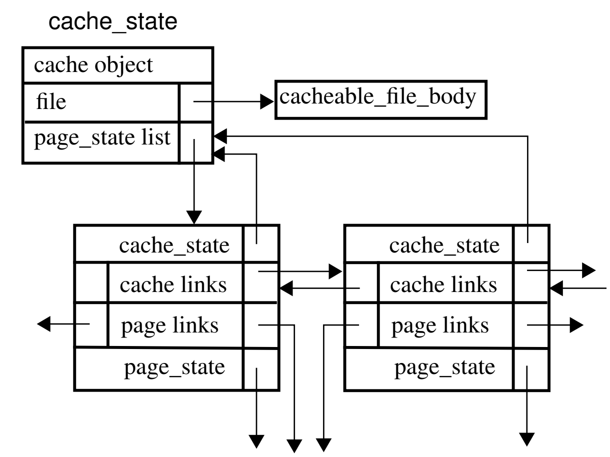

FIGURE 14. cache_state structure

Each cache_state structure contains a pointer to a list of pages that are used by the cache. Each element in this list appears both in the cache_state page list and in the appropriate page_state cache list. Each list element contains pointers to the cache _state and page_state structures whose list the element appears in and two sets of list links so that the element can appear simultaneously in both lists.

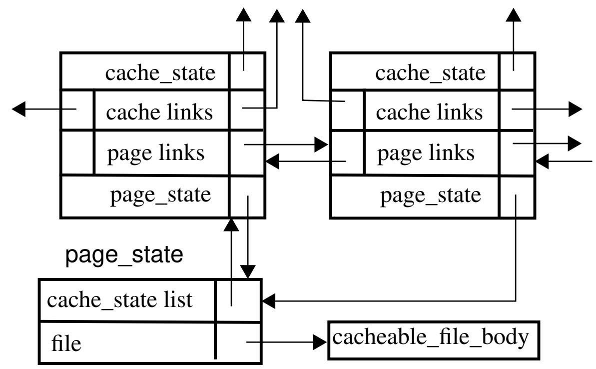

FIGURE 15. page_state structure

Each page_state structure contains a pointer to a list of caches that are caching the page. Each element in this list appears both in the page_state cache list and in the appropriate cache_state page list. Each list element contains pointers to the cache _state and page_state structures whose list the element appears in and two sets of list links so that the element can appear simultaneously in both lists.

contains a reference to the appropriate cache_state structure. The cache_state structure contains many fields including:

- A list of page_state structures for the pages that are being cached by this cache.
- The VMM or CFS cache objects needed for coherency of this cache.
- A pointer back to the cacheable_file_body structure.

10.3.4 Page State

There is one instance of the *page_state* structure for each page that is being cached by some VMM (see Figure 15). This structure contains:

- A list of cache_state structures for the caches that are caching this page.
- A pointer back to the *cacheable_file_body* structure.

10.3.5 Context State

Each context being cached by a storage file server is represented by one instance of a per-context data structure of type *fs_context_body*. The file system context hash table contains a reference to this data structure and each fs_context object implemented by the storage file server contains a reference to the appropriate fs_context_body instance.

This structure only contains simple bookkeeping information.

10.3.6 Front Objects

Each file, fs_context, and fs object implemented by a storage file server has a front object associated with it. Each front object encapsulates the access rights and possibly the principal of the object instance and contains a reference to the appropriate body object:

- *cacheable_file_front* objects point to a cacheable_file_body instance.
- *fs_context_front* objects point to a fs_context_body instance.
- *fs_front* objects point to an fs_body instance.

Each fs_pager object implemented by a storage file server also has a front object, but these objects don't need encapsulated access rights or encapsulated principals. The fs_pager object front object is of type *fs_pager_object_impl* and contains a pointer to a cache_state structure.

10.4 Local File Server

The LFS provides a generic implementation of a SunOS local file system. It caches control information but does not currently use the VMM to cache data for read and write operations. The LFS currently relies on the CFS to provide read and write operations that use VMM data caches for high performance[4].

10.5 SOUP File Server

The SOUP File Server implementation has some peculiarities because it is implemented on top of UNIX. In this section we will discuss these peculiarities.

10.6 Bounding the File Descriptors Used

UNIX has a limit on the number of file descriptors and hence the number of open files that a process can have. Since the SFS can cache more files than there are allowed file descriptors, the SFS must multiplex the file descriptors. This is done by keeping the UNIX file name as part of the state of each cached file and only opening files when necessary. Once a file is opened it will remain open until it has to be closed because the SFS has run out of file descriptors.

There is one other detail about unbinding not given in the previous section. Even if a UNIX file has more than one link, if the name that is being unbound is the same as the UNIX file name that the SFS uses to open the file, then the file will be renamed *%SFS#*. Otherwise the SFS would not be able to reopen the file later.

10.7 Data Caching

The SFS relies on UNIX to cache its data. All reads, writes, page-ins, and page-outs are done directly on the appropriate UNIX file object.

10.8 Coherency with Direct UNIX Accesses

The files cached by the SFS can be directly accessed by UNIX processes. This is commonly done when users rebuild programs on UNIX that are then run on Spring. The SFS cannot hope to keep the data coherent on each data access, but it does attempt to keep files coherent across resolutions. When a file is resolved that matches some already cached file, the SFS compares the UNIX modify time of the file to the time that the SFS believes the file was last modified. If they don't match, then the SFS will flush the cached copy from its cache, and put the new UNIX file in its cache. Clients that are still using the original cached version of the file can continue to do so. However, if the UNIX file was modified in place, clients who continue using it may be surprised.

This coherency support provided by the SFS isn't free. The SFS must *stat* the UNIX file after every write or truncate to get the UNIX view of the modified time. Thus all writes and truncates are slower to provide this level of coherency. This slowdown isn't significant on UNIX files that are stored on the SFS's machine but is very significant for files that are accessed via NFS. NFS appears not to be optimized to handle *stat* calls after writes. Evidently a *stat* call requires all data for the file to be synchronously written to disk because the *stat* after a write on an NFS file takes on the order of 100 milliseconds.

4. In general the CFS is only supposed to cache information for remote files. However, in our current implementation the CFS caches data for all files, even locally implemented ones. This issue is discussed in Section 13.

11 Performance

The Spring File System does extensive amounts of caching to provide high performance. In this section we will examine just how effective this caching could be and how effective it really is.

11.1 Potential Improvements

The caching by the CFS provides the ability for tremendous increases in performance. Table 1 gives two examples of sequences of operations that clients will do on files and how caching dramatically reduces network accesses. The first example is the use of a 1 Mbyte temporary file accessed via the read-write interface. This shows the effect of the data and length caching done by the CFS. In this example, when caching is used there is virtually no network activity; this file can be read and written as fast as the local file system can copy data. The second example shows the use of a 1 Mbyte file accessed via memory mapping. This shows the effect of the zero-fill optimization. With the zero-fill optimization and length caching, there are virtually no network operations.

11.2 Measured Improvements

In the previous section we discussed the maximum benefit from caching. Table 2 gives measurements of some com-

Operation	Network Operations Without Caching	Network Operations With Caching
Create file	1	1
Write 1 Mbyte	256	1 (bind)
Read 1 Mbyte	256	0
Remove	1	1
Total	**514**	**3**
Create file	1	1
Map file	1	1
Set length	1	0
Modify pages	256	0
Total	**259**	**2**

TABLE 1. Possible improvements with caching

The page size is assumed to be 4 Kbytes. The first example involves reading and writing a 1 Mbyte file in its entirety where each read and write transfers 4 Kbytes of data. Without caching, 256 network reads and 256 network writes are required. The second example involves accessing a 1 Mbyte file through the mapping interface. Without the zero-fill optimization 256 network page faults are required if all of the pages are touched.

mon file operations. The client machine is a SPARCstation 2. The operations without caching go to a SOUP File Server that is running on a SPARCstation 2. These measurements show that caching allows the operations to be executed at least five times faster than without caching. We obviously need to do much more extensive performance evaluation of the Spring File System including comparing its performance to other systems. However, for now it is encouraging that caching is very effective for these simple measurements.

12 Related Work

There have been many instances of file systems that employ caching. Examples are the Sprite File System [1], the Andrew File System [2], and NFS [4]. All three of these file systems provide some level of caching of both data and attributes and some level of coherency. However, none of them provide distributed shared memory (DSM), and they were all built as part of or on top of monolithic operating systems. As a result many of the issues

Operation	Without Caching	With Caching
read 4K	11 ms	1.9 ms
write 4K	51 ms	2.1 ms
set_offset	3.4 ms	0.11 ms
map/unmap	10.5 ms	2.1 ms

TABLE 2. Measured performance

addressed by the Spring File System, such as dealing with external pagers, the separation of naming from the file system, and dynamically locating a per-machine cacher, were not addressed by these file systems.

There have also been several instances of systems that provide DSM including [5], [6], and [7]. However, these systems also did not address the issues involved in a system like Spring.

To our knowledge the only system that has addressed the caching problems in a distributed modular system besides Spring is CHORUS [8]. The CHORUS system implements distributed shared memory by having one global coherency manager that interacts with a per-machine cache manager. Each access to a file object is indirected through the local cache manager by using a *coherent* capability. When a file object is created it contains the known port of the local cache manager.

The special coherent capability in CHORUS provides functionality similar to the subcontract mechanism in Spring. However, the Spring subcontract mechanism is more general since it allows the local cacher to not exist and the cacher is identified by a name instead of a specific port number.

The notions of length and attributes coherency are not mentioned in [8]. Other issues, such as binding to caches, naming, and cache reclamation, are not mentioned either. Thus although CHORUS has implemented something similar to the Spring File System, it is unclear if they have solved all of the hard problems solved by the Spring File System.

13 Current Status and Future Work

Everything described in this paper is implemented with two exceptions. First, the CFS caches files for all file servers, even local ones. This is because in Spring there is currently no way for the caching subcontract to determine if an object is implemented by a remote server or a local server. Once it is possible to make this determination, then the CFS can be made to no longer cache files implemented by the LFS.

If the CFS no longer caches files implemented by the LFS, then the LFS must be changed so that reads and writes don't go directly to the disk. This can be done by using the local VM system in the same manner that the CFS does read and write caching: the LFS will map the file being read or written into its own address space and then do copies from/to the mapped location.

The other unimplemented feature of the Spring File System is the full naming authentication checking described in Section 4.1. Currently when an operation is being performed on an fs_context object (e.g., resolve), the storage file servers do not check the permission of the encapsulated principal *P* as each component of a multi-component name is processed. The file servers instead only check the permission of *P* when an object is to be returned. The result is a file server could allow a given principal to resolve through many intermediate contexts to get to an object even though the intermediate contexts should be unreadable to that principal.

The reason that we have not implemented full naming authentication checking is that we let the underlying on-disk file system (the particular UFS implementation) handle multi-component resolves. In order to do the proper checking the storage file servers need to do lookups a component at a time. An alternative for the LFS is to modify the UFS lookup code to take in the principal that is doing the naming operation.

14 Conclusions

File caching is crucial to good system performance in a distributed environment. The Spring File System provides effective caching in an environment different than the previous environments where caching was implemented. The Spring caches not only provide good performance but they are fully coherent as well. The Spring File System demonstrates that caching can be as effective in a highly modular distributed system as it is in monolithic systems such as UNIX and Sprite.

15 References

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

[2] Howard, J.H. ET AL., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems* 6, 1 (February 1988), pp. 51-81.

[3] Khalidi, Y.A. and Nelson, M.N., "The Spring Virtual Memory System," Sun Microsystems Laboratories, Technical Report SMLI TR-93-009, February 1993.

[4] Sandberg, R. ET AL., "Design and Implementation of the Sun Network Filesystem," *Proceedings of the USENIX 1985 Summer Conference*, June 1985, 119-130.

[5] Li, K., *Shared Virtual Memory on a Loosely Coupled Multiprocessor*, Ph.D. Thesis, Yale University, 1986.

[6] Leach, P., Levine, P., Hamilton, J., and Stumpf, B., "The File System of an Integrated Local Network," *Proceedings of the 1985 ACM Computer Science Conference,* March 1985, 309-324.

[7] Ramachandran, U. and Khalidi, Y.A., "An Implementation of Distributed Shared Memory," *Software-Practice & Experience* 21, 5 (May 1991), pp. 443-464.

[8] Abrosimov, V., Armand, F. and Ortega, M.I., "A Distributed Consistency Server for the CHORUS system," *Proceedings of USENIX Symposium on Experiences with Distributed and Multiprocessor Systems,* March 1992, pp. 129-148.

[9] Acceta, M. ET AL, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the USENIX 1986 Summer Conference,* June 1986.