Performance of Cache Coherence in Stackable Filing*

John Heidemann Gerald Popek
University of California, Los Angeles

Abstract

Stackable design of filing systems constructs sophisticated services from multiple, independently developed layers. This approach has been advocated to address development problems from code re-use, to extensibility, to version management.

Individual layers of such a system often need to cache data to improve performance or provide desired functionality. When access to different layers is allowed, cache incoherencies can occur. Without a cache coherence solution, layer designers must either restrict layer access and flexibility or compromise the layered structure to avoid potential data corruption. The value of modular designs such as stacking can be questioned without a suitable solution to this problem.

This paper presents a general cache coherence architecture for stackable filing, including a standard approach to data identification as a key component to layered coherence protocols. We also present a detailed performance analysis of one implementation of stack cache-coherence, which suggests that very low overheads can be achieved in practice.

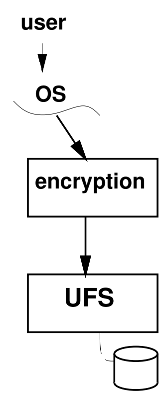

Figure 1: A sample application of the stackable layers. Each layer is connected by a standard interface.

1 Introduction

Stackable filing: Filing services are one of the most user-visible parts of the operating system, so it is not surprising that there are both many filing services suggested by operating systems researchers and a variety of third parties interested in providing these solutions. Of the many innovations which have appeared recently, very few of them have become widely available in a timely fashion. We believe this delay results from two deficiencies in the practice of current file system development. First, file systems are large and difficult to implement, and new filing systems often cannot take advantage of existing services. Second, file systems today are built around a few fixed interfaces which fail to accommodate change and evolution inherent in modern operating systems. Today's filing interfaces vary from system to system, and even between point releases of a single operating system. These differences imply that third parties cannot adapt filing interfaces to suit their own needs, nor can they expect their software to function as the base system evolves.

Stackable filing [7, 22, 8] presents a new approach to the construction of filing services to address these problems. Inspired by Streams [21], stackable filing is based on two key ideas. First, to allow rapid development of new services, stacking constructs filing services from *layers* which are combined into *stacks* to provide a complete filing environment. (We formalize these concepts in Section 2.) In this approach a new service is provided as a layer; it reuses existing services by stacking over them. Each layer is bounded above and below by a syntactically identical interface. This precisely defined interface allows layers to be provided in a binary form without source code. In spite of this "hands-off" design, changes to an existing service can often be accomplished easily by simply "pulling apart" any two layers and inserting a new module. Finally, since all layers meet the same interface, semantically equivalent layers can be swapped to improve performance or portability.

The second key idea in stacking is that the interface which bounds layers is *extensible*, allowing layers to be robust both to internal and external change. An extensible interface allows third parties to independently grow and adapt the filing interface to their needs. It also allows developers to incrementally evolve the base operating system without invalidating existing layers. Evidence suggests that the ability to construct and install binary modules that extend and alter filing services can lead to much more rapid evolution of available services and greatly increase reuse of existing service implementations.

An example of stackable filing is shown in Figure 1. A standard Unix file system (UFS) manages disk storage while a layer stacked above encrypts and decrypts data passing through it. One could imagine adding compression to the top of this stack as another layer. At UCLA we have constructed a number of services with stackable layers including replicated filing, user-id mapping, a persistent, object-oriented storage service, and prototypes of compression and encryption [7, 8]. All of these have been integrated into a full function filing service (SunOS 4.1.1).

*This work was sponsored by the Advanced Research Projects Agency under contract N00174-91-C-0107. Gerald Popek is also affiliated with Locus Computing Corporation. The authors can be reached at 3564 Boelter Hall, UCLA, Los Angeles, CA, 90095, or by electronic mail to johnh@ficus.cs.ucla.edu or popek@ficus.cs.ucla.edu. References to work related to this paper can be found at ⟨http://ficus-www.cs.ucla.edu/ficus-members/johnh/work.html⟩.

In UCLA stacking, each layer provides a potentially different view of the underlying data; a user can select different views by accessing a file through different layers. Access to different views is important to meet changing user needs and to provide for administrative services and sophisticated layer configurations. In Figure 1, for example, a user might write a file through the encryption layer while a backup program archives the encrypted data directly from the UFS storage layer. A directory-union layer might present several underlying directories as a single directory. The unified view would be most often used, but the underlying directories would be required for new software installation. Finally, we describe in Section 5.4 how internal access to different layers occurs in sophisticated filing services.

Stacking has been adopted in BSD 4.4 and the Spring operating system [12], and it has been employed extensively at UCLA to develop distributed filing services [7]. However, we argue below that suitable incorporation of multi-level cache management is essential to the success of modular systems.

File system caching: Caching can be used to improve performance in a system with stackable layers just as elsewhere: commonly used data is kept "on the side" by an upper layer to avoid repeating prior work. Stackable caching is particularly important for layers such as encryption and compression since the computation these layers perform is relatively expensive.

In addition to caching as a performance optimization, caching is also a required filing service in modern operating systems. Many systems employ an integrated file system cache and virtual memory system; such systems require caching to implement program execution.

For these reasons caching is a required part of any modern filing environment, and we expect caching to be important in file systems constructed from stackable layers. As described above and in Section 5.4, data can be accessed and cached at multiple layers of a single stack. Yet data caches in multiple layers raise several questions. How can these caches be kept coordinated? If layers are provided by different parties, how can they cooperate to provide coherence? Consider Figure 1. Both layers are likely to cache pages. However, when the same data is cached in both the encryption and UFS layers, updates to one cache must be coordinated with the other cache, or reads can return stale data and multiple updates can lose data. Some form of *cache coherence* is required. These problems are not issues in a monolithic file system where there is only one file system and one cache[1].

Thus far we have presented the problem of file data coherence in a multi-layer caching system. File system data is only one aspect of file system state which requires consistency guarantees. The more general problem is that many assertions easy to make in a monolithic system become difficult or impossible to make when state is distributed across several layers of a file system stack. Several such assertions are important in file systems: file data coherence, file attribute (meta-data) coherence, name lookup cache coherence, consistency of user-level locking, and internal concurrency control. This paper presents a system capable of addressing all of these areas.

Therefore, to summarize the focus of this paper:

1. file system stacking, if feasible in practice, would be very attractive;
2. practical stacking often requires concurrent access to multiple points in the stack;
3. various stack layers must cache information of different sorts in order to provide satisfactory performance;
4. those intra-layer caches must be kept coherent, or the accesses implied in the second point above can give incorrect results; and
5. a general framework for cache coherence is needed, since no individual third-party layer can solve the problem alone.

That is, cache coherence is essential to allow stacking to reach its full potential. This paper provides a modular solution to this problem.

Related work and directions for this paper: Our work builds upon two areas of prior research. First, we draw cache coherence algorithms from research in the areas of hardware multiprocessing, distributed filing, and distributed shared memory. Second, we build upon stacking work done at Sun Microsystems [22] and UCLA [7, 8], and cache-coherent stacking work also done at Sun [12]. A complete discussion of related work follows in Section 6.

This paper contributes to the architecture of cache coherence in a stacking system. We refine the notion of separation of the manager and the provider of cached objects introduced in Spring [12]. We show that consistent *identification* of cached objects is an important component of a coherence solution and can simplify the burden cache coherence places on layers. There are also several important structural characteristics to this work, principally a design that requires minimal change to common virtual memory architectures, and the freedom to access intermediate layers directly. We believe that these contributions are essential to more completely exploit the capabilities of stacking.

We also present a detailed performance analysis of our system. Conflicting costs of the framework and benefits of caching make careful performance analysis important. An understanding of the performance trade-offs in cache coherence is critical to the application of these results to other systems.

For concreteness we have focused our efforts in cache coherence on file system stacking. With the widespread deployment of object-oriented software development, many large-scale software systems are structured similarly to file system stacks. To the extent that caching is important at multiple levels of such systems, cache coherence also will be important, and the techniques employed in this paper may also be relevant.

2 Overview of UCLA Stacking

Our work on cache coherence takes place in the context of the UCLA stackable filing environment [8]. This system has been developed at UCLA since 1990, and portions of it have more recently been incorporated into 4.4BSD Unix. The research environment at the UCLA Ficus project has been hosted under a replicated file system built with stacking since 1991.

To provide a uniform vocabulary for the remainder of the paper, we now briefly summarize the original vnode interface and the UCLA stackable filing framework. (Detailed descriptions of the vnode interface [14] and UCLA stacking [8, 9] are available elsewhere.)

The vnode interface: The vnode interface separates the upper-level kernel from different file system implementations in an object-oriented manner. The upper-level kernel treats files as nearly opaque

[1] Some user-level systems (such as stdio) do caching. Such packages and caching are typically avoided when cache coherence is needed. This alternative are not possible when the services of a filing layer are required.

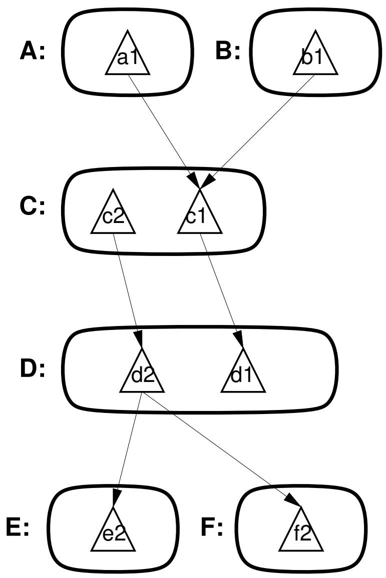

Figure 2: A configuration of several layers. The ovals represent layers; the figure as a whole represents a stack. Each triangle is a vnode, while each collection of joined triangles represents a file.

objects, *vnodes*. Actions on files are invoked via *vnode operations* which match the desired action with the a particular implementation at run-time.

At UCLA [8] and Sun [22, 25, 12] the vnode interface has been extended to support file system stacking, extensibility, and distributed filing. We next discuss each component of stackable filing at UCLA, drawing on Figure 2 for illustration.

Layer configuration: Each filing service is provided as a *layer*. Layers can be distributed as binary-only modules by third-parties and are configured into the kernel at system startup[2].

Layer instantiation: Layers are an abstract facility provided by the kernel. Before a layer is used it must be *instantiated* in a running system. Instantiation attaches a layer to a part of the file system namespace; all user actions in this part of the file system will be forwarded to the layer. (In our system, layer instantiation is done with the mount system call.)

Layers take layer-specific configuration information when instantiated. These parameters provide any additional information needed for layer execution, typically the name of the layer to stack upon. For example, configuration of layer D in Figure 2 would specify layers E and F. This approach is analogous to the specification of disk partition identity when configuring a physical file system.

More commonly, layers will rely upon other layers for lower-level resources. We describe such a combination of layers as a *stack*. By combining the services of several existing layers, stacks can form sophisticated filing services.

Each layer of a stack has a different place in the file system namespace, so different layers of the stack may be referenced at the same time.

[2] In principle, layers can be dynamically loaded into the kernel, but this is not supported by our current implementation. Regardless of how layers are loaded, new layers can be instantiated at any time.

Figure 2 is a single stack composed of six layer-instantiations labeled A through F.

Files in a layer: Stacks and layers are units of file system configuration. Translation of a path-name that enters the namespace under control of a layer creates a *vnode* to represent the state of that layer. The vnode represents its layer's "view" of the file. Just as layers build on other layers to form stacks, vnodes may build on vnodes from lower layers. Taken together these vnodes represent a *file*.

Each triangle in Figure 2 represents a vnode. Vnodes a1, b1, c1, and d1 combine to form file 1, while c2, d2, e2, and f2 form file 2.

This figure also shows several different configurations. Vnode c1 has two vnodes stacked on top of it, called *fan-in*. Vnode d2 stacks over two vnodes and so exhibits *fan-out*. Fan-in and fan-out allow stacks to form a directed acyclic graph (DAG) but complicate the problem of cache management. As an example of the use of fan-in, layer A might be an on-disk caching layer while layer B provides remote access, both of which stack over layer C which provides the Unix-specific extensions to layer D, a CD-ROM file system. Fan-out is common when a replication layer stacks over two storage layers.

File information: As a user extracts information from a file, a vnode of that file may cache some of this information, typically to improve performance. There are several types of information a layer may cache, including file data pages, file attributes, and directory name lookups. Collectively, such information will be termed *cache-objects*.

In Figure 2, a user might write to file 2 through vnode c2. Initially the user's data might be cached with vnode c2. Later it would be written down the stack, through layer D and to E and F.

A cache-object is a layer's representation of some type of file information (data, attributes, etc.). Cache-objects representing logically the same data may be held in different vnodes of the same stack. For example, all vnodes of file 2 in Figure 2 might store the "file length attribute" cache-object for the file. Because there are potentially multiple copies of what is logically the same data, some mechanism must be provided to keep them synchronized. This paper describes a *cache coherence* strategy to insure that cache-objects of layers from different parties and in arbitrary configurations can remain synchronized.

3 Architecture

Cache management is more difficult in a layered system than in a monolithic system because state (cache contents and restrictions) previously concentrated in a single location is now distributed across several modules. Our approach to cache coherence is to unify this state in a centralized *cache manager*. The cache management service is known to all stack layers and records the caching behavior of different layers. If it detects caching requests that would violate existing coherence constraints, it revokes caching privileges as necessary to preserve coherence.

An example of a potential stack and cache manager configuration can be seen in Figure 3. When a request is made to cache an object and that request conflicts with existing usage, existing cache holders are required to flush their caches before the request is allowed to proceed. In this example the encryption layer might request the cache manager to grant it exclusive caching rights to object A. The cache manager knows this request conflicts with the outstanding UFS cache of A, and so it will require the UFS to flush its cache before continuing. If the encryption layer allowed shared ac-

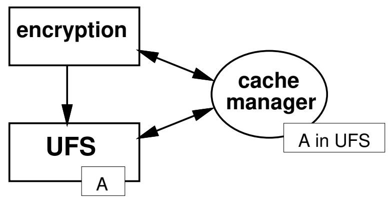

Figure 3: A sample application of the cache manager.

cess of A, the cache manager would verify that this request was compatible with the UFS's outstanding request (breaking this request if not) and then continue.

We next examine the design considerations which influence our approach to cache coherence. We then examine each of the subproblems facing our cache manager: identifying when different layers cache the same logical object, deciding if such concurrent cache requests are compatible, deadlock avoidance, and the relationship of caching and distributed filing.

3.1 Design constraints

Several constraints influence our choice and design of a solution. Good performance is the first constraint; support for the coherence framework should have little performance impact on an otherwise unaltered system.

To manage data, the cache manager must be able to identify it. A flexible and extensible identification scheme is a second requirement. Extensibility is critical because we already cache different kinds of data (names, file data, attributes); we anticipate caching other data and attribute types in the future. Flexible cache-object naming is also important because logically identical data-objects may be labeled differently in different layers. For example, "file data bytes 15–20" has a different meaning above and below a compression layer.

Additional design requirements include a strategy for deadlock avoidance (an important special case of stack-wide state) and the desire to make minimal changes to the virtual memory (VM) system. Several similar VM designs are widely available; the applicability of our work is maximized by focusing on the file system and its limited interactions with the VM rather than requiring significant changes to both systems. We comment as we proceed regarding the impact of these constraints on our design and implementation.

3.2 Data identification

To explore the services and level of generality required by a stackable cache management service, consider the analogy of identifying shared memory. In a simple shared-memory application where all processes share identical address spaces, data can be identified by its offset from the beginning of memory. A more sophisticated shared-memory application might allow independent processes on the same host to share memory by adding a second level of naming. Processes identify shared data with a memory segment name and shared data as offsets in that segment. More general still is a distributed shared memory system where host identification must be added to segment and byte names. A common characteristic of all of these examples is that all active agents (threads or processes) ultimately refer to the same thing: a particular byte of memory. Increasing generality of agents requires more sophisticated addressing, but fundamentally the problem is still the same.

The problem of data identification becomes more difficult with a general stacking model. Stack layers can arbitrarily change the semantics of the data representation above and below the layer. For example, layers may choose to rename data obtained from below, or may dynamically compute new data. Because new filing layers can be configured into the system dynamically, the scope of data change cannot be predicted until run-time. Data must be kept coherent in spite of these difficulties.

Our cache manager design addresses this problem in a manner analogous to how DSM addressing was identified: layers use more sophisticated identification as increasing generality is required. With the goal to "make simple things simple and complex things possible", the cache manager provides significant support for the common case where layers do not change naming of cachable objects. Layers with more sophisticated needs are allowed complete control over caching behavior. We examine each of these cases below.

3.2.1 Cache-object naming: simple layers

Layers cache several kinds of cache-objects, so a first component of cache-object identification must distinguish different cache-objects held by a single vnode. To identify cache-objects the cache manager uses a cache-object type and a type-specific name. Type-specific names are easily generated. (For example, each attribute or group of attributes is given a unique name, and file data bytes are identified by their location in the file. Section 4.2 discusses name selection in more detail.) Figure 4a shows how a single vnode might identify several cache-objects.

The cache manager can identify a cache-object held by a single vnode with specific names for each cache-object. The cache manager must be able to identify when cache-objects held by different vnodes alias one another. We solve this problem in two ways. The next section describes a solution for the general problem, but here we examine an important special case.

Often a layer assigns cache-object names in the same way as the layer it is stacked upon. We optimize our cache manager to support this kind of *simply-named* layer. Since information is identified the same way by each vnode of a simply-named file, the cache manager can automatically identify and avoid cache aliases if it can determine which vnodes belong to the same file.

The cache manager associates vnodes by tagging vnodes of the same simply-named file with a special token. The mapping ⟨ file-token, co-type, co-name ⟩ → vnode allows the cache manager to determine that ⟨ `file-2`, `attrs`, `length` ⟩ → `vp-c2` and ⟨ `file-2`, `attrs`, `length` ⟩ → `vp-d2` refer to the same object and must be kept coherent. In Figure 4b the cache manager has recorded both vnodes of a two-vnode file as caching the file length attribute.

3.2.2 Cache-object naming: general layers

Not all layers are simply-named. A layer that alters a cache-object in a way that changes its naming violates the simply-named restriction. Without help the cache manager cannot insure cache coherence above and below such a layer since it cannot anticipate how that layer alters cache-objects. For example, a file's length and the location of file data are altered by a compression layer in a layer-specific manor.

To solve this problem, generally-named layers must become involved in the cache coherence process. The cache manager supervises data above and below this layer as if there were two separate, simply-named files (each with a separate file-token). The generally-named layer is responsible for this division and knows about the two different "files". It informs the cache manager that it must see

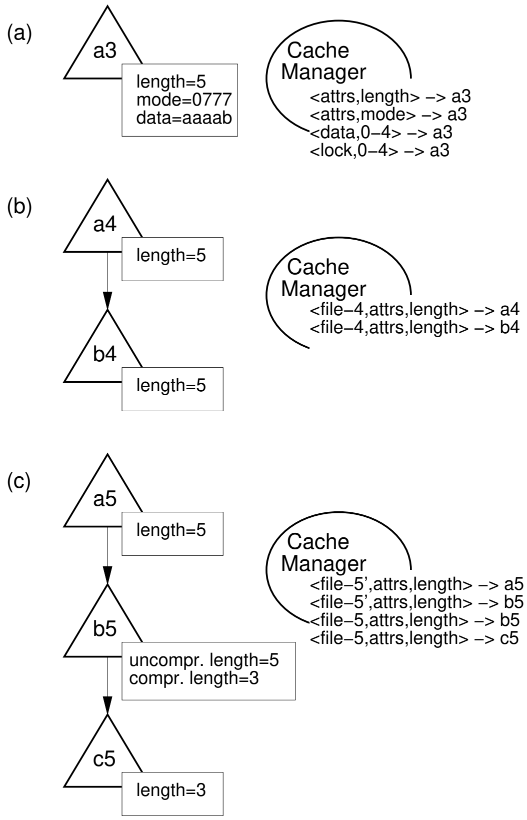

Figure 4: Levels of cache-object identification described in Section 3.2. In (a) a single vnode identifies cache-objects by type and name. In (b) a file-token is added. Part (c) shows how a general layer can map between different file tokens.

all caching events occurring in either simply-named file. That layer then relays and translates cache coherence events as necessary.

Figure 4c shows the general cache management case. Vnode b5 is cache-name-complex and divides the stack into simply-named files 5 and $5'$. The cache manager has a record for b5 with both of these simply-named file-tokens, allowing b5 to map any cache actions to the other side of the stack. The details of this mapping are dependent on b5's implementation. The details of one possible implementation are discussed in Section 4.4.

We provide cache coherence in two flavors to support simple layers with very little work while still providing a solution for the general case. For example, addition of coherent data page caching to a "null" layer (which uses simple naming) required only 70 lines of code, while support in a layer requiring general naming can easily be 5 to 10 times longer.

3.3 Cache-object status

A cache manager employs cache-object identification to track which layers cache what information. Tracking cache-objects allows the cache manager to implement a simple coherence policy by never allowing concurrent caching of the same object.

A better solution can be obtained if we employ knowledge of cache-object semantics to specify when cache-objects require exclusive access and when they can be safely cached in multiple layers. For example, some file attributes are immutable and so can be cached by multiple layers without penalty, other attributes change frequently enough to preclude caching, and an intermediate policy would be suitable for still others.

We require that a layer's cache request include not only what object is to be cached, but also its desired *status*. The status specifies if the layer intends to cache the object and whether other layers are allowed to concurrently cache it also. To handle a cache request the cache manager compares the incoming request against other outstanding cache requests, invalidating layers with conflicting requirements. If the new request indicates that the object is to be cached, the cache manager then records what layer will hold the data, promising to inform that layer if future actions require invalidation.

In addition to the standard cache-object requests, a layer can simply register interest in watching caching behavior for a given object. It will then be notified of all future cache actions. This facility is used to implement cache coherence across general layers.

3.4 Deadlock prevention

An operating system must either avoid or detect (and break) deadlock. In operating systems, deadlock avoidance is usually preferred to avoid the expense of deadlock detection and the difficulty of deadlock resolution.

Without cache coherence our style of stacking does not contribute to deadlock. Locks are not held across operations and since operations proceed only down the vnodes of a file, file vnodes form an implicit lock order. Cache coherence callbacks violate this implicit lock order; callbacks can be initiated by any vnode (in any stack layer) and can call any other vnode of that file.

To prevent deadlock from concurrent cache-management operations we protect the whole file with a single lock during cache manipulation. This approach has the disadvantage of preventing multiple concurrent caching operations on a single file, but in many environments that event is quite unlikely. In most cases cache operations are either already serialized by a pre-existing lock (such as during disk I/O) or can be processed rapidly (as with name lookup caching). Although a single lock works well in these environments, an aggressive multiprocessor system may wish to provide additional, finer granularity locking to reduce lock contention.

We guarantee deadlock avoidance by insuring a one-to-one association between stack locks and files. In Figure 4, for example, files 3, 4 and 5 each have a single lock, even though file 5 requires general naming. Run-time changes to stack configuration can violate this rule if a new layer with fan-out merges two existing files into a single new file. When this occurs the new layer must acquire both locks and then replace all references of the second lock with references to first.

3.5 Relationship to distributed computing

Cache coherence in stacking as described so far will keep all layers in a single operating system coherent[3]. Of course, shared filing is a useful service beyond the kernel of a single processor or small multiprocessor. Clusters of independent workstations and large-scale multiprocessors often have a shared filing environment among independent kernels and operating systems. Cache coherence on a single machine must not interfere with the overall distributed filing environment.

Cache coherence in a distributed system is subject to a wide range of latencies and degrees of autonomy. This range has prompted the

[3] Although we expect all layers to be cache coherent, layers which do not participate in coherence protocols are possible. Stacks involving such layers cannot make coherence guarantees.

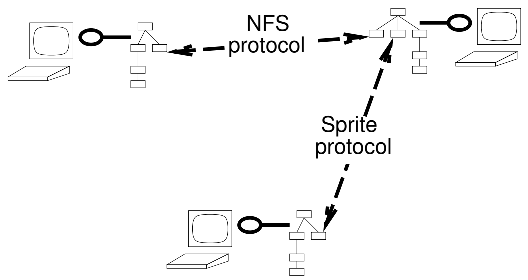

Figure 5: Distributed cache coherence involving different network protocols. Cache managers maintain coherence local to each machine while different protocols are employed for inter-machine coherence.

development of a number of different distributed file systems (for example, Locus, NFS, Sprite, AFS, and Ficus). Each of these file systems are designed for different environments and as a result have different internal coherence algorithms; the variety of solutions suggests that no single approach is best for all environments.

Cache coherence in stackable files on a single node of a distributed system must interact with the distributed filing coherence protocol, but we cannot require generalization of our protocol to the whole distributed system and successfully match all environments already served. Neither is it suitable to adopt different distributed filing semantics on a single machine where we can often provide a much better service. Instead, each particular distributed filing protocol interacts with the stackable coherence algorithms to maintain local consistency, but also communicates with its peers to provide its distributed policy. Figure 5 illustrates this concept. The cache manager at each site (the small ovals) maintains local coherence, while the layers implementing different distributed protocols (such as NFS or Sprite) implement their own coherence protocols independently. Distributed coherence and locking issues are thus the responsibility of the distributed filing protocol. Recognizing the variety of distributed protocols suggests that this "hands-off" distributed concurrency policy is the only one that will permit stacking to be widely employed.

4 Implementation

An implementation of this coherence framework is an important step in validating and evaluating the approach. This section briefly summarizes important points of our implementation, highlighting optimizations and other relevant implementation choices. We conclude by drawing the design and implementation together in an extended example.

4.1 Implementation overview

In general, a cache coherent stack behaves just as any other file system stack. A user invokes operations upon a layer, the operation passes down the stack and the results are returned back up the stack.

A layer may employ cached data to service a request. If the data already exists in the cache, that data is assumed to be coherent and the layer can use it. If the data is not in the cache, the layer will typically acquire the data and place it in the cache.

Before acquiring data to be cached a layer must gain *ownership* of that data. To acquire ownership a layer first locks the stack and then makes a cache-ownership call to the cache manager, providing its simply-named stack token, the identity of the cache-object it wishes to cache, and what restrictions it places on concurrent use of that cache object. The cache manager returns with a guarantee that the request has been met and the layer can acquire data without fear of coherence problems.

To make this guarantee the cache manager examines its records. If any other layers in the same simply-named stack have conflicting requests, the cache manager calls them back and asks them to relinquish their cached data. Other layers may have also registered "watch" interest in the stack to provide cache coherence between general layers. If so, the cache manager informs them of the incoming cache request, allowing them to translate and propagate the cache message throughout the whole stack.

When designing our cache manager we identified several kinds of cache-objects in need of coherence. We also realized that there would likely be other kinds of cache-objects in the future. To allow cache requests to be processed efficiently we apply three generic "classes" of cache-objects to several situations. The next sections discuss these classes and their application to actual cached data. In addition, Appendix A.2 presents the interfaces between the cache manager and a layer.

4.2 Cache-object classes

For efficiency we structured our implementation around three types of cached objects: whole files, named objects, and byte-ranges. We examine each of these classes briefly here; we apply them in the following section.

Whole-file identification: Successful use of stacking in a multiprocessing context requires coordination of multiple streams of control within a single file. Per-file locking provides an approach that can achieve this goal. Key design concerns are lightweight identification, support for arbitrarily complex stacks (since stacks can be DAGs), and careful attention to deadlock.

Whole-file identification is accomplished by recursively labeling the vnodes of the file. The lowest vnode in the file generates a unique token to identify that file. (In our implementation, the memory location of the vnode is used as a token[4].) As vnodes representing upper layers of the file are created, they inherit the identity of the vnodes they stack upon As each vnode making up the file is created it identifies itself as part of the same file as the vnode it stacks upon. (Fan-out vnodes which stack over multiple children employ general naming as described in Section 3.2.2.)

Whole-file identification solves a unique problem. More general services such as named-object and byte-range identifiers discussed in the following sections handle other stack identification needs.

Named-object identification: The fundamental service provided by the cache manager is maintenance of a central database of cache-object usage. Generic "names" of variable-length byte-strings provide a general way of object naming. The *named-object* subsystem implements this general model of cached object identification.

Named-objects are identified by the layer and a short string of bytes (the name). The cache manager uses these names to identify when layers of the same stack are caching related information. Services with a few objects may use fixed, pre-defined names; services that require more general naming might use application-specific names. Named-objects are suitable for file attribute (and extended attribute) cache management and name lookup validation. Details of name assignment for these applications follow in the next section.

[4] While suitable for our prototype, a better long-term implementation would use 32-bit counters to avoid name-reuse issues.

Byte-range Identification: *Byte-range* identification is a more specific scheme then named-objects. Byte-ranges support efficient association of caching information with specific areas in a file, identified as segments specified by file offset and length. Byte-range identification is suitable for user-level file locking and data cache coherence.

4.3 Application and optimizations

Our current system supports cache coherent file data, name-lookup caching, and attributes. Although application of byte-range or named-object cache management to each of these problems is relatively straightforward, several important optimizations are discussed below.

4.3.1 Data page caching

Our approach to data page caching is influenced by the observation that a sophisticated distributed shared memory system is *not* required to support inter-layer coherence. We adopt this view for two reasons. First, we expect most user action to be focused on one view of each file at a time and so concurrent sharing of a single file between layers will be rare. We explore the implications and the reasoning behind this assumption in Section 5.6. Second, we did not choose to provide stronger consistency than that provided by the filing system today. Multi-file consistency is left to the application, or to a separate layer.

An expected low rate of concurrent access to data pages implies that the simplest possible synchronization policy is warranted. We therefore protect each page with a single logical token and only allow a single layer to cache that page at any instant. (With byte-range identification we represent the logical tokens for contiguous pages efficiently.) When cache coherence requires pages to be flushed (because of potential cache incoherence) the current owning layer writes the pages to the bottom stack layer, insuring that future requests anywhere in the stack retrieve the most recent data.

Page flipping: A first optimization one might consider is moving pages between layers by changing page identification in the VM system. (In SunOS, each page is named and indexed by its vnode and file-offset. The most efficient way to move a page from one layer to another is to adjust this information.) For brevity we will term this optimization "page flipping". A key problem in page flipping is recognizing between which layers the page should be moved.

Consider the need to flip a page from vnode a1 to b1 in Figure 2. The minimal action required would be to move the page down the stack to vnode c1, the "greatest common layer" of a1 and b1, then back up to b1. Identification of the greatest common layer is difficult given the limited knowledge a layer has of the whole stack, particularly when non-linear stacks are considered. Our implementation therefore employs a simplification by approximating the greatest common layer with the bottom-most stack layer (vnode d1 in the figure). Stacks with fan-in will move the page to each bottom layer. With this optimization pages can move between layers without incurring any disk activity or data copying.

Page sharing: Allowing multiple layers to concurrently share the same physical page representation is a desirable optimization to avoid page thrashing and page duplication when two active layers have identical page contents. This optimization requires support from the VM system, like that provided by Spring [13]. Unfortunately, the SunOS 4.x VM system serving as our test-bed associates each page with a single vnode, and so we were unable to explore this optimization.

Read-only and read/write pages: Another possible optimization is to coordinate page access with reader/writer tokens instead of simple tokens. Reader/writer tokens allow multiple read-only copies of pages to exist in the stack concurrently. If pages are used primarily for read access, then this optimization avoids needless page flipping. We chose not to implement this optimization because of our expectation that concurrent data page sharing will be rare.

4.3.2 File attribute caching

File system layers often must alter their behavior based on file metadata. Current file systems may depend on file type or size; replicated file systems such as Ficus must know replica storage locations. Good performance often requires these sorts of attributes be cached in multiple filing layers, particularly when files are accessed remotely. Reliable behavior requires that such attributes be kept cache coherent. Our implementation of attribute cache coherence is therefore based on the assumption that multiple layers will need to cache attributes concurrently.

The cache manager handles coherent attributes as a class of named-objects. Groups of related attributes are each given a unique name when designed and are managed together. Because named-object cache management places no restrictions on the number of groups, this system extends easily to support file-specific attributes and potentially generic "extended attributes". There are many possible attribute-group naming schemes; we employ one modeled on a $\langle$ host-id, time-stamp $\rangle$ tuple to allow simple distributed allocation.

Our current implementation provides coherence for standard attributes; coherent Ficus extended attribute support is underway. Standard attributes are broken into three groups (frequently changing, occasionally changing, and unchanging) as an optimization to avoid unnecessary invalidation.

4.3.3 Directory name lookup caching

Pathname translation is one of the most frequently employed portions of the file system. The directory name lookup cache (DNLC) is a cache of directory and pathname component-to-object mappings which has been found to substantially improve file system performance. Cached name translations must be invalidated when the name is removed. In a multi-layer system the name may be cached in one layer and removed through another; a cache coherence system must insure that a removal in any layer invalidates any cached names in other layers.

A cache coherent DNLC must coordinate name caching and invalidation in several layers. Several approaches are possible to solve this problem. We considered merging the DNLC with our cache manager, but we rejected it for our research environment to keep our code cleanly separated from the remainder of the operating system. Instead we experimented with two different mappings between DNLC entries and the named-object cache manager. We first recorded all names and removals with the cache manager, directly using file names as cache-object names. This initial approach did not match typical DNLC usage (cache invalidations are rare) and so performance suffered. Our final approach tags directories that have any cached name translations; an invalidation in a tagged directory is sent to all layers. We found that occasional "broadcasts" prove more efficient than the bookkeeping necessary for more precise invalidation.

4.3.4 File data locks

User-level programs employ file-locking system calls to manage concurrency between independent user programs. For file locks to provide effective concurrency control they must apply to all stack layers, otherwise programs modifying a file through different layers could unwittingly interfere with each other. User-level file locking can be provided with the byte-range cache manager in a manner analogous to file data cache coherence[5].

4.3.5 Whole-file locking

Just as user-level programs employ locking for concurrency control, the kernel employs locking internally to keep file data structures consistent. Stacking requires serialization of access to stack-wide data structures as well as per-layer data. Whole-file locking provides this serialization.

We implement whole-file locking with a streamlined protocol separate from other forms of cache coherency. Stack-locking calls bracket other cache coherence mechanisms to avoid deadlock, so a separate protocol is required and minimal overhead is important.

4.4 An extended example

To bring together the design and implementation of cache coherence we next consider an example. We will examine stacks b and c in Figure 4 as data is cached.

Stack b represents the case of two layers with simple naming. Consider a user reading data from the top layer. Assuming the file's data structures do not already exist in memory, the pathname-translation operation is passed down the stack. As it returns up the stack, vnodes b4 and a4 are built. Creation of vnode b5 allocates a file-token, cache management structure, and lock for file 4, and vnode a4 uses this same information. Name-lookup caching may occur as a side effect of pathname translation; if so, one of the layers (typically the top) would register this fact with the cache manager of the parent directory of the file.

After the vnodes are created, a user reads from a4. Vnode a4 locks the stack and passes the read operation down the stack, specifying a4 as the caching vnode. The operation arrives at b4 (the bottom layer) which requests that a4 be given ownership of $\langle$ $4'$, `data`, 0–8k $\rangle$. The cache manager grants ownership of the entire file immediately (initially pages are unowned), and b4 reads the pages, placing them directly into a4's cache.

The stack in Figure 4c presents a more difficult case since general naming is required. Again, creation of c5 allocates cache management structures. Layer b is a compression layer which requires general naming, so it allocates a new file-token $5'$ to represent the "uncompressed file", and layer b registers "watch" interest in all caching occurring to layer $5'$. No new lock is created since each file must have only one lock. Finally, vnode a5 is created and returned.

Next assume that the user writes data into bytes 0–32k of our file through the top layer. Before the data can be written, a5 must acquire page ownership of $\langle$ $5'$, `data`, 0–32k $\rangle$. Vnode b5 watches caching operations to file-token $5'$, so the cache manager makes a callback and b5 translates this request and registers ownership of $\langle$ 5, `data`, 0–24k $\rangle$ (assuming 25% compression). Ownership is now assured and the read operation can take place.

To demonstrate cache interference, another user now will read the file back through vnode a5. Without cache coherence the results of this request are indeterminate. With coherence, a5 must register ownership of the data before the read. Currently b5 has ownership of part of file 5 so the cache manager calls back b5. Before b5 releases ownership of $\langle$ 5, `data`, 0–24k $\rangle$ it synchronizes $\langle$ $5'$, `data`, 0–32k $\rangle$. Vnode a5 owns this data, so the cache manager calls a5 to synchronize the pages; vnode a5 writes the pages, calling on b5 to compress them, ultimately delivering them to c5.

[5] Our current prototype does not yet implement cache-coherent, user-level locking.

(a) non-layered caching:

1. If data is in cache, use it.
2. Read data into the cache; use it.

(b) layered caching:

1. If data is in our layer's cache, use it.
2. Register ownership of data with the cache manager.
3. If registration conflicts with outstanding requests, revoke them.
4. If caching data-pages currently in another layer's cache, page-flip data into our layer and use it.
5. Read data into our layer's cache; use it.

Figure 6: Caching algorithms with and without layering. We use the layered caching algorithm in our system.

These examples present some of the most important details of our cache coherence protocol, both with simple- and general-naming.

5 Performance Evaluation

Performance evaluation of large software systems is difficult, and caching file systems is particularly difficult. When examining the performance of a cache coherence framework, particular care must be taken to separate the overhead of the framework from the benefits of caching. (The LADDIS NFS server benchmark, for example, carefully exercises NFS to gain useful measurements [27].) The next sections examine components of our coherence approach that impact performance, the benchmarks we use to examine that performance, and finally the performance of our system from several perspectives.

5.1 Performance components

A cache coherent, layered file system is composed of a number of cooperating components. Some of these components improve overall performance while others impose limits. (Of course, we expect better performance overall with caching than without.) This section examines the caching algorithms before and after our addition of cache coherence, with the goal of identifying which changes alter performance.

An abstract form of the algorithms used to access data through the cache is shown in Figure 6. Step 1 of both algorithms is the same, but the following steps differ and so may influence performance. Because Step 1 is identical, the cost of accessing already-cached data should not change. This fact is critical to overall performance, since a high cache hit rate significantly reduces average access time even if the cache miss penalty is also high.

Step 2, cache-object registration, is a new step and represents overhead of the cache coherence framework. The cost of this step is examined in Section 5.5.

Conflicting cache requests in Step 3 also represent a cost of cache coherence. This overhead is distinct from framework overhead, though, since it is a property of client usage patterns. We therefore characterize it as client overhead and examine it in Section 5.6.

Step 4 is an optimization to the basic cache coherence algorithm. For data pages the cost of servicing a cache miss is high (because they are large and require hardware interaction, see Section 4.3.1), so it is profitable to move cache-objects from layer to layer rather than regenerate them. The effects of this optimization are discussed in Section 5.6.

On the surface the last step is identical in the two algorithms; however their implementations differ. In a monolithic system, the same module generates and caches data. In a layered system one layer might generate the data, another may modify this data somehow, and a third may cache the data. An important aspect of the cost of layered caching is passing data between layers. For example, if data must be copied each time it moves between layers, bulk-data copy overhead would quickly limit layer usage. Such costs might not be present in a monolithic implementation where there is only one kind of buffering.

Typical vnode interfaces were not constructed with layered filing in mind; some aspects of their interfaces require excessive copying in a multi-layered filing environment. We have extended the interface to avoid this problem. We examine the implementation and performance costs of these changes and Step 5 in Section 5.3.

We have identified several differences between the layered and non-layered caching algorithms. We expect some of these differences not to significantly affect performance while others may improve or limit performance. After discussing our benchmarks and methodology, we will examine each difference with several experiments.

5.2 Performance experiments and methodology

Benchmarks: We examined our system with several sets of benchmarks. Our benchmarks can be divided into three groups. First are a set of benchmarks that operate recursively over a directory hierarchy. These benchmarks include recursive copy, find, find and grep, and remove. We selected this set because they intensively exercise the file system in different ways. Find accesses a large number of files without generating much caching. Copy accesses files and their data.

The second class of benchmarks is represented by the Modified Andrew Benchmark [18]. The Modified Andrew Benchmark consists of five phases: four brief file system operations and a large-program build. In our environment we found the first four phases too short to allow good statistical comparisons, and all were dominated by the compile phase. We therefore present only aggregate performance of all phases of this benchmark.

The final set of benchmarks is employed to measure cache interference. We describe them in Section 5.6.

Measurement times: We examined all benchmarks with two different measurement times: elapsed time and system time. Elapsed time represents the performance observed by a typical workstation user. System time represents only time spent in the kernel. Since all of our overhead is in the kernel, this measure exaggerates the impact of our changes.

Test environment: All tests were performed on a Sun SPARCstation IPC with 12 Mb of memory and a Sun 207 Mb hard disk with 16 msec average seek time. Our test machine runs a modified version of SunOS 4.1.1.

All data is stored in a stack-enabled version of the standard SunOS 4.1.1 file-system (UFS), a version of Berkeley's Fast File-system [16]. For multi-layer tests we add one or more null layers [8]. A null layer ordinarily passes all filing operations down the stack for processing; for these experiment we modified the null layer to cache file data pages internally.

5.3 Costs of layered data caching

The modularity enforced by a layered system limits information exported by a layer to that provided by its interface. A minimal, clear interface is both a benefit and a curse to a multi-layer system. A minimal interface simplifies multiple service implementations, but a minimal interface appropriate to a monolithic system may not admit efficient caching in a multi-layer system. Most current file system interfaces (for example, the SunOS and SVR4 vnode interfaces) do not provide the necessary services to allow efficient multi-layer caching.

One cost of caching in a layered system is therefore creation of new interface operations to allow efficient caching. This cost takes two forms: increased interface complexity and run-time overhead due to added code. We examine each of these issues below.

Implementation cost: Rather than engineer a completely new file system/virtual-memory system interface, we provided "stack-friendly" caching by minimal modifications to relevant existing vnode operations. The number of modifications required can be used as a measure of additional complexity required for efficient stackable caching. We currently cache three types of objects: attributes, file name translations, and data pages. Efficient caching of the first two of these objects is possible with no interface changes. Attribute manipulation already avoids unnecessary data copies, and name translation is internal to a each layer. Data page caching, the final case, was the only class of operations that required change. We next examine modifications required for this class of operations.

Data page caching required some interface changes to avoid repeated data copies. The caching operations (`putpage` and `getpage`) manage caching file data. The process of caching file data consists logically of two separate components; first data is read from stable storage, then it is placed in the VM cache. In a monolithic system such as the UFS, the same layer performs both of these operations. As first noted by the Spring project [12], successful layered caching benefits from a separation of these functions. In Spring terms, one object will serve as the *pager*, performing actual data I/O, while another object (the *cacher*) may be actively caching data. Our system restructures the file system paging operations to allow different layers to assume each of these functions. We modify three vnode operations (`putpage`, `getpage`, and `rdwr`) and their support code to accept vnodes representing both the cache and pager objects, rather than a single vnode representing both. The interfaces of these modified operation are listed in Appendix A.1.

Another operation requiring slight modification was the data read/write operation. Writing beyond the end of a file automatically extends the file length; our modifications keep file data and length information synchronized.

Our experiences modifying SunOS to support efficient data caching across multiple layers suggest that relatively few changes are required. The other relevant aspect of performance is the run-time cost of these changes, to which we now turn.

Performance cost: To investigate the performance cost of these changes we ran our benchmarks on kernels using standard and stack-friendly data acquisition. Neither case employed cache coherence; the measurement results are intended to evaluate the cost of the stack-friendly framework. Table 1 compares the standard Unix file system with and without these changes. Figure 7 presents these results graphically.

A comparison of individual benchmarks from these results shows a performance difference of $\pm 4\%$ for different tests, and that several of the tests show no statistically significant difference. Taken as a

	standard		stack-friendly		
benchmark	**mean**	**%RSD**	**mean**	**%RSD**	**% difference**
elapsed time:					
cp	159.0	12.67	154.2	12.18	–3.02
find	79.2	6.78	81.1	5.99	2.40
findgrep	205.2	5.90	197.7	1.22	–3.66
grep	61.6	3.60	61.9	1.68	0.487∗
rm	58.0	8.35	57.9	2.49	–0.172∗
mab	147.7	2.44	149.2	5.52	1.02
system time:					
cp	22.9	1.66	23.1	1.85	0.873∗
find	51.8	13.68	52.7	14.04	1.74∗
findgrep	102.4	1.38	101.5	1.58	–0.879
grep	19.2	1.97	20.1	2.41	4.69
rm	6.3	11.93	6.1	10.62	–3.17∗
mab	37.3	1.11	37.9	1.35	1.61

Table 1: Elapsed- and system-time performance comparisons of UFS performance with standard and stack-friendly cache operations. %RSD is σ_x/μ_x. Differences marked with an asterisk are less than the 90% confidence interval and so are not statistically significant. These values are derived from 25 sample runs. Section 5.3 interprets this data; Figure 7 presents it graphically.

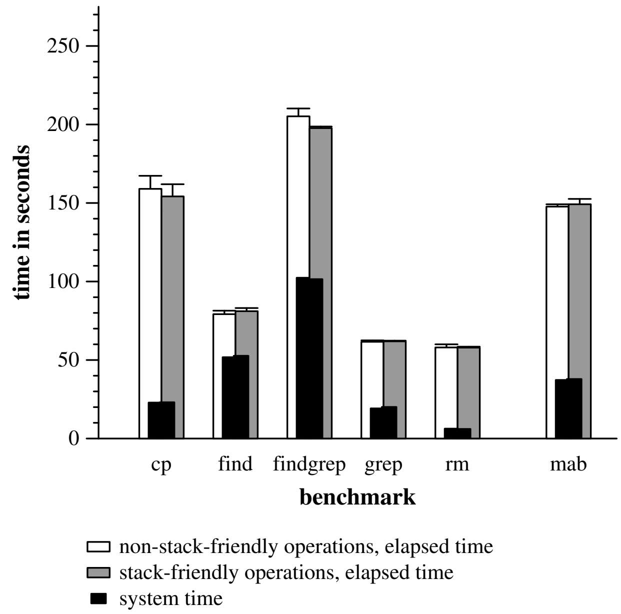

Figure 7: Benchmarks comparing a UFS with and without stack-friendly data acquisition. Error bars show one standard deviation. (This figure illustrates the data presented in Table 1.)

whole the tests suggest that there is some performance variation, but there is no consistent bias for either type of data acquisition.

5.4 Cache coherence benefits

Given operations that permit efficient caching in multiple layers, the next important issue is to examine what benefits cache coherence provides. The most important benefit is improved system reliability. Although instances of cache incoherence are usually rare, even occasional incoherence is not permissible in many critical applications. A related benefit is that cache coherence allows improved structure of multi-layer filing systems. File system implementations often require the ability to make assertions about data; without cache coherence these assertions are more difficult and often force a less modular structure. Finally, cache coherence and caching can improve system performance. We consider these benefits in turn, drawing on Ficus replication for illustrations.

The most important benefit of cache coherence is its support for correct system behavior. Without coherence unusual combinations of user activity can result in cache incoherence and incorrect results. Potential caching problems would force many developers to structure their systems in a less modular way, or prevent user access to lower layers. An example of this problem occurs in Ficus (see Figure 8 for the Ficus layer configuration). Ficus caches pathname translations in the selection layer (step 1). A file removal action by the remote user is directed to the physical layer on the local user's replica (steps 2 and 3). Without cache coherence the local user can still employ the cached name at the selection layer (step 4). With cache coherence, step 3 would have also removed the cached entry. Restructuring Ficus to avoid this problem would require that operations always pass through all layers, adding overhead and artificially distorting layer configuration. Although this problem occurs only occasionally in daily use, it would almost certainly require a solution should Ficus be deployed in a production setting. Moreover, fear of this sort of problem would curtail use of stacking as a structuring technique in many settings.

Another benefit of cache coherence is that by providing a rich environment within which correct behavior is easily achieved, layer development is made easier. One is also led into increased separation of function into separate layers, improving reusability. Two examples in Ficus illustrate how lack of coherence altered the desired system structure. First, without cache coherence Ficus cannot completely support memory-mapped data access. We work around this problem in several ways, but in a widely deployed system this problem may prevent the use of layering techniques. Second, the selection layer requires file attribute information when accessing a file. The overhead of an attribute fetch for each file access is significant (particularly if the file is remote), yet the selection layer could not cache attributes because its cache may have become invalid. Instead we were forced to build an ad hoc facility to work around the problem.

Performance is another important motivation for caching. Performance can be improved when the cache coherence service permits caching where it could otherwise not be used. The degree of performance change is highly application-dependent. For example, a software encryption layer which could not cache decrypted pages in memory would be unusable for executing programs (although it might be acceptable for logged output). To quantify the benefits of caching, we stacked three null layers over a UFS, simulating the layering overhead in the Ficus stack. We measured benchmark performance with and without name-lookup caching in the top null layer. Results were quite dependent on the pattern of use. In some cases, improvement was insignificant. Elapsed time of the copy case in fact showed a 10% increase; caching is of no benefit in a single-pass copy. In other cases overhead was cut up to 40%.

5.5 Cache coherence performance: no interference

We have suggested that there are both performance and structural advantages when layers employ cache coherence. Even when a layer experiences substantial overall speedup due to caching, there is still some overhead due to the cache coherence framework effort

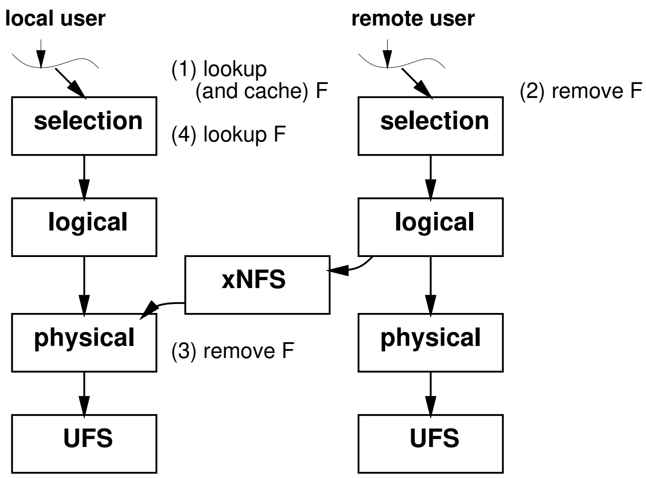

Figure 8: The Ficus layer configuration. Each column represents a particular host. The logical layer controls access to different replicas, accessing remote replicas through stack-enabled NFS. The sequence of operations listed results in cache coherence problems; see Section 5.4 for details.

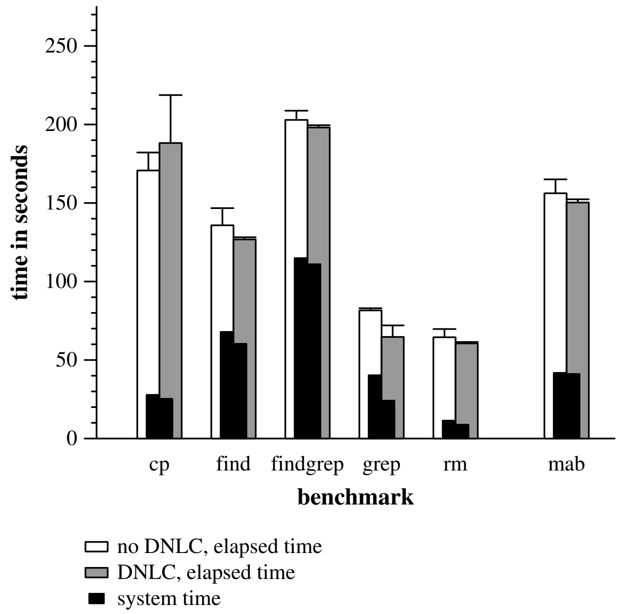

Figure 9: Benchmarks comparing three null layers stacked over a UFS with and without coherent name-lookup caching. (This figure illustrates the data presented in Table 2.)

	without DNLC		with DNLC		
benchmark	**mean**	**%RSD**	**mean**	**%RSD**	**% difference**
elapsed time:					
cp	170.7	8.03	188.2	19.40	10.3
find	135.8	9.63	126.8	1.31	–6.63
findgrep	202.9	3.46	198.1	0.87	–2.37
grep	81.6	2.03	64.7	13.59	–20.7
rm	64.5	9.71	60.6	1.75	–6.05
mab	156.2	6.79	150.3	1.63	–3.78
system time:					
cp	27.8	1.29	25.3	1.20	–8.99
find	67.9	15.11	60.3	2.17	–11.2
findgrep	114.9	2.40	111.0	0.75	–3.39
grep	40.3	0.80	24.1	33.24	–40.2
rm	11.4	6.77	8.8	6.94	–22.8
mab	41.8	1.23	41.1	0.93	–1.67∗

Table 2: Elapsed- and system-time performance comparisons of a stack of three null layers over a UFS without and with name-lookup caching. Differences marked with an asterisk are less than the 90% confidence interval and so are not statistically significant. These values are derived from 8 sample runs. Section 5.4 characterizes this data.

spent in step 2 of the layered caching algorithm (Figure 6b).

Measuring cache coherence framework overhead is crucial for several reasons. First, framework overhead can be used as a metric to select between different cache coherence implementations. Second and perhaps more importantly, framework overhead is required of all layers involved in cache coherence. Framework overhead therefore represents an additional cost applied to existing file system layers if they wish to participate in cache coherent stacks. Finally, cache coherence is an important component to a robust and general environment for stackable filing, so its performance is critical.

To investigate the cost of the framework alone, independent of any performance benefits of caching, we compare a layer with and without the cache coherence framework. Table 3 compares our disk-based file system (UFS) with and without the framework. Since only a single layer is employed in these tests all overhead observed is due to the framework as opposed to cache interference. Figure 10 reproduces these results graphically.

Cache coherence overhead on these benchmarks varies but is typically about 3–5%. Of the measured benchmarks, find exhibits the most overhead (15%) while findgrep and grep show the least cost (1–2%).

A 3–5% performance cost is not unreasonable when providing new functionality, but it is an unfortunate cost for existing services. This overhead represents the cost of setting up and maintaining cache coherence data structures. We expect that some of this cost can be avoided by internally preserving partially built data structures [2]. Careful tuning and examination of fast-path opportunities could also likely improve our prototype system; we project that a production quality service is quite feasible.

The cost of this overhead must also be weighed against the benefits of cache coherence. Caching in a multi-layer system can dramatically improve overall performance, often more than accounting for

benchmark	non-coherent mean	non-coherent %RSD	coherent mean	coherent %RSD	% difference
elapsed time:					
cp	228.1	12.81	218.9	17.61	–4.03
find	73.2	11.36	84.8	12.74	15.8
findgrep	212.0	2.19	216.7	2.14	2.22
grep	60.1	1.61	61.1	1.26	1.66
rm	73.4	1.62	79.8	17.89	8.72
mab	151.6	2.60	157.2	4.90	3.69
system time:					
cp	22.5	2.36	23.2	2.02	3.11
find	46.2	7.18	53.4	9.59	15.6
findgrep	98.3	1.61	103.1	1.54	4.89
grep	18.6	1.94	19.5	2.25	4.85
rm	6.2	14.99	6.3	11.70	1.61∗
mab	36.9	1.21	38.4	1.51	4.07

Table 3: Elapsed- and system-time performance comparisons of non-coherent and coherent caching kernels. Differences marked with an asterisk are less than the 90% confidence interval and so are not statistically significant. These values are derived from 30 sample runs. (The data in table is shown graphically in Figure 10.)

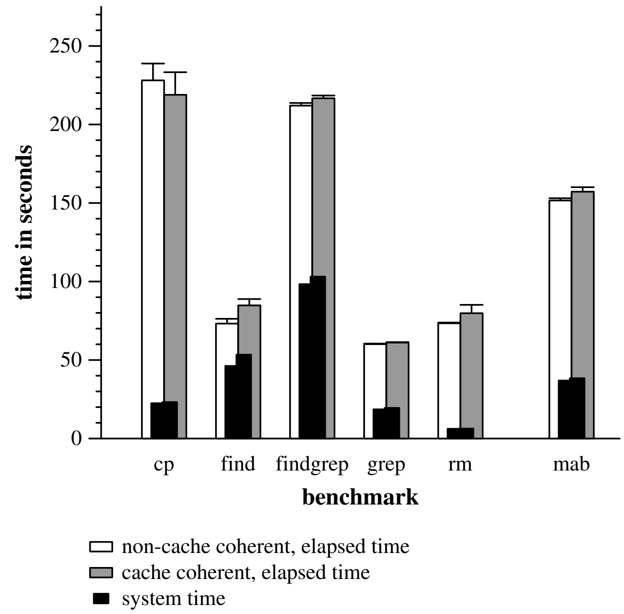

Figure 10: Benchmarks comparing a UFS in kernels with and without cache coherence. (This figure illustrates the data presented in Table 3.)

cache coherence overhead. In addition, cache coherence is an important part of providing a robust layered system by allowing layer designers to accommodate caching across all layers of a stack.

5.6 Cache coherence performance: interference

The experiments described thus far describe the performance of cache coherence when a stack is exercised with current styles of usage (all access through a single layer). Cache coherence is designed for a broader environment where access is possible through multiple layers. Shared access to the same data through different layers results in competition for caching this data. We next examine the effect this competition can have on performance.

Inter-layer cache interference is highly application-dependent and is not easily tested by standard benchmarks. We have therefore constructed two synthetic benchmarks to stress interference: sequential and random updates to potentially different file layers. We relate these benchmarks to practical applications below.

For each benchmark we stack one or two null layers on a UFS. Once layers are configured we map the file data into memory and exercise it according to the pseudo-code of Figure 11. Files are small enough to fit into physical memory, so all overhead measured is the effect of cache interference.

The results of these benchmarks appear in Table 4. Since the range of data is so great, some measurements are on the order of timer granularity (one-tenth of a second); in these cases measurement error is relatively high (10–14%).

We draw two conclusions from Table 4. First, the random-update benchmark shows that cache coherent access to multiple layers is extraordinarily expensive. Random updates exhibit more than 20 times greater elapsed time and 400 times greater system time when cache interference is present. This performance is a direct result of the lack of locality across multiple stack layers (*layer* locality) in a random access reference pattern. If this case were common, full function stacking would not be viable. However, we are not aware of any applications that exhibit this reference pattern; we discuss this problem in detail below. Furthermore, sequential file access presents a much different story: elapsed time is practically equivalent regardless of the degree of interference, although system time degrades by a factor of five.

Poor performance of the two-null-layer case with respect to the one-null-layer case is due to lack of layer locality. With one null layer the entire file is brought into memory and updates then happen without operating system intervention. With multiple null layers pages move between layers; each move requires a page fault which is several orders of magnitude more expensive than a direct memory reference.

We can analytically determine the number of page faults expected for each benchmark. Using file length and access conditions specified in Figure 11 and assuming a 4kbyte page size, any one-null-layer benchmark will page the entire file into the null layer with 250 faults. By comparison, the two-null-layer sequential benchmark will require 8000 faults to move the file between layers 32 times. In the two-null-layer random access benchmark each access has a 50% chance of requiring a fault[6]. In this case, where no layer locality is exhibited, randomly updating only four-tenths of the file results in 204,800 faults on average. (We have verified this figure, counting about 270,000 faults in a typical two-null-layer, random-update trial.)

These benchmarks suggest that, like virtual memory, locality is required for efficient use of cache coherence. With stacking, the reference stream must exhibit good *layer* locality to avoid cross-layer page faults. To interpret the results of these synthetic benchmarks in the context of real applications, we must characterize expected layer locality.

We have proposed file system layers as an approach to building rich filing services from composable layers. Currently (with the exception of direct disk access) filing environments export only one service; all user applications access this "top layer". We expect that

[6] In the n-active layer steady-state each access has a $1/n$ chance of a cache hit. First access to a page are not part of steady state; for our 1000k file with 4k pages these first 250 page accesses are not significant.

time	benchmark	one mean	one %RSD	two mean	two %RSD	%difference	90% confidence interval, % diff.
elapsed:	**random-update**	14.67	7.92	350.53	1.2	2289.90	9.44
	sequential-update	441.04	1.45	443.49	0.46	0.56	0.4
system:	**random-update**	0.72	10.02	289.48	0.87	40,291.00	134.17
	sequential-update	1.42	13.74	9.13	3.38	544.10	29.96

Table 4: Elapsed- and system-time performance comparisons of files with and without cache contention. The columns headed "one" show access through a single null layer stacked over a UFS; the columns headed "two" add a second null layer to this stack. Layer accesses are distributed across all null layers according to benchmark type. There is no contention with one null layer; contention is possible with multiple layers. These values are derived from 12 sample runs. These benchmarks exercise worst-case performance and are not representative of typical behavior; see Section 5.6 for discussion.

Random-update:
```
for i = 1 to random-scale(file-length)
begin
    layer = random(file-layers)
    offset = random(file-length)
    data[layer][offset]++
end
```

Sequential-update:
```
for i = 1 to sequential-scale(file-length)
begin
    layer = (i div file-length) mod file-layers
    offset = i mod file-length
    data[layer][offset]++
end
```

Constants:
```
random-scale(length) = (length / 10) * scale
sequential-scale(length) = (length * 8) * scale
length = 1,024,000 bytes
scale = 4
```

Figure 11: Benchmarks and parameters used to test cache interference for memory-mapped files.

a primary benefit of multi-layered filing will be to allow users to customize and extend their filing environment. Once configured, we believe that most user access will be to the "top layer" representing a particular configuration of a multi-layer stack. For example, most user access to the stack in Figure 1 would be to the cleartext provided through the encryption layer, not the encrypted-text presented by the UFS. No interference would occur in the common case of two programs reading (or memory mapping) a file through the same layer. When all user access occurs through a particular layer, no cache interference occurs and we expect performance results equivalent to the one-null-layer case.

Nevertheless, although most applications access a single layer, we have identified several cases where multi-layer access is important. In these cases, access to multiple layers may cause cache interference. Continuing our example, the user in Figure 1 may wish to transmit the encrypted-text of a file, and so after updating the file via the encryption layer, the user would read the file directly from the UFS. As in this example, we expect that the majority of such access will be sequential. Floyd's studies of Unix applications in an academic environment suggest that 70–90% of opened files are read sequentially [4]. For these cases, the sequential-update benchmark is representative. Sequential-update performance shows some system-time performance cost, but no noticeable elapsed-time performance penalty.

The remaining random access case is exemplified by database applications. Recall, however, that the random-update benchmark is a stream of randomly located updates to random layers. We do not expect a single database application would need to access multiple layers of the same file concurrently, or that two independent databases would access the same file concurrently through different layers, so this synthetic, worst case seems unlikely to occur in practice.

We selected these benchmarks to push the bounds of our system, and their worst-case results show significant overhead. Fortunately, we believe that they also suggest that practical applications will not suffer significant performance degradation with expected patterns of layer locality.

5.7 Performance experiences

Cache coherence in stackable filing is important to manage cache coherence problems that can arise from access to different stack layers. Both multi-layer access and caching are required in many practical layering systems. Administrative programs and sophisticated stack configurations require access to different stack layers, while caching is required for good performance.

Our performance experiments suggest a layer actively caching data will experience about a 3–5% overhead for typical benchmarks, although some may be higher or lower. Our use of stack-friendly cache access operations does not seem to be a significant portion of this cost. Instead we believe that the cost is primarily due to the maintenance of additional data structures and to comparison of our prototype implementation with carefully tuned file system code.

We also investigated system performance when different layers contend for the same cached objects. When applications that exhibit no locality compete for cached objects, significant overhead occurs. Common patterns of file usage and the expected uses of cache coherence suggest that typical applications will see minimal or no overhead due to contention.

We find a powerful analogy between virtual memory and cache coherence in stacking. The performance of both is strongly dependent on the locality exhibited by given applications; VM requires spatial locality while stack cache coherence requires "layer locality". Virtual memory frees many application designers from detailed concerns about memory management, often allowing applications to be more naturally structured. Similarly, stack cache coherence frees the designer from concerns about inter-layer consistency, providing a rich framework in which each layer truly can be independently developed and employed.

6 Related Work

Our work on cache coherence is based on several bodies of existing work including distributed filing, shared memory multiprocessing and distributed shared memory, and stackable layering. Each of these areas evolved slightly different solutions to cache coherence, but the central problem is determining *who* holds *what* data. We examine different applications from this perspective, categorizing how this information is stored and collected.

6.1 Distributed filing

Early distributed file systems such as Cedar and NFS avoid the problem of cache coherence by disallowing file mutation [24] and not providing strong coherence [23]. Locus provides strong coherence with a distributed token passing algorithm [20], while Sprite detects concurrent update at a central site and disables caching for coherence [17]. Later systems provide variations on the token algorithm: AFS's callbacks are essentially centrally-managed tokens [11]; Gray's leases are tokens that can time-out to simplify error recovery [6].

Cache coherence in stacking borrows the basic coherence approach used in these systems. Unlike these systems, stacking faces the unique problem of data identification across different data representations.

6.2 Multiprocessors and distributed shared memory

As with distributed filing, early approaches to shared memory multiprocessing avoid multiple caches or do not provide strong coherence (Smith surveys such systems [26]). More sophisticated systems broadcast and multicast coherence information to some or all processors. The constraints of a hardware implementation limit the scale of these approaches.

In distributed shared memory systems software plays a larger role in coherence. Li proposes strong consistency with both centralized and distributed algorithms [15]. Recent work has focused on employing application-specific knowledge to relax the consistency model and obtain better performance [5, 3].

6.3 Stackable layering

Early work in joining layers with a symmetric interface developed in several contexts: the Unix shell [19], the Streams I/O system [21], and x-kernel network protocols [10]. Most data in these systems is transient, and so they do not address cache coherence problems[7].

Databases and file systems have both persistent data and a need for caching. The Genesis work in databases [1], and file system stacking work from UCLA [7, 8], Rosenthal and Skinner at SunSoft [22, 25] and the Spring project at Sun Laboratories [12] approach cache coherence in different ways.

The Genesis work focuses on modularity and stacking for databases. Cache coherence problems are avoided by not allowing multi-layer access.

Rosenthal identifies the problem of cache coherence in stackable filing [22]. His early system restricts access and caching to the top stack layer, avoiding coherence problems at the cost of prohibiting multi-layer access. Skinner's later work provides two kinds of stacking termed "interposition" and "composition" [25]. Access to multiple stack layers created with composition is allowed, but with interposition is not. Issues of cache coherence are mentioned but not addressed in Skinner's paper. Locking among interposition layers is provided with a single readers/writers lock; locking between composition layers is not discussed.

Spring is an operating system that closely ties the virtual memory and file systems to provide distributed shared memory [12]. Cache coherent file system stacking is a natural result of this architecture, and with it come two important results. First, they recognize that separation of the data provider and the data manager is necessary for efficient, layered caching. In Spring terminology this concept is the separation of the cacher and pager objects. Second, they recognize that general cache coherence can be provided if each layer acts recursively as cacher and pager objects for the layer it stacks upon. We build upon these results.

Our work differs from the Spring work in several respects. We see cache-object identification as the central problem in cache-coherent stacking. To aid the layer designer, we provide two approaches to object identification, a fast, simple one for the dominant case and a richer solution for the general case. Our cache manager handles all aspects of the simple case and can directly invalidate data in any layer. The Spring work only provides (in our terms) the general model, potentially placing additional burden on designers of new layers and raising performance questions.

A second difference is application of cache coherence to all aspects of filing. The Spring project discusses coherent sharing of data pages and some file attributes. They recommend use of Spring object-oriented inheritance to provide coherence for other file attributes. We instead provide a cache coherence framework suitable for file data pages, attributes, generic extended attributes, and name lookup caching. We expect that this framework will extend easily to accommodate future data types, for example file locks.

A third difference in our work and Spring is the degree of independence or integration between stacking and the rest of the system. Spring is a complete operating system. Its virtual memory system, distributed shared memory, and stackable filing share an integrated implementation. While such an approach may be attractive, it limits portability. We instead focus on stackable filing. We require few modifications of and limited interaction with the VM system. Our system is designed to function with drop-in file systems in a binary-only kernel distribution, and we are intentionally distinct from distributed filing. We believe that a more modular approach is essential to allow wider application.

A final important difference between our work and Spring is that of performance evaluation. Performance analysis of the Spring file system and file system layering has focused on the cost of layering and the benefits of caching. While it is clear that caching is of substantial benefit in Spring (as in many other systems), it is not clear what overhead is paid for cache coherence. Because our system has evolved to support cache coherence, we are able to present a "before-and-after" performance analysis of cache coherence.

7 Conclusion

Stackable filing has the potential to significantly simplify file-system development, allowing a substantially richer filing environment through third-party contribution. To make full use of the stackable model, developers must have control over caching and cache coherence; without cache coherence the layer structure can be impaired, performance can suffer, and incorrect results can occur.

Good performance, ease of layer construction, modularity with the rest of the operating system, and extensibility are a few of the requirements of a successful cache coherence system. We believe we have an architecture and an implementation that meets these criteria. We were pleased that these goals could be achieved by a modular "drop-in" design which has few interactions with the rest of the op-

[7] User data in networking traffic (at the TCP level) is transient and so is not suitable for caching. Routing information is often cached, both between hosts (IP routing and ARP translation), and between TCP/UDP and IP layers of some implementations. Occasional cache incoherence in these systems is either tolerated, or the cache is not considered authoritative and is verified before each use. These approaches do not generalize to filing environments where cached data is considered authoritative and employed without verification.

erating system and yet sacrifices neither function nor performance. With the architecture and performance analysis presented in this paper we are optimistic that file-system stacking can be more generally employed.

Acknowledgments

The authors thank the anonymous reviewers and Greg Minshall, our paper shepherd, whose comments significantly improved this paper. We also thank members of the Ficus project for reviews of early drafts of this paper.

References

[1] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, November 1988.

[2] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Conference Proceedings*, pages 87–98. USENIX, June 1994.

[3] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164. ACM, October 1991.

[4] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report TR-177, University of Rochester, March 1986.

[5] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26. IEEE, May 1990.

[6] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 202–210. ACM, December 1989.

[7] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[8] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994. Preliminary version available as UCLA technical report CSD-930019.

[9] John Shelby Heidemann. *Stackable Design of File Systems*. Ph.D. dissertation, University of California, Los Angeles, 1995.

[10] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. RPC in the x-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 91–101. ACM, December 1989.

[11] Michael Leon Kazar. Synchronization and caching issues in the Andrew File System. In *USENIX Conference Proceedings*, pages 31–43. USENIX, February 1988.

[12] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proceedings of the 14th Symposium on Operating Systems Principles*. ACM, Dec 1993.

[13] Yousef A. Khalidi and Michael N. Nelson. The Spring virtual memory system. Technical Report SMLI TR-93-09, Sun Microsystems, February 1993.

[14] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun Unix. In *USENIX Conference Proceedings*, pages 238–247. USENIX, June 1986.

[15] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[16] Marshall McKusick, William Joy, Samuel Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[17] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[18] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Conference Proceedings*, pages 247–256. USENIX, June 1990.

[19] Rob Pike and Brian Kernighan. Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal*, 63(8):1595–1605, October 1984.

[20] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture*. The MIT Press, 1985.

[21] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[22] David S. H. Rosenthal. Evolving the vnode interface. In *USENIX Conference Proceedings*, pages 107–118. USENIX, June 1990.

[23] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *USENIX Conference Proceedings*, pages 119–130. USENIX, June 1985.

[24] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 25–34. ACM, December 1985.

[25] Glenn C. Skinner and Thomas K. Wong. "Stacking" vnodes: A progress report. In *USENIX Conference Proceedings*, pages 161–174. USENIX, June 1993.

[26] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[27] Mark Wittle. LADDIS: The next generation in NFS file server benchmarking. In *USENIX Conference Proceedings*, pages 111–128. USENIX, June 1993.

A Interface Changes

A goal of our work is to provide the minimal changes to existing systems and allow a modular adoption of cache coherence. This appendix summarizes our interface changes. Considerable mechanism underlies them, as the body of the paper presumably makes clear.

All new code in our implementation is freely available under a BSD-style copyright. A complete distribution is available to those with a SunOS 4.x source-code license. The implementation includes modules to manage byte-range and named-object lists, locking, and modifications to make the UFS and null layer cache coherent. The authors welcome inquiries.

In the following sections we present interfaces with C-like declarations. In these declarations, IN, OUT, and INOUT denote the direction of data movement. Ordinarily vnodes are adjusted to refer to the current layer as operations move down and up the stack; the NOTRANSLATE modifier indicates that this mapping should not occur. (This option is required for vnodes in an interface when vnodes must refer to a particular layer of the file, rather than the "current" layer.) All operations return "errno"-style error codes.

A.1 Stack-friendly interface changes

As described in Section 5.3, efficient data-page caching in a multiple-layer stack requires changes to three vnode operations. These operations are based on the corresponding SunOS 4.x operations but are modified to separate pager and cacher functionality. In the interface this change is reflected by replacing the original *vp* argument (which served as both the paging and caching agent) with three parameters: *vp*, the paging vnode; *mapvp*, the caching vnode; and *name*, a reference to cache-manager information.

vop_stackgetpage (IN struct vnode **vp*, IN struct svcm_name **name*, IN NOTRANSLATE struct vnode **mapvp*, IN u_int *offset*, IN u_int *length*, IN u_int **protection_p*, INOUT struct page ***page_list*, IN u_int *page_list_size*, IN struct seg **segment*, IN addr_t *address*, IN enum seg_rw *rw*, IN struct ucred **cred*)

A *getpage* operation is invoked to service a page fault in memory backed by a layer. We have expanded the original *vp* argument into *vp*, *name*, and *mapvp*. *Offset* and *length* specify the required data. The remaining arguments are employed by the VM system.

vop_stackputpage (IN struct vnode **vp*, IN struct svcm_name **name*, IN NOTRANSLATE struct vnode **mapvp*, IN u_int *offset*, IN u_int *length*, IN int *flags*, IN struct ucred **cred*)

The *putpage* operation is the opposite of *getpage*: it writes dirty pages back to stable storage. We change *vp*, *name*, and *mapvp*.

vop_stackrdwr (IN struct vnode **vp*, IN struct svcm_name **name*, IN NOTRANSLATE struct vnode **mapvp*, INOUT struct uio **uiop*, IN enum uio_rw *rw*, IN int *ioflag*, IN struct ucred **cred*)

A *rdwr* operation is used to read or write data. Again, we change *vp*, *name*, and *mapvp*. The *uio* specifies what data will be read or written.

A.2 Cache-coherence interfaces

Below are the two vnode operations which have been added to support cache coherence, and the cache-object registration interface exported by the cache manager.

vop_cachenamevp (IN struct vnode **vp*, OUT NOTRANSLATE svcm_name_token **token*, IN struct ucred **cred*)

Vop_cachenamevp is called when an upper-layer creates a new vnode. It returns the token representing the simply-named part of the stack. This token is then used to build an *svcm_name*, the data structure used by the cache manager to record caching information.

vop_cache_callback (IN struct vnode **vp*, IN struct svcm_name **name*, IN enum svcm_obj_classes obj_*class*, IN void **obj*, IN struct ucred **cred*)

Vop_cache_callback is invoked when the cache manager invalidates a cache-object. The *obj* parameter specifies the cache-object to be purged. For byte-range classes, *obj* specifies the region's offset and length; for named-objects it points to a length-counted string.

svcm_register (INOUT struct svcm_name **name*, IN struct vnode **own_vp*, IN enum svcm_obj_classes *obj_class*, IN u_int *obj_name_length*, IN void **obj_name*, IN enum svcm_status *status*, IN struct ucred **cred)*;

Svcm_register is called by each layer implementation after it has locked the file, but before it attempts to cache data. It informs the cache manager that *own_vp* wishes to cache object *obj_name* of class *obj_class* with *status* rights in the simply-named file *name*. The cache manager will consult its records and call-back any vnodes with conflicting cache requests and all vnodes with general-naming.