Stackable Design of File Systems

John Shelby Heidemann

University of California, Los Angeles

September, 1995

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

UCLA Computer Science Department
Technical Report UCLA-CSD-950032

Thesis committee:
Gerald J. Popek, co-chair
D. Stott Parker, co-chair
Richard Muntz
Rajive L. Bagrodia
Kirby A. Baker

To my family—
my mother Dorothy
and my brother Ben

Contents

List of Figures

List of Tables

Abstract of the Dissertation

Stackable Design of File Systems

by

John Shelby Heidemann
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1995
Professor Gerald J. Popek, Co-chair
Professor D. Stott Parker, Co-chair

This dissertation presents the design, implementation and evaluation of file-system design with *stackable layers*. Stackable layering addresses two significant problems in file-system development. First, existing services are difficult both to extend incrementally and to re-use in new work. Stacking addresses this problem by constructing sophisticated new services as a *stack* of new and existing layers. Layers work together since each layer is bounded above and below by the same (*symmetric*) interface; layer configuration is limited only by semantic constraints. Layers can be independently developed and distributed as binary-only modules to protect the investment in their development. Incremental improvements to existing services can be provided through new, thin layers.

Second, evolution of filing interfaces presents a problem to development and maintenance of services. In some respects, evolution is often too fast, as when vendor changes to interfaces invalidate existing third-party layers, greatly adding to their development and maintenance costs. At the same time, evolution is too limited and slow, as when developers and especially third parties cannot provide new services because of the constraints of old, centrally-managed interfaces. We address these problems by providing an *extensible* layering interface which supports managed interface evolution by both vendors and third parties. When interface changes are too large or are incompatible with existing practice, a *compatibility layer* can smooth over the changes. With an extensible interface, a layer may be confronted by an operation it does not understand. A standard mechanism allows layers to handle these operations by sending the operation to a lower layer for processing.

Stacking enables and simplifies several design techniques. A transport layer may move operations between machines and allows user-level layer development. Our stacking solution also includes a cache-coherence protocol to synchronize state across stack layers and a lightweight layering protocol allowing the benefits of independent development to extend even to very “thin” layers. We have constructed several layers using our stacking facilities.

This dissertation describes both the implementation of these services and their measurement and evaluation. We examine the performance of the stacking framework, cache-coherence protocols, and lightweight layers, concluding that stacking often adds little or no cost to user-observed performance and minimal additional kernel overhead. Finally, our experiences using stacking to develop and deploy several layers suggest that new services can be provided significantly easier with stacking than with traditional methods.

Acknowledgments

I first would like to thank my advisor, Jerry Popek, for his advice and support during this research. His encouragement has helped take this work further than it otherwise might have gone. I would also like to acknowledge his contributions, both to the direction and focus of this research and particularly to its presentation.

I have been very fortunate that this work has taken place in the context of a larger research effort, the Ficus project. I am indebted to Richard Guy, David Ratner, and Ashvin Goel for their discussions and support throughout this research. I would also like to thank the other members of the Ficus project. Chronologically, the project has included Ted Kim, Dieter Rothmeier, Wai Mak, Tom Page, Yu Guang Wu, Jeff Weidner, Greg Skinner, Michial Gunter, Steven Stovall, Geoff Kuenning, John Salomone, Andrew Louie, Qian Qin, Noah Haskell, Mark Yarvis, Alexy Rudenko, and Andy Wang. Each of these people contributed to the project in a different way. I would like to highlight Yu Guang Wu for his early work on the null layer, Jeff Weidner, Ashvin Goel and Ted Kim for implementing other layers, and Dieter Rothmeier, Wai Mak, and David Ratner for being early users of the new interface. I would especially like to thank Janice Martin for a careful proofreading of the dissertation (quotation placement and any remaining errors are my responsibility). Finally, I would like to thank Alta Stauffer for keeping Jerry's life organized and parts of this thesis on his plate, and Monique Bennarosh and Janice Martin for keeping things together at UCLA.

As a part of this work I added a subset of the UCLA stacking interface into the 4.4BSD operating system. I am grateful to Kirk McKusick for this opportunity (as well as some constructive criticism). I would also like to thank Jan-Simon Pendry for his loopback file-system, upon which the 4.4BSD null layer was based.

This dissertation draws upon two papers for some of its material, one published in ACM Transactions on Computing [HP94], and the other in the ACM Symposium on Operating Systems Principles [HP95]. In addition to those already mentioned, I am indebted to the Greg Minshall, SOSP paper shepherd, and anonymous reviewers of these papers for their comments which improved those papers and, indirectly, this dissertation.

Systems performance analysis is greatly aided by widely available tools and benchmarks. In this work I have employed both kitrace, by Geoff Kuenning [Kue95], and the Andrew benchmark from the Andrew File-System project [HKM88], as modified by John Ousterhout [Ous90].

I gratefully acknowledge the financial support of the Advanced Research Projects Agency and the USENIX Association toward this research. Development of stacking at UCLA would have been impossible without their aid. In the course of this research I was supported by ARPA contracts F29601-87-C-0072 and N00174-91-C-0107, and the 1990-91 USENIX Graduate Research Scholarship.

All trade-marked terms which appear in this document, including Unix, SunOS, NFS, and PostScript, are held by their respective owners.

Finally, I would like to thank Karen Schulz for her advice and encouragement during this work.

I welcome comments about this dissertation. I can be reached by e-mail at ⟨johnh@ficus.cs.ucla.edu⟩.

Further information about his work can be found on the World-Wide Web at ⟨http://ficus-www.cs.ucla.edu/ficus-members/johnh/work.html⟩.

Chapter 1

Introduction

Filing services are one of the most user-visible parts of the operating system, so it is not surprising that many new services are proposed by researchers and that a variety of third parties are interested in providing these solutions. Of the many innovations which have been proposed, very few have become widely available in a timely fashion. We believe this delay results from two deficiencies in practices of current file-system development. First, file systems are large and difficult to implement. This problem is compounded because no good mechanism exists to allow new services to build on those which already exist. Second, file systems today are built around a few fixed interfaces which fail to accommodate the change and evolution inherent in operating systems development. Today's filing interfaces vary from system to system, and even between point releases of a single operating system. These differences greatly complicate and therefore discourage third-party development and adoption of filing extensions.

These problems raise barriers to the widespread development, deployment, and maintenance of new filing services. The thesis of this dissertation is that a layered, *stackable* structure with an *extensible* interface provides a much better methodology for file-system development. We propose construction of filing services from a number of potentially independently developed modules. By stackable, we mean that these modules are bounded by identical, or *symmetric*, interfaces above and below. By extensible, we mean that these interfaces can be independently changed by multiple parties, without invalidating existing or future work.

To validate this thesis we developed a framework supporting stackable file-systems and used that framework to construct several different filing services. This dissertation describes the design, implementation, and evaluation of this system.

1.1 Motivation

This dissertation explores stackable layering in three stages. First we discuss the issues and approaches involved in stacking. We then explore issues in cache-coherence and lightweight layering which follow from this model. This section introduces each of these topics.

1.1.1 Stacking

Modularity is widely recognized as a necessary tool in the management of large software systems. By dividing software into small, easily managed pieces, modularity provides advantages in organization and verification throughout the software life-span. The hallmark of modularity is a set of independent software components joined by well-defined interfaces.

When modular interfaces are carefully documented and published, they can also serve as an important tool for compatibility and future development. By providing a common protocol between two subsystems, such an interface allows either or both systems to be replaced without change to the other. Improved modules can therefore be independently developed and added as desired, improving the computing environment. Interfaces such as POSIX.1 [IEE90] and NFS [SGK85] are examples of interfaces widely used to provide operating system and remote filing services.

Because operating systems represent such a widely used service, the development of modular systems interfaces can have particularly wide impact. The best example of standard systems interfaces is probably POSIX.1. Programs based on this interface are widely portable and can execute on a wide range of today's hardware, from personal computers to the largest supercomputer.

One would like to see this same level of portability currently present for application programs in operating systems themselves. Large portions of an operating system are hardware independent and should run equally well on any computer. Such portability has been largely achieved, as exemplified by portable operating systems such as Unix [RT74].

What has not been achieved to the same extent is portability of major kernel subsystems. Because of the exacting nature of software, and because of the lack of modular interfaces within the operating system itself, the Unix kernel has been slow to evolve to new software technologies. While individual vendors have adopted new kernel technologies such as STREAMS [Rit84], new virtual memory approaches, and new file-systems, such additions have only come slowly and at considerable expense.

Micro-kernel designs are one approach to kernel modularity. Kernels such as Mach [ABG86] and Chorus [RAA90] divide the operating system into two parts: a core of memory management, process control, and simple inter-process communication; and a server (or servers) supporting the remainder of the traditional operating system, including accounting, protection, file-system and network services, and backwards compatibility. For the case of Mach and Unix, as a figure of merit, the core is on the order of 15% of the total operating system kernel. This intra-kernel boundary is an important structuring tool, particularly because it offers a platform on top of which third parties can offer a variety of services. But this approach does not provide a total solution, as it fails to address the modularity of the remaining 85% of the system.

File systems, a rich portion of the remaining kernel, are an active area of research. Many file-system services have been proposed, including version management, user-customizable naming, fast log-structured storage, replication, and large-scale distributed filing. All of these have well-developed prototypes, but appearance in commercially available systems has been both slow and piecemeal.

Adoption of these new filing services has been slow in part because file systems are large, monolithic pieces of code with limited internal modularity. Although recent approaches to file-system modularity (such as Sun's VFS interface [Kle86]) allow easy substitution of *entire* file-systems, they do little to support modularity within file systems themselves. As a result, it is not easy to replace or enhance separate portions of the file system; for example, keeping the physical disk management and installing a new directory layer.

Another problem with existing approaches to file-system modularity is that they are particularly fragile in the face of change, one of the goals modularity is intended to facilitate. Evolution of the kernel to more efficient mechanisms, and addition of new file-systems have required frequent changes to the interface, resulting in incompatibility between vendors of similar operating systems and even between different releases of the "same" operating system. Frequent change and the resulting incompatibilities have largely discouraged third-party innovation, restricting introduction of new filing services to the primary operating system vendors alone [Web93]. This contrasts sharply with other operating system interfaces such as device access and graphical user interfaces, where standard interfaces have allowed competition and rapid development of a wide array of services.

The problems of re-use and change are recognized by developers. Current approaches to file-system development begin to address these problems, but few do so satisfactorily. A common approach to filing development is to take an existing system and begin modifying it. Direct modification achieves good code re-use but greatly hinders change as new services become bound to the licensing and portability constraints of the original code. An approach common in the operating systems research community is to develop new services as user-level NFS servers (for example, see Deceit [SBM90], semantic filing [GJS91], and Alex [Cat92]). Because the NFS protocol is very well specified and nearly universally available, this approach is very robust to external change, but it offers no support for *internal* change. Interfaces for new services must be supplied with new protocols in parallel to NFS, at great expense in implementation cost and maintenance, or with modifications to the NFS protocol, greatly reducing portability. A final approach commonly taken is to provide a new service at the VFS-level. A VFS can achieve some re-use, but this interface provides little support to manage change. Third-party experience developing for the VFS interface documents the burden in keeping up with inter- and intra-vendor change [Web93].

Difficulties with current approaches suggest that a better solution to filing service design is needed. For inspiration and potential solutions we examine how these problems are managed in other large software systems which allow third-party contribution.

Unix shell programming is one example of a successful development environment. Individual programs are easily connected by a flexible, standard interface, the pipe [RT74]. Programs can be combined quickly and easily in the shell with a simple programming language. New programs are widely and independently developed by a number of vendors. These features combine to provide an excellent environment for rapid prototyping and development.

This approach to software modularity has also been applied to kernel-level subsystems. The STREAMS system is Ritchie's redesign of Unix terminal and network processing. STREAMS modules are bounded above and below by a syntactically identical interface, allowing very flexible module configuration. Because this interface is *symmetric* in this way, users are encouraged to combine a number of small modules into protocol *stacks*. Furthermore, because the interface is formally defined these modules can be independently developed by third parties and combined to address the task at hand. As a result, third parties have built commercial quality layers that integrate well with other protocol modules. This modular approach allowing multiple, independent groups to contribute to communications facilities is one of the reasons Unix is attractive as a base for networking and distributed systems software in engineering and commercial use.[1]

This dissertation seeks to apply the principles of stackable layering to file-system development. We envision a situation where a user's filing environment is composed of stacks of independently developed filing *layers*. Like STREAMS, the interface between layers will be symmetric to allow flexible configuration. The interface must also be *extensible* and robust to internal and external change. Chapter 2 explores these issues and requirements in more detail. Chapter 3 examines different ways stacking can be used to address problems unique to filing. Finally, Chapters 4 and 5 present and evaluate our prototype system developed at UCLA.

1.1.2 Cache coherence

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 services 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. Caching will also be important in file systems constructed from stackable layers. For best results data will be cached in the layer closest to the user. With layering, though, a user may choose to access a stack through different layers at different times. For example, administrative actions can be performed more easily at lower stack layers. Distributed filing systems too can produce data accesses to different stack layers (we consider one such case in detail in Section 8.4). If data is always cached near the point-of-access, access to multiple layers may result in the same logical data cached in different layers.

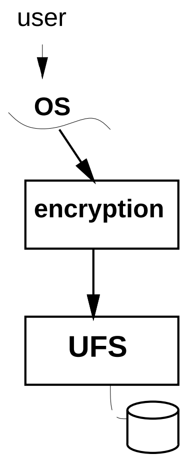

Figure 1.1: A sample application of the stackable layers. Each layer is connected by a standard interface.

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 1.1. 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. If layers are provided by different parties, how can they cooperate to provide coherence?

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

[1] In fact, commercial systems such as Novell's Netware-386 have adopted the STREAMS framework, presumably for similar reasons.

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, user-level file-locking consistency, and internal concurrency-control.

Therefore, to summarize the issue of cache coherence:

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.
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. Chapter 6 discusses the characteristics required of a solution to this problem. Chapters 7 and 8 present our prototype solution and evaluate its effectiveness.

1.1.3 Featherweight layering

Code reuse is on one significant motivation for stacking. Large layers which encompass several abstractions and services cannot easily be reused due to their weight and inflexibility. Thus, an ideal filing environment would be composed of stacks of several "thin" layers.

Two tensions push against the decomposition of filing services into multiple layers. First is the design effort required. Selection and definition of components requires careful thought. There are often several different ways to decompose a service; a poor selection can complicate layer implementation and limit reusability.

Second, layering overhead also constrains service decomposition. Our layering mechanism was designed to minimize overhead, but full generality in a layering mechanism implies a certain amount of overhead. Our measurements suggest a 1–2% system-time overhead for general-purpose layers (see Section 5.1.3 for details of this evaluation).

Although a 1–2% system-time overhead is not significant for a layer providing a new service to the user, this overhead is a consideration if layering is to be used internally to structure such services. This limitation is unfortunate since there are several thin layers (such as name-lookup caching, VM/file-system interaction, and compatibility layers) that are common across a number of filing services. These layers individually make only minor alterations to the interface, but they still incur the overhead of the full layering mechanism. Adding several such layers to a stack would add noticeable overhead; and several of these layers will often be added to *each* layer of a multi-layer stack.[2] A general-purpose layering mechanism is not suitable for these lightweight services.

Featherweight layers are special "lightweight" layers designed to address the problem of layer overhead. Featherweight layers obtain performance improvements over general layering mechanisms by restricting the capabilities they provide and by "piggy-backing" on the administrative machinery of a "host" layer. Since featherweight layers provide only a subset of stacking functionality they cannot be used to implement all layered services. Instead they provide the lightweight portions of a stack in cooperation with a few general-purpose layers.

Chapters 9 and 10 present the design, implementation, and evaluation of a featherweight layering service. By placing a few limitations on layering functionality they show that it becomes possible to create featherweight layers with library-routine-like performance while retaining benefits of stackable layering design such as third-party development and late binding.

1.2 Related Work

Modularity in systems programming has a rich history. Our work builds upon this background, inspired by advances in symmetric module design, general file-system structuring, distributed shared memory protocols. and some recent work on stackable filing. We next briefly summarize related work. We cover the relationship between our work and others more completely in Chapter 11.

1.2.1 Symmetric interfaces and stacking

Unix shell programming with pipes [RT74] is now the widest use of a symmetric interface, for software development and other applications [PK84]. Ritchie

[2]For example, vendors may configure compatibility layers on to all stacks by default to insure backwards compatibility. Similarly, layers implementing cache coherence would need to be configured into any layer which might cache data.

then applied these principles to kernel structure in his STREAMS I/O system [Rit84]. Such work has since been adopted in a number of versions of Unix.

The x-kernel [HP88] is a new kernel designed originally to provide customized network protocols. Using a symmetric interface for all kernel services ("everything is a protocol"), great flexibility in protocol selection and combination is provided. They employ both run-time protocol selection and an efficient implementation to demonstrate that layering can be performance competitive with monolithic protocol implementations.

1.2.2 File-system structure

Research in the late 1960s and early 1970s modularized operating systems, proposing a multi-layer implementation.

To provide for multiple file-systems, several "file-system switch" mechanisms have been developed [Kle86, RKH86, KM86]. These typically found quick use in the support of network file access [SGK85, RFH86] and have since been applied to the support of other file systems [Koe87]. None of these approaches provide explicit support for stacking or extensibility, but all provide basic modularity.

1.2.3 Stackable filing systems

Sun Microsystems applied the vnode interface to build two-layer file system stacks in their loopback and translucent file-systems [Hen90]. Internal to the operating system, stacking is used to support device special files.

More recently, Rosenthal [Ros90] and later Skinner and Wong [SW93] at SunSoft have experimented with a modified vnode interface to provide dynamic file-system stacking. The Spring project (at Sun Laboratories) has also developed stackable filing technology [KN93a].

1.2.4 Cache coherence

Our cache-coherence protocols build 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. We review this work in Section 11.2. Second, we build upon the cache-coherent stacking work of the Spring project at Sun Laboratories [KN93a].

1.2.5 Featherweight layering

Featherweight layering is inspired by the observation that the performance of a layered system is often best when logically independent layers share implementation details. Others have suggested that performance of layered systems is improved by avoiding a process-per-layer [Rit84, HP88] or by employing continuations [DBR91]. We improve file-system layering performance by restricting layer state. We expand on these issues in Section 11.3.

1.3 Road Map to the Dissertation

The thesis of this dissertation is that stackable filing with an extensible interface improves file-system development. We begin exploring this thesis in the next chapter by motivating the need for stackable layering and extensible interfaces. We also introduce the problem of maintaining data coherence across layers of a stack, and we suggest the need for very lightweight stackable layers. The remainder of the thesis considers each of these topics, discussing in turn the design, implementation, and evaluation of stacking, cache coherence, and lightweight layering. The dissertation concludes with an extended discussion of related work and issues for future study.

Chapter 2

Stacking Model

We have identified several problems that exist with current approaches to file-system development, problems that we believe stackable filing can address. In this chapter we describe the characteristics which are desirable in an improved filing environment:

extensibility Filing must be robust to both internal and external change.

stacking It must be possible to add new functionality to existing services.

coherence Assertions about data consistency must be possible across multiple layers.

In addition, several secondary goals place restrictions on the final solution:

distributable Computers today are increasingly networked with shared filing environments. Furthermore, microkernel operating systems may place the filing service in one or more server processes, each with a different address space. Filing must work in each of these environments.

scalability Each requirement must meet a wide range of demands. Extensibility must work equally well for vendors, third parties, and independent developers. Stacking must work both for complex services and for small, lightweight additions. Distribution must apply from different server processes of a microkernel to multiple machines on a LAN to computers cooperating across an internetwork.

ease-of-use If meeting these goals results in a system which is difficult to use, the ultimate goal of an improved file-system-development environment will be defeated.

efficiency If these services impose excessive overhead, then they will not be used. The cost of services must be proportional to the service provided.

The remainder of this chapter discusses each of these characteristics (extensibility, stacking, and coherence) in light of these restrictions.

2.1 Extensibility

Webber characterizes the dilemma of third-party vendors quite well [Web93]:

> Unix kernels with a VFS architecture have been commercially available for many years. Sun Microsystems, for example, described their VFS architecture in the 1986 Summer Usenix proceedings [Kle86]. By many measures the VFS concept has been quite successful, but from a third-party point of view there are two major problems:
>
> - Few vendors have the same VFS interface.
> - Few vendors provide release-to-release source or binary compatibility for VFS modules.
>
> We call these two problems the *VFS portability* problem and the *lock-step release* problem, respectively. Together, they make VFS modules expensive to produce, expensive to port, and expensive to maintain.

To these observations we add one additional problem: few third parties can change and extend the interface. We call this limitation the *extension* problem. If third parties are to provide truly novel new services, then it must be possible for them to add operations to the interface. These new operations must be equivalent to vendor-supplied operations in terms of performance and capability.

We view these problems as evidence that any file-system interface which is successful in the long-term must provide *extensibility*. We next consider evidence of change in existing systems, ways to delay evolution, and finally, how our secondary goals influence this design.

2.1.1 Evidence of evolution

Rosenthal has examined the gradual evolution of the SunOS file-system interface [Ros90]. He found significant changes to the interface in every major operating system release. Table 2.1 shows his comparison of changes.

Rosenthal's study demonstrates the frequency of evolution through one version of Unix. It is also interesting to note that the designers of SVR4 Unix recognized the inevitability of change and allocated space for the future addition of 32 operations. We discuss later how space reservation only addresses part of the problem in Section 11.1.3.

2.1.2 Alternatives to manage change

Given the inevitability of software evolution, there are surprisingly few ways to accommodate it in current filing interfaces. Without a formal way to manage evolution, two kinds of problems quickly appear: development without evolution, and managing change when it does arrive.

Several approaches are possible to avoid evolution. A common one is to require that everyone use the same version of software; change is prohibited by fiat. While this approach works for small groups over short periods of time, it fails as scale and duration increase. The longer a configuration is frozen, the greater users' demands for new software. As the user population grows from tens of machines to hundreds or thousands, the different goals and requirements of multiple administrative domains mandate different software configurations.

Often, pressures to adopt new tools force their use before change can be fully accommodated. If existing interfaces must remain unchanged, the only alternative is to create an additional, parallel interface. While this approach allows support of new services, it also complicates the environment. Such work needlessly duplicates existing efforts as similar goals are accomplished in different ways. In the long run, this *ad hoc* approach to evolution will likely cause difficulties in maintenance and further development.

Eventually, change must occur. Barriers to evolution imply that, in practice, widely used operating system modifications derive only from a few major systems-software vendors and research institutes in occasional, perhaps annual, systems software releases. While this policy of change delays problems resulting from change to an occasional event, eventually these difficulties must be faced.

Because the authority of operating system change is vested largely in the systems software vendor, potential for third-party enhancement is greatly restricted. Unavailability of source code, incompatibility with other third-party changes and even vendor-supplied updates together discourage third-party innovation. Finally, the methods used by manufacturers to improve services are often not available to third parties. As a result, third-party modifications suffer delay, increased complexity, and performance penalties compared to vendor-supplied improvements, further handicapping independent development.

2.1.3 Design constraints

Third-party support for software evolution is critical to the timely development of new capabilities. The filing interface must be able to evolve as needs and capabilities change. Our secondary goals influence this design in several ways.

It must be easy to provide extensibility in a distributed file-system as well as to layers on a single host. Extensibility requires that each operation be formally defined. Support for extensibility in distributed filing requires that this definition must include information sufficient to allow an RPC protocol to reproduce the operation on a different machine or in a different address space.

Extensibility must be scalable in several ways. It must scale in those allowed to initiate change. The process of evolution cannot be controlled by any central authority. Multiple organizations and individuals must be able to contribute, and their extensions must co-exist in a single system. Scalability also implies that there be no fixed limit on the number of extensions provided.

Ease-of-use implies that changes can occur incrementally and independently, and that they must not invalidate existing or future services. Software must gracefully adapt to its environment, both as a result of the presence of unexpected, new extensions, and the lack of expected support. Ideally, a new software module could be added without source code changes to it or any other mod-

release	vnode fields	vnode size	operation count
SunOS 2.0 (1985)	11 fields	32 bytes	24 operations
SunOS 4.0 (1988)	14	40	29
SunOS 4.1 (1990)	14	40	30
SVR4 without fill (1989)	11	40	37
SVR4 with fill (1989)	19	72	69
Rosenthal's prototype (1990)	6	20	39

Table 2.1: A slightly expanded version of Rosenthal's evaluation of vnode interface evolution in SunOS (derived from [Ros90]). Fill indicates space left in SVR4 for future expansion; Rosenthal's prototype is discussed in Section 11.6.1.

ule. Finally, ease-of-use requirements for stacking imply that layers are configured at run-time. We discuss these requirements more in the next section, but for the interface they imply that the caller and callee must be matched at run-time; at least this level of dynamic binding is required.

Since file-system operations are often in the tight loop of computation, efficiency is of primary concern.

2.2 Stacking

File systems frequently implement very similar abstractions. Nearly all file systems ultimately are grounded in disk access and file and directory allocation, for example. This observation motivates file-system *stacking*. If a complex filing service can be decomposed into several *layers*, then potentially each layer can be developed independently. Furthermore, in the future, layers can be individually upgraded as need or desire arises. Finally, a set of filing layers serve as building blocks for the construction of future services. Together, these examples show how stacking can reduce the cost of file-system development.

An example of layered filing is seen in Figure 2.1. The operating system vendor provided a standard file storage layer (the Unix file-system, or UFS). On the left stack a user has configured a compression layer over this basic file service.

A key characteristic of a stackable layer is that it possess a *symmetric* interface; it should export an interface to its clients which is syntactically the same as that which it depends upon from layers it stacks over. Layers bounded by a symmetric interface can be inserted between any existing stack layers (subject to semantic constraints, of course). For example, in the right-hand stack of Figure 2.1, the user has "pulled apart" the compression layer and UFS and inserted an encryption layer for more secure data storage.

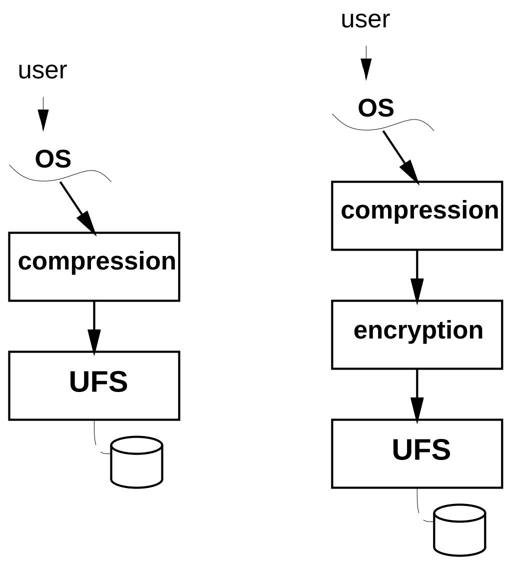

Figure 2.1: Two file-system stacks providing encryption and compression.

2.2.1 Design constraints

Again, our secondary constraints of distribution, scalability, ease-of-use, and efficiency all have implications on the design of stacking.

Distributed stacking requires that layers can bridge address space, protection domain, and machine boundaries. A convenient way to cross protection domains is with a *transport layer* which conceptually has ends in each domain and a transport protocol between. It may be advantageous to have multiple transport layers, each customized to serve a particular need (for example, transport between processes on a single machine compared to across a LAN or WAN).

Layer scalability implies that the cost of each layer is proportional to its capabilities. Very "thin" layers should be possible with minimal overhead, while "thick" layers may require additional mechanism. To scale in numbers of layers, per-layer memory requirements must be reasonable.

Layer ease-of-use is improved by run-time layer configuration. It should be possible for a user to easily create new layer instances as needed. In addition, dynamic loading of new layers should be possible.

Finally, the performance cost of layering must be minimized. There are several aspects to layering cost (described later in Section 9.2); the costs of providing layer abstractions and the cost of using those abstractions must be proportional to the services provided.

2.2.2 Stacking and extensibility

As described thus far, a conflict between stacking and extensibility is apparent. Stacking is based on the premise that each layer is bounded (above and below) by the same interface. Extensibility implies that layer users can independently change and evolve the interface.

Extensibility requires that layers be robust to change. In a non-layered environment, this means that a layer must respond to unknown operations with an error message. For example, in Figure 2.2a the UFS must reject `vop_set_extent_size` operation (returning an error code) which would be handled by an extent-based file-system (in Figure 2.2b). Any system with extensibility must specify some (possibly configurable) default action for unknown operations.

In a layered environment intermediate layers often act as "filters", providing a small service by changing a few operations, but relying on lower layers to provide most aspects of storage. When presented with an unknown operation, intermediate layers therefore *bypass* that operation to a lower layer for processing. Figure 2.2c illustrates bypassing `vop_set_extent_size`.

2.2.3 Generalized stacking

The linear file-system stacks presented thus far are really a special case of general layering. In general, *trees* of layers are possible, a single layer can stack-upon or be stacked-upon by multiple other layers.

We distinguish between two kinds of "forked" stacking. *Fan-out* occurs when a layer stacks "outwards" over multiple layers. Figure 2.3 illustrates how fan-out might be used to implement disk mirroring.

Fan-in allows multiple clients access to a particular layer. Fan-in is useful when when different clients of a service desire different views of the data. For example, in Figure 2.4 the UFS has fan-in. Section 3.3 discusses advantages and uses of fan-in.

2.2.4 Stacking and concurrency

A complete definition of stacking must consider the effects of stacking on other processes. When stack configuration is changed by one process, how does this affect other processes that are actively using the stack? On one hand, perhaps all processes should always see exactly the same stack configuration. In this case, pushing a layer on a stack should interpose that layer between the prior layer and all of its clients. On the other hand, perhaps clients should get what they asked for when they asked for it. New clients will, of course, see the new layer, but existing clients should continue to see the configuration they've been seeing.

Different choices on this issue make sense in different contexts. If a "lock-out" layer were placed on a stack to deny access, it might be required to deny access to all clients (current and future), not just future clients. On the other hand, a client in the midst of reading an encrypted file probably does not want to see decrypted data in the middle of the data stream as some other client changes the stack.

We discuss alternatives to this issue in more detail later.

2.3 Coherence

In a monolithic file-system the file-system designer has complete control over execution. Locking and caching are feasible because the designer has control over all data access and execution paths, and can insure that deadlock

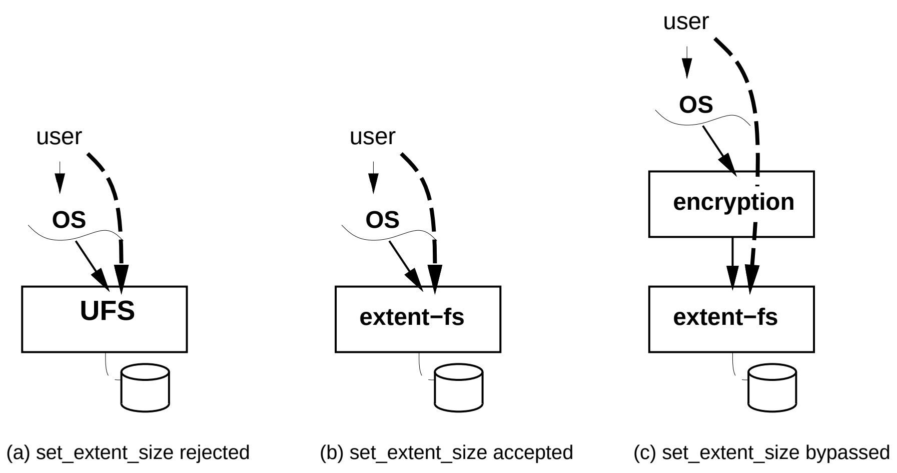
user
OS
UFS
(a) set_extent_size rejected
user
OS
extent−fs
(b) set_extent_size accepted
user
OS
encryption
extent−fs
(c) set_extent_size bypassed

Figure 2.2: Treatment of `vop_set_extent_size` by different layers.

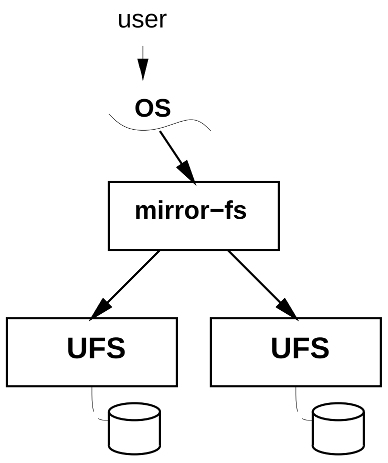

Figure 2.3: A tree of file-system layers to provide disk mirroring. The mirror-fs layer exhibits *fan-out*.

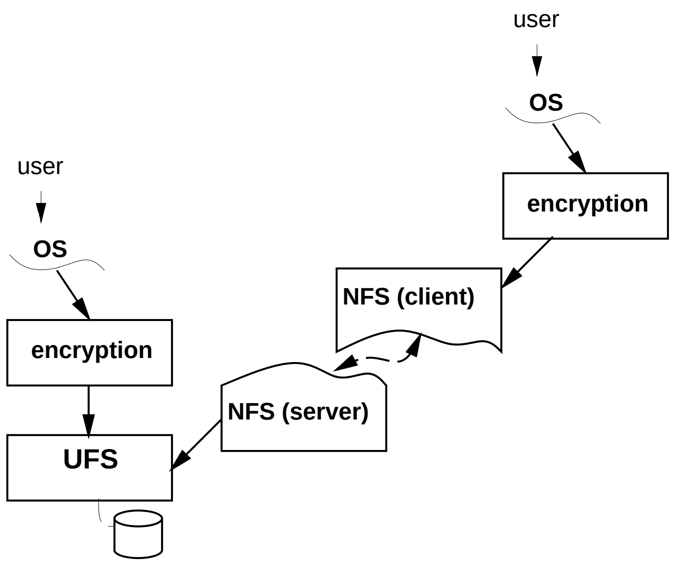

Figure 2.4: A tree of file-system layers exhibiting fan-in.

and access to old data are not possible. In short, the designer's complete control over the situation allows him or her to make assertions about file-system state.

The designer of a file-system layer loses this ability. The layer may be combined at run-time with services from other developers, each with their own views of locking and caching. Late layer binding and distribution of functionality among layers from multiple vendors makes it extremely difficult for the designer of any individual layer to make assertions about the global state of "filing". Unfortunately, such assertions are required to insure freedom from deadlock and coherence of cached data.

To address the problem of state assertions in a multi-vendor, multi-layer system, a general coherence mechanism is required. An important special case of this mechanism is cache coherence: a protocol to keep copies of data in different layers up-to-date.

2.3.1 Design constraints

The constraints of distribution, scalability, ease-of-use, and efficiency affect coherence.

A number of protocols for distributed coherence exist, yet the wide variety of performance constraints present from sharing on a single machine to across the Internet make it unlikely that any single solution can meet all needs. We discuss how this observation influences cache coherence in stacking in Section 6.5.

Scaling is of concern in several different dimensions for coherence. Coherence solutions must adapt to support a large number of objects of several different types. In addition, coherence protocols must adapt to meet future needs as well as current needs.

Cache-coherence protocols in distributed shared memory systems have become quite sophisticated, involving compiler and programmer support and sporting several policies for different data objects. Stackable filing is not frequently employed to serve as primary store for large multiprocessor compute tasks, so a simpler, easier-to-use solution is required for coherence in stackable filing. Furthermore, such complex solutions would quickly overwhelm what should be relatively simple filing layers such as encryption. Such a result would defeat our goal of improving the filing development environment.

Finally, the services provided by coherence must be proportional to their overheads.

2.4 Model Summary

This chapter presented the three distinguishing features of stackable layering: stacking, extensibility, and coherence. In the broadest sense these characteristics have been goals of software engineers for many years: stacking is "just" modularity; extensibility, change management; and coherence, successful design. But truly successful application of these principles to *file-system* layering add new dimensions of distribution, scaling, ease-of-use, and efficiency. This chapter explored how these constraints affect a solution. Following chapters will explore how stacking can be used to simplify filing development and the details of one system implementing these characteristics.

Chapter 3

Stacking Techniques

This section examines in detail a number of different file-system development techniques enabled or simplified by stackable layering.

3.1 Layer Composition

One goal of layered file-system design is the construction of complex filing services from a number of simple, independently developed layers. If file systems are to be constructed from multiple layers, one must decide how services should be decomposed to make individual components most reusable. Our experience shows that layers are most easily reusable and composable when each encompasses a single abstraction. This experience parallels those encountered in designing composable network protocols in the x-kernel [HPA89] and tool development with the Unix shells [PK84].

As an example of this problem in the context of file-system layering, consider the stack presented in Figure 3.1. A compression layer is stacked over a standard Unixfile-system (UFS); the UFS handles file services while the compression layer periodically compresses rarely used files.

A compression service provided above the Unix directory abstraction has difficulty efficiently handling files with multiple names (hard links).[1] This is because the UFS was not designed as a stackable layer; it encompasses several separate abstractions. Examining the UFS in more detail, we see at least three basic abstractions: a disk partition, arbitrary length files referenced by fixed names (inode-level access), and a hierarchical directory service. Instead of a single layer, the "UFS service" should be composed of a stack of directory, file, and disk

[1]Consider, for example, a layer which detects compressed files by their extension. When the file is uncompressed, the file will be renamed to indicate its new status. It is difficult to rename all names of a file with multiple links because some names are unknown.

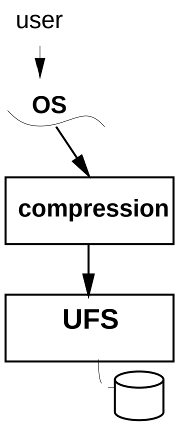

Figure 3.1: A compression service stacked over a Unix file-system.

layers. In this architecture the compression layer could be configured directly above the file layer. Multiply-named files would no longer be a problem because multiple names would be provided by a higher-level layer. One could also imagine re-using the directory service over other low-level storage implementations. Stacks of this type are show in Figure 3.2.

3.2 Layer Substitution

Figure 3.2 also demonstrates layer substitution. Because the log-structured file-system and the UFS are semantically similar, the compression layer can stack equally well over either. Substitution of one for the other is possible, allowing selection of low-level storage to be independent of higher-level services. This ability to have "plug-compatible" layers not only supports higher-level services across a variety of vendor-customized storage fa-

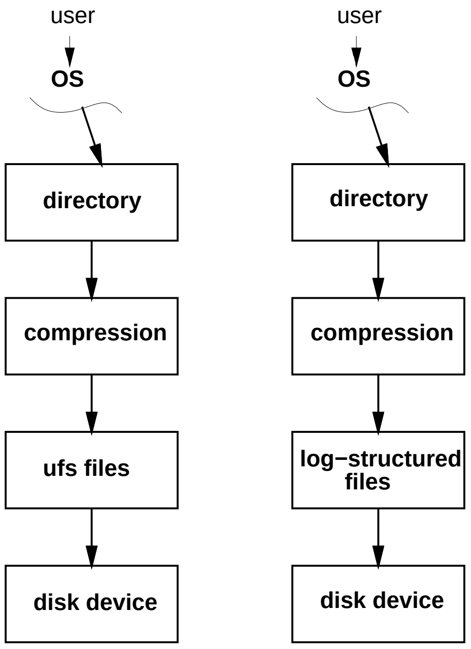

Figure 3.2: A compression layer configured with a modular physical storage service. Each stack also uses a different file storage layer (UFS and log structured layout).

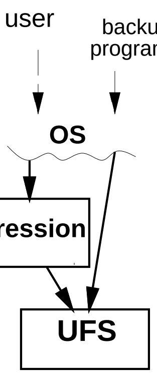

Figure 3.3: Multiple-layer access through and beneath a compression layer. Access through the compression layer provides users transparently uncompressed data. Fan-in allows a backup program to directly access the compressed version.

cilities, but it also supports the evolution and replacement of the lower layers as desired.

3.3 Multi-Layer Access

Modularity, re-use, and third-party involvement advocate the construction of filing services from a number of layers. If these layers are stackable (each exporting the same interface), then fan-in allows a layer to export its services not only to a single layer above, but also to a user or other layers. From the perspective of the stack as a whole, this sort of *multi-layer access* allows a stack to export several potentially different views of file data.

The ability to export multiple views of the same data is useful in several different ways. The most common use is to allow occasional access to lower stack layers for administrative purposes such as file backup and debugging. For example, in Figure 3.3, normal user access proceeds through a compression layer, allowing data on disk to be stored in a compressed format and transparently uncompressed on demand. With multi-layer access a backup program can bypass the compression layer and directly transfer the compressed data to the backup media, saving both time and backup storage.

Multi-layer access can be directly employed by users. Union-mounts in Plan 9 [PPT91] and 4.4BSD [McK95] for example, create a single "unified" directory from several underlying directories, yet users may need access to the underlying directories to install new software. A user of the compression service in Figure 3.3 also may wish to directly access the compressed data when making a copy of the file.

Finally, sophisticated stack configurations often employ multi-layer access internally. Multi-layer access occurs in Ficus; we describe this case in detail in Section 8.4. Similarly, multi-layer access can be employed with encryption layers to preserve data security over a network as shown earlier in Figure 2.4.

3.4 Cooperating Layers

Layered design encourages the separation of file systems into small, reusable layers. Sometimes services that could be reusable occur in the middle of an otherwise special-purpose file-system. For example, a distributed file-system may consist of a client and server portion, with a remote access service in-between. One can envision several possible distributed file-systems offering simple stateless service, exact Unix semantics, or even file replication. Each might build its particular semantics on top of an "RPC" remote access service, but if remote access is buried in the internals of each specific file-system, it will be unavailable for reuse.

Cases such as these call for *cooperating layers*. A "semantics-free" remote access service is provided as a reusable layer, and the remainder is split into two separate, cooperating layers. When the file-system stack is composed, the reusable layer is placed between the others. Because the reusable portion is encapsulated as a separate layer, it is available for use in other stacks. For example, a new secure remote filing service could be built by configuring encryption/decryption layers around the basic transport service.

An example of the use of cooperating layers in the Ficus replicated file-system [GHM90] is shown in Figure 3.4. The logical and physical layers of the Ficus stack correspond roughly to a client and server of a replicated service. A remote access layer is placed between them when necessary.

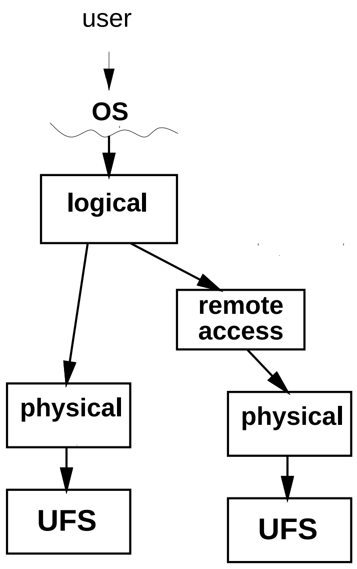

Figure 3.4: Cooperating Ficus layers (logical and physical) in the 1992 Ficus stack. Fan-out allows the logical layer to identify several replicas, while a remote access layer is inserted between cooperating Ficus layers as necessary.

3.5 Compatibility With Layers

The flexibility stacking provides promotes rapid interface and layer evolution. Unfortunately, rapid change often rapidly results in precisely the incompatibility this effort is intended to address. Interface change and incompatibility today often prevent the use of existing filing abstractions [Web93]. A goal of our design is to provide approaches to cope with interface change in a binary-only environment.

File-system interface evolution takes a number of forms. Third parties wish to extend interfaces to provide new services. Operating system vendors must change interfaces to evolve the operating system, but usually also wish to maintain backwards compatibility. Stackable layering provides a number of approaches to address the problems of interface evolution.

Extensibility of the file-system interface is the primary tool to address compatibility. Any party can add operations to the interface; such additions need not invalidating existing services. Third-party development is facilitated, gradual operating system evolution becomes possible, and the useful lifetime of a filing layer is greatly increased, protecting the investment in its construction.

Layer substitution (see Section 3.2) is another approach to address simple incompatibilities. Substitution of semantically similar layers allows easy adaption to differences in environment. For example, a low-level storage format tied to particular hardware can be replaced by an alternate base layer on other machines.

Resolution of more significant problems may employ a *compatibility layer*. If two layers have similar but not identical views of the semantics of their shared interface, a thin layer can easily be constructed to map between incompatibilities. This facility could be used by third parties to map a single service to several similar platforms, or by an operating system vendor to provide backwards compatibility after significant changes.

A still more significant barrier is posed by different operating systems. Although direct portability of layers between operating systems with radically different system services and operation sets is difficult, limited access to remote services may be possible. Transport layers can bridge machine and operating system boundaries, extending many of the benefits of stackable layering to a non-stacking computing environment. NFS can be thought of as a widely used transport layer, available on platforms ranging from personal computers to mainframes. Although standard NFS provides only core filing services, imparts restrictions, and is not extensible, it is still quite useful in this limited role. Section 5.3 describes how this approach is used to make Ficus replication available on PCs.

3.6 User-Level Development

One advantage of micro-kernel design is the ability to move large portions of the operating system outside of the kernel. Stackable layering fits naturally with this approach. Each layer can be thought of as a server, and operations are simply RPC messages between servers. In fact, new layer development usually takes this form at UCLA (Figure 3.5). A transport layer (such as NFS) serves as the RPC interface, moving all operations from the kernel to a user-level file-system server. Another transport service (the “u-to-k layer”) allows user-level calls on vnodes that exist inside the kernel. With this framework layers may be developed and executed as user code. Although inter-address space RPC has real cost, caching may provide reasonable performance for an out-of-kernel file-system [SS90] in some cases, particularly if other characteristics of the filing service have inherently high latency (for example, hierarchical storage management).

Nevertheless, many filing services will find the cost of frequent RPCs overly expensive. Stackable layering offers valuable flexibility in this case. Because file-system layers each interact only through the layer interface, the transport layers can be removed from this configuration without affecting a layer’s implementation. An appropriately constructed layer can then run in the kernel, avoiding all RPC overhead. Layers can be moved in and out

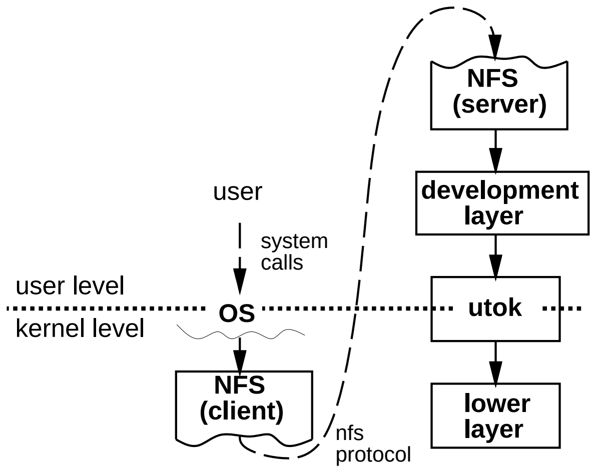

Figure 3.5: User-level layer development via transport layers.

of the kernel (or between different user-level servers) as usage requires. By separating the concepts of modularity from address space protection, stackable layering permits the advantages of micro-kernel development and the efficiency of an integrated execution environment.

3.7 Interposition

Some stacking implementations support *interposition*; layers added to the top-of-stack are used not only for future operations on the stack, but are interposed between *existing* users of the stack and the old stack-top. This capability to alter existing clients can be used effectively in several ways.

One example of the use of interposition is illustrated by the 1992 Ficus stack. Before the selection layer was added, the logical layer handled replica failure and switch-over. When a replica failure occured, we would like to interpose “redirection vnode” on the stack-top to transparently redirect existing clients of that vnode.[2] This configuration can be seen in Figure 3.6. (This approach is now accomplished more cleanly in the selection layer.)

Interposition is useful when operations must happen at

[2]The UCLA stackable interface does not support interposition, so we emulated it in this case by altering the operations vector of the vnode for the failed replica. This approach to emulation works well when a layer alters its own behavior but is not available when a third-party layer is involved.

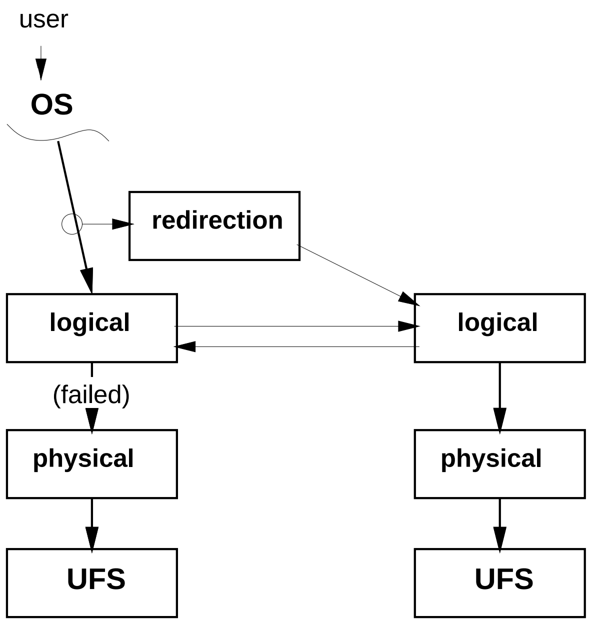

Figure 3.6: Interposition in the 1992 Ficus stack.

run-time and be seen by all existing users. Another example of this would be dynamic addition and removal of a measurement layer.

A final example of the use of interposition (suggested by Rosenthal [Ros90] and Skinner [SW93]) is for attachment of new file-systems into the namespace. Operations directed at the interposed-upon layer are reflected up to the interposing layer. When a file system is mounted (attached to the namespace) in Unix, actions to the mounted-on directory must be reflected to the root of the new file-system. (Similarly, the root of the new file-system is also interposed upon to handle lookups on its parent directory.)

Chapter 4

UCLA Stacking Implementation

The UCLA stackable layers interface and its environment are the results of our efforts to tailor file-system development to the stackable model. Sun's vnode interface is extended to provide extensibility, stacking, and address-space independence. We describe this implementation here, beginning with a summary of the vnode interface and then examining important differences in our stackable interface.

4.1 Existing File-System Interfaces

Sun's vnode interface is a good example of several "file-system switches" developed for the Unix operating system [Kle86, RKH86]. All have the same goal, to support multiple file-system types in the same operating system. The vnode interface has been quite successful in this respect, providing dozens of different filing services in several versions of Unix.

The vnode interface is a method of abstracting the details of a file-system implementation from the majority of the kernel. The kernel views file access through two abstract data types. A *vnode* identifies individual files. A small set of file types is supported, including regular files, which provide an uninterpreted array of bytes for user data, and directories which list other files. Directories include references to other directories, forming a hierarchy of files. For implementation reasons, the directory portion of this hierarchy is typically limited to a strict tree structure.

The other major data structure is the *vfs*, representing groups of files. For configuration purposes, sets of files are grouped into *subtrees* (traditionally referred to as file systems or disk partitions), each corresponding to one vfs. Subtrees are added to the file-system namespace by *mounting*.

Mounting is the process of adding new collections of files into the global file-system namespace. Figure 4.1 shows two subtrees: the root subtree, and another attached under `/usr`. Once a subtree is mounted, name translation proceeds automatically across subtree boundaries, presenting the user with an apparently seamless namespace.

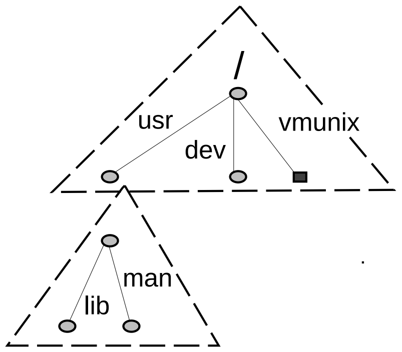

Figure 4.1: A namespace composed of two subtrees.

All files within a subtree typically have similar characteristics. Traditional Unix disk partitions correspond one-to-one with subtrees. When NFS is employed, each collection of files from a remote machine is assigned a corresponding subtree on the local machine. Each subtree is allowed a completely separate implementation.

Data encapsulation requires that these abstract data types for files and subtrees be manipulated only by a restricted set of operations. The operations supported by vnodes, the abstract data type for "files", vary according to implementation (see [Kle86] and [KM86] for semantics of typical operations).

To allow this generic treatment of vnodes, binding of desired function to correct implementation is delayed until kernel initialization. This is implemented by associating with each vnode type an *operations vector* identifying the correct implementation of each operation for that vnode type. Operations can then be invoked on a given vnode by looking up the correct operation in this vector (this mechanism is analogous to typical implementations of C++ virtual class method invocation).

Limited file-system stacking is possible with the standard vnode interface using the mount mechanism. Sun Microsystems' NFS [SGK85], loopback, and translucent [Hen90] file-systems take this approach. Information associated with the mount command identifies the existing stack layer and where the new layer should be attached into the filing name space.

4.2 Extensibility in the UCLA Interface

Accommodation of interface evolution is a critical problem with existing interfaces. Incompatible change and the lock-step release problem [Web93] are serious concerns of developers today. The ability to add to the set of filing services without disrupting existing practices is a requirement of diverse third-party filing development and would greatly ease vendor evolution of existing systems.

The vnode interface allows that the association of an operation with its implementation be delayed until run-time by fixing the formal definition of all permissible operations before kernel compilation. This convention prohibits the addition of new operations at kernel link time or during execution, since file systems have no method of insuring interface compatibility after change.

The UCLA interface addresses this problem of extensibility by maintaining all interface definition information until execution begins, and then dynamically constructing the interface. Each file-system provides a list of all the operations it supports. At kernel initialization, the union of these operations is taken, yielding the list of all operations supported by this kernel. This set of operations is then used to define the global operations vector dynamically, adapting it to arbitrary additions.[1] Vectors customized to each file-system are then constructed, caching information sufficient to permit very rapid operation invocation. During operation these vectors select the correct implementation of each operation for a given vnode. Thus, each file-system may include new operations, and new file-systems can be added to a kernel with a simple reconfiguration.

New operations may be added by any layer. Because the interface does not define a fixed set of operations, a new layer must expect "unsupported" operations and accommodate them consistently. The UCLA interface requires a *default* routine which will be invoked for all operations not otherwise provided by a file system. File systems may simply return an "unsupported operation" error code, but we expect most layers to pass unknown operations to a lower layer for processing.

The new structure of the operations vector also requires a new method of operation invocation. The calling sequence for new operations replaces the static offset into the operations vector of the old interface with a dynamically computed new offset. These changes have very little performance impact, an important consideration for a service that will be as frequently employed as an inter-layer interface. Section 5.1 discusses performance of stackable layering in detail.

4.3 Stack Creation

This section discusses how stacks are formed. In the prototype interface, stacks are configured at the file-system granularity, and constructed as required on a file-by-file basis.

4.3.1 Stack configuration

Section 4.1 described how a Unix file-system is built from a number of individual subtrees by mounting. Subtrees are the basic unit of file-system configuration; each is either mounted making all its files accessible, or unmounted and unavailable. We employ this same mechanism for layer construction.

Fundamentally, the Unix mount mechanism has two purposes: it creates a new "subtree object" of the requested type, and it attaches this object into the file-system name-space for later use. Frequently, creation of subtrees uses other objects in the file system. An example of this is shown in Figure 4.2 where a new UFS is instantiated from a disk device (`/layer/ufs/crypt.raw` from `/dev/sd0g`).

Configuration of layers requires the same basic steps of layer creation and naming, so we employ the same mount mechanism for layer construction.[2] Layers are

[1] For simplicity, we ignore here the problem of adding new operations at run-time. This "fully dynamic" addition of operations can be supported with traditional approaches to run-time extensions. Either operation vectors can reserve additional space for run-time operations at configuration time, or vectors can be reallocated while "in-use".

[2] Although mount is typically used today to provide "expensive" services, the mechanism is not inherently costly. Mount constructs an object and gives it a name; when object initialization is inexpensive, so

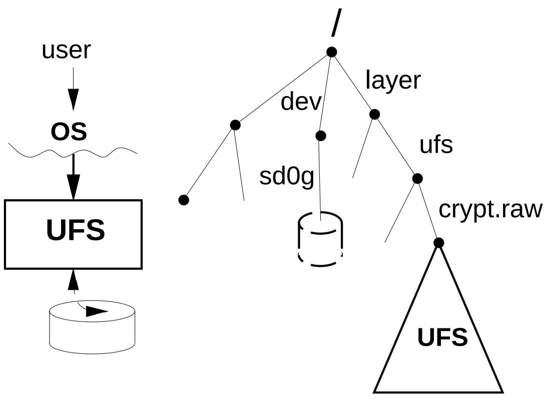

Figure 4.2: Instantiating a UFS with the Unix mount mechanism. The new layer is instantiated at `/layer/ufs/crypt.raw` from a disk device `/dev/sd0g`.

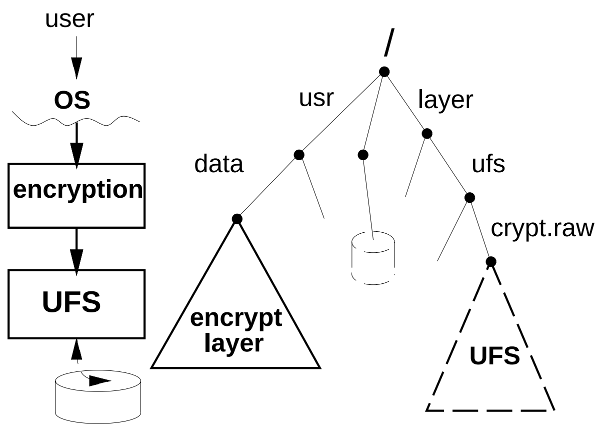

Figure 4.3: Instantiating an encryption layer over an existing UFS. The encryption layer `/usr/data` is instantiated form an existing UFS layer `/layer/ufs/crypt.raw`.

built at the subtree granularity, a mount command creating each layer of a stack. Typically, stacks are built bottom up. After a layer is mounted to a name, the next higher layer's mount command uses that name to identify its "lower-layer neighbor" in initialization. Figure 4.3 continues the previous example by stacking an encryption layer over the UFS. In this figure, an encryption layer is created with a new name (`/usr/data`) after specifying the lower layer (`/layer/ufs/crypt.raw`). Alternatively, if no new name is necessary or desired, the new layer can be mounted to the same place in the namespace.[3] Stacks with fan-out typically require that each lower layer be named when constructed.

Stack construction does not necessarily proceed from the bottom up. Sophisticated file-systems may create lower layers on demand. The Ficus distributed file-system takes this approach in its use of volumes. Each volume is a subtree storing related files. To insure that all sites maintain a consistent view about the location of the thousands of volumes in a large-scale distributed system, volume mount information is maintained on disk at the mount location. When a volume mount point is encountered during path name translation, the corresponding volume (and lower stack layers) is automatically located and mounted.

is the corresponding mount.

[3]Mounts to the same name are currently possible only in 4.4BSD-derived systems. If each layer is separately named, standard access control mechanisms can be used to mediate access to lower layers.

4.3.2 File-level stacking

While stacks are configured at the subtree level, most user actions take place on individual files. Files are represented by vnodes, with one vnode per layer.

When a user opens a new file in a stack, a vnode is constructed to represent each layer of the stack. User actions begin in the top stack layer and are then forwarded down the stack as required. If an action requires creation of a new vnode (such as referencing a new file), then as the action proceeds down the stack, each layer will build the appropriate vnode and return its reference to the layer above. The higher layer will then store this reference in the private data of the vnode it constructs. Should a layer employ fan-out, each of its vnodes will reference several lower-level vnodes similarly.

In Figure 4.4, the file stack of vnodes is shown paralleling the stack of file-system layers. If this file were created (for example, by `vop_create`), the operation would proceed down the stack to the UFS layer, which would construct vnode u1. As the operation returns to the encryption layer, it would build and return vnode e1 to the user.

Since vnode references are used both to bind layers and to access files from the rest of the kernel, no special provision need be made to perform operations between layers. The same operations used by the general kernel can be used between layers; layers treat all incoming operations identically. In Figure 4.4 the reference binding

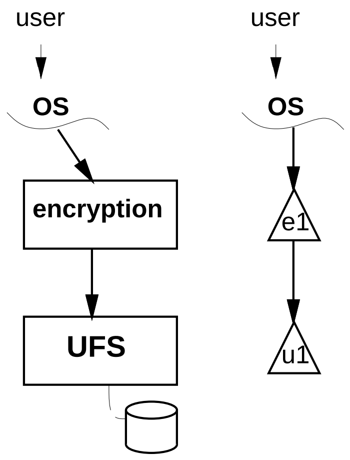

Figure 4.4: File-level stacking.

the user to vnode e1 is the same as that joining vnodes e1 and u1.

Although the current implementation does not explicitly support stack configuration at a per-file granularity, there is nothing in the model which prohibits finer configuration control. To divorce UCLA stacking from the mount-model of configuration, a "typing" layer would identify each file's configuration and then construct the corresponding vnode stack when the file is accessed. Kim's object-oriented filing system (under development at UCLA) represents approach to per-file typing [Kim95].

4.3.3 Stack data caching

When the same data is cached in different stack layers, cache incoherence becomes possible. UCLA stacking employs a cache-coherence protocol described in Chapters 6 and 7.

4.4 Stacking and Extensibility

One of the most powerful features of a stackable interface is that layers can be stacked together, each adding functionality to the whole. Often layers in the middle of a stack will modify only a few operations, passing most to the next lower layer unchanged. For example, although an encryption layer would encrypt and decrypt all data accessed by read and write requests, it may not need to modify operations for directory manipulation. Since the inter-layer interface is extensible and therefore new operations may always be added, an intermediate layer must be prepared to forward arbitrary, new operations.

One way to pass operations to a lower layer is to implement, for each operation, a routine that explicitly invokes the same operation in the next lower layer. This approach would fail to adapt automatically to the addition of new operations, requiring modification of all existing layers when any layer adds a new operation. The creation of new layers and new operations would be discouraged, and the use of unmodified third-party layers in the middle of new stacks would be impossible.

What is needed is a single *bypass* routine which forwards new operations to a lower level. Default routines (discussed in Section 4.2) provide the capability to have a generic routine intercept unknown operations, but the standard vnode interface provides no way to process this operation in a general manner. To handle multiple operations, a single routine must be able to handle the variety of arguments used by different operations. It must also be possible to identify the operation taking place, and to map any vnode arguments to their lower level counterparts.[4]

Neither of these characteristics are possible with existing interfaces where operations are implemented as standard function calls. And, of course, support for these characteristics must have absolutely minimal performance impact.

The UCLA interface accommodates these characteristics by explicitly managing operations' arguments as collections. In addition, meta-data is associated with each collection, providing the operation identity, argument types, and other pertinent information. Together, this explicit management of operation invocations allows arguments to be manipulated in a generic fashion and efficiently forwarded between layers, usually with pointer manipulation.

These characteristics make it possible for a simple bypass routine to forward all operations to a lower layer in the UCLA interface. By convention, we expect most file-system layers to support such a bypass routine. More importantly, these changes to the interface have minimal impact on performance. For example, passing meta-data requires only one additional argument to each operation. See Section 5.1 for a detailed analysis of performance.

Appendix A.1 shows a C-based implementation of `vop_create` with both the Sun- and UCLA-vnode inter-

[4] Vnode arguments change as a call proceeds down and then back up the stack, much as protocol headers are stripped off as network messages are processed. No other argument processing is required in order to bypass an operation between two layers in the same address space.

faces. Appendix A.3 shows a bypass routine.

4.5 VFS Stacking and Extensibility

Thus far we have focused on the vnode interface and how it can be modified to support stacking and extensibility. As the vnode interface implements the file abstraction, the VFS interface provides an abstraction for file systems and layers. The VFS interface provides a much smaller set of services (shown in Table 4.1), but many of the reasons which motivate extensibility of filing operations also suggest extensibility of file-system operations.

Of these operations, the first four (mount, unmount, mountroot, and swapvp) should not be stacked-upon. To provide stacking for the remainder of the operations, we re-implement them as vnode operations. When implementing a VFS operation as a vnode operation, we simply replace the vfs argument with a vnode; an vnode from the layer can represent the vfs. New vfs operations can be implemented as vnode operations in the same way, and so take advantage of vnode operation extensibility.

An alternative to this approach would be to provide extensibility for the VFS interface itself. The small number of VFS operations suggests to us that in many cases the approach taken in our prototype is preferable.

4.6 Inter-Machine Operation

A *transport layer* is a stackable layer that transfers operations from one address space to another. Because vnodes for both local and remote file-systems accept the same operations, they may be used interchangeably, providing network transparency. Sections 3.5 and 3.6 describe some of the layer configurations which this transparency allows.

Providing a bridge between address spaces presents several potential problems. Different machines might have differently configured sets of operations. Heterogeneity can make basic data types incompatible. Finally, methods to support variable length and dynamically allocated data structures for traditional kernel interfaces do not always generalize when crossing address space boundaries.

For two hosts to inter-operate, it must be possible to identify each desired operation unambiguously. Well-defined RPC protocols such as NFS insure compatibility by providing only a fixed set of operations. Since restricting the set of operations frequently restricts and impedes innovation, each operation in the UCLA interface is assigned a universally unique identifier when it is defined.[5] Inter-machine communication of arbitrary operations uses these labels to reject locally unknown operations.

Transparent forwarding of operations across address space boundaries requires not only that operations be identified consistently, but also that arguments be communicated correctly in spite of machine heterogeneity. Part of the meta-data associated with each operation includes a complete type description of all arguments. With this information, an RPC protocol can marshal operation arguments and results between heterogeneous machines. Thus a transport layer may be thought of as a semantics-free RPC protocol with a stylized method of marshaling and delivering arguments.

NFS provides a good prototype transport layer. It stacks on top of existing local file-systems, using the vnode interface above and below. But NFS was not designed as a transport layer; its supported operations are not extensible and its implementations define particular caching semantics. We extend NFS to automatically bypass new operations. We have also prototyped a cache consistency layer providing a separate consistency policy.

We have tried two approaches to export operation descriptions to NFS. In our first implementation each new operation was accompanied with a pointer to a procedure which would marshal data into the canonical representation for NFS (external data representation, or XDR [Sun87]). This approach provides support for machine heterogeneity, but it supports only a single protocol (XDR); protocol heterogeneity is not addressed. To relax this constraint, our second implementation lists hierarchically the type of each data object. This hierarchy is interpreted as the data is marshaled. Each network protocol can provide its own interpreter, allowing our internal data description to service multiple networking protocols.

In addition to the use of an NFS-like inter-address space transport layer, we employ a more efficient transport layer operating between the user and the kernel level. Such a transport layer provides "system call" level access to the UCLA interface, allowing user-level development of file-system layers and providing user-level access to new file-system functionality. The desire to support a system-call-like transport layer placed one additional constraint on the interface. Traditional system calls expect the user to provide space for all re-

[5]Generation schemes based on host identifier and time-stamp support fully distributed identifier creation and assignment. We therefore employ the NCS UUID mechanism.

operation	**description**
`vfs_mount`	Configure a new file system or layer.
`vfs_unmount`	Remove an existing file system or layer.
`vfs_mountroot`	Configure the root file-system.
`vfs_swapvp`	Create a vnode corresponding to a file-identifying token.
`vfs_statfs`	Return statistics about the file system or layer.
`vfs_sync`	Flush any pending I/Os to backing store.
`vfs_rootvp`	Return a vnode for the file-system root.
`vfs_vget`	Create a vnode corresponding to a file-identifying token.

Table 4.1: VFS operations provided in SunOS 4.x.

turned data. We have chosen to extend this restriction to the UCLA interface to make the user-to-kernel transport layer universal. In practice, this restriction has not been serious since the client can often make a good estimate of storage requirements. If the client's first guess is wrong, information is returned, allowing the client to correctly repeat the operation.

4.7 Centralized Interface Definition

Several aspects of the UCLA interface require precise information about the characteristics of the operation taking place. Network transparency requires a complete definition of all operation types (as described above), and a bypass routine must be able to map vnodes from one layer to the next (as described in Appendix A.3). The designer of a file system employing new operations must provide this information.

Detailed interface information is needed at several different places throughout the layers. Rather than require that the interface designer keep this information consistent in several different places, operation definitions are combined into an *interface definition*. Similar to the data description language used by RPC packages, this description lists each operation, its arguments, and the direction of data movement. An interface compiler translates this into forms convenient for automatic manipulation.

Appendix A.2 shows the interface definition of `vop_create`.

4.8 Framework Portability

The UCLA interface has proven to be quite portable. Initially implemented under SunOS 4.0.3, it has since been ported to SunOS 4.1.1. In addition, the in-kernel stacking and extensibility portions of the interface have been ported to 4.4BSD. Although BSD's *namei* approach to pathname translation required some change, we are largely pleased with our framework's portability to a system with an independently derived vnode interface. Section 5.3 discusses portability of individual layers.

While the UCLA interface itself has proven to be portable, portability of individual layers is somewhat more difficult. None of the implementations described have identical sets of vnode operations, and pathname translation approaches differ considerably between SunOS and BSD.

Fortunately, several aspects of the UCLA interface provide approaches to address layer portability. Extensibility allows layers with different sets of operations to co-exist. In fact, interface additions from SunOS 4.0.3 to 4.1.1 required no changes to existing layers. When interface differences are significantly greater, a compatibility layer (see Section 3.5) provides an opportunity to run layers without change. Ultimately, adoption of a standard set of core operations (as well as other system services) is required for effortless layer portability.

Chapter 5

UCLA Stacking Evaluation

While a stackable file-system design offers numerous advantages, file-system layering will not be widely accepted if layer overhead is such that a monolithic file-system performs significantly better than one formed from multiple layers. To verify layering performance, overhead was evaluated from several points of view.

If stackable layering is to encourage rapid advance in filing, it must have not only good performance, but it also must facilitate file-system development. Here we also examine this aspect of "performance", first by comparing the development of similar file-systems with and without the UCLA interface, and then by examining the development of layers in the new system.

Finally, compatibility problems are one of the primary barriers to the use of current filing abstractions. We conclude by describing our experiences in applying stacking to resolve filing incompatibilities.

5.1 Layer Performance

To examine the performance of the UCLA interface, we consider several classes of benchmarks. First, we examine the costs of particular parts of this interface with "micro-benchmarks". We then consider how the interface affects overall system performance by comparing a stackable layers kernel to an unmodified kernel. Finally we evaluate the performance of multi-layer file-systems by determining the overhead as the number of layers changes.

Measurements in this chapter were collected from a machine running a modified version of SunOS 4.0.3. All benchmarks were run on a Sun-3/60 with 8 Mb of RAM and two 70 Mb Maxtor XT-1085 hard disks. This machine is rated at 3 MIPS (it predates the SPEC benchmarks). The measurements in Section 5.1.2 used the new interface throughout the new kernel, while those in Section 5.1.3 used it only within file systems.

5.1.1 Micro-benchmarks

The new interface changes the way every file-system operation is invoked. To minimize overhead, operation calls must be very inexpensive. Here we discuss two portions of the interface: the method for calling an operation, and the bypass routine. Cost of operation invocation is key to performance, since it is an unavoidable cost of stacking no matter how layers themselves are constructed.

To evaluate the performance of these portions of the interface, we consider the number of assembly language instructions generated in the implementation. While this statistic is only a very rough indication of true cost, it provides an order-of-magnitude comparison.[1]

We began by considering the cost of invoking an operation in the vnode and the UCLA interfaces. On a Sun-3 platform, the original vnode calling sequence translates into four assembly language instructions, while the new sequence requires six instructions.[2] We view this overhead as not significant with respect to most file-system operations.

We are also interested in the cost of the bypass routine. We envision a number of "filter" file-system layers, each adding new abilities to the file-system stack. File compression or local disk caching are examples of services such layers might offer. These layers pass many operations directly to the next layer down, modifying the user's actions only to uncompress a compressed file, or to bring a remote file into the local disk cache. For such layers to be practical, the bypass routine must be inexpens-

[1]Factors such as machine architecture and the choice of compiler have a significant impact on these figures. Many architectures have instructions which are significantly slower than others. We claim only a rough comparison from these statistics.

[2]We found a similar ratio on SPARC-based architectures, where the old sequence required five instructions, the new eight. In both cases these calling sequences do not include code to pass arguments of the operation.

ive. A complete bypass routine in our design amounts to about 54 assembly language instructions.[3] About one-third of these instructions are not in the main flow, being used only for uncommon argument combinations, reducing the cost of forwarding simple vnode operations to 34 instructions. Although this cost is significantly more than a simple subroutine call, it is not significant with respect to the cost of an average file-system operation. To further investigate the effects of file-system layering, Section 5.1.3 examines the overall performance impact of a multi-layered file-system.

5.1.2 Interface performance

While instruction counts are useful, actual implementation performance measurements are essential for evaluation. The first step compares a kernel supporting only the UCLA interface with a standard kernel.

To do so, we consider two benchmarks: the modified Andrew benchmark [Ous90, HKM88] and the recursive copy and removal of large subdirectory trees. In addition, we examine the effect of adding multiple layers in the new interface.

The Andrew benchmark has several phases, each of which examines different file-system activities. Unfortunately, the brevity of the first four phases relative to granularity makes accuracy difficult. In addition, the long compile phase dominates overall benchmark results. Nevertheless, taken as a whole, this benchmark probably characterizes "normal use" better than a file-system intensive benchmark such as a recursive copy/remove.

The results from the benchmark can be seen in Table 5.1. Overhead for the first four phases averages about two percent. Coarse timing granularity and the very short run times for these benchmarks limit their accuracy. The compile phase shows only a slight overhead. We attribute this lower overhead to the fewer number of file system operations done per unit time by this phase of the benchmark.

To exercise the interface more strenuously, we examined recursive copy and remove times. This benchmark employed two phases, the first doing a recursive copy and the second a recursive remove. Both phases operate on large amounts of data (a 4.8 Mb `/usr/include` directory tree) to extend the duration of the benchmark. Because we knew all overhead occurred in the kernel, we measured system time (time spent in the kernel) instead of total elapsed time. This greatly exaggerates the impact of layering, since all overhead is in the kernel and system time is usually small compared to the elapsed "wall clock" time a user actually experiences. As can be seen in Table 5.2, system time overhead averages about 1.5%.

5.1.3 Multiple layer performance

Since the stackable layers design philosophy advocates using several layers to implement what has traditionally been provided by a monolithic module, the cost of layer transitions must be minimal if layering is to be used for serious file-system implementations. To examine the overall impact of a multi-layer file-system, we analyze the performance of a file-system stack as the number of layers employed changes.

To perform this experiment, we began with a kernel modified to support the UCLA interface within all file systems and the vnode interface throughout the rest of the kernel.[4] At the base of the stack we placed a Berkeley fast file-system, modified to use the UCLA interface. Above this layer we mounted from zero to six null layers, each of which merely forwards all operations to the next layer of the stack. We ran the benchmarks described in the previous section upon those file-system stacks. This test is by far the worst possible case for layering since each added layer incurs full overhead without providing any additional functionality.

Figure 5.1 shows the results of this study. Performance varies nearly linearly with the number of layers used. The modified Andrew benchmark shows about 0.3% elapsed time overhead per layer. Alternate benchmarks, such as the recursive copy and remove phases, also show less than 0.25% overhead per layer.

To get a better feel for the costs of layering, we also measured system time, time spent in the kernel on behalf of the process. Figure 5.2 compares recursive copy and remove system times (the modified Andrew benchmark does not report system time statistics). Because all overhead is in the kernel, and the total time spent in the kernel is only one-tenth of elapsed time, comparisons of system time indicate a higher overhead: about 2% per layer for recursive copy and remove. Slightly better performance for the case of one layer in Figure 5.2 results from a slight caching effect of the null layer compared to the standard UFS. Differences in benchmark overheads are the result of differences in the ratio between the number of vnode operations and benchmark length.

[3]These figures were produced by the Free Software Foundation's gcc compiler. Sun's C compiler bundled with SunOS 4.0.3 produced 71 instructions.

[4]To improve portability, we desired to modify as little of the kernel as possible. Mapping between interfaces occurs automatically upon first entry of a file-system layer.

	vnode interface		UCLA interface		percent
phase	**time**	**%RSD**	**time**	**%RSD**	**overhead**
MakeDir	3.3	16.1	3.2	14.8	−3.03
Copy	18.8	4.7	19.1	5.0	1.60
ScanDir	17.3	5.1	17.8	7.9	2.89
ReadAll	28.2	1.8	28.8	2.0	2.13
Make	327.1	0.4	328.1	0.7	0.31
Overall	394.7	0.4	396.9	0.9	0.56

Table 5.1: Modified Andrew benchmark results running on kernels using the vnode and the UCLA interfaces. Time values (in seconds, timer granularity one second) are the means of elapsed time from 29 sample runs; %RSD indicates the percent relative standard deviation (σ_X / μ_X). Overhead is the percent overhead of the new interface. High relative standard deviations for MakeDir are a result of poor timer granularity.

	vnode interface		UCLA interface		percent
phase	**time**	**%RSD**	**time**	**%RSD**	**overhead**
recursive copy	51.57	1.28	52.55	1.11	1.90
recursive remove	25.26	2.50	25.41	2.80	0.59
overall	76.83	0.87	77.96	1.11	1.47

Table 5.2: Recursive copy and remove benchmark results running on kernels using the vnode and UCLA interfaces. Time values (in seconds, timer granularity 0.1 second) are the means of system time from twenty sample runs; %RSD indicates the percent relative standard deviation. Overhead is the percent overhead of the new interface.

We draw two conclusions from these figures. First, elapsed time results indicate that under normal load usage, a layered file-system architecture will be virtually undetectable. Also, system time costs imply that during heavy file-system use a small overhead will be incurred when numerous layers are involved.

5.2 Layer Implementation Effort

An important goal of stackable file-systems and this interface is to ease the job of new file-system development. Importing functionality with existing layers saves a significant amount of time in new development, but this savings must be compared to the effort required to employ stackable layers. The next three sections compare development with and without the UCLA interface, and examine how layering can be used for both large and small filing services. We conclude that layering simplifies both small and large projects.

5.2.1 Simple layer development

A first concern when developing new file-system layers was that the process would prove to be more complicated than development of existing file-systems. Most other kernel interfaces do not support extensibility; would this facility complicate implementation?

To evaluate complexity, we choose to examine the size of similar layers implemented both with and without the UCLA interface. A simple "pass-through" layer was chose for comparison: the loopback file-system under the traditional vnode interface, and the null layer under the UCLA interface.[5] We performed this comparison for both the SunOS 4.0.3 and the 4.4BSD implementations, measuring complexity as numbers of lines of comment-free C code.[6]

Table 5.3 compares the code length of each service in the two operating systems. Closer examination revealed that the majority of code savings occurs in the implementation of individual vnode operations. The null layer implements most operations with a bypass routine, while the loopback file-system must explicitly forward each operation. In spite of a smaller implementation, the services provided by the null layer are also more general; the same implementation will support the addition of future operations.

For the example of a pass-through layer, use of the

[5] In SunOS the null layer was augmented to exactly reproduce the semantics of the loopback layer. This was not necessary in 4.4BSD.

[6] While well-commented code might be a better comparison, the null layer was quite heavily commented for pedagogical reasons, while the loopback layer had only sparse comments. We chose to eliminate this variable.

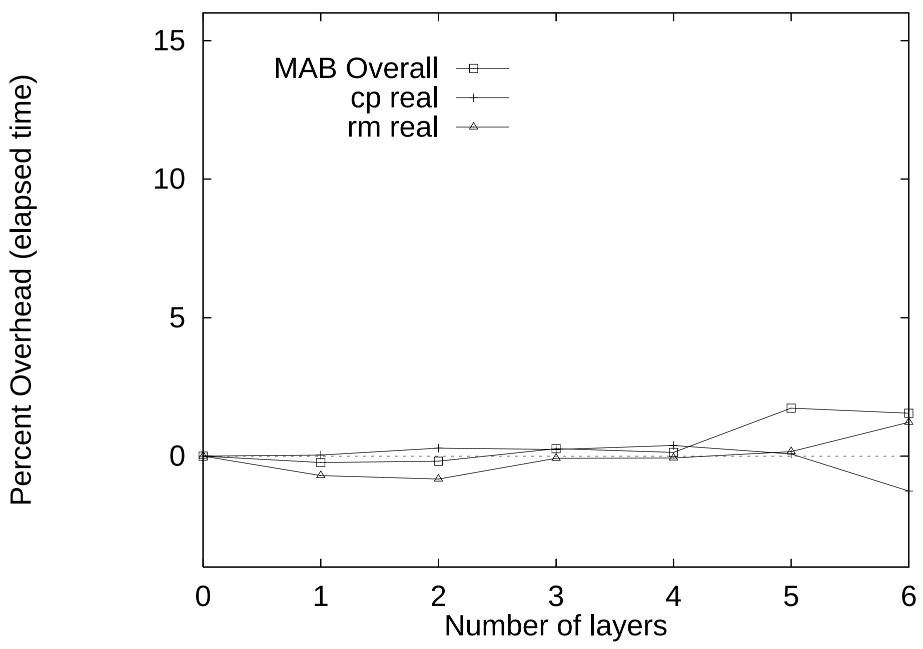

Figure 5.1: Elapsed time of recursive copy/remove and modified Andrew benchmarks as layers are added to a file-system stack. Each data point is the mean of four runs.

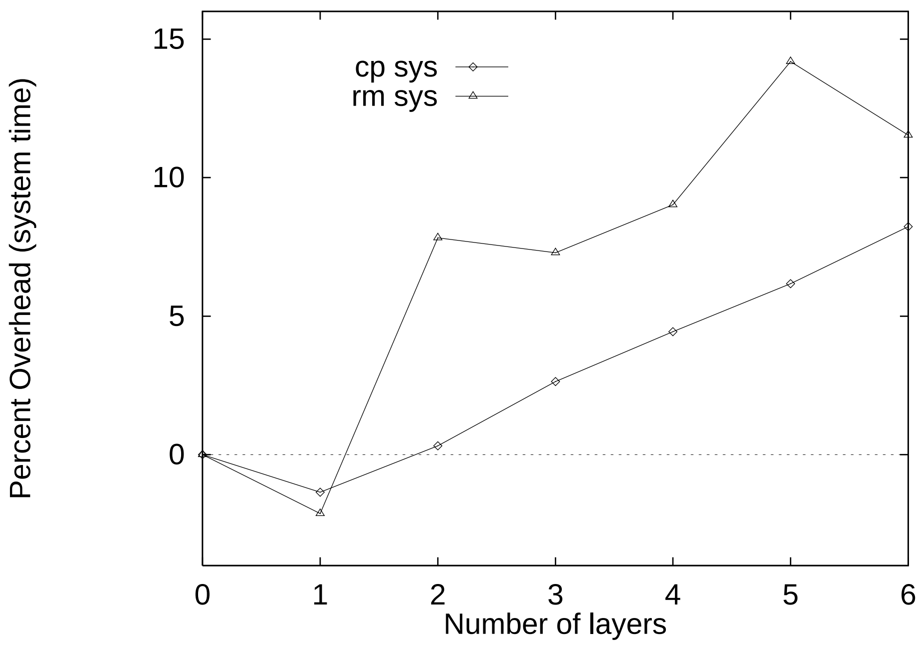

Figure 5.2: System time of recursive copy/remove benchmarks as layers are added to a file-system stack (the modified Andrew benchmark does not provide system time). Each data point is the mean of four runs. Measuring system time alone of a do-nothing layer represents the worst possible layering overhead.

	SunOS	BSD
loopback-fs	743 lines	1046 lines
null layer	632 lines	578 lines
difference	–111 lines	–468 lines
	–15%	–45%

Table 5.3: Number of lines of comment-free code needed to implement a pass-through layer or file system in SunOS 4.0.3 and 4.4BSD.

UCLA interface enabled improved functionality with a smaller implementation. Although the relative difference in size would be less for single layers providing multiple services, a goal of stackable layers is to provide sophisticated services through multiple, reusable layers. This goal requires that minimal layers be as simple as possible.

We are currently pursuing strategies to further reduce the absolute size of null layer code. We expect to unify vnode management routines for null-derived layers, centralizing this common service.

5.2.2 Layer development experience

The best way to demonstrate the generality of a new design technique is through its use by different parties and in application to different problems.

To gain more perspective on this issue students were invited to design and develop new layers as part of a graduate class at UCLA. While all were proficient programmers, their kernel programming experience ranged from none to considerable. Five groups of one or two students each were provided with a null layer and a user-level development environment.

All projects succeeded in provided functioning prototype layers. Prototypes include a file-versioning layer, an encryption layer, a compression layer, second class replication as a layer, and an NFS consistency layer. Other than the consistency layer, each was designed to stack over a standard UFS layer, providing its service as an optional enhancement. Self-estimates of development time ranged from 40 to 60 person-hours. This figure included time to become familiar with the development environment, as well as layer design and implementation.

Review of the development of these layers suggested three primary contributions of stacking to this experiment. First, by relying on a lower layer to provide basic filing services, detailed understanding of these services was unnecessary. Second, by beginning with a null layer, new implementation required was largely focused on the problem being solved rather than peripheral framework issues. Finally, the out-of-kernel layer development platform provided a convenient, familiar environment compared to traditional kernel development.

We consider this experience a promising indication of the ease of development offered by stackable layers. Previously, new file-system functionality required in-kernel modification of current file-systems, and therefore knowledge of multi-thousand-line file-systems and low-level kernel debugging tools. With stackable layers, students in the class were able to investigate significant new filing capabilities with knowledge only of the stackable interface and programming methodology.

5.2.3 Large-scale example

The previous section discussed our experiences in stackable development of several prototype layers. This section concludes with the the results of developing a replicated file-system suitable for daily use.

Ficus is a “real” system, both in terms of size and use. It is comparable in code size to other production file-systems (12,000 lines for Ficus compared to 7–8,000 lines of comment-free NFS or UFS code). Ficus has seen extensive development over its three-year existence. Its developers’ computing environment (including Ficus development) is completely supported in Ficus, and it is now in use at various sites in the United States.

Stacking has been a part of Ficus from its very early development. Ficus has provided both a fertile source of layered development techniques, and a proving ground for what works and what does not.

Ficus makes good use of stackable concepts such as extensibility, cooperating layers, an extensible transport layer, and out-of-kernel development. Extensibility is widely used in Ficus to provide replication-specific operations. The concept of cooperating layers is fundamental to the Ficus architecture, where some services must be provided “close” to the user while others must be close to data storage. Between the Ficus layers, the optional transport layer has provided easy access to any replica, leveraging location transparency as well. Finally, the out-of-kernel debugging environment has proved particularly important in early development, saving significant development time.

As a full-scale example of the use of stackable layering and the UCLA interface, Ficus illustrates the success of these tools for file-system development. Layered file-systems can be robust enough for daily use, and the development process is suitable for long-term projects.

5.3 Compatibility Experiences

Extensibility and layering are powerful tools to address compatibility problems. Section 3.5 discusses several different approaches to employ these tools; here we consider how effective these tools have proven to be in practice. Our experiences here primarily concern the use and evolution of the Ficus layers, the user-id mapping and null layers, and stack-enabled versions of NFS and UFS.

Extensibility has proven quite effective in supporting "third party"-style change. The file-system layers developed at UCLA evolve independently of each other and of standard filing services. Operations are frequently added to the Ficus layers with minimal consequences on the other layers. We have encountered some cache consistency problems resulting from extensibility and our transport layer. Chapters 6 and 7 discuss our approach to cache coherence. Without extensibility, each interface change would require changes to all other layers, greatly slowing progress.

We have had mixed experiences with portability between different operating systems. On the positive side, Ficus is currently accessible from PCs running MS-DOS (see Figure 5.3). The PC runs an NFS implementation to communicate with a Unix host running Ficus. Ficus requires more information to identify files than will fit in an NFS file identifier, so we employ an additional "shrinkfid" layer to map over this difference.

Actual portability of layers between the SunOS and BSD stacking implementations is more difficult. Each operating system has a radically different set of core vnode operations and related services. For this reason, and because of licensing restrictions, we chose to reimplement the null and user-id mapping layers for the BSD port. Although we expect that a compatibility layer could mask interface differences, long-term interoperability requires not only a consistent stacking framework but also a common set of core operations and related operating system services.

Finally, we have had quite good success employing simple compatibility layers to map over minor interface differences. The shrinkfid and umap layers each correct deficiencies in interface or administrative configuration. We have also constructed a simple layer which passes additional state information (opens and closes) through extensible NFS as new operations.

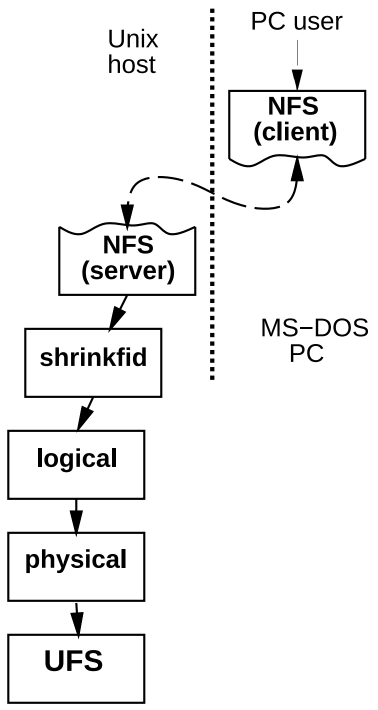

Figure 5.3: Access to Unix-based Ficus from a PC running MS-DOS. NFS bridges operating system differences; the shrinkfid layer addresses minor internal interface differences.

5.4 Summary

This chapter evaluated the performance of file-system layering, considering the performance of individual layers and file-system stacks. It also considered how layering can improve the file-system development by allowing code reuse and out-of-kernel development. To summarize the development environment, consider the comments of one of the students who developed a file-system layer [Kue91]:

> For me, the really big advantage of the stackable layers was the ease of development. Combined with the ook [out-of-kernel] development, the testing cycle was vastly shorter than other kernel work I've done. I could compile, mount, debug, and unmount in the time that it would have taken to just link a kernel, and of course I had `dbx` available instead of struggling with lousy kernel debuggers.

Chapter 6

Coherence Architecture

Nearly all file systems today provide a layer of abstraction over the raw disk geometry; it is inconceivable that any new file system would lack such an abstraction. Similarly, data caching has become a required filing technique. The performance improvements of caching are well known; a well integrated cache allows efficient use of resources. For these reasons a filing cache is part of all modern, general-purpose operating systems.

We have already argued in Section 1.1.2 that file-system caching becomes more difficult in a multi-layered filing environment. To recap briefly, a first problem is that different, independently derived layers may choose to cache the same data, and uncoordinated updates to these caches can result in data loss. The second and more general problem is that the separation of filing services into multiple layers makes it difficult for any individual layer to make assertions about the current state of file as a whole. Yet all too often caching must or will occur at different stack layers due to multi-layer access (Section 3.3).

This chapter proposes a cache-coherence protocol to address both of these problems. We begin by outlining the general approach, and then discuss design constraints, the central problem of identifying what data is to be cached, and other design issues.

6.1 General Approach to Cache Coherence

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

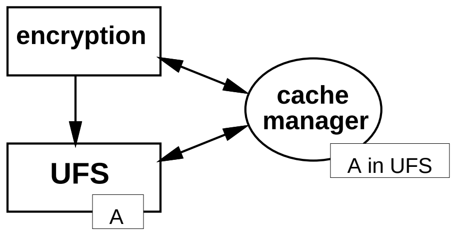

Figure 6.1: A sample application of the cache manager.

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 6.1. 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 access 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.

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. A variety of VM systems are in use. 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.

6.2 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 (DSM) 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.

6.2.1 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 7.2 discusses name selection in more detail.) Figure 6.2a 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 $\langle$ file-token, co-type, co-name $\rangle \rightarrow$ vnode allows the cache manager to determine that $\langle$ `file-2`, `attrs`, `length` $\rangle \rightarrow$ `vp-c2` and $\langle$ `file-2`, `attrs`, `length` $\rangle \rightarrow$ `vp-d2` refer to the same object and must be kept coherent. In Figure 6.2b the cache manager has recorded both vnodes of a two-vnode file as caching the file length attribute.

6.2.2 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

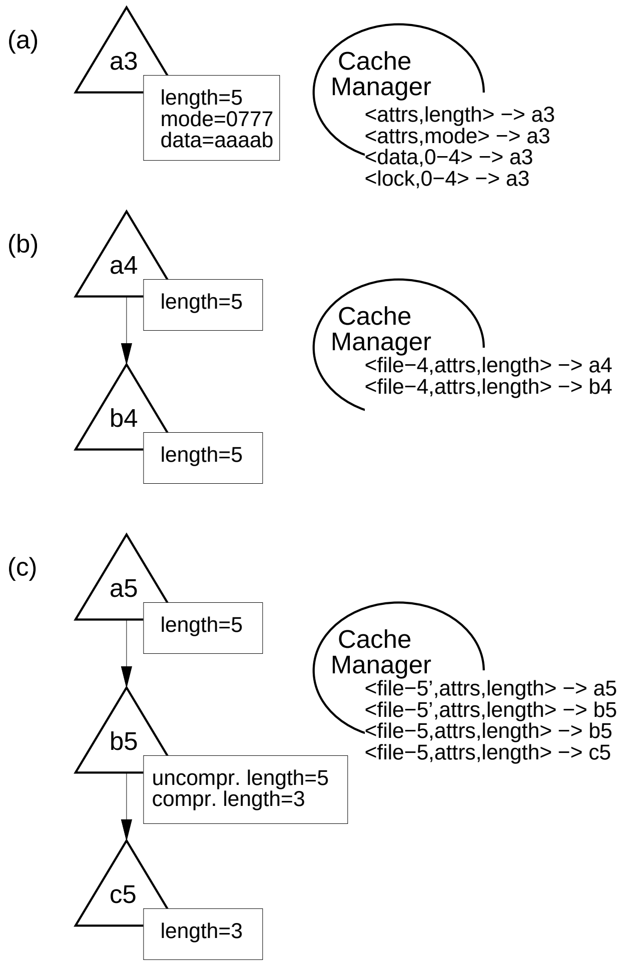

Figure 6.2: Levels of cache-object identification described in Section 6.2. In (a) a single vnode identifies cache-objects by type and name. In (b) a file-token is added. Part (c) shows how a general layer can map between different file tokens.

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

Figure 6.2c 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 7.4.

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.

6.3 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.

Appendix B.3 lists each of the requests a layer can make upon the cache manager and their interactions.

6.4 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 6.2, 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. We describe a protocol for this procedure in Section 7.3.5.

6.5 Relationship to Distributed Computing

Cache coherence in stacking as described so far will keep all layers in a single operating system coherent.[1] 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 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 6.3 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.

6.6 Summary

This chapter has explored the architecture of our cache management protocol: layers cooperate with a central

[1] 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.

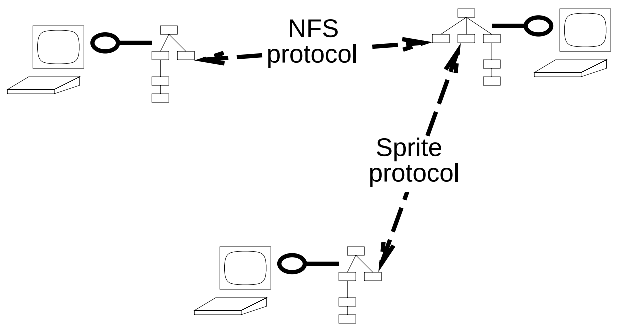

Figure 6.3: Distributed cache-coherence involving different network protocols. Cache managers maintain coherence local to each machine while different protocols are employed for inter-machine coherence.

cache manager, consistently identifying what data is to be cached and with what constraints. The real contribution of this work is not simply the centralized cache manager (which has been provided before in other environments), but a cache manager which is robust to independent layer development and semantics- and data-identity-changing layers, and which provides good performance. The next chapters examine and evaluate our implementation of this protocol.

Chapter 7

Coherence Implementation

Chapter 6 described our approach to cache coherence. An implementation of this framework is an important step in validating and evaluating this design. This chapter briefly summarizes important points of our implementation, highlighting optimizations and and other relevant implementation choices. We conclude by drawing the design and implementation together in an extended example.

7.1 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, however, 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, and the stack is unlocked.

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 B.2 presents the interfaces between the cache manager and a layer.

7.2 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.

7.2.1 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.[1]) 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

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

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 6.2.2.)

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.

7.2.2 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.

7.2.3 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.

7.3 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.

7.3.1 Data-page caching

Our approach to data-page caching is influenced by the observation that it is *not* necessary to provide a sophisticated distributed shared memory system 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 8.6. Second, we assert that it is inappropriate 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 a simple synchronization mechanism 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 7.1. 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.

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 [KN93b]. 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.

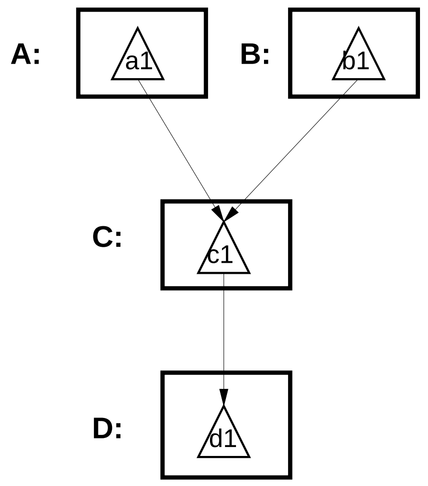

Figure 7.1: A configuration of several layers. The ovals represent layers; the figure as a whole represents a stack. Each triangle is a vnode, while each collection of joined triangles represents a file.

Read-only page replication Another possible optimization is to coordinate page access with reader/writer tokens instead of simple tokens. Reader/writer tokens allow read-only copies of pages to be replicated (possibly in different formats) in different layers of the stack concurrently, or allow a single writable page. 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 across multiple layers will be rare.

Page sharing and read-only page replication optimize for similar but not identical scenarios. For example, consider caching data in vnodes c1 and d1 of Figure 7.1. Page sharing is effective only if layer C has the same data representation as layer D, regardless of page read or write status. Page sharing also reduces memory usage. Read-only page replication is effective regardless of data representation, but only if pages are not used for updates.

7.3.2 File attribute caching

file-system layers often must alter their behavior based on file meta-data. 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 based 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.

7.3.3 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.

7.3.4 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.[2]

7.3.5 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.

As described in Section 6.4 deadlock prevention requires a one-to-one relationship between files and locks. Stack layers with fan-out can violate this relationship if they merge several files into one. In such cases, references to all locks must be replaced by references to a single lock. For example, the addition of layer D in Figure 7.2a joins files 1, 2 and 3 through the common point d4. Layer D must replace locks for files 1–3 with a single lock, perhaps that of lock 4.

In general, the procedure for lock reallocation is layer-specific. Reallocation itself must be done with some care to avoid deadlock. A general protocol is to release all lower-layer locks, re-acquire them in some canonical order, and then replace each with a reference to a new lock. For example, in Figure 7.2 we might release locks 1–3, re-acquire them in low-to-high memory address order, and then replace the references with lock 4. Significantly simpler procedures are possible in many cases, such as when a layer with general naming stacks above a single other layer.

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

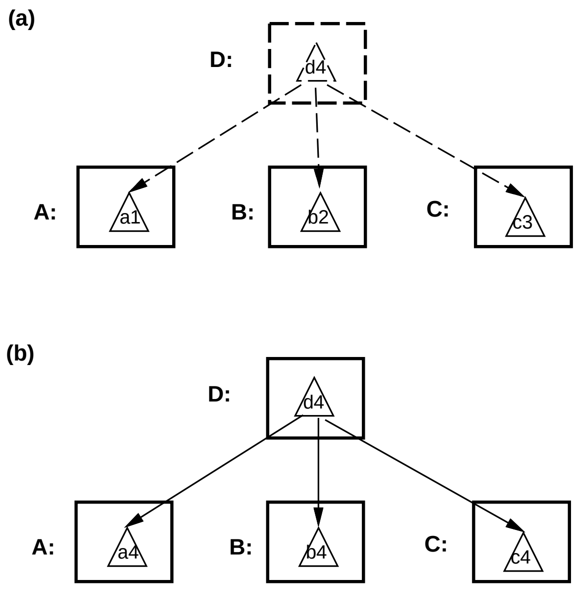

Figure 7.2: Lock merging due to layer addition.

7.4 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 6.2 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 6.2c 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.

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

Chapter 8

Coherence 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 [Wit93].) We next examine components of our coherence approach that impact performance, the benchmarks we use to examine that performance, and finally several perspectives on the performance of our system.

8.1 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 8.1. 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 8.5.

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 8.6.

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

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 7.3.1), 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 8.6.

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 8.3.

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.

8.2 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 [Ous90]. 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 8.6.

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 measurements in this chapter were taken on a Sun SPARCstation IPC (a 13.8 SPECint92 machine) 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 Filesystem [MJL84]. For multi-layer tests we add one or more null layers. 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.

8.3 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 (`vop_putpage` and `vop_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 [KN93a], 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 (`vop_putpage`, `vop_getpage`, and `vop_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 B.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.

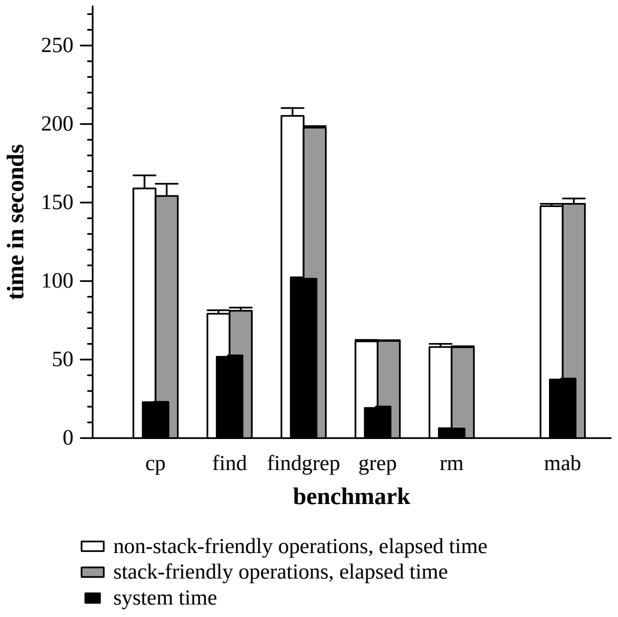

Figure 8.2: Benchmarks comparing a UFS with and without stack-friendly data acquisition. Error bars show one standard deviation. (This figure illustrates the data presented in Table 8.1.)

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 8.1 compares the standard Unix file-system with and without these changes. Figure 8.2 presents these results graphically.

A comparison of individual benchmarks from these results shows a performance difference of ±4% for different tests, and that several of the tests show no statistically significant difference. Taken as a whole the tests suggest that there is some performance variation, but there is no consistent bias for either type of data acquisition.

8.4 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 8.3 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

	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 8.1: 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 8.3 interprets this data; Figure 8.2 presents it graphically.

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 other-

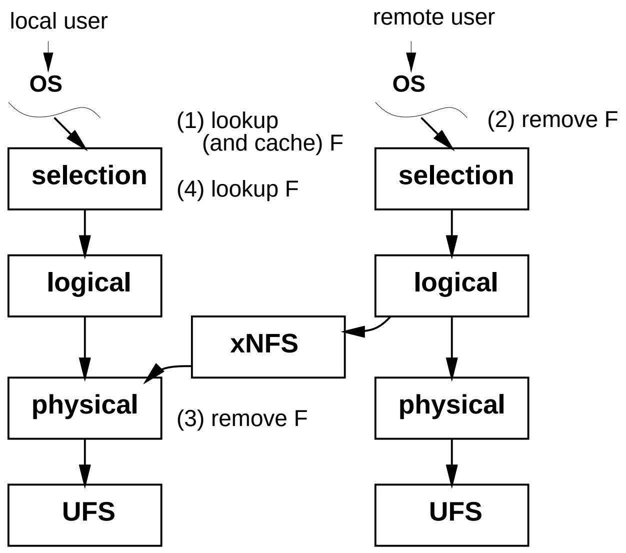

Figure 8.3: Layer configuration for Ficus replication. Each column represents a particular host. The logical layer controls access to different replicas, accessing remote replicas through stack-enabled NFS. The sequence of operations listed results in cache-coherence problems; see Section 8.4 for details.

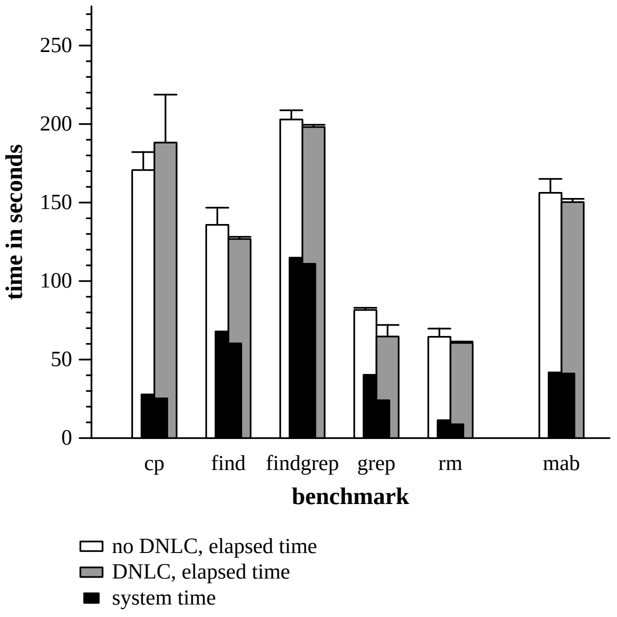

Figure 8.4: Benchmarks comparing three null layers stacked over a UFS with and without coherent name-lookup caching. (This figure illustrates the data presented in Table 8.2.)

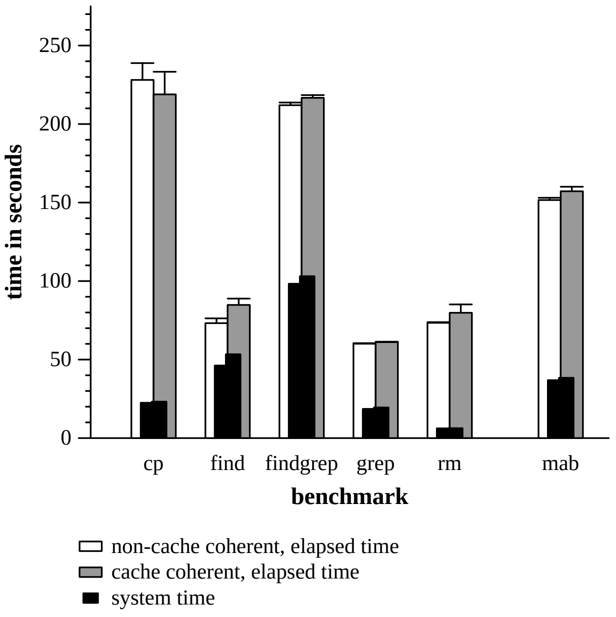

Figure 8.5: Benchmarks comparing a UFS in kernels with and without cache coherence. (This figure illustrates the data presented in Table 8.3.)

wise 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%.

8.5 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 spent in step 2 of the layered caching algorithm (Figure 8.1b).

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 8.3 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 8.5 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%).

	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 8.2: 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 8.4 characterizes this data.

	non-coherent		coherent		
benchmark	**mean**	**%RSD**	**mean**	**%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 8.3: 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 8.5.)

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 [Bon94]. 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 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.

8.6 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 8.6. 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 8.4. 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 8.4. 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 8.6 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.[1] 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 a

[1] 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.

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

		one		two			90% confidence
time	**benchmark**	**mean**	**%RSD**	**mean**	**%RSD**	**%difference**	**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	40291.00	134.17
	sequential-update	1.42	13.74	9.13	3.38	544.10	29.96

Table 8.4: 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 8.6 for discussion.

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 1.1 would be to the clear-text 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 1.1 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 [Flo86]. 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.

8.7 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.

Chapter 9

Featherweight Layer Design and Implementation

General-purpose layering has been successful at structuring file-system services. For significant new services, the 1–2% system-time overhead associated with layering is a small fraction of total costs. However, we have also argued (in Section 1.1.3) that there are numerous “thin” services that would also benefit from a layered structure. Unfortunately, the overhead observed for general-purpose layering becomes quite significant when compared to the services provided by a thin layer. This observation motivated the exploration of *featherweight layering*, a lightweight approach applicable to the structure of simple layers.

The success of featherweight layering is based on two assumptions: first, that there are a number of interesting layers that can be constructed in a restricted layering environment, and second, that a restricted layering environment can allow a more efficient implementation than a fully general system. We explore each of these assumptions in the next two sections.

Featherweight layers also have potential costs: they may complicate the layering model with two different mechanisms accomplishing a similar purpose, and the cost of any new mechanisms required for featherweight layering may counter the performance improvements that would otherwise be seen. We address the layering model in Section 9.3 and performance concerns by examining our implementation in Section 10.2.

9.1 Potential Featherweight Layers

Ideally, one would like to have all layers be as lightweight as possible. The key issue here is “as possible”—some layers require complex stacking facilities while others admit simpler solutions.

To judge the potential of a lightweight layering protocol, we examined existing file-system implementations, other work in file-system structuring, and our own experience in development of layered filing. We found two different areas where lightweight layering seemed applicable: compatibility layers and miscellaneous “library” services present in existing file-systems. Figure 9.1 lists several examples of featherweight layering in these areas.

Later in this chapter we will show that each of the services described in Figure 9.1 can be provided as a featherweight layer. We have prototyped each of these layers at UCLA. These examples validate our assumption that useful services can be provided without a fully general layering service. We next examine the second assertion, that a restricted layering service can improve efficiency relative to a general service.

9.2 Costs of Fully General Layering

Featherweight layers are beneficial only if such a restricted layering environment can provide a significant performance advantage. The overhead experienced by the null layer places an upper bound on the performance gains expected from featherweight layering. At best we can hope to eliminate all null-layer overhead, although this goal may not prove possible in practice.

To understand which areas of the null layer would benefit from a different implementation we employed several benchmarks to measure layering and cache-coherence overheads. (These benchmarks are described in Sections 5.1 and 8.2.) We examined these benchmarks

Compatibility layers:

oldiftonewif Map the pre-stacking vnode interface to the stack-enabled vnode interface. (Allows a stack-enabled file-system to offer service to an unchanged higher-level clients.)

pathconf Implement a default `vop_pathconf` operation added to SunOS 4.1.

maptostackmap Convert pre-cache-coherence vnode-operations into their cache-coherent equivalents.

Internal utilities:

vmio Implement `vop_rdwr` by coping from memory-mapped file data.

fdnlc Implement name-lookup caching.

fsync Implement a generic `vop_fsync` required by the vmio layer and the virtual memory system.

specref Construct vnodes for "special" files (devices, pipes, and sockets) as necessary.

Miscellaneous user services:

frl Implement file/record locking.

ro Make a file system read-only.

Figure 9.1: Potential applications of featherweight-layering technology. The dnlc, frl, specref, and ro layers are inspired by Skinner's work [SW93].

in a test environment identical to that described in Section 8.2. Because these benchmarks are exploratory in nature and are intended to show qualitative results rather than provide controlled, quantitative results, we ran the benchmarks several times and selected a representative run rather than averaging successive runs. (We will validate the effectiveness of featherweight layering in Section 10.2 in a controlled experiment.)

Figure 9.2 shows the overheads of varying numbers of null layers for these benchmarks. The top two graphs present elapsed time, the lower two, system time. The left two graphs represent absolute values while the right graphs show overhead relative to the zero-additional-layer case.

Overheads vary significantly depending on the benchmark employed; in particular, real-time costs of the cp and rcp benchmarks show substantial variation. These results are not unexpected since these measurements examine only a single run. In spite of this variation, we can draw at least two qualitative conclusions from this data. First, the overhead for most benchmarks is basically linear in the number of layers. Second, several benchmarks (notably find and grep) show substantial overhead when multiple layers are employed. The high overhead present in these benchmarks suggests that large numbers of general layers cannot be employed for trivial purposes. Instead of the performance exhibited in Figure 9.2, a picture qualitatively like that in Figure 9.3 is desired. A few general-purpose layers are present, but much of the stack is composed of featherweight layers, considerably reducing total overhead.

9.2.1 Where is the expense of general layering?

Figure 9.2 suggests that there is significant overhead as the number of layers rises. To determine where this overhead occurs, we profiled several benchmarks executing on a stack of ten null layers. (The cumulative overhead of ten layers emphasizes where layering overhead occurs.) Like the measurements of Figure 9.2, these profiles are a single representative run taken from several observations. Again, these measurements are derived in a test environment identical to that described in Section 8.2.

Table 9.1 shows the five most expensive routines in the null layer (ranked the in-kernel execution time spent in that function and its descendents). The bypass routine is called very frequently (once per layer, per operation) and so contributes significantly to overhead. Also, for the find benchmark vnode creation time is significant.

Bypassing is expensive for two reasons. First, it is ex-

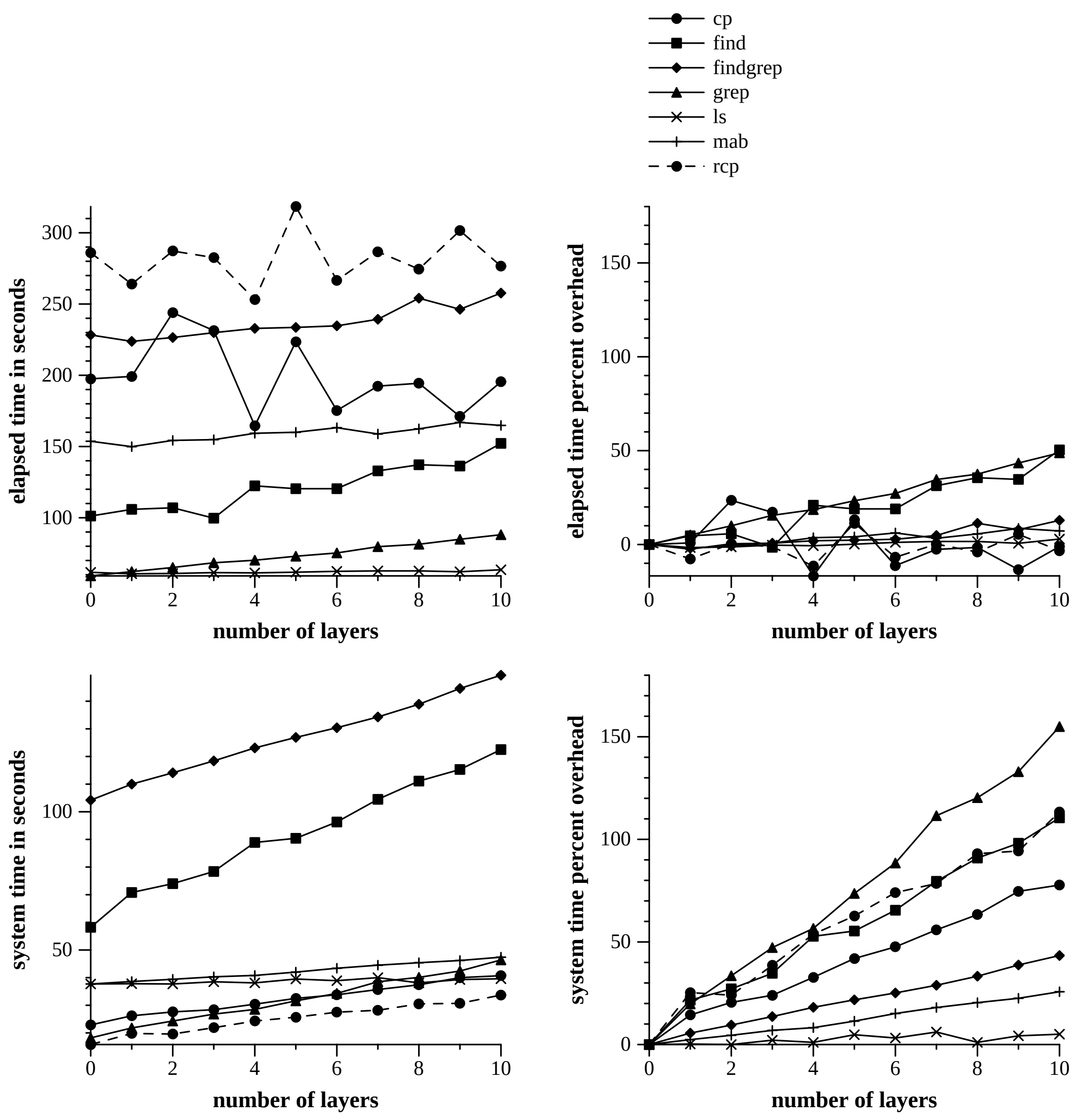

Figure 9.2: Layering overhead as the number of null layers vary. Please note that several of these graphs have different scales. Since this data consists of only one sample run and certain benchmarks exhibit high variability, these measurements should be considered qualitative and not quantitative.

Mab (Modified Andrew Benchmark)

rank	%time	self time	descendent time	call count	routine
30	4.0	2.70	0.11	149,662	`null_bypass`
100	1.1	0.05	0.70	5010	`null_make_nullnode`
119	0.9	0.17	0.45	17,656	`null_lookup`
154	0.6	0.24	0.18	18,150	`null_getattr`
167	0.5	0.10	0.27	3103	`null_getpage`

Total useful execution time: 69.55 seconds.

Find

rank	%time	self time	descendent time	call count	routine
10	20.1	3.10	34.37	263,010	`null_make_nullnode`
12	15.9	27.05	2.57	1,996,920	`null_bypass`
14	13.7	25.43	0.00	501,230	`null_find_nullnode`
19	7.1	12.06	1.15	237,280	`null_free_nullnode`
23	4.5	2.30	6.15	267,249	`null_lookup`

Total useful execution time: 186.09 seconds.
`null_find_nullnode` is called by `null_make_nullnode`.

Grep

rank	%time	self time	descendent time	call count	routine
7	20.1	6.64	0.35	417,555	`null_bypass`
18	12.3	0.48	3.81	21,400	`null_rdwr`
39	3.0	0.93	0.12	105,000	`null_open`
48	2.0	0.15	0.55	11,476	`null_lookup`
116	0.4	0.01	0.14	1080	`null_make_nullnode`

Total useful execution time: 34.81 seconds.

Table 9.1: Null-layer routine usage from three benchmarks. *Rank*, routine ranked in all kernel routines by %time; *%time*, time spent in routine or child routine as fraction of useful time spent in the kernel; *self-time*, time spent in the routine; *descendent-time*, time spent executing in descendent routines on behalf of this routine; *call-count*, total times this routine was called. These values are taken from one profiling run of benchmarks over a stack of ten null layers. Although they are typical, minor variation in future runs is not unlikely.

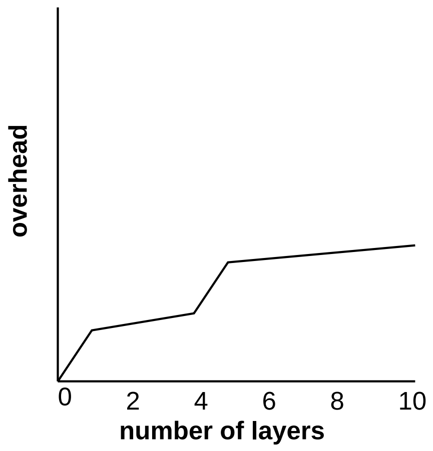

Figure 9.3: A qualitative picture of desired featherweight layering performance. The general layers 1 and 5 add some overhead, while the more numerous featherweight layers show substantially less overhead.

ecuted once per layer in the stack for all operations not otherwise implemented by that layer. Second, the generality of a bypass routine (which must be prepared to handle arbitrary operations and arguments) limits tuning. Featherweight layers address the first problem by not requiring bypass code for operations they do not modify. When bypassing cannot be avoided, the second problem can be addressed by providing bypass routines customized to particular common operations. Such a bypass routine would still need to map vnodes to the lower-layer, but one customized to a particular operation would be more efficient than the general case described in Appendix A.3.

Vnode creation is expensive because it requires object allocation and initialization. Just as with bypassing, vnode creation overhead is best reduced by avoiding it. A large vnode cache addresses this problem if vnodes are used repeatedly. Featherweight layers also avoid vnode creation by employing vnodes of their stacked-upon layer. Finally, when vnode allocation cannot be avoided, costs can be minimized using techniques such as those described by Bonwick [Bon94]. Bonwick suggests caching partially deallocated data structures. If such all cached objects share generic substructures requiring initialization (for example, locks in an SMP kernel), this initialization can be avoided by re-using the substructures left by the prior owner.

Featherweight layers therefore provide steps to improve performance both by eliminating unnecessary bypasses and by reducing the amount of data structure maintenance.

9.3 Design of Featherweight Layering

The central goal of featherweight layering is a gain in performance over general layering. The benefits of better performance must be weighed against the costs of complexity and new overhead described in the introduction, so a secondary goal must be to minimize these costs. The primary design issue for featherweight layering is therefore to determine the subset of functionality which should be selected.

To avoid complicating the stacking model, featherweight layer code should be a subset of that employed by a standard layer. This keeps the programmer's view of layering similar and allows featherweight layers to be easily "upgradable" to general layers. Any performance gains of featherweight layers will arise from a less general layering mechanism.

The generality of standard layers derives by each layer having its own representation of files in the form of private vnodes. Private vnodes imply the capability to have per-layer:

- operation implementations
- cached data objects
- file locks
- vnode locking
- private location in the file-system namespace
- private state
- stacked-upon vnode or vnodes (an important special case of private state)

All featherweight layers will require some of these features. (For example, all non-terminal layers must identify their stacked-upon vnode.) An important design issue is therefore which of these features should be made available and how they should be made available. Each unavailable feature restricts which layers can be provided with featherweight technology, but exposing features through new interfaces can be costly in performance, implementation, and complexity.

9.4 Implementation of Featherweight Layers

We have outlined the design constraints of featherweight layering above. We next discuss the implementation we've chosen and how these design constraints affected that implementation.

9.4.1 Featherweight layer configuration

Featherweight layers are configured into a stack of the general layer they stack upon at the mount-time.[1] Each featherweight layer is listed in the mount options of the general layer.

A featherweight layer depends on the layer it stacks upon for all services, except for operations it modifies. For these operations, the featherweight layer's implementation must take priority. This preemption is accomplished by building a custom operations vector for each extant configuration of featherweight layers. At mount-time the operations vector for the general layer is taken as a starting point. Each featherweight layer is then "patched in" to any operations it modifies. If the featherweight layer has no need to call the operation it replaces, its implementation simply overrides the existing operation. On the other hand, some featherweight layers may implement an operation by first invoking that operation in the lower-level layer and then modifying the result. Since that lower-level layer can be a general layer or another featherweight layer, *operation doubling* (described below) is used to preserve the stacked-upon definition.

9.4.2 Featherweight layering restrictions

Featherweight layers derive any performance improvement by providing a restricted layering environment. Inherent in featherweight layer design is the tension of expressiveness against speed and simplicity. To achieve a reasonable balance we examined the requirements of our list of potential featherweight layers (Figure 9.1).

operation implementations A featherweight layer can alter any operation. Just as with a regular layer, it can replace the operation with a completely new implementation. This new implementation can be self-contained, or it can invoke operations on lower layer vnodes.

cached data objects Data protected by the cache manager cannot be manipulated without obtaining that layer's cache name. General layers (for caching) can have multiple cache names, so this information is part of each layer's private state. To allow featherweight layers to alter cached data, simple layers can export their cache name via a new vnode operation. (This procedure is described below in "Controlled export of private state".)

file/record locks Status of user-level-lock requests are part of the public vnode state. Access to this data must be cache coherent and so is dependent on access to the cache-name.

vnode locking Per-vnode locking protocols are part of a vnode's private state and so are not accessible to a featherweight layer. Stack-wide locking is part of the cache manager and therefore is available.

private state Featherweight layers have no in-memory, private state. A number of straightforward schemes are possible to allocate and coordinate such state, but the overhead of such solutions is comparable to the cost of a general-purpose layer and the complexity added is significant. Barring a lightweight state allocation mechanism, a general purpose layer provides an appropriate mechanism for services with private state.

An extensible-attribute storage service (such as that described by Weidner [Wei95] or present in OS/2 [Dun90]) might allow on-disk private state. Full exploration of this potential is the subject of future work.

stacked-upon vnodes A featherweight layer is required to stack over exactly one other vnode. No fan-out is possible (references to multiple stacked-upon vnodes would require private state). For the kinds of services provided by featherweight layers, this restriction has not been significant.

portion of namespace Featherweight layers cannot have a namespace independent of the general-purpose layer they stack upon. A private namespace must be represented by independent vnodes; a featherweight layer lacks its own vnodes.

The central limitation of a featherweight layer is its lack of private state. We employ two techniques to overcome this limitation: export of private state and operation doubling.

Controlled export of private state

The central idea behind featherweight layering is to allow a lightweight service to make use of mechanisms

[1] In our prototype, featherweight layer configuration with its general-purpose layer implies that addition of featherweight layers after mount-time requires addition of another general-purpose layer. This restriction is an artifact of our prototype; it could be relaxed if necessary.

provided by a general layer. The featherweight layer makes use of the vnode and public data of a general layer. Other aspects of layers (such as cache management) are part of the private state of the stacked-upon layer. Often these aspects must remain part of the private state because different layers may implement them differently, or not at all.

If a featherweight layer must make use of such facilities, we export them via new vnode operations, rather than with a completely public interface. This approach allows *controlled* access to these services. Layers that do not or cannot provide such services can gracefully return an error.

Although this approach entails slightly higher overhead than a completely public interface, it allows featherweight layer access to "private" state while allowing the stacked-upon layer to retain control as necessary. We examine the performance of this approach in Section 10.2.2.

Operation doubling

Private-state export allows a featherweight layer to take advantage of some of the private state present in the general-purpose layer it stacks upon while avoiding maintenance of its own state. In some cases, though, the featherweight layer *requires* internal state. If a layer overrides an existing operation and also needs to call the old implementation of that operation, references to both operations must be stored somewhere. This storage would traditionally occur in private state, but traditional approaches to private state cannot be applied since a featherweight layer lacks private per-mount or per-vnode storage.

We solve this problem with *operation doubling*. Observing that the *only* private data allocated to an instance of a featherweight layer is the vnode operations vector, we allocate additional space in the operations vector. If a featherweight layer overloads an operation, we allocate an additional entry in the operations vector for a reference to the stacked-upon implementation.

The three approaches to vector management are illustrated in Table 9.2. The *general layer* shows a typical operations vector configuration. In the *FWL without doubling* column we add an fsync layer to this stack. The fsync layer does not need to call the underlying fsync operation, so `fsync_fsync` simply takes the place of the existing fsync vnode operation. In the final column we stack a vmio layer over the ufs. The vmio implementation of `vop_rdwr` sometimes requires calling the underlying operation, so that operation is doubled. The old operation is stored in the `vop_vmio_doubled_rdwr` slot.

Table 9.3 shows the calling sequences of invocations of each kind of operation.

Several observations are relevant to this approach. First, as an optimization, the stacked-over operation need not be saved if the featherweight layer never calls it. This was the case with the fsync layer of Table 9.2. Second, operation doubling is successful even if several featherweight layers in the same stack alter the same operation. Each layer has its own doubling slots. Finally, space in the operations vector could be used for other purposes, at least in principle. The penalty is that operations vector entries are allocated in all existing operations vectors, and the stacked-upon layer may use different vectors for different objects.

9.4.3 Commentary on the implementation

Our implementation of featherweight layering changes one aspect of vnode semantics. Although the operations vector is supposed to be opaque, there are several places in SunOS 4.x where this field is checked to determine vnode type. Since featherweight layers allocate new operations vectors for each mount-instance, the operations vector can no longer be used to determine type.

We address this problem in two ways. First, explicit type-checks are often not necessary in an object-oriented system. When we can, we restructure the code to avoid the type check. Such restructuring was not always possible, so a complimentary approach is to provide a new "vnode-type" operation, formalizing the type-checking interface.[2]

9.5 Summary

If stackable filing is to be extended to very lightweight services, a comparable layering mechanism must be provided. This chapter has suggested that featherweight layering fills that role; that by limiting per-layer private state, substantial performance improvements can be achieved. The next chapter validates these claims by examining our prototype featherweight layer implementation.

[2]Explicit type-checks are often considered bad style in an object-oriented system. Rosenthal argues for their complete elimination to improve flexibility in his design for stackable filing [Ros90]. However, there are times when type-checks are very convenient. For example, when the UFS is out of inodes, it attempts to free inodes in use by name-lookup caching. To do so it scans the name-lookup cache looking for UFS-type vnodes.

slot	**general layer**	**FWL without doubling**	**FWL with doubling**
`vop_open`	`ufs_open`	`ufs_open`	`ufs_open`
`vop_close`	`ufs_close`	`ufs_close`	`ufs_close`
`vop_rdwr`	`ufs_rdwr`	`ufs_rdwr`	`vmio_rdwr`
`vop_fsync`	`ufs_fsync`	`fsync_fsync`	`vmio_fsync`
`vop_vmio_doubled_rdwr`	`ufs_bypass`	`ufs_bypass`	`ufs_rdwr`
	...		

Table 9.2: Operations vector configurations for several layer combinations.

general layer
call `ops_vector[vop_open]` [`ufs_open`]
    `ufs_open` handles operation

FWL layer without doubling
call `ops_vector[vop_fsync]` [`vmio_fsync`]
    `fsync_fsync` handles operation

FWL layer with doubling
call `ops_vector[vop_rdwr]` [`vmio_rdwr`]
    `vmio_rdwr` gets operation, calls `vop_vmio_doubled_rdwr` (`ufs_rdwr`)
        `ufs_rdwr` handles operation
    `vmio_rdwr` does additional work, if necessary

Table 9.3: Calling sequence of `vop_rdwr` for different layer combinations.

Chapter 10

Evaluation of Featherweight Layering

In the prior chapter we described the need for lightweight, layered services, and we suggested featherweight layering as a potential solution. In this chapter we evaluate the proposed model and its prototype implementation. First, we examine the impact featherweight layers have on the programming model. We then consider the performance of our implementation.

10.1 Programming Model

For featherweight layering to be successful, it must be the case that featherweight layers substantially reduce overhead, that there are needed services which can be provided as featherweight layers, and that featherweight layers do not add significant complexity to the basic programming model. A later section discusses the performance characteristics of featherweight layers; here we consider their expressive power and potential complexity.

10.1.1 Expressive power

Featherweight layering is advantageous only if it can be employed to solve useful tasks. It derives its performance improvements by restricting layering functionality; these restrictions must not be too oppressive if beneficial featherweight layers are to be constructed.

We have found that several classes of application are amenable to featherweight layer solutions. There are a number of small, internal services used by layer designers, there are a few user services common to several layers, and several simple versioning problems can be effectively managed with featherweight layering. Figure 9.1 lists nine layers which have been prototyped at UCLA and shows which categories they fall into. These layers demonstrate that featherweight layering can be particularly effective in providing these kinds of services to multiple layers. We therefore conclude that featherweight layering is expressive enough to provide useful services.

10.1.2 Programming model complexity

We have shown that featherweight layering can be applied to several different layers, and in the next section we will show that it can significantly reduce layering overhead. But if the cost of these improvements is a significant increase in design complexity, then featherweight layering would fail the ultimate goal of improving the filing-development environment.

Featherweight layering adds two kinds of complexity to layer design. First, a second kind of layering requires that the developer choose which mechanism will be employed. If this choice is difficult or if this choice is likely to be incorrect and expensive to correct, then the choice itself adds significant complexity. Second, featherweight layering may be more difficult to use than regular layering.

It is difficult to quantitatively evaluate the cost of choosing a layering strategy. A formal experiment requires use of featherweight layers by a statistically significant number of developers in a controlled setting. This experiment has not been conducted.

Although we lack the means necessary for an experimental evaluation of differences in complexity, we have taken every step possible to minimize complexity. Good documentation of the differences in the layering schemes and examples of existing layers best serve to simplify choice of layering protocols.

We have taken several steps to minimize the cost of converting between layering structures. A featherweight layer implementation is a strict subset of a general layer implementation with only one exception. Like a general-purpose layer, a featherweight layer lists all operations it wishes to change. It lacks the vnode maintenance and layer configuration code of a general-purpose layer. Operation invocation is largely similar between featherweight layers and general-purpose layers; the only major difference is that doubled operations are invoked with a slightly different syntax. (Invocation of non-doubled operations occurs with the normal format.) This difference is due to limitations of our current interface compiler; if necessary it could be removed. We believe by subsetting general-purpose layers to make featherweight layers, we minimize the cost of transition between schemes. Choice of layering mechanism can usually be made simply by determining if in-memory state is required.

Finally, we believe that featherweight layers have an very low inherent complexity. A featherweight layer consists only of the operations it wishes to change; no external mechanisms are required. Figure 10.1 presents the complete source code of the fsync featherweight layer, for example.

10.2 Performance

There are two components to the cost of featherweight layering: a run-time performance cost and a featherweight layer instantiation cost. To completely evaluate run-time performance, we will examine both micro- and macro-benchmarks. We first examine the minimal featherweight layer instantiation costs.

10.2.1 Featherweight layer instantiation costs

file-system instantiation (mounting) is a small component of typical file-system operation because configuration occurs only occasionally. On many systems, all file-systems are configured at system power-up and are unaltered thereafter. In other environments instantiation might take place with manipulation of removable media such as a floppy diskette or CD-ROM. In these cases physical hardware latency dwarfs featherweight layer instantiation time.

For these reasons we only briefly examine instantiation costs of featherweight layers. We consider two dimensions of this cost: time and space.

Instantiation time

Initialization of a featherweight layer requires cloning and then altering the stacked-upon, general-purpose layer's operations vector. To examine instantiation cost we measured the time required to mount when mounting seven featherweight layers over a single null layer. The results of this experiment are shown in Table 10.1. Standard deviation for the total instantiation time is quite high because context switches resulted in four samples that were 5–30 times higher then the rest. When these outliers were eliminated (as shown in the right-most column) we found an average total time fell to 20,185 μseconds with a 8.38% RSD.

These measurements suggest that instantiation time of a single featherweight layer is about $\frac{1}{3}\%$ of the total configuration time. This number is quite low and, given the frequency of layer instantiation, it is difficult to imagine cases where this slight overhead would be problematic.

Instantiation memory requirements

Featherweight layers are piggy-backed on existing general-purpose layers and employ most of the services of their stacked-upon layer. The only state unique to a particular featherweight layer instantiation is its operations vector. This situation differs from a system without featherweight layering where all instances of a given layer share the same operations vector. The memory requirements of additional operations vectors therefore represent an additional memory cost of featherweight layering, a cost which we examine next.

The amount of memory required is dependent upon the number of mounted layers, operations vectors per layer, and how many total vnode operations are configured into the current system. The following formulae specify the relationship:

$$\begin{aligned}\text{ops vector memory} \;\le\; & (\text{mounted general layers}) \times \\ & (\text{ops vectors per layer}) \times \\ & (\text{ops vector size})\end{aligned}$$

$$\begin{aligned}\text{ops vector size} \;=\; & ((\text{ops for all general layers}) + \\ & (\text{doubled ops})) \times \\ & (\text{size of vector element})\end{aligned}$$

While these formulas allow us to compute featherweight layer memory requirements, for several reasons it is difficult to get good estimates for their parameters. Nearly all of them (except for the 4-byte size of a vector element) are strongly dependent on layer configuration and deployment, and on local needs and practices. A user of a small laptop computer might have 1–3 mounted general layers, 1 vector per layer, and $\sim$30

```
#include <sys/param.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <i405/i405.h>

int
i405_vn_fsync(ap)
    struct nvop_fsync_args *ap;
{
    USES_NVOP_PUTPAGE;
    return NVOP_PUTPAGE(ap->a_vp, 0, 0, 0, ap->a_cred);
}

/*
 * fsync_fwl_vnodeop_entries specifies what operations
 * the fsync layer overrides.
 */
struct fwl_vnodeopv_entry_desc fsync_fwl_vnodeop_entries[] = {
    { &nvop_fsync_desc, i405_vn_fsync, 0, NULL },
    { NULL, NULL, 0, NULL },
};
```

Figure 10.1: Annotated source code for the fsync featherweight layer.

	featherweight layer instantiation time	**total instantiation time**	**total time (without outliers)**
mean time:			
total	468 μseconds	66,208 μseconds	20,185 μseconds
per layer	67 μseconds		
% RSD	4.98%	201%	8.38%

Table 10.1: Initialization time to stack seven featherweight layers (pathconf, fsync, maptostackmap, vmio, frl, fdnlc, and oldtonew) over a null layer. These values are the mean of 20 samples of the mount command. The right-most column represents the same experiment as the center column with outliers (4 of the 20 samples) removed from the data.

parameter	workstation	server
systems examined	16	6
mounted file-systems	6–10	15–25
ops vectors per layer	1	1
operations in system	29–37	29–37
memory present	12–36 Mb	32–96 Mb

Table 10.2: Measurements of parameters of file-system usage for workstations and servers in an non-stacking academic environment. Section 10.2.1 describes how these measurements were taken.

operations per vector. A server computer running with multi-layer replicated filing system with compression and backwards compatibility layers may well have ∼100 general-purpose layers, 5 vectors per layer, and ∼80 operations per vector. Because of the multiplicative affect of these parameters, slight errors in estimation can significantly effect memory-usage estimates. While we have experience with our use of featherweight layering at UCLA, our environment is different from most because of layer development.

As an initial estimate of these parameters, we measured a number of Unix workstations and servers currently deployed in an academic setting. We measured the dynamic number of local and remote file-systems employed by machines on the Ficus project and for the UCLA computer science department. These machines were various kinds of Sun workstations and servers running either SunOS 4.x or Solaris 2.4. To avoid the bias of our use of Ficus replication, we factored out Ficus-specific volumes. The results of this survey can be seen in Figure 10.2. We recognize that these figures are dependent on their environment; however, we believe they provide order-of-magnitude figures typical to academic environments in 1994–95.

From the values of Figure 10.2 we can calculate current memory usage if current systems were converted to use featherweight layering. Current systems do not yet employ layering, and so such a calculation would underestimate memory requirements. A better estimate can be determined if we extrapolate current figures to account for layering. Table 10.3 shows the result of two such estimations, assuming moderate and heavy use of layering. For "moderate" use we project three general-purpose layers taking the place of each current file-system, each with twice the operations vectors as currently used, and a system with twice the number of vnode operations as the current maximum, and 5 doubled operations. For "heavy" use we project eight layers in place of each file-system, three operations vector per layer, three times the current maximum vnode operations, and ten doubled operations.

Memory usage of these configurations is shown in Table 10.3. Depending on layering use, memory requirements vary substantially, but even in heaviest usage featherweight layering consumes a fraction of a percent of total system memory, about the memory footprint of a small utility program. This memory usage does not seem significant in the current computing environment, and it will become even less significant as memory sizes continue to grow. Finally, if necessary, featherweight layer memory usage could be reduced by having layers with identical configurations share operations vectors.

10.2.2 Performance of featherweight layer details

We next examine particular aspects of featherweight layer performance. Just as with general-purpose layers, some aspects of featherweight layer construction are critically important to overall performance. We examine two of these: the cost of a "minimal" featherweight layer and the cost of a featherweight layer bypass operation. A combination of this detailed analysis with general benchmarks (described in Section 10.2.3 following) give a complete performance picture.

First let us consider the minimal featherweight layer. A featherweight layer consists of a list of operations to override. Since the shortest such list is empty, a "minimal" featherweight layer will not alter filing behavior in any way and will have no run-time overhead. While this example may appear contrived, it compares favorably to the overhead of the minimal general-purpose layer which includes scaffolding to set-up and maintain per-layer data structures.

The minimal featherweight layer that alters filing behavior would replace a single filing operation. The fsync layer does so, implementing code to allow a user to write all file data to disk. When added to a stack, its `fsync_fsync` implementation of `vop_fsync` is patched into the operations vector and executes directly when invoked. There is no change in the performance of any other operation. Again, performance is exactly as if the fsync service had been constructed as part of the original layer.

Featherweight layering adds overhead only when the featherweight layer requires information present only in the stacked-upon layer. An example of this overhead is present in the vmio layer which

	current		moderate		heavy	
parameter	**workstation**	**server**	**workstation**	**server**	**workstation**	**server**
mounted GP layers	6–10	15–25	30	75	80	200
ops vectors per layer	1	1	2	2	4	4
operations in system	29–37	29–37	74	74	111	111
doubled operations	0	0	5	5	10	10
FWL memory	0.7–1.4kbytes	1.7–3.6k	18.5k	46.3k	151.3k	378.1k

Table 10.3: Estimates of parameters for layer instantiation for memory usage in different system configurations. Section 10.2.1 describes how these numbers and projections were determined.

```
static int
null_get_svcm_name(ap)
    struct nvop_get_svcm_name_args *ap;
{
    *ap->a_name_p =
        VTONULL(ap->a_vp)->null_name;
    return 0;
}
```

Figure 10.2: C source code for an implementation of `vop_get_svcm_name`.

maps a standard `vop_rdwr` to a stack-friendly operation (`vop_stackrdwr`). The specification of `vop_stackrdwr` requires that the stack's cache-lock must be acquired before the call. The cache-lock is part of the stacked-upon layer's private state, so the vmio layer must retrieve this information (with another vnode operation) before locking the stack. This extra vnode operation represents overhead due to featherweight layering; if constructed as a monolithic layer, this information would not then require a vnode operation.

To quantify the cost of this extra vnode operation, we examine `vop_get_svcm_name`, the routine used by the vmio layer to access cache management information from the stacked-upon layer's private state. Figure 10.2 shows the C code required to implement this function. This subroutine expands into 7 optimized SPARC instructions, and requires about 15 lines to set up the vnode operation and check a potential error return. If this information were instead made publicly available, access cost would be less than 6 instructions. This cost is small but not insignificant; we believe that it contributes to the few percent overhead measured in the grep benchmark of Figure 10.6.

These examples illustrate how featherweight layering provides very low-overhead layering. Operations which are not modified execute as if there were no featherweight layer. Even when new operations are provided, often they can be provided as efficiently as they would have been with a monolithic implementation. While general-purpose layers have a minimal amount of overhead regardless of what service is provided, in featherweight layering overhead is proportional to the services used even if no services are employed.

10.2.3 Macro-benchmarks

Micro-benchmarks are useful to examine particular aspects of featherweight layer performance, but overall performance is often better judged through higher-level benchmarks. To investigate overall performance we compare three layer configurations (see Figure 10.3). Each of these configurations represents one possible strategy for providing lightweight services. The single-null-layer case presents the traditional approach: all services are hard-coded as part of a single layer. This approach is taken by most existing file-systems and layers.

The multiple-null-layer case approximates the performance that would be observed if lightweight services were built with general-purpose layers. This approximation is not perfect: we actually implement all services in the top layer and then emulate the framework of six other layers. Since all of the new services are provided by the top layer, we believe that this approximation slightly underestimates the overhead that would exist if services were spread throughout all layers.

Finally, the third case provides these same services with featherweight layer mechanisms. We have stripped all lightweight services out of a null layer, creating a "minimal" layer. We then added these services back as seven featherweight layers stacked over that minimal layer.

We ran our suite of benchmarks against these three configurations. (Section 8.2 describes these benchmarks

single null layer A single null layer stacked over a UFS. This “null” layer includes name-lookup and file-system data caching.

multiple null layers A stack of seven null layers over a UFS. All null layers are configured as in the single-null-layer case, but caching is done only at the top layer (to avoid double-caching effectively reducing cache size).

minimal layer with featherweight layers A minimal layer (a null layer without name-lookup and file-system caching) is stacked over a UFS. On the minimal layer are added the pathconf, fsync, maptostackmap, vmio, frl, fdnlc, and oldtonew featherweight layers.

Figure 10.3: Layer configuration for the featherweight layer macro-benchmarks.

and the specific hardware employed in these tests.) The results of these experiments can be seen in Table 10.4, and graphically in Figure 10.4.

These measurements show that the featherweight layer case has performance comparable to the single-null-layer case, while the multiple-null-layer case typically shows performance considerably worse than either of the others. This observation is most apparent for system-time measurements of these benchmarks because all overhead occurs in the kernel. There is one main exception to this observation: elapsed time for the copy benchmark. However, elapsed time of this benchmark exhibits a very high standard deviation (more than 30% for the featherweight layer case and more than 14% in the other cases), and so measurement error many be responsible for this anomaly. Additionally, there are several cases where the featherweight layer case performs slightly better (0.1 or 0.2 seconds) than the single-null-layer case. With a timer granularity of 0.1 seconds, this behavior is not unexpected.

Rather than compare these three measurements directly, it is helpful to compute the performance of these benchmarks relative to some baseline and then compare these relative measurements. We adopt the single-null-layer case as a baseline; however minimal, additional code required for layering in the other cases should add some cost to the implementation of these services. Against this baseline we compare the multiple-null-layer and the featherweight layer cases, as seen in Table 10.5 and Figure 10.5. Figure 10.6 presents the same data with

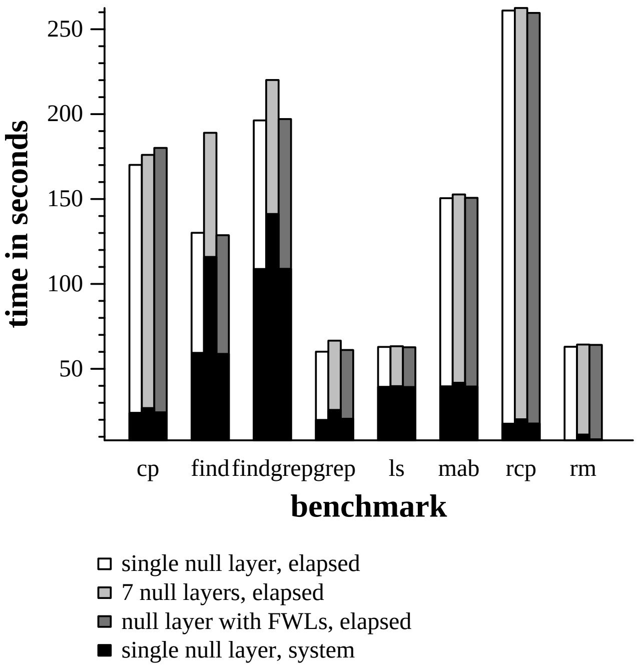

Figure 10.4: Benchmarks comparing monolithic, general, and featherweight layer configuration of seven services. Section 10.2.3 interprets this data; Table 10.4 presents it in tabular form.

		single-null		multiple-null		multiple-FWL	
time	**benchmark**	**mean**	**%RSD**	**mean**	**%RSD**	**mean**	**%RSD**
elapsed:	cp	170.1	15.09	176.0	14.22	180.1	31.23
	find	130.1	8.37	189.0	6.68	128.7	8.91
	findgrep	196.3	2.60	220.1	1.81	197.1	2.19
	grep	60.1	1.73	66.6	1.95	61.1	1.34
	ls	62.9	1.55	63.3	1.78	62.7	1.53
	mab	150.5	2.30	152.7	1.34	150.7	1.86
	rcp	261.0	8.36	262.5	8.39	259.6	5.39
	rm	63.0	0.99	64.3	1.22	64.1	2.32
system:	cp	24.1	1.35	26.9	1.34	24.4	2.39
	find	59.4	15.04	115.9	8.73	58.8	15.20
	findgrep	108.8	2.79	141.2	1.52	108.9	1.85
	grep	19.9	2.21	25.8	2.97	20.6	3.40
	ls	39.4	1.69	39.8	2.06	39.3	2.05
	mab	39.7	1.23	41.8	1.11	39.6	1.11
	rcp	17.7	4.67	20.3	3.80	17.8	5.31
	rm	7.9	5.70	11.3	4.31	8.5	7.26

Table 10.4: Benchmarks comparing monolithic, general, and featherweight layer configuration of seven services. Differences marked with an asterisk are less than the 90% confidence interval and so are not statistically significant. These values are derived from 11 sample runs of each benchmark. Section 10.2.3 interprets this data; Figure 10.4 presents it graphically.

the find benchmark omitted since the benchmark distorts the remainder of the graph.

There is substantial variation between the overheads observed in these benchmarks. This behavior is expected; different benchmarks exercise the file system in different ways. Recall that we observed two primary contributors to null-layer overhead in Section 9.2.1: vnode creation and bypass cost. The multi-null-layer case suffers from both of these penalties, while the featherweight layer case avoids vnode creation completely and suffers bypass overhead only from operations that are changed. The relative prevalence of these operations in the different benchmarks is reflected in the different overheads observed in Table 10.5 and Figure 10.5. For the find benchmark, both are extremely common, while in the Modified Andrew Benchmark both are fairly rare.

These benchmarks allow us to draw several conclusions about how to structure lightweight services. First, they suggest that a general purpose layering mechanism is too expensive a delivery mechanism for very lightweight services. Even at only 2% system-time overhead per layer (a cost needed to maintain minimal per-layer data structures) the overhead of layering quickly overwhelms lightweight services. The cumulative effect of this small overhead implies that general-purpose layers cannot be used indiscriminately, as demonstrated by the 1–30% overheads seen in the multiple-null-layer benchmarks.

Our second conclusion is that featherweight layering, by contrast, does allow a successful, layered implementation of very lightweight services. For none of our benchmarks does the featherweight layer case exhibit more than an 8% overhead, and for most of them the overhead is less than the amount of error inherent in measuring the data, and therefore is statistically insignificant.

10.3 Summary

In this chapter we have evaluated featherweight layering from several perspectives. We have shown that the programming model offered by featherweight layering is both sufficiently powerful to allow significant featherweight layers to be constructed, and yet that featherweight layers do not introduce significant additional complexity into the stackable programming model. We have looked at the performance of featherweight layering both by a close examination of the technique and by high-level benchmarks. We have found that featherweight

		multiple-null		multiple-FWL	
time	**benchmark**	**absolute change**	**% change**	**absolute change**	**% change**
elapsed:	cp	5.9	3.47	10	5.88
	find	58.9	45.27	-1.4	-1.08
	findgrep	23.8	12.12	0.8	0.41
	grep	6.5	10.81	1	1.66
	ls	0.4	0.64	-0.2	-0.32
	mab	2.2	1.46	0.2	0.13
	rcp	1.5	0.57	-1.4	-0.54
	rm	1.3	2.06	1.1	1.75
system:	cp	2.8	11.62	0.3	1.24
	find	56.5	95.12	-0.6	-1.01
	findgrep	32.4	29.78	0.1	0.09
	grep	5.9	29.65	0.7	3.52
	ls	0.4	1.02	-0.1	-0.25
	mab	2.1	5.29	-0.1	-0.25
	rcp	2.6	14.69	0.1	0.56
	rm	3.4	43.04	0.6	7.59

Table 10.5: Benchmarks comparing the relative performance of general and featherweight layer configuration of seven services to a single-layer implementation. These values are derived from 11 sample runs of each benchmark. Section 10.2.3 interprets this data; Figures 10.5 and 10.6 present it graphically.

layers can provide a very low-overhead layering mechanism and can be used successfully to implement lightweight services for which general-purpose layering is prohibitively expensive. Featherweight layering extends the applicability of UCLA layering to very "thin" layers while preserving the characteristics of binary-only, independent, third-party development and flexible configuration and installation.

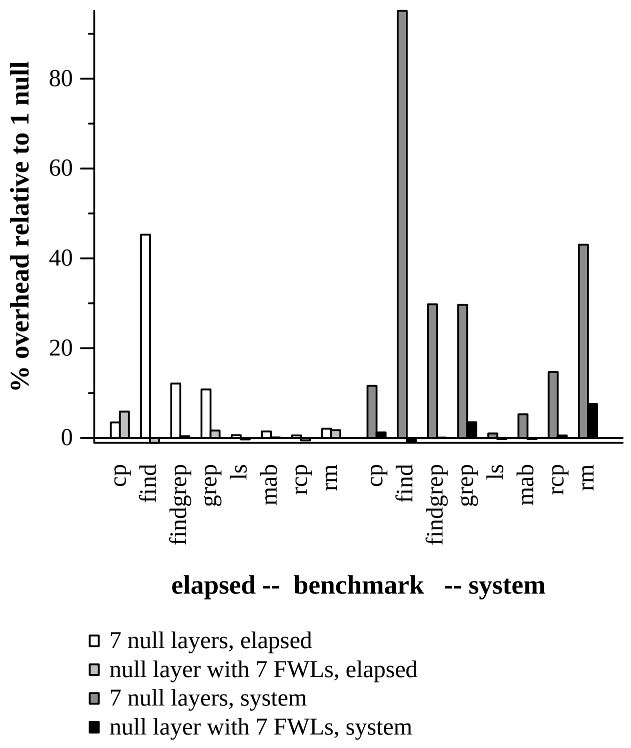

Figure 10.5: Benchmarks comparing the performance of general and featherweight layer configuration of seven services to a single-layer implementation. Methodology for these benchmarks is described in Table 10.5 and Section 10.2.3.

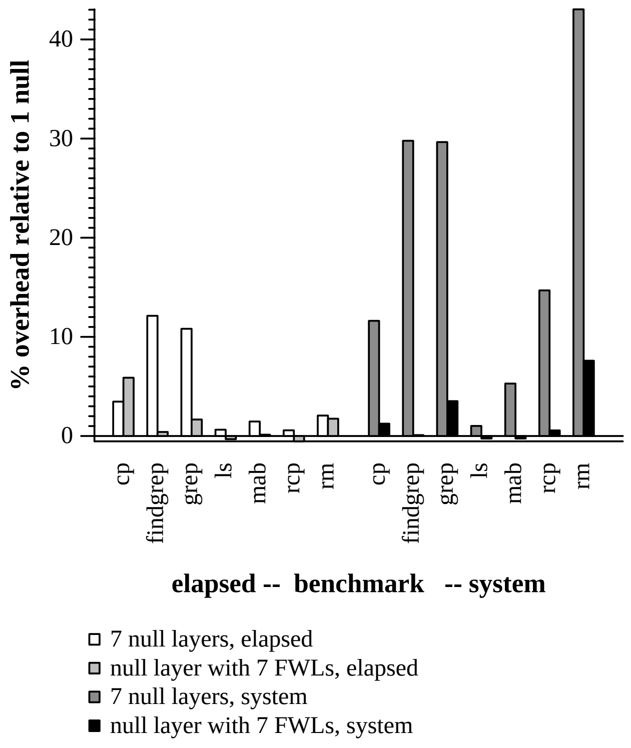

Figure 10.6: Benchmarks comparing the performance of general and featherweight layer configuration of seven services to a single-layer implementation, with the find benchmark omitted and graph scaling adjusted. Methodology for these benchmarks is described in Table 10.5 and Section 10.2.3.

Chapter 11

Related Work

File-system stacking builds on a long tradition of operating-systems research in modularity and layering. Our work at UCLA is derived directly from the vnode interface developed at Sun Microsystems [Kle86] as inspired by Ritchie's STREAMS I/O system [Rit84]. Two other groups have worked contemporaneously in file-system stacking. At SunSoft, Rosenthal and later Skinner and Wang evolved the vnode interface to support stacking [Ros90, SW93]. The Spring project at Sun Laboratories instead has created a brand-new operating system supporting coherent, stackable filing [KN93a].

This chapter begins with a review of existing work in stacking and operating system modularity. We then examine recent approaches to file-system stacking, investigating their influences, similarities, and differences.

11.1 Stacking Fundamentals

File-system stacking is grounded in work on file-system structure and symmetric interfaces. Cache coherence builds upon distributed filing and distributed shared memory. In this section we briefly review research in these areas.

11.1.1 File-system structure

Dijkstra describes early approaches to modular operating system design [Dij67, Dij68]. Madnick and Alsop [MA69], and later Madnick and Donovan [MD74] discuss modular and layered approaches to file-system design, concluding with a six-layer design. The design of Unix adopted simpler approaches, resulting in a two-layer design (file system and physical devices) [Bac86].

11.1.2 Modular file-systems

In the mid-1980s, pressure to add distributed filing systems prompted Unix vendors to develop several abstract interfaces to filing services. The vnode interface at Sun [Kle86], the generic file-system interface at DEC [RKH86] and the file-system switch at AT&T are all examples of these interfaces. The primary initial motivation was networked filing, but vendors also introduced support for other physical and logical filing systems.

One of these interfaces, Sun's vnode interface [Kle86], serves as a foundation for our stackable file-systems work. In Section 4.1 we briefly describe this interface, our changes, and the motivations for those changes.

The standard vnode interface has been used to provide basic file-system stacking. Sun's loopback and translucent file-systems [Hen90], and early versions of the Ficus file-system were all built with a standard vnode interface. These implementations highlight the primary differences between the standard vnode interface and our stackable environment; with support for extensibility and explicit support for stacking, the UCLA interface is significantly easier to employ (see Section 5.2.1). Our approaches to cache coherence and lightweight layering also journey beyond the original scope of the vnode interface.

11.1.3 Extensibility

The System V, Release 4 version of the vnode interface recognized the problem of interface extensibility. To aid future expansion of the interface, it allocates space for 32 additional operations [AT90]. (It also adds extra space in the in-memory vnode.) While these capabilities are a step towards a binary-interface standard for filing, they provide no support for third-party extensions, and they impose a significant space penalty [Ros90].

NeFS describes one proposal to provide an extensible file-system interface [Sun90], focusing exclusively on remote file access. An alternative to the NFS protocol for remote access, NeFS allows remote execution of PostScript-like programs for file access.

11.1.4 Symmetric interfaces

Unix shell programming with pipes [RT74] is an example of a widely used symmetric interface. Pike and Kernighan describe this work for software development [PK84]; other applications are as rich as text formatting [KP84] and music processing [Lan90].

Ritchie applied these principles to one kernel subsystem with the STREAMS device I/O system [Rit84]. Ritchie's system constructs terminal and network protocols by composing stackable modules which may be added and removed during operation. Ritchie's conclusion is that STREAMS significantly reduces complexity and improves maintainability of this portion of the kernel. Since its development STREAMS has been widely adopted.

The x-kernel is an operating system nucleus designed to simplify network protocol implementation by implementing all protocols as stackable layers [HPA89]. Key features are a uniform protocol interface, allowing arbitrary protocol composition; run-time choice of protocol stacks, allowing selection based on efficiency; and very inexpensive layer transition. The x-kernel demonstrates the effectiveness of layering in new protocol development in the network environment, and that performance need not suffer.

Shell programming, STREAMS, and the x-kernel are all important examples of stackable environments. They differ from our work in stackable file-systems primarily in the richness of their services and the level of performance demands. The pipe mechanism provides only a simple byte-stream of data, leaving it to the application to impose structure. Both STREAMS and the x-kernel also place very few constraints or requirements on their interface, effectively annotating message streams with control information. A stackable file-system, on the other hand, must provide the complete suite of expected filing operations under reasonably extreme performance requirements

Caching of persistent data is another major difference between STREAMS-like approaches and stackable file-systems. File systems store persistent data which may be repeatedly accessed, making caching of frequently accessed data both possible and necessary. Because of the performance differences between cached and non-cached data, file caching is mandatory in production systems. Network protocols operate strictly with transient data, and so caching issues need not be addressed.

11.1.5 User-level layering with NFS

To avoid problems with kernel-level filing development and inconsistent interfaces, a number of research projects have chosen to develop experimental services as user-level NFS servers. Examples include replication in Deceit [SBM90], automatic semantic indexing [GJS91], and ftp-access through the file-system [Cat92], among others.

NFS-servers have several advantages as a development platform. The NFS protocol provides a well-defined and widely available interface to build upon, and a user-level server can build on a local disk or another NFS server for file storage. Yet a user-level NFS server also has several very serious disadvantages. First, NFS servers gain portability because the interface is fixed. Services requiring new interfaces must either overload the existing interface, modify the basic protocol, or supply another parallel protocol. Each of these approaches has been taken in different systems, and each has significant expense in implementation and maintenance cost, and can limit portability. In addition to the set of operations, NFS clients implement a particular coherence protocol which may or may not be appropriate for a new service. Finally, communications to a user-level NFS server can pose a significant performance bottleneck as data is copied multiple times as it moves from disk to user-level server and to client, all through network buffers. For these reasons NFS-servers have limited applicability to development of new services.

11.1.6 Object-oriented design

Strong parallels exist between "object-oriented" design techniques and stacking. Object-oriented design is frequently characterized by strong data encapsulation, late binding, and inheritance. Each of these has a counterpart in stacking. Strong data encapsulation is required; without encapsulation one cannot manipulate layers as black boxes. Late binding is analogous to run-time stack configuration. Inheritance parallels a layer providing a bypass routine; operations inherited in an object-oriented system would be bypassed through a stack to the implementing layer.

Stacking differs from object-oriented design in two broad areas. Object-orientation is often associated with a particular programming language. Such languages are typically general purpose, while stackable filing can be instead tuned for much more specific requirements. For example, languages usually employ similar mechanisms (compilers and linkers) to define a new class of objects and to instantiate individual objects. In a stackable filing environment, however, far more people will configure (instantiate) new stacks than will design new layers. As a result, special tools exist to simplify this process.

A second difference concerns inheritance. Simple stackable layers can easily be described in object-oriented terms. For example, the encryption layer of Figure 1.1 can be thought of as adding encryption after inheriting "files" from the UFS "base-class". Similarly, a remote-access layer could be described as a sub-class of "files". But with stacking it is not uncommon to employ multiple remote-access layers. It is less clear how to express this characteristic in traditional object-oriented terms.

11.2 Coherence Fundamentals

Our work on cache coherence is based on several bodies of existing work, including distributed filing, hardware coherence in shared-memory multiprocessors, distributed shared memory (DSM) systems, 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.

11.2.1 Distributed filing

Early distributed file-systems such as Cedar and NFS avoid the problem of cache coherence by disallowing file mutation [SGN85] and not providing strong coherence [SGK85]. Locus provides strong coherence with a distributed token passing algorithm [PW85], while Sprite detects concurrent update at a central site and disables caching for coherence [NWO88]. Later systems provide variations on the token algorithm: AFS's callbacks are essentially centrally-managed tokens [Kaz88]; Gray's leases are tokens that can time-out to simplify error recovery [GC89].

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.

11.2.2 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 [Smi82]). 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 [LH86]. Recent work has focused on employing application-specific knowledge to relax the consistency model and obtain better performance [GLL90, CBZ91].

11.2.3 Networking protocols

We have already described early work concerning stacking of network protocols (for example, the Unix shell [PK84], the STREAMS I/O system [Rit84], and the x-kernel [HPA89]). Cache coherence is typically not an issue in networking systems since data that passes through the layers of a network stack is transient and so not suitable for caching.

Network protocols often cache routing information, 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.

11.3 Featherweight Layering Fundamentals

Featherweight layering is inspired by the desire to "compile away" layers of abstraction. Although layering is often a useful tool to logically describe a process, an implementation of each layer need not be fully general. This concept appears many times in the literature in different forms. Several groups advocate network layering without devoting a process per layer (see, for example, STREAMS [Rit84] and the x-kernel [HP88]). The x-kernel can bypass protocol layers to improve performance [OP92]. A version of Mach employs continuations to improve performance [DBR91], again discarding the process. Proponents of such systems typically cite reduced overheads in memory usage and context-switch times.

Others have suggested approaches to minimize or reduce the amount of state required in a layered system. Careful allocation techniques (for example, in the x-kernel [HMP89] and the slab memory allocator [Bon94]) reduce the cost of state initialization and reuse. Rosenthal [Ros90] limits the required per-vnode state to reduce vnode allocation costs. Finally, Massalin and others advocate run-time code generation to eliminate state [Mas92, KEH93].

To our knowledge, we are the first to suggest that interesting services can be provided with only restricted kinds of state.

11.4 Extensible Databases in Genesis

Genesis is a layered database management system developed at the University of Texas at Austin [BBG88]. Genesis decomposes the database into a number of separate services such as file storage, indexing, and data transformation. With the aid of an authoring tool [BB92], a database implementor can create a custom database by selecting particular implementations from these services.

The parallels between stackable filing and "stackable databases" in Genesis are strong. Both advocate the use of layers bounded by symmetric interfaces. Because the range of services needed for a database is so large, Genesis classifies layers into different *realms*. A realm is defined as all layers exporting the same interface; thus only layers within a single realm are interchangeable.

Genesis layers are distributed and managed as source code. This approach allows some performance optimizations. For example, binding of inter-layer operations can be done at compile time, implementing this binding as a standard function call [BO92]. It should be possible to use similar techniques to avoid data structure overheads (as we do with featherweight layering in a more restrictive context); it is not clear that Genesis employs this optimization.

Genesis does not address the general issue of inter-layer state coherence. Multi-layer access is prohibited so data cache-coherence is not an issue. Concurrency control issues are addressed by locking. Locks can apply to different granularities at different layers of the system. As locks propagate through the system, each layer is allowed to map lock ranges appropriately in a manner analogous to general cache-coherence.

11.5 Hierarchical Storage Management

Since there is no widely available hierarchical storage management (HSM) system, several third parties have developed commercial HSM solutions. Transparent HSM requires additional kernel service, often implemented as a VFS, and third parties are naturally interested in making their services available across a variety of platforms and operating system releases. Thus, HSM systems have revealed many of the limitations of the vnode interface. Webber cites portability and the lock-step release problems as significant factors in the cost of HSM solutions. (See Section 2.1 for his description of these problems.)

Faced with the difficulties of VFS-level portability, Webber proposes an in-kernel *event detector* as a hook to a *file monitor* [Web93]. The set of events monitored include both system-call-class file operations (read, write, stat, chmod, etc.) and low-level file operations (allocate and free inode, error return). Events can be processed in several different ways, including pass-through, deny, and forward to the file monitor. Epoch has successfully used this interface to implement both hierarchical storage management and on-line backup systems.

Comparison Webber's protocol has several advantages. Portability is greatly enhanced by providing a fixed set of filing events and constructing new services at the user-level. The relatively limited set of kernel-changes required to support monitoring improves chances that this service will become widely available.

Unfortunately, these strengths are also its weaknesses. A fixed set of events provides no mechanism to manage future growth and change and so limits the generality of this solution. The context switches required of the user-level file monitor also raise significant questions of performance. Webber cites overhead as about 625 microseconds per event. Such overhead is not significant when providing high-latency services such as HSM and perhaps compression, but it would be a serious limitation for many applications.

11.6 Stackable Filing at SunSoft

Rosenthal [Ros90] and later Skinner and Wong [SW93] have investigated stackable filing at SunSoft. Like our work, theirs is also inspired by Ritchie's work with STREAMS. However, differences in focus have resulted

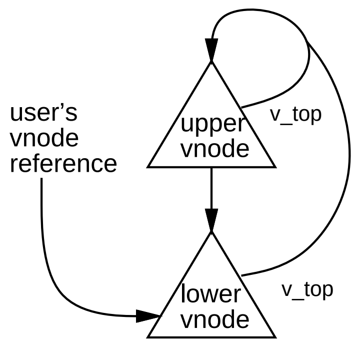

Figure 11.1: Interposition in Rosenthal's stacking forwards all operations through the "`v_top`" pointer to the top of the stack.

in substantially different stacking models. We examine each of these proposals below.

11.6.1 Rosenthal

Rosenthal's revision of the vnode interface has two main goals [Ros90]:

- To make an interface that would evolve to meet new demands more gracefully by supporting *versioning*.
- To reduce the effort needed to implement new file-system functionality by allowing vnodes to be *stacked*.

Rosenthal's approach to versioning employs compatibility layers (this technique is described in Section 3.5). Rosenthal's stacking model is based on interposition. To implement interposition, each vnode contains a reference to the top-of-stack (`v_top` in Figure 11.1). Each "external" vnode operation indirects through this reference to the top-of-stack. (Although not stated explicitly in his paper, presumably there is a second kind of vnode operation employed internally to a stack to invoke operations on the stacked-upon layer without this extra indirection.)

Rosenthal provides two new stack operations ("push" and "pop") to interpose layers on a stack at a file-by-file granularity.

Finally, Rosenthal describes several applications of stacking technology, including quotas as a layer, a "less temporary" file-system where files remain in memory until an explicit synchronization, user-level filing, read-only caching, read-write caching, a "fall-back" file-system where load is spread over several servers, and a replication service.

Interposition As described in Section 3.7, the primary difference between interposition and other kinds of stacking is that a layer interposed on a stack becomes immediately visible to all current and future clients of any layer of that stack.

Interposition in Rosenthal's stack model requires that all users see an identical view of stack layers; dynamic changes of the stack by one client will be perceived by all other clients. As a result, it is possible to push a new layer on an existing stack and have all clients immediately begin using the new layer. We describe in Section 3.7 how this feature can be employed to redirect the clients of an existing layer, to add and remove a measurements layer at run-time, or to implement the join of two file-systems at a mount-point.

However, it is not clear how widely this facility is required. Section 3.7 describes different applications of interposition, but often other techniques can be substituted with little or no loss in functionality. We believe interposition is rarely required because stack layers typically have semantic content. A client selects a particular combination of semantics when opening a file. If the client wants to change the semantics of its stack, it can do so by opening the file through other layers. It is rare that a client wishes to allow *other* clients' preferences for stack semantics to influence its own. Consider, for example, Figure 1.1. One client is reading the encrypted data directly from the UFS. If the encryption layer were provided with interposition, another client opening the file through the encryption layer would force the first to switch from encrypted to decrypted data mid-stream. This example argues that mid-stream change is rarely necessary, and, to the extent that it adds complexity and overhead, it is undesirable.

In addition, insuring that all stack clients agree on stack construction has a number of drawbacks. As discussed in Section 3.3, access to different stack layers is often useful for special tasks such as backup, debugging, and remote access. Such diverse access is explicitly prohibited if only one stack view is allowed. Insuring a common stack top also requires very careful locking in a multiprocessor implementation, at some performance cost. Since the UCLA interface does not enforce atomic stack configuration, it does not share this overhead.

The most significant problem with Rosenthal's method of dynamic stacking is that for many stacks there is no well-defined notion of "top-of-stack". Stacks with fan-in have *multiple* stack tops. Encryption is one service requiring fan-in with multiple stack "views" (see Section 3.3). With multiple top-of-stacks there is no single stack view, and so interposition does not make sense. Furthermore, with transport layers, the correct stack top could be in another address space, making it impossible to keep a top-of-stack pointer. For all these reasons, our stack model explicitly permits different clients to access the stack at different layers. Skinner proposes extensions to Rosenthal's model which remove this limitation by providing two stacking mechanisms. We examine these extensions in the next section.

Other differences Per-file configuration allows additional configuration flexibility, since arbitrary files can be independently configured. However, this flexibility complicates the task of maintaining this information; it is not clear how current tools can be applied to this problem. A second concern is that these new operations are specialized for the construction of linear stacks. Push and pop do not support more general stack fan-in and fan-out.

Another difference between Rosenthal's vnode interface and the UCLA interface concerns extensibility. Rosenthal discusses the use of versioning layers to map between different interfaces. Version-mapping layers work well to manage differences caused by occasional change originating from a single source, such as periodic vendor releases of an operating system. Mapping layers provide little support for third-party-initiated change, however, since the number and overhead of mappings grow significantly as additional changes must be supported. A more general solution to extensibility is preferable, in our view.

Finally, Rosenthal advocates minimizing the amount of state required of each vnode. "Cheap" vnodes encourage layer use. The sentiment is well-founded. However, our experience with lightweight layering suggests that even cheap vnodes still have noticeable overhead. In our view a solution such as featherweight layering is required to extend stacking to exceedingly lightweight layers.

11.6.2 Skinner and Wong

Skinner and Wong revised Rosenthal's stacking model [SW93] based on further experience with that model [Ros92] and prompted by Unix International's Stackable File-Systems Working Group [Gro92]. The primary innovation in their new model is to employ two kinds of file-system "stacking": interposition and composition. Interposition retains the desirable features of Rosenthal's stacking mechanism. Composition adds fan-out capability and is achieved with the mechanisms similar to those used for stacking in the UCLA model and in standard vnode environments.

To support two styles of stacking they divide the traditional functionality of the vnode into two separate abstractions: *cvnodes*, for use in composition; and *ivnodes*, for interposition.[1] Cvnodes are the abstraction used by the upper-level kernel and for composition. They are implemented simply as a reference to a chain of ivnodes; this reference is the equivalent of Rosenthal's `v_top`. Ivnodes reproduce the data present in the original vnode and are used for interposition. Figure 11.2 shows C-code proposed for these data structures.

Decomposition of vnodes into cvnodes and invodes is also reflected in vnode operations. Vnode operations are permitted only one "vnode" argument; this argument is implemented as a pair ⟨cvnode, ivnode⟩. When an operation is invoked on a cvnode (indicated by a NULL ivnode field), the ivnode is automatically set to the head of the interposition chain. As the operation moves down the chain of interposition nodes the ivnode field is updated. This approach to operation invocation allows the two kinds of operations (those directed to the top of an interposition chain and those intended for the next link in the chain) to be provided with only one form of operation.

Several existing operations have multiple vnodes as arguments. Skinner and Wong use two different mechanisms to meet this restriction. First, they decompose operations such as `vop_link` and `vop_rename` into the more basic operations listed in Table 11.1. Second, they replace a second vnode in an operation by a file-identifier.

A locking protocol manages concurrency during "plumbing" operations (such as `vop_pop` or `vop_lookup`) which create new cvnodes or change the interposition state. Their implementation supports multiprocessing.

Skinner and Wong support vnode stacking but leave the VFS interface unchanged. To allow changes in VFS operation behavior, they converted several vfs operations into vnode operations

As applications for stacking, Skinner and Wong provide a "toolkit" of several interposition layers implementing services such as stack configuration, name-lookup caching, file/record locking, mount-point management, device and special file support, and write-prohibition (for a read-only file-system).

[1] In Skinner and Wong's terminology, cvnodes are simply new "vnodes" and ivnodes are called pvnodes. We adopt different terminology here to reserve the term "vnode" to refer to the original concept.

```
struct cvnode {
    struct ivnode *v_chain; /* head of interposition chain */
};
struct ivnode {
    struct vfs *i_vfsp;    /* owning VFS */
    struct vnodeops *i_op; /* the ops vector */
    struct ivnode *i_link; /* next older interposer */
    void *i_private;       /* this interposer’s state */
};
```

Figure 11.2: A C implementation of cvnodes and ivnodes. (Derived from Figure 2 of Skinner and Wong [SW93].)

name	**purpose**
`vop_fid`	get a file’s file identifier (fid)
`vop_mkobj`	create an object and return its vnode and fid
`vop_diraddentry`	add a directory entry for a fid-identified file
`vop_dirrmentry`	remove a directory entry
`vop_inclink`	increment a file’s link count
`vop_declink`	decrement a file’s link count

Table 11.1: Decomposed vnode operations. This table appears as Table 1 in Sinner and Wong [SW93].

Stacking

Provision for both layer interposition and composition allows more flexibility than either Rosenthal’s interposition or composition alone. However, the question of the need for interposition raised above apply here as well. Interposition is required to implement mount-point handling through stacking, and in a few other scenarios (see Section 3.7). Interposition can also replace composition in some instances, if desired. But support for both composition and interposition has potential costs in both complexity and performance.

Two mechanisms accomplishing similar tasks adds substantial additional design complexity. A layer designer now must select what kind of stacking mechanism is to be used early in layer design. Minimizing design complexity was of primary concern in featherweight layering; our approach to reduce complexity was to make one service a subset of the other. This simplification does not to apply to interposition and composition.

In addition to the complication of two stacking mechanisms, the requirement that each operation have only a single vnode argument adds a substantial burden to those employing the interface. Skinner and Wong work around this problem by using file-identifiers to represent other vnodes and by decomposing high-level operations (such as link and rename) into a sequence of lower-level operations (name entry, link incrementing, etc.). Again, the use of two abstractions (vnodes for the first file in an operation and file-identifiers for subsequent files) adds complexity. In addition, exposing a lower-level interface to directories complicates the implementation of file systems that don’t match the traditional Unix disk model (such as NFS and the MS-DOS FAT format). Skinner and Wong recognize this problem in the “Impedance Mismatches” section of their paper [SW93].

Finally, the costs of the protocol changes made by Skinner and Wong are not entirely clear. They present performance analysis of several user-level benchmarks; their primary benchmark is simulation of a C-program development-environment with varying levels through-put. For this benchmark they show costs from between 2.3% improvement to 9.9% overhead (depending on workload). Our experience suggests that overheads are most easily detected in the system-time of file-system intensive benchmarks.

Other differences Like Rosenthal, Skinner and Wong recognize the problem of cache coherence as an area of future work.

Skinner and Wong advocate lightweight layering and propose layered solutions for several small services. In fact, several of their “toolkit” layers provided inspiration for individual featherweight layers in our system.

Our experiences suggest that construction of these services with general-purpose layering techniques would show noticeable system-time overhead. They describe a file-creation benchmark which shows “no significant performance degradation” even with five of these layers. Few details are provided about this benchmark and how its performance was measured; however our experience shows that file creation benchmarks can easily be dominated by synchronous disk-write times rather than software overheads.

Finally, Skinner and Wong suggest that addition of a lightweight transaction mechanism to the vnode interface would simplify error recovery. We agree with their analysis that future work and prototyping is required before transaction support will be widely accepted at the vnode level.

11.7 Spring

The Spring operating system is an object-based, distributed operating system developed at Sun Laboratories [MGH94]. Objects in Spring implement an interface specified by an interface-definition language [HR94]. Objects can be distributed transparently between the kernel, user-level servers, and remote machines.

Notable features in Spring include a virtual memory system supporting external pagers [KN93b] and a coherent distributed filing system [NKM93] implemented with stackable layers [KN93a].

11.7.1 Stacking

Several aspects of file-system layering in Spring [KN93a] are similar to ours. In Spring, a filing service is provided by a layer which implements the Spring filing interface. An implementation of a filing layer might build upon other filing layers. Layers are combined and configured using the Spring naming service [RNP93]. In the terminology of Section 11.6.2, Spring employs composition to build layers. Section 5 of Khalidi and Nelson suggests that their object service provides a general mechanism for run-time object interposition [KN93a], but few details of this mechanism are available. Spring also supports cache-coherence between layers; we describe this service below in Section 11.7.3.

The Spring approach to stacking is aided the fact that Spring is a new operating system, built from scratch. Spring employs an object-based interface-definition language throughout their system, and it structures all system interfaces as necessary to support stacking. However, with the exception of interposition, basic stacking in Spring is functionally similar to that provided by UCLA stacking.

11.7.2 Extensibility

Spring manages interface extensibility through its object-oriented interface-definition language [HR94]. An interface is defined as a class; new versions inherit from this class to add features. Type-checking at both compile- and run-time can be used to insure that the client and provider of a service communicate with consistent interface versions. Run-time type conversion can be used to export or employ older interfaces for backwards compatibility.

The Spring approach to versioning makes elegant use of object-oriented technology to address the lock-step release problem. An operating system vendor can introduce new services by sub-classing existing interfaces, yet can maintain backwards compatibility if desired using type-checking and run-time type conversion. Interface inheritance doesn’t directly address the VFS portability problem; portability must be addressed by standardization on a few interfaces. Interface inheritance seems unlikely to satisfactorily address the extension problem. Independent evolution of interfaces does not map well to the single version-hierarchy suggested for Spring.

Finally, while the extensibility mechanisms proposed in UCLA stacking may lack the elegance of expressing versioning with an interface class-hierarchy, their much simpler implementation makes broad deployment more likely. Operation-granularity evolution and extensibility (rather than whole interface evolution as suggested in Spring) also seems much more likely to permit change by multiple, independent third parties.

11.7.3 Cache-coherence in Spring

Virtual memory and file systems are very closely related in Spring. The virtual memory system includes support for distributed shared memory [KN93a]. Cache-coherent file-system stacking is a natural result of this architecture. The Spring cache-coherence work highlights two important results. First, the Spring developers 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. Our work in cache coherence builds 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.

11.8 Meta-Descriptions of Stacking

Most of this chapter has discussed other systems which provide layering and stacking in some form. Inspired by their experiences with Genesis [BBG88] and Avoca [OP92] (a version of the x-kernel), Batory and O'Malley describe a meta-model for hierarchical software systems [BO92].

In their model hierarchical software systems are built from *components*. Each component is a member of a *realm*, a group of components which implement the same interface. Components may be built from other components. A component is *symmetric* if it builds upon components from the same realm.

Stackable filing and UCLA stacking easily fit into this model. All filing layers are part of a single realm. Each layer corresponds to a component.

Batory and O'Malley's experiences in two such different areas suggest to us that our approaches to stackable filing may also find wider applicability.

Chapter 12

Conclusion

file-system development has long been an area of fruitful research. Unfortunately, broad application of this research has been difficult. Implementation of new filing ideas has been slow because services were built from scratch or modifications to existing systems added undesirable encumbrances to distribution. Even when completed, new services often have failed to work together, and have failed to be robust across system changes.

Stackable file-system development offers an alternative. Stacking allows new services to build on (instead of re-build) well-understood filing services. Distinct layers help to confine changes and focus testing efforts. A consistent approach to interface extension means that vendors and third parties can provide new services without invalidating existing work. A cache-coherence protocol insures that designers and users can construct and access their stacks with confidence that they see up-to-date information. Finally, a featherweight layering protocol allows stacking to apply to very thin layers as well as to major new services. Together, these capabilities offer the potential for broader acceptance and deployment of new filing services.

Through our prototype implementation we demonstrate the effectiveness of stacking, both as a paradigm and as a means to provide replication and other services. We provided a detailed performance analysis. Most important, we show that stackable layers can in many ways offer a development environment superior to the alternatives.

12.1 Research Contributions

The primary contribution of this dissertation is the validation of the thesis: that a layered, stackable structure with an extensible interface provides a better methodology for file-system development than current approaches. Evidence for this thesis is provided through the following means:

1. Design of:
   - A file-system interface supporting easy extensibility by multiple third parties.
   - A bypass mechanism which allows file-system stacking and extensibility to work together.
   - A cache-coherence protocol supporting safe data caching in third-party filing layers.
   - A lightweight subset of stacking extending file-system layering to very low-overhead layers.
2. Description of:
   - File-system structuring techniques enabled and simplified by stacking.
   - Experiences using stacking in the classroom.
3. Production-quality implementations of:
   - An extensible file-system interface in use by a community of thousands of people and distributed with several Unix implementations.
   - A system for user-level layer development used by a community of two dozen.
   - Two transport-layers which transparently pass new operations between address spaces.
   - Use of file-system layering to provide a variety of services including:
     - optimistic replication
     - configurable replication consistency protocols
     - user-identity mapping
     - compatibility-mapping between differing file-identifier sizes

    – namespace duplication

  These layers have been used by over two dozen people for more than four years; they currently host the complete stacking development environment and Ficus-project user population.

4. Prototype implementations of:
   - Cache-coherence protocols which insure that data can be safely cached in multiple stack layers.
   - A featherweight layering protocol providing "thin" layers with very little overhead.

5. Verification of the following statements through performance evaluation of our system:
   - Addition of stacking to a system incurs minimal overhead.
   - Per-layer stack costs are minimal: general purpose layers incur about 2% system-time overhead, featherweight layers incur negligible overhead.
   - Addition of cache coherence to a system incurs minimal overhead.

6. Substantial empirical experience suggesting that:
   - Our extensible interface is portable to systems with independently implemented vnode interfaces.
   - Stacking makes it substantially easier to implement small layers.
   - Stacking can be used to significant advantage for large, heavily used filing services.
   - Cache-coherence for file-system layering can be implemented without significant changes to the virtual memory system.

12.2 Future Work

Work as broad as a new filing substrate borders on many areas of related work. This section lists some of these areas that this dissertation does not address.

12.2.1 Implementation enhancements

Over the course of this work typical memory capacities of workstations have advanced from 8 megabytes to 40 megabytes. The operating systems world has similarly advanced. Our implementation must keep pace.

Nearly all operating systems today support kernel-module dynamic loading. We anticipate that addition of dynamic loading to stackable layers modules will be possible with the usual dynamic loading techniques (for example, pre-reservation of table space).

The new services offered through stackable layers often need "just a little more" information stored with the file. For example, compression might need an "is-compressed" flag or might be aided by an uncompressed-size field, and encryption would require space for an encryption key and related information. A convenient way to provide such additional storage would be through a generic extensible-attributes service. We have prototyped such a system at UCLA [Wei95], and designs and implementations for other environments exist [And90, Dun90, Ols93]. Adoption of any one of these extended attribute services is another way to ease file-system development.

Finally, we would like to move our stacking environment to a kernel with symmetric multiprocessing (SMP). We have kept SMP support in mind throughout implementation, and so we expect few problems; a prototype SMP-based implementation is the best way to answer this question definitively.

12.2.2 Stacking

Current mechanisms manage large-granularity file-system stacking well. The best way to manage per-file stack configuration is less clear. A standard "stack composition" attribute, possibly stored in some kind of extensible attribute system, might provide a solution. One intriguing implementation of such a system in a slightly different context is contained in Kim's object-oriented filing work [Kim95]. More experience and wider deployment of such systems is required to evaluate the approach, however.

Current disk-based file-systems are almost exclusively constructed as large, monolithic layers. This situation is unfortunate because it means that several potentially separable services are tied into one package; to take one part requires using it all. It would be very useful to decompose existing storage services into separate flat-file and directory service abstractions, and possibly others as well. 4.4BSD takes some steps in this direction, employing the vnode interface internally between UFS directory routines and the on-disk log-structured and fast filesystems [McK95]. Unfortunately it is not currently possible to separate these services into independent layers.

12.2.3 Extensibility

Our approach to extensibility has worked well, both by allowing us to add to the interface and in managing interface changes from the operating-systems vendor. Unfortunately, there are at least three independently derived “vnode” interfaces in use currently (SunOS [Kle86], BSD [KM86], and Linux [Joh92]). Each system has a substantially different operation mix and collection of supporting services. Simple extensibility is insufficient to bridge these differences. Compatibility layers (see Section 3.5) offer substantial hope of addressing these differences, but technical challenges still remain. The best long-term solution would be migration to a set of much more similar interfaces.

As a “third party” we have made substantial use of interface extensibility for Ficus replication. Often we have found ourselves needing an operation just like the current one but with a small change, such as a single additional argument. Our approach to extensibility allows new operations, but there is no easy way to add arguments to existing operations. An object-oriented interface (such as that in Spring) might allow operation addition with subclassing, but more work is needed in this area to support third-party additions.

12.2.4 Cache coherence

We have explored cache-coherence across the layers of a stack on a single system and in a distributed environment with NFS. In Section 6.5 we argued that different distributed environments require different levels of coherence. A closer examination of the interactions between cache-coherence and distributed systems with stronger guarantees would be interesting.

12.2.5 Lightweight layering

Featherweight layers minimize layer cost by restricting per-layer state, thus allowing very lightweight layers to be provided with correspondingly little overhead. Another approach to improve layering performance would be to relax general-purpose layering’s prohibition on source code. For example, source code allows Genesis to avoid dynamic operation invocation by matching operation calls with targets at compile time. While source-code-optimized layering might not be suitable for commercial distribution, it might allow a single development organization to construct a system as several “logical” layers which are “compiled away” to run with the best possible performance.

12.3 Closing Remarks

This dissertation has presented stackable layering as an approach for file-system construction. We have saved substantial effort through the use of stacking for the development of file-replication and other services at UCLA, both because stacking has allowed the *re*-use of existing services, and because extensibility has permitted us to evolve different parts of our service at different rates and to easily manage external change.

This dissertation has described a general approach for file-system stacking and extensibility, and has presented solutions for the important issues of cache-coherence and lightweight layering. We have demonstrated the effectiveness of these solutions both through our own use and through performance analysis and experimentation. We believe that adoption of these techniques has and will substantially improve file-system development, and thereby allow future filing research to be both more effective and relevant than it has been.

Appendix A

Stacking Appendix

This appendix summarizes the interface changes needed to support stacking, as described in Chapter 4.

A.1 A Sample Vnode Operation

As described in Section 4.4, we made several changes to the vnode interface to support stacking and extensibility. Here we summarize differences in the calling sequence.

In Figure A.1 we show the original operation invocation sequence. The operation is an indirect function call through a vnodeops structure defined at compile time.

Figures A.2 and A.3 shows two variations of the new calling sequence. There are three important differences between old and new invocations. The most obvious change is that parameters are passed in an arguments-structure rather than on the stack. An arguments-structure has the advantage that it avoids re-copying all arguments each time an operation moves down a stack layer. In addition, a pointer to an arguments-structure serves as a generic "handle" to allow manipulation of any set of vnode-operation arguments. A second change is that we add an operation description as the first parameter of the call. This description includes the operation's name, and information needed to marshal operation arguments for RPC and bypass operations. (A complete list of this information appears in Figure A.4.) By placing this description in a well-known location we can identify and manipulate arguments to an arbitrary operation. In Section A.3 we describe how the argument structure and descriptive information are employed to bypass an operation through a layer.

The final important difference in the calling sequence is that the operation vector is managed dynamically rather than fixed at compile time. The global variable `vop_create_offset` is set to that operation's position in the operations vector when operations are configured.

Figures A.2 and A.3 show two slightly different approaches to argument-structure allocation. In Figure A.2 the programmer explicitly declares information with `USES_VOP_CREATE`. In Figure A.3 the compiler assumes this duty in an in-lined function call. The in-line approach provides a slightly more appealing interface to the programmer but requires compiler support for function in-lining, a widely available but non-ANSI standard feature. Both constructs should generate comparable code.

Finally, Figure A.5 shows old and new implementations of `vop_create`. In the new form, parameters arrive in an arguments-structure.

A.2 A Sample Operation Declaration

Support for operation bypassing and transport layers requires a complete definition of each operation and its arguments. In Section 4.7 we described these requirements. Figure A.6 shows a sample interface definition of `vop_create` from our prototype.

A definition lists operation arguments and their types. In addition, the direction of data movement must be indicated with an IN, OUT, or INOUT tag. This information is required to generate RPC code. Finally, we add a UUID to uniquely identify the operation in communications between different address spaces.

A.3 A Sample Bypass Routine

Figure A.7 presents the bypass routine for the null layer. For the null layer, the only requirement of the bypass routine is that vnodes are mapped to their lower-layer equivalents before calling the lower layer, and then restored upon return. In addition, if a new vnode is re-

```
struct vnodeops {
    int (*vn_create)();
    ....
};
struct vnode {
    struct vnodeops *v_op;
    ...
};
#define VOP_CREATE(VP,NM,VA,E,M,VPP,C) \
    (*(VP)->v_op->vn_create) (VP,NM,VA,E,M,VPP,C)

void
call_demonstration(struct vnode *dvp,
    struct vattr *va,
    struct vnode **result_vp,
   struct ucred *cred)
{
    int error = VOP_CREATE(dvp, "file", &va, NONEXCL, 0,
                           &result_vp, cred);
}
```

Figure A.1: Old calling sequence for `vop_create`.

```
struct vnode {
    int (**v_op)();
    ...
};
#define USES_VOP_CREATE struct vop_create_args vop_create_a;
#define VOP_CREATE(VP,NM,VA,E,M,VPP,C) \
    ( vop_create_a.a_desc = &vop_create_desc, \
    vop_create_a.a_vp = (VP), \
    vop_create_a.a_nm = (NM), \
    vop_create_a.a_vap = (VA), \
    vop_create_a.a_exclusive = (E), \
    vop_create_a.a_mode = (M), \
    vop_create_a.a_vpp = (VPP), \
    vop_create_a.a_cred = (C), \
    (( *( (VP)->v_op [vop_create_offset] )) (&vop_create_a))

void
call_demonstration(struct vnode *dvp,
    struct vattr *va,
    struct vnode **result_vp,
   struct ucred *cred)
{
    USES_VOP_CREATE;
    int error = VOP_CREATE(dvp, "file", &va, NONEXCL, 0,
                           &result_vp, cred);
}
```

Figure A.2: Macro-based new calling sequence for `vop_create`.

```
struct vnode {
    int (**v_op)();
    ...
};
inline int
VOP_CREATE(struct vnode *VP,
    char *NM,
    struct vattr *VA,
    enum vcexcl E,
    int M,
    struct vnode **VPP,
    struct ucred *C)
{
    struct vop_create_args vop_create_a;
    vop_create_a.a_desc = &vop_create_desc;
    vop_create_a.a_vp = VP;
    vop_create_a.a_nm = NM;
    vop_create_a.a_vap = VA;
    vop_create_a.a_exclusive = E;
    vop_create_a.a_mode = M;
    vop_create_a.a_vpp = VPP;
    vop_create_a.a_cred = C;
    return (( *( VP->v_op [vop_create_offset] )) (&vop_create_a));
}

void
call_demonstration(struct vnode *dvp,
    struct vattr *va,
    struct vnode **result_vp,
   struct ucred *cred)
{
    int error = VOP_CREATE(dvp, "file", &va, NONEXCL, 0,
                           &result_vp, cred);
}
```

Figure A.3: In-line-based new calling sequences for `vop_create`.

```
struct vnodeop_desc {
    int vdesc_offset;   /* offset in vector */
    char *vdesc_name;   /* a user-readable string */
    int vdesc_flags;

    /*
     * This information is used by bypass routines
     * to map and locate arguments.
     */
    int *vdesc_vp_offsets;
    int vdesc_vpp_offset;
    int vdesc_cred_offset;
    int vdesc_proc_offset;

    /*
     * The following data is for transport layers aid.
     * vdesc_datatype describes the arguments in detail.
     * vdesc_uuid_p is a uuid that uniquely names the operation.
     * (Although technically the uuid should be transport-layer
     * specific, its kept here because it’s common to several
     * transport layers.)  Finally, vdesc_uuidhash
     * is used to quickly locate operations by uuid.
     */
    struct xport_datatype *vdesc_datatype;
    struct nca_uuid *vdesc_uuid_p;
    struct vnodeop_desc *vdesc_uuidhash;
};
```

Figure A.4: Descriptive information accompanying each vnode operation.

(a) old vnode-operation implementation: `int`
```
    xfs_create(struct vnode *vp,
        char *name,
        struct vattr *vattrs,
        enum vcexcl exclusive,
        int mode,
        struct vnode **vpp,
        struct ucred cred)
    {
        /*
         * Do the work to create ‘‘name’’
         * in directory ‘‘vp’’, returning
         * ‘‘vpp’’.
         */
    }
```

(b) new vnode-operation implementation: `int`
```
    xfs_create(struct vop_create_args *ap)
    {
        /*
         * Do the work to create
         * ‘‘ap->a_name’’ in directory
         * ‘‘ap->a_vp’’, returning
         * ‘‘ap->a_vpp’’.
         */
    }
```

Figure A.5: Old and new implementations of `vop_create`.

```
OPERATION
vop_create {
    IN struct vnode *a_vp;
    IN char *a_nm;
    IN struct vattr *a_vap;
    IN enum vcexcl a_exclusive;
    IN int a_mode;
    OUT struct vnode **a_vpp;
    IN struct ucred *a_cred;
} uuid(2ac783c60000.02.83.b3.c0.45.00.00.00);
```

Figure A.6: An interface definition of `vop_create`.

turned as part of the operation, we create a new null-node to stack over it.

Three observations are important about the bypass routine. First, it refers to the arguments through a generic pointer, so it can accept arguments for any operation. Second, it uses information obtained from the vnode-operation description to manipulate well-known arguments such as vnodes. Finally, more sophisticated layers can modify other arguments in the bypass routine. The user-identity layer, for example, maps user-ids just before the call to the lower layer.

```
int
null_bypass(ap)
    struct vop_generic_args *ap;
{
    int error, flags, i;
    struct vnode *old_vps[VDESC_MAX_VPS];
    struct vnode **vps_p[VDESC_MAX_VPS];
    struct vnode ***vppp;
    struct vnodeop_desc *descp = ap->a_desc;

    /* Map the vnodes going in.
     * Later, we’ll invoke the operation based on
     * the first mapped vnode’s operation vector.
     */
    flags = descp->vdesc_flags;
    for (i = 0; i < VDESC_MAX_VPS; flags >>= 1, i++) {
        if (descp->vdesc_vp_offsets[i] == VDESC_NO_OFFSET)
            break;   /* bail out at end of list */
        if (flags & 1)   /* skip vps that aren’t to be mapped */
            continue;
        vps_p[i] = VOPARG_OFFSETTO(struct vnode**,descp->vdesc_vp_offsets[i],ap);
        old_vps[i] = *(vps_p[i]);
        *(vps_p[i]) = NULLTOLOWERV(VTONULL(*(vps_p[i])));
    };

    /* Call the operation on the lower layer
     * with the modified argument structure.
     */
    error = VCALL(*(vps_p[0]), descp->vdesc_offset, ap);

    /* Maintain the illusion of call-by-value
     * by restoring vnodes in the argument structure
     * to their original value.
     */
    flags = descp->vdesc_flags;
    for (i = 0; i < VDESC_MAX_VPS; flags >>= 1, i++) {
        if (descp->vdesc_vp_offsets[i] == VDESC_NO_OFFSET)
            break;   /* bail out at end of list */
        if (flags & 1)   /* skip vps that aren’t to be mapped */
            continue;
        *(vps_p[i]) = old_vps[i];
    };

    /* Map the possible out-going vpp.
     */
    if (descp->vdesc_vpp_offset != VDESC_NO_OFFSET &&
            !(descp->vdesc_flags & VDESC_NOMAP_VPP) &&
            !error) {
        struct ucred **credpp = VOPARG_OFFSETTO(struct ucred**,
                      descp->vdesc_cred_offset, ap);
        vppp = VOPARG_OFFSETTO(struct vnode***, descp->vdesc_vpp_offset, ap);
        **vppp = null_make_nullnode(**vppp, VFSTONULLINFO(old_vps[0]->v_vfsp), *credpp);
     };

     return (error);
}
```

Figure A.7: The bypass routine for the null layer.

Appendix B

Cache-Coherence Appendix

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 dissertation 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.

B.1 Stack-Friendly Interface Changes

As described in Section 8.3, 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.

B.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.

B.3 Cache-Coherence Storage Options

Different data types have different caching needs. Often data semantics and run-time behavior limit data mutability, allowing data to be duplicated in different caches while guaranteeing coherence. In Section 6.3 we described how knowledge about these semantics allows the cache manager to improve performance.

In Table B.1 we summarize the five levels of service a layer can request when caching an object. The first four options combine two facts: will the layer invoking the cache manager cache the data-object, and can other layers which currently cache the data-object continue to cache it. This view of these states can be seen in Table B.2. If the current layer asks for rights to cache the object (as in the shared and exclusive cases), the cache manager promises to call those layers back should that cache need to be invalidated.

option	**meaning**
uncached	I don't cache; don't care about others.
non-cachable	I don't cache; others can't cache now.
shared	I cache (want callback); others can too.
exclusive	I alone can cache and want callback.
watch	I expect to see all cache actions.

Table B.1: Cache registration options.

	other layers	
your layer	**can cache**	**can't cache**
does cache	shared	exclusive
doesn't cache	uncached	non-cachable

Table B.2: An alternate view of cache registration options.

Independent of these options, with the "watch" request a layer will see all cache interactions for the given data-object and file. Layers requiring general naming register a "watch" request to monitor the cache and translate names as required.

In Table B.3 we enumerate the possible interactions between an outstanding cache request (one already registered with the cache manger) and a new request. In each case, if the new request conflicts with an existing request, a callback is made to invalidate the existing cache.

	outstanding cache request			
new cache request	**uncached/ non-cachable**	**shared**	**exclusive**	**watch**
uncached	—	—	error	callback
non-cachable	—	callback	callback	callback
shared	—	—	callback	callback
exclusive	—	callback	callback	callback
watch	—	—	—	callback

Table B.3: Interactions between a new cache request and existing cached objects. *Callback* indicates that the old layer has its caching privileges revoked. Error indicates an invalid state. No action is required for other states.

References

[ABG86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for UNIX Development." In *USENIX Conference Proceedings*, pp. 93–113. USENIX, June 9-13 1986.

[And90] Curtis Anderson. "UNIX System V Extended File Attributes Feature Requirements—Filesystem Dependent Attributes— Issue 1 (Draft 3)." Unix International Memorandum, October 1990.

[AT90] AT&T. "Design of the Virtual File System Features for UNIX System V Release 4." Internal memorandum, January 1990.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, 1986.

[BB92] D. S. Batory and J. R. Barnett. "DaTE: The Genesis DBMS Software Layout Editor." In Pericles Loucopoulos and Roberto Zicari, editors, *Conceptual modeling, databases, and CASE: an integrated view of information systems development*, chapter 8, pp. 201–221. John Wiley & Sons, Inc., 1992.

[BBG88] 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.

[BO92] Don Batory and Seam O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components." *ACM Transactions on Software Engineering and Methodology*, **1**(4):355–398, October 1992. Also available as University of Texas TR-91-22.

[Bon94] Jeff Bonwick. "The Slab Allocator: An Object-Caching Kernel Memory Allocator." In *USENIX Conference Proceedings*, pp. 87–98. USENIX, June 1994.

[Cat92] Vincent Cate. "Alex—a Global Filesystem." In *Proceedings of the Usenix File Systems Workshop*, pp. 1–11, May 1992.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. "Implementation and Performance of Munin." In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 152–164. ACM, October 1991.

[DBR91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems." In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 122–136. ACM, October 1991.

[Dij67] Edsger W. Dijkstra. "The structure of the THE multiprogramming system." In *Proceedings of the Symposium on Operating Systems Principles*. ACM, October 1967.

[Dij68] Edsger W. Dijkstra. "Complexity controlled by hierarchical ordering of function and variability." Working paper for the NATO conference on computer software engineering at Garmisch, Germany, October 1968.

[Dun90] Ray Duncan. "Power Programming: Using Long Filenames and Extended Attributes, Part 1." *PC Magazine*, (April 24):317–328, 1990.

[Flo86] Rick Floyd. "Short-Term File Reference Patterns in a UNIX Environment." Technical Report TR-177, University of Rochester, March 1986.

[GC89] 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*, pp. 202–210. ACM, December 1989.

[GHM90] 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*, pp. 63–71. USENIX, June 1990.

[GJS91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. "Semantic File Systems." In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 16–25. ACM, October 1991.

[GLL90] 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*, pp. 15–26. IEEE, May 1990.

[Gro92] Unix International Stackable Files Working Group. *Requirements for Stackable Files*. Unix International, Parsippany, New Jersey, October 1992.

[Hen90] David Hendricks. "A Filesystem for Software Development." In *USENIX Conference Proceedings*, pp. 333–340. USENIX, June 1990.

[HKM88] John Howard, Michael Kazar, Sherri Menees, David Nichols andMahadev Satyanarayananand Robert Sidebotham, and Michael West. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[HMP89] Norman C. Hutchinson, Shivakant Mishra, Larry L. Peterson, and Vicraj T. Thomas. "Tools for Implementing Network Protocols." *Software—Practice and Experience*, **19**(9):895–916, September 1989.

[HP88] Norman C. Hutchinson and Larry L. Peterson. "Design of the x-Kernel." In *Proceedings of the 1988 Symposium on Communications Architectures and Protocols*, pp. 65–75. ACM, August 1988.

[HP94] 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.

[HP95] John Heidemann and Gerald Popek. "Performance of Cache Coherence in Stackable Filing." In *Proceedings of the 15th Symposium on Operating Systems Principles*. ACM, December 1995.

[HPA89] 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*, pp. 91–101. ACM, December 1989.

[HR94] Graham Hamilton and Sanjay Radia. "Using Interface Inheritance to Address Problems in System Software Evolution." In *Proceedings of the AC Workshop on Interface Definition Languages*. ACM, January 1994. Also available as Sun Laboratories technical report SMLI TR-93-21.

[IEE90] IEEE. "Standard for Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API)." Technical Report IEEE Std. 1003.1-1990, IEEE, 1990. Also available as ISO/IEC 9945-1: 1990s.

[Joh92] Michael K. Johnson. "The Linux Kernel Hackers' Guide." Anonymous ftp as ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/kernel-hackers-guide/khg-0.6.ps.gz, 1992.

[Kaz88] Michael Leon Kazar. "Synchronization and Caching Issues in the Andrew File System." In *USENIX Conference Proceedings*, pp. 31–43. USENIX, February 1988.

[KEH93] David Keppel, Susan J. Eggers, and Robert R. Henry. "Evaluating Runtime-Compiled Value-Specific Optimizations."

Technical Report 93-11-02, University of Washington, November 1993.

[Kim95] Ted Kim. *“Frigate: An Object-oriented File-System.”*. Master’s thesis, University of California, Los Angeles, 1995. To appear.

[Kle86] S. R. Kleiman. “Vnodes: An Architecture for Multiple File System Types in Sun Unix.” In *USENIX Conference Proceedings*, pp. 238–247. USENIX, June 1986.

[KM86] Michael J. Karels and Marshall Kirk McKusick. “Toward a Compatible Filesystem Interface.” In *Proceedings of the European Unix User’s Group*, p. 15. EUUG, September 1986.

[KN93a] 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. Also available as Sun Laboratories technical report SMLI TR-93-18.

[KN93b] Yousef A. Khalidi and Michael N. Nelson. “The Spring Virtual Memory System.” Technical Report SMLI TR-93-9, Sun Microsystems, February 1993.

[Koe87] Matt Koehler. “GFS Revisited or How I Lived with Four Different Local File Systems.” In *USENIX Conference Proceedings*, pp. 291–305. USENIX, June 1987.

[KP84] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice-Hall, 1984.

[Kue91] Geoff Kuenning. “Comments on CS239 Class Projects.” Personal communication, June 1991.

[Kue95] Geoffrey H. Kuenning. “Kitrace: Precise Interactive Measurement of Operating Systems Kernels.” *Software—Practice and Experience*, **25**(1):1–22, January 1995.

[Lan90] Peter S. Langston. “Unix Music Tools at Bellcore.” *Software—Practice and Experience*, **20**(S1):47–61, June 1990.

[LH86] 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*, pp. 229–239. ACM, August 1986.

[MA69] Stuart E. Madnick and Joseph W. Alsop, II. “A modular approach to file system design.” In *AFIPS Conference Proceedings Spring Joint Computer Conference*, pp. 1–13. AFIPS Press, May 1969.

[Mas92] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[McK95] Marshall Kirk McKusick. “The Virtual Filesystem Interface in 4.4BSD.” *Computing Systems*, **8**(1):3–26, Winter 1995.

[MD74] Stuart E. Madnick and John J. Donovan. *Operating Systems*. McGraw-Hill, 1974.

[MGH94] James G Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. “An Overview of the Spring System.” In *Proceedings of the Spring 1994 IEEE COMPCON*. IEEE, February 1994.

[MJL84] 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.

[NKM93] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany. “The Spring File System.” Technical Report SMLI TR-93-10, Sun Microsystems, February 1993.

[NWO88] 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.

[Ols93] Michael A. Olson. “The Design and Implementation of the Inversion File System.” In *USENIX Conference Proceedings*, pp. 205–217. USENIX, January 1993.

[OP92] Sean W. O’Malley and Larry L. Peterson. “A Dynamic Network Architecture.” *ACM Transactions on Computer Systems*, **10**(2):110–143, May 1992.

[Ous90] John K. Ousterhout. “Why Aren’t Operating Systems Getting Faster As Fast as Hardware?” In *USENIX Conference Proceedings*, pp. 247–256. USENIX, June 1990.

[PK84] Rob Pike and Brian Kernighan. “Program Design in the UNIX Environment.” *AT&T Bell Laboratories Technical Journal*, **63**(8):1595–1605, October 1984.

[PPT91] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. “Plan 9, A Distributed System.” In *Proceedings of the Spring 1991 EurOpen*, pp. 43–50, May 1991.

[PW85] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture*. The MIT Press, 1985.

[RAA90] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. “Overview of the Chorus Distributed Operating System.” Technical Report CS/TR-90-25, Chorus systèmes, April 1990.

[RFH86] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, and Kang Yueh. “RFS Architectural Overview.” In *USENIX Conference Proceedings*, pp. 248–259. USENIX, June 1986.

[Rit84] Dennis M. Ritchie. “A Stream Input-Output System.” *AT&T Bell Laboratories Technical Journal*, **63**(8):1897–1910, October 1984.

[RKH86] R. Rodriguez, M. Koehler, and R. Hyde. “The Generic File System.” In *USENIX Conference Proceedings*, pp. 260–269. USENIX, June 1986.

[RNP93] S. Radia, M. Nelson, and M. Powell. “The Spring Name Service.” Technical Report SMLI TR-93-16, Sun Microsystems, October 1993.

[Ros90] David S. H. Rosenthal. “Evolving the Vnode Interface.” In *USENIX Conference Proceedings*, pp. 107–118. USENIX, June 1990.

[Ros92] David S. H. Rosenthal. “Requirements for a “Stacking” vnode/VFS Interface.” Unix International document SF-01-92-N014, 1992.

[RT74] Dennis M. Ritchie and Ken Thompson. “The UNIX Time-sharing System.” *Communications of the ACM*, **17**(7):365–375, October 1974.

[SBM90] Alex Siegel, Kenneth Birman, and Keith Marzullo. “Deceit: A Flexible Distributed File System.” In *USENIX Conference Proceedings*, pp. 51–61. USENIX, June 1990.

[SGK85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. “Design and Implementation of the Sun Network File System.” In *USENIX Conference Proceedings*, pp. 119–130. USENIX, June 1985.

[SGN85] 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*, pp. 25–34. ACM, December 1985.

[Smi82] Alan J. Smith. “Cache Memories.” *ACM Computing Surveys*, **14**(3):473–530, September 1982.

[SS90] David C. Steere and James J. Kistler andMahadev Satyanarayanan. “Efficient User-Level File Cache Management on the Sun Vnode Interface.” In *USENIX Conference Proceedings*, pp. 325–332. USENIX, June 1990.

[Sun87] Sun Microsystems. “XDR: External Data Representation standard.” Technical Report RFC-1014, Internet Request For Comments, June 1987.

[Sun90] Sun Microsystems. “Network Extensible File System Protocol Specification, *draft*.” Available for anonymous FTP on titan.rice.edu as public/nefs.doc.ps, February 1990.

[SW93] Glenn C. Skinner and Thomas K. Wong. ““Stacking” Vnodes: A Progress Report.” In *USENIX Conference Proceedings*, pp. 161–174. USENIX, June 1993.

[Web93] Neil Webber. “Operating System Support for Portable Filesystem Extensions.” In *USENIX Conference Proceedings*, pp. 219–228. USENIX, January 1993.

[Wei95] Jeff Weidner. "The UCLA Extensible Attributes Layer." In preparation, 1995.

[Wit93] Mark Wittle. "LADDIS: The Next Generation In NFS File Server Benchmarking." In *USENIX Conference Proceedings*, pp. 111–128. USENIX, June 1993.

Colophon

This document was produced using LaTeX2e on a a Sun IPC running SunOS 4.1.1 Unix modified to support UCLA stacking and the Ficus replicated file-system, and on a Dell Latitude XP portable computer running the Linux operating system (a version of Unix). I employed idraw for figures and jgraph for most graphs.