Efficient User-Level File Cache Management on the Sun Vnode Interface

David C. Steere James J. Kistler M. Satyanarayanan

April 18, 1990
CMU-CS-90-126

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in Summer Usenix Conference Proceedings, Anaheim, June (1990).

Abstract

In developing a distributed file system, there are several good reasons for implementing the client file cache manager as a user-level process. These include ease of implementation, increased portability, and minimal impact on kernel size. For reasons of compatibility it is also desirable to use a standard file intercept mechanism on the client. The Sun VFS/Vnode file system interface is such a standard. However, this interface is designed for kernel-based file systems, and a user-level cache manager that used the Vnode mechanism would pay a large performance penalty due to the high number of kernel to cache manager context switches per file system call.

This paper describes our solution to the problem for the Coda file system. By using a relatively small amount of kernel code to cache critical information, we are able to retain the much larger and more complex components of the Coda cache manager in a user level process. The measurements of Coda presented here confirm the performance benefits of this strategy, and indicate the relative merits of caching different kinds of information in the kernel.

This research was supported in part by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20 and in part by the National Science Foundation under Contract No. CCR-8657907, IBM Corporation (Faculty Development Award and Graduate Fellowship), Digital Equipment Corporation (Equipment Grant).

Keywords: Operating Systems, File Systems, Performance, Cache Management, Coda, Vnode Interface.

1. Introduction

In this paper we describe and evaluate our approach for implementing a user-level client cache manager on top of the Sun Vnode interface. Our work was carried out in the context of the *Coda file system* [5], a descendant of the *Andrew file system* (AFS) [4]. Coda is a highly available file system for a large-scale distributed computing environment composed of Unix[1] workstations. It provides resiliency to server and network failures through the use of two distinct but complementary mechanisms. One mechanism, *server replication*, stores copies of a file at multiple servers. The other mechanism, *disconnected operation*, is a mode of execution in which a caching site temporarily assumes the role of a replication site. Disconnected operation is particularly useful for supporting portable workstations.

The design of Coda optimizes for availability and performance, and strives to provide the highest degree of consistency attainable in the light of these objectives. In the presence of network partitions, Coda places availability above consistency and allows potentially conflicting updates to occur. Such updates are detected as *inconsistencies* at the earliest possible time. The most complex part of Coda is the client cache manager, known as *Venus*. Besides cache management, Venus is responsible for emulation of Unix semantics, coordination of the replica control algorithm, detection of inconsistencies, and disconnected operation.

Given the complexity of Venus, we saw good reasons for it to be a user-level process rather than part of the kernel. Our experience has been that kernel code takes much longer to develop, debug and maintain. We felt that designing Venus as a user-level process would substantially ease our implementation effort and also increase Coda's portability.

A different design consideration, with conflicting implications, was our desire to adhere to standards. We saw many advantages, particularly that of portability, in using the Sun Vnode interface [2]. This interface is a *de facto* standard for file system call interception in Unix. Unfortunately, it imposes a component-by-component interaction between the kernel and the cache manager when translating pathnames. Consequently, a naive user-level cache manager implementation on this interface would result in unacceptable overhead.

To reduce this overhead, we decided to build a caching module in the kernel, called the *MiniCache*. Since most Unix applications show strong locality of reference, we felt that this would achieve a large reduction in the number of calls to Venus. We also conjectured that much of this reduction could be achieved with a MiniCache whose size and complexity were a small fraction of that of Venus. Our conjecture has indeed proved to be true. The performance of Coda with the MiniCache is comparable to that of the current production version of AFS, whose cache manager is in the kernel.

2. Design Considerations

The dominant influence on the design of the MiniCache was the Vnode interface. This interface was created in conjunction with the Sun *Network File System* (NFS) [3]. It behaves as a file system multiplexor, permitting multiple file systems to coexist in the same name space. The interface specifies two types of operations: those on a *Virtual File System* (VFS), and those on a *Virtual Inode* (Vnode). VFS operations

[1]Unix is a trademark of AT&T.

such as `mount` and `unmount` pertain to the binding of different file systems into one name space. Vnode operations such as `open`, `close`, and `getattr` pertain to individual files and directories. To add a new file system, one need only provide a *VFS driver*, which provides an entry point for each VFS and Vnode operation.

An implicit assumption of the Vnode interface is that the VFS driver resides in the kernel. The most obvious manifestation of this assumption is the strategy used to perform pathname translation. Rather than passing the entire pathname to the driver, translation is performed on a component-by-component basis. Although this simplifies symbolic link traversal across file systems, it would impose a serious performance penalty on an out-of-kernel driver when translating multi-component pathnames.

We therefore decided to split the VFS driver into two pieces: a small piece inside the kernel (the MiniCache), and a more sophisticated piece outside the kernel (Venus). Essentially, the MiniCache behaves as buffering agent between the Vnode interface and Venus, as illustrated in figure 1. When an application program generates a file system call on an object in Coda, it is intercepted by the Vnode interface and routed to the MiniCache. If the MiniCache has sufficient information, it services the call. Otherwise, it passes the request to Venus. Venus handles the request, possibly contacting Coda servers in the process. Control is returned to the application via the MiniCache and the Vnode interface.

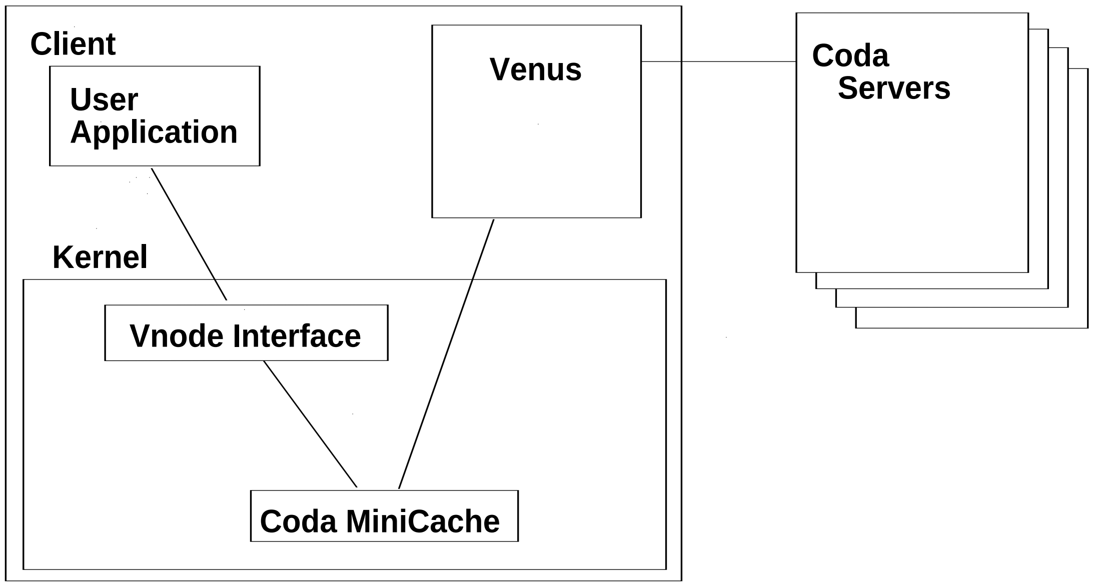

Figure 1: Coda Structure

A consequence of the splitting of the VFS driver is that state is shared between its two components. The correctness of the system requires coherence in this shared state. It is clear to see that mutating file system calls will change MiniCache state, and that these changes must be propagated to Venus and thence to the servers. But there are also situations in Coda where Venus may need to invalidate MiniCache state. Such invalidations may arise in a number of different ways. First, a server may invalidate a cache entry in Venus because of a modification at another client. Second, authentication tokens held by Venus on behalf of a user may expire, implicitly revoking his privileges. Third, Venus may discover that an object is inconsistent, and therefore discard all local knowledge of it. Shared state also has an important implication for the communication mechanism between the MiniCache and Venus: both parties need to be able to initiate message exchanges.

A broader consideration is the need to strike the right balance between complexity and efficiency. The MiniCache needs to be sophisticated enough to improve performance substantially, but should not be so complex that it amounts to moving Venus completely into the kernel. Further, the MiniCache needs to be particularly efficient because it is in the critical path of many operations.

3. The MiniCache

This section describes four major aspects of the MiniCache. The first subsection discusses the main data structures. The second describes the communication between Venus and the MiniCache. The third presents the performance enhancements we have implemented, and the fourth discusses a limitation of the Vnode interface that we encountered when dealing with the `exec` system call.

3.1. Data Structures

The MiniCache consists of two interdependent caches. The first cache, the Cnode cache, contains a set of *Cnodes*, which are Vnodes augmented with Coda-specific fields. The latter include the *CFid* (an identifier unique to each Coda object) as well as fields used in the performance enhancements described in Section 3.3. For efficiency, Cnodes are kept on an internal freelist rather than being deallocated. To provide Venus with a fast access path, this cache can be looked up via a hash table keyed on CFids.

The second cache, the name cache, contains precomputed translations of pathnames. Each entry in the cache is indexed by a triple: a pointer to the Cnode for a directory, a component name in that directory, and a user id. A Coda object may have multiple entries in the cache, one for each user who has accessed the file recently. The value of the cache entry is a pointer to the Cnode corresponding to that component. The name cache supports the basic operations of insertion, lookup, and deletion. Deletion happens to be a particularly complex operation since it occurs in multiple flavors. The simplest case is when a deletion occurs via the Vnode interface. More complex cases occur when Venus deletes entries for one of the three reasons cited in Section 2.

The functioning of the name cache can be best understood by following a typical pathname translation. At any step in the translation one has a pointer, *DCP*, to the Cnode of a directory, a name, *N*, in that directory, and the id, *U*, of the user on whose behalf the translation is being performed. If the triple *(DCP, N, U)* is missing from the name cache, Venus is contacted to fill the entry. Translation continues by replacing *DCP* by the value of the cache entry, *N* by the next component of the pathname, and repeating the cache lookup. Translation begins with *DCP* set to the root (for absolute pathnames) or the current directory (for relative pathnames).

Our name cache is similar in many ways to the Directory Name Lookup Cache (DNLC), supplied with the Sun Vnode interface[3]. Unfortunately the DNLC does not provide the full range of invalidation functions needed by Venus. Rather than modify it, we chose to build our own name cache.

3.2. Communication

In addition to the need for either peer to initiate action, the communication mechanism between the MiniCache and Venus needs to satisfy two conditions. First, it has to be interruptible in a manner that allows cleanup of state at both ends. Second, it must provide at-most-once semantics since many of the operations are not idempotent.

We initially tried using Sun RPC [6] for communication between the MiniCache and Venus. This would have enhanced the portability of the MiniCache, since Sun RPC is normally available in any kernel that has the Vnode interface. Unfortunately, we found it impossible to meet both the constraints mentioned above using that approach.

Consequently, we settled on a pseudo-device as the basis of our communication. The device is used to pass messages between the MiniCache and Venus. At-most-once semantics is trivially obtained because there are no retransmissions or timeouts. Interruptibility is ensured by careful implementation of the device driver. In addition, the pseudo-device supports concurrency by permitting an arbitrary number of outstanding messages.

Although the use of a pseudo-device limits portability to Unix-like systems, the communication code is limited to a small section of the MiniCache. The pseudo-device could easily by replaced by any other reasonable IPC mechanism.

3.3. Performance Enhancements

There are many conceivable ways in which the MiniCache can be used to improve performance. Given our desire to keep the MiniCache simple, we have only chosen to make those performance optimizations that provide substantial benefit relative to implementation complexity. We elaborate on these optimizations in this section.

The first, and most obvious, optimization is the reduction in MiniCache-Venus traffic during pathname translation. This reduces the average cost of the `lookup` Vnode operation, and is provided by the name cache described in Section 3.1.

The second optimization eliminates the need for the MiniCache to contact Venus on individual read and write operations. Besides reducing the number of MiniCache-Venus interactions, this substantially reduces the amount of data copying done by Coda. On a successful `open` of a file, the MiniCache saves a pointer to the Vnode of the locally cached copy. It can then service `rdwr` Vnode operations on the file by redirecting them to the local file system. `Readdir` Vnode operations on directories are handled in a similar manner. The MiniCache notifies Venus of `close` operations, giving Venus the opportunity to write a file back to the server if necessary.

The next performance enhancement we implemented was to provide an attribute cache, allowing `getattr` operations to be serviced by the MiniCache without contacting Venus. Such operations occur frequently, typically due to the `stat` system call which is extensively used by Unix applications. Space for the cache is provided as an extension of each Cnode, with a flag indicating whether the attribute fields are valid. The attributes for an object are invalidated when Venus flushes the object from its cache. Attributes are also invalidated in two other cases: when a file that has been modified is closed, and when a child of a directory is modified. Access permissions are correctly enforced for `getattr` since a user needs appropriate access rights to lookup the Cnode.

The fourth enhancement takes advantage of the presence of the name cache to reduce the number of `access` calls to Venus. `Access` is a Vnode operation that is used to determine a user's permissions on an object. Checking access rights in Coda is substantially more complex than checking Unix mode bits because Coda uses an access list mechanism with group inheritance. A full implementation of `access` in the MiniCache would greatly add to its complexity. Fortunately, we have been able to trivially implement an important subset of the full access check. It turns out that a large percentage of the `access` operations occur during pathname translation, and simply determine if a user has permission to lookup entries in a directory. Since name cache entries include the user id as part of the key, we are able to infer lookup permission on a cache hit. In all other cases, the MiniCache forwards `access` operations to Venus.

The last performance enhancement is to speed up symbolic link traversal by caching the link value within the Cnode. Symbolic links are frequently encountered during pathname translation, particularly in Unix environments that support heterogeneous machines. The symbolic link cache does not violate access constraints because a user needs to have proper access to look up the Cnode of the directory containing a symbolic link. In Unix, lookup permission on a directory automatically implies permission to read symbolic links in that directory.

3.4. Program Execution

In the course of our implementation we ran into a serious limitation of the Vnode interface: there is no provision for a VFS driver to be notified of `exec` and `exit` system calls. All the driver sees are `read` operations on the individual pages of an executable file. This is a problem for any stateful file system, since it now has to infer when to allocate and free internal state. For example, it needs to guess when a file must be cached, and when it is safe to flush it.

This problem is magnified for Coda since Venus cannot directly access kernel data structures. Although it is possible for the MiniCache to infer opens, it does not possess enough of Venus' state to determine when it is appropriate to flush a file. On the other hand, Venus knows when it is appropriate to flush a file, but does not possess enough kernel context to know if this would be safe. So the flushing or replacement of a cache entry by Venus is preceded in our implementation by a call to the MiniCache to determine if the operation is safe. Besides degrading performance, this approach also limits portability since it depends on specific details of the virtual memory subsystem.

4. Performance Evaluation

Our evaluation of the MiniCache focuses on two important questions. First, how much improvement does the MiniCache provide, both in absolute terms and relative to the current production version of AFS with an in-kernel VFS driver? Second, what is the individual contribution of each performance enhancement?

The primary basis for this evaluation is the Andrew Benchmark[1], consisting of a series of Unix operations on a subtree containing the source code of a Unix application. The input to the benchmark consists of 70 files, totalling about 200 Kbytes. The benchmark consists of five phases: `MakeDir`, which constructs a copy of the source subtree; `Copy`, which copies the source files to the new subtree; `ScanDir`, which examines the status of each file in the target subtree; `ReadAll` which scans every byte in every file in the target subtree; and `Make`, which compiles and links all the files.

Since the Andrew benchmark does not make extensive use of symbolic links, we use a different criteria for measuring the effect of caching them. Details of this test are provided in Section 4.3.

The tests reported here were conducted on a single client and server located on a single segment of Ethernet. Both machines were IBM APC-RTs with 12 MB of memory, running the Mach 2.5 operating system. To limit interference, nonessential processes on the client were killed before running the benchmarks. Although experimental control on the servers and the network was less strict, the low variance in our results indicates that this was not a problem.

4.1. Comparison with AFS

In estimating the absolute and relative performance benefits of the MiniCache we have chosen to focus on two phases of the Andrew benchmark. These phases, `ScanDir` and `ReadAll`, are the most demanding phases in terms of pathname translation. They involve no server interactions in Coda and AFS because they operate on data cached in an earlier phase of the benchmark. Hence the performance degradation due to an out-of-kernel VFS driver would be most apparent in these phases.

Table 1 presents the elapsed times, in seconds, of these phases averaged over four runs of the benchmark. With the MiniCache turned on, Coda performs as well as AFS. Our original goal of obtaining good performance without moving Venus into the kernel has thus been met. Without the MiniCache, substantial degradation is apparent. This is to be expected, since all Vnode operations are simply redirected to Venus, suffering at least two context switches in the process.

Configuration	ScanDir	ReadAll
Coda without MiniCache	44 (1)	71 (1)
Coda with MiniCache	26 (1)	43 (1)
AFS (in-kernel cache manager)	26 (0)	43 (1)

Running times in seconds of the third and fourth phases of the Andrew Benchmark. Numbers in parentheses indicate standard deviations.

Table 1: Performance Improvements Due to the MiniCache

4.2. Individual Contributions

In this section we present the observed performance improvement due to each of the enhancements described in Section 3.3. The metric used is the number of Vnode operations seen by Venus during the running of the Andrew benchmark. This is a completely deterministic metric, for three reasons: our measurements were made after an initial run to warm cache entries for the source subtree, the benchmark is small enough to avoid cache overflow, and target subtrees were deleted between benchmark runs.

Table 2 shows the reduction in number of Vnode operations due to the individual performance enhancements, with the listed operations accounting for about 80% of the total number in the benchmark. The table indicates that the name cache eliminates all `lookup` operations on Coda.

VFS Operation	Enhancement		MkDir	Copy	ScanDir	ReadAll	Make	Total
`lookup`	Name Caching	off	33	745	1586	2139	886	5369
		on	0	0	0	0	0	0
`rdwr/readdir`	RdWr Intercept	off	0	332	136	498	2107	3073
		on	0	0	0	0	0	0
`getattr`	Attribute Caching	off	0	70	361	360	84	875
		on	0	0	83	0	29	112
`access`	Access Caching	off	0	70	241	500	202	1013
		on	0	70	68	234	201	573

This table contains a summary of the number of operations seen by Venus during a run of the Andrew Benchmark. The values for lookup are the number of lookup operations performed by Venus that succeeded. The Name Cache row indicates that all lookups of Coda files were handled in the kernel. ReadWrite is a combination of the number of rdwr and readdir Vnode operations seen by Venus.

Table 2: Contribution of Performance Enhancements

Although `rdwr` calls to Venus have been eliminated, our measurements indicate that there is not a correspondingly large savings in elapsed time. We suspect that this is because the cost of accessing the local disk far outweighs the cost of contacting Venus.

The number of `getattrs` that reach Venus has been reduced by nearly a factor of 8. The reason that `getattrs` are not totally eliminated is that attributes are flushed whenever a file is modified or a directory is updated. Thus the 83 getattrs in the `ScanDir` phase correspond to the 70 files that were created in the `Copy` phase and the 13 directories in which files were modified.

Table 2 also shows that our simple strategy for reducing `access` calls to Venus works very well: the number of such calls is reduced by nearly half. Complete elimination of `access` calls to Venus would have required much greater effort.

4.3. Symbolic Link Caching

To evaluate the effect of caching symbolic links, we wrote a small program to repeatedly invoke the `stat` system call on the head of a chain of symbolic links. The first name in the chain pointed to the second, which pointed to the third, and so on. We tried two variants of this test: one in which the symbolic links were relative, and the other in which the symbolic links were absolute pathnames that were 5 components long. The target file was the same in both cases, and all links and the target file were in the same directory.

Each `stat` generates a `lookup` on the name of symbolic link, which generates a `readlink` to get its value. This is then followed by a `lookup` on the name specified in the link and then a `getattr`. With relative pathnames, this amounts to two `lookups`, one `readlink`, and one `getattr`. The use of absolute pathnames adds five more `lookup` calls, one for each directory in the pathname.

Table 3 shows mean and standard deviation of the elapsed time in seconds to perform 10000 `stats` on each link of the chains. The table indicates that it takes about 10 seconds longer per link with absolute pathnames

Number of links	*Without Cache* Rel Path		*Without Cache* Abs Path		*With Cache* Rel Path		*With Cache* Abs Path	
1	84.0	(.21)	94.6	(.30)	9.9	(.07)	20.1	(.13)
2	160.4	(.19)	182.5	(1.56)	13.1	(.15)	33.2	(.10)
3	237.4	(.32)	268.3	(.29)	16.3	(.08)	46.5	(.12)
4	313.5	(.45)	359.1	(1.55)	19.4	(.14)	59.7	(.11)
5	393.2	(2.99)	443.9	(1.58)	22.6	(.08)	73.0	(.15)

This table contains the time in seconds to execute 10000 `stat` calls on symbolic links. Two kinds of links were tested, one using absolute pathnames as the values of the links and the other using relative pathnames. The number of links refers to the number of symbolic link expansions needed to find the real file. Each number is the average of four runs, with the standard deviation in parentheses.

Table 3: Performance of the Symlink Cache

than with relative pathname links. With the name cache enabled, all the `lookups` can be satisfied in the kernel. Thus the cost of looking up an entry in the name cache is roughly two tenths of a millisecond.

The table also shows that each additional link adds about 3 seconds with the symlink cache enabled, and about 77 seconds otherwise. Each link causes an additional `readlink` and `lookup` operation. Since the name cache was enabled in both cases, almost all of the 77 seconds is due to calls to Venus.

Symbolic links are used quite heavily in our environment. In the Coda source subtree alone, there are 439 symbolic links out of roughly 10,000 files and directories. Typically symbolic links occur high in the subtree, and are encountered a disproportionate fraction of the time. The benefits of symbolic link caching are therefore quite substantial.

5. Conclusion

Although early versions of AFS demonstrated that efficient user-level cache management is possible, they depended on a customized system call intercept mechanism. When AFS adopted the Vnode interface, Venus was moved into the kernel to avoid excessive performance degradation. Rather than follow this approach, we decided to investigate the possibility of retaining Venus as a user-level process.

Our approach has been to move relatively simple, yet critical, pieces of Venus functionality into the kernel. Our analysis of representative Unix file system activity indicates that five such pieces (pathname translation, data read/write, attribute read, access checking, symlink expansion) account for the bulk of the Vnode operations. A key aspect of our solution is that the code we have added to the kernel is relatively small and simple. Our results indicate that we are able to match the performance of an in-kernel client cache manager, AFS, in the most demanding phases of the Andrew benchmark.

Two aspects of the MiniCache limit its portability slightly. One is our use of a Unix device driver, rather than Sun RPC, for communication. The other is our dependence on code specific to the Mach virtual memory implementation in order to correctly handle the execution of files. These limitations are unfortunate, since portability was one of the factors motivating our use of the Vnode interface.

The concerns which motivated this work are fundamental to a distributed system. On one hand, performance concerns lead one to design the system as part of the kernel. On the other, issues of portability and maintainability suggest that development should proceed at the user-level. The key result of this work is that an efficient portable user-level cache manager can be built on the Sun Vnode interface. In addition, it can be achieved via a small and simple piece of code in the kernel.

References

[1] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988).

[2] Kleiman, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Summer Usenix Conference Proceedings, Atlanta* (1986).

[3] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and Implementation of the Sun Network File System. In *Summer Usenix Conference Proceedings, Portland* (1985).

[4] Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., Spector, A., and West, M. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island* (Dec. 1985).

[5] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput. 39*, 4 (Apr. 1990).

[6] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol specification version 2. Tech. Rep. RFC-1057, SRI Network Information Center, June 1988.