# Evolving the Vnode Interface

*David S. H. Rosenthal*

Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043

*ABSTRACT*

The vnode interface has succeeded in supporting a wide range of file system implementations over its 6-year history. During that time it has also had to accommodate evolution in file system semantics, and in the relationship between the file system and the virtual memory system. The effects of this evolution have been less than elegant, and pressures for further evolution are mounting.

The evolution of the interface is reviewed in order to identify the problems it has caused, and a more robust revision of the interface design proposed. This design also permits new file systems to be implemented in terms of pre-existing file system implementations; it is more like the Streams interface in this respect. The current state of a prototype implementation is described and its performance characterized.

MELENE (*lazily*). We are very well as we are. Life without a care − every want supplied by a kind and fatherly monarch, who, despot though he be, has no other thought than to make his people happy − what have we to gain by the great change that is in store for us?

Gilbert & Sullivan, *Utopia, Ltd.*

## 1. Introduction

The vnode interface was developed in 1984 to abstract out file system operations in order to support multiple file system implementations, in particular NFS[8] and the Berkeley file system.[9] Over the 6 years since then it has been very successful; many versions of the UNIX† kernel use it in some form, and many different file system implementations have been based on it. During that time it has also had to accommodate evolution in file system semantics, and in the relationship between the file system and the virtual memory system.[2-4,12] The effects of this evolution have been less than elegant, and pressures for further evolution are mounting.

I review the evolution of the interface in order to identify the problems it has caused and, using experience with a SunOS 4.1-based prototype, I propose a more robust revision of the interface design. This design also permits new file systems to be implemented in terms of pre-existing file system implementations; it is more like the Streams interface[11,16] or layered protocols in the *x*-kernel[7] in this respect. I also characterize the performance of the prototype.

It is important to stress that I am merely reporting the current state of research work in progress; I know of no current plans to make changes like the ones I describe in Sun products.

## 2. Review of Vnode Evolution

The original vnode design had the following goals:

- Provide a well-defined interface between file system implementations and the rest of the kernel.
- Support but not require Unix file system semantics. In particular it should support local disk file systems, statefull and stateless remote file systems, and other file systems such as that of MS-DOS.



† UNIX is a trademark of Bell Laboratories.

- Define an interface that can be used by kernel-resident implementations of remote file servers.
- All file system operations should be atomic.

```
enum vtype {
        VNON, VREG, VDIR, VBLK,
        VCHR, VLNK, VSOCK, VBAD
};
struct vnode {
        u_short         v_flag;
        u_short         v_count;
        u_short         v_shlockc;
        u_short         v_exlockc;
        struct vfs      *v_vfsmountedhere;
        struct vnodeops *v_op;
        union {
                struct socket   *v_Socket;
                struct stdata   *v_Stream;
        } v_s;
        struct vfs      *v_vfsp;
        enum vtype      v_type;
        caddr_t         v_data;
};
struct vnodeops {
        int     (*vn_open)();
        int     (*vn_close)();
        int     (*vn_rdwr)();
        int     (*vn_ioctl)();
        int     (*vn_select)();
        int     (*vn_getattr)();
        int     (*vn_setattr)();
        int     (*vn_access)();
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_bmap)();
        int     (*vn_strategy)();
        int     (*vn_bread)();
        int     (*vn_brelse)();
};
```

Figure 1: Original Vnode Interface

To these, the implementation added:

- There should be little or no performance degradation.
- Static table sizes should not be required.
- File systems should not be forced to use central resources.
- The interface should be re-entrant.
- An ‘‘object-oriented’’ programming approach should be used.
- Each operation is done on behalf of the current process.

These goals resulted in the design outlined in Figure 1. Each vnode contains a small amount of data and a pointer to an ‘‘ops vector’’, a structure defining the operations that the rest of the kernel can invoke on the vnode object. In C++ terminology,[15] the entries in the ops vector are the virtual functions of the vnode class. The detailed semantics of the virtual functions of the various vnode versions aren’t important for the argument of this paper.

These operations are invoked via macros like:

```
#define VOP_FOO(vp) (*(vp)->v_op->vn_foo)(vp)
```

Note that for a particular machine architecture, these structures and the calling conventions for the functions define a binary interface; C is used here merely as a way of making it legible.

### 2.1. SunOS 4.X

By the release of SunOS 4.0 in 1988, considerable evolution had occurred in the vnode interface, as shown in Figure 2. Three fields had been added to the vnode, increasing its size by 8 bytes, and 9 entries had been added to and 4 entries deleted from the ops vector (see Table 1 below). These changes were motivated by:

- The rewrite of the virtual memory system. This unified file I/O and paging, replacing the buffer cache operations (*vn_bmap*, *vn_strategy*, *vn_bread*, *vn_brelse*) with paging operations (*vn_getpage*, *vn_putpage*, *vn_map*), and required a new field (*v_Pages*) in the vnode.
- The representation of special files as a file system type. This added a field (*v_rdev*) and an operation (*vn_realvp*).
- System V support. This added a field (*v_filocks*), an operation (*vn_cntl*), and a new vnode type (VFIFO).

### 2.2. System V Release 4

By the release of System V Release 4 in 1989, the vnode had evolved further. As shown in Figure 3, the structure had lost 3 fields and gained 8 (all reserved for future use), expanding to 72 bytes. The ops vector had gained 8 actual operations plus another 32 reserved for future use. Among the motivations for these changes were:

```
enum vtype {
        VNON, VREG, VDIR, VBLK,
        VCHR, VLNK, VSOCK, VBAD, VFIFO
};
struct vnode {
        u_short         v_flag;
        u_short         v_count;
        u_short         v_shlockc;
        u_short         v_exlockc;
        struct vfs      *v_vfsmountedhere;
        struct vnodeops *v_op;
        union {
                struct socket   *v_Socket;
                struct stdata   *v_Stream;
                struct page     *v_Pages;
        } v_s;
        struct vfs      *v_vfsp;
        enum vtype      v_type;
        dev_t           v_rdev;
        long            *v_filocks;
        caddr_t         v_data;
};
struct vnodeops {
        int     (*vn_open)();
        int     (*vn_close)();
        int     (*vn_rdwr)();
        int     (*vn_ioctl)();
        int     (*vn_select)();
        int     (*vn_getattr)();
        int     (*vn_setattr)();
        int     (*vn_access)();
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_lockctl)();
        int     (*vn_fid)();
        int     (*vn_getpage)();
        int     (*vn_putpage)();
        int     (*vn_map)();
        int     (*vn_dump)();
        int     (*vn_cmp)();
        int     (*vn_realvp)();
        int     (*vn_cntl)();
};
```

Figure 2: SunOS 4.X Vnode Interface

- The replacement of Unix domain sockets by Streams, which removed a vnode type and a field in the vnode.
- The need to support Xenix semantics, which added a vnode type.
- The removal of Berkeley-style locks, which removed two vnode fields.
- Additional remote file system support, which added 5 new operations.

### 2.3. Problems of Evolution

The evolution of the vnode is summarized in Table 1. There has been a steady growth in the size of the vnode and the number of operations in the ops vector.

Table 1: Vnode Evolution

| Release | Year | Fields | Bytes | Ops |
|---|---|---|---|---|
| SunOS 2.0 | 1985 | 11 | 32 | 24 |
| SunOS 4.0 | 1988 | 14 | 40 | 29 |
| SVR4 - fill | 1989 | 11 | 40 | 37 |
| SVR4 + fill | 1989 | 19 | 72 | 69 |
| Prototype | 1989 | 6 | 20 | 39 |

In System V Release 4 almost half the vnode structure and almost half the ops vector are devoted to preparing for future evolution. I believe this demonstrates that our current techniques for dealing with the evolution of kernel interfaces are inadequate. It cannot be a good idea to impose 80+% space overheads on data structures in order to cope with future change. It appears that a revision of the vnode interface to be more robust in the face of changing demands for file system functionality is required.

## 3. Design

What is this 80% overhead intended to achieve? The problem is that customer kernels have to be built from components supplied in object form by a number of suppliers. We want to let independent software vendors (ISVs) supply file system implementations if they need new file system semantics to achieve their ends, and to do so in object form to protect their investment. The processes of releasing and distributing software mean that customers cannot expect to get both a new operating system release and the corresponding release of the ISV's product at the same time.

Thus, it must be possible for the customer to build a working kernel from object components with the kernel at a higher release number than some of the file system implementations it is using. Note that there is in general no customer demand for kernels in which the file system implementations are at a higher level than the rest of the kernel; in this situation it is relatively easy for the ISV to supply both old and new versions of their file system.

In attempting to revise the vnode interface, I had two main goals:

```
enum vtype {
        VNON, VREG, VDIR, VBLK,
        VCHR, VLNK, VFIFO, VXNAM, VBAD
};
struct vnode {
        u_short         v_flag;
        u_short         v_count;
        struct vfs      *v_vfsmountedhere;
        struct vnodeops *v_op;
        struct vfs      *v_vfsp;
        struct stdata   *v_stream;
        struct page     *v_pages;
        enum vtype      v_type;
        dev_t           v_rdev;
        caddr_t         v_data;
        struct filock   *v_filocks;
        long            v_filler[8];
};
struct vnodeops {
        int     (*vop_open)();
        int     (*vop_close)();
        int     (*vop_read)();
        int     (*vop_write)();
        int     (*vop_ioctl)();
        int     (*vop_setfl)();
        int     (*vop_getattr)();
        int     (*vop_setattr)();
        int     (*vop_access)();
        int     (*vop_lookup)();
        int     (*vop_create)();
        int     (*vop_remove)();
        int     (*vop_link)();
        int     (*vop_rename)();
        int     (*vop_mkdir)();
        int     (*vop_rmdir)();
        int     (*vop_readdir)();
        int     (*vop_symlink)();
        int     (*vop_readlink)();
        int     (*vop_fsync)();
        void    (*vop_inactive)();
        int     (*vop_fid)();
        void    (*vop_rwlock)();
        void    (*vop_rwunlock)();
        int     (*vop_seek)();
        int     (*vop_cmp)();
        int     (*vop_frlock)();
        int     (*vop_space)();
        int     (*vop_realvp)();
        int     (*vop_getpage)();
        int     (*vop_putpage)();
        int     (*vop_map)();
        int     (*vop_addmap)();
        int     (*vop_delmap)();
        int     (*vop_poll)();
        int     (*vop_dump)();
        int     (*vop_pathconf)();
        int     (*vop_filler[32])();
};
```

Figure 3: SVR4 Vnode Interface

- 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* .

The idea of stacking vnodes is not new; several of the file systems in SunOS 4.1 (the *translucent*[6] and the *loopback* file systems, for example) implement vnodes whose operators simply invoke the operators of an underlying vnode with slightly altered arguments. But there had always been severe restrictions on the ways in which vnode operations could be overridden. I wanted to be able to override all vnode operations in a completely general way that would support breaking file system functionality down into small modules like Streams modules. Instead of viewing file systems as large monolithic structures, I wanted to be able to plug them together from smaller pieces.

### 3.1. Design Guidelines

To investigate solutions to these problems I evolved a prototype, starting with an alpha version of SunOS 4.1. The prototype has been running for quite some time, but it is very much the result of evolution not design. As I experimented with it I evolved a set of design guidelines:

- *Vnodes should stack.* In other words, it should be possible to interpose new functionality on all operations invoked on an existing vnode without locating all pointers to it and updating them. This is especially important since each page structure contains a vnode pointer, and requiring file system code to find and update all the page structures leads to an inadmissible mixing of the file and virtual memory sub-systems.
- *In fact, vnodes should tree.* It should be possible for one higher-level vnode to represent a number of lower-level vnodes. As an example, consider a fan-out-fs whose operations simply invoke the corresponding operation on all of a set of underlying vnodes (Figure 4).

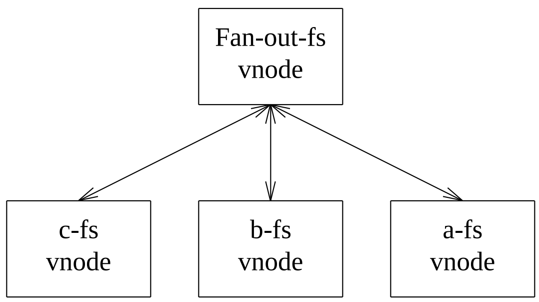


Figure 4: A vnode tree

- *No special-case code for mount.* The whole concept of stacking vnodes is a generalization of the concept of mount. It should be possible to replace the special-case code in name lookup that currently implements mounts, and the *vfs_mountedhere* field in the vnode.
- *Vnodes should be opaque.* In other words, the data structure visible to higher levels of the kernel should contain no data, only a pointer to the ops vector. A visible datum in the vnode represents a possible operation (updating the value of the datum) that cannot be overridden via the ops vector, because it doesn't need to go via the ops vector.

  Further, one of the problems of evolving the interface is that the visible data in the vnode structure changes. If there is no data in the vnode, it cannot change.
- *Vnodes should be cheap.* If vnodes are to stack (or tree) there are likely to be several vnodes where at present there is only one. So, space in the vnode is at a premium. An important advantage of opaque vnodes is that the data that they don't contain doesn't take up space. Nor is there any need to reserve space in the shape of *v_filler[8]* in case the non-existent data expands.

  It might be argued that all that opaque vnodes achieve is to move data from the public to the private part of a vnode. But, to my surprise, much of the data in the public part of the SunOS 4.X vnode was often present in some other form in the private part.
- *No vnode type.* A vnode type field or a vnode type operator is an invitation for the higher-level kernel code to test the type and behave differently depending on the result. This is likely to cause difficulties for file systems trying to intercept all operations. It is better for the rest of the kernel to treat all vnodes equally and leave all special case catching and error generation to the file-system specific code.

  For example, the SunOS 4.X kernel checks to see if the vnode in which it is being asked to look up a name is a directory, and generates ENOTDIR if it isn't. Why not simply call the vnode's *vn_lookup()* operator and let it generate the ENOTDIR if the vnode doesn't represent something that can have names looked up in it?
- *There should be a cheap way to lock the whole stack of vnodes.* While one process is manipulating a stack of vnodes, other processes must be prevented from invoking operations on any of the vnodes in the stack. Locking the stack in this sense shouldn't involve, for example, traversing the stack setting a lock flag in each vnode.
- *Vnodes should support versioning.* It should be possible for the higher-level kernel code to use file system implementations constructed with several versions of the interface, and similarly to build file system implementations that work with kernels that implement several different versions of the interface.

### 3.2. The New Vnode

I believe that the new vnode should look like Figure 5. This isn't quite the way it currently looks in my prototype*, but the prototype bears the scars of a lot of exploratory hacking.

Note that the vnode is almost opaque; except for the reference count all the visible fields are needed to implement the vnode stack. This is held together by the *v_top* and *v_above* fields as shown in Figure 6.

There is no public *v_below* pointer. The representation of the set of vnodes, if any, below a vnode is private to that vnode's file system implementation; no higher-level code knows or cares about the vnodes that may exist below a vnode. The absence of a public down pointer is a principle; if there were one it would allow higher-level code to traverse the vnode trees and invoke lower-level vnode's operations without the intervening vnode finding *out*.

Instead of indirecting through the vnode to find the ops vector and invoking the appropriate operation as with previous vnode interfaces, the macro finds the top vnode of the stack and invokes the appropriate operation from *its* ops vector. Even if the vnode pointer points into the middle of the stack, the code that gets invoked will be the corresponding operation of the top vnode of the stack. Whether that vnode executes the operation itself, or forwards it to other vnodes

* The prototype hasn't yet demonstrated versioning, some of its vnode operations return values other than an error code, and its ops vector still contains some operations that should be obsoleted. Further, it isn't compatible with SunOS 4.1 in that it returns the wrong code on some errors.

```
struct vnode {
        struct vnode    *v_top;
        struct vnode    *v_above;
        struct vnodeops *v_op;
        u_short         v_readers;
        u_short         v_count;
        caddr_t         v_data;
};
struct vnodeops {
        int     (*vn_version)();
        int     (*vn_open)();
        int     (*vn_close)();
        int     (*vn_rdwr)();
        int     (*vn_ioctl)();
        int     (*vn_select)();
        int     (*vn_getattr)();
        int     (*vn_setattr)();
        int     (*vn_access)();
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_lockctl)();
        int     (*vn_fid)();
        int     (*vn_getpage)();
        int     (*vn_putpage)();
        int     (*vn_map)();
        int     (*vn_dump)();
        int     (*vn_cmp)();
        int     (*vn_cntl)();
        int     (*vn_vfsp)();
        int     (*vn_socket)();
        int     (*vn_stream)();
        int     (*vn_bsdlock)();
        int     (*vn_bsdunlock)();
        int     (*vn_isswap)();
        int     (*vn_swapon)();
        int     (*vn_push)();
        int     (*vn_pop)();
        int     (*vn_namepage)();
        int     (*vn_unnamepage)();
        int     (*vn_pagecount)();
        int     (*vn_pageapply)();
};
```

Figure 5: The Ideal Vnode

The higher-level code invokes operations using its vnode pointer like this:

```
#define VOP_FOO(vp) (*(vp)->v_top->v_op->vn_foo)(vp)
```

below it, is no concern of the invoker's. In effect, a pointer to a vnode becomes an alias for the top vnode of the stack.

Reference counting of these links is simple; the *v_above* pointers hold a counted reference to the vnode they point to but the *v_top* pointers do not. Normally, the file-system specific downward links do hold their vnode.

## 4. Implementation

### 4.1. Stack Manipulation Operators

Vnode stacks can be manipulated by two new vnode operators, VOP_PUSH and VOP_POP.

```
VOP_PUSH(below, above)
```

arranges that *above* is the new top vnode of the stack containing *below*. If *vp* is the top of a stack,

```
VOP_POP(vp)
```

pops it off. The implementation of these operators is file-system specific because they must deal with the private representation each implementation uses for the set of vnodes below a vnode, and because these operations may have file-system specific side effects.

The mount operations use these facilities. A mount is simply a stack of a root directory over a mount point directory; lookup operations on the lower vnode automatically look up in the upper vnode with no special-case code in the file-system independent parts of the kernel. The only special-case code left is in each file system. It arranges to fall through to the lower vnode if ".." is looked up in the upper one.

In the current implementation, the definition of "top" is stretched slightly. The first time VOP_PUSH is applied to a vnode, a special stack head vnode is created which floats above any vnodes pushed on the stack. VOP_POP removes the vnode below this stack head. This avoids the need to descend the stack and update the *v_top* pointers when pushing and popping, and simplifies locking.

### 4.2. Stacks and the VM System

In the SunOS VM system, the page cache uses the vnode pointer and the offset to establish the identity of the page. Allowing multiple aliases for a vnode pointer introduces the potential for aliasing in the page cache. To avoid this, it is necessary to establish a rule for file system implementations.

The rule is, if a vnode belonging to a file system implementation that wants control over the page cache is pushed onto a stack, it must claim all existing pages for the stack below. When it is popped, it must either ensure there are none of its pages in the cache or restore the previous identity.

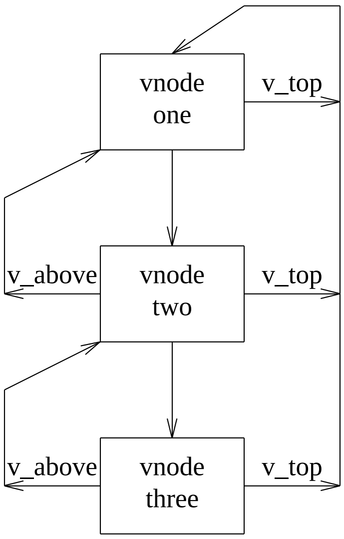


Figure 6: Vnode Links

A page in the cache is labelled with its identity by the VOP_NAMEPAGE operator, and unlabelled by VOP_UNNAMEPAGE. With the current VM system, these operators add and delete pages from the page hash lists. A function, such as one that renames pages, can be applied to all the cached pages for a vnode by:

```
error = VOP_PAGEAPPLY(vp, fn, data);
```

In principle, this interface allows each file system implementation to choose its own representation for the set of in-memory pages. In practice, more work is needed to refine the VM/FS interface to make this feasible. The VM cache and locking (see the next section) have an incestuous relationship, and I'm far from happy with the current implementation in either respect.

### 4.3. Locking

In order to ensure other processes don't see a malformed vnode stack, there must be a method of locking the entire stack while it is being manipulated. It must be cheap and cannot involve traversing the stack itself.

The technique I'm currently working on puts a readers/writers lock in the special stack head vnode. Before changing the stack, a writer switches the ops vector in the stack head from one containing operations like Figure 7(a) to one containing operations like Figure 7(b). These then ''capture'' any process invoking an operation on any vnode pointer in the stack and puts it to sleep. The writer waits until there are no readers, edits the stack, and switches the ops vector of the stack head back. This technique has no overhead for vnodes that aren't part of a stack, and only a tail-recursion-type procedure call for stack vnodes that are not locked. However, the cost of maintaining the count of the number of readers in a stack is significant.

```
static int
snarefs_foo(vp)
        struct vnode *vp;
{
        return(VOP_FOO(vp));
}
```

Figure 7(a): Vnode Operation When Unlocked

```
static int
snarefs_foo(vp)
        struct vnode *vp;
{
        int error;
        int s;

        s = splhigh();
        (void) VOP_HOLD(vp);
        while (vp->v_top->v_op == &snarefs_slow_ops) {
                (void) sleep((caddr_t)
                      &(vp->v_top->v_op), PSNARE);
        }
        (void) splx(s);
        error = VOP_FOO(vp);
        (void) VOP_RELE(vp);
        return (error);
}
```

Figure 7(b): Vnode Operation When Locked

### 4.4. Versioning

As described above, one advantage of opaque vnodes is that there is no need to version the vnode structure itself. But the evolution so far indicates that change in the ops vector must be anticipated. Fortunately, stacking opaque vnodes allows an ''adaptor file system'' to be defined to convert from the new version of the interface used by the kernel to the older interface used by the file system implementation (see Figure 8). All that the adaptor-fs vnode needs in its private data is the down pointer.

This technique's overheads are very small:

- no increase in size of the vnode structure,
- no increase in size of the ops vector,
- a small increase in run time for those vnodes supported by back-level file system implementations.

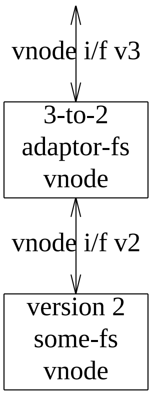


Figure 8: Adaptor-fs - v3 kernel to v2 fs

- a small increase in space consumption, for the adaptor-fs vnode for each vnode supported by a back-level file system implementation.

Despite this, the technique allows for more change in the interface than adding fillers. There are no arbitrary limits on how much the interface can change, provided the adaptor-fs can emulate the minimum required new functionality in the old interface's terms.

### 4.5. Performance

In my work on the prototype I have been exploring rather than tuning, but I have found one performance technique that could also be applied to the existing vnode interfaces. Current file system implementations have a single vnode ops vector; all vnodes they create have the same ops vector pointer (in fact, unpleasant parts of some kernels even use the ops vector pointer to decide which file system the vnode comes from!).

I changed the file systems to use a separate ops vector for each type of vnode they support. For example, the UFS file system implementation has a different vector for normal files and for directories. The *vn_lookup* entry in the directory vector points to *ufs_lookup*, but in the file vector it points to a routine like:

```
int
vfs_enotdir(vp)
        struct vnode *vp;
{
        return (ENOTDIR);
}
```

In this way, the file system implementation doesn't have to start all its operations by examining the type field and generating errors if it's wrong. This computation is done once at vnode create time instead of every time an operation is invoked. In fact, the prototype contains a vnode ops vector supported by the ‘‘Nancy Reagan’’ file system – *all* its ops vector entries point to error stubs like *vfs_enotdir()*.

In this framework, just as each vnode type can have its own ops vector, it can also have its own private data. When defining the inode-equivalent for a new file system, there is no need to provide space for the directory offset in the private data for a regular file.

Running one of Sun's synthetic workload benchmark suites on the prototype kernel initially revealed approximately 7% degradation, with special benchmarks for name lookup showing approximately 30% degradation. This was obviously unacceptable, even for a prototype. The major causes of degradation turned out to be:

- The vnode reference counting; in my zeal for completely opaque vnodes I had turned the VN_HOLD and VN_RELE macros into full-blown vnode operations called through the ops vector (8%).
- Detecting vnodes that are the root of a mounted file system; I had eliminated the VROOT flag in a very expensive way (18%).

Fixing these resulted in a kernel with no detectable degradation on the synthetic workload benchmarks and about 1% degradation in name lookup. The worst-case path I found was looking up ".." across a mount point, with about 6% degradation. However, this was all before I started working on locking. The various locking implementations I've tried so far have degradations in the 15-25% range for the worst-case paths in name lookup.

## 5. Using File System Modules

As well as providing for evolution this way of decomposing file system functionality into modules and connecting them at run-time in various ways allows many opportunities for new ways of implementing file system functionality. To illustrate them, I describe a few possible modules. I haven't implemented these ideas yet, I'm only discussing them to show the potential of the concept.

### 5.1. Quotas

Support for file-space quotas is an interesting example of the possible use of file system modules. In the current SunOS source, *#ifdef*

*QUOTA* appears in 22 files, ranging from *machdep.c* to *init_main.c*. Despite this, only the *ufs* file system supports quotas.

Suppose we construct a file system module that can be pushed on top of mounted file systems to provide quota services. It would intercept all operations to the file systems underneath and maintain an internal space usage database. Operations violating the quotas would be rejected. The quota-fs would use normal file system operations on the underlying file system to externalize the space database.

In this way, a single file system module implementation with no *#ifdef QUOTA* elsewhere in the kernel could provide quotas for *all* other file system implementations.

### 5.2. A Less Temporary File System

One of the performance problems with *ufs* is the need to write directories synchronously in order to ensure that they will still be there in a consistent form if the system crashes. This is a particular problem in */tmp*, where the user may not care if the files survive a crash. SunOS includes a file system implementation called *tmpfs* which represents files in virtual memory; they may get paged out to the swap area but they are never written to disk. Mounting this on */tmp* and */usr/tmp* improves performance significantly, at the cost of ensuring that *no* files in */tmp* and */usr/tmp* survive a crash.

There is only one problem with this approach. Some applications, *vi* for example, require temporary files to survive crashes and be scavenged before */tmp* is cleaned out. Fortunately, these applications normally *fsync()* their temporary files when checkpointing to make sure they don't hang around in the buffer cache and get caught by a crash. We can exploit this by building a write-on-fsync module like Figure 9.

The module would route all writes to the *tmpfs* vnode until and unless the process invoked the *fsync()* operator. At that point, the file would be copied to the underlying file system. In this way, files that were never *fsync()* ed would get the full benefit of being really temporary, and files that were would actually appear in permanent storage.

### 5.3. Watchdogs

Another possible module would implement *Watchdogs* as described by Bershad and Pinkerton.[1] The watchdog-fs vnode would intercept all or only the selected operations, convert them into

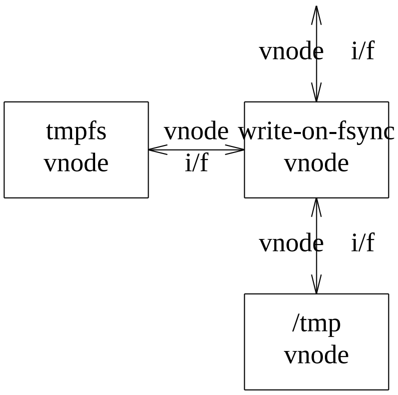


Figure 9: Less Temporary File System

IPC messages on a Stream, and wait for a response from the user-level watchdog process reading the Stream. The content of this response would determine if the operation was to be passed on down the stack, or returned to the invoker.

### 5.4. Read-Only Caching

Consider a module like that in Figure 10. When one of its operations is invoked from above, it forwards it to the a-fs vnode. If that operation succeeds the result is returned. Otherwise, the operation represents a cache miss and must be forwarded to the b-fs vnode. Typically, in this case the cache-fs will also try to prevent future cache misses, for example by copying the file from the b-fs to the a-fs.

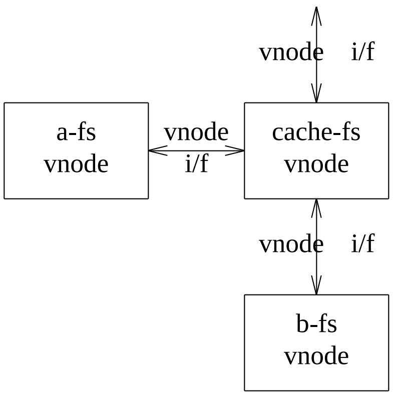


Figure 10: RO Cache File System

This simple module can use any file system as a file-level cache for any other (read-only) file system. It has no knowledge of the file systems it is using; it sees them only via their opaque vnodes. Figure 11 shows it using a local writable ufs file system to cache a remote read-only NFS file system, thereby reducing the load on the server. Another possible configuration would be to use a local writable ufs file system to cache a CD-ROM, obscuring the speed penalty of CD.

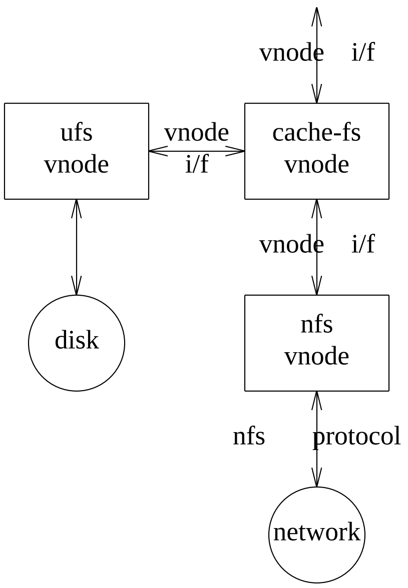


Figure 11: Local disk cache for RO NFS

File-level caching can be inefficient. If the file system being used as a cache can be persuaded, for example via a flag on the VOP_RDWR call, to reject reads to holes in files instead of returning zeros it can be used as a block-level cache. A new entry in the cache is created as a file full of holes, which are filled in as reads to the a-fs vnode fail with an appropriate error such as EHOLE. This is a very simple change to UFS.

### 5.5. Read-Write Caching

Srinivasan and Mogul[13,14] modified the NFS protocols by adding the cache-consistency protocols from the Sprite operating system.[10] In this environment, adding cache-consistency and thus enabling a read-write file cache need not involve modifying the NFS protocol. By allowing the cache-fs implementations to communicate amongst themselves using a cache-consistency protocol such as Sprite's or the V system's *leases*[5] alongside the NFS protocol we should be able to add caching without changing the protocol used to move data to and fro (see Figure 12).

### 5.6. Fall-back

Another useful module could be the fall-back-fs, shown in Figure 13. Each operation it receives from above is sent to one of the N underlying

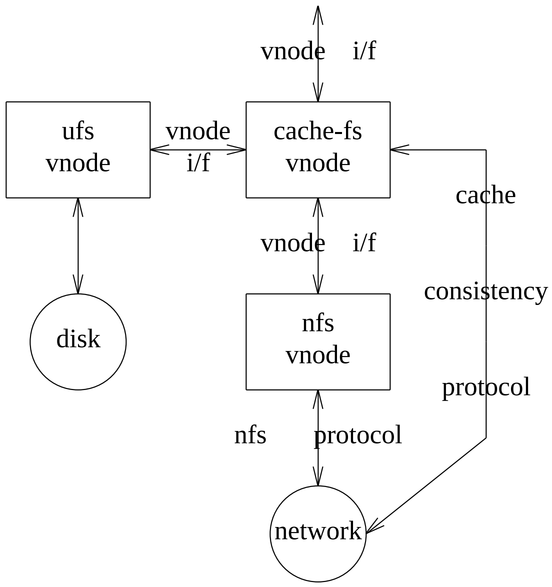


Figure 12: Local disk cache for RW NFS

vnodes. If it fails with a retryable error, or does not return before a timeout, another of the vnodes is chosen and the operation tried on it. In this way, the reliability and availability of the file system seen through the top vnode interface is greater than any of the individual underlying file systems, and this has been accomplished with no knowledge of or modification to the underlying file systems.

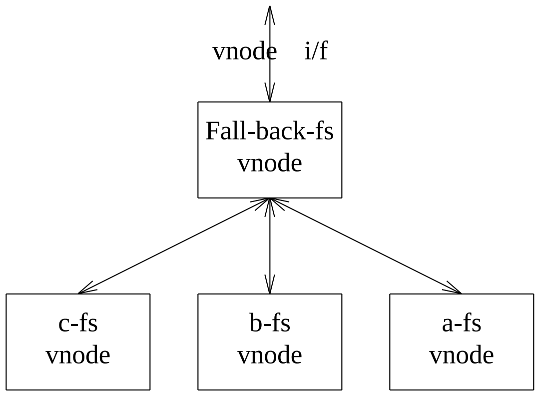


Figure 13: Fall-back-fs

Typically, this would be used to spread the load among a number of NFS file servers for read-only file systems, and to avoid clients being blocked when one of the servers went down.

### 5.7. Replication

A similar useful module is the replicate-fs, shown in Figure 14. Each operation it receives from above is sent to *each* of the N underlying vnodes. When M (<=N) of them has returned successfully, replicate-fs returns upwards. Read operations choose one of the N randomly. Implementations of replicate-fs communicate with each other using an ordering protocol to ensure that each sees the same sequence of operations and therefore stays consistent.

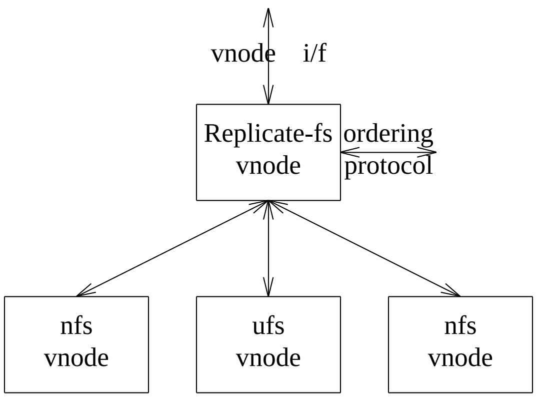


Figure 14: Replicate-fs

Using this module, a reliable replicated configuration could be assembled from unreliable pieces. For example, a set of machines could each operate by replicating operations across a local copy of a file system and a remote mount of each of the other system's copies.

## 6. Future Work

I have only started exploring these ideas. I have a working prototype SunOS 4.1 kernel that is close to the vnode interface I prefer, but I have done little work on the versioning aspects of the problem. I have converted most of the existing file system implementations in SunOS 4.1 to use the new interface, and I have started work on implementing new file system modules like cache-fs. But there is a great deal of work to do:

- I'm still not happy with the locking technique, nor with its relationship to the VM page cache. There are performance and semantic issues still to be resolved.
- The System V Release 4 kernel and its file system implementations need to be modified in the same way. I don't anticipate any major problems in doing this.
- More file system modules need to be implemented.
- The adaptor-fs versioning technique needs to be implemented and tested.
- Experiments using the same techniques on other internal kernel interfaces, particularly the VFS interface, are needed.
- The whole question of multi-threading has been totally ignored.

Assuming that these are all achieved, there are a large number of questions to be answered about what to do with this technology. The vnode interface is part of the System V Release 4 standard, and is thus under formal change control. Discussions will be needed to investigate what role these techniques might play in the future of System V.

## 7. Conclusions

As the demands on the system have changed, the vnode interface has evolved to match them. The techniques currently in use to cope with this evolution are too expensive. The alternative techniques of opaque vnode structures and stacking cope with evolution better at lower cost. But, more importantly, opaque vnodes can be assembled into tree structures. This allows file system functionality to be dissected into small, independent modules akin to Streams modules that are interconnected at run-time.

Of course, this is just another example of object-oriented programming. But unlike most examples, this is object-oriented programming at the binary level.

**Acknowledgements**

Many of these ideas have been rattling around in the back of my mind since the early discussions about the Andrew File System at Carnegie-Mellon's Information Technology Center, and they owe much to Bob Sidebotham. In particular, a somewhat hysterical brainstorming session at an ITC party between Bob, David Nichols and myself led to the ''holey cache-file'' concept.

Steve Kleiman and Bill Joy designed the original vnode interface, and have been extraordinarily helpful in my efforts to change it. The same applies to all my colleagues in the Systems Group at Sun, especially to Bill Shannon, Rob Gingell, Mike Powell, and Glenn Skinner. Special thanks are due to Steve Baumel for penetrating reviews of drafts of the paper. I'm also very grateful to Sun's management for their patience; although

this project has already taken many times longer than my initial estimate, and it is still far from finished, they have always encouraged me to take the time to ‘‘do the right thing’’.

**References**

1. Brian N. Bershad and C. Brian Pinkerton, *Watchdogs: Extending the UNIX File System,* pp. 267-276, Proceedings of the Winter 1988 Usenix Conference, Dallas, TX, February 1988.

2. Howard Chartock, *RFS in SunOS,* pp. 281-290, Proceedings of Summer Usenix Conference, Phoenix, AZ, June 1987.

3. Robert A. Gingell, Joseph P. Moran, and William A. Shannon, *Virtual Memory Architecture in SunOS,* pp. 81-94, Proceedings of Summer Usenix Conference, Phoenix, AZ, June 1987.

4. Ed Gould, *The Network File System Implemented on 4.3BSD,* pp. 294-298, Proceedings of Summer Usenix Conference, Atlanta, GA, June 1986.

5. Cary G. Gray and David R. Cheriton, *Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency,* pp. 202-210, Proceedings of the 12th ACM Symp. on Operating Systems Principles, Litchfield Park, AZ, December 1989.

6. David Hendricks, *The Translucent File Service,* pp. 87-93, Proceedings of the Autumn 1988 EUUG Conference, Vienna, Austria, October 1988.

7. Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O’Malley, *RPC in the x-Kernel: Evaluating New Design Techniques,* pp. 91-101, Proceedings of the 12th ACM Symp. on Operating Systems Principles, Litchfield Park, AZ, December 1989.

8. Steven R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX,* pp. 238-247, Proceedings of Summer Usenix Conference, Atlanta, GA, 1986.

9. M. Kirk McKusick and *et al*, ‘‘A Fast File System for UNIX,’’ *ACM TOCS*, vol. 2, no. 3, pp. 181-197, August 1984.

10. Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, ‘‘Caching in the Sprite Network File System,’’ *ACM Trans. on Computer Systems*, vol. 6, no. 1, pp. 134-154, February 1988.

11. Dennis M. Ritchie, ‘‘A Stream Input-Output System,’’ *AT&T Bell Laboratories Tech. J.*, vol. 63, no. 8, October 1984.

12. Mordecai B. Rosen and Michael J. Wilde, *NFS Portability,* pp. 299-305, Proceedings of Summer Usenix Conference, Atlanta, GA, June 1986.

13. V. Srinivasan and Jeffery C. Mogul, ‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols,’’ Research Rept. 89/5, Digital Western Research Lab., Palo Alto, CA, May 1989.

14. V. Srinivasan and Jeffery C. Mogul, *Spritely NFS: Experiments with Cache-Consistency Protocols,* pp. 45-57, Proceedings of the 12th ACM Symp. on Operating Systems Principles, Litchfield Park, AZ, December 1989.

15. Bjarne Stroustrup, *The C++ Programming Language,* pp. 201-203, Addison-Wesley, Reading, MA, 1987.

16. Ian Vessey and Glenn Skinner, *Implementing Berkeley Sockets in System V Release 4,* pp. 177-193, Proceedings of the Winter 1990 USENIX Conference, Washington, DC, January, 1990.

**About the Author**

David Rosenthal is a Distinguished Engineer at Sun Microsystems. He holds a M. A. degree from Cambridge University and a Ph. D. from the University of London, and has worked on computer graphics, user interface technologies, and operating systems at the University of Edinburgh’s Architecture Dept., the CWI Amsterdam, and Carnegie-Mellon University’s Information Technology Center.

He looks on this paper as a sign that he is making progress in recovering from a seven-year addiction to window systems.