LFS -- A Local File System for Multiprocessor NFS Network Servers

Allan M. Schwartz
David Hitz
William M. Pitts

Technical Report 4 December 1989

ABSTRACT

This report discusses the design and implementation of a special-purpose Unix file system interface, called LFS, that lies at the heart of a new multiprocessor NFS file server. This server addresses the latest generation of RISC-based Unix workstations, which have I/O demands that can be satisfied only by very high-performance file servers directly connected to multiple networks. LFS is organized into client and server modules that execute on different physical processors. The LFS interface (operation set) is an extension of the NFS interface. The LFS server implementation is based on the BSD 4.2 file system (UFS) with split data and control caches. The LFS SunOS client code fits easily into the SunOS kernel by sandwiching between the standard Virtual File System (VFS) and block I/O interfaces. Because the LFS server executes on a non-Unix processor, the normal procedure calls that occur between these kernel interfaces are transparently replaced with extremely fast message exchanges. These message operations are managed by a lightweight multiprocessor kernel that executes in each processor.

Revision 1.3.4; 910318.
Auspex Systems Inc.
2952 Bunker Hill Lane
Santa Clara, California 95054 USA
Phone: 408/492-0900 . Fax: 492-0909
Email: Info@Auspex.com or uunet!Auspex!Info
This paper appears in the *Proceedings of the IEEE Systems Design & Networks Conference `90*, Santa Clara, California, 8-10 May 1990. An early version of this paper appeared in the *Proceedings of the Sun User Group*, Anaheim, California, 6-8 December 1989.

TABLE OF CONTENTS

1 OVERVIEW

The latest generation of Unix workstations have I/O requirements that file servers are largely unable to satisfy. This network I/O performance gap--between client and server--has developed because dramatic jumps in microprocessor performance have not been matched by similar boosts in server I/O channel performance.

This paper explores the file system of a new NFS server--the Auspex NS 5000--that delivers significantly greater network I/O performance than conventional servers. The NS 5000 is a multiprocessor network server that distributes performance-limiting I/O functions to independent processors. This paper provides an overview of Auspex's hardware architecture and details one aspect of NS 5000 software architecture--the separation of the Unix file and storage processing components onto dedicated processors. In particular, it discusses how the NS 5000's Local File System was easily constructed using the standard VFS and NFS software interfaces of SunOS Unix.

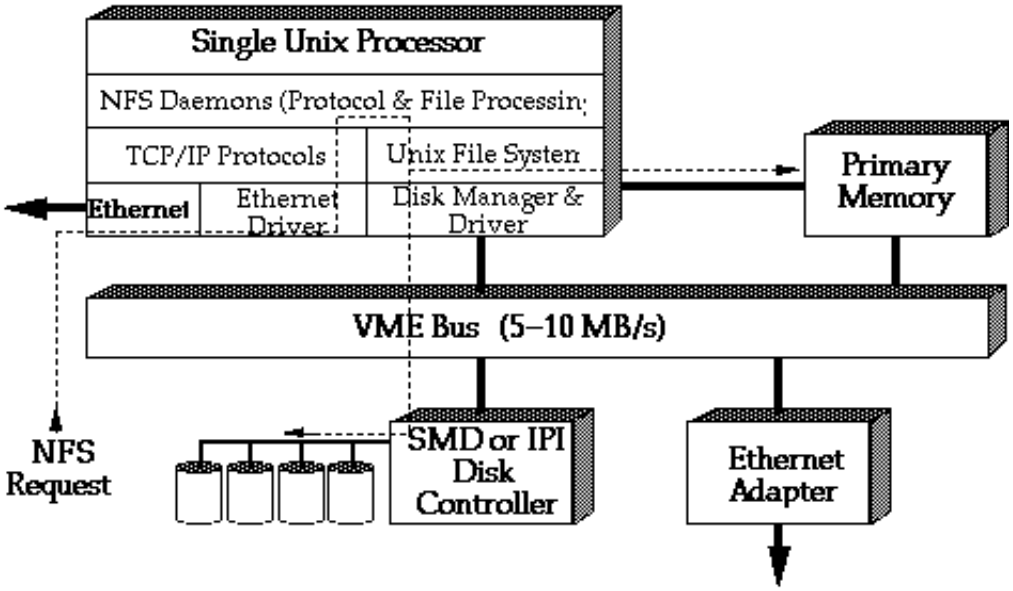

Figure 1: Conventional file server architecture. Using a single CPU, conventional servers share the execution of server functions with the Unix operating system. The dashed line indicates the basic flow of an NFS request through the processor. All network, file system, and most storage functions are performed by this single processor. Today, most of these processors--especially the latest RISC designs--are optimized for instruction fetch and execution, not I/O latency and bandwidth. This has exacerbated the I/O performance gap. In an attempt to address the storage speed issue, some vendors have recently added higher performance, microprocessor-based, caching disk controllers to their systems. But this is only an incremental step along the path to a server with balanced I/O throughput.

1.1 Performance Limitations of Conventional Server Architectures

In CASE and CAD environments, a frequent rule-of-thumb is that a single CISC- or even RISC-based rack-mounted file server can support from 5 to 10 diskless client workstations. This low client-to-server ratio is the result of a traditional approach to server design: repackaging a workstation in a rack. While conversion of a display-less workstation into a server may address disk capacity issues, but it does nothing to address fundamental I/O limitations. As a Network File System (NFS) server [Sandberg86], the one-time workstation must sustain 5-10 or more times the network, disk, backplane, and file system *throughput* than it was designed to support as a client. Adding larger disks, more network adaptors, or extra CPU memory does not resolve basic architectural I/O constraints. Adding CPU MIPS does *not* solve the problem. None of these steps increases overall network I/O throughput, as shown in figure 1.

Closing the NFS I/O performance gap requires a new, server-specific I/O architecture that can flexibly balance client data demand with server I/O throughput. The new I/O architecture must scale in power and capacity just as easily as client workstations scale in performance. The new architecture must focus on optimizing a Unix NFS network server's most common actions--NFS operations--just as RISC processors focus on optimizing a CPU's most common instructions.

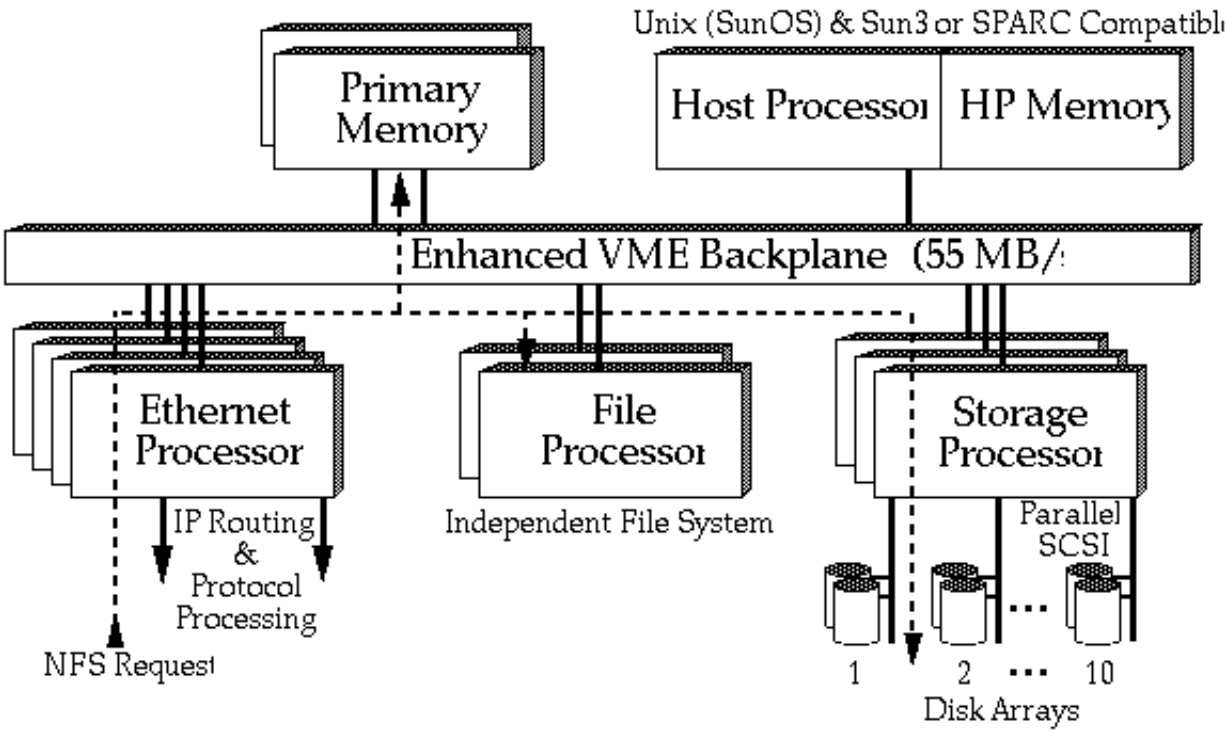

Figure 2: The NS 5000 functional multiprocessing I/O architecture. All network, file system, and storage processing is completely removed from the Unix host processor and performed instead by dedicated processors. Primary memory contains only file data and no instructions. The dashed line shows the data flow of an NFS read request through the system.

1.2 A High Performance Network Server Architecture

Auspex developed a network server, the NS 5000, with a *functional multiprocessing* (FMP) I/O architecture [Auspex89]. The NS 5000 emphasizes file--and not compute--service by maximizing utilization of the data paths between client workstations and the server's network interfaces, file system, and disk storage subsystem. As shown in figure 2, the NS 5000 achieves its performance advantage with highly intelligent processors that operate in parallel to optimize NFS throughput. The Unix operating system is completely eliminated from all network, file, and storage processing.

A key feature of the NS 5000 software architecture is that the Unix file system is totally separated--factored out--of the kernel and executes on its own independent file processor. Because optimizing NFS was our primary goal, the file processor presents a streamlined, tailored-for-NFS file system interface--called the *Local File System*, or LFS--as its primary software interface. LFS presents this interface to its two types of clients: the local Unix host processor and remote Unix workstations:

n The Unix host processor accesses the NS 5000 file system through an LFS interface layer of the standard virtual file system interface executing in the host processor.

n Remote Unix workstations send NFS requests to an Ethernet processor, which converts NFS request packets from the network into LFS message requests to the file processor.

Some background on the basic Unix file system will be useful before a discussion of LFS.

2 UNIX FILE SYSTEM BACKGROUND

In Unix before the advent of NFS, the Unix system call layer was directly connected to the underlying file system. These versions of

Unix include V7, BSD4.2, and System V R.2. The file system interface in these versions of Unix is known as the *Unix File System*, or UFS.

2.1 Typical UFS Control Flow

UFS maps the file abstraction to its representation on the disk using *inodes*, the kernel's internal representation of a file. The inode contains information such as the device (disk) number, the partition holding the file, and a list of the file's first *n* block numbers. Application calls through the system call layer translate a file descriptor (or path) to an inode pointer.

Figure 3 illustrates the UFS control flow of a disk I/O request from an application to the disk device driver in BSD 4.2. An application program making a system call, for example *write,* would call *ufs_write*, then the block I/O routine (*bwrite*), and finally the disk driver (*dkstrategy*) which would schedule the disk transfers.

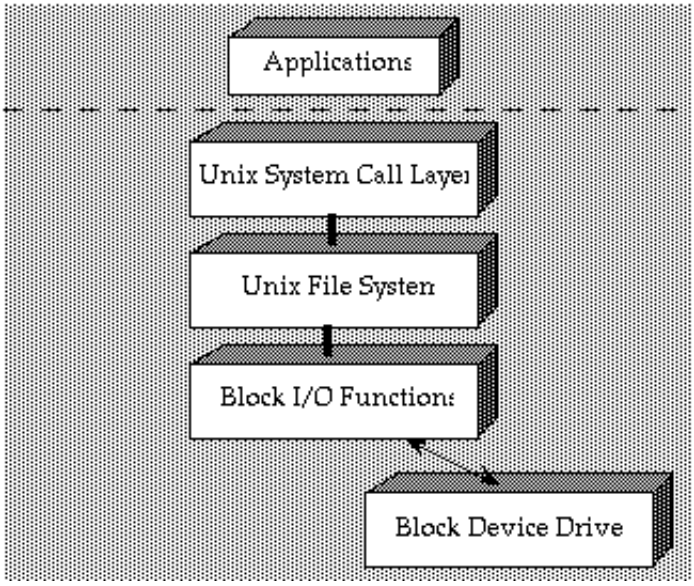

Figure 3: Unix file system processing before the NFS era. Application calls to the file system map directly to the disk block I/O functions.

2.2 The Virtual File System Interface

When Sun Microsystems introduced NFS in SunOS V2.0, it also introduced a new file system abstraction--called the *virtual file system* or VFS [Kleiman86]--so that NFS could compatibly and transparently coexist with UFS. VFS included two new fundamental data structures: the *vnode* and the *vnodeops* vector. The VFS interface has a level of indirection so a file vnode can refer to either a UFS file, to an NFS file, or to a file on some other type of file system. Each vnode refers to a vector of vnodeops--the vnode's supported operations--such as *create, read, write, getattr,* and *setattr*. Table 1 has the complete operations list. Each operation is performed with a call through a *V_OP* entry.

access	close	cmp	create	dump	fid
fsync	getattr	getpage	inactive	ioctl	link
lockctl	lookup	map	mkdir	putpage	rdwr
readdir	readlink	realvp	remove	rename	rmdir
select	setattr	open	symlink		

Table 1: A table of the *vnodeops* operations.

For example, the *rdwr* (read-write) system call with a vnode pointer *vp* basically contains the macro

VOP_RDWR(vp, uiop, rw, f, c)

which translates into

*(*vp->v_op->vn_rdwr)(vp, uiop, rw, f, c).*

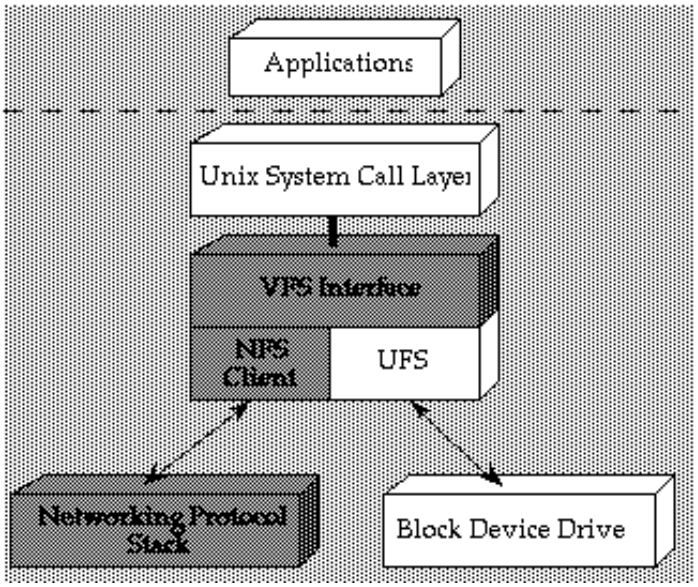

Figure 4: File system processing through the VFS interface. NFS and UFS are transparently supported through the VFS abstraction.

With VFS, file system processing has evolved as shown in figure 4. If an application makes a *write* system call to an open remote file--a file on an NFS server--the write routine is called with a vnode pointer which refers to an rnode. The VOP_RDWR macro will therefore call the *nfs_write* routine. This eventually initiates an *nfs_write* remote procedure call (RPC) on that file. The remote call is synchronous in that it blocks until the RPC returns, indicating that the data has been safely written by the NFS server.

3 THE LOCAL FILE SYSTEM INTERFACE

The Local File System (LFS) is implemented as a service running on an independent file processor. The LFS client code runs on both the Unix host processor as a VFS layer and on the Ethernet processor(s). Since these processors do not share instruction memory, the client and server portions of LFS communicate through message passing. Message operations are provided by a kernel, discussed next.

3.1 FMK--The Functional Multiprocessing Kernel

The NS 5000's functional multiprocessing architecture uses four types of loosely coupled processors, each with local instruction and data memory. All of these processors, including the Unix host processor, run a *functional multiprocessor kernel* (FMK) [Auspex90]. FMK is a small kernel for writing operating systems, and as such it provides only fundamental services such as lightweight processes, process scheduling, message passing, and memory allocation. A library of standard functions and processes provide services such as *sleep*, *wakeup*, error logging, and real time clocks, which are an integral part of the kernel itself in the Unix system.

The FMP architecture uses Unix to provide the large set of ancillary services that are not sufficiently frequent or time critical to justify implementation on dedicated processors. As a result, FMK was integrated into the Unix kernel, so Unix heavyweight processes appear to be compatible peers to the lightweight processes on non-host processors.

3.2 LFS Control Flow

Using FMK to direct LFS messages between processors, the previous file processing diagram evolves again as shown in figure 5.

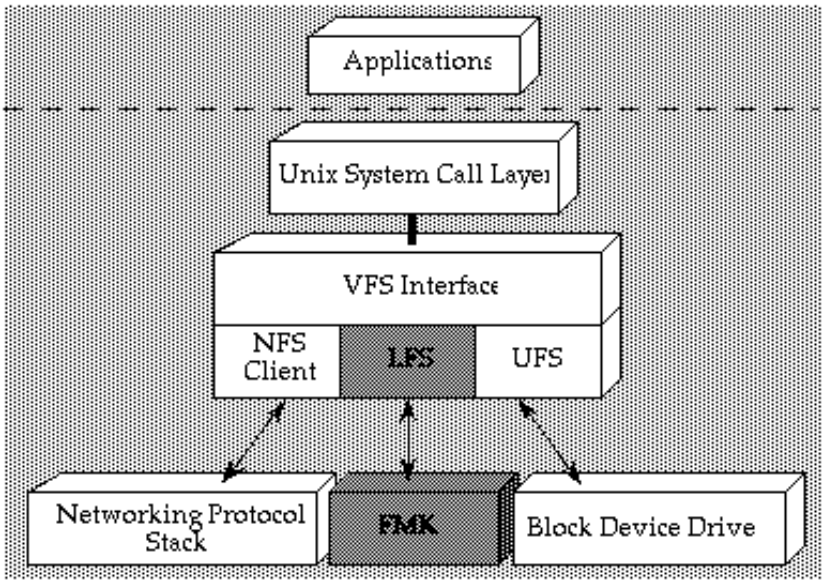

Figure 5: LFS control flow using VFS. LFS requests and responses are exchanged through message passing and not procedure calls. The FMK is a lightweight message-passing kernel used for interprocess communication between NS 5000 physical processors. From the FMK, LFS messages pass to the file processor (not shown).

For example, when an application makes a *write* to an open file which is an LFS file--i.e., a file mounted by a file processor--*write* is called with a vnode pointer which refers to an LFS-flavored rnode. The *VOP_RDWR* macro calls the *lfs_write* routine which eventually sends a message to the file processor, specifically to the *FP_Manager* process in charge of that file system. The message requests a write operation on that file. The call is still completely synchronous; however, the entire operation occurs more quickly than an NFS operation because the LFS message and the associated data are transfered across the NS 5000 backplane, not across an Ethernet.

3.3 LFS Operations

The Unix host processor--and other LFS clients--use FMK to send messages to the LFS server on the file processor. The LFS server on the file processor supports a request-reply message interface whose 24 message operations are based on NFS. LFS includes the 17 basic NFS messages, as well as *mount*, *umount*, and the VFS functions *fsync*, *access,* and *syncfs*. The LFS data transfer messages (*read*, *write*, *readdir*) are unlike NFS in that the file processor does not copy data. Instead these operations return the address of a data buffer in NS 5000 primary memory. When an LFS client finishes with the data, it sends a second LFS message back to the LFS server to release the buffer.

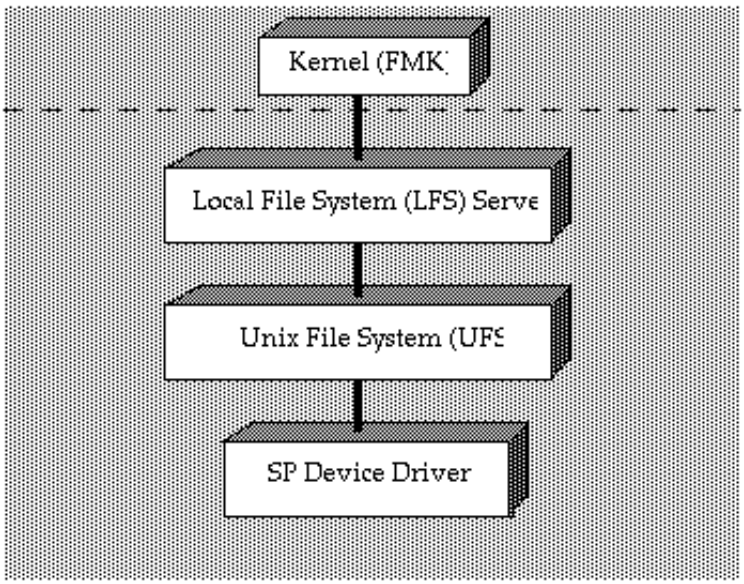

Figure 6: The architecture of the file processor software. FMK routes LFS requests to the LFS server. The LFS server makes UFS calls, and UFS calls *FP_strategy*.

4 SEPARATING THE FILE SYSTEM FROM UNIX

4.1 File Processor Software Organization

In separating LFS from Unix, we found that the existing VFS and block driver interfaces were natural places to partition file and disk processing into separate components that could run on specialized file and storage processors. As shown in figure 6, the file processor runs almost standard UFS code with veneers on the top and bottom to provide the Unix kernel services required by UFS. On the top, a layer of LFS server code cracks the LFS messages and makes the appropriate UFS subroutine calls as described in the previous section. On the bottom, the block I/O layer (*bread*, *bwrite*, etc.) was modified to communicate with the storage processor. All calls through the device driver switch were replaced with simple calls to *FP_strategy,* which sends FMK messages to the storage processor to initiate I/O.

4.2 Raw I/O and UFS Interfaces for Utilities

The NS 5000 also has a normal block device interface to the storage processor from the Unix host processor. Therefore, file systems can be mounted directly on /dev/ad*nn*, the special device name for the storage processor, using the normal UFS code in the host processor's kernel. Utilities that need to access the raw disk device work normally on the NS 5000, specifically *dkinfo*, *fsck*, *newfs, tunefs*, *format*, etc.

4.3 Overall Software Architecture

Figure 7 shows the NS 5000 software architecture. Note, in particular, how the VFS, block I/O, and UFS interfaces discussed above actually fit into the overall picture.

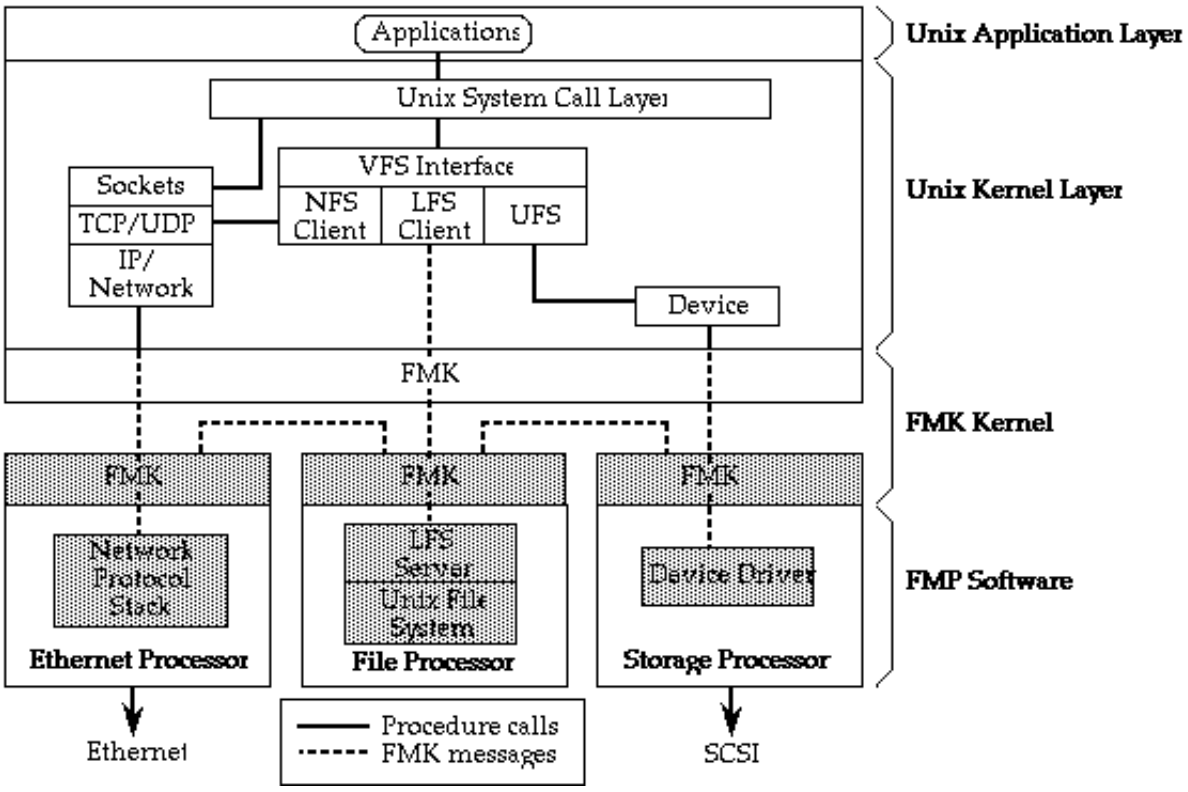

Figure 7: NS 5000 software structure. FMK executes on each hardware processor (large boxes). Solid connecting lines represent function calls. Dashed lines denote control paths using FMK messages. While the software structure is complex, the optimized NFS path is simple. The modules performing primary NFS processing are denoted *FMP Software* (shaded boxes).

4.4 Splitting the Buffer Cache

The major modification to UFS was splitting the buffer cache into two parts: a large *data cache* in primary memory and a smaller *control cache* in file processor private memory. The data cache contains all user data blocks for files and symlinks. The control cache contains all other disk data, including super-blocks, disk inodes, directory contents, and indirect blocks.

Putting file data into primary memory allows the storage processors, Ethernet processors, and the Unix host processor to access it without interfering with file processor private memory. Similarly, the file processor can access control structures without using backplane bandwidth. Thus, the split control and data caches allow each processor appropriate access with minimal memory contention.

4.5 Origin of LFS Components

The LFS server code was modeled after the NFS server code provided in the licensed NFS 4.0 VAX reference release. Like the NFS server code, LFS uses VFS to access the UFS layer; however, we could remove the VFS layer for efficiency. Handling multiple file system types can be more easily accomplished with a separate FMK server process for each type. The UFS code, from the bottom of the VFS layer down through the block-I/O layer, is also derived from the NFS reference release.

5 CONCLUSION

Separating the file system from the Unix operating system was a sound decision for increasing server I/O throughput. The VFS and NFS interfaces made the implementation fairly straightforward, given the existence of the FMK. File system performance improved significantly by executing it on a dedicated file processor with unhindered access to buffer cache. In addition, the file system runs unencumbered by normal Unix context switching and scheduling overhead. Finally, and most importantly, the file processor sits in the middle position between the network and storage processors, coordinating the NS 5000's highly streamlined datapath that services NFS requests.

6 REFERENCES

[Auspex89] Auspex Systems Inc.
An Overview of Functional Multiprocessing for Network Servers.
Technical Report 1, Fourth Edition, Auspex Systems Inc., February 1991.

[Auspex90] David Hitz, Guy Harris, James K Lau, and Allan M Schwartz.
Using Unix as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers.
In *Proceedings of the Winter 1990 USENIX Conference*,
Washington, DC, 23-26 January 1990.
Also Technical Report 5, Auspex Systems Inc., January 1990.

[Kleiman86] S.R. Kleiman.
Vnodes: An Architecture for Multiple File System Types in Sun Unix.
Proceedings of the Summer 1989 USENIX Conference, Atlanta, Georgia.

[Sandberg86] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon.
Design and Implementation of the Sun Network File System.
Proceedings of the Summer 1985 USENIX Conference, pp. 119-30, Portland, OR, June 1985.