Using Unix as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers

David Hitz
Guy Harris
James K. Lau
Allan M. Schwartz

January 1990

ABSTRACT

Auspex builds fast NFS file servers designed to satisfy the I/O demands of large networks and high-performance workstations. The architecture handles NFS operations quickly and efficiently by *completely eliminating* Unix from the normal path of NFS service. We designed a message passing kernel that allows a slightly modified Unix kernel to execute as a peer processor with Ethernet processors, filesystem processors, and disk storage processors. These non-Unix processors respond efficiently to NFS requests and perform IP packet routing. A separate host processor running SunOS4.0 provides full Unix compatibility by servicing less time critical and less frequent requests such as Yellow Pages. Our message passing kernel is small (15 kbytes of object) and fast (10,000 messages per second into a Motorola 68020) and provides source code debugging for all processors.

1. 1INTRODUCTION
2. 2MODEL FOR A FUNCTIONAL MULTIPROCESSING KERNEL
    1. 2.1Overview of Model
    2. 2.2Fundamental Primitives
    3. 2.3Omissions from FMK
3. 3THE FMK PROGRAMMING ENVIRONMENT
    1. 3.1Development Tools
    2. 3.2A Library of Standard Services
4. 4EXTENDING FMK TO COEXIST WITH UNIX
    1. 4.1Handling Unix Process Destruction
    2. 4.2FMK in the Bottom Half of the Unix Kernel
5. 5FMK IMPLEMENTATION
6. 6FMK IN THE NS 5000
7. 7CONCLUSION
8. 8REFERENCES

1 INTRODUCTION

Today's NFS servers are hard-pressed to meet the demands of large networks populated with new high-performance client workstations. As client-server computing enters its midlife, client workstations are severely constrained by server I/O limitations. This I/O performance gap has developed because dramatic jumps in microprocessor performance have not been matched by similar boosts to server I/O channel performance. Unix workstation vendors have traditionally designed servers by repackaging workstations in a rack, but adding larger disks, more network adaptors, or extra memory does not resolve basic architectural I/O constraints; neither does adding CPU MIPS.

At Auspex we have designed and built a high performance NFS file server called the NS5000 with a Functional Multiprocessing (FMP) architecture that distributes NFS protocol processing across several highly intelligent processors [Auspex89]. In a nutshell, Auspex's Functional Multiprocessing architecture removes Unix from the normal NFS processing path, using it instead to provide compatibility with standard Unix services such as Yellow Pages and system administration. It optimizes a file server's most common actions--NFS operations--just as RISC processors optimize a CPU's most common instructions.

Figure 1 compares NFS software processing on the NS ;5000 with that of a normal Unix server.

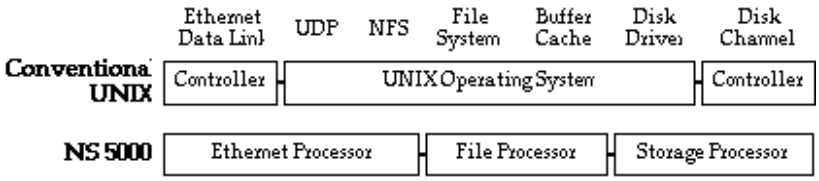


Figure 1: The distribution of NFS software functions using conventional and functional multiprocessing architectures.

In more detail, the processors perform the following functions:

One Unix host processor (HP) runs SunOS 4.0 to handle all non-NFS services. This provides complete compatibility with standard Unix NFS servers but is not part of the normal NFS path.

One or more Ethernet processors (EPs) receive and process Ethernet packets from the network. For NFS packets, IP, UDP, XDR, RPC and NFS layers are all handled locally, as is packet routing. EPs forward all unsupported protocols to Unix.

One or more file processors (FPs) manage local filesystems. FPs service file system requests from the Unix host processor as well as those from the EPs.

One or more storage processors (SPs) control SCSI disks and tapes. Each SP contains 10 parallel high-speed SCSI channels and provides driver-level services such as elevator sorting and retry.

The architecture is fast because the processors are designed specifically for NFS file service. It scales well because it supports multiple instances of each processor.

A key feature of the NS5000 is a simple, high-performance message-passing kernel that provides a common development base for all non-Unix processors. This Functional Multiprocessing Kernel (FMK) provides services such as lightweight process scheduling, message passing and memory allocation. Each processor executes an instance of FMK in its local memory and passes messages over the backplane. Some key features of FMK include:

Very small size--less than 15 kilobytes of object code.

Fast context switching--over 20 thousand / second on a 68020.

Fast message-passing between processors--10,000 / second into a 68020.

Full support for *dbx* source code debugging.

Making the NS5000 look like standard Unix requires the host processor to integrate completely with the other processors. With FMK this is possible because the host processor looks just like another FMK peer processor in the system. The host processor provides Unix compatibility services; other processors provide NFS services.

The remainder of this paper explores the design and implementation of FMK, the work required to integrate FMK and Unix, and a more detailed description of how FMK and Unix interact in the NS5000 architecture.

2 MODEL FOR A FUNCTIONAL MULTIPROCESSING KERNEL

Most of the code running on the NS5000 comes from Unix. The TCP/IP protocol stack and Fast File System come from BSD 4.3, and we license NFS from Sun. All of this code requires Unix kernel support, but running Unix on all of our processors--even a stripped down Unix--would be cumbersome and ill-suited to our needs. We wanted the smallest, fastest kernel that would satisfy our needs: a reduced primitive set kernel, if you will.

FMK is the result. FMK is a kernel for operating system development, not application development, and as such it provides a minimal set of fundamental services including lightweight process scheduling, message passing, memory allocation, simple timer services, and interrupt handling. It does not support memory management, process destruction or a robust user interface. The original specification provided just 16 primitives, and although it has now grown to thirty-something, we still consider it lean.

2.1 Overview of Model

FMK supports lightweight processes that communicate via synchronous (*i.e.* blocking) message-passing primitives. Each process is identified by an FMK process-id or PID and consists of little more than a stack, a thread of control, a scheduling priority, and a queue

of messages that have been sent to it. These processes are similar to, though simpler than, Unix processes running in kernel mode. As with Unix kernel processes, no preemption is allowed. A process continues to run until it explicitly gives up the CPU. Nothing in FMK precludes preemption; however, since most code running under FMK comes from the Unix kernel we would have had to modify that code or disable preemption anyway. It seemed more sensible to omit it.

Message passing works identically whether the destination process is on the same processor or a different one. This makes it easy to tune the system by moving a particular process from one processor to another. As an example of this flexibility, we developed the file processor code under FMK on Unix before moving it to its own processor.

Processes on the same processor share the same address space, while those on different processors do not. When a group of processes share data structures in their common address space, as opposed to communicating exclusively through messages, they may be moved to another processor only if they are kept together. Of course, processes that manage hardware such as Ethernet chips or SCSI channels must run on the processor with that hardware.

An FMK message consists of 128 bytes of data, the last few bytes of which are reserved for the kernel. The data always starts with a message type that specifies the operation requested of the receiving process. All messages have an associated C structure that defines their format. The FMK part of the message includes a pointer for queuing the message into linked lists, source and destination PIDs, and the like.

2.2 Fundamental Primitives

The following list summarizes the most frequently used FMK primitives. All primitives begin with "*k_*" to reduce naming conflicts. The "*k*" stands for kernel.

k_register(name) Assign the specified name to the current process.

k_resolve(name) Return the PID of the process with the specified name.

k_alloc_msg() Return a newly allocated message.

k_free_msg(msg) Free the specified message.

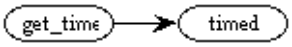
k_send(msg, pid) Send message to the specified process. Block till it returns.

k_receive() Receive message sent to this process. Block if none ready.

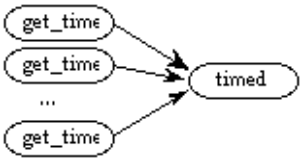
k_reply(msg) Return message to process that originally called *k_send*().

The code in figure 2 shows how these primitives can be used. The *get_time*() function sends a message to a TIMED process that handles simple timer services. The *timed*() function implements the TIMED process. Note that *timed*() is intended to run in Unix user space and uses standard Unix system calls to get and set the time.

We can represent this message passing relationship in a graph like this:



Boxes represent processes, and the arrow represents a message being sent. Since TIMED could service multiple clients, the picture really looks like this:



There is just one TIMED process so only one GET_TIME or SET_TIME message can be handled at a time. For many services, this is desirable. For instance, requests to a DMA service process *should* be queued for sequential processing because a DMA channel can handle only one request at a time. On the other hand, messages to a filesystem service process must not back up just because the current message is waiting for disk I/O.

```
struct time /* Ask the TIMED what time it is. */
get_time()
{
struct time time;
```

```
struct time_msg *msg;
K_PID pid;

msg = k_alloc_msg();
msg->type = GET_TIME;

pid = k_resolve("TIMED"); /* Find "timed" process. */
msg = k_send(msg, pid ); /* Send message, await reply. */

time = msg->time;
k_free_msg( msg );

return time;
}

timed() /* Implementation of TIMED process. */
{
struct time_msg *msg;

k_register("TIMED");
while (1) {
msg = k_receive();

switch (msg->type) {
case GET_TIME: gettimeofday(&msg->time, 0); break;
case SET_TIME: settimeofday(&msg->time, 0); break;
}

k_reply(msg);
}
}
```

Figure 2: An example of timer services implemented with FMK primitives.

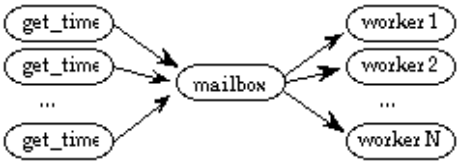
FMK provides mailboxes to allow more than one process to service messages sent to a particular PID. Sending a message to a mailbox is identical to sending a message to a process. Typically the creator of a mailbox also creates several worker processes to receive messages from it using

k_create() Create a new process.

The arguments to *k_create*() include the address of a function to run in the new context, the size of the stack required for that function, a scheduling priority, and one argument for the new process. These processes receive messages from the mailbox using

k_receive_mbox(id) Receive a message from a mailbox. Block if none ready.

If timed were reimplemented using mailboxes, the graph would become



Most services implemented in the NS5000 use this model.

The examples above give a flavor of development with FMK. FMK also includes primitives to check whether messages are available without blocking, primitives to wait for interrupts and to signal their occurrence, and primitives to adjust the scheduling behavior of processes. These are less frequently used, but are required to provide particular services or to improve performance in special cases. FMK has also incorporated the Unix kernel memory allocation functions from the BSD 4.3 Tahoe release with some modifications for supporting multiple memory types [McKusick88].

2.3 Omissions from FMK

Having reviewed what FMK offers, it makes sense to examine what it does not. We believe that FMK is more notable for features it omits than for those it includes. In particular, there is no mechanism for killing an FMK process; once a process is created, it lives forever. Processes cannot send messages asynchronously; they must wait for the reply. Processes can wait for only one event at a time; there is nothing like Unix's *select*(2). Finally, FMK has no memory management.

Each of these features would require extra processing in the low-level code that handles process scheduling and message delivery. FMK is fast in large part because of these simplifying assumptions.

In Unix, leaving out these features would be painful. Processes are expensive and context switching is slow. Every process is expected to perform a reasonable amount of work. On the other hand, when processes are cheap and context switching rapid, groups of processes can perform functions that would be handled by a single process in Unix.

These restrictions are not an exercise in masochism. Code based on a group of processes built from simple primitives can be simpler and more provably correct than code based on fewer processes with more complex primitives.

These principles are by no means new to FMK. Most of the fundamental ideas were originally developed in Thoth, from which the V kernel and Port are also derived [Cheriton79, Cheriton84]. Unlike FMK, which has no memory management, these kernels all support memory management models that allow processes to have private address spaces. They also allow process destruction. FMK has made some subtle but important changes in the relationship between processes and messages that have eased its integration with Unix, but perhaps more importantly, FMK has carried the idea of a very small and very simple set of primitives to an almost obsessive extreme.

3 THE FMK PROGRAMMING ENVIRONMENT

We believe that time spent building a high quality development environment and good debugging tools is more than repaid in reduced development and debugging time and improved product quality. Our work focussed on making important Unix tools like *dbx*(1) and *prof*(1) work with code running under FMK on the non-Unix processors and on developing new tools where useful. For FMK in Unix user processes, standard development tools work fine. For Unix kernel debugging *kadb*(8) is archaic, but we haven't had time to write *kdbx*.

To compile FMK code to run on an FMK processor, one simply links normally compiled code with the `fmk.o` file for the target processor (*i.e.* `fmk_ep.o`, `fmk_fp.o`, or `fmk_sp.o`). Code that does not depend on hardware specific to a processor may be linked with `fmk_Unix.o` and tested under Unix. A new *ax_startup*(8) utility in Unix ("*ax_*" for Auspex) downloads images to each processor and initiates communication between them.

3.1 Development Tools

Running *dbx* on a peer processor from Unix is also simple. The image is downloaded using *ax_dbx* instead of *ax_startup*. As with ordinary *dbx*, one starts the program by typing run and stops it with control-C. Single stepping, breakpoints, and tracing all work as expected. The only noticeable difference is that printouts from a processor come out on its console, not from *ax_dbx*.

Another program named *ax_util* provides a collection of services related to the various processors. It can collect core files (interpretable by *ax_dbx*) or profiling data (interpretable by *ax_prof*), and can also read and write a processor's local memory. The NS5000 is designed to collect a core file from each processor in the case of a panic, although we don't expect this ever to happen except in carefully controlled test situations.

An interactive shell process runs under FMK. It supports several commands such as one that shows all FMK processes and their states and another that enables verbose tracing from FMK. Custom commands can be added for individual processors. The Ethernet processor, for instance, supports a command that shows network MBUF statistics for each interface.

Finally, *ax_util* can provide a virtual console connection from Unix to any other processor. As an example of the power of this feature, one can *telnet* to an NS5000, establish a virtual connection to any processor, and issue any command its shell supports. Of course, one must be super-user to do this, and some commands may be dangerous to the health of the system, but for debugging this feature is wonderful.

3.2 A Library of Standard Services

Although FMK itself is very lean, we have developed a library of standard functions and processes that provide useful services.

An FMK process that registers the name AX_ERRD provides a link to the Unix *syslogd*(8) message logging service. From any

processor, one can send a message to AX_ERRD to have a message printed on the Unix console or appended to an administrative log file.

An AX_TIMED process running under Unix provides real time clock services to all processors, and can also be instructed to notify a processor when Unix's time gets changed.

The library also includes the simple shell, some standard functions from libc such as *bcopy*(3) and *printf*(3), and FMK implementations of Unix kernel functions such as *sleep*() and *wakeup*().

4 EXTENDING FMK TO COEXIST WITH UNIX

As an essential step in making our Functional Multiprocessing architecture look like standard Unix, FMK must connect Unix with the other processors. Unix must integrate with FMK as a compatible peer processor.

Minimizing modifications to SunOS is also important because Unix implementations are a moving target and we expect Sun and others to continue providing new and improved versions of Unix in the foreseeable future. We want to smoothly track these new releases.

To implement services like a device driver interface to the storage processor, FMK primitives are provided within the Unix kernel. On the other hand, we also wanted to support FMK primitives in Unix user level processes because it is so much faster and easier to debug code there than in the kernel.

Two differences between Unix and FMK make this integration difficult. First, Unix processes can die whereas FMK processes must live forever. Second, FMK provides quick context switching for its processes, so interrupts typically wake up a service process. Unix context switching is slower, so many device driver services are performed in the bottom half of the kernel at interrupt level where there is no process context and blocking is forbidden. It is difficult to provide FMK services in the bottom half because FMK requires processes to block in order to send and receive messages.

The general solution to these problems is to observe that the procedural interface to FMK under Unix need not look exactly like that on the other processors as long as messages to and from Unix look *just as if* they had been sent to standard FMK lightweight processes. On the other hand, we tried hard to minimize the differences between FMK on Unix and FMK on other processors to make it easier to move code between the two and to reduce general confusion.

4.1 Handling Unix Process Destruction

We solved the process destruction problem by waving our hands. We observed that while Unix process-ids change often, there are only a small fixed number of `user` and `proc` structures in the kernel. We reasoned that processes aren't really destroyed in Unix, their contents and process-ids just change.

While this observation may sound fatuous, in practice it allowed the system to work. When Unix boots, each Unix process slot gets a unique FMK PID, and it retains that FMK PID even as the Unix PID for the process slot changes.

This approach introduces two possibilities for error:

If a Unix process is killed during *k_send*(), the reply message returns to an empty process slot, or worse yet to a new process running in the same slot.

If a Unix process that has *k_register*()ed a named service dies, the new process occupying that slot will almost certainly not know how to handle arriving messages.

We solved the first problem by making *k_send*() uninterruptible. A process in *k_send*() cannot be killed until after it gets its reply. Most Unix processes that use FMK don't provide named services, and for them this solution is sufficient.

For processes providing named services, we made a simple rule: Don't die. To insure that such processes aren't killed we added a system call to Unix to declare a process immortal--like *init*(8). If that process dies, Unix dies. There are three such service processes in the NS5000.

4.2 FMK in the Bottom Half of the Unix Kernel

Providing FMK service in the bottom half of the kernel was harder. At this level Unix provides no process context and blocking is not allowed. Yet the FMK message passing model requires a process context because it expects message senders and receivers to block.

One solution would have been to add lightweight processes that could run in the bottom half of the Unix kernel. This didn't fit with

our goal of keeping Unix modifications simple and easy to port to new versions.

Instead we provided additional primitives to FMK that allow bottom half code to send messages without blocking and to receive messages at interrupt time. Although we dislike adding primitives to FMK, the requirements of interfacing with Unix overrode that concern in this case.

5 FMK IMPLEMENTATION

For the non-Unix processors, the implementation consists of two separate layers: a processor-independent FMK layer, and a processor-specific layer that handles hardware initialization and message passing between processors.

Although we have currently used the FMK layer only on 68K-based processors, it was written to be portable to other processors with the same byte ordering. The use of C structures to define message formats precludes processors with different byte orders. The cost of XDR-like format conversion for messages within our system would have been unacceptable.

The hardware layer consists of additional code unique to the various processors. It supports not only FMK, but also features like DMA channels and RS232 console ports. The entire FMK kernel, including both FMK and hardware layers, compiles to between 12 and 14 kilobytes of object code for all non-Unix processors.

Although the FMK primitives in Unix are almost identical to those in other processors, the implementation is separate. Under Unix FMK is implemented as a collection of functions that can be used anywhere in the top half of the kernel. We also added a device driver that provides access to the functions in user space. FMK user processes link with a special Auspex library to access these functions. This code should be easily portable to other BSD-derived Unix implementations.

Since the design goal of the NS5000 is to service many NFS clients, the ultimate performance measure for FMK is its ability to efficiently support large volume message traffic among peer processors. Between processes on a single processor, FMK can support up to 15 thousand k_send/k_reply pairs per second, and over the backplane FMK can deliver up to 10 thousand messages per second to one processor. The context switch time is under 50 microseconds. All of these numbers are for 20-MHz 68020 processors. Because the architecture was designed to remove Unix from the standard datapath, we did not optimize FMK as carefully for the host processor.

6 FMK IN THE NS 5000

Our goal in developing FMK was not to introduce yet another operating system into the world, but to build a tool for use in our file server. Figure 3 shows how FMK connects the various components of the NS5000. The very top of the diagram represents user space with the Unix kernel in the middle and the three special purpose processors below. Additional processors can be added, but they have been omitted for simplicity.

In keeping with our strategy of modifying Unix as little as possible, we put FMK interfaces at standard cleavage points: Unix speaks to the storage processor through a device driver, to the file processor through a new VFS file system type, and to the Ethernet processor through the standard network "if" driver.

The interface to the storage processor--implemented as a simple Unix device driver--is the cleanest. The device driver for /dev/ad* (Auspex disk) simply converts incoming requests into FMK messages to the SP using the FMK primitives provided in the kernel. Since disk interactions are always initiated by top half Unix routines, the extended FMK primitives for the bottom half of the kernel are never needed. This driver allows utilities like *fsck*(8), *newfs*(8), and *format*(8) which read disks directly to work unmodified.

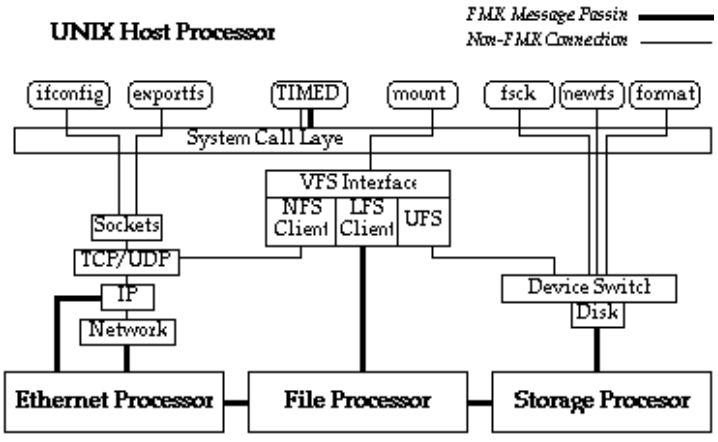


Figure 2: NS5000 software architecture using FMK for interprocess communication.

The interface to the file processor is more complicated. To give Unix access to filesystems that are mounted on a file processor, we created an entirely new filesystem type using Sun's VFS (Virtual File System) architecture [Kleiman86]. We call this new file system LFS, for Local File System [Schwartz89]. For each request through the VFS interface, it generates one or more messages to the file processor. LFS looks much like NFS, but with modifications to make operation over a local bus more efficient, and with additions to support mounting, exporting and quota control.

The interface to the Ethernet processor is the most complex. Unlike the SP or FP, the EPs must communicate with the bottom half of Unix when packets using non-EP supported protocols arrive. An EP must also send messages to Unix when IP addresses not in its route cache are encountered and to resolve keys for secure RPC. We made other modifications where Unix was unable to handle the eight separate Ethernets.

Finally, there are many Unix user level programs that use FMK to communicate with the other processors. The FMK service programs that provide timer and error logging services have already been mentioned. During development we also used FMK programs to query or control the Auspex processors. For instance, one program queries FMK to gather statistics such as the number of messages sent and the load average of that processor's CPU. These tools are not part of the standard Unix interface because they refer to features not available in standard Unix, but they are useful in developing and tuning the system.

7 CONCLUSION

Auspex's FMP architecture achieved its principal goal of providing fast NFS file service by removing Unix from time-critical operations. Moreover, FMK provided a common and consistent platform for system development outside of Unix. Finally, FMK provided a standard interface to Unix for those nonstreamlined services that Unix exports to the clients.

Beyond these primary achievements, there have been some surprising additional FMK benefits. The designers of most disk and Ethernet controllers cannot easily add debugging messages that print directly on the Unix console. Neither can they use *dbx* to debug their firmware on a running system. Most of us at Auspex believe that this is the best kernel development environment we have used. It is certainly much easier than debugging the Unix kernel with *kadb*, for instance. Not all of these features were part of our original design requirements, but our simple and consistent development environment made them possible.

Our approach of using standard Unix to provide compatibility allowed us to focus development effort on the I/O subsystem, where the greatest improvements could be made. Our use of FMK and its environment allowed us to develop this code quickly and efficiently.

8 REFERENCES

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

[Cheriton79] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager.
Thoth, a portable real-time operating system.
Communications of the ACM 22(2):105-115, February, 1979.

[Cheriton84] David R. Cheriton.
The V Kernel: A software base for distributed systems.
IEEE Software 1(2):19-42, April, 1984.

[Kleiman86] S.R. Kleiman.
Vnodes: An Architecture for Multiple File System Types in Sun Unix.
Proceedings of the Summer 1986 USENIX Conference, Atlanta, Georgia.

[McKusick88] Marshall Kirk McKusick and Michael J. Karels
Design of a General Purpose Memory Allocator for the 4.3BSD Unix Kernel.
Proceedings of the Summer 1988 USENIX Conference, San Francisco, CA.

[Row84] John Row and David Daugherty.
Operating System Extensions Link Disparate Systems.
Computer Design, July 1984.

[Row85] John Row.

Lan Software Links Diverse Machines, OSs.
Mini Micro Systems, September 1985.

[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.

[Schwartz89] Allan M. Schwartz, David Hitz, and William M. Pitts.
LFS--A Local File System for Multiprocessor NFS Network Servers.
Proceedings of the Sun User Group, Anaheim, CA, 6-8 December 1989.
Also Technical Report 4, Auspex Systems Inc., December 1989.