TECHNICAL REPORT 98-01

Operating System Support for Easy Development of Distributed File Systems

Kenichi Kourai, Shigeru Chiba, and Takashi Masuda

August 6, 1998

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113–0033 Japan

TITLE
Operating System Support for Easy Development of Distributed File Systems

AUTHORS
Kenichi Kourai, Shigeru Chiba, and Takashi Masuda

KEY WORDS AND PHRASES
multi-level protection, fail-safe mechanism, distributed file system, operating system

ABSTRACT
A number of new distributed file systems have been developed, but the development of such file systems is not a simple task because it requires the operating system kernel to be modified. We have therefore developed an operating system in which a new file system is implemented as an extension module separated from the kernel. The operating system makes the file system easy to debug by protecting the kernel from errors of the file system, and it differs from similar operating systems in that it avoids unnecessary performance degradation by enabling the protection level to be changed without modifying the source code. We have implemented this operating system on the basis of NetBSD 1.2 and confirmed that the file system can run more efficiently when the protection level is decreased.

REPORT DATE	**WRITTEN LANGUAGE**
August 6, 1998	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
9	13

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
A short version of this paper will appear in Proceedings of the 1998 Parallel and Distributed Computing and Systems.

DISTRIBUTION STATEMENT
First issue 35 copies. This technical report is available via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113–0033, Japan

Operating System Support for Easy Development of Distributed File Systems

Kenichi Kourai, Shigeru Chiba, and Takashi Masuda

August 6, 1998

1 Introduction

Distributed file systems are not easy to develop because they are a part of an operating system and are implemented in the kernel. The development of such a file system is made easier by first implementing it as a user-level library that emulates the system calls for file access. This library is linked to an application program so that the program can use the new file system. The distributed file systems are eventually re-implemented in the kernel for the efficiency. The developers must debug both the library for the emulation and the file system embedded in the kernel, and this debugging makes the development laborious.

We have therefore developed an operating system that makes distributed file systems easier to develop. This operating system allows the users to develop a file system as an extension module. Because the extension module is installed in the operating system on demand, the users can debug their file systems without continually rebooting the computer. The operating system also uses a new fail-safe mechanism, called multi-level protection, that allows the users to install an extension module at various protection levels without modifying the source code. The users can therefore avoid performance penalties by running the released module at a lower protection level than that used for debugging the file system.

We have implemented this operating system on the basis of NetBSD 1.2. The multi-level protection has been implemented as a set of protection managers, each of which provides a different level of protection. The users can easily change the protection level of the extension module simply by choosing one of these protection managers. Because the protection manager provides APIs for the file system module, the users do not need to modify the source code of the module when changing the protection level.

We experimented to make sure of the usefulness of the multi-level protection and confirmed (1)that the performance of a file system module is improved when the protection level is decreased, (2)that the overhead at the maximal protection level is acceptable, and (3)that the overhead at the minimal protection level is almost negligible compared with those of the file system directly implemented in the kernel by hand.

2 Multi-level protection

In this section, we first describe problems of developing distributed file systems and then explain an operating system that we have developed to address the problems.

2.1 Developing distributed file systems

A number of distributed file systems such as NFS [7], AFS [8], and Coda [9] have already been developed, and most of them are embedded in the monolithic kernel like UNIX systems. This makes it difficult to develop distributed file systems not only because it takes much time to compile and link a developing file system with the huge kernel and the computer must thereafter be rebooted, but also because it is difficult to debug the file system. This is because the developers cannot use tools like a symbolic debugger that help identify the cause of errors and because they must reboot the computer every time an error occurs.

Extensible operating systems such as SPIN [2] and VINO [10] alleviate these problems in part by allowing the users to add a new distributed file system easily. Because the users can implement a file

system as an extension module and install it in the operating system on demand, they can try using their debugged file system without rebooting the computer. Another way the extensible operating systems can help is by using a fail-safe mechanism to protect the kernel from errors due to the extension modules. If the fail-safe mechanism detects an error of the module, it safely detaches the erroneous module from the operating system. Because it is not necessary to reboot the computer at this time and because the users can identify the cause of the error, the fail-safe mechanism makes it easy to debug the file system modules.

A problem is that no fixed level of fail-safety is appropriate in all phases of the development of distributed file systems. If the fail-safe mechanism provides complete fail-safety, distributed file systems can be debugged safely but the finished file systems are inefficient. On the other hand, if no fail-safety is provided, the operating system is not useful for debugging distributed file systems.

Many distributed file systems are therefore first implemented as a user-level library that emulates the system calls for file access and are later re-implemented in the kernel. The emulation makes it easy for the developers to implement and debug the file system because they need only modify the user-level library. After the prototype of file system developed using the emulation is implemented, they evaluate the file system experimentally. Considering the results of this evaluation, they can easily change the policy of the distributed file system before re-implementing the file system in the kernel. This way of development, however, makes the burden on the developers heavy. Because of the differences in data structure, APIs, and timing, they must debug both the library for the emulation and the file system embedded in the kernel.

2.2 Multi-level protection

On our operating system, a file system is developed as an extension module to the kernel so that it can be installed without rebooting the system. To make the file system easy to debug, our operating system protects the kernel from an erroneous extension module. To avoid the problems of other extensible operating systems, we have developed a new fail-safe mechanism, called multi-level protection, that allows the users to change the protection level of the extension modules according to the stability of the file system under development.

The multi-level protection enables the users to change the protection level without modifying the source code of the extension modules. They need only select the appropriate protection level: if they select the highest protection level, all errors are detected and recovered; if they select lower one, some errors are neither detected nor recovered. To change the protection level without modifying no source code, the multi-level protection provides APIs for the extension modules and hides the differences between the implementations of the protection.

This reduces the difficulty of debugging a new file system. The developers can, for example, implement a prototype of a distributed file system almost as easily as the library for the emulation. Because the multi-level protection fully protects the file system implemented as an extension module in this phase of debugging, the developers can obtain accurate error information and therefore easily identify the causes of the errors and fix them. And an extension module in which an error has been detected is safely detached so that it does not compromise the rest of the operating system.

After the prototype is implemented and experimentally evaluated, the multi-level protection allows the developers to decrease the protection level without modifying the source code. Because the performance is thereby improved, it becomes easier to run the file system longer in order to find more errors. It is generally sufficient to detect only errors depending on timing, like deadlocks, because in this phase of testing the distributed file systems seem to be rather stable.

Finally, the released version of the distributed file system developed with the multi-level protection is embedded into the kernel without modifying the source code. It can run almost as efficiently as one directly implemented in the kernel by hand without any protection.

3 Implementation

We have implemented our operating system on the basis of NetBSD 1.2. In this section, we first mention how the users change the protection level of the extension modules. Next, we explain the protection

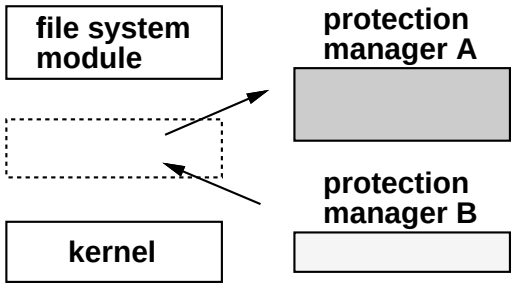

Figure 1: Exchange of the protection managers.

manager that has an important role on changing the protection level and describe the techniques of protection used in it.

3.1 Changing protection level

As shown in Figure 1, our system allows the users to change the protection level of an extension module by exchanging protection managers. The protection manager is inserted between the kernel and the module and implemented as a library linked with the module. In our system there are many protection managers, each of which can detect or recover from a different kind of error. The users can change the protection level by selecting a different protection manager.

To change the protection level of the extension module, the users relink the module with the protection manager which provides a desirable protection level. If the protection managers use macro or inline expansion for the purpose of performance improvement or for some other purposes, they need to recompile the module. After relinking, they restart the module in order to make the new protection level available. In the current implementation, if the address space where the module is located changes from the user space to the kernel space, the computer needs to be restarted.

3.2 Protection manager

3.2.1 API

To change the protection level without modifying the source code of the extension modules, the protection manager provides APIs, to which all the modules must conform. These APIs enable the modules to deal with a kernel data structure of low abstraction as if it were one of high abstraction. There is one API for manipulating the kernel data and another for registering callback functions. Although forcing all the modules to conform to these APIs restricts the programming of the extension modules, the advantages of not requiring source-code modification outweigh the disadvantages of this restriction.

Manipulating kernel data A part of the kernel data structure, such as `vnode` and `buf`, are hidden and direct access to them is not allowed. The protection manager therefore provides an API enabling the extension modules to access these hidden parts. This API hides complex operations and dangerous pointer manipulation from the developers of the modules. The protection manager also provides an API with the functionality equivalent to one provided in the kernel. Examples of these APIs are listed in Table 1.

Registering callback functions The protection manager provides an API to register callback functions invoked by upcalls from the kernel. These upcalls are made at the VFS [3] level of file systems and are not made to the extension modules directly but through the protection manager. After receiving an upcall, the protection manager invokes the corresponding callback function registered. At this time the protection manager transforms the data structure passed as the arguments to a more abstract structure. Examples of the upcalls are listed in Table 2.

API	Functionality
API to manipulate hidden parts	
`GetNextBlk`	Take the next element of `buf` chain
`MCbuild`	Pack raw data to `mbuf` chain
API provided in the kernel	
`Vref`	Increment a reference count of `vnode`
`Bread`	Read data from a file to `buf`

Table 1: Examples of APIs provided by the protection manager to manipulate kernel data.

Type	Request
`VOP_READ`	Read data from a file
`VOP_LOCK`	Lock a file

Table 2: Examples of upcalls from the kernel to file system modules.

3.2.2 Targeting errors

The protection manager treats an illegal memory access as an error to be detected and recovered from. There are two kinds of illegal memory access: a hardware-trapped memory access and semantically illegal data modification. A hardware-trapped memory access is detected easily. A segmentation fault, for example, happens when the extension modules access different address spaces; an alignment fault happens when the modules access with a bad alignment.

The protection manager can detect semantically illegal data modification as an error only if, by illegal modification, the value of data exceeds the range that the data can take. Data whose reference count becomes negative, for example, is detected as an error because the reference count must be positive or zero, and a pointer that points to an out-of-range address is detected as an error.

The protection manager also treats a deadlock as an error to be detected and recovered from. For example, a deadlock is detected if, when two threads in a file system lock different files, each thread tries to lock the file locked by the other.

3.3 Techniques of protection

Our system uses some techniques of protection in order to detect and recover from an illegal memory access and a deadlock. Various combinations of these techniques enable the fail-safe mechanism of various protection levels.

3.3.1 Detection of an illegal memory access

Hardware-trapped memory access Our system exploits switching address spaces in order to detect an illegal memory access. If an extension module is located in an address space different from the kernel address space, any illegal memory accesses to the kernel memory by the module are disabled. It is very easy to detect this kind of illegal memory access because it is necessarily trapped by hardware and notified to the operating system. The overheads for enabling this protection, however, are rather large due to increasing the number of context switches and the amount of data copies between address spaces. To decrease these overheads, our system also allows the users to locate the module in the kernel address space.

When the extension module is located in an address space different from the kernel address space, the module and the kernel use shared memory to communicate with each other. Such shared memory, however, violates the protection of the kernel by compromising the isolation of address space. To prevent the module from illegally accessing the kernel memory, our system protects the shared memory by using

virtual memory system, using system calls such as `mmap` and `mprotect`, except when the code of the protection manager is executed.

Our system allows the users to select the way where the shared memory is protected and to change the protection level of the extension module. If the users want to prevent data in the shared memory from being destroyed and to detect an illegal memory read, the protection manager unmaps the shared memory so that the module cannot access it directly. If they want only to keep the data from being destroyed, the protection manager changes the protection of the shared memory to read-only. This way keeps the overheads lower than the way of unmapping the shared memory although it does not necessarily enable the protection manager to detect an illegal memory read earlier and more accurately. If the users do not need to detect illegal memory accesses to the shared memory, the protection manager does not protect the shared memory and there is thus no overhead for the protection.

A malicious extension module can destroy the data in the shared memory intentionally because the protection manager is located in the same address space with the module and because the module can invalidate the protection of the shared memory made by the protection manager. However, because we assume that any malicious extension modules are not installed, this cannot happen.

Semantically illegal data modification Our system replicates the kernel data in order to detect semantically illegal data modification. The extension modules freely access the kernel data replicated by the protection manager, and the protection manager writes it back to the kernel periodically. When writing it back, the protection manager checks the data by examining various properties of the data structure; for example, that the reference count of the data is positive or zero, that the upper and lower bounds of buffer size are fixed, and that the pointers to a certain kind of data point to the address of shared memory.

Our system allows the users to change the protection level by selecting what types of data are checked and how they check the data. For example, if the protection manager stops tracing pointers, the overhead is reduced because it does not need to check any loops caused by pointers. It is possible, however, that such errors are detected lately or cannot be detected.

3.3.2 Recovery from an illegal memory access

When an error due to an extension module, such as an illegal memory access, is detected, the kernel data modified by the module must be restored so that the operating system is kept stable. To restore the modified data, our system uses a log in which manipulation modifying the kernel data is recorded. On recovering, our system checks the log and executes the manipulation reverse to that recorded in the log. For example, if a file-lock manipulation is recorded in the log, our system executes an unlock manipulation.

Our system allows the users to adjust the overheads for recording the log and restoring the modified data by selecting what manipulations are recorded and what data is restored. If they want to remove all influences that the extension module has had on the kernel data, our system records all manipulations in the log and restores all the data modified by the manipulations. If they do not need to record all manipulations, our system reduces the overhead by recording only some of them. For example, in many cases a little memory leak is negligible even if it is not restored.

3.3.3 Detection of a deadlock

Our system periodically checks whether the system falls into a deadlock state. When locking, unlocking, and waiting for resources like `vnode`, the extension modules notify the system and the system records this information. To check for a deadlock, our system creates a wait-for-graph based on the information and checks for loops.

Our system allows the users to select the interval between checks for a deadlock and thereby change the protection level. If the users want to detect a deadlock earlier and shorten the time it takes to stop the services of the deadlocking modules, our system makes this interval short. This degrades the performance of the whole system. If they want to decrease the overhead, our system makes the interval long. But then it takes more time to detect a deadlock, and the services of the deadlocking modules become unavailable for a longer time.

Protection technique	1	2	3	4	5
Shared memory protection	√*	√**			
Kernel data replication	√	√	√		
Address space switch	√	√	√	√	
Log record	√	√	√	√	
Deadlock check	√	√	√	√	

Table 3: Five characteristic combinations of protection techniques. *Unmap shared memory. **Change the protection of shared memory to read-only.

Error	1	2	3	4	5
Illegal shared memory read	√				
Illegal shared memory write	√	√			
Semantically illegal data modification	√	√	√		
Illegal kernel data access	√	√	√	√	

Table 4: Detectable errors on memory protection at each protection level.

3.3.4 Recovery from a deadlock

Because the occurrence of deadlocking depends on timing, our system resolves the deadlock, and in many cases the deadlocking modules can continue to run. Our system resolves the deadlock by destroying a loop in the wait-for-graph. Our system first finds out where the loop is and then temporarily releases one of the locks in the loop. Although our system should release only the lock that causes the deadlock, because finding that lock is difficult, our system releases one of locks in the loop randomly and stops the module whose lock is released temporarily until it can obtain the lock again.

4 Experiments

We experimented to make sure of the usefulness of the multi-level protection. The purposes of these experiments were (1)to make sure that the execution performance is improved when the users change the protection level and degrade the level of fail-safety, (2)to measure the overhead of the maximal protection level, and (3)to measure the overheads needed to make the APIs that all protection managers provide common.

Because it would have been too difficult to experiment with all combinations of the protection techniques our system provides, we selected five characteristic combinations, listed in Table 3, and experimented with them. These combinations can detect the errors on memory protection listed in Table 4. We call the combination yielding the highest protection level the first level and call the combination yielding the lowest level the fifth level.

We developed a file system module for these experiments: Simple Network File System (SNFS). We used a SPARCstation 5 (MicroSPARC2/85MHz) for the SNFS client and a PC (Cyrix 6x86/133MHz) for the SNFS server. Each operating system was the one we developed. The client and server were connected with a 10Mbps network.

4.1 The overhead of protection

To obtain the overhead of each of the protection techniques used for the fail-safe mechanism in our system, we first measured the time needed to copy a file. The size of the copied file was 64KB and the block size for each `read` and `write` system call was 8KB. The results of this experiment, shown in Figure 2, mean that the performance of SNFS is improved when the protection level is made lower. The first level takes 2.2 times as much time as the fifth level, and the performance is rather degraded. It is acceptable, however, for debugging the SNFS module.

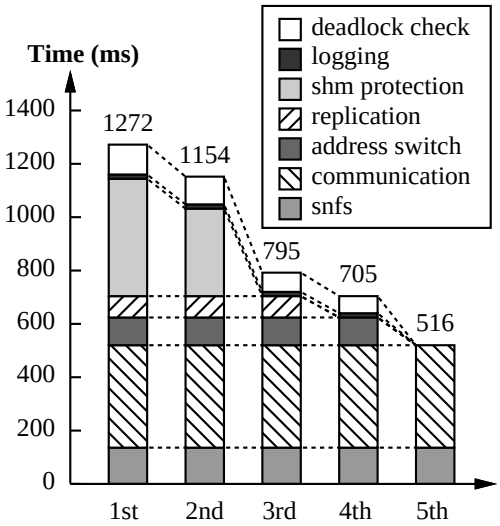

Figure 2: The time needed to copy a 64KB file in SNFS.

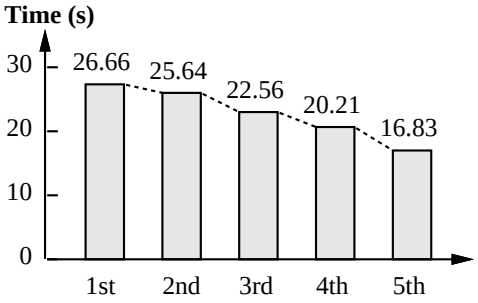

Figure 3: The time needed to compile a `ps` program in SNFS.

We then measured the time to compile a `ps` program of NetBSD 1.2. This program consisted of five source files and two header files, and had 2,000 lines. We compiled this program with `gcc`, and Figure 3 shows the results. In practical circumstances like those in this experiment, the first level takes 1.6 times as much time as the fifth level, and the performance is good enough even for normal use.

4.2 The overhead of API

We compared SNFS of the fifth level, which was embedded in the kernel, with SNFS directly implemented in the kernel by hand in order to obtain the overhead for making the APIs that all protection managers provide common. Figure 4 shows the result obtained with a 64KB file copy (the block size is 8KB).

This overhead was about 0.1%, and we think that the causes of this overhead are excess function calls, copies for transforming data structure, the increase of cache miss due to increasing the object size, and so on. This result means that the performance of a SNFS module implemented using multi-level protection is almost as good as that of SNFS directly implemented in the kernel by hand if the protection level of the module is the lowest and the module is embedded in the kernel.

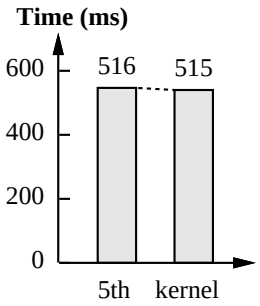

Figure 4: The overhead of API in SNFS.

5 Related work

Several UNIX systems like NetBSD allow the users to dynamically link extension modules called loadable kernel modules to the kernel. A loadable kernel module is implemented as a part of the kernel and runs very efficiently, but the errors due to the module can make the whole operating system crash because fail-safety is not considered.

The microkernel operating systems such as Mach [1] and Amoeba [4] provide complete fail-safety. Since an extension module is implemented by a user process, the errors due to the module are not propagated to the rest of the operating system. The erroneous module is simply terminated by the kernel. However, the performance is sacrificed because the overheads of IPC and context switches are large.

An extensible operating system SPIN [2] allows the users to download the extension modules into the kernel and provides fail-safety by using language supports. The extension module is written in the type-safe language Modula-3 [5] so that it does not cause memory access violation. SPIN does not consider other errors, however, and the type-safe language restricts the programming of the extension modules.

VINO [10, 11] uses Software Fault Isolation [13, 12] to protect the kernel from problems due to downloaded extension modules. It also limits the maximum amount of resources that the extension module can use at any given time and automatically releases the resources if a certain timeout expires. To recover from errors, VINO provides a kernel transaction system. VINO thus provides a relatively light-weight and sufficient fail-safety, but VINO always entails certain fixed overheads.

Chorus [6], on the other hand, allows the users to select whether or not the extension modules use the fail-safety feature. An extension module is first developed as a user process and protected by complete fail-safety. After debugging this module, the users install it in the kernel and run it without fail-safety. This approach is similar to ours, but the extension modules in Chorus continue to have overheads for complete fail-safety until the developers finish debugging the modules and the errors are exterminated.

6 Conclusion

This paper has described an operating system that makes distributed file systems easier to develop. In this operating system, a new file system is implemented as an extension module separated from the kernel. To make the file system easy to debug, the operating system kernel is protected from errors of the file system. Unnecessary performance degradation can be avoided by using a new fail-safe mechanism, called multi-level protection, to change the protection level. We have implemented this operating system and developed a distributed file system on it. We experimented to make sure of the usefulness of the multi-level protection and confirmed that the file system can run more efficiently when the protection level is decreased.

This multi-level protection is so far implemented only for file systems, but distributed file systems often require the customization of network protocols. We will therefore apply the multi-level protection to network system in the future.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX 1986 Summer Conference*, pages 93–112, 1986.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, S. Chambers, and C. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings 15th ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.

[3] M. K. McKusick. The Virtual Filesystem Interface in 4.4BSD. *Computing Systems*, 8(1):3–25, 1995.

[4] S. J. Mullender, C. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Stavern. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[5] G. Nelson. *System Programming with Modula-3.* Prentice Hall, 1991.

[6] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Lé oñard, and W. Neuhauser. CHORUS Distributed Operating System. *Computing Systems*, 1(4):305–370, 1988.

[7] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX 1985 Summer Conference*, pages 119–130, 1985.

[8] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, 1985.

[9] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[10] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. An Introduction to the Architecture of the VINO Kernel. Technical Report TR–34–94, Harvard University Computer Science, 1994.

[11] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, 1996.

[12] C. Small. MiSFIT: A Minimal i386 Software Fault Isolation Tool. Technical Report TR–07–96, Harvard University Computer Science, 1996.

[13] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, 1993.