Extending the Operating System at the User Level: the Ufo Global File System

Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheima
Department of Computer Science
University of California, Santa Barbara

Extending the Operating System at the User Level: the Ufo Global File System *

Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{berto,ibel,schauser,chriss}@cs.ucsb.edu
http://www.cs.ucsb.edu/research/ufo

Abstract

In this paper we show how to extend the functionality of standard operating systems completely at the user level. Our approach works by intercepting selected system calls at the user level, using tracing facilities such as the /proc file system provided by many Unix operating systems. The behavior of some intercepted system calls is then modified to implement new functionality. This approach does not require any re-linking or recompilation of existing applications. In fact, the extensions can even be dynamically "installed" into already running processes. The extensions work completely at the user level and install without system administrator assistance.

We used this approach to implement a global file system, called Ufo, which allows users to treat remote files exactly as if they were local. Currently, Ufo supports file access through the FTP and HTTP protocols and allows new protocols to be plugged in. While several other projects have implemented global file system abstractions, they all require either changes to the operating system or modifications to standard libraries. The paper gives a detailed performance analysis of our approach to extending the OS and establishes that Ufo introduces acceptable overhead for common applications even though intercepting system calls incurs a high cost.

Keywords: operating systems, user-level extensions, /proc file system, global file system, global name space, file caching

* This work was supported by the National Science Foundation under NSF CAREER Award CCR-9502661 and NSF Postdoctoral Award ASC-9504291. Computational resources were provided by the NSF Instrumentation Grant CDA-9529418 and Sun Microsystems. The software is available on-line under http://www.cs.ucsb.edu/research/ufo

1 Introduction

Computer users have always had the desire to extend operating systems functionality to support new protocols or meet new usage patterns. In this paper we show how to extend a standard Unix operating system (Solaris) completely at the user level. Our approach — which is similar to interposition agents [Jon93] — uses tracing facilities to intercept selected system calls at the user level. The behavior of intercepted system calls is then modified to implement new functionality. We use this to implement Ufo,[1] a global file system, which supports file access through the FTP and HTTP protocols and can easily be expanded to support new protocols.

While this paper focuses on extending the file system services, our approach provides a general way of expanding operating system functionality without any kernel changes or library modifications. Our extensions, which are not just limited to Solaris, do not require any re-linking or re-compilation of existing applications. In fact, they can even be dynamically "installed" into already running processes. Extensions can be added on a per-user basis, i.e., extensions for one user do not affect other users. Actually, even a single user could run different jobs with different extensions without interference.

An important advantage of this method is that developing the OS extensions can be done entirely at the user level and without access to OS source code. This makes our approach an excellent way for testing new kernel extensions and for providing OS extensions that are not performance critical.

1.1 Personalized Global File Systems

With the recent explosive growth of the Internet an increasing number of users, including us, have access to

[1] The acronym Ufo stands for User-level File Organizer.

multiple computers that are geographically distributed. The initial motivation for our work was the desire to have transparent file access from our Unix machines to our personal accounts at remote sites. In addition, we also wanted to present the resources from the large number of existing HTTP and anonymous FTP servers as if they were local files. This would allow all local applications to transparently access remote files.

Ufo implements a global file system that provides this functionality. It is a user-level process that runs on multi-user Unix systems and connects to remote machines via authenticated and anonymous FTP and HTTP protocols. It provides read and write caching with a weak cache consistency policy.

It was important to us that the file system not only run at the user-level but that it also be user-installable (i.e. installing it does not require root access). For example, assume one of us obtains a new account at an NSF supercomputer center. Once we log into that account, we would like to transparently see all remote files we have some way of accessing (be it via telnet, FTP, rlogin, NFS, or HTTP); all without having to ask the system administrator to install anything. This is not necessarily easy to do in a Unix environment since most current file system software must be installed by a system administrator. For example, systems such as NFS and AFS allow sharing of files across the Internet, but they require root access to mount or export new file partitions. The system administrator may not have the time or, due to security concerns, may not be willing to install a new piece of software or export a file system resource.

A user-installable file system does not have these problems. Not only can users install it themselves, but it does not introduce any additional security holes in the underlying operating system or network protocol. To guarantee that a file system can indeed be installed by the user, it should only rely on functionality provided by standard (unmodified) operating systems.

1.2 Personalizing the Operating System: The Ufo Approach

In order to provide a global file system we need extensions to the operating system that handle file accesses (and related functions) properly. By modifying the behavior of the system calls, we can add new functionality to the operating system. In our approach we modify the system call behavior by inserting a user-level layer, the *Catcher*, between the application and the operating system.

The Catcher is a user-level process which attaches to an application and intercepts selected system calls issued by the application. From the user's perspective, the Catcher provides a user-level layer between the user's application processes and the original operating system, as shown in Figure 1. This extra layer does not change the existing OS, but allows us to control the user's environment, either by modifying function parameters, or issuing additional service requests.

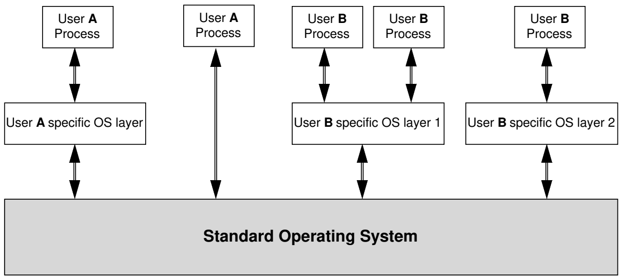

Figure 1: A new view of the operating system.

The Catcher operates as follows: Initially, it connects to the user process and tells the operating system which system calls to intercept. Our implementation which runs under Solaris 2.5.1 uses the System V /proc interface,[2] which was originally developed for debugging purposes [FG91]. Instead of just tracing the system calls, we actually change at the user-level the semantics of some of them to implement the global file system. Whenever a system call of interest begins (or completes), the operating system stops the subject process and notifies the Catcher. The Catcher calls the appropriate extension function, if needed, and then resumes the system call.

For our global file system, we intercept the *open*, *close*, *stat*, and other system calls that operate on files. When we intercept a system call which accesses a remote file, we first ensure that an up-to-date copy is available locally. Then, we patch the system call to refer to the local copy and allow it to proceed. System calls which only access local files are not modified (they can just continue), while most systems calls not related to files are not even intercepted. Since no application binaries are changed, this approach works transparently with any existing executable (with the exception of the few programs requiring setuid).

A potential concern with our approach is its performance overhead. While the cost for intercepting system calls is significant, our performance analysis shows that Ufo introduces acceptable overhead for common applications.

Since Ufo runs fully at the user-level, if one user runs it there is no performance penalty on another user. Furthermore, a user can run Ufo only on some selected applications without impacting other applications,

[2] Similar functionality is provided by Digital Unix, IRIX, BSD and Linux. This mechanism is used by system call tracing applications such as truss or strace.

or even dynamically "install" (attach) or "uninstall" (detach) Ufo while applications are running.

1.3 Using Ufo

Installing Ufo can be done by any user without needing root assistance. The simplest way to start using Ufo is to explicitly start processes under its control, e.g.

```
tcsh% ufo csh
csh% grep UCSB http://www.cs.ucsb.edu/index.html
csh% cd /ftp/schauser@cheetah.cs.ucsb.edu/
csh% emacs papers/ufo/introduction.tex &
```

In the example above, the new shell running under Ufo can use the global file system's services. Ufo automatically attaches to any child that the shell spawns, like the grep and emacs processes above. Alternatively, Ufo can be instructed to dynamically attach to an already running process by providing its pid.

```
tcsh% emacs &
[3] 728
tcsh% ufo -pid 728
```

1.4 Outline

The remainder of the paper is structured as follows. Section 2 reviews related work and compares our method for operating system extension with alternative approaches. Section 3 describes in detail the Catcher and how it intercepts system calls at the user-level. Section 4 discusses the design decisions of Ufo, the user-level global file system. Section 5 presents experimental results for a variety of micro-benchmarks, standard Unix file system benchmarks, and full application programs. Section 6 concludes this paper and offers an outlook on future research directions.

2 Related Work

Before presenting implementation details of the Catcher (in Section 3), we will put our work in context by comparing our approach with alternative ways of extending operating system functionality. The eager reader can skip directly to the discussion of our implementation in Section 3. We first introduce a classification of different approaches to extending the operating system. We then discuss the relevant research projects on extending operating system and file system functionality in more detail.

2.1 Approaches for Extending the Operating System

There has been a considerable amount of work on extending operating systems with new functionality. We can classify these approaches into the following categories:

Change Operating System: The most straightforward approach is to just modify the operating system itself and incorporate the desired functionality. This requires access to the OS sources and the privileges to install the new kernel.

Device Driver: Instead of changing the kernel itself, modifications can be limited to a new device driver which implements the desired functionality. Root access is required to install the device drivers.

Network Server: A clean solution with minimal intrusion to the operating system is to install a network server, which provides the additional services through an already existing standardized interface. Installing the server and mounting remote directories requires root capabilities.

We want to re-iterate, these first three approaches require super-user intervention and affect everybody using the system, since everybody will see the modifications to the operating system. If there is a bug or security hole in the newly installed software, the whole system's integrity and security can be compromised. A user-level approach avoids this problem.

User-level Plug-Ins: When a one-time modification to the operating system can be tolerated, a flexible strategy is to add hooks to the operating system so that system calls can trigger additional functions that extend the functionality. This approach is especially appropriate if the OS has already been designed to be flexible and support extensions.

User-level Libraries (Static or Dynamic Linking): Most applications do not directly access the operating system, but use library functions embedded in standard libraries. Instead of modifying all binaries or the OS kernel, it suffices to make changes to the libraries. Super-user privileges are only necessary if the original libraries/binaries need to be replaced.

Application Specific Modifications: Instead of incorporating the modifications into the library, we can also incorporate them directly into the application, avoiding the operating system altogether.

Intercept System Calls: Most modern operating systems provide the functionality of intercepting system calls at the user level. A process can be notified when another process enters or exits selected system calls. While the original motivation for this functionality was debugging and tracing of system calls, this mechanism can also be used

Method	Examples and References
Change Operating System	Sprite [NWO88], Plan 9 [PPTT90]
Device Driver	AFS, NFS, SLIC [GPA96] & WebFS [VDA96]
Network Server	ftp2nfs [Gsc94], Alex [Cat92]
User-level Plug-Ins	extended OS: SLIC [GPA96], UserFS [Fit96]
	flexible/extendible OS: SPIN [BSP+94], Exokernel [EKO95],
Statically Linked Library	Newcastle Connection [BMR82], Prospero [NAU93], Condor [Con95]
Dynamically Linked Library	Jade [RP93], IFS [EP93]
Application Specific	Ange-ftp [Nor]
Intercept System Calls	Interposition Agents [Jon93], Confinement [GWTB96], Ufo

Table 1: Different methods of extending operating system functionality and examples.

Method	Modify OS source	Requires root access	Re-compile applications	Re-link applications	Range of applications	Performance overhead
Change Operating System	X	X			all	very low
Device Driver		X			all	very low
Network Server		X			all	medium
User-level plug-ins	once				all	low-high
Statically Linked Library			X	X	user	low
Dynamically Linked Library				some	dyn. linked	low
Application Specific			X	X	single	low
Intercept System Calls					all (no setuid)	high

Table 2: Different methods of extending operating system functionality and their limitations.

to alter their behavior. This mechanism, which first was used in the context of Mach to implement interposition agents [Jon93], forms the basis for our Ufo implementation.

Table 1 lists examples of the above approaches, while Table 2 summarizes their limitations and identifies the context in which they can be applied. We wanted an approach which works with most existing applications without the need for recompiling, and more importantly, which can be used without requiring root access. Therefore we decided to use the mechanism of intercepting system calls.

2.2 Related OS Extensions

The project that is the closest to our own is the work on interposition agents [Jon93] which also makes use of the mechanism of intercepting system calls. Interposition agents provide a general system call tracing toolbox, which allows different system calls to be intercepted and handled in alternate ways, as we do in Ufo. Three example agent applications were implemented: spoofing the time of day, tracing system calls (as in truss), and transparently merging the contents of separate directories. The interposition agents work is based on Mach. While Mach is a Unix variant, it was designed to be more flexible and extensible. In particular, when calls are intercepted in Mach 2.5, they can be redirected to the process' *own* address space. Thus, the interposition agents are run in the user process' own memory. Approaches that use a more standard Unix, such as ours, are more constrained (and more complicated to implement) since it is more difficult to access the user process state from outside of the process' address space.

Another research project that uses the Unix trace mechanism for implementing an OS extension is Janus [GWTB96] which provides a secure, confined environment for running untrusted applications safely by intercepting and selectively denying system calls. Like ours, the Janus implementation has been designed for Solaris.

A lot of current research deals with designing operating systems such that they allow for easier and more efficient user-level extension. Engler et al. [EKO95] carry the Mach micro-kernel methodology [ABB+86] further by removing as much kernel abstraction as possible from the OS. This pushes the kernel/user-level boundary as low as possible, placing most of the OS services outside of the kernel. Another approach, taken by VINO [SESS94] and SPIN [BSP+94], is to allow injection of user-written kernel extensions into the kernel domain. A discussion of the issues involved can be found in [SS96].

Another recent project, SLIC [GPA96], is an OS extension to Solaris that allows for plug-ins at both the user and the kernel level.

We now discuss operating system extensions specific to our particular application: remote file transfer.

2.3 OS Extensions for Remote File Systems

There are a number of systems that provide transparent access to remote resources on the Internet, many of which have been very successful. Examples include NFS [SGK$^+$85], AFS [MSC$^+$86], Coda [SKK$^+$90], ftpFS in Plan 9 [PPTT90] and Linux [Fit96], Sprite [Wel91, NWO88], WebFS [VDA96], Alex [Cat92], Prospero [NAU93], and Jade [RP93]. They all have one significant drawback, however: they either require root access or modifications to the existing operating system, applications or libraries. Ufo is distinct in that it requires no such modifications to any existing code and runs entirely at the user-level.

There are a few systems for global file access that run entirely at the user-level and are user-installable. They are also similar to Ufo in that they extend a local file system to provide uniform and transparent access to heterogeneous remote file servers. Prospero [NAU93] and Jade [RP93] both provide access to NFS and AFS file systems, and to FTP servers. Prospero runs at user-level by replacing standard statically linked libraries. This avoids changes to the operating system, but requires re-linking of existing binaries. Jade [RP93] uses dynamic libraries instead and allows most dynamically linked binaries to run unmodified. Changing application libraries works well for most applications, especially when combined with dynamic linking. The drawback of this approach is that it does not work for statically linked applications not owned by the user as well as for applications that circumvent the standard librtaries and execute system call instructions directly.

Other global file systems also run at the user level, but are not user-installable, since they require extensions to the operating system itself, which in turn requires root access. One such example is WebFS [VDA96], a global user-level file system based on the HTTP protocol. To run at the user level, WebFS relies on the OS extensions provided by SLIC [GPA96], which implements a callback mechanism to a user process. (WebFS also requires the HTTP server be extended with a set of CGI scripts that service requests.) Similar to SLIC, UserFS [Fit96] is an OS extension that enables user-level file systems to be written for Linux. While installing UserFS itself requires kernel recompilation, installing new file modules, such as ftpFS, does not. Plan 9 [PPTT90] also includes an FTP based file system (also called ftpFS). At least two projects provide access to FTP servers by implementing an NFS server that functions as an FTP-to-NFS gateway. Alex [Cat92] supports read-only access to anonymous FTP servers, while [Gsc94] additionally allows read and write access to authenticated FTP servers.

3 Catcher Implementation

In this section we discuss the details of our implementation of the Catcher inside Ufo. We start by describing the high-level architecture and the role of the Catcher in Ufo.

3.1 The Ufo Architecture

Ufo is a user-level process which provides file system services to other user-level processes by *attaching* to them. Once attached to a subject process, it intercepts system calls and services them if they operate on remote files. The application is unaware of the existence of the Ufo, but, with Ufo's help, it can operate on remote files as if they were local.

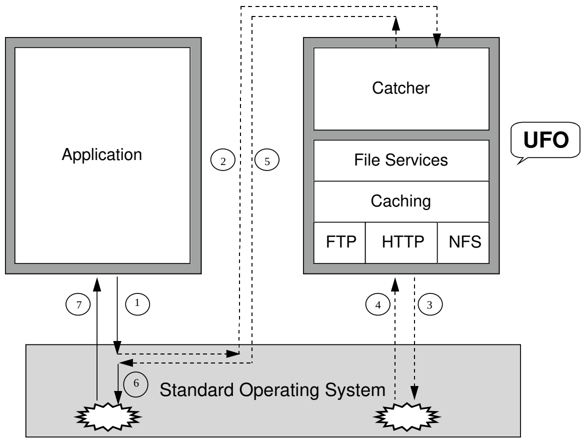

Figure 2: General architecture of Ufo.

Ufo is implemented in two modules: the Catcher and the Ufo module (Figure 2). The Catcher is responsible for intercepting system calls and forwarding them to the Ufo module. The Ufo module implements the remote file system and consists of three layers: the File Services layer which identifies remote files, the Caching layer, and the Protocol layer containing different plug-in modules implementing the actual file transfer protocols.

Figure 2 shows the steps involved in servicing a remote file request. When the application issues a system call (1), it can go directly to the kernel or, if it is file-related, get intercepted by the Catcher (2). For intercepted calls, Ufo determines whether the system call operates on a remote or a local file, possibly using kernel services (3,4). If the file is local, the request proceeds unmodified. If the file is remote, Ufo creates a local cached copy, patches the system call by modifying its parameters, and lets the request proceed to the kernel (5). After the request is serviced in the kernel (6), the result is returned to the application (7). The return from the system call may also be intercepted and patched by Ufo, though the figure does not show this.

3.2 Catcher Implementation Details

In our Solaris implementation, the Catcher monitors user processes using the /proc virtual file system [FG91]. This is the same method used by monitoring programs, such as truss or strace, which are also available on a number of other UNIX platforms, including Digital Unix, IRIX, BSD or Linux. The System V /proc interface allows us to monitor and modify an individual process by operating on the file associated with a user process.

In particular the Catcher attaches to a subject process *pid* by opening the /proc/*pid* file. Once attached, the Catcher uses *ioctl* system calls on the open file descriptor to control the process. It can instruct the operating system to stop the subject process on a variety of events of interest. In Ufo there are two events of interest: system call entry into the kernel, and system call exit from the kernel. Once a subject process has stopped on an event of interest, the Catcher can read and write the registers and read and write in the address space of the process. The Catcher uses this to examine and modify the parameters or the result of system calls like *open*, *stat*, and *getdents*. Finally, the /proc interface allows us to restart the execution of a stopped process. Figure 3 summarizes how the discussed functionality is used in the Catcher.

```
connect to subject process
register the set of system calls to intercept
while subject process is running do
        wait for process to stop on system call entry or exit
        determine system call number and its parameters/result
        call handler function for this system call
        if system call is fork then
                begin monitoring new child process
        resume system call
endwhile
```

Figure 3: Outline of the Catcher algorithm.

Conceptually, Ufo implements the system calls intercepted by the Catcher, but in practice Ufo does not service them directly. Although implementing the system calls directly in Ufo would be possible, it would require reimplementing existing OS functionality. Instead, we "patch" the system calls by *(i)* modifying the call's parameters, *(ii)* changing the file system state (e.g., fetching a file from a remote server) and *(iii)* modifying the result returned from the operating system. A good example for the first two actions is the *open* system call. On the entry of an open call, we may have to modify the file name string to point to the locally cached copy. Before allowing the system call to continue, the Catcher may have to wait for Ufo to download the file from the remote site. Implementing the name change is somewhat complicated since we must modify the user's address space. We cannot just change the filename in place since the new filename might be longer than the old one. Also, the filename could be in a segment which is read-only or shared among threads. Currently we solve these problems by writing the new file name in the unused portion of the application's stack and changing the system call argument to point to the new string.

The open system call needs to be intercepted on exit from the kernel as well. Although the returned result is not modified, Ufo must remember the correspondence between the returned file handle and the file name, which is needed when the file is closed.

Besides file related system calls, there are several others that must be intercepted. For example, to track child processes we intercept the *fork* system call. Given the child pid, we can open its associated /proc file and monitor it as well. System V allows the set of trapped system calls to be automatically inherited from parent to child, so this setup is only needed for the initial process.

3.3 User-Level Restrictions

Implementations of file system functionality at the user level must obey some restrictions since user-level processes cannot perform arbitrary actions and cannot access the whole file system related state of the kernel.

One problem is that the Catcher cannot control setuid process since the security policy of the operating system disallows user-level processes from attaching to other users' processes. In practice we have found this not to be a problem since very few programs are installed with setuid. And for most of those programs, e.g., rlogin, it is not clear whether one really needs the file system extensions. In the current implementation, whenever the Catcher detects that a subject process is about to spawn a setuid program, it just does not trace the child process.

In Solaris, the /proc interface allows the controlling process to write in the subject process' address space. This is important if the Catcher needs to change some system call arguments such as filename strings for Ufo. If we are to port the Catcher to other operating systems that do not provide the capability of writing into the subject process, we would not be able to implement this feature. Although writing in the user process is not necessary for the basic functionality of Ufo, a Ufo implementation on such an operating system would have some limitations. Features such as the URL naming scheme and mountpoints in the root directory require changing of string arguments to system calls and therefore would not be possible to implement (see Subsection 4.1).

Another problem arises when the Catcher process is killed. Being a regular user-level process, the Catcher cannot protect itself against the SIGKILL signal. There is no graceful way to handle such a situation if the subject processes running under the Catcher continue working on remote files. In the current Ufo implementation the subject process will be trapped on the next intercepted system call and stay trapped until killed.

3.4 Catcher Discussion

The Catcher mechanism allows us to create a personalized operating system. Requests made to the kernel can be re-interpreted, in effect allowing individual users to run their own OS. Any user can use the "new" OS without having to modify the original operating system or needing root access. Although, the current Catcher only intercepts system calls, System V allows the user to also intercept and act on signals and hardware faults. This allows for a wide range of OS functionality to be extended using the Catcher mechanism. Other potential uses of the Catcher for personalized OS extensions include encrypting file systems, file systems which store files in compressed form, confined execution environments for runing untrusted binaries [GWTB96], virtual memory paging [DWAP94, FMP^{+}95], and process migration [Con95].

A potential concern with our approach is its performance overhead. Indeed, intercepting individual system calls is quite expensive and for some OS extensions this overhead would be unacceptable. Nevertheless, Ufo is an example that there are OS extensions for which the Catcher mechanism works well. Our performance analysis shows that Ufo introduces moderate overhead for common applications. This is due to the fact that typical applications issue relatively few system calls, and not all system calls are intercepted in Ufo.

4 Ufo's Global File System Module

Ufo provides read and write access to FTP servers and read-only access to HTTP servers. The remote file access functionality is implemented in Ufo's file system module which is responsible for resolving remote file names, transfering files, and caching.

4.1 Naming Strategies

Ufo supports three ways of specifying names of remote files: *(i)* through a URL, *(ii)* through a regular filename implicitly containing the remote host, user name, and access mode, and *(iii)* through mount points.

The first way to specify a remote file is through its URL syntax. Unfortunately, some applications cannot handle URL names. Make and gmake cannot handle the colon in the URL, while Emacs considers // to be the root of the file system and thus discards everything to the left.

To alleviate these problems we also support specifying a remote file through a regular filename. The general syntax is `/protocol/user@host/filename` where `protocol` is the file transfer protocol, e.g., `ftp` or `http`.

Lastly, Ufo allows the user to specify explicit mountpoints for remote servers or access protocols in a *.uforc* file. For example, the line

```
local /csftp remote /
             machine ftp.cs.ucsb.edu method FTP
```

specifies that accesses relative to `/csftp` refer to the root directory of the `ftp.cs.ucsb.edu` anonymous FTP server. The user can also specify mountpoints for access methods. In fact that is how the second naming scheme is implemented: if the user does not explicitly specify a mount point for the HTTP method, for example, Ufo uses the implicit mountpoint:

```
local /http method HTTP
```

Similarly to Sprite [NWO88], we have implemented mount points using a prefix table which, given a filename, searches for the longest matching prefix in the list of mount points.

Ufo also supports symbolic links. A user can create links to frequently accessed remote directories. While links simplify accesses to remote files, they actually present quite an implementation challenge, since they require following all link components to determine the true name of a file.

4.2 Accessing Remote Files and Directories

Ufo transfers only whole files to and from the remote file system. Whenever Ufo intercepts the *open* system call for a remote file, it ensures that a local copy of the file exists in the cache, and then redirects the system call to the local copy. *Read* and *write* system calls don't even have to be intercepted since they operate on file descriptors returned by the *open*; they will correctly access the local copy in the cache. Finally, on a *close* system call, Ufo checks whether the file has been modified and if so, stores the file back to the server (the store may be delayed if write-back caching is in effect). Ufo uses whole file transfers for two reasons: this minimizes the number of system calls that need to be intercepted, and protocols such as FTP only support whole file transfers.

When an application requests information about a remote file, e.g., through a *stat* or *lstat* system call, Ufo satisfies the request by creating a local *file stub* and redirecting the system call to it. The file stub has the correct modification date and size of the remote file but contains no actual data.[3] With this approach Ufo neither has to re-implement the *stat* system call, nor download the whole file. Only if the application wants to open a file stub later, will Ufo actually download the remote file. Similarly, when a system call such as *getdents* (get directory entries) is issued on a remote directory, Ufo creates a copy of the directory in the local cache and puts file stubs in it. Then, it redirects the system call to the so created skeleton directory.

4.3 Caching and Cache Consistency

Since remote data transfers can be quite slow, Ufo implements caching of remote files to achieve reasonable performance. Instead of downloading a file each time the user opens it for reading, Ufo keeps local copies of previously accessed files. Ufo can reuse the local copy on a subsequent access, as long as it is up-to-date. Similarly, we use write-back caching which delays writing a modified file back to the remote server. While files are the primary objects cached, Ufo also caches directory information (directory contents), and file information (size, modification time, permissions). The FTP module additionally caches open control connections. Since establishing a new connection to the remote server for each transfer is expensive, we reuse open control connections by keeping them alive for a period of time after a transfer has completed.

The cache consistency policy governs whether we are allowed to use a local copy on a read, and whether we can delay the write-back of a modified file. To efficiently support a wide range of usage patterns, Ufo provides an adjustable consistency policy based on timeouts (a read and write delay). The policy guarantees that *(i)* when a file is opened it is no more than T_{read} seconds out of date; and *(ii)* changes made to a file will be written back to the server within T_{write} seconds after the file is closed. To verify that a local file is up to date (i.e., is not stale), Ufo checks whether the file on the remote site has changed (*validate on open*). T_{read} and T_{write} can have a zero value. In this case files opened for reading are never stale and modified files are written back to the server immediately after they are closed.

The write timeout of a file is always a certain number of seconds. The read timeout can optionally be specified as a percentage of the file's age as in Alex [Cat92]. This method is based on the observation that older files are less likely to change than newer files. Therefore older files need to be validated less often. Files can have individual timeouts and Ufo provides mechanisms for the user to define default timeouts for all files, or for all files on a server. This allows the user to adjust the tradeoff between performance and consistency based on known usage patterns. For example, when mounting read-only binaries large read timeouts can be used since these files change rarely.

4.4 Authentication and Security

Ufo relies on the underlying access protocols for authentication. Currently, passwords are only required for authenticated FTP servers and are not needed for HTTP and anonymous FTP accesses. Ufo allows the passwords to be stored in the *.uforc* or *.netrc* files, or alternatively, Ufo asks for the password on the first access to a remote server.

Since Ufo is running entirely at the user level with the access permissions of its owner, it does not introduce new security problems in the system. The only potential security concern is to ensure that other users do not gain undesired access to the files in the private Ufo cache. We avoid this problem by creating the topmost cache directory with read and write permissions for the owner only.

4.5 Implementation Trade-Offs

In implementing Ufo, we tried to minimize the amount of operating system functionality that we had to reimplement. First, we attempted to minimize the number of intercepted system calls in order to minimize the execution overhead that Ufo introduces. This lead to the whole file caching policy. Second, we wanted to minimize the implementation effort by modifying/reimplementing as few system calls as possible. This lead to our decision to create file stubs and skeleton directories for the *stat* and *getdents* calls.

Of course, there is a trade-off between execution overhead and implementation effort. For example, the advantage of creating file stubs and skeleton directories is that we do not have to reimplement the *stat* and *getdents* system calls. The disadvantages are that creating file stubs may have high overhead. Also for efficiency, we rely on the support for holey files by the local file system. For example, on our machines the /tmp file system does not support holey files, thus

[3] Creating a stub is done by seeking to the desired position in a newly created file and then writing a single byte. On most file systems, the so created stub occupies a small amount of disk space, independent of the reported file size.

if we use /tmp for the Ufo cache the stubs for large files do use all the disk space indicated by their size. The NFS-mounted file systems at our site do support holey files, but the stub creation there is an order of magnitude slower than on /tmp. For these reasons we are considering implementing the *stat* and *getdents* system calls completely inside the next version of Ufo to improve its performance. In fact, we are already partially implementing (patching) the *getdents* system call in order to support Ufo mountpoints in user-unwritable areas such as the root directory.

Transferring only whole files introduces three well known problems for extremely large files [Cat92]. First, when only a small fraction of a file is actually accessed, a lot of unnecessary data may be transferred. Second, the whole file has to fit on the local disk. In practice we don't expect these two problems to occur frequently. With the exception of databases, most applications tend to access files nearly in their entirety [BHK+91]. Furthermore, Ufo allows any local file system to be used for file transfers, thus reducing the danger of insufficient local disk space. A third problem comes from our decision not to intercept the *read* and *write* system calls. In our approach the open call blocks until the whole file has been transferred. It is possible to intercept and handle *read* and *write* system calls in Ufo. The benefit is that *open* would not always block:[4] reads that operate on the already present part of a file could be executed without waiting for the completion of the whole transfer (see Alex [Cat92]). The drawback is that intercepting *read* and *write* calls incur a high overhead and requires extra implementation effort.

5 Performance Measurements

The main goal of our performance analysis is to measure the overhead introduced by the Catcher mechanism in Ufo. This information is necessary to determine the usability of our method for operating system extension.

We first present the results of several microbenchmarks, which measure the overhead of intercepting individual Unix system calls. To demonstrate the overall impact of this overhead on whole applications we also present measurements for a set of file system benchmarks and a set of real-life applications. While the microbenchmarks show that intercepting system calls is expensive, the real-life applications exhibit much lower overhead.

All tests were run on a 143 MHz Sun Ultra 1 workstation with 64 megabytes of main memory running Solaris 2.5.1.

[4] Several other system calls like *lseek* also have to be intercepted for this to work.

5.1 Microbenchmarks

The microbenchmark results present the user-perceived run times (measured as wall clock times) for *open*, *close*, *stat*, *read*, *write*, and *getpid* system calls. The results are shown in Table 3. The columns show the numbers for the normal user program, for the Catcher-monitored program (Catcher only, no calls to Ufo functions), and for the Ufo program (Catcher and Ufo functionality). In the latter case, we examine the run times for a local file, for a cached remote file and for a remote file that has not been cached.

The *Catcher only* and *Ufo local file* numbers are of special significance. They show the cost of running a process under the Catcher or under Ufo when the process accesses local files only and does not require any of the extended OS functionality. This is the the fundamental overhead introduced by our method of extending the OS. The numbers for remote files are a measure of the combined effect of our remote file system implementation, our caching policy, the efficiency of the underlying access protocol (FTP in this case), and the quality of the network connection.

In order to measure the cost of the Solaris system calls themselves and not the network speed or the NFS overhead, we used the local /tmp file system. Accesses to /tmp are very fast and do not involve disk, network traffic or protocol overhead. As a result the microbenchmarks present the Catcher and Ufo overhead in the worst-case scenario. The relative Catcher and Ufo overhead for accessing non-cached NFS files, for example, is much lower.

The microbenchmarks were run on a lightly loaded workstation by taking the *wall-clock* time just before and just after the system call. The timing was done using the high resolution timer *gethrtime* which has a resolution of about 0.5 microseconds on the Ultra 1 workstation. Since individual system calls are very fast, normal system activity such as interrupts and context switches distorts some of the measurements. This produces a small percentage of outliers that are several times larger than the rest of the measurements. To ensure we do not include unrelated system activity in our measurements, in each test run we recorded 100 measurements and discarded the highest 10% of them. The remaining times were then averaged. The numbers in the table are the arithmetic mean of five such runs. The standard deviation for the five runs was below 2% for all tests, except for *getpid*, for which the standard deviation was at most 6%.

The *Catcher only* numbers show the cost of intercepting system calls. The results are obtained by running the benchmark program under the control of the Catcher alone. The Catcher simply intercepts the *open*, *close* and

System Call	Standard OS		Catcher only		Ufo local file		Ufo remote cached		Ufo remote no cache	
open	28 μs	(1.00)	211 μs	(7.54)	611 μs	(21.8)	2383 μs	(85.1)	531 ms	(18964)
close	12 μs	(1.00)	108 μs	(9.00)	300 μs	(25.0)	1530 μs	(128)	452 ms	(37667)
stat	33 μs	(1.00)	133 μs	(4.03)	472 μs	(14.3)	1744 μs	(52.8)	198 ms	(6000)
getpid	3 μs	(1.00)	5 μs	(1.67)	5 μs	(1.67)	5 μs	(1.67)	5 μs	(1.67)
write 1b	23 μs	(1.00)	26 μs	(1.13)	26 μs	(1.13)	26 μs	(1.13)	26 μs	(1.13)
read 1b	25 μs	(1.00)	28 μs	(1.12)	28 μs	(1.12)	28 μs	(1.12)	28 μs	(1.12)
write 8K	97 μs	(1.00)	101 μs	(1.04)	101 μs	(1.04)	102 μs	(1.05)	101 μs	(1.04)
read 8K	75 μs	(1.00)	78 μs	(1.04)	78 μs	(1.04)	79 μs	(1.05)	79 μs	(1.05)

Table 3: Run times in microseconds for various system calls for accessing files in /tmp (the numbers are the arithmetic mean of 5 runs, each executing 100 iterations). The numbers in parentheses represent the ratio normalized to the standard Solaris OS.

stat system calls executed by the benchmark program, and lets them continue immediately without modifying them. The *read*, *write* and *getpid* system calls are not intercepted at all. Even though one may expect that these system calls will not be affected, they do incur a small overhead: whenever there is even a single intercepted system call for a process, the operating system takes a different execution path for all system calls of that process, independent of whether they are intercepted or not. The results demonstrate that for *read* and *write* of 1 byte blocks this overhead is small and for 8K blocks it is negligible. Because *getpid* is so fast, it has a substantial relative overhead, but still only 2 μs total. On the other hand, system calls that must be trapped by the Catcher incur a factor of 4–9 overhead. During this extra time, control is passed from the program to the Catcher, (which performs *ioctl* calls to read information from the */proc* file system), and then back again.

The *Ufo local file* column shows how much extra overhead is introduced by Ufo in addition to the Catcher. The benchmark program is running under Ufo and is accessing local files only. Even though no remote files are accessed, Ufo still introduces some overhead in addition to the Catcher overhead. The extra overhead comes from the analysis of the parameters of the intercepted system calls. For system calls that reference a file, Ufo determines whether the file is indeed local or remote. Since a system call does not necessarily take an absolute path name as an argument, Ufo has the responsibility of determining it. Determining the true filename can involve a number of *stat* system calls, similar in flavor to the pwd command, and this can add a noticeable overhead.

The remaining two columns measure the overhead of Ufo when working with remote files. These numbers are measured as with *Ufo local file*, except that the accesses are to remote files. For the *Ufo remote cached* tests, a locally cached copy of the remote file is accessed. Note that in either case (cached or uncached), the *read* and *write* system calls operate on the locally cached copy of the file. Thus, these numbers are consistent across all of the tests. On the other hand, *open* and *stat* calls to uncached remote files require remote accesses, and the overhead increases dramatically when Ufo uses the FTP protocol to retrieve the file. This overhead is almost entirely determined by the quality of the network connection and the FTP protocol. In our measurements we accessed files located at UC Berkeley. From a UC Santa Barbara machine, opening a remote file of size 1024 bytes residing at a UC Berkeley host requires 531ms using FTP. Closing the same remote file after modifying it takes 452ms since the file must be written back to the remote server. If the file is cached, the *open*, *close* and *stat* overhead is much smaller, but it still has roughly four times the overhead compared to a local file. This is due to two reasons: the additional work to manage the cache, and several remaining inefficiencies in our prototype implementation which will be corrected in future versions of Ufo.

5.2 File System Benchmarks

Table 4 reports the absolute execution times in seconds for two file system benchmarks run on the local /tmp file system with and without Ufo and on a remote FTP-mounted file system with and without caching. For these tests, the FTP host was a machine on the local 100Mbit/s Ethernet network. The remote tests with caching were with a warm cache and read and write delays set to infinity. Thus, these measurements represent the best-case scenario for remote files. For the remote tests without caching, the read and write delays were set to zero, forcing every *open*, *close* and *stat* system calls to go to the remote site. These tests are the worst-case scenario for accessing remote files under Ufo.

Iostone and *Andrew* are standard file system benchmarks. We chose these as examples of applications that execute a lot of file system calls that Ufo intercepts

Benchmark	System calls total	System calls intercepted	Standard OS		Ufo local file		Ufo remote cached		Ufo remote no cache	
iostone	99203	48762	3 s	(1.00)	25 s	(8.33)	102	(34)	7744	(2581)
andrew										
makedir	641	465	0 s[†]	(1.00)	1 s	(-)	1 s	(-)	3 s	(-)
copy	9596	6999	3 s	(1.00)	6 s	(2.00)	7 s	(2.33)	68 s	(22.7)
scandir	8674	4115	6 s	(1.00)	7 s	(1.17)	10 s	(1.67)	61 s	(10.2)
read all	12907	6084	10 s	(1.00)	13 s	(1.30)	16 s	(1.60)	68 s	(6.80)
make	8407	3107	12 s	(1.00)	14 s	(1.17)	16 s	(1.33)	22 s	(1.83)
total	40225	20770	30 s	(1.00)	40 s	(1.33)	50 s	(1.67)	321 s	(10.7)

Table 4: Run times for the Iostone and Andrew file system benchmark programs with and without Ufo. Times are in seconds, with the ratios normalized to the standard OS shown in parenthesis. ([†]The Andrew benchmark reports its timing results with a resolution of 1 second. The 0 seconds in the table indicate a measurement between 0 and 1 second.)

and handles. The *Iostone* benchmark [IOS87] performs thousands of file accesses (opening, reading, and writing). Because of the large amount of file opens and closes, Ufo runs about 8 times slower on the local file system. The *Andrew* benchmark [HKM$^+$88] measures five stages in the generation of a software tree. The stages *(i)* create the directory tree, *(ii)* copy source code into the tree, *(iii)* scan all the files in the tree, *(iv)* read all of the files, and finally *(v)* compile the source code into a number of libraries. For this benchmark the Ufo overhead on local files is a factor of 1.33, much lower than the overhead for *Iostone*. For both *Andrew* and *Iostone*, the results for the uncached remote tests are orders of magnitude worse than for the local /tmp file system. This is not not surprising since the network latency and the FTP protocol overhead are quite large compared to the fast accesses in /tmp.

5.3 Application Programs

We also tested Ufo with a number of larger Unix applications: *latex*, *ghostscript*, a *make* of the Ufo executable, and the integer applications from the SPEC95 benchmark suite. The results are shown in Table 5. As with the file system benchmarks, each test was run without Ufo, under Ufo on local files only, and under Ufo on remote files with and without caching.

The first set of benchmarks are programs that we run frequently. The *latex* test measures the time to latex three times a 20 page paper consisting of 8 tex files and then produce a postscript from the dvi file. The *make* test compiles Ufo itself using g++. The *ghostscript* test displays a 20 page postscript document. The table shows that *latex* and *make* perform a relatively large number of system calls that Ufo intercepts, mainly *open*, *close*, and *stat*. This results in Ufo overheads of 24% and 22% respectively, when run locally, and higher overheds, when run remotely. The remote overheads, while large, should be acceptable to the user, since accessing remote files is expected to cost extra time. The local overheads on the other hand, are incurred only because the application is running under Ufo even though it is not using any of its functionality. To avoid unnecessary local overhead, applications that only access local files can be run without Ufo, and Ufo can be detached from applications once they stop accessing remote files.

The *ghostscript* test on the other hand performs few calls that Ufo intercepts and never writes to the remote server; as a result the Ufo overhead is very low even in the remote test. This sort of overhead should be unnoticeable to the user.

The last eight tests are the integer applications from the SPEC95 benchmark suite. These were chosen as examples of compute intensive applications that do not perform extensive file system operations. For these applications the observed overhead is very small in the local and even in the remote tests. Small perceived overheads should also be expected for interactive applications such as text editors since the user is not likely to notice the difference between $28\mu s$ and $611\mu s$ when opening a local file.

5.4 Summary of Experimental Results

As expected, we find that intercepting system calls can be very expensive, and remote accesses are orders of magnitude higher than local accesses. For programs such as the Iostone benchmark — which performs many *open*, *close* and *stat* calls — the Ufo overhead for local files is too large to be ignored. Clearly, such programs should not be run under Ufo if they only access local files since this will incur a large overhead even though the program does not utilize any of the extended functionality. If remote files need to be accessed, then programs like Iostone will run slow, but this is mainly due to the network latency and access protocol overhead which

Application	System calls total	System calls intercepted	Standard OS		Ufo local file		Ufo remote cached		Ufo remote no cache	
latex	7638	4396	13.0 s	(1.00)	16.1 s	(1.24)	17.6	(1.35)	90.0 s	(6.92)
make	22783	6762	31.0 s	(1.00)	36.5 s	(1.22)	38.5	(1.24)	104.8 s	(3.38)
ghostscript	5495	167	4.1 s	(1.00)	4.3 s	(1.05)	4.4	(1.07)	5.6 s	(1.37)
0.99.go	131	48	833 s	(1.00)	876 s	(1.05)	880	(1.06)	914	(1.10)
124.m88ksim	161	18	469 s	(1.00)	470 s	(1.00)	472	(1.01)	506	(1.08)
126.gcc	21234	1008	343 s	(1.00)	348 s	(1.01)	348	(1.01)	365	(1.06)
129.compress	40	14	344 s	(1.00)	348 s	(1.01)	348	(1.01)	344	(1.00)
130.li	250	60	476 s	(1.00)	478 s	(1.00)	486	(1.02)	498	(1.05)
132.ijpeg	1050	45	484 s	(1.00)	486 s	(1.00)	482	(1.02)	498	(1.03)
134.perl	6019	42	446 s	(1.00)	454 s	(1.02)	445	(1.00)	458	(1.03)
147.vortex	8490	27	602 s	(1.00)	604 s	(1.00)	602	(1.00)	607	(1.01)

Table 5: Relative run times for some file system benchmarks and larger Unix applications. Times are in seconds, and the relative speed in parentheses. The first column shows the number of system calls executed by the application.

by far outweighs the Catcher and Ufo overhead as shown in Table 3. In this case Ufo proves to be a convenient tool. Furthermore, Ufo allows the user to dynamically attach to a running process and detach from it, so the choice of running under Ufo or not is always available. Applications that need access to remote files can have it, and the remaining processes will not incur any overhead.

Other applications, such as *make*, and *latex* incur a 22-24% overhead on local files — noticeable, but perhaps acceptable to the user even when the functionality of Ufo is not required. For remote files these applications incur overheads of 24% in the best case and 600% in the worst case depending on the kind of file caching used. In most cases the user expects that working on remote files would be slower, so the use of the extra functionality provided by Ufo should be worth the additional overhead, especially when the only alternative is to manually transfer files using FTP. Many other applications, such as compute intensive programs or text editors, make infrequent use of the system calls trapped by Ufo (though they may use other calls such as *read* and *write*). For such applications, user-perceived delays are much smaller: on the order of a few percent. In this case, running applications under Ufo makes no appreciable difference.

From these observations we can draw the conclusion that the Catcher is a good tool for implementing operating system extensions that require the interception only of relatively infrequent system calls. An example of such an extension is the Ufo file system when running real-life applications. On the other hand, this method is not ideal for extensions which intercept frequently occurring system calls.

Finally, we would like to mention that we are aware of some opportunities for improving the current Catcher prototype and many opportunities for improving Ufo. For example, optimizing the filename check in Ufo (to determine whether a file is local or remote) alone will result in a significant reduction of the running time.

6 Conclusions and Future Work

In this paper, we presented a general way of extending operating systems functionality, using the debugging and tracing facilities provided by many Unix operating systems. Selected system calls are intercepted at the user-level and augmented to obtain the desired functionality. This mechanism forms the basis for Ufo, a file system providing transparent access to remote files on FTP and HTTP servers. Ufo proved to be a useful tool which we now use daily. As our experimental results show, its overhead, while quite large for intercepted system calls, is acceptable for most applications.

We believe that our approach is a promising way for individual users to develop and experiment with future operating system extensions, since this can be done completely at the user-level. Essentially, each user sees a personalized version of the operating system, extensions do not affect other users and are compatible with existing applications as those need not be re-compiled or re-linked. In the past, operating systems research had a hard time to carry over to the general public. With our approach, researchers can make their extensions easily available, and users can run them without relying on the system administrator for installation.

There are plenty of avenues for future work and research. For example, we have several ideas on how to improve the performance of the Catcher and Ufo. We also plan to implement new protocol modules in Ufo, e.g. based on NFS, WebNFS, and the rlogin protocols. We have experimented with several other OS extensions suitable for cluster of workstation environments. For example, we have developed a prototype that attaches to a process, checkpoints it, and then can restart it at a later time or migrate it to another processor. Similiarly, we have a prototype Catcher which intercepts all forks and execs and sometimes decides to execute some processes on other workstations. While both tools are still at a very crude stage, we have already seen some of their potential benefits. Similar benefits can be expected for paging virtual memory to the memory of idle processors instead of to a slow local disk.

Another interesting research area is protected computing. The system calls define the capabilities a process has and resources it can obtain (memory, disk access, CPU time). We can use the Catcher to limit the resources a process can access or obtain. This approach, implemented in Janus [GWTB96], is especially interesting in the current development of global computing, where one user may run an untrusted binary fetched from the Internet.

Finally, we intend to generalize our design of the Catcher since it can not only intercept system calls, but also signals and hardware traps which are delivered to the application. We intend to build a Catcher toolbox which can be used for OS courses and research projects.

7 Acknowledgements

We would like to thank Urs Hölzle for insightful discussion of this paper and for helping test early versions of Ufo. We would also like to thank Dave Probert, Chad Yoshikawa, Roger Faulkner, Arvind Krishnamurthy, and the anonymous referees for their valuable feedback.

References

[ABB+86] M. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.

[BHK+91] M. G. Baker, J. H. Hartmann, M.D. Kupfer, K. W. Shirrif, and J. K. Ousterhout. Measurement of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, Pacific Grove, CA, 1991.

[BMR82] D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle Connection, or UNIXes of the world unite! *Software — Practice and Experience*, 12, 1982.

[BSP+94] B. N. Bershad, S. Savage, P. Pardyak, E. F. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, 1994.

[Cat92] V. Cate. Alex — a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, Ann Arbor, MI, May 1992.

[Con95] The Condor Team. Checkpoint & migration of UNIX processes in the Condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[DWAP94] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating System Design and Implementation*, May 1994.

[EKO95] D. Engler, F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource managment. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[EP93] P. R. Eggert and D. S. Parker. File systems in user space. In *Proceedings of the Usenix Winter 1993 Technical Conference*, Berkeley, CA, 1993. Usenix Association.

[FG91] R. Faulkner and R. Gomes. The process file system and process model in UNIX system V. In *Proceedings of the 1991 USENIX Winter Conference*, 1991.

[Fit96] J. Fitzhardinge. Userfs: A user file system for Linux. ftp://sunsite.unc.edu:pub/Linux/ALPHA/userfs/userfs-0.9.tar.gz, 1996.

[FMP+95] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[GPA96] D. P. Ghormley, D. Petrou, and T. E. Anderson. SLIC: Secure loadable interposition code. Technical Report CSD-96-920, University of California, Berkeley, 1996.

[Gsc94] M. Gschwind. FTP — access as a user-defined file system. *ACM Operating Systems Review*, 1994.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications — Confining the Wily Hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[HKM+88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[IOS87] IOStone. A synthetic file system performance benchmark. Technical Report TR-074-87, Princeton University, 1987.

[Jon93] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th Symposium on Operating Systems Principles*, New York, NY, 1993.

[MSC+86] J. Morris, M. Satyanarayananan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3), 1986.

[NAU93] B. C. Neumann, S. S. Augart, and S. Upasani. Using Prospero to support integrated location-independent computing. In *Proceedings of the Symposium on Mobile and Location-Independent Computing*, Cambridge, MA, 1993.

[Nor] A. Norman. *Ange-Ftp Manual*. Free Software Foundation, Inc.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[PPTT90] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell labs. In *Proceedings of the UKUUG Conference*, July 1990.

[RP93] H. C. Rao and L. L. Peterson. Accessing files in an internet: The JADE file system. *IEEE Transactions on Software Engineering*, 19(6), June 1993.

[SESS94] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the VINO architecture. Technical Report TR34-94, Harvard University, 1994.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of the Summer USENIX conference*, June 1985.

[SKK+90] M. Satyanarayananan, 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. *IEEE Transactions on Computers*, 39(4), 1990.

[SS96] M. Seltzer and C. Small. A comparison of OS extension technologies. In *Proceedings of the 1996 Usenix Technical Conference*, San Diego, CA, January 1996.

[TLC85] M. Theimer, K. Landtz, and D. Cheriton. Preemptable remote execution facilities for the V system. In *Proceedings of the 10th ACM Symposium on Operating System Priciples*, December 1985.

[VDA96] A. Vahdat, M. Dahlin, and T. Anderson. Turning the web into a computer. Technical report, University of California, Berkeley, 1996.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[VGA94] A. M. Vahdat, D. P. Ghormley, and T. E. Anderson. Efficient, portable, and robust extension of operating system functionality. Technical Report CS-94-842, University of California, Berkeley, December 1994.

[Wel91] B. B. Welch. Measured performance of caching in the Sprite network file system. *Computer Systems*, 3(4), 1991.