Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System

ALBERT D. ALEXANDROV, MAXIMILIAN IBEL, KLAUS E. SCHAUSER, and CHRIS J. SCHEIMAN
University of California at Santa Barbara

In this article we show how to extend a wide range of 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 relinking 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. Individual users can choose what extensions to run, in effect creating a personalized operating system view for themselves. 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 article 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 individual system calls incurs a high cost.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*access methods*; *distributed file systems*

General Terms: Performance

Additional Key Words and Phrases: File caching, global name space, proc file system, user-level operating system extensions

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.
Authors' address: Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106; email: {berto; ibel; schauser; chriss}@cs.ucsb.edu.

1. INTRODUCTION

Extensibility is becoming a very important issue in modern operating systems. Extensible operating systems can easily adapt to satisfy the requirements of new emerging applications. The growing importance of extensibility is evidenced by the numerous projects focusing on designing new extensible operating systems [Bershad et al. 1995; Engler et al. 1995; Ford et al. 1996; Seltzer et al. 1994], or implanting extensibility in existing ones [Fitzhardinge 1996; Ghormley et al. 1996].

In this article we present a new method for extending existing operating systems entirely at the user level. We achieve most of the benefits of an extensible operating system without having to modify the existing OS or designing a new one. In our approach, extensions can be installed and run by an individual user without the help of a system administrator and without affecting other users. One advantage of this method is that extensions become very simple to install and use. Another major benefit is that each user can choose what extensions to run, in effect creating a personalized operating system view for him or herself.

We demonstrate this method for operating system extension by building a significant extension—the Ufo[1] global file system extension that installs and works completely at the user level. The original motivation for this extension came from the recent explosive growth of the Internet that gave an increasing number of users, including us, access to multiple computers that are geographically distributed. Our desire was to have transparent file access from our local Unix machines to our personal accounts at remote sites. In addition, we also wanted to present resources from the large number of existing HTTP and anonymous FTP servers as if they were local to allow applications to transparently access remote files.

Ufo is based on the Catcher, our tool for extending a standard Unix operating system (Solaris) completely at the user level. The Catcher extension approach, which is similar to Interposition Agents [Jones 1993], uses standard tracing facilities to intercept selected system calls. The behavior of intercepted system calls is then modified to implement new functionality.

While this article focuses on extending the file system services, our approach provides a general way of expanding the operating system. The main advantages of this approach are that extensions

—install and work completely at the user level,

—require no modifications to the kernel,

—require no recompilation or relinking of existing applications,

—require no modification of standard shared libraries,

—can be dynamically “installed into” and “uninstalled from” already running processes,

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

—can be added on a per-user and per-process basis, and

—can be developed at the user level without access to OS source code.

Like all user-level extension methods, our approach has certain limitations. First, some OS extensions are not possible to implement due to the fact that there is hidden kernel state inaccessible from the user level. Second, there is a performance overhead associated with system call interception which may be too big for some extensions. Last, the extensions do not work for setuid programs due to the security policy enforced by the operating system.

In this article we aim to examine the benefits of this method of extending the OS and to evaluate the performance of the implemented extensions in the context of remote file systems. We also give a good insight into the range of possible extensions by implementing a large OS extension for remote file access (Ufo) and suggesting other possible extensions.

The remainder of the article is structured as follows. Section 2 discusses the significance of the user-level approach for Ufo and gives a high-level overview of the OS extension mechanism. Section 3 reviews related work and compares our method for operating system extension with alternative approaches. Section 4 gives a detailed description of the Catcher and explains how it intercepts system calls at the user-level. Section 5 discusses the design decisions of Ufo, the user-level global file system. Section 6 presents experimental results for a variety of microbenchmarks, standard Unix file system benchmarks, and full application programs. Section 7 concludes this article and offers an outlook on future research directions.

2. UFO OVERVIEW

Ufo implements a global file system that allows local applications to transparently access files on remote machines. It is a user-level process that runs on 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.

2.1 User-Level Motivation

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 an easy task 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.

2.2 Personalizing the Operating System: The Catcher Approach

In order to provide a global file system, we extend the operating system to handle file accesses (and related functions) properly. By modifying the behavior of 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.

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 Sun Solaris 2.5.1 uses the System V /proc interface,[2] which was originally developed for debugging purposes [Faulkner and Gomes 1991]. 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 allowed to proceed unmodified, 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 analy-

[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`.

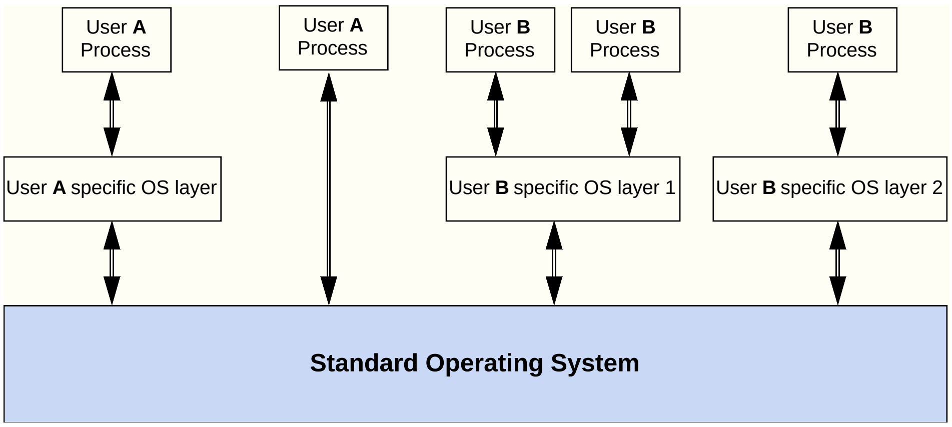

Fig. 1. A new view of the operating system.

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

2.3 Using Ufo

Ufo can be installed by any user without 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, cd, 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
```

3. RELATED WORK

Before presenting implementation details of the Catcher (in Section 4), 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 4. 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.

3.1 Approaches for Extending the Operating System

There has been a considerable amount of work on extending operating systems with new functionality. We can divide the research in two main groups: designing new extensible operating systems and adding extensibility to existing operating systems. There are multiple research projects on designing new extensible operating systems, some of which are mentioned in Section 3.2.

Here we discuss approaches for extending mainstream monolithic operating systems. We can classify these approaches into the following categories:

—*Change the 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 reiterate that these first three approaches require superuser 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 these libraries. Superuser 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.

Table I. Different Methods of Extending Operating System Functionality and Examples

Method	Examples and References
Change the OS	Sprite [Nelson et al. 1988], Plan 9 [Pike et al. 1990]
Device Driver	AFS [Morris et al. 1986], NFS [Sandberg et al. 1985], SLIC [Ghormley et al. 1996], and WebFS [Vahdat et al. 1996]
Network Server	ftp2nfs [Gschwind 1994], Alex [Cate 1992]
User-Level Plug-Ins	extended OS: SLIC [Ghormley et al. 1996], UserFS [Fitzhardinge 1996] flexible/extensible OS: SPIN [Bershad et al. 1994], Exokernel [Engler et al. 1995]
Statically Linked Library	Newcastle Connection [Brownbridge et al. 1982], Prospero [Neuman et al. 1993], Condor [Tannenbaum and Litzkow 1995]
Dynamically Linked Library	Jade [Rao and Peterson 1993], IFS [Eggert and Parker 1993]
Application Specific	Ange-ftp [Norman 1992]
Intercept System Calls	Interposition Agents [Jones 1993], Confinement [Goldberg et al. 1996], Ufo

Table II. Different Methods of Extending Operating System Functionality and Their Limitations

Method	Modify OS Source	Needs Root Access	Recompile Appli- cations	Relink Appli- cations	Range of Appli- cations	Perfor- mance Overhead
Change OS	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

—*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 to alter their behavior. This mechanism, which first was used in the context of Mach to implement interposition agents [Jones 1993], forms the basis for our Ufo implementation.

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

3.2 Related OS Extensions

The project that is the closest to our own is the work on interposition agents [Jones 1993] 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 harder 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 [Goldberg et al. 1996] 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 that allow for easier and more efficient user-level extension. The microkernel approach is to remove functionality from the kernel and place it in user-level servers. This reduces the complexity of the kernel and allows for easier extensibility, since changes to the user-level servers do not affect the kernel. Engler et al. [1995] propose to implement the operating system as a set of untrusted application libraries. In their approach the kernel is reduced to a very small exokernel that is only responsible for securely exporting the hardware resources to the library operating system. Fluke [Ford et al. 1996] combines the microkernel approach with recursive virtual machines [Goldberg 1974]. This produces a modular and extensible operating system which can be decomposed both horizontally (as in microkernel OS) and vertically through stackable virtual machine monitors. Another approach, taken by VINO [Seltzer et al. 1994] and SPIN [Bershad et al. 1995], is to allow injection of user-written kernel extensions into the kernel domain. A discussion of the issues involved can be found in Seltzer and Small [1996]. SLIC [Ghormley et al. 1996], another recent project, is an OS extension to Solaris that allows for plug-ins at both the user and the kernel level.

Lastly, while we extend OS functionality by interposing at the user-level/OS boundary, interposition is also possible at other places. For example, Disco [Bugnion et al. 1997] interposes a virtual machine monitor layer between the OS and the machine hardware. This layer allows multiple independent operating systems to run concurrently on a single large-scale shared memory multiprocessor without large implementation overhead.

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

3.3 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 (network file system) [Sandberg et al. 1985], AFS (Andrew file system) [Morris et al. 1986], Coda [Satyanarayananan et al. 1990], ftpFS in Plan 9[Pike et al. 1990] and in Linux [Fitzhardinge 1996], Sprite [Welch 1991; Nelson et al. 1988], WebFS [Vahdat et al. 1996], Alex [Cate 1992], Prospero [Neumann et al. 1993], and Jade [Rao and Peterson 1993]. They all have one significant drawback: 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 [Neumann et al. 1993] and Jade [Rao and Peterson 1993] both provide access to NFS and AFS file systems, and to FTP servers. Prospero runs at the user level by replacing standard statically linked libraries. This avoids changes to the operating system, but requires relinking of existing binaries. Jade [Rao and Peterson 1993] 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 libraries 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 [Vahdat et al. 1996], 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 [Ghormley et al. 1996], which implements a call-back 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 [Fitzhardinge 1996] 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 [Pike et al. 1990] 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 [Cate 1992] supports read-only access to anonymous FTP servers, while ftp2nfs [Gschwind 1994] additionally allows read and write access to authenticated FTP servers.

Last, but not least, web browsers such as Netscape and Internet Explorer are universally used to access global files through the HTTP and FTP protocols. One can view these browsers as implementing a global files system. This file system, though, is application specific, providing services to a single application—the browser itself. Ufo on the other hand can provides file system services to any application.

4. 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.

4.1 The Ufo Architecture

Ufo is a user-level process that 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.

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 and 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.

4.2 Catcher Implementation Details

In our Solaris implementation, the Catcher monitors user processes using the /proc virtual file system [Faulkner and Gomes 1991]. 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

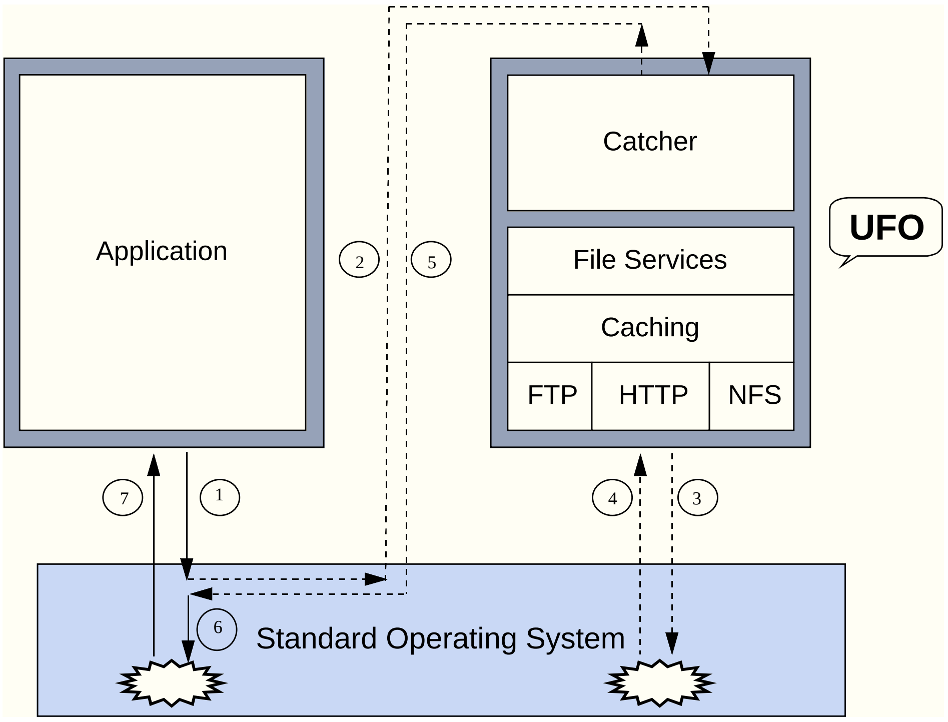

Fig. 2. General architecture of Ufo.

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 having a process identifier *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. Eventually, when the subject process terminates or is killed, the Catcher detects this and stops tracing the process. Figure 3 summarizes how the discussed functionality is used in the Catcher.

Conceptually, Ufo implements the system calls intercepted by the Catcher, but in practice Ufo does not service them directly. The /proc interface requires that the intercepted system calls always go through the kernel. It is possible to "abort" a system call on its entry to the kernel, intercept its exit form the kernel, and have Ufo service the call. We chose

```
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
```

Fig. 3. Outline of the Catcher algorithm.

not to do this in Ufo, in order to avoid reimplementing much of the existing OS functionality. Instead, we "patch" the system calls by (1) modifying the call's parameters, (2) changing the file system state (e.g., fetching a file from a remote server), and (3) 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 file name in place, since the new file name might be longer than the old one. Also, the file name 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 returned file handle, 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.

4.3 Catcher Discussion

The Catcher mechanism allows us to create a personalized operating system. We can reinterpret requests made to the kernel, allowing individual users to run their own private 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 also allows to intercept and act on signals and hardware faults. This allows for an even wider range of extended OS functionality to be implemented using the Catcher mechanism.

As with any user-level approach, there are limitations to the range of possible extensions. First, there is a considerable amount of hidden kernel state that is inaccessible from the user level. Second, intercepting system calls introduces an overhead. In the remainder of this section, we assert that although the limitations are not negligible, the Catcher is a valuable and flexible tool with small overhead for most applications.

The lack of access to internal kernel state prevents us from implementing certain time-critical extensions, such as a modified scheduler. However, a wide range of extensions to the Catcher approach is a great implementation alternative. Some sample applications are encrypted or compressed file systems, confined execution environments for running untrusted binaries [Goldberg et al. 1996], virtual memory paging [Dahlin et al. 1994; Feeley et al. 1995], and process migration [Tannenbaum and Litzkow 1995].

Surprisingly, the Catcher overhead proved relatively small when amortized over the execution of a whole application. Although we saw significant performance drops for specialized file service benchmarks under Ufo, which experience a slowdown of as much as 10 times, most real-life applications, even disk-intensive ones like latex or make, run with a moderate overhead of not more than 5%–29%. We believe that most applications for the Catcher can amortize the cost for intercepting systems calls or page faults over the total running time of a process, since typical applications issue relatively few system calls, and for most extensions, not all system calls need to be intercepted.

Other potential concerns with the Catcher mechanism are the execution of setuid programs and dealing with the premature demise of the Catcher process. Again, we found that, in practice, these limitations are not posing any additional burden to the user. Setuid programs always run under the identity of the owner of the executable file rather than under the identity of the user that starts them. The Catcher cannot control setuid processes, 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 as setuid, and those few, e.g., rlogin, do not really require the file system extensions of Ufo. In the current implementation, whenever the Catcher detects that a subject process is about to execute a setuid program, it simply stops tracing the process. 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. However, this is not an newly introduced limitation—for example, a process using an NFS-mounted volume is also bound to block if the NFS daemon or the remote server goes down.

The last issue of concern is the portability of the Catcher mechanism and of the extensions built on top of it. Fortunately, most Unix operating systems provide at least a core of the Solaris /proc functionality to port most of the Catcher's functionality. Even when the full /proc functionality is not available, we can still port the Catcher and extensions that use it, albeit with slightly reduced range of functionality. For example, the current Ufo implementation requires a certain amount of functionality from the Catcher and therefore from the /proc interface (or from its equivalent). This functionality includes the ability to (1) intercept the entry and exit of system calls; (2) read and modify the system call arguments and return values; (3) read from, and write to, the subject process' address space. The Solaris /proc interface provides all this functionality; other operating systems may have restrictions on (2) and (3). This is important if the Catcher needs to change some system call arguments such as file name strings for Ufo. If we are to port the Catcher to other operating systems that do not allow writing into the subject process, we would not be able to implement this feature. We can still port the basic functionality of Ufo, but with some restrictions. Features such as a 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 Section 5.1). The resulting Ufo will be somewhat less convenient to use, but nevertheless fully functional.

In summary, the Catcher does have its limitations. We can certainly come up with a number of applications for which the use of Catcher is not a wise choice. However, the Catcher is flexible enough to be an excellent implementation tool for a wide range of applications.

5. 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, transferring files, and caching.

5.1 Naming Strategies

Ufo supports three ways of specifying names of remote files: (1) through a URL, (2) through a regular file name implicitly containing the remote host, user name, and access mode, and (3) 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 file name. The general syntax is `/protocol/user@host/file name` 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 anonymous FTP server `ftp.cs.ucsb.edu`. The user can also specify mount points 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 mount point:

```
local /http method HTTP
```

Similarly to Sprite [Nelson et al. 1988], we have implemented mount points using a prefix table which, given a file name, 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. This both increases the complexity and the performance cost of file name resolution in Ufo.

5.2 Accessing Remote Files and Directories

Similarly to other file systems, such as AFS [Morris et al. 1986], 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 do not 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 services 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 reimplement 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

[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.

stubs in it. Then, it redirects the system call to the so-created skeleton directory.

5.3 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 an *open*, and whether we can delay the write-back of a modified file on a *close*. To efficiently support a wide range of usage patterns, Ufo provides an adjustable consistency policy based on timeouts (a read and a write delay). The policy guarantees that (1) when a file is opened it is no more than T_{read} seconds out-of-date; and (2) 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 given in seconds. The read timeout can optionally be specified as a percentage of the file's age as in Alex [Cate 1992]. 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 trade-off 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.

5.4 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 the protection of cached files. Each Ufo user has a distinct private cache. To ensure that others cannot gain undesired access to a user's cached files, Ufo creates the topmost cache directory with read and write permissions for the owner only.

5.5 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 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 mount points in user-unwritable areas such as the root directory.

Transferring only whole files introduces three well-known problems for extremely large files [Cate 1992]. 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 do not expect these two problems to occur frequently. With the exception of databases, most applications tend to access files nearly in their entirety [Baker et al. 1991]. 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 [Cate 1992]). The drawback is

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

Table III. Run Times for Various System Calls for Accessing Files in /tmp. The numbers are the arithmetic mean of 5 runs, each executing 100 iterations. The Standard OS numbers are in microseconds, and the remaining columns show times relative to Standard OS, e.g., 7.54 means 7.54 times slower.

System Call	Standard OS	Catcher Only	Ufo Local File	Ufo Remote Cached	Ufo Remote (no cache)
open	28 μsec.	7.54	21.8	85.1	18964
close	12 μsec.	9.00	25.0	128	37667
stat	33 μsec.	4.03	14.3	52.8	6000
getpid	3 μsec.	1.67	1.67	1.67	1.67
write 1 byte	23 μsec.	1.13	1.13	1.13	1.13
read 1 byte	25 μsec.	1.12	1.12	1.12	1.12
write 8K	97 μsec.	1.04	1.04	1.05	1.04
read 8K	75 μsec.	1.04	1.04	1.05	1.05

that intercepting *read* and *write* calls incurs a high overhead and requires extra implementation effort.

6. 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 143MHz Sun Ultra 1 workstation with 64MB of main memory running Solaris 2.5.1.

6.1 Microbenchmarks

The microbenchmark results present the user-perceived run times (measured as wall-clock times) for the *open*, *close*, *stat*, *read*, *write*, and *getpid* system calls. The results are shown in Table III. 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 noncached 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 which has a resolution of about 0.5 microseconds on the 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 *stat* system calls executed by the benchmark program, and lets them continue immediately without modifying them. The *open* system call is intercepted both on entry to, and on exit from, the kernel; *close* is only intercepted on exit; and *stat* is only intercepted on entry. This explains why *open* incurs about twice as much overhead as *close* and *stat*. 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 one-byte blocks this overhead is small and for 8K blocks is negligible. Because *getpid* is so fast, it has a substantial relative overhead, but still only 2 μsec. 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. The files are located in the /tmp system and have five name components. 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 and from bookkeeping tasks for open files. For system calls that reference a file (e.g., *open*, *stat*), 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 file name can involve a number of *stat* system calls, similar in flavor to the pwd command, and this can add a noticeable overhead. In addition the *open* and *close* (but not *stat*) system calls have to update Ufo's internal table of open files.

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 531 msec. using FTP. Closing the same remote file after modifying it takes 452 msec., 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.

6.2 File System Benchmarks

Table IV reports the absolute execution times in seconds for two file system benchmarks run on the local file system with and without Ufo and on a remote FTP-mounted file system with and without caching. The local tests were run on the /tmp file system in order to factor out the overhead of our NFS file system and to provide the fastest possible execution times. The FTP host was a machine on the local 100Mbit/s Ethernet network. The remote tests with caching used 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 call 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 and handles. The *Iostone* benchmark [Park and Becker 1990]

Table IV. Run Times for the Iostone and Andrew File System Benchmark Programs with and without Ufo. The Standard OS numbers are in seconds. The remaining columns show times relative to Standard OS (e.g., 8.33 means 8.33 times slower). The first two columns show the total number of system calls executed by the application and of system calls intercepted by Ufo. The number of system calls per second is given in parentheses.

Benchmark	System Calls Total	System Calls (Calls/sec.)	System Calls Intercepted Total	System Calls Intercepted (Calls/sec.)	Standard OS (in sec.)	Ufo Local File	Ufo Remote Cached	Ufo Remote No Cache
iostone	99203	(33068)	48762	(16254)	3	8.33	34	2581
andrew								
makedir	641	(—)	465	(—)	0*	—	—	—
copy	9596	(3199)	6999	(2333)	3	2.00	2.33	22.7
scandir	8674	(1446)	4115	(686)	6	1.17	1.67	10.2
read all	12907	(1291)	6084	(608)	10	1.30	1.60	6.80
make	8407	(701)	3107	(259)	12	1.17	1.33	1.83
Total	40225	(1341)	20770	(692)	30	1.33	1.67	10.7

* 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.

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 [Howard et al. 1988] measures five stages in the generation of a software tree. The stages (1) create the directory tree, (2) copy source code into the tree, (3) scan all the files in the tree, (4) read all of the files, and finally (5) 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 surprising, since the network latency and the FTP protocol overhead are quite large compared to the fast accesses in /tmp.

6.3 Application Programs

We also tested Ufo with a number of larger Unix applications: *latex*, *spell*, *latex2html*, *make* of Ufo, *ghostscript*, and the integer applications from the SPEC95 benchmark suite. The results are given in Table V and Figure 4 graphically shows the results for a subset of the benchmarks. 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; *spell* checks the spelling of 16 Latex files; and *latex2html* converts a Latex document to HTML format. *Make* compiles Ufo using *g++*. The *ghostscript* test displays a 20-page PostScript document. The table shows that *latex*, *spell*,

Table V. Run Times for Some Larger Unix Applications. The Standard OS numbers are in seconds, and the remaining columns show times relative to Standard OS (e.g., 1.24 means 1.24 times slower). The first two columns show the total number of system calls executed by the application and of system calls intercepted by Ufo. The number of system calls per second is given in parentheses.

Application	System Calls		System Calls Intercepted		Standard OS (in sec.)	Ufo Local File	Ufo Remote Cached	Ufo Remote No Cache
	Total	(Calls/sec.)	Total	(Calls/sec.)				
latex	7638	(588)	4396	(338)	13.0	1.24	1.35	6.92
spell	3168	(1320)	810	(338)	2.4	1.17	1.21	4.58
latex2html	53569	(1145)	14523	(310)	46.8	1.29	1.76	5.01
make	22783	(735)	6762	(218)	31.0	1.22	1.24	3.38
ghostscript	5495	(1340)	167	(41)	4.1	1.05	1.07	1.37
0.99.go	131	(0.16)	48	(0.06)	833	1.05	1.06	1.10
124.m88ksim	161	(0.34)	18	(0.04)	469	1.00	1.01	1.08
126.gcc	21234	(61.91)	1008	(2.94)	343	1.01	1.01	1.06
129.compress	40	(0.12)	14	(0.04)	344	1.01	1.01	1.00
130.li	250	(0.53)	60	(0.13)	476	1.00	1.02	1.05
132.ijpeg	1050	(2.17)	45	(0.09)	484	1.00	1.00	1.03
134.perl	6019	(13.50)	42	(0.09)	446	1.02	1.00	1.03
147.vortex	8490	(14.10)	27	(0.04)	602	1.00	1.00	1.01

latex2html, and *make* perform a relatively large number of system calls that Ufo intercepts, mainly *open*, *close*, and *stat*. This results in Ufo overheads between 17% and 29%, when run locally, and higher overheads, 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 overhead, the user may choose to run only part of the applications under Ufo. For example, the authors achieve this by configuring their X Windows environment to start a terminal window under Ufo and another without Ufo. All future applications started from the Ufo-enabled terminal window will have access to the remote file system services, while the applications from the other terminal window do not use Ufo and thus incur no overhead. Ufo also provides the option for the user to manually detach from running applications should the need for this arise.

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

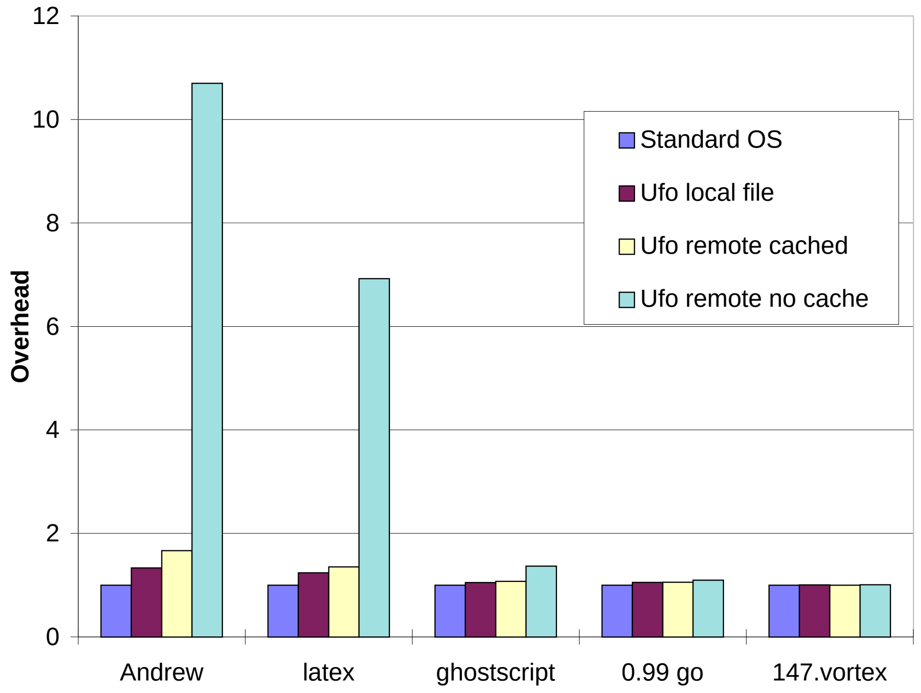

Fig. 4. Performance overhead for some of the benchmarks.

notice the difference between 28 μsec. and 611 μsec. when opening a local file.

6.4 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 (Table IV)—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 by far outweighs the Catcher and Ufo overhead as shown in Table III. In this case, Ufo proves to be a convenient tool. Furthermore the user can choose to run only a portion of his or her applications under Ufo by setting up his or her X Windows environment as explained before. Applications that need access to remote files can have it, and the remaining processes will not incur any overhead. An additional benefit from the fact that Ufo uses system call interposition is that the user can have Ufo dynamically attach to a running process and detach from it. This gives the user extra flexibility

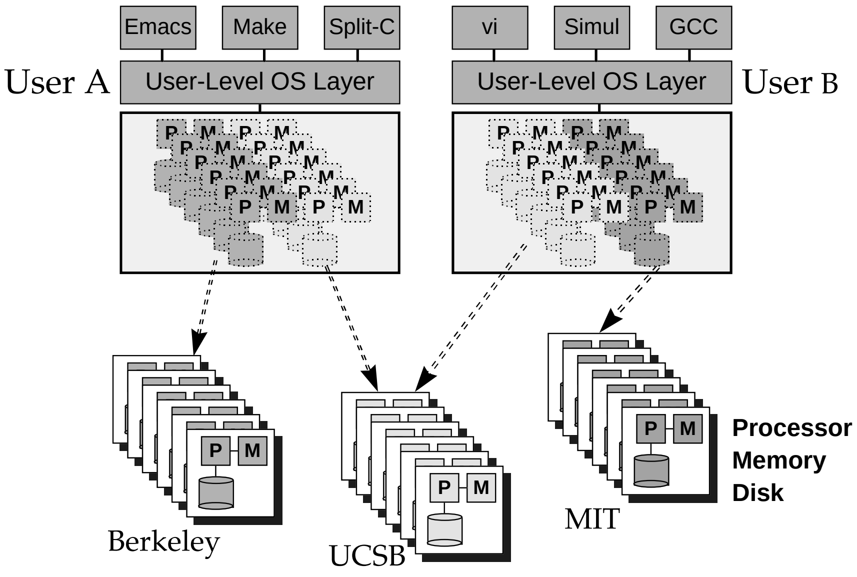

Fig. 5. Personalized user-level OS extensions for accessing a variety of remote resources.

unavailable with alternative methods for user-level OS extension (e.g., modified standard libraries).

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 file name check in Ufo (to determine whether a file is local or remote) alone will result in a significant reduction of the running time. For example, we expect to reduce the overhead for the *open* system call from 611 to below 400 microseconds.

7. CONCLUSIONS AND FUTURE WORK

In this article, 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 on a daily basis. 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 recompiled or relinked. 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. Similarly, 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. By combining these extensions, one could build a personalized OS layer that provides transparent access to a variety of remote resources such as CPU time, memory, and file systems (Figure 5). With the Catcher method, different users can choose to have different personalized views of the operating system by running different OS extensions.

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 [Goldberg et al. 1996], 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.

ACKNOWLEDGMENTS

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

REFERENCES

Baker, M. G., Hartmann, J. H., Kupfer, J. H., Shirrif, K. W., and Ousterhout, J. K. 1991. Measurement of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (SOSP '91, Pacific Grove, CA, Oct. 13–16). ACM Press, New York, NY.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. 1995. Extensibility safety and performance in the SPIN operating system. *ACM SIGOPS Oper. Syst. Rev. 29*, 5 (Dec.), 267–283.

Brownbridge, D. R. and Marshall, L. F. 1982. The Newcastle Connection, or UNIXes of the world unite!. *Softw. Pract. Exper. 12*.

Bugnion, E., Devine, S., and Rosenblum, M. 1997. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles* (SOSP '97, Saint-Malo, France). ACM Press, New York, NY.

Cate, V. 1992. ALEX—A global file system. In *Proceedings of the 1992 USENIX File System Workshop* (Ann Arbor, MI, May). USENIX Assoc., Berkeley, CA.

Dahlin, M., Wang, R., Anderson, T., and Patterson, D. 1994. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation* (Monterey, CA, May). USENIX Assoc., Berkeley, CA.

Eggert, P. R. and Parker, D. S. 1993. File systems in user space. In *Proceedings of the USENIX Winter 1993 Technical Conference* (Berkeley, CA). USENIX Assoc., Berkeley, CA.

Engler, D. R., Kaashoek, M. F., and O'Toole, J. 1995. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Oper. Syst. Rev. 29*, 5 (Dec.), 251–266.

Faulkner, R. and Gomes, R. 1991. The process file system and process model in UNIX system V. In *Proceedings of the 1991 Winter USENIX Conference*. USENIX Assoc., Berkeley, CA.

Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M., and Thekkath, C. A. 1995. Implementing global memory management in a workstation cluster. *ACM SIGOPS Oper. Syst. Rev. 29*, 5 (Dec.), 201–212.

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

Ford, B., Hibler, M., Lepreau, J., Tullman, P., Back, G., and Clawson, S. 1996. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (OSDI '96, Seattle, WA, Oct.). ACM Press, New York, NY.

GHORMLEY, D. P., PETROU, D., AND ANDERSON, T. E. 1996. SLIC: Secure loadable interposition code. Tech. Rep. CSD-96-920. Computer Science Department, University of California at Berkeley, Berkeley, CA.

GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. 1996. A secure environment for untrusted helper applications—Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*. USENIX Assoc., Berkeley, CA.

GOLDBERG, R. P. 1974. Survey of virtual machine research. *Computer 7*, 6 (June).

GSCHWIND, M. K. 1994. FTP—Access as a user-defined file system. *ACM SIGOPS Oper. Syst. Rev. 28*, 2 (Apr.), 73–80.

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 51–81.

JONES, M. B. 1993. Interposition agents: Transparently interposing user code at the system interface. *ACM SIGOPS Oper. Syst. Rev. 27*, 5 (Dec.), 80–93.

MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S., AND SMITH, F. D. 1986. Andrew: A distributed personal computing environment. *Commun. ACM 29*, 3 (Mar.), 184–201.

NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. 1988. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 134–154.

NEUMANN, B. C., AUGART, S. S., AND UPASANI, S. 1993. Using Prospero to support integrated location-independent computing. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*. USENIX Assoc., Berkeley, CA.

NORMAN, A. 1992. Ange-ftp manual. Free Software Foundation, Cambridge, MA. ftp://alpha.gnu.ai.mit.edu/ange-ftp/ange-ftp.tar.gz

PARK, A. AND BECKER, J. C. 1990. IOStone: A synthetic file system benchmark. *SIGARCH Comput. Arch. News 18*, 2 (June), 45–52.

PIKE, R., PRESOTTO, D., THOMPSON, K., AND TRICKEY, H. 1990. Plan 9 from Bell Labs. In *Proceedings of the UK Unix Users Group*.

RAO, H. C. AND PETERSON, L. L. 1993. Accessing files in an internet: The Jade File System. *IEEE Trans. Softw. Eng. 19*, 6 (June), 613–624.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementaiton of the Sun network file system. In *Proceedings of the Summer USENIX Conference*. USENIX Assoc., Berkeley, CA.

SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4 (Apr.), 447–459.

SELTZER, M. AND SMALL, C. 1996. A comparison of OS extension technologies. In *Proceedings of the 1996 USENIX Technical Conference* (San Diego, CA, Jan.). USENIX Assoc., Berkeley, CA.

SELTZER, M., ENDO, Y., SMALL, C., AND SMITH, K. 1994. An introduction to the VINO architecture. Tech. Rep. TR34-94. Harvard Univ., Cambridge, MA.

TANNENBAUM, T. AND LITZKOW, M. 1995. The Condor distributed processing system. *Dr. Dobb's J. 20*, 2 (Feb.).

VAHDAT, A., DAHLIN, M., AND ANDERSON, T. 1996. Turning the Web into a computer. Tech. Rep. Computer Science Department, University of California at Berkeley, Berkeley, CA.

WELCH, B. B. 1991. Measured performance of caching in the Sprite network file system. *Comput. Syst. 3*, 4.

Received: May 1997; revised: January 1998; accepted: March 1998