Rapid File System Development Using `ptrace`

Richard P. Spillane, Charles P. Wright, Gopalan Sivathanu, and Erez Zadok

Stony Brook University

Abstract

Developing kernel-level file systems is a difficult task that requires a significant time investment. For experimental file systems, it is desirable to develop a prototype before investing the time required to develop a kernel-level file system. We have built a `ptrace` monitoring infrastructure for file system development. Because our system runs entirely in user-space, debugging is made easier and it is possible to leverage existing tested user-level libraries. Because our monitor intercepts all OS entry points (system calls and signals) it is able to provide more functionality than other prototyping techniques, which are limited by the VFS interface (FUSE) or network protocols (user-level NFS servers). We have developed several example file systems using our framework, including a pass-through layered file system, a layered encryption file system, and a user-level ISO9660 file system. We analyzed the complexity of our code using cyclomatic complexity and other metrics. We show savings for a pass-through file system of 53% compared to existing user-level pass-through file systems and a factor of 4.7 reduction for an in-kernel pass-through file system. Our performance evaluation demonstrates that our infrastructure has an acceptable overhead of 18.4% for a pass-through file system.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Complexity measures, Rapid prototyping, Files, Monitors, Modules and interfaces, Elicitation methods

General Terms

Experimentation, Management, Design

Keywords

Rapid Prototyping, Monitors

ExpCS, 13-14 June 2007, San Diego, CA.

1 Introduction

File system prototyping is difficult: developers currently have to struggle with a massive body of complex kernel-level code to test their ideas. Attempts have been made to address this issue, but most of them lack the added flexibility and expressiveness that kernel developers often require (e.g., being able to modify how the VFS caches inodes). This forces some developers to resort to prototyping in the kernel, and to cope with weaker debugging tools, lengthy reboot cycles, and the inability to capitalize on existing user level libraries.

We created a user-level file system development environment using a `ptrace` [9] *monitor*: a program that uses `ptrace` to manage resources shared between multiple processes. Our development environment is more powerful and convenient for developers than existing user-level frameworks. We call our monitor *Goanna* (after the Australian genus of monitor lizards). The `ptrace` process-tracing facility, commonly used by debuggers, allows us to intercept OS entry points (i.e., system calls, signals, and more) and insert customized code before, after, or as a replacement for the kernel-level implementations—all without the process's knowledge. The underlying OS kernel handles all process management, memory management, the networking, and other core OS facilities. We have followed the Unix tradition of using a single name-space, so that new file systems and existing file systems can co-exist. System calls destined for paths that Goanna does not handle are simply passed on to the underlying kernel. Thus, developers can focus only on their specific extension code. Moreover, because developers are writing user-space code, a plethora of libraries and debugging tools that do not work in the kernel are available.

Goanna's `ptrace` monitor framework significantly improves the experience which kernel developers have by providing the following three advantages:

Rapid development

There are many user-level libraries that perform useful tasks for file systems (e.g., the Berkeley DB provides transactional B-trees and hash tables [20]). In user space, developers also have a wide variety of good debugging techniques at their disposal. A good prototyping framework should leverage these technologies as much as possible. Unfortunately, user-level libraries are not suitable for the kernel, and debugging the kernel is difficult. Some kernel debuggers are available, but they are cumbersome to use, and they tend to change timing conditions enough to make it more difficult to debug complex problems; for these reasons, many developers often choose to insert print statements instead. These problems, combined with the raw complexity of most commonly used OS kernels, introduce debugging difficulties that almost never occur in user space. Because a `ptrace` monitor runs entirely in user-space, it can link with existing user-space libraries and interoperate with user-level debugging tools.

Powerful prototypes

A prototyping framework should allow developers to change significant aspects of the system. Other file system frameworks such as FUSE [22] or NFS toolkits [15] that address rapid development, fall short here. For example, FUSE can only intercept VFS operations and is unaware of system calls like `fork`. Many research ideas for which a prototyping framework would be useful can not afford to be limited to a predetermined VFS interface. For example, if special properties inherited accross processes are required, instrumentation of `fork` would be crucial. A `ptrace` monitor addresses this issue by intercepting more of a process's entry points (i.e., all of the signals and system calls). Thus, for traced processes, a monitor is able to provide most of the kernel's functionality.

Modularity

A prototyping framework should allow developers to write code only for those systems they wish to change, without modifying other systems. An OS kernel is a large piece of software, with many interacting components. For example, the Linux virtual file system (VFS) has more than 48,000 lines of code, but if it is changed in any significant way, then the developer must update dozens of file systems, the memory management system, and more. Not only is there a large amount of code, but the code is also very complex. Frameworks that enable kernel development in user space (such as UML [3]) also suffer from this lack of modularity. In a monitor, the developer begins with a clean slate. Goanna has only 6,429 lines of code, and consequently it is simpler to understand and, if necessary, rewrite any of its components. Also, we designed Goanna such that prototyping a file system is rather easy by including a simple file-system switch that operates at the system call level.

The major objection to using a user-level monitor rather than kernel development for prototyping is that to get better performance, one must reimplement the system after the prototyping stage to deploy it. This is often true, but we feel that mitigating the opportunity cost associated with prototyping a new file system is worth the amount of work that may be lost when transitioning to a deployable system. There are also many file systems where performance is not critical. For example, an SSH file system performs many network and cryptographic operations, which would dominate any performance degradation caused by running it in user-level. Moreover, the amount of work to be redone should be small. Most of the design and algorithms should translate easily to the kernel. Excepting references to user-level libraries or modifications to the monitor itself, the code can also be reused. Finally, when one considers that most systems (especially in research) never make it past the prototype phase, it becomes evident that a more efficient means of prototyping file systems is desirable.

User-level monitors are also useful beyond prototyping. For example, browsing the contents of an ISO 9660 CD-ROM image would normally be done by creating a loopback device and mounting it. However, it is unsafe to allow non-root users to mount arbitrary devices; so users cannot easily browse these images. Goanna provides a solution: users can virtually mount any ISO image they like by running any program (e.g., a shell) through Goanna (which does not require root privileges). This improves the system's security, because the user does not require any root privileges to browse the image. Also, using a `ptrace` monitor to allow new kernel-like functionality to be offloaded to user space increases reliability of the kernel code base.

When we compared the lines of code, number of tokens, number of identifiers, and the McCabe complexity of file systems developed with our framework vs. those developed with FUSE, user-level NFS servers, or the kernel, we found our file systems were less complex.

The rest of this paper is organized as follows. In Sections 2, 3, and 4 we discuss the current state of the art in file system prototyping, an overview of Goanna's design, and three example file systems developed in Goanna, respectively. In Section 5 we evaluate Goanna's complexity and performance. Sections 6 discusses related work, and we conclude in Section 7.

2 Background

To evaluate prototyping frameworks, we developed a taxonomy that includes the three criteria from Section 1 (rapid prototyping, powerful prototypes, and modularity) with design transferability, compatibility, and performance. We used these six dimensions to evaluate six common prototyping frameworks: the kernel itself, User Mode Linux (UML) [3], FUSE [22], a user-level NFS server toolkit [15], an `LD_PRELOAD` library, a modified C library [13], and our `ptrace` monitor.

User Mode Linux runs the Linux kernel in user space by using `ptrace`. UML's primary goal is to provide a mechanism to run GDB on the kernel. FUSE is a hybrid approach that routes VFS calls to a user-level daemon so developers can implement a file system with the traditional VFS interface in user space. User-level NFS toolkits allow developers to construct their prototype as NFS servers that run in user space. The `LD_PRELOAD` facility allows C library functions, including system call wrappers, to be overridden. A modified C library [13] can be used in much the same way as the `LD_PRELOAD` option. Table 1 summarizes our qualitative estimates on how each alternative meets the six criteria. Often, improving one criterion comes at the expense of others; therefore, no single framework is suitable for all circumstances.

Rapid development

A prototyping framework should enable developers to use external libraries and debugging tools to speed up development. The kernel cannot link against user libraries, and is the most difficult to debug. UML can be debugged in user space, but it cannot link with user libraries. Similarly, a modified C library would have circular dependencies if it links against a user library that in turn links back to the C library (which almost all user libraries do). Conversely, FUSE, user-level NFS toolkits, an `LD_PRELOAD` approach, and a monitor framework all use the standard user runtime environment, so they can easily use user-level libraries and be debugged.

Powerful prototypes

A prototyping framework should impose as few limitations as possible. In-kernel development is the most powerful prototyping framework for file system development, because any aspect of the system's behavior can be changed. Some examples requiring a prototyping framework more powerful than a VFS interface are:

	Rapid Development	Powerful Prototypes	Modularity	Design Transferability	Runtime Compatibility	Performance
In-kernel file system	★	★★★	★	★★★	★★★	★★★
UML	★★	★★	★	★★★	★	★
FUSE	★★★	★	★★	★★ ½	★★★	★★
User-level NFS	★★★	★	★★	★★	★★	★
LD_PRELOAD	★★★	★★	★★	★	★	★★ ½
Modified C library	★★	★★	★	★	★	★ ½
`ptrace` monitor	★★★	★★	★★★	★★	★★	★

Table 1. We evaluated each framework according to our six criteria. Each row represents a framework, and each column represents a criterion. The number of stars within a column can be qualitatively compared, but the number in different columns have no direct relation to each other.

- *Process Inheritence*: A file system which requires that one process inherits vital attributes from another process would have to instrument `fork`. One such application would be in a file system that supports mandatory locks where children of a process can optionally be given permission to access files locked by their parent.
- *Recovery Actions*: A file system that allows a process to perform system-condoned tasks before it exits would require instrumentation of the `exit` call. One application of this would be to allow a process to run a system tool that undos its changes to the file system if the process exits with a non-zero code.
- *Security Tracking*: A file system that associates important security metadata with each process would need to be aware of `fork`, `exec`, and `exit`. To ensure that absolutely all exit points are caught (e.g., being killed by the Out of Memory or OOM killer in Linux), a small kernel patch would be required [24].

In order to provide the most general operating system prototyping, the broadest possible interface to the operating system must be instrumented, and the system call layer is broader than the VFS layer. UML runs in the user-level, and therefore it cannot fully control its environment (e.g., its scheduling policies are limited by those of the host OS). FUSE is even more inflexible than UML as it forces the developer to interface with the kernel strictly through the VFS interface, making it impossible to catch process-spawn events, manipulate caches (e.g., the inode, dentry, and buffer caches), and more. For example, an access-control layer could chose to grant access based on the PID. Because FUSE file systems cannot intercept process termination, they cannot reliably revoke a process's access when it exits [24]. A user-level NFS toolkit is limited in a similar way to FUSE, though it is limited by the NFS protocol rather than the VFS interface. The monitor can intercept all calls made by the process, and the `LD_PRELOAD` and modified C library frameworks can intercept most calls (but not in-lined assembly).

Modularity

A prototyping framework should limit the amount of code that developers must learn and change. This is especially true for components that developers are not interested in changing.

The `ptrace`-based monitor framework is the most modular of all the options since it is least dependent on limited protocols or interfaces; it allows developers to override only those system calls they intend to work on; and developers can override these calls without any knowledge of the kernel. The next most modular frameworks are FUSE and NFS toolkits, because they prevent developers from crossing clearly defined interfaces. However, neither of these approaches are as modular as a monitor framework, because developers must implement all of the protocol's methods. The `LD_PRELOAD` library-based approach is also not as modular as a monitor framework because developers must implement not only I/O-related system calls (e.g., `write`), but also I/O-related library calls like `fprintf`. The C library is worse than FUSE, the NFS toolkit, and the `LD_PRELOAD` library, because the code for most C libraries is large, and is not designed with intercepting calls to the kernel in mind. Thus as with UML and in-kernel development, the prototype would be tightly integrated with the library.

Design transferability

A prototype's design and algorithms should be applicable to the deployed system with minimal changes. The code of a prototype developed in the kernel or using UML, need not change for deployment. FUSE is almost as transferable as UML because it asks developers to design their extensions with an interface similar to the kernel's VFS. NFS servers are less transferable than FUSE because the NFS protocol is not as close to the VFS as FUSE's protocol. The monitor framework is similar to NFS, in that the system call interface is similar to the VFS interface, but not quite as close as FUSE's protocol. The `LD_PRELOAD` and modified C library techniques force developers to structure their code in a different way than what the kernel would require, which has a non-trivial impact on their design.

Runtime compatibility

A prototype should be capable of interfacing correctly with any program. A kernel prototype fully satisfies this criteria. One specific application of UML as a runtime environment is running untrusted services while mitigating the risk to the physical hardware [3]. Using UML as a runtime environment for more general usage suffers from the separation between the host and guest kernels: they cannot share the same name space or memory, and must use the network to interact. In comparison to UML, FUSE is highly compatible with applications because the kernel routes all VFS events to the FUSE daemon. The NFS toolkit approach is compatible with programs that are not affected by network anomalies, such as latency; so it is more compatible than other approaches, but not as compatible as FUSE or the kernel itself. The monitor framework has similar limitations: it cannot run set-UID programs or those that execute `ptrace` on other processes, but it is otherwise compatible with all programs and does not require any recompilation of their source code. The `LD_PRELOAD` technique also does not require recompilation of programs, but programs that do not use the C library or are statically linked cannot be run with the new file system. A modified C library requires relinking of programs so programs that make system calls directly cannot be supported.

Performance

The performance of the deployed system should be as good as possible. This is obviously true of kernel prototypes, because kernel code can use in-kernel caches and avoid context switches and data copies. FUSE adds the overhead of user-level context switches when servicing VFS events, but because FUSE uses the in-kernel caches, the number of data copies is reduced. Overall, FUSE comes the closest to a kernel-level file system in terms of performance. UML and the monitor both use `ptrace`, which adds context switches and data copies. Similarly, an NFS toolkit runs over the network subsystem incurring data copies within the network stack. The `LD_PRELOAD` and C library run in the address space of the process, so they do not incur as much overhead for context switches and data copies as other user-level techniques. However, they introduce additional function calls and must call the kernel to perform meaningful work. Moreover, they behave sufficiently differently than the in-kernel VFS, that estimating the potential performance improvements from porting such code to the kernel is more difficult.

It is clear that no single solution is the answer to all prototyping needs, because each prototype could have different requirements. However, a `ptrace`-based monitor is best suited for circumstances where significant OS changes are required, yet rapid development and modularity are desired. Though all the user-level approaches try to support rapid development, they make trade-offs that weaken the power of their prototypes or their modularity—whereas the monitor framework provides these criteria while not overly sacrificing design transferability, compatibility, or performance. For example, FUSE and NFS toolkits trade prototyping power for improved modularity by strongly enforcing the VFS interface or the NFS protocol, respectively. On the other hand, UML forces the developer to contend with a large body of kernel code in exchange for a more flexible framework. The `LD_PRELOAD` approach is competitive with a monitor, but it is not as modular and it sacrifices more in terms of transferability and compatibility.

3 Design

Our `ptrace`-based monitor named *Goanna* (after the genus of Australian monitor lizards) intercepts and modifies a process's system calls and signals [9]. From the perspective of the application, Goanna is equivalent to the OS, so no application modifications are required. As shown in Figure 1, Goanna runs at user-level, so file system prototypes do not need to execute within the kernel. Unlike an `LD_PRELOAD` or C library, a single instance of Goanna can handle multiple processes, so it is simpler to share data, caches, and other resources.

The major disadvantage of the `ptrace` approach is that performance may suffer for system-call–intensive programs, as more context switches are required for each system call. The criterion of performance is often not as important as rapid development for a prototype, so this is an acceptable trade off, especially for file systems with superseding bottlenecks (e.g., the network).

In the remainder of this section, we describe the design of Goanna. In Sections 3.1, 3.2, 3.3, and 3.4 we explain Goanna's system-call interception, structure, process handling, and path resolution. Address space issues are discussed in Section 3.5. In Sections 3.6, 3.7, and 3.8 we describe our `mmap` design, file system switch, and performance enhancements.

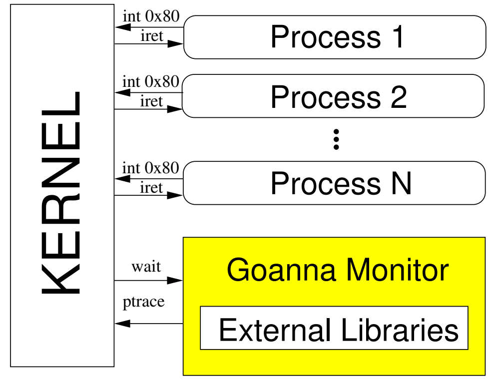

Figure 1. At system call entry, the kernel signals Goanna via the `wait` system call. Goanna manipulates the monitored processes' state using `ptrace` primitives. Libraries execute within Goanna's address space and use standard system calls.

3.1 Process Tracing Primitives

Goanna begins by forking a new child. The child issues `PTRACE_TRACEME` to request interception, and then the child executes the to-be-traced executable. From this point onward, Goanna is notified via the `wait` system call whenever the child needs attention.

Many debuggers like GDB use `ptrace`, so its primitives are similar to those provided by a debugger. Goanna allows the child process to fork another child and so on. Each child will be monitored by Goanna. The monitor can use three primitives to control the execution of the child process:

- `PTRACE_SYSCALL` continues execution until the next entry or exit from a system call. The child is stopped before entering or after leaving the kernel, and the parent is notified each time.
- `PTRACE_CONT`, analogous to `continue` in GDB, continues execution until the child receives a signal.
- `PTRACE_SINGLESTEP`, analogous to `nexti` in GDB, continues until the next instruction.

When the child is in the stopped state, a monitor can use four primitives to observe and manipulate the child process: `PTRACE_GETREGS`, `PTRACE_SETREGS`, `PTRACE_PEEKDATA`, and `PTRACE_POKEDATA`.

`PTRACE_GETREGS` retrieves the values of the process's registers. On the Intel 80x86 architecture, the `eip` register contains the program counter, the `eax` register indicates what system call the process wants to execute, and the remaining general-purpose registers contain the system call's arguments. Goanna's implementation is tied to the 80x86 architecture, because it references these registers, but it is simple to add support for other architectures because the application binary interface, or ABI is similar on all Linux platforms. In our reference prototype, only 411 out of 6,429 lines of code reference 80x86-specific registers.

A monitor can manipulate registers with the `PTRACE_SETREGS` primitive. Before a system call, the call to execute can be changed by setting `eax`, and the arguments can be changed by updating the corresponding registers. After a system call is executed, the return value can be set by updating the value of `eax`. At any point in time, the execution flow of the program can be changed by modifying `eip`. This is required when a single system call must be implemented in terms of several other system calls that have to execute in the user process's address space.

Finally, there are two primitives to examine and update a word in the child process's memory: `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`. These primitives are used when the system call takes pointer arguments (e.g., file names are passed as strings, and `stat` fills in a user-supplied buffer).

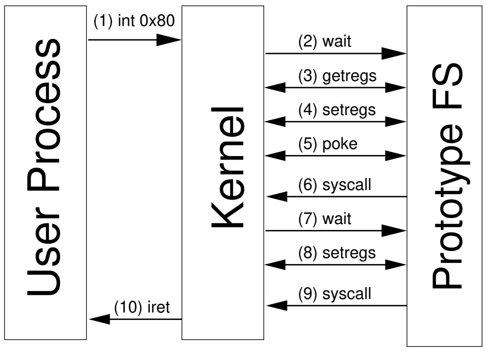

Figure 2. `ptrace` primitives used to handle a `read` system call. Arrows indicate control transfer. Double arrows indicate that the function was called and returned immediately.

Figure 2 shows an example of how Goanna handles a `read` system call, destined for a file stored in a prototype file system, on behalf of a user process. There are ten steps involved in this call:

1. The user process issues a system call by storing the call's number in `eax` and invoking the system call trap `0x80`.
2. The `wait` system call in the monitor returns the user process's PID.
3. Goanna issues a `PTRACE_GETREGS` call to retrieve the value of `eax`. Based on `eax` and the call's arguments, Goanna can determine how to treat the call. This is useful for cases where developers want to intercept only a subset of calls that are relevant to their file system.
4. If this call is to be intercepted, then Goanna changes the registers to prevent the kernel from handling the call. To nullify the kernel call, Goanna sets `eax` to –1; the kernel consequently ignores the call because no handler is associated with –1.
5. Goanna executes the prototype's `read` operation, and uses `PTRACE_POKEDATA` to write the returned data into the user process's address space (we describe an optimization in Section 3.5).
6. Goanna instructs the kernel to continue execution until the end of the call and calls `wait`. In this case the call returns immediately without performing any service, because `eax` was set to –1 in step 4.
7. The kernel skips the call, and returns from `wait`.
8. Goanna uses the `PTRACE_SETREGS` primitive to store the return value of the previously executed read in `eax`
9. Goanna uses the `PTRACE_SYSCALL` primitive to allow the user process to continue executing.
10. The kernel issues an `iret` instruction to return control to the user process. The user process reads the return value from `eax`, and it is as if the system call was serviced by the kernel.

3.2 Monitor Structure

In Goanna, execution begins by forking a child process to trace. After the fork, the child executes the program to be monitored. All of the process's descendants are also monitored, and each monitored process is assigned a state. The two most common states are INUSER and INCALL, which indicate that the process is executing user-level code or that it is executing a system call, respectively. To service requests, Goanna calls the `wait` system call. When a process requires attention, usually because it is entering or exiting a system call, the kernel returns its ID as the result of the `wait` call (`wait` also returns when a signal is delivered or a process exits).

After returning from `wait`, Goanna retrieves the current process's state and performs an appropriate action. There are currently 19 states (including INUSER and INCALL). Most of the states indicate that the user process is in the midst of a specific call, (e.g., `clone`, `exec`, `chdir`, or `dup`), and allow Goanna to remember what it was doing before it called wait. One of the most important states is INFORCERET, which indicates that the return value of the presently executing system call should be overridden by a given value. This is useful if the prototype file system needs to pass back status information. In the example in Section 3.1, the return value of the `read` is determined in step 5, but is not yet returned. When the return value is determined in step 5, Goanna sets the state to INFORCERET. After step 7, Goanna looks up the state and because its state is set to INFORCERET, it can determine that it must return a value; Goanna then sets the value of `eax` to the proper return value. Two other states of note are REDOCALL, which indicates that the current system call should be repeated, and RESTOREREGS, which indicates that the process's registers should be set to their original values. REDOCALL allows us to insert a new system call into the stream (e.g., to create shared memory regions in the user process's address space), and RESTOREREGS is used when we need to change system call arguments (e.g., when rewriting file names).

3.3 Process Control Blocks

Goanna maintains each process's state in a private *process control block* (PCB). This allows Goanna to map file descriptors to open files for a particular process, record the current working directory, track mount points, and store other process-specific metadata. Goanna's PCB is independent of the OS PCB, and contains the process ID to use as a search key, a copy of the process's registers, the current state of the process (e.g, INFORCERET), and all state-specific information (e.g., the return value to be passed back to the application). Encapsulating all of this information in a single structure allows Goanna to handle concurrent processes.

Goanna needs to map file descriptors to files in order to intercept system calls that identify files by their file descriptors (e.g., `fstat`). As such, like an OS PCB, Goanna's PCB contains an open-file table and stores the present working directory (PWD). The open-file table is a simple array with a slot for each possible file descriptor. If a given file descriptor is connected to a file that is to be handled by the prototype file system, then its slot contains a pointer to a structure describing the file; otherwise it is empty (`NULL`). If a system call uses a file descriptor as an argument, it is looked up in the open-file table. If the file descriptor's slot is empty, then the system call proceeds with no further intervention. Otherwise, Goanna uses information stored in the structure describing the file to proceed according to the intended semantics of the prototype file system (e.g., an encrypting file system may encrypt the data before writing it to disk on a write call).

An interesting problem with open is that a monitor cannot arbitrarily assign file descriptors to the user-level process, because the kernel would not know that a given file descriptor is in use. To handle this situation, Goanna uses *shadow descriptors*. When opening a file in a prototype system, Goanna changes the path name to "/" before letting the system call proceed. The resulting file descriptor (in the child process) is used as a place holder, and no system calls are issued against it. The kernel does not assign the resulting descriptor to any other file, so Goanna can correctly identify the calls that it handles. For efficiency, Goanna reuses this file descriptor with dup on subsequent open calls.

3.4 Mount Subsystem

To determine which instance of a file system is being operated on in an intercepted system call, Goanna maintains a mount table. This mount table associates pathnames with different operations vectors and instance-specific data (e.g., the device to read and write from). On startup, Goanna reads a configuration file that provides a list of paths to manage, and for each path, the mount type and data (the configuration file is essentially equivalent to /etc/fstab). When Goanna intercepts a system call that references one of these paths, Goanna passes it to the appropriate routine.

Pathnames passed to system calls can be rather complex. If they are relative path names, then they depend on the process's context. Any path can use the ".." operator to move one level up the directory tree. We store paths as stacks, with the root path represented as an empty stack, and a path such as /usr/local/bin is represented by a stack containing usr, local, and bin. If a path is managed by Goanna, then it is a child of one of the mount-table entries described in the configuration file. To rapidly determine if one path is a child of another, the path structure also contains a depth and a length for each path component.

Each PCB contains a path stack for the PWD. When a chdir or fchdir system call is issued, the new PWD is stored as a candidate. If the system call is successful, then the candidate becomes the PWD. The mount table also uses a path stack to identify the path for each mount.

To resolve a path that is passed to a system call, first the process's PWD is copied to a new stack. If the path begins with a "/," then the stack is emptied. Each subsequent component is pushed onto the stack. If the component is ".."," then an element is popped off (unless the stack is already empty). After converting the string pathname into a path stack, Goanna searches the mount table for any mount that contains this path. The path structure is optimized for this purpose: if the path has a lower depth than the mount, then it cannot be a child; and the length is stored with each component so the component names need to be compared only if they have equal length. If one is found, then the path components after the root of the mount are extracted; for example, if the path is /usr/local/src/linux and the mount is rooted at /usr/local, then src/linux is extracted. Any mount-specific data associated with the file name is passed to the file system (e.g., the device it is stored on). If the path name is not contained in a mount, then Goanna allows the system call to go through unchanged.

3.5 Address Spaces

There are two distinct address spaces involved in executing a monitor: the address space of the monitor and the address space of the user process. An important issue for performance is that the ptrace primitives to access the user process's address space are rather limited—they can examine or change only one word at a time (PTRACE_PEEKDATA in Section 3.1). Thankfully, Linux provides a more flexible interface through the /proc file system. A process with permission to ptrace another process may read from the traced process's memory using the /proc/*pid*/mem file, where *pid* is the PID of the traced process. This allows the transfer of up to a page (1,024 words on the 80x86) in a single system call. Linux also has support for writing to /proc/*pid*/mem, but it is disabled by default. For our prototype, we enabled a writable /proc/*pid*/mem to allow bi-directional bulk transfers. If the /proc/*pid*/mem interface is not available for reading or writing, then Goanna falls back to PTRACE_PEEKDATA and PTRACE_POKEDATA. Allowing regions of the child's address space to be memory-mapped into a monitor, thus providing a zero-copy transfer method, would be more efficient.

Another issue is that all system call arguments need to be in the user process's address space. For example, the first argument to open is a pointer to a string. If a monitor needs to update these values, then it must manipulate the child's address space. It is not always possible to manipulate the file name in place, because the new file name may be longer than the existing file name, and the memory segment may be read only. To address this issue, previous ptrace monitors have modified either the stack, or the first writable segment. In Goanna, we establish a System-V shared-memory region between each user process and Goanna. When the first system call is issued with an argument that needs to be updated, Goanna creates a shared memory region. Next, Goanna inserts a shared-memory attach operation into the child's system-call stream. At this point, Goanna writes the new file name into its own address space, and updates the child's registers to point to the shared memory in the child's address space. After the call, the child's original registers are restored. Subsequent arguments can be rewritten by simply updating the local region and the child's registers. This approach has the advantage of requiring no data copies. Also the child's existing memory is not modified, therefore the child's memory need not be restored after the call.

3.6 Memory-Mapped Operations

Many applications (e.g., linkers) take advantage of memory-mapped operations, which allow access to the file system through an efficient memory-like interface. Therefore, supporting mmap is essential to providing good compatibility and design transferability. Goanna provides support for memory-mapped operations by intercepting the mmap system call and any SIGSEGV signals that are delivered to a monitored process.

Upon intercepting an mmap system call destined for one of its file systems, Goanna behaves much like an OS kernel: it establishes an empty region and services page faults for that region. To create the empty region, Goanna converts the process's mmap system call to an anonymous region that the process is not allowed to read or write to. Goanna also records the address of the memory-mapped region and its backing file in the process's PCB.

Normally, the OS handles the page faults through a hardware trap triggered by the MMU. Goanna handles page faults through a software trap. Whenever an application accesses an invalid page (either because it does not have permission or the page does not exist), the OS sends it a SIGSEGV signal. Before a monitored application receives the signal, Goanna examines the signal information including the address that faulted. If Goanna finds the address in that PCB's memory-mapped region list, then Goanna reads the page into the process's address space. Next, Goanna issues an `mprotect` system call in the context of the application to allow the process to read the page. If the address is not found, then a `SIGSEGV` signal is delivered to the process, usually resulting in a core dump.

When the region that was read into the process's address space is written to, a second page fault is generated. Goanna marks the page's state as dirty. It then allows the process to change the page (using `mprotect`). Unfortunately, the signal information structure informs us only that an access violation occurred; it does not inform us whether the requested access was a read or write. If we had this extra piece of information, we could reduce the number of traps into the monitor that are required for memory-mapped writes. On `munmap` or `msync`, Goanna writes dirty pages to the backing file. Although Goanna does not currently write dirty pages in other circumstances, it would be possible to create a separate thread for flushing dirty pages (analogous to Linux's `pdflush` or FreeBSD's `vm_pageout`).

3.7 File System Switch

Goanna uses a file system switch (a.k.a., a virtual file system or VFS). The VFS is responsible for resolving path names and then passing the operation down to an appropriate file system. The major advantage of a VFS is that hard-coded function calls are avoided, thus allowing multiple file systems to be easily developed. Moreover, the VFS provides a convenient boundary between Goanna's core (e.g., `ptrace` primitives) and a file system. Our experience developing layered file systems has given us unique insight into VFS design from the perspective of extensibility [26]. VFSs today either encapsulate quite a bit of functionality within the VFS itself, making it difficult to extend, or they include too little generic functionality, making it difficult to develop file systems.

Like a standard VFS, our system uses operations vectors to direct intercepted calls to the appropriate method. Our VFS differs in that it has several levels of methods, which may call one another. The highest level of methods correspond to system calls. To provide functionality within the VFS, generic methods can be assigned to these operations. For example, a file system need only implement one `write_internal` method to write data to a file. The `write` and `writev` operations can then be handled by the `generic_write` and `generic_writev` methods. Those methods in turn call the `write_internal` method. Using generic operations for some operations is not new, but our VFS differs in that no functionality is actually built into our VFS: it is all delegated to generic methods. This is important because the generic methods may not be suitable for all file systems. For example, a transactional file system must start a transaction at the beginning of `writev` and commit it at the end (so that earlier writes do not affect the file system if later writes fail). As the `generic_writev` method does not know about transactions, it is unsuitable for a transactional file system. In our architecture, a transactional file system can implement its own `writev` method without being constrained by existing VFS functionality.

Our design also allows for efficient implementation of layered file systems. For file systems that simply add functionality to an existing lower-level file system, it is possible to add a new layer without introducing method calls for operations handled by the lower-level file system. For example, an encryption file system does not affect the `fchown` operation, so the file-system switch directly calls the lower-level `fchown` operation.

3.8 `ptrace` Enhancements

The standard `ptrace` interface requires at least six context switches for each system call:

1. The traced process traps into the kernel.
2. The kernel transfers control to the monitor.
3. The monitor transfers control to the kernel.
4. After executing the system call, the kernel transfers control back to the monitor so that the return value can be manipulated.
5. The monitor transfers control back to the kernel.
6. The kernel transfers control to the traced process.

In reality, more context switches are required as the monitor must retrieve the values of traced process's registers, issue system calls to provide OS-like services, etc.

Clearly, reducing the number of times that the monitor is called improves performance. For most calls, Goanna needs to be notified only on entry. If the call is not destined for a monitored prototype file system, Goanna does not need to examine the return value so the call could execute without further intervention by Goanna. If the call is handled by the prototype file system, the return value could be set and the monitor need not be notified. Unfortunately, these two modes of operations are not possible under the current `ptrace` interface.

We created two new `ptrace` operations: `PTRACE_CHECKEMU` and `PTRACE_SYSSKIP`. The `PTRACE_CHECKEMU` operation is similar to the `PTRACE_SYSEMU` operation that was recently introduced to improve the performance of User Mode Linux [3]. The primitive `PTRACE_SYSEMU` allows all of a process's system calls to be emulated, but it is not suitable for Goanna, because we emulate only a subset of the system calls. Our `PTRACE_CHECKEMU` interface allows Goanna to determine whether emulation is required after examining the registers (the UML developers agree that our more general `PTRACE_CHECKEMU` interface is an improvement over the existing `PTRACE_SYSEMU` [5]). The corollary to `PTRACE_CHECKEMU` is `PTRACE_SYSSKIP`. When Goanna does not implement a call, it issues `PTRACE_SYSSKIP` instead of `PTRACE_SYSCALL` to bypass notification of this system call's return value and goes directly to the start of the next system call. Together, these primitives reduced traps into Goanna by 30.8% during an OpenSSH compile.

Finally, there are also many non-file-system system calls that Goanna need not intercept at all (e.g., `time` or `getpid`). To reduce the number of extraneous calls into the monitor, we added an optional bitmap of system calls to the task structure. By using a new `ptrace` primitive, `PTRACE_SELECT`, Goanna selects precisely the set of calls that need to be traced. This method reduced the number of traps to Goanna by an additional 12.8% during an OpenSSH compilation. Overall, these techniques reduced the number of traps to Goanna by 43.7%.

Our improvements can benefit a wide variety of `ptrace` monitors. For example, `PTRACE_CHECKEMU` grew out of work for User Mode Linux, but provides a more flexible interface that can be used by a monitor that emulates a subset of system calls. Many security-oriented monitors need to examine only which system calls are being executed and their arguments, but not their return value. For these types of monitors, `PTRACE_SYSSKIP` would greatly improve their performance. The `strace` program provides support for filtering the set of system calls to display (e.g., file system, process, or IPC- related calls), but this filtering is done in user-space. By using `PTRACE_SELECT`, `strace` could have the kernel perform this filtering more efficiently.

4 Example File Systems

In this section we describe the design and implementation of three file systems that we created using Goanna. In Section 4.1 we describe a simple pass-through layer that handles file system operations by passing them down to another directory. This pass-through layer serves as the basis for our AES encryption file system described in Section 4.2. In Section 4.3 we describe a user-level ISO file system, which allows users to browse CD-ROM images. In Section 4.4 we briefly describe our transactional file system called Amino.

4.1 Pass-Through Layer

We developed a simple pass-through file system layer for two reasons. First, it serves as an example for other file system extensions. We developed it in such a way that its operations could be reused for other file systems (e.g., the encryption file system described in Section 4.2). Second, it provides a suitable basis for evaluating Goanna's overhead (Section 5.2). The pass-through file system takes a single mount-time argument: the name of the directory to which operations should be redirected.

The pass-through file system implements 21 operations, 17 of which are simple wrappers around another system call. It also defines two new operations: `encodename` and `decodename`. File systems built on top of this pass-through layer can override these operations to manipulate file names. These methods translate upper-level file names to the corresponding lower-level names (and vice versa). A representative method of the pass-through file system is `unlink`, which has only three function calls: (1) the argument is converted to a lower-level name using `encodename`; (2) the lower-level name is unlinked; and (3) the lower-level name is freed. The `read`, `write`, and `lseek` methods are similar to the system-call-based wrappers, but take internal monitor objects (i.e., mount and open file structures) as arguments instead of operating at the ABI level. This allows the methods to be re-used for many types of system calls (e.g., the `read` operation is used for both the `read` and `readv` system calls in addition to memory-mapped reads). The last two methods are `open` and `close`, both of which wrap underlying system calls and manage monitor state (e.g., the open file structure).

4.2 AES Encryption Layer

We have developed an AES encryption file system on top of the pass-through layer described in Section 4.1. This encryption layer allows users to encrypt the contents of a directory, thereby preventing a breach of confidentiality if the hard disk is stolen.

Encryption scheme

Our file system layer encrypts both file names and file data. However, to simplify development and administration, we chose to preserve the existing structure of files by encrypting each file separately because users are used to dealing with a traditionally organized file hierarchy [2, 24]. This convenience, however, comes at the expense of revealing some information about the structure of the files (e.g., how many files exist in a given directory and their size). Several systems have made theses choice, including CFS [2], NCryptfs [24], and eCryptfs [10]. The data and names in our system are encrypted using a key that is read with `getpass` on startup.

We use a separate scheme for file name encryption and data encryption. For file names, we must to encrypt the parent directory name and a name within that directory. Each parent directory has an associated initialization vector (IV), which means that a file with the same name in two different directories does not encrypt to the same text. We chose to use the AES-CBC mode to encrypt file names. This has the disadvantage of causing the file name's length to be rounded up to the nearest cipher block size (16-bytes), but it is more secure: more malleable cipher modes (e.g., CFB and CTR) are inappropriate because they do not permit the reuse of an IV for different cipher texts. After the file name is encrypted, it is base-64 encoded so that illegal (i.e., "`/`" and "`\0`") or control characters, which can disrupt the user's terminal and confuse utilities, are not written to the file system.

For data we need a scheme that has four properties:

- Two different files with the same plaintext have different cipher text.
- Two different regions of the same file that contain the same plaintext have different cipher text.
- We can rewrite regions of the file with the same IV.
- Random access has a constant penalty.

The scheme we developed, inspired by Blaze's OFB/ECB hybrid [2], is a hybrid of AES-CTR and ECB mode that satisfies each of these properties. This scheme has the advantage over Blaze's that there is no need to store precomputed data; it supports arbitrarily large files; and we use a distinct random stream for each file.

Extended attributes

For each encrypted file we must store two pieces of information: its initialization vector and its actual size, because the file's size is rounded up to the nearest cipher block size. In both cases we use the extended attribute API supported by Ext2, Ext3, Reiserfs, and many other file systems [8]. Storing the data in the file itself would change the file size and expected performance characteristics (because data would no longer be block aligned).

Implementation

The encryption file system overrides a subset of the operations for the pass-through file system described in Section 4.1. The `mount` operation initializes the AES encryption and decryption keys and locks them in memory so that the OS does not write them to swap. The `unmount` operation zeros out the keys before freeing them. The encryption layer defines four generic VFS-like operations: `open`, `read`, `write`, and `lseek`. It retrieves the file's IV via the extended attribute interface. If the file does not yet have an IV, then a new one is generated. The `read` and `write` functions are more complex than the others because they must correctly handle I/O operations that are not aligned on the AES block size, forcing us to use padding.

The encryption layer implements two internal methods for the pass-through file system: `encodename` and `decodename`. The `encodename` method converts a decrypted file name (e.g., `/home/rick/goanna/paper.pdf`) to an encrypted file name. The `encodename` operation is used for `open`, `mkdir`, and other operations that take a pathname as an argument. Conversely, the `decodename` operation is used for directory-reading operations. It retrieves the IV of the parent, and then decrypts the name.

Finally, the encryption layer implements five system-call-level functions: `stat`, `fstat`, `truncate`, `ftruncate`, and `read`. The `stat` and `fstat` functions retrieve the file size using extended attributes. The `truncate` and `ftruncate` functions fill holes that could be created by sparse files, and align all truncate operations on AES block-size boundaries.

4.3 ISO9660 File System

CD-ROM images, also known as ISOs, are formatted according to the ISO9660 standard. ISOs are a convenient way of transferring large collections of files, such as Linux distributions, software backups, or even family photos. However, to access the files in an ISO, users must first mount it using a loop device. Unfortunately, mounting a file system requires root privileges. It would be possible to create set-UID programs to allow a user to mount ISO images, but even if developed securely, there is always a potential for bugs or misconfigurations that could compromise the security of the system.

To address this issue, we developed a user-level file system using Goanna, built around `libiso9660` from GNU `libcdio` [18]. Goanna's user-space nature allowed us to link against this library and leverage its 3,449 lines of already tested code.

Because ISO9660 file systems are read-only by their nature, we needed to implement only nine methods for this file system: `mount`, `unmount`, `open`, `close`, `read`, `lseek`, `fstat`, `getdents`, and `fcntl`. The most complex method was `read`. For `read`, much of the code complexity was caused by a limitation of the `libiso9660` library, which only allows 2KB-blocks to be read. To implement read efficiently, we wrote more code to avoid extra data copies for unaligned access.

4.4 Amino File System

The original application of Goanna was toward developing a transactional file system called Amino [25]. Amino allows user-level applications to group operations into transactions that satisfy ACID semantics. Amino is based on the user-level Berkeley Database [20], so user-level development allowed us to develop it rapidly. Because Amino must support transaction rollback for VFS caches (i.e., the inode, directory-name-lookup, and page caches), it would require invasive changes to the VFS if developed in the kernel. This would have required a large investment of time to begin our investigation into transactional file systems, and our initial results would have been significantly delayed. Moreover, because one of the major facets of our investigation required us to make changes to the VFS, it would not have been possible to develop Amino as a FUSE file system or a user-level NFS server. Amino is the most demanding of all the file systems—it has almost six times as many lines of code (6,173) as the encryption file system, requiring each of the three properties that Goanna's monitoring infrastructure was designed for: rapid development, modularity, and powerful prototypes.

5 Evaluation

We evaluated Goanna in two dimensions. In Section 5.1 we analyze how complex our file system extensions are compared to other methods of developing file systems. In Section 5.2 we evaluate Goanna's performance.

Method	LoC	Tokens	Identifiers	MC
Framework Implementation				
`ptrace`	**6,429**	**40,121**	**13,927**	**1,145**
Kernel	48,072	255,969	109,081	8,152
FUSE	8,481	46,051	17,772	1,338
NFS	18,480	103,960	42,734	2,726
Pass-Through File System				
`ptrace`	732	**3,948**	**1,403**	**127**
Kernel	6,079	34,612	14,146	599
FUSE	**706**	5,010	1,659	149
Cryptographic File System				
`ptrace`	**1,131**	**7,483**	**2,260**	**212**
Kernel	9,780	57,489	23,130	943
FUSE	2,396	19,297	7,468	423
NFS	1,556	8,981	3,663	251
ISO9660 File System				
`ptrace`	**539**	**2,700**	**994**	**88**
Kernel	3,769	22,158	8,666	616
FUSE	1,704	11,890	4,315	363

Table 2. We evaluated different types of file systems implemented using different frameworks according to four metrics. Bold entries are the smallest in their class. (LoC means Lines of Code; MC means the McCabe cyclomatic complexity).

5.1 Complexity Evaluation

We used four metrics to compare the amount of development effort different frameworks require to write pass-through, encryption, and ISO9660 file systems. For each type of system that we evaluated, Table 2 shows the number of lines, tokens, identifiers, and the McCabe [16] cyclomatic complexity. McCabe's metric is the most precise: it measures the number of linearly independent paths through a program. We used the C and C++ Code Counter (CCCC) [14] to compute the complexity of each function, and then summed the results.

Framework implementation

Goanna and the FUSE frameworks require a similar amount of development effort to implement. We chose the SFS toolkit [15] as an example of a user-level NFS server. It is more than twice as complex than either FUSE or Goanna, but it is tightly integrated with a simple pass-through file system, which we did not remove from its complexity metric. The kernel's VFS system is the largest framework by any metric. This is not surprising because it cannot rely on external libraries and includes caching, quota management, support for several binary formats, asynchronous I/O, and many other tightly integrated facilities. This tight integration means that a kernel developer has to be familiar with a large body of complex code to develop file systems.

Pass-through layer

Goanna's and FUSE's pass-through file systems have similar complexity: FUSE is 3% shorter, but has 27% more tokens, 18% more identifiers, and a 17% higher cyclomatic complexity. Although our pass-through file system is 26 lines longer, it has more functionality than its FUSE counterpart. Our file system transforms names before passing the operation down to the lower-level file system, which enables us to mount on any lower-level directory (FUSE is limited to "/") and build our encryption file system on our pass-through file system. When this additional functionality is removed from our file system, its cyclomatic complexity is reduced to 97 (or 53% less than FUSE's). The SFS toolkit provides a built-in loopback NFS server, but the toolkit itself is more complex than Goanna or FUSE and the corresponding pass-through file system put together. Wrapfs, a pass-through file system for the Linux kernel, has the highest complexity, because it must perform elaborate operations on reference counts and cached objects, and emulate much of the VFS's functionality.

Encryption file system

We compared Goanna to the in-kernel eCryptfs [10], FUSE's EncFS [7], and the encryption file system from the SFS toolkit. Our file system and the one from the SFS toolkit have similar complexity. This is as expected because both allow a file system layer to extend an existing pass-through layer. Because EncFS was developed in FUSE, where the interface is similar to the VFS's, developers had to implement a less abstract interface, and thus they had to implement more routines. EncFS originally had over 14,000 lines of code and a McCabe complexity of 1,323. Even when we removed all code related to configuration, abstract classes, header files used by C++, and specialized caching, we still found EncFS to be twice as complex as our file system. eCryptfs suffers from the same problems as Wrapfs (because it is essentially a modified copy). Thus it is twice as complex as the other file systems.

ISO9660 file system

We compared Goanna's ISO9660 file system to the kernel's and to the FUSE-based `fuseiso` [17]. Goanna's ISO9660 file system is 539 lines of code with a McCabe complexity of 88. The size and complexity of `fuseiso` was greater: 1,704 lines and a cyclomatic complexity of 423. This increase is for two reasons: (1) Goanna uses `libiso9660`, whereas `fuseiso` does not use any external libraries, so it has code for reading ISO9660 directories, and (2) FUSE requires its file systems to handle more VFS objects than our framework. The kernel implementation is larger than `fuseiso`. This is because it cannot use external libraries such as `libiso9660` or even system calls, so interfacing with the device it is mounted on is more complicated.

In sum, FUSE and the NFS toolkit fall short in prototyping power, but they compare favorably with Goanna in terms of rapid prototyping capabilities and modularity. However, when rapid prototyping, modularity, *and* powerful prototypes (Section 2) are required, a `ptrace`-based monitor framework is a better choice than either FUSE or NFS toolkits.

5.2 Performance Evaluation

Our testbed ran Fedora Core 4 with all updates as of July 19, 2005. All experiments were conducted on a dedicated 40GB Maxtor IDE disk. We compared our pass-through and encryption file systems to XFS because it is highly scalable [21] and has mature extended attribute support [8]. We compared our ISO9660 file system to the kernel's over the loop device. To ensure a cold cache, we remounted the file systems between each iteration of a benchmark. For all tests, we computed 95% confidence intervals for the mean elapsed, system, and user time using the Student-*t* distribution. In each case, the half-widths of the intervals for the elapsed and system times were less than 5% of the mean. We also compute wait time, which is elapsed time less CPU time. Wait time is mostly spent waiting on I/O, but can also be affected by the scheduler. We do not evaluate Amino in this paper, because it is outside the scope of this paper, and a thorough evaluation can be found in a separate technical report [25].

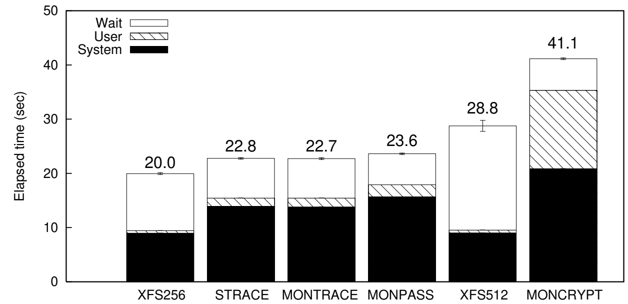

Figure 3. Postmark: 5,000 files and 20,000 transactions.

Pass-through file system

We used the following four configurations to evaluate our pass-through file system:

XFS256 XFS with the default 256-byte inodes, which is the basis for the following three configurations.

STRACE XFS monitored by `strace -cf`. This configuration shows the overhead of the `ptrace` facilities when used by a standard tool, but does not modify any system calls or produce any output during execution.

MONTRACE XFS monitored by Goanna. This configuration shows the overhead of `ptrace` and our path-name resolution infrastructure.

MONPASS Our pass-through file system.

We used Postmark v1.5 [12] to evaluate the performance of our system. Postmark is an I/O and system-call intensive workload that simulates a busy mail server by creating, deleting, reading, and writing to small files. We used the default parameters but increased the number of files to 5,000 and the number of transactions to 20,000, because the defaults do not exercise the file system sufficiently. Figure 3 shows the results of this experiment. XFS256 took 20.0 seconds to execute. It used 9.0 seconds of CPU time in the kernel (system time) and 0.5 seconds of user time. STRACE was 14.1% slower than XFS256; it consumed 55.4% more system time and 3.2 times more user time. This increase is caused by additional context switches and examining the application's registers. MONTRACE was similar to STRACE: it took 13.8% longer than XFS256; it used 54.1% more system time and 3.5 times more user time. Goanna used less system time than STRACE, because it retrieves the registers more efficiently. However, it used more user time, because it performs path name resolution. MONPASS took 18.3% longer than XFS256 and it took 74.6% more system time and 4.8 times more user time. The additional CPU overhead was caused by the additional data copies and system calls required to service requests. Our results show that Goanna is as efficient as a standard tool that uses `ptrace`, and that file systems can be prototyped with acceptable overheads even for system call and data-intensive workloads like Postmark.

Encryption file system

To evaluate our encryption file system, we used the configuration MONCRYPT, in which our AES encryption file system was layered over XFS formatted with 512-byte inodes; we call this configuration XFS512. The increased inode size allows extended attributes (EAs) to be stored directly in the inode block, thus improving EA performance. When no EAs are used, XFS512 is 44.1% slower than XFS256.

Figure 3 shows the results. MONCRYPT's elapsed time overhead was 106.2% over XFS256. This overhead is caused by three factors: (1) encryption increases the amount of user time by a factor of 31, (2) extended attributes must be written, and (3) more data must be read and written to ensure proper cipher block size alignment. These overheads are similar to what we have observed for Postmark running on CFS in the past, which runs in user-level and provides the same type of encryption (123–223%) [23, 24]. When MONCRYPT compared to XFS512, the elapsed time overhead is reduced to 43.1%.

ISO9660 file system

To evaluate the read-only ISO9660 file systems, we did not run Postmark because it modifies files and directories. Instead, we wrote a program to read all of the files from Fedora Core 4 (i386) Disc 1. There are a total of 628 files, which consume 650MB. We used two new configurations:

LOOPISO The kernel's ISOFS over a loop-back device.

MONISO Our ISO9660 file system.

The LOOPISO configuration ran for 11.2 seconds. Of this time, 2.1 seconds were spent in the kernel, 0.03 seconds were spent executing in user-space, and 9.1 seconds were spent waiting on I/O. The MONISO configuration ran for 23.6 seconds, or 110.0% longer than LOOPISO. The major cause of this increase was an increase in CPU time. System time increased to 12.84 seconds (a six-fold increase), and user time increased to 4.9 seconds. The user time is mainly the amount of time spent within the monitor. Clearly, there is a performance overhead for using a monitoring framework, but the rapid development and convenience of not requiring a kernel modification can offset this performance impact. Additionally, we are considering using zero-copy methods to improve performance for data-intensive workloads such as this one (see Section 7).

6 Related Work

The Ufo Global File system uses an interposition technique similar to Goanna's [1]. Ufo provides transparent access to remote files via FTP or HTTP. Ufo's monitor uses the Solaris `/proc` file system. Ufo operates only on system calls such as `open`, `close`, and `stat`. When an access to a remote file is detected, the file is transparently fetched, and the system call is changed to open the local copy. Creating a copy is suitable for small files, but is not appropriate for large files, those which are accessed randomly, or files that are shared. Ufo does not implement other calls such as `read`, `write`, `getdents`, or `stat` internally. Once the file is copied locally, Ufo relies on the existing file system's methods. Moreover, by relying on the existing OS to provide most file and directory operations, Ufo severely limits the types of file systems that can be developed. For example, Goanna's encryption file system must override `read`, and therefore could not be developed on Ufo. The existing methods also use the OS caches, which means that the BDB file system we developed with Goanna is not possible to develop with Ufo.

The Janus framework uses the `ptrace` interface to sandbox untrusted applications [6]. Janus monitors file-system and network-related system call invocations, and applies configurable policies to allow or deny system call execution. The `ptrace` interface has since been used for several other security monitors. For example, model-carrying code verifies that an application's sequence of system calls fits within a model [19].

Several other systems intercept system calls to provide new functionality, but do not use `ptrace` to do so. SLIC [4] is an OS extensibility system that allows kernel-level extensions or user-level servers to register handlers for system calls, signals, and other OS entry points. SLIC has been used to patch security holes, encrypt files, and provide a restricted execution environment. User-level SLIC extension servers are quite similar to the `ptrace` interface, but have two key disadvantages: (1) they must be trusted, and (2) SLIC uses self-modifying code for interception. Interposition agents provide higher-level abstractions for system call interception [11]. Jones stresses that interposition agents allow portable user-level extensions to existing system abstractions (i.e., pathnames, descriptors, files, users, etc.) that would normally need to be developed in the kernel.

7 Conclusions

We built a `ptrace`-based framework to enable rapid prototyping of file systems. Our user-level approach allows file system prototypes to leverage existing and time-tested user-level libraries, and use the powerful debugging facilities available in user space. Because `ptrace` allows us to intercept all system calls and signals, Goanna can override most OS functionality, enabling more powerful prototypes than are possible with FUSE or user-level NFS servers. Our approach is also highly modular: it can handle library calls, can rely on existing OS system calls, and developers need not be concerned with the large body of existing kernel code; The design of a file system developed with `ptrace` can be readily adapted to the kernel, as it operates at the traditional boundary between user-space and the kernel. This level of abstraction also provides a high degree of compatibility with user-level applications. Our performance evaluation demonstrates that user-level file systems can provide acceptable performance for prototypes. We have used `ptrace` to implement more file system functionality than existing solutions can. Goanna allows developers to modify all file system calls, including `read`, `write`, and even memory-mapped reads and writes. Goanna's VFS infrastructure is also an improvement over existing VFSs in that it allows the file system to extend the OS, beginning within the system call interface.

We developed three example file systems using Goanna. First, we developed a simple, yet extensible, pass-through file system. Second, we built a highly secure encryption layer using the pass-through file system as a basis. Third, we developed a user-level ISO9660 file system. We showed that file systems developed with our framework have comparable complexity to ones developed with FUSE and user-level NFS toolkits, but Goanna has a higher degree of power (see Section 4.4). We also show that each of these frameworks have less complexity than the kernel.

To allow other developers to benefit from our approach, we have released Goanna, the example file systems, and benchmarks described in this paper. They can be downloaded from `www.fsl.cs.sunysb.edu`.

7.1 Future Work

Currently, to use a Goanna file system, a process must be started through the monitor. We plan to create a simple framework that will allow processes to request an extension, or for users to attach an extension to an existing user-level process (e.g., by clicking on its window as in the `xkill` interface). A third option would be to run all of a user's processes through Goanna, and dynamically insert extensions based on the system call stream. For example, when a ZIP archive is opened, it could be made to appear as a directory. This mode would be convenient because users would not need to predefine a mount table in the Goanna configuration file.

The second aspect of our future work is to improve performance. We plan to further improve the `ptrace` interface. To reduce both the number of context switches and data copies between the kernel and Goanna we will: (1) use a shared-memory segment to manipulate the user process's registers so that data copies and context switches are reduced; (2) map regions from the user process's address space into Goanna's; and (3), where appropriate, bundle several `ptrace` operations into a single system call to reduce context switches (e.g., waiting for notification could be combined with retrieving registers). We may also port performance-sensitive subsets of Goanna to the kernel (e.g., path name resolution and file-table lookups). Finally, Goanna's implementation is currently single threaded (though many monitors can run concurrently). We plan to make Goanna multi-threaded to improve performance when a single monitor handles many processes.

Acknowledgements

We would like to thanks everyone who helped review drafts of this work, as well as the reviewers for the ExpCS 2007 workshop for their valuable comments. We would also like to thank the members of the File Systems and Storage Laboratory at Stony Brook for their helpful comments and advice. This work was made possible partially thanks to NSF award CNS-0614784 (CSR-PDOS).

8 References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 77–90, Anaheim, CA, January 1997. USENIX Association.

[2] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, 1993. ACM.

[3] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 63–72, Atlanta, GA, October 2000. USENIX Association.

[4] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 39–52, Berkeley, CA, June 1998. ACM.

[5] P. Giarrusso. Fwd: Re: [patch 1/4] UML Support - Ptrace: adds the host SYSEMU support, for UML and general usage, July 2005. `www.uwsg.iu.edu/hypermail/linux/kernel/0507.3/1992.html`.

[6] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *Proceedings of the Sixth USENIX UNIX Security Symposium*, pages 1–13, San Jose, CA, July 1996. USENIX Association.

[7] V. Gough. Encfs, November 2005. `http://arg0.net/wiki/encfs`.

[8] A. Grünbacher. POSIX Access Control Lists on Linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 259–272, San Antonio, TX, June 2003. USENIX Association.

[9] M. Haardt and M. Coleman. *ptrace(2)*. Linux Programmer's Manual, Section 2, November 1999.

[10] M. A. Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, pages 201–218, Ottawa, Canada, July 2005. Linux Symposium.

[11] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '93)*, pages 80–93, Asheville, NC, December 1993. ACM.

[12] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[13] D. G. Korn and E. Krell. A New Dimension for the Unix File System. *Software-Practice and Experience*, 20(S1):19–34, June 1990.

[14] T. Littlefair. *An Investigation into the use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Faculty of Communications, Health, and Science, Edith Cowan University, Mount Lawley Campus, June 2001.

[15] D. Maziéres. A toolkit for user-level file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 261–274, Boston, MA, June 2001. USENIX Association.

[16] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[17] D. Morozhnikov. FUSE ISO File System, January 2006. `http://fuse.sourceforge.net/wiki/index.php/FuseIso`.

[18] H. V. Riedel and R. Bernstein. GNU Compact Disc Input and Control Library. `www.gnu.org/software/libcdio/`, October 2005.

[19] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 15–28, Bolton Landing, NY, October 2003.

[20] Sleepycat Software, Inc. *Berkeley DB Reference Guide*, 4.3.27 edition, December 2004. `www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html`.

[21] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[22] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[23] C. P. Wright, J. Dave, and E. Zadok. Cryptographic File Systems Performance: What You Don't Know Can Hurt You.

In *Proceedings of the Second IEEE International Security In Storage Workshop (SISW 2003)*, pages 47–61, Washington, DC, October 2003. IEEE Computer Society.

[24] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[25] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. Technical Report FSL-06-01, Computer Science Department, Stony Brook University, January 2006. `www.fsl.cs.sunysb.edu/docs/amino-tr/amino.pdf`.

[26] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright. On incremental file system development. *ACM Transactions on Storage (TOS)*, 2(2):161–196, May 2006.