User-Level Extensibility in the Mona File System*

Paul W. Schermerhorn Robert J. Minerick
Peter W. Rijks Vincent W. Freeh
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
{pscherm1,rmineric,prijks,vin}@cse.nd.edu

Abstract

An extensible file system raises the level of file abstraction which provides benefits to both the end-user and programmer. The Modify-on-Access file system provides safe and simple user-defined extensibility through *transformations*, which are modular operations on input and output streams. A user inserts transformations into input and output streams, which modify the data accessed. Untrusted transformations execute in user space for safety. Performance of user-level transformations, although much slower than that of in-kernel transformations, is comparable to other user-level approaches, such as pipes.

This paper presents several interesting user-level transformations. For example, the `command` transformation executes a shell script whose input and output are routed from/to the file system. A file guarded by the `ftp` transformation is a "mount" point to an FTP server. The `php` transformation creates dynamic documents from PHP source when read. A file written to a sound device that is guarded by the `mp3` transformation is decoded on the fly, in the file system, before reaching the sound device.

Mona is a novel approach to file system extensibility that provides heretofore unseen flexibility. Mona is fine-grained: a user defines actions on a per-file basis. It is modular: transformations can be stacked upon one another. Mona supports two classes of transformations: kernel-resident and user-level.

1 Introduction

Unix-like operating systems have commonly used the file system to provide hardware abstractions for applications programmers. For example, Linux provides the devices `/dev/audio` and `/dev/dsp` (among others) to provide easy access to sound devices. Placing these abstractions in the file system allows programmers to simply write to a file rather than have to go through the difficult process of passing data to a kernel module. This is one of the tasks of an operating system—to facilitate commonly-performed operations. However, the semantic function of the file system has changed little over the years. File systems can do more than provide generic access to hardware devices. Raising the semantic level of the file system provides benefits to both the end-user and the programmer.

Performing common tasks at the file system level allows applications to be written at a higher level. For example, a file system that can decode an MP3 audio file is able to provide a *decoder* file to which an application writes a raw MP3 file, rather than decoding and writing it to a device file. This relieves application programmers of the responsibility to implement widely shared functionality. Furthermore, portability will be enhanced if the programmer does not have to rewrite these common operations for every target platform. End-users benefit by having more stable software (because upgrades to common operations can be achieved more easily at a single common point) and greater functionality (because applications will leverage common operations more readily). The Modify-on-Access (Mona) file system provides safe and simple extensibility, allowing file system extensions to provide much of the functionality common to many applications.

Mona is an extensible file system based on Linux's *ext2* file system [9]. Mona has recently been ported to the 2.4 series of the kernel from the 2.2 series. Mona allows users to associate actions, called *transformations*, with the input and output streams of files. These transformations operate on the data before it is passed on to the user process accessing the file. This technique of pushing operations out of the application and into the file system extends the capabilities of traditional file systems. Furthermore, transformations may be stacked upon one

*This research was supported by NSF CAREER grant CCR-9876073, the JPL HTMT Project, the Arthur J. Schmitt Fellowship, and the ND Faculty Research Program.

another to create complex functionality out of simpler components.

The Mona file system supports both kernel and user-level extensions. Many existing schemes to extend file system functionality focus solely on kernel extensions. While there are many cases in which kernel extensions are the most appropriate mechanism, for many operations user-level extensibility is the better choice. We enumerate several reasons below.

- First, programming in user-space is much easier than in the kernel. Commonly available libraries (e.g., *libc*) can be used just as they can in application programs, whereas few functions are available to the kernel programmer.
- Second, debugging is much easier in user space than in the kernel.
- Third, time-consuming extensions are scheduled automatically by the system when implemented in user-space. On the other hand, because the Linux kernel is not preemptive, a long running extension implemented in the kernel must explicitly schedule itself.
- Finally, while it is obvious that no system can be completely safe from malicious users and faulty code, executing in user-space is much safer than executing in the kernel.

This paper describes user-level extensibility in the Mona file system [9, 10]. Untrusted user-defined transformations execute safely outside the kernel, modifying data as it is read and written. Users associate zero or more transformations the with input and output streams of a file, which specialize the file system for a particular application or use. Although executing transformations in user-space is less efficient than executing in the kernel, user-level transformations are efficient. Read and write operations are approximately five times slower when guarded by user-level transformations than when guarded by kernel transformations. This overhead is not noticeable for interactive operations, and is comparable to the performance of Unix pipes.

Additionally, this paper describes several interesting user-level file transformations, illustrating the possible uses of a higher-level file system. For example, the `ftp` transformation provides transparent access to files on a remote FTP server. On a file access, the `ftp` transformation issues an FTP request to the designated remote site, waits, and fills local buffers with the returned data. The `php` transformation reads a raw PHP file from disk. It parses the contents of the file and creates data for the file buffers dynamically. Further, the `mp3` transformation decodes MP3 files on the fly. Thus, an application can simply write a raw MP3 file to a special file. The file system takes care of decoding. Lastly, the `command` transformation executes a program—often a simple shell script. The `command` transformation redirects the I/O of the program from/to the file system. As a result, extending the Mona file system is as simple as writing a shell script.

The remainder of this paper is organized as follows. Section 2 provides an overview of the Mona file system. Section 3 describes several transformations. Section 4 discusses the `export` transformation that safely executes user-level transformations in user space. Section 5 presents measurements of the Mona file system and the `export` transformation. Section 6 discusses related work. The last section presents our conclusions.

2 Mona File System Overview

Mona provides file system extensions as *transformations*, which modify streaming data [9]. A typical transformation acts as a filter, reading input, then pushing modified data downstream. Mona supports two types of transformations: *kernel* and *user-level*. Common kernel transformation code is downloaded into the kernel at the time the Mona kernel module is inserted. Less frequently used kernel transformations may also be added (and removed) dynamically. User-level transformations are executed in a user-level process that communicates with the file system via the *export* kernel transformation, as described in Section 4.

The Mona file system creates *virtual* files, whose contents exist only in the file system while the virtual file is open. There are two mechanisms for creating virtual files: persistent and transient. A *persistent* transformation link, which is similar to a symbolic link, creates an access point in the file system. Any program can access a transformation link exactly as it would access an ordinary file. A persistent transformation link exists until it is explicitly removed.

The `lnx` utility creates persistent transformation links—it is similar to `ln`. In the example below, the `lnx` utility is used to create a persistent link to the file `/dev/audio` with the `fail` transformation on the input (read) and the `mp3` transformation on the output (write).

```
lnx /dev/audio mp3 -r fail_xform \
  -w mp3_xform
```

When a program writes to the virtual file, `mp3`, the MP3 data that the program writes is decoded and written directly to the audio device `/dev/audio`. In this case, there is no reason to read the virtual file, and so the `fail` transformation guards the input stream. In this persistent case, the virtual file exists for all programs to use until the user explicitly deletes it.

Conversely, a *transient* transformation exists only as long as the virtual file is open. A user creates a transient data view by manipulating transformations at runtime. A user pushes and pops transformations on the data streams of an open file using the `ioctl` system call. The example below pushes the `fail` transformation on the input stream and the `mp3` transformation on the output stream.

```
ioctl(fd, PUSH_INPUT,
  "fail_xform /dev/audio");
ioctl(fd, PUSH_OUTPUT,
  "mp3_xform /dev/audio");
```

This creates a virtual file identical to the persistent transformation example above. However, the virtual file is only accessible through the file descriptor, and is destroyed when the file is closed.

Mona adds new functionality to the file system without sacrificing backwards compatibility. Because Mona is compatible with ext2, either file system may be used to read and write media based on the other (of course, ext2 does not support the added functionality of Mona). The Mona file system is a Linux kernel module, which can be loaded and unloaded as needed.

In addition to maintaining compatibility with the ext2 file system, we have also demonstrated that the overhead of using the Mona file system is negligible. When using Mona as a traditional file system, i.e. without utilizing its enhanced capabilities, an overhead of less than 1% is incurred on `read`, `write`, and `open` system calls. Additionally, tests have shown that Mona performs similarly to ext2 for the PostMark suite of benchmarks, the Andrew benchmark, and for kernel compilation. When utilizing user-level transformations we have found that little performance is sacrificed relative to what we believe is a large gain in functionality. In cases where a network of transformations is used, we have found that Mona can perform better than current models. Preliminary experiments have shown this to be true for emulating the functionality of Unix pipes, where we have shown a clear performance advantage [8].

3 User-level Transformations

This section first presents a number of user-level transformations that we have implemented to address real-world problems. These transformations allow existing mechanisms to be applied more broadly via the file system, allowing users to solve new problems or to solve existing problems more easily. After demonstrating some uses of user-level transformations, a brief tutorial on constructing user-level transformations is presented. Finally, the section concludes with a short discussion of the benefits of user-level file system extensions.

3.1 Applications of User Transformations

An active file system raises the level of the file system abstraction. The higher level of abstraction provides significant benefit to both developers and end-users.

- Application code development is simpler. First, there is less for the application to implement. Second, code re-use is more likely because programmers can use the well understood file system interface instead of having to learn a new API. Moreover, there is a central location for code, which simplifies upgrading systems.
- All applications can use the expanded capability–even those that are not modified. For example, encryption in the file system provides security to any application that reads or writes data from a file.
- Novel file system semantics and structures can be created. For example, locking or data consistency can be provided by the file that is concurrently shared—rather than by the applications (which may not even be aware the file is shared). Other examples are remote access (FTP, HTTP), logging, journaling, dynamic file creation (PHP), and file conversion.

The above are the potential advantages of an active file system. It is also a sampling of the potential uses of user-level transformations. Mona's flexibility and ease of programming allow for many other uses.

Mona user-level transformations give programmers a novel interface for code reuse. One example of this is the `mp3` transformation, which decodes MP3 audio files on the fly. Suppose a virtual file guards the audio device with the `mp3` transformation. Any program can now write an MP3 encoded file to the virtual file. The `mp3` transformation decodes the data in the file system, which is given to the device. Not only do programmers avoid the need to implement their own MP3 decoders, but having the code at a centralized point makes updates easier. Moreover, the code is associated with the device, not the applications.

Some user-level transformations provide new ways of using existing resources. The `ftp` transformation creates a common interface to local and remote files, allowing users to navigate a remote FTP site as if it were part of the local file system. A user executes ordinary Unix executables (e.g., `cd`, `ls`, `cp`, etc.) to manipulate remote files, eliminating the need for explicit file transfer requests. A virtual file named `foo.remote` can be a link to a file `foo` on a remote FTP site. When the user reads the virtual file, the ftp transformation automatically negotiates with the remote host and transfers foo to the user's local machine. Except for the latency of the file transfer, the user may never know that the base file exists only on a remote server. For public FTP access, a user can use the generic ftp transformation. If a user wants to access private data via FTP, she can recompile the transformation to contain her username and password and place the new shared library in her personal transformation directory.

A variety of archive transformations allow access to the content of various archival formats, including tar, rpm, deb, zoo, and zip files without unpacking the archive. These transformations allow users read and write access to the various archives without the need to manually extract and re-archive those files. The similar gzip and bzip2 transformations allow the same type of manipulation on files in compressed form. The ftp, archive, and compression user-level transformations all demonstrate Mona's flexibility and usefulness in providing new views and interfaces to existing resources.

User-level transformations can make use of existing tools. The php transformation is an example of this. PHP is a server-side, cross-platform, HTML embedded scripting language. It allows developers to create scripts that dynamically generate web pages. PHP is the most popular module for the Apache web server, and is employed in creating dynamic content for a large number of commercial web sites. The Mona php transformation combines the power of PHP with the flexibility of the Mona programming model. By moving PHP into a Mona user-level transformation, we allow the creation and delivery of dynamic content via any transport method (e.g. FTP, text editor, web browser, etc.).

Many web sites use PHP to parse a database file and generate the day's or week's events on the fly. Using the Mona PHP transformation, one can create a similar utility that does not depend on server processing. Consider guarding a user's .plan file with the php transformation. The raw data in the .plan is a PHP script that parses a database file. When any user reads this .plan, the php transformation dynamically generates a user's plan from a database file that is specific to the current date. This is an example of Mona's ability to extend the usefulness of an existing tool: e.g., serverless PHP.

The command transformation makes programming file system extensions even easier. Comparable in some ways to a traditional Unix command-line pipeline, this transformation allows the user to execute arbitrary executables or shell scripts on input and output streams. Text processing tools such as grep, sed, and gawk can be placed on streams to automatically filter out unwanted data. For example, if a user wants to print out only the first field of each line, a transformation link bar.first can be created pointing to a file bar.raw with the below transformation on the input stream.

```
command /usr/bin/gawk '{print \$1}'
```

The file bar.first appears as a normal file containing just the first field of the records in the file bar.raw.

There are many other potential uses of the command transformation. For example, a simple script could guard an HTML file, which forces a reload of a browser whenever the base HTML changes. In this way, users edit HTML files with their favorite editor, but can view up-to-date renderings of the files in web browsers. In another example, version control could be automated using the rcs utility. Finally, if a user wishes to be notified when a file is read or modified, the command transformation can send mail automatically. In short, any program can be associated with a data stream using the Mona file system. This illustrates the flexibility that the Mona system affords the user. Just as the export transformation is a kernel-resident transformation built to allow transformations to be executed in user space, the command transformation is a standard user-level transformation built to allow executables to be associated with data streams.

3.2 Implementing a User Transformation

This section describes the construction of simple user-level transformations. The implementation is primarily a matter of creating a filter that conforms to a well-defined function-call interface. A transformation has the following calling interface:

```
int decompress_xform(
  XformInfoPtr xf_ptr,
  int input_bytes,
  char *buffer,
  int *state);
```

The transformation info pointer, xf_ptr, uniquely identifies each instance of a transformation. In particular, it contains information about data in the transformation network and a pointer to memory that may be utilized by the user to store data private to the current instance of the transformation. The parameter input_bytes provides the size of the data passed to the transformation via the character pointer buffer. The last of the arguments passed to a transformation is state, an integer pointer to the "global" transformation state variable. Each transformation returns the total number of bytes placed in the buffer to be sent downstream to the next transformation in the network, if one exists. Additionally, it sets state, indicating whether all input has been consumed.

Figure 1 lists the source code of a simple decompression transformation. It takes an input buffer and passes it to a decompression routine. The decompression routine

```
int decompress(char *, char *, int);
typedef struct decomp_data { int size; int sent; char* data; } d_data;
int dec_xform(XformInfoPtr xf_ptr, int input_bytes, char *buffer, int *state) {
  d_data *data_ptr = (d_data *)xf_ptr->private_data;
  int output_bytes;

  if(decomp_data == NULL) {
    decomp_data = (d_data *)malloc(sizeof(d_data)):
    decomp_data->size = decompress(buffer, data_ptr->data, input_bytes);
    decomp_data->sent = 0;
  }
  if((output_bytes = decomp_data->size - decomp_data->sent) > BLOCK_SIZE) {
    memcpy(buffer, decomp_data->data, BLOCK_SIZE);
    decomp_data->sent += BLOCK_SIZE;
    *state = XF_CURRENT_HAS_DATA;
  } else {
    memcpy(buffer, decomp_data->data, output_bytes);
    free(decomp_data->data);
    free(decomp_data);
    xf_ptr->private_data = NULL;
    *state = XF_READY;
  }
  (output_bytes > BLOCK_SIZE) ? return(BLOCK_SIZE) : return(output_bytes);
}
```

Figure 1: Source code for the *decompress* transformation.

is responsible for managing the space required for the new data, using the pointer passed by the transformation to allocate memory. If the `private_data` element of the structure `xf_ptr` is NULL, then a new block of data has been sent. In that case, some initialization tasks are performed, along with the decompression itself. If there is more data than can fit in one block, one block is sent along, and `state` is set to `XF_CURRENT_HAS_DATA`. This ensures that the current transformation will be called again before more data from the source file is passed in the buffer, allowing `decompress_xform` to continue sending the decompressed data until it has been consumed. Once the remaining decompressed data is smaller than the block size, it can be sent, cleanup can be performed, and `state` can be set to `XF_READY`, indicating that the transformation is ready for new data.

To make use of this transformation, the user must compile it as a shared library and put the shared library where Mona can find it:

```
gcc decompress_xform.c -shared \
 -o decompress.so
mv decompress.so ~/.mona
```

Mona user-level transformations are implemented as functions. When size is conserved, the programmer does not have to be concerned about setting flags or computing return values. More complex transformations require the programmer to maintain buffers, set state variables, and compute return values. These additional programming tasks are minor in comparison to the added work of creating more complex transformations. When complete, the function is compiled into a shared library and is ready to use.

3.3 Advantages of User Transformations

The Mona API makes transformation development easy for the programmer. While kernel transformations in Mona share the same interface as user-level transformations, there are important advantages to programming user-level transformations. User-level transformations offer the programmer access to the libraries of code that have become a part of every programmer's tool box. Common debugging tools can also be used on user-level transformations. Additionally, since kernel transformations run in kernel mode, they are not preempted. This means that kernel transformations must be explicitly scheduled by the programmer. Because user-level transformations are scheduled like any other user-level process, they do not suffer from these complications. These three advantages of user-level transformations together with Mona's interface make user-level transformations an easy target for any developer.

In addition to the advantages discussed above, user-level transformations are also much safer than kernel transformations. Since user-level transformations run in

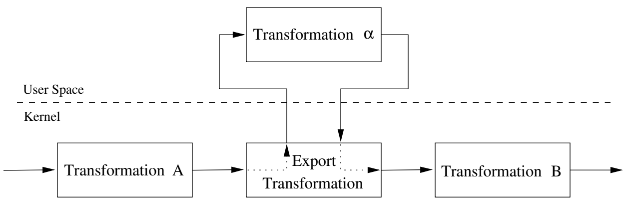

Figure 2: Illustration of `export` transformation.

user space, they cannot access internal kernel data structures and therefore cannot compromise the integrity of the kernel. This means that any user can write and test a user-level transformation without the intervention of a system administrator.

Mona user-level transformations provide users with a virtually unlimited number of ways to extend the file system. The ability to use existing code (and even existing applications) greatly reduces implementation time even for complex extensions. Mona's fine-grained per-file approach makes it more useful than many previous approaches to file system extension.

4 Export Transformation

In order to realize the advantages of executing in user space Mona needs a mechanism for exporting transformations. The `export` kernel transformation safely executes untrusted or computationally expensive transformations outside the kernel in user space. For example, consider executing three consecutive transformations, A, α, and B, where A and B are kernel-resident transformations and α is a user transformation. Mona inserts the `export` transformation between A and B, as shown in Figure 2. The `export` transformation passes its input data across the kernel/user space boundary to a user-level transformation α. When transformation α completes, it returns output data to the `export` transformation which then passes it on to B.

4.1 Export Execution Model

The `export` transformation executes user transformations in user-level processes for two reasons. First, processes allow the Mona implementation to easily support transformation concurrency. Second, the *setuid* system call provides control over the access permissions of a process (and any transformations within it).

The Mona implementation uses a daemon to supervise all transformations that execute outside of the kernel. When the file system instantiates an `export` transformation during an *open* system call, the daemon forks

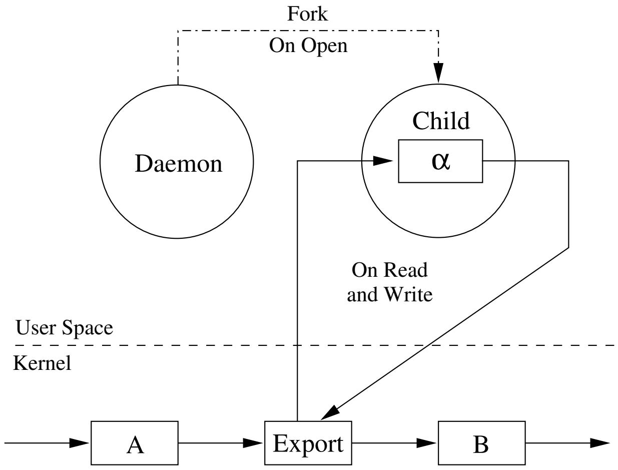

Figure 3: Execution model of `export` transformation.

a child process to handle accesses to the file. Any subsequent read or write to the file passes data up to the child process, which transforms the data and returns it to the `export` transformation, as shown in Figure 3. This figure provides an in-depth look at the example first introduced in Figure 2. Transformations A and B reside within the kernel, but α is exported to user space. Any data that streams through the network passes up through the `export` transformation to user space, through α, the transformation in the child process, and back to the kernel.

4.2 Export Implementation

The `export` transformation requires three extensions to the Mona file system. First, we extend the `ioctl` system call to enable communication between the kernel and user-space I/O streams. The `MONA_IOC_K2U_MASTER` option to the `ioctl` call allows the Mona daemon to detect when the file system instantiates a new `export` transformation. Two other `ioctl` options, `MONA_IOC_K2U_SLAVE` and `MONA_IOC_U2K`, allow children of the daemon to request and submit transformation data. Both request `ioctl` options block their calling processes until data is available. As a result, there is a clean, well-defined interface between the `export` transformation and the Mona daemon.

Second, Mona implements an initialization queue, which is a queue of transformations that are waiting to be pushed into user space. The Mona options for the `ioctl` system call give the Mona daemon access to this queue. The daemon blocks on the `MONA_IOC_K2U_MASTER` `ioctl` call until the file system places a transformation in the initialization queue. When this occurs the daemon awakens and performs the following actions to initialize the transformation in the queue. The daemon reads a key from the kernel that uniquely identifies the transformation. Then it forks a child process and sets the UID and GID (user and group ID) of the child to that of the owner of the transformation, in order to preserve file permissions. (Section 4.3 discusses the security issues in detail.) The child process services subsequent requests, as described below.

The final extension required is an execution queue of initialized transformations that are awaiting execution. The file system moves a transformation from the initialization queue to the execution queue after the Mona daemon forks a child process for the transformation. The child process blocks in the queue until there is data on which it can execute. Included with the transformation data are the names of the shared library and the function to be called. Each child then opens the shared library file using the Linux `dlopen` call and loads the appropriate function with the `dlsym` call. Transformation data on the execution queue is uniquely identified by transformation identification keys. A child of the daemon uses its identification key, which was provided by the Mona daemon, as an argument to the `ioctl` `MONA_IOC_K2U_SLAVE` system call in order to request data from the execution queue. After the child transforms its data, it sends a reply back to the kernel through another `ioctl` call using the `MONA_IOC_U2K` option. This process repeats until the file access completes and the child terminates.

4.3 Export Security

For the exported transformation to be safe, it must execute with proper permissions and maintain the integrity of the kernel. Before a child of the Mona daemon executes a transformation, it changes its UID and GID to a safe permission level. There are three obvious choices for a UID and GID in this situation, the owner of the base file, the owner of the virtual file, or the user accessing the virtual file. However, two of these are unsafe. If a transformation ran under the permissions of the user accessing a file, the transformation creator would have access to the user's files. Setting a transformation's permissions to that of the owner of the base file is also unsafe. For example, a user could point a transformation link to a file owned by root and have the transformation generate a shell which has root permissions. Therefore, the Mona daemon uses the `setuid` system call to change the effective user id of the child process to that of the user who created the virtual file and specified the code to be executed [17]. Consequently, a transformation will not perform actions that the owner of the virtual file is not allowed to perform.

The `export` transformation does not compromise the integrity of the kernel even though data originating in user space flows into the kernel. First, the mechanism for passing data across the kernel-user boundary truncates the kernel buffer if the child process, which is executing a user transformation, attempts to exceed the space allocated for the kernel buffer. Second, data that passes through a transformation is never executed, it is only appended to an I/O stream on a read or write.

Furthermore, the Mona file system enforces proper use of the `ioctl` extensions. The `MONA_IOC_K2U_MASTER` option restricts accesses to processes owned by root and exits with an error message for any other user. The other new options, `MONA_IOC_K2U_SLAVE` and `MONA_IOC_U2K`, allow any user to request and submit transformation data. However, each call requires a valid transformation identification key before the kernel accepts the call. It is conceivable that a user could randomly guess keys and attempt to insert or remove data from another user's I/O stream. However, with a large enough key the probability of successfully guessing a random key is essentially zero.

In summary, the `export` transformation overcomes the three difficulties of kernel-resident transformations. First, the Mona daemon allows an unprivileged user to extend file system capabilities in a secure manner. Second, transformation code that executes for extended periods of time runs in user space, where it is time shared along with all other processes to maintain fair scheduling of resources. However, when performance is paramount, one can implement a kernel-resident transformation. Finally, user-level transformations are much easier to implement than kernel-resident transformations due to the availability of user-space tools such as libraries and debuggers.

5 Results and Evaluation

Mona provides greater functionality through file system extensions. As such, the focus of the project has not been quantitative (i.e., user-level transformations do not focus on increasing system performance). Using Mona transformations will provide performance wins in some cases, but the purpose of Mona is to provide ways to make systems more useful to users. However, if the cost of a user-level transformation is large, its benefits could be outweighed. For this reason, we performed several tests to measure the overhead that Mona adds to a system.

To determine the baseline overhead associated with using Mona instead of ext2, we compare system call execution times for files in ext2 and unguarded (i.e., no transformations) files in Mona. The results indicate that the overhead added to these system calls is less than 1%. This overhead applies only to the `open`, `read`, and `write` system calls. To find out the effect using Mona

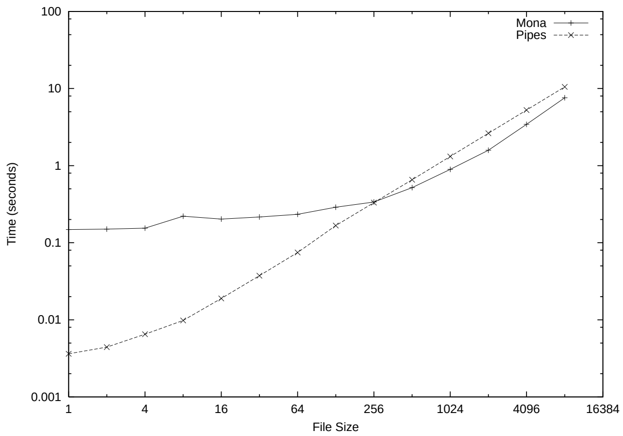

Figure 4: Unix pipes vs. Mona user-level transformations (Linux 2.2 kernel)

.

has on aggregate system performance, we also performed tests using several file system benchmark suites. Our tests have shown that the PostMark suite had the largest overhead at 2%, whereas the Andrew and kernel compile tests were both below 1%. This amount of overhead on unguarded files is small, and the added functionality provided by the Mona file system outweighs the performance penalty [8].

Mona's ability to allow unprivileged users to extend the file system is one of its key novel aspects. The overhead of using a Mona user transformation under the Linux 2.2 kernel is approximately 150 μs. This cost is quickly amortized as file size increases, and is negligible for large files. The same principle applies for transformations of increasing complexity. For a transformation that is computationally non-trivial, the latency contributed by this overhead will be considered acceptable by the user.

In many cases the added functionality is similar to that of standard Unix pipes. To compare our performance to that of pipes, we ran several tests implementing the same computation as both processes communicating via pipes and as Mona transformations. This test sends input data through two transformations using a Unix pipe and the Mona file system. Figure 4 shows that Mona transformations compare favorably with Unix pipes. Due to the overhead of instantiating the new process, user transformation performance is worse than that of pipes until the file size approaches 128K, although only by a little over a tenth of a second in the worst case. For file sizes over 128K, Mona user-level transformations perform better than their pipe counterparts by as much as 65%. The two Mona user-level transformations operate in the same user process. Consequently there is no overhead from switching processes. Mona incurs no overhead from buffer copying because it passes a pointer to a buffer between transformations. Mona's performance advantage increases when several operations are stacked upon one another.

6 Related Work

An adaptive I/O subsystem was implemented in Streams [16, 2, 13] and is a component of several System V variants. A stream is a connection between a device driver and a user process. The `ioctl` system call pushes stream modules (similar to Mona transformations) into the stream. When data flows through the stream and reaches a module, code from the module executes on the data before passing modified data downstream. Like Mona, the Streams system enables dynamic extensibility. However, there are some differences between the systems. First, all Stream modules execute with full permissions in the kernel. Consequently, only privileged and expert users can create new extensions. In addition, a Stream module is inherently duplex and adds a stage in both the input and output pipeline. This works well for operations that have natural inverses, such as compression/decompression. However, if an operation does not have or require an inverse operation (such as PHP) for data traveling the opposite direction, the technique wastes resources and adds an extra pipeline stage.

The watchdog system provides extended file semantics by guarding file accesses with special processes [3]. Each watchdog process is a user-level program associated with either a file or directory. When a guarded file opens, the kernel negotiates with the watchdog guarding the file to determine how to handle accesses. Like user-space transformations, watchdog processes provide a simple mechanism to add user-defined extensibility to a file system. However, creating a new process for each open guarded file is expensive in system resources and interprocess communication. Managing an entire process is excessive overhead for simple transformations, such as *lock*. The watchdog system cannot push common operations into the kernel where they can execute quickly. The flexibility of the Mona system allows the user to decide the best execution environment for any particular operation.

BSD Portals extend the file system by exporting certain open system calls to a user-space daemon [12]. A portal daemon is mounted as a standard file system. When the kernel resolves a pathname that leads through the portal daemon mount, the remainder of the path is sent to the daemon. Depending on the type of daemon that is mounted, some type of open occurs, and a file descriptor is returned. This allows for arbitrary code to be executed on opens, but this functionality is specific to the open system call. Translators provided by GNU/Hurd [14] are very similar to Mona user-level transformations. A translator is a program inserted between the content of a file and the user process. Translators are user programs, and as such can be installed and modified by regular users. Translators provide the file system interface for Hurd programs.

Stackable file systems derive functionality from preexisting file systems [6, 11]. By stacking a file system on top of another in a file system hierarchy, the operations provided by the lower level are inherited by the higher. A stackable file system is most effective when a handful of operations are required for a large set of files and the operations change infrequently. However, because a system administrator must implement all stacking, the benefit to ordinary users will be somewhat limited. Furthermore, the layering structure (and thus the functionality) of stackable file systems cannot be modified dynamically. Stackable file systems use mount points rather than files as targets for extensibility, and as such are a much more coarse-grained approach than Watchdogs or Mona, where users define their own operations and implement them on a per-file basis. Apollo's DOMAIN file system [15] is quite similar to stackable file systems. It allows users to define new file types and associate associate user defined procedures with these new types. However, extensible code resides in user processes.

Stackable templates alleviate the usability difficulties of stackable file systems by abstracting complex kernel code into templates [20]. Consequently, *Wrapfs*, a stackable template file system, provides a much simpler (and more usable) interface than previous stackable file systems by hiding details of the operating system internals. Wrapfs extends the vnode interface to enable stacking, as originally proposed by Rosenthal [18]. As a result, Wrapfs supports unmodified native file systems while providing users an extended vnode interface. This approach contrasts the Mona file system, which is implemented as a peer to other native file systems within an unmodified virtual file system interface, and maintains full compatibility with the *ext2* file system. Furthermore, unlike Mona, Wrapfs does not allow streams to change size at runtime.

A simulation of Active Disks uses transformation-like *disklets* to operate on data streams entering and exiting intelligent disk drives [1]. The Active Disk architecture integrates processors and large amounts of memory onto a disk drive. An analysis of several algorithms on this architecture found that an Active Disk using application-specific disklets outperforms conventional disk drives. Additionally, Active Disks scale considerably better than traditional disk architectures. Active Disks and Mona demonstrate the potential of modular, stream-oriented processing.

The *userfs* package implements customizable file systems as user processes [5]. Unlike the stackable systems discussed above, mounting a user file system does not require privileged access. As a result, unprivileged users can customize their environments without the assistance of a system administrator. This system thus allows simple file system extensibility, but is more coarse-grained than Mona. Entire hierarchies are mounted with the userfs, whereas the flexibility of the Mona file system allows actions to be associated with individual files or hierarchies.

There are many projects whose goal is to provide general kernel extensibility, including file system extensibility [4, 7, 19]. These systems address the safety issues associated with downloading untrusted code into the kernel. The Mona file system provides only a subset of the extensibility offered by general kernel extensibility projects (i.e., the subset that relates to the file system). Mona only allows trusted code in the kernel and concentrates on user-space extensibility for untrusted code.

Through the use of traditional Unix tools like filter programs and pipes, we can achieve a series of stacked operations on a data stream, but to use these, we need to either explicitly create the pipes in a process, or use the pipe functionalities of a shell. The former requires significant modification to applications, while the latter allows us to only operate on standard input or output, not arbitrary files. Shell associations perform actions (e.g., launching an application) based on the type of the underlying file. This could be similar to Mona's functionality in some cases, but does not, for example, allow dynamic insertion and deletion of functionality at runtime. Another Unix mechanism that can be used is the LD_PRELOAD functionality of dynamic libraries. Library preloading can be used to intercept file system calls and perform operations on the data between the user program and the operating system. However, when using library preloading, all calls that are intercepted will have to go through the filter code. Mona allows filters to be specified for files on an individual basis.

7 Conclusions

This paper describes how to raise the level of abstraction provided by a file system. It presents the Modify-on-Access (Mona) file system, which supports safe and simple user-defined extensions called transformations, which are modular, stream-oriented operations that are inserted into an I/O stream during a file access. This paper presents several examples in which extending the structure and semantics of a file simplifies applications and benefits both the application programmer and the end user.

The Mona file system provides a novel combination of granularity, modularity, and usability. Mona supports transformations on a fine-grained per-file basis. In addition, transformations are modular and can compose larger operations. Finally, Mona can execute a transformation within the kernel or in a user-space process. As a result, a user has the ability to choose appropriate levels of performance, safety, and ease of use. The flexibility in all three areas distinguishes Mona from previous extensible file systems.

This paper presents several lessons. First, although transformations are a limited form of computation, there are many useful operations that fit the form. Moreover, it shows that the model (and our implementation) are ideally suited for extensions. The `export` transformation is a kernel transformation that enables user-level transformations. Similarly, the `command` transformation is a user-level transformation that enables shell script transformations.

This paper examines the cost of user-level transformations. We show that the cost is comparable to pipes, and in some cases better. The cost of user-level transformations is negligible when a transformation involves a latency-dependent operation, like the `ftp` transformation. Additionally, as the complexity of a transformation increases, the relative cost of a user-level transformation decreases. Even in the worst case–a trivial transformation–the absolute cost of invoking a user-lever transformation is low enough (approximately 150μs) that there is not a noticeable additional delay for interactive environments.

Finally, this paper shows a simple but effective technique for maintaining security with user-level extensions. Our technique sets the UID and GID of the child helper processes such that code executes with permissions that are allowable by the system. Consequently, Mona ensures that the use of user-level transformations will never compromise security.

Acknowledgements

We would like to thank our reviewers for their comments, and especially Alan Nemeth for his help in shepherding us through the review and submission process.

We would also like to thank Richard Kendall for his foundational contributions to the Mona project.

Mona is available as open source in conformance with the Open Source Initiative's Open Source Definition. Mona is available at http://www.cse.nd.edu/˜ssr/projects/mona.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proc. Eighth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[3] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the UNIX file system. *USENIX Winter Conference*, pages 267–275, Winter 1988.

[4] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. Fifteenth ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, December 1995.

[5] Jeremy Fitzhardinge. Userfs–file systems implemented as user processes. http://sunsite.unc.edu/pub/micro/pc-stuff/Linux-/ALPHA/userfs/!INDEX.html, 1997.

[6] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[7] M. Frans Kashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russel Hunt, David Mazières, Thomas Pickney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. Sixteenth ACM Symp. on Operating Systems Principles*, pages 52–65, Saint Malo, France, October 1997.

[8] H. Richard Kendall, Vincent W. Freeh, Paul W. Schermerhorn, and Peter W. Rijks. Streaming extensibility in the modify-on-access file system. To appear in the Journal of Systems and Software.

[9] H. Richard Kendall. The modify-on-access file system. Master's thesis, University of Notre Dame, Notre Dame, IN 46556, July 1998.

[10] H. Richard Kendall and Vincent W. Freeh. The modify-on-access file system: An extensible Linux file system. In *Proc. of LinuxWorld Conference and Expo*, San Jose, CA, March 1999.

[11] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. *ACM SIGOPS*, pages 1–14, December 1993.

[12] A. David McNab. BSD portals for Linux 2.0. Technical Report NAS-99-008, NASA Ames Research Center, 1999.

[13] Steve D. Pate. *UNIX Internals*. Addison-Wesley, 1996.

[14] Gnu Hurd Project. Debian GNU/Hurd translators. http://www.debian.org/ports/hurd/hurd-doc-translator.

[15] Jim Rees, Paul H. Levine, Nathaniel Mishkin, and Paul J. Leach. An extensible I/O system. *Proceedings of 1986 Summer USENIX Conference*, pages 114–125, June 1986.

[16] Dennis M. Ritchie. A stream input output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[17] Dennis M. Ritchie and Ken Thompson. The UNIX time–sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[18] David S. H. Rosenthal. Evolving the vnode interface. *USENIX Summer Conference*, pages 107–117, Summer 1990.

[19] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. Fourteenth ACM Symp. on Operating Systems Principles*, pages 203–216, Ashville, NC, December 1993.

[20] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proc. 1999 USENIX Annual Technical Conf.*, June 1999.