Streaming Extensibility in the Modify-on-Access File System

H. Richard Kendall  Vincent W. Freeh  Paul W. Schermerhorn
Robert J. Minerick  Peter W. Rijks
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

Abstract
This paper presents the Modify-on-Access (Mona) file system that provides extensibility through transformations applied to streams of data. Mona overcomes two limitations of prior extensible file systems. First, the Mona file system offers two levels of extensions (kernel and user) that share a common interface. It allows performance-critical operations to execute with modest overhead in the kernel and untrusted or more complex operations to safely execute in user space. Second, Mona enables fine-grained extensions which allow an application to customize the file system at runtime.

This paper discusses the implementation of the Modify-on-Access file system. Our implementation adds modest overhead of 0–3% (0.01–0.21 µs) to file system operations. This overhead has even less effect on net system performance for several benchmarks. Moreover, this paper describes applications that achieve 4–5 times speedup using custom transformations. This paper also describes several transformations that increase functionality. Among these are the find transformation that allows a user to browse a remote file as though it were local and the command transformation which invokes an arbitrary executable (even a shell script) on a data stream.

1 Introduction
Over the last decade, research in system software has demonstrated that extensibility offers increased performance and functionality [5, 10, 28, 8]. However, fully extensible operating systems are little used, primarily due to the high cost associated with adopting a new operating system. Moreover, some argue that the cost of an extensible operating system in terms of complexity and performance does not justify the added benefits [7]. On the other hand, extensions at the file system or device driver level have little cost of adoption and retain most of the benefits of extensibility.

This paper presents the Modify-on-access (Mona) file system, a file system that applies dynamic, fine-grained extensions, called transformations, to streams of data [15, 17]. Mona demonstrates increased performance and functionality over traditional file systems with virtually no cost of adoption. Mona uses the existing Linux virtual file system interface and maintains complete compatibility with ext2, the standard file system from which it is derived.

Mona is an active file system that cooperates with processes to accomplish tasks. This is in contrast to conventional file systems, which are passive entities that transport data between peripheral storage and main memory. Additionally, Mona can provide extensibility on a per-file basis. Moreover, modifying data concurrent with its transfer has the potential for speedup because cached data is reused.

While other extensible file systems have been implemented, the Mona file system is novel in two ways. First, it supports both privileged and unprivileged extensions. A kernel transformation is part of the Mona file system module and executes with low overhead and full privilege inside kernel space. In contrast, a user-level transformation executes in user space, in an ordinary, unprivileged user-level process. It is not part of the file system. A user-level transformation trades in-kernel performance for increased security.

*This research was supported by NSF grant CCR-9676073 and the Arthur J. Schmitt fellowship. This paper will be published in the Journal of Systems and Software in 2002.
enabling unprivileged users to safely extend the file system or kernel. The second novel feature of Mona is the ability to easily and efficiently apply extensions on a per-file basis at runtime. A process can push and pop transformations on an open file stream, allowing much greater flexibility than previous extensible file systems.

In Mona, the basic unit of computation and extensibility is the transformation. A transformation is an arbitrary operation on a stream of data. Mona transforms data as it is accessed. On a read, the file system applies transformations to raw data before presenting it to the user process. Similarly, it applies transformations to data before writing it to disk. This technique pushes operations out of the application and into the file system. Furthermore, an application may stack transformations upon one another, creating a complex operation out of a network of simpler transformations. In this way, transformations extend the capabilities of traditional file systems.

Compression is a straightforward and obvious use of transformations. On a read, the input transformation decompresses data as it is read from the file, and on a write the output transformation compresses. Thus the Mona file system creates a transparently compressed file, hiding details of the compression from users, who access it as though it were a normal file. Furthermore, Mona can avoid unnecessary compression and decompression because the file system provides both standard and compressed versions of the same file. As a result, Mona can copy a compressed file to another location without the unnecessary steps of decompressing and re-compressing it.

The primary novelty in the Mona file system, its unique two-level extension model, promotes the widespread use of extensions. An operation that is not trusted, requires user-level resources, or takes too much time can be implemented as a user-level transformation. On the other hand, a trusted, simple, and short operation can become a kernel transformation. There are several advantages to this two-level model. With Mona, an ordinary, unprivileged user, in addition to having access to existing transformations, may create new transformations as desired. Additionally, the model simplifies the scheduling of long-running kernel extensions. The typical kernel cannot be preempted, so an extension running in kernel space must explicitly schedule other jobs if an operation takes too long. In Mona, such an extension can be exported to user space to be automatically scheduled just like an ordinary process. Lastly, the two-level model greatly simplifies development of kernel extensions. A transformation may be developed and debugged in user space and only migrated into the kernel after it is trusted.1

The Mona file system provides extensibility without compromising compatibility with existing systems. Mona has virtually no cost of adoption because Mona is a drop-in replacement for ext2, the standard Linux file system. This, combined with Mona's unique ability to provide a single interface for dynamic, fine-grained extensions that execute either in the kernel or in user space, enables Mona to provide greater flexibility and performance than existing systems.

The remainder of this paper is organized as follows. Section 2 discusses related work and provides some background knowledge that is necessary to understand the tools and environment the file system uses. Section 3 gives a high-level view of the Mona file system. It describes transformations in detail, explains the user interface to the file system, lists unconventional operations enabled by transformations, and concludes with an examination of the export transformation, which is the mechanism for executing untrusted code safely outside of the kernel in user space. Next, Section 4 explains implementation details. These include the framework for executing transformations both inside and outside of the kernel. Section 5 presents the results of an analysis of the Mona file system. Finally, Section 6 summarizes the conclusions drawn from the research and explains the direction of future work.

2 Related Work

An adaptive I/O subsystem was implemented in Streams [24, 2, 21] 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 calls push 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 several differences between the

1There are differences in capabilities because some data is only available in the kernel and standard libraries are only available in user space. However, there is no difference in the programming model or interface.
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 require an inverse operation for data traveling the opposite direction, the technique wastes resources and adds an extra pipeline stage.

Three systems—watchdogs [4], stackable file systems [11, 18], and Apollo’s DOMAIN file system [23]—are closely related to Mona. The watchdog system provides extended file semantics by guarding file accesses with special processes. 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. In contrast, stackable file systems derive functionality from preexisting file systems. 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. In this manner, a compression or encryption layer can be added to an existing file system. DOMAIN allows users to define new types of objects, which includes file systems. Furthermore, a user can associate procedures with types (objects).

Like user-level 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. In addition, managing an entire process has excessive overhead for simple transformations. The watchdog system cannot push common operations into the kernel where they can execute quickly. Mona allows users the flexibility to determine which level best suits their needs.

In contrast, a stackable file system provides fast kernel operations similar to kernel transformations, but lacks flexibility. A stackable file system is effective if only a few operations are required for a large set of files and the operations change infrequently. However, stackable file systems fail to propagate their full extensibility down to the typical user because a system administrator must implement all stacking. Furthermore, the layering structure (and thus the functionality) of stackable file systems cannot be modified dynamically. This is a much more coarse-grained approach than watchdogs or the Mona file system, where users define their own operations and implement them on a per-file basis. DOMAIN is quite similar to stackable file systems. However, extensible code executes in user processes.

Stackable templates alleviate the usability difficulties of stackable file systems by abstracting complex kernel code into templates [29]. 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 [26]. 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.

3 Mona Programming Model

The Modify-on-access (Mona) file system supports a programming model in which operations are implicitly performed on input and output data streams. We call this model an active file system. This section describes the programming model for an active file system and outlines some benefits that result.

First, an application written for an active file system can insert (or remove) operations into an I/O stream. This allows an application to decide at runtime whether and where to perform an operation. For example, a word processor supporting a single file format could push conversion filters at runtime onto any stream accessing a file in another format. Furthermore, all data elements are touched by the file system when a page of data is fetched from disk, so it can be more efficient to perform simple modifications while moving the data because it is already in cache.

Another benefit arises when the semantics of a file operation can change to better suit an application. For example, standard file systems are not well suited for concurrency. Consequently, applications must provide necessary additional semantics themselves. Many applications create an external lock file that indicates a file is being accessed. Transformations in the Mona file system provide lockable files by acquiring a lock during an open system call and releasing the lock upon a close.
Finally, streaming data operations are common and often critical. There are many useful filter operations, and the demand for operations on streaming input is increasing. Furthermore, as data set sizes grow, streaming operations become more desirable because one does not need the entire data set at any given time [25]. These benefits justify the use of transformations in the Mona programming model.

The remainder of this section presents an overview of the Mona file system. First, basic transformations demonstrate the benefits of simple filtering operations. This is followed by more complex examples demonstrating capabilities not found in conventional file systems. Next, the user interface to Mona is outlined. The section concludes with a description of the mechanism for exporting operations outside the kernel into user space.

3.1 Basic File Operations

Basic filtering transformations that read, modify, and write streams add considerable functionality to a system. A transformation (or series of transformations) generates data that logically, but not physically, resides within the file system. The *compress* transformation is an example of a common filtering transformation. It reads a stream of data and outputs the data in compressed blocks.

Similarly, the *passwd* transformation is another basic filtering transformation that addresses security problems inherent to the original Unix password implementation. All users needed some of the information in */etc/passwd*, but the file also contained privileged information (encrypted passwords). Modern Unix systems separate public and privileged information into two files, because access to these encrypted passwords greatly assisted malicious users in cracking passwords. Because Unix systems do not provide extensibility at the file system level, they required ad hoc additions to restrict access to private data. On the other hand, the *passwd* transformation removes passwords from the input when unprivileged users read */etc/passwd*. Thus, it allows both classes of data (privileged and unprivileged) safely to remain in the same file. It was trivial to write the *passwd* transformation in the Mona framework. Similar Mona transformations easily can implement other unforeseen protection schemes. In fact, the *passwd* transformation is, for the most part, an instance of the more general *select* transformation, which strips out all but selected fields from a data stream.

Transformations may be stacked upon one another, creating a transformation network. Mona successively applies the set of stacked transformations to all data in a stream. One example stacks encryption and compression on a data stream, presenting the user with transparently encrypted and compressed output.

3.2 Extended File Operations

Mona supports arbitrary transformation operations in addition to the basic filter operations described in Section 3.1. As a result, the Mona file system provides extended operations on files that are not found in conventional file systems.

Mona also supports enhanced backup and logging functions. For example, the *access-log* transformation monitors a file by writing information to a log when someone accesses it. The *mirror* transformation applies updates to two files simultaneously, which provides simple redundancy. A transformation could also provide automatic version control and a history of modifications to a base file. In addition, Mona can trigger other actions when a file is accessed. Examples of such actions are sending a mail message or incrementing a hit count. This action could be context-sensitive, performing a different action based on the user ID or time of the access, or even on the current system load.

The *pme* file system [3, 9] demonstrates the benefits of providing system information through pseudo-files. The Mona file system also implements pseudo-files, in which transformations generate their own input data rather than reading it from a file. For example, the pseudo-file */etc/time*, a virtual file created by the *current-time* transformation, returns the system time when it is read. Another example is */etc/random*, which provides a stream of random numbers; a write to */etc/random* is cast as an integer that acts as the seed value. Another useful pseudo-file for parallel computing is the *ticket broker*, which issues a unique ticket on every read.

A transformation can create implicit file structures. It can maintain a file in a format, such as a queue or heap, that is transparent to users. For a file guarded by the *queue* transformation, a write automatically inserts data at the tail of the queue and a read removes data from the head of the queue. Normal reads and
writes access these files, but a transformation that maintains the data structures supervises the interface to the base file. Because accesses to the base queue file are atomic, a shared queue file provides a simple communication channel between processes.\footnote{\textit{Queues and heaps have well-defined insertion and deletion points and thus match the conventional read and write API. A more complex data structure, such as a relational database, requires a more elaborate interface that uses auxiliary functions or \texttt{ioctl}.}}

In addition to the above transformations, we have implemented several larger and more complex transformations. The \textit{ftp} transformation allows a user to navigate a remote FTP site as if it were part of the local file system. A user executes ordinary shell commands (\texttt{cd, ls, grep, more}) locally, but manipulates remote files. A similar transformation provides users read and write access to files in \texttt{tar} archives without the need to manually extract and re-archive those files. The \textit{command} transformation is another useful transformation which applies a shell script or binary program to data streams in user space. For example, one could strip passwords from \texttt{/etc/passwd} by calling the cut program within the \textit{command} transformation.

In summary, the Mona file system surpasses the capabilities of conventional file systems through transformations that operate on data streaming either to or from a file. As a result, extended file operations such as implicit file structures, context-sensitive triggers, and pseudo-file services provide enhanced functionality.

\subsection{Mona Interface}

The Mona file system creates \textit{virtual files} whose data resides logically, but not physically, within the file system. This contrasts with \textit{actual files} whose data is physically within the system. A user accesses virtual files and actual files in the same way because they are indistinguishable to the user. Mona creates virtual files on demand. For example, applying the \textit{decompress} transformation to a compressed file creates a virtual file that is the decompressed image of the underlying file. Mona decompresses page data on-demand as the virtual file is read. If only part of the virtual file is read, then only part of the underlying file is decompressed. Furthermore, if the underlying compressed file is modified, the decompressed virtual file will reflect this change.

Conventional system calls access files in the Mona file system. The \texttt{open, read, write, and close} calls use typical parameters and return values. In addition, the \texttt{ioctl} system call can poll an opened virtual file to determine the state of its transformation network.

For usability, the Mona file system provides a user two types of virtual files: \textit{transient} and \textit{persistent}. A user creates a transient virtual file by manipulating transformations at runtime. To accomplish this, the user pushes and pops transformations on the data streams of an open file using the \texttt{ioctl} system call. These transformations are not visible outside the user process, and the information associating them with a file is discarded when the file closes. The source code in Figure 1 illustrates this procedure. First, the code opens an empty heap file. Next, the \texttt{ioctl} call specifies a heap read and write. Reads and writes manipulate the heap structure by removing and inserting keys, respectively. The \texttt{close} system call then discards internal transformation data structures in addition to closing the file. To a user accessing the heap virtual file, the final state of the heap contains the values $[X, W]$ because four bytes were written and only two were read.

\begin{verbatim}
fld = open("heap1", O_RDWR); /* Open an empty "heap" raw file */
ioctl(fd, PUSH_INPUT, "extract_heap"); /* Add transformations */
ioctl(fd, PUSH_OUTPUT, "insert_heap");
write(fd, "WXYZ", 4);      /* heap = [Z, Y, X, W] */
read(fd, &string, 2);     /* string = "ZY"; heap = [X, W] */
close(fd);                /* Pop transformations; heap = [X, W] */
\end{verbatim}

Figure 1: Sample code using the \textit{heap} file structure.
A user creates a persistent virtual file through a transformation link, an extension of a symbolic link. A conventional symbolic link is a file that names another file. When accessing a symbolic link, an ordinary file system accesses the file named in the link. In addition to this, the Mona symlink system call copies configuration information to the link's inode. When the file system follows a transformation link to the underlying file, it also reads the additional information stored in the link that indicates the transformations to apply to the data. For example, a file named foo might be a transformation link to a raw file named foo.gz, with appropriate transformations. Opening foo automatically builds the transformation network to decompress and compress data.

The lnx (link transformation) utility binds transformations to a persistent virtual file. This utility takes a raw file name and a link name as arguments (like the ln utility). A user specifies which transformations to use on input and output streams with the -i and -o command line options, respectively. As an alternative, a user can call lnx with the -f option which reads equivalent information from a file. This is especially useful for complex networks.

The Mona file system, with the exception of the open and close calls, treats a transformation link as a conventional symbolic link. A transformation link can point to either a raw file or the output of another link. The standard ls command displays information about a transformation link, as shown below.

```
% lnx raw_heap heap -i extract_heap -o insert_heap
% ls -l
lwrxrwxw 1 hrk hrk 58 Mar 1 7:43 heap -> /mona/raw_heap
    extract_heap
    insert_heap
-rw-r--r-- 1 hrk hrk 42 Mar 1 7:42 raw_heap
```

In this example, a user creates a transformation link named heap to a raw heap file structure. This link uses the extract_heap and insert_heap transformations to read and write, respectively, to the heap raw file.

Transient and persistent virtual files are complementary features in the Mona file system. Although they are functionally equivalent, a user may desire one over the other depending on the context of an access. Transient virtual files present a process with fine-grained control over opened files. Persistent virtual files change the semantics of a file for all processes in the system. Once created, both transient and persistent virtual files are accessed through the conventional Unix file interface.

3.4 The Export Transformation

It is often necessary to perform transformations outside of the kernel. First, it greatly simplifies security concerns. The kernel operates in privileged mode, so arbitrary user-defined code cannot be executed in the kernel for obvious security reasons. Second, exporting code outside the kernel facilitates scheduling. A kernel thread is not preempted; therefore, a kernel transformation cannot execute for long periods of time without explicitly handling the scheduling of concurrent processes.

Consequently, the Mona file system provides the export transformation to safely execute untrusted or computationally expensive transformations outside of the kernel in user space.\(^3\) Figure 2 illustrates a transformation network generated by connecting a combination of kernel and user-level transformations. A and B are kernel transformations, but α is a user-level transformation. Therefore, the file system inserts the export transformation between A and B. This transformation passes its input data across the kernel-user boundary to a user-level transformation network that runs in user space. When transformation α completes, the output data returns to the export transformation which then passes it on to transformation B. Section 4 describes the mechanics of this process in detail.

Supporting both kernel and user-level transformations makes developing kernel transformations easier. Transformation code can be developed and debugged in user space first and then inserted into the kernel. This saves time because the compile-test-debug cycle is greatly reduced. There is no need to insert and remove a kernel module (and mount and unmount file systems) because a user transformation is an ordinary relocatable object file. In addition, a symbolic debugger and printf calls can be used to debug a user-level

\(^3\)The export transformation refers to the interface responsible for executing transformations in user space. It should not be confused with a transformation that is exported to user space for execution.
transformation. Lastly, a user-level bug causes a core dump, whereas a kernel bug causes a kernel panic. Developing in user space does not obviate the need for testing a transformation in the kernel, but it does reduce the time spent debugging.

The Mona file system is extensible for unprivileged users as well as privileged users. Although both cases have identical interfaces from the perspective of a transformation, there are differences in a transformation’s location and capabilities. A super-user compiles a kernel transformation directly into the file system module. Therefore, the kernel loads a kernel transformation when it inserts the Mona module. Alternatively, an unprivileged user can implement a transformation by compiling the function and appending it to a shared library.

Kernel and user-space transformations also differ in capability. A kernel transformation runs in the kernel with the same privileges as system code. Consequently, a kernel transformation can directly access kernel data structures to determine the precise state of the system at runtime. A user-space transformation does not have access to kernel structures. However, it does have high-level libraries available. For example, a user-space transformation can use libc or any of the other standard C libraries. In contrast, a kernel transformation only has access to a reduced set of functions in a limited kernel library.

The Mona file system surpasses the capabilities of conventional file systems. It accomplishes this through transformations, which operate on the data streaming either to or from an individual file. As a result, extended file operations such as implicit file structures, context-sensitive triggers, and pseudo-file services provide enhanced functionality. In addition, the export transformation complements the base file system by pushing user-defined transformations outside of the kernel and into user space.

4 Mona Implementation

The implementation of the Modify-on-Access file system is straightforward and efficient. This section describes the components of the Mona implementation. It first presents general information and then discusses the novel aspects of the file system. The section concludes by describing the implementation of the export transformation.

The Mona file system evolved from the standard Linux file system, ext2[3], and maintains complete compatibility with it. As a result, the ext2 and Mona file systems are interchangeable. The Linux kernel can mount an ext2 partition as a Mona partition without any modifications. Similarly, the kernel can mount a Mona partition as an ext2 partition. However, attempting to follow a transformation link in ext2 results in a “File not found” error. Likewise, using the ioctl system call with Mona specific options will return an error in the ext2 file system. Although the Mona file system contains additional meta-information in transformation links, removing the link within either Mona or ext2 will also remove any non-standard file system structures.

The Mona file system operates on data during reads and writes by inserting modular transformations within the input and output data streams. A conventional file system only supports the copy operation on data flowing to or from a disk. The Mona file system extends this behavior by supporting any arbitrary
data stream operation. Each file in the Mona file system has associated input and output streams which are used on reads and writes, respectively. However, unlike a conventional file system, transformation networks guard these streams. It is possible, however, for a Mona stream to have no associated transformations. In this case, like a standard file system, the Mona file system performs the *copy* transformation directly to or from user-space buffers.

Mona generalizes standard file systems by performing computation on data during transit between the kernel and secondary storage. Figure 3 illustrates three transformation networks. In this example, all three input networks share a common sink, a user process that initiates file accesses. In contrast, the output networks each have a distinct sink, a base file on disk. File system buffers in the kernel lie between the user process and the disk. The top network shows two transformations on the input stream and two corresponding inverse transformations on the output stream. Transformations that have natural inverses, such as compression/decompression or encryption/decryption fall into this class. However, not all transformation networks are symmetric on input and output streams, as shown by the asymmetric middle transformation network. The bottom transformation network illustrates an unguarded file, which is equivalent to conventional file systems.

Kernel transformation code is part of the downloaded Mona module. Additionally, the module contains a table of all kernel transformations. This table is searched on an open call and information in the table is used to construct the transformation network.

4.1 Transformation Implementation

A transformation is a modular operation on a data stream. It takes input data, performs an operation using the data, and then passes data to its output. Correctly-behaving implementations of the transformation mechanism can be optimized for either speed or ease of use. The Modify-on-Access file system chooses a transformation implementation that is optimized for passing data through a transformation network quickly. This means that implementing transformations is more complex than it could be, but it is still straightforward.

The Mona file system implements a transformation as a function call, as shown in the prototype below:

```c
int t(T_info * thisXform, char *inBuf, int inBufSize,
     int *networkStatePtr)
```

The first argument is a pointer to a data structure that maintains static information about the transformation
typedef struct transform_info {
    T_info *next;
    int (*fn)(T_info *, int, char *, int *);
    struct inode *i_node;
    struct file filp;
    void *private_data;
    int state;
} T_info;

Figure 4: The transformation information data structure.

between invocations. The next arguments are an input buffer and its size. The final argument is a pointer to a state variable, shared with the other transformations in the current transformation network, that maintains the current state of the network. The transformation returns an integer specifying the total number of bytes that were output.

Each instance of a transformation contains a private copy of the data structure shown in Figure 4. This contains a pointer to the next transformation in the list, a pointer to the transformation function, the inode and file pointer of the base file, a private pointer for any persistent storage requirements, and the current state of the transformation.

A transformation passes information downstream (the direction the data is flowing) by calling its neighboring transformation on a buffer of data. Transformations update the state of the network by modifying a variable that is passed to them as a pointer. This mechanism passes state information both up and down a network. Furthermore, a transformation neither knows nor cares whether it is on an input or output stream.

The file system does not require that transformations conserve the size of data. For example, a compression transformation generates less data than it consumes. If it needs additional data, the transformation sets the state variable with a request for more information from its calling procedure before passing the final compressed data down the network. Likewise, a decompression transformation will generate more data than it consumes. It may need to pass data downstream in multiple sets before returning with a request for more data. As a result, a transformation network may store excess data in the network between reads and writes. If this is the case, the file system first reads or writes buffered data and only refills the network after it is completely empty.

A stream with size-conserving transformations does not require any additional storage. First, data is copied into a page and passed to the first transformation in a network. Ideally, the transformation then modifies the buffer and passes it directly to the next transformation without using extra storage. However, a transformation is allowed dynamically to add buffers. For example, the decompress transformation may generate more data than it can hold in the buffer passed to it. As a result, the transformation dynamically creates a second buffer to hold the overflow data which will be required during subsequent invocations.

4.2 Transformation Networks

The Mona file system builds linked lists of transformation data structures that correspond to the series of transformations associated with a given file. This framework connects each transformation with its neighbor. Furthermore, the mechanism requires very little space in the kernel.

Each instance of a transformation contains the corresponding information structure introduced in Figure 4. This contains a pointer to the next transformation in the list, a pointer to the transformation function, the inode and file pointer of the raw file, a private pointer that transformations use for any persistent storage requirements, and the current state of the transformation. A process indexes the list of transformation networks with a file descriptor, which then follows a link to the start of the corresponding input or output network. Figure 5 illustrates this framework by showing the relationship between processes and transformations.

The straightforward implementation of a user-level transformation network would create one process per transformation. However, this will have significant overhead due to the cost of process creation and the
management of kernel resources, as in the case of Unix pipes. Furthermore, each process in the pipeline must read and write the data, which is a great hindrance to performance. Moreover, the overhead grows with the number of processes, which discourages the modular use of transformations. Instead, the Mona file system merges transformations into one process. Because transformations send data to other transformations without crossing process boundaries, the Mona file system avoids costly kernel context switches implicit to Unix pipes [22].

4.3 Transformation Example

The following section describes the application programmer’s interface (API) necessary for designing transformations. It also gives the complete source code for a simple, yet general, transformation that performs a modification on each byte of data that it reads.

The implementation of transformations is optimized for passing data quickly through a transformation network. Ideally, a transformation takes an input buffer, modifies it, and then sends the same buffer to the next transformation. However, many transformations do not conserve the size of data and require a more intricate protocol that dynamically services requests for additional data that is generated by overflows.

A transformation must have a means of communicating with its calling function. First, it must be able to report how many bytes of data were sent through the entire network. The integer return value handles this task. Second, it must be able to indicate whether it has more data to send downstream. The state parameter handles this task. The state can either be $XFORM\_READY$, denoting that all computations have completed, or $XFORM\_FULL$, indicating that the current transformation has buffered data and should be called again before additional data is introduced into the transformation network. Because the state parameter is passed as a pointer, modifying it locally within a transformation will update the state variable for the network.

Figure 6 lists the source code for the upper2lower transformation. This transformation takes an input buffer and scans it one byte at a time, converting any ASCII upper case characters to lower case. The transformation is general and can be compiled either within the kernel (as part of the file system module) or within a user-defined shared library.

The transformation begins by converting the first $inBufSize$ bytes of the buffer from upper case to lower case. Then it returns the number of bytes sent downstream. In this case, the transformation does not modify the state of the transformation network (via the $networkStatePtr$ variable). However, if the transformation
int upper2lower_xform(T_info *thisXform, char *inBuf,
    int inBufSize, int *networkStatePtr) {

    int i;

    for(i = 0; i < inBufSize; i++) {
        if ((inBuf[i] <= 'Z') && (inBuf[i] >= 'A'))
            inBuf[i] += ('a' - 'A');
    }

    return inBufSize;
}

Figure 6: Source code for the upper2lower transformation.

![Diagram](image)

Figure 7: Execution model of export transformation.

did not consume all its input, it would set networkStatePtr so that Mona will invoke the transformation again in order to guarantee that the network flushes overflowing data from a previous access before reading in new data.

In the Mona model a transformation that does not consume all its input is invoked again. Mona user-level transformations are implemented as functions. The above example is the most trivial of transformations: it conserves size and outputs data in the same order as received. 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. For example, a typical decompression transformation firsts read a dictionary without outputting any data. Then, because it writes more than it reads, it might fill an output buffer before exhausting all the input. The transformation must be invoked again to flush unused input and unused input must be remembered between transformation invocations.

4.4 The Implementation of export

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

A user process initiates a transformation network through either a read or write system call. The
network then passes data while modifying it appropriately at each transformation stage. However, rather
than performing the modification locally, the export transformation passes its input data and the name of a
user transformation to a daemon running in user space.

The Mona implementation uses the 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 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 7.
This figure provides a detailed 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.

For the export transformation to be safe, it must execute with proper permissions and maintain the
integrity of the kernel. The Mona file system keeps proper permissions by executing transformations in user
processes. Before a child of the Mona daemon executes a a user-level transformation, it changes user and
group IDs (UID and GID) for safety. 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 runs with the permissions of the user accessing a virtual file, then it executes operations under the guise of the user accessing the file, which would allow malicious transformation
authors access to unsuspecting users’ resources. Similarly, setting a transformation’s permissions to that of
the owner of a base file is also unsafe. For example, the owner of a virtual file could point a transformation
link to /etc/passwd, which is owned by root, and have a transformation that executes with root permissions.
Therefore, the Mona file system changes the permissions of the process executing a user-level transformation
to those of the user who created the virtual file. Consequently, a transformation is only capable of doing
what its owner can do.

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 user process 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 that read and write streams.
The option that allows kernel access on the open system call only allows access by processes owned by root
and exits with an error message for any other user. The remaining enhancements of the ioctl system call
allow any user to request and submit transformation data. However, the file system assigns a unique 64-
bit identification key to each stream and requires a valid key before servicing an ioctl system 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 rapidly
approaches zero.

Unlike kernel transformations, a user-level transformation is not one of a fixed set downloaded into the
kernel. User-level transformation code is a library file that is dynamically loaded. 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 upper2lower.c -shared -o upper.so
mv upper.so /lib/mona
```

In order to guard a file, both the name of the transformation and the name of the library file are needed.

The export transformation overcomes the two limitations of kernel transformations. First, the Mona
deamon allows an unprivileged user to extend file system capabilities in a secure manner. All user-defined
code is executed outside of the kernel with the permissions of the user who instantiated the transformation.
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.

In summary, the Mona file system implementation is efficient and extensible, yet maintains a modular
design that is compatible with ext2. By instantiating transformations as function calls and transformation
networks as successive function calls, the implementation executes operations on data quickly with minimal space overhead. Furthermore, through the export transformation, the file system extends user capabilities by supporting arbitrary user-defined operations on a per-file basis.

5 Results

The Modify-on-Access file system provides an extensible environment that simplifies user tasks. However, for the system to be practical, adding this functionality must have little overhead. In addition, Mona must demonstrate an overall speedup for applications which leverage the fine-grained control provided by the file system.

This section presents the micro and macro benchmarks of the Mona file system. It will be shown that transformations introduce an acceptable amount of overhead and have little adverse effect on system performance. Furthermore, Section 5.3 and Section 5.4 present several applications that exploit the potential for increased performance and functionality.

Tests were conducted with the following parameters. The microbenchmarks were run using the Linux 2.4 kernel on both the Pentium III and PowerPC architectures. The Pentium III runs at 650MHz with a 256k backside cache, has 512MB of RAM, and uses an IDE hard disk. The Power Macintosh G3 runs at 300MHz, has a 512k backside cache, 160MB or RAM, and an IDE hard disk. The macrobenchmarks were run using the Linux 2.2 kernel on the same Power Macintosh G3 with a SCSI disk, and a Pentium Pro running at 200 MHz with 64MB of RAM and an IDE hard disk. All micro-benchmarks were averaged over 10,000 iterations on an unloaded system in order to amortize the cost of disk accesses and pinpoint costs attributed exclusively to the file system.

5.1 Micro-benchmarks

System calls in the Mona file system incur overhead not found in conventional file systems. There are two costs associated with executing transformations. The first is determining whether a transformation needs to be applied. The second is invoking the transformation. The cost of transformation execution is not file system overhead because it is part of the application. The following two sections examine these costs for both kernel and user-level transformations.

5.1.1 Kernel Transformations

Table 1 shows the results of three system calls in four different configurations on both test machines. The first, ext2, forms the baseline for performance because Mona is derived from it. The remaining four tests use the Mona file system. The second test shows the time for accessing a file that has no transformation associated with it, i.e., it is unguarded. The next three show the time for three kernel transformations: identity, xor, and op100. The first transformation merely copies its input to the output, the second performs a bit-wise XOR on every byte, and the third applies a bit-wise XOR one hundred times to simulate a more intensive computation. Each test yielded highly consistent results. Except where noted, the standard deviation fell within 1% of the average time in most cases. The largest deviations in our results came when testing the
<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th></th>
<th>write</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kernel</td>
<td>user</td>
<td>ratio</td>
<td>kernel</td>
</tr>
<tr>
<td>identity</td>
<td>30.57</td>
<td>143.23</td>
<td>4.7</td>
<td>10.18</td>
</tr>
<tr>
<td>xor</td>
<td>55.77</td>
<td>170.12</td>
<td>3.1</td>
<td>35.41</td>
</tr>
<tr>
<td>op100</td>
<td>1304.03</td>
<td>2686.70</td>
<td>2.1</td>
<td>1283.65</td>
</tr>
</tbody>
</table>

Table 2: Comparison of execution time (in μs) between kernel and user-level transformations on the Pentium III architecture.

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th></th>
<th>write</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kernel</td>
<td>user</td>
<td>ratio</td>
<td>kernel</td>
</tr>
<tr>
<td>identity</td>
<td>56.21</td>
<td>236.76</td>
<td>4.2</td>
<td>28.78</td>
</tr>
<tr>
<td>xor</td>
<td>117.59</td>
<td>297.05</td>
<td>2.5</td>
<td>90.46</td>
</tr>
<tr>
<td>op100</td>
<td>6871.43</td>
<td>25709.64</td>
<td>3.7</td>
<td>6852.65</td>
</tr>
</tbody>
</table>

Table 3: Comparison of execution time (in μs) between kernel and user-level transformations on the PowerPC architecture.

open system call on the PowerPC, where we observed standard deviations of 3.9% and 2.1% for the identity and xor transformation tests, and 3.2% when testing the read system call for the xor transformation.

The cost of an open is independent of file size. Table 1 shows the times for open on a single transformation. Since the xor and xor100 transformations are simple transformation networks of length one, their times are the same as that of the identity transformation, and are, therefore, not repeated. The cost of an open grows approximately linearly with the length of the transformation network. This makes sense, because these transformations are not passed any data on the open system call, and therefore are effectively the same as the identity transformation. However, for read and write, the cost is proportional to the size of the file. Table 1 shows the cost per page of data (4KB) read or written. Although the overhead for read and write grows with the length of the transformation network, it is minor compared to overhead due to file size.

The first cost is determining whether to apply a transformation. If the access is unguarded, no transformation is applied and the cost is the execution of an additional conditional expression and accessing some system data. This overhead is shown in Table 1. It is the difference between the base file system cost, ext2, and unguarded. This overhead is usually insignificant, ranging from 0 to 3% (a fraction of a microsecond).

The second cost is invoking a transformation. For a non-trivial transformation, this cost is dominated by the operation, which is transformation-specific. The cost of such an operation is more accurately charged to the application rather than to the file system.

The open system call involves finding the transformation in a table, which can nearly double the cost of an open (relative to ext2). Table 1 shows opening a persistent transformation link, which has overhead due to reading the structure of a transformation network from disk and creating it within the kernel. Although the overhead with one transformation is significant in relative terms (up to 86%), it is small in absolute numbers: 2 and 6 microseconds on the Pentium III and PowerPC, respectively. The read performance is significantly less than ext2 and the write performance because we perform an extra copy on reads to maintain consistency.

5.1.2 User-level Transformations

A kernel transformation has less overhead than a user-level transformation. As a result, opening a user-level transformation is approximately twice as expensive as opening a kernel transformation. First, the export kernel transformation is applied, which is identical to opening a kernel transformation. Then the name and any other arguments are passed into the export transformation, which accounts for the additional cost.

Table 2 and Table 3 show the execution times of kernel and user-level transformations for the Pentium III and PowerPC architectures. These tables present the same three transformations as discussed above. The ratio column represents the relative speedup that would occur if a user-level transformation were placed
in the kernel. The time for read and write are on data sizes of 4KB. The standard deviations of the above tests are all less than 1% of the mean.

As the complexity of a transformation increases the one-time cost of crossing the user-kernel boundary is better amortized. The execution time of a user-level transformation can be an order of magnitude greater (in the case of the PowerPC) than the kernel transformation for the identity transformation. However, this cost drops to only a factor of 2 for the more complex op100 transformation. There is a large increase in absolute time of op100, which is due to preemptive scheduling. The transformation executes long enough for the user-level daemon process to be re-scheduled. On the other hand, complex kernel transformations easily can cause a process to exceed its time-slice, which violates the scheduling policy and leads to inefficiencies such as increased response time.

5.2 Macro-benchmarks

This section shows the effect on aggregate system performance resulting from the overhead described in Section 5.1. We present the performance of three different benchmarks.

PostMark: A modern file system benchmark from Network Appliances.

Andrew: A file system benchmark from the Andrew File System [12].

Kernel compile: A compilation of the Mona module.

These three benchmarks are quite different. PostMark is designed to evaluate the performance of file systems [13]. It consists only of file operations. We use the default settings with the exception of increasing the number of transactions in order to get a longer test. This represents the load on a file server that provides services such as electronic mail, net news, and commerce service, which depend on enormous numbers of relatively short-lived files. The kernel compile benchmark has a higher computation to I/O ratio than PostMark. The Andrew benchmark sits between the above: it has phases that only perform file operations, followed by a compilation.

Table 4 presents the results of the three benchmarks. The PostMark benchmark consists almost entirely of file operations. Consequently, Mona has greater overhead running Postmark than the other benchmarks. Both the Andrew and compile benchmarks have negligible overhead associated with the Mona file system. This is due to relatively lower I/O demands. The difference between the base (ext2) and Mona unguarded shows the cost of performing ordinary accesses in Mona. Executing the trivial xor\(^4\) transformation costs at most 7.5% and roughly half of this cost is associated with the application, not the file system. This shows that doubling the overhead of individual file system operations (as in xor) has a small net effect on an application overall. There is an anomaly for the unguarded PostMark test. On both the Pentium and the PowerPC, unguarded takes longer than identity. However, the latter test involves an additional call to the identity transformation. Therefore, it should not run faster—which is the case for the other benchmarks (Andrew and kernel compile). In spite of an in-depth examination, we have not discovered the source of this anomaly.

\(^4\)In this test the xor transformation is performed twice, which results in no net changes to the data.
<table>
<thead>
<tr>
<th></th>
<th>Mona</th>
<th>ext2</th>
</tr>
</thead>
<tbody>
<tr>
<td>best case</td>
<td>154</td>
<td>122</td>
</tr>
<tr>
<td>average case</td>
<td>174</td>
<td>838</td>
</tr>
<tr>
<td>worst case</td>
<td>195</td>
<td>1700</td>
</tr>
</tbody>
</table>

Table 5: Speeding up file lookup (times in microseconds).

5.3 Examples of Increased Performance

There are two reasons for extensibility: increased performance and increased functionality. This section presents applications in which Mona speeds up performance. Section 5.4 complements this by demonstrating applications with increased functionality. The results presented here are from the Pentium Pro test machine.

CGI scripts The first example of performance speedup is pushing the operations of a typical CGI script into the kernel. The performance improvement is no surprise because it is obvious that it is more efficient to execute a few more operations concurrent with a file transfer than to fork another process. Nevertheless, this is a big improvement in performance. The typical CGI counter script was mimicked in a kernel transformation. We saturated an Apache web server with requests for a file containing a hit count. The web server processed 4.7 times as many requests per minute when accessing files with a Mona transformation than when accessing files on an  ext2  partition using a CGI script. In this test, the server processed 17,240 requests/minute in Mona versus 3,708 requests/minute in  ext2.

We compare a kernel transformation running with full permissions to a CGI script running in user space as user nobody. Although this may appear to be an unfair comparison, CGI scripts and kernel transformations are both implemented by a privileged user and must be validated before use. The system administrator must decide if the extra performance is worth the risk. Moreover, unlike a CGI script, a transformation is not limited to use in web servers. The same transformation could also implement a reference count on an FTP server or a user’s .plan file.

Hashing name lookup The  ext2  file system uses a linear search to find file names. Consequently, the time to locate a file grows with the number of files in a directory. This discourages building flat directories. However, some applications are better served by a flat hierarchy [29]. For example, a usenet news server places all articles for a particular newsgroup into one directory. Because news files tend to be small, the lookup is particularly costly. This illustrates how to overcome a performance problem that does not merit a drastic change to the kernel.

We have implemented a transformation for the name lookup that hashes names into several buckets, which are subdirectories. This lowers the cost of name lookup. There are numerous parameters that can be adjusted for this test, such as number of files, size of files, percentage of reads and writes, etc. Because we are interested in the lookup we use small files. Furthermore, because we are comparing hashing to linear search, the speed of access depends where the target file is located. Therefore, we show best, average, and worst case performance for each method in Table 5.

The test bed for this is 10,000 small files. The results are highly dependent on the distribution of the file names. In the hash test, these files are perfectly hashed over fifty subdirectories. Thus, the longest linear lookup is 200 files for the hash or 10,000 files of the straight linear lookup. Tests show that the hash lookup is about 5 times faster in the average case. Different tests produce larger or smaller wins for the hashing lookup, depending on the number of files and the effectiveness of the hash. These results are validated by the usenetfs that is implemented in Wrapfs [29], which reports a 22% overall speedup for a similar optimization.

5.4 Examples of Increased Functionality

Mona provides the ability to exploit application-specific customizations. As a result, significant speedups can result, as shown in the previous section. In addition, this extensibility enables an application to easily
enhance the capabilities of the file system. The sample transformations this section introduces were relatively easy to implement, yet provide substantial improvements in system functionality.

**FTP Transformation** One Mona user-level transformation allows users to navigate a remote FTP site as if it were part of the local file system. A user executes ordinary shell commands (e.g., cd, ls, cp) to manipulate remote files, which eliminates the need for explicit downloads. A similar transformation allows users read and write access to files in tar archives without the need to manually extract and re-archive those files. Other such transformations would be equally easy to construct.

**Command Transformation** Constructing transformations as executables using the command transformation greatly simplifies transformation creation. This approach is an example of the flexibility that the Mona system affords the user, even within the system. Just as the export transformation is a kernel transformation built to allow transformations to be executed in user space, the command transformation is a user-level transformation built to allow executables to be associated with data streams.

The command transformation function takes as an argument the name of an executable object file or shell script. It forks a new process for the executable, redirects standard input and standard output of the new process to shared pipes, and executes the selected file. The command transformation then passes data from the input stream to standard input of the newly created process and copies data to the output data stream from standard output of the process. The user can execute an existing program, write a normal program and compile it as for regular use, or create a shell script that ties two or more such programs together. This provides the user with the opportunity to associate the actions of existing Unix utilities with data streams or to create new utilities if appropriate ones do not exist. This ease of use is, however, accompanied by a performance hit. Forking new processes is much more costly than executing a function in a library, and this approach requires at least two forks.

Shell transformations are especially useful when duplication of existing code would be necessary to accomplish the goal. If a user is interested in knowing when a file is accessed, the mail program can be placed in a command transformation. Text processing tools such as grep, sed, and awk can be placed on streams to filter data. Additionally, the rcs utility provides implicit version control. In short, any program can be associated with a data stream using the Mona file system.

In summary, the Mona file system provides the opportunity to increase performance through application-specific customization and extend the capabilities of an application. Mona incurs little overhead and has benchmark execution times comparable to conventional file systems.

### 6 Conclusions

This paper discusses the design and implementation of the Modify-on-Access (Mona) file system. It explains how to use Mona, an active file system that performs computation on behalf of processes during I/O. Section 5 shows that the overhead of a kernel transformation is negligible. In addition, we present applications that show performance improvements of up to 5 times by customizing file operations.

Additionally, this paper presents the Mona programming model. It describes modular, stream-oriented operations, called transformations, that the file system applies to an I/O stream during a file access. It also shows that associating transformations with a file provides the ability to create application-specific file structures and file semantics, a feature that conventional file systems lack. We have presented several examples in which extending the structure and semantics of a file simplifies applications and benefits both the system programmer and ordinary user.

Our implementation of the Mona file system provides a novel combination of granularity, modularity, and multiple levels of execution. Mona supports transformations on a fine-grained, per-file or per-access basis. In addition, transformations are modular and can compose larger operations. Finally, Mona can execute a transformation within the kernel, from a user-level library, or as a shell script or native binary. As a result, a user has the flexibility to choose an appropriate level of performance, safety, and ease of use. The flexibility in all three areas distinguishes Mona from previous extensible file systems.

17
The extensibility of the model has been validated twice, with two major enhancements that were provided without changing the model. Kernel transformations form the basis of our system. The export transformation is a standard kernel transformation that provides user-level transformations. Similarly, the command transformation is an ordinary user-level transformation that executes shell scripts or native binaries.

The Mona file system presents several opportunities for improvements. As a result, we are expanding our implementation in three major areas. First, we are continuing the development of a suite of kernel and user-level transformations. Second, we are developing random-access protocols for Mona that complement stream-oriented transformations. Third, there is some opportunity to avoid kernel bloat and increase the flexibility of Mona kernel transformations by allowing transformations to be downloaded into the kernel at run time without removing and inserting the module. Furthermore, we plan to release the file system source in the near future.

The presence of the Mona file system adds an interesting design tradeoff. An application can place functionality in a kernel transformation, a user-level transformation, or in the application itself. The tradeoffs between kernel and user-level have been discussed above. The tradeoffs between implementing functionality in the application or in the file system are similar in many ways. We have demonstrated opportunities for increased performance by performing simple tasks in the file system. Also, if the functionality is either already in the file system or predicted to be widely used, using Mona transformations can simplify the programming task. On the other hand, it is easier in some cases to forego learning the transformation interface to implement simple operations outside of the application. In short, there is no hard and fast rule, and users should decide what is most appropriate to their given situations.

Lastly, “intelligent” devices, those with integrated processing capabilities, will be the standard in the future. Today, processors are being incorporated in disk drives [1, 6], communication devices [20, 27], and even memory [19, 14]. To effectively harness the power of these intelligent devices, an appropriate systems programming model is needed. The model should:

- share the workload between applications and intelligent devices,
- support the increased functionality of intelligent devices, and
- do the above dynamically and/or adaptively.

Clearly, the programming model must support generic use of intelligent device operations, but that is not sufficient. To fully harness their potential, it must also support application-specific operations, and it must do so without being a burden to use. The model must be highly flexible so that the system and the devices can adapt dynamically to changes in the environment. The Mona programming model is one realization of such a programming model for use with intelligent disk drives.

We have a prototype Mona file system running on an “intelligent” disk drive [16]. This system offloads operations to the device, freeing the main CPU for other activity. The Active Disk [1] simulation demonstrates that executing streaming operations locally on a disk can significantly increase disk performance.

References


