The Design and Implementation of the Exported Procedure Call

H. Richard Kendall and Vincent W. Freeh

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
{rkendall,vin}@cse.nd.edu

May 30, 2001

Abstract

This paper describes the exported procedure call, a mechanism that pushes computation out of the operating system kernel and into user space. It supports a simple, secure model for system extensions. An exported procedure call incurs overhead crossing the kernel-user boundary, but once in user space, it has greater security and usability and is significantly simpler. This paper demonstrates the capabilities of the exported procedure call by discussing two implementations. One is the Modify-on-access (Mona) file system and the other is the Magi device interface. Mona and Magi use the exported procedure call in order to safely execute untrusted or complex system extensions. This paper shows that in situations where raw kernel performance is not paramount, the exported procedure call is desirable.

1 Introduction

Much debate has centered around the proper place for the user-kernel boundary in an operating system. Lowering the boundary, such as in a micro-kernel, provides greater extensibility, security, and customization [5, 2, 4, 14, 15, 7]. However, as the boundary is lowered performance tends to decrease due to the overhead of additional boundary crossings. Today, monolithic kernels proliferate, while micro-kernels are still in the fringes.

Because configurability is important, newer monolithic operating system kernels provide some configurability. For example, modern operating systems support inserting or removing a kernel module at runtime. However, a module, which runs in the kernel in supervisor mode, is an extension to the operating system kernel and is undesirable for several reasons. First, kernel code is harder to develop and debug because one cannot use standard libraries or debuggers. Second, uninterrupted sequences of code must be kept short in the kernel. Third, kernel code runs in supervisor mode with minimal security. Even though kernel code has the above drawbacks, when performance is paramount, a service needs to be in the kernel.

On the other hand, many operations are not on the critical path, such as a latency-bound task. In this case, the overhead of pushing the operation out of the kernel (an extra boundary crossing

*This research was supported by NSF grant CCR-9876073 and the Arthur J. Schmitt fellowship.
or two) is hardly noticeable. Consequently, the added benefits of executing the operation outside
of the kernel are more than justified. As an analogy consider extending a kernel with a device
driver versus extending the Apache web server with a module. The latter is significantly simpler,
even when debugging. Few programmers will ever write even a trivial device driver, but a
sophomore computer science student could write an Apache module. Furthermore, Patterson argues
that performance is mostly solved, the current challenges are reliability and serviceability [17].

An exported procedure call (EPC) resembles a remote procedure call in which an operating
system kernel is the client and a user-level process is the server. During an EPC, the operating
system kernel will invoke a procedure performed in a user-level process, then wait for it to return.
These stages, invoke and return, correspond to sending a request across the user-kernel boundary
and receiving the response. A user-space server continuously passes through get, service, and put
stages during a transaction. In these stages, a process gets a request from the kernel, services the
request, and puts the result back into the kernel.

This paper describes the design and implementation of the exported procedure call. It also
discusses mechanisms for and tradeoffs of pushing tasks out of an operating system kernel and into
user-space processes. We present two systems, the Modify-on-access (Mona) file system [8, 10] and
the Magi device interface [9], that implement variations of the EPC to safely support user-defined
extensions.

The remainder of this paper is organized as follows. The next section discusses the general
mechanisms for crossing the user-kernel boundary. Section 3 examines the generic export model
and discusses how the Mona and Magi implementations support the model. We analyze the Mona
and Magi implementations with respect to performance and security in Section 4. Lastly, Section 5
presents conclusions and future work.

2 Overview

Executing in user space is a fundamental micro-kernel idea. There are several advantages to the
micro-kernel design. Services use mechanisms provided by the micro-kernel like any other program.
Further, a malfunction in a service is isolated like a malfunction to a user program. Finally, a
micro-kernel system is more flexible and tailor-able: different implementations of the same service
can easily co-exist. A micro-kernel design is far superior to a monolithic kernel in terms of software
engineering [12, 13]. However, most micro-kernels suffer from poor performance, which has
prevented their wide-spread adoption.

The exported procedure call (EPC) provides many of the above advantages without forcing the
adoption of an entirely new operating system. There is some overhead to exporting an operation.
Therefore, an EPC is most useful for latency-bound operations, which are less effected by local
performance. Executing in user space provides greater security and flexibility. It is also simpler
and more reliable. Furthermore, complicated features and operations tend to appear sooner in user
space than in the kernel. EPC can bridge this gap.

The largest obstacle to exporting code from the kernel to user space is developing a communica-
tion mechanism for crossing the user-kernel boundary. Some of the difficulties include maintaining
security and controlling overhead. This section provides an overview of several possible implementa-
tions. It discusses the advantages and disadvantages of each. The discussion begins with the
most simple and proceeds to the more complex.
Each of the mechanisms discussed below uses a helper process, which is in user space. This process completes an operation on behalf of the kernel. The helper process gets a request with a system call that blocks the process. The kernel unblocks the helper process when there is work to be performed. Therefore, a helper process starts servicing a request when it returns from the blocking system call. The information needed by the helper is supplied in the parameters of the get system call. When the request is serviced, the helper process replies with a system call, which may be the same as the blocking one.

Shared File  The file system is the simplest mechanism for communicating across the user-kernel boundary. Consequently, it was the choice for the initial prototype of the Mona file system. Two dedicated files provide communication channels between user and kernel: \texttt{kernel2user} and \texttt{user2kernel}. The former is for messages written by the kernel and read by the user-space daemon. The latter is for communication in the other direction. Implementing communication via shared files is extremely easy and results in very fast prototype development. However, using a shared file becomes cumbersome as the system scales. The system either creates more such dedicated files, or these files or it multiplexes between the many helper processes accessing the two dedicated files. The methods below are much easier to scale. Additionally, performance using shared files is roughly an order of magnitude slower than the next three alternatives described.

New System Call  Creating a new system call is the standard method for providing new cross-boundary communication requirements. Most operating systems have a well-defined interface for adding a new system call to the kernel. However, adding a system call involves modifying the operating system kernel source tree. As a result, it is necessary to create non-standard patches to the kernel source code, recompile the kernel, and reboot the system. Although this may be acceptable in some instances, many developers prefer that the operating system source code remain as close to the standard as possible.

Existing System Call  The third technique for crossing the user-kernel boundary is enhancing an existing system call, this is used by the Mona file system. In the case of Mona, the \texttt{ioctl} call is extended, adding four new options to the existing \texttt{ioctl} system call to handle communication between user-space servers and the kernel. Because the implementation embeds all new kernel code within the Mona module, the main kernel source code requires no changes to support the additional cross-boundary communication. Moreover, the extension can be added (or removed) at runtime without rebooting the kernel. However, this technique will not work unless an existing system call is available for modification. The Stream I/O system extends the \texttt{ioctl} call [16].

Device Driver  A device driver is a standard extension to the kernel and can be created to communicate with a user process. For example, the Magi device interface uses a device driver to pass messages across the user-kernel space boundary. An EPC server process reads and writes to the device \texttt{/dev/magi} using the existing \texttt{read} and \texttt{write} system calls. The device driver executes within the kernel and services user-space accesses. The device driver method is similar to the shared file technique described above in terms of how it is used. However, a device driver is able to process accesses much faster than an ordinary file. A device driver is completely self-contained and does not require any modification to the kernel source tree. In Linux, a user may dynamically insert a
driver module into a running kernel without recompiling the kernel or rebooting the system. This method is less intrusive than adding a new system call and more general than modifying an existing system call.

The exported procedure call modifies a current, existing operating system in order to provide the many benefits of user-level extensibility. This differs significantly from the research projects that create new extensible operating systems [5, 2, 4, 14, 15, 7]. As a result, the exported procedure call does not achieve the same level of performance as measured in overhead. However, within the application base for which EPC is designed—latency-bound tasks—the performance of EPC is comparable to the exotic methods simply because disk or network latency dominates.

3 Export Model

The Mona file system and the Magi device interface both implement an instance of a general exported procedure call model that pushes kernel code to user space. The following section examines the export model in detail and presents the Mona and Magi implementations. The section concludes with a discussion that compares the implementations and explains how they relate to the general model.

In the general exported procedure call model, the operating system kernel and a user-space EPC server pass through complementary execution stages, as shown in Figure 1. In the invoke stage, the kernel initiates a request to an EPC server running in user space. After passing a request up to user space, the kernel enters the return stage and suspends the calling process until a server sends back the result of the request. In user space, an EPC server reads a request in the get stage. The server then services the request and sends the result back to the kernel in the service and put stages, respectively.

3.1 Mona Export Model

Mona is an active file system that cooperates with processes to accomplish tasks. In Mona, an application pushes and pops transformations, stream-oriented data operations, onto streams of data flowing to or from a process. When an application performs a read or write on a stream containing a transformation, the file system executes the transformation on all data flowing down
the stream. As a result, Mona enables numerous file system extensions, such as compression or encryption.

The Mona file system supports in-kernel transformations that are pushed on to (or popped off of) files. In this sense, it is similar to streams [16]. However, Mona is a block-oriented system and it supports exporting computation out of the kernel [11]. The Virtual File System (VFS) in Linux abstracts many file system operations, such as open, read, and ioctl [1] in order to support multiple file systems easily and uniformly. Mona is derived from the VFS.

A conventional file system copies data between secondary storage and buffers in the kernel. The Mona file system enhances this model by enabling applications to insert operations into a data stream between a source and a sink. Figure 2 illustrates this mechanism. To the left of the figure, a user-space process accesses kernel buffers through read and write system calls. The operating system then either fills the buffers from disk or flushes the buffers to disk. Mona extends conventional file system functionality by applying transformations A and B to data flowing from disk and to disk, respectively.

The Mona file system allows a process to open a file and insert application-specific operations into the data streams flowing to and from the file. However, because streaming kernel extensions may be arbitrary user-defined code, Mona executes the extensions in user space to maintain system security. In order to push kernel code into user space, Mona implements a version of the general exported procedure call model described at the beginning of this section. In this model, the operating system kernel initiates an EPC that is serviced by a user-space process.

There are two types of EPC servers in the Mona implementation. The first server is a daemon process executing in user space. When the Mona file system first pushes an extension onto a stream, the daemon process is responsible for initializing the extension and spawning an instance of the second type of server, a helper process. On subsequent stream reads and writes, the helper process services accesses by executing the extensions assigned to the stream.

In the Mona implementation of the EPC model, the kernel passes through the invoke and return stages. During the invoke stage, the kernel places a message to an EPC server in one of two kernel queues. If the message requests that an extension be applied to a stream, the message is placed on a queue accessed by the daemon process. Similarly, if the message is data waiting to be transformed, it is placed on a queue for helper processes. After the kernel inserts the message into a queue, the kernel sends a signal to wake up the respective daemon or helper process. For efficiency, Mona then
suspends the process that initiated exporting code to user space. When the server completes the requested action and replies with the result, a signal wakes the suspended process. Lastly, in the *return* stage, the client wakes and receives the buffer containing the result of the transaction.

Both EPC servers pass through *get*, *service*, and *put* stages. The first type of server process is a daemon. The *get* stage involves invoking the *ioctl* system call with Mona-specific options to read a message requesting that a transformation be applied to a stream. In the second stage, *service*, the daemon spawns a new helper process to handle subsequent accesses. Lastly, the *put* stage passes the result, an integer indicating a success or an error, back to the kernel by invoking the *ioctl* system call.

The second type of EPC server is a helper process. In the *get* stage, the helper receives data to be transformed. Next, in the *service* stage, the server applies an extension to the data received in the previous stage. Finally, the *put* stage sends the transformed data back to the kernel. At the conclusion of the *put* stage, the daemon and helper servers suspend execution and wait for another request.

The implementation of the export model in Mona is illustrated in Figure 3. This figure is the user-space equivalent of the example shown in Figure 2. An application process opens two streams for reading and writing and assigns the user-space transformations $A$ and $B$ to the streams. The Mona file system inserts a kernel transformation, called the *export* transformation, into the portion of the streams residing in the kernel. In addition, Mona passes transformations $A$ and $B$ to the daemon running in user space. At the conclusion of the *open* call, the daemon spawns two helper processes to handle all subsequent accesses to the streams. When the application process reads or writes data, the *export* transformation passes streaming data from the kernel up to the corresponding helper process in user space. The helper process then applies its transformation to the data and passes the result back across the user-kernel space boundary to the *export* transformation by invoking the *ioctl* system call.

The Mona implementation moves extensions, called transformations, from the kernel to user space through the *export* transformation. The export model resembles a basic remote procedure
call in which user-space processes service requests generated by the kernel. Many details, such as handling expanding and contracting streams and advanced buffering techniques, are beyond the scope of this paper and can be found in [8, 10].

3.2 Magi Export Model

Due to decreasing processor and memory costs, it is common for manufacturers to build “intelligent” peripheral devices by integrating substantial computing capabilities. Intelligent devices have the potential to mitigate the current I/O bottleneck by preprocessing data and restructuring data during device idle cycles [9]. However, such devices are failing to reach their full potential due to a lack of support in system software [6].

The Magi device interface is an extension of the Mona file system that enables transformations to migrate out of the file system to an intelligent peripheral device. As a result, a host system can push operations onto a device capable of processing data locally. Both kernel and user-space transformations in the Mona file system always execute on the processor of the host machine, as shown in the top half of Figure 4. However, the bottom half of the figure illustrates that a transformation may migrate to a device for execution in Magi.¹

In Magi, an intelligent device cooperates with a host system to execute extensions. Each intelligent device in Magi executes a local operating system that is separate from that of its host system. In order to facilitate the migration of code from one system to the other, the Magi prototype runs Linux on both the host and the device. The model for exporting tasks to user space is implemented in the Magi runtime, which is the operating system executing on intelligent devices.

The Magi runtime is composed of three primary components, an interrupt handler, a device driver, and a user-space daemon. Figure 5 illustrates a typical transaction in the Magi runtime using these components. A request from the host system enters the Magi runtime in the interrupt

¹Migrating operations between devices involves many issues that are beyond the scope of this paper, such as handling heterogeneous devices. See [9] for more discussion.
The Magi runtime services requests from a host system.

The handler then passes the message through a queue in the device driver /dev/magi up to the daemon in user space. The daemon then services the request and sends the results back to a second queue in the driver. The driver packages the result into a message and sends it back to the host system.

The export mechanism is a vital aspect of the Magi system because it allows all users, regardless of privilege, to safely execute operations on an intelligent device. In the Magi model, a host processor generates a request to an intelligent device which then services the host’s request. The request enters the device runtime in the interrupt handler in the kernel. However, it is not safe to execute arbitrary user-defined code in the kernel. As a result, Magi passes all requests up to user space where they safely execute in a restricted process.

The Magi implementation that exports operations to user space is similar to the implementation in Mona. The kernel of the device operating system invokes operations that execute in a user-space daemon server. Unlike Mona, the Magi prototype does not handle more than one transaction at a time. Consequently, helper process servers are not necessary in the Magi system. The method of crossing the user-kernel space boundary is the most substantial difference between the Magi and Mona export models. Magi does not add or modify any system call to support its unique communication requirements. Instead, Magi uses the device driver, /dev/magi, to pass messages across the boundary. A process receives a message from the kernel by reading /dev/magi. Likewise, a process sends a message back to the kernel by writing a message to /dev/magi.

In the Magi EPC implementation, the device kernel passes through the invoke and return stages. A message from the host generates an interrupt on the intelligent device and initiates the invoke stage. The interrupt handler then examines the message and places it on a queue that is accessed by an EPC server through the device /dev/magi. Unlike Mona, there is no need to suspend a process in the device runtime because the process that generated the request is on the host system. Lastly, the return stage triggers when a server writes the result of a computation back to the file /dev/magi. In this stage, the Magi device driver reads the result and sends it from the intelligent device back to the host system.

The only server in the Magi EPC implementation is a user-space daemon. Like the Mona model, the daemon passes through the get, service, and put stages. When the kernel wakes the Magi daemon, the daemon reads the file /dev/magi looking for a request to service. In the next
stage, *service*, the daemon performs the requested task. The final stage, *put*, involves packaging the result into a message and writing it to /dev/magi.

### 3.3 Example: FTP File System

The *ftp* transformation is a Mona user-level transformation that 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.

When a user accesses a file that is guarded by the *ftp* transformation, the kernel invokes an EPC to complete the system call in user space. The additional overhead incurred using an EPC is negligible because of the latencies involved in FTP transfers. Connecting to an FTP server and accessing a remote file in user space is very straightforward. Creating an FTP file system extension in Mona is simple. However, it is difficult to put such functionality in the kernel. For example, a kernel module must explicitly schedule itself, yielding to other processes when waiting for a remote transfer. On the other hand, the operating system schedules the user-level transformation.

### 3.4 Example: Versioning File System

The *vc* transformation is also a Mona user-level transformation that automatically creates versions of a file. Obviously, the file system interface is not sufficient to put the user in touch with all the features and capabilities of a version-control system, such as CVS [3] or RCS [18]. So this transformation provides single-file, single-branch version control.

When a file being guarded by the *vc* transformation is opened, the most recent revision of the file is checked out. When a modified, checked out file is closed, the changes are written out as a new revision. Only the open and close system calls are guarded. Therefore, reading and writing proceed at native speeds.

The Mona transformation uses RCS programs to check files in and out.\(^2\) Specifically, it uses *ci* (check in) on close and *co* (check out) on open. Because it uses an existing program, the *vc* transformation is trivial to construct. It is comparable to a shell script.

To get the full power of a version control system, this transformation must be supplemented with other utilities. In some cases, it makes sense to augment *ioctl* with modes to access version control features. However, in most cases, separate programs are best. In the above transformation, ordinary, unmodified RCS programs can be used but must be applied to the raw file (bypassing the Mona transformation).

In summary, exporting kernel code to user space for execution is a valid technique for handling untrusted kernel extensions. The problem reduces to a general client-server model in which a user-space server processes handle requests generated by the kernel. The Mona file system and Magi device interface both implement working versions of the general model.

\(^2\)CVS is built on top of RCS and provides features for maintaining multiple files. The extra functionality is more than is needed or usable by the transformation.
4 Analysis

The following section analyzes the mechanisms for exporting kernel code to user space that are implemented in the Mona file system and the Magi device interface. Crossing the user-kernel boundary has the potential to introduce security violations. As a result, this section examines the implementation to verify that system security is maintained. In addition, the model for exporting computation to user space must be efficient in order to be viable. Consequently, this section also presents an analysis of the performance of Mona and Magi.

4.1 Security

The Mona file system introduces four new options to the *ioctl* system call that enable crossing the user-kernel boundary. These options enable requests for data by the daemon and helper processes and responses to those requests. Mona allocates buffers in the kernel for these transactions. If a process attempts to overflow a buffer, the transaction is truncated to the size of the buffer. Furthermore, Mona prevents a process from executing invalid *ioctl* by guarding each open stream with a key that grants access to a stream.

The technique of requiring keys to access a stream is a simple, yet effective, security measure. When a process opens a stream, Mona assigns the stream a unique 64-bit identification key. This key is not used for encryption; rather, it can be viewed as a password that grants access to data. A user-space helper process accesses its corresponding kernel stream by sending its key with every call to the *ioctl* system call. Because *ioctl* requires a valid key to access a stream, Mona prohibits a process from accessing a restricted stream. With a large enough random key, it is virtually impossible to successfully access a restricted stream by a process of guessing keys. Moreover, if a 64-bit key does not provide enough security, the key easily can be increased to 128 or 256 bits because the size of the key has little impact on system performance. Passing a 256-bit key to enable the transfer of an entire page (4 KB) of data is negligible.

The Mona file system also prohibits user-space extensions from accessing restricted resources. When the Mona daemon forks a child helper process, the system changes the helper’s user and group identification values (UID and GID) to a safe setting. This corresponds to the UID and GID of the user who instantiated the transformation (not the one that is accessing the file). As a result, a user-space extension is not allowed access to any resource that is prohibited to the user that created the file.

The Magi device interface must coordinate permissions between the host operating system and the device runtime. The current implementation assumes that both systems have identical users and permissions. When the host system generates a request to an intelligent device, the host sends the user and group identification values (UID and GID) under which the request should execute. The device runtime then passes the UID and GID to the daemon process servicing the request.

4.2 Scheduling

The use of an ordinary process to complete a system call can cause priority anomalies. The EPC server is another user process, but it may not have the same priority as the user process on who’s behalf it is executing. Priority inversion can occur if an EPC process has a higher priority than its user. Second, a user process may receive less than its fair share of the CPU when there are several peer processes concurrently executing.
Mona uses two classes of server processes, daemon and helper. The daemon servers perform
open and close operations, which occur less frequently than read and write operations. These
operations can be viewed as the completion of an open or close. The daemons servers do not
execute arbitrary user code and each operation is short. Therefore, daemon servers have a very
high priority, the same as other system daemons, such as the print spooler. On the other hand,
the helper server processes execute user-level operations. No guarantees can be made on these.
Therefore, helper processes take the priority of the invoking user process.

Technically, Mona permits priority inversion when the daemon is completing an open or close.
The effects are minor because the daemon server operations are bounded and short. Moreover, there
is little chance of priority inversion because it only occurs when a high priority process awakens
while the daemon server is executing on behalf of a low priority process. However, if one views
the daemon server execution as the completion of a system call, then there is no inversion because
system code is not pre-empted.

A more troublesome effect of using EPC is that pre-emptive scheduling can introduce delays
and cause a process to not receive its share of the CPU. Consider a system with several processes
of equal priority all ready to run. If a process invokes an EPC that is serviced by the helper server,
the “system call” is placed on the back of the queue and must wait until it gets to the front. This
is particularly costly if the EPC was called at the beginning of a time slice.

A simple solution is to set the helper server to a priority one higher than the invoker. This
ensures that the completion of the “system call” occurs before any other processes of equal priority.
However, this is not workable. First, the operation performed by a helper server can be of arbitrary
complexity and unbounded in length. This could cause even worse unfair sharing than that which it
is trying to eliminate. Second, there is no restriction on a user-level transformation calling another
user-level transformation. Thus, a user-level process could raise its level arbitrarily high by nesting
EPC calls.

We have not found the unfair sharing to be a significant problem for the application base for
which Mona is designed: That is, latency-bound applications. In such cases, the performance
bottleneck is not at the local CPU. Even though there are some additional costs due scheduling,
these cost seem to be negligible.

4.3 Performance

The exported procedure call adds usability and security to kernel extensions, but at some per-
formance penalty. This section examines the cost of EPC relative to performing the equivalent
operation in the kernel. This section only examines the cost of EPC, not the benefits because the
benefits are non easily quantified. The benefits are qualitative: ease-of-use, security, and simplicity.
For most users, the cost of kernel extension is prohibitive, whereas user-space extensions are very
common. Additionally, security is very complex for arbitrary code downloaded into the kernel,
whereas it is implicit for user-space code.

The overhead of the exported procedure call is primarily based on the number of user-kernel
boundary crossings, which is generally proportional to the amount of data being accessed. The
relative overhead of an EPC depends on the complexity of the exported operation. Complex
operations amortize the overhead better than simple operations. This section examines the overhead
for different complexities. However, complexity is only one aspect of an exported operation. One
also exports an operation because there is added functionality in user space, because there is
<table>
<thead>
<tr>
<th></th>
<th>Pentium Pro</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>kernel</strong></td>
<td>72.6</td>
<td>51.7</td>
</tr>
<tr>
<td><strong>user</strong></td>
<td>341</td>
<td>292</td>
</tr>
<tr>
<td><strong>ratio</strong></td>
<td>4.7</td>
<td>5.6</td>
</tr>
</tbody>
</table>

Table 1: Execution time (in µs) for kernel and user-level transformations on 4 KB reads.

<table>
<thead>
<tr>
<th></th>
<th>Pentium Pro</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>identity</strong></td>
<td>43.2</td>
<td>34.7</td>
</tr>
<tr>
<td><strong>xor</strong></td>
<td>898</td>
<td>428</td>
</tr>
<tr>
<td><strong>ratio</strong></td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2: Execution time (in µs) for kernel and user-level transformations on 4 KB writes.

latency that is more easily masked in user space, etc. This section does not present tests using latency-bound transformations, such as the ftp transformation, because performance is dominated by latency.

### 4.3.1 Mona Performance

The following section demonstrates that the Mona file system implements an efficient mechanism for exporting computation from the kernel to user space. The test environment is the Linux 2.2 kernel executing on 200 MHz Pentium Pro and 300 MHz PowerPC architectures. The Pentium Pro has 64 MB of EDO RAM and uses an IDE disk drive. The PowerPC has 160 MB of EDO RAM and uses a SCSI disk drive. All tests presented here average the results of 10,000 iterations on an unloaded system to amortize the cost of disk accesses.

The primary overhead is crossing the user-kernel boundary. As a result, in order to amortize costs, it is desirable to bundle as much computation as possible for a boundary crossing.

The results in Table 1 and Table 2 present the performance of kernel and user-space transformations on 4 KB (one page) reads and writes, respectively. The ratio column shows the relative speedup that would occur from moving computation in user space back into the kernel. Kernel and user-space execution times are measured on three synthetic benchmarks, varying from trivial to moderately complex: identity, xor, and xor100. The identity benchmark performs a simple read or write and does not execute any operation on transferred data. The next benchmark, xor, illustrates the cost of applying a single exclusive-or operation to each transferred byte. The final benchmark, xor100, performs 100 consecutive exclusive-or operations on each byte that is read or written.

The benchmarks isolate the costs incurred by exporting an operation out of the kernel and demonstrate that complex operations better amortize the cost of crossing the user-kernel boundary. The identity benchmark illustrates the worst-case scenario, in terms of the relative cost of EPC: exporting an operation that performs no work. In this case, it is 5–20 times more expensive to export the identity operation than to perform it in the kernel. But in absolute cost, it is between
<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>11.7</td>
<td>22.1</td>
<td>1.89</td>
<td>7.10</td>
</tr>
<tr>
<td>xor</td>
<td>15.3</td>
<td>23.2</td>
<td>1.50</td>
<td>11.6</td>
</tr>
<tr>
<td>xor100</td>
<td>1250</td>
<td>1310</td>
<td>1.05</td>
<td>1240</td>
</tr>
</tbody>
</table>

Table 3: Execution time (in ms) on 4 KB reads and writes in Magi.

240 and 850 $\mu$s. A slight increase in the complexity of a transformation greatly reduces the relative overhead. The xor transformation, which is still extremely simple, has much less relative overhead than identity.

As the ratio column indicates, the relative cost of exporting computation to user space decreases substantially as the complexity of the exported computation increases. For xor100, a moderately complex operation, the export mechanism increases execution time by only a factor of 2 on the Pentium Pro architecture and a factor of 4 on the PowerPC architecture. For many applications, the overhead is an acceptable trade for the increased flexibility and security provided by exporting tasks out of the kernel.

The xor100 transformation computes longer than the Linux time-slice. Therefore, some of the cost of executing xor100 in user space is due to scheduling. However, executing the xor100 operation for 10 ms in the non-preemptive Linux kernel violates the Linux time quantum of 9 ms. As a result, the kernel version of xor100 must explicitly schedule other processes or violate the scheduling policy. On the other hand, the system automatically schedules the user-space xor100 benchmark which takes multiple time quanta to complete. As a result the absolute overhead of the complex operation is much larger than the trivial operation, even though there are an equal number of user-kernel boundary crossings. This illustrates the other significant overhead: pre-emption. However, this is not exactly overhead. Pre-emption is a policy decision. The xor100 kernel transformation violates this policy. If it obeyed the policy, some overhead due to scheduling would occur.

4.3.2 Magi Performance

The Magi system requires attaching an intelligent device to a host system. In our prototype, the intelligent device is a co-processor card designed for a standard ISA bus. The card integrates a 33MHz 486 processor, 16MB of RAM, and an IDE disk controller. The host system and the device both execute the Linux 2.2 kernel.

The tests described in this section measure the overhead of exporting operations from the kernel to user space on the intelligent device. The device is configured to act as an intelligent disk drive for the host machine. The device implements two ramdisks to service the host. The first ramdisk is in the kernel and immediately responds to requests from the host. The second ramdisk is implemented in a daemon process and requires the kernel to export an operation to user space. The execution times for each type of read or write request are averaged over 1000 iterations. The host sends messages to the device requesting that the device read and write data. In addition, the host requests that the device perform a specified operation on the transferred data. The following benchmarks perform the identity, xor, and xor100 operations on data, as described in Section 4.3.1.
Magi demonstrates that executing kernel code in user space achieves reasonable performance when the complexity of an operation justifies exporting it out of the kernel. As expected, Table 3 shows that exporting code from the kernel to user space imposes overhead. The kernel and user columns represent the execution times for applying an operation to data in the kernel and user space, respectively. The ratio column shows the relative slowdown from pushing a kernel operation into user space.

Once again, the identity benchmark is the worst-case scenario. When Magi exports an operation that performs no work, servicing reads and writes takes roughly twice as long to complete in user space than in the kernel. However, as the complexity of an operation increases, the relative overhead decreases. For the single exclusive-or operation, the table shows that overhead decreases from 100% down to 50%. Similarly, for the more complex xor100 benchmark, the overhead drops down to only 5% of the total execution time.

In summary, implementations of the Mona file system and Magi device interface export operations from the kernel to user space. This section demonstrates that both systems maintain security. In addition, Mona and Magi achieve reasonable performance when exporting complex operations to user space.

5 Conclusion
The standard system call interface executes operations in an operating system kernel on behalf of an application in user space. However, systems do not implement the corresponding inverse operation, executing code in user space on behalf of the kernel. Although pushing tasks into user space incurs overhead, the cost is offset by benefits, such as increased ease of use, greater security, and simple extension.

This paper presented the design and implementation of the exported procedure call, an interface for pushing operations from the kernel to user space. It described a general export model that outlines the requirements for pushing operations out of an operating system kernel into user-space processes. The exported procedure call model resembles a remote procedure call in which the kernel invokes an operation in a user-space server.

In addition, this paper examined two systems, the Mona file system and the Magi device interface, which implement versions of the exported procedure call. Mona utilizes the export mechanism to push stream-oriented kernel extensions into user space. As a result, a kernel extension can be written by an unprivileged user yet execute safely in a restricted process. Furthermore, a kernel developer may debug an extension in user space before implementing it in the kernel. Likewise, the Magi device interface pushes requests from a host system out of the device operating system kernel into a restricted process. Consequently, Magi enables a host machine to safely migrate application-specific operations onto an intelligent device.

An analysis of the Mona and Magi implementations demonstrates that overhead cost are contained and acceptable for many operations. It is shown in Section 4.3 that for both systems the relative cost of exporting an operation from the kernel to user space decreases as the complexity of the exported operation increases. For a reasonably complex operation, the Mona file system executes 2-4 times faster in the kernel than in user space. Similarly, the Magi device interface incurs an overhead of roughly 5% to push a complex operation out of the kernel. Neither system sacrifices security to maintain this level of performance.
References


