A Comparison of OS Extension Technologies

Christopher Small and Margo Seltzer
Harvard University

Abstract

The current trend in operating systems research is to allow applications to dynamically extend the kernel to improve application performance or extend functionality, but the most effective approach to extensibility remains unclear. Some systems use safe languages to permit code to be downloaded directly into the kernel; other systems provide in-kernel interpreters to execute extension code; still others use software techniques to ensure the safety of kernel extensions. The key characteristics that distinguish these systems are the philosophy behind extensibility and the technology used to implement extensibility. This paper presents a taxonomy of the types of extensions that might be desirable in an extensible operating system, evaluates the performance cost of various extension technologies currently being employed, and compares the cost of adding a kernel extension to the benefit of having the extension in the kernel. Our results show that compiled technologies (e.g. Modula-3 and software fault isolation) are good candidates for implementing general-purpose kernel extensions, but that the overhead of interpreted languages is sufficiently high that they are inappropriate for this use.

1 Introduction

The motivation that led to the emergence of microkernels in the early 80's leads now to the emergence of extensible operating systems. It is unreasonable to expect any one system to possess all of the functions needed by all applications. Rather, vendors of sophisticated applications, which require application-specific operating system customization, sell these extensions as well. These kernel extensions enable new functionality, but they also introduce the possibility of compromising system reliability. If an application consistently brings a system down, its additional functionality is hardly worthwhile. Therefore, one of the major challenges that arises in supporting easily extensible operating systems is being able to do so in a manner that does not compromise the reliability of the underlying system.

Commercial systems have traditionally taken the route of supporting dynamically loadable extensions (e.g. device drivers, new subsystems, and new file systems). This is a practical approach, but does not address the reliability issue; it is assumed that it is acceptable for the kernel to crash if the module has bugs in it and if the module does not contain security violations. This scenario makes sense when a small group of trusted users are the only ones who can load such extensions, but a general purpose facility must be more robust. Several current research projects are developing more general-purpose extensible systems that provide reliability and security. The goal of this paper is to describe the assortment of technologies available and the types of extensions one might envision, and then to assess the performance tradeoffs of the various choices.

In Section 2 we discuss several of the extensible systems currently under development. Section 3 describes the different types of kernel extensions that might be useful. Section 4 discusses the various extension technologies, and section 5 introduces our test cases and presents our performance results and analysis.

2 Related Work

If one chooses to ignore the potential reliability problems introduced by adding code to the kernel, the no-overhead solution is to allow unprotected patching. Microcomputer operating systems (e.g. MS-DOS and the Macintosh OS) allow arbitrary code to be patched into the kernel, providing no protection from misbehaved applications. On the other hand, because applications have complete control, it is much easier to manipulate and extend the kernel to provide for the needs of applications. It may be easier to write a real-time system on a microcomputer operating system than on a workstation operating system *because* the kernel lacks multitasking and protection.

The move towards safe extensible operating systems is an evolutionary step on the path from

This research was supported by Sun Microsystems Laboratories.

conventional monolithic kernels to microkernel architectures. The Mach microkernel [ACCE86] was designed to allow kernel functionality to be moved out of the kernel address space into user-level external servers in order to increase safety, robustness, and flexibility. However, the expense of frequent upcalls to user-level code motivated the current generation of microkernel-based systems, which link server code directly into the kernel address space [GUIL91]. The CMU Bridge project follows this model, placing servers in the kernel, but protects the kernel using software fault isolation techniques such as *sandboxing* [WAHBE93]. Measurements of sandboxing have shown it to have a much lower overhead than hardware protection mechanisms (on the order of a few percent).

The Exokernel project [ENGL95] also strives to reduce the number of user-kernel protection boundary crossings, but it takes a different approach. Rather than support safe downloading of code into the kernel, it moves as much functionality as possible from kernel to user-level. In this reductionist approach to kernel design, kernel abstractions are thought to be the reason that systems have poor performance, hence they are removed from the kernel and placed into the application. System abstractions are available from user-level libraries, but applications control which abstractions they use. The kernel is left only with responsibility for controlling access to physical devices. There is little or no reason to extend the kernel; nearly all functionality is under the control of the application.

Along with adding functionality, extensible systems can provide applications with the ability to override policy decisions. Cao et al. [CAO94] motivate this sort of extension by examining the performance improvement achieved by allowing an application to control the buffer cache eviction policy. Their system did not allow applications to add new policy code to the kernel; rather, multiple policies were compiled into the kernel and an application chose among them. This work showed the benefit of allowing applications to control policy; however, we believe that it is not possible to determine (and implement) all policies *a priori*; a more general extensibility mechanism is required.

The HiPEC system [LEE94] is similar to, but more flexible than, Cao's system. HiPEC allows applications to control VM caching policy using programs written in a simple, assembler-like, interpreted language designed specifically for the task of managing a queue of VM pages. The performance impact of executing a program in this language is low, but the expressiveness of the HiPEC language is limited (it has only 20 basic instructions). The language would have to be augmented if it were to be used for other applications.

Packet filters are used to demultiplex a stream of network packets by examining the contents of each packet header. Often, packet filters are implemented in a simple interpreted language (e.g. [MOGUL87, MCCAN93, YUHARA94]). A special language, designed to efficiently describe packet headers, is used to write packet filters. The performance of interpreted packet filters is close to that of compiled code, but, like HiPEC, the expressiveness is limited to the specific domain.

Instead of starting with a minimal language and extending it, the SPIN system [BERS95] is written in, and uses as its extension language, Modula-3 [NELS91]. Modula-3 is a strongly-typed, garbage-collected language, designed so that it is impossible for a program to have a "dangling pointer" to deleted data or to construct a pointer to an arbitrary memory location. Because of the design of the language, code in "safe" modules is not able to reach outside its bounds and violate the integrity of the program in which it is running.

The μChoices operating system [CAMP95] proposes using "a simple flexible scripting language similar to Tcl" to aggregate multiple kernel calls or remove control traffic between user-level and kernel-level. Although Tcl was not designed with this application in mind, if extensions are relatively small, the raw performance of the extension technology will be of little or no consequence.

Extension technology is also useful outside the kernel. Some database servers allow clients to load query- or datatype-specific code into the server to improve performance. The Thor database server uses a typesafe language designed for writing extensions [LISK95]; the Illustra database server is extended by writing DataBlades, which add support for new data types to the server [ILLU94]. The Illustra server does not currently protect itself from misbehaved DataBlade code, although Illustra is evaluating software fault isolation techniques.

The HotJava Web browser from Sun Microsystems can be extended with "applets" written in Java, a Modula-3 like language with a C++ syntax [GOSL95]. Java code is compiled to a machine-independent byte-code that is downloaded by the HotJava browser and interpreted or compiled into native code on-the-fly. Like Modula-3, Java is designed to reduce or eliminate dangling or stray pointers and safety problems.

In this work, we examine the extension technologies being proposed by these systems: unsafe

C (the DOS approach), C in user-level servers (the microkernel approach), software fault isolation, Modula-3, Java, and Tcl. Our goal is to understand the performance implications of the extension technology choice.

3 Graft Taxonomy

We refer to a kernel extension as a *graft* and the process of adding a graft to the running kernel *grafting*[1]. We have found three motivations for grafting code into the operating system kernel:

- *Policy:* the application wants to control kernel policy, e.g. how it manages the buffer cache, VM cache, or process scheduling.
- *Performance:* the application wants to migrate portions of itself into the kernel in order to improve overall performance. This technique can save data copies between kernel and user space (e.g. when copying data from the disk to the network), and upcalls into user space (e.g. to handle a mouse event).
- *Functionality:* the application wants to add general functionality to the kernel, e.g. support for access control lists, automatically compressed files, or new communication models.

Although there are an unlimited number of ways in which a graft can be structured, we have identified three basic structures into which the implementation for most grafts will fall: *Prioritization* grafts, *Stream* grafts, and *Black Box* grafts.

3.1 Prioritization

There are numerous places in a kernel where one of a set of entities is selected. Choosing a victim is the central policy decision made by the virtual memory system (which page to evict), the buffer cache manager (which buffer to evict), and the process scheduler (which process to schedule next). We call this a *Prioritization* policy decision, and a graft that replaces prioritization policy code a *Prioritization graft*.

Normally, the VM system and the buffer cache use a least-recently-used (LRU) policy (or a variant thereof), although in some cases a different policy works better. An application might know that each block of a file will be read once, in order, and not read again, in which case it makes sense to use a most-recently-used (MRU) strategy for that file. As another example, while processing a query, a database system knows which blocks of the database will be needed soon and which blocks are no longer needed.

The access pattern of a set of pages or file blocks is often a good heuristic for deciding which pages or blocks will be used in the near future, but it is not infallible. For example, a garbage collector copies live objects from partially filled pages with the goal of decreasing fragmentation. After a collection, a page from which live objects have been taken has been accessed recently, but contains no useful data; on the other hand, pages that are full of live data have not been accessed recently. Using an LRU strategy, the former will be retained in memory and the latter will be evicted, which is the opposite of what is desired.

Process scheduling is another example of a prioritization policy. At each scheduling point the kernel has a list of candidates, and chooses one to run. No scheduling algorithm is appropriate for all application mixes; the demands of interactive applications differ from those of applications with real-time deadlines, and a scheduler optimized for one class will not satisfy the other. Processes may wish to be scheduled as a group; a client-server application may not want the server to be scheduled unless there is an outstanding client request, in which case it should be scheduled ahead of any client.

In general, a Prioritization graft is one that is presented with a list of options and must select the item of highest priority. It uses some internally defined weighting function to choose a candidate. The weighting function may use just the information in the list, or it may have access to some data from the application (e.g. which blocks or pages will not be needed again).

Our Prioritization graft benchmark models a VM page eviction policy. We assume that the kernel maintains a list of pages in LRU order; when it comes time to evict a page, the kernel normally chooses the page at the head of the LRU queue as its candidate. In our model, instead of immediately evicting the candidate, the kernel determines which process owns the candidate page and allows the process to offer one of its other resident pages for eviction. Our model application keeps a “hot list” of pages that will be needed in the near future. The goal of the page eviction graft is to ensure that none of these pages are evicted.

The page eviction graft receives a pointer to the head of the LRU queue. The graft checks to see if the candidate (the head of the queue) is on the application’s hot list; if it is not, it accepts the kernel’s candidate. If the candidate is on the hot list, the graft searches through the queue for an acceptable page

1. This is a (weak) pun on the name of our project, the VINO kernel.

that is not on the application's hot list. This page is returned to the kernel.

For this benchmark we model a TPC-B transaction processing benchmark database [TPCB90]. The database holds 1,000,000 records in a four-level b-tree; the b-tree has approximately 400 internal pages (16MB) and 50,000 data pages (200 MB). The b-tree is 50% full, and has one root page, four pages at the second level, 391 pages at the third level, and approximately 50,000 pages at the fourth level; each third-level page in the b-tree points to up to 128 fourth level pages.

The server accesses the database by mapping it into its virtual address space. When the server does a non-keyed lookup, it traverses the b-tree in depth-first order, starting from the root. When it reaches a third-level page it knows which 128 fourth-level pages it will access next, hence which of the memory mapped pages of the database should not be paged out. During such a search, it constructs a hot list of these 128 pages. If a page fault occurs during a search, our graft is called with the LRU chain head, and it uses the hot list to find an eviction candidate that is acceptable to the application. As each page is processed, its entry is removed from the hot list, so as the simulation runs, the queue grows shorter. We presume that the kernel keeps track of candidate pages and graft-proposed alternates, as in Cao's system [CAO94], to ensure that an application does not manipulate the VM system to gain more physical memory than it would receive under the default strategy.

This test is not particularly compute-intensive. Instead, it is sensitive to the overhead associated with traversing a list of items. If the extension technology requires extensive pointer checking, or does not support traversal of linked lists, this test will highlight it.

Because there are 50,000 data pages in the database, there is a low probability that a page that the application will need is already in the cache (roughly 64/50,000, or once every 781 times). However, if one of the pages is in memory, we want to ensure that it is not evicted. For the graft to be successful, the overhead of checking on each eviction should be low relative to the cost of evicting and then re-faulting a page. Our test computes the break-even point, showing how often the graft will need to save an eviction in order to pay for its per-eviction cost.

3.2 Stream

Our second graft model is based on the idea of Unix filters and pipes. Frequently, it is useful to apply a set of filters to a data stream. For example, we might want the kernel to transparently compress a file when it is written and decompress it when it is read, or automatically encrypt a file when written and decrypt it when read by the appropriate user. For security reasons we might want to compute a secure checksum, or *fingerprint*, of an executable when it is loaded, to verify that it has not been compromised by a virus. A *stream* graft is such a filter. It consists of filtering code that is inserted into a data stream, normally between the storage system and application level.

A journaling file system is one that accepts a stream of I/O requests, saves a journal of the metadata changes, and passes along the original requests. A standard filesystem could be transformed into a journaling filesystem by inserting into the request stream a graft that journals the changes made to the metadata.

The Stream Input-Output System of UNIX [RITCH84] decomposed the character I/O system of UNIX into a set of filters. Network and terminal protocols were built up by linking filters into chains. Characters read from (or sent to) a device were passed to the first filter in the chain; each filter processed the characters in its input queue and moved them on to the next filter in the chain. This mechanism was used not only to handle erase and kill processing, but also to create pseudo-devices, such as virtual displays and keyboards, to construct multiple virtual terminals from a single terminal.

Another example of a stream graft is one that takes a data source (e.g. the disk) and, rather than modifying it on the way to application level, writes it elsewhere (e.g. the network). Recent research into fast path connections, such as the *x*-kernel work at Arizona [DRUS93], the video server benchmark of the SPIN operating system [BERS95], and Fall's work in decreasing I/O time through use of in-kernel copying [FALL93] show that there is a substantial performance gain from saving copies to and from user-level. A stream graft that takes its input and directs it to an output connection, perhaps after transforming the data, could be used to build this type of fast path connection.

Our representative stream graft is an implementation of the MD5 Message-Digest Algorithm [RFC1321], which produces a 128-bit fingerprint of a file. The MD5 fingerprint is both expensive to compute and computationally infeasible to forge. MD5 is useful for ensuring that a file has not been tampered with; a change to the contents of the file will result in a change to the fingerprint. If the fingerprint is kept separate from the file (say, on a small amount of safe media, such as a read-only floppy disk), a change to the file can be detected by computing its MD5 fingerprint and comparing it to the saved fingerprint.

Like many compression and encryption algorithms, MD5 is stream-based. The algorithm maintains a small amount of state as it processes the data; unlike compression and encryption algorithms, the data output is the same as the input; when the algorithm completes, the graft can be queried for the fingerprint, and the computed fingerprint can then be compared to the saved fingerprint.

If the MD5 fingerprint can be computed as quickly as data can be read from the disk, the time spent in the MD5 code can be overlapped with I/O activity. However, if it takes longer to compute a fingerprint than it takes to read the data from the disk, the overall processing time will increase. Our test measures whether a graft written in a given technology can keep up with the disk.

3.3 Black Box

The Black Box graft structure is more general than that of a Prioritization or Stream graft. A Black Box graft has some number of inputs, some state, and a single output. We see it operating as a “black box” function, normally producing a single output value. For example, at the center of the code that implements Access Control Lists is a small database that (at an abstract level) accepts a triple containing a file access request, a user ID, and a file ID, and responds “yes” or “no.”

File system read-ahead code, which determines how many (and which) blocks of a file to prefetch, is another example of a “black box” function. If the application knows ahead of time the order in which blocks of a file will be read, the kernel can use this information to make read-ahead decisions. In some cases, an application will read a subset of the blocks of a file in order, and then skip to another region of the file. If the kernel uses heuristics (rather than application knowledge) to choose a read-ahead policy, it can not cope with arbitrary application behavior. With the cooperation of the application, it can make more appropriate read-ahead decisions.

A Logical Disk facility (LD) [DEJON93] sits between the filesystem and the physical disk. The filesystem reads and writes logical blocks, and the LD maps the logical requests to locations on the physical disk. The LD can be used to transparently replicate data, by writing it in multiple places on the same disk or multiple disks, and speed write performance, by writing logically discontiguous blocks on a physically contiguous region. A log-structured file system [ROSE91] can be implemented using a logical disk facility; the filesystem lays out blocks as its sees fit, and the Logical Disk reorders and buffers writes to improve write performance.

Our black box test application is a simple logical disk facility that converts random writes to sequential writes. It accepts block write requests, batches them into physical segments, and maintains a mapping from logical block numbers to physical block numbers. If the savings in I/O time due to batching is greater than the cost of translating logical block numbers on each read/write, the graft is effective.

4 Extension Technologies

Each extension technology offers a different level of safety and imposes a different level of trust. An extension written in an unsafe language (e.g. C) can read, write, or jump to any location in the address space (using pointer arithmetic); one written in a safe language (e.g. Modula-3 or Java) is restricted to code addresses exported to the extension. In either case, we need a mechanism to ensure that extension code not monopolize the CPU; we must be able to preempt an extension that runs too long. This means that extension code should not be able to disable interrupts.

For an operating system extension, read protection is important, even if it is not required for reasons of data privacy. Because an extension running in the kernel has access to memory mapped devices, an extension can destructively *read* a device register.

Once safety is ensured, the performance of a technology determines its suitability for different applications. The overhead of a graft intended to increase performance should not cause performance to degrade.

We also need to make sure that an extension technology is sufficiently expressive, so that it is possible to implement the grafts we want to write. A small, specialized language designed around a single problem domain may perform better than a general-purpose language, but will be difficult to reuse in other domains.

The size of the runtime environment can have an impact on overall performance. The runtime code size of support environments ranges from tens of kilobytes to several megabytes. If we require that the support environment run in the kernel and remain resident, any memory used by the environment is memory unavailable for other uses.

The extension technologies we examine fall into three basic trust models: *hardware protection*, *software protection*, and *interpretation*.

4.1 Hardware Protection

The simplest, and perhaps most dependable, method for ensuring the safety of the kernel is to place extensions outside the kernel's address space in order to take advantage of the protection offered by the hardware. When the kernel wants to run an extension, it upcalls into user-level code; when the code returns, the kernel continues. Along with memory protection, the kernel can time-slice the extension to ensure that it does not monopolize the CPU. If the extension runs too long, the kernel can abort it and carry on without it.

The primary disadvantage of this model is that there is a cost associated with an upcall; for small extensions, this per-invocation overhead can be much larger than the cost of running the extension.

4.2 Software Protection

Using software protection, we place extension code in the same address space as the kernel, but restrict the instructions that are evaluated by the extension. By controlling the language (e.g. Modula-3), or the compiler, or by patching the binary code [WAHBE93], we can control the instructions evaluated by the extension. This allows us to ensure that the extension does not read or write outside its bounds, or jump to arbitrary kernel code. In addition, we can ensure that it does not issue instructions that disable interrupts or that initiate I/O operations.

Software protection can offer the highest performance of the three options, but requires control of, and trust in, the language translation tools used to process the extension source. The security of the kernel is a function of the correctness of the generated code; if the translation tool can be coaxed into generating unsafe code, the security of the kernel can not be guaranteed. If, for example, under certain circumstances the array bounds checking of a compiler can be subverted, it may be possible to write code that overwrites its stack and circumvents the kernel's security mechanisms.

Even assuming that the translation tools work correctly, the kernel still needs to verify that code loaded into the kernel was in fact processed by a trusted language tool. This can be accomplished by performing the translation at load time, or by marking the code (say, with a cryptographic checksum) at the time it is generated. The former technique requires a high cost at extension load time; the latter implies that we are able to embed a secret key of some sort in the language tool and mark the generated code with this key. Methods for generating cryptographic checksums are well known; the social issues of key distribution and management are outside the scope of this paper.

The goal of *software fault isolation* (SFI) is to make it more efficient to ensure the validity of memory references using software than by using hardware. One type of SFI, *sandboxing*, ensures that the high bits of a memory address match those of the *sandbox* region assigned to the function or module. In this way, a module can, at worst, overwrite its own data with a stray pointer or jump to locations in its own code. Sandboxing can be done at compile time or performed as a post-processing phase on object files, allowing separation of the sandboxing tools from the compiler. At load time, a linear-time algorithm can be used to guarantee that all memory references in a piece of object code have been correctly sandboxed.

Omniware C++ [COLU95] is a commercial system that includes a compiler that generates machine independent code. A run-time system translates the machine independent code into native code with software fault isolation instructions and links the code into the executable.

Many modern languages, such as Modula-3 and ML, do not suffer the safety problems of C. In a *typesafe* and *pointer-safe* language it is impossible to construct a pointer to an arbitrary memory location, or be left with a "dangling" pointer to deallocated memory. If type casting is available, it is combined with compile-time analysis or a runtime type check to ensure that the cast is valid. Similarly, array bounds are checked on access to ensure that a program does not access memory outside the array.

Although the definition of a language may ensure type and pointer safety, the language tools written to compile and run the language may not correctly implement the definition. It is infeasible (using present technology) to verify the correctness of a Modula-3 compiler; because of this, we can not be sure that a compiler will not generate incorrect code. (In fact, while running our VM Page Eviction benchmark, we found a bug of this type in the Modula-3 compiler.)

4.3 Interpretation

Instead of depending on the safety and security of a compiler, we can create a virtual machine, an interpreter, to run grafts. The source language (which could be C, Modula-3, or anything else) would be compiled to intermediate or machine code for a real or virtual machine. The kernel would include an interpreter to run the code, ensuring its safety. This model allows complete control over the behavior of the extension by implementing only safe operations in the interpreter.

The intermediate code could also be used as the input to a runtime code generator. A reasonably fast interpreter runs 10 to 100 times more slowly than compiled code [MAY87]; however, using incremental code generation techniques, performance can approach that of compiled code [HOLZE94]. (Note that there is a flexible line between generating native code at load time – as above – and dynamically generating native code from interpreted code.)

Java is compiled to a compact byte code for the Java Virtual Machine. As interpreters go, the Java interpreter is fairly fast. In addition, Sun plans to release a runtime code generator for Java in the near future. The authors of Java expect that compiled Java will run at about the same speed as compiled C or C++ ([GOSL95], p. 48), which, based on current technology, is believable.

The currently available Java system is Alpha-release software. Versions for Sparc/Solaris and Windows are available from Sun (on java.sun.com); Java has been ported to HP-UX and UnixWare (by OSF) and Linux (found on java.blackdown.org).

Another technique for building an interpreted language is not to transform the source to an intermediate format, but rather to interpret it directly. This technique, used in `awk`, `sh`, and `Tcl`, leads to a smaller start-up time, with a higher overhead per statement. Because source-interpreted scripting languages are immensely popular, and have been proposed as a vehicle for writing grafts [CAMP95], we include Tcl as one of our tested technologies.

5 Performance Analysis

Given the wide range of extension technologies available, it is not obvious which is "best" in any dimension. In fact, extensible systems are being built that employ nearly every technology described. In this section, we measure and analyze the performance of our sample extensions.

5.1 Hardware

We ran tests on four hardware platforms: three commercial workstations and one Intel x86 "PC".

- *Alpha:* DEC AlphaStation 400 4/233 (233MHz), running DEC OSF/1 v3.2A, 64MB of memory.
- *HP-UX*: HP PA-RISC 9000/735 (99MHz), running HP-UX A.09.03, 80MB of memory.
- *Linux:* "PC"-class Pentium (90MHz), running Linux 1.1.95, 16MB of memory.
- *Solaris:* Sun SPARCStation 20 (75MHz), running SunOS 5.4, 128MB of memory.

5.2 Extension Technologies

We implemented our grafts on five different extension technologies. Not all tests were run on all platforms; for example, the current release of the Omniware compiler runs only on Solaris.

- *C:* compiled with `gcc -O` (version 2.6.3 on HP-UX, 2.7.0 on all other platforms).
- *Java:* release Alpha 3.
- *Modula-3:* version 3.5.3 of DEC SRC Modula-3.
- *Omniware:* compiled with Colusa's omniC++ compiler, version 1.0 beta, release 1.5. (This commercially available compiler generates machine-independent code that is translated to native, fault-isolated code at runtime. This pre-release version of the compiler supports write and jump protection, but no read protection, and does not include an optimizer for the SFI instructions.)
- *Tcl:* Tcl version 3.7.

5.3 Upcall Overhead

As a baseline for comparison, we implemented each benchmark in C. Since C is not a safe language, we estimated the cost of implementing the benchmark as a user-level server by determining the break-even point as a function of the time required to upcall to such a server; if an upcall is free (takes no time), the performance of a system written using user-level servers is equal to that of one with unsafe C linked into the kernel. As the cost of an upcall increases, the performance of a system written using user-level servers will decrease.

When the kernel makes an upcall to a user-level server, it pushes a new call frame on the server's stack and switches to the server. When the server is done handling the upcall, it returns to the kernel. This process is similar to (but simpler than) the one followed by the kernel when it posts a signal to a process. First, the kernel pushes a call frame for the signal handler on the application's stack, then schedules the application process. When the process returns from the signal handler, it re-enters the kernel. The kernel cleans up the state of the process and returns it to the point where the signal was handled.

To give a feeling for the cost of an upcall on each platform, we measured the time required to handle a signal sent to a process. The test program forks a child process, which registers handlers for a group of twenty signals and then suspends itself (by sending itself `SIGTSTP`). When the parent is notified that the child is suspended, it posts the handled signals to the child, then wakes it (by sending it `SIGCONT`). The child awakes, handles the signals, and suspends itself again. When the parent is notified that the child has once again been suspended, it knows that all signals have been handled.

We then measured the time to post the signals to the child when the child ignores (rather than handles) the group of signals. The latter time is subtracted from the former; the result is divided by the number of signals handled, which gives an estimate of the time required to handle a single signal. The results of this test are shown in Table 1.

Platform	**Signal Handling Time**
Alpha	19.5µs(7.5%)
HP-UX	25.8µs(1.4%)
Linux	55.9µs(0.1%)
Solaris	40.3µs(3.8%)

Table 1. Signal Handling Time We measure the time required to send twenty signals to a child process that handled the signals, then subtract the time required to send twenty signals to a child process that ignores the signals. The difference is divided by the number of signals to give a per-signal handling time. Each time is the mean of thirty runs of 1000 iterations each (standard deviations in parenthesis).

We implemented and measured the performance of a simple upcall mechanism on BSD/OS 2.0. On a 486-DX266 we measured a signal handling time of 63.1µ, and an upcall time of 37.2µs (about 40% quicker).

These upcall estimates are fairly conservative. Work done to improve exception handling times ([THEK94]) and cross-domain procedure calls ([Ford94]) lead us to believe that it is feasible to implement an upcall mechanism that takes less than one fourth the time we measured for signal delivery.

5.4 VM Page Eviction

As described in Section 3.1, our sample Prioritization graft takes the LRU-ordered list of page eviction candidates and returns a candidate not on its hot list.

We assume that the application keeps the hot list (of active pages) in its memory at a known location, so that it is accessible to the graft. In addition, we assume that the application keeps the hot list in a form that can be easily traversed by the graft; for example, the C graft searches a linked list of structs, where the Modula-3 graft searches a linked list of Modula-3 RECORDs.

In our model application, the hot list starts out with 128 entries (the number of data pages referenced by a level-three internal page), and as each page is faulted, it removes that page from the hotlist. On average, the hotlist contains 64 pages, which is the number we simulate.

We determined the time to check the 64 element hotlist in each of the supported technologies, and present that time in Table 2. To give an intuitive feeling for the performance of each technology, we also normalized the time relative to unsafe C code.

To determine the break-even point for this graft, we measured the page fault time on each of our test platforms (Table 3). The times listed indicate how long it takes to handle a page fault event, not how long it takes to bring in a single faulted page; Alpha and HP-UX bring in multiple disk pages on each fault.[2] We assume that the systems are performing read-ahead in order to take advantage of (expected) locality of reference. However, our model database server would not be able to take advantage of this behavior, as the faulted data pages are scattered throughout the database.

(The page fault read-ahead policy exhibited here is an obvious candidate for grafting; if we are able to control how many pages the system brought in on a fault, we can reduce the per-fault time.)

Once we have computed the page fault time, we can determine the break-even point for this graft. We divide the page fault time by the time required to run the graft; the result is the number of times we can run the graft for each page eviction saved and still be ahead of the game.

Remember that our model application would find a page to save, on average, once ever 781 invocations. If the break-even point is less than this, our model application would not benefit from this graft. Worse yet, if the number is less than one, the amount of time to run the graft is greater than the page fault time, hence under no circumstances would the graft be beneficial.

In Table 2 we see that on Solaris the relative times of Omniware and Modula-3 are quite close, each running about 40% slower than unprotected C.

2. The number of pages faulted was determined by running the page fault test on an otherwise unloaded system while watching the output of iostat or vmstat.

Platform		**C**	**Java**	**Modula-3**	**Omniware**
Alpha	raw	2.9µs(0.2%)		3.2µs(1.5%)	
	normalized	1.0	N.A.	1.1	N.A.
	break-even	8655		7843	
HP-UX	raw	6.0µs(0.4%)	159µs(0.8%)	6.8µs(1.7%)	
	normalized	1.0	26.5	1.1	N.A.
	break-even	2983	113	2632	
Linux	raw	3.7µs(0.1%)	237µs(0.1%)	9.1µs(0.1%)	
	normalized	1.0	64	2.5	N.A.
	break-even	1270	20	516	
Solaris	raw	4.5µs(0.1%)	141µs(0.3%)	6.3µs(2.8%)	6.3µs(0.2%)
	normalized	1.0	31.3	1.4	1.4
	break-even	1533	49	1095	1095

Table 2. **VM Page Eviction Test** We measure the mean time required to search a 64 element “hot list” of page numbers. Raw times and time normalized to unprotected C code (on the same platform) are given. The break-even point is the number of times the graft can run in the time it takes handle a page fault. Each time is the mean of 30 runs of 100,000 searches each (standard deviations in parenthesis).

(Remember, however, that this version of Omniware does not include read protection, which gives it a performance advantage over Modula-3.) On Alpha and HP-UX the Modula-3 code runs 10% slower than the C code, which is not a significant difference for this test.

Platform	**Fault Time**	**Num Pages**
Alpha	25.1ms(5.0%)	16
HP-UX	17.9ms(0.8%)	4
Linux	4.7ms(0.5%)	1
Solaris	6.9ms(3.2%)	1

Table 3. Page Fault Time Measured using *lmbench* (standard deviations in parenthesis). Alpha and HP-UX bring in more than one disk page on a fault, performing read-ahead, even though the test performs random accesses to memory.

On Linux, we see a 150% slowdown for Modula-3, a greater difference than we see on other platforms and with other tests. Examining the code generated by the Modula-3 compiler, we found that it includes a runtime check against NIL (location zero) on each pointer access. The code generated on the other platforms (Solaris, Alpha, and HP-UX) does not include explicit NIL checks. The Modula-3 language specification [NELS91, p. 50] states that dereferencing NIL should cause a runtime error; on Solaris and Alpha, dereferencing location zero causes a segmentation violation, which is trapped by the Modula-3 runtime system. This is not the case on Linux, so the runtime check is needed. (We found that although no runtime checks are generated by the HP-UX version of the Modula-3 compiler, dereferencing location zero does not cause a segmentation violation. This appears to be a bug in the Modula-3 compiler.)

For our purposes (i.e., operating system extensions), the Alpha and Solaris slowdowns are the more appropriate comparison – because we can ensure that dereferencing location zero causes a fault, we would not need runtime NIL checks. (This fault would not be handled by a signal mechanism in the kernel, but would instead require some support by the normal kernel fault logic.)

On Solaris, the Java code runs at about 1/30th the speed of compiled C code, and about 1/20 of Modula-3. On this platform we see that the break-even point for Modula-3 is about 1100 pages; for Java, the break-even point is about 50 pages, too low to benefit our model application.

To compare these results to the overhead of performing an upcall on each eviction, we computed the break-even point as a function of upcall time, allowing the upcall time to range from 0 to 50µs (see Figure 1). When compared with the break-even points for Modula-3 and Omniware, we find that we would need an upcall time on the order of 5µs for upcalls to compete with the compiled extension technologies, which would be difficult to achieve.

Unsurprisingly, we found that Tcl is not the appropriate tool for this job. Measurements of Tcl showed that it was four orders of magnitude slower than the compiled languages (C and Modula-3), taking 40ms on Solaris, as compared with the C version, which took 4.5µs. With a break-even point

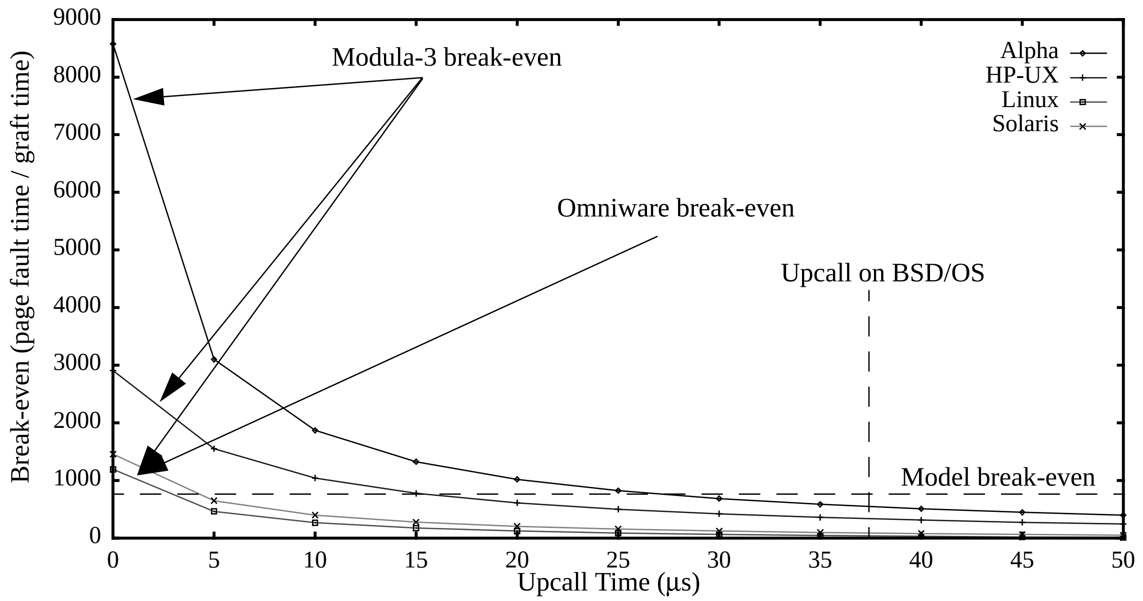

Figure 1. Break-Even vs. Upcall Time The break-even point for the VM Page Eviction test. Break-even is inversely proportional to the upcall time. The break-even points for Modula-3 and Omniware are included, showing that a sub-10µs upcall time is needed for user-level servers to compete with compiled, downloaded code here.

close to or less than one (i.e. the Tcl code would have to save a page each time it was called in order to not degrade performance), Tcl is not competitive here with the other technologies.

5.5 MD5 Fingerprinting

We took the standard C implementation of the MD5 algorithm (found in RFC1321) and modified or reimplemented it for each of our test platforms. The MD5 algorithm makes heavy use of array access and unsigned 32-bit arithmetic. The standard implementation takes advantage of C's behavior of silently ignoring numeric overflow by performing computation modulo 2^{32}. The code worked correctly in C on all platforms, and was easily translated into Java and Modula-3.

This test is very compute intensive. It uses almost none of the support facilities of the extension technology (e.g. no data types other than array), but instead gives an indication of the cost of performing array access and numerical computation. The results are shown in Table 5.

Because the cost of computing the fingerprint is so high, the cost of an upcall is comparatively low. If we assume that there would be one upcall to the user-level extension for every 64KB read from disk, we would need to add the cost of 1MB/64KB = 16 upcalls. Assuming a (pessimistic) upcall time of 50µs, this would add 8ms to the raw C time, which is not significant when added to compute times of 150ms to 800ms.

To compute a break-even point, we measured the write bandwidth of the disk on each system (Table 4). This measurement was used to estimate the time required to read 1MB of data. If the time needed to compute the checksum of 1MB of data is less than that required to read the data from the disk, computation can time can be hidden by I/O time. (This optimistically assumes that the disk read requires no CPU cycles, and is a best-case break-even calculation.)

Platform	**Bandwidth (KB/s)**	**1MB access time**
Alpha	4364(1.2%)	235ms
HP-UX	1855(13%)	552ms
Linux	1694(5.7%)	604ms
Solaris	3126(11%)	320ms

Table 4. Disk I/O Time Write bandwidth in KB/s on each platform, measured using *lmbench*. From this, the time to access 1MB of data is computed. Each time is the mean of 30 runs (standard deviations in parenthesis).

The Omniware code is faster than the Modula-3 code, but slower than compiled C. To see if the size of the test was affecting the results, we ran a larger test (64MB) and saw a similar overhead (14480ms for Omniware vs. 9498ms for C, an overhead of 50%). We believe that this overhead is representative of this implementation of Omniware for MD5. (Once again,

Platform		C	Java	Modula-3	Omniware
	raw	159ms(1.8%)		207ms(0.4%)	
Alpha	normalized	1.0	N.A.	1.3	N.A.
	MD5/disk	0.67		0.9	
	raw	239ms(1.6%)	23987ms(2.5%)	352ms(0.3%)	
HP-UX	normalized	1.0	100	1.5	N.A.
	MD5/disk	0.43	43	0.64	
	raw	202ms(0.3%)	22887ms(1.0%)	387ms(0.1%)	
Linux	normalized	1.0	113	1.9	N.A.
	MD5/disk	0.33	38	0.64	
	raw	146ms(1.7%)	10368ms(0.3%)	294ms(0.1%)	219ms(0%)
Solaris	normalized	1.0	71	2.0	1.5
	MD5/disk	0.46	32	0.92	0.68

Table 5. MD5 Fingerprinting Mean time required to compute the MD5 fingerprint of 1MB of data. The time is compared to the time needed to read 1MB from the disk. If this number is less than one, the computation of the fingerprint can be overlapped with I/O. If it is greater than one, computing the fingerprint will decrease throughput. The mean of 30 runs is reported (standard deviations in parenthesis).

this version of Omniware does not include read protection, which gives it a performance advantage over Modula-3.)

Unlike C, Modula-3 does not ignore arithmetic overflow. The Word package supports computation modulo the native wordsize, so on the 32-bit platforms (HP-UX, Linux, and Solaris) we were able to implement MD5 efficiently. On the 64-bit Alpha, the Word package performs computation modulo 2^{64}, which produces incorrect results for MD5. We implemented two versions of MD5 on Alpha: one uses 64-bit integers and the Word package, doing roughly the same amount of work as the 32-bit implementations (but computing an incorrect checksum), and the other using 32-bit integers, which produces the correct checksum, but generates more instructions (by a factor of four). For our performance measurements we used the 64-bit version, which gives a more accurate comparison of the relative performance of the technologies. We found that the 32-bit version took approximately ten times as long to run as the 64-bit version. (We see this as an artifact of the compiler implementation, and not a characteristic of the Alpha processor or of Modula-3.)

The Modula-3 code runs from 1/2 to 3/4 the speed of compiled C. There is no reason for the numerical computation to be slower; we attribute the difference to run-time array bounds checking. On all platforms, the time required to compute the fingerprint was less than the time to read the data from the disk, so a Modula-3 implementation of MD5 could keep up with disk access and overlap its computation with I/O time.

On the other hand, we found that neither Java nor Tcl were able to keep up with the disk. The Java code, at best 1/30th disk speed, seems unlikely to be used for this application. And, as above, we found that our Tcl implementation was four orders of magnitude slower than one written in a compiled language, and hence too slow for this type of graft. (On Solaris it took 50 minutes to complete, as compared with 1.9 seconds for the C code.)

5.6 Logical Disk

Our simulation models a logical disk designed to support a log-structured layer between a filesystem and the physical disk. The simulation accepts write requests for logical blocks and maintains the mapping between these logical blocks and the physical blocks onto which they are stored. As with the system implemented by de Jonge et al. [DEJON93], our simulation maintains all data structures in main memory.

We simulate a 1GB physical disk with 4KB blocks and 64KB (16 block) segments. Our simulation uses a stream of block write requests that are skewed so that 80% of the requests are for 20% of the blocks. Because our simulation does not include a cleaner, we run it for 262144 iterations (the number of blocks on the disk).

We measured the absolute time required to maintain the mapping between logical and physical blocks for the entire run. To justify using this graft, it must save more time than it takes: the overhead incurred per write should be less than the time saved

Platform		**C**	**Java**	**Modula-3**	**Omniware**
	raw	0.74s(1.9%)		1.3s(2.0%)	
Alpha	normalized	1.0	N.A.	1.75	N.A.
	per block	2.8µs		5.0µs	
	raw	1.3s(1.3%)	32.2s(0.6%)	2.1s(0.7%)	
HP-UX	normalized	1.0	25	1.6	N.A.
	per block	5.0µs	123µs	8.0µs	
	raw	1.3s(0.8%)	46.5s(0.1%)	1.7s(0.9%)	
Linux	normalized	1.0	36	1.3	N.A.
	per block	5.0µs	177µs	6.6µs	
	raw	1.9s(0.2%)	24.6s(0.4%)	2.9s(0.4%)	2.2s(0.1%)
Solaris	normalized	1.0	13	1.5	1.16
	per block	7.2µs	94µs	11.1µs	8.4µs

Table 6. Logical Disk Time to handle bookkeeping for 262,144 writes to a Logical Disk. The time is normalized to compiled C code. The per-block overhead is how much time must be saved on each write in order for the graft to break even. The mean of 30 runs is reported (standard deviations in parenthesis).

by batching writes into segments. We found that the compiled technologies (Omniware and Modula-3) add a sub-10µs overhead per write, which is on the order of 1% of a typical disk seek time.

The Java overhead is on the order of 100µs, roughly 10% of a typical seek, so we would need to save one seek for every ten blocks written. This is not an unreasonable assumption to make; interpreted Java would work for this task. (Because of performance of Tcl on the first two tests, we did not take Tcl measurements for this test.)

When looking at a user-level server for managing the mapping, we assume that there is one upcall on each block write, with a upcall estimate of 10µs. This would double the overhead per write, but still keep it substantially lower than that of interpreted Java. The performance of a user-level server for this task would be relatively close to that of compiled code.

6 Conclusions

We believe that our taxonomy of grafts and of graft architectures encompasses a significant share of the operating system extension space. Our sample grafts (VM page eviction, MD5 fingerprinting, and Logical Disk) are equivalent in structure and performance characteristics to the policy, performance, and functionality grafts we envision. Given the performance of these representative grafts, we are able to determine what kinds of extensions are worth building.

The anecdotal evidence has been that structuring a system with fine-grained user-level extensions and upcalls is not feasible; our VM page eviction test supports this position. On the other hand, the coarse-grained and computationally expensive MD5 test shows that there are some situations where this overhead does not matter, and the Logical Disk simulation shows that upcalls can be successfully hidden in the face of a large number of I/O operations.

Given that we want both safety and performance, a compiled technology such as Modula-3 or SFI is our best choice. Because Modula-3 is unfamiliar to many developers, it may be that a hybrid language with the syntax of C or C++ but the safety of Modula-3 would be more widely accepted. The two compelling candidates are compiled Java and SFI with full (read, write, and jump) protection. Neither is available today, but both are currently under development. In the near future we will be able to measure their performance and compare them directly.

In their current state, neither of the interpreted technologies are up to the task. While Tcl and Java are well suited for interactive applications, where the relevant metric is human perceptual time, they are not suitable for building kernel extensions because system events occur at a finer timing granularity.

7 Status and Availability

All code is available on the World Wide Web, at `http://www.eecs.harvard.edu/~chris`.

Acknowledgments

The authors are grateful to the reviewers for providing valuable feedback. David Black of OSF, the shepherd, was invaluable in helping with the paper, especially in coming up with the scheme for measuring signal delivery time. Special thanks to OSF for allowing us to use a pre-release copy of their port of Java to HP-UX. David Holland kindly lent us his Linux box (and all its free disk space) to obtain the Linux measurements. Dawson Engler of MIT made the helpful suggestion of presenting the break-even times as a function of upcall time. And special thanks to Keith Smith and Chris Thorpe for their help brainstorming and polishing the paper.

We are very grateful to Larry McVoy of SGI for use of his *lmbench* suite [MCVOY96]. We used tests `lmdd.c,v 1.12` and `lat_pagefault.c,v 1.2`.

Bibliography

[ACCE86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: a New Kernel Foundation for UNIX Development," *1986 Summer USENIX Conference* (July 1986).

[BERS95] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proceedings of the 15th SOSP*, Copper Mountain, CO (December 1995).

[CAMP95] Campbell, R., Tan. S.-M., ''µChoices: An Object-Oriented Multimedia Operating System," *Proceedings of HotOS V*, pp. 90–94, Orcas Island, WA (May 1995).

[CAO94] Cao, P., Felten, E., Li, K., "Implementation and Performance of Application-Controlled File Caching," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation*, pp. 165-177, Montery, CA (November 1994).

[COLU95] "Omniware Technical Overview", Colusa Software, http://www.colusa.com, (1995).

[DEJON93] de Jonge, W., Kaashoek, M. F., Hsieh, W., "The Logical Disk: A New Approach to Improving File Systems," *Proceedings of the 14th SOSP*, pp. 15–28, Asheville, NC (December 1993).

[DRUS93] Druschel, P., Peterson, L., "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proceedings of the 14th SOSP*, pp. 189–202, Asheville, NC (December 1993).

[ENGL95] Engler, D., Kaashoek, M. F., and O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proceedings of the 15th SOSP*, Copper Mountain, CO (December 1995).

[FALL93] Fall, K., Pasquale, J., "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability," *1993 Winter USENIX Conference*, pp. 327–334, San Diego, CA (January 1993).

[FORD94] Ford, B, Lepreau, J, "Evolving Mach 3.0 to a Migrating Thread Model", *1994 Winter USENIX Conference*, pp. 97-114, San Francisco, CA (January 1994).

[GOSL95] Gosling, J., McGilton, H., "The Java Language Environment," available from `http://java.sun.com` (May 1995).

[GUIL91] Guillemont, M., Lipkis, J., Orr, D., Rozier, M., "A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility," *1991 Winter USENIX Conference*, pp. 13–21, Dallas, TX (January 1991).

[HOLZE94] Hölze, U., Ungar, D., "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback", *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL (June 1994).

[ILLU94] "Illustra DataBlade Developer's Kit Architecture Manual, Release 1.1," Illustra Information Technologies, Oakland, CA (1994).

[LEE94] Lee, C.-H., Chen, M., Chang, R. C., "HiPEC: High Performance External Virtual Memory Caching," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation*, pp. 153–164, Montery, CA (November 1994).

[LISK95] Liskov, B., Curtis, D., Day, M., Ghemaway, S., Gruber, R., Johnson. P., Myers, A., "Theta Reference Manual," MIT LCS Programming Methodology Group Memo 88 (February 1995).

[MAY87] May, C., “MIMIC: A Fast System/370 Simulator,” *Proceedings of the SIGPLAN ‘87 Symposium on Interpreters and Interpretive Techniques*, in *SIGPLAN Notices*, *22*, 7, pp. 1-13, St. Paul, MN (July 1987).

[MCCAN93] McCanne, S., Jacobson, V., “The BSD Packet Filter: A New Architecture for User-level Packet Capture,” *1993 Winter USENIX Conference,* San Diego, CA (January 1993).

[MCVOY96] McVoy, L., Staelin, C., “lmbench: Portable Tools for Performance Analysis,” *1996 USENIX Conference*, San Diego, CA (January 1994).

[MOGUL87] Mogul, J., Rashid, R., Accetta, M., “The Packet Filter: An Efficient Mechanism for User-level Network Code,” *Proceedings of the 11th SOSP*, pp. 39–52, Austin, TX (November 1987).

[NELS91] *Systems Programming with Modula-3*, Nelson, G., ed., Prentice Hall, Englewood Cliffs, NJ (1991).

[RFC1321] Rivest, R., “The MD5 Message-Digest Algorithm,” *Network Working Group RFC 1321* (April 1992).

[RITCH84] Ritchie, D., “A Stream Input-Output System,” *AT&T Bell Laboratories Technical Journal, 63,* 8 pp. 1897–1910 (October 1984).

[ROSE91] Rosenblum, M., Ousterhout, J., “The Design and Implementation of a Log-Structured File System,” *Proceedings of the 13th SOSP,* pp. 1–15, Pacific Grove, CA (October 1991).

[TPCB90] Transaction Processing Performance Council, “TPC Benchmark B,” Standard Specification, Waterside Associates, Fremont, CA (1990).

[THEK94] Thekkath, C., Levy, H, “Hardware and Software Support for Efficient Exception Handling,” *Proceedings of ASPLOS VI,* pp. 110-119, San Jose, CA (October 1994).

[WAHBE93] Wahbe, R., Lucco, S., Anderson, T., Graham, S., “Efficient Software-Based Fault Isolation,” *Proceedings of the 14th SOSP,* pp. 203–216 Asheville, NC (December 1993).

[YUHURA94] Yuhara, M., Bershad, B., Maeda, C., Moss, J., “Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages,” *1994 Winter USENIX Conference*, pp. 153–166, San Francisco, CA (January 1994).

Christopher A. Small (chris@eecs.harvard.edu) is a Ph.D. candidate in Computer Science at Harvard University. His research interests include the interaction of language systems, database systems, and operating systems. He received his B.A. in Mathematics and his M.A. in Computer Science from Boston University in 1984, where he was a Trustee Scholar. He worked at an unreasonably large number of companies in the Boston area before entering Harvard’s Ph. D. program in 1993.

Margo I. Seltzer (margo@eecs.harvard.edu) is an Assistant Professor of Computer Science in the Division of Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She is the co-author of several widely-used software packages including database and transaction libraries and the 4.4BSD log-structured file system. Dr. Seltzer spent several years working at start-up companies designing and implementing file systems and transaction processing software and designing microprocessors. She is a Sloan Foundation Fellow in Computer Science and was the recipient of the University of California Microelectronics Scholarship, the Radcliffe College Elizabeth Cary Agassiz Scholarship, and the John Harvard Scholarship. Dr. Seltzer received an A.B. degree in Applied Mathematics from Harvard/Radcliffe College in 1983 and a Ph. D. in Computer Science from the University of California, Berkeley in 1992.
