Application Performance and Flexibility on Exokernel Systems

M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger,
Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney,
Robert Grimm, John Jannotti, and Kenneth Mackenzie
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A
http://www.pdos.lcs.mit.edu/

Abstract

The exokernel operating system architecture safely gives untrusted software efficient control over hardware and software resources by separating management from protection. This paper describes an exokernel system that allows specialized applications to achieve high performance without sacrificing the performance of unmodified UNIX programs. It evaluates the exokernel architecture by measuring end-to-end application performance on Xok, an exokernel for Intel x86-based computers, and by comparing Xok's performance to the performance of two widely-used 4.4BSD UNIX systems (FreeBSD and OpenBSD). The results show that common unmodified UNIX applications can enjoy the benefits of exokernels: applications either perform comparably on Xok/ExOS and the BSD UNIXes, or perform significantly better. In addition, the results show that customized applications can benefit substantially from control over their resources (e.g., a factor of eight for a Web server). This paper also describes insights about the exokernel approach gained through building three different exokernel systems, and presents novel approaches to resource multiplexing.

1 Introduction

In traditional operating systems, only privileged servers and the kernel can manage system resources. Untrusted applications are restricted to the interfaces and implementations of this privileged software. This organization is flawed because application demands vary widely. An interface designed to accommodate every application must anticipate all possible needs. The implementation of such an interface would need to resolve all tradeoffs and anticipate all ways the interface could be used. Experience suggests that such anticipation is infeasible and that the cost of mistakes is high [1, 4, 8, 11, 21, 39].

The *exokernel architecture* [11] solves this problem by giving untrusted applications as much control over resources as possible. It does so by dividing responsibilities differently from the way conventional systems do. Exokernels separate protection from management: they protect resources but delegate management to applications. For example, each application manages its own disk-block cache, but the exokernel allows cached pages to be shared securely across all applications. Thus, the exokernel protects pages and disk blocks, but applications manage them.

Of course, not all applications need customized resource management. Instead of communicating with the exokernel directly, we expect most programs to be linked with libraries that hide low-level resources behind traditional operating system abstractions. However, unlike traditional implementations of these abstractions, library implementations are unprivileged and can therefore be modified or replaced at will. We refer to these unprivileged libraries as *library operating systems,* or libOSes.

We hope the exokernel organization will facilitate operating system innovation: there are several orders of magnitude more application programmers than OS implementors, and any programmer can specialize a libOS without affecting the rest of the system. LibOSes also allow incremental, selective adoption of new OS features: applications link with the libOSes that provide what they need—new OS functionality is effectively distributed with the application binary.

The exokernel approach raises several questions. Can ambitious applications actually achieve significant performance improvements on an exokernel? Will traditional applications—for example, unaltered UNIX applications—pay a price in reduced performance? Is global performance

This research was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985 and by a NSF National Young Investigator Award. Robert Grimm is currently at the University of Washington, Seattle.

compromised when no centralized authority decides scheduling and multiplexing policies? Does the lack of a centralized management policy for shared OS structures lower the integrity of the system?

This paper attempts to answer these questions and thereby evaluate the soundness of the exokernel approach. Our experiments are performed on the Xok/ExOS exokernel system. Xok is an exokernel for Intel x86-based computers and ExOS is its default libOS. Xok/ExOS compiles on itself and runs many unmodified UNIX programs (e.g., perl, gcc, telnet, and most file utilities). We compare Xok/ExOS to two widely-used 4.4BSD UNIX systems running on the same hardware, using large, real-world applications.

ExOS ensures the integrity of many of its abstractions using Xok's support for protected sharing. Some abstractions, however, still use shared global data structures. ExOS cannot guarantee UNIX semantics for these abstractions until they are protected from arbitrary writes by other processes. In our measurements, we approximate the cost of this protection by inserting system calls before all writes to shared global state.

Our results show that most unmodified UNIX applications perform comparably on Xok/ExOS and on FreeBSD or OpenBSD. Some applications, however, run up to a factor of four faster on Xok/ExOS. Experiments with multiple applications running concurrently also show that exokernels can offer competitive global system performance.

We also demonstrate that application-level control can significantly improve the performance of applications. For example, we describe a new high-performance HTTP server, Cheetah, that actively exploits exokernel extensibility. Cheetah uses a file system and a TCP implementation customized for the properties of HTTP traffic. Cheetah performs up to eight times faster than the best UNIX HTTP server we measured on the same hardware.

In addition to evaluating the exokernel approach, this paper presents new kernel interfaces that separate protection from management. We discuss the disk subsystem, XN, and explain how unprivileged applications can define new file systems and how these file systems can safely multiplex the same disk at a fine granularity. Finally, we summarize what we have learned from building three complete exokernel systems (Xok, Aegis [11] for DECstations, and Glaze [29] for the Fugu multiprocessor).

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 summarizes the exokernel architecture. Section 4 provides a detailed example of reconciling application control with protection by presenting the disk system XN. Section 5 briefly overviews Xok/ExOS, the experimental environment for this paper. Section 6 reports on the performance of unaltered UNIX applications, while Section 7 reports on the performance of aggressively-specialized applications, such as the high-performance Cheetah web server. Section 8 investigates global performance on an exokernel system. Section 9 discusses our experiences with building three different exokernel systems. Section 10 concludes.

2 Related Work

The exokernel architecture was proposed in [11], which described a research prototype that performed significantly better than Ultrix on microbenchmarks. While the paper provided evidence that the exokernel approach was promising, it left many questions unanswered.

There is a large literature on extensible operating systems, starting with the classic rationales by Lampson and Brinch Hansen [19, 25, 26]. Previous approaches to extensibility can be coarsely classified in three groups: better microkernels, virtual machines, and downloading untrusted code into the kernel. We discuss each in turn.

The principal goal of an exokernel—giving applications control—is orthogonal to the question of monolithic versus microkernel organization. If applications are restricted to inadequate interfaces, it makes little difference whether the implementations reside in the kernel or privileged user-level servers [20, 18]; in both cases applications lack control. For example, it is difficult to change the buffer management policy of a shared file server. In many ways, servers can be viewed as fixed kernel subsystems that happen to run in user space. Whether monolithic or microkernel-based, the goal of an exokernel system remains for privileged software to provide interfaces that do not limit the ability of unprivileged applications to manage their own resources.

Some newer microkernels push the kernel interface closer to the hardware [8, 20, 36], obtaining better performance and robustness than previous microkernels and allowing for a greater degree of flexibility, since shared monolithic servers can be broken into several servers. Techniques to reduce the cost of shared servers by improving IPC performance, moving code from servers into libraries, mapping read-only shared data structures, and batching system calls [2, 18, 28, 30] can also be successfully applied in an exokernel system.

Virtual machines [5, 12, 17] (VMs) are an OS structure in which a privileged virtual machine monitor (VMM) isolates less privileged software in emulated copies of the underlying hardware. Unfortunately, emulation hides information. This can lead to ineffective use of hardware resources; for instance, the VMM has no way of knowing if a VM no longer needs a particular virtual page. Moreover, VMs can only share resources through remote communication protocols. This prevents VMs from sharing many OS abstractions such as processes or file descriptors with each other. Thus, VMMs confine specialized operating systems and associated processes to isolated virtual machines, while exokernels let applications use customized libOSes without sacrificing a single view of the machine,

Downloading code into the kernel is another approach to extensibility. In many systems only trusted users can download code, either through dynamically-loaded kernel extensions or static configuration [13, 21]. In the SPIN and Vino systems, any user can safely download code into the kernel [4, 39]. Safe downloading of code through type-safety [4, 37] and software fault-isolation [39, 42] is comple-

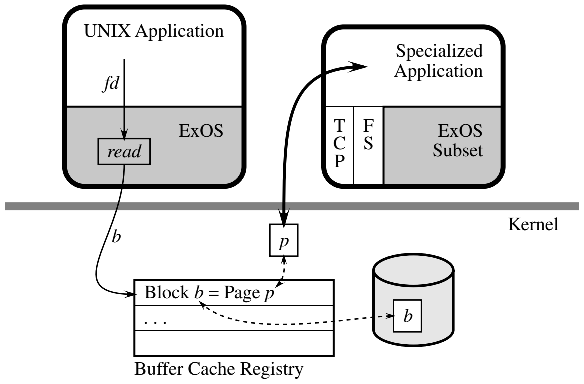

Figure 1. A simplified exokernel system with two applications, each linked with its own libOS and sharing pages through a buffer cache registry.

mentary to the exokernel approach of separating protection from management. Exokernels use downloading of code to let the kernel leave decisions to untrusted software [11].

In addition to these structural approaches, much work has been done on better OS abstractions that give more control to applications, such as user-level networking [40, 41], lottery scheduling [43], application-controlled virtual memory [22, 27] and file systems [6, 35]. All of this work is directly applicable to libOSes.

3 Exokernel Background

This section briefly summarizes the exokernel architecture. Figure 1 shows a simplified exokernel system that is running two applications: an unmodified UNIX application linked against the ExOS libOS and a specialized exokernel application using its own TCP and file system libraries. Applications communicate with the kernel using low-level physical names (e.g., block numbers); the kernel interface is as close to the hardware as possible. LibOSes handle higher-level names (e.g., file descriptors) and supply abstractions.

We briefly describe the exokernel principles, motivated in [11]. These principles illustrate the mechanics of exokernel systems and provide important motivation for many design decisions discussed later in this paper. In addition, we show how the principles can be applied and discuss the general issue of protected sharing.

3.1 Exokernel principles

The goal of an exokernel is to give efficient control of resources to untrusted applications in a secure, multi-user system. We follow these principles to achieve this goal:

Separate protection and management. Exokernels provide primitives at the lowest possible level required for protection—ideally, at the level of hardware (disk blocks, context identifiers, TLB, etc.). Resource management is restricted to functions necessary for protection: allocation, revocation, sharing, and the tracking of ownership.

Expose allocation. Applications allocate resources explicitly. The kernel allows specific resources to be requested during allocation.

Expose names. Exokernels use physical names wherever possible. Physical names capture useful information and do not require potentially costly or race-prone translations from virtual names.

Expose revocation. Exokernels expose revocation policies to applications. They let applications choose which instance of a resource to give up. Each application has control over its set of physical resources.

Expose information. Exokernels expose all system information and collect data that applications cannot easily derive locally. For example, applications can determine how many hardware network buffers there are or which pages cache file blocks. An exokernel might also record an approximate least-recently-used ordering of all physical pages, something individual applications cannot do without global information.

These principles apply not just to the kernel, but to any component of an exokernel system. Privileged servers should provide an interface boiled down to just what is required for protection.

3.2 Kernel support for protected abstractions

Many of the resources protected by traditional operating systems are themselves high-level abstractions. Files, for instance, consist of metadata, disk blocks, and buffer cache pages, all of which are guarded by access control on high-level file objects. While exokernels allow direct access to low-level resources, exokernel systems must be able to provide UNIX-like protection, including access control on high-level objects where required for security. One of the main challenges in designing exokernels is to find kernel interfaces that allow such higher-level access control without either mandating a particular implementation or hindering application control of hardware resources.

Xok meets this challenge with three design techniques. First, it performs access control on all resources in the same manner. Second, Xok provides software abstractions to bind hardware resources together. For example, as shown in Figure 1, the Xok buffer cache registry binds disk blocks to the memory pages caching them. Applications have control over physical pages and disk I/O, but can also safely use each other's cached pages. Xok's protection mechanism guarantees that a process can only access a cache page if it has the same level of access to the corresponding disk block. Third, and most general, some of Xok's abstractions allow applications to download code. This is required for abstractions whose protection does not map to hardware abstractions. For example, files may require valid updates to their modification times.

The key to these exokernel software abstractions is that they neither hinder low-level access to hardware resources nor unduly restrict the semantics of the protected abstractions they enable. Given these properties, a kernel software abstraction does not violate the exokernel principles.

Though these software abstractions reside in the kernel on Xok, they could also be implemented in trusted user-level servers. This microkernel organization would cost many additional context switches; these are particularly expensive on the Intel Pentium Pro processors on which Xok runs. Furthermore, partitioning functionality in user-level servers tends to be more complex.

3.3 Protected sharing

The low-level exokernel interface gives libOSes enough hardware control to implement all traditional operating system abstractions. Library implementations of abstractions have the advantage that they can trust the applications they link with and need not defend against malicious use. The flip side, however, is that a libOS cannot necessarily trust all other libOSes with access to a particular resource. When libOSes guarantee invariants about their abstractions, they must be aware of exactly which resources are involved, what other processes have access to those resources, and what level of trust they place in those other processes.

As an example, consider the semantics of the UNIX fork system call. It spawns a new process initially identical to the currently running one. This involves copying the entire virtual address space of the parent process, a task operating systems typically perform lazily through copy-on-write to avoid unnecessary page copies. While copy-on-write can always be done in a trusted, in-kernel virtual memory system, a libOS must exercise care to avoid compromising the semantics of fork when sharing pages with potentially untrusted processes. This section details some of the approaches we have used to allow a libOS to maintain invariants when sharing resources with other libOSes.

The exokernel provides four mechanisms libOSes can use to maintain invariants in shared abstractions. First, *software regions*, areas of memory that can only be read or written through system calls, provide sub-page protection and fault isolation. Second, the exokernel allows on the-fly-creation of *hierarchically-named capabilities* and requires that these capabilities be specified explicitly on each system call [31]. Thus, a buggy child process accidentally requesting write access to a page or software region of its parent will likely provide the wrong capability and be denied permission. Third, the exokernel provides *wakeup predicates*: small, kernel-downloaded functions that wake up processes when arbitrary conditions become true (see Section 5.1 for details). Wakeup predicates can ensure that a buggy or crashed process will not hang a correctly behaved one. Fourth, the exokernel provides robust critical sections: inexpensive critical sections that are implemented by disabling software interrupts [3]. Using critical sections instead of locks eliminates the need to trust other processes.

Three levels of trust determine what optimizations can be used by the implementation of a shared abstraction.

Optimize for the common case: Mutual trust. It is often the case that applications sharing resources place a considerable amount of trust in each other. For instance, any two UNIX programs run by the same user can arbitrarily modify each others' memory through the debugger system call, ptrace. When two exokernel processes can write each others' memory, their libOSes can clearly trust each other not to be malicious. This reduces the problem of guaranteeing invariants from one of security to one of fault-isolation, and consequently allows libOS code to resemble that of monolithic kernels implementing the same abstraction.

Unidirectional trust. Another common scenario occurs when two processes share resources and one trusts the other, but the trust is not mutual. Network servers often follow this organization: a privileged process accepts network connections, forks, and then drops privileges to perform actions on behalf of a particular user. Many abstractions implemented for mutual trust can also function under unidirectional trust with only slight modification. In the example of copy-on-write, for instance, the trusted parent process must retain exclusive control of shared pages and its own page tables, preventing a child from child making copied pages writable in the parent. While this requires more page faults in the parent, it does not increase the number of page copies or seriously complicate the code.

Defensive programming for mutual distrust. Finally, there are situations where mutually distrustful processes must share high-level abstractions with each other. For instance, two unrelated processes may wish to communicate over a UNIX domain socket, and neither may have any trust in the other. For OS abstractions that can be shared by mutually distrustful processes, libOSes must include defensive implementations that give reasonable interpretations to all possible actions by the foreign process (for instance a socket write larger than the buffer can be interpreted as an end of file).

Fortunately, sharing with mutual distrust occurs very infrequently for many abstractions. Many types of sharing occur only between child and parent processes, where mutual or unidirectional trust almost always holds. Where mutual distrust does occur, defensive sanity checks are often not on the critical path for performance. In the remaining cases, as is the case for disk files, we have carefully crafted kernel software abstractions to help libOSes maintain the necessary invariants.

4 Multiplexing Stable Storage

An exokernel must provide a means to safely multiplex disks among multiple library file systems (libFSes). Each libOS contains one or more libFSes. Multiple libFSes can be used to share the same files with different semantics. In addition to accessing existing files, libFSes can define new on-disk file types with arbitrary metadata formats. An exokernel must give libFSes as much control over file management as possible while still protecting files from unauthorized access. It therefore cannot rely on simple-minded solutions like partitioning to multiplex a disk: each file would require its own partition.

To allow libFSes to perform their own file management, an exokernel stable storage system must satisfy four requirements. First, creating new file formats should be simple and lightweight. It should not require any special privilege. Second, the protection substrate should allow multiple libFSes to safely share files at the raw disk block and metadata level. Third, the storage system must be efficient—as close to raw hardware performance as possible. Fourth, the storage system should facilitate cache sharing among libFSes, and allow them to easily address problems of cache coherence, security, and concurrency.

This section describes how Xok multiplexes stable storage, both to show how we address these problems and to provide a concrete example of the exokernel principles in practice. First, we describe XN, Xok's extensible, low-level in-kernel stable storage system. We also describe the general interface between XN and libFSes and present one particular libFS, C-FFS, the co-locating fast file system [15].

4.1 Overview of XN

Designing a flexible exokernel stable storage system has proven difficult: XN is our fourth design. This section provides an overview of UDFs, the cornerstone of XN; the following sections describe some earlier approaches (and why they failed), and aspects of XN in greater depth.

XN provides access to stable storage at the level of disk blocks, exporting a buffer cache registry (Section 4.3.3) as well as free maps and other on-disk structures. The main purpose of XN is to determine the access rights of a given principal to a given disk block as efficiently as possible. XN must prevent a malicious user from claiming another user's disk blocks as part of her own files. On a conventional OS, this task is easy, since the kernel itself knows the file's metadata format. On an exokernel, where files have application-defined metadata layouts, the task is more difficult.

XN's novel solution employs *UDFs* (*untrusted deterministic functions*). UDFs are metadata translation functions specific to each file type. XN uses UDFs to analyze metadata and translate it into a simple form the kernel understands. A libFS developer can install UDFs to introduce new on-disk metadata formats. The restricted language in which UDFs are specified ensures that they are deterministic—their output depends only on their input (the metadata itself). UDFs allow the kernel to safely and efficiently handle any metadata layout without understanding the layout itself.

UDFs are stored on disk in structures called *templates.* Each template corresponds to a particular metadata format; for example, a UNIX file system would have templates for data blocks, inode blocks, inodes, indirect blocks, etc. Each template T has one UDF: *owns-udf*$_T$, and two untrusted but potentially nondeterministic functions: *acl-uf*$_T$ and *size-uf*$_T$. All three functions are specified in the same language but only *owns-udf*$_T$ must be deterministic. The other two can have access to, for example, the time of day. The limited language used to write these functions is a pseudo-RISC assembly language, checked by the kernel to ensure determinacy. Once a template is specified, it cannot be changed.

For a piece of metadata m of template type T, *owns-udf*$_T(m)$ returns the set of blocks which m points to and their respective template types. UDF determinism guarantees that *owns-udf* will always compute the same output for a given input: XN cannot be spoofed by *owns-udf.* The set of blocks *owns-udf* returns is represented as a set of tuples. Each tuple constitutes a range: a block address that specifies the start of the range, the number of blocks in the range, and the template identifier for the blocks in the range. Because owned sets can be large, XN allows libFSes to partition metadata blocks into disjoint pieces such that each set returned is (typically) a single tuple.

For example, say a libFS wants to allocate a disk block b by placing a pointer to it in a metadata structure, m. The libFS will call XN, passing it m, b, and the proposed modification to m (specified as a list of bytes to write into m). To enforce protection, XN needs to know that the libFS's proposed modification actually does what it says it does—that is, allocates b in m. Thus, XN runs *owns-udf*$_T(m)$; makes the proposed modification on m', a copy of m; and runs *owns-udf*$_T(m')$. It then verifies that the new result is equal the old result plus b.

The *acl-uf* function implements template-specific access control and semantics; its input is a piece of metadata, a proposed modification to that metadata, and set of credentials (e.g., capabilities). Its output is a Boolean value approving or disapproving of the modification. XN runs the proper *acl-uf* function before any metadata modification. *acl-uf*s can implement access control lists, as well as providing certain other guarantees; for example, an *acl-uf* could ensure that inode modification times are kept current by rejecting any metadata changes that do not update them.

The *size-uf* function simply returns the size of a data structure in bytes.

4.2 XN: Problem and history

The most difficult requirement for XN is efficiently determining the access rights of a given principal to a given disk block. We discuss the successive approaches that we have pursued.

Disk-block-level multiplexing. One approach is to associate with each block or extent a capability (or access control list) that guards it. Unfortunately, if the capability is spatially separated from the disk block (e.g., stored separately in a table), accessing a block can require two disk accesses (one to fetch the capability and one to fetch the block). While caching can mitigate this problem to a degree, we are nervous about its overhead on disk-intensive workloads. An alternative approach is to co-locate capabilities with disk blocks by placing them immediately before a disk block's data [26]. Unfortunately, on common hardware, reserving space for a capability would prevent blocks from being multiples of the page size, adding overhead and complexity to disk operations.

Self-descriptive metadata. Our first serious attempt at efficient disk multiplexing provided a means for each instance of metadata to describe itself. For example, a disk block would start with some number of bytes of application-specific data and then say "the next ten integers are disk block pointers." The complexity of space-efficient self-description caused us to limit what metadata could be described. We discovered that this approach both caused unacceptable amounts of space overhead and required excessive effort to modify existing file system code, because it was difficult to shoehorn existing file system data structures into a universal format.

Template-based description. Self-description and its problems were eliminated by the insight that each file system is built from only a handful of different on-disk data structures, each of which can be considered a type. Since the number of types is small, it is feasible to describe each type only once per file system—rather than once per instance of a type—using a *template*.

Originally, templates were written in a declarative description language (similar to that used in self-descriptive metadata) rather than UDFs. This system was simple and better than self-descriptive metadata, but still exhibited what we have come to appreciate as an indication that applications do not have enough control: the system made too many tradeoffs. We had to make a myriad of decisions about which base types were available and how they were represented (how large disk block pointers could be, how the type layout could change, how extents were specified). Given the variety of on-disk data structures described in the file system literature, it seems unlikely that any fixed set of components will ever be enough to describe all useful metadata.

Our current solution uses templates, but trades the declarative description language for a more expressive, interpreted language—UDFs. This lets libFSes track their own access rights without XN understanding how they do so; XN merely verifies that libFSes track block ownership correctly.

4.3 XN: Design and implementation

We first describe the requirements for XN and then present the design.

4.3.1 Requirements and approach

In our experience so far, the following requirements have been sufficient to reconcile application control with protected sharing.

1. To prevent unauthorized access, every operation on disk data must be guarded. For speed, XN uses *secure bindings* [11] to move access checks to bind time rather than checking at every access. For example, the permission to read a cached disk block is checked when the page is inserted into the page table of the libFS's environment, rather than on every access.

2. XN must be able to determine unambiguously what access rights a principal has to a given disk block. For speed, it uses the UDF mechanism to protect disk blocks using the libFS's own metadata rather than guarding each block individually.

3. XN must guarantee that disk updates are ordered such that a crash will not incorrectly grant a libFS access to data it either has freed or has not allocated. This requirement means that metadata that is persistent across crashes cannot be written when it contains pointers to uninitialized metadata, and that reallocation of a freed block must be delayed until all persistent pointers to it have been removed.

While isolation allows separate libFSes to coexist safely, protected sharing of file system state by mutually distrustful libFSes requires three additional features:

1. Coherent caching of disk blocks. Distributed, per-application disk block caches create a consistency problem: if two applications obliviously cache the same disk block in two different physical pages, then modifications will not be shared. XN solves this problem with an in-kernel, system-wide, protected cache registry that maps cached disk blocks to the physical pages holding them.

2. Atomic metadata updates. Many file system updates have multiple steps. To ensure that shared state always ends up in a consistent and correct state, libFSes can lock cache registry entries. (Future work will explore optimistic concurrency control based on versioning.)

3. Well-formed updates. File abstractions above the XN interface may require that metadata modifications satisfy invariants (e.g., that link counts in inodes match the number of associated directory entries). UDFs allow XN to guarantee such invariants in a file-system-specific manner, allowing mutually distrustful applications to safely share metadata.

XN controls only what is necessary to enforce these protection rules. All other abilities—I/O initiation, disk block layout and allocation policies, recovery semantics, and consistency guarantees—are left to untrusted libFSes.

4.3.2 Ordered disk writes

Another difficulty XN must face is guaranteeing the rules Ganger and Patt [16] give for achieving strict file system integrity across crashes: First, never reuse an on-disk resource before nullifying all previous pointers to it. Second, never create persistent pointers to structures before they are initialized. Third, when moving an on-disk resource, never reset the old pointer in persistent storage before the new one has been set.

The first two rules are required for global system integrity—and thus must be enforced by XN—while a file system violating the third rule will only affect itself.

The rules are simple but difficult to enforce efficiently: a naive implementation will incur frequent costly synchronous disk writes. XN allows libFSes to address this by enforcing the rules without legislating how to follow them. In particular, libFSes can choose any operation order which satisfies the constraints.

The first rule is implemented by deferring a block's deallocation until all on-disk pointers to that block have been deleted; a reference count performed at crash recovery time helps libFSes implement the third rule.

The second rule is the hardest of the three. To implement it, XN keeps track of *tainted* blocks. Any block is considered tainted if it points either to an uninitialized block or to a tainted block. LibFSes must not be allowed to write a tainted block to disk. However, two exceptions allow XN to enforce the general rule more efficiently:

First, XN allows entire file systems to be marked "temporary" (i.e., not persistent across reboots). Since these file systems are not persistent, they are not required to adhere to any of the integrity rules. This technique allows memory-based file systems to be implemented with no loss of efficiency.

The second exception is based on the observation that unattached subtrees—trees whose root is not reachable from any persistent root—will not be preserved across reboots and thus, like temporary trees, are free of any ordering constraints. Thus, XN does not track tainted blocks in an unreachable tree until it is connected to a persistent root.

4.3.3 The buffer cache registry

Finally, we discuss the XN buffer cache registry, which allows protected sharing of disk blocks among libFSes. The registry tracks the mapping of cached disk blocks and their metadata to physical pages (and vice versa). Unlike traditional buffer caches, it only records the mapping, not the disk blocks themselves. The disk blocks are stored in application-managed physical-memory pages. The registry tracks both the mapping and its state (dirty, out of core, uninitialized, locked). To allow libFSes to see which disk blocks are cached, the buffer cache registry is mapped read-only into application space.

Access control is performed when a libFS attempts to map a physical page containing a disk block into its address space, rather than when that block is requested from disk. That is, registry entries can be inserted without requiring that the object they describe be in memory. Blocks can also be installed in the registry before their template or parent is known. As a result, libFSes have significant freedom to prefetch.

Registry entries are installed in two ways. First, an application that has write access to a block can directly install a mapping to it into the registry. Second, applications that do not have write access to a block can indirectly install an entry for it by performing a "read and insert," which tells the kernel to read a disk block, associate it with an application-provided physical page, set the protection of that page page appropriately, and insert this mapping into the registry. This latter mechanism is used to prevent applications that do not have permission to write a block from modifying it by installing a bogus in-core copy.

XN does not replace physical pages from the registry (except for those freed by applications), allowing applications to determine the most appropriate caching policy. Because applications also manage virtual memory paging, the partitioning of disk cache and virtual memory backing store is under application control. To simplify the application's task and because it is inexpensive to provide, XN maintains an LRU list of unused but valid buffers. By default, when LibOSes need pages and none are free, they recycle the oldest buffer on this LRU list.

XN allows any process to write "unowned" dirty blocks to disk (i.e., blocks not associated with a running process), even if that process does not have write permission for the dirty blocks. This allows the construction of daemons that asynchronously write dirty blocks. LibFSes do not have to trust daemons with write access to their files, only to flush the blocks. This ability has three benefits. First, the contents of the registry can be safely retained across process invocations rather than having to be brought in and paged out on creation and exit. Second, this design simplifies the implementations of libFSes, since a libFS can rely on a daemon of its choice to flush dirty blocks even in difficult situations (e.g., if the application containing the libFS is swapped out). Third, this design allows different write-back policies.

4.4 XN usage

To illustrate how XN is used, we sketch how a libFS can implement common file system operations. These two setup operations are used to install a libFS:

Type creation. The libFS describes its types by storing templates, described above in Section 4.1, into a *type catalogue.* Each template is identified by a unique string (e.g., "FFS Inode"). Once installed, types are persistent across reboots.

LibFS persistence. To ensure that libFS data is persistent across reboots, a libFS can register the root of its tree in XN's *root catalogue.* A root entry consists of a disk extent and corresponding template type, identified by a unique string (e.g., "mylibFS").

After a crash, XN uses these roots to garbage-collect the disk by reconstructing the free map. It does so by logically traversing all roots and all blocks reachable from them: reachable blocks are allocated, non-reachable blocks are not. If rebuilding the free map after a crash needs to be fast, this step can be eliminated by ordering writes to the free map.

After initialization, the new libFS can use XN. We describe a simplified version of the most common operations.

Startup. To start using XN, a libFS loads its root(s) and any types it needs from the root catalogue into the buffer cache registry. Usually both will already be cached.

Read. Reading a block from disk is a two-stage process, where the stages can be combined or separated. First, the libFS creates entries in the registry by passing block addresses for the requested disk blocks and the metadata blocks controlling them (their *parents*). The parents must already exist in the registry—libFSes are responsible for loading them. XN uses *owns-udf* to determine if the requested blocks are controlled by the supplied metadata blocks and, if so, installs registry entries.

In the second stage, the libFS initiates a read request, optionally supplying pages to place the data in. Access control through *acl-uf* is performed at the parent (e.g., if the data loaded is a bare disk block), at the child (e.g., if the data is an inode), or both.

A libFS can load any block in its tree by traversing from its root entry, or optionally by starting from any intermediate node cached in the registry. Note that XN specifically disallows metadata blocks from being mapped read/write.

To speculatively read a block before its parent is known, a libFS can issue a raw read command. If the block is not in the registry, it will be marked as "unknown type" and a disk request initiated. The block cannot be used until after it is bound to a parent by the first stage of the read process, which will determine its type and allow access control to be performed.

Allocate. A libFS selects blocks to allocate by reading XN's map of free blocks, allowing libFSes to control file layout and grouping. Free blocks are allocated to a given metadata node by calling XN with the metadata node, the blocks to allocate, and the proposed modification to the metadata node. XN checks that the requested blocks are free, runs the appropriate *acl-uf* to see if the libFS has permission to allocate, and runs *owns-udf*, as described in Section 4.1, to see that the correct block is being allocated. If these checks all succeed, the metadata is changed, the allocated blocks are removed from the free list, and any allocated metadata blocks are marked tainted (see Section 4.3.2).

Write. A libFS writes dirty blocks to disk by passing the blocks to write to XN. If the blocks are not in memory, or they have been pinned in memory by some other application, the write is prevented. The write also fails if any of the blocks are tainted and reachable from a persistent root. Otherwise, the write succeeds. If the block was previously tainted and now is not (either by eliminating pointers to uninitialized metadata or by becoming initialized itself), XN modifies its state and removes it from the tainted list.

Since applications control what is fetched and what is paged out when (and in what order), they can control many disk management policies and can enforce strong stability guarantees.

Deallocate. XN uses UDFs to check deallocate operations analogously to allocate operations. If there are no on-disk pointers to a deallocated disk block, XN places it on the free list. Otherwise, XN enqueues the block on a "will free" list until the block's reference count is zero. Reference counts are decremented when a parent that had an on-disk pointer to the block deletes that pointer via a write.

4.5 C-FFS: a library file system

This subsection briefly describes C-FFS (co-locating fast file system [15])—a UNIX-like library file system we built—with special reference to additional protection guarantees it provides.

XN provides the basic protection guarantees needed for file system integrity, but real-world file systems often require other, file-system-specific invariants. For instance, UNIX file systems must ensure the uniqueness of file names within a directory. This type of guarantee can be provided in any number of ways: in the kernel, in a server, or, in some cases, by simple defensive programming. C-FFS currently downloads methods into the kernel to check its invariants. We are currently developing a system similar to UDFs that can be used to enforce type-specific invariants in an efficient, extensible way.

Our experience with C-FFS shows that, even with the strongest desired guarantees, a protected interface can still provide significant flexibility to unprivileged software, and that the exokernel approach can deal as readily with high-level protection requirements as it can with those closer to hardware.

C-FFS makes four main additions to XN's protection mechanisms:

1. Access control: it maps the UNIX representation and semantics of access control (uids and gids, etc.) to those of exokernel capabilities.

2. Well-formed updates: C-FFS guarantees UNIX-specific file semantics: for example, that directories contain legal, aligned file names.

3. Atomicity: C-FFS performs locking to ensure that its data is always recoverable and disk writes only occur when metadata is internally consistent.

4. Implicit updates: C-FFS ensures that certain state transitions are implicit on certain actions. Some examples are that modification times are updated when file data are changed, and that renaming or deleting a file updates the name cache.

It is not difficult to implement UNIX protection without significantly degrading application power. C-FFS protection is implemented mainly by a small number of if-statements rather than by procedures that limit flexibility. The most intricate operation—ensuring that files in a directory have unique names—is less than 100 lines of code that scans through a linked list of cached directory blocks to ensure name uniqueness.

4.6 Future work

Stable storage is the most challenging resource we have multiplexed. Future work will focus on two areas. First, we plan to implement a range of file systems (log-structured file systems, RAID, and memory-based file systems), thus testing if the XN interface is powerful enough to support concurrent use by radically different file systems. Second we will investigate using lightweight protected methods like UDFs to implement the simple protection checks required by higher-level abstractions.

5 Overview of Xok/ExOS

For the experiments in this paper, we use Xok/ExOS. This section describes both Xok and ExOS.

5.1 Xok

Xok safely multiplexes the physical resources on Intel x86-based computers. Xok performs this task in a manner similar to the Aegis exokernel, which runs on MIPS-based DECstations [11]. The CPU is multiplexed by dividing time into round-robin-scheduled slices with explicit notification of the beginning and the end of a time slice. Environments provide the hardware-specific state needed to run a process (e.g., an exception stack) and to respond to any event occurring during process execution (e.g., interrupts and exceptions). The network is multiplexed with dynamic packet filters [10]. This subsection briefly describes the differences between Aegis and Xok.

Physical memory. Unlike the MIPS architecture, the x86 architecture defines the page-table structure. Since x86 TLB refills are handled in hardware, this structure cannot be overridden by applications. Additionally, since the hardware does not verify that the physical page of a translation can be mapped by a process, applications are prevented from directly modifying the page table and must instead use system calls. Although these restrictions make Xok less extensible than Aegis, they simplify the implementation of libOSes (see Section 9) with only a small reduction in application flexibility.

Like Aegis, Xok allows efficient and powerful virtual memory abstractions to be built at the application level. It does so by exposing the capabilities of the hardware (e.g., all MMU protection bits) and exposing many kernel data structures (e.g., free lists, inverse page mappings). Xok's low-level interface means that paging is handled by applications. As such, it can be done from disk, across the network, or by data regeneration. Additionally, applications can readily perform per-page transformations such as compression, verification of contents using digital signatures (to allow untrusted nodes in a network to cache pages), or encryption.

Wakeup predicates. Applications often want to sleep until a condition is true. Unfortunately, it may be difficult for an application to express this condition to the kernel. This problem is more prevalent on exokernels because the bulk of OS functionality resides in the application.

To solve this problem, Xok provides applications with the ability to inject wakeup predicates into the kernel. Wakeup predicates are boolean expressions used by applications to sleep until the state of the system satisfies some condition; they are evaluated by the kernel when an environment is about to be scheduled. The application is not scheduled if the predicate does not hold.

Predicate evaluation is efficient. Like dynamic packet filters, Xok compiles predicates on-the-fly to executable code. The significant overhead of an address space context switch is eliminated by evaluating the predicates in the exokernel and pre-translating all predicate virtual addresses to their associated physical addresses. When a virtual page referenced in a predicate is unmapped, the physical page is not marked as free until a new predicate is downloaded or until the application exits. Furthermore, the implementation of wakeup predicates is simple (fewer than 200 lines of commented code) because careful language design (no loops and easy to understand operations) allows predicates to be easily controlled.

Predicates are simple but powerful. Coupled with Xok's exposure of data structures, they have provided us with a robust wakeup facility—none of the new uses of wakeup predicates required changes to Xok. For example, to wait for a disk block to be paged in, a wakeup predicate can bind to the block's state and wake up when it changes from "in transit" to "resident." To bound the amount of time a predicate sleeps, it can compare against the system clock. The composition of multiple predicates allows atomic checking of disjoint data structures.

Access control Unlike Aegis, Xok performs access control through hierarchically-named capabilities [31]; despite the name, these capabilities more closely resemble a generalized form of UNIX user and group ID than traditional capabilities [9]. All Xok calls require explicit credentials. We believe that the combination of an exokernel interface, hierarchically-named capabilities, and explicit credentials will simplify the implementation of secure applications, as we hope to demonstrate in future work.

5.2 ExOS 1.0

ExOS is a libOS that supports most of the abstractions found in 4.4BSD. It runs many unmodified UNIX applications, including all of the applications that are needed to build the complete system (kernel, ExOS, and applications) on itself. It also runs most shells, file utilities (wc, grep, ls, vi, etc.), and many networking applications (telnetd, ftp, etc.). The most salient missing functions are full paging, process swapping, process groups, and a windowing system. There is no fundamental reason why these are not supported; we simply have not yet had the time to implement or port them. On Aegis, for instance, ExOS supported full paging to disk and over the network.

The primary goals of ExOS are simplicity and flexibility. To allow applications to override any implementation feature, we made the system entirely library based, rather than place objects such as process tables in non-customizable servers. As a result, customization of the resulting system is limited only by an application's understanding of the system interfaces and by the protection enforced by shared abstractions—any ExOS functionality can be replaced by application-specific code.

The two primary caveats of the current implementation are that the system is research, not production quality and that it uses shared global state for some abstractions. These limitations are not fundamental and we do not expect removing either caveat to have a significant impact on our results. To compensate for the effects of shared state on performance, measurements in Sections 6 and 8 include the cost of inserting system calls before all writes to shared state. This represents the overhead of invoking the kernel to check writes to shared state.

5.2.1 Implementing UNIX abstractions on Xok

To implement UNIX abstractions in a library, we partitioned most of the UNIX kernel state and made it private to each process. The remainder is shared. Most critical shared state (inode table, file system metadata, page tables, buffer cache, process table, and pipes), is protected using Xok's protections mechanisms. However, for some shared state (the process map, file descriptor table, sockets, TTYs, mount table, and system V shared memory table), ExOS uses shared memory. Using software regions, we plan to make this shared state fully protected in the near future. A limited degree of fault isolation is provided for these abstractions by mapping shared data at addresses far from the application text and data.

Processes. The *process map* maps UNIX process identifiers to Xok environment numbers using a shared table. The *process table* records the process identifiers of each process, that of its parent, the arguments with which the process was called, its run status, and the identity of its children. The table is partitioned across application-reserved memory of Xok's environment structure, which is mapped readable for all processes and writeable for only the environment's owning process. ExOS uses Xok's IPC to safely update parent and child process state. The UNIX *ps* (process status) program is implemented by reading all the entries of the process table.

UNIX provides the *fork* system call to duplicate the current process and *exec* to overlay it with another. *Exec* is implemented by creating a new address space for the new process, loading on demand the disk image of the process into the new address space, and then discarding the address space that called *exec*. Implementing fork in a library is peculiar since it requires that a process create a replica of its address space and state *while it is executing*. To make fork efficient, ExOS uses copy-on-write to lazily create separate copies of the parent's address space. ExOS scans through its page tables, which are exposed by Xok, marking all pages as copy-on-write except those data segment and stack pages that the *fork* call itself is using. These pages must be duplicated so as not to generate copy-on-write faults while running the *fork* and page fault handling code. Groups of page table entries are updated at once by batching system calls to amortize the system call overhead over many updates.

Interprocess communication. UNIX defines a variety of interprocess communication primitives: signals (software interrupts that can be sent between processes or to a process itself), pipes (producer-consumer untyped message queues), and sockets (differing from pipes in that they can be established between non-related processes, potentially executing on different machines).

Signals are layered on top of Xok IPC. Pipes are implemented using Xok's software regions, coupled with a "directed yield" to the other party when it is required to do work (i.e., if the queue is full or empty). Sockets communicating on the same machine are currently implemented using a shared buffer.

Inter-machine sockets are implemented through user-level network libraries for UDP and TCP. The network libraries are implemented using Xok's timers, upcalls, and packet rings, which allow protected buffering of received network packet,

File descriptors. File descriptors are small integers used to access many UNIX resources (e.g., files, sockets, pipes). On ExOS they name entries in a global *file descriptor table*, which is currently stored in shared memory. As in the UNIX kernel itself, ExOS accesses each table element in an object-oriented manner: each resource is associated with a table of pointers to functions implementing each operation (read, write, etc.). However, unlike UNIX, ExOS allows applications to install their own methods.

Files. Local files are accessed through C-FFS, which uses XN to protect file metadata; remote files are accessed through the Network File System protocol (NFS) [38]. Both file systems are library based. ExOS uses XN's buffer cache registry to safely share both C-FFS and NFS disk blocks.

UNIX allows different file systems to be attached to its hierarchical name space. ExOS duplicates this functionality by maintaining a currently unprotected shared mount table that maps directories from one file system to another.

5.2.2 Shared libraries

Since ExOS is implemented as a library, shared libraries are crucial. Without shared libraries, every application would contain its own copy of ExOS, wasting memory and making process creation expensive. We employ a simple but primitive scheme for shared libraries. ExOS is linked as a stand-alone executable with its base address starting at a reserved section of the application's address space. Its exported symbols are then extracted and stored in an assembly file. To resolve calls to library routines, the application links against this assembly file. During process creation the application is loaded and ExOS maps the library at its indicated address.

This organization separates the file that the libOS resides in from applications, allowing multiple applications to share the same on-disk copy and, more importantly, any cached disk blocks from this file. Code sharing reduces the size of ExOS executables to roughly that of normal UNIX applications. Unlike traditional dynamic linking, procedure calls are no more expensive than for normal code since they do not require the use of a relocation table.

6 Application Performance on Xok

This section shows that unmodified UNIX applications run as fast on Xok/ExOS as on conventional centralized operating systems. In fact, because of C-FFS, some applications run considerably faster on Xok/ExOS. We compare Xok/ExOS to both FreeBSD 2.2.2 and OpenBSD 2.1 on the same hardware. Xok uses device drivers that are derived from those of OpenBSD. ExOS also shares a large source code base with OpenBSD, including most applications and most of libc. Compared to OpenBSD and FreeBSD, ExOS has not had much time to mature; we built the system in less than two years and moved to the x86 platform only a year ago.

All experiments are performed on 200-MHz Intel Pentium Pro processors with a 256-KByte on-chip L2 cache and 64-MByte of main memory. The disk system consists of an NCR 815 SCSI controller connecting a fast SCSI chain with one or more Quantum Atlas XP32150 disk drives to the PCI bus (vs440fx PCI chip set). Reported times are the minimum time of ten trials (the standard deviations of the total run times are less than three percent).

The measurements establish two results. First, the base performance of unaltered UNIX applications linked against ExOS is comparable to OpenBSD and FreeBSD. Untrusted libOSes on an exokernel can support unchanged UNIX applications with the same performance as centralized monolithic UNIX operating systems. Second, because of ExOS's high-performance file system, some unaltered UNIX applications perform better on ExOS than on FreeBSD and OpenBSD. Applications do not need to be re-written or even modified in order to take advantage of an exokernel.

It is important to note that a sufficiently motivated kernel programmer can implement any optimization that is implemented in an extensible system. In fact, a member of our research group, Costa Sapuntzakis, has implemented a version of C-FFS within OpenBSD. Extensible systems (and we believe exokernels in particular) make these optimizations significantly easier to implement than centralized systems do. For example, porting C-FFS to OpenBSD took more effort than designing C-FFS and implementing it as a library file system. The experiments below demonstrate that by using unprivileged application-level resource management, any skilled programmer can implement useful OS optimizations. The extra layer of protection required to make this application-level management safe costs little.

Benchmark	Description (application)
Copy small file	copy the compressed archived source tree (cp)
Uncompress	uncompress the archive (gunzip)
Copy large file	copy the uncompressed archive (cp)
Unpack file	unpack archive (pax)
Copy large tree	recursively copy the created directories (cp).
Diff large tree	compute the difference between the trees (diff)
Compile	compile source code (gcc)
Delete files	delete binary files (rm)
Pack tree	archive the tree (pax)
Compress	compress the archive tree (gzip)
Delete	delete the created source tree (rm)

Table 1. The I/O-intensive workload installs a large application (the lcc compiler). The size of the compressed archive file for lcc is 1.1 MByte.

6.1 Base system performance

We test ExOS's base performance by running the I/O-intensive benchmarks from Table 1 over ExOS's library implementation of C-FFS on top of XN and comparing it to OpenBSD with a C-FFS file system. The workload in the experiments represents unmodified UNIX programs involved with installing a software package: copying a compressed archive file, uncompressing it, unpacking it (which results in a source tree), copying the resulting tree, comparing the two trees, compiling the source tree, deleting binaries, archiving the source tree, compressing the archive file, and deleting the source tree (see Table 1).

Figure 2 shows the performance of these applications over Xok/ExOS, OpenBSD/C-FFS, OpenBSD, and FreeBSD. To establish base system performance, we compare Xok/ExOS with OpenBSD/C-FSS, since they both use a C-FFS file system. The total running time for Xok/ExOS is 41 seconds and for OpenBSD/C-FFS is 51 seconds. Since ExOS and OpenBSD/C-FFS use the same type of file system, one would expect that ExOS and OpenBSD perform equally well. As can be seen in Figure 2, Xok/ExOS performance is indeed comparable to OpenBSD/C-FFS on eight of the 11 applications. On three applications (pax, cp, diff), Xok/ExOS runs considerably faster (though we do not yet have a good explanation for this).

From these measurements we conclude that, even though ExOS implements the bulk of the operating system at the application level, common software development operations on Xok/ExOS perform comparably to OpenBSD/C-FFS. They demonstrate that—at least for this common domain of applications—an exokernel's flexibility can be provided for free: even without aggressive optimizations ExOS's performance is comparable to that of mature monolithic systems. The cost of low-level multiplexing is negligible.

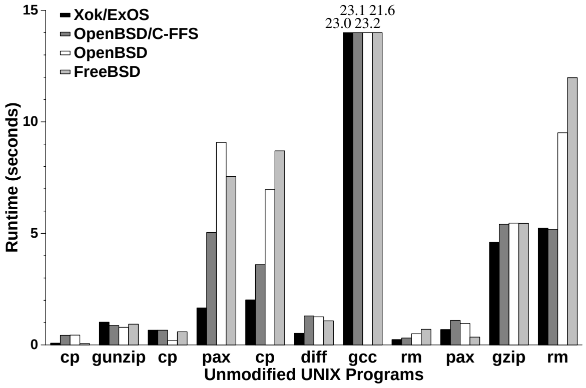

Figure 2. Performance of unmodified UNIX applications. Xok/ExOS and OpenBSD/C-FFS use a C-FFS file system while Free/OpenBSD use their native FFS file systems. Times are in seconds.

6.2 Invisible optimization using C-FFS

These comparisons concentrate on I/O intensive operations that exploit the C-FFS library file system [15]. We again use the I/O-intensive benchmarks described in Table 1, but now compare Xok/C-FFS with OpenBSD and FreeBSD. As Figure 2 shows, unaltered UNIX applications can run significantly faster on top of Xok/ExOS. Xok/ExOS completes all benchmarks in 41 seconds, 19 seconds faster than FreeBSD and OpenBSD. On eight of the eleven benchmarks Xok/ExOS performs better than Free/OpenBSD (in one case by over a factor of four). ExOS's performance improvements are due to its C-FFS file system.

We also ran the Modified Andrew Benchmark (MAB) [33]. On this benchmark, Xok/ExOS takes 11.5 seconds, OpenBSD/C-FFS takes 12.5 seconds, OpenBSD takes 14.2 seconds, and FreeBSD takes 11.5 seconds. The difference in performance on MAB is less profound than on the I/O-intensive benchmark, because MAB stresses fork, an expensive function in Xok/ExOS. ExOS's fork performance suffers because Xok does not yet allow environments to share page tables. Fork takes six milliseconds on ExOS, compared to less than one millisecond on OpenBSD.

6.3 The cost of protection

In this section, we investigate the cost of protection on Xok/ExOS. As discussed in the previous section, we have not yet completed the protected implementation of all data structures. ExOS stores some tables in writeable global shared memory, including the file descriptor table. In order for our measurements to estimate the performance of a fully protected ExOS, we inserted three system calls before every write to these shared tables. All measurements reported in Section 6 include these extra calls.

To measure the costs of all protection we ran the benchmarks presented in Figure 2 without XN or any of the extra system calls. This reduces the overall number of Xok system calls from 300,000 to 81,000, but only changes the total running time from 41.1 seconds to 39.7 seconds. Real workloads are dominated by costs other than system call overhead.

Benchmark	Shared memory	Protection	OpenBSD
Latency 1-byte	13	30	34
Latency 8-Kbyte	150	148	160

Table 2. The cost of a local-trust implementation of pipes (times in microseconds).

To investigate the cost of protection in more detail, we measure the cost of the protection mechanisms described in Section 3. We do so by comparing two implementations of pipes (see Table 2). The first implementation places all data in shared memory and performs no sanity checking. The second implementation uses software regions to protect pipe data and installs a wakeup predicate on every read (something unnecessary even with mutual distrust). The results show that even with gratuitous use of Xok's protection mechanisms, user-level pipes can still outperform OpenBSD.

7 Exploiting Extensibility in Applications

This section demonstrates some of the interesting possibilities in functionality and performance enabled by application-level resource management. We report on a binary emulator, a "zero-touch" file-copy program, and the Cheetah web server. Because XN was developed recently, the applications in this section were not measured with XN.

7.1 Fast, simple binary emulation

Xok provides facilities to efficiently reroute specific INT instructions. We have used this ability to build a binary emulator for OpenBSD applications by capturing the system calls made by emulated OpenBSD programs. This binary emulator is useful for OpenBSD programs for which we do not have source code. Although the emulator is only partially completed (it supports 90 of the approximately 155 OpenBSD system calls), initial results are promising: it has been able to execute large programs such as Mosaic.

The main interesting feature of the emulator is that it runs in the same address space as the emulated program, and consequently does not need any privilege. Measurements show that most programs on the emulator run only a few percent slower than the same programs running directly under Xok/ExOS.

A counter-intuitive result is that, because the emulator runs in the same address space as ExOS, it is possible to run emulated programs faster than on their native OS. For example, the trivial "get process id" system call takes 270 cycles on OpenBSD and 100 cycles on the emulator running on Xok/ExOS (on a 120-MHz Intel Pentium). This difference comes from the fact that the emulator replaces OpenBSD system calls with procedure calls into ExOS. ExOS can omit many expensive checks that UNIX must perform in order to guard against application errors (on an exokernel, if an application passes the wrong arguments to a libOS, only the application will be affected).

7.2 XCP: a "zero-touch" file copying program

XCP is an efficient file copy program. It exploits the low-level disk interface by removing artificial ordering constraints, by improving disk scheduling through large schedules, by eliminating data touching by the CPU, and by performing all disk operations asynchronously.

Given a list of files, XCP works as follows. First, it enumerates and sorts the disk blocks of all files and issues large, asynchronous disk reads using this schedule. (If multiple instances of XCP run concurrently, the disk driver will merge the schedules.) Second, it creates new files of the correct size, overlapping inode and disk block allocation with the disk reads. Finally, as the disk reads complete, it constructs large writes to the new disk blocks using the buffer cache entries. This strategy eliminates all copies; the file is DMAed into and out of the buffer cache by the disk controller—the CPU never touches the data.

XCP is a factor of three faster than the copy program (CP) on Xok/ExOS that uses UNIX interfaces, irrespective of whether all files are in core (because XCP does not touch the data) or on disk (because XCP issues disk schedules with a minimum number of seeks and the largest contiguous ranges of disk blocks).

The fact that the file system is an application library allows us both to have integration when appropriate and to craft new abstractions as needed. This latter ability is especially profitable for the disk both because of the high cost of disk operations and because of the demonstrated reluctance of operating systems vendors to provide useful, simple improvements to their interfaces (e.g., prefetching, asynchronous reads and writes, fine-grained disk restructuring and "sync" operations).

7.3 The Cheetah HTTP/1.0 Server

The exokernel architecture is well suited to building fast servers (e.g., for NFS servers or web servers). Server performance is crucial to client/server applications [23], and the I/O-centric nature of servers makes operating system-based optimizations profitable.

We have developed an extensible I/O library (XIO) for fast servers and a sample application that uses it, the Cheetah HTTP server. This library is designed to allow application writers to exploit domain-specific knowledge and to simplify the construction of high-performance servers by removing the need to "trick" the operating system into doing what the application requires (e.g., Harvest [7] stores cached pages in multiple directories to achieve fast name lookup).

An HTTP server's task is simple: given a client request, it finds the appropriate document and sends it. The Cheetah

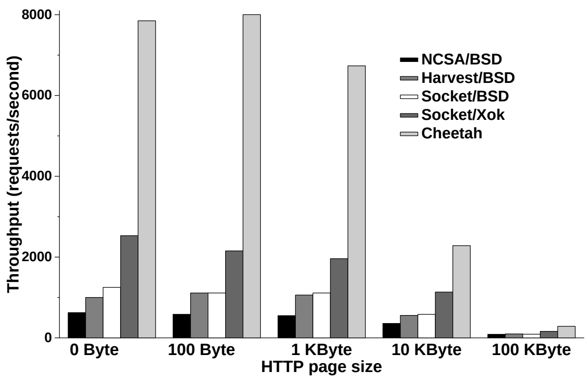

Figure 3. HTTP document throughput as a function of the document size for several HTTP/1.0 servers. **NCSA/BSD** represents the NCSA/1.4.2 server running on OpenBSD. **Harvest/BSD** represents the Harvest proxy cache running on OpenBSD. **Socket/BSD** represents our HTTP server using TCP sockets on OpenBSD. **Socket/Xok** represents our HTTP server using the TCP socket interface built on our extensible TCP/IP implementation on the Xok exokernel. **Cheetah/Xok** represents the Cheetah HTTP server, which exploits the TCP and file system implementations for speed.

Web server performs the following set of optimizations as well as others not listed here.

Merged File Cache and Retransmission Pool. Cheetah avoids all in-memory data touching (by the CPU) and the need for a distinct TCP retransmission pool by transmitting file data directly from the file cache using precomputed file checksums (which are stored with each file). Data are transmitted (and retransmitted, if necessary) to the client directly from the file cache without CPU copy operations. (Pai et al. have also used this technique [34].)

Knowledge-based Packet Merging. Cheetah exploits knowledge of its per-request state transitions to reduce the number of I/O actions it initiates. For example, it avoids sending redundant control packets by delaying ACKs on client HTTP requests, since it knows it will be able to piggy-back them on the response. This optimization is particularly valuable for small document sizes, where the reduction represents a substantial fraction (e.g., 20%) of the total number of packets.

HTML-based File Grouping. Cheetah co-locates files included in an HTML document by allocating them in disk blocks adjacent to that file when possible. When the file cache does not capture the majority of client requests, this extension can improve HTTP throughput by up to a factor of two.

Figure 3 shows HTTP request throughput as a function of the requested document size for five servers: the NCSA 1.4.2 server [32] running on OpenBSD 2.0, the Harvest cache [7] running on OpenBSD 2.0, the base socket-based server running on OpenBSD 2.0 (i.e., our HTTP server without any optimizations), the base socket-based server running on the

Xok exokernel system (i.e., our HTTP server without any optimizations with vanilla socket and file descriptor implementations layered over XIO), and the Cheetah server running on the Xok exokernel (i.e., our HTTP server with all optimizations enabled).

Figure 3 provides several important pieces of information. First, our base HTTP server performs roughly as well as the Harvest cache, which has been shown to outperform many other HTTP server implementations on general-purpose operating systems. Both outperform the NCSA server. This gives us a reasonable starting point for evaluating extensions that improve performance. Second, the default socket and file system implementations built on top of XIO perform significantly better than the OpenBSD implementations of the same interfaces (by 80–100%). The improvement comes mainly from simple (though generally valuable) extensions, such as packet merging, application-level caching of pointers to file cache blocks, and protocol control block reuse.

Third, and most importantly, Cheetah significantly outperforms the servers that use traditional interfaces. By exploiting Xok's extensibility, Cheetah gains a four times performance improvement for small documents (1 KByte and smaller), making it eight times faster than the best performance we could achieve on OpenBSD. Furthermore, the large document performance for Cheetah is limited by the available network bandwidth (three 100Mbit/s Ethernets) rather than by the server hardware. While the socket-based implementation is limited to only 16.5 MByte/s with 100% CPU utilization, Cheetah delivers over 29.3 MByte/s with the CPU idle over 30% of the time. The extensibility of ExOS's default unprivileged TCP/IP and file system implementations made it possible to achieve these performance improvements incrementally and with low complexity.

The optimizations performed by Cheetah are architecture independent. In Aegis, Cheetah obtained similar performance improvements over Ultrix web servers [24].

8 Global Performance

Xok/ExOS's decentralization of resource management allows the performance of individual applications to be improved, but Xok/ExOS must also guarantee good global performance when running multiple applications concurrently. The experiments in this section measure the situation where the exokernel architecture seems potentially weak: under substantial load where selfish applications are consuming large resources and utilizing I/O devices heavily. The results indicate that an exokernel can successfully reconcile local control with global performance.

Global performance has not been extensively studied. We use the total time to complete a set of concurrent tasks as a measure of system throughput, and the minimum and the maximum latency of individual applications as a measure of interactive performance. For simplicity we compare Xok/ExOS's performance under high load to that of FreeBSD; in these experiments, FreeBSD always performs better than OpenBSD, because of OpenBSD's small, non-unified buffer cache. While this methodology does not guarantee that an exokernel can compare to any centralized system, it does offer a useful relative metric.

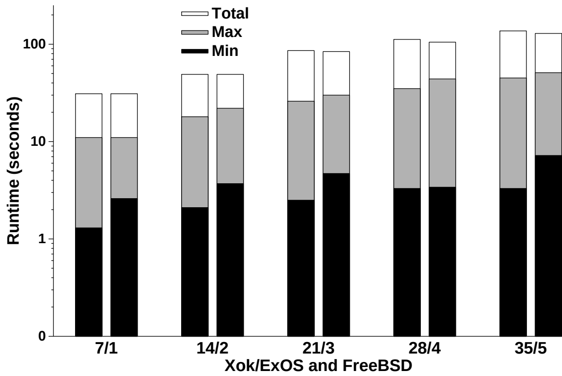

Figure 4. Measured global performance of Xok/ExOS (the first bar) and FreeBSD (the second bar), using the first application pool. Times are in seconds and on a log scale. *number/number* refers to the the total number of applications run by the script and the maximum number of jobs run concurrently. **Total** is the total running time of each experiment, **Max** is the longest runtime of any process in a given run (giving the worst latency). **Min** is the minimum.

The space of possible combinations of applications to run is large. The experiments use randomization to ensure we get a reasonable sample of this space. The inputs are a set of applications to pick from, the total number to run, and the maximum number that can be running concurrently. Each experiment maintains the number of concurrent processes at the specified maximum. The outputs are the total running time, giving throughput, and the time to run each application. Poor interactive performance will show up as a high minimum latency.

The first application pool includes a mix of I/O-intensive and CPU-intensive programs: pack archive (pax -w), search for a word in a large file (grep), compute a checksum many times over a small set of files (cksum), solve a traveling salesman problem (tsp), solve iteratively a large discrete Laplace equation using successive overrelaxation (sor), count words (wc), compile (gcc), compress (gzip), and uncompress (gunzip). For this experiment, we chose applications on which both Xok/ExOS and FreeBSD run roughly equivalently. Each application runs for at least several seconds and is run in a separate directory from the others (to avoid cooperative buffer cache reuse). The pseudo-random number generators are identical and start with the same seed, thus producing identical schedules. The applications we chose compete for the CPU, memory, and the disk.

Figure 4 shows on a log scale the results for five different experiments: seven jobs with a maximum concurrency of one job through 35 jobs with a maximum concurrency of

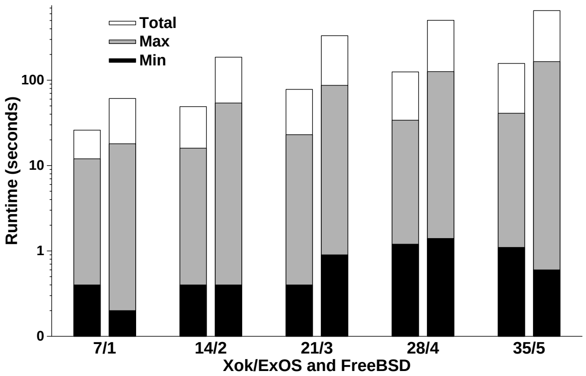

Figure 5. Measured global performance of Xok/ExOS (the first bar) and FreeBSD (the second bar), using the second application pool. Methodology and presentation are as described for Figure 4.

five jobs. The results show that an exokernel system can achieve performance roughly comparable to UNIX, despite being mostly untuned for global performance.

With a second application pool, we examine global performance when specialized applications (emulated by applications that benefit from C-FFS's performance advantages) compete with each other and non-specialized applications. This pool includes tsp and sor from above, unpack archive (pax -r) from Section 6, recursive copy (cp -r) from Section 6, and comparison (diff) of two identical 5 MB files. The pax and cp applications represent the specialized applications.

Figure 5 shows on a log scale the results for five experiments: seven jobs with a maximum concurrency of one job through 35 jobs with a maximum concurrency of 5 jobs. The results show that global performance on an exokernel system does not degrade even when some applications use resources aggressively. In fact, the relative performance difference between FreeBSD and Xok/ExOS increases with job concurrency.

The central challenge in an exokernel system is not *enforcing* a global system policy but, rather, *deriving* the information needed to decide what enforcement involves and doing so in such a way that application flexibility is minimally curtailed. Since an exokernel controls resource allocation and revocation, it has the power to enforce global policies. Quota-based schemes, for instance, can be trivially enforced using only allocation denial and revocation. Fortunately, the crudeness of successful global optimizations allows global schemes to be readily implemented by an exokernel. For example, Xok currently tracks global LRU information that applications can use when deallocating resources.

We believe that an exokernel can provide global performance *superior* to current systems. First, effective local optimization can mean there are more resources for the entire system. Second, an exokernel gives application writers machinery to orchestrate inter-application resource management, allowing them to perform domain-specific global optimizations not possible on current centralized systems (e.g., the UNIX "make" program could be modified to orchestrate the complete build process). Third, an exokernel can unify the many space-partitioned caches in current systems (e.g., the buffer cache, network buffers, etc.). Fourth, since applications can know when resources are scarce, they can make better use of resources when layering abstractions. For example, a web server that caches documents in virtual memory could stop caching documents when its cache does not fit in main memory. Future research will pursue these issues.

9 Experience

Over the past three years, we have built three exokernel systems. We distill our experience by discussing the clear advantages, the costs, and lessons learned from building exokernel systems.

9.1 Clear advantages

Exposing kernel data structures. Allowing libOSes to map kernel and hardware data structures into their address spaces is a powerful extensibility mechanism. (Of course, these structures must not contain sensitive information to which the application lacks privileges.) The benefits of mapping data structures are two-fold. First, exposed data structures can be accessed without system call overhead. More importantly, however, mapping the data structures directly allows libOSes to make use of information the exokernel did not anticipate exporting.

Because exposed data structures do not constitute a well-defined API, software that directly relies on them (e.g., the hardware abstraction layer in a libOS) may need to be recompiled or modified if the kernel changes. This can be seen as a disadvantage. On the other hand, code affected by changes in exposed data structures will typically reside in dynamically-linked libOSes, so that applications need not concern themselves with these changes. Moreover, most improvements that would require kernel modification on a traditional operating systems need only effect libOSes on exokernels. This is one of the main advantages of the exokernel, as libOSes can be modified and debugged considerably more easily than kernels. Finally, we expect most changes to the exokernel proper to be along the lines of new device drivers or hardware-oriented functionality, which expose new structures rather than modify existing ones.

In the end, some aggressive applications may not work across all versions of the exokernel, even if they are dynamically linked. This problem is nothing new, however. A number of UNIX programs such as top, gated, lsof, and netstat already make use of private kernel data structures through the kernel memory device /dev/kmem. Administrators have simply learned to reinstall these programs whenever major kernel data structures change.

The use of "wakeup predicates" has forcefully driven home the advantages of exposing kernel data structures. Frequently, we have required unusual information about the system. In all cases, this information was already provided by the kernel data structures.

The CPU interface. The combination of time slices, initiation/termination upcalls, and directed yields has proven its value repeatedly. (Subsequent to our work, others have found these primitives useful [14].) We have used the primitives for inter-process communication optimization (e.g., two applications communicating through a shared message queue can yield to each other), global gang-scheduling, and robust critical sections (see below).

Libraries are simpler than kernels. The "edit, compile, debug" cycle of applications is considerably faster than the "edit, compile, reboot, debug" cycle of kernels. A practical benefit of placing OS functionality in libraries is that the "reboot" is replaced by "relink." Accumulated over many iterations, this replacement reduces development time substantially. Additionally, the fact that the library is isolated from the rest of the system allows easy debugging of basic abstractions. Untrusted user-level servers in microkernel-based systems also have this benefit.

9.2 Costs

Exokernels are not a panacea. This subsection lists some of the costs we have encountered.

Exokernel interface design is not simple. The goal of an exokernel system is for privileged software to export interfaces that let unprivileged applications manage their own resources. At the same time, these interfaces must offer rich enough protection that libOSes can assure themselves of invariants on high-level abstractions. It generally takes several iterations to obtain a satisfactory interface, as the designer struggles to increase power and remove unnecessary functionality while still providing the necessary level of protection. Most of our major exokernel interfaces have gone through multiple designs over several years.

Information loss. Valuable information can be lost by implementing OS abstractions at application level. For instance, if virtual memory and the file system are completely at application level, the exokernel may be unable to distinguish pages used to cache disk blocks and pages used for virtual memory. Glaze, the Fugu exokernel, has the additional complication that it cannot distinguish such uses from the physical pages used for buffering messages [29]. Frequently-used information can often be derived with little effort. For example, if page tables are managed by the application, the exokernel can approximate LRU page ordering by tracking the insertion of translations into the TLB. However, at the very least, this inference requires thought.

Self-paging libOSes. Self-paging is difficult (only a few commercial operating systems page their kernel). Self-paging libOSes are even more difficult because paging can be caused by external entities (e.g., the kernel touching a paged-out buffer that a libOS provided). Careful planning is necessary to ensure that libOSes can quickly select and return a page to the exokernel, and that there is a facility to swap in processes without knowledge of their internals (otherwise virtual memory customization will be infeasible).

9.3 Lessons

Provide space for application data in kernel structures. LibOSes are often easier to develop if they can store shared state in kernel data structures. In particular, this ability can simplify the task of locating shared state and often avoids awkward (and complex) replication of indexing structures at the application level. For example, Xok lets libOSes use the software-only bits of page tables, greatly simplifying the implementation of copy on write.

Fast applications do not require good microbenchmark performance. The main benefit of an exokernel is not that it makes primitive operations efficient, but that it gives applications control over expensive operations such as I/O. It is this control that gives order of magnitude performance improvements to applications, not fast system calls. We heavily tuned Aegis to achieve excellent microbenchmark performance. Xok, on the other hand, is completely untuned. Nevertheless, applications perform well.

Inexpensive critical sections are useful for LibOSes. In traditional OSes, inexpensive critical sections can be implemented by disabling interrupts [3]. ExOS implements such critical sections by disabling software interrupts (e.g., time slice termination upcalls). Using critical sections instead of locks removes the need to communicate to manage a lock, to trust software to acquire and release locks correctly, and to use complex algorithms to reclaim a lock when a process dies while still holding it. This approach has proven to be similarly useful on the Fugu multiprocessor; it is the basis of Fugu's fast message passing.

User-level page tables are complex. If page tables are migrated to user level (as on Aegis), a concerted effort must be made to ensure that the user's TLB refill handler can run in unusual situations. The reason is not performance, but that the naming context provided by virtual memory mappings is a requirement for most useful operations. For example, in the case of downloaded code run in an interrupt handler, if the kernel is not willing to allow application code to service TLB misses then there are many situations where the code will be unable to make progress. User-level page tables made the implementation of libOSes tricky on Aegis; since the x86 has hardware page tables, this issue disappeared on Xok/ExOS.

Downloaded interrupt handlers are of questionable utility on exokernels. Aegis used downloaded code extensively in interrupt servicing [44]. The two main benefits are elimination of kernel crossings and fast upcalls to unscheduled processes, thereby reducing processing latency (e.g., of send-response style network messages). On current generation chips, however, the latency of I/O devices is large compared to the overhead of kernel crossings, making the first benefit negligible. The second does not require downloading code, only an upcall mechanism. In practice, it is the latter ability that gives us speed. Downloading interrupt handlers seems more useful on commercial operating systems with extremely high overhead for kernel crossing than on exokernel systems. It is easier to download interrupt handlers into an existing commercial OS than to turn the commercial OS into an exokernel system.

Downloaded code is powerful. Downloaded code lets the kernel leave decisions to untrusted software. We have found this delegation invaluable in many places. The main benefit of downloaded code is *not* execution speed, but rather trust and consequently power: The kernel can invoke downloaded code in cases where it cannot trust application code. For example, packet filters are downloaded code fragments used by applications to claim incoming network packets. Because they are in the kernel, the kernel can inspect them and verify that they do not steal packets intended for other applications. The alternative, asking each application if it claims a given packet, is clearly unworkable; the kernel would not know how decisions were made and could not guarantee their correctness. Another example is the use of downloaded code for metadata interpretation: since the kernel can ensure that UDFs are deterministic and do not change, it can trust their output without having to understand what they do.

10 Conclusion

This paper evaluates the exokernel architecture proposed in [11]. It shows how we built an exokernel system that separates protection from management to give untrusted software control over resource management. Our exokernel system gives significant performance advantages to aggressively-specialized applications while maintaining competitive performance on unmodified UNIX applications, even under heavily multitasked workloads. Exokernels also simplify the job of operating system development by allowing one library operating system to be developed and debugged from another one running on the same machine. The advantages of rapid operating system development extend beyond specialized niche applications. Thus, while some questions about the full implications of the exokernel architecture remain to be answered, it is a viable approach that offers many advantages over conventional systems.

Acknowledgments

Over the last three years many people have contributed to the exokernel project. In particular, we thank Deborah Wallach and Doug Wyatt for their many contributions. We also thank Josh Cates, Erik Nygren, Constantine Sapuntzakis, Yonah Schmeidler, and Elliot Waingold for porting drivers and applications to Xok/ExOS. We thank Eddie Kohler for his help with writing this paper. Finally, we thank Josh Cates, John Chapin, Matt Frank, John Guttag, Anthony Joseph, Hank Levy (our shepherd), Erik Nygren, Max Poletto, Deborah Wallach, David Wetherall, Emmett Witchel, and the anonymous referees for their careful reading of earlier versions of this paper and their valuable feedback.

References

[1] T. Anderson. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems*, pages 92–94, 1992.

[2] J. Barrera. Invocation chaining: manipulating light-weight objects across heavy-weight boundaries. In *Proc. of 4th IEEE Workshop on Workstation Operating Systems*, pages 191–193, October 1993.

[3] B.N. Bershad, D.D. Redell, and J.R. Ellis. Fast mutual exclusion for uniprocessors. In *Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 223–237, October 1992.

[4] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[5] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[6] P. Cao, E.W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[7] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A hierarchical Internet object cache. In *Proceedings of 1996 USENIX Technical Conference*, pages 153–163, January 1996.

[8] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, November 1994.

[9] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[10] D.R. Engler and M.F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1996*, pages 53–59, August 1996.

[11] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–152, October 1996.

[13] B. Ford, K. Van Maren, J. Lepreau, S. Clawson, B. Robinson, and Jeff Turner. The FLUX OS toolkit: Reusable components for OS implementation. In *Proc. of Sixth Workshop on Hot Topics in Operating Systems*, pages 14–19, May 1997.

[14] B. Ford and S.R. Susarla. CPU inheritance scheduling. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 91–106, October 1996.

[15] G. Ganger and M.F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Technical Conference*, pages 1–18, 1997.

[16] G. Ganger and Y. Patt. Metadata update performance in file systems. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 49–60, November 1994.

[17] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer*, pages 34–45, June 1974.

[18] D. Golub, R. Dean, A. Forin, and R. Rashid. UNIX as an application program. In *USENIX 1990 Summer Conference*, pages 87–95, June 1990.

[19] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970.

[20] H. Hartig, M. Hohmuth, J. Liedtke, and S. Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[21] J.H. Hartman, A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting. Scout: A communication-oriented operating system. Technical Report TR 94-20, University of Arizona, Tucson, AZ, June 1994.

[22] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *Fifth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 187–199, October 1992.

[23] D. Hitz. An NFS file server appliance. Technical Report 3001, Network Applicance Corporation, March 1995.

[24] M.F. Kaashoek, D.R. Engler, D.H. Wallach, and G. Ganger. Server operating systems. In *SIGOPS European Workshop*, pages 141–148, September 1996.

[25] B.W. Lampson. On reliable and extendable operating systems. *State of the Art Report, Infotech*, 1, 1971.

[26] B.W. Lampson and R.F. Sproull. An open operating system for a single-user machine. *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 98–105, December 1979.

[27] C.H. Lee, M.C. Chen, and R.C. Chang. HiPEC: high performance external virtual memory caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 153–164, 1994.

[28] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 237–250, December 1995.

[29] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M.F. Kaashoek. UDM: user direct messaging for general-purpose multiprocessing. Technical Memo MIT/LCS/TM-556, March 1996.

[30] C. Maeda and B.N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, 1993.

[31] D. Mazières and M.F. Kaashoek. Secure applications need flexibile operating systems. In *Proc of 6th Workshop on Hot Topics in Operating Systems*, pages 56–61, May 1997.

[32] NCSA, University of Illinois, Urbana-Champaign. NCSA HTTPd. http://hoohoo.ncsa.uiuc.edu/index.html.

[33] J.K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[34] V. Pai, P. Druschel, and W. Zwaenepoel. I/O-lite: a unified I/O buffering and caching system. Technical Report http://www.cs.rice.edu/~vivek/IO-lite.html, Rice University, 1997.

[35] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[36] D. Probert, J.L. Bruno, and M. Karzaorman. SPACE: a new approach to operating system abstraction. In *International Workshop on Object Orientation in Operating Systems*, pages 133–137, October 1991.

[37] D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell. Pilot: an operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[38] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proc. of the 1985 Summer USENIX conference*, pages 119–130, 1985.

[39] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 213–228, October 1996.

[40] C.A. Thekkath, H.M. Levy, and E.D. Lazowska. Separating data and control transfer in distributed operating systems. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 2–11, San Francisco, CA, October 1994.

[41] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 40–53, 1995.

[42] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

[43] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, November 1994.

[44] D.A. Wallach, D.R. Engler, and M.F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM '96)*, pages 40–52, August 1996.