Exokernel: an operating system architecture for application-level resource management

Dawson R. Engler, M. Frans Kaashoek and James O'Toole Jr.
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139

{engler,kaashoek,james}@lcs.mit.edu

March 24, 1995

Abstract

We describe an operating system architecture that securely multiplexes machine resources while permitting an unprecedented degree of application-specific customization of traditional operating system abstractions. By abstracting physical hardware resources, traditional operating systems have significantly limited the performance, flexibility, and functionality of applications. The exokernel architecture removes these limitations by allowing untrusted software to implement traditional operating system abstractions entirely at application-level.

We have implemented a prototype exokernel-based system that includes Aegis, an exokernel, and ExOS, an untrusted application-level operating system. Aegis defines the low-level interface to machine resources. Applications can allocate and use machine resources, efficiently handle events, and participate in resource revocation. Measurements show that most primitive Aegis operations are 10–100 times faster than Ultrix, a mature monolithic UNIX operating system. ExOS implements processes, virtual memory, and inter-process communication abstractions entirely within a library. Measurements show that ExOS's application-level virtual memory and IPC primitives are 5–50 times faster than Ultrix's primitives. These results demonstrate that the exokernel operating system design is practical and offers an excellent combination of performance and flexibility.

1 Introduction

Operating systems define the interface between applications and physical resources. Unfortunately, this interface can significantly limit the performance and implementation freedom of applications. This problem arises because the operating system abstracts the details of hardware resources to provide a more portable and more full-featured interface than is directly implemented by the hardware. The end result of such a full-featured interface is an approach to resource

management that is strongly centralized. Centralized management can conflict with application needs, limiting both performance and flexibility. We believe these problems can be solved through distributed, *application-level*, resource management. To this end, we have designed a kernel that securely multiplexes machine resources and permits traditional operation system abstractions to be implemented efficiently at application-level, so that they can easily be extended, specialized, or even replaced.

Traditionally, operating systems hide information about machine resources behind high-level *core abstractions*, choosing particular implementations of abstractions such as processes, file system storage, address spaces, inter-process communication, exception handling, etc. Core abstractions define a virtual machine on which applications execute, and their implementation cannot be replaced by untrusted applications. We believe that fixing the implementations of these traditional operating system abstractions is unacceptable because this denies applications the advantages of domain-specific optimizations. More important, it restricts the flexibility of application builders in adding new resource abstractions to the operating system because they must resort to emulating the new abstraction on top of high-level core abstractions.

Substantial evidence exists that applications can benefit greatly from having more control over how machine resources are used to implement higher-level abstractions. Appel et al. [4] reported that the high cost of general-purpose virtual memory primitives reduces the performance of persistent stores, garbage collectors, and distributed shared memory systems. Cao et al. demonstrated that application-level control over file caching can reduce the number of I/O operations by up to 80% [9]. Cheriton et al. [21] and Krueger et al. [25] showed how application-specific virtual memory policies can increase application performance. Stonebraker [43] demonstrated that inappropriate file-system implementation decisions can have a dramatic impact on the performance of databases. Thekkath et al. [45] showed that by deferring signal handling to applications the cost of exceptions can be reduced by an order of magnitude.

We have designed a new operating system architecture in which traditional operating system abstractions are imple-

mented entirely at application level by untrusted software. In this architecture, an *exokernel* securely multiplexes available hardware resources. Using the exokernel, applications can securely bind to machine resources, efficiently handle events, and participate in a resource revocation protocol. The exokernel interface is very low-level and can be implemented extremely efficiently. Library operating systems, working above the exokernel interface, implement higher-level abstractions and can define special-purpose implementations that best meet the performance and functionality goals of applications.

We have implemented a prototype exokernel-based system that includes an exokernel (Aegis) and an untrusted library operating system (ExOS). This system demonstrates several important properties of the exokernel architecture:

- Low-level secure multiplexing of hardware resources can be implemented efficiently.

- Traditional core abstractions can be implemented efficiently at application-level.

- Applications can create special-purpose implementations of core abstractions.

In practice, our implementation provides applications with greater flexibility and better performance than in a monolithic system. Aegis's low-level interface allows application-level software, such as ExOS, to manipulate resources very efficiently. Aegis's protected control transfer is three times faster than the best reported implementation [29]. Aegis's exception forwarding and control transfers are close to 100 times faster than in Ultrix 4.2, a mature monolithic system using identical hardware. Because of this efficiency, ExOS is able to implement virtual memory entirely at application level.

Aegis also permits ExOS (or other application-level software) flexibility that is not available in microkernel-based systems. Aegis's efficient protected control transfer allows applications to trade between a wide array of IPC semantics that differ in performance by a factor of 10. In contrast, microkernel systems such as Amoeba [44], Chorus [39], Mach [1], and V [13], do not allow untrusted application software to define specialized IPC primitive because virtual memory and message passing services are implemented by the kernel and trusted servers. Similarly, many other microkernel abstractions, such as page-table structures and process abstractions, are fixed. Finally, many of the hardware resources in microkernel systems, such as the network, screen, and disk, are encapsulated in heavy-weight servers that cannot be bypassed or tailored to application-specific needs.

The focus of this paper is on how the exokernel architecture can be designed and implemented securely and efficiently. The remaining sections provide a more detailed case for exokernels (Section 2), discuss the issues that arise in their design (Section 3), present the implementation and summarize performance measurements of Aegis and ExOS (Sections 4 and 5), discuss global optimizations (Section 6), summarize related work (Section 7), and report our conclusions (Section 8).

2 Motivation for Exokernels

Traditionally, operating systems have centralized resource management in a set of core abstractions that cannot be specialized, extended, or replaced. Whether provided by the kernel or by trusted user-level servers, these core abstractions are implemented by privileged software that must be used by all applications, and therefore cannot be changed by untrusted software. Typically, the core abstractions defined by the operating system include processes, file storage, address spaces, and inter-process communication.

In this section, we argue that fixing the implementation of these high-level abstractions can reduce the performance, increase the complexity, and limit the functionality of application programs. We then give an end-to-end argument for the exokernel architecture and discuss the role of application-level library operating systems.

2.1 The Cost of Core Abstractions

Application performance suffers because there is no single way to abstract physical resources or to implement a core abstraction that is best for all applications. In implementing a core abstraction, the operating system is forced to make trade-offs between support for sparse or dense address spaces, read-intensive or write-intensive workloads, etc. Any such trade-off penalizes some applications, and often the applications that suffer most are those whose behavior is the most predictable. Relational databases and garbage collectors sometimes have very predictable data access patterns, and their performance suffers when a general-purpose page replacement strategy such as LRU is imposed by the operating system.

High-level core abstractions hide information from application-level (untrusted) software. For example, most current systems do not make low-level exceptions, timer interrupts, or raw device I/O directly available to applications. Unfortunately, hiding this information makes it difficult or impossible for applications to implement their own resource management abstractions. For example, database implementations must struggle to emulate random-access record storage on top of file systems [43]. Implementing light-weight threads on top of heavy-weight processes usually requires compromises in correctness and performance because the operating system hides page faults and timer interrupts [3]. In both of these cases, the complexity of applications increases because of the difficulty of getting good performance from high-level core abstractions.

Core abstractions can limit the functionality of applications because they are the only available interface between

applications and hardware resources. Because all applications must share the core abstractions, changes to core abstractions occur rarely, if ever. This is perhaps why few good ideas from the last decade of operating systems research have been adopted into widespread use. What operating systems support scheduler activations [3], multiple protection domains within a single address space [10], efficient IPC [29], or efficient and flexible virtual memory primitives [4, 21, 25]?

2.2   Exokernels: An End-to-End Argument

The essential observation about core abstractions in traditional operating systems is that they are overly general. Traditional operating systems attempt to provide all the features needed by all applications. As previously noted by Lampson et al. [28], Anderson et al. [3] and Massalin [31], general-purpose implementations of core abstractions force applications that do not need a given feature to pay substantial overhead costs. This longstanding problem has become more important with explosive improvements in raw hardware performance and enormous growth in diversity of the application software base.

The familiar "end-to-end" argument applies as well to low-level operating system software as it does to low-level communications protocols [40]. Applications know better than operating systems what the goal of their resource management decisions should be, and should therefore be given as much control as possible over those decisions. Our proposed solution is a new operating system architecture in which traditional abstractions are implemented entirely at application level.

To provide maximum opportunity for application-level resource management, the exokernel architecture consists of a thin exokernel veneer that multiplexes physical resources securely, and library- and server-based operating systems that implement system objects and policies (see Figure 1). This structure allows the extension, specialization and even replacement of abstractions. For example, page-table structures can vary across different applications. To the best of our knowledge, no other secure operating system architecture gives applications so much useful freedom.

We expect that an exokernel structure is an effective way to address the problems listed in Section 2.1. Efficient implementation of basic abstractions at application level solves many of them, since conflicts between application needs and available abstractions can be resolved without the intervention of kernel architects. Furthermore, since the kernel only multiplexes resources, its implementation is simple. Secure multiplexing does not require complex algorithms; it mostly requires tables to keep track of ownership. A simple kernel improves reliability and ease of maintenance, consumes few resources, and enables quick adaptation to new requirements (e.g., gigabit networking). Furthermore, as is true in RISC architectures, the simplicity of exokernel operations allows them to be implemented very efficiently.
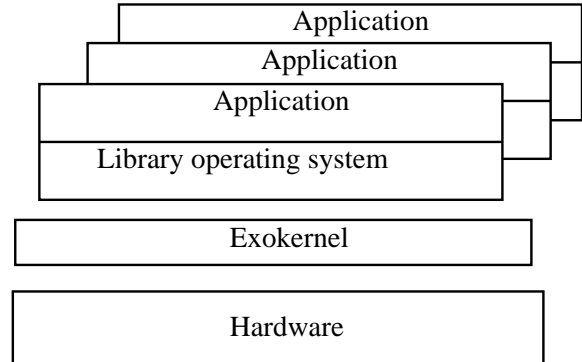

Figure 1: An example exokernel-based system consisting of a thin exokernel veneer that multiplexes physical resources and library operating systems, each linked with an application. Each library operating system implements its own system objects and policies.

2.3   Library Operating Systems

In addition to providing flexibility and efficiency, an exokernel-based system has a number of potential implementation and performance advantages. Since library operating systems need not multiplex a resource among competing applications with widely different demands, their implementation can be more specialized and simpler than corresponding kernel-level implementations. In addition, since libraries are not trusted by the exokernel, they are free to trust the application. For example, if an application passes the wrong arguments to a library, only that application will be affected. Finally, the number of kernel transitions in an exokernel system can be smaller, since most the operating system is running in the address space of the application.

The unprecedented implementation freedom available to applications using an exokernel system may create portability and compatibility problems. Software that uses an exokernel interface directly will not be portable because the interface will include hardware-specific information. However, library operating systems can use a low-level machine-dependent layer to hide hardware details and can implement industry standard (e.g., POSIX) interfaces. In short, library operating systems can provide as much portability as is desirable to applications. However, unlike in other operating systems, an application running on an exokernel can freely replace these library operating systems without needing any special privileges, which simplifies the addition and development of new standards and features not anticipated by kernel architects.

The library software that provides applications with higher-level operating system services might be considerably simplified by modular design. It is possible that object-oriented programming methods, overloading, and inheritance can provide useful operating system service imple-

mentations that can be easily specialized and extended, as in Anderson's VM++ library [25]. To control the space used by these libraries, good support for shared libraries and dynamic linking will be an essential part of a complete exokernel-based system.

Backward compatibility can be provided as in microkernel-based systems. Three viable solutions exist: (1) binary emulation of the operating system and its programs; (2) porting the operating system by implementing its hardware abstraction layer on top of the exokernel; (3) re-implementing the operating system's core abstractions on top of the exokernel. All of these approaches can be used with an exokernel-based system.

3 Exokernel Design

In order to allow applications to extend, specialize, and replace traditional operating system abstractions, the exokernel architecture is designed so that untrusted software can efficiently implement basic operating system services at application level. The exokernel design philosophy encourages *distributed control*: while traditional operating systems tend to centralize resource control in core abstractions, an exokernel strives to give resource control to applications. To enable untrusted application-level software to implement higher-level abstractions efficiently, the exokernel interface must be secure, yet permit very direct and efficient control over hardware resources.

In exporting these resources the exokernel has responsibility for three important tasks: (1) tracking ownership of resources, (2) performing access control by guarding all usage or binding points to ensure that security is not violated, and (3) revoking access to resources. In this section, we will explore the design of exokernels first in general, giving an overview of the main tasks performed by an exokernel and the principles that have guided our approach. Then we discuss in detail the central issues in exokernel design: secure multiplexing and resource revocation.

3.1 Design Principles

The central tenet of an exokernel is that the kernel should not manage resources besides that required for protection. For instance, an exokernel designer strives to safely export all privileged instructions, hardware DMA capabilities, and machine resources. The resources exported are those provided by the underlying hardware: physical memory, the CPU, disk memory, translation look-aside buffer, addressing context identifiers, and interrupt/trap events.

The exokernel must specify the details of the interface that applications use to claim, release, and use machine resources. Here are some principles that have guided our efforts to design an exokernel interface that provides applications the maximum degree of control:

- **Expose Hardware**: Permit complete and fine-grained allocation of all hardware resources and privileged hardware operations.

- **Expose Names**: Define enumerable resource namespaces that can be used to allocate specific physical resources.

- **Expose Events**: Provide a visible resource revocation protocol that allows well-behaved applications to respond to scarcity, but also controls rogue applications.

The exokernel should allow applications to allocate all resources, such as physical memory, the processor, and hardware devices. Most of these resources should be finely subdivided so that multiple applications can use particular pieces of the resource. Some resources are subdivided in time, or are not subdivided at all, typically when tracking ownership would be expensive or infeasible. For example, on a MIPS processor, the general-purpose registers are best allocated entirely to a single application at a time, because subdividing the registers seems inconvenient. However, on a SPARC processor [24] it could be useful to allocate register windows individually. The number, format, and current set of TLB mappings should be visible to and replaceable by applications, as should other "privileged" co-processor state. The exokernel must export privileged instructions to applications to enable them to implement traditional operating system abstractions such as processes and address spaces. Each exported operation can be encapsulated within a system call that checks the ownership of any resources involved.

Hardware resources should be named using physical names. Physical names are easy to implement, since the kernel does not have to perform translation. Furthermore, physical names encode useful resource attributes. For instance, in a system with direct-mapped caches, the name of the physical page (i.e., the page number) determines which pages it conflicts with. If applications can request specific physical pages they can reduce cache conflicts among the pages in their working set [38]. The hardware namespaces should be enumerable by applications so that applications can tailor their requests to the available resources.

Resource revocation should be visible to applications to support lightweight application-level resource management. For example, it allows physical names to be used easily and permits applications to respond rapidly to the loss of physical resources.

3.2 Secure Resource Multiplexing

One of the primary tasks of an exokernel is to multiplex resources *securely*, meaning that mutually distrustful applications can be given access to resources. Many resources, such as physical memory, CPU, TLBs, addressing-context identifiers, and traps, can be multiplexed using simple access control mechanisms. The exokernel can enforce security by checking access privileges each time a resource

is used. Providing low-level protection checking in the exokernel enables mutually distrustful applications to access resources directly.

However, the exokernel does not know about the ownership and access privileges for high-level objects such as files, directories, windows, and network connections. The complex semantics associated with such resources are determined by application-level software. Therefore, we use *secure binding* to control access to protected hardware resources. Secure binding decouples application-level access authorization decisions from low-level protection checking.

When an application binds to a resource, complex access control calculations may be required, but need not be performed by the exokernel. Securely binding to a resource means gaining controlled access to the resource such that later operations can be efficiently checked without recourse to high-level authorization information. For example, a file server can buffer data in memory pages and grant access to authorized applications by providing them with the capabilities for the physical pages. The exokernel would enforce the capability checking without needing any information about the file system's access control mechanisms. We will discuss how this idea applies to the secure multiplexing of memory, frame buffers, and network devices.

Multiplexing Physical Memory

Secure bindings to physical memory can be implemented using self-authenticating capabilities and address translation hardware. When an application allocates a physical memory page, the exokernel creates a secure binding for that page by recording the owner and the read and write capabilities specified by the application. The owner of a page has the power to change its capability and to deallocate it.

To ensure protection, the exokernel guards every access to a physical memory page by requiring that the capability be presented by the application requesting access. If the capability is insufficient, the request is denied. Typically, the processor contains a translation look-aside buffer (TLB), and the exokernel must check memory capabilities when an application attempts to enter a new virtual-to-physical mapping.

If the underlying hardware defines a page table interface, then the exokernel must guard the page table instead of the TLB. Although the details of how to implement secure memory bindings will vary depending on the details of the address translation hardware, the basic principle is straightforward. Privileged machine operations such as TLB loads and DMA must be guarded by the exokernel.

Using capabilities to secure resource access enables applications to grant access rights to other applications without kernel intervention. Applications can also use "well-known" capabilities to share resources easily. The overhead of capabilities is fairly small. For example, two 64-bit capabilities per 4 kilobyte page is a 0.4% space overhead.

To break a secure binding, the exokernel must change the associated capabilities and mark the resource as free. In the case of physical memory, the exokernel would flush all TLB mappings and any queued DMA requests. In practice, these operations are deferred until the resource is reallocated, so that the cost of TLB flush operations is amortized.

Multiplexing a Frame Buffer

Device multiplexing presents problems because of the almost organic nature of complex device interfaces and because assigning ownership to pieces of a device can be difficult without detailed knowledge of the device. Some devices, such as disk drives, could be partitioned using a capability-based method similar to that used for physical memory. However, the value of a centralized I/O scheduling policy may make it more desirable to assign ownership of the entire device to a single application.

When the device hardware contains a low-level protection mechanism, it can be used to implement secure binding efficiently. For example, some Silicon Graphics frame buffer hardware associates an ownership tag with each pixel. This mechanism can be used by the window manager to set up a binding between an application and a portion of the frame buffer. The application can access the frame buffer hardware directly because the hardware checks the ownership tag when I/O takes place. Similarly, the "label" feature of the Xerox Alto disk device [28] could be used by an exokernel to cheaply implement secure bindings for individual disk blocks.

Multiplexing the Network

If there is no hardware capability support that can be used to efficiently multiplex a device, secure bindings can be implemented by the exokernel. We have already seen that the exokernel can maintain software capabilities for every physical memory page and check capabilities when TLB operations are attempted. Network devices offer a greater challenge because protocol-specific knowledge is normally required to identify packet ownership.

In some cases, the network hardware may offer a uniform way to demultiplex the incoming data stream. For example, ATM cells contain a virtual circuit identifier that might uniquely identify the application that should receive the data. However, in general, protocol-specific knowledge must be used to interpret the contents of incoming messages and identify the intended recipient. Packet filters [32] can be used, with simple security precautions, to distribute incoming messages among applications without incorporating protocol-specific knowledge into the exokernel.

Sharing the network interface for outgoing messages is relatively much easier. Transmission buffers can be allocated, shared, and protected by the exokernel just as easily as physical memory pages. Applications could map these message transmission buffers into their address space as suggested by Druschel et al. [18]

3.3 Revocation

An exokernel pushes as much resource management as it can to application level. Once resources have been allocated to applications there must be a way to reclaim them. Revocation can either be *invisible* or *visible* to applications. Traditionally, operating systems have performed revocation invisibly, deallocating resources without application involvement. For example, with the exception of some external pagers [1, 39], most operating systems deallocate (and allocate) physical memory without informing applications. This form of revocation has lower latency than visible revocation since it requires no application involvement. Its disadvantage is that applications have no control over deallocation and no knowledge of whether resources are scarce.

An exokernel uses visible revocation for most resources. Even the processor is explicitly revoked at the end of a time slice; the application can react by saving only the required processor state. For example, an application could avoid saving the floating point state or other registers that are not live. However, there are situations where invisible revocation performs much better because revocations occur very frequently. Processor addressing-context identifiers are a stateless resource that may be revoked very frequently and are best handled by invisible revocation.

Revocation and Physical Naming

Although well-behaved applications are expected to give up resources when requested to do so by the exokernel, we call this interaction "revocation" because future access to the relinquished resources must be prevented by the exokernel. Revocation has interesting trade-offs because applications use physical names to refer to resources. The main constraint that the use of physical names places on the exokernel is that revocation must be revealed to the application.

The application must be notified so that it can properly manage the loss of the resource. For example, an application that relinquishes physical page "5" should update any of its page-tables that refer to this page. This is easy for an application to do when it chooses to deallocate a resource in reaction to an exokernel revocation request.

We view the revocation process as a dialogue between the exokernel and the application. Applications (or library operating systems) are responsible for organizing resource lists so that resources can be deallocated quickly. For example, an application could have a simple vector of physical pages that it owns: when the kernel indicates that a page should be deallocated, the application selects one of these pages, writes it to disk, and frees it. The revocation protocol could allow the application to make the exokernel aware of "good faith" operations such as writing a page to disk in preparation for deallocation.

The Abort Protocol

The exokernel must also be able to take resources from applications that fail to respond satisfactorily to revocation requests. The exokernel can define a second stage of the revocation protocol in which the revocation request ("please return a memory page") becomes an imperative ("return a page within 50 micro-seconds"). However, if the application fails to respond quickly, some emergency action must be taken.

We rejected the idea of simply killing any application that fails to respond quickly to revocation requests because we believe that programmers have great difficulty reasoning about hard real-time bounds. We expect that such a rule is unnecessarily strict. Instead, if an application fails to comply with the revocation protocol, the *abort protocol* defines what action the exokernel will take. The exokernel will take some resources away "by force", and will inform the application.

To record the forced loss of a resource, we use a *repossession vector*. When the exokernel takes a resource from an application, this fact is registered in the vector and the application receives a "repossession" exception so that it can update any mappings that use the resource. For resources with state, the exokernel can write the state into another memory or disk resource. In preparation, the application can pre-load the repossession vector with a list of resources that can be used for this purpose. For example, it could provide names and capabilities for disk blocks that should be used as backing store for physical memory pages.

Another complication is that the exokernel should not arbitrarily choose the resource to repossess; for example, the application uses some physical memory to store vital bootstrap information such as exception handlers and page tables. The simplest way to deal with this is to guarantee the application a small number of resources that will not be repossessed (e.g., 5–10 physical memory pages). If even those resources must be repossessed, some emergency exception that tells an application to submit itself to a "swap server" would be required.

3.4 Summary

We discussed design guidelines for exokernels. The main task of the exokernel is to securely expose machine resources to applications. The exokernel employs access control and secure bindings to achieve this goal safely. To allow effective application-level resource management, the exokernel uses physical names and visible resource revocation. An abort protocol is used to protect against uncooperative applications.

4 Aegis: an Exokernel

This section and Section 5 describe two software systems that follow the principle of exposing all hardware functionality: *Aegis*, a prototype exokernel, and *ExOS*, a prototype
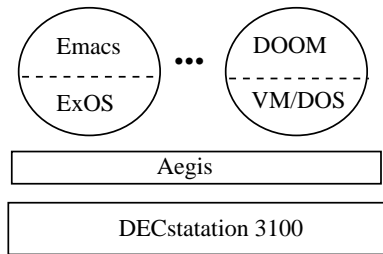
Figure 2: Aegis securely multiplexes the hardware resources that are exported by DECstations to library operating systems. ExOS, an library operating system, implements processes, virtual memory, user-level exceptions, and various interprocess abstractions.

Machine	SpecInt89	MIPS	Memory
DEC2100 (12.5 MHz)	8.7	11	12 MB
DEC3100 (16.67 MHz)	11.8	15	24 MB

Figure 3: Experimental platforms

library operating system (see Fig. 2). Another prototype exokernel, Glaze, is being built for an experimental SPARC-based shared-memory multiprocessor, supporting PhOS, a parallel operating system library. Glaze differs from Aegis in details (they implement fewer hardware resources), but they both share the same basic exokernel design. One key difference between PhOS and ExOS is that PhOS supports multiple page tables: (1) an inverted one for shared memory and (2) an hierarchical one for local memory. In this paper, we will focus on Aegis and ExOS.

The outline of this section is as follows: We first discuss the experimental environment. Then, we discuss Aegis's implementation. Included in the discussion are experiments that test the efficacy of the exokernel approach. In addition, we compare the performance of Aegis with the performance of Ultrix, a mature monolithic UNIX operating system. It is important to note that Aegis does not offer the same level of functionality as Ultrix: it has no disk support, and only rudimentary software. While we do not expect these additions to cause large fluctuations in our measurements, we emphasize that ours is *not* a widely-used, robust implementation: the entire user community is, at the moment, three people. The performance results for Aegis test our first hypothesis from the introduction: low-level multiplexing is not expensive. Our experiments indicate that the cost of flexibility in an exokernel system is either negligible or easily recouped. In fact, the low-level nature of the exokernel allows many basic operations to be implemented an order of magnitude more efficiently than in Ultrix. The next section describes ExOS and presents results supporting the other two hypotheses of this paper.

4.1 Experimental Configuration

We run experiments within the DECstation/MIPS family; the machine configurations we use are shown in Figure 3. The two machine configurations are used to get a tentative measure of the scalability of the exokernel. All times are measured using the "wall-clock." We used `clock` on the

Unix implementations and a micro-second counter on the exokernel. The exokernel's time-quantum was set at 15.625 milliseconds. All benchmarks were compiled using an identical compiler and flags: `gcc` version 2.6 with optimization flags "-O2". None of the benchmarks use floating-point instructions; we do not, therefore, save floating-point state. Both systems were run in "single-user" mode.

All experiments were measured by performing a large number of trials and dividing by this number to get the base cost of a single operation. Because such measurements do not consider cold start misses in the cache or TLB, they represent a "best case". However, Ultrix has a much larger cache and virtual memory footprint than Aegis, making this form of measurement more favorable to it than to the exokernel. Additionally, none of the experiments used loop unrolling to minimize looping overhead. Since this overhead is equivalent on both systems, it again understates the performance gains that our system obtains over Ultrix. We therefore believe that the difference in performance between the two systems is a conservative one. In closing, we note that Ultrix, despite its poor performance relative to Aegis, is *not* a poorly tuned system. It is a mature monolithic system that performs quite well in comparison to other research operating systems. For example, it performs two to three times better than Mach 3.0 in a set of I/O benchmarks [33]. Also, its virtual memory performance is approximately twice that of Mach 2.5 and three times that of Mach 3.0 [4].

A few of our benchmarks were extremely sensitive to instruction cache conflicts. In some cases the effects amounted to a factor of three performance penalty. Changing the order in which ExOS's object files were linked was sufficient to remove most conflicts. A happy side-effect of using application-level libraries is that object code rearrangement is extremely straightforward (i.e., a "makefile" edit). Furthermore, with instruction cache tools, conflicts between application and library operating system code can be removed automatically — an option not available to applications using traditional operating systems! We believe that the large impact of instruction cache conflicts is due to the fact that most Aegis operations are performed at near hardware speed; as a result, even minor conflicts are noticeable.

4.2 Aegis Overview

Table 4 lists a subset of the Aegis interface. We will discuss the implementation of most of the system calls in this section. In addition to the system calls listed in the table, Aegis supports a set of primitive operations that encapsulate privileged instructions (see Figure 5 for some examples). These primitive operations can be viewed as pseudo-

System call	Description
Yield	Yield processor
Scall	Synchronous protected control transfer
Acall	Asynchronous protected control transfer
Alloc	Allocation of resources (e.g., page)
Dealloc	Deallocation of resources
DMA	Two DMA calls (not discussed in this paper)
Clock	Two calls related to the clock

Figure 4: Subset of the Aegis system call interface.

instructions (similar to the Alpha's use of PALcode [42]). In this subsection we examine how Aegis protects time-slices and environments; other resources are protected as described in the previous section.

4.2.1 Processor Time Slices

The CPU representation is unique and deserves a brief discussion. Aegis treats the CPU as a space-multiplexed device: it is represented as a linear vector, where each element corresponds to a time-slice. Time-slices are partitioned at the clock granularity and can be allocated in a manner similar to physical memory. Scheduling is done "round robin" by cycling through the vector of time-slices. A crucial property of this representation is *position*, which encodes an ordering and an approximate upper bound on when the time-slice will be run. Position can be used to meet deadlines, and to trade off latency for throughput. For example, a long-running scientific application could allocate contiguous time-slices in order to minimize the overhead of context-switching, while an interactive application could allocate several equidistant time-slices in order to maximize responsiveness.

Timer interrupts denote the beginning and end of time-slices, and are delivered in a manner similar to exceptions: a register is saved in the "interrupt save area", the exception program counter is loaded, and Aegis jumps to user-specified interrupt handling code with interrupts re-enabled. The application's handlers are responsible for general-purpose context-switching: saving and restoring live registers, releasing locks, etc. The flexibility this framework provides allows a number of optimizations. For example, context-switching code can implement efficient uni-processor synchronization by moving ("pc-lusering") the program counter out of critical sections at context-switching time [8].

Fairness is provided by bounding the time an application takes to save its context: each subsequent timer interrupt is recorded, and when a threshold is exceeded, the environment is destroyed. In a more mature implementation the kernel will simply context-switch the application "by hand." When a time-slice is selected to run, this record is checked: if its value is non-zero, the count is decremented and the time-slice is skipped; if its value is zero, the time-slice is initiated.

This simple scheduler can support a wide range of higher-level scheduling policies. For example, a server could enforce proportional sharing (perhaps through lottery scheduling [47]) on a collection of sub-processes by allocating a number of time-slices; as each time-slice is initiated the server first determines which of its sub-process should run and then enables it by performing a `yield` system call to the chosen process. The exokernel's efficient implementation of `yield` allows high-level schedulers to perform their operations with minimal overhead.

4.2.2 Processor Environments

An Aegis processor environment is a structure that stores the information needed to deliver events to applications. All resource consumption is associated with an environment, because Aegis must deliver events associated with a resource to its designated owner.

Four kinds of events are delivered by Aegis: an exception, an interrupt, a protected control transfer, and an address translation. Processor environments contain the four contexts required to support these events:

Exception context: includes starting program counter addresses and a pointer to physical memory for saving registers, for each of several exceptions.

Interrupt context: includes interrupt vector program counter values and register-save regions for dispatching interrupts. For timer interrupts, the interrupt context specifies separate program counters for start-time-slice and end-time-slice cases, as well as status register values that control co-processor and interrupt-enable flags.

Protected Entry context: specifies legal program counter values for synchronous and asynchronous protected control transfers from other applications. Aegis allows any processor environment to transfer control into any other; access control is managed by the application itself.

Addressing context: consists of a set of guaranteed address translations that the application relies on for bootstrapping page-tables, exception handling code, exception stacks, an address space identifier, a status register, and a tag used to hash into the Aegis software TLB (see Section 4.5). To switch from one environment to another, Aegis must install these values.

These are the event-handling contexts required to define a process. Each context depends on the others for validity: for example, an addressing context does not make sense without an exception context, since it does not define any action to take when an exception or interrupt occurs.

Primitive operations	Description
TLBwr	Insert mapping into TLB
FPUmod	Enable/disable FPU
CIDswitch	Install context identifier
TLBvadelete	Delete virtual address from TLB

Figure 5: A sample of Aegis's primitive operations.

Machine	OS	procedure call	syscall (getpid)
DEC2100	Ultrix4.2	.57	32.2
DEC2100	Aegis	.56	3.2 / 4.7
DEC3100	Ultrix4.2	.42	33.7
DEC3100	Aegis	.42	2.9 /3.5

Figure 6: Null procedure and system call. Aegis has two paths: (1) with stack and (2) without stack. Times are in micro-seconds

Machine	OS	unalign	overflow	coproc	prot
DEC2100	Ultrix	n/a	272	n/a	294.
DEC2100	Aegis	2.8	2.8	2.8	3.0
DEC3100	Ultrix	n/a	200.	n/a	242.
DEC3100	Aegis	2.1	2.1	2.1	2.3

Figure 7: `Trap` benchmarks; times are in micro-seconds

4.3 Basic Costs

The base cost for null procedure and system calls are shown in Figure 6. The null procedure call is presented as a sanity check: given its minimal operating system requirements, it should be (and is) the same on both operating systems. It obliquely shows that Aegis' scheduling flexibility does not add overhead to base operations.

Aegis has two system call paths: one for system calls that do not require a stack and another for those that do. With the exception of protected control transfers, which are special-cased for efficiency, all Aegis system calls are vectored along one of these paths. Ultrix's `getpid` is approximately an order of magnitude slower than Aegis' slowest system call path — this suggests that the base cost of demultiplexing system calls is noticeably higher in Ultrix. Part of the reason Ultrix is so much less efficient on this basic operation is that it performs a more expensive demultiplexing operation. For example, on the MIPS, kernel TLB faults are vectored through the same fault handler as system calls. Therefore, Ultrix must take great care not to disturb any registers that will be required to "patch up" an interrupted TLB miss. Because Aegis does not map its data structures (and has no page tables) it can avoid such intricacies. We expect that this will be the common case with all exokernels, since they should be quite small and therefore not require paging of kernel text and code.

4.4 Exceptions

Aegis forwards to applications all hardware exceptions save for system calls and interrupts, using techniques similar to those described in Thekkath et al. [45]. To forward an exception, Aegis performs the following actions:

1. It saves three scratch registers into an agreed-upon "save area". (To avoid TLB exceptions, Aegis does this operation using physical addresses.)

2. It loads the exception program counter, the last virtual address that failed to have a valid translation, and the cause of the exception.

3. It uses the exception cause to perform an indirect jump to an application-specified program counter value, where execution resumes with the appropriate permissions set (e.g., in user-mode with interrupts re-enabled).

After processing the exception, applications can immediately resume execution without entering the kernel. Ensuring that applications can return from their own exceptions (without kernel intervention) requires that all exception state be available for user reconstruction. The means that all registers that are saved must be in user-accessible memory locations, etc.

Fast exceptions enable a number of intriguing applications [4, 45]. For example, efficient page-protection traps can be used by applications such as distributed shared memory systems, persistent object stores and garbage collectors [4, 45]. In general, most of these operations could be done by inserting explicit checks in code (e.g., hardware page-protection can be emulated by checking every load and store). The obvious advantage to using exceptions instead of explicit checks is efficiency. A more subtle advantage is that the use of explicit checks requires compiler support. Writing a well-tuned, *correct* compiler that is portable and generates efficient code is a difficult problem; eliminating this requirement aids the efficient, simple implementation of many operations.

Currently, Aegis dispatches exceptions in 18 instructions. The low-level nature of Aegis allows an extremely efficient implementation: exception forwarding requires almost *four times* fewer instructions than the most highly-tuned implementation in the literature [45]. Part of the reason for this improvement is that Aegis does not used mapped data structures, and so does not have to carefully separate out kernel TLB misses from the more general class of exceptions in its exception demultiplexing routine.

We test Aegis' overhead on exceptions for unaligned pointer accesses (`unalign`), arithmetic overflow (`overflow`), attempted use of the floating point coprocessor when it is disabled (`coproc`) and access to protected pages (`prot`). The times for `unalign` are not available under Ultrix since the kernel attempts to "fix up" the unaligned access and writes an error message to standard error. Additionally, Ultrix does not allow applications to disable co-processors, and hence cannot utilize the `coproc` exception. Times are given in Figure 7. Careful tuning of the exception path (aided by the minimal kernel functionality Aegis provides) allows all traps to be dispatched approximately *two orders of magnitude* faster than Ultrix.

4.5 Address Translations

We look at two problems in supporting application-level virtual memory (AVM): bootstrapping and efficiency.

An exokernel must provide support for bootstrapping the virtual naming system (i.e., supporting translation exceptions on both application page-tables and exception code). Aegis provides a simple bootstrapping mechanism through the use of a small number of *guaranteed mappings*. An application's virtual address space is partitioned into two segments. The first segment holds normal application data and code. The second segment is used to hold exception handling code, page-tables, etc. The exokernel allows mappings in the second segment to be "pinned" through guaranteed mappings. A miss on a guaranteed mapping will be handled automatically by Aegis. This frees the application from dealing with the intricacies of boot-strapping the TLB and exception handlers that can take TLB misses.

On a TLB miss, the following actions occur:

1. Aegis checks which segment the virtual address resides in. If it is in the standard user segment, the exception is forwarded directly to the application. If it is in the second region, Aegis first checks to see if it is a guaranteed mapping: if so, it installs the TLB entry and continues, otherwise it forwards it to the application.

2. The application looks up the virtual address in its page-table structure, and if the access is not allowed raises the appropriate exception (e.g., "segmentation fault"). If the mapping is valid, the application constructs the appropriate TLB entry and its associated capability and invokes the appropriate exokernel system routine.

3. Aegis checks that the given capability corresponds to the access rights requested by the application. If so, the mapping is installed in the TLB; control is then returned to the application. Otherwise an error is returned.

4. The application performs cleanup and resumes execution.

The obvious challenge in supporting AVM is making it *fast*. The primary bottleneck that must be overcome is the cost of TLB refills. We do this by overlaying the hardware TLB with a large software TLB (STLB) to absorb capacity misses [5, 23]. On a TLB miss, Aegis first checks to see whether the required mapping is in the STLB; if so, Aegis installs it and resumes execution. Otherwise, the miss is forwarded to the application. Aegis currently uses a unified STLB. To improve hashing coverage and to decrease the number of TLB flushes that occur when context identifiers are recycled, each process is associated with an 11 bit tag field: this field is constant over the environment's life-time and is recycled infrequently. This tag is used to compute the hash function (by xoring it with virtual addresses during lookup): to decrease the likelihood of "worst-case" hashing collisions, the tag is selected randomly from a collection of $2^{11} - 1$ tags. The STLB contains 4096 entries of 8 bytes each; it is a direct-mapped, resides in unmapped physical memory, and on an STLB "hit", replaces the desired mapping in 18 instructions. By looking at the base system call

Machine	OS	matrix
DEC2100	Ultrix4.2	7.1
DEC2100	Aegis	7.0
DEC3100	Ultrix4.2	5.2
DEC3100	Aegis	5.2

Figure 8: 150x150 matrix multiplication (time in seconds)

cost presented in Figure 6 we can see that replacing a TLB mapping from the STLB is 2-3 microseconds (approximately a factor of two) less expensive than doing so from application level using an upcall and a system call. Compared to a single-level page table (e.g., as supported by Mach), the STLB requires an additional load in order to check the virtual address space tag. However, this load resides in the same cache line as the mapping itself, and so does not add additional cache miss overhead. To avoid the worst-case behavior of a direct mapped STLB, we will likely move to a two-way set-associative structure as the implementation matures (as is used in the Rialto [17] and PA-RISC operating systems [23]).

As dictated by the exokernel principle of exposing kernel book-keeping structures, the STLB is mapped using a well-known capability, allowing applications to efficiently probe for entries, etc.

The overhead of application-level memory is measured by performing a 150 by 150 matrix multiplication. Because this naive version of matrix multiply does not use any of the special abilities of ExOS or Aegis (e.g., page-coloring to reduce cache conflicts), we expect that it will perform equivalently on both operating systems. The times in Figure 8 give a tentative indication that application-level virtual memory does not add a noticeable overhead to operations that have large virtual memory footprints. Of course, this is hardly a conclusive proof; see Section 5.2 for a discussion of the ExOS virtual memory system.

4.6   Protected Control Transfers

Aegis provides a *protected control transfer* mechanism as a substrate for implementing efficient IPC mechanisms [6, 22, 29]. Operationally, a protected control transfer changes the program counter to an agreed-upon value in the callee, donates the current time-slice to the callee's processor environment, and installs required elements of the callee's processor context (addressing-context identifier, address space tag, and processor status word).

Aegis provides two forms of protected control transfers: *synchronous* and *asynchronous*. The difference between the two is what happens to the processor time slice. Asynchronous calls donate the remainder of the current time slice to the callee. Synchronous calls tranfers the current time slice period as well as all future instantiations of it: the callee can then return the time-slice via a synchronous control transfer call back into the original caller. Both forms

OS	Machine	Transfer
Aegis	DEC2100/ 12.5MHz	2.89
L3	486/50MHz (normalized)	9.1
Aegis	DEC3100/ 16.67MHz	2.2
L3	486/50MHz (normalized)	6.67

Figure 9: Protected control transfer overhead; times are in micro-seconds

of control transfer guarantee two important properties: (1) to applications, a protected control transfer is atomic, and (2) Aegis will not overwrite any application-visible register, allowing the large register set of modern processors to be used as a temporary message buffer [12].

Currently, our synchronous protected control transfer operation costs 30 instructions. Roughly ten of these instructions are required in order to distinguish the system call "exception" from other hardware exceptions on the MIPS architecture. The remaining twenty instructions could benefit from additional optimizations. Because Aegis implements the minimum required for any control transfer mechanism, applications can introduce additional protection checks only if required. For example, control transfers between clients and trusted servers can be optimized by allowing the server to save and restore only the registers it uses, rather than requiring that the client save and restore the entire register state on every call. L3 appears to provide similar IPC semantics [29].

We measure the "bare-bones" overhead of our protected control transfer mechanism in Figure 9. Times are given in micro-seconds, and were derived by dividing the time to perform a call and reply by 2 (i.e., we measure the time to perform a uni-directional control transfer). We measured a trusted control transfer: only the callee saves and restores the registers it uses. These measurements also include the overhead cost of incrementing a counter and performing a branch, due to our measurement code. The performance of Aegis shown in Figure 9 is one to two orders of magnitude faster than any similar operation available under Ultrix (in fact, they are an order of magnitude more efficient than `getpid`!).

We attempt a crude comparison of our protected control transfer operation to the L3 RPC mechanism. The L3 implementation is the fastest published result [29], but it runs on an Intel 486. For Figure 9, we scaled the published L3 results based on the MIPS rating of our DECstation. Aegis's trusted control transfer mechanism performs *3 times* faster than L3's trusted RPC mechanism.

We have not tuned the Aegis protected control transfer implementation aggressively; architectural characteristics of the MIPS are one of the main determinants of our better performance relative to L3. For example, L3 pays a heavy penalty to enter and leave the kernel (71 and 36 cycles, respectively). While our base cost is not so high, much of the Aegis code does deal with required operations: demultiplexing the system call exception and setting the status, co-processor and address tag registers.

5 ExOS: an extensible OS

The most unusual aspect of ExOS is that it manages fundamental operating system abstractions (e.g., virtual memory and process mechanisms) *at application-level*, completely within the address space of the application that is using it. To the best of our knowledge ExOS and the Cache Kernel [11] are the first general-purpose library operating systems implemented in a multiprogramming environment. The Cache Kernel, however, supports library operating systems primarily for kernel simplification instead of for performance and extensibility. The goal of this sections is to demonstrate that (1) basic system abstractions can be implemented at application level in a direct manner and (2) specialization and extensibility of these abstractions can result in substantial performance improvements. Due to space constraints we focus on IPC and virtual memory.

5.1 Fast IPC Abstractions

Fast inter-process communication is crucial for building efficient and decoupled systems [6, 22, 29]. As described in Section 4, the Aegis protected control transfer mechanism is an efficient substrate for implementing fast IPC mechanisms. We measure the efficiency of IPC primitives that are constructed in ExOS on top of the Aegis primitive.

pipe: measures the time needed to send a word-sized message from one process to another using pipes. It was measured by "ping-ponging" a counter between two processes. The Ultrix `pipe` implementation uses the standard UNIX pipe implementation. The ExOS `pipe` implementation uses a shared-memory circular buffer. Writes to full buffers and reads from empty ones cause the current time slice to be yielded by the current process to the reader or writer of the buffer, respectively. The `pipe` implementation is an application-level library; the only kernel primitives used are the `yield` system call and those primitives required to construct the application-level virtual memory. We use two `pipe` implementations: the first is a naive implementation, while the second exploits the fact that this library exists in application space by simply inlining the read and write calls. ExOS' unoptimized `pipe` implementation is an order of magnitude more efficient than the equivalent operation under Ultrix. Much of this performance is due to the efficient implementation of `yield` in Aegis.

shmem: this experiment measures the time for two processes to "ping-pong" using a shared counter. The exokernel implementation uses Aegis' `yield` system call to yield the current time-slice between partners. Because Ultrix does not provide a yield-like primitive, acceptable efficiency can only

Machine	OS	pipe	pipe-opt	shmem	lrpc	tlrpc
DEC2100	Ultrix4.2	334	n/a	334	680.	n/a
DEC2100	Aegis	30.9	24.8	12.4	13.9	8.6
DEC3100	Ultrix4.2	231	n/a	231	457.	n/a
DEC3100	Aegis	22.6	18.6	9.3	10.4	6.4

Figure 10: IPC benchmarks; times are in micro-seconds

be achieved by using pipes to emulate the required functionality. As in the other IPC tests, the difference between ExOS and Ultrix is large: in this test ExOS is almost thirty times faster than Ultrix.

lrpc: this experiment measures the time to RPC into another address space, increment a counter and return its value. ExOS's LRPC is built on top of the exokernel's protected control transfer mechanism. There are two implementations: `tlrpc` and `lrpc`. `tlrpc` only saves and restores the stack pointer: the called processor environment is a trusted server that will restore any registers that it uses. `lrpc` saves all general-purpose callee-saved registers. Ultrix does not have an RPC mechanism; we emulated RPC functionality through a server process that waited on a well known pipe: a client sends an index, the server calls the appropriate function, and returns the result through the pipe.

Both implementations assume that only a single function is of interest (i.e., neither uses the RPC number to index into a table, etc.) and do not check permissions. Both implementations are also single-threaded. ExOS's untrusted `lrpc` ranges between 44 and 49 times faster than Ultrix, while the trusted version ranges between 71 and 79 times faster: almost a two order of magnitude differential. The most important reason for this difference is the efficiency of the control transfer mechanism.

In summary, Aegis' efficient protected control transfer and yield mechanisms allow very efficient IPC primitives to be constructed at application-level. Furthermore, doing so is profitable: exploiting both application-specific requirements (e.g., RPC between clients and trusted servers) and the characteristics of application-level (e.g., simple inlining) gives marked performance improvements.

5.2 Application-level Virtual Memory

ExOS provides a rudimentary virtual memory system (its size is approximately 1000 lines of heavily commented code). Its two main limitations are that it does not handle swapping and that page-tables are implemented as a linear vector (address translations are looked up in this structure using binary search). Barring these two implementation constraints, its interface is richer than other virtual memory systems we know of: it provides flexible support for aliasing, sharing, disabling and enabling of caching on a per-page basis, specific page-allocation, DMA, etc.

We compare Aegis and ExOS to Ultrix on seven virtual memory experiments, based on those used by Appel and Li [4]:

dirty: Measures the time to query whether a page is "dirty" or not. Since it does not require examination of the TLB, this measurement is used to test the base cost of looking up a virtual address in ExOS's page-table structure. This operation is not provided by Ultrix.

(un)prot1: Measures the time required to change the page protection of a single page.

prot100: Measures the time required to "read-protect" 100 pages.

unprot100: Measures the time required to remove read-protections on 100 pages.

trap: Time to take a page-protection trap.

appel1: Time to access a random protected page and, in the fault-handler, protect some other page and unprotect the faulting page (this benchmark is "prot1+trap+unprot" in Appel et al. [4]).

appel2: Time to protect 100 pages, then access each page in a random sequence and, in the fault-handler, unprotect the faulting page (this benchmark is "protN+trap+unprot" in Appel et al. [4]). Note that `appel2` requires less time than `appel1` since `appel1` must both unprotect and protect different pages in the fault handler.

`dirty` measures the average time to parse the page-table for a random entry. If we compare the time required for `dirty` to the time required to perform `(un)prot1`, we see that over half the time spent in `(un)prot1` is due to the overhead of parsing the page-table. This overhead can be directly eliminated through the use of a data structure more tuned to efficient lookup (e.g., a hash-table). Even with this penalty, our system performs these operations close to two times faster than Ultrix. The likely reason for this difference is that, as shown in Figure 6, Aegis dispatches system calls an order of magnitude more efficiently than Ultrix.

In general, our exokernel-based system performs well on this set of benchmarks. The sole exceptions are `prot100` and `unprot100`. Ultrix is extremely efficient in protecting and unprotecting contiguous ranges of virtual addresses: it performs 20% to 60% more efficiently than Aegis in these operations. One reason for this difference is the immaturity of our implementation; another is that changing page

Machine	OS	dirty	(un)prot1	prot100	unprot100	trap	appel1	appel2
DEC2100	Ultrix4.2	n/a	51.6	175.	175.	297.	438.	392.
DEC2100	Aegis	17.5	32.5	213.	275.	13.9	74.4	45.9
DEC3100	Ultrix4.2	n/a	47.8	140.	140.	240.	370.	325.
DEC3100	Aegis	13.1	24.4	156.	206.	10.1	55.	34.

Figure 11: Virtual memory benchmarks; times are in micro-seconds

protections in ExOS requires access to two data structures (Aegis' STLB and ExOS's page-table). We anticipate that these times will improve as we tune the system. However, even with poor performance on these two operations, the benchmark that uses this operation (`appel2`) is close to an order of magnitude more efficient on ExOS than on Ultrix.

`trap` is another area where the exokernel system performs extremely well (i.e., 21 to 24 times faster than Ultrix). This performance differential is achieved even though the `trap` benchmark on Aegis is implemented with Unix signal-like semantics: for example, all caller-saved registers are saved. If these semantics were violated by ExOS, the performance difference would become even larger.

Finally, the higher-level benchmarks, `appel1` and `appel2`, also show impressive speedup: up to an order of magnitude in some cases and never less than a factor of five.

These speedups were achieved because the virtual memory management was performed at application-level. Application-level virtual memory support might be expected to add a large overhead to basic memory operations because of the protected nature of the exokernel interface and because of more frequent user/kernel crossings. These benchmarks show that this is not the case. In fact, we can expect further improvements in performance from more sophisticated page-table structures and hand-coded assembly language for some operations. The use of a high-level language (C) currently wastes time saving and restoring registers when handling exceptions.

5.3 Summary

We have shown how inter-process communication and virtual memory can be implemented efficiently and directly at application level. Frequently, the performance differential between ExOS and Ultrix is more than an order of magnitude. We have shown that specialization can provide significant performance improvements: for example, trusted LRPC is close to a factor of two faster than its untrusted counter-part.

6 Discussion

The focus of this paper has been on how the exokernel architecture can be designed and implemented securely and efficiently. In this section we touch upon how an exokernel architecture can deal with policy conflicts between competing applications and how global system optimizations can be realized. The impact of distributed resource management on both issues is determined by whether a policy requires information, or *feedback*, from the system. If it does not, then an exokernel implementation of the policy is no more challenging than in a traditional operating system. For example, proportional sharing does not require detailed feedback from the system. Therefore, enforcing proportional sharing of resources in an exokernel can be implemented in a manner analogous to on a traditional operating system: through the exokernel's control over allocation and revocation. As shown in the last sections, allocation and revocation are inexpensive operations, so we expect that exokernel systems behave as well under the high load as traditional operating systems.

Most modern operating systems expect that applications do not lie about the resources they need. Systems like UNIX will give applications all the resources they request until they reach their quota, even though some application may have no use for the resources it requests. Under this assumption, policies that require feedback can easily be realized in an exokernel architecture. For example, working sets can be approximated by monitoring TLB insertions and DMA operations. Applications can use this information to decide whether giving up a page will improve overall system performance, and do so if another application has more need for physical memory. As in traditional operating systems, indiscriminate resource requests can be discouraged by suspending or swapping an application that requests a scarce resource.

As another example, consider disk arm latency, which is the crucial bottleneck in a disk system. An effective global policy attempts to ensure that the disk arm does not need to move frequently. This can be done by reordering reads and writes, file-caching, and allocating (or migrating) frequently accessed blocks along a narrow band (which lowers seek time to requested blocks). Disk operation reordering can be done in an exokernel system: for security the exokernel controls all reads and writes from the disk, and so can trivially reorder any that are performed. The difference in file-caching under an exokernel system is that the applications manage these caches: whether this is done through proxy servers or directly is of little concern. Application-controlled file caching, as explored by Cao et al. [9], can be directly used in an exokernel architecture.

The exokernel protects and guards applications from each other, but expects, for example, that an application does not allocate all of its quota of physical memory, if it only needs a couple of pages. The problem of guarding against such malicious applications (or badly-written applications) is a hard one, since it is difficult to distinguish between an application that needs many resources and one that is abusing the available resources. Like in many other systems, the exokernel architecture as described in this paper relies on social mechanisms, such as users who refuse to run the application, to deal with malicious applications. We think this degree of trust is acceptable for many computer systems; it is acceptable in most UNIX systems and it also seems to work in the Internet. Even lower levels of trust and fault isolation are accepted in single-user operating systems such as DOS and MacOS. An interesting research question is whether distributed control can be extended to an environment with malicious applications; we plan to investigate this question in future research.

7 Related work

Many early OS papers discussed the need for extendible, flexible kernels [27, 37, 48]. Lampson's description of CAL-TSS [27] and Brinch Hansen's microkernel paper [19] are two classic rationales. Hydra was the most ambitious system to have the separation of kernel policy and mechanism as one of its central tenets [48]. Modern revisitations of microkernels have also argued for kernel extensibility [1, 15, 36, 39, 44].

The most important difference between our work and previous approaches is the explicit view that the kernel should not provide high-level core abstractions. In other systems, the effective operating system interface is much higher-level (e.g., page-tables are implemented by the kernel).

Current extensible OS projects include Scout [20], Bridge [30], and Vino [41]. Some of the techniques used in these systems, such as type-safe languages [7, 32, 37] and software fault-isolation [16, 46], are also applicable to exokernels. These systems are just beginning to be constructed, so it is difficult to determine their relationship to exokernels in general and Aegis in particular.

Another current extensible OS project, the SPIN project, investigates adaptable kernels that allow applications to make policy decisions [7]. The SPIN system encapsulates policies in *spindles* that can be dynamically loaded into the kernel. To ensure safety, spindles will be written in a pointer-safe language and will be translated by a trusted compiler. We view the SPIN project as complementary to the exokernel design and hope to use their results to optimize application-level library operating systems.

The interface provided by the VM/370 OS [14] is very similar to what would be provided by our ideal OS: namely, the raw hardware. However, the important difference is that VM/370 provides this interface by *virtualizing* the entire base-machine. Since this machine can be quite complicated and expensive to emulate faithfully, virtualization can result in a complex and inefficient OS. In contrast, our approach *exports* hardware resources rather than emulating them, allowing an efficient and fast implementation. Furthermore, the central tenet of the virtual machine movement (and VM/370 in particular) is that an application should not be able to detect that it is not executing on the native hardware. Supporting this illusion precludes application resources management: Because the application is not supposed to see VM/370 it is unable to communicate with it about issues such as explicit allocation, revocation, naming and sharing (sharing is particularly difficult across virtual machines [26]).

The four approaches we view as most similar to the exokernel philosophy are the SPACE kernel [35], the open operating system [28], Anderson's argument for application-specific operating systems [2] and the Cache Kernel [11].

SPACE is a "submicro-kernel" that provides only low-level kernel abstractions defined by the trap and architecture interface [35]. Its close coupling to the architecture makes it similar in many ways to an exokernel, but we have not been able to make detailed comparisons because its design methodology and performance are not yet published.

The open operating system for a single-user machine [28] is motived by a rationale similar to that of the exokernel. However, the approach taken to extensibility is very different because it is designed for a single-user machine. Therefore, protection is not an issue in the open operating system, whereas secure multiplexing is the main task of an exokernel. In addition, the exokernel attempts to define no core abstractions, while in the open operating systems the file system and communications are standardized.

Anderson [2] made a clear argument for application-specific library operating systems and proposed that the kernel concentrate solely on the adjudication of hardware resources. The exokernel design addresses how to provide secure multiplexing of physical resources in such a system, and moves the kernel interface to a lower level of abstraction. In addition, Aegis and ExOS demonstrate that low-level secure multiplexing and library operating systems can offer excellent performance.

Like Aegis, the Cache Kernel [11] provides a low-level kernel that can support multiple application-level operating systems. The difference between the Cache Kernel and Aegis is mainly one of high-level philosophy: the Cache Kernel focuses primarily on reliability, rather than securely exporting hardware resources to applications. For example, the Cache Kernel attempts to eliminate all dynamic memory allocation (similar to Popek and Klines' Data Secure Unix [34]). Unsurprisingly, this single constraint lowers the kernel interface as compared to traditional operating systems. However, the deemphasis on application flexibility and extensibility is telling; the Cache Kernel is biased towards a server-based system structure (for example, it supports only 16 "application-level" kernels concurrently). In spite

of these differences, we believe that with several straightforward changes the Cache Kernel would fit within our definition of an exokernel.

8 Conclusion

We have argued that the benefits of distributed, application-specific resource management are compelling enough that the entire operating system structure should be organized to maximize it. We have defined a new OS architecture, the exokernel, to accomplish this goal and have provided a set of principles to guide exokernel design. We presented and tested two prototype systems built on our ideas: Aegis, a prototype exokernel, and ExOS, a prototype library operating system. Two interesting features of these systems are that the whole of virtual memory management occurs at application-level and that basic operations can be performed one to two orders of magnitude faster than in a mature monolithic structure.

Acknowledgments

We like to thank Bob Gruber, Sandeep Gupta, Wilson Hsieh, Butler Lampson, Ulana Legedza, Massimiliano Poletto, Raymie Stata, and Debby Wallach for insightful discussions and careful reading of earlier versions of this paper.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. *Proc. Summer 1986 USENIX Conference*, pages 93–112, July 1986.

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

[3] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. Thirteenth Symposium on Operating System Principles*, pages 95–109, October 1991.

[4] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on ASPLOS*, pages 96–107, Santa Clara, CA, April 1991.

[5] K. Bala, M.F. Kaashoek, and W.E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First Symposium on OSDI*, pages 243–253, November 1994.

[6] B. N. Bershad. High performance cross-address space communication. Technical Report 90-06-02 (PhD Thesis), University of Washington, June 1990.

[7] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN - an extensible microkernel for application-specific operating system services. TR 94-03-03, Univ. of Washington, February 1994.

[8] 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.

[9] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First Symposium on OSDI*, pages 165–178, November 1994.

[10] Jeffrey S. Chase, Henry M. Levy, Michel Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report TR 92-03-02, University of Washington, 1992.

[11] 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*, November 1994.

[12] D. R. Cheriton. An experiment using registers for fast message-based interprocess communication. *Operating Systems Review*, 18:12–20, [10] 1984.

[13] D. R. Cheriton. The v kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[14] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM J. Research and Development*, 25(5):483–490, September 1981.

[15] H. Custer. *Inside Windows/NT*. Microsoft Press, Redmond, WA, 1993.

[16] P. Deutsch and C.A. Grant. A flexible measurement tool for software systems. *Information Processing 71*, 1971.

[17] Richard Draves. Private Communication, December 1994.

[18] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM '94*, pages 2–13, 1994.

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

[20] J.H. Hartman, A.B. Montz, David 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.

[21] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on ASPLOS*, pages 187–199, October 1992.

[22] W.C. Hsieh, M.F. Kaashoek, and W.E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Fourth Workshop on Workstation Operating Systems*, pages 186–190, October 1993.

[23] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[24] SPARC International. *The SPARC Architecture Manual Verson 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[25] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for development of application-specific virtual memory management. In *Proceedings of OOPSLA*, pages 48–64, October 1993.

[26] B. W. Lampson. Hints for computer system design. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 33–48, December 1983.

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

[28] 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, 1979.

[29] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–188, 1993.

[30] Steven Lucco. High-performance microkernel systems (abstract). In *Proc. of the first Symp. on OSDI*, November 1994.

[31] H. Massalin. *Synthesis: an efficient implementation of fundamental operating system services*. PhD thesis, Columbia University, 1992.

[32] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th SOSP*, pages 39–51, Austin, TX, November 1987.

[33] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed TLBs. *20th Annual International Symposium on Computer Architecture*, pages 27–38, 1993.

[34] G.J. Popek et al. UCLA data secure UNIX. In *Proc. of the 1979 National Computer Conference*, pages 355–364, 1979.

[35] D. Probert, J.L. Bruno, and M. Karzaorman. SPACE: A new approach to operating system abstraction. In *IWOOS*, 1991.

[36] R.F. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[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] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the First Symposium on OSDI*, pages 255–266, November 1994.

[39] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[40] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *Trans. on Computer Systems*, 2(4):277–288, November 1984.

[41] Margo Seltzer et al. An introduction to the architecture of the VINO kernel, November 1994.

[42] R.L. Sites. Alpha axp architecture. *Comm. of the ACM*, 36(2), February 1993.

[43] M. Stonebraker. Operating system support for database management. *CACM*, 24(7):412–418, July 1981.

[44] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S.J. Mullender, A. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[45] C. A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[46] 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, 1993.

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

[48] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessing operating system. *Communications of the ACM*, 17(6):337–345, July 1974.