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, U.S.A
{engler, kaashoek, james}@lcs.mit.edu

Abstract

Traditional operating systems limit the performance, flexibility, and functionality of applications by fixing the interface and implementation of operating system abstractions such as interprocess communication and virtual memory. The *exokernel* operating system architecture addresses this problem by providing application-level management of physical resources. In the exokernel architecture, a small kernel securely exports all hardware resources through a low-level interface to untrusted library operating systems. Library operating systems use this interface to implement system objects and policies. This separation of resource protection from management allows application-specific customization of traditional operating system abstractions by extending, specializing, or even replacing libraries.

We have implemented a prototype exokernel operating system. Measurements show that most primitive kernel operations (such as exception handling and protected control transfer) are ten to 100 times faster than in Ultrix, a mature monolithic UNIX operating system. In addition, we demonstrate that an exokernel allows applications to control machine resources in ways not possible in traditional operating systems. For instance, virtual memory and interprocess communication abstractions are implemented entirely within an application-level library. Measurements show that application-level virtual memory and interprocess communication primitives are five to 40 times faster than Ultrix's kernel primitives. Compared to state-of-the-art implementations from the literature, the prototype exokernel system is at least five times faster on operations such as exception dispatching and interprocess communication.

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. Traditionally, operating systems hide information about machine resources behind high-level abstractions such as processes, files, address spaces and interprocess communication. These abstractions define a virtual machine on which applications execute; their implementation cannot be replaced or modified by untrusted applications. Hardcoding the implementations of these abstractions is inappropriate for three main reasons: it denies applications the advantages of domain-specific optimizations, it discourages changes to the implementations of existing abstractions, and it restricts the flexibility of application builders, since new abstractions can only be added by awkward emulation on top of existing ones (if they can be added at all).

We believe these problems can be solved through *application-level* (*i.e.,* untrusted) resource management. To this end, we have designed a new operating system architecture, *exokernel*, in which traditional operating system abstractions, such as virtual memory (VM) and interprocess communication (IPC), are implemented entirely at application level by untrusted software. In this architecture, a minimal kernel—which we call an *exokernel*—securely multiplexes available hardware resources. Library operating systems, working above the exokernel interface, implement higher-level abstractions. Application writers select libraries or implement their own. New implementations of library operating systems are incorporated by simply relinking application executables.

Substantial evidence exists that applications can benefit greatly from having more control over how machine resources are used to implement higher-level abstractions. Appel and Li [5] 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.* [10] reported that application-level control over file caching can reduce application running time by 45%. Harty and Cheriton [26] and Krueger *et al.* [30] showed how application-specific virtual memory policies can increase application performance. Stonebraker [47] argued that inappropriate file-system implementation decisions can have a dramatic impact on the performance of databases. Thekkath and Levy [50] demonstrated that exceptions can be made an order of magnitude faster by deferring signal handling to applications.

To provide applications control over machine resources, an exokernel defines a low-level interface. The exokernel architecture is founded on and motivated by a single, simple, and old observation: the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions.

To provide an interface that is as low-level as possible (ideally, just the hardware interface), an exokernel designer has a single overriding goal: to separate protection from management. For instance, an exokernel should protect framebuffers without understanding windowing systems and disks without understanding file systems. One approach is to give each application its own virtual machine [17]. As we discuss in Section 8, virtual machines can have severe performance penalties. Therefore, an exokernel uses a different approach: it *exports* hardware resources rather than emulating them, which allows an efficient and simple implementation. An exokernel employs three techniques to export resources securely. First, by using *secure bindings*, applications can securely bind to machine resources and handle events. Second, by using *visible re-*

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.

Appears in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

source revocation, applications participate in a resource revocation protocol. Third, by using an *abort protocol*, an exokernel can break secure bindings of uncooperative applications by force.

We have implemented a prototype exokernel system based on secure bindings, visible revocation, and abort protocols. It includes an exokernel (Aegis) and an untrusted library operating system (ExOS). We use this system to demonstrate several important properties of the exokernel architecture: (1) exokernels can be made efficient due to the limited number of simple primitives they must provide; (2) low-level secure multiplexing of hardware resources can be provided with low overhead; (3) traditional abstractions, such as VM and IPC, can be implemented efficiently at application level, where they can be easily extended, specialized, or replaced; and (4) applications can create special-purpose implementations of abstractions, tailored to their functionality and performance needs.

In practice, our prototype exokernel system provides applications with greater flexibility and better performance than monolithic and microkernel systems. Aegis's low-level interface allows application-level software such as ExOS to manipulate resources very efficiently. Aegis's protected control transfer is almost seven times faster than the best reported implementation [33]. Aegis's exception dispatch is five times faster than the best reported implementation [50]. On identical hardware, Aegis's exception dispatch and control transfer are roughly two orders of magnitude faster than in Ultrix 4.2, a mature monolithic system.

Aegis also gives ExOS (and other application-level software) flexibility that is not available in microkernel-based systems. For instance, virtual memory is implemented at application level, where it can be tightly integrated with distributed shared memory systems and garbage collectors. Aegis's efficient protected control transfer allows applications to construct a wide array of efficient IPC primitives by trading performance for additional functionality. In contrast, microkernel systems such as Amoeba [48], Chorus [43], Mach [2], and V [15] do not allow untrusted application software to define specialized IPC primitives because virtual memory and message passing services are implemented by the kernel and trusted servers. Similarly, many other abstractions, such as page-table structures and process abstractions, cannot be modified in microkernels. Finally, many of the hardware resources in microkernel systems, such as the network, screen, and disk, are encapsulated in heavyweight servers that cannot be bypassed or tailored to application-specific needs. These heavyweight servers can be viewed as fixed kernel subsystems that run in user-space.

This paper focuses on the exokernel architecture design and how it can be implemented securely and efficiently. Section 2 provides a more detailed case for exokernels. Section 3 discusses the issues that arise in their design. Section 4 overviews the status of our prototype and explains our experimental methodology. Sections 5 and 6 present the implementation and summarize performance measurements of Aegis and ExOS. Section 7 reports on experiments that demonstrate the flexibility of the exokernel architecture. Section 8 summarizes related work and Section 9 concludes.

2 Motivation for Exokernels

Traditionally, operating systems have centralized resource management via a set of abstractions that cannot be specialized, extended, or replaced. Whether provided by the kernel or by trusted user-level servers (as in microkernel-based systems), these abstractions are implemented by privileged software that must be used by all applications, and therefore cannot be changed by untrusted software. Typically, the abstractions include processes, files, address spaces, and interprocess communication. In this section we discuss the problems with general-purpose implementations of these abstractions and show how the exokernel architecture addresses these problems.

2.1 The Cost of Fixed High-Level Abstractions

The essential observation about 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 and Sproul [32], Anderson *et al.* [4] and Massalin and Pu [36], general-purpose implementations of 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. We argue that preventing the modification of the implementation of these high-level abstractions can reduce the performance, increase the complexity, and limit the functionality of application programs.

Fixed high-level abstractions *hurt application performance* because there is no single way to abstract physical resources or to implement an abstraction that is best for all applications. In implementing an abstraction, an 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 class of applications. For example, 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. The performance improvements of such application-specific policies can be substantial; Cao *et al.* [10] measured that application-controlled file caching can reduce application running time by as much as 45%.

Fixed high-level abstractions *hide information* from applications. For instance, most current systems do not make low-level exceptions, timer interrupts, or raw device I/O directly available to application-level software. Unfortunately, hiding this information makes it difficult or impossible for applications to implement their own resource management abstractions. For example, database implementors must struggle to emulate random-access record storage on top of file systems [47]. As another example, implementing lightweight threads on top of heavyweight processes usually requires compromises in correctness and performance, because the operating system hides page faults and timer interrupts [4]. In such cases, application complexity increases because of the difficulty of getting good performance from high-level abstractions.

Fixed high-level abstractions *limit the functionality* of applications, because they are the only available interface between applications and hardware resources. Because all applications must share one set of abstractions, changes to these abstractions occur rarely, if ever. This may explain why few good ideas from the last decade of operating systems research have been adopted into widespread use: how many production operating systems support scheduler activations [4], multiple protection domains within a single address space [11], efficient IPC [33], or efficient and flexible virtual memory primitives [5, 26, 30]?

2.2 Exokernels: An End-to-End Argument

The familiar "end-to-end" argument applies as well to low-level operating system software as it does to low-level communication protocols [44]. Applications know better than operating systems what the goal of their resource management decisions should be and therefore, they should be given as much control as possible over

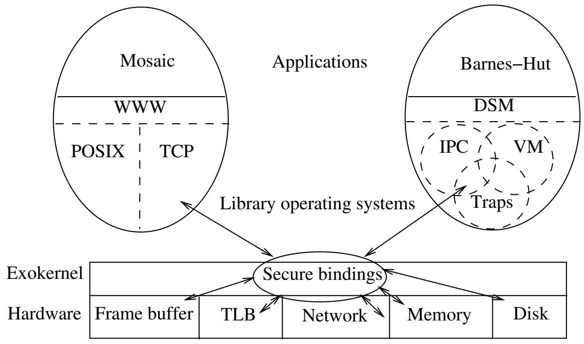

Figure 1: An example exokernel-based system consisting of a thin exokernel veneer that exports resources to library operating systems through secure bindings. Each library operating system implements its own system objects and policies. Applications link against standard libraries (*e.g.,* WWW, POSIX, and TCP libraries for Web applications) or against specialized libraries (*e.g.,* a distributed shared memory library for parallel applications).

those decisions. Our solution is to allow traditional abstractions to be implemented entirely at application level.

To provide the maximum opportunity for application-level resource management, the exokernel architecture consists of a thin exokernel veneer that multiplexes and exports physical resources securely through a set of low-level primitives. Library operating systems, which use the low-level exokernel interface, implement higher-level abstractions and can define special-purpose implementations that best meet the performance and functionality goals of applications (see Figure 1). (For brevity, we sometimes refer to "library operating system" as "application.") This structure allows the extension, specialization and even replacement of abstractions. For instance, page-table structures can vary among library operating systems: an application can select a library with a particular implementation of a page table that is most suitable to its needs. To the best of our knowledge, no other secure operating system architecture allows applications so much useful freedom.

This paper demonstrates that the exokernel architecture is an effective way to address the problems listed in Section 2.1. Many of these problems are solved by simply moving the implementation of abstractions to application level, since conflicts between application needs and available abstractions can then be resolved without the intervention of kernel architects. Furthermore, secure multiplexing does not require complex algorithms; it mostly requires tables to track ownership. Therefore, the implementation of an exokernel can be simple. A simple kernel improves reliability and ease of maintenance, consumes few resources, and enables quick adaptation to new requirements (*e.g.,* gigabit networking). Additionally, as is true with RISC instructions, the simplicity of exokernel operations allows them to be implemented efficiently.

2.3 Library Operating Systems

The implementations of abstractions in library operating systems can be simpler and more specialized than in-kernel implementations, because library operating systems need not multiplex a resource among competing applications with widely different demands. In addition, since libraries are not trusted by an 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 crossings in an exokernel system can be smaller, since most of the operating system runs in the address space of the application.

Library operating systems can provide as much portability and compatibility as is desirable. Applications that use an exokernel interface directly will not be portable, because the interface will include hardware-specific information. Applications that use library operating systems that implement standard interfaces (*e.g.,* POSIX) will be portable across any system that provides the same interface. An application that runs on an exokernel can freely replace these library operating systems without any special privileges, which simplifies the addition and development of new standards and features. We expect that most applications will use a handful of available library operating systems that implement the popular interfaces; only designers of more ambitious applications will develop new library operating systems that fit their needs. Library operating systems themselves can be made portable by designing them to use a low-level machine-independent layer to hide hardware details.

Extending or specializing a library operating system might be considerably simplified by modular design. It is possible that object-oriented programming methods, overloading, and inheritance can provide useful operating system service implementations that can be easily specialized and extended, as in the VM++ library [30]. To reduce the space required by these libraries, support for shared libraries and dynamic linking will be an essential part of a complete exokernel-based system.

As in microkernel systems, an exokernel can provide backward compatibility in three ways: one, binary emulation of the operating system and its programs; two, by implementing its hardware abstraction layer on top of an exokernel; and three, re-implementing the operating system's abstractions on top of an exokernel.

3 Exokernel Design

The challenge for an exokernel is to give library operating systems maximum freedom in managing physical resources while protecting them from each other; a programming error in one library operating system should not affect another library operating system. To achieve this goal, an exokernel separates protection from management through a low-level interface.

In separating protection from management, an exokernel performs three important tasks: (1) tracking ownership of resources, (2) ensuring protection by guarding all resource usage or binding points, and (3) revoking access to resources. To achieve these tasks, an exokernel employs three techniques. First, using *secure bindings*, library operating systems can securely bind to machine resources. Second, *visible revocation* allows library operating systems to participate in a resource revocation protocol. Third, an *abort protocol* is used by an exokernel to break secure bindings of uncooperative library operating systems by force.

In this section, we enumerate the central design principles of the exokernel architecture. Then, we discuss in detail the three techniques that we use to separate protection from management.

3.1 Design Principles

An exokernel specifies the details of the interface that library operating systems use to claim, release, and use machine resources. This section articulates some of the principles that have guided our efforts to design an exokernel interface that provides library operating systems the maximum degree of control.

Securely expose hardware. The central tenet of the exokernel architecture is that the kernel should provide secure low-level primitives that allow all hardware resources to be accessed as directly as possible. An exokernel designer therefore 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 (TLB), and addressing context identifiers. This principle extends to less tangible machine resources such as interrupts, exceptions, and cross-domain calls. An exokernel should not impose higher-level abstractions on these events (*e.g.,* Unix signal or RPC semantics). For improved flexibility, most physical resources should be finely subdivided. The number, format, and current set of TLB mappings should be visible to and replaceable by library operating systems, as should any "privileged" co-processor state. An exokernel must export privileged instructions to library operating systems 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.

Phrased negatively, this principle states that an exokernel should *avoid resource management*. It should only manage resources to the extent required by protection (*i.e.,* management of allocation, revocation, and ownership). The motivation for this principle is our belief that distributed, application-specific, resource management is the best way to build efficient flexible systems. Subsequent principles deal with the details of achieving this goal.

Expose allocation. An exokernel should allow library operating systems to request specific physical resources. For instance, if a library operating system can request specific physical pages, it can reduce cache conflicts among the pages in its working set [29]. Furthermore, resources should not be implicitly allocated; the library operating system should participate in every allocation decision. The next principle aids the effectiveness of this participation.

Expose Names. An exokernel should export physical names. Physical names are efficient, since they remove a level of indirection otherwise required to translate between virtual and physical names. Physical names also encode useful resource attributes. For example, in a system with physically-indexed direct-mapped caches, the name of a physical page (*i.e.,* its page number) determines which pages it conflicts with. Additionally, an exokernel should export bookkeeping data structures such as freelists, disk arm positions, and cached TLB entries so that applications can tailor their allocation requests to available resources.

Expose Revocation. An exokernel should utilize a visible resource revocation protocol so that well-behaved library operating systems can perform effective application-level resource management. Visible revocation allows physical names to be used easily and permits library operating systems to choose which instance of a specific resource to relinquish.

Policy

An exokernel hands over resource policy decisions to library operating systems. Using this control over resources, an application or collection of cooperating applications can make decisions about how best to use these resources. However, as in all systems, an exokernel must include policy to arbitrate between competing library operating systems: it must determine the absolute importance of different applications, their share of resources, *etc.* This situation is no different than in traditional kernels. Appropriate mechanisms are determined more by the environment than by the operating system architecture. For instance, while an exokernel cedes management of resources over to library operating systems, it controls the allocation and revocation of these resources. By deciding which allocation requests to grant and from which applications to revoke resources, an exokernel can enforce traditional partitioning strategies, such as quotas or reservation schemes. Since policy conflicts boil down to resource allocation decisions (*e.g.,* allocation of seek time, physical memory, or disk blocks), an exokernel handles them in a similar manner.

3.2 Secure Bindings

One of the primary tasks of an exokernel is to multiplex resources *securely*, providing protection for mutually distrustful applications. To implement protection an exokernel must guard each resource. To perform this task efficiently an exokernel allows library operating systems to bind to resources using *secure bindings*.

A secure binding is a protection mechanism that decouples authorization from the actual use of a resource. Secure bindings improve performance in two ways. First, the protection checks involved in enforcing a secure binding are expressed in terms of simple operations that the kernel (or hardware) can implement quickly. Second, a secure binding performs authorization only at bind time, which allows management to be decoupled from protection. Application-level software is responsible for many resources with complex semantics (*e.g.,* network connections). By isolating the need to understand these semantics to *bind time*, the kernel can efficiently implement access checks at *access time* without understanding them. Simply put, a secure binding allows the kernel to protect resources without understanding them.

Operationally, the one requirement needed to support secure bindings is a set of primitives that application-level software can use to express protection checks. The primitives can be implemented either in hardware or software. A simple hardware secure binding is a TLB entry: when a TLB fault occurs the complex mapping of virtual to physical addresses in a library operating system's page table is performed and then loaded into the kernel (bind time) and then used multiple times (access time). Another example is the packet filter [37], which allows predicates to be downloaded into the kernel (bind time) and then run on every incoming packet to determine which application the packet is for (access time). Without a packet filter, the kernel would need to query every application or network server on every packet reception to determine who the packet was for. By separating protection (determining who the packet is for) from authorization and management (setting up connections, sessions, managing retransmissions, *etc.*) very fast network multiplexing is possible while still supporting complete application-level flexibility.

We use three basic techniques to implement secure bindings: hardware mechanisms, software caching, and downloading application code.

Appropriate hardware support allows secure bindings to be couched as low-level protection operations 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 capabilities for the physical pages. An exokernel would enforce capability checking without needing any information about the file system's authorization mechanisms. As another 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 a library operating system 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.

Secure bindings can be cached in an exokernel. For instance, an exokernel can use a large software TLB [7, 28] to cache address translations that do not fit in the hardware TLB. The software TLB can be viewed as a cache of frequently-used secure bindings.

Secure bindings can be implemented by downloading code into the kernel. This code is invoked on every resource access or event to determine ownership and the actions that the kernel should perform. Downloading code into the kernel allows an application thread of control to be immediately executed on kernel events. The advantages of downloading code are that potentially expensive crossings can be avoided and that this code can run without requiring the application itself to be scheduled. Type-safe languages [9, 42], interpretation, and sandboxing [52] can be used to execute untrusted application code safely [21].

We provide examples of each of these three techniques below and discuss how secure bindings apply to the secure multiplexing of physical memory and network devices.

Multiplexing Physical Memory

Secure bindings to physical memory are implemented in our prototype exokernel using self-authenticating capabilities [12] and address translation hardware. When a library operating system 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 library operating system. The owner of a page has the power to change the capabilities associated with it 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 library operating system requesting access. If the capability is insufficient, the request is denied. Typically, the processor contains a TLB, and the exokernel must check memory capabilities when a library operating system attempts to enter a new virtual-to-physical mapping. To improve library operating system performance by reducing the number times secure bindings must be established, an exokernel may cache virtual-to-physical mappings in a large software TLB.

If the underlying hardware defines a page-table interface, then an 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 an exokernel. As dictated by the exokernel principle of exposing kernel book-keeping structures, the page table should be visible (read only) at application level.

Using capabilities to protect resources enables applications to grant access rights to other applications without kernel intervention. Applications can also use "well-known" capabilities to share resources easily.

To break a secure binding, an exokernel must change the associated capabilities and mark the resource as free. In the case of physical memory, an exokernel would flush all TLB mappings and any queued DMA requests.

Multiplexing the Network

Multiplexing the network efficiently is challenging, since protocol-specific knowledge is required to interpret the contents of incoming messages and identify the intended recipient.

Support for network demultiplexing can be provided either in software or hardware. An example of a hardware-based mechanism is the use of the virtual circuit in ATM cells to securely bind streams to applications [19].

Software support for message demultiplexing can be provided by packet filters [37]. Packet filters can be viewed as an implementation of secure bindings in which application code—the filters—are downloaded into the kernel. Protocol knowledge is limited to the application, while the protection checks required to determine packet ownership are couched in a language understood by the kernel. Fault isolation is ensured by careful language design (to bound runtime) and runtime checks (to protect against wild memory references and unsafe operations).

Our prototype exokernel uses packet filters, because our current network does not provide hardware mechanisms for message demultiplexing. One challenge with a language-based approach is to make running filters fast. Traditionally, packet filters have been interpreted, making them less efficient than in-kernel demultiplexing routines. One of the distinguishing features of the packet filter engine used by our prototype exokernel is that it compiles packet filters to machine code at runtime, increasing demultiplexing performance by more than an order of magnitude [22].

The one problem with the use of a packet filter is ensuring that that a filter does not "lie" and accept packets destined to another process. Simple security precautions such as only allowing a trusted server to install filters can be used to address this problem. On a system that assumes no malicious processes, our language is simple enough that in many cases even the use of a trusted server can be avoided by statically checking a new filter to ensure that it cannot accept packets belonging to another; by avoiding the use of any central authority, extensibility is increased.

Sharing the network interface for outgoing messages is easy. Messages are simply copied from application space into a transmit buffer. In fact, with appropriate hardware support, transmission buffers can be mapped into application space just as easily as physical memory pages [19].

3.2.1 Downloading Code

In addition to implementing secure bindings, downloading code can be used to improve performance. Downloading code into the kernel has two main performance advantages. The first is obvious: elimination of kernel crossings. The second is more subtle: the execution time of downloaded code can be readily bounded [18]. The crucial importance of "tamed" code is that it can be executed when the application is not scheduled. This decoupling allows downloaded code to be executed in situations where context switching to the application itself is infeasible (*e.g.,* when only a few microseconds of free processing time is available). Packet filters are an example of this feature: since the packet-filter runtime is bounded, the kernel can use it to demultiplex messages irrespective of what application is scheduled; without a packet filter the operating system would have to schedule each potential consumer of the packet [37].

Application-specific Safe Handlers (ASHs) are a more interesting example of downloading code into our prototype exokernel. These application handlers can be downloaded into the kernel to participate in message processing. An ASH is associated with a packet filter and runs on packet reception. One of the key features of an ASH is that it can initiate a message. Using this feature, roundtrip latency can be greatly reduced, since replies can be transmitted on the spot instead of being deferred until the application is scheduled. ASHs have a number of other useful features (see Section 6).

A salient issue in downloading code is the *level* at which the code is specified. High-level languages have more semantic information, which provides more information for optimizations. For example, our packet-filter language is a high-level declarative language. As a result packet filters can be merged [56] in situations where merging a lower-level, imperative language would be infeasible. However, in cases where such optimizations are not done, (*e.g.,* in an exception handler) a low-level language is more in keeping with the exokernel philosophy: it allows the broadest range of application-level languages to be targeted to it and the simplest implementation. ASHs are another example of this tradeoff: most ASHs are imported into the kernel in the form of the object code of the underlying machine; however, in the few key places where higher level semantics are useful we have extended the instruction set of the machine.

3.3 Visible Resource Revocation

Once resources have been bound to applications, there must be a way to reclaim them and break their secure bindings. Revocation can either be *visible* or *invisible* to applications. Traditionally, operating systems have performed revocation invisibly, deallocating resources without application involvement. For example, with the exception of some external pagers [2, 43], 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 disadvantages are that library operating systems cannot guide deallocation and have no knowledge that resources are scarce.

An exokernel uses visible revocation for most resources. Even the processor is explicitly revoked at the end of a time slice; a library operating system can react by saving only the required processor state. For example, a library operating system could avoid saving the floating point state or other registers that are not live. However, since visible revocation requires interaction with a library operating system, invisible revocation can perform better when 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

The use of physical resource names requires that an exokernel reveal each revocation to the relevant library operating system so that it can relocate its physical names. For instance, a library operating system that relinquishes physical page "5" should update any of its page-table entries that refer to this page. This is easy for a library operating system to do when it deallocates a resource in reaction to an exokernel revocation request. An abort protocol (discussed below) allows relocation to be performed when an exokernel forcibly reclaims a resource.

We view the revocation process as a dialogue between an exokernel and a library operating system. Library operating systems should organize resource lists so that resources can be deallocated quickly. For example, a library operating system could have a simple vector of physical pages that it owns: when the kernel indicates that some page should be deallocated, the library operating system selects one of its pages, writes it to disk, and frees it.

3.4 The Abort Protocol

An exokernel must also be able to take resources from library operating systems that fail to respond satisfactorily to revocation requests. An 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 microseconds"). However, if a library operating system fails to respond quickly, the secure bindings need to be broken "by force." The actions taken when a library operating system is recalcitrant are defined by the *abort protocol.*

One possible abort protocol is to simply kill any library operating system and its associated application that fails to respond quickly to revocation requests. We rejected this method because we believe that most programmers have great difficulty reasoning about hard real-time bounds. Instead, if a library operating system fails to comply with the revocation protocol, an exokernel simply breaks all existing secure bindings to the resource and informs the library operating system.

To record the forced loss of a resource, we use a *repossession vector*. When an exokernel takes a resource from a library operating system, this fact is registered in the vector and the library operating system receives a "repossession" exception so that it can update any mappings that use the resource. For resources with state, an exokernel can write the state into another memory or disk resource. In preparation, the library operating system 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 an exokernel should not arbitrarily choose the resource to repossess. A library operating system may use 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 each library operating system a small number of resources that will not be repossessed (*e.g.,* five to ten physical memory pages). If even those resources must be repossessed, some emergency exception that tells a library operating system to submit itself to a "swap server" is required.

4 Status and Experimental Methodology

We have implemented two software systems that follow the exokernel architecture: *Aegis*, an exokernel, and *ExOS*, a library operating system. Another prototype exokernel, *Glaze*, is being built for an experimental SPARC-based shared-memory multiprocessor [35], along with *PhOS*, a parallel operating system library.

Aegis and ExOS are implemented on MIPS-based DECstations. Aegis exports the processor, physical memory, TLB, exceptions, and interrupts. In addition, it securely exports the network interface using a packet filter system that employs dynamic code generation. ExOS implements processes, virtual memory, user-level exceptions, various interprocess abstractions, and several network protocols (ARP/RARP, IP, UDP, and NFS). A native extensible file system that implements global buffer management is under development. Currently, our prototype system has no real users, but is used extensively for development and experimentation.

The next three sections describe the implementation of Aegis, ExOS, and extensions to ExOS. Included in the discussion are experiments that test the efficacy of the exokernel approach. These experiments test four hypotheses:

- Exokernels can be very efficient.
- Low-level, secure multiplexing of hardware resources can be implemented efficiently.
- Traditional operating system abstractions can be implemented efficiently at application level.
- Applications can create special-purpose implementations of these abstractions.

On identical hardware we compare the performance of Aegis and ExOS with the performance of Ultrix4.2, a mature monolithic UNIX operating system. It is important to note that Aegis and

Machine	Processor	SPEC rating	MIPS
DEC2100 (12.5 MHz)	R2000	8.7 SPECint89	~ 11
DEC3100 (16.67 MHz)	R3000	11.8 SPECint89	~ 15
DEC5000/125 (25 MHz)	R3000	16.1 SPECint92	~ 25

Table 1: Experimental platforms.

ExOS do not offer the same level of functionality as Ultrix. We do not expect these additions to cause large increases in our timing measurements.

The comparisons with Ultrix serve two purposes. First, they show that there is much overhead in today's systems, which can be easily removed by specialized implementations. Second, they provide a well-known, easily-accessible point of reference for understanding Aegis's and ExOS's performance. 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 operating systems [39]. For example, it performs two to three times better than Mach 3.0 in a set of I/O benchmarks [38]. Also, its virtual memory performance is approximately twice that of Mach 2.5 and three times that of Mach 3.0 [5].

In addition, we attempt to assess Aegis's and ExOS's performance in the light of recent advances in operating systems research. These advances have typically been evaluated on different hardware and frequently use experimental software, making head-to-head comparisons impossible. In these cases we base our comparisons on relative SPECint ratings and instruction counts.

Table 1 shows the specific machine configurations used in the experiments. For brevity, we refer to the DEC5000/125 as DEC5000. The three machine configurations are used to get a tentative measure of the scalability of Aegis. All times are measured using the "wall-clock." We used `clock` on the Unix implementations and a microsecond counter on Aegis. Aegis's time quantum was set at 15.625 milliseconds. All benchmarks were compiled using an identical compiler and flags: `gcc` version 2.6.0 with optimization flags "-O2." None of the benchmarks use floating-point instructions; therefore, we do not save floating-point state. Both systems were run in "single-user" mode and were isolated from the network.

The per-operation cost was obtained by repeating the operation a large number of times and averaging. As a result, the measurements do not consider cold start misses in the cache or TLB, and therefore represent a "best case." Because, Ultrix has a much larger cache and virtual memory footprint than Aegis, this form of measurement is more favorable to Ultrix. Because Ultrix was sensitive to the instance of the type of machine it was run on, we took the best time measured. The exokernel numbers are the median of three trials.

A few of our benchmarks are 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 are 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.

System call	Description
`Yield`	Yield processor to named process
`Scall`	Synchronous protected control transfer
`Acall`	Asynchronous protected control transfer
`Alloc`	Allocation of resources (*e.g.,* physical page)
`Dealloc`	Deallocation of resources

Table 2: A subset of the Aegis system call interface.

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

Table 3: A sample of Aegis's primitive operations.

5 Aegis: an Exokernel

This section describes the implementation and performance of Aegis. The performance numbers demonstrate that Aegis and low-level multiplexing can be implemented efficiently. We describe in detail how Aegis multiplexes the processor, dispatches exceptions, translates addresses, transfers control between address spaces, and multiplexes the network.

5.1 Aegis Overview

Table 2 lists a subset of the Aegis interface. We discuss the implementation of most of the system calls in this section. Aegis also supports a set of *primitive operations* that encapsulate privileged instructions and are guaranteed not to alter application-visible registers (see Table 3 for some typical examples). These primitive operations can be viewed as pseudo-instructions (similar to the Alpha's use of PALcode [45]). In this subsection we examine how Aegis protects time slices and processor environments; other resources are protected as described in Section 3.

5.1.1 Processor Time Slices

Aegis represents the CPU 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 to maximize responsiveness.

Timer interrupts denote the beginning and end of time slices, and are delivered in a manner similar to exceptions (discussed below): 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*. This framework gives applications a large degree of control over context switching. For example, it can be used to implement scheduler activations [4].

Fairness is achieved by bounding the time an application takes to save its context: each subsequent timer interrupt (which demarcates a time slice) is recorded in an excess time counter. Applications pay for each excess time slice consumed by forfeiting a subsequent time slice. If the excess time counter exceeds a predetermined threshold, the environment is destroyed. In a more friendly implementation, Aegis could perform a complete context switch for the application.

This simple scheduler can support a wide range of higher-level scheduling policies. As we demonstrate in Section 7, an application can enforce proportional sharing on a collection of sub-processes.

5.1.2 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 (such as revocation exceptions) to its designated owner.

Four kinds of events are delivered by Aegis: exceptions, interrupts, protected control transfers, and address translations. Processor environments contain the four contexts required to support these events:

Exception context: for each exception an exception context contains a program counter for where to jump to and a pointer to physical memory for saving registers.

Interrupt context: for each interrupt an interrupt context includes a program counters and register-save region. In the case of 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: a protected entry context specifies program counters 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: an addressing context consists of a set of *guaranteed mappings*. A TLB miss on a virtual address that is mapped by a guaranteed mapping is handled by Aegis. Library operating systems rely on guaranteed mappings for bootstrapping page-tables, exception handling code, and exception stacks. The addressing context also includes an address space identifier, a status register, and a tag used to hash into the Aegis software TLB (see Section 5.4). To switch from one environment to another, Aegis must install these three 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.

5.2 Base Costs

The base cost for null procedure and system calls are shown in Table 4. The null procedure call shows that Aegis's scheduling flexibility does not add overhead to base operations. Aegis has two system call paths: the first for system calls that do not require a stack, the second for those that do. With the exception of protected control transfers, which are treated as a special case for efficiency, all Aegis system calls are vectored along one of these two paths. Ultrix's `getpid` is approximately an order of magnitude slower than Aegis's slowest system call path—this suggests that the base cost of demultiplexing system calls is significantly 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 a MIPS processor, kernel TLB faults are vectored

Machine	OS	Procedure call	Syscall (getpid)
DEC2100	Ultrix	0.57	32.2
DEC2100	Aegis	0.56	3.2 / 4.7
DEC3100	Ultrix	0.42	33.7
DEC3100	Aegis	0.42	2.9 / 3.5
DEC5000	Ultrix	0.28	21.3
DEC5000	Aegis	0.28	1.6 / 2.3

Table 4: Time to perform null procedure and system calls. Two numbers are listed for Aegis's system calls: the first for system calls that do not use a stack, the second for those that do. Times are in microseconds.

Machine	OS	unalign	overflow	coproc	prot
DEC2100	Ultrix	n/a	208.0	n/a	238.0
DEC2100	Aegis	2.8	2.8	2.8	3.0
DEC3100	Ultrix	n/a	151.0	n/a	177.0
DEC3100	Aegis	2.1	2.1	2.1	2.3
DEC5000	Ultrix	n/a	130.0	n/a	154.0
DEC5000	Aegis	1.5	1.5	1.5	1.5

Table 5: Time to dispatch an exception in Aegis and Ultrix; times are in microseconds.

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 this to be the common case with exokernels.

5.3 Exceptions

Aegis dispatches all hardware exceptions to applications (save for system calls) using techniques similar to those described in Thekkath and Levy [50]. To dispatch 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 cause of the exception to perform an indirect jump to an application-specified program counter value, where execution resumes with the appropriate permissions set (*i.e.,* in user-mode with interrupts re-enabled).

After processing an 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. This means that all registers that are saved must be in user-accessible memory locations.

Currently, Aegis dispatches exceptions in 18 instructions. The low-level nature of Aegis allows an extremely efficient implementation: the time for exception dispatching on a DECstation5000/125 is 1.5 microseconds. This time is over five times faster than the most highly-tuned implementation in the literature (8 microseconds on DECstation5000/200 [50], a machine that is 1.2 faster on SPECint92 than our DECstation5000/125). Part of the reason for this improvement is that Aegis does not use mapped data structures, and so does not have to separate kernel TLB misses from the more general class of exceptions in its exception demultiplexing routine. Fast exceptions enable a number of intriguing applications: efficient page-protection traps can be used by applications such as distributed shared memory systems, persistent object stores, and garbage collectors [5, 50].

Table 5 shows exception dispatch times for unaligned pointer accesses (**unalign**), arithmetic overflow (**overflow**), attempted use of the floating point co-processor 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" an 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 Table 5. In each case, Aegis's exception dispatch times are approximately two orders of magnitude faster than Ultrix.

5.4 Address Translations

This section looks at two problems in supporting application-level virtual memory: bootstrapping and efficiency. An exokernel must provide support for bootstrapping the virtual naming system (*i.e.,* it must support 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. A miss on a guaranteed mapping will be handled automatically by Aegis. This organization frees the application from dealing with the intricacies of boot-strapping TLB miss and exception handlers, which can take TLB misses. To implement guaranteed mappings efficiently, an application's virtual address space is partitioned into two segments. The first segment holds normal application data and code. Virtual addresses in the segment can be "pinned" using guaranteed mappings and typically holds exception handling code and page-tables.

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 dispatched directly to the application. If it is in the second region, Aegis first checks to see if it is a guaranteed mapping. If so, Aegis installs the TLB entry and continues; otherwise, Aegis 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 Aegis system routine.

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

4. The application performs cleanup and resumes execution.

In order to support application-level virtual memory efficiently, TLB refills must be fast. To this end, Aegis caches TLB entries (a form of secure bindings) in the kernel by overlaying the hardware TLB with a large software TLB (STLB) to absorb capacity misses [7, 28]. 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.

The STLB contains 4096 entries of 8 bytes each. It is direct-mapped and resides in unmapped physical memory. An STLB "hit" takes 18 instructions (approximately one to two microseconds). In contrast, performing an upcall to application level on a TLB miss, followed by a system call to install a new mapping is at least three to six microseconds more expensive.

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

OS	Machine	MHz	Transfer cost
Aegis	DEC2100	12.5MHz	2.9
Aegis	DEC3100	16.67MHz	2.2
Aegis	DEC5000	25MHz	1.4
L3	486	50MHz	9.3 (normalized)

Table 6: Time to perform a (unidirectional) protected control transfer; times are in microseconds.

5.5 Protected Control Transfers

Aegis provides a *protected control transfer* mechanism as a substrate for efficient implementations of IPC abstractions. 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 the 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 only the remainder of the current time slice to the callee. Synchronous calls donate the current time and all future instantiations of it; the callee can return the time slice via a synchronous control transfer call back to the original caller. Both forms of control transfer guarantee two important properties. First, to applications, a protected control transfer is atomic: once initiated it will reach the callee. Second, Aegis will not overwrite any application-visible register. These two properties allow the large register sets of modern processors to be used as a temporary message buffer [14].

Currently, our synchronous protected control transfer operation takes 30 instructions. Roughly ten of these instructions are used to distinguish the system call "exception" from other hardware exceptions on the MIPS architecture. Setting the status, co-processor, and address-tag registers consumes the remaining 20 instructions, and could benefit from additional optimizations. Because Aegis implements the minimum functionality required for any control transfer mechanism, applications can efficiently construct their own IPC abstractions. Sections 6 and 7 provide examples.

Table 6 shows the performance in microseconds of a "bare-bone" protected control transfer. This time is derived by dividing the time to perform a call and reply in half (*i.e.,* we measure the time to perform a unidirectional control transfer). Since the experiment is intended to measure the cost of protected control transfer only, no registers are saved and restored. However, due to our measurement code, the time includes the overhead of incrementing a counter and performing a branch.

We attempt a crude comparison of our protected control transfer operation to the equivalent operation on L3 [33]. The L3 implementation is the fastest published result, but it runs on an Intel 486 DX-50 (50 MHz). For Table 6, we scaled the published L3 results (5 microseconds) by the SPECint92 rating of Aegis's DEC5000 and L3's 486 (16.1 vs. 30.1). Aegis's trusted control transfer mechanism is 6.6 times faster than the scaled time for L3's RPC mechanism.

Filter	Cold Cache	Warm Cache
MPF	71.0	35.0
PATHFINDER	39.0	19.0
DPF	7.5	1.5

Table 7: Time on a DEC5000/200 to classify TCP/IP headers destined for one of ten TCP/IP filters; times are in microseconds.

Architectural characteristics of the Intel 486 partially account for Aegis's better performance. L3 pays a heavy penalty to enter and leave the kernel (71 and 36 cycles, respectively) and must flush the TLB on a context switch.

5.6 Dynamic Packet Filter (DPF)

Aegis's network subsystem uses aggressive dynamic code generation techniques to provide efficient message demultiplexing and handling. We briefly discuss some key features of this system. A complete discussion can be found in [22].

Message demultiplexing is the process of determining which application an incoming message should be delivered to. Packet filters are a well-known technique used to implement extensible kernel demultiplexing [6, 56]. Traditionally, packet filters are interpreted, which entails a high computational cost. Aegis uses Dynamic Packet Filter (DPF), a new packet filter system that is over an order of magnitude more efficient than previous systems.

The key in our approach to making filters run fast is dynamic code generation. Dynamic code generation is the creation of executable code at *runtime*. DPF exploits dynamic code generation in two ways: (1) by using it to eliminate interpretation overhead by compiling packet filters to executable code when they are installed into the kernel and (2) by using filter constants to aggressively optimize this executable code. To gain portability, DPF compiles filters using VCODE, a portable, very fast, dynamic code generation system [20]. VCODE generates machine code in approximately 10 instructions per generated instruction and runs on a number of machines (*e.g.,* MIPS, Alpha and SPARC).

We measured DPF's time to classify packets destined for one of ten TCP/IP filters, and compare its times to times for MPF [56] (a widely used packet filter engine) and PATHFINDER [6] (the fastest packet filter engine in the literature). Table 7 presents both the "cold-cache" and "warm-cache" times to perform this message classification. Warm cache refers to two properties. First, the packet filter is in the processor cache. Second, both MPF and PATHFINDER (but not DPF) maintain a cache of filters that recently accepted messages. This experiment and the numbers for both MPF and PATHFINDER are taken from [6].

To ensure meaningful comparisons between the systems, we ran our DPF experiments on the same hardware (a DECstation 5000/200) in user space. For cold cache performance we ensure that the filter code does not reside in the instruction cache by allocating 64 contigous pages, placing the filter code at the beginning of each page, and cycling through these pages in "round-robin" order. Because the DECStation5000 has a physically-addressed, direct-mapped cache and because Ultrix guarentees that contigous virtual pages within a page-aligned 64K-region will not conflict, this methodology is sufficient to ensure that the filter is not in the instruction cache. It is unclear why the performance difference between warm and cold caches for MPF and PATHFINDER is only a factor of two. Due to its use of dynamic code generation, DPF is 10 to 20 times faster than MPF and five to 10 times faster than PATHFINDER.

Machine	OS	pipe	pipe'	shm	lrpc
DEC2100	Ultrix	326.0	n/a	187.0	n/a
DEC2100	ExOS	30.9	24.8	12.4	13.9
DEC3100	Ultrix	243.0	n/a	139.0	n/a
DEC3100	ExOS	22.6	18.6	9.3	10.4
DEC5000	Ultrix	199.0	n/a	118.0	n/a
DEC5000	ExOS	14.2	10.7	5.7	6.3

Table 8: Time for IPC using pipes, shared memory, and LRPC on ExOS and Ultrix; times are in microseconds. Pipe and shared memory are unidirectional, while LRPC is bidirectional.

5.7 Summary

The main conclusion we draw from these experiments is that an exokernel can be implemented efficiently. The reasons for Aegis's good performance are the following. One, keeping track of ownership is a simple task and can therefore be implemented efficiently. Two, since the kernel provides very little functionality beyond low-level multiplexing, it is small and lean: for instance, it keeps its data structures in physical memory. Three, by caching secure bindings in a software TLB, most hardware TLB misses can be handled efficiently. Four, by downloading packets filters and by employing dynamic code generation, secure binding to the network can be implemented efficiently.

6 ExOS: a Library Operating System

The most unusual aspect of ExOS is that it manages fundamental operating system abstractions (*e.g.,* virtual memory and process) at *application level*, completely within the address space of the application that is using it. This section demonstrates that basic system abstractions can be implemented at application level in a direct and efficient manner. Due to space constraints we focus on IPC, virtual memory, and remote communication.

6.1 IPC Abstractions

Fast interprocess communication is crucial for building efficient and decoupled systems [8, 27, 33]. As described in Section 5, the Aegis protected control transfer mechanism is an efficient substrate for implementing IPC abstractions. This section describes experiments used to measure the performance of ExOS's IPC abstractions on top of the Aegis primitives. The results of these experiments are summarized in Table 8. The experiments are:

pipe: measures the latency of sending a word-sized message from one process to another using pipes by "ping-ponging" a counter between two processes. The Ultrix **pipe** implementation uses standard UNIX pipes. 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. We use two pipe implementations: the first is a naive implementation (**pipe**), while the second (**pipe'**) exploits the fact that this library exists in application space by simply inlining the read and write calls. ExOS's unoptimized **pipe** implementation is an order of magnitude more efficient than the equivalent operation under Ultrix.

shm: measures the time for two processes to "ping-pong" using a shared counter. ExOS uses Aegis's `yield` system call to switch between partners. Ultrix does not provide a `yield` primitive, so we synthesized it using signals. ExOS's **shm** is 15 to 20 times faster than Ultrix's **shm**. ExOS's **shm** is about twice as fast as its **pipe** implementation, which must manipulate circular buffers.

Machine	OS	matrix
DEC2100	Ultrix	7.1
DEC2100	ExOS	7.0
DEC3100	Ultrix	5.2
DEC3100	ExOS	5.2
DEC5000	Ultrix	3.8
DEC5000	ExOS	3.7

Table 9: Time to perform a 150x150 matrix multiplication; time in seconds.

lrpc: this experiment measures the time to perform an LRPC into another address space, increment a counter and return its value. ExOS's LRPC is built on top of Aegis's protected control transfer mechanism. **lrpc** saves all general-purpose callee-saved registers. The **lrpc** implementation assumes that only a single function is of interest (*e.g.,* it does not use the RPC number to index into a table) and it does not check permissions. The implementation is also single-threaded.

Because Ultrix is built around a set of fixed high-level abstractions, new primitives can be added only by emulating them on top of existing ones. Specifically, implementations of **lrpc** must use pipes or signals to transfer control. The cost of such emulation is high: on Ultrix, **lrpc** using pipes costs 46 to 60 more than on ExOS and using signals costs 26 to 37 more than on ExOS. These experiments indicate that an Ultrix user either pays substantially in performance for new functionality, or is forced to modify the kernel.

6.2 Application-level Virtual Memory

ExOS provides a rudimentary virtual memory system (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 limitations, 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, and DMA.

The overhead of application-level memory is measured by performing a 150 by 150 integer 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 it to perform equivalently on both operating systems. The times in Table 9 indicate that application-level virtual memory does not add noticeable overhead to operations that have reasonable virtual memory footprints. Of course, this is hardly a conclusive proof.

Table 10 compares Aegis and ExOS to Ultrix on seven virtual memory experiments based on those used by Appel and Li [5]. These experiments are of particular interest, since they measure the cost of VM operations that are crucial for the construction of ambitious systems, such as page-based distributed shared memory systems and garbage collectors. Note that Ultrix's VM performance is quite good compared to other systems [5]. The operations measured are the following:

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

prot1: time to change the protection of a single page.

prot100: time to "read-protect" 100 pages.

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

trap: time to handle 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 and Li [5]).

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 and Li [5]). Note that **appel2** requires less time than **appel1** since **appel1** must both unprotect and protect different pages in the fault handler.

The **dirty** benchmark measures the average time to parse the page-table for a random entry. This operation illustrates two consequences of the exokernel architecture. First, kernel transitions can be eliminated by implementing abstractions at application level. Second, application-level software can implement functionality that is frequently not provided by traditional operating systems.

If we compare the time for **dirty** to the time for **prot1**, we see that over half the time spent in **prot1** is due to the overhead of parsing the page table. As we show in Section 7.2, this overhead can be reduced through the use of a data structure more tuned to efficient lookup (*e.g.,* a hash table). Even with this penalty, ExOS performs **prot1** almost twice as fast as Ultrix. The likely reason for this difference is that, as shown in Table 4, 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 exceptions are **prot100** and **unprot100**. Ultrix is extremely efficient in protecting and unprotecting contiguous ranges of virtual addresses: it performs these operations 1.1 to 1.6 times faster than Aegis. One reason for this difference is the immaturity of our implementation. Another is that changing page protections in ExOS requires access to two data structures (Aegis's STLB and ExOS's page-table). 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. 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) to handle exceptions adds overhead for saving and restoring all caller-saved registers when a trap handler starts and returns.

6.3 Application-Specific Safe Handlers (ASH)

ExOS operates efficiently in spite of executing at application level in part because the cost of crossing between kernel and user space is extremely low in our prototype (18 instructions). Most application-specific optimizations can therefore be implemented in libraries at application level. However, in the context of networking, there are two reasons for ExOS to download code: the first is technology driven, while the second is more fundamental. First, the network buffers on our machines cannot be easily mapped into application space in a secure way. Therefore, by downloading code into the kernel, applications can integrate operations such as checksumming during the copy of the message from these buffers to user space. Such integration can improve performance on a DECstation5000/200 by almost a factor of two [22]. Second, if the runtime of downloaded code is bounded, it can be run in situations when performing a full context switch to an unscheduled application is impractical. Downloading code thus allows applications to decouple latency-critical operations such as message reply from process scheduling.

We examine these issues using application-specific handlers (ASHs). ASHs are *untrusted* application-level message-handlers

Machine	OS	dirty	prot1	prot100	unprot100	trap	appel1	appel2
DEC2100	Ultrix	n/a	51.6	175.0	175.0	240.0	383.0	335.0
DEC2100	ExOS	17.5	32.5	213.0	275.0	13.9	74.4	45.9
DEC3100	Ultrix	n/a	39.0	133.0	133.0	185.0	302.0	267.0
DEC3100	ExOS	13.1	24.4	156.0	206.0	10.1	55.0	34.0
DEC5000	Ultrix	n/a	32.0	102.0	102.0	161.0	262.0	232.0
DEC5000	ExOS	9.8	16.9	109.0	143.0	4.8	34.0	22.0

Table 10: Time to perform virtual memory operations on ExOS and Ultrix; times are in microseconds. The times for **appel1** and **appel2** are per page.

Machine	OS	Roundtrip latency
DEC5000/125	ExOS/ASH	259
DEC5000/125	ExOS	320
DEC5000/125	Ultrix	3400
DEC5000/200	Ultrix/FRPC	340

Table 11: Roundtrip latency of a 60-byte packet over Ethernet using ExOS with ASHs, ExOS without ASHs, Ultrix, and FRPC; times are in microseconds.

that are downloaded into the kernel, made safe by a combination of code inspection [18] and sandboxing [52], and executed upon message arrival. The issues in other contexts (*e.g.,* disk I/O) are similar.

An ASH can perform general computation. We have augmented this ability with a set of message primitives that enable the following four useful abilities:

1. Direct, dynamic message vectoring. An ASH controls where messages are copied in memory, and can therefore eliminate all intermediate copies, which are the bane of fast networking systems.

2. Dynamic integrated layer processing (ILP) [1, 16]. ASHs can integrate data manipulations such as checksumming and conversion into the data transfer engine itself. This integration is done at the level of *pipes*. A pipe is a computation that acts on streaming data. Pipes contain sufficient semantic information for the ASH compiler to integrate several pipes into the message transfer engine at *runtime*, providing a large degree of flexibility and modularity. Pipe integration allows message traversals to be modularly aggregated to a single point in time. To the best of our knowledge, ASH-based ILP is the first to allow either dynamic pipe composition or application-extended in-kernel ILP.

3. Message initiation. ASHs can initiate message sends, allowing for low-latency message replies.

4. Control initiation. ASHs perform general computation. This ability allows them to perform control operations at message reception time, implementing such computational actions as traditional active messages [51] or remote lock acquisition.

It is important to note the power of the ASH computational model. It allows the vectoring process to be completely dynamic: the application does not have to pre-specify that it is waiting for a particular message, nor does it have to pre-bind buffer locations for the message. Instead, it can defer these decisions until message reception and use application-level data structures, system state, and/or the message itself to determine where to place the message. Capturing the same expressiveness within a statically defined protocol is difficult.

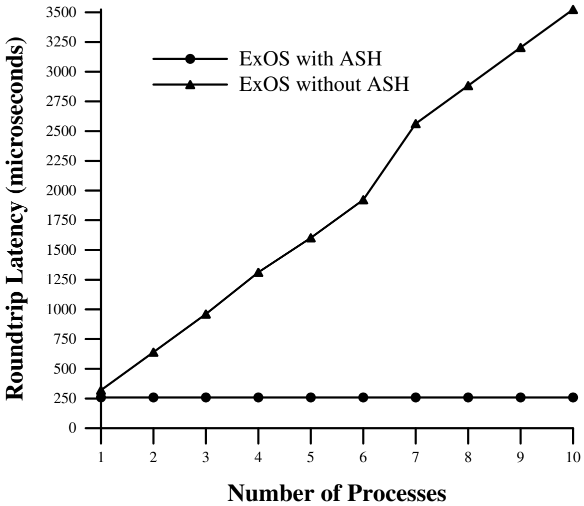

Figure 2: Average roundtrip latency with increasing number of active processes on receiver.

Table 11 shows the roundtrip latency over Ethernet of ASH-based network messaging and compares it to ExOS without ASHs, Ultrix, and FRPC [49] (the fastest RPC in the literature on comparable hardware). Roundtrip latency for Aegis and Ultrix was measured by ping-ponging a counter in a 60-byte UDP/IP packet 4096 times between two processes in user-space on DECstation5000/125s. The FRPC numbers are taken from the literature [49]. They were measured on a DECstation5000/200, which is approximately 1.2 times faster than a DECstation5000/125 on SPECint92.

The message processing at each node consisted of reading the 60-byte message, incrementing the counter, copying the new value and a precomputed message header into a transmission buffer, and then sending the reply message. In comparison to a complete application-level implementation, ASHs save 61 microseconds.

Despite being measured on a slower machine, ExOS/ASH is 81 microseconds faster than a high-performance implementation of RPC for Ultrix (FRPC) running on DECstation5000/200s and using a specialized transport protocol [49]. In fact, ExOS/ASH is only 6 microseconds slower than the lower bound for cross-machine communication on Ethernet, measured on DECstation5000/200s [49].

ASHs can be used to decouple latency-critical operations such as message reply from the scheduling of processes. To measure the impact of this decoupling on average message roundtrip latency we performed the same experiment as above while increasing the number of active processes on the receiving host (see Figure 2). With ASHs, the roundtrip latency stays constant. Without them, the latency increases, since the reply can be sent only when the application is scheduled. As the number of active processes increases, it becomes less likely that the process is scheduled. Since processes

Machine	lrpc	tlrpc
DEC2100	13.9	8.6
DEC3100	10.4	6.4
DEC5000	6.3	2.9

Table 12: Time to perform untrusted (**lrpc**) and trusted (**tlrpc**) LRPC extensions; times are in microseconds.

are scheduled in "round-robin" order, latency increases linearly. On Ultrix, the increase in latency was more erratic, ranging from .5 to 4.5 milliseconds with 10 active processes. While the exact rate that latency increases will vary depending on algorithm used to schedule processes, the implication is clear: decoupling actions such as message reception from scheduling of a process can dramatically improve performance.

7 Extensibility with ExOS

Library operating systems, which work 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 demonstrate the flexibility of the exokernel architecture by showing how fundamental operating system abstractions can be redefined by simply changing application-level libraries. We show that these extensions can have dramatic performance benefits. These different versions of ExOS can co-exist on the same machine and are fully protected by Aegis.

7.1 Extensible RPC

Most RPC systems do not trust the server to save and restore registers [27]. We implemented a version of **lrpc** (see Section 6.1) that trusts the server to save and restore callee-saved registers. We call this version **tlrpc** (trusted LRPC). Table 12 compares **tlrpc** to ExOS's more general IPC mechanism, **lrpc**, which saves all general-purpose callee-saved registers. Both implementations assume that only a single function is of interest (*e.g.,* neither uses the RPC number to index into a table) and do not check permissions. Both implementations are also single-threaded. The measurements show that this simple optimization can improve performance by up to a factor of two.

7.2 Extensible Page-table Structures

We made a new version of ExOS that supports inverted page tables. Applications that have a dense address space can use linear page tables, while applications with a sparse address space can use inverted ones. Table 13 shows the performance for this new version of ExOS. The inverted page-table trades the performance of modifying protection on memory regions for the performance of faster lookup. On the virtual memory benchmarks of Section 6.2, it is over a factor of two more efficient on **dirty**, 37% faster on **appel1**, and 17% faster on **appel2**. Because VM is implemented at application level, applications can make such tradeoffs as appropriate. This experiment emphasizes the degree of flexibility offered by an exokernel architecture.

7.3 Extensible Schedulers

Aegis includes a yield primitive to donate the remainder of a process' current time slice to another (specific) process. Applications can use this simple mechanism to implement their own scheduling algorithms. To demonstrate this, we have built an application-level scheduler that implements *stride scheduling* [54], a deterministic, proportional-share scheduling mechanism that improves on recent work [53]. The ExOS implementation maintains a list of processes for which it is responsible, along with the proportional share they are to receive of its time slice(s). On every time slice wakeup, the scheduler calculates which process is to be scheduled and yields to it directly.

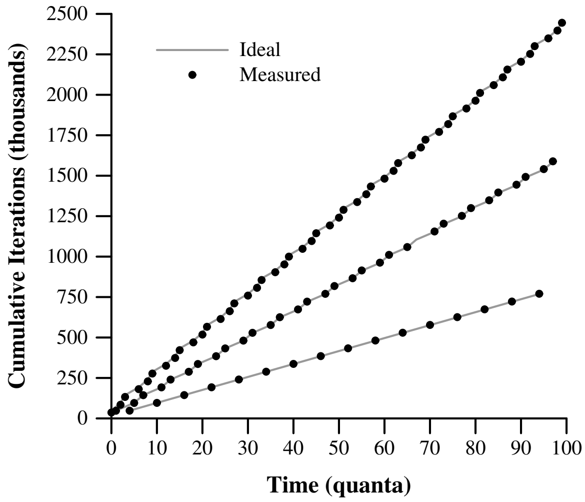

Figure 3: Application-level stride scheduler.

We measure the effectiveness of this scheduler by creating three processes that increment counters in shared memory. The processes are assigned a 3:2:1 relative allocation of the scheduler's time slice quanta. By plotting the cumulative values of the shared counters, we can determine how closely this scheduling allocation is realized. As can be seen in Figure 3, the achieved ratios are very close to idealized ones.

It is important to note that there is nothing special about this scheduler either in terms of privileges (*any* application can perform identical actions) or in its complexity (the entire implementation is less than 100 lines of code). As a result, any application can easily manage processes. An important use of such fine-grained control is to enhance the modularity of application design: previously, applications that had subtasks of different/fluctuating priorities had to internalize them in the form of schedulable threads. As a result, the likelihood of software errors increased, and the complexity of the design grew. By constructing a domain-specific scheduler, these applications can now effectively and accurately schedule sub-processes, greatly improving fault isolation and independence.

8 Related work

Many early operating system papers discussed the need for extendible, flexible kernels [32, 42]. Lampson's description of CAL-TSS [31] and Brinch Hansen's microkernel paper [24] are two classic rationales. Hydra was the most ambitious early system to have the separation of kernel policy and mechanism as one of its central tenets [55]. An exokernel takes the elimination of policy one step further by removing "mechanism" wherever possible. This process is motivated by the insight that mechanism *is* policy, albeit with one less layer of indirection. For instance, a page-table is a very detailed policy that controls how to translate, store and delete mappings and what actions to take on invalid addresses and accesses.

VM/370 [17] exports the ideal exokernel interface: the hardware interface. On top of this hardware interface, VM/370 supports a number of virtual machines on top of which radically different operating systems can be implemented. However, the important

Machine	Method	dirty	prot1	prot100	unprot100	trap	appel1	appel2
DEC2100	Original page-table	17.5	32.5	213.	275.	13.9	74.4	45.9
DEC2100	Inverted page-table	8.0	23.1	253.	325.	13.9	54.4	38.8
DEC3100	Original page-table	13.1	24.4	156.	206.	10.1	55.0	34.0
DEC3100	Inverted page-table	5.9	17.7	189.	243.	10.1	40.4	28.9

Table 13: Time to perform virtual memory operations on ExOS using two different page-table structures; times are in microseconds.

difference is that VM/370 provides this flexibility by *virtualizing* the entire base-machine. Since the base machine can be quite complicated, virtualization can be expensive and difficult. Often, this approach requires additional hardware support [23, 40]. Additionally, since much of the actual machine is intentionally hidden from application-level software, such software has little control over the *actual* resources and may manage the virtual resources in a counter-productive way. For instance, the LRU policy of pagers on top of the virtual machine can conflict with the paging strategy used by the virtual machine monitor [23]. In short, while a virtual machine can provide more control than many other operating systems, application performance can suffer and actual control is lacking in key areas.

Modern revisitations of microkernels have argued for kernel extensibility [2, 43, 48]. Like microkernels, exokernels are designed to increase extensibility. Unlike traditional microkernels, an exokernel pushes the kernel interface much closer to the hardware, which allows for greater flexibility. An exokernel allows application-level libraries to define virtual memory and IPC abstractions. In addition, the exokernel architecture attempts to avoid shared servers (especially trusted shared servers), since they often limit extensibility. 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 run in user-space. Some newer microkernels push the kernel interface closer to the hardware [34], obtaining better performance than previous microkernels. However, since these systems do not employ secure bindings, visible resource revocation, and abort protocols, they give less control of resources to application-level software.

The SPIN project is building a microkernel system that allows applications to make policy decisions [9] by safely downloading *extensions* into the kernel. Unlike SPIN, the focus in the exokernel architecture is to obtain flexibility and performance by securely exposing low-level hardware primitives rather than extending a traditional operating system in a secure way. Because the exokernel low-level primitives are simple compared to traditional kernel interfaces, they can be made very fast. Therefore, the exokernel has less use for kernel extensions.

Scout [25] and Vino [46] are other current extensible operating systems. These systems are just beginning to be constructed, so it is difficult to determine their relationship to exokernels in general and Aegis in particular.

SPACE is a "submicro-kernel" that provides only low-level kernel abstractions defined by the trap and architecture interface [41]. 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 have not yet been published.

Anderson [3] makes a clear argument for application-specific library operating systems and proposes 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 [13] provides a low-level kernel that can support multiple application-level operating systems. To the best of our knowledge ExOS and the Cache Kernel are the first general-purpose library operating systems implemented in a multiprogramming environment. 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. As result, it is biased towards a server-based system structure. For example, it supports only 16 "application-level" kernels concurrently.

9 Conclusion

In the exokernel architecture, an exokernel securely multiplexes available hardware resources among applications. Library operating systems, which work above the low-level exokernel interface, implement higher-level abstractions and can define special-purpose implementations that best meet the performance and functionality goals of applications. The exokernel architecture is motivated by a simple observation: the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions. To achieve a low-level interface, the exokernel separates management from protection. To make this separation efficient it uses secure bindings, implemented using hardware mechanisms, software caches, or downloading code.

Experiments using our Aegis and ExOS prototypes demonstrate our four hypotheses. First, the simplicity and limited number of exokernel primitives allows them to be implemented very efficiently. Measurements of Aegis show that its basic primitives are substantially more efficient than the general primitives provided by Ultrix. In addition, Aegis's performance is better than or on par with recent high-performance implementations of exceptions dispatch and control transfer primitives.

Second, because exokernel primitives are fast, low-level secure multiplexing of hardware resources can be implemented efficiently. For example, Aegis multiplexes resources such as the processor, memory, and the network more efficiently than state-of-the-art implementations.

Third, traditional operating system abstractions can be implemented efficiently at application level. For instance, ExOS's application-level VM and IPC primitives are much faster than Ultrix's corresponding primitives and than state-of-the-art implementations reported in the literature.

Fourth, applications can create special-purpose implementations of abstractions by merely modifying a library. We implemented several variations of fundamental operating system abstractions such as interprocess communication, virtual memory, and schedulers with substantial improvements in functionality and performance. Many of these variations would require substantial kernel alternations on today's systems.

Based on the results of these experiments, we conclude that the exokernel architecture is a viable structure for high-performance, extensible operating systems.

Acknowledgments

We thank Henri Bal, Robert Bedichek, Matthew Frank, Greg Ganger, Bob Gruber, Sandeep Gupta, Wilson Hsieh, Kirk Johnson, Butler Lampson, Ulana Legedza, Hank Levy (our shepherd), David Mosberger-Tang, Massimiliano Poletto, Robbert van Renesse, Satya (M. Satyanarayanan), Raymie Stata, Carl Waldspurger, and Deborah Wallach for insightful discussions and careful reading of earlier versions of this paper. We also thank the anonymous referees for their valuable feedback. We thank Jochen Lietdke for his aid in comparing the IPC mechanisms of Aegis and L3. We thank Ken Mackenzie for many insights and discussions. In addition, we thank Hector Briceno for porting NFS and SUN RPC; Sandeep Gupta for developing an inverted page-table; Robert Grimm for porting a disk driver; and Tom Pinckney for porting gdb and developing an application-level file system. Finally, we thank Deborah Wallach for her input on the design of ASHs, the software she developed to evaluate them, and porting Aegis to the DECstation5000. Her help was invaluable.

References

[1] M. B. Abbot and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

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

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

[4] 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 *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[5] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991.

[6] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, November 1994.

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

[8] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[9] 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*, December 1995.

[10] 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.

[11] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–308, November 1994.

[12] D. L. Chaum and R. S. Fabry. Implementing capability-based protection using encryption. Technical Report UCB/ERL M78/46, University of California at Berkeley, July 1978.

[13] 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.

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

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

[16] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990*, September 1990.

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

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

[19] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, pages 2–13, October 1994.

[20] D. R. Engler. VCODE: a very fast, retargetable, and extensible dynamic code generation substrate. Technical Memorandum MIT/LCS/TM534, MIT, July 1995.

[21] D. R. Engler, M. F. Kaashoek, and J. O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the Sixth SIGOPS European Workshop*, pages 62–67, September 1994.

[22] D. R. Engler, D. Wallach, and M. F. Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT, March 1995.

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

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

[25] 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.

[26] K. Harty and D.R. 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.

[27] 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.

[28] 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*, pages 39–51, May 1992.

[29] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-index caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[30] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for development of application-specific virtual memory management. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1993*, pages 48–64, October 1993.

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

[32] 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.

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

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

[35] K. Mackenzie, J. Kubiatowicz, A. Agarwal, and M. F. Kaashoek. FUGU: Implementing translation and protection in a multiuser, multimodel multiprocessor. Technical Memorandum MIT/LCS/TM503, MIT, October 1994.

[36] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 191–201, 1989.

[37] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[38] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design tradeoffs for software-managed TLBs. In *20th Annual International Symposium on Computer Architecture*, pages 27–38, May 1993.

[39] 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.

[40] G. J. Popek and C. S. Kline. The PDP-11 virtual machine architecture. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 97–105, November 1975.

[41] 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.

[42] 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.

[43] 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.

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

[45] R. L. Sites. Alpha AXP architecture. *Communications of the ACM*, 36(2), February 1993.

[46] C. Small and M. Seltzer. Vino: an integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard, 1994.

[47] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[48] 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.

[49] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[50] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 110–121, October 1994.

[51] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–267, May 1992.

[52] 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, December 1993.

[53] 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.

[54] C. A. Waldspurger and W. E. Weihl. Stride scheduling: deterministic proportional-share resource management. Technical Memorandum MIT/LCS/TM528, MIT, June 1995.

[55] 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.

[56] M. Yahara, B. Bershad, C. Maeda, and E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, 1994.