Aegis: A Secure Programmable Exokernel

Dawson R. Engler M. Frans Kaashoek James W. O'Toole Jr.

MIT Laboratory for Computer Science
Cambridge, MA 02139, USA

June 27 1994

Abstract

To provide high performance and a high degree of modularity, operating systems should provide only minimal embedded kernel functionality. The *Exokernel* is a new operating system structure that achieves this minimalist goal by letting applications manage the underlying hardware resources directly instead of hiding the hardware functionality in layers of operating system software. Applications can customize the operating system by extending exokernel interfaces. Exokernels can use code inspection, inlined cross-domain procedure calls, and secure languages to allow the safe execution of user code in the kernel. By having direct access to the physical hardware applications can implement policies that incorporate application-specific knowledge. To test and evaluate exokernels and their customization techniques a prototype exokernel system, *Aegis*, is being developed.

1 Introduction

An operating system is a virtual machine that abstracts physical resources. Existing operating systems do more than encapsulate resources: they also implement policies, which control the use of these resources. In addition, operating system implementors typically seize upon the hardware supervisor mode to define the interface between applications and the virtual machine. The fixed resource policies and the supervisor barrier prevent applications from obtaining the full functionality and performance of the hardware. In our view, the operating system should defer policy decisions to applications and provide a secure programmable machine interface, so that applications have direct access to the hardware and are able to set their own policies [15]. We are designing and implementing *Aegis*, a prototype system that will test the feasibility of this approach.

A monolithic operating system typically provides a single fixed policy for resource management. However, no single policy is best for all applications. For example, file system and relational database applications often have different storage access patterns, and therefore benefit from different buffer and disk management policies. Similar policy conflicts arise in almost every area of resource management [2, 13, 19, 42]. While microkernels have pushed some policy decisions out of the kernel into user-level processes, the supervisor mode still defines a fixed interface among the kernel, user-level services, and applications. Applications then suffer from a large number of protection domain crossings. These crossings are typically expensive, as they involve entering the supervisor mode, saving and restoring registers, scheduling a thread of control, and manipulating address spaces [3, 9, 38]. For example, many applications run slower on the microkernel implementation of Mach than on the monolithic implementation [3].

The disadvantages of fixed operating system policies are made worse by the growing diversity in hardware platforms and rapid advances of technology. For example, in a mobile computer it is of crucial importance to save power by carefully managing the spin-down policy of the disk. Using application-specific knowledge, spin-down policies can be much more effective than predictive policies or static ones [13, 33]. New architectures are arriving that support protected user-level communication, requiring little involvement of the operating system [8, 27]. By fixing policies, operating system software prevents applications from evolving with the hardware.

In this paper we propose a new operating system organization, the *exokernel*, that delivers hardware performance and hardware functionality directly to applications. An exokernel is a virtual machine that lets applications manage their own physical resources (for example through the use of libraries). By moving the management of resources out of the operating system kernel, an exokernel allows for a high degree of customization. We see three reasons for customization:

- Tailor policies to given hardware configurations, allowing applications to advance with the hardware.
- Tailor policies to given applications. For example, the interaction of application and the virtual memory system is crucial to the performance of applications. Closer integration with virtual memory management can enable distributed shared memory systems and garbage collectors to run an order of magnitude faster [4].
- Improve performance by reducing the overhead of interrupts, translation look-aside buffer (TLB) faults, traps, context switches, and cache misses, which are the primary sources of overhead in today's operating systems.

By lowering the kernel interface to calls that match the physical hardware resources, we expect the number of kernel crossings to be reduced, as most data structures and code will be living in the application address space. To minimize the performance impact of crossing the kernel boundary for privileged instructions and interrupts, and switching from one process to another process, the exokernel allows applications to run user code safely in the kernel. In this paper we discuss three techniques to make the exokernel design safe and efficient: (1) code inspection, (2) inlined cross-domain calls, and (3) type-safe languages.

The contributions of this paper are fourfold. First, we propose a new operating system organization that allows applications to control the hardware directly and allows applications to "program" the operating system kernel by downloading safe extensions into the kernel. Second, we have built a prototype exokernel system, *Aegis*, which runs on a MIPS R2000/R3000; it is based on an efficient, truly minimal kernel: it implements only five system calls, it supports the entering of another addressing context from within the kernel in approximately 20 MIPS assembly instructions, and it reflects all interrupts, exceptions, and traps to the application in approximately 9 MIPS assembly instructions. Third, we introduce an efficient variant of sandboxing [45] in which only unrestricted jumps and privileged instructions have to be controlled instead of all indirect stores and loads. Fourth, we show how portable dynamic runtime code generation can be used in order to improve the performance of safe user code that is imported into the kernel; we implemented Mogul's packet-filter language [35], compared its performance to an efficient byte-code interpreter, and measured a factor of 10 performance improvement. All four of our contributions are preliminary, as we do not yet have fully integrated system that supports all of them in a single system.

The outline of the rest of the paper is as follows. Section 2 discusses the design of a prototype exokernel, *Aegis*. Section 3 discusses our proposal for a secure programmable machine and the implementation techniques that permit safe customization across the supervisor mode barrier. Section 4 reports on an implementation of the *Aegis* system. In Section 5 we briefly discuss related work and in Section 6 we conclude.

2 The *Aegis* Exokernel

We have designed a prototype exokernel, *Aegis*, to test the feasibility of a system that provides minimal resource allocation, arbitration services, and a programmable interface. In designing *Aegis*, we strictly limited the functionality that is embedded and non-customizable. Many of the primitives supplied by other "minimalistic" kernels are viewed as applications by the exokernel. For example, the *Aegis* kernel does not include page-table management, a file-system, remote procedure call, nor device drivers.

To allow efficient checking we have chosen a capability-based approach for our prototype. As in Amoeba, *Aegis* capabilities are large random numbers [36]. In a capability-based system specific resources can be awarded to individual applications. Capabilities enable individual applications to efficiently manipulate the resources they own. The use of probabilistic capabilities also enables efficient revocation of resource access.

The design so far has been focused on the minimal set of abstractions that enable applications to share the available physical hardware. Although it seems difficult to predict how hardware trends will affect exokernel design, we do not expect high-level services such as network protocol implementations or file system management to be exokernel primitives. In the current prototype for a DECstation, *Aegis* reflects the following hardware abstractions in its interface: physical memory pages, the TLB, processors, addressing context identifiers, and interrupt/trap events. In the near future we will also add direct memory access (DMA) channels and I/O devices.

Aegis currently supports five system calls: *alloc*, *dealloc*, *restrict*, *yield*, and *enter*. *Alloc* allocates a physical resource (e.g., a time slice, TLB entry, physical memory page, or addressing context identifier) and returns a capability. *Dealloc* releases a physical resource. *Restrict* restricts a capability. *Yield* allows another user to use the resource temporarily. *Enter* is a primitive call to enter another *environment* (i.e., an addressing context) through a controlled entry point, called a *gate*. As our design progresses, we will only add more systems calls if absolutely needed.

The entity of accountability in the *Aegis* kernel is a *domain*, which consists of one or more *environments* and their associated physical resources (e.g., context identifier, physical pages, and time slices). For example, a database package may be a single domain with multiple environments, each running a module of the package. The environments are fault isolated from each other. In addition to the enter mechanism to switch control between environments in different domains, cooperating environments in the same domain can pass control by just changing their context identifiers.

A low-level interface allows applications to set their own policies. Such local policy decisions may conflict with more effective global policies. For example, two applications may make conflicting policies for a magnetic disk with a single arm. For such a device a global policy is often more effective than application-specific policies. We are exploring this tradeoff between application independence and global system performance.

3 A Secure Programmable Machine

Aegis provides a programmable virtual machine that permits applications to safely and efficiently manage system resources. *Aegis* executes in supervisor mode. Our design protects the secure machine against unsafe environments by using traditional methods: unsafe application code executes in separate addressing contexts.

In addition, to permit efficient management of system resources, our design permits controlled execution of user code in the kernel. This makes the secure machine programmable and enables safe, efficient customization of the operating system. However, only safe application code may execute privileged instructions and may execute in the kernel. We are examining several implementation techniques that permit safe customization across the supervisor mode barrier. These techniques ensure that safe application code will not execute privileged instructions or access protected data in uncontrolled ways:

1. **Code Inspection.** Code introduced into the secure programmable machine must be free of privileged instructions. As shown by Deutsch [12], machine code can be inspected to guard against wild loads, stores, and jumps. Execution time can also be controlled by bounding loop iteration counts. With sufficiently stringent restrictions on the application code (or by augmenting the application code with checks), it can be safely executed in supervisor mode, and primitives of the secure machine that use privileged instructions can be inlined into the inspected code.

2. **Inlined cross-domain calls.** Code inspection that prohibits indirect loads and stores will be too restrictive for many application purposes. Wahbe [45] shows that sandboxing can be

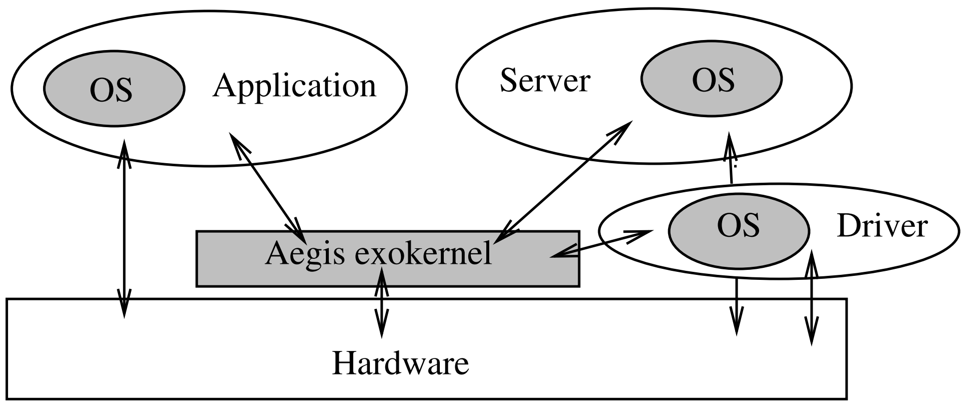

Figure 1. Software structure of the *Aegis* system. Each application manages its own physical resources by containing operating system (OS) code. A small exokernel immediately reflects interrupts, exceptions, and traps back to the application. Applications can "program" the kernel interface by loading safe pieces of program into the kernel. (We do not have a file system or a device driver yet.)

used to restrict the domain of indirect memory operations, but with an overhead of several instructions per guarded operation. The motivation behind sandboxing is to simulate address-space protection domains when cross-domain calls are expensive. We propose a variant of sandboxing that allows user code to execute in supervisor mode, but in a restricted addressing context. This scheme eliminates the need to control all indirect loads and stores. Furthermore, because *Aegis* hands out context identifiers and allows cooperating environments to alter the context identifier register directly, switching among these environments can be done in only a few instructions. Like *Aegis* they can even safely download foreign application code in their environment.

3. **Type-safe language.** If applications need to customize the operating system kernel with operations that require frequent and intensive manipulation of shared protected data structures, then page-granularity protections may not suffice. The use of a type-safe language and a trusted compiler might enable closer integration of application and kernel code, while simultaneously increasing portability. Runtime code generation and other modern compilation techniques enable the kernel to merge related pieces of code that are downloaded by applications.

We expect each of these techniques to be useful for exokernel extensions, depending on the tradeoff involved.

4 Implementation of the *Aegis* system

We have implemented a prototype of the *Aegis* system on a MIPS R2000; Figure 1 depicts its structure. In addition, we experimented with two customization techniques: one based on safe languages and runtime code generation, and another based on inlined cross-domain calls.

The *Aegis* kernel is an efficient, truly minimal kernel. It does not implement page table management, file systems, device drivers, nor remote procedure call; all of these are (or will be) implemented by applications. It supports only the five system calls described in Section 2. The implementation of *alloc*, *dealloc*, *restrict*, and *yield* are straightforward. Enter allows communication between different *Aegis* environments (addressing contexts) by making calls to *gates* (i.e., entry points) in an another environment. Each environment supports one gate. A environment calling *enter* donates the rest of its time slice to the called environment. *Enter* implements the bare minimum to allow control to pass from one environment to another environment. For example, it does not save and restore registers. Aegis can pass control from one environment to another environment in only 20 MIPS assembly instructions. A very efficient RPC from client to server can be built if the client trusts the server to save and restore registers it uses [21]; traditionally RPC systems save and restore at least all general purpose registers. On the R2000 on which *Aegis* runs, this costs approximately 64 load and store instructions; as memory speeds are not scaling with processor performance and processor is increasing, this overhead will become increasingly expensive.

An *Aegis* environment is a kernel data structure that consists of an exception handling vector, a context identifier, and pointers to an application page table, a gate, an epilogue, and a prologue. The prologue and epilogue code allows applications to save and restore registers under their own control, and allows them to customize context switches. For example, this allows an application to implement Bershad's fast uniprocessor synchronization [7] by moving the program counter out of a critical section and releasing locks on a forced context switch. When *Aegis* schedules another environment, it calls the epilogue code of the running environment, installs pointers to the dispatch table, page table, and context identifier of the new environment, and then calls the new environment's prologue code. As even saving and restoring of registers is under application control, applications can reduce context switching overhead by saving the minimal number of registers; for example, applications can use their knowledge of active registers to decide which ones to save and restore.

Aegis supports context identifiers; address spaces and mappings from virtual to physical are implemented by applications. This organization allows each application to choose its own page table layout. For example, an object-oriented system with many small objects may implement each object in a small address space with a simple page table, while a traditional UNIX system may implement a multilevel page table hierarchy. In addition, applications have more control over TLB management, which allows applications to prefetch their own TLB entries, resulting in better overall performance [5]. The synthetic applications we are currently running all use a very simple page table layout.

All interrupts (except for timer interrupts), exceptions, and traps are reflected back through the environment's dispatch table to the applications in 9 MIPS assembly instructions, allowing each application to customize its usage of the physical resources. For example, garbage collectors and distributed shared memory systems can effectively be integrated with the virtual memory systems, which is also under application control. Thekkath has also recently advocated and exploited the usage of such inexpensive user-level traps [44]. *Aegis* can support efficient emulation of multiple operating system APIs (e.g., BSD4.3, Win32) by tailoring the usage of physical resources to a specific system and by taking advantage of inexpensive user-level traps.

Timer interrupts are the only physical resource that is managed by *Aegis* directly to allow time sharing of the processor. At a timer interrupt *Aegis* can also take back physical resources from badly behaving applications. For example, to revoke physical memory that has been allocated to a given application, Aegis flushes translations to the allocated physical memory out from the TLB. *Due to space restrictions we restricted our discussion of revocation of physical resources.*

We are investigating multiple ways of safe customization: one using code inspection and inlined cross-domain calls, and two language-based approaches. We have experimented with an efficient code inspection scheme that only needs to control jumps and privileged instructions (instead of jumps and indirect load and stores as in sandboxing [45]) that takes advantage of the architecture of the MIPS R2000/R3000. The R2000/R3000 can be programmed in such way that user programs can execute privileged instructions by writing and reading coprocessor 0 [24]. Using this feature, applications can run controlled privileged instructions in user space while maintaining fault isolation among different environments. This scheme will work on any architecture that decouples addressing contexts from the ability to execute privileged instructions.

We also have experimented with two language-based approaches to safe customization. Using portable runtime code generation [16] we implemented Mogul's safe packet-filter language and compared its performance to an efficient byte-code interpreter of the packet filter. We measured a factor of 10 performance improvement. In addition, we have configured a small Scheme48 [25] byte-code interpreter so that it can run inside the Aegis exokernel, allowing us to experiment with executing safely arbitrary user code in the kernel. *We have no results yet on the scheme48 interpreter, but we will report on them in the full paper.*

5 Related Work

Many early operating systems papers discussed the need for extensible kernels [18, 23, 29, 30, 46, 47]. Lampson's description of the CAL-TSS [28] and Brinch Hansen's microkernel paper [17] are two classic rationales. Hydra was the most ambitious system that had the separation of kernel policy and mechanism as one of its central tenets [47, 32]. Modern revisitations of microkernels have also argued for kernel extensibility [1, 10, 11, 39, 41, 43]. Anderson and Kiczales et al. also recently argued for minimalism and customizability [2, 26]. Two important differences between our work and previous approaches are that: (1) the fixed kernel interface in these systems is a high-level one (e.g., page-tables are implemented by the kernel) and (2) customization carries a heavy performance penalty since it is usually accomplished through the addition of servers that interact with the kernel via IPC.

To combat this performance penalty, current microkernels allow reintroduction of code into the kernel: the QNX operating system allows user-level handlers for device I/O [20]; Chorus servers are loaded into the kernel [41]; Mach 3.0 has migrated some of the AFS cache manager back into the kernel [38]; the Amoeba file server is usually configured to run in the kernel [43]; Windows/NT runs only non-critical services outside of the kernel [11]; Lepreau *et. al.* has provided a tool to reintroduce servers into Mach 3.0 [31]. Draves has argued for selecting among several implementations of a specific kernel abstraction to allow customization [14]. However, none of these customizations guarantee the safety of the code once introduced.

Synthesis is an innovative operating system that has inspired some of our work [34]; for example, the use of runtime code generation. The *Aegis* system, however, uses a portable runtime code generation system [16], while the synthesis kernel is written in assembly, and uses ad-hoc runtime code generation macros to specialize system calls. Furthermore, the *Aegis* system will use runtime code generation to efficiently execute application kernel extensions.

Concurrently with our work the SPIN project also investigates adaptable kernels that allow applications to make policy decisions efficiently [6]. The SPIN system encapsulates policies in *spindles* that can be dynamically loaded into the kernel. To ensure safety, spindles will be written in a type-secure language and will be translated by a trusted compiler. In contrast, we anticipate using a variety of customization techniques that permit safe execution of small pieces of application code within the exokernel, some of which will allow the kernel interface to be extended using existing application code. In addition, the exokernel aggressively moves code out of the operating system kernel into user-space; its kernel interface provides a much low-level interface than microkernels do. The SPIN project, on the other hand, concentrates more on moving user code into a microkernel.

Like exokernels the Scout project intends to use modern compiler and language technology to improve operating systems. Scout mainly intends to use this technology to optimize critical paths in a communication kernel that is based on their high-performance x-kernel architecture [22]. Exokernels are employ compiler technology both for improving performance and controlling the execution of user code in the kernel.

Techniques for certifying the safety of code include inspection and sandboxing. Deutsch showed how to safely introduce user-written instrumentation code into a running kernel by bounding loads, stores, jumps and runtime [12]. Making existing binaries safe has been explored by Wahbe [45]. Although this technique will be useful for controlling entry points, we expect that loads and stores can be constrained more efficiently by changing the address context quickly within supervisor mode.

The use of a type-safe language to allow user applications to augment the operating system has been explored in a personal workstation environment [40]. Myers argued for using a type safe language to load application code into the object manager in a persistent object system [37].

In summary, the *Aegis* system builds on a rich tradition of operating system and language technology. *Aegis* extends it in two key ways: (1) it makes hardware resources directly available to the application; (2) it award treats application kernel extensions as first class objects by safely incorporating them into the operating system. This organization allows aggressive user customization and subsequently high performance and flexibility.

6 Conclusions

Operating systems must shed the burden of their traditional policy-laden services. We suggest exokernels as an new minimal operating system design. To minimize the overhead of protection domain crossings we proposed three customization techniques. We reported on an experimental exokernel, *Aegis*. *Aegis* currently supports five system calls that allow applications to control the available physical hardware directly.

Exokernels allows unprecedented control of the physical hardware resources. We expect that an exokernel will be an important technology to allow software to scale from small hand-held devices to powerful general-purpose workstations and high-performance supercomputers.

References

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

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

[3] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The interaction of architecture and operating system design. In *Proc. of the Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

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

[5] K. Bala, M.F. Kaashoek, and W.E. Weihl. Software prefetching and caching for translation lookaside buffers. In *OSDI (submitted)*, June 1994.

[6] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN - an extensible microkernel for application-specific operating system services. Technical Report TR94-03-03, University of Washington, February 1994.

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

[8] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *The 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[9] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.

[10] D.R. Cheriton. The V distributed system. *Commun. ACM*, 31(3):314–333, March 1988.

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

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

[13] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power hungry disk. In *Proceedings of 1994 Winter Usenix conference*, pages 293–306, January 1994.

[14] R. Draves. The case for run-time replaceable kernel modules. In *Fourth Workshop on Workstation Operating Systems*, pages 160–165, October 1993.

[15] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. The operating system kernel as a secure programmable machine. *Proceedings of the Sixth SIGOPS European Workshop*, September 1994.

[16] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. October 1994.

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

[18] M.C. Harrison and J.T. Schwartz. SHARER, a time sharing system for the CDC 6600. *Communications of the ACM*, 10(10):659–665, October 1967.

[19] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. *Proc. of the Fifth Conf. on Architectural Support for Programming languages and Operating Systems*, pages 187–199, October 1992.

[20] D. Hildebrand. An architectural overview of QNX. *Proc. Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.

[21] 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.

[22] N.C. Hutchinson and L.L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Trans. on Soft. Eng.*, 17(1), Jan. 1991.

[23] D.H.R. Huxtable and M.T. Warwick. Dynamic supervisors — their design and construction. *Proceedings of the First ACM Symposium on Operating Systems Principles*, 1967.

[24] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[25] Richard A. Kelsey and Jonathan A. Rees. *Tractable Scheme Implementation*. Kluwer.

[26] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The need for customizable operating systems. In *Fourth Workshop on Workstation Operating Systems*, pages 165–170, October 1993.

[27] J. Kuskin et al. The Stanford FLASH multiprocessor. *The 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

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

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

[30] B.W. Lampson and H.E. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, May 1976.

[31] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeff Law. In-kernel servers in Mach 3.0: implementation and performance. *Proc. of the Third Usenix Mach Symposium*, 1993.

[32] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism seperation in HYDRA. *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[33] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A quantative analysis of disk drive power management in portable computers. In *Proceedings of 1994 Winter Usenix conference*, pages 279–291, January 1994.

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

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

[36] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[37] A.C. Myers. Resolving the integrity/performance conflict. In *Fourth Workshop on Workstation Operating Systems*, pages 156–160, October 1993.

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

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

[40] 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.

[41] 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.

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

[43] 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.

[44] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (to appear)*, October 1994.

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

[46] B.A. Wichmann. A modular operating system. *Proc. IFIP Cong. 1968*, 1968.

[47] 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.