# The Case for Application-Specific Operating Systems

Thomas E. Anderson

Division of Computer Science
University of California, Berkeley
Berkeley, CA 94720

Computer systems of today have the potential for vastly improved application performance. Advances in hardware technology have led to systems with more, faster processors, higher bandwidth networks, and larger amounts of primary, secondary, and tertiary storage. Relative to systems of only a decade ago, improvements of at least two orders of magnitude have occurred in each of these areas, and it appears likely that the next decade will see equally large improvements.

Recent evidence suggests that these trends will require re-thinking the traditional role of operating systems. Increasingly, applications programmers and compiler writers have found that achieving the performance potential of modern computer systems requires control over the physical resources of the machine. Traditional operating systems, however, invisibly manage physical resources on behalf of applications. As a result, some have suggested the only way to obtain good application performance is to simply "turn off" the operating system [Agarwal et al. 90, Black 90], although this loses the advantages that led to running operating systems on top of bare hardware in the first place. As one example, database systems have long been forced to re-implement major parts of the operating system, such as threads, memory management, and I/O, in order to gain control over the machine's physical resources. High performance architectures and algorithms exacerbate both the problems with traditional operating systems and the potential benefits of appropriate operating system support.

The challenge to operating systems designers, then, is to deliver to applications the performance available now only from dedicated hardware, combined with the ease of sharing resources and data among multiple applications and the simpler programming model found in general-purpose operating systems.

The operating systems community has spent much of the last few years debating the value of "monolithic" and "small-kernel" operating system structures — whether operating system modules should be included in the kernel or separated into distinguished application-level servers. I believe this debate will be largely irrelevant to future computer systems, since both manage physical resources behind the back of application software.

Instead, I propose an *application-specific* structure where as much of the operating system as possible is pushed into runtime library routines linked in with each application. The operating system kernel is stripped to its bare minimum functionality: at a minimum, the kernel must adjudicate among application requests for physical resources and it must enforce hardware protection boundaries between applications. Everything else, including resource management and communication/sharing between applications, can and should be done by operating system code running as library routines in each application. The key is that the operating system must notify each application of changes in its resource allocation, to allow the application the chance to adapt to make best use of whatever resources are available to it.

Pushing operating system functionality into each application need not necessarily complicate the task of the application programmer. The runtime system linked into each application can provide the programmer with the same interface now provided by a traditional operating system; however, because these library routines can be made application-specific, the programmer has the flexibility to easily modify them whenever that is necessary for performance.

As one example, parallel applications are typically written and compiled assuming the number of processors running the program does not change during its execution. This prevents long running jobs from sharing processors with interactive jobs, since time-slicing behind the back of a parallel application often yields terrible performance. It also prevents periods of low parallelism in one job from being overlapped with periods of high parallelism in another. Node failures are another reason for changes in the number of processors available to an application; a fully configured Thinking Machines CM-5 is expected to have a mean time between failures of only one hour. At least on shared-memory multiprocessors, though, processor and thread management can be pushed to the user level, improving performance and flexibility with no loss in functionality [Anderson et al. 92].

As another example, many supercomputer applications are as performance-limited by memory size and I/O bandwidth as by processor cycles. Pre-fetching is a promising way of hiding memory and I/O latency, but performance can be drastically reduced if there is insufficient buffer space to hold all of the pre-fetched material, for instance, if more than one application is pre-fetching at the same time. This is another case where application knowledge of dynamic resource allocation could lead to better performance.

As a third example, there has been recent interest in network file systems that stripe files across multiple servers to obtain higher bandwidth. The traditional approach would be to stripe files transparently to the user; an application may be able to achieve better performance, however, if it knows how its data is stored, so that it can co-locate pieces of its computation near the data being used.

Several recent research efforts have made an argument for pushing certain pieces of the operating system into each application [Young et al. 87, Bershad et al. 91, Anderson et al. 92]. But these arguments have each focused on only an isolated piece of the operating system. Rather, I believe that the same argument can be made for *every* physical resource managed by a traditional operating system, be it processors, memory, the file cache, network bandwidth, or secondary/tertiary storage, and for *every* medium of sharing between applications, be it local RPC, remote RPC, or the file system. The goal: major improvements in application performance on high performance computer systems.

# References


[Agarwal et al. 90] Agarwal, A., Lim, B.-H., Kranz, D., and Kubiatowicz, J. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 104–114, May 1990.

[Anderson et al. 92] Anderson, T., Bershad, B., Lazowska, E., and Levy, H. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems 10*(1), February 1992.

[Bershad et al. 91] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[Black 90] Black, D. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer Magazine*, 23(5):35–43, May 1990.

[Young et al. 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.