This page intentionally left blank.

The Interaction of Architecture and Operating System Design

Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Today's high-performance RISC microprocessors have been highly tuned for integer and floating point application performance. These architectures have paid less attention to operating system requirements. At the same time, new operating system designs often have overlooked modern architectural trends which may unavoidably change the relative cost of certain primitive operations. The result is that operating system performance is well below application code performance on contemporary RISCs.

This paper examines recent directions in computer architecture and operating systems, and the implications of changes in each domain for the other. The requirements of three components of operating system design are discussed in detail: interprocess communication, virtual memory, and thread management. For each component, we relate operating system functional and performance needs to the mechanisms available on commercial RISC architectures such as the MIPS R2000 and R3000, Sun SPARC, IBM RS6000, Motorola 88000, and Intel i860.

Our analysis reveals a number of specific reasons why the performance of operating system primitives on RISCs has not scaled with integer performance. In addition, we identify areas in which architectures could better (and cost-effectively) accommodate operating system needs, and areas in which operating system design could accommodate certain necessary characteristics of cost-effective high-performance microprocessors.

This work was supported in part by the National Science Foundation under Grants No. CCR-8703049, CCR-8619663, and CCR-8907666, by the Washington Technology Center, by the Digital Equipment Corporation Systems Research Center and External Research Program, and by IBM and AT&T Fellowships. Bershad is now with the School of Computer Science, Carnegie Mellon University.

1 Introduction

The last decade has seen substantial change in computer architecture. This change has resulted primarily from the RISC/CISC discussions of the early 1980s [Patterson & Ditzel 80, Patterson & Sequin 82, Hennessy et al. 82, Radin 82], along with new hardware technology that has both enabled and required new architectural trends. Among the most important of these trends are (1) a move towards simple, directly-executed instruction sets, (2) an open-architecture philosophy that makes implementation visible to higher-level software, (3) the migration of function from hardware to software, and (4) a performance-oriented approach in which architectural features are removed unless they can be justified on a cost/performance basis.

Over the same period, operating systems too have seen an evolution. Operating system functions have changed to meet new requirements: fast local communication, distributed programming, parallel programming, virtual memory, and others. For improved extensibility, maintainability, and fault tolerance, modern operating systems such as Mach [Jones & Rashid 86] are moving from the traditional monolithic structure to a more modular organization. These requirements have led to much research in new underlying structures. Thus, while the Unix system interface has become standard in engineering and scientific computing, future Unix systems are unlikely to resemble current Unix implementations at lower levels. Newer operating systems will simply support Unix as one of several available interfaces, as is done, for example, on V [Cheriton et al. 90], Mach [Golub et al. 90], and Topaz [Thacker et al. 88].

Unfortunately, modern operating systems and architectures have evolved somewhat independently:

- While simulation and measurement studies (such as [Katevenis 85]) have been used to guide hardware design tradeoffs, these studies have tended to overlook the operating system. The problem is partly technological: most early program tracing tools were unable to trace operating system code. But the amount of information overlooked can be huge. In trace-driven studies gathered through a microcode-based tool, Agarwal et al. found that during the execution of two VAX Ultrix workloads, over 50% of the references were system references [Agarwal et al. 88]; worse, this study and others (such as [Clark & Emer 85]) have shown operating system behavior to differ significantly from application behavior. Thus, the result of ignoring such a large execution component could be dramatic.

- In those modern architectures where the needs of operating systems have been carefully considered, "traditional Unix" has driven the design. But, as noted, a variety of new requirements are changing the design of operating systems in ways that make "traditional Unix," and applications built on top of it, inappropriate benchmarks.
- Operating system research has strongly focused on performance. In many cases, the performance of operating system facilities has been optimized to where it meets hardware limits [Schroeder & Burrows 90, Bershad et al. 90a, van Renesse et al. 88, Massalin & Pu 89], leaving hardware, not software, as the bottleneck. Yet while these facilities make optimal use of hardware, they often are predicated on the performance characteristics of earlier processors. They may not exploit the strengths or minimize the weaknesses of newer architectures, and may not perform adequately in the future, given current architectural trends.
- While some high-level operating system functions have been heavily optimized, software implementors may have overlooked the importance of low-level functions. In particular, some functions that were formerly optimized through microcode must now be carefully implemented in software for the system to perform effectively.

This paper examines recent changes and directions in computer architecture and operating systems, and the implications of changes in each domain on the other. Ousterhout has already demonstrated that operating systems are not receiving the same benefit as applications from new hardware platforms [Ousterhout 90b]. He attributes this principally to the operating system's appetite for memory bandwidth and disk performance. In this paper we look at a different granularity, focusing instead on the relationship between essential operating system primitives and hardware support. Relative to previous-generation machines, new RISC-oriented architectures have added and removed features; both the additions and the deletions have affected the performance of operating system primitives. At the same time, relative to previous-generation operating systems, newer operating systems have added features that may require (for acceptable performance) hardware support that no longer exists, and have failed to optimize certain functions that have recently become more important.

1.1 Motivation

To motivate our study, we compared the performance of several modern microprocessors executing the following primitive operating system functions:

- Null system call — the time for a user program to enter a null C procedure in the kernel, with interrupts (re-)enabled, and then return.
- Trap — the time for a user program to take a data access fault (e.g., on a protection violation), vector to a null C procedure in the kernel, and return to the user program. This requires saving and restoring any registers that are not preserved across procedure calls.
- Page table entry change — the time, once in the kernel, to convert a virtual address into its corresponding page table entry, update that entry to change protection information, and then update any hardware (e.g., the translation buffer) that caches this information.
- Context switch — the time, once in the kernel, to save one process context and resume another, including the time to change address spaces in the hardware. This does not include the time to find another process to run.

Our measurements examined a CISC implementation, the VAXstation 3200 (11.1 MHz CVAX [Leonard 87]), and four RISC implementations: the Tektronix XD88/01 (20 MHz Motorola 88000 [Mot 88a, Mot 88b]), DECstation 3100 (16.6 MHz MIPS R2000 [Kane 87]), DECstation 5000/200 (25 MHz MIPS R3000[1]), and SPARCstation 1+ (25 MHz Sun SPARC [Sun 87, Cyp 90]). For brevity, our tables list the architecture or microprocessor names, rather than the system names, although performance is of course affected not only by instruction set architecture and processor technology, but by attributes specific to particular system-level implementation choices, such as cache size and organization.

For software, we began with the vendor-supplied Unix handlers shipped with these systems. Our objective was to isolate the architectural impact on these operating system functions. Therefore, we attempted to restructure the drivers in order to remove operating system dependencies and measure only the operating system independent parts; this often reduced the execution times by a factor of two from the vendor-supplied versions. We further optimized where possible (e.g., removing extraneous procedure calls) subject to the rules that our resultant code would still be able to boot Unix and would still maintain the standard conventions for register usage between the compiler, operating system, and applications. The resulting handlers were almost entirely written in assembler.

The system call time was measured directly by repeated calls from user level to an otherwise unused system call code. The time to change a page table entry (PTE) and to context switch was measured by writing special system calls, and then subtracting the system call time from the measured time. For the trap time, we repeatedly called the system call to unmap a page from the test program's address space, then referenced the unmapped page from user-level, and within the trap handler re-mapped the page. The trap time is this measurement minus the system call, unmap, and remap times. For all measurements, the operating system was allowed to assume that a purely integer application was running (e.g., that there were no floating point operations in progress and that floating point registers did not need to be saved on context switches). The performance of some of the RISC architectures relative to the CVAX would be worse without this assumption; we consider this effect in Sections 3 and 4.

[1] The MIPS R3000 uses the same instruction set as the R2000.

Operation	Time (μseconds) DEC CVAX	Motorola 88000	MIPS R2000	MIPS R3000	Sun SPARC	Relative Speed (RISC/CVAX) 88000	R2000	R3000	SPARC
Null system call	15.8	11.8	9.0	4.1	15.2	1.3	1.8	3.9	1.0
Trap	23.1	14.4	15.4	5.2	17.1	1.6	1.5	4.4	1.4
Page table entry change	8.8	3.9	3.1	2.0	2.7	2.3	2.8	4.4	3.3
Context switch	28.3	22.8	14.8	7.4	53.9	1.2	1.9	3.8	.5
Application Performance						3.5	4.2	6.7	4.3

Table 1: Relative Performance of Primitive OS Functions

Our measurements, presented in Table 1, show that while the application code performance of the RISCs is excellent relative to the CVAX (we use the SPECmark [SPEC 90] as a measure of application performance), the performance of these operating system functions has not scaled in a commensurate way.

In the following sections we will discuss why these primitive functions are important, and why they have not received the expected performance gain. In part, we will show that current architectures have made the implementation of these functions more difficult, and therefore performance is harder to achieve. To illustrate this point, Table 2 shows a count of the number of instructions executed along the shortest path in our drivers used for Table 1 (along with a similar estimate for the Intel i860 [Int 89]). These numbers are meant only to give a *qualitative* indication of the added complexity; obviously some VAX instructions, such as those used for context switching, do large amounts of work in microcode. However, some may still be surprised by the order of magnitude difference in the number of instructions needed in some cases by the RISCs relative to the VAX, and by the variation in instruction counts even among the RISCs.

Our objective in implementing drivers for Tables 1 and 2 was to remove operating system dependencies and to do reasonable optimizations in a way that was equitable to all of the architectures. We do not claim that our driver implementations are optimal. With some effort, we are certain that the performance of all of these functions on all of the architectures could be further improved. In the past, these functions have typically been implemented and highly-optimized through the use of microcode. Implementors of current RISC operating systems may not have paid as much attention to performance as previous microprogrammers, or may not yet have as much experience optimizing code for their architectures and implementations.

1.2 Organization of the Paper

This paper is organized around the characteristics of three major components of operating system design. In each case we first discuss the functional and performance needs of the operating system component, and then describe how it can be facilitated or inhibited by particular architectural features. Section 2 describes interprocess communication mechanisms; architecturally, it concentrates on the system call and interrupt processing facility of modern microprocessors. Section 3 describes virtual memory systems and the functions they provide; it focuses on page fault handling as well as page table and translation buffer management. Section 4 describes lightweight thread support in multiprocessor operating systems; it examines the management of processor state in run-time thread systems. In each section we select architectural examples from contemporary microprocessor architectures.

Finally, Section 5 presents frequency measurements to show why the performance of operating system primitives is important to application performance. Section 6 summarizes our results.

2 Interprocess Communication

Interprocess communication is central to modern operating system structure and performance. Operating systems have evolved from monolithic, centralized kernels to a more decentralized structure [Baskett et al. 75, Rashid & Robertson 81, Cheriton 84, Jones & Rashid 86]. The reasons for this evolution are two-fold. First, by structuring the operating system as independent address spaces communicating through messages, modularity, fault tolerance, and extensibility are enhanced. Second, using messages rather than shared memory as the principal communication mechanism simplifies the move to a physically distributed topology.

While monolithic Unix systems are still the basis for much of today's computing, newer implementations such as Mach are moving to a "small-kernel" or "kernelized" structure, in which many operating system components are implemented as servers outside of the kernel. These servers communicate with users, with the kernel, and with each other through message passing. As operating system structure continues to decentralize, communication performance becomes crucial to overall system performance. In fact, good communication performance is key to enabling this structure; poor communication performance is likely to guarantee maintenance of the status quo, i.e., monolithic design.

Communication performance also is key to effective use of contemporary distributed computing environments. Although many modern processors are designed as standalone workstations, the need for data and device sharing, the potential of networks of computers to be used as multiprocessors, and the growing computational power of new processors has led to increasing use of file and computation servers in

Operation	CVAX	88000	R2/3000	SPARC	i860
Null system call	12	122	84	128	86
Trap	14	156	103	145	155
Page table entry change	11	24	36	15	559
Context switch	9	98	135	326	618

Table 2: Instructions Executed for Primitive OS Functions

Function	Time
Network transfer time	17%
Interrupt processing	30%
Wakeup receiving thread	17%
Stub processing	29%
Checksums	7%

Table 3: RPC Processing Time in SRC RPC

networks. This use is predicated on the ability of processors to communicate efficiently across networks. In fact, in many environments, truly stand-alone computers no longer exist.

2.1 Cross-Machine Communication

To simplify communication for the programmer, most systems now support Remote Procedure Call (RPC) [Birrell & Nelson 84]. RPC has become the preferred method to communicate between address spaces, both on the same machine and across a network, because it encapsulates message-oriented communication in procedure call semantics. In an RPC system, clients make procedure calls to remote servers. The client's call actually is made to an automatically-generated stub procedure that is linked into the client. The stub is responsible for packaging the procedure's parameters in a message packet, and transmitting that packet to a similar stub linked with the server. The server stub receives the message, removes the parameters, and makes a procedure call into the server. The return travels back in the opposite direction, with the server stub sending the results in a message packet to the client stub.

Surprisingly, communication performance typically is not limited by network speed. Because cross-machine RPC involves communication between two remote address spaces, the operating system must be involved to transfer control and data between the client and server. This operating system involvement adds overhead that usually dominates network latency. For example, Table 3 shows the distribution of time spent in a round-trip cross-machine null RPC for a small (74-byte) packet in SRC RPC [Schroeder & Burrows 90] executing on μVAX-II Firefly multiprocessors [Thacker et al. 88] connected by an Ethernet. For SRC RPC, one of the fastest RPC systems, only 17% of the time for a small packet is spent on the wire.

Given the significant CPU requirements of cross-machine RPC, one would expect a substantial RPC performance improvement to accompany any improvement in CPU performance. Indeed, Schroeder and Burrows suggest that tripling CPU speed would reduce SRC RPC latency for a small packet by about 50%, on the expectation that the 83% of the time not spent on the wire will decrease by a factor of 3. Looking at Table 3, however, we see that much of the RPC time goes to functions that may not benefit proportionally from modern architectures. For example, Table 1 shows that system calls and traps do not scale well, and the stub and interrupt processing components of the round-trip RPC include several system calls and interrupts. Large register sets and pipelines, present on most modern RISCs, are not likely to benefit interrupt processing and thread management because of the additional state to examine, save, and restore. The only real "computation" in RPC, in the traditional sense, is the checksum processing, and this in fact is memory intensive and not compute intensive; each checksum addition is paired with a load (which on some RISCs will likely fetch from a non-cached I/O buffer). Thus, Ousterhout found in the Sprite operating system [Ousterhout et al. 88] that kernel-to-kernel null RPC time was reduced by only half when moving from a Sun-3/75 to a SPARCstation-1, even though integer performance increased by a factor of five [Ousterhout 90a].

For larger RPC requests, network transmission time becomes more significant: e.g., nearly 50% for SRC RPC with a 1500-byte result packet. However, the checksum component also doubles in percentage, and the cost of parameter copying (called "marshalling") becomes substantial since it too is memory intensive. Thus, larger packets also have higher CPU costs. Moreover, network bandwidths are increasing quickly; with 10- to 100-fold improvements likely over the next several years, the lower bound on RPC performance will be due to the cost of operating system primitives that include interrupt processing, thread management, and memory-intensive byte copying or checksum operations. We evaluate the costs of interrupt processing in more detail in Section 2.3 and the costs of thread management in Section 4.

2.2 Local Communication

The performance of cross-machine RPC determines how effectively programs can use the network. The performance of *local* cross-address space RPC determines how effectively the operating system can be decomposed, as well as how rapidly clients can communicate with local servers. One mechanism geared to rapid local communication is called lightweight remote procedure call (LRPC) [Bershad et al. 90a]. LRPC achieves performance for the null call that only marginally exceeds the optimal time permitted by the hardware. For the simplest local calls, LRPC achieves a 3-fold performance improvement over previous methods. This gain is obtained through the use of shared, statically-mapped buffers for pa-

Operation	Hardware Minimum	LRPC Overhead
Procedure Calls	4%	–
Kernel Traps	23%	–
Context Switches	42%	–
RPC Stubs	–	13%
Kernel transfer	–	17%
Total	69%	30%

Table 4: LRPC Processing Time

rameter passing, and by allowing the client's thread to directly execute in the server's address space. The second technique nearly eliminates the thread management overhead inherent in message-oriented RPCs.

Table 4 shows the distribution of time for a null LRPC on a CVAX Firefly. With LRPC, the real factor limiting performance is the hardware cost of communicating through the kernel. Each LRPC must enter the kernel twice (once for the call and once for the return). Once inside the kernel, the kernel must perform a context switch, changing the hardware address mapping context from the client to the server address space (vice versa on the return path). Unfortunately, this kernel bottleneck is even worse on newer architectures, because the overhead of system calls and context switching has not decreased proportionally with overall processor speed (cf. Table 1).

With respect to the context switch needed for cross-address space communication, there are two components: the cost of saving and restoring state, and the cost of translation lookaside buffer (TLB) misses caused by the address space change. Both of these will be discussed in later sections. However, it is worth noting that the number of address spaces, as well as the number of cross-address space calls, will be larger for kernelized operating systems (as we will show in Section 5). This may have an effect on TLB efficiency.

2.3 System Calls and Interrupt Handling

Part of the reason for the high cost of system calls on modern processors is the mismatch between the hardware trap support and the software requirements of system calls. Table 5 shows the distribution of time spent by the CVAX, the R2000, and the SPARC in the three components of the null system call: kernel entry/exit, call preparation, and C call/return. In aggregate, we see that the MIPS R2000 and SPARC, while substantially faster than the CVAX on integer performance, are only marginally faster for a null system call.

Looking at the data in a slightly different way, RISCs such as the SPARC and the 88000 (from Table 1) spend many more cycles in software for a system call than the CVAX hardware support requires in microcode. Thus, while the RISCs move the work done by VAX microcode into software, they seem to require much more work overall due to the management of new features. The MIPS R2000 requires 15% fewer cycles than the CVAX for a system call, but this is small compared to the advantage the R2000 should have over the CVAX in total required cycles [Clark & Bhandarkar 91].

The most telling row of Table 5 shows the cost of call preparation for these three machines — the work needed, following the trap, to prepare the processor to execute a procedure call to a C-level operating system routine. Call preparation includes vectoring from the trap entry point to the specific exception handler, window management (on the SPARC), machine state management (e.g., manipulation of machine registers and kernel stack pointers), and saving/restoring of registers that must be preserved across system calls. Because the VAX performs some of these functions in hardware as part of the system call and return from exception instructions, the time to enter and exit the kernel is longer, but the cost once in the kernel is much less.

One reason for this is the addition of new RISC features requiring software management. For example, SPARC has register windows to improve application performance, but the windows reduce kernel trap performance; we estimate that 30% of the null system call time on the SPARC is associated with register window processing. In SPARC, hardware ensures that one register frame is available for execution of the trap handler on exceptions. When a system call occurs, the trap handler must then ensure that *another* frame is available for its call to the specified operating system routine. This requires examination of the register window pointers and possible saving (and later restoration) of frames to memory. Because a frame for the low-level handler is interposed between the user-level caller and the system routine being called, parameters and results must be copied an extra time.

Similarly, the Motorola 88000 loses much of its performance advantage because of the complexity of managing its pipelines in software when a trap occurs. The 88000 includes a large number of registers containing pipeline state, and these must be examined and manipulated on a trap to check for and service any outstanding faults.

While RISCs such as the SPARC and 88000 have directly vectored interrupt dispatching, some RISCs have eliminated this, choosing instead to do vectoring through software. For example, nearly all exceptions on the MIPS R2000, and all exceptions on the Intel i860, are vectored through one handler. In their paper on MIPS operating system support, DeMoney et al. [1986] claim that separate vectoring is unnecessary, and that "most Unix systems fill these [vector] addresses with code to save the cause and then jump to a common interrupt handler, thus adding several cycles." This assumes that a specific implementation of a last-generation operating system will be sufficient for the future. Luckily, despite this argument, the designers provided a separate handler for user-level TLB misses, recognizing that a TLB miss is not an "exceptional" event. However, a system call is not an exceptional event either, and the frequency of system calls, and hence the importance of efficient traps, is increased by modern small-kernel operating system structure (as we will show in Section 5).

Other factors contributing to relatively poor system call performance on RISCs are more subtle. For example, low-

Function	Time (μseconds) CVAX	R2000	SPARC	Relative Speed RISC/CVAX R2000	SPARC
Kernel entry/exit	4.5	.6	.6	7.5	7.5
Call preparation	3.1	6.3	13.1	.5	.24
Call/return to C	8.2	2.1	1.4	3.9	5.8
Total	15.8	9.0	15.2	1.8	1.0

Table 5: Time in Null System Call

level trap handling code on the R2000 makes relatively poor use of load and branch delay slots. Nearly 50% of the delay slots in this code path are unfilled, accounting for approximately 13% of the null system call time on the R2000. Furthermore, the R2000-based DECstation 3100 has a 4-deep write-through buffer, but will stall for 5 cycles on every successive write once the buffer is full. Successive stores are frequent in many operating system functions, such as trap handling or context switch, where registers must be written to memory. We estimate that write buffer stalls account for 30% of the interrupt overhead on the DECstation 3100.

In contrast, the DECstation 5000 has a 6-deep write buffer that can retire a write every cycle if successive writes are to the same page, as they typically are in trap handling. This accounts in part for the the fact that trap performance of the DECstation 5000 is better relative to the DECstation 3100 than one would expect based on their integer performance. With its write buffer, the DECstation 5000 runs without memory-induced slowdown in the common case of cache hits on reads and successive writes to a page, but even so, its trap performance still does not scale with the CVAX.

2.4 Data Copying

As noted by Ousterhout [Ousterhout 90b], data copying is another area in which modern processors have not scaled proportionally to their integer performance, yet it is an important aspect of local and remote communication. In a conventional message passing or RPC system, arguments may have to be copied as many as 4 times: from client to kernel, from kernel to server, from server to kernel, and from kernel to client. Various optimizations can be applied, but even in LRPC which uses a shared client/server buffer, two copies are necessary: one to copy arguments from the invocation stack on the call, and one to copy results on the return.

The problem is that faster systems often have a mismatch between memory speed and processor speed. In fact, Ousterhout found that for a range of RISCs and CISCs, "the relative performance of memory copying drops almost monotonically with faster processors, both for RISC and CISC machines [Ousterhout 90b]." This should not be surprising, in part because the same commodity memory parts are typically used for main memories on both RISCs and CISCs in various performance ranges. Most modern processors have on-chip caches, as well as second-level caches, but these on-chip caches are still relatively small. A limitation during parameter copying is the large number of consecutive write requests, which will stall on some write-through caches. A write-back cache or DECstation 5000-style memory can help, at the cost of added complexity. The problem of data copying for message passing and cache interference is significant enough that some researchers have proposed special architectural support to optimize copy operations [Cheriton et al. 88].

2.5 Summary

In modern operating systems, communication performance is crucial, because it enables good operating system structure and allows effective use of networks and distribution. Good communication performance relies on several primitive functions — including system calls, interrupt handling, context switching, and data copying — that do not fully benefit from modern architectural innovations. In some cases, architectures could improve on the performance of these primitives. For example, on a system call, which is a voluntary exception, a processor like the 88000 could wait for other exceptions to occur before servicing the call, reducing the processing needed in the trap handler to check for faults. Similarly, the SPARC could take a window fault if needed before the call, rather than emulating the check within the trap handler.

We have also seen that techniques used to improve application performance, such as branch delays and write buffers, work less well on some operating system code. Operating system designers must be aware that architectural trends will lead towards relatively more expensive system calls, and should look for mechanisms that avoid the kernel when possible (e.g., [Bershad et al. 90b]).

3 Virtual Memory

The most basic use of virtual memory is to support address spaces larger than the physical memory. For this function, all that is needed is a level of indirection between virtual and physical addresses, provided through TLBs and page tables, plus the ability to fault on an instruction referencing a nonresident page. In general, performance of a virtual memory system is related to the ratio of physical to virtual memory size, the size and organization of the TLB, the cost of servicing a fault, and the page replacement algorithms used. For the operating system, the main issues are the flexibility of the addressing mechanism, the information provided by that mechanism, and the ease of handling faults and changing hardware VM addressing state.

In addition to supporting large address spaces, modern operating systems are making new demands on virtual memory, often to enhance system performance. For example, Accent and Mach use a copy-on-write mechanism to speed program startup and cross-address space communication for large data messages [Fitzgerald & Rashid 86, Young et al. 87]. In the latter case, the kernel maps large message buffers into the receiver's address space, so they are shared read-only by both sender and receiver. Copy-on-write saves memory and avoids copying in the case where the message is not modified after it is sent. If either the sender or receiver attempts to write the buffer, a trap occurs and the buffer is copied. This relies on the ability to quickly trap and change page protection bits.

Virtual memory also can be used to transparently support parallel programming across networks. Such loosely-coupled multiprocessing will become increasingly common as today's Ethernets are replaced by much faster networks. In systems such as Ivy [Li & Hudak 89], a network-wide shared virtual memory is used to give the programmer on a workstation network the illusion of a shared-memory multiprocessor. Pages can be replicated on different workstations as long as the copies are mapped read-only. When one node attempts a write, it faults. Software then executes an invalidation-based coherence protocol, invalidating all copies except the writer's, whose mapping is changed to read-write. The writer then makes any changes on its unique copy. Later execution of a read request on a remote node faults, causing another replica to be created and the writer's copy to be changed back to read-only.

Along with copy-on-write and distributed virtual memory, other operating system functions are being overloaded on virtual memory protection bits as well: these include garbage collection [Ellis et al. 88], checkpointing [Li et al. 90], recoverable virtual memory [Eppinger 89], and transaction locking [Radin 82]. Because these functions often are implemented at the run-time level, their implementations are simplified by user-level handling of page faults and efficient modification of TLB or page table entry access bits. As a result, systems must find a way of quickly reflecting page faults back to the user level, so that user-level code can make an appropriate management decision [Young et al. 87]. This requires both efficient dispatching of the fault within the kernel (i.e., trap handling) and efficient crossing from kernel space to user space and back (i.e., system calls).

3.1 Handling Memory Management Faults

Unfortunately, at the same time that operating systems are making more demands of the virtual memory system, memory management on newer architectures has become more difficult. We have already mentioned that on some RISC processors, exception handling is more complex. There are two other factors that can make the situation worse.

First is the existence of pipelined execution units. While many previous machines have been pipelined, the pipelines were typically invisible to software. On some current RISCs, the pipeline structure is "exposed," that is, compilers and operating systems must deal with the pipeline as part of a process' visible context. On these machines, pipeline visibility is required both for performance and for correctness, and the operating system must manage the pipeline when an exception arises.

As one example, the Motorola 88000 has 5 internal pipelines, including an instruction fetch pipeline, each of which must be restarted after a fault. Associated with these pipelined execution units are nearly 30 internal registers. During an exception, many of these registers must be read, saved, and restored, adding to the complexity and latency of fault handling. When a page fault occurs on the 88000, several instructions may be in the process of execution, and instructions following the faulting instruction may have already completed. For this reason, the program cannot be restarted at the faulting instruction. Instead, the operating system must examine a collection of special registers to find the types of memory accesses underway, the addresses of reads in progress, and the addresses and data values of writes in progress. Then the operating system must *emulate* the execution of the store or read request that caused the fault.

Even the existence of a floating point pipeline can complicate page fault handling. When a page fault occurs on the 88000, hardware "freezes" execution of the floating point unit. Because the floating point unit performs integer multiplication instructions, though, it must be restarted for the fault handler to proceed. Unfortunately, when the floating point unit is reenabled it may complete outstanding operations, writing their results into general-purpose registers that are in use by the fault handler. In effect, a trap must be handled as though it were a full context switch to the FPU; the fault handler must store needed interrupt context in memory, enable the floating-point unit, allow the pipeline to clear, and then save general-purpose registers once they are safe from corruption — all before starting to handle the memory management condition. Similarly, on an interrupt the Intel i860 must save the current state of its pipelines and restore them when the interrupted process is continued. If the floating point pipeline could be in use, the save/restore process adds 60 or more instructions to i860 page fault and other exception handling. Such complexities are not necessary; for example, the IBM RS6000 [IBM 90], the SPARC, and the R2/3000, each of which has several independent pipelined functional units, implement precise interrupts [Smith & Pleszkun 88], thereby shielding software from much of the detail of pipelined processing.

A second factor in fault handling is the reduction, in some processors, of the information provided to handle faults. On the Intel i860, for example, the processor provides no information on the faulting address; in fact, it provides little information about why the fault occurred. On a page fault, the i860 trap handler knows only that a data access fault of some type occurred, and the address of the faulting instruction. The fault handler must then *interpret* the faulting instruction to determine the type of fault and the offending address. This requirement adds 26 instructions to our trap handler in Table 2, despite the fact that the hardware must have the faulting address available when the fault occurs.

3.2 Translation Buffers and Page Tables

While most modern computers have rather standard, 32-bit paged virtual memories, address space management has changed somewhat over the past few years. On conventional systems, virtual address translation conceptually requires one or two overhead memory references for each user memory access. These page table accesses are usually avoided through the use of a TLB, whose entries are searched by hardware prior to a physical cache access. Formerly, TLB miss handling was hidden from the operating system, but some new RISC architectures have moved TLB management into software. For example, the MIPS R2000 has a 64-entry TLB that is loaded by operating system software on misses. One great advantage of this structure is that the architecture does not dictate page table structure. The operating system is free to choose whatever page table structure it likes, and handling of sparse address spaces, which is problematic on a linear page table system like the VAX, is greatly simplified.

While the MIPS TLB structure has moved in the direction of added operating system flexibility for page tables, it is also more restrictive in some ways. Like the VAX, the MIPS virtual address space is divided into two halves: user space and system space. User space addresses are always translated through the TLB. System space, however, is again subdivided into four regions: unmapped and cached, unmapped and uncached, mapped and cached, and mapped and uncached. The reason for having uncached regions is to avoid cache replacements on, for example, writes to I/O buffers that are unlikely to be referenced again. The reason for an unmapped (i.e., physically based) segment is to avoid the translation overhead — in this case to save TLB entries — for operating system components that are typically memory resident anyway. Furthermore, this can be justified because, as with caches, TLBs are poorly used by the operating system relative to user-mode programs. For example, in a study of TLB performance on the VAX-11/780, Clark and Emer found that while the VMS operating system accounts for only one fifth of all references, it accounts for more than two thirds of all TLB misses [Clark & Emer 85].

On the other hand, this system space structure has several problems with respect to modern operating systems. Because the unmapped region is accessed directly through a physical base register, there is no indirection and therefore no ability to specify page-level protection or access control, except to the entire region. The unmapped region of the address space is protected only by kernel mode execution, i.e., code executing in kernel mode can access this region while non-kernel mode code cannot. This organization is therefore best suited to a monolithic kernel structure, like current Unix implementations. With small-kernel operating systems, much operating system code runs in user mode and therefore cannot benefit from this hardware.

Perhaps a better solution to increasing the utilization of TLB entries, particularly for the operating system, is found in the SPARC/Cypress [Cyp 90] implementation. In this case, the architecture supports a 3-level page table structure. The first-level table maps the entire 4GB address space; it contains pointers to second-level tables, each of which maps a 16MB region. Each second-level table contains pointers to third-level tables, each of which maps 256KB of 4KB pages. At each level, an entry can either be a pointer to the next-level table, or a terminal page table entry. If a terminal page table entry is found in the second level, for example, it maps a contiguous 256KB region, and a single TLB entry can be used to hold the mapping for this entire region. Because the region is addressed using PTEs and TLB entries, the standard protection mechanism is still utilized. Furthermore, an operating system specified portion of the 64-entry TLB can be "locked" to prevent hardware from replacing entries in that section.

Many of the newer RISCs have process ID tags in their TLB entries, which allows the entries to live across context switches. This gives them an advantage over untagged systems such as the VAX. In fact, in a null LRPC (Table 4), an estimated 25% of the time is lost to TLB misses on the CVAX, because the entire TLB must be purged twice, once during the call and once on return. It is worth noting that kernelized operating systems will increase the demand for tag bits and TLB size, since a kernelized structure increases the number of address spaces and context switches.

A final complexity for new architectures is the inclusion of virtually addressed caches. Virtually addressed caches are attractive because they can reduce cycle time by removing the need for a virtual address translation preceding cache lookup. Virtually addressed caches are similar to translation buffers in two ways: (1) the cache address tags are context dependent, and therefore the cache must be flushed on a context switch, and (2) each cache entry contains protection bits, so entries must be invalidated when a PTE is changed.

Depending on the architecture, virtually addressed caches can increase the cost of low-level operating system functions by an order of magnitude. Cache flushing at context switch time can be extremely expensive, as shown by the high instruction count for the i860 context switch in Table 2. Process IDs can eliminate the need for this, though.

Efficiently changing page table protection information in a virtually addressed cache is more difficult. At most one entry in a TLB need be invalidated when a page's protection is changed. With a virtually addressed cache, however, any change to a page's protection requires a complete search of the cache and (typically) invalidation of any blocks on that page; this is a time-consuming operation. On the i860, for example, 536 out of the 559 instructions required to change a PTE are concerned with flushing the virtual cache.

3.3 Summary

New architectures can simplify memory management, but they also can add significant complexity and latency to the memory management task. At a time when operating systems are making new uses of memory management, e.g., for copy-on-write message passing or transaction support, architectures can help by not hiding information, such as the fault address needed for fast fault handling. On the other side, with the potential performance advantages of virtually addressed caches and imprecise interrupts, operating systems for modern architectures may need to be less aggressive in their use of copy-on-write and similar mechanisms that rely on fast fault handling.

4 Threads and Multiprocessing

Threads, or "lightweight processes," have become a common and necessary component of new languages and operating systems [Jones & Rashid 86, Halstead 85, Bershad et al. 88]. Threads allow the programmer or compiler to express, create, and control parallel activities, contributing to the structure and performance of parallel programs. A thread is simply one stream of control within a parallel program. Because threads within a single application are "lightweight" (i.e., they share a single address space), threads do not require the large amount of software and hardware state needed by a full process that contains hardware context for address space management.

Threads can be supported by the operating system, by the application run-time level, or by both. At the operating system level, threads allow the application to create multiple units of work within an address space that are individually schedulable by the operating system. The advantage is that the operating system provides a uniformity of function. At the run-time level, threads are completely managed by user-level code invisibly to the operating system. The advantage is performance and flexibility; thread operations do not need to cross kernel boundaries, and a thread system can be specially tailored to an application's needs.

With carefully-implemented user-level thread systems, it is possible to provide high-performance parallel programming primitives that approach minimal hardware costs, e.g., new thread creation in 5–10 times the cost of a procedure call [Anderson et al. 89, Massalin & Pu 89]. This is due to the low cost of communication within a single address space running in a single protection mode. Also, through careful kernel-to-user interface design, user-level threads can provide all of the function of kernel-level threads without sacrificing performance [Anderson et al. 90].

It is likely that the importance of threads will continue to increase in the future, as programmers and compilers seek speedup through finer and finer grained parallelism. The ability to obtain speedups in this way — through fine-grained parallelism — is dependent to a great extent on the cost of thread creation and thread operations. If thread operations are inexpensive, then threads can be freely used for fine-grained activities; if thread operations are costly, then only coarse-grained parallelism can be effectively supported.

4.1 Architecture and User-Level Threads

While fine-grained user-level threads require no special architectural support, the architecture can have a large impact on thread performance. Most crucial is the amount of state that must be managed to context switch from one thread to another thread in the same address space. (Note that the context switch measured in Table 1 includes this cost, plus the cost of changing the address space.) Table 6 shows the amount of thread state for several modern architectures. Most of the newer RISC processors, such as the Sun SPARC, the MIPS R2000 and the IBM RS6000, have more than 64 registers, compared to 16 in most earlier 32-bit CISC architectures, such as the VAX. On a context switch, these registers must be written into a thread control block, and an equal number of reads are required to load the registers for the newly scheduled thread. In a context switch that passes through the operating system, this cost may be noticeable but small relative to the total context switch overhead. But, in a fine-grained user-level thread system, these reads and writes become the dominating cost. Optimizations that reduce the amount of state saving, e.g., saving only those registers that are in active use [Wall 86], may become crucial to minimizing context switch costs.

On the SPARC architecture, the number of registers to be saved depends on the number of register windows in use at the time of the context switch. Measurements of Sun Unix have shown that for SPARC systems with 8 windows, on average three need to be saved/restored on each context switch [Kleiman & Williams 88]. Our SPARC context switch driver for Table 1, which assumes the SUN Unix average, spends 70% of its time saving and restoring windows (12.8 μseconds per window). Thus, the cost of reading and writing these registers makes fine-grained threads highly inefficient. Worse, because SPARC's current window pointer is in a privileged register, a completely user-level thread context switch is impossible; a kernel trap is required to change the current window pointer in order to switch from the current thread's register windows to the new thread's windows.

The large register sets on architectures such as SPARC are not gratuitous; they were motivated by scores of measurements taken primarily on sequential Unix application programs. Larger register sets can reduce memory accesses, which have become relatively more costly as processor speeds have increased. In particular, register windows greatly reduce the cost of parameter passing and register saving on procedure call. The assumption was that procedure calls are much more frequent than context switches. Since context switching was expensive in earlier operating systems, it was avoided if possible. But to realize potential speedup on current parallel machines, thread context switches implemented completely at user-level may become more frequent.

As one test, we ran several experiments with the Synapse parallel simulation environment [Wagner 89] executing on a Sequent shared-memory multiprocessor. Synapse is an object-oriented system with lightweight threads scheduled at user-level. Across the experiments measured, we found that the ratio of procedure calls to context switches varied from 21:1 to 42:1 (8 calls were made by the run-time system, the rest by the application). Synapse is object-oriented, which tends to be procedure call-intensive, and its run-time system implementation explicitly attempts to reduce the number of context switches. Even so, on a SPARC Synapse would spend more of its time doing context switches than procedure calls, because the cost of a thread context switch is 50 times that of a procedure call, assuming 3 window save/restores for each context switch. Thus, for this parallel application, the SPARC tradeoff of improving procedure call time at the expense of increasing the time to do context switches is less cost effective than on sequential programs. It is not surprising, then, that some researchers use a SPARC register window per thread as a way of optimizing context switches instead of procedure calls [Agarwal et al. 90].

	VAX	88000	R2/3000	SPARC	i860	RS6000
Registers	16	32	32	136	32	32
F.P. State	0	0	32	32	32	64
Misc. State	1	27	5	6	9	4

Table 6: Processor Thread State (32-bit words)

Although many context switches in a user-level thread package are voluntary, such packages must also perform involuntary swaps as a result of asynchronous events, for instance due to signals or exceptions. As noted in Section 3.1, exception handling is more difficult in newer architectures.

Kernel-level threads can be problematic too, e.g., causing decreased TLB effectiveness due to an increased number of thread context switches between threads in separate address spaces. This is a particular problem for architectures with small numbers of TLB entries. The problem occurs especially if threads are scheduled independently of the address space with which they are associated.

A final issue for multiprocessing is support for atomic memory lock instructions, required to synchronize fine-grained access to shared resources. The MIPS R2000/R3000 has no atomic semaphore instruction; this omission hurts uniprocessor performance as well as multiprocessor performance, since threads often are used for program structure as well as for parallel programming. On the MIPS, threads that wish to synchronize must either trap into the kernel, where interrupts can be disabled, or resort to a complex locking algorithm. Both are expensive. For example, *parthenon*, a resolution-based theorem prover that exploits or-parallelism, is able to decrease its total execution time by 10% on a MIPS R3000-based *uniprocessor* through the use of multiple threads. However, this program spends roughly 1/5 of its time synchronizing through the kernel.

Lock management can be particularly difficult on architectures in which the operating system must handle exceptions that arise in the middle of an atomic instruction or critical section. On the Intel i860, for example, software must be able to restart the critical section at the lock instruction when a fault occurs inside a critical section. If the critical section includes non-reexecutable store instruction, the software must first store unmodified values in store targets of those non-reexecutable instructions to ensure that no fault occurs. The Motorola 88000 has a similar situation in which faults cannot be allowed on the write portion of a read-modify-write instruction. These problems are not insurmountable, and code can be written to perform an atomic sequence correctly, but not without a cost in the latency of lock acquisition and a possible expansion of the critical section, reducing throughput and potential parallelism.

4.2 Summary

Thread management seems to be an area in which architectures and operating systems have moved at cross purposes. While architectural mechanisms have been added to speed the execution of sequential programs, operating systems have been changing to support fine-grained parallel decomposition through threads. Unfortunately, the best assumptions for efficient sequential execution (e.g., long sequences of uninterrupted instructions) and the best architectural features (e.g., large amounts of processor state) are the worst choices for fine-grained multithreaded programs.

5 The Behavior of Operating Systems and Applications

We have shown that the performance of primitive operating system functions has not scaled with overall processor performance, and we have explored a number of the reasons for this fact. In this section, we support the importance of these primitives by showing that:

1. low-level primitives such as trap and context switch are frequently used, and
2. these primitives are becoming more frequently used as operating system structure evolves.

To demonstrate the first point, we monitored operating system behavior during the execution of a set of applications. To demonstrate the second point, we ran each application on two different versions of Mach, a binary compatible reimplementation of Unix. The first version, Mach 2.5, is monolithic: the entire operating system executes in a privileged kernel address space. The second version, Mach 3.0, has a small message-based kernel on which traditional operating system services are implemented as user-level programs. It is important to note that Mach 3.0 is not a "completely decomposed" operating system: many services are provided by a single application-level server which could more logically be provided by multiple servers. Our measurements indicate that the performance of operating system primitives on current architectures may limit the extent to which systems such as Mach can be further decomposed without compromising application performance.

We selected applications that we believe are representative of a typical workload in a workstation environment. These were: *spellcheck-1* (spellcheck a 1 page document); *latex-150* (format a 150 page document); *andrew-local* (a script of file system intensive programs such as copy, compile and search, run using an entirely local file system); *andrew-remote* (the same script run using a remote file system); *link-vmunix* (the final link phase of a Mach kernel build) and *parthenon* (a resolution-based theorem prover that uses multiple threads to exploit or-parallelism).

All applications were run on a MIPS R3000-based DECstation 5000/200 with 24 megabytes of memory. We ran each program three times to smooth out the effects of paging and file caching. Over the second and third runs there was little variation in performance. We instrumented the operating

	Time (sec.)	Address Space Switches	Thread Context Switches	System Calls	Emul. Instrs.	KTLB Misses	Other Exceptions	% Time in OS Prims.
Mach 2.5 (monolithic)								
spellcheck-1	2.3	139	238	802	39	2953	2274	
latex-150	69.3	2336	2952	5513	320	34203	15049	
andrew-local	73.9	3477	5788	35168	331	145446	67611	
andrew-remote	92.5	3904	6779	35498	410	205799	67618	
link-vmunix	25.5	537	994	13099	137	46628	15365	
parthenon (1 thread)	22.9	171	309	257	1395555	1077	2660	
parthenon (10 threads)	20.8	176	1165	268	1254087	2961	3360	
Mach 3.0 (decomposed)								
spellcheck-1	1.4	1277	1418	1898	13807	22931	2824	20%
latex-150	80.9	16208	19068	16561	213781	378159	19309	5%
andrew-local	99.2	41355	50865	70495	492179	1136756	144122	12%
andrew-remote	150.0	128874	144919	160233	1601813	1865436	187804	16%
link-vmunix	29.9	24589	25830	26904	164436	423607	28796	16%
parthenon (1 thread)	28.8	1723	2211	1308	1406792	12675	3385	18%
parthenon (10 threads)	26.3	1785	3963	1372	1341130	18038	4045	19%

Table 7: Application Reliance on Operating System Primitives

system kernels to count the occurrences of the primitive operations described in previous sections.

Table 7 summarizes our measurements. For each application running on each Mach version, it shows (for the final two runs) the average elapsed time, and the average number of address space context switches, kernel-level thread context switches[2], kernel-handled system calls, kernel-emulated instructions, kernel-mode address TLB misses, and other exceptions (primarily interrupts and page faults, but excluding user-mode TLB misses). For Mach 3.0 we also show the percentage of its elapsed time that each application spends executing these low-level system primitives (*not including*, in the case of system calls and other exceptions, the time spent actually executing the called system procedure).

The table illustrates several key points about application behavior and operating system structure. First, and most important, operating system primitives occur frequently, particularly in a small-kernel operating system such as Mach 3.0, and their performance has a clear effect on application performance. Under Mach 3.0, most of the applications spend between 15 and 20 percent of their time executing these primitives. For example, execution of these primitives during the remote Andrew script consumes nearly 26 seconds out of 150. Clearly this overhead is substantial, and any attempt to optimize the execution of this application would have to take this component into consideration. Further, while the percentages reported in Table 7 are specific to the MIPS R3000, the performance of the primitives would also be important on other architectures. For example, the combination of Tables 1 and 7 indicates that a SPARC would spend 9.4 seconds just in the overhead for system calls and context switches in executing the remote Andrew script on Mach 3.0.

[2] In Mach 3.0, an address space context switch implies a kernel-level thread context switch, but not vice versa.

Second, from the two halves of Table 7 we see that a decomposed system will execute more low-level system functions than a monolithic system. Many operating system calls which in Mach 2.5 are implemented in the kernel, are provided in Mach 3.0 by cross-address space RPCs to operating system servers running at user-level. Each invocation of an operating system service via an RPC requires at least two system calls and two context switches (one to send the request; another to send the reply) to do the work of one system call in a monolithic system. In addition, the number of context switches for a program running on a decomposed system increases because the operating system servers are themselves multithreaded and can run concurrently with applications. As an example of this effect, there is a 33-fold increase in context switches for the remote Andrew benchmark on Mach 3.0 over Mach 2.5. This benchmark modifies a large number of files, and each open and close operation involves at least two local RPCs — one to the local Unix server and another to the local file cache manager. In contrast, the Mach 2.5 kernel implements all this functionality itself; one system call handles everything.

Third, the number of kernel-level TLB misses is significantly larger for all applications running under Mach 3.0 than it is under Mach 2.5. TLB misses on the R3000 are handled by one of two software exception handlers. One deals with user-space misses and has a latency of about a dozen cycles. The second handles misses in kernel space and is presumed to be infrequently invoked because much of kernel space can run unmapped (thereby increasing the effectiveness of the fixed-size TLB), but has a latency of a few hundred cycles. Page tables, for instance, remain mapped in kernel mode; TLB entries are needed to map the page tables themselves. With much of the operating system moved to the user level, less code and data are using the unmapped regions, and frequent context switching stresses the limited number of TLB entries on the R3000. These effects increase the number of second-level misses by an order of magnitude.

Finally, the emulated instruction counts demonstrate the significant performance impact that can arise from the omission of an important architectural feature, in this case the lack of an atomic *Test-and-Set* instruction. While critical sections are commonplace in the Mach 2.5 operating system, they execute in kernel mode and can simply disable interrupts. But in Mach 3.0 the operating system's critical sections execute at user-level; a trap to the kernel is needed to provide mutual exclusion. (Existing non-trap based solutions to the mutual exclusion problem (e.g., [Lamport 87]) still have overheads on the order of dozens of cycles.)

6 Conclusions

We have described the operating system needs in three fundamental areas and the relationship of modern architectures to those needs. In general, we have seen that:

- Operating systems are being decomposed into kernelized structures with independent servers executing in separate address spaces. These separate address spaces communicate with users and with each other using RPC. At the same time, architectures have made message-based communication (relatively) more expensive because system calls, interrupt handling, and byte copying are (relatively) more expensive.
- Operating systems are requiring more use of memory management, at a time when handling memory management events has become more difficult.
- Operating systems are moving towards support of fine-grained multithreaded application programs. At the same time, architectures are adding more processor state, which makes fine-grained threads more expensive.

The current state of affairs is definitely not catastrophic — the performance of today's desktop workstations was unimaginable (or at least unaffordable) only 5 years ago. On the other hand, unless architects pay more attention to operating systems, and operating system designers pay more attention to architecture, operating system performance will become a severe bottleneck in next-generation computer systems.

7 Acknowledgements

We would like to thank Robert Bedichek, John Ousterhout, and Robert Short, who helped us to understand several of the architectures and to obtain some of the data presented in this paper. Jeff Chase, Susan Eggers, Sabine Habert, Eric Koldinger, and John Zahorjan provided helpful reviews of early versions of this paper. We would also like to thank the referees, who caused us to greatly improve this paper.

References

[Agarwal et al. 88] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[Agarwal et al. 90] A. Agarwal, B.-H. Lim, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114, May 1990.

[Anderson et al. 89] T. Anderson, E. Lazowska, and H. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[Anderson et al. 90] 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. Technical Report 90-04-02, Department of Computer Science and Engineering, University of Washington, October 1990.

[Baskett et al. 75] F. Baskett, J. H. Howard, and J. T. Montague. Task communication in DEMOS. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, pages 23–31, November 1975.

[Bershad et al. 88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[Bershad et al. 90a] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[Bershad et al. 90b] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared-memory multiprocessors. Technical Report TR-90-05-07, Department of Computer Science and Engineering, University of Washington, July 1990.

[Birrell & Nelson 84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Cheriton 84] D. R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[Cheriton et al. 88] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen. The VMP multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 410–421, May 1988.

[Cheriton et al. 90] D. R. Cheriton, G. R. Whitehead, and E. W. Sznyter. Binary emulation of Unix using the V kernel. In *Proceedings of the Summer 1990 USENIX Conference*, pages 73–85, June 1990.

[Clark & Bhandarkar 91] D. W. Clark and D. Bhandarkar. VAX versus RISC: Quantitative evidence from comparable implementations. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, April 1991.

[Clark & Emer 85] D. W. Clark and J. S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[Cyp 90] Cypress Semiconductor, San Jose, CA. *SPARC RISC User's Guide*, 1990.

[DeMoney et al. 86] M. DeMoney, J. Moore, and J. Mashey. Operating system support on a RISC. In *Proceedings of the 31st Computer Society International Conference (Spring Compcon '86)*, pages 138–143, March 1986.

[Ellis et al. 88] J. R. Ellis, K. Li, and A. W. Appel. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[Eppinger 89] J. L. Eppinger. *Virtual Memory Management for Transaction Processing Systems.* PhD dissertation, Carnegie-Mellon University, February 1989.

[Fitzgerald & Rashid 86] R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[Golub et al. 90] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Halstead 85] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hennessy et al. 82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill. Hardware/software tradeoffs for increased performance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM, March 1982.

[IBM 90] IBM Corporation, Advanced Workstation Division, Austin, Texas. *POWER Processor Architecture*, 1990.

[Int 89] Intel Corporation. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1989.

[Jones & Rashid 86] M. B. Jones and R. F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 67–77, October 1986.

[Kane 87] G. Kane. *MIPS R2000 RISC Architecture.* Prentice Hall, Englewood Cliffs, N.J., 1987.

[Katevenis 85] M. G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* The MIT Press, Cambridge, Massachusetts, 1985.

[Kleiman & Williams 88] S. Kleiman and D. Williams. SunOS on SPARC. *Sun Technology*, Summer 1988.

[Lamport 87] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987.

[Leonard 87] T. E. Leonard, editor. *VAX Architecture Reference Manual.* Digital Press, Bedford, MA, 1987.

[Li & Hudak 89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li et al. 90] K. Li, J. F. Naughton, and J. S. Plank. Real-time concurrent checkpoint for parallel programs. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, March 1990.

[Massalin & Pu 89] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Mot 88a] Motorola, Inc., Phoenix, AZ. *MCS 88100 RISC Microprocessor User's Manual*, 1988.

[Mot 88b] Motorola, Inc., Phoenix, AZ. *MCS 88200 Cache/Memory Management Unit User's Manual*, 1988.

[Ousterhout 90a] J. K. Ousterhout. Personal communication, July 1990.

[Ousterhout 90b] 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.

[Ousterhout et al. 88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[Patterson & Ditzel 80] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *Computer Architecture News*, 8(6):25–33, October 1980.

[Patterson & Sequin 82] D. A. Patterson and C. H. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, September 1982.

[Radin 82] G. Radin. The 801 minicomputer. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47. ACM, March 1982.

[Rashid & Robertson 81] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[Schroeder & Burrows 90] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Smith & Pleszkun 88] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[SPEC 90] SPEC newsletter benchmark results. Systems Performance Evaluation Cooperative, 1990.

[Sun 87] Sun Microsystems, Inc., Mountain View, CA. *The SPARC Architecture Manual*, 1987.

[Thacker et al. 88] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[van Renesse et al. 88] R. van Renesse, H. van Staveren, and A. S. Tanenbaum. Performance of the world's fastest distributed operating system. *ACM Operating Systems Review*, 22(4):25–34, October 1988.

[Wagner 89] D. B. Wagner. *Conservative Parallel Discrete-Event Simulation: Principles and Practice.* PhD dissertation, Univeristy of Washington, September 1989.

[Wall 86] D. W. Wall. Global register allocation at link time. In *ACM SIGPLAN Symposium on Compiler Construction*, June 1986.

[Young et al. 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. 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.