CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 5(3), 171–191 (MAY 1993)

Kernel–Kernel communication in a shared-memory multiprocessor

ELISEU M. CHAVES,* JR., PRAKASH CH. DAS,* THOMAS J. LEBLANC, BRIAN D. MARSH* AND MICHAEL L. SCOTT

Computer Science Department
University of Rochester
Rochester, New York 14627-0226, U.S.A.

SUMMARY

In the standard kernel organization on a bus-based multiprocessor, all processors share the code and data of the operating system; explicit synchronization is used to control access to kernel data structures. Distributed-memory multicomputers use an alternative approach, in which each instance of the kernel performs local operations directly and uses remote invocation to perform remote operations. Either approach to interkernel communication can be used in a large-scale shared-memory multiprocessor.

In the paper we discuss the issues and architectural features that must be considered when choosing between remote memory access and remote invocation. We focus in particular on experience with the Psyche multiprocessor operating system on the BBN Butterfly Plus. We find that the Butterfly architecture is biased towards the use of remote invocation for kernel operations that perform a significant number of memory references, and that current architectural trends are likely to increase this bias in future machines. This conclusion suggests that straightforward parallelization of existing kernels (e.g. by using semaphores to protect shared data) is unlikely in the future to yield acceptable performance. We note, however, that remote memory access is useful for small, frequently-executed operations, and is likely to remain so.

1. INTRODUCTION

Computer architecture has a strong influence on the design of multiprocessor operating system kernels, affecting the distribution of kernel functionality among processors, the form of interkernel communication, the layout of kernel data structures and the need for synchronization. For example, in bus-based shared-memory multiprocessors it is easy for all processors to share the code and data of the operating system.[1] Explicit synchronization can be used to control access to kernel data structures. Both distributed-memory multicomputers (e.g. hypercubes and mesh-connected machines) and distributed systems (e.g. workstations on a network) use an alternative organization, with kernel data distributed among the processors, each of which executes a copy of the kernel.

*Eliseu Chaves is with the Universidade Federal do Rio de Janeiro, Brazil. He spent six months on leave at the University of Rochester in 1990. Prakash Das is now with Transarc Corp. in Pittsburgh, PA. Brian Marsh is now with the Matsushita Information Technology Lab in Princeton, NJ.

[1] It is customary to refer to bus-based machines as UMA (uniform memory access) multiprocessors, but the terminology can be misleading. Main memory (if present) is equally far from all processors, but caches are not, and caches are the dominant determinant of memory performance.

1040–3108/93/030171–21$15.50

Received 24 February 1992
Revised 4 September 1992

Each instance of the kernel performs operations on local data directly and uses remote invocation to request operations on remote data. Non-pre-emption of the kernel (other than by interrupt handlers) provides a significant amount of implicit synchronization among the kernel threads sharing a processor.

Although very different, these two organizations each have their advantages. A shared-memory kernel is similar in structure to a uniprocessor kernel, with the exception that access to kernel data structures requires explicit synchronization. As a result, it is relatively straightforward to port a uniprocessor implementation to a shared-memory multiprocessor. Having each processor execute its own operations directly on shared memory is also reasonably efficient, at least on small machines. In addition, this kernel organization simplifies load balancing and global resource management, since all information is globally accessible to all kernels.

Message-passing (i.e. remote invocation) kernels, on the other hand, are naturally suited to architectures that do not support shared memory. Each copy of the kernel is able to manage its own data structures, so the source of errors is localized. The problem of synchronization is simplified because all contention for data structures is local, and can be managed in large part using non-pre-emption. This kernel organization scales easily, since each additional processor has little impact on other kernels, other than the support necessary to send invocations to one more kernel.

Large-scale shared-memory multiprocessors have properties in common with both bus-based machines and distributed-memory multicomputers. All of memory can be accessed directly, but at very different costs. The programmer may be responsible for allocating data to processors (as in so-called NUMA—non-uniform memory access—machines such as the BBN Butterfly[1], IBM RP3[2], Illinois Cedar[3], and Toronto Hector[4]), or the hardware may provide coherent caches (as in the Scalable Coherent Interface[5] and the Stanford DASH[6], MIT April/Alewife[7], and Kendall Square[8] machines). In either case, the performance and conceptual trade-offs between the use of remote invocation and remote memory access in the kernel are not well understood, and depend both on architectural parameters and on the overall design of the operating system.

In this paper we focus on the trade-offs between remote memory access and remote invocation in kernel-level software on NUMA multiprocessors. We begin in Section 2 by discussing the various aspects of locality, and the range of options available for kernel–kernel communication. We then explore the implementation of one of these options in detail in Section 3. Specifically, we note that remote invocation can cause a kernel operation to execute either at interrupt level or in a normal kernel process context, and that the choice between these alternatives has a major effect on the generality and performance of the mechanism, and on its impact on other aspects of kernel execution.

With implementation details in hand, we examine remote access/invocation trade-offs in Section 4. We consider direct costs (latency) for individual remote operations, indirect costs imposed on local operations by the choice of remote communication mechanism, processor and memory contention, and the conceptual compatibility of communication mechanisms with common models of kernel organization. The importance of these trade-offs can be seen in current trends: operating system overhead has grown to 15–20% of execution time on modern microprocessors[9], and the growing complexity of parallel systems software demands that kernels be made as clean and maintainable as possible.

Our observations are made concrete in Section 5 through a series of experiments with our implementation of the Psyche multiprocessor operating system on the BBN Butterfly

Plus multiprocessor (the hardware base of the GP1000 product line). Our conclusions appear in Section 6. Briefly, we find that remote memory access provides reasonable performance only for interprocessor kernel operations that perform less than a few dozen memory references. Remote invocation enjoys a significant performance advantage for lengthier operations, an advantage that is likely to increase in future machines. Since lengthy operations generally require a kernel process context, interrupt-level remote invocation is useful only in special cases.

2. KERNEL–KERNEL COMMUNICATION OPTIONS

As multiprocessors increase in size, it becomes increasingly difficult to construct operating systems for them that perform well. Machines with a very small number of processors can use a lightly modified version of an existing operating system (e.g. Unix) in a master–slave configuration. All kernel calls execute on a single processor; other processors request services via traps to a remote invocation mechanism. Unfortunately, as the scale of the machine increases, the master processor inevitably becomes a bottleneck. By using locks to protect shared data structures, several manufacturers have parallelized the Unix kernel for concurrent execution on bus-based shared-memory multiprocessors of up to 30 processors[10]. Even on this scale, however, the modifications required to avoid performance-degrading contention are non-trivial[11–13].

Operating systems for machines with large numbers of processors (hundreds or even thousands) will require extensive rewrites of existing code, or will need to be written from scratch. The latter approach has been employed successfully by the vendors of distributed-memory multicomputers. The former approach is being pursued by a variety of groups (e.g. OSF), but has yet to be demonstrated on a large-scale machine. In developing the nX operating system for the Butterfly GP1000 and TC2000 machines, BBN ACI explicitly eschewed the goal of large-scale parallelism within the Unix kernel, opting instead for a resident front-end. Unix applications under nX run only within one small *cluster* of the machine, with a master–slave kernel organization. The bulk of the machine is dedicated to running parallel applications with little kernel support other than high-speed parallel I/O. Similarly, the version of Mach developed for the IBM RP3 performs best when most of the processors do not make system calls[14].

We focus in the remainder of this paper on design alternatives for general-purpose parallel operating systems, in which the full range of kernel services are available with reasonable performance on every processor. We consider a machine organization consisting of a collection of *nodes*, each of which contains memory and one or more processors (possibly with caches). Each processor can access all of the memory on the machine, but it can access data located at the local node much more quickly than it can access data located at a remote node. When a processor at node i begins executing an operation that must access data on node j, interaction among nodes is required.

Experience with several multiprocessor kernels indicates that most kernel operations can be performed primarily using local memory references on some node. This *node locality* in kernel operations is crucial for reasonable performance in large machines. It implies that most memory accesses will be local even when using remote memory accesses for interkernel communication, and that the total amount of time spent waiting for replies from other processors when using remote invocation will be small compared to the time spent on other operations. At the same time, experience with uniprocessor

operating systems suggests that it is very hard to build a kernel with a high degree of *address locality*. Consecutive memory references tend not to lie in any small set of dense address ranges[15], due to heavy use of pointer-based dynamic data structures, operations on multiple process contexts, interrupt-driven activity and a lack of nested loops.

On a NUMA multiprocessor (without coherent caches) there are three principal alternatives for kernel–kernel communication:

Remote memory access: The operation executes on node i, reading and writing node j's memory as necessary. The memory at node j may be mapped by node i statically, or it may be mapped on demand.

Remote invocation: The processor at node i sends a message to a processor at node j, asking it to perform the operation on its behalf. The operation may be executed directly by the message interrupt handler, or indirectly via wake-up of a kernel process.

Bulk data transfer: The kernel moves the data required by the operation from node j to node i, where it is inspected or modified, and possibly copied back. The kernel programmer may request this data movement explicitly, or it may be implemented transparently by lower-level software using page faults.

Cache-coherent multiprocessors blur the distinction between remote memory access and bulk data transfer. As in software bulk transfer requests, cache-coherent machines move multiple words of data at a time, potentially improving performance both through amortized movement costs (prefetching) and through repeated local access during an operation (caching). On the other hand, cache-coherent machines migrate data automatically, and generally use cache lines that are smaller than a page. The lack of kernel intervention during data migration means that logical structure, locking and synchronization issues that pertain to remote memory access in NUMA machines also pertain in cache-coherent machines. Moreover, in a kernel with a high degree of node locality, most data items will have a node at which they usually reside (even on cache-only machines[8] with no 'main' memory), and will migrate back to that node if temporarily accessed elsewhere. Most of the trade-offs between remote memory access and remote invocation discussed in the following Sections apply to both classes of architecture, though we will couch our discussion in terms of NUMA machines.

In terms of the communication options listed above, the lack of address locality in the kernel suggests that data accessed by any particular kernel operation are unlikely to be contiguous, even if they reside on the same node. We therefore concentrate here on a comparison between remote memory access and remote invocation. We present two different versions of remote invocation in Section 3. One is fast but of limited use; the second is slower but more general. In the subsequent section, and in the case study that follows, we focus on the trade-offs between remote memory access and these two forms of remote invocation. We consider direct, measurable costs of individual remote operations, indirect costs imposed on local operations, the effects of competition among remote operations for processor and memory cycles, and the extent to which different communication mechanisms complement or clash with the structural division of labour among processes in the kernel. Ultimately, we find that the more general form of remote invocation is best for most operations, but that the other two mechanisms (remote memory access and the faster form of invocation) are both better in certain cases.

3. IMPLEMENTATION OF REMOTE INVOCATION

The simplest way to perform a remote invocation is to execute the requested operation in the interrupt handler of the interprocessor communication mechanism. In Unix terminology[16], this places the code for the operation in the 'bottom half' of the target processor's kernel. Alternatively, one can arrange to execute remote invocations in a normal process context in the 'top half' of the kernel. Doing so can be expensive: it requires a context switch out of the interrupt handler, and may require synchronization with other process(es) running in the kernel. An interrupt-level remote invocation (RI) may be very fast, but can only be used for operations that can be executed safely in an interrupt handler. A process-level RI is slower, but of more general utility.

3.1. Interrupt-level remote invocation

Since interrupts occur at unpredictable times from the target processor's point of view, and since interrupt handlers cannot block (because they lack a process context), mutual exclusion for data structures shared between interrupt handlers and the rest of the kernel must be achieved by masking interrupts. On a uniprocessor these data structures consist primarily of I/O buffers. On a multiprocessor with interrupt-level RIs, they may be more numerous and varied, and normal (non-interrupt) kernel routines may need to lock more than one data structure at a time. Simply turning interrupts on and off may not suffice for lock acquisition and release.

One standard solution is to maintain a count of the number of critical sections currently active in normal kernel routines. Interrupt-level RIs are permitted only when the counter is at zero (i.e. when interrupts are enabled). The lock acquisition and release routines turn interrupts on or off when changing the counter from 1 to 0 or vice versa. Alternatively, the handler for interrupt-level RIs could place requests in a queue when the counter is non-zero (rather than executing them immediately) and the lock release routine could regenerate an interprocessor communication interrupt when changing the counter from 1 to 0 in the presence of a non-empty queue. This latter approach minimizes the period of time during which interrupts are masked, and may improve performance by reducing the probability of lost interrupts.[2]

Interrupt-level RI has several limitations. Because its handlers lack a process context, it cannot be used for operations that may block. If used extensively it may require more data structures to be available to interrupt routines than would otherwise have been the case. The only form of synchronization available for these data structures is short-term mutual exclusion (masking interrupts), and that occurs at a very coarse grain. Even invocations that do not touch data of any interest to current process-level activity are likely to be disabled much of the time. To prevent deadlock, we must prohibit outgoing invocations when incoming invocations are disabled. This rule severely limits the circumstances under which an interrupt-level RI is permitted. In particular, it precludes the use of interrupt-level RI for operations that must access data on two different processors as a

[2] Message-based multicomputers generally incorporate hardware queuing to avoid lost messages. Shared-memory multiprocessors generally incorporate a single interprocessor interrupt vector, with no lower bound on the time between interrupts from separate processors. They rely on software queuing in shared memory to tolerate lost interrupts.

single atomic operation, unless the programmer is willing to detect and recover from deadlock.

It may be possible on some machines to interrupt a remote processor at a lower priority than device interrupts (or to re-post a high-priority interrupt at a lower level).[3] It may therefore be feasible to use interrupt-level RI for lengthy operations. Several other factors, however, suggest that it be used only for short operations. Longer operations are more likely to need condition synchronization, or to require locks on more than one processor, or to require that large amounts of data be accessible to interrupt routines. Longer operations are also more able to tolerate the latency of a general-purpose, process-level invocation. Finally, during short operations it makes sense for the requesting processor to busy-wait for notice of completion, which in turn makes it possible for us to request an interrupt-level RI from within the handler for another. Rescheduling is likely to be slower than busy-waiting when requesting an interrupt-level RI from a normal kernel routine. Rescheduling is not possible in an interrupt-level routine, but busy-waiting is, provided that we restore data structures to a consistent state and re-enable interrupt-level RIs before making a nested invocation.[4] We assume in the rest of this paper that interrupt-level RI is used for short operations only, and that the requesting processor spins.

3.2. Process-level remote invocation

To execute a remote invocation in the normal (process-level) part of the target processor's kernel, the interprocessor communication interrupt handler uses the same mechanism employed by device handlers to initiate 'top-half' activity. If the processor was executing in user space prior to the interrupt, the handler performs an asynchronous trap and the remote invocation executes immediately. If the processor was executing in the kernel prior to the interrupt, the handler queues the invocation for execution the next time control returns to user space or enters the kernel's idle loop. Deadlock prevention requires that a process blocks when making a process-level RI, rather than busy-waits, because busy-waiting would lock out incoming process-level RIs. Blocking also makes sense in other ways: it may take a significant amount of time to get around to executing a request on the target processor, and we assume that process-level RI will be used for longer operations anyway.

Because it executes in a process context, the requested operation can block during its execution, e.g. for condition synchronization. A process can perform a process-level RI while holding semaphores or other scheduler-based locks. Processes can therefore synchronize with the execution of process-level RIs from other processors. Deadlock is still possible, but only as a result of algorithmic problems in the kernel, not because of the over-coarse locking inherent in the invocation mechanism.

Remote memory access and the two forms of remote invocation are to a large extent compatible, and can be used in the same system if certain guidelines are followed. It is easy to use different mechanisms for unrelated data structures. It is almost as easy

---

[3] Note that the ability to interrupt a remote processor at *high* priority is required for operations such as TLB shootdown[17–19].

[4] Nested interrupt-level RIs must be performed with care. They may require large interrupt stacks, and raise the possibility that a processor may spin for an unbounded amount of time while the processor to which it made an interrupt-level RI services unrelated interrupts.

to use remote access and process-level RI on the same data structure, provided that the synchronization mechanisms are also compatible. Either spin locks or semaphores can be used, though performance may suffer if a process performs a process-level RI (and blocks) while holding a spin lock, and the scheduling mechanisms that underlie semaphores will need to be implemented with atomic instructions or interrupt-level RI to work across nodes.

We can also use remote access and interrupt-level RI on the same data structure, but only with a 'hybrid' lock that uses both interrupt masking *and* spinning (such locks are always required for data that are accessed both by remote processors and by interrupt-level routines on their home processor). Pseudocode for a simple hybrid lock appears in Figure 1. Its key task is to ensure that an interrupt-level routine that attempts to acquire a lock is given top priority, and will therefore succeed after a bounded amount of spinning. For the sake of deadlock avoidance, we must refrain from performing an interrupt-level RI or acquiring a remote lock while holding a local lock, because we mask out interrupts.

```
acquire:
    if on home node
        nested_maskings +:= 1
        disable interrupts
        lock.urgently_needed := true
        loop
            exit if TAS (lock)
            pause
        lock.urgently_needed := false
    else
        loop
            exit if not lock.urgently_needed and then TAS (lock)
            pause
release:
    lock := 0
    if on home node
        nested_maskings -:= 1
        if nested_maskings = 0 then
            enable interrupts
```

Figure 1. Pseudo-code for a 'hybrid' lock that can protect data structures manipulated both by remote memory access and by interrupt-level RI. Nested_maskings is a per-processor private counter. TAS is assumed to atomically test and set a flag bit in the lock. If contention is expected to be high, one should instead use a version of the algorithm that spins only on local locations[30]

Finally, we can use both interrupt-level and process-level RI on the same data structure so long as we respect the deadlock-avoidance rule: it must always be possible to perform incoming invocations while waiting for an outgoing invocation. In particular, a process cannot request a process-level RI while it has locked out interrupt-level RIs in order to touch a data structure that is shared between normal and interrupt-level code.

4. REMOTE ACCESS/INVOCATION TRADE-OFFS

In this Section we consider four dimensions along which to compare remote memory access and remote invocation. The first of these dimensions is the latency of an operation in isolation, based on architectural constants. The second dimension is the impact on local operations of the organization and synchronization required for compatibility with remote operations. The third dimension is contention and throughput. The fourth dimension is the extent to which kernel–kernel communication alternatives complement or clash conceptually with the basic organizational structure of the kernel.

4.1. Direct costs of remote operations

A first cut at deciding between remote memory access or remote invocation for a particular operation can be made on the basis of the latency incurred under the two different implementations. For example, consider an operation O invoked from node i that needs to perform n memory accesses to a data structure on another node j. We can perform those memory accesses remotely from node i, or we can perform a remote invocation to node j, where they will be performed locally. For the sake of simplicity, suppose that O must perform a fixed number of local memory accesses (e.g. to stack variables) and a fixed number of register–register operations regardless of whether it is executed on node i or on node j. If the remote/local memory access time ratio is R and the overhead of a remote invocation is C times the local memory access time, then it will be cheaper to implement O via remote memory access when $(R-1)n<C$.[5]

The fixed overhead of remote invocation, independent of operation complexity, suggests that operations requiring a large amount of time should be implemented via remote invocation (all other things being equal).[6] Back-of-the-envelope calculations should suffice in many cases to evaluate the performance trade-off. Many operations are simple enough to make a rough guess of memory access counts possible, and few are critical enough to require a truly definitive answer. For critical operations, however, experimentation is necessary.

4.2. Indirect costs for local operations

Kernel operations will often be organized differently when performed via remote invocation, instead of remote memory access. They may require context available on the invoking node to be packaged into parameters. They may be rearranged to increase node locality, so that accesses to data on the invoking and target processor are not interleaved. Most importantly, perhaps, the use of process-level RI for *all* remote accesses to a particular data structure may allow that data structure to be implemented without explicit synchronization, depending instead on a lack of pre-emption within the kernel to provide implicit synchronization. Explicit synchronization is still required, of course, for any data structures that a process needs to keep locked on the local node during an outgoing

[5] This formula assumes, of course, that we are executing on a NUMA machine. If the hardware provides coherent caches, then only the first reference to data in a particular cache line will incur a remote access cost.

[6] We did not include the cost of parameter passing in our simple analysis. Nearly all our kernel operations take only one or two parameters, and require no reply other than a notice of completion, so our assumption of a fixed cost for remote invocation is realistic.

process-level RI. Because interrupt-level RI handlers can always execute unless explicitly locked out (even if they interrupt normal kernel activity), explicit synchronization is also required for data structures accessible to interrupt routines. Depending on architectural parameters, locks that inhibit interrupt-level RI may be faster than semaphores or process-level spin locks; in particular, they are slightly faster on the Butterfly Plus.

Avoiding explicit synchronization can improve the speed not only of remote operations but also of the (presumably more frequent) local operations that access the same data structure. The impact of explicit synchronization on local operations is easy to underestimate. The case study in the following Section includes operations in which lock acquisition and release account for 49% of the total execution time (in the absence of contention). This overhead could probably be reduced by a coarser granularity of locking, but only with considerable effort: fine-grain locking introduces fewer opportunities for deadlock, and allows for greater concurrency.

On a machine in which nodes are bus-based multiprocessors (with parallel execution of one local copy of the kernel), explicit synchronization may be required for certain data structures even if remote invocation is always used for operations on those data structures requested by other nodes. On the other hand, clever use of atomic fetch-and-Φ operations to create concurrent no-wait data structures[20] may allow explicit synchronization to be omitted even for data structures whose operations are implemented via remote memory access.[7]

If remote memory accesses are used for many data structures, large portions of the kernel data space on other processors will need to be mapped into each instance of the kernel. Since virtual address space is limited (at least on 32-bit processors), this mapping may make it difficult to scale the kernel design to very large machines, particularly if kernel operations must also be able to access the full range of virtual addresses in the currently running user process. Mapping remote kernel data structures on demand is likely to cost more than sending a request for remote invocation. Mechanisms to cache information about kernel data structures may be limited in their effectiveness by the lack of address locality. Systems that map remote kernel data into a separate kernel–kernel address space[21] may waste large amounts of time switching back and forth between the kernel–kernel space and the various user-kernel spaces.

4.3. Competition for processor and memory cycles

Operations that access a central resource must serialize at some level. Operations implemented via remote invocation serialize on the processor that executes those operations. Operations implemented via remote memory accesses serialize at the memory. Because an operation does more than access shared data, there is more opportunity with remote memory access for overlapped computation. Operations implemented via remote memory access may still serialize if they compete for a common coarse-grain lock, but operations implemented via remote invocation will serialize even if they have no data in common whatsoever.

[7] A parallel data structure is said to be *wait-free* if each of its access functions is guaranteed to complete within a bounded amount of time. As designed by Herlihy and others, wait-free data structures employ fetch-and-Φ operations (e.g. compare_and_swap, fetch_and_store, fetch_and_add, and the newer load_linked and store_conditional provided by the MIPS R4000 and DEC Alpha processors) to divert competing processors into different code paths, without ever spinning or blocking.

If competition for a shared resource is high enough to have a noticeable impact on overall system throughput, it will be desirable to reorganize the kernel to eliminate the bottleneck. The amount of competition that can occur before inducing a bottleneck may be slightly larger with remote memory access because of the ability to overlap computation. Even in the absence of bottlenecks, we expect that operations on a shared data structure will occasionally conflict in time. The coarser the granularity of the resulting serialization, the higher the expected variance in completion time will be. The desire for predictability in kernel operations suggests that operations requiring a large amount of time should be implemented via remote memory access, in order to serialize at the memory instead of the processor. This suggestion conflicts with the desire to minimize operation latency, as described above; it may not be possible to minimize latency and variance simultaneously.

Given that the requestor of an interrupt-level RI busy-waits for notice of completion, the desirability of remote invocation in comparison to remote memory access may then depend on whether we are interested in latency or throughput. An interrupt-level RI may execute quickly from the requesting processor's point of view, but in the absence of unrelated interrupts it ties up both the requesting and responding processors for the duration of the requested operation.

4.4. Compatibility with the conceptual model of kernel organization

There are two broad classes of kernel organization, identified by Lauer and Needham[22] as the message-based and procedure-based approaches (see Figure 2). In a procedure-based kernel there is no fundamental distinction between a process in user space and a process in the kernel. Each user program is represented by a process that enters the kernel via traps, performs kernel operations and returns to user space. Kernel resources are represented by data structures shared between processes. In a message-based kernel each major kernel resource is represented by a separate kernel process, and a typical kernel operation requires communication (via queues or message-passing) among the set of kernel processes that represent the resources needed by the operation.

We have found the choice between the procedure-based and message-based organizations to have a more pervasive impact on the rest of the operating system than any other single design decision. (Psyche is procedure-based, but we have built message-based kernels as well.) Both approaches can be aesthetically appealing, depending on one's point of view. The procedure-based organization presents a uniform model for user- and kernel-level processes, and closely mimics the hardware organization of a UMA multiprocessor. The message-based organization, on the other hand, leads to a compartmentalization of the kernel in which all synchronization is subsumed by message-passing. The message-based organization closely mimics the hardware organization of a distributed-memory multicomputer. Because it minimizes context switching, the procedure-based organization is likely to perform better on a machine with uniform memory[23]. The message-based organization may be easier to debug[24]. Most Unix kernels are procedure-based. Demos[25] and Minix[26] are message-based.

Remote invocation seems to be more in keeping with the message-based approach to kernel design. Remote memory access seems appropriate to the procedure-based approach. When porting an operating system from some other environment, the pre-existence of a procedure-based or message-based bias in the implementation may suggest the use of the corresponding mechanism for kernel–kernel communication, though mixed

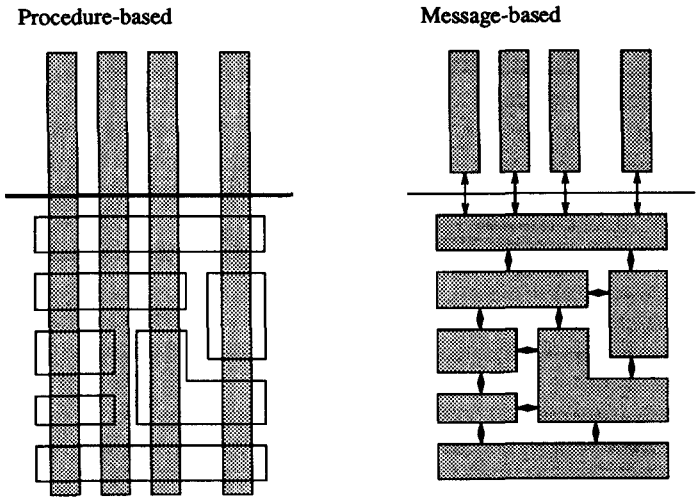

Figure 2. Alternative kernel organizations. Shaded boxes represent processes; unshaded boxes represent data abstractions

approaches are possible[27]. If a procedure-based kernel is used on a uniprocessor, the lack of context switching in the kernel may obviate the need for explicit synchronization in many cases. Extending the procedure-based approach to include remote memory access may then incur substantial new costs for locks. On a machine with multiprocessor nodes, however, such locking may already be necessary.

5. CASE STUDY: PSYCHE ON THE BBN BUTTERFLY

Our experimentation with alternative communication mechanisms took place in the kernel of the Psyche operating system[21] running on a BBN Butterfly Plus multiprocessor[28]. The Psyche implementation is written in C++, and uses shared memory as the primary kernel communication mechanism. The Psyche kernel was modified to provide performance figures for remote invocation as well, with and without fine-grain locking. Our results are based on experiments using these modified versions of the kernel.

The Psyche implementation displays a high degree of node locality. The kernel object representing an application-level abstraction (process, address space, memory segment) is allocated and initialized on a single node, either on the node where the creation request originated or another specified node. Other kernel data structures associated with a node's local resources are also local to that node. It is quite common, therefore, for a kernel operation not to need access to data on another node. In those cases where kernel–kernel communication is required, local accesses still tend to dominate.

Among those kernel operations requiring access to data on more than one node, it was common in the original Psyche implementation for remote memory accesses to occur at several different times in the course of the operation. In an attempt to optimize our remote procedure call mechanism we found that many, though not all, of these accesses could be grouped together by restructuring the code, thereby permitting them to be implemented by a single remote invocation.

5.1. Fundamental costs

The Butterfly Plus is a NUMA machine with one processor per node, no caches and a remote-to-local memory access time ratio of approximately 12:1. The average measured execution time[29] of an instruction to read a 32-bit remote memory location using register indirect addressing is 6.88 μs; the corresponding instruction to read local memory takes 0.518 μs. The time to write memory is slightly lower: 4.27 μs and 0.398 μs for remote and local memory, respectively.[8] Microcoded support for block copy operations can be used to move large amounts of data between nodes in about a fifth of the time required for a word-by-word copy (345 μs instead of 1.76 ms for 1K byte). None of the experiments reported below moved enough data to need this operation.

Our remote invocation mechanisms rely on remote memory access and on the ability of one processor to cause an interrupt on another. A processor that requires a remote operation writes an operation code and any necessary parameters into a preallocated local buffer. It then writes a pointer to that buffer into a reserved location on the remote node, and issues a remote interrupt. The requesting processor then spins on an 'operation received' flag in the local buffer. When the target processor receives the interrupt, it checks its reserved location to obtain a pointer to the buffer. It sets the 'operation received' flag, at which point the requesting kernel process either blocks (in the case of a process-level RI), or begins to spin on an 'operation completed' flag (in the case of an interrupt-level RI). If another request from a different node overwrites the original request, the second request will be serviced instead. After a fixed period of unsuccessful waiting for the 'operation received' flag, the first processor will time-out and resend its request. In case a processor's request is completed just before a resend, receiving processors ignore request buffers whose 'operation received' flag is already set.

The mechanism to trigger a remote invocation is *optimistic*, in that it minimizes latency in the absence of contention and admits starvation in the presence of contention. The average latency of an interrupt-level RI, excluding parameter copying and operation costs, is 56 μs (measured by timing a large number of consecutive invocations, and dividing). An earlier, non-optimistic, implementation relied on microcoded atomic queues, but these required approximately 60 μs for the enqueue and dequeue operations alone. The average latency of a process-level RI, again excluding parameter copying and operation costs, is about 421 μs. Process-level RI could be made faster with some more hand optimization, but it is unlikely that we could get it under 300 μs. Interrupt-level RI is highly optimized; we see no way to make it significantly faster.

5.2. Explicit synchronization

Psyche uses spin locks to synchronize access to kernel data structures. To achieve a high degree of concurrency within the kernel, access to each component data structure requires possession of a lock. This approach admits simultaneous operations on different parts of the same kernel data structure, but also introduces a large number of synchronization points in the kernel. Mapping a memory segment into the current address space, for

[8] The original Butterfly architecture had a remote-to-local access time ratio of approximately 5:1. The speed of local memory was significantly improved in the Butterfly Plus, with only a modest improvement in the speed of remote accesses.

example, can require up to nine lock acquisitions. Creating a segment can require 38 lock acquisitions. A cheap implementation of locks is critical.

We use a test-and-test&set lock[30] to minimize latency in the absence of contention. If the lock is in local memory, we use the native MC68020 TAS instruction. Otherwise, we use a more expensive atomic instruction implemented in microcode on the Butterfly. (TAS is not supported on remote locations.) The slight cost of checking to see whether the lock is local (involving a few bit operations on its virtual address) is more than balanced by the use of a faster atomic primitive in the common, local case. Moreover, this cost must also be incurred when using remote invocation, rather than remote access, to determine the node to interrupt.

A lock can be acquired and released manually by calling in-line subroutines, or automatically using features of C++. The automatic approach passes the lock as an initialization parameter to a dummy variable in the block of code to be protected. The constructor for the dummy variable acquires the lock; the destructor (called by the compiler automatically at the end of a scope) releases it. Constructor-based critical sections are slightly slower, but make it harder to forget to release a lock. Manual locking is used for critical sections that span function boundaries or that do not properly nest. Acquiring and releasing a local lock manually requires a minimum of 5 μs, and may require as much as 10 μs, depending on instruction alignment, the ability of the compiler to exploit common subexpressions and the number of registers available for temporary variables. Acquiring and releasing a remote lock manually requires 38 to 45 μs. The additional time required to acquire and release a lock through constructors is about 1 to 3 μs. Synchronization using remote locks is expensive because the Butterfly's microcoded atomic operations are significantly more costly than native processor instructions. Extensive use of no-wait data structures[20] might reduce the need for fine-grain locks, but would probably not be faster, given the cost of atomic operations.

The performance of semaphores is indistinguishable from that of spin locks in the absence of lock competition; the only thing that differs on the code path is a check in the V operation to determine whether any processes are waiting. Disabling and enabling of interrupt-level RIs is slightly cheaper: a critical section counter can be incremented and then decremented again in just over 5 μs.

5.3. Impact on the cost of kernel operations

To assess the impact of alternative kernel–kernel communication mechanisms on the performance of typical kernel operations, we measured the time to perform several such operations via local memory access, remote memory access, and remote invocation, with and without explicit synchronization. The results appear in Table 1. The first three lines give times for low-latency operations. The first of these inserts and then removes an element in a doubly linked list-based queue; the second and third search for elements in a list. Remote invocations for all three are implemented in interrupt-level routines. The last three lines give times for high-latency operations: creating a segment, mapping a segment and adding a new process to an address space. Remote invocations for these are implemented at the process level. All times are accurate to about ± 3 in the third significant digit. Times for the low-latency operations are averaged over 10,000 consecutive trials. They are stable in any particular kernel load image, but fluctuate with

Table 1. Latency of kernel operations

Operation		Local access locking		Remote access locking		Remote inv.* locking	
		on	off	on	off	on	off
enqueue + dequeue		42.4	21.6	247	154	197	174
find last in list of 5	(μs)	25.0	16.1	131	87.6	115	96.7
find last in list of 10		40.6	30.5	211	169	125	105
create segment		6.20	5.69	14.8	13.1	7.42	6.88
map segment	(ms)	0.96	0.86	3.05	2.62	1.94	1.77
create process		1.43	1.35	3.30	3.04	1.89	1.75

*Interrupt-level remote invocation is used for low-latency operations (top half of table); process-level remote invocation is used for high-latency operations (bottom half of table).

changes in instruction alignment. They are also sensitive to the context in which they appear, due to variations in the success of compiler optimizations.[9] Times for the high-latency operations are averaged over a small number of consecutive trials in several separate runs.

Times in columns 1 and 2 are with all data on the local node. Times in columns 3 to 6 are with target data on a remote node, but with temporary variables still in the local stack. Columns 1 and 3 give times for the original, unmodified version of the Psyche kernel. Column 2 indicates what operations would cost if synchronization were achieved through lack of context switching, with no direct access to remote data structures. Column 4 indicates what operations on remote data structures would cost if subsumed in some other operation with coarse-grain locking. Column 6 indicates what remote operations would cost if the data structures they manipulate were always accessed via remote invocation, with no explicit synchronization required beyond recognizing that invocations were not disabled. Column 5 indicates the cost of performing operations via remote invocation in a hybrid kernel that continues to rely on locks.

5.3.1 Explicit synchronization

We can calculate the fraction of the cost of each of our kernel operations due to synchronization by comparing figures in adjacent columns of Table 1. The comparisons appear in Figure 3. Synchronization clearly dominates the cost of simple operations on queues, contributing in some cases nearly 50% for local operations and 40% for remote operations. Although less overwhelming, synchronization impacts more complex operations as well, due to the use of fine-grain locks. Segment creation requires acquiring and releasing approximately 38 constructor-based locks, contributing over 500 μs, or 8%, to the cost in the local case and 1.7 ms, or 11%, to the cost in the remote case. The overhead of fine-grain locking combined with automatically acquired locks is clearly significant. More to the point, in the case of process-level data structures this overhead is imposed on local access simply to *permit* remote access. We could reduce the cost of synchronization by locking data structures at a coarser grain. This change would reduce

[9] We have read through the assembly language output of the compiler to make sure the optimizer is not removing any apparently useless code of importance to the timing tests.

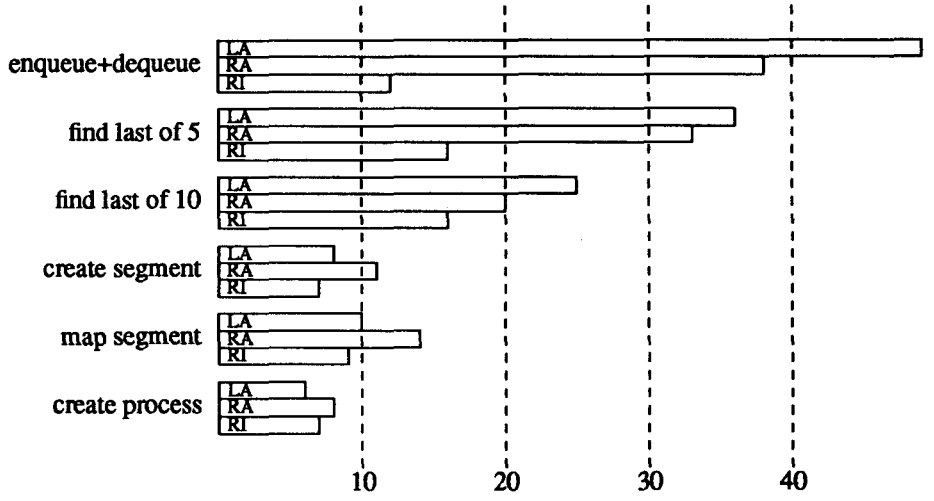

Figure 3. Percentage of latency due to explicit locking, for local access (LA), remote access (RA), and remote invocation (RI)

the number of locks required by a typical operation, but would simultaneously reduce the potential level of concurrency.

5.3.2 Remote references

We can assess the impact of remote memory references by comparing the cost of local and remote operations in Table 1. Figure 4 indicates the marginal cost of remote references for each of our kernel operations. Without locking, this marginal cost is 86% for a remote enqueue/dequeue operation pair; remote references exclusive of synchronization account for 54% of the cost even when locking is used (154 μs to perform the operation remotely excluding synchronization costs minus 21.6 μs to perform the operation locally over 247 μs total time). When searching for the tenth element in a list, remote references exclusive of synchronization account for two-thirds of the cost of the operation. Even for complex operations such as segment creation, which performs much of its work using stack variables, remote references account for over half of the total cost.

The overhead associated with explicit synchronization and remote references is a function of the complexity of the operation, while the overhead associated with remote invocation is fixed. In addition, if using process-level RI exclusively we can rely on implicit synchronization (non-pre-emption in the kernel), thereby reducing the cost of operations significantly. In Table 1 the times in the last three rows of column 6 are not only much faster than the corresponding times in column 3, they are in several cases close to the times in column 1; the ability to avoid lock acquisition and release almost hides the cost of remote invocation and parameter passing. Given that a remote memory access costs more than 6 μs more than a local access, the 60 μs overhead of an interrupt-level RI with a single parameter can be justified on the Butterfly Plus to avoid 11 remote references. The 421 μs overhead of a process-level RI can be justified

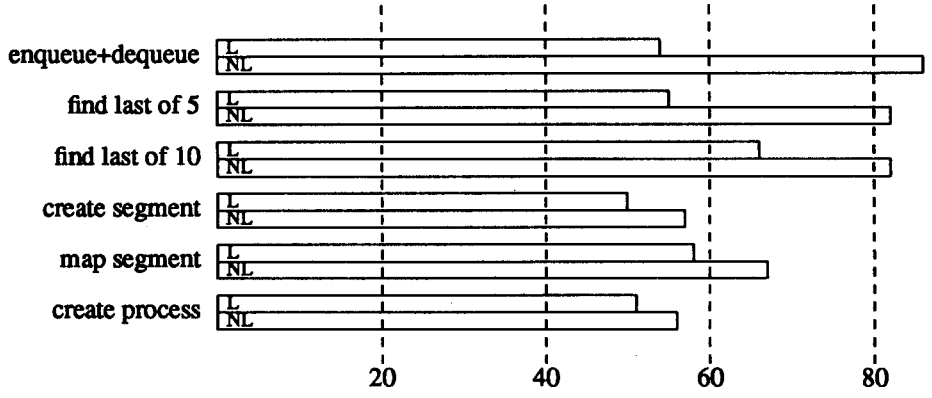

Figure 4. Remote access penalty for kernel operations, with (L) and without (NL) locking

to avoid 71-7k remote references, where k is the number of lock acquisitions that can be eliminated by exclusive use of process-level RI. If hand optimization were to yield an implementation of process-level RI that cost 300 μs, it could be justified to avoid 50-7k remote references.

5.3.3 Comparative latency of remote access and remote invocation

Figure 5 presents three different comparisons of the cost of remote invocation and the cost of remote memory access. The first bar in each group expresses column 6 in Table 1 as a percentage of column 3; it compares remote memory access with explicit locking to remote invocation without explicit locking. The second bar in each group expresses column 5 in Table 1 as a percentage of column 3; it includes locking overhead for both remote memory access and remote invocation, as would exist in a hybrid kernel. The third bar in each group expresses column 6 in Table 1 as a percentage of column 4; it considers the case in which the desired operation is subsumed in some other operation with coarse-grain locking.

Figures greater than 1 indicate scenarios in which remote memory access displays a lower latency. Based on this metric alone, our experiments would seem to indicate that remote memory access is justified only for the most trivial of operations. Other factors soften this conclusion, however, and make remote invocation less of a clear win than it might at first appear. In particular, our experiments highlight the extreme cases of operations simple enough to perform via interrupt-level RI, or complex enough to absorb the overhead of a process-level RI. For medium-size operations we may be unwilling to accept either the limitations of interrupt-level RI or the overhead of process-level RI. Moreover, even for tiny operations the *throughput* of interrupt-level RI may be unacceptably low, as discussed in the following Section.

5.3.4. Throughput for interrupt-level remote invocations

On the Butterfly Plus remote memory accesses steal bus cycles from the processor on which the memory resides, thereby slowing that processor's progress. Remote

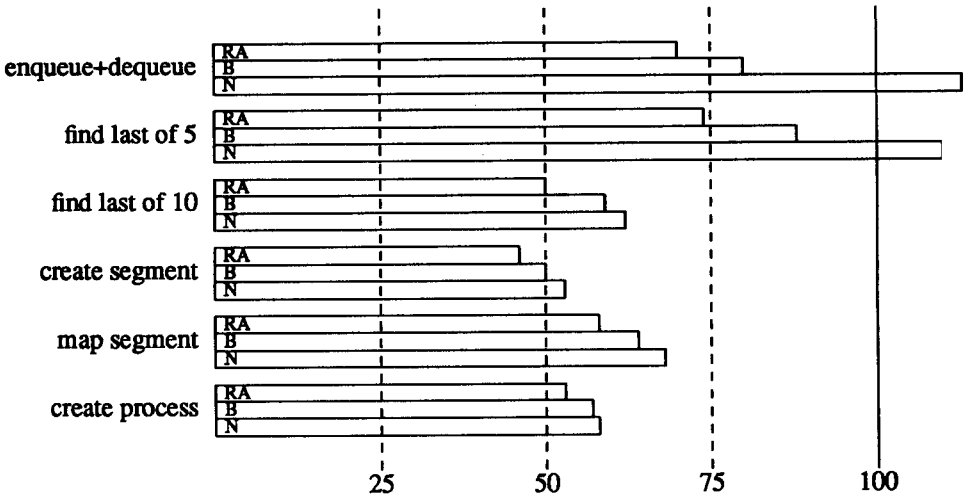

Figure 5. Remote invocation time as a percentage of remote memory access time, with locking in the case of the latter (RA), both (B), or neither (N)

invocations, however, steal the entire processor for the duration of the operation. In the case of interrupt-level RIs, parallelism is lost: the requesting processor remains idle until the invocation completes. In the case of process-level RIs, the requesting processor has the opportunity to do something else instead. These observations suggest that latency provides a reasonable measure of the performance of process-level RI, but that throughput must also be considered for interrupt-level RI.

We conducted a simple experiment to compare the impact of remote memory accesses and interrupt-level RIs on the progress of other computations on a given processor. We timed the slowdown of a compute-bound application on a processor subjected to remote memory accesses and interrupt-level RIs. For a set of low-latency operations whose aggregate latency was 19 ms when using remote memory access and 22 ms when using interrupt-level RI, we measured application slowdowns of 1 ms and 17 ms, respectively. From these figures we can see that interrupt-level RI significantly affects the throughput of the remote processor, whereas remote access does not. Thus, in all cases where the latency of the two alternatives is comparable, and even in some cases where the latency of remote access is higher, remote access would be preferred for reasons of throughput.

6. CONCLUSIONS

Architectural features strongly influence operating system design. The choice between remote invocation and remote access as the basic communication mechanism between kernels on a shared-memory multiprocessor is highly dependent on the cost of the remote invocation mechanism, the cost of the atomic operations used for synchronization, and the ratio of remote to local memory access time. On a cache-coherent machine it is also dependent on the cache line size, the degree to which lines are falsely shared, and the extent to which prefetching and caching effects can reduce the cost of typical kernel operations. Since the overhead associated with remote access scales with the operation,

while the overhead associated with remote invocation is fixed, there will for any machine be a break-even point above which process-level RI enjoys an increasingly compelling performance advantage. For smaller operations the operating system designer must weigh the issues of latency, throughput, conceptual appeal and the possibility of eliminating explicit synchronization in order to make a choice between remote access and the two forms of remote invocation.

Our experience with Psyche indicates that the natural node locality of kernel operations is sufficient to allow us to perform most large operations with only one or two remote invocations. Without this locality many invocations would be needed just to collect the data necessary to perform an operation. Under those circumstances remote access would be competitive with process-level RI, but performance would be poor; a kernel without node locality is not a reasonable option for the Butterfly architecture.

Our original decision to use remote memory access as the principal kernel–kernel communication mechanism was based primarily on the conceptual appeal of a uniform procedure-based organization across the entire machine. We underestimated the impact that locking would have on the cost of typical operations, and did not give adequate consideration to lengthy operations early in the design process. If we were to rebuild the kernel at this point, we would make more extensive use of process-level RI for lengthy operations. We would also attempt to identify data structures for which process-level RI alone would suffice, allowing us to eliminate explicit locking. Finally, we would attempt where possible to increase the granularity of our remaining locks, being careful to avoid the introduction of bottlenecks.

There are only a small number of cases in which interrupt-level RI is the mechanism of choice. The most plausible scenario arises with low-latency operations that cannot be performed via remote access. TLB shootdown is such an operation on most machines; instructions that manipulate the TLB cannot be invoked remotely. We also use interrupt-level RI for console I/O, and to implement our remote kernel debugging facility[21]. Interrupt-level RI may also be preferred over remote access when the target processor of a remote operation is idle, or when the latency of remote access is more than twice the latency of interrupt-level RI (e.g. for a memory-intensive operation on a machine in which remote memory is exceptionally slow). In both these latter scenarios interrupt-level RI will outperform remote memory access even in terms of throughput.

On the Butterfly Plus, remote invocation is relatively fast, explicit synchronization is costly and remote references are significantly more expensive than local references. Increases in processor speed relative to memory and interconnect latencies are likely to make remote references even more expensive in future machines. All but the shortest operations on the Butterfly Plus display lower latency with remote invocation than they do with remote memory access, and the disparity between the two options is likely to increase.

It is not yet clear whether the large-scale multiprocessors of the future will provide system-wide hardware cache coherence. Several projects are moving in that direction[5–8], but others are pursuing software-managed coherence beyond the bounds of a single bus[4,31,32]. NUMA machines are likely to be cheaper to build than their cache-coherent cousins, and recent studies suggest[33] that they can provide comparable performance for reasonable applications. Our results apply most directly to NUMA machines, but suggest that remote invocation should be attractive for kernel–kernel communication on cache-coherent machines as well. Deliberate exploitation of node locality in the kernel for a

cache-coherent machine would associate each data structure with a 'home node' on which that structure is accessed most often. As interprocessor communication becomes slower and slower relative to processor performance[34], operations that touch several cache lines that 'belong' to another processor might profitably be dispatched to that processor for execution rather than running locally. Further performance studies will be needed to quantify kernel–kernel communication trade-offs more precisely on large cache-coherent machines.

For small operations, interrupt-level RI can display lower latency than remote access, but it may display lower throughput as well, and is limited by the need to use coarse, interrupt-masking locks and to restore all data structures to a consistent state before performing an outgoing interrupt-level RI. Moreover, we strongly suspect that the desire to keep data structures out of the interrupt-level portion of the kernel will mean that some operations that are too small to absorb the overhead of process-level RI will still touch too much data to use interrupt-level RI. These observations imply that performance will be maximized if remote access is used for operations comprising up to a few dozen remote memory references, and process-level RI is used above this limit. Interrupt-level RI should be reserved for special cases.

ACKNOWLEDGEMENTS

Our thanks to Rob Fowler and to the referees for their helpful comments on this paper, and to Tim Becker for his invaluable assistance with experiments. An earlier version of this paper was presented at the Second USENIX Symposium on Experiences with Distributed and Multiprocessor Systems; this current version benefited significantly from discussions with symposium attendees, particularly Jim Gibson of BBN ACI. This research was supported by NSF grant no. CCR-9005633, NSF Institutional Infrastructure grant no. CDA-8822724, a DARPA/NASA Graduate Research Assistantship in Parallel Processing, the Federal University of Rio de Janeiro, and the Brazilian National Research Council.

REFERENCES

1. W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken and T. Blackadar, 'Performance measurements on a 128-node Butterfly parallel processor,' *Proceedings of the 1985 International Conference on Parallel Processing*, 20–23 August 1985, pp. 531–540.
2. G. R. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, 'The IBM Research parallel processor prototype (RP3): introduction and architecture,' *Proceedings of the 1985 International Conference on Parallel Processing*, 20–23 August 1985, pp. 764–771.
3. D. J. Kuck, E. S. Davidson, D. H. Lawrie and A. H. Sameh, 'Parallel supercomputing today and the cedar approach,' *Science*, **231** 967–974, (1986).
4. Z. G. Vranesic, M. Stumm, D. M. Lewis and R. White, 'Hector: a hierarchically structured shared-memory multiprocessor,' *Computer*, **24**(1), 72–79, (1991).
5. D. V. James, A. T. Laundrie, S. Gjessing and G. S. Sohi, 'Scalable coherent interface,' *Computer*, **23**(6) 74–77, (1990).
6. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, 'The directory-based cache coherence protocol for the DASH multiprocessor,' *Proceedings of the Seventeenth International Symposium on Computer Architecture*, 28–31 May 1990, pp. 148–159. (in *CAN 18:2*).
7. A. Agarwal, B. Lim, D. Kranz and J. Kubiatowicz, 'APRIL: a processor architecture for multiprocessing,' *Proceedings of the Seventeenth International Symposium on Computer Architecture*, 28–31 May 1990, pp. 104–114 (in *CAN 18:2*).

8. T. H. Dunigan, 'Kendall square multiprocessor: early experiences and performance,' ORNL/TM-12065, Oak Ridge National Laboratory, May 1992.
9. T. E. Anderson, H. M. Levy, B. N. Bershad and E. D. Lazowska, 'The interaction of architecture and operating system design,' *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 8–11 April 1991, pp. 108–120 (in *ACM SIGARCH Computer Architecture News 19:2, ACM SIGOPS Operating Systems Review 25* (special issue) and *ACM SIGPLAN Notices 26:4*).
10. M. J. Bach and S. J. Buroff, 'Multiprocessor Unix systems,' *Bell Lab. Tech. J.*, **63**(8), 1733–1750 (1984).
11. J. Boykin and A. Langerman, 'The Parallelization of Mach/4.3BSD: design philosophy and performance analysis,' *Proceedings of the First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, 5–6 October 1989, pp. 105–126.
12. M. D. Campbell, R. Holt and J. Slice, 'Lock granularity tuning mechanisms in SVR4/MP,' *Proceedings of the Second USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 21–22 March 1991, pp. 221–228.
13. N. Paciorek, S. LoVerso and A. Langerman, 'Debugging multiprocessor operating system kernels,' *Proceedings of the Second USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 21–22 March 1991, pp. 185–202.
14. R. Bryant, H. Chang and B. Rosenburg, 'Experience developing the RP3 operating system,' *Proceedings of the Second USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 21–22 March 1991, pp. 1–18.
15. D. W. Clark and J. S. Emer, 'Performance of the VAX-11/780 translation buffer: simulation and measurement,' *ACM Trans. Compu. Syst.* 3(1), 31–62, (1985).
16. S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, The Addison-Wesley Publishing Company, Reading, MA, 1989.
17. D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill and R. V. Baron, 'Translation lookaside buffer consistency: a software approach,' *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 3–6 April 1989, pp. 113–122.
18. A. L. Cox and R. J. Fowler, 'The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with PLATINUM,' *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3–6 December 1989, pp. 32–44 (in *ACM SIGOPS Operating Systems Review 23:5*).
19. B. Rosenburg, 'Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors,' *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3–6 December 1989, pp. 137–146 (in *ACM SIGOPS Operating Systems Review 23:5*).
20. M. Herlihy, 'Wait-free synchronization,' *ACM Trans. Programming Lang. Syst.*, **13**(1) 124–149, (1991).
21. M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos and N. G. Smithline, *'Implementation issues for the Psyche multiprocessor operating system,' Comput. Syst.* 3(1), 101–137, (1990). Earlier version presented at the *First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL, 5–6 October, 1989.
22. H. C. Lauer and R. M. Needham, 'On the duality of operating system structures,' *ACM SIGOPS Operating Systems Review*, **13**(2), 3–19, (1979). Originally presented at the *Second International Symposium on Operating Systems*, October 1978.
23. D. Clark, 'The structuring of systems using upcalls,' *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1–4 December 1985, pp. 171–180 (in *ACM SIGOPS Operating Systems Review 19:5*).
24. R. A. Finkel, M. L. Scott, Y. Artsy and H. Chang, 'Experience with Charlotte: simplicity and function in a distributed operating system,' IEEE *Trans.* **SE-15**(6), 676–685, (1989).
25. F. Baskett, J. H. Howard and J. T. Montague, 'Task communication in Demos,' *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, November 1977, pp. 23–31.

26. A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
27. T. J. LeBlanc, J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl and P. C. Dibble, 'The Elmwood multiprocessor operating system,' *Software—Practice and Experience*, **19**(11), 1029–1056, (1989).
28. BBN Advanced Computers Incorporated, *Inside the Butterfly Plus*, Cambridge, MA, 16 October 1987.
29. A. L. Cox, R. J. Fowler and J. E. Veenstra, 'Interprocessor invocation on a NUMA multiprocessor,' TR 356, Computer Science Department, University of Rochester, October 1990.
30. J. M. Mellor-Crummey and M. L. Scott, 'Algorithms for scalable synchronization on shared-memory multiprocessors,' *ACM Trans. Compu. Syst.*, **9**(1), 21–65, (1991).
31. R. Bisiani and M. Ravishankar, 'PLUS: a distributed shared-memory system,' *Proceedings of the Seventeenth International Symposium on Computer Architecture*, 28–31 May 1990, pp. 115–124 (in *CAN 18:2*).
32. D. R. Cheriton, H. A. Goosen and P. D. Boyle, 'Paradigm: a highly scalable shared-memory multicomputer architecture,' *Computer*, February 1991, pp. 33–46.
33. W. J. Bolosky and M. L. Scott, 'A trace-based comparison of shared memory multiprocessor architectures,' TR 432, Computer Science Department, University of Rochester, July 1992.
34. E. P. Markatos and T. J. LeBlanc, 'Shared-memory multiprocessor trends and the implications for parallel program performance,' TR 420, Computer Science Department, University of Rochester, May 1992.