2nd USENIX Symp. on Experiences with Distributed and Multiprocessor Systems, Atlanta, GA, March 1991, pp. 105–116

Kernel-Kernel Communication in a Shared-Memory Multiprocessor[†]

Eliseu M. Chaves, Jr.[*]
Thomas J. LeBlanc
Brian D. Marsh
Michael L. Scott

Computer Science Department
University of Rochester
Rochester, New York 14627
(716) 275-9491
{chaves,leblanc,marsh,scott}@cs.rochester.edu

Abstract

In the standard kernel organization on a shared-memory 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 inter-kernel communication can be used in a NonUniform Memory Access (NUMA) multiprocessor, although the performance tradeoffs may not be apparent in advance.

In this paper we compare the use of remote access and remote invocation in the kernel of a NUMA multiprocessor operating system. We discuss the issues and architectural features that must be considered when choosing an inter-kernel communication scheme, and describe a series of experiments on the BBN Butterfly designed to empirically evaluate the tradeoffs between remote invocation and remote memory access. We conclude that the Butterfly architecture is biased towards the use of remote invocation for most kernel operations, but that a small set of frequently executed operations can benefit from the use of remote access.

1. Introduction

An important consideration in the design of any multiprocessor operating system kernel is the host architecture, which will often dictate how kernel functionality is distributed among processors, the form of inter-kernel communication, the layout of kernel data structures, and the need for synchronization. For example, in uniform memory access (UMA) multiprocessors, it is easy for all processors to share the code and data of the operating system. 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 use an alternative organization, wherein the kernel data is distributed among the processors, each of which executes

[†] 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.

[*] Visiting Scientist on leave from the Universidade Federal do Rio de Janeiro, Brazil.

a copy of the kernel. Each kernel performs operations on local resources directly and uses remote invocation to request operations on remote resources. Nonpreemption of the kernel (other than by interrupt handlers) provides 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 straightforward to port a uniprocessor implementation to a shared-memory multiprocessor.[1] Having each processor execute its own operations directly on shared memory is also very efficient. 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 don't 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, since all contention for data structures is local, and can be managed using nonpreemption. 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.

NonUniform Memory Access (NUMA) multiprocessors, such as the BBN Butterfly [2], IBM 8CE [9], and IBM RP3 [15] have properties in common with both shared-memory multiprocessors and distributed-memory multicomputers. Since NUMA multiprocessors support both remote memory access and remote invocation, kernel data can be accessed using either mechanism. The performance tradeoffs between the use of remote invocation and remote access in the kernel of a NUMA machine are not well understood, however, and depend both on the specific architecture and on the overall design of the operating system.

In this paper we explore the tradeoffs between remote access and remote invocation in the kernel, and the related issues of locality, synchronization, and contention. Our observations about the tradeoffs are made concrete through a series of experiments comparing the direct and indirect costs associated with each design decision. We conclude with a summary of the relationship between architectural characteristics and kernel organization for NUMA multiprocessors.

2. Kernel-Kernel Communication Options

We consider a machine organization consisting of a collection of *nodes*, each of which contains memory and one or more processors. Each processor can access all the memory on the machine, but it can access the memory of the local node more quickly than the memory of 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. There are three principal classes of implementation alternatives:

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.

---

[1] Several versions of Unix have been ported to multiprocessors simply by protecting operating system data structures with semaphores [3]. An alternative approach is to use a master/slave organization wherein all kernel calls are executed on a single node; other nodes contain only a trap handler and a remote invocation mechanism. BBN's nX version of Unix uses this approach, as do the Unix portions of Mach [1], from which nX was derived.

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.

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 a lower-level system using page faults.

In evaluating the tradeoffs between these three options, we distinguish between *node locality* and *address locality*. Address locality captures the traditional notion of spatial locality in sequential programs. We say that a program (e.g. the kernel) displays a high degree of address locality if most of the memory locations accessed over some moderate span of time lie within a small set of dense address ranges. Node locality, by contrast, captures the notion of physical locality in a NUMA multiprocessor. We say that the kernel displays a high degree of node locality if most operations can be performed primarily using local memory references on some node.

Whatever the mechanism(s) used to communicate between instances of the kernel, performance clearly depends on the ability to maximize node locality. Any operating system that spends a large fraction of its time on operations that require interaction between nodes is unlikely to perform well. It seems reasonable to expect, and our experience confirms, that a substantial amount of node locality can in fact be obtained. This implies that most memory accesses will be local even when using remote memory accesses for kernel-kernel 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. There are several reasons for this difficulty. Kernels operate on behalf of a potentially large number of user processes, whose actions are generally unrelated to each other. To the extent that they are related, the most pronounced effect is likely not to be continuity of working set across context switches, but rather fragmentation of the working set of any particular process, as it incorporates common data structures. Typical kernel construction techniques rely heavily on pointer-based linked data structures, the pieces of which are often dynamically allocated.

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 in physical proximity, casting doubt on the utility of bulk data transfer for the implementation of kernel-kernel communication. Of course, we have not deliberately attempted to organize data structures to maximize address locality in any of the systems we have built, nor are we aware of any attempts to do so in other projects. It is therefore possible that the lack of address locality is simply an artifact of avoidable design decisions. Large parts of the PLATINUM kernel [6] are implemented on top of a "coherent memory" system that replicates and migrates data in response to page faults. Experiments with PLATINUM may eventually lead to a better understanding of the utility of bulk data transfer in the kernel.

In the remainder of this section, and in the case study that follows, we focus on the choice between remote memory access and 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 labor among processes in the kernel. Ultimately, we argue in favor of a mixture of both mechanisms, since no one mechanism will be the best choice for all operations.

2.1. Direct Costs of Remote Operations

A reasonable 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 $(R-1)n<C$.

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).[2] Back of the envelope calculations should suffice in many cases to evaluate the performance tradeoff. 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.

2.2. Indirect Costs for Local Operations

An important factor that we ignored in the above comparison based on latency is that operations will often be organized differently when performed via remote invocation. They may require context available on the invoking node to be packaged into parameters. They may be reorganized in order to segregate accesses to data on the invoking processor into code that can be executed before or after the remote invocation. Most important, perhaps, the use of remote invocation for *all* accesses to a particular data structure may allow that data structure to be implemented without explicit synchronization, depending instead on the implicit synchronization available via lack of context-switching as in a uniprocessor kernel. Although preemption is still possible from interrupt handlers, the cost of disabling interrupts is typically much lower than the cost of explicit synchronization.

Avoiding explicit synchronization can improve the speed not only of the 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. We will see operations in our case study 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 requires less thought and allows greater concurrency.

On a machine in which individual nodes are 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 fetch-and-Φ operations to create concurrent no-wait data structures [10, 13] may allow explicit synchronization to be omitted even for data structures whose operations are implemented via remote memory access.

---

[2] We did not include the cost of parameter passing in our simple analysis. Nearly all our kernel operations take only one parameter, and the reply value is used to signal completion of the operation, so our assumption of a fixed cost for remote invocation is realistic.

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, 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 kernel-kernel data into a separate kernel address space [16] may waste large amounts of time switching back and forth between the kernel-kernel space and the various user-kernel spaces.

2.3. 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 memory, 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.

If competition for a shared resource is high enough to have a noticeable impact on overall system throughput it will clearly 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 simultaneously to minimize latency and variance.

2.4. Compatibility With the Conceptual Model of Kernel Organization

There are two broad classes of kernel organization, which we refer to as the horizontal and vertical approaches. These alternatives correspond roughly to the message-based and procedure-based approaches, respectively, identified by Lauer and Needham in their 1978 paper [11]. In a vertical 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 horizontal 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.

Both approaches to kernel organization can be aesthetically appealing, depending on one's point of view. The vertical organization presents a uniform model for user- and kernel-level processes, and closely mimics the hardware organization of an UMA multiprocessor. The horizontal organization, on the other hand, leads to a compartmentalization of the kernel in which all synchronization is subsumed by message passing. The horizontal organization closely mimics the hardware organization of a distributed-memory multicomputer. Because it minimizes context switching, the vertical organization is likely to perform better on a machine with uniform memory [5]. The horizontal organization may be easier to debug [8]. Most Unix kernels are vertical. Demos [4] and Minix [17] are horizontal.

Remote invocation seems to be more in keeping with the horizontal approach to kernel design. Remote memory access seems appropriate to the vertical approach. If porting an operating system from some other environment, the pre-existence of a vertical or horizontal bias in the implementation may suggest the use of the corresponding mechanism for kernel-kernel communication, though mixed approaches are possible [12]. If a vertical 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 vertical 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.

3. Case Study: Psyche on the BBN Butterfly

Our experimentation with alternative communication mechanisms took place in the kernel of the Psyche operating system [16] running on a BBN Butterfly Plus multiprocessor [2]. 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 basic abstraction provided by Psyche is the *realm*, a passive object containing code and data. A *process* is a thread of control representing concurrent activity within an application. Processes are created, destroyed, and scheduled by user-level code, without requiring kernel intervention. User-level processes interact with one another by invoking realm operations. Processes are executed by *virtual processors*, or *activations*. The kernel time-slices the processor among the activations located at a node. Activations execute in address spaces known as *protection domains*, and obtain access to realms by means of an *open* operation that maps a realm into the caller's domain.

The implementation of the Psyche abstractions favors node locality. The kernel object representing an application-level abstraction is allocated and initialized on a single node, either on the node where the 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 *protected procedure call* mechanism (a form of RPC) we found that many, though not all, of these accesses could be grouped together by re-structuring the code, thereby permitting them to be implemented by a single remote invocation.

3.1. Fundamental Costs

The Butterfly Plus is a NUMA machine with a remote:local memory access time ratio of approximately 12:1. The average measured execution time [7] 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.[3] Microcoded support for block copy operations can be used to move large amounts data between nodes in about a fifth of the time

[3] 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.

required for a word-by-word copy (345 μs instead of 1.76 ms for 1K bytes). None of the experiments reported below moved enough data to need this operation.

Our remote invocation mechanism relies 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 a "operation received" flag in the local buffer. When the remote 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 processor begins to spin on an "operation completed" flag. 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 remote invocation mechanism is *optimistic*, in that it minimizes latency in the absence of contention and admits starvation in the presence of contention. Its average latency, excluding parameter copying and operation costs, is 56 μs. An earlier, non-optimistic, implementation relied on microcoded atomic queues, but these required approximately 60 μs for the enqueue and dequeue operations alone.

3.2. Explicit Synchronization

Psyche uses spin locks to synchronize access to kernel data structures. In order 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. Opening (mapping) a realm, for example, can require up to nine lock acquisitions. Creating a realm can require 38 lock acquisitions. A cheap implementation of locks is critical.

We use a test-and-test&set lock [14] 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.

A lock can be acquired and released manually, by calling inline 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 end of 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 [10] might reduce the need for fine-grain locks, but would probably not be faster, given the cost of atomic operations.

3.3. 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. The last three lines give times for high-latency operations: creating a realm, opening (mapping) a realm, and adding a new activation to a protection domain. All times are accurate to about ±3 in the third significant digit. Times for the low-latency operations are averaged over 10,000 trials. They are stable in any particular kernel load image, but fluctuate with changes in instruction alignment. They are also sensitive to the context in which they appear, due to variations in the success of compiler optimizations. We have read through the assembly language output of the compiler for our timing tests, to make sure the optimizer isn't removing anything important. Times for the high-latency operations are averaged over 1 to 10 trials. They are limited by the resolution of the 62.5 μs clock.

Times in columns 1 and 2 are with all data on the local node. Times in columns 3 through 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 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 always executed via remote invocation, so that the lack of context-switching would obviate the need for locks. Column 5 indicates the cost of performing operations via remote invocation in a hybrid kernel that continues to rely on locks.

In actuality, of course, the use of remote invocation for all remote operations eliminates the need for true mutual exclusion locks, but retains the problem of synchronization between normal activity and the remote invocation interrupt handler. The times in columns 2 and 6 may therefore underestimate real costs. (The times in column 2 do apply, as shown, to subsumption in larger operations with coarser locking.) A more realistic implementation of remote invocation without explicit locking would employ a bit indicating whether normal execution was currently in the kernel. If the kernel were already active, the remote invocation handler would queue its request

Operation		Local Access locking		Remote Access locking		Remote Inv. locking	
		on	off	on	off	on	off
enqueue+dequeue	(μs)	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	(μs)	40.6	30.5	211	169	125	105
create realm	(ms)	6.20	5.69	14.8	13.1	6.87	6.37
open realm	(ms)	0.96	0.86	3.05	2.62	1.15	1.09
create activation	(ms)	1.43	1.35	3.30	3.04	1.53	1.43

Table 1: Overhead of Kernel Operations

for execution immediately prior to the next return to user space. If the kernel were not active, the interrupt handler could execute its operation immediately, at interrupt level, or it could use a mechanism such as the VAX AST to force a context switch out of user space and into the kernel upon return from the interrupt handler. Execution directly from the interrupt handler is clearly faster, but may or may not be appropriate for high-latency operations. We have used it in all our tests, and our figures indicate the performance that results when the kernel is not otherwise active. With the exception of diagnostic serial lines, devices in the Butterfly are attached to a single "king" node; processors other than the king are in no danger of losing device interrupts due to extended computation at high priority.

Explicit Synchronization

As seen in Table 1, the cost of synchronization dominates in simple operations on queues, introducing in some cases nearly 100% overhead for local operations and 60% overhead for remote operations. Though less overwhelming, synchronization impacts more complex operations as well, due to the use of fine-grain locks. Realm creation requires acquiring and releasing approximately 38 constructor-based locks, adding over 500 μs, or 9%, to the cost in the local case and 1.7 ms, or 13%, 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, this overhead is imposed on local access to data structures in order to *permit* remote access to those structures. We could reduce the cost of synchronization by locking data structures at a coarser grain. This change would reduce the number of locks required by a typical operation, but would simultaneously reduce the potential level of concurrency.

Remote References

We can assess the impact of remote memory references by comparing the cost of local and remote operations in Table 1. Without locking, the marginal cost of remote references accounts for 86% of the cost of 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 10th element in a list, remote references exclusive of synchronization account for 2/3 of the cost of the operation. Even for complex operations such as realm creation, which performs much of its work out of the stack, remote references account for 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 remote invocation exclusively we can rely on implicit synchronization (non-preemption 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 comparable to the times in column 1; the ability to avoid lock acquisition and release hides the cost of remote invocation and parameter passing. The enqueue/dequeue operation and the search in a list of 5 both take less time via remote invocation, without synchronization, than they take via remote memory access with synchronization. If we could avoid the need for synchronization, however, (e.g. by coarse-grain locking), remote access would be cheaper. Since a remote memory access costs more than 6 μs more than a local access, and a remote lock/unlock pair costs about 40 μs, the 60 μs overhead of a remote invocation with a single parameter can be justified to avoid four remote references and a lock/unlock pair. If synchronization were free, remote invocation could still be justified to avoid eleven remote references.

4. 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 atomic operations used for synchronization, and the ratio of remote-to-local memory access time. Since the overhead associated with remote access scales with the operation, while the overhead associated with remote invocation is fixed, we would expect remote access to outperform remote invocation only on relatively simple operations. The operating system designer must determine exactly which operations, if any, would benefit from the use of remote access, and whether the impact on the overall design of the operating system would be justified.

On the Butterfly Plus, remote invocation is relatively fast, explicit synchronization is costly, and remote references significantly more expensive than local references. As a result, few operations can be executed more efficiently with remote access than with remote invocation. In fact, remote invocation dominates even if the kernel exhibits node locality. Although originally introduced to minimize remote references, node locality can also ensure that only one remote invocation is required per kernel operation. If no attempt had been made to maximize node locality, no complex operation could have been performed with one remote invocation; many invocations would be needed just to collect the data necessary to perform an operation. Under those circumstances remote access would be competitive; however, the resulting organization would not be a reasonable one for the Butterfly architecture.

Given the obvious advantages of remote invocation on the Butterfly Plus, why did we consider using shared memory in the original Psyche kernel? First, we were attracted to the shared-memory kernel model on aesthetic grounds and believed that remote accesses could be minimized and overall performance made acceptable with an appropriate degree of node locality. We did not realize the extent to which the cost of remote access would be dominated by synchronization, nor did we explicitly recognize that node locality would also improve the performance of remote invocation. We expected that the remote accesses required by a typical operation, even if few in number, would often be separated by significant amounts of local computation, and would therefore require several separate remote invocations. Second, our original experience with remote invocation suggested that it took over 150 μs to perform a remote operation, which increased the appeal of remote access. Unfortunately, this experience was based on an implementation that used Butterfly atomic operations extensively. Our current implementation does not use those operations at all. Third, our estimates of the cost of synchronization were based heavily on the efficiency of the MC68020 test&set instruction, and did not sufficiently consider such additional factors as the need to differentiate between local and remote locks, the overhead of constructors and destructors, and the frequency of synchronization.

Despite our conclusions regarding the advantages of remote invocation, remote access can play an important role in the kernel. For example, the PLATINUM kernel on the Butterfly [6] uses remote access to manage page tables and can free a page in 10 μs. Efficiency in managing page information is particularly important in PLATINUM because the operating system replicates and migrates pages frequently to create the illusion of uniform access memory. Since most operations on page tables and memory management data structures are simple operations, they are particularly well-suited to remote access. Thus, there clearly is a role for remote access in the kernel. The significance of that role will vary from machine to machine, depending on architectural parameters. On the Butterfly, remote invocation should be the dominant mechanism, reserving remote access for frequently executed, specialized operations.

Acknowledgments

Our thanks to Rob Fowler for his helpful comments on this paper, and to Tim Becker for his invaluable assistance with experiments.

References

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proc. of the Summer 1986 USENIX Technical Conference and Exhibition*, Pittsburgh, PA, June 1986.
2. BBN Advanced Computers Inc., Inside the Butterfly Plus, Oct 1987.
3. M. J. Bach and S. J. Buroff, "Multiprocessor Unix Systems," *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct 1984), pp. 1733-1750.
4. F. Baskett, J. H. Howard and J. T. Montague, "Task Communication in Demos," *Proc. 6th ACM Symp. on Operating System Principles*, West Lafayette, IN, Nov 1977, pp. 23-31.
5. D. Clark, "The Structuring of Systems Using Upcalls," *Proc. 10th ACM Symp. on Operating System Principles*, Orcas Island, WA, Dec 1985, pp. 171-180.
6. A. L. Cox and R. J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proc. 12th ACM Symp. on Operating System Principles*, Litchfield, AZ, Dec 1989, pp. 32-44.
7. A. L. Cox, R. J. Fowler and J. E. Veenstra, "Interprocessor Invocation on a NUMA Multiprocessor," TR 356, Department of Computer Science, University of Rochester, Oct 1990.
8. R. A. Finkel, M. L. Scott, Y. Artsy and H. Chang, "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," *IEEE Transactions on Software Engineering 15*, 6 (June 1989), pp. 676-685.
9. A. Garcia, D. Foster and R. Freitas, "The Advanced Computing Environment Multiprocessor Workstation," IBM Research Report RC-14419, IBM T.J. Watson Research Center, Mar 1989.
10. M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Structures," *Proc. of the Second PPOPP*, Seattle, WA, Mar 1990, pp. 197-206.
11. H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *Operating Systems Review 13*, 2 (Apr 1979), pp. 3-19.
12. T. J. LeBlanc, J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl and P. C. Dibble, "The Elmwood Multiprocessor Operating System," *Software—Practice & Experience 19*, 11 (Nov 1989), pp. 1029-1056.
13. J. M. Mellor-Crummey, "Concurrent Queues: Practical Fetch-and-Phi Algorithms," TR 229, Department of Computer Science, University of Rochester, Nov 1987.
14. J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems*, to appear. Earlier version published as TR 342, Department of Computer Science, University of Rochester, April 1990, and COMP TR90-114, Center for Research on Parallel Computation, Rice University, May 1990.

15. 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," *Proc. 1985 International Conference on Parallel Processing*, St. Charles, IL, Aug 1985, pp. 764-771.

16. 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," *Computing Systems 3*, 1 (Winter 1990), pp. 101-137 .

17. A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.