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
{prakash,leblanc,marsh,scott}@cs.rochester.edu
COS99286@UFRJ.BITNET (Chaves)

April 1991

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 and conceptual tradeoffs may not be apparent in advance.

In this paper we compare the use of remote memory 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 Plus designed to empirically evaluate the tradeoffs between remote memory access and remote invocation. 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 memory 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 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 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, at least on UMA 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 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 in large part 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 [11], and IBM RP3 [18] 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 and conceptual 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.

remote invocation

The processor at node *i* sends a message to a processor at node *j*, asking it to perform the

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.

operation on its behalf. The operation may be executed directly by the message interrupt handler, or indirectly via wakeup 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.

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 mechanisms are 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 inter-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. In studies of VMS on the VAX-11/780, for example, Clark and Emer [6] found that for a workload in which user-level code generated over three quarters of all memory references, kernel-level code generated over two thirds of all TLB misses. 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. Finally, kernels spend little of their time in nested loops.

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. The PLATINUM kernel [8] in particular would benefit from address locality, since large parts of PLATINUM 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.

Given our apprehension regarding bulk data transfer as a basic inter-kernel communication mechanism, we are primarily interested in 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 tradeoffs 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 labor among processes in the kernel. Ultimately, we argue in favor of a mixture of all three mechanisms (remote memory access and the two forms of remote invocation), since no one mechanism will be the best choice for all operations.

3. Remote Invocation Details

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 parlance [15], this simple mechanism 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 kernel process context on the target processor. Doing so requires a transfer into the *top half* of the kernel. A bottom-half remote invocation (BHRI) may be very fast, but can only be used for operations that can be executed safely in an interrupt handler. A top-half remote invocation (THRI) is slower, but of more general utility.

3.1. Bottom-Half Remote Invocation

The bottom half of a uniprocessor kernel contains only those routines that service asynchronous devices. All other routines reside in the top half of the kernel. In essence, bottom-half routines execute at interrupt level in response to asynchronous interrupts; top-half routines execute in a normal process context in response to system calls. Since interrupts occur at unpredictable times, and interrupt handlers cannot block, mutual exclusion for data structures shared between the top and bottom halves of the kernel must be achieved by masking interrupts. On a uniprocessor these data structures consist primarily of I/O buffers. On a multiprocessor using BHRIs, the number and variety of data structures shared with the bottom half may increase, and it may be necessary to lock more than one of them at a time. Critical sections may nest or interleave. Simply turning interrupts on and off may not suffice for lock acquisition and release.

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

BHRI has several limitations. Because its handlers lack a process context, it cannot be used for operations that may block. If used extensively it may tend to push more data structures into the bottom half of the kernel than would otherwise have been there. The only form of synchronization available for these data structures is short-term mutual exclusion, and that occurs at a very coarse grain. Even BHRIs that do not touch data of any interest to current top-half activity are likely to be disabled much of the time. To prevent deadlock, we must prohibit outgoing BHRIs when incoming BHRIs are disabled. This rule severely limits the circumstances under which 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.

BHRI is permitted. In particular, it precludes the use of BHRI for operations that must access data on two different processors atomically, unless the programmer is willing to detect and recover from deadlock.

It may be possible on some machines to deliver interprocessor interrupts 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 BHRI for lengthy operations. Several other factors, however, suggest that we use it only for short operations. Longer operations are likely to need condition synchronization, or to require locks on more than one processor, or to force large amounts of data into the bottom half. Longer operations are also more able to tolerate the latency of a general-purpose, top-half 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 a BHRI from the handler for another. Re-scheduling is likely to be slower than busy-waiting when requesting a BHRI from a top-half routine. Re-scheduling is not possible in a bottom-half routine, but busy-waiting is, provided that we restore data structures to a consistent state and re-enable BHRIs before making a nested invocation.⁴ We assume in the rest of this paper that BHRI is used for short operations only, and that the requesting processor spins.

3.2. Top-Half Remote Invocation

To execute a remote invocation in the top half 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 execution is interrupted in user space, the handler schedules an asynchronous trap immediately. If execution is interrupted in the kernel, the handler queues the request for examination the next time execution returns to user space or enters the idle loop in the kernel. Deadlock prevention requires that a process block when making a THRI, rather than busy-wait, because busy-waiting would lock out incoming THRI. 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 THRI will be used for longer operations anyway.

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

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

We can also use remote access and BHRI on the same data structure, but only with a hybrid lock that uses both interrupt masking *and* spinning, with a queuing discipline that guarantees

³ Note that *high* priority interprocessor interrupts are required for operations such as TLB shutdown [5, 8, 19].

⁴ Nested BHRIs must be performed with care. They suggest the need for very 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 a BHRI services unrelated interrupts.

service to bottom-half routines. The critical point is to ensure that a bottom-half routine that attempts to acquire a lock is guaranteed to succeed after a bounded amount of spinning. We must therefore lock out interrupts while holding a hybrid lock, and consequently cannot perform a BHRI while holding such a lock. Finally, we can use both THRI and BHRI 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 THRI while it has locked out BHRIs in order to touch a data structure that is shared between the kernel's top and bottom halves.

4. Remote Access/Invocation Tradeoffs

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 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 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).⁵ 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.

4.2. Indirect Costs for Local Operations

An important factor that is ignored in comparisons 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 THRI for *all* remote 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

⁵ 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.

context-switching as in a uniprocessor kernel. Explicit synchronization is still required, of course, for any data structures that a process needs to keep locked on the local node while making a THRI. Because BHRI handlers can always execute unless explicitly locked out (even if they interrupt top-half kernel activity), explicit synchronization is still required for data structures accessible to bottom-half routines. Depending on architectural parameters, BHRI-inhibiting locks may be faster than semaphores or top-half 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. 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 [12, 16] may allow explicit synchronization to be omitted even for data structures whose operations are implemented via remote memory access.

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 remote kernel data into a separate kernel-kernel address space [20] 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 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.

Given that the requestor of a BHRI 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. A BHRI 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, 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 [13]. 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 [7]. The horizontal organization may be easier to debug [10]. Most Unix kernels are vertical. Demos [4] and Minix [21] 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 [14]. 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.

5. Case Study: Psyche on the BBN Butterfly

Our experimentation with alternative communication mechanisms took place in the kernel of the Psyche operating system [20] 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 *protec-*

tion 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.

5.1. 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 [9] of an instruction to read a 32-bit remote memory location using register indirect addressing is 6.88 μ s; the corresponding instruction to read local memory takes 0.518 μ s. The time to write memory is slightly lower: 4.27 μ s and 0.398 μ s for remote and local memory, respectively.⁶ 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 μ s instead of 1.76 ms for 1K bytes). 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 THRI), or begins to spin on an "operation completed" flag (in the case of a BHRI). 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 a BHRI, excluding parameter copying and operation costs, is 56 μ s. An earlier, non-optimistic, implementation relied on microcoded atomic queues, but these required approximately 60 μ s for the enqueue and dequeue operations alone. The average latency of a THRI, again excluding parameter copying and operation costs, is about 421 μ s. THRI could be made faster with some more hand optimization, but it is unlikely that we could get it under 300 μ s. BHRI is highly optimized; we see no way to make it significantly faster.

⁶ 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.

5.2. 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 [17] 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, in order to determine the node to interrupt.

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 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 μ s, and may require as much as 10 μ s, 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 μ s. The additional time required to acquire and release a lock through constructors is about 1 to 3 μ s. 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 [12] 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 activations are waiting. Locks to disable BHRIs are slightly cheaper, averaging just over 5 μ s.

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 the kernel's bottom half. 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. Remote invocations for these are implemented in the kernel's top half. 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

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

Table 1: Latency of Kernel Operations

appear, due to variations in the success of compiler optimizations.⁷ Times for the high-latency operations are averaged over 1 to 10 trials. They are limited by the resolution of the 62.5 μ s 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 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.

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 Table 2. Synchronization clearly dominates the cost of simple operations on queues, contributing in some cases nearly 50% for local operations and 40% 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, contributing over 500 μ s, 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 top-half 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 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. Table 3 indicates the marginal cost of remote references for each of our kernel operations. Without locking, this marginal cost is 86% for a remote

⁷ 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.

Operation	Local Access	Remote Access	Remote Invocation
enqueue+dequeue	49%	38%	12%
find last of 5	36%	33%	16%
find last of 10	25%	20%	16%
create realm	8%	11%	7%
open realm	10%	14%	9%
create activation	6%	8%	7%

Table 2: Percentage of Latency Due to Explicit Locking

enqueue/dequeue operation pair; remote references exclusive of synchronization account for 54% of the cost even when locking is used (154 μ s to perform the operation remotely excluding synchronization costs minus 21.6 μ s to perform the operation locally, over 247 μ s 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 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 THRI 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 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 μ s more than a local access, the 60 μ s overhead of a BHRI with a single parameter can be justified on the Butterfly Plus in order to avoid eleven remote references. The 421 μ s overhead of a THRI can be justified to avoid k lock acquisitions and 71–7 k remote references. If hand optimization were to yield an implementation of THRI that cost 300 μ s, it could be justified to avoid only j lock acquisitions and 50–7 j remote references.

Comparative Latency of Remote Access and Remote Invocation

Table 4 presents three different comparisons of the cost of remote invocation and the cost of remote memory access. The first column is the ratio of columns 6 and 3 in Table 1; it compares remote memory access with explicit locking to remote invocation without explicit locking. The second column is the ratio of columns 5 and 3 in Table 1; it includes locking overhead for both

Operation	With Locking	Without Locking
enqueue+dequeue	54%	86%
find last of 5	55%	82%
find last of 10	66%	82%
create realm	50%	57%
open realm	58%	67%
create activation	51%	56%

Table 3: Remote Access Penalty for Kernel Operations

remote memory access and remote invocation, as would exist in a hybrid kernel. The third column is the ratio of columns 6 and 4 in Table 1; it considers the case in which the desired operation is subsumed in some other operation with coarse-grain locking.

Figures greater than one 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 BHRI, or complex enough to absorb the overhead of a THRI. For medium-size operations we may be unwilling to accept either the limitations of BHRI or the overhead of THRI. Moreover even for tiny operations the *throughput* of BHRI may be unacceptably low, as discussed in the following section.

Throughput for Bottom-Half 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 invocations, however, steal the entire processor for the duration of the operation. In the case of THRI, the requesting processor has the opportunity to do something else while the invocation is in progress; in the case of BHRI it does not. These observations suggest that latency provides a reasonable measure of the performance of THRI, but that throughput must also be considered for BHRI.

We conducted a simple experiment to compare the impact of remote memory accesses and BHRI 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 BHRI. For a set of low-latency operations whose aggregate latency was 19 ms when using remote memory access and 22 ms when using BHRI, we measured application slowdowns of 1 ms and 17 ms, respectively. From these figures we can see that BHRI 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. Since the overhead associated with remote access scales with the operation, while the overhead associated with remote invocation is fixed, there will for any

Operation	Locking		
	Remote Access Only	Both	Neither
enqueue+dequeue	0.70	0.80	1.13
find last of 5	0.74	0.88	1.10
find last of 10	0.50	0.59	0.62
create realm	0.46	0.50	0.53
open realm	0.58	0.64	0.68
create activation	0.53	0.57	0.58

Table 4: Ratio of Latencies for Remote Invocation and Remote Memory Access

machine be a break-even point above which THRI 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, THRI, and BHRI.

Our experience with Psyche suggests 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 THRI, 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 vertical organization across the entire machine. Fortunately remote access appears to be the most efficient mechanism for many operations. Nonetheless, we definitely underestimated the impact that locking would have on the cost of typical operations. If we were to re-build the kernel at this point, we would make more extensive use of THRI for lengthy operations. We would also attempt to identify data structures for which THRI 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 BHRI is the mechanism of choice. The most plausible scenario arises with low-latency operations that cannot be performed via remote access. TLB shutdown is such an operation on most machines; instructions that manipulate the TLB cannot be invoked remotely. We also use BHRI for console I/O, and to implement our remote debugging facility [20]. BHRI 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 BHRI (e.g. for a memory-intensive operation on a machine in which remote memory is exceptionally slow). In both these latter scenarios BHRI 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. As a result, most operations display lower latency with remote invocation than they do with remote memory access. For small operations, however, BHRI displays lower throughput than remote access, 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 BHRI. Moreover, we strongly suspect that the desire to keep data structures out of the bottom half of the kernel will mean that many operations become too big to use BHRI before they become big enough to absorb the overhead of THRI. These observations suggest that performance will be maximized if remote access is used for operations comprising up to a few dozen memory references and THRI is used above this limit. BHRI should be reserved for special cases.

Acknowledgments

Our thanks to Rob Fowler for his 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.

References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevarian and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986, pp. 93-112.
- [2] BBN Advanced Computers Incorporated, "Inside the Butterfly Plus," Cambridge, MA, 16 October 1987.
- [3] M. J. Bach and S. J. Buroff, "Multiprocessor Unix Systems," *AT&T Bell Laboratories Technical Journal* 63:8 (October 1984), pp. 1733-1750.
- [4] 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.
- [5] 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.
- [6] D. W. Clark and J. S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Transactions on Computer Systems* 3:1 (February 1985), pp. 31-62.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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. Extended abstract presented at the *IEEE Workshop on Design Principles for Experimental Distributed Systems*, Purdue University, 15-17 October 1986.
- [11] A. Garcia, D. Foster and R. Freitas, "The Advanced Computing Environment Multiprocessor Workstation," RC-I4419, IBM T. J. Watson Research Center, March 1989.
- [12] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Structures," *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, 14-16 March 1990, pp. 197-206.
- [13] H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *ACM SIGOPS Operating Systems Review* 13:2 (April 1979), pp. 3-19. Originally presented at the *Second International Symposium on Operating Systems*, October 1978.
- [14] 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 (November 1989), pp. 1029-1056.

- [15] 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.
- [16] J. M. Mellor-Crummey, "Concurrent Queues: Practical Fetch-and- Φ Algorithms," TR 229, Computer Science Department, University of Rochester, Nov 1987.
- [17] 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, Computer Science Department, University of Rochester, April 1990, and COMP TR90-114, Center for Research on Parallel Computation, Rice University, May 1990.
- [18] 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.
- [19] B. Rosenberg, "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. 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. Earlier version presented at the *First Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL, 5-6 October, 1989, 227-236.
- [21] A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.