4

Interprocessor Invocation on a NUMA Multiprocessor

Alan L. Cox, Robert J. Fowler, and Jack E. Veenstra

Technical Report 356
October 1990

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Interprocessor Invocation on a NUMA Multiprocessor

Alan L. Cox
Robert J. Fowler
Jack E. Veenstra

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 356

October 1990

Abstract

On a distributed shared memory machine. the problem of minimizing accesses to remote memory modules is crucial for obtaining high performance. We describe an object-based, parallel programming system called OSMIUM to support experiments with mechanisms for performing invocations on remote objects. The mechanisms we have studied include: non-cached access to remote memory, data migration, and function-shipping using an interprocessor invocation protocol (IIP). Our analyses and experiments indicate that IIP competes well with the alternatives, especially when the structure of user programs requires synchronized access to data structures. While these results are obtained on a NUMA multiprocessor, they are also applicable to systems that use hardware cache coherency techniques.

This work is supported in part by ONR/DARPA research contract no. N00014-82-K-0193 and in part by NSF research grant no. CDA-8822724.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 356	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Interprocessor Invocation on a NUMA Multiprocessor		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alan L. Cox, Robert J. Fowler, & Jack E. Veenstra		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department 734 Computer Studies Bldg. University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE October 1990
		13. NUMBER OF PAGES 12
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. *(of this report)* unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* Distribution of this document is unlimited.		
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*		
18. SUPPLEMENTARY NOTES none		
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)* RPC, remote invocation, NUMA memory, object-oriented, caching, scheduling, migration		
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)* (see next page)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

On a distributed shared memory machine, the problem of minimizing accesses to remote memory modules is crucial for obtaining high performance. We describe an object-based, parallel programming system called OSMIUM to support experiments with mechanisms for performing invocations on remote objects. The mechanisms we have studied include: non-cached access to remote memory, data migration, and function-shipping using an interprocessor invocation protocol (IIP). Our analyses and experiments indicate that IIP competes well with the alternatives, especially when the structure of user programs requires synchronized access to ıta structures. While these results are obtained on a NUMA multiprocessor, they are also applicable to systems that use hardware cache coherency techniques.

1 Alternative Mechanisms for Remote Object Invocation.

While the physical distribution of active and passive entities in large parallel computer systems adversely affects the latency of individual operations, the alternative of co-locating all parts of a program at a single processing element is unsatisfactory. Reducing the execution time of a computation by exploiting parallelism requires partitioning and distributing (mapping) sub-computations and data among the processors and memories of the system [7, 22]. OSMIUM (Object System Mechanisms Implementing Universal Mobility) is a parallel, object-based programming system designed to support experiments with dynamic mapping on large shared-memory computers. OSMIUM provides a general set of alternative implementations for invoking objects that are not co-located with the invoking thread. These include: non-cached fine-grain access to the remote memory, implicit object migration by the kernel's software caching system, explicit migration assisted by the OSMIUM run-time library, and function-shipping using an interprocessor invocation protocol (IIP). In most cases the choice of which of the alternative mechanisms to use is strictly a performance issue; these decisions can be made transparently by the kernel, the compiler, and the language run-time library, or they can be made explicitly by the programmer late in the program development process. While it was implemented specifically to run on a BBN Butterfly GP1000 [2], a NUMA multiprocessor, the ideas incorporated in OSMIUM are also applicable to any multiprocessor architecture in which the physical distribution of the memory hierarchy interacts with user programs to affect performance. The techniques discussed in this paper will become more important as processor speed increases relative to the latency of a remote memory access such as a cache miss.

OSMIUM runs on the PLATINUM operating system kernel [10]. The memory model implemented by the kernel resembles a simplified Mach [21], from which it is derived. Each virtual address space can be shared by kernel-scheduled *threads* on multiple processors, where, unlike Mach, a thread is bound to a single processor for its lifetime. To decrease the average latency of memory access and to reduce contention, the kernel uses a software implementation of a directory-based cache coherency protocol to control the migration and replication of data on a per-page basis. Because of the large block size and the software implementation, the overhead of caching could be very large for data that is write-shared at a fine grain. When the kernel detects thrashing it temporarily disables caching on a per-page basis. User code may explicitly specify the placement of data on a per-page basis. It may also specify that a write-update protocol be used to maintain coherency for a page instead of the default write-invalidate protocol.

OSMIUM is a parallel extension of C++ [25] in which a parallel application program occupies a single PLATINUM address space. Its run-time library provides a *lightweight task* package that uses a single kernel thread on each of the available processors for the execution of the user program. Tasks can be created with an explicit operation or implicitly as part of the IIP. Although the entire virtual address space is accessible from any processor, the memory allocator maintains a set of zones so that data with different access patterns can be allocated to distinct virtual address ranges. Thus, if a particular range is occupied only by data accessed by a single processor, the kernel will keep that range in memory local to that processor. Similarly, large migratory objects are allocated in zones distinct from those occupied by fine-grain write-shared objects such as locks.

Our research using OSMIUM is designed to answer the following questions:

1. What are the costs and benefits of the different forms of mobility associated with each of the mechanisms for invoking operations on remote data?

2. Can we justify the inclusion of each alternative on performance grounds? To be worth including in a system there must be a case in which each mechanism outperforms all of the others. Furthermore, the cost of including the mechanism should be balanced against the total benefit of using it.

3. Since making explicit placement decisions is a burden on users, it is desirable to make mapping decisions automatic and transparent. Are there effective automatic policies for making placement decisions and choosing among invocation mechanisms? At what level of the system can and should these choices be made? Operations implemented in hardware or in the kernel can be very efficient, and decisions made at these levels can be sensitive to rapidly changing performance feedback while being functionally transparent to the user. A limitation of these on-line policies implemented at the lower levels of the system is that they neither have the benefit of the kind of static analyses implementable in compilers nor do they know the programmer's intentions.

While ultimately we are interested in all of these issues, they must be answered in order. In this paper we present results addressing the first two issues.

1.1 Summary of Results.

This paper concentrates on the quantitative comparison of the use of the IIP to object migration or non-cached remote memory reference.[1] The results comparing the relative merits of remote memory access to object migration are specific to our prototype implementation of OSMIUM; the lessons are most applicable to other systems that are based on an object-migration mechanism or that have large cache block sizes. On the other hand, our comparison of the IIP to these mechanisms has broader consequences for appropriate programming styles on any system in which there are high relative costs due to latency and contention for memory operations that miss the local parts of the memory hierarchy. In systems in which processors become faster with respect to the interconnect latency [9, 12, 11, 17, 4], the use of interprocessor invocation will become more attractive compared to naive shared-memory implementations of the same operations.

2 Rationale and Related Work.

In contrast with theoretical models of parallel computation such as the PRAM, in real parallel systems data are partitioned among physically distributed memories and computational operations are similarly assigned to physically distinct processors. To get good performance in a parallel program, most operands should be in memories close to the processors that execute the operations on them. Unless the processors do not share or communicate data among themselves, there will be circumstances in which a program will specify that a thread of control on one processor should perform an operation on data in a remote memory, usually associated with another processor. Table 1 presents the set of alternative strategies for executing such remote operations. The simplest strategy is to move neither the object nor the thread of control and to execute the operation remotely. Remote execution in OSMIUM is done using remote memory references, but it may also be possible to recursively partition the operation and to choose among the mechanisms for each component.

Another possible implementation of the remote operation is to migrate the data to the site of the thread. This can be done at many levels including hardware caches, software caching schemes such as shared virtual memory [18], software-controlled distributed object stores [20, 14], or transfer of the data explicitly using message-passing. In some cases it may be more economical to move the thread of control so that the operation is executed at the location of the data. The migration of the thread can be either permanent or temporary, as in the execution of an operation in a combining switch [13], remote procedure call, or location-independent invocation [5]. Finally, the thread, the

[1] On a *multi*, a hardware cache-coherent multiprocessor, migration is performed through a sequence of cache misses and the latter option is not available.

		OBJECT		
		Does not move	Migrate/ Replicate to Thread node	Migrate/ Replicate to another node
THREAD	Does not move	R	L (M)	R (M)
	Moves to Object node	L (I)	R (IM)	R (IM)
	Moves to another node	R (I)	R (IM)	L (IM)

Table 1: Alternatives for performing an operation on a remote object. Each uses some combination of (R)emote memory references, (L)ocal memory references, remote (I)nvocation, or object (M)igration.

data, or both can be moved to a third location. These alternatives might be attractive to control contention for processor and memory bandwidth.

While it provides mechanisms for all of these alternatives, the first use of OSMIUM is to test the viability of interprocessor invocation on a shared-memory multiprocessor. The choice between the IIP and data migration or remote memory access is not determined automatically but by the programmer. A policy in the software caching system automatically chooses between data migration and remote memory access.

The development of OSMIUM is also motivated by our experiences writing and porting applications to PLATINUM. For the best performance, data with different access and sharing patterns must be separated into different pages. This requires that users understand a program in some detail and write appropriate code for memory allocation. Our experience has also been that a lot of problems arise due to compilers and run-time libraries that are inappropriate for multiprocessor, especially NUMA, use. For example, a naive placement of data by the linker can be a significant source of false sharing. Co-location on a cache block of data such as a lock variable and a message buffer with very different patterns of true sharing results in similar problems. OSMIUM addresses some of these issues by providing a run-time library that insures the separation of its own data from user data and includes routines to simplify user memory allocation.

2.1 Related Work.

Our work on OSMIUM touches on several areas of current research interest and provides a unique synthesis of ideas and techniques.

There have been many distributed object-based programming systems, including Eden [6], Emerald [15], Amber [8], and Clouds [20]. While they all run on multiple nodes on a local area network, Amber is the only one of the group designed to run on multiprocessor nodes. Unlike OSMIUM interprocessor invocation, remote invocation on Amber is used solely to communicate between threads on different multiprocessors, while ordinary procedure call and shared memory are used between threads on the same multiprocessor. OSMIUM uses the object model to help provide a modular structure to application programs compatible with the need to avoid very fine-grain sharing in large systems. The spatial modularity of object data structures and the temporal modularity of the operations that manipulate them are visible both to the compiler and to the run-time library. Because method entry and exit mark changes in locality of reference, object-based programming provides convenient demarcation points for the evaluation of (re)location decisions.

On a multi, a thread executing on a particular processor will load its *footprint* [26] into the local cache. Rescheduling the thread on another processor separates it from its footprint, which will then be migrated to the new location by a burst of cache misses. Scheduling methods that take processor-cache affinity into account are intended to reduce the number and magnitude of such transients [24]. A dual phenomenon occurs when a thread invokes an operation on often-used shared data that is in another processor's cache; the new footprint will be remote from the invoking thread. As an alternative to migrating the data with a burst of cache misses, IIP is a mechanism to move the operation to the processor that owns the cache.

Bershad has described a fast, user-level RPC (URPC) mechanism for shared-memory multiprocessors [3]. The implementation techniques for the IIP and URPC are similar, but the goals are very different. The major goal of URPC is to provide fast cross-address space communication, whereas the IIP is used as a mechanism for improving processor-data locality for object invocations. We call our protocol IIP rather than RPC because an interprocessor invocation in OSMIUM does not cross address space boundaries.

One approach is the construction of high-performance mechanisms for blocking synchronization for avoiding the loss of resources associated with mechanisms that stall the hardware or use spin locks in software. At one extreme are the so-called "latency tolerant" processors such as HEP [23] or Tera [16] that switch among multiple hardware contexts on memory references. At a coarser grain, recent work on the Alewife multiprocessor is examining the use of lightweight software context switches to hide the latency of cache misses in a scalable shared memory system [1]. Multithreaded architectures such as HEP and Alewife can hide the latency of data retrieval, but do not reduce the original amount of traffic to the remote memories. The IIP blocks at the object invocation granularity, but this is part of a mechanism to reduce the traffic rather than to hide the latency. Blocking at any of these granularities affects the programming style. To maintain throughput each processor must be given enough tasks to insure that it never idles.

3 The OSMIUM Prototype.

OSMIUM is an augmented version of C++. The main visible additions to the standard C++ runtime environment are a lightweight task package and a memory allocation package that partitions the virtual address space into a set of heaps called *zones*. An OSMIUM program is contained within a single PLATINUM address space that has one kernel-scheduled thread per processor to run user code. While the entire address space is visible to any of the threads, the use of zones allows a virtual address range to be associated with the memory (or cache) of a particular processor. This association can be a hard binding enforced by directives to the kernel or it can be a *de facto* relationship for an address range accessed by only that processor. The zones are also used to partition data according to sharing pattern.

When complete, OSMIUM will allow dynamic flexibility in the choice of mechanism for each invocation, but in the prototype that we built to perform the experiments reported here, some of the choices are constrained statically at compile-time. We modified the AT&T C++ translator to generate inline stubs that use the IIP for all calls to the methods of any class derived from the distinguished base class `uses_IIP`. Public data is still directly accessible by remote memory access or migration and inline methods provide direct access to both the private and public data. Methods of classes not derived from `uses_IIP` are always invoked using ordinary local procedure call regardless of location. To summarize, if an operation is a call to a method of a class derived from `uses_IIP` that is not declared inline, that operation will use IIP, even if the object is local. Otherwise, the operation is executed locally, accessing the object using remote memory access, data page migration, or write-update to replicated data. The choice of mechanism is made by PLATINUM using its own dynamic policy [10] as well as directives generated implicitly by the run-time library or explicitly by user code.

On each node the lightweight task package maintains a single task queue. Entries in this queue are either requests to create tasks or replies to existing tasks. A task can be created either explicitly by user code or implicitly by the IIP. We make the fundamental assumptions that the tasks within an OSMIUM program are all cooperating towards a common goal, that they are doing necessary work, and that it is more important to maximize throughput than to minimize the response time of individual operations. Therefore, rather than use pre-emptive scheduling to ensure fairness, we use run-until-block scheduling to reduce overhead.

The execution of the IIP begins with the inline stub generated by the translator. The stub determines the location of the object from the *object dictionary*. A call is performed to the *IIP_call* procedure which marshalls the parameters into a request message allocated on the stack and enqueues the message on the receiving node. The caller then prepares to service an incoming request by switching to a new task from a cache of free tasks. If the next queue entry is a request, it is executed by the new task. If the entry is a reply, the new task returns to the free cache and the task specified by the reply runs. This strategy reduces latency, overlapping task allocation by the caller with request processing by the receiver without reducing throughput.

The task created to service a request determines the number of arguments passed by the caller, copies the arguments to its stack, and performs a procedure call to the method. Upon return, the request message is recycled into a reply containing the return value, if any. The task enqueues the reply on the calling node and resumes service as above.

Because of the run-until-block scheduling, an object derived from **uses_IIP** can be viewed as a simple form of monitor in which the lock is released on nested calls to objects also derived from **uses_IIP**. Unsynchronized operations can be implemented using inline methods or public data. We have not yet implemented condition variables *per se*, but we do have blocking semaphores.

The object dictionary is implemented as a hash table. Since invocations are very frequent while object creation and deletion are comparatively rare, it is appropriate to maintain the dictionary in replicated form. Software cache coherency using a write-invalidate protocol works well for data that are migratory, broadcast, or otherwise write-shared at a coarse grain. However, the cost of invalidating a page is not just the cost of the invalidation operation. It includes the cost of replicating the modified page back to the processors that share it. If updates are relatively infrequent and involve a small part of the page, a write-update protocol works well. Dictionary data structures, for instance, the OSMIUM object directory, have this kind of access pattern. At initialization OSMIUM instructs the kernel to use its write-update protocol to maintain the coherency of the object dictionary.

Objects are assigned to nodes during the execution of the constructor for **uses_IIP**. The default placement policy is implemented as a C++ virtual function. The user can override the default policy for any given class by redefining this function.

The only feature distinguishing processors in an OSMIUM address space is that an address register is reserved using the GNU C compiler (as the back-end for the C++ translator) for use as a pointer to a *nodestructure* containing data private to the processor executing the current thread. Also using the GNU C compiler to generate Motorola 68020 code, Mellor-Crummey and LeBlanc [19] evaluated the cost of reserving a data register. A set of common Unix programs was slowed down by 1% in the worst case. Since this private data includes the queues and other data used by the task package on every invocation, the loss of the use of this register to the compiler was deemed a reasonable price to pay.

Category	Caller	Receiver
data marshalling	7	4
message enqueue and dequeue	11	10
synchronization	14	14
dispatch	3	4
thread switch	6	
procedure call linkage	5	11

Table 2: Instruction Breakdown.

3.1 IIP Latency and Throughput.

IIP requires a minimum of 46 instructions by the caller[2] and 43 instructions by the receiver. Table 2 shows the breakdown by category for the caller and the receiver. During a remote object invocation using th P the caller performs one remote read and four remote writes, whereas the receiver performs eight remote reads and ten remote writes. Each parameter requires one additional instruction and remote memory reference by the receiver. The average execution time for an instruction such as `movl a2@,d0` accessing remote memory is $6.88\mu s$ versus $0.518\mu s$ accessing local memory, whereas the average execution time for a write such as `movl d0,a2@` to remote memory is $4.27\mu s$ versus $0.398\mu s$ to local memory.[3]

We measured the minimum latency of the IIP by timing 10000 calls to an empty procedure that takes any number of parameters and does not return a value. The curve labelled "call latency" in Figure 1 shows the average measured execution time per invocation as the number of parameters is varied from 0 to 8. In this experiment, the calling processor busy-waits while the receiver executes its part of the protocol. Because some instructions in the caller and receiver parts of the protocol are executed concurrently, *this measurement of latency is an optimistic estimate of the overhead of* the IIP.

To obtain a more accurate estimate of overhead, hence throughput, we performed two related experiments. The execution of the IIP on a local object takes $90.9\mu s$ if there are no parameters. The single processor in this experiment executes both the caller and receiver parts of the protocol using only local memory references. Exactly the same path through the code is executed in each iteration of an experiment in which each of a pair of processors repeatedly invokes the empty procedure on the other. Since we use run-until-block scheduling, the two processors are guaranteed to run in lock-step. The average measured time per iteration is summarized by the curve labelled "call overhead" in Figure 1. If there are no parameters, this experiment takes $196\mu s$ per iteration. The difference compared with local execution of $105\mu s$ is consi tent with the 23 remote memory references used by the protocol.

To separate the costs of the caller and receiver parts of the protocol, we made another measurement of throughput by varying the number of processors that repeatedly invoke an empty method of an object on an otherwise idle processor. We measured the rate at which one of those processors is served. Increasing the number of processors from 1 to 2 only increases the average time per invocation by $55\mu s$ because of overlapping execution. From 2 to 15 processors, the average time per invocation increases linearly at $117\mu s$ per processor with a correlation coefficient $\rho = 0.99997$. This is the average time for the receiver to service and reply to a request. Since the average time to perform both the caller and receiver parts of the protocol together is $196\mu s$, the average time for the

[2]This excludes extra instructions executed due to collisions in the hash table and while busy-waiting if no task is ready to run.

[3]All timings were gathered on a 16-processor BBN Butterfly GP1000. A processing node on this machine consists of a 16.67MHz MC68020 with a MC68851 MMU and 4 MBytes of physical memory.

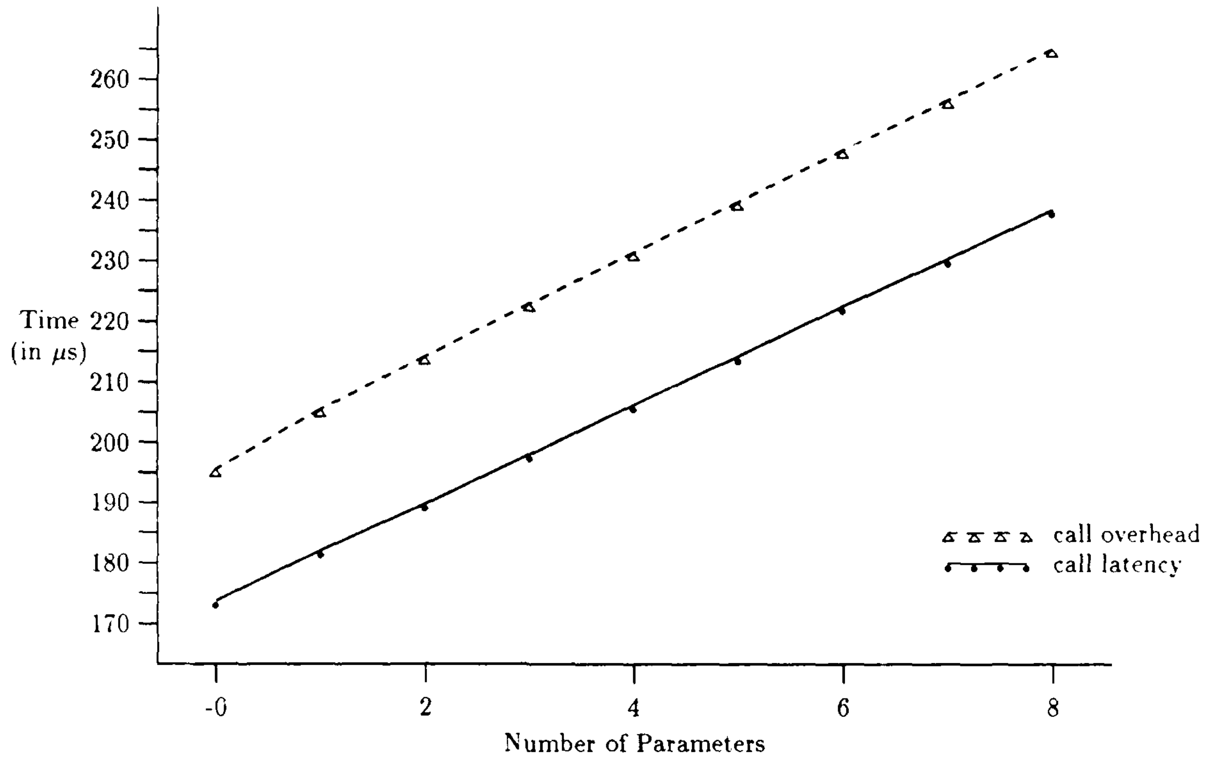

Figure 1: IIP Execution Times.

caller to issue a request and process the reply must be 79μs. The linear behavior of this experiment indicates that the high-level contention for the object is not accompanied by low-level contention in the IIP mechanism; the time to service and reply to a request using the IIP remains constant.

4 Comparing the IIP to the Alternatives.

An evaluation of the choice of mechanism for performing an invocation might consider only the costs of the alternatives when the operation is performed in isolation. Such an analysis would consider these factors:

1. the number of memory references to the object by the operation (the cost of remote access),
2. the size of the object (the cost of copying),
3. the overhead associated with object migration and with the IIP, and
4. the number of memory references to other data such as parameters.

On this basis, we hypothesize that an operation performing very few references is best done using remote memory access. Data migration and the IIP are the main candidates when the number of references is large. The overhead of the IIP implemented at user level is much less than that of page migration in the kernel. Unless page migration can co-locate the object with parameters and other data, the IIP will be the preferred mechanism. The results discussed in the remainder of this section confirm these hypotheses.

In general, considering each invocation in isolation will not result in the best possible performance. In any system with a lot of concurrent activity contention can be an important, if not the dominant, factor affecting performance. Each mechanism will experience different amounts of contention and each will experience it at a different level. Operations using remote memory access contend at a fine grain for memory and interconnect bandwidth, while operations using the IIP contend in software for access to the object. Because of contention, the tradeoffs among invocation mechanisms depend not just on the inherent properties of each individual operation but also on the properties of the operations with which it contends. Since it is difficult to perform analytic estimations of the amounts and effects of contention, we present some measurements to help guide our insights.

Another problem with considering an invocation in isolation is that it is only part of a larger computation. For example, if a thread of control is about to perform a long sequence of invocations on a single object, it would be appropriate to co-locate the thread and the object for the entire sequence, even though it may be cheaper to perform each individual operation using remote memory access. The optimal dynamic placement problem is sure to be at least as hard as static placement and mapping [7, 22].

4.1 The IIP versus Remote Memory Access.

To determine analytically which of the IIP or remote memory access is the faster mechanism for a given operation, we would like to have a model that predicts the overhead for that operation. If we can develop a simple, reliable model, it could be used by a compiler or run-time system to determine automatically which mechanism to use.

If an operation accesses only the invoked object and the parameters of the operation, an estimate of the overhead of using remote memory access to perform an operation is

$$t_{overhead} = 6.36 n_{reads} \mu s + 3.87 n_{writes} \mu s$$

where n_{reads} and n_{writes} are the number of remote reads and writes that are required by the operation to access the object. An estimate of the overhead of using IIP to perform an operation is given by the "call latency" curve in Figure 1.

We validated our overhead estimates using data gathered from the execution of two operations varying the number of references they performed and the mechanism used. One of the operations sums the elements of an integer vector. The other performs an integer version of *axpy* on two integer vectors.[4] This operation performs two reads and one write for each of the corresponding pairs of vector elements. By varying the lengths of the vectors we can change the number of memory references performed by both operations. Based on the estimated overheads, the execution time using remote memory access or the IIP should be equal when the vector length is 28.6 for the vector sum operation and 11 for the axpy operation.

We measured the execution time for 10000 calls to each operation using both remote memory access and the IIP. We performed both the "call latency" and the "call overhead" experiments described earlier. For the vector sum operation, remote memory access was faster up to a vector length of 28, but beginning at a vector length of 29 the IIP was faster. For the axpy operation, remote memory access was faster up to a vector length of 11, but beginning at a vector length of 12 the IIP was faster.

We believe that the results for the axpy operation varied from the prediction because the pipelining of the MC68020 at least partially overlaps a multiply with a remote access. We will see in a later section that as the number of processors accessing a memory module increases, the ability of pipelining to hide memory access latency diminishes.

[4] We consider the integer version of axpy to better approximate the ratio of floating point to memory access latencies typical of machines based on recent processor architectures, e.g. the BBN TC2000.

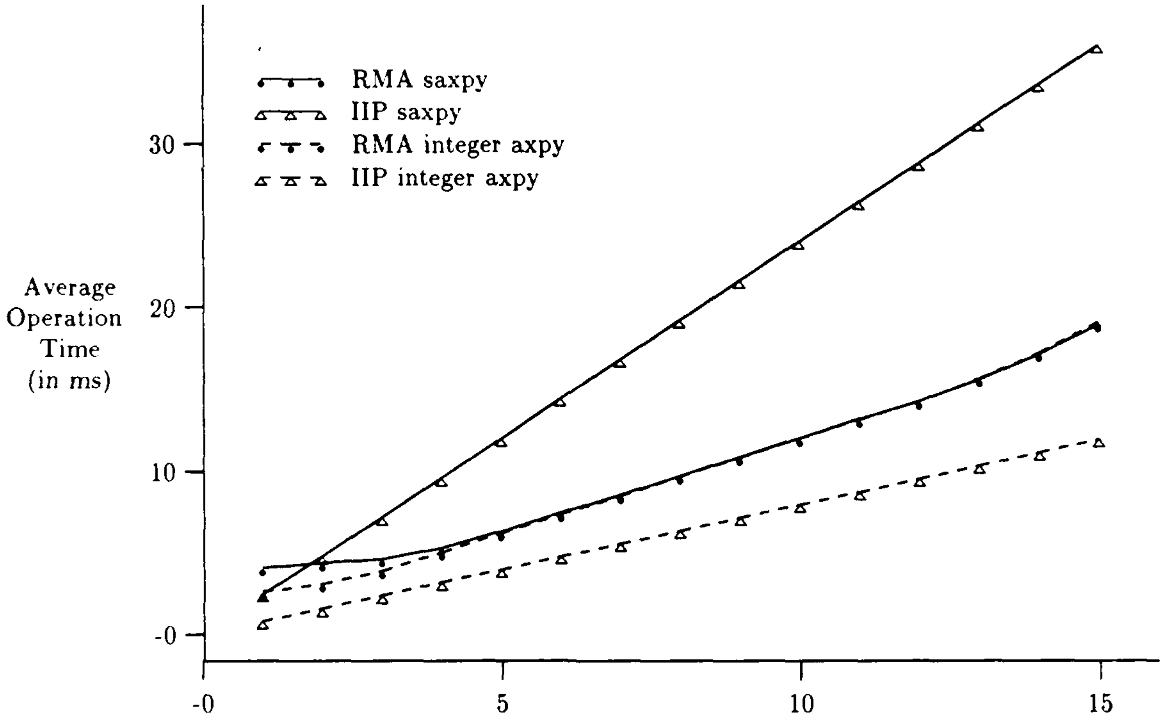

Figure 2: IIP Execution Time.

4.2 The IIP versus Data Migration.

We have previously measured the time to migrate a 4Kbyte page in the absence of contention on PLATINUM to be between 1.38 ms and 2.04 ms [10]. The migration time varies depending on the placement of the kernel data structures and whether the page is first replicated by a read and then written requiring an additional fault to invalidate other copies. Thus the number of references necessary to amortize the cost of a migration varies from 217 to 527 depending on the cost of the migration and the ratio of reads to writes.

It will always be cheaper to use the IIP to perform an operation than to move the page or pages containing the object to the caller, unless the number of parameters as well as the location and number of accesses to other data are considered. In fact, the most efficient way to perform an operation may be to use the IIP and to use data migration to move some data with the invocation.

4.3 Using the IIP to Reduce Contention.

Contention is a problem that manifests itself at many levels. In addition to memory and interconnect, tasks contend for processors and for objects. To compare the effects of contention at the object level to the machine (or memory module) level, we varied the number of processors performing 10000 invocations of the integer axpy and the single-precision axpy or *saxpy* on two vectors of length 128 located on an idle processor. We compared the use of remote memory reference to the IIP for both operations for up to 15 processors. Operations performed using remote memory access are only serialized by memory module bandwidth limitations. Figure 2 shows the average time per invocation as the number of processors is increased.

For both operations the IIP is the fastest mechanism at one processor, but from the second processor the results begin to differ. Memory bandwidth limitation is sufficient for the integer axpy to run faster using the IIP instead of remote memory access. But because floating-point operations are much slower and can be overlapped with remote memory access, remote memory access is the faster mechanism for saxpy. On the other hand, that overlap is not available when IIP and local memory operations are used.

The integer axpy and saxpy curves for remote memory access begin to converge at five processors. This indicates that remote memory reference overhead becomes the bottleneck and the added cost of floating-point operations is completely hidden by overlapping.

These experiments illustrate that concurrent memory-intensive operations on a single object will contend with one another at a low level for memory modules and interconnect. By introducing high-level synchronization in the IIP we reduce the level of logical concurrency and trade low-level for high-level contention. This reduction of low-level contention, combined with the advantages of always making local memory references, offsets the overhead of the IIP as well as the apparent loss of concurrency in the user program.

4.4 The IIP versus Explicit Synchronization.

Blocking synchronization has an advantage over busy-waiting: it allows the processor to perform useful work while waiting for a synchronization event. The use of user-level lightweight tasks dramatically reduces the cost of using blocking synchronization. For example, in OSMIUM the overhead of obtaining and releasing a local, blocking semaphore is $61\mu s$ ($88\mu s$ for a remote semaphore) in the case in which the semaphore is obtained immediately. If the task has to block, the overhead is comparable to the IIP.

The low cost of blocking means that busy-waiting should be used in far fewer cases than in other systems, and it should *never* be used *when any significant contention for a resource is anticipated.* In such cases, the IIP not only offers efficient blocking synchronization, but also the advantages discussed above.

5 Conclusions and Open Questions.

Shared memory in a parallel computer system provides convenient distributed name spaces and the means to access objects in those spaces. The judicious use of remote memory operations can be very valuable. Used either explicitly or implicitly in the form of a cache miss, a remote memory operation can be much more expensive than a local operation, for example, a cache hit, but it is far less expensive than a messag-passing operation that uses both memory components and processors. Access to shared memory is a very valuable resource and should be used wisely.

Our experiments using the IIP show that even with shared memory the use of function shipping can yield higher performance than the direct, naive use of shared memory. For certain classes of operations the IIP can dominate memory access just on the basis of latency reduction. In a highly-concurrent program, however, there will be contention for various abstractions. By moving the synchronization and communication to a higher, more abstract level of the system, a programmer or execution environment can reduce the total amount of contention. While these results were obtained on a NUMA system in which data migration is performed by software, they are also applicable to systems in which migration is performed by coherent hardware caches.

Based on the measurements presented here, we have begun to explore automatic techniques for choosing among the mechanisms, using both static compiler analysis and dynamic protocols in the run-time system and kernel.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[2] BBN Laboratories, Cambridge, Massachusetts. *Butterfly Parallel Processor Overview*, June 1985.

[3] Brian N. Bershad. *High Performance Cross-Address Space Communication*. PhD thesis, Department of Computer Science and Engineering, University of Washington, June 1990.

[4] Roberto Bisiani and Mosur Ravishankar. PLUS: A distributed ired-memory system. In *Proceedings of the 17th Annual International Symposium on Comp..er Architecture*, pages 115–124, June 1990.

[5] Andrew P. Black and Yeshayahu Artsy. Implementing location independend invocation. *IEEE Transactions on Parallel and Distributed Comuting*, 1(1):107–119, January 1990.

[6] Andrew P. Black, Edward D. Lazowska, Jerre D. Noe, and Jan Sanislo. The Eden project: A final report. Technical Report 86-11-01, Department of Computer Science, University of Washington, November 1986.

[7] Sahid H. Bohkari. *Assignment Problems in Parallel and Distributed Computing*. Kluwer, Boston, 1987.

[8] Jeffery S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 147–158, Litchfield Park, AZ, December 1989.

[9] D. Cheriton, A. Gupta, P. Boyle, and Hendrik Goosen. The VMP Multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 410–421, June 1988.

[10] Alan L. Cox and Robert J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 32–44, Litchfield Park, AZ, December 1989.

[11] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[12] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, June 1988.

[13] Allan Gottlieb. An overview of the NYU Ultracomputer project. In Jack Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 25–95. North-Holland, 1987.

[14] Gary M. Johnston and Roy H. Campbell. An object-oriented implementation of distributed virtual memory. In *Proceedings of the USENIX Association Distributed and Multiprocessor Systems Workshop*, pages 39–57, 1989.

[15] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, pages 109–133, February 1988.

[16] James T. Kuehn and Burton J. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings: Supercomputing '88*, pages 28–34, 1988.

[17] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, June 1990.

[18] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, September 1986.

[19] John M. Mellor-Crummey and Thomas J. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 1989.

[20] Da[illegible] V. Pitts and Partha Dasgupta. Object memory and storage management in the *clouds* kernel. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 10–17, June 1988.

[21] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[22] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, Massachusetts., 1989.

[23] Burton J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[24] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. Technical report, Department of Computer Science and Engineering, University of Washington, February 1990.

[25] Bjarne Stoustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[26] Dominique Thiebaut and Harold S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.