Optimizing IPC Performance for Shared-Memory Multiprocessors

Benjamin Gamsa
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4
Email: ben@cs.toronto.edu

Orran Krieger and Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 1A4
Email: okrieg@eecg.toronto.edu
Email: stumm@eecg.toronto.edu

Abstract

We assert that in order to perform well, a shared-memory multiprocessor inter-process communication (IPC) facility must avoid a) accessing any shared data, and b) acquiring any locks. In addition, such a multiprocessor IPC facility must preserve the locality and concurrency of the applications themselves so that the high performance of the IPC facility can be fully exploited.

In this paper we describe the design and implementation of a new shared-memory multiprocessor IPC facility that in the common case internally requires no accesses to shared data and no locking. In addition, the model of IPC we support and our implementation ensure that local resources are made available to the server to allow it to exploit any locality and concurrency available in the service. To the best of our knowledge, this is the first IPC subsystem with these attributes.

The performance data we present demonstrates that the end-to-end performance of our multiprocessor IPC facility is competitive with the fastest uniprocessor IPC times.

1 Introduction

The design of the inter-process communication (IPC) facility of a shared memory multiprocessor seriously affects performance and scalability. The IPC facility must maximize locality and concurrency, and export these attributes to applications and servers. In particular, it should efficiently enable independent requests to be serviced in parallel, whether they originate from a large number of different programs or a smaller number of large-scale parallel programs, and whether they are targeted at one or many servers. The IPC facility must also allow locality in the interactions between the clients and the servers (such as a client repeatedly accessing the same server resource) to be easily exploited.

In this paper we describe the design and implementation of a new shared-memory multiprocessor IPC facility that in the common case internally requires no accesses to shared data and no locking.[1] In addition, the model of IPC we support and our implementation ensure that local resources are made available to the server to allow it to exploit any locality and concurrency available in the service. To the best of our knowledge, this is the first IPC subsystem with these attributes. The performance data we present in Section 3 demonstrates that the end-to-end performance of our multiprocessor IPC facility is competitive with the fastest uniprocessor IPC times.

To date, the majority of the research on performance conscious IPC has been done on uniprocessor systems. Excellent results have been reported for these systems, to the point where Bershad argues that the IPC overhead has become largely irrelevant [1]. For example, Liedtke reports requiring 60 μsecs on a 20 MHz 386 system and 10 μsecs on a 50MHz 486 system for a null, round-trip RPC [12]; a recent version of Mach requires 57 μsecs on a 25 MHz MIPS R3000 and 95 μsecs on a 16 MHz MIPS R2000 [1, 8]; and QNX requires 76 μsecs on a 33MHz 486 [11]. These implementations apply a common set of techniques to achieve good performance: i) registers are used to directly pass data across address spaces, circumventing the need to use slow memory [6]; ii) the generalities of the scheduling subsystem are avoided with hand-off scheduling techniques [4, 6]; iii) code and data is organized to minimize the number of cache misses and TLB faults; and iv) architectural and machine-specific features are exploited or avoided depending on whether they help or hinder performance.

While in the multiprocessor case it is important to exploit all of the optimizations listed above, additional work is also required. In particular, direct translation of the uniprocessor IPC facilities to multiprocessors generally results in accesses to shared data and locks along the critical path. In a multiprocessor, accesses to shared data can result in cache misses or increased cache invalidation traffic which can add hundreds of cycles to the cost of an operation. (The relative cost of cache misses and invalidations is still increasing as processor cycle times are further reduced.) With the increases in efficiency of IPC implementations, locks can quickly saturate, even if the critical sections are very short.

The new IPC facility we introduce in this paper is based on the Protected Procedure Call (PPC) model rather than a message passing model. In the PPC model, a client process is thought of as crossing directly into the server's address space when making a call. This model together with our implementation has a number of important properties. First, the model inherently provides as much concurrency in the server as the requesting clients, while the implementation uses no locks in the common case thereby imposing no constraints of its own on concurrency. Second, no

[1] Although our approach works equally well on smaller bus-based systems, we are primarily concerned with shared-memory multiprocessors that can scale to hundreds of processors. Large scale shared-memory multiprocessors generally have memory distributed among the processors, such that the access time and available bandwidth varies with the distance between the memory module and the processor. Such multiprocessors are often called Non-Uniform Memory Access time (NUMA) multiprocessors.

shared data is accessed in the common case, minimizing cache consistency traffic. Finally, the model dictates that requests are always handled on the same processor as the client, allowing the server to keep state associated with the client's requests local to the client. With our implementation, the resources provided to the server to handle a request (in particular, the server's stack) are local to the processor on which the request is being serviced, hence minimizing implicit accesses by the server to shared data.

2 Implementation Overview

The common model of a system based on Protected Procedure Calls (PPCs) is that servers are passive objects, consisting of simply an address space, and that client threads move from address space to address space as they invoke services. This immediately implies that: i) client requests are always handled on their local processor; ii) clients and servers share the processor in a manner similar to handoff scheduling (since there is logically only one thread of control); and iii) there are as many threads of control in the server as client requests (again because each client provides its own thread of control). The PPC model is thus a key component to enabling locality and concurrency within servers.

Although PPCs are most naturally implemented by having the invoking process cross directly into the server's address space [2, 5, 7], our implementation uses separate worker processes in the server to service client calls. Worker processes are created dynamically as needed and (re)initialized to the server's call handling code on each call, affecting an upcall directly into the service routine.

The decision to use separate worker processes to implement PPCs is motivated by three factors.[2] First, it simplifies exception handling. Although we would like a PPC to appear similar to a traditional procedure call, we would prefer its failure modes to more closely follow those of a message exchange (for example, an exception raised against the client while executing in the server should not effect the server). Second, having the client itself cross into the server's address space would still necessitate a separate stack while executing in the server (for robustness and security) as well as a change of identity to allow the client to acquire the privileges of the server, which reduces the savings of such an approach. The third, more pragmatic factor is that having a separate worker process service PPC calls fits more naturally with the traditional process model upon which our operating system is based.

To maximize locality within the IPC subsystem, each processor independently maintains a local collection of all resources required to complete a PPC call (Figure 1). This includes a pool of worker processes for each server[3], and a pool of call descriptors (CDs) shared among all the servers for use on that processor. The per-server worker pools most commonly contain only a single worker, but can grow and shrink dynamically as needed. The call descriptors serve two purposes: they store return information during a call, and they point to physical memory used for the stack of a worker process during a call. These pools are accessed exclusively by the local processor.

[2] Similar factors were coincidentally noted in Spring [10].

[3] If a server supports multiple services, there is one pool per service.

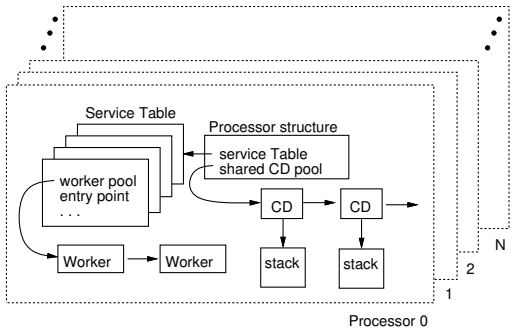

Figure 1: Per-processor PPC data structures.

When a call is made, a worker process is allocated from the server's pool, and a call descriptor (CD) is allocated from the per-processor pool; if necessary, new worker processes and call descriptors are created for this purpose. Next, the return information for the calling process is stored in the CD, and the physical memory associated with the CD is mapped into the server's address space to be used as the worker's stack. The worker is then used to perform an upcall into the server's address space and immediately starts executing the server's call handling code. When the call completes, the stack is unmapped from the server's address space, the CD and worker are returned to their respective pools, and the information in the CD is used to return control back to the caller.

The key benefits of our approach all result from the fact that resources needed to handle a PPC are accessed exclusively by the local processor. By using only local resources during a call, remote memory accesses are eliminated. More importantly, since there is no sharing of data, cache coherence traffic is also eliminated. Since the resources are exclusively owned and accessed by the local processor, no locking is required (apart from disabling interrupts, which is a natural part of system traps). By eliminating memory, network, and lock contention, the PPC facility imposes no constraints on concurrency for the system.

Since the physical stacks (and CDs) used by the worker processes are not bound to particular workers or even particular servers, but instead are assigned to workers on an as-needed basis, they are effectively recycled on each call. This improves the overall cache performance of the system, due to the smaller cache footprint that arises when multiple servers are called in succession and sequentially share physical stack pages. This also reduces the physical memory requirements of the system, since multiple servers called in succession may share a single CD and stack, and extra stacks created during peak call activity can easily be reclaimed.

A concern of reusing stacks in this way is the possible security risk of sharing stacks amongst potentially untrusting servers. This is currently addressed by permitting workers to permanently hold on to a CD and stack, allowing them to safely put sensitive information on their stack. Although, as a side effect, this allows individual calls to complete more quickly in the best case, it removes the advantages of sharing stacks, and may ultimately result in overall lower performance. A possible compromise solution would collect servers that trust each other into groups and only share stacks between servers in the same

group.

Probably the most important prior work in performance conscious multiprocessor IPCs is Bershad's Light-Weight RPC (LRPC) facility [2]. On the surface LRPC appears similar to our own. For example, it uses the same PPC model as its basic abstraction, and it is designed to minimize the use of shared data and locks. The key difference is that not all resources required by an LRPC operation are exclusively accessed by a single processor. This has implications for the LRPC facility itself as well as the servers. The LRPC facility accesses shared data which must be locked and may cause additional bus traffic. From a server perspective, the stacks used to handle the calls are not reserved on a per-processor basis, and hence the server may implicitly access remote data.

It is also interesting to observe how the recent changes in technology lead to design tradeoffs far different from what they used to be. The Firefly multiprocessor [14] on which Bershad's IPC work was developed has a smaller ratio of processor to memory speed, has caches that are no faster than main memory (but are used to reduce bus traffic), and uses an updating cache consistency protocol. For these reasons, it was far less important to avoid accessing shared data, maximize the cache hit rate, or to pass arguments across address spaces directly in the processor registers. Bershad found that he could improve performance by idling server processes on idle processors (if they were available), and having the calling process migrate to that processor to execute the remote procedure. This approach would be prohibitive in today's systems with the high cost of cache misses and invalidations.

3 Performance

The platform used for our implementation is the Hurricane operating system [13, 15] running on the Hector shared memory multiprocessor [16]. These experiments were performed on a fully configured but otherwise idle 16 processor system. This prototype system uses Motorola 88100/88200 processors running at 16.67 MHz, with 16KB data and instruction caches and a 16 byte line size. Each processor has a local portion of the globally accessible memory. Hector has no hardware support for cache coherence. Uncached local memory accesses require 10 cycles, while cache loads and writebacks require 20 cycles plus an extra 10 for the first store to a clean cache line. The Motorola 88200 has a dual context TLB (user/supervisor bit) which takes 27 cycles on our hardware to handle a TLB miss. A trap to (and return from) supervisor mode requires approximately 1.7 μsec.

Hector is a NUMA multiprocessor, with memory access costs increasing with the distance between processors and memory. However, because of the emphasis on locality in the design of the PPC facility, we found that the non-uniform memory access times had no measurable impact on performance. Hence, the NUMAness is not addressed here.

To measure the cost of individual PPC operations, we used a microsecond timer (with 10 cycle access overhead), thus eliminating some of the problems associated with microbenchmarking [3]. Figure 2 shows the performance of the PPC operations under a variety of conditions. A round trip user-to-user null call (with up to 8 arguments) requires approximately 34.1 μsec if the cache is warm. This is reduced by 3–4 μsec if the worker process holds on to its call descriptor and stack (although, as already mentioned, this may ultimately hurt performance in the average case due to lower cache hit rates).

A call to a service in the supervisor address space does not require a TLB flush and thus incurs fewer TLB misses. This brings the cost down to 23.5 μsec for the normal case, and 18.7 μsec if the worker process holds on to its call descriptor and stack.

The cost of these operations is significantly higher if the cache is not fully primed. Approximately half the instructions executed for a PPC operation are loads and stores to (local) memory, and hence their cost will vary according to the number of cache misses. For example, with the data-cache flushed before each call, times increase consistently by about 20 μsec, about half of which is due to the cost of saving registers at user level on the user stack, and half due to cache misses while manipulating the call data structures inside the kernel. Dirtying the cache and flushing the instruction cache can increase the times by another 20-30 μsec. However, the hit rates for both instructions and data can be expected to be high because this code most likely will be heavily used (and if the code is infrequently used, its performance is unlikely to be of concern).

The performance results presented in Figure 2 are for a quiet system with only a single client making calls. Results from measurements with multiple clients making requests to a common server are presented in [9], and show a near-linear increase in throughput as the number of clients is increased.

4 Concluding Remarks

We have described the design and implementation of a new shared-memory multiprocessor client-server interprocess communication system, and discussed the most important tradeoffs in its design. Although many of the design decisions were influenced by our particular hardware base, a M88000-based, non-cache-coherent multiprocessor [16], we argue that similar design decisions would apply to most current multiprocessors. In particular, the strategies used to maximize locality (such as using processor-specific worker processes), to eliminate the need for locking, and to maximize the cache hit rate (such as the serial sharing of stacks) will continue to be appropriate as long as the difference between the cost of a cache hit and a cache miss is large, regardless of whether the system has hardware support for cache coherence or not.

On our system with 16 MHz Motorola M88100 processors, a null-RPC from a client process to a user-level server and back, with 8 words of data passed in each direction, costs 34.1 μsecs if the cache is warm, and as low as 18.7 μsecs if the call is to a kernel-level server. Although multiprocessor IPC can generally be expected to be slower than uniprocessor IPC because of the need for locking, an increase in the cache miss rate and an increase in cache invalidation traffic, our IPC overhead is comparable to the best times achieved on uniprocessor systems. We believe that the overhead of our facility is close to the minimum achievable on our platform.

We have incorporated this facility into the Hurricane operating system [13, 15], and adapted most of the servers to use it. The implementation entails approximately 2000 lines of commented code, of which only 200 instructions and 6 data cache lines are

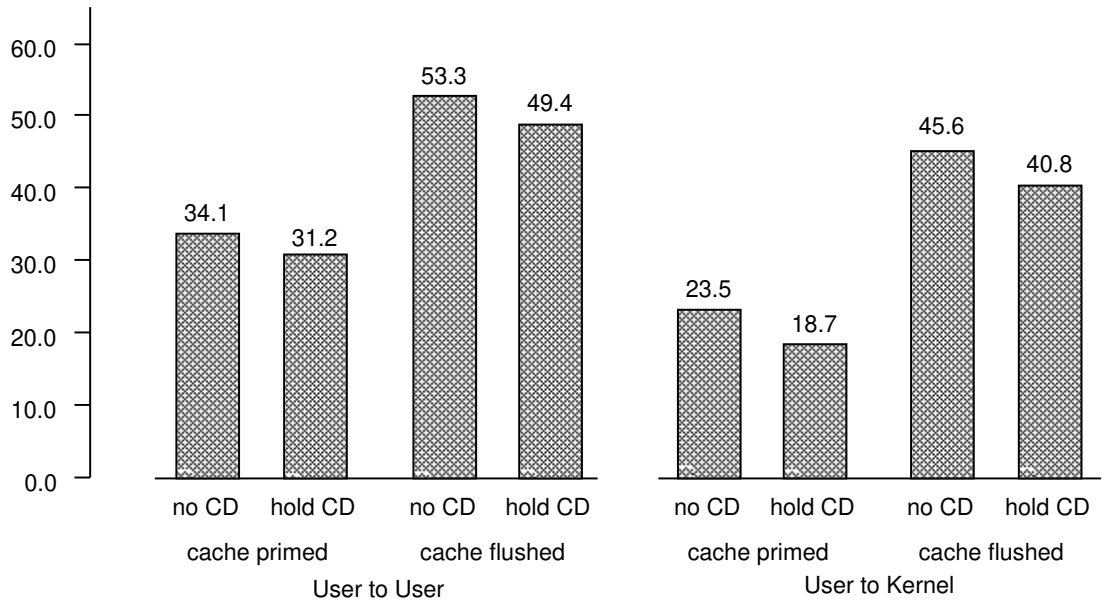

Figure 2: Times in microseconds for round-trip PPC calls for all combinations of: a user-level server and a kernel level server, with and without the cache primed, and with and without a held Call Descriptor.

required to complete most calls; the vast majority of the code is needed to handle exceptions and to integrate the new facility with the pre-existing message passing facility. Generally, not much effort is required to modify servers to use this facility. Large changes are necessary only when adapting a single threaded server to now be multithreaded (although a single lock on entry would be sufficient if concurrency is not needed).

References

[1] Brian N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–212, Seattle, 1992. Usenix.

[2] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 102–13, 1989.

[3] Brian N. Bershad, Richard P. Draves, and Alessandro Forsin. Microbenchmarks to evaluate system performance. In *Proceedings of the Third Workshop on Workstation Operating Systems(WWOS-3)*, 1992.

[4] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, 1990.

[5] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. Technical Report TR-93-04-02, Department of Computer Science and Engineering, Unversity of Washington, Seattle, WA 98195, April 1993.

[6] David R. Cheriton. An experiment using registers for fast message-based interprocess communication. *Operating System Review*, (4):12–20, 1984.

[7] Partha Dasgupta, Richard J. Leblanc, Jr., and William F. Appelbe. The Clouds distributed operating systems: Functional description, implementation details and related work. In *The 8th International Conference on Distributed Computer Systems*, pages 2–9, S. José CA (USA), June 1988. (IEEE).

[8] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 122–36. Association for Computing Machinery SIGOPS, October 1991.

[9] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. Technical Report 294, CSRI, University of Toronto, 1993.

[10] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer Usenix Conference*. Usenix, 1993.

[11] Dan Hildebrand. Architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, 1992. Usenix.

[12] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–187, 1993.

[13] Michael Stumm, Ron Unrau, and Orran Krieger. "Designing a Scalable Operating System for Shared Memory Multiprocessors". In *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 285–303, Seattle, Wa., April 1992.

[14] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr. "Firefly: A Multiprocessor Workstation". *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[15] R. Unrau, M. Stumm, O. Krieger, and B. Gamsa. Hierarchical clustering: A structure for scalable multiprocessor operating system design. Technical Report CSRI-268, Computer Systems Research Institute, University of Toronto, Toronto, Canada, March 1992.

[16] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. "The Hector Multiprocessor". *IEEE Computer*, 24(1), January 1991.