Optimizing IPC Performance for Shared-Memory Multiprocessors

Benjamin Gamsa, Orran Krieger, and Michael Stumm

Technical Report CSRI-294
January 1994

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Optimizing IPC Performance for Shared-Memory Multiprocessors

Benjamin Gamsa, Orran Krieger, and Michael Stumm
Department of Computer Science
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 1A4
Email: ben@cs.toronto.edu
Phone: (416) 978-1685

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 i) the end-to-end performance of our multiprocessor IPC facility is competitive with the fastest uniprocessor IPC times, and ii) that our implementation can sustain this performance with perfect speedup regardless of the degree of concurrency, even if all requests are directed to the same server.

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

[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.

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 i) the end-to-end performance of our multiprocessor IPC facility is competitive with the fastest uniprocessor IPC times, and ii) that our implementation can sustain this performance with perfect speedup regardless of the degree of concurrency, even if all requests are directed to the same server.

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 [2]. 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 [13]; a recent version of Mach requires 57 μsecs on a 25 MHz MIPS R3000 and 95 μsecs on a 16 MHz MIPS R2000 [2, 10]; and QNX requires 76 μsecs on a 33MHz 486 [12]. 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 [8]; ii) the generalities of the scheduling subsystem are avoided with hand-off scheduling techniques [5, 8]; 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 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[3, 6, 9], 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

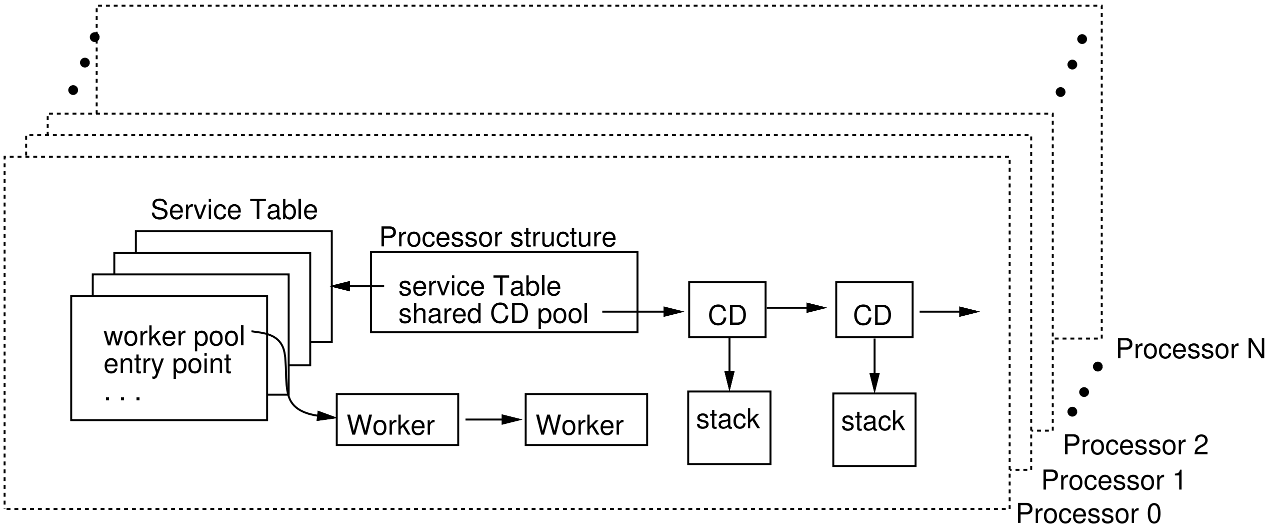

Figure 1: Per-processor PPC data structures

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 to 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 the 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.

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

[2] Similar factors were coincidentally noted in Spring [11].

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

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 [3]. 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 IPC facility itself as well as the servers. The IPC 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 [15] 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 [14, 16] running on the Hector shared memory multiprocessor [17]. 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.

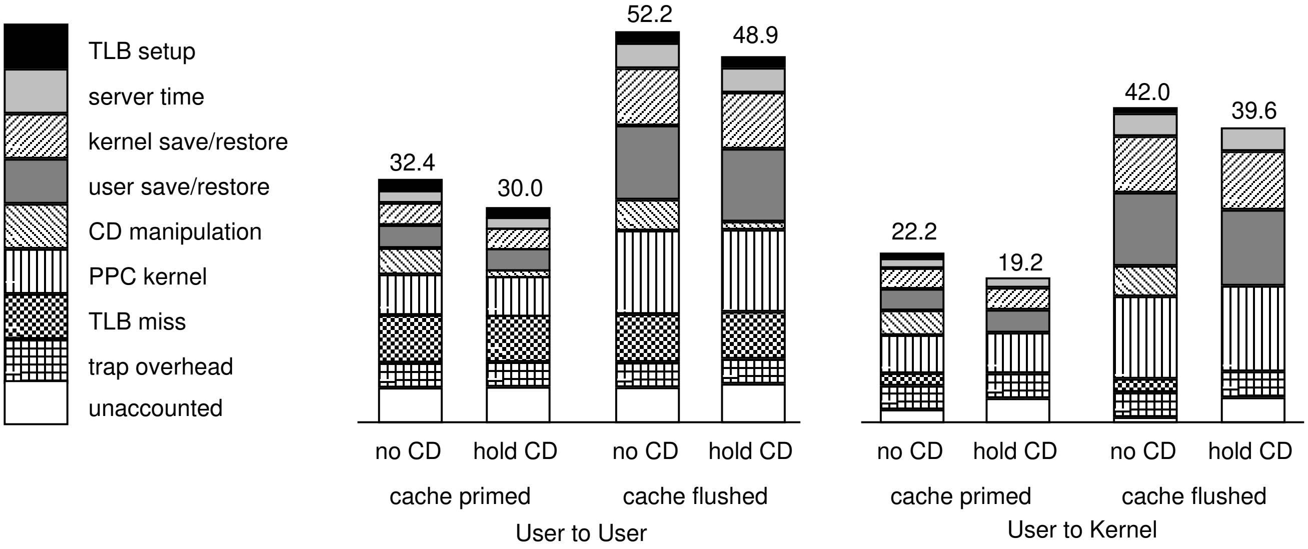

Figure 2: The breakdown provided in this graph is for the total round-trip time for a PPC (in microseconds). It is based on a detailed description of the architecture, low-level measurements, and direct inspection of the compiler generated assembly code. *TLB setup* are operations required to modify the current virtual to physical mappings; *server time* is the time spent in the worker executing the dummy server code (saving and restoring a few registers); *kernel save/restore* includes operations required to save and restore the minimum processor state required for a process switch; *user save/restore* includes the operations required to save and restore user-level registers that might be over-written during the call; *CD manipulation* includes operations to manipulate call descriptors, including free list and stack management operations; *PPC kernel* encompasses all operations other than those covered elsewhere required to implement the PPC call model; *trap overhead* is the time required for two traps and corresponding return-from-interrupts; and *unaccounted* covers the time not yet accounted for in the other categories (likely pipeline stalls, extra TLB misses, and cache misses caused by cache interference);

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 in this section.

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 [4]. Finer detail was obtained using a detailed description of the architecture, low-level measurements, and direct inspection of the compiler generated assembly code. Figure 2 depicts the breakdown of the time to performance PPC operations under a variety of conditions. A round trip user-to-user null call (with up to 8 arguments) requires approximately 32.4 μsec if the cache is warm. This is reduced by 2-3 μsec if the call descriptor and stack are locked to the worker process (although 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 22.2 μsec without a dedicated call descriptor and stack, and 19.2 μsec with a dedicated 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

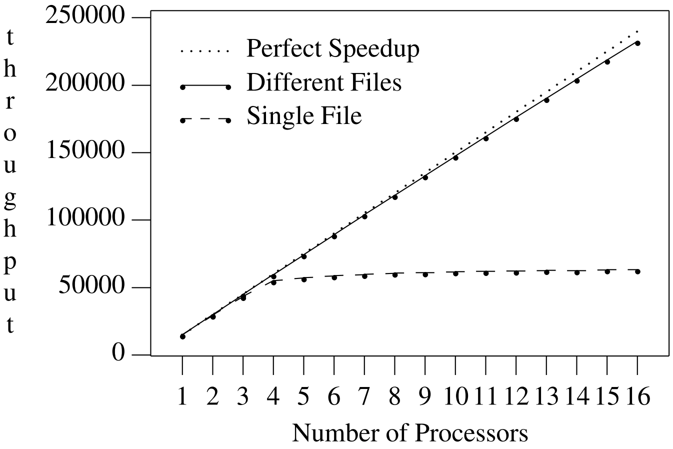

Figure 3: Throughput for independent clients repeatably requesting the length of a file from the file server. The Y axis is the number of calls per second. The X axis is the number of independent clients, each running on a separate processor. The dotted line at the top is the ideal throughput assuming perfect speedup. The solid line is the throughput actually achieved when each client is requesting the length of different files. The dashed line is the throughput when the clients request the length of a single common file.

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).

Although we feel that some performance improvement is still possible, we believe that we are fairly close to the minimum for our hardware platform. Out of all the overhead categories described in Figure 2, the only ones we (potentially) have any control over are: call descriptor manipulation, PPC kernel code, and some portion of the unaccounted-for time. Over all the test cases, the categories for which we had no control accounted for between 52% and 60% of the total execution time. Almost all of this is due to architectural "features" (large register set, trap time, TLB miss time, etc). Moreover, we believe that a significant fraction of the remaining 40-48% is either unavoidable, required to perform the basic functionality, or attributable to the basic structure of the operating system on which our facility is based.

The performance results presented in Figure 2 above are for a quiet system with only a single client making calls. To determine how well the system performs as concurrency increases, we measured the number of calls per second that can be serviced by a single server, in this case the file server, handling requests to obtain the length of an open file. The base time for the sequential case is 66 μsec, with half of the time attributable to the IPC facility and half to the file system server. The solid curve in Figure 3 shows the throughput in the case that independent clients issue the `GetLength` request to different files (but to the same server). This figure clearly shows linear increase in throughput with each processor contributing a constant increase in the rate of calls serviced. This example illustrates how significant IPC overhead can be for fine-grained interactions.

The dashed line in Figure 3 shows the throughput of clients concurrently making `GetLength` requests for a single common file. In this case the throughput saturates at four processors. Since there are only a very small number of memory accesses in the critical section in the file server, this experiment illustrates the dramatic impact any locks in the IPC path might have.

4 Design and Implementation Issues

This section discusses some of the issues that arose during the design and implementation stage of the PPC facility.

4.1 Separation of Authentication and Naming

As opposed to systems like Mach [1] or Spring [11] that use capabilities both for naming and for providing security, we specifically chose to separate the two issues. Callers are identified to servers by their program ID, which can then be used by the server to retrieve client-specific state so they can verify whether the client is permitted to make the call. This allows the service provider to choose the type of security and protection it desires. More importantly, it eliminates the need to globally change per address space data structures when capabilities are added and deleted, thus facilitating the implementation of a highly concurrent and localized system.

4.2 Moving Data Across Calls

Our PPC model provides explicit transfer of 8 words in both directions, but does not directly address how to transfer larger amounts of data. We provide a mechanism borrowed from the V system [7] where a caller may give permission to the server to read and write selected portions of its address space. The actual transfer of data is done by a separate `CopyTo` or `CopyFrom` request.[4] This mechanism is less efficient than other mechanisms that more closely integrate the bulk data transfer with the transfer of control, however we have generally found that in a shared-memory multiprocessor environment there is a stronger need to exploit service-specific techniques for optimizing data transfer (e.g. shared buffers, and virtual memory manipulation techniques) and hence did not concentrate on this aspect of our facility.

4.3 Cross-Processor Calls

Protected procedure calls only deal with the problem of crossing from one address space to another; they do not address how to transfer control between processors. In a fully symmetric multiprocessor, it is rare that one needs to cross processor boundaries, and hence it is critically important to concentrate on optimizing the local case. In those cases where cross-processor requests are necessary (usually for accessing devices, or low-level operating system functions) we currently use solutions tailored to the specific situations. For example, interactions with a disk only involve accesses to shared queues: in the case of a busy disk, appending the request to the end of the disk queue; in the case of an idle disk, additionally adding the disk device driver process to the ready queue. For completeness we do eventually expect to develop a cross-process PPC variant.

4.4 PPC Variants

We have found that the semantics and implementation of our PPC facility closely match not only the IPC requirements of our operating system, but also the requirements for other types of control transfer that appear throughout the system. Three important examples are: 1) asynchronous requests (where the requester is not blocked waiting for the call to complete), 2) interrupt dispatching, and 3) upcalls.

Asynchronous requests are implemented in the PPC facility by putting the calling process onto the processor ready-queue rather than linking it into the call descriptor of the worker. This allows the caller

[4] `CopyTo` and `CopyFrom` are normal PPC requests made to the CopyServer.

and worker to proceed independently. When the worker completes, the fact that there is no caller waiting is discovered, and another process is selected for execution. Asynchronous PPC requests are used, for example, to initiate a file block prefetch request.

Interrupt dispatching is integrated into the PPC facility using a technique similar to that used for asynchronous requests. An asynchronous request from the kernel to the device server is manufactured by the interrupt handler and dispatched as for a normal call. From the device server's point of view, it appears as a normal PPC request.

Upcalls are essentially software-based interrupts. They use the same implementation as the interrupt dispatcher, but may be triggered by an arbitrary system event, rather that just an interrupt. They have wide application, and are currently used for debugging and exception handling.

All these situations benefit from bypassing the general scheduling facility, maximizing locality, the dynamic creation of workers, and unconstrained concurrency. In the past we have used a variety of special case solutions for each of these situations. We have found that the above minor variants of our base PPC facility allow us to replace these special case solutions, resulting in greatly reduced code complexity, and in many cases substantially improved performance.

4.5 Secondary Design and Implementation Issues

4.5.1 PPC Call Interface

Normal calling conventions (for C/C++ on our platform) support passing up to eight procedure arguments in registers, but only permit one result to be returned in the registers. Ideally we would like to preserve the procedure call interface as much as possible, but have found the limit of a single result to be too constricting.

We therefore use a C macro with some assertions to the compiler (gcc) that allows us to pass the values of eight variables in a PPC call, and use those same variables to return eight values. To the user of the macro, it appears like a normal procedure call that happens to modify the arguments for the caller. By using a macro, the entire sequence of instructions required to accomplish this is exposed to the compiler for optimization, with the result that most calls generate little additional code other than the trap call.

Figure 4 shows a typical use of the PPC facility. The example shows a typical client stub on the left, and the corresponding compiler output on the right. In this example, as in many cases, the parameters of the stub library code are passed directly through to the trap call. Since exactly eight parameters are required, dummy variables are used to fill in the extra places. All that is required in the stub is to load the appropriate opcode and flags, and invoke the `PPC_CALL` macro. The return value is placed in the last parameter (by convention) and is returned to the caller.[5]

As can be seen from the compiled code on the right, apart from saving and restoring the return address, loading the service entry point number and opcode parameters, and calling the trap handler stub (required to save and restore the callee-save registers), no extra code is required (such as to marshal arguments).

4.5.2 Death and Destruction

Service entry point may be deallocated (through explicit requests or exceptions) using one of two strategies: a soft-kill removes the entry point and all associated data structures immediately, but allows calls in progress to complete; and a hard-kill frees all resources and aborts any calls in progress. The hard-kill is required in cases where the server may be faulty. The soft-kill is a more gentle way of deallocating an entry point, and may prove particularly useful in conjunction with an `Exchange` call, allowing on-line replacement of executing servers.

[5] Although this example uses only one return value, all eight variables are available.

```
int DoStuff( unsigned arg1, char *arg2, void *arg3 )   subu    r31,r31,0x0030
  {                                                    or.u    r9,r0,0x0100
    register int t5,t6,t7,opflags;                     or      r10,r0,0x0000
    opflags = PPC_OP_FLAGS( PPC_DO_STUFF, 0 ) ;        st      r1,r31,0x0024
    PPC_CALL(SOME_EP, arg1, arg2, arg3, t4,            bsr     _ppccall
                    t5, t6, t7, opflags ) ;            ext     r2,r9,0<16>
    return PPC_RC(opflags) ;                           ld      r1,r31,0x0024
  }                                                    jmp.n   r1
}                                                      addu    r31,r31,0x0030
```

Figure 4: Example PPC library call, and compiler output.

The destruction of service entry points (or workers) is complicated somewhat by the fact that all PPC resources may only be accessed from the processor they are associated with. This requires some cleanup operations to be performed by interrupting the appropriate processor.[6]

4.5.3 Worker Initialization

We have found that in some servers the workers need to execute initialization code once when they are first created (e.g. registering themselves with an exception server, or allocating a buffer, for example). In order to prevent this one-time cost from impacting the other calls, we allow workers to change their own call handling routine independently. Thus, when a service entry point is created, the call handling routine given is the initialization routine. The first call to a newly created worker enters at the initialization routine which promptly changes the worker's call handling routine, thus bypassing the initialization routine on subsequent calls handled by this worker. Although primarily targeted to one-time initialization, the call handling routine may be changed at any time.

4.5.4 Stack Size

One of the limitations of our current implementation is that we restrict stacks to one page, which is 4KB on our platform. Although this has proven sufficient for all of our current servers, we have considered a number of ways of addressing this limitation.

It is straightforward to support stacks of some fixed multiple of the page size, chosen on a service by service basis. It simply requires keeping an independent list of stack pages (rather than associating them with call descriptors) and mapping as many as required into the appropriate location. For speed, this would be treated as an exceptional case.

Another approach is to keep the current implementation and simply assign a larger virtual space for the stack. Accesses beyond the first page would result in a page fault and be handled by the normal page-fault handling mechanisms. This would keep the common case fast and only penalize those servers that require the extra space (which are likely to execute longer and more easily amortize the cost of the page-fault). Cleanup on return would be required to return the extra pages to the system, but this can also be implemented so as not to slow the common case.

[6] Many systems initiate processor-specific code execution through remote interrupts, for example, for TLB shootdown.

Finally, it is also possible for the server to acquire, internally, a larger stack if required (using whatever stack management strategy is appropriate for the server).

4.5.5 Naming

We use an approach similar to V[7] or L3 [13] where a client makes a request to a server and identifies in its arguments the object to which the request is directed. This is in contrast to Mach [1] or Spring [11] where requests are directed to the object rather than to the server that implements it.

In order for a program to become a PPC server, it must first obtain an unused *entry point ID* and call a special server (discussed further in Section 4.5.6) to bind this ID to its call handling routine. The ID can then be registered with the Name Server (which has a well-known entry point ID). A client that wants to call the server, obtains the server's entry point ID from the Name Server, and uses the ID as an argument on subsequent PPC operations. The ID is used to locate the entry point information, such as the pool of workers and address space of the server, used to complete a PPC call.

Since authentication is performed by each server and not by the PPC facility itself, small integers may safely be used as entry point IDs. By keeping the maximum number of service entry points restricted to a small number (currently 1024), a simple array with direct indexing can be used with each processor having its own copy. This results in fast call processing with an acceptable space overhead (as little as a single pointer per service entry point per processor is necessary).

In the future, we may extend the maximum number of entry points, using a fixed sized array (indexed by a small ID) to directly locate service entry points that require high performance, and using a more complex data structure (e.g. hash table with overflow buckets) to locate service entry points for the rest.

4.5.6 Frank

A kernel-level server, called *Frank*,[7] is used to manage the PPC resources. Service entry points are allocated and deallocated with PPC calls to Frank, which has a well-known service ID. Frank is also responsible for handling exceptional PPC conditions. Calls that fail due to a lack of resources (e.g. an empty worker or call descriptor list) are redirected to Frank for handling. For example, if no worker process is available during a call, the call is redirected to Frank, who creates a new worker process, initializes it for the particular target entry point, and forwards the call to the original target entry point. Frank is a normal server executing in the kernel address space, and is special only in that all its resources are preallocated, it may not block, and it may not be preempted.

5 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 [17], 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.

[7] The name *Frank* was chosen so that *Bob*, our file server, would not be the only server with an eccentric name.

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 32.2 μsecs if the cache is warm, and as low as 19.2 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 [14, 16], 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 cache lines are 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] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1993 Summer Usenix Conference*, pages 93–112. Usenix, 1986.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] D. R. Cheriton. “The V Kernel: A Software Base for Distributed Systems”. *IEEE Software*, 1, April 1984.

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

[9] 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).

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] Ron Unrau, Michael Stumm, and Orran Krieger. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. Technical report, Computer Systems Research Institute, University of Toronto, 93.

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