The Measured Performance of a Fast Local IPC

Jonathan S. Shapiro
David J. Farber
Jonathan M. Smith
Distributed Systems Laboratory
University of Pennsylvania*

Abstract

Protected application decomposition is limited by the performance of the local interprocess procedure call implementation. In this paper, we measure the performance of a new IPC implementation, and show that the cost of IPC can be reduced to the cost of raw memory copy plus a small overhead. Even on machines with poor context switching performance, our implementation compares favorably with bcopy*() for surprisingly small payloads.*

1. Introduction

Decomposing applications into separate, protected components improves the security, reliability, testability, and in some cases performance of the application. The practical limitation in such decomposition is the latency of cross-process procedure invocations within the same machine (local IPC). Recent work in Mach [6], L3 [9, 10] and EROS [13] has yielded an order of magnitude improvement in local IPC performance.

A number of current research efforts, including *SPIN* [1] and Exokernel [5], are now considering the problem of untrusted loadable modules. One option is to place such modules in individually protected processes, ensuring security without placing restrictions on source language. Unfortunately, doing so places a great deal of pressure on the IPC implementation. To determine the feasibility of process-based protection in a performance critical system, we measured the performance of this IPC system.

Previous work has shown that the limiting factor in high-speed network protocols is the performance and number of data copies made by the protocol implementation [3, 14]. In this paper, we show that the performance of a careful IPC implementation is also dominated by data copy cost for surprisingly small payloads. Among other things, our measurements substantially contradict the conclusions of Bershad and Chen [2], and show that measurements of Mach 3.0 do not generalize to properly architected and implemented microkernels. Our results are primarily a consequence of careful implementation, and should migrate readily to other architectures.

2. Preliminaries

The performance of an IPC subsystem depends on the processor architecture and speed, the memory hierarchy of the measured system, and definition of the IPC operations. Before proceeding to the measurements, we briefly describe the relevant aspects of the implementation architecture and the EROS IPC system.

2.1. Processor and Memory Hierarchy

Our initial implementation runs on the Pentium processor family [8]. Four aspects of the Pentium impact the IPC performance of the machine:

1. Switching from user mode to supervisor mode takes 75 cycles, as compared to 6 to 10 cycles on modern RISC processors. This accounts for 25.5% of the minimal (null payload) IPC time.

*This work was supported by the Hewlett-Packard Research Grants Program, the AT&T Foundation, CNRI as manager for NSF and ARPA under cooperative agreement #NCR-8919038, NSF #CDA-92-14924, and ARPA #MDA972-95-1-0013.

2. The TLB is untagged. Switching address spaces completely flushes the TLB, and the associated TLB faults account for 16.9% of the minimal IPC time.

3. The first level (L1) cache is write back, but not write allocate. For research purposes this is beneficial; since write-allocation can be simulated in software it is possible to examine the behavior of both cache policies on block transfer operations.

4. The Pentium architecture implements two integer pipelines. In the minimal IPC, only 27.4% of the instructions are able to execute in parallel. For comparison, a good compiler is able to sustain 2-way integer parallelism close to 85% on typical application code streams. This means that a high-performance IPC implementation is mostly limited by the serial nature of its data accesses.

Except where indicated, all of the performance measurements reported in this paper were measured on a 120 MHz Pentium system with a 256K synchronous-burst second level cache and 16 megabytes of EDO memory on a TYAN Titan-III motherboard. The figures reported are in-memory benchmarks; the results should be equally applicable to any motherboard and I/O subsystem.

2.2. IPC Semantics

Like Mach 4.0 [6], EROS implements a procedure-oriented IPC system [12]. Our IPC interface includes three primitive invocations:

Send Sends a message via a named port without waiting for a response.

CallAndWait Sends a message via a named **entry port** and waits for a response.

ReturnAndReceive Returns a reply to the most recently received message, and blocks until the next invocation.

Combining the Call/Wait and Reply/Receive into single operations reduces the number of privilege crossings required by the IPC interface. Other IPC systems have implemented half a dozen variations on this basic model, most of which are functionally equivalent.

The EROS IPC system is taken with minor revisions from KeyKOS [7]. Like Mach 3.0 [11], the EROS IPC architecture provides explicitly named activation records. Every **call** invocation fabricates a **reply port**, and blocks until the reply port is invoked.[1] All copies of the reply port are consumed as a consequence of its invocation, ensuring that there is at most one return per call. Message transfer is atomic, and copies up to 64 KBytes of contiguous data from the sender to the receiver.

Unlike conventional IPC systems, EROS permits **call** invocations to be performed on reply ports and **return** invocations to be performed on entry ports. A **call** invocation on a reply port provides coroutine invocation. A **return** invocation on an entry port acts as a "tail call", and allows the return path of a sequence of protected modules to be short circuited. Figure 1 shows how tail call optimization might be used in the implementation of a network protocol stack. The performance impact of tail call support is considered in section 4.2.

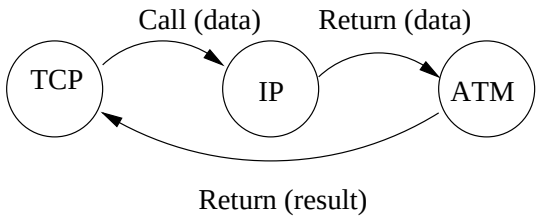

Figure 1. Short circuiting the return path

3. Block Move Performance

The purpose of an IPC invocation is to move some amount of data from the sender to the recipient. Our first step was therefore to examine the block move performance provided by three implementations: the hardware block move facility, a software implementation unwound to optimize for the L1 cache line length, and the same software implementation modified to read each destination cache line before dirtying it (simulating a write-allocate policy). IPC operations normally proceed with a warm source and cold destination, and the cold-destination results for small payloads are shown in Figure 2. In transfers larger than 4096 bytes, the write-allocate copy suffers from self-contention in the data cache, and is significantly *worse* than the no-write-allocate version.

The measurements show that there is a penalty for write allocation, but a small advantage (0.13 μS to 0.16 μS) to hand-copying rather than using the block move hardware for transfers smaller than 1024 bytes. Since this makes up for about 5% of the minimal privilege crossing time on the Pentium, we have implemented a hybrid copy design that uses the hardware block move only for large transfers. Coroutine processing is sometimes able to proceed from a hot cache, and the hybrid design does not penalize performance in this case.

[1] The EROS system uses capabilities instead of ports. Capabilities eliminate a level of indirection, but for the purposes of this paper, capabilities may be considered equivalent to ports. Rather than introduce unfamiliar terminology, this paper uses the term "port" throughout.

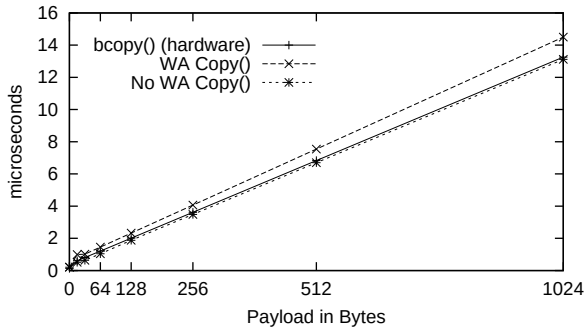

Figure 2. Comparison of bcopy with and without write-allocate, cold destination

4. Basic IPC Performance

Having chosen a suitable block move implementation, we now turn our attention to copying data across process boundaries.

The basic steps in a cross-process procedure call are as follows:

1. Enter the kernel
2. Verify that the recipient is in the proper state
3. Check if data is being transferred. If so:
   (a) Verify that the sender's buffer is present in memory and is readable to the sender.
   (b) Verify that the recipient's receive buffer is present in memory and is writable to the recipient
   (c) Transfer the data
4. Transfer any register-based information, such as the request code.
5. Transfer any ports (Mach), capabilities (EROS), or file descriptors (UNIX streams)
6. Capture an activation record so that we know who to return to (mechanism varies).

No activation record is constructed in a **return** operation, but the caller activation must be consumed to ensure that at most one return occurs per call.

There are a number of techniques for doing these steps quickly [9, 10, 6]. For IPC's that include a message buffer (in our experience over 50% contain only a request or return code), the data copy step has considerable impact. At payloads of 64 bytes or more, the copy step dominates the transfer performance. Because EROS IPC unconditionally transfers four ports, the data copy step (including address probes) is the only portion of this algorithm that has variable latency.

4.1. Measuring the Basic IPC

To test the implementation, we ran two microbenchmarks with payloads ranging from 0 bytes to 32 kilobytes:

1. *One-Way* copies a request code and some message data from client to a server. The server replies with a single "OK" result code. The request and result codes are transferred directly from sender registers to recipient registers; no address space checks are required to return the response code, and no string move occurs in the reply.
2. *Echo* copies a request code and some message data from the client to the server. The server echoes whatever is received back to the client verbatim. The echo benchmark approximates the behavior of hot-cache coroutines.

No processing of the payload is done by client or server. In the *One-Way* microbenchmark, the received data is not examined by the recipient; the one-way benchmark (Figure 3) therefore reports cold-destination results.

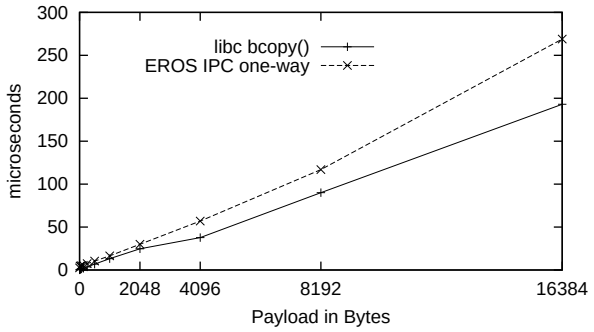

Figure 3. IPC send vs. payload (send only)

The null IPC (empty payload) round trip time is 4.9 μS. At 4096 bytes, the overhead of IPC relative to local copy is 50.8%. At 32 KBytes, it is 33.2%. The slopes of the lines, however, are very close. The discrepancy above 2048 bytes is due to differences in the cache alignment of the buffers between the bcopy test and the EROS IPC test case.

The effects of cache residency can be seen in the *echo* results. In spite of the fact that *echo* transfers twice as much data as *one-way*, *echo* outperforms *one-way* for small transfers (Figure 4) due to the fact that the destination buffer is cache resident.

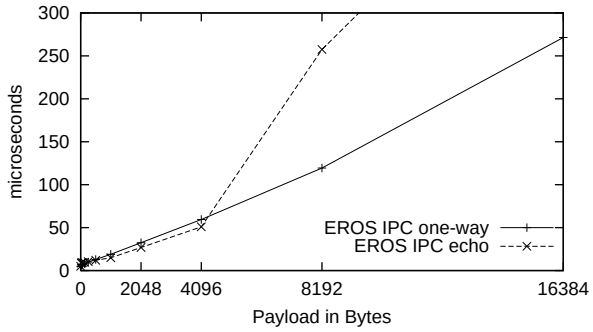

Figure 4. One-way vs. echo IPC (round trip)

4.2. Tail Call Optimization

One difference between the EROS IPC system and previous IPC implementations is the support for tail call optimization, which proves to be useful in stream-oriented processing. Figure 1 illustrates a tail call implementation of the Internet Protocol (IP). In this implementation, the IP module performs a **return** operation to the ATM **entry port**, forwarding the **reply port** that was generated by the call from the TCP module. The ATM module accepts this message, and returns to the **reply port** supplied by the IP module, which is in fact a direct reply to the TCP module.

To measure the effect of the tail call optimization, we set up three processes as shown in Figure 1, and compared the performance to the use of conventionally nested IPC. The first module passes data payload to the second which forwards it to the third, who responds with a return code directly to the originator. The marginal cost of the extra return exceeds 4.9% for payloads up to 2048 bytes. (Figure 5).

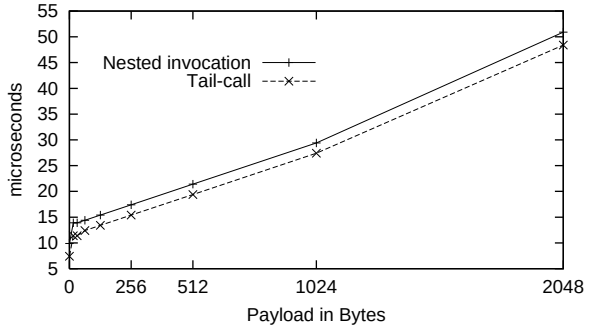

Figure 5. Performance of 2-hop IPC

The performance benefit of tail call is most apparent for small payloads, ranging from 25.25% on null IPC's down to 9.34% for 512 byte transfers. Past this point, the transfer cost dominates. The tail call optimization is therefore especially important for small-payload applications such as serial line discipline management, small packet routing, user-level demultiplexing, or connection setup. The IPC bandwidth across the two processes using the 2-hop mechanism and 16 byte packets is 2.16 megabytes/sec, as compared to 1.61 megabytes/sec without this optimization (34.1% improvement). Tail call is also useful for applications such as databases where the large payloads tend to be on the return path.

Unlike multihop invocations, the tail call optimization eliminates a cross-process copy at each stage of the pipeline. As a result, the advantage relative to conventional IPC grows as the invocation depth increases. In principle, the tail call optimization should save at least 2 μS for each invocation eliminated on this machine.

Tail call can sometimes be combined with shared memory to further improve the benefit. A reasonable engineering compromise in the implementation of the TCP/IP stack would be to have the TCP and IP modules share a memory region containing a packet buffer, and have TCP simply pass to IP the pointer or name of the packet. In such an implementation, both the extra data copy and the extra reply are eliminated. This is similar to the single-copy TCP implementation used by Hewlett Packard in the afterburner implementation [4].

5. Interpreting the Results

In evaluating interprocess communication, the cost of null round-trip IPC operations is usually compared to the cost of procedure calls or the native block move operation. This seems an unfair comparison. Having chosen the metric that maximally penalizes the IPC invocation, the conclusion that IPC is unacceptably expensive is immediate. Regrettably, this conclusion is accurate for many popular IPC implementations (Windows/NT, Windows/95, Mach, Chorus, UNIX, VMS). In spite of the inherent unfairness of the comparison, the EROS IPC implementation compares surprisingly well to the native block move operation.

In practice, large systems are not decomposed at arbitrary procedure boundaries. Appropriate decomposition points are those places where a significant data payload must be copied across a functional boundary within the application, or where the routine invoked is long-running relative to the IPC cost (e.g. sorting).

5.1. Marginal Overhead

A common approach to understanding IPC performance is to examine the percentage overhead relative to a data transfer within a single process using *bcopy*(). In the EROS

IPC system, the marginal overhead for 8192 byte transfers is 29.9%, and for 16384 byte transfers is 39.37%. While high, this is more than a factor of 10 improvement in overhead relative to previous generation IPC systems. The user/supervisor crossing penalty (1.2 μS per round trip) would be further reduced on a RISC processor, and an additional 1.2 μS are eliminated on tagged TLB's and single-address-space machines (HP-PA, RS6000, PowerPC).

By contrast, the Pentium Pro processor is 40% *slower* at null IPC operations than the Pentium. The IPC path is characterized by serial data dependency. Because of its deeper pipelines, the Pentium Pro operates at a disadvantage on serially data dependent code sequences.

5.2. Throughput

A more useful way of examining IPC performance is to examine throughput relative to the cost of conventional local interprocess communication mechanisms. To do so, we first measured the EROS IPC throughput at various payloads, where the cost of a single transfer includes the cost of the reply. Using *ttcp*, we then measured the throughput of UDP and TCP sockets for the same payloads over the IP loopback driver. Finally, we modified a version of *ttcp* to measure local pipe bandwidth. These provide reasonably good metrics for stream-oriented processing.

TCP implementations ordinarily aggregate requests on the sender end of the connection. This feature improves stream throughput (the normal case), but must be turned off to avoid delays in remote procedure call invocations. To get a first-order comparison to RPC/TCP performance, we used *ttcp* to measure TCP throughput when aggregation was disabled by setting `TCP_NODELAY`.

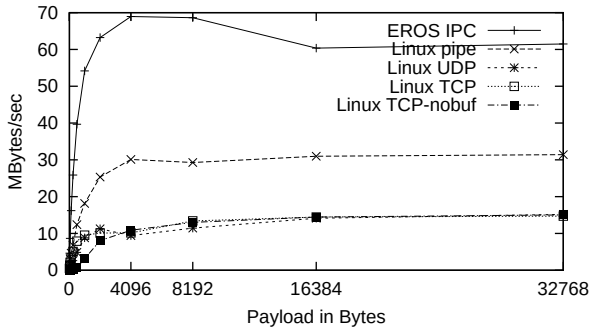

Figure 6. IPC Throughput

The results are shown in Figure 6. Raw data may be found in Table 1. Throughput peaks at 4096 byte payloads, and is sustained until the size of the cache is reached at 8192 bytes. At this point, the source buffer no longer fits in the cache, and above 8192 the graph shows the sustainable throughput if the source buffer is cold.

For comparison, the best-case transfer performance of the current-generation PCI bus is 133 MBytes/sec, and the bandwidth of currently-deployed 100 Mbit local-area networks is 12.5 MBytes/sec.

6. Conclusions

We have shown that the latency of a fast local IPC implementation is dominated by the data copy costs intrinsic to the memory system, even on machines with poor context switching times. In addition, we have shown that not all cache misses are equally expensive; in particular, a properly designed hardware store buffer makes a tremendous difference in the delay induced by data write misses.

The performance of microkernels depends on the quality of the system architecture and implementation, and varies widely. Previous measurements of microkernel systems, notably Mach, have therefore yielded disappointing performance. This low performance is an artifact of poor IPC implementations, rather than intrinsic to the microkernel approach to system construction.

Generalizations about performance from one system to another are at best highly misleading, and in some cases outright mistaken.

Both EROS and the L3 system [9] provide IPC latencies that compare favorably with monolithic systems, even for trivial system calls. As the ideas in these IPC mechanisms become more widely deployed, we expect to see application workloads showing better performance on microkernels than on monolithic implementations. Both EROS and L3 have room for further improvement.

6.1. Status of EROS

EROS provides a highly secure and reliable platform on which to build applications. Among other features, the system includes a single level store and a periodic system-wide checkpoint facility that enables rollback-based recovery. The implementation is nearly complete, and will be made available to the research community. The IPC implementation is essentially operating system independent, and we are actively seeking the loan of a PowerPC and a MIPS workstation to demonstrate our IPC implementation on other platforms.

Information concerning EROS, including mailing lists, can be obtained from the EROS home page at `http://www.cis.upenn.edu/~eros`.

Payload Bytes	EROS IPC	Linux pipe	Linux UDP	Linux TCP	Linux TCP-nodelay	Type
16	2.32	0.58	0.221	0.511	0.013	Serial traffic
32	4.64	1.32	0.359	1.38	0.025	RPC
64	8.65	3.10	0.682	2.15	0.05	X11 packets
128	16.20	4.65	1.376	3.33	0.10	
256	25.86	6.77	2.62	5.11	0.20	
512	39.69	12.38	4.86	7.90	0.68	
1024	54.18	18.16	8.81	9.54	3.15	
2048	63.21	25.34	11.28	10.11	8.06	
4096	68.95	30.12	9.41	10.16	10.87	Pages
8192	68.61	29.26	11.45	13.42	13.00	NFS xfers
16384	60.37	30.97	14.14	14.44	14.46	
32768	61.49	31.40	15.18	14.70	15.03	

Table 1. Throughput in MBytes/sec

6.2. Acknowledgements

Scott Nettles (University of Pennsylvania) provided ongoing commentary during the data collection phase that significantly shifted the structure and argument of this paper. Scott Alexander provided insight into TCP/IP that helped us choose what and how to measure.

The KeyKOS IPC architecture, from which EROS derives, is due to Norm Hardy, the senior architect of the KeyKOS system. The EROS project owes Norm and the entire KeyKOS team an immeasurable debt.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Backer, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the *SPIN* Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, 1995.

[2] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM, 1993.

[3] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of ACM SIGCOMM '90*, September 1990.

[4] A. Edwards, G. Watson, J. Lumley, and C. Calamvokis. User-space Protocols Deliver High Performance to Applications on a Low-Cost Gb/s LAN. In *Proceedings, 1994 SIGCOMM Conference*, pages 14–23, August 1994.

[5] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, 1995.

[6] B. Ford and J. Lepreau. Evolving Mach 3.0 to a Migrating Threads Model. In *Proceedings of the Winter USENIX Conference*, January 1994.

[7] N. Hardy. The KeyKOS Architecture. *Operating Systems Review*, pages 8–25, October 1985.

[8] Intel Corporation. *Pentium Processor Family User's Manual*, 1996.

[9] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM, 1993.

[10] J. Liedtke. Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces. Technical Report 933, November 1995.

[11] K. Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, July 1992.

[12] J. S. Shapiro. A Programmer's Introduction to EROS. Available via the EROS home page at `http://www.cis.upenn.edu/~eros`.

[13] J. S. Shapiro, D. J. Farber, and J. M. Smith. State Caching on the EROS System. In *Proceedings of the 7th International Persistent Objects Workshop*. Morgan-Kaufman, October 1996.

[14] J. M. Smith and C. B. S. Traw. Hardware/Software Organization of a High-Performance ATM Host Interface. *IEEE Journal on Selected Areas in High Speed Computer/Network Interfaces*, 11(2):240–253, February 1993.