The following paper was originally published in the
Proceedings of the USENIX MACH III Symposium
Santa Fe, New Mexico, April 1993.

In-Kernel Servers on Mach 3.0: Implementation and Performance

Jay Lepreau, Mike Hibler, Bryan Ford,
Jeffrey Law, and Douglas Orr
University of Utah

In-Kernel Servers on Mach 3.0: Implementation and Performance

Jay Lepreau *Mike Hibler* *Bryan Ford* *Jeffrey Law*

Center for Software Science
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
E-mail: {lepreau,mike,baford,law}@cs.utah.edu

Abstract

The advantages in modularity and power of microkernel-based operating systems such as Mach 3.0 are well known. The existing performance problems of these systems, however, are significant. Much of the performance degradation is due to the cost of maintaining separate protection domains, traversing software layers, and using a semantically rich inter-process communication mechanism. An approach that optimizes the common case is to permit merging of protection domains in performance critical applications, while maintaining protection boundaries for debugging or in situations that demand robustness. In our system, client calls to the server are effectively bound either to a simple system call interface, or to a full RPC mechanism, depending on the server's location. The optimization reduces argument copies, as well as work done in the control path to handle complex and infrequently encountered message types. In this paper we present a general method of doing this for Mach 3.0 and the results of applying it to the Mach microkernel and the OSF/1 single server. We describe the necessary modifications to the kernel, the single server, and the RPC stub generator. Semantic equivalence, backwards compatibility, and common source and binary code are preserved. Performance on micro and macro benchmarks is reported, with RPC performance improving by a factor of three, Unix system calls to the server improving between 20% and a factor of two, and 4–13% performance gain on large benchmarks. A breakdown of the times on the RPC path is also presented.[1] [2]

1 Introduction

The modularity of microkernel-based operating systems (OS's) provides them well-known benefits. These benefits include improved debugging facilities and software engineering improvements which result from a higher degree of system structuring. The use of separate protection domains to implement systems on top of a microkernel provides robustness; the ability to compose servers to exhibit different OS "personalities" offers flexibility. In addition, microkernels export powerful abstractions which are useful for building distributed systems and multicomputer OS's. However, this modularity and power has come at some cost. Because the modularity is strictly at the task level and tasks communicate via a powerful but "heavy" interprocess communication (IPC) mechanism[3], all inter-task interactions are expensive. The expensive inter-task operations include the common

[1]This research was sponsored by the Hewlett-Packard Research Grants Program.

[2]Appears in "Proceedings of the Third Usenix Mach Symposium," Apr. 1993.

[3]Although tasks can communicate through shared memory, much inter-task interaction is handled via remote procedure calls (RPCs). A few common I/O operations, such as Unix `read`, are often implemented with shared memory and can be reasonably fast.

case of untrusted clients calling trusted servers on the same machine, passing small amounts of data back and forth.

Although the Mach 3.0[1, 8] microkernel has shown isolated instances of performance comparable to that of macrokernel systems[5], this has not been observed in general. Many software layers and some hardware boundaries must be traversed even for the most trivial interactions between tasks. All messages exercise much of the IPC mechanism, even though common servers, such as the Unix server, use only a small subset of its capabilities. The conclusion is that for inter-task calls, Mach has failed to optimize the common case.

Instrumentation of the Mach IPC path on the Hewlett-Packard PA-RISC has shown that at least on this platform, the traditional "context switch" from one address space to another is only a minor component of same-machine Mach RPC costs. As detailed later, most of the cost is in kernel entry and exit code, data copies in the MIG-generated stub routines, and in port shuffling and related code. We have largely eliminated the last two costs in the common case.

To optimize the common case, we have constructed a separately linked OSF/1 server image (referred to as the *in-kernel server* or *INKS*) that can be loaded into the kernel protection domain at run time. We dynamically replace heavyweight IPC-based communication mechanisms with lighter-weight trap and procedure call mechanisms where possible. We have extended the RPC stub generator to produce special stubs which enact these new mechanisms. Since the change of communication mechanism is essentially a change in procedure binding, the presence or absence of these mechanisms is transparent to clients.

In the rest of this paper we discuss the goals, constraints, and design in Section 2, give implementation details in Section 3, and give status and performance results in Section 4, including breakdown of the RPC path, microbenchmarks, and macrobenchmarks. Finally, we examine related work, discuss planned and possible future work, and give our conclusions.

2 Design

2.1 Goals

Our goals were five-fold. Most importantly, we wanted to improve the *performance* of systems based on Mach — both those implementing Unix and more specialized systems, such as network protocol servers. Nearly as important, we sought to be as *compatible* as possible, as elaborated below. We wanted to make the required system transformations *acceptable*, by imposing minimal constraints on Mach program structure. We wanted to *explore the issues* involved, and finally, we wanted a sample system as a *target* for a more dynamic binding mechanism, based on an object server we have implemented[9, 10].

2.2 Compatibility Constraints

To maintain compatibility, we imposed a number of constraints on the design:

- The source for in- and out-of-kernel servers must be identical, with little use of conditional compilation.
- All changes to the microkernel must be backwards compatible. That is, old clients and servers that use normal RPC's must still run.
- New clients must be able to function with old servers. With a small amount of effort, new clients and servers should function with old microkernels. Note that for Unix programs, compatibility is not an issue, because the "emulator" encapsulates all of the interface to Mach, and will normally match the installed server.

Beyond our primary goals, we have achieved additional compatibility:

- Fully linked binaries of in- and out-of-kernel servers are identical.
- "Conformant" servers, which are defined to be those which don't directly call `mach_msg`, but only use MIG-generated stubs, currently require only a few lines of "boilerplate" source code in order to take full advantage of our system. (These could be eliminated by modest further changes to MIG, or to pieces of the Mach library.) "Non-conformant" servers run correctly, but with only a slight speed increase over out-of-kernel servers.
- The C threads library required no changes (with the exception of one bug fix).
- Changes to the OSF/1 server and emulator were modest, and predominantly stemmed from its code heritage as a macrokernel.

2.3 Design Overview

First we will outline the changes to the kernel and to the RPC stub generator. In principle, these are the only changes needed to allow "conformant" servers to run inside the kernel, with full performance. Then we will trace a client-to-server call, which will demonstrate the overall design. Later sections will describe the special problems that the OSF/1 server and emulator presented.

When loaded into the kernel, a server continues to exist as a distinct task but with two unique attributes: it shares the kernel address map (i.e., lives in the kernel address space) and its threads run in the same privileged mode as kernel threads. Two changes were made to the kernel in support of this. First, a mechanism was added to load a server into the kernel address space and instantiate it as a separate task. Second, existing support for kernel tasks had to be extended, both to support preemption of its threads and to allow them to perform system calls. In addition, to support user to server system calls, we added kernel infrastructure for server stack management and call "registration."

The other key component in our design is the RPC stub generator, which for typical Mach clients/server applications, is the Mach Interface Generator (MIG). We have produced an extended version of MIG, called KMIG, which includes facilities to supplement remote procedure calls by generating special traps through a dispatch table. The special stubs produced by KMIG allow user-mode client programs to use more efficient control paths when making calls to servers loaded into the kernel protection domain. This mechanism could also be made available to those servers, when making calls to other servers in the same domain.

If a server is linked with the corresponding server-side KMIG-generated stub routines, when it is started it can optionally be loaded into the kernel. Once loaded, the server's first action is to register with the kernel the RPC procedure ID ranges it supports as client system calls, along with a vector of pointers to its server-side stubs. These tables are automatically generated by KMIG.

The KMIG-generated client stubs are essentially conventional system call traps. The system call numbers are used by the kernel to index into the appropriate server registered table.

When a client executes a server-provided system call, the kernel observes that it is out of the normal range, finds the server's receive port from the client's send right name, and looks up the registered trap table for that server. If such a table exists, and the kernel finds the vector index within it, it sets up some server context (stack, global base pointers, and task identity) and dispatches to the associated server-side stub. Otherwise it returns an error code which tells the client to build a message and send it normally.

The server-side stub copies in to kernel space any necessary arguments and calls the actual work function. Upon return from the work function, the stub copies out any returned values and returns to its caller. The kernel then restores state, and returns to user mode. The client-side stub checks the return value for the special error code, and sends a real message if so. Otherwise it just passes back the return code, since kernel code has done all the parameter handling.

In the design outlined here for traps to in-kernel servers, Mach ports are not used as communication channels, but are still used for addressing and protection. A client task must own a send right for an in-kernel server in order for it to be able to make trap calls to that server. Also, instead of being global, trap tables for in-kernel servers are attached to the receive ports owned by that server. In a multiserver environment, this allows multiple in-kernel servers to field overlapping sets of call (trap) numbers, using message ports to distinguish between them. This contrasts with conventional (macrokernel-style) trap semantics, in which there is only one global trap table shared by all user-mode processes.

Note that in this model, user threads are performing server functions for themselves rather than having these functions performed on their behalf by a server-provided thread. If effect, the user thread has "migrated" from its task into the server task. This has numerous implications that are briefly discussed later, and in more detail in [4].

3 Implementation

To date we have experimented with the OSF/1 server and a number of special purpose microservers. The following sections detail the changes necessary to make these servers run both in and out of the kernel and also discusses the shortcomings of various aspects of the implementation.

3.1 Server Loading

An in-kernel server is currently loaded at boot time using a slightly modified version of the existing bootstrap routines. The server is assumed to have been statically linked at an address that doesn't conflict with any allocated kernel space. During an interactive boot, in addition to prompting for a server name, the bootstrap code now asks whether the selected server should be loaded in or out of kernel.

During a normal boot, the kernel creates a separate kernel bootstrap task (which shares the address space of the kernel), gives it send rights for the master host and device ports and creates a thread to run in it. Once running, this thread uses the standard external kernel interfaces to create a user task and thread, load its address space with the server image and send it the host and device ports. This kernel thread then becomes the initial thread for the default pager.

For the in-kernel case, the kernel creates a second kernel task and thread in addition to the previously described bootstrap task. This second task also shares the kernel address space and will become the server. The bootstrap task is given send rights to the task and thread kernel ports for the eventual server. When the bootstrap thread begins execution it skips creation of a user task and thread and uses the kernel created equivalents instead. From this point on, the bootstrap procedure continues as in the out-of-kernel case. It should be noted that even though the server is now loaded in the kernel's address space, it is still pageable.

3.2 Server/Kernel Interaction

At the current time, we do not attempt to short-circuit any server to kernel interactions. A server running in-kernel still has the same kernel interface as it would running out of kernel. Though considerably simplifying our job, there were still some problems.

One issue that arose is the semantic difference between kernel and user tasks. In standard Mach 3.0, kernel tasks are created with a special kernel function which returns an internal structure identifier instead of a port name. In order to avoid changes to the server code, this name had to be "externalized" and the in-kernel server's task had to be made to work like a normal task port name. Since the kernel only creates other kernel tasks at boot time, the amount of code affected was small and this was not a major problem.

In addition, kernel tasks are directly linked to the kernel address map, instead of having a private map referencing the same memory objects, as is the case when two user tasks share memory. If a server performs a "random" `vm_deallocate` it can actually unmap part of the kernel. Though this seems unlikely in a well-behaved server, it happens in the Unix server.[4] Our solution was to fix the offending code. A related problem is that the kernel address map contains a particular construct, a "sub_map," that does not appear in user task maps and which some of the virtual memory routines cannot handle. One of these routines is `vm_map_fork`, the address map duplication primitive called when a newly created task inherits memory from the parent. Hence, if a server attempts to create a new task inheriting memory from the server, a kernel panic results. This again happens in the Unix server where it "forks" a child task for the `init` process. In this case, since the child task never actually references the inherited memory (the address space is almost immediately deallocated) it was easy to work around.

Though not strictly necessary for correct behavior of the servers involved, server thread preemptibility was implemented. This is a problem unique to in-kernel servers since server threads are running as kernel threads and Mach kernel threads are not preemptible. This creates a potential mismatch with server threads which might assume preemptibility and that, in turn, could lead to latency problems in the kernel.[5] To be safe, kernel threads running server code are now subject to rescheduling at the "traditional" points (leaving traps, interrupts, system calls). There were two primary difficulties in doing this. One was differentiating a kernel thread running server code from a normal kernel thread. The other was in implementing a new mechanism to force a rescheduling trap since the conventional technique is to use an AST that is triggered upon return to user mode. Both were solved in highly machine-dependent, and temporary, ways.

A special, lighter-weight system call interface was introduced to handle Mach system calls from the server to the kernel. Since it is a kernel-to-kernel transition, less state saving and setup were needed and there was no need to check for emulated (i.e., server-handled) system calls. Measurement of a "do nothing" kernel call (`task_terminate` with a NULL argument) shows that this modified path cuts over 30% from the overhead of the standard system call interface.

Saving server state when entering a system call presented a minor problem. In the normal server/kernel model, a user thread (application or OS server) has two sets of saved state. Whenever it traps into the kernel (system call or interruption) some or all register state is saved in the process control block (PCB) in a "saved state" structure. When it is context switched (always from within the kernel) the volatile kernel state for the thread[6] is saved at the base of the active kernel stack. In the INKS world, an additional level of state saving is necessary since a user thread may first trap into the kernel to perform a server function and then, in the context of the server, trap into the kernel again to execute a Mach service. The latter cannot use the standard saved state structure since it contains state from the initial user to server transition. In our implementation we allocate another saved state structure in the PCB for use during server to kernel transitions.

Kernel stacks were another source of problems. In principle, during a Mach system call an in-kernel server thread could continue to run on its own stack instead of switching to a kernel allocated stack. This would eliminate the need to copy arguments, switch stacks, and in general, would simplify the system call path. The problem, however, is that the special optimized paths through the existing kernel assume they have complete control of the stack the thread is running on (not an unreasonable assumption in a "pure" kernel environment). In particular, continuations are a problem because their effect is to throw away the contents of the kernel stack. Since there is now pre-system call server state on the "kernel" stack we could not allow this.[7]

Though general disabling or modification of the continuation system and the related stack handoff code is a possibility, it would have required code changes we judged too extensive at this time. Therefore, in order to preserve the semantics of kernel stacks, we chose to have the server run off of its own stack and switch to a kernel stack on system call entry. This means we now need to copy

[4] On booting, the server deallocates page zero of "its" address space but since the server is no longer loaded at zero, this deallocates the first page of the kernel.

[5] As an extreme example, a naive server running in-kernel might loop on a spin lock causing the kernel to deadlock.

[6] There may be no volatile state if a continuation was specified at context switch time.

[7] We implemented a prototype of this and obtained about a 35% speedup of the server-kernel system call path. As a workaround in this prototype, the affected continuations were identified and disabled. (This allowed further streamlining of the system call path as we no longer needed to save callee-save registers for thread_syscall_return.)

arguments, save registers for continuations, and do some additional stack management. Essentially, we were forced to retain much of the original system call path, even though no protection boundary is being crossed. We still save some overhead, but the principal performance benefits arise from coupling this with changes to the RPC stubs.

3.3 User/Server Interaction

KMIG treats specially routines with "normal" arguments and message options (in particular, no port rights or out-of-line memory regions). It generates client stubs which are merely system calls whose number corresponds to the `msgh_id` number (RPC subsystem base + procedure ID number). The user stack already contains the arguments, including the server's port name, just as the user supplied them in the C-style function call, so no argument copying is needed.

When a trap is made to the kernel, it first determines whether it is a "normal" trap (to Mach primitives or Unix emulation) or a trap to an in-kernel server. This distinction is easy to make because normal traps use small positive or negative numbers, while INKS traps, whose trap numbers correspond to `msgh_id` numbers, are large positive numbers.

When the kernel determines that the trap is to an in-kernel server, instead of looking up the trap in a global table as normal traps are handled, it first determines *which* server is being called. It retrieves the server's port name, which the user supplied as the first parameter in the call[8], and decodes it in the context of the client's IPC space to find the destination port and the task in which it resides (the server task). From that task, it extracts the trap table previously registered by the in-kernel server and looks up the stub entrypoint based on the `msgh_id` number originally supplied as the trap number.

Before calling the server-side stub, the kernel switches from its stack to a new server-provided stack. Note that this violates our "ideal" in that it requires server changes to make stacks available for trapping clients. There are three major reasons why we must do this. First is the continuation/stack-handoff issue discussed earlier: remaining on a kernel stack while running in the server limits optimizations that the kernel can do when the server later makes a Mach system call. A second, more serious problem is that server routines may make assumptions about the layout of the stack they are running on. For example, the OSF/1 server allocates a thread specific data structure (the `uthread` structure) on the stack of every service thread.[9] All references to this structure are made via a constant offset from the base of the current stack, which is assumed to be a fixed size Cthread stack. Hence, not only does the server have knowledge of the layout of the stack but it also make assumptions about its size and alignment. Also, since OSF/1 service threads are Cthreads, there is additional Cthread-specific state on the stack as well. Finally, kernel stacks are relatively small and wired-down in the kernel. The former means that they may not be large enough for complex or deeply nested server calls, while the latter could lead to excessive wiring of physical memory.

After the kernel has switched to a service stack, the system call arguments are copied from the user's address space to the server's. (Note that these are normal, short, call-by-value arguments.) The kernel's final action before the KMIG-generated server stub is called, is to change the current thread's "task identity" to that of the server task. Since our desire is to run server code largely unchanged, we must preserve the notion that it is running in a separate task. Hence, server code which performs operations on "mach_task_self" should affect the in-kernel server task and not the user task, even though it is running from a user thread. Our current approach to solving this is ad hoc. When a user thread traps into the server, we change its containing-task pointer to that of the server task. We do not currently remove the thread from the task's thread list and insert it into the server task's.

Once the server stub receives control, it allocates local variables for the storage of pass-by-

[8] MIG allows the request port name to be in other positions in the argument list, but we do not know of any case in which this feature is actually used. If this feature were to be used, KMIG would simply generate a standard message stub for that procedure.

[9] This structure contains process information that is only valid while in a system call, so the service thread stack is a convenient place for it.

reference or `Out` arguments, copies pass-by-reference arguments from user space with the kernel's standard `copyin` procedure, and calls the server work function. Upon return from the work function, the server copies any `Out` or `InOut` arguments back into user space via `copyout` and returns to the kernel. The locations of the kernel's copyin and copyout procedures were returned as part of the server's trap table registration.

Finally, the kernel resets the thread's task pointer to its pre-call value, switches back to the original stack and exits the client's trap.

The Unix server required two deviations from the standard procedure. The first deals with copying arguments in to and out of the kernel address space and is explained below. The other is a violation of our "conformance" constraint that servers not directly call `mach_msg`. For brevity, the Unix server and emulator implement a number of services with a "hand-rolled" generic RPC stub, without using MIG. We had to similarly hand-implement a generic INKS trap to handle these system calls.

3.4 Issues

3.4.1 Thread Model

One important implementation issue we had to confront is the thread semantics employed during traps to in-kernel servers. Should the client's thread "move" into the server task for the duration of the RPC, or should the kernel switch to a "real" thread created and owned by the server? The latter adheres to existing Mach semantics, and therefore would be more straightforward. However, for our initial implementation we chose the former option, for several reasons: it offers the greatest potential for performance improvement from INKS, it tests the feasibility of such a stretching of Mach semantics, it offers the flexibility to support other specialized communication mechanisms, and we did not realize the extent to which the Unix server relied on "specializing" its service stacks.

In [4] we discuss in detail the issues related to this decision, so only a brief summary is presented here. A thread switching model would have resulted in a cleaner, safer[10] implementation and would have fit better with Mach's current thread semantics. It would probably be slower than a migrating threads implementation, but perhaps not significantly overall, especially for a single server system. In the long term, however, we believe that thread-switching is only a temporary way to avoid a problem in Mach. If a general-purpose migrating threads model such as that exploited in LRPC[3] is introduced to Mach in the future, this approach to in-kernel server traps will become both faster and cleaner.

3.4.2 Server Task

Many of the complexities and shortcomings of the thread migration model could be avoided and the benefits retained by altogether eliminating the notion of a separate server task. By viewing the server code as down-loadable kernel code running as part of the "kernel task" we no longer have to worry about thread migration or service threads. User threads trapping into the server are just trapping into an extended kernel, and the user thread is just "running in kernel mode." The semantics of thread operations applied to user threads running in the kernel are well-defined, so threads running in server code introduce no new problems. Coupling this with short-circuiting server/kernel calls, we have essentially re-created a monolithic kernel.

There are drawbacks, however. One is the kernel/server stack issue. User threads entering the kernel will still need to switch to a service stack or "prepare" the kernel stack so it appears as the server expects. For the latter, the size and non-pageable nature of kernel stacks would again be a concern. In either case, the kernel would have to be modified to avoid using the continuation and stack handoff optimizations or those optimizations would have to be modified extensively. This

[10] Currently, when a thread migrates into a server, it can still be manipulated by any other thread that has access to its thread control port.

approach also does not address the preemptibility problem. Since server threads would now be true kernel threads, they would no longer be preemptible, leading to all the attendant latency problems. It is also not clear how much an existing server would need to be changed to fit this model. While many of the kernel to server interface issues can be addressed by KMIG or more sophisticated linking technology, much depends on what assumptions a server makes about its address space and other resources.

3.4.3 Trap Tables

In our current implementation of INKS, the trap tables for an in-kernel server are attached to its task, instead of individually to each of its service ports. While this simplifies the implementation and works well in the case of the Unix server, it may have to be changed in the future. The problem is that, because one receive port in the server's task is not distinguished from another, *any* send right whose receive right is in the in-kernel server's task may be used by other tasks to make INKS traps into the server, even if that port is not one that would respond to corresponding messages. This is not a problem in the Unix server, because there is only one "type" of service port that clients know about: the BSD request port. However, there could be a problem in a server that exports multiple "types" of send rights, each responding to a different set of request messages.

There are two apparent ways of solving this problem. First, if the implementation of INKS is left the same, server work functions could simply check the receive port that corresponds to the send right the client used to make the trap, ensuring that the call being made is in fact valid for the given port. However, a better and more "correct" way to do it would be to simply change INKS so that trap tables are associated with individual receive ports in the server, instead of the server's task. This would be somewhat more difficult to implement than the current strategy, but we foresee no major problems.

3.4.4 Copyin and Copyout

As described earlier, KMIG generates calls to the kernel copyin and copyout routines to deal with copying arguments and results across the user/kernel boundary. This led to an unexpected inefficiency in the Unix server. Since it is derived from a monolithic kernel, most of the system call service routines already do their own copyin and copyout. Hence we were doing two copies for every data item, in or out.

To solve this, KMIG was modified to optionally omit generation of all copyin and most copyout calls. The only KMIG-generated copyouts which remain are for integer `Out` parameters which are traditionally return values.

Unfortunately, eliminating KMIG-generated copies uncovered a few places in the Unix server where arguments were not being handled consistently. For example, the `bsd_execve` routine (not inherited from the monolithic kernel) dereferences some of its out pointer parameters assuming they will be accessible in the server's address space. However, the KMIG stub no longer allocates a local array for these parameters, expecting the service routine to do its own copyout. We view this as a server problem and fixed the offending routines.

In total, the changes to perform this copy optimization in the OSF/1 server were limited to 3 files and 7 functions. Whether the KMIG option to suppress copies is generally useful or peculiar to the Unix server remains to be seen.

3.4.5 Miscellaneous

Finally, there were a number of other issues that came up and were not resolved to our satisfaction:

- The problem of inheriting an in-kernel server's address space (i.e., the kernel address space) on task creation needs to be addressed. Although the problem was easily avoided in the case of the Unix server, in general, other servers may want to export portions of their address space.

- A related creation issue is whether a kernel task should be able to spawn only kernel tasks, just user tasks, or both. Currently, `task_create` will always create a user task, which is sufficient for the Unix server.

- There is a thread analog to the "task identity" problem discussed earlier. Just as server code may make the assumption that `mach_task_self` does not refer to the user task, it may also assume that `mach_thread_self` does not refer to the user thread, or, more importantly, that the currently running thread is not the user thread. An example of this in the OSF/1 server is the `exit` system call which attempted to terminate the currently active user task and threads, one of which is running the exit code.

- The existing fixed partitioning of a thread's saved state may need reworking. For the single in-kernel server experiments we were doing, simply adding another "saved state" structure to the PCB for server/kernel transitions was sufficient. However, introducing multiple servers with server to server interactions will add arbitrary levels of state saving. Presumably such inter-server calls can be handled by saving the necessary old server state onto the new server's stack thereby requiring no additional static saved state areas.

- If a server is overloaded, with messages built up in its port queue, clients invoking it by the system call interface will get preferential treatment, since the direct invocation bypasses the port's queue. Handling this is straightforward: if the kernel finds the server's receive port has a non-empty queue, it returns MACH_SEND_INTERRUPTED, signaling the client stub to send a real message.

- To this point we have concentrated solely on increasing execution speed and have not considered the effect of in-kernel servers on memory usage or the effect of its potentially increased memory use on overall system performance. Since in-kernel servers are pageable their mere existence in the kernel does not imply increased physical memory usage. However, it does increase virtual memory use within the kernel address space.[11] The current implementation, with its private service stack pool for trapping clients, certainly increases memory usage.

4 Results

4.1 Status

The identical OSF/1 single server binary runs multiuser, in or out of the kernel. With trapping clients there are occasional robustness problems, but the system typically stays up for hours under benchmark loads (SPEC SDM, Andrew, kernel builds). Some types of signals crash the system, but we have not examined this yet.

4.2 Experimental Environment

All timings were collected on a single HP9000/730 with 32 MB RAM and one 425 MB SCSI-2 disk, on an isolated ethernet segment. The HP730 has a 67 Mhz PA-RISC 1.1 processor, 128K offchip Icache, 256K offchip Dcache, 96 entry ITLB, and 96 entry DTLB, with a pagesize of 4KB. The caches are virtually addressed with a cache miss cost of about 14 cycles. RPC test times were collected by reading the PA's clock register, CR16, which increments every cycle, and can be read in user mode. This was done with an inline function and `asm`. Other times are obtained from the Unix server.

[11] It has been suggested that the presense of the server in the kernel's address map may affect the performance of map entry lookups at fault time, especially on server faults where we might potentially need to scan past all the kernel entries just to reach the server's "neighborhood."

Table 1: Non-trapping RPC Test Program Breakdown

RPC Stage	Description	Out of Kernel Cycles	In Kernel Cycles
1	Procedure call to MIG stub	24	24
2	Stub operations before `mach_msg_trap` call	199	199
3	Call to `mach_msg_trap`, trap into kernel	350	369
4	`mach_msg_trap` operations before message copyin	18	18
5	Copyin message to kernel address space	205	205
6	"Red tape"	159	159
7	Context switch from client task to server task	178	178
8	"Red tape"	122	121
9	Copyout message to server, trap return, MIG demux	443	404
10	Server MIG stub processing before work function	19	19
11	Null work function (just returns)	17	17
12	Server MIG stub processing after work function	172	172
13	Server call to `mach_msg_trap`, trap into kernel	378	304
14	`mach_msg_trap` operations before message copyin	18	18
15	Copyin reply message to kernel address space	211	212
16	"Red tape"	144	144
17	Context switch from server task to client task	178	179
18	"Red tape"	90	90
19	Copyout reply message to client, trap return	364	364
20	Client stub operations after `mach_msg_trap` call	172	173
21	Procedure return from MIG stub to client	17	17
	Total	3478	3386

The operating system software is the Mach 3.0 kernel, version NMK13, and the OSF/1 single server, version 1.0.4b1 (derived from OSF/1 1.0.4). This is the port done by Utah, and does not yet include the virtual cache improvements from CMU[13]. Little of the port has been optimized, and in particular, the existing implementation of the system call/context switch path is not at all optimal. This likely affects the breakdown of RPC costs, but probably has only a marginal affect on the overall results. The compiler is GCC 2.3.3.u3 (a Utah distribution of 2.3.3 with additional optimizations for the PA). All RPC timings were made with only the micro "RPC server" running, and all Unix benchmark measurements were made with the system in single-user mode. Each Unix run was repeated at least three times. The "RPC server" timings showed essentially no variance, with only 1-3 cycle differences out of thousands.

4.3 RPC Breakdown and Analysis

In Table 1 we present a breakdown of an RPC to an out-of-kernel server, containing one `InOut` parameter of 32 words, which is used to store timestamps at successive stages of the message path. This test does not, and is not meant to, completely represent "typical" RPCs. The most common RPCs pass only a few (three to ten) discrete integer parameters, most of which are `In` or `Out`; `InOut` arguments are less common. However, the purpose of this test is not to measure the total performance of an RPC, but to determine comparatively how much time the various parts of the RPC path take. The most important requirement is that all measured code remain on the "optimized path" within `mach_msg_trap`, and our test fulfills this requirement. We now briefly describe the

Table 2: Trapping Test Program Breakdown

Trap Stage	RPC Stage	Description	Cycles
1	1	Procedure call to user stub	24
2	3	Trap into kernel, recognize the trap as INKS	213
3		Port, task, and trap entrypoint lookups	116
4		Find and allocate a service stack	22
5		Switch stacks, change task context	92
6		Call server stub	29
7	10	Server stub processing before work function (`copyin`)	232
8	11	Null work function (just returns)	20
9	12	Server stub processing after work function (`copyout`)	229
10		Return from server stub to kernel	17
11	19	Deallocate stack and return from trap	163
12	21	Procedure return from user stub to client	17
		Total	1174

contents of the table.

Stages 1–2 and 20–21 occur in the client's address space. Stage 2 is the marshalling of the request message, and stage 20 is the unmarshalling of the reply message. Note that both of these stages are somewhat higher in our test case than they would be in common RPCs, because of the larger-than-normal `InOut` parameter that must be copied into and out of the message.

Stages 3–9 and 13–19 occur mostly in the kernel's address space (in particular, in the `mach_msg_trap` function). Stages 3–9 are essentially identical in overall function to stages 13–19. (The code path executed in stage 6 is actually different from the one executed in stage 16, but the overall functionality and execution time for each is very close.)

"Red tape" (stages 6, 8, 16, and 18) is code in `mach_msg_trap` that performs administrative tasks necessary to the passing of simple messages: port name lookups, generating reply rights (in the case of request messages), consuming request or reply rights, locking and unlocking kernel data structures, as well as numerous tests for exceptional situations that would require control to be transferred off of the optimized path.

Stages 10–12 happen in the server side. Stage 12, like stages 2 and 20, is much larger than it would be for "real" RPCs. In the server, the received request message is stored in a message buffer separate from the reply message to be built and returned. While this does not matter for `In` or `Out` arguments, which are the most common, `InOut` arguments must be copied by the server stub from the request message to the reply message. Most of the time spent in stage 12 is in this copy, and if the argument was not `InOut`, the stage 12 time could be expected to be similar to the stage 10 time.

Table 2 gives the figures for the same test using the in-kernel server trap mechanism. Again, a single 32-word `InOut` argument is passed. The "Trap Stage" column independently enumerates the stages of an INKS trap, while the "RPC Stage" column lists the stage numbers from Table 1 that correspond to the stages in Table 2, where appropriate.

In the trapping case, the time spent in the client-side stub (trap stages 1 and 12) is very small because the client stub is a trivial assembly language fragment that simply loads the trap number into a register and makes the trap.

Table 3: RPC test results: cycles (ratio to trapping INKS)

Test	Configuration		
	Unoptimized	INKS Message	INKS Trap
Null RPC	2312 (3.5)	2220 (3.4)	656
64 `In`	2555 (3.1)	2462 (3.0)	830
1K `In`	5843 (3.3)	5890 (3.3)	1791
1K `Out`	8366 (3.0)	7512 (2.7)	2741
128 `InOut`	3385 (3.0)	3291 (2.9)	1122

Trap stages 2–6 contain most of the processing done by the microkernel for the INKS trap. In stage 3, the mapping from the client's port name to the kernel's `ipc_port` pointer is done, as well as the lookup of the requested trap in the server's INKS trap table. Stages 4 and 5 handle the selection, setup, and activation of a new service stack for use by the server's work function. Stage 5 also includes changing the task context and copying the client's basic (call-by-value) parameters to the new server stack.

Trap stages 7–9 correspond closely to RPC stages 10–12, except that *all* required argument copying is done in the server stub. The large times for stages 7 and 9 stem from the explicit copying of the 32-word argument into the server's address space and, later, back out to the client's. This is the *only* argument copying that is done in the trapping case, unlike the RPC case where the arguments are copied in the client stub and in the kernel as well. However, note that while the large time spent in RPC stage 12 is an artifact of the test implementation as discussed previously, the time spent in the corresponding trap stage 9 is *not* such an artifact—this copy would in fact have to be done for `Out` arguments as well as `InOut` arguments, which are quite common.

For comparison, a simple procedure call on the same machine typically takes 15 to 25 cycles.

Based on these tables, an approximate categorization can be made of the time spent in different types of processing during the RPC path. Argument copying accounts for about 1350 cycles in the RPC case, but only about 450 for the INKS trap. Trap entry and exit times are also significantly better in INKS, from about 1100 cycles down to about 400. Finally, most of the "red tape" involved in passing Mach messages is eliminated.

It is clear that much less argument copying is done in INKS traps than in RPC. Simple stack-based `In` parameters are copied only once (client stack to server stack) instead of four times (client stack to client message, client message to kernel, kernel to server, server message to server stack). Pointer-based `In` parameters are copied once instead of three times (the copy from server message to server stack is not necessary in the RPC case). `Out` parameters are similarly reduced from three copies to one. `InOut` parameters are copied only twice instead of seven times (client to message, message to kernel, kernel to server request message, request message to reply message, reply message to kernel, kernel to client message, reply message to original buffer).

4.4 Micro Benchmark results

Table 3 shows several small RPC benchmarks, run directly on the Mach kernel, with no Unix server running. Five tests were run: a null RPC, a 64 byte `In` parameter, a 1024 byte `In`, a 1024 byte `Out` parameter, and a 128 byte `InOut` parameter (the same test as in Tables 1 and 2 without the instrumentation code). The cycle count for each test under each configuration is shown, along with a ratio to the trapping INKS configuration, in parentheses. The unoptimized message configuration and INKS message configuration typically execute three times as many cycles as the INKS trap configuration. Variance in these tests was almost nonexistent; on rare occasions the results would vary by a single cycle. From the `InOut` results in the three RPC tables, one can measure the overhead

Table 4: Syscall test results: time in microseconds (ratio to trapping INKS)

Test	Configuration Unoptimized	 INKS Message	 INKS Trap
nop (emulator)	11.3 (0.95)	11.3 (0.95)	11.9
nop (server)	92.7 (2.08)	86.7 (1.94)	44.6
getpid	11.9 (1.00)	11.9 (1.00)	11.9
getgid	92.6 (2.18)	85.9 (2.02)	42.5
setrlimit	114.2 (1.94)	105.6 (1.80)	58.7
4096 byte-read	813.8 (1.19)	802.1 (1.18)	682.4
64 byte-read	310.0 (1.70)	301.2 (1.64)	183.5
4096 byte-write	489.7 (1.23)	483.2 (1.22)	396.5
64 byte-write	179.4 (1.98)	172.7 (1.91)	90.5
stat	280.5 (1.55)	273.2 (1.51)	180.9
create	535.6 (1.69)	518.8 (1.64)	317.2
signal	394.1 (1.46)	372.5 (1.38)	269.1

Table 5: Server to kernel syscalls: Time in microseconds (ratio to INKS)

	Configuration	
Test	Unoptimized	INKS
Message Syscall	18.81 (1.13)	16.71
Trapping Syscall	6.86 (1.44)	4.75

imposed by instrumenting the message and trap paths. It is both minimal and consistent: 2.6% (93 and 95 cycles) for the two message paths, and 4.6% (52 cycles) for the trapping path.

As an aside, during this work we discovered a source of huge inefficiency in the standard MIG-generated stubs. To copy an argument "struct" into and out of messages, MIG generated a structure containing a single `char` array, and used structure assignment to effect the copy. Both GCC 1.39 and HP's optimizing C compiler generate code which copies a byte at a time, even if the array is word-aligned (as it usually is). Moving to GCC 2, or replacing the code with a `bcopy`, improved RPC message performance by a factor of two; before this INKS trapping RPCs attained a factor of six speedup. The lesson is that large potential improvements can lie in mundane and seldom-examined portions of the system.

Table 4 shows timings for a small variety of arbitrarily selected, but hopefully representative, Unix system calls. These were the only system call benchmarks we ran: no result filtering has been done. Times are in microseconds with the time relative to the trapping INKS configuration shown in parentheses. In calls to the Unix server, the trapping INKS configuration improves Unix system call performance by a minimum of 20%, ranging up to a factor of two. "Nop (emulator)" and "getpid" are handled completely within the emulator and therefore should experience no performance improvement under trapping-INKS. The times on the two message configurations typically varied by 2% to 5%, while the trapping times varied by only 1% to 3%.

As discussed previously, certain aspects of Mach's implementation prevented us from achieving a large performance gain in calls from an in-kernel server to the microkernel. However, the fact that the server runs in the kernel address space still allowed some minor optimizations to be made along the trap path, resulting in slightly higher performance. Table 5 compares the times for a do-nothing server-to-kernel system call in the normal case (server running in user mode) and the INKS case (server running in kernel mode). Message syscalls account for most of the general-purpose calls in

Table 6: Macro benchmarks: Time in seconds (ratio to trapping-INKS)

Test	Configuration		
	Unoptimized	INKS Message	INKS Trap
Full kernel build	1682 (1.15)		1463
Andrew (filesystem phases)	48.8 (1.04)	49.5 (1.05)	47.0

Table 7: Communication Counts

	From User	
Emulation Traps	82474	34.9%
INKS Traps	48142	20.4%
Mach Syscalls (trap)	2748	1.2%
Mach Syscalls (msg)	16024	6.8%
Total From User	149388	63.2%
	From Server	
Mach Syscalls (trap)	41480	17.5%
Mach Syscalls (msg)	45654	19.3%
Total From Server	87134	36.8%

the Mach interface such as device I/O, while trapping syscalls are used for the very common cases like `vm_allocate`.

4.5 Macro Benchmark Results

We also ran somewhat more realistic Unix workloads, in particular the modified Andrew benchmark[11] and a Mach kernel build. The more realistic kernel build benchmark showed a 13% speedup under trapping INKS, while the Andrew benchmark yielded 4%. Table 6 shows these timing in seconds, with the ratio to trapping INKS in parentheses.

All runs were done with the system in single-user mode. “Andrew” includes only the four filesystem phases of the benchmark. For that test, after the initial run we found results only varied by at most a second. The kernel build compiles the Mach kernel from scratch, with all files local. Six runs of this test were made, with the last five showing run time variation of less than 1%.

On all system configurations, since the first run of any benchmark was significantly slower than succeeding ones, we factored out the first run of any macro benchmark. An anomaly we noted is that trapping INKS usually paid a higher startup penalty than either message case, typically twice as high. We have not yet investigated the cause of this effect, but speculate that once found, fixing it will yield further improvements in performance, for all runs.

We also ran the SPEC SDM SDET benchmark, and obtained about 8% throughput increase for the low-load “1 script” case. This gain abruptly decreased for higher loads, to less than 1%.

Table 7 shows the usage of the different communication mechanisms during one run of the Andrew benchmark. Table 8 breaks down by message type and source, the messages seen during the benchmark. The dominance of “simple” RPCs confirms our decision to concentrate on this type of RPC. However, the significant number of server-originated messages suggests that optimization of the server-kernel path would be worthwhile.

The data in these tables, together with our micro-benchmark numbers, can be used to validate the system time reduction obtained in larger benchmarks.

Table 8: Mach Message Type Breakdown

	From User	
Sync Simple	7810	12.7%
Sync Complex	8214	13.3%
Total From User	16024	26.0%
	From Server	
Sync Simple	22176	35.9%
Sync Complex	4666	7.6%
Async Send	6552	10.6%
Async Recv	12260	19.9%
Total From Server	45654	74.0%

4.6 Cache Issues

One can speculate that some of the performance improvement stems from better cache use. However, the expected cache effect is minor and would tend to worsen performance. (Note that the portion of the virtual address used to access the cache on the PA includes the address space identifier.) Moving the server and kernel into the same address space, thus making equal the space identifier portion of the cache index, should tend to worsen cache performance slightly. Linking the server, as we do, at an address that doesn't conflict with the kernel or user processes, should tend to improve cache performance over a server linked at the normal address. However, in both in- and out-of-kernel cases we use the same server, so this is not a factor. In general, we do not expect cache effects to be significant and *consistent* in this experiment. Other studies have shown that random changes can have significant cache effects[2].

5 Limitations

A serious limitation that arises from use of this framework is the loss of the protection that separate address spaces provides. Hardware protection is an important mechanism used to provide robustness in the face of unfriendly or malicious programs. Use can only be made of the merging of protection domains in situations where this loss of protection will not introduce new risks or inconveniences. For example, in the case of a Unix server which is a crucial and trusted operating system component, when loss of the server means (as it typically does) loss of all relevant system services, reduction in protection introduces no new limitations from a user's point of view. The difference between the Unix system dying on a fault in kernel space and dying on a fault in user space is moot. If, however, the service is part of a fault-tolerant system and is capable of recovery from faults, or if it is only one component in a system with other important, independent services, the loss of protection introduces real drawbacks to users and should be carefully weighed.

A limitation of the current implementation is that transferring port rights or out-of-line memory forces a normal message. This is not necessarily a drawback, for if the operation is rare enough, we increase our overall performance by simplifying the code for the common case. Most servers, including the Unix server, do not transfer port rights between themselves and clients or other servers. The vnode pager in the Unix server does issue substantial numbers of Mach kernel calls to do port manipulation, but short-circuiting the server-kernel path is a prerequisite to optimizing these calls. Further measurement of the argument type and message frequencies is needed to determine whether it is worthwhile to sort-circuit messages for these arguments.

6 Related Work

The most similar work to ours is that done in the Chorus[12, 6] microkernel/multi-server Unix system. Chorus "supervisor actors" correspond to our in-kernel servers, but there are significant differences. Chorus adopts the "migrating-thread" model, with a single kernel stack which remains with the thread. This is easy because continuations do not exist and existing servers do not make assumptions about the intimate details of their stacks (they use native kernel threads). All potential optimizations are performed: user-server, user-kernel, server-server, and server-kernel interactions. The linkage is typically through traps, although procedure call linkage is also used. Supervisor actors are linked separately from the kernel and each other, as we do. In principle, supervisor actors can be loaded and unloaded dynamically, although some violate the rules for this. The client interface to the local Unix "process manager" is *always* through traps; local messages "cannot happen." Thus there is no check for a non-0 length message queue. Asynchronous messages are not optimized. Chorus's lack of an RPC stub generator contributes to a relative lack of flexibility and transparency compared to our scheme. The name space (service id's) of multiple servers cannot overlap in Chorus.

7 Future Work

There are a number of ways to continue this work. A useful short term effort is to do the same experiment on another hardware architecture, in order to help factor out machine-dependent effects. We feel that the most important task is to concentrate on *measurements*, so informed decisions can be made between the various implementation strategies. Further investigation of the "ramp up" and scaling problems will likely reveal bottlenecks. Completing the support for server-server interactions would allow us quantify the improvement in a multi-server environment where the speedup should be substantially greater, due to the increased numbers of inter-domain transfers[7].

Optimizing server-kernel interactions should be examined, as it could potentially result in up to two orders of magnitude reduction in overhead (message vs. procedure call). If this optimization were done, note that it generalizes the existing mechanism supporting alternate system call paths to a few Mach kernel calls. This would remove it from the realm of special-purpose "hacks" and make it a universally applied optimization.

In the longer view, we plan to use the powerful linking capabilities of the OMOS meta-object/object server[9, 10] to bind more flexibly and dynamically. OMOS can hook pre-existing programs and modules together in a structured and programmable way. We will move toward making OMOS capable of dynamically binding arbitrary modules with arbitrary interfaces (C-style function calls, C++ method invocations) communicating across arbitrary channels. In this role OMOS is analogous to the nameserver and binding server in a traditional RPC system, but is capable of much tighter and more flexible binding.

8 Conclusion

The client-server Mach IPC path can be heavily and transparently optimized in the common case of a trusted Unix server, with little impact on server code. Major improvements in IPC and Unix system call performance result, with significant improvements on Unix benchmarks. The Mach thread abstraction has the potential to evolve to a migrating-threads model.

Acknowledgements

We are grateful to Douglas Orr for enlightening discussion and review of earlier drafts, Allan Bricker for patient explanation of the Chorus mechanism, and Michael Condict for helpful discussion of the OSF/1 server and INKS issues. We also thank Robert Kessler for reviewing earlier drafts, Leigh Stoller for useful suggestions, Peter Hoogenboom for extensive time in table formatting, and

John Bonn for help running benchmarks. In addition, we thank Hewlett-Packard for register CR16 which makes precise low-overhead timing easy.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, GA, June 9–13, 1986. Usenix Association.

[2] B. Bershad, R. Draves, and A. Forin. Using microbenchmarks to evaluate system performance. Technical report, Carnegie Mellon University, November 1991.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[4] Bryan Ford, Mike Hibler, and Jay Lepreau. Extending the Mach 3.0 thread model. Technical Report UUCS-93-012, University of Utah Computer Science Department, April 1993.

[5] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–96, Anaheim, California, June 11–15, 1990. Usenix Association.

[6] M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A second-generation micro-kernel based unix; lessons in performance and compatibility. In *Proceedings of the Winter 1991 USENIX Conference*, pages 13–22, Dallas, TX, Winter 1991.

[7] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized emulation services for Mach 3.0 — overview, experiences and current status. In *Proc. of the Second Usenix Mach Symposium*, pages 13–26, 1991.

[8] Open Systems Foundation and Carnegie Mellon University., Cambridge MA. *MACH 3 Kernel Interface*, 1992.

[9] Douglas B. Orr and Robert W. Mecklenburg. OMOS — an object server for program execution. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 200–209, Paris, France, September 1992. IEEE Computer Society.

[10] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993.

[11] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Conference Proceedings*, pages 247–256, Anaheim, CA, Summer 1990. Usenix.

[12] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–367, 1988.

[13] Bob Wheeler and Brian N. Bershad. Consistency management for virtually indexed caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 124–136, October 1992.