MICROKERNEL MODULARITY WITH INTEGRATED KERNEL PERFORMANCE

Improving the Performance of MK OSF/1 Via Collocation, RPC Short-Circuiting and Other Optimizations[1]

Michael Condict, Don Bolinger, Eamonn McManus, Dave Mitchell, Steve Lewontin
OSF Research Institute

Abstract

Microkernel modularity has typically had a significant performance cost. The Microkernel Performance Project has been pursuing a variety of performance optimizations in an effort to achieve integrated kernel-level performance with a microkernel design. Although this project is still underway, we have already exceeded our goal of 95% of integrated kernel performance. Much of our work is based on collocation of programs in the same address space. A major benefit of collocation is the ability to short-circuit RPC between collocated clients and servers. The project's largest single performance gain comes from short-circuiting the system call path when the OS server is collocated in the kernel address space. Kernel collocation of the OS server has also enabled other important performance improvements, including optimized copy-in/copy-out. RPC short-circuiting is based on a new model that raises RPC to the status of first class object in the Mach kernel. A key element in the new RPC model is a migrating threads mechanism that avoids scheduler and context-switching overhead, providing a more efficient transition from client to server. Based on the new RPC model, we are developing a general Mach RPC service to enable efficient, non-message-based RPC between clients and servers, whether or not they are collocated. In addition to collocation and RPC short-circuiting, we have made a number of other significant improvements, including C-threads optimizations, elimination of MP lock overhead on uniprocessors, FS cache optimization, and a variety of VM optimizations. These miscellaneous improvements have contributed significantly to the overall performance improvement.

1 Introduction

Microkernel (MK) architecture brings major advantages in terms of operating system modularity. For example, the MK design allows us to construct a kernel that can support multiple operating system personalities while making OS enhancement and maintenance far simpler. Still, MK designs have always exacted a performance penalty when compared with integrated kernels (IKs), and we have long recognized that wider acceptance of MK systems will depend on achieving performance comparable to that of IK systems. The principal goal of the Microkernel Performance Project is to demonstrate IK-like performance in an MK system.

The project brings together a variety of performance optimizations, and the results so far have been extremely positive. Our original goal was to achieve approximately 95% of the performance of an OSF/1 integrated kernel. Although the MK Performance Project is still underway, we have already successfully met this goal.

[1]This research was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Material Command (AFMC).

April 13, 1994 5:38 pm

Most of the performance penalties associated with MK systems result from locating the OS server in user space and using a single, general-purpose IPC message mechanism to implement RPC. This adds much overhead to the system call path and makes OS access to both kernel and user memory more costly. Therefore, our principal strategy for improving MK performance has been to move the OS server task back into the kernel address space and to optimize the RPC mechanism for the case where the client and server share an address space.

Much of this work is based on the *collocation* design originally reported in Condict[1]. Collocation means locating separately-compiled and linked programs in the same address space[2]. When programs share the same address space, it becomes possible to replace IPC message-based RPC between them with a form of direct procedure call that we call *short-circuited RPC*. Short-circuited RPC is many times more efficient than message passing, and provides the basis for the largest single performance improvement we made, the short-circuited system call path.

One of the requirements for this project was to maintain the modularity of the MK design. In the case of RPC, this means that collocated programs must not be aware that they are communicating via short-circuited rather than message-based RPC. Message-based RPC provides a clear boundary between communicating programs. For example, a server need not be concerned that the thread handling an RPC will be aborted by the client; under IPC message-based RPC, the effects of **`thread_abort`** are confined to the caller. RPC short-circuiting needs a way to maintain these boundaries so that programs can be safely collocated without having to change their assumptions about the semantics of operations like **`thread_abort`**. We have adopted a version of the Utah *migrating threads* mechanism [2] for this purpose.

In addition to shortening the RPC call path, collocation also offers the obvious possibility of passing data more efficiently. Direct RPC parameters can be passed on the stack, and in some cases it may be possible to avoid VM copy operations by passing pointers to out-of-line data.

While collocation is a general mechanism, applicable to programs in any address space, collocation of programs in the kernel address space has special importance to performance optimization and has so far been the principal focus of our development work. In the first place, by moving the OSF1 server into the kernel address space, we can apply RPC short-circuiting to the system call path. Second, we can reduce the overhead of copy-in/copy-out operations between the OS server (now in the kernel) and the user-space application. Optimized copy-in and copy-out was a major source of performance improvement in the enhanced MK operating system.

This paper focuses on these and other collocation-related performance optimizations. However, a number of other optimizations, not related to collocation, have also been incorpo-

[2]In the current implementation of kernel collocation, each collocated program runs in a separate task. This corresponds to our original model of collocating several tasks in the same address space. However, our model of collocation is evolving in the direction of having several separately compiled and linked programs loaded into a *single task*. Whenever possible in this paper we speak of collocating *programs in an address space* because this terminology covers both cases. However, when we describe aspects of the current implementation that apply only to collocated tasks, we use the term *task* rather than *program*.

rated into our work, and these are reported as well. Section 2 of this paper gives a quick overview of the performance overhead associated with traditional MK designs. Sections 3-6 describe collocation, migrating threads, short-circuited RPC, and copy-in/copy-out optimizations in more detail. Section 7 summarizes other performance improvements. Section 8 describes a set of performance measurements that demonstrate the benefits of our work. Finally, section 9 discusses our plans for further work.

2 MK Performance Costs

We analyzed the relative performance of comparable MK and IK versions of the OSF/1 operating system in order to identify the places where performance optimization would have the highest payoff. When measured with 8-user AIM III[3] runs, the OSF/1 1.3 MK operating system exacts a performance penalty of about 48% (46 seconds) compared with the OSF/1 IK release 1.3 (see Figure 1). Of that difference, nearly half (22 seconds) can be attributed to the inefficient IPC message-based system call path. We also identified a number of smaller, but significant costs, including:

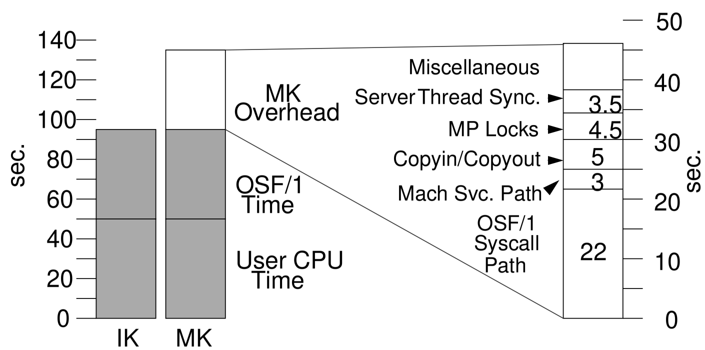

Figure 1. Overhead of OSF/1 MK 1.3 Compared With OSF/1 1.3 IK Running AIM III on Intel 486DX50

- Inefficient copy-in/copy-out when the OS server is not located in the kernel address space
- MP locking that is unnecessarily fine-grained for non-MP systems
- Inefficient, message-based thread synchronization
- Expensive, message-based OS server requests for kernel services

[3] See Section 8 for a description of the AIM III tests.

April 13, 1994 5:38 pm

Given the distribution of performance costs, it is obvious that optimizing the system call path has a high payoff. Furthermore, a substantial proportion of the remaining overhead time is due, at least in part, to the location of the OS server in user address space. In particular, both the copy-in/copy-out path and OS server requests for kernel services—which together account for 8 seconds—are much less efficient because the OS server is in user space. Also, while our thread synchronization optimizations are not based on collocation, it turns out that they can be made even more efficient with a collocated OS server. In sum, kernel collocation of the OS server has the potential to affect more than 30 of the 46 seconds of measured overhead. Therefore, collocation is an essential enabling technology, underlying much of our work.

It might seem that *all* the MK overhead should be in some way due to the location of the OS server in user space, since this is the chief distinguishing feature of the MK design[4]. This is particularly costly in Mach because all service requests are based on a heavy-weight general-purpose IPC mechanism. However, we recognize that the principal benefit of MK design is modularity and that deploying the OS server as a user-space task is simply a means to achieve this. Some of the overhead of an MK system is therefore due to modularity rather than the location of the OS *per se*, and we expect some of this modularity-related overhead to persist in any MK design.

3 Collocation

Collocation is the enabling technology underlying many of the performance optimizations we have made. In general, collocation means loading more than one program into the same address space. In our current implementation of kernel collocation, each kernel-loaded program runs in a separate task. However, our model of collocation has evolved to include collocation of separately-compiled and linked programs in a single task, and we expect that future implementations will use this model.[5]

Our design contemplates collocation in both user and kernel address spaces. However, up to this point our development work has concentrated on programs collocated in the kernel address space, because this is where we gain the most in performance with such optimizations as short-circuiting the system call path, and faster copy-in/copy-out. Therefore, the discussion in this section concentrates mainly on the specifics of kernel collocation.

Our objective for kernel collocation is to enable an ordinary user program to run unchanged in the kernel address space. Such a program can check to see if it is running kernel-loaded in order to enable specific performance optimizations, but should not need to do so in order to run normally.

As we noted above, in the current implementation each kernel-loaded program runs in a separate task. A kernel-loaded task is created by a new **`kernel_task_create`** service as an ordinary task with the following special characteristics:

- Its address space is allocated out of a submap of the kernel map.

[4] More generally, the MK overhead is due to the decomposition of traditional OS functions into separate tasks running in different address spaces. Here, however, we are particularly concerned with the division of labor between the OS server and the kernel.

[5] See the footnote on terminology on page 1.

- It runs at kernel privilege and has access to the whole kernel address space.
- It is marked as kernel-loaded (by setting a new **`kernel_loaded`** flag).
- All kernel-loaded tasks share the same port space, in which port names are pointers to port data structures. This imposes various restrictions on port manipulation by collocated servers, as noted below.

Some low-level, machine-dependent changes are required in the kernel to support collocation of tasks. For example, kernel loading of servers forces us to make a distinction between running in the kernel task and running at kernel privilege. Threads in a user-space task execute at user privilege level and must switch privilege level when they enter the kernel; hence running at kernel privilege is synonymous with running in the kernel task. However, when a server task is collocated in the kernel, the server threads execute at kernel privilege level even when they are in the server task. This can require changes in low-level kernel code. For example, some hardware architectures (notably 80x86) depend on the change of privilege level for correct handling of exception stacks. In such cases, it is necessary to modify the relevant machine-dependent code so that threads from kernel-loaded tasks function as if they were user threads.

One task (currently the OSF/1 server) is loaded into the kernel at boot time. We have also provided a new **`kernel_execve`** system call so that other user tasks can be collocated on the fly. Currently, kernel loaded-tasks must be statically linked at a fixed address, but we will also enable dynamic relocation in the future.

In addition to providing the new **`kernel_execve`** system call, the OSF/1 server has itself been modified to take explicit advantage of collocation in certain cases. For example the copy-in and copy-out paths explicitly check for kernel-loading to take advantage of optimizations available when the OS server is in the kernel address space. (See Section 6 for more details.)

While it is our goal to be able to collocate ordinary programs without modification, some minor constraints apply to kernel-loaded servers. The most important issue was raised in the previous section: all kernel-loaded tasks share the same port name space. This means that servers must not rename or deallocate ports. Also, once servers have been collocated they cannot be separately unloaded, because doing so might invalidate data, such as port right counts, shared with other collocated servers. The are a few other restrictions. Servers must allocate memory out of their submaps and they must not perform operations that could interfere with the Mach kernel's use of its address space, such as VM-protecting page zero. Finally, any server loaded into the kernel address space and operating a kernel privilege must be trusted by the Mach kernel[6].

It is also important to point out that for a collocated server to receive short-circuited RPCs, it must do additional setup. This is not a requirement of collocation per se, but of the short-circuiting mechanism, so this issue is left for Section 5.

[6]An additional requirement on architectures (such as the 80x86) where a stack is needed to handle exceptions, is that thread stacks must be wired.

4 Migrating Threads

Collocation allows for major performance gains because it permits many message-based operations to be converted into direct procedure calls. However, the modularity requirements of an MK system do not permit us to assume that an arbitrary server can be freely entered by a client thread. The **`thread_abort`** problem described above is a good example of this. In general, any RPC short-circuiting mechanism must be able to provide the same degree of server isolation as message-based RPC. Since we currently implement collocation by having each collocated program run in a separate task, we need to provide a point of control as threads cross task boundaries. We have adopted a version of the Utah migrating threads mechanism [2] for this purpose.[7]

The migrating threads mechanisms falls somewhere between two extremes. At one pole, a thread of a client collocated in the same address space could simply enter the server via a direct procedure call. No server resources would consumed on the client's behalf; the server would simply be a passive repository for code executed by the client. Such a thread could be considered to have "fully entered" the server since all of the client's resources would still be engaged in its execution.

The other extreme is typified by message-passing RPC, in which the recipient task allocates a call thread to handle the call and passes the results back as a message. In the message-passing model, there are two separate threads of control, the active thread in the server, and the (suspended) thread in the client which initiated the RPC and which will receive the results (although the RPC model attempts to attach single thread semantics to the relationship between client and server).

In the migrating threads model, there is only one flow of control over the RPC, but the client thread does not fully enter the server. We have implemented this by splitting each thread into two components: the *activation* and the *shuttle*. An activation holds the per-task runtime state of the thread, including machine registers, stacks, and a control port. A shuttle holds the scheduling and resource accounting elements of a thread. Figure 2 shows the major components of each structure.

The shuttle is the portion of the thread that migrates from one task to another. As the shuttle migrates, activations are pushed onto and popped off of an activation stack associated with the shuttle. (See Figure 3.) The activation stack is maintained as a linked list using the higher and lower activation pointers shown in Figure 2.

When a thread migrates to a new task, an activation associated with that task is pushed onto the thread's activation stack. When a thread migrates back to a previous task, the current (topmost) activation is popped from its activation stack. The thread can therefore be thought of as a single logical flow of control operating on a stack of activations. As the shuttle migrates, it carries with it the scheduling and resource accounting elements of the thread, while the activations maintain the thread state specific to each task.

[7]Again, it is important to note that our model is evolving from collocation of tasks in an address space towards collocation of programs within a task. In the new model, a thread will no longer cross a task boundary, and a different thread migration mechanism will be required. However, the migrating threads mechanism described here will continue to be used for RPCs among non-collocated tasks under the Mach RPC service that we are developing. (See Section 9.2.)

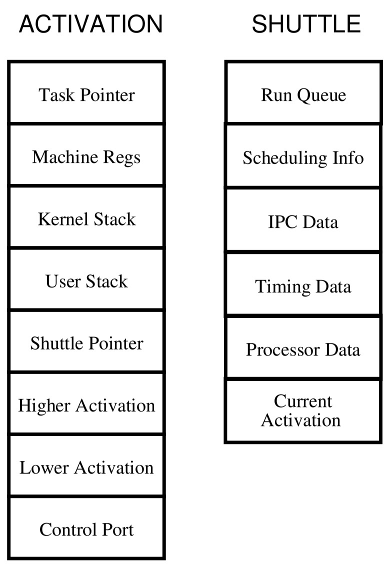

Figure 2. Activation and Shuttle Components

In our current design, each task maintains a pool of “empty” activations to which a shuttle can be attached as it migrates in. When the thread migrates back out of a task, the popped activation is returned to its pool. Besides pushing and popping activations, thread migration also involves switching stacks and whatever other saving and restoration of state is appropriate for the transition from task to task. An outline of the full procedure is:

1. Remove an activation from the pool in the server task.
2. Push the activation onto the activation stack of the incoming shuttle.
3. Transfer the caller’s arguments to the stack of the newly-pushed activation.
4. Switch stacks, call the work function, and switch back.
5. Pop the activation, return it to the server pool, and return to the caller.

Splitting the thread into a shuttle and activations gives us an opportunity to establish a point of control at the task boundary. Thread operations may be directed at the activation of a specific task and need not affect the thread in another task. Control can be asserted in the migration procedure. For example, a possible implementation of **`thread_abort`** that

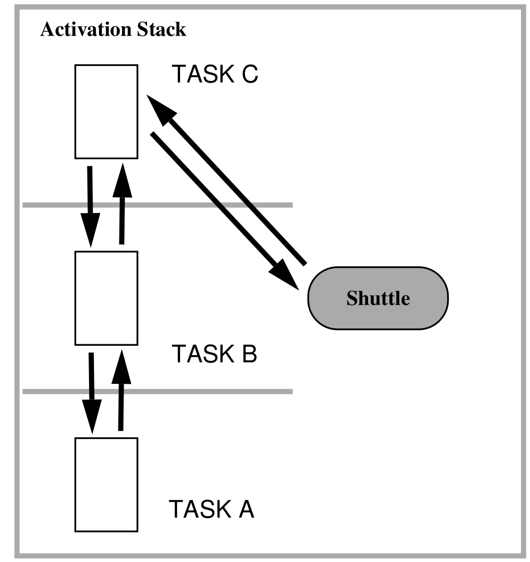

Figure 3. The Activation Stack

maintains the current message-based RPC semantics would be to split the thread in two at the task boundary by cloning the shuttle. One shuttle would return immediately to the aborting task, and the other could run to completion. (In fact, we have not implemented **`thread_abort`** this way because we believe these semantics are undesirable.)

Thread migration via this mechanism avoids involving the scheduler. The thread continues to run with the same scheduling attributes in each task, and the migration is a relatively lightweight procedure compared with a full context switch. Nevertheless, migration can be made even lighter-weight in some cases. When calling across a boundary that does not require a point of control, we can avoid the full thread migration algorithm. This is most obviously the case with the Mach kernel itself, which is already designed to permit unrestricted migration of user-space threads (via traps). Therefore, in designing the RPC short-circuiting mechanism, we have optimized to avoid the full migration algorithm—including the activation switch—where it is not required, such as when a collocated server makes a request to the kernel for Mach services.

5 RPC Short-Circuiting

We have developed a variety of means to shorten the RPC path from the traditional Mach IPC message-based mechanism. An optimized RPC mechanisml needs to address two broad cases: where the client and server share an address space and where they do not[8]. The traditional IPC message-based mechanism provides a general solution that is not well optimized for either case. We have been working to provide optimizations for both cases.

Our current development work has focussed on the collocation case. We have developed an RPC short-circuiting mechanism, based on collocation and migrating threads, that radically speeds up RPC performance. This is described in Section 5.1. One major benefit of this RPC short-circuiting mechanism is that it can be used to optimize the system call path for a kernel-collocated OS server. In a microkernel OS with user-space OS services, the system call path always involves IPC messages.[9] By moving the OS server into the kernel, system calls can instead use short-circuited RPCs. System call short-circuiting is described in Section 5.2

Because of the high payoff from short-circuiting the system call path, we have focussed our work on optimizing the RPC path between kernel-collocated programs. However, it is important to note that the RPC short-circuiting mechanism is also designed to work for programs collocated in user address space. While we have not yet done any development in this area, we plan to address this issue in the next stage of the MK Performance Project.

In addition to optimizing the RPC call path itself, we have also given attention to the problem of passing indirect parameters more efficiently. When the client and server are collocated, the problem can often be reduced to passing pointers to the data, although some extensions of MIG are required to maintain pass-by-value semantics. This issue is discussed in Section 5.3.

Up to this point we have not done any development for the case of optimized RPC between programs that are not collocated. Here, the current model is to fall back on traditional IPC message-based RPC. However, we are developing a design for a general Mach RPC service that any client can use to invoke an efficient, non-message-based RPC to any server. The Mach RPC Service is briefly described in Section 9.

5.1 RPC Short-Circuiting Design

The RPC short-circuiting design raises RPC to the status of a first class object in Mach. In the traditional Mach design, RPC was unknown to the kernel and the objects—ports, messages, tasks, and threads—managed by the kernel. RPC semantics were imposed on IPC messages by MIG-generated stubs. In our new design, support for the RPC short-circuit path is explicitly installed in task-level data structures, known to the kernel.

While the new RPC design required changes to the kernel and to kernel-managed objects, it was also subject to the constraint that the changes remain transparent to RPC clients and have minimal impact on servers. Short-circuited RPC had to have exactly the same semantics as IPC message-based RPC. Generic clients could not be required to change their behavior or in any way be required to have knowledge that the RPC path was different. This

[8] This assumes that the client and server are on the same node. A third case is that the client and server are on different nodes, and a network RPC must be used.

[9] The paths are somewhat different for emulator and non-emulator based servers. Our current work was focussed mainly the non-emulator model, although the architecture also contemplates improvements in the emulator model.

meant that changes in the application interface to RPC had to be confined to the MIG-generated stubs as much as possible.

Some changes in server initialization are inevitable. Servers must now establish a pool of empty activations and set up new RPC data structures (described below) in order to receive RPCs. This requirement does not really violate our intention that the new RPC mechanism be as transparent as possible, since servers are already required to do setup specifically to receive IPC message-based RPCs. We have simply added to the setup required in order to take advantage of the new RPC path.

The general short-circuiting design is to have the MIG user stub act as a switch among the possible RPC paths. (Since the kernel is being changed anyway, short-circuited RPC can also be in-lined into specific kernel calls.) What is required to make short-circuiting work between collocated programs is:

1. To make the server entry points available to callers which are collocated, but which are separately linked and do not share symbol tables.

2. To provide a general mechanism for safely migrating a thread of control from the caller to the callee.

For collocated tasks, requirement 2 is met by the Utah migrating threads mechanism. To implement this, all tasks collocated in an address space are given access to a common routine that can migrate threads from task to task. A pointer to this routine is installed in a “glue vector” that contains pointers to any address space-specific routines related to RPC short circuiting. The glue vector is made accessible via a well-known pointer in each task. To short-circuit a call, the client stub calls the appropriate thread-migration glue routine with arguments that determine the server entry point.

Requirement 1 is met, compatibly with the message passing RPC mechanism, by having the target port name point (somewhat indirectly) to the server entry point for the routine being called. In the kernel, where ports are always pointers to data, this is easily arranged. In user-space collocation, ports will be renamed to point to data structures in the user-space target task. The mapping of IPC ports to server entry points is implemented at the task level with the data structures shown in Figure 4. Each port name is a pointer to an **`rpc_port`** structure. (In the kernel it is actually a pointer to a kernel **`ipc_port`** structure, but the only part we let the stub see is the first few fields, which correspond to the user-visible **`rpc_port`** structure that will be used in user-space short-circuiting.) The **`rpc_port`** in turn contains a pointer to an **`rpc_subsystem`** structure that holds a jump table of entry points for RPC-able server functions. The **`rpc_subsystem`** structure is generated by MIG for each subsystem. It describes a set of server routines that can be invoked by RPCs on the ports registered with the subsystem. The jump table includes the entry points and argument descriptors for all of the subsystem functions.

Port names that are pointers to port structures have one of the lower order bits unset. In the case of a send/receive right, both lower order bits are 0. This allows a simple stub test of a port to see whether it is a legitimate pointer to data in the local (kernel or user) address space or simply the local port name for some foreign port. If the port is a pointer, the stub

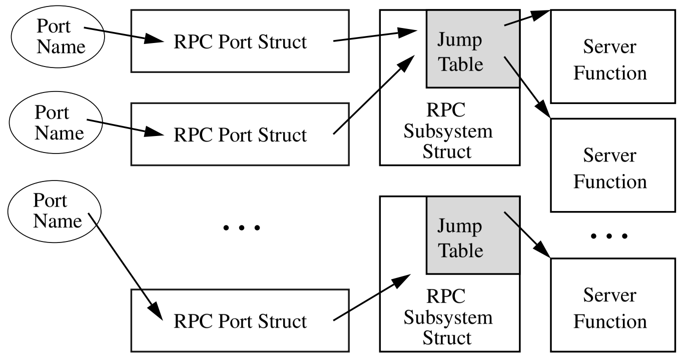

Figure 4. RPC Data Structures

selects the short-circuited RPC path. Otherwise, the stub uses a traditional IPC message-based RPC.

To follow the short-circuited path (shown schematically in Figure 5), the stub calls the migration glue routine with the port and operation number[10]. Once the migration routine has performed the actual thread migration steps, it calls the server operation by dereferencing the port name and indexing into the entry point vector with the entry point number.

Collocated programs use ports to derive short-circuited RPC entry points, but the stubs make this use transparent to the application code. The client application simply makes an RPC on a port as usual. The server arranges for the port to point to the appropriate data structures when it registers the subsystem. The client also requires access to the appropriate glue routines. This is accomplished by setting a new **`user_data`** member in each program to hold a pointer to the common glue vector. The first program initialized in an address space sets **`user_data`**. Subsequent programs are loaded with **`user_data`** initialized to the same value.

The client and server application code are unaware of the RPC path. In the short-circuited case, the client MIG stub makes the decision to short-circuit the call, and the short-circuit path is handled entirely in target task glue code and target task RPC data structures. Server stub unmarshalling is bypassed, and the call is made directly to the server work routine. When the IPC message-based RPC path is followed, the work routine is instead called from the server stub, but the server application code is unaware of this difference.

As we noted previously, when calling into the Mach kernel, the full thread migration algorithm is overkill. Because the kernel is a known entity, designed to be entered by and passively executed by foreign threads, the protection at the task boundary provided by this

[10]Here we ignore argument processing for the moment. This topic is taken up in Section 5.4.

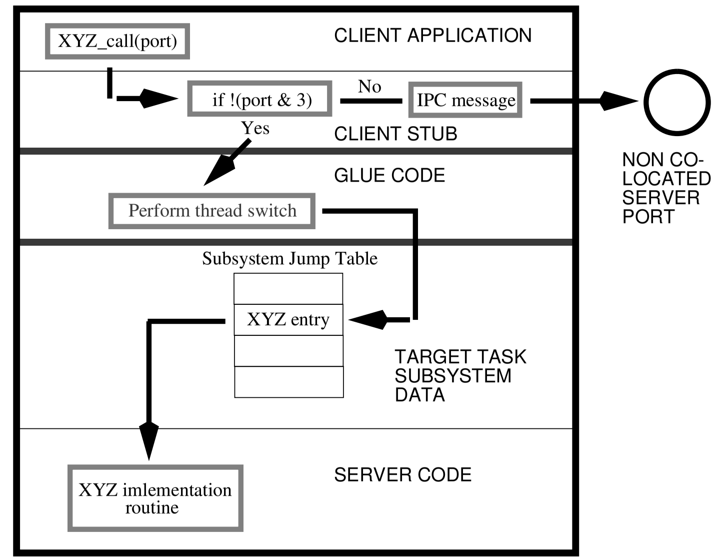

Figure 5. Short-Circuited RPC Path

mechanism is not needed. Therefore, the short-circuited path is tuned to avoid pushing and popping an activation in this case.

5.2 The Short-Circuited System Call Path

The objective in short-circuiting the system call path is to obtain performance comparable to that of an integrated kernel. This means avoiding messages and the traps and context switches they require. In our design (see Figure 6), the path starts with a trap into kernel mode(1). The same user thread that requests the service executes the OS server code with no context switch, via short-circuited RPC (2,4). The OS server itself can obtain any necessary Mach services by direct procedure call rather than by IPC messages(3) [11]. Finally, the OS server returns to the user task by a simple return from interrupt(5). No messages are required and, for a non-blocking call, no context switches are required either.

The short-circuited path can be compared to the system call path with a user-space OS server (Figure 7). With a user-space server, the application again begins the call with a trap

[11]In fact, we have not yet implemented such short-circuited access to kernel services in general. A few kernel services—copy-in and copy-out, and the new thread synchronization primitives—are implemented via short-circuitable calls that the OS server can call without using MIG stubs. But the majority of kernel services requests are now being optimized by replacing IPC messages with traps. See Section 7.2 for further discussion of this issue.

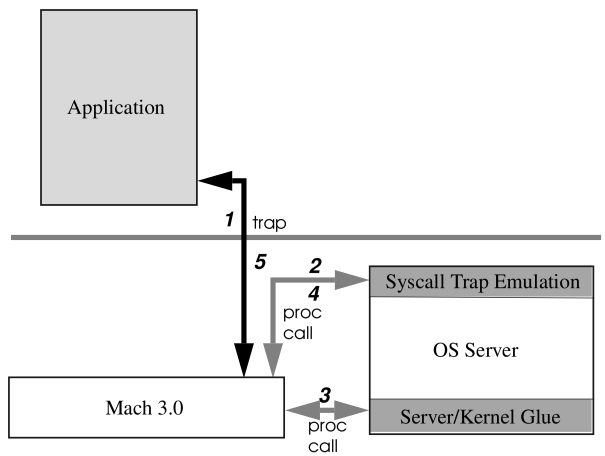

Figure 6: System call path with a kernel-loaded OS server

into the kernel (1). The kernel then context switches to a thread of the user-mode server (2), which it has awakened to receive the system call message. If the server uses kernel services via RPC(3), these are also IPC message-based with the consequent trap and context-switching overhead. Finally, the server sends a reply message to end the system call, causing another trap into the kernel (4) and a context switch back to the user thread that initiated the system call (5). The grand total is at least two user-user messages, which imply two user-user context switches, not counting any kernel services the OS server may request.

The short-circuited design provides performance much more like that of an integrated kernel (Figure 8). In the IK design, the path again begins with a user trap into the kernel (1), which determines the correct OS function based on the trap argument. The thread that trapped is transferred (in kernel mode) to the OS function (2), where it carries out the requested service, possibly requiring one or more procedure calls to the Mach part of the kernel (3). When finished, it returns to the top-level trap interface of the kernel (4), which returns to user mode (5), allowing the user to continue. The only important performance differences between the kernel-loaded server and the integrated kernel are:

- The procedure calls in the integrated kernel are more efficient, because they avoid the call through glue code, the thread migration algorithm, and some of the indirection.
- The OS code in the integrated kernel can directly access the kernel data structures, whereas the collocated OS server must go through the documented kernel interfaces. But the prevention of this violation of modularity is the whole point of a microkernel-based OS.

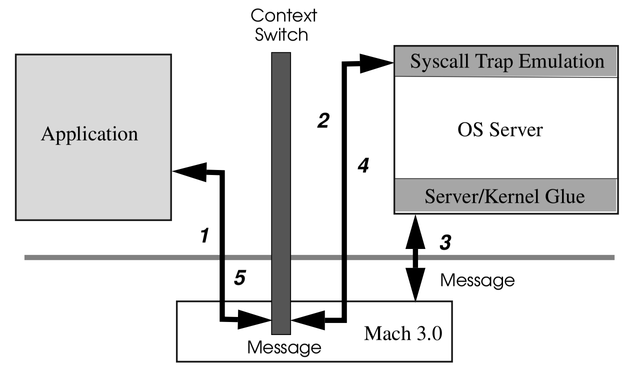

Figure 7: System call path with a user-mode OS server

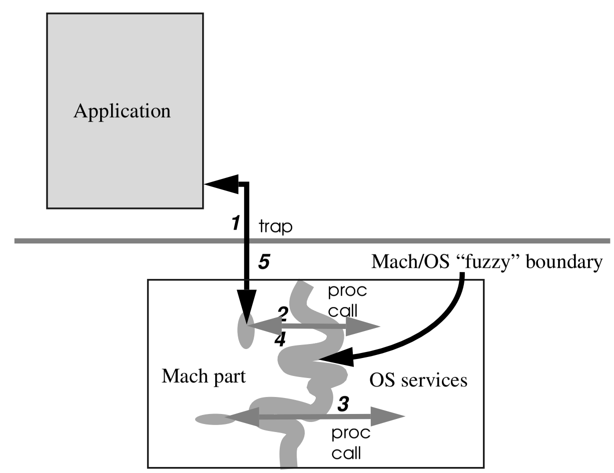

Figure 8: System call path in an integrated kernel

This design preserves the modularity at a small cost in efficiency, unlike the user-mode OS server, which enforces the modularity at a much higher cost.

5.3 Parameter Passing

In IPC message-based RPC, data need to be copied, although the kernel makes this as efficient as possible by manipulating VM mappings. With short-circuited RPC, several optimizations are possible. In general, direct parameters can be passed on the stack, just as they are in a local procedure call. Indirect data (arrays pointed to by direct parameters) may be copied or, in some cases, a pointer may simply be passed from client to server.

Generic client argument processing is as follows:

- Before entering the server, handle input arguments:
  - **vm_allocate** mapped copies of indirect data
  - allocate new port rights for input port arguments
- Migrate to server work function.
- On return, handle output arguments:
  - **vm_allocate** mapped copies of indirect data that were not deallocated by the server
  - allocate port right copies for output port arguments that were not deallocated by the server

Since the server stub is bypassed in short-circuited RPC, the client stub does all the argument passing work. However, depending on the MIG declaration, the server may or may not deallocate some arrays before returning to the client. When the server does not deallocate an array, the client stub must be responsible for this.

The VM copy operations may be avoided by passing pointers to indirect data if the server agrees not to violate pass-by-value semantics; that is, the server must not modify the data or access them after the call returns. Two new MIG argument keywords are available to tell the stubs that a pointer may be safely passed. The keyword **const** signifies that the server agrees not modify the referenced data. The keyword **auto** signifies that the server agrees not to reference the data after the call returns. When the two keywords are used together, the stub passes a pointer instead of copying the indirect argument. Since the call bypasses the server stub, it automatically bypasses the server stub code that would normally deallocate the memory passed out of line[12].

The case of a system call along the short-circuited RPC exception path is special. Rather than passing the system call arguments as explicit parameters to the exception RPC, the

[12]It would also be possible to extend MIG to permit pass-by-reference semantics for short-circuited RPC. However, IPC message-based RPC does not implement pass by reference semantics, so we have not done this.

kernel simply copies-in to the saved thread state a block of the user stack sufficient to hold any possible system call arguments. The thread state is then passed to the exception RPC as a parameter. This adds a small amount of unnecessary overhead to some calls, since the kernel always copies-in the same-sized block of the stack, even when there are no system call arguments. However, having the kernel copy-in the arguments is more efficient than having the server do so, and the performance gain is much more significant than the small extra overhead for those few system calls that have no arguments.

6 Copy-in/Copy-out Optimization

Optimized copy-in/copy-out is a major benefit of collocation not specifically dependent on RPC short-circuiting. The basic idea is that, when a server that is collocated in the kernel performs copy-in or copy-out, we can avoid the VM manipulations normally required to map the server's address space into the kernel. Without collocation, the kernel has to map both the server and the target address spaces in sequence to perform a copy operation, and must map or copy the target data into its own address space in order to transfer it. With the server in the kernel address space, only the target address space needs to be mapped in, and the data can be copied directly between the server and target.

A kernel-collocated OS server also has direct, short-circuited access to the kernel copy-in and copy-out primitives, thus making these operations even more efficient. These are currently implemented using specialized short-circuited kernel copy-in (**klcopyin**) and copy-out (**klcopyout**) routines that attempt to find the target task map on the activation stack and perform the copy directly. However, to generalize the implementation, we will replace these with a short-circuited path in the **vm_read_overwrite** and **vm_write** system calls.

A related optimization is also possible for device access. Device access typically requires kernel address space buffers that can be wired for the duration of the IO. When the OS server is in user space, the target memory needs to be remapped into kernel memory. By placing the OS server in the kernel, the device can access the server's virtual memory directly, without remapping. This optimization is applied to **write** and **read_overwrite** device accesses.[13]

7 Other Optimizations

In addition to the collocation-related optimizations described in Sections 5 and 6, the MK Performance Project also added a number of other performance enhancements. As we will see when we examine the performance results (in Section 8), these currently add up to a little less than half of the total savings achieved by the project, although we expect this proportion to fall somewhat as we complete the collocation-related work. This section briefly describes the major miscellaneous optimizations.

[13]Note that we actually implemented a more general, non-collocation-related optimization of **device_read_overwrite.** In the past, this allocated kernel memory, read data into it, and then swapped the kernel memory with the target memory. The optimized version remaps the target into the kernel and reads directly into it. In the collocated case, this is further optimized because the remapping is not necessary.

7.1 Cthreads Optimizations

Cthreads optimizations constituted a substantial sub-project within the MK Performance Project. This work is fully described in [3]. The most important change is optimization of the thread synchronization mechanisms. The basic elements are:

- Continuations were implemented for transfers of control among Cthreads. Continuations reduce the amount of state that needs to be saved and restored at context switch time, thereby reducing the size of critical sections.

- The multiple spin locks protecting the run queue and the internal queues of each mutex and condition variable were replaced with a single master lock. This further simplifies and shortens the critical section of the context switch path.

- The use of continuations and simpler locking makes possible much more efficient thread synchronization. Among other optimizations, IPC messaging was removed from the synchronization primitives and, for multiprocessors, various types of spinning were added to the blocking process. These optimizations account for the bulk of the Cthread-related performance gains.

While these optimizations are not collocation-related, a further gain in efficiency is realized with collocation because the OS server has direct, short-circuited access to the kernel synchronization primitives. Two synchronization operations **(klthread_switch** and **klthread_depress_abort**) were made directly available to the OS server for short-circuited access without using MIG-generated stubs.[14]

7.2 Traps for Common Kernel Services

We have added traps for many common Mach services that were previously accessed via Mach messages. At first sight, this optimization appears to be out of keeping with our RPC short-circuiting design. RPC short-circuiting can potentially make OS server requests for kernel services highly efficient. However, we are currently using this technique only for a limited number of kernel services—copy-in/copy-out and the thread synchronization primitives—and we do not provide this short-circuiting via the MIG stubs and the full short-circuit algorithm described in Section 5. Instead, these interfaces have been made available by installing them in the kernel glue vector.

There are two reasons for the currently limited, non-MIG-based, use of short-circuited RPC for OS server requests for Mach services. First, some services, such as **thread_self** were never Mach message-based because they do not have a target port. While these services can be short-circuited, they cannot use the port-based RPC algorithm described in Section 5. Second, even for those services that can be short-circuited through MIG stubs, we have not yet generalized this mechanism due to some added complexity required in the MIG stubs for calls into the kernel. The kernel uses a MIG feature that allows a server to specify translation functions for RPC arguments, typically for the purpose of mapping ports

[14] A number of short-circuited kernel routines were made available to the OS server by installing them in the kernel glue vector, where they can be directly accessed by the OS server without going through the MIG stubs and the short-circuiting algorithm described in Section5. The reasons why short-circuiting was provided this way are discussed in Section 7.2.

to object pointers. We have not yet completed our implementation of this feature for short-circuited RPC, so we are still unable to provide a generic MIG stub-based short-circuit interface to kernel services.

Despite this limitation, we have done parallel work to provide trap-based access to many common kernel services that were previously accessed via IPC messages. We have added new traps for about 15 functions—including **vm_protect**, **vm_region**, **mach_port_destroy**, **task_info**, **task_set_exception_ports**, and others—bringing the total number of trap-accessible kernel services to 35. While this is not quite as efficient as providing short-circuited RPC access to these services, it is still a significant optimization compared with IPC messaging. Once we have completed the necessary MIG work, we expect a modest further performance gain in this area.

7.3 Other VM Optimizations

In addition to the collocation-related optimizations reported in Section 6, we have added a variety of other important VM optimizations. These include:

- Improvement of the page fault handler by careful optimization of many the common cases, including: map-in with page present, zero-fill, and shadow-based copy on write.

- Caching of pmaps, an optimization that is especially beneficial on the 80x86 architecture.

- Bypassing the kernel iodone thread. The iodone thread was necessary because I/O completion was signalled via an IPC message, which cannot be sent from interrupt level. By eliminating the message, I/O completion can be queued from interrupt level and retrieved directly by the server. This eliminates a context switch and the expense of an IPC message.

7.4 Miscellaneous Optimizations

A variety of other optimization, some of which resulted in considerable performance gains, have also been incorporated in our enhanced MK system. These include:

- Elimination of MP lock overhead for the uniprocessor configuration of the server. This turns out to be the largest single non-collocation-related optimization. For the uniprocessor configuration we have replaced the fine-grained MP locking with a single master lock that each thread holds as long as it does not block. This makes the server threads non-preemptible, just as they are in the integrated kernel. The considerable performance benefits arise because context switching and costly locking is greatly reduced.

- Avoiding a call to **task_thread** when non-superuser tasks exit. The OS server uses this call to discover the list of threads in a task. However, a non-superuser task can only have created threads by calling the OS server, hence such a task should not contain any threads unknown to the OS server. This makes the call to **task_thread** redundant.

- Changing the file system buffer cache from write-through to write-back.
- Tuning of OS server thread priorities.

8 Results

We performed a variety of comparative tests on the enhanced MK, OSF/1 MK 1.3, and OSF/1 1.3 IK operating systems to measure the efficacy of our performance optimizations. We chose to compare the enhanced MK with a production-configured OSF/1 1.3 integrated kernel because our general goal is a micro-kernel that demonstrates performance comparable to a normally configured version of an established IK design. To gauge overall performance we chose the AIM III test, which simulates a mixed work load with a varying number of users. Because AIM III typically gives the worst results for MK systems when compared with IK systems we feel that the positive results we got are highly significant.

Figure 9 shows the overall results on an 8-user AIM III test. Our goal was to have the enhanced MK run the test at no more than 105% of the OSF/1 IK timing. As the chart shows, the enhanced MK actually ran 1.4 seconds faster than our target time. When we ran the AIM III test at loads varying from 2 to 30 users we achieved our goal of 95% or better performance throughout the range, as shown in Figure 10.

Performance Summary

8 User AIM III in i486 50 MHz - Real Time Performance

Untuned OSF/1 MK5	133 seconds
OSF/1 Integrated Kernel	85.5 seconds
Goal (105% of IK)	90 seconds
ACHIEVED	**88.6 seconds**
Total Savings vs MK5	44 seconds

Figure 9. AIM III Performance Summary

We broke down the results according to type of optimization by adding one optimization at a time and testing the system as each new optimization was added. Figure 11 shows the breakdown of the real-time results.[15] As expected, collocation-related optimizations (roughly 27 seconds) account for well over half the total performance gain.[16] By far the most important single optimization was system call short-circuiting (19 seconds), which accounts for nearly four times the gain of the next best optimizations. Certain non-collocation-related gains were also highly significant, including the bulk of the C-threads

[15] The buffer cache results are for CPU rather than real time.

[16] As explained in Section 7.1, some part of the improvement in Cthreads performance is attributable to collocation. This is not reflected in the total figure for collocation-related gains.

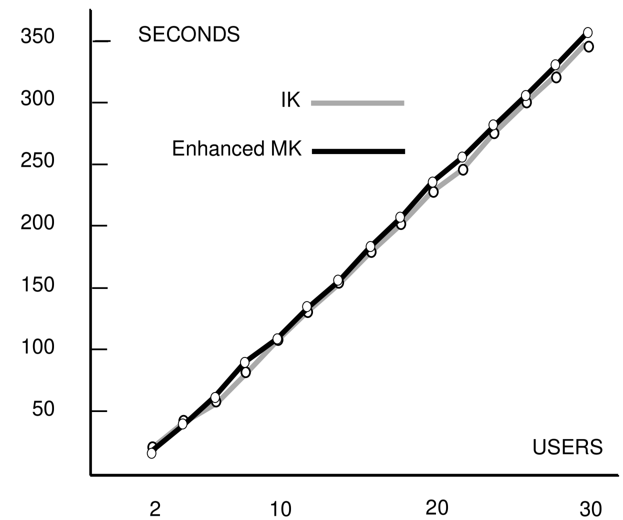

Figure 10. Enhanced MK vs. OSF/1 1.3 IK on 1-30 User AIM III

optimizations (5.5 seconds), the non-MP lock optimization (4 seconds), and the change in the file system buffer cache (4 seconds).

It is important to note that the results of the separate optimizations, when measured this way, are not additive. Frequently, adding a new optimization reduces the benefits of some previous optimization. For example, the benefits of optimizing the device access path for a collocated server tended to be offset by the FS buffer cache optimization, which reduced the number of device writes. Nevertheless, our figures give at least a first-order approximation of the benefits associated with each optimization. And whatever the breakdown, we have clearly achieved our goal of a microkernel system that performs on a par with a production quality integrated kernel under the AIM III test.

9 Future Directions

Although we have already achieved our original performance goals, we are continuing to evolve our work along two lines. First, we are moving away from a collocation model in which tasks are collocated in a single address space and towards one in which programs are collocated within a single task. Second, we are working to develop a general Mach RPC

MKPP Sources of Improvement

8 User AIM III on i486AT 50 MHz
Real Time Performance Gains vs MK5

Short Circuited System Calls*	19 seconds
Cthreads Optimizations	5.5 seconds
Replace Mapping of User Memory into Collocated Server with Optimized Copy-in/Copy-out*	5 seconds
Eliminate MP Lock Overhead for Uniprocessor	4.2 seconds
Optimize FS Buffer Cache**	2 seconds
Traps for Common Mach Requests by Collocated Server*	1.5 seconds
Avoid task_thread call on exit	1.1 seconds
Mach Copy-in of Syscall Arguments for Collocated Server*	2 seconds
Use Same Config Switches as IK	1 second
Tune Server Thread Priorities	1.5 seconds
Optimize VM Fault Path in the Kernel	1.7 seconds
Optimize Device Access Path for Collocated Server*	.6 seconds

*Collocation-specific **CPU Time

Figure 11. Sources of Performance Improvement

service that can be used by non-collocated programs to make efficient, non-message-based RPCs.

9.1 Collocation

The current design of collocation, in which multiple tasks are collocated in a single address space, was convenient to implement, but it still requires a fairly costly thread migration mechanism. Our goal is to move to collocation of programs within a single task and to use a more efficient thread migration mechanism. Under this model, each server will export a thread migration routine tailored to the server and machine environment. Such a server-specific mechanism should generally be more efficient than the generic thread migration algorithm—requiring an activation switch—that we currently use.

Under this new model, kernel collocation can be implemented in one of two ways. In one version, the Mach kernel continues to be a separate task, but all collocated servers are loaded into a single task. In the other version, all servers are collocated in the same task as the Mach kernel itself. The latter version may be somewhat more difficult to implement, but it is more appealing because it consistent with the general collocation model that we will use

in user space. Given that kernel collocated tasks already share the same port name and VM address space, there seems little reason to keep tasks separate.

One way we can improve the efficiency of kernel-to-kernel short circuited RPC is to avoid switching stacks. Under the current optimization of server calls into the kernel, we already avoid the activation switch in the thread migration algorithm. With modifications to the kernel, and possibly to Cthreads, it should be possible to jump into kernel code without a stack switch.

9.2 A Mach RPC Service

Once the migrating threads mechanism is in place, it becomes possible to implement a general service to allow clients to invoke efficient, non-message-based RPCs by trapping into the kernel. The kernel can use the migrating threads mechanism to invoke an RPC in which the client thread migrates to a server in any address space. Since RPC has been raised to the status of a first class object in the kernel, it makes sense to provide such generic service to invoke RPCs, just as we currently provide a generic Mach message service. We have therefore designed a new Mach RPC service that allows clients to invoke RPCs via a trap-based Mach RPC system call.

MIG client stubs will no longer make use of Mach messages. Instead, they will make the new Mach RPC system call, passing the MIG message number, target port, and the MIG arguments. A new trap handler in the kernel will use the message number and target port to invoke an RPC in which the client thread migrates to the server, using the migrating threads mechanism described in this paper.

The Mach RPC service can be used both for user-to-kernel RPC and for user-to-user RPC when the user-space client and server are not collocated. (It can also be used in the case where client and server are collocated, but it is less efficient than short-circuited RPC.) As such it nicely complements the RPC short-circuiting mechanism. Together, these will provide a set of efficient RPC solutions, using a small number of mechanisms, that cover all cases: kernel-to-kernel, user-to-kernel, kernel-to-user, user-to-user, collocated, and not collocated.

The Mach RPC service can also be extended transparently to inter-node RPC. For example, Mach RPC can be used make location-transparent requests for OS services. When OS services are based on service-specific traps, it must be assumed that the OS server is on the local node. When OS services are requested via the Mach RPC service, the OS server need not be on the local node, and the client need not have any knowledge of its location.

One obvious use for the Mach RPC service is to optimize system calls in emulator-based systems. In current emulator-based systems, application emulator code makes message-based RPCs to user-space OS server emulator code. The system call path can be made far more efficient by using the Mach RPC service instead a message-based RPC. Because the Mach RPC service can be used to make both user-to-user and user-to-kernel RPCs, this mechanism will work for both kernel collocated and user-space OS servers.

Acknowledgments

Besides the authors of this paper, significant contributors to the MK Performance Project at the OSF Research Institute include David Black, Randall Dean, and Franklin Reynolds. We also collaborated closely with Brian Ford and Jay Lepreau at the University of Utah in adopting the migrating threads algorithm.

References

[1] Michael Condict, Dave Mitchell, and Franklin Reynolds. “Optimizing Performance of Mach-based Systems By Server Co-location: A Detailed Design”. *Operating Systems, Collected Papers*, Vol. 2. The OSF Research Institute, October 1993.

[2] Bryan Ford and Jay Lepreau. “Evolving Mach 3.0 to a Migrating Thread Model”. UUCS-93-022.

[3] Randall W. Dean. “Using Continuations to Build a User-Level Threads Library”. *USENIX Mach III Symposium Proceedings*, April 19-21, 1993.