Evolving Mach 3.0 to a Migrating Thread Model

Bryan Ford *Jay Lepreau*

University of Utah

Abstract

We have modified Mach 3.0 to treat cross-domain remote procedure call (RPC) as a single entity, instead of a sequence of message passing operations. With RPC thus elevated, we improved the transfer of control during RPC by changing the thread model. Like most operating systems, Mach views threads as statically associated with a single task, with two threads involved in an RPC. An alternate model is that of migrating threads, in which, during RPC, a single thread abstraction moves between tasks with the logical flow of control, and "server" code is passively executed. We have compatibly replaced Mach's static threads with migrating threads, in an attempt to isolate this aspect of operating system design and implementation. The key element of our design is a decoupling of the thread abstraction into the *execution context* and the *schedulable thread of control*, consisting of a chain of contexts. A key element of our implementation is that threads are now "based" in the kernel, and temporarily make excursions into tasks via upcalls. The new system provides more precisely defined semantics for thread manipulation and additional control operations, allows scheduling and accounting attributes to follow threads, simplifies kernel code, and improves RPC performance. We have retained the old thread and IPC interfaces for backwards compatibility, with no changes required to existing client programs and only a minimal change to servers, as demonstrated by a functional Unix single server and clients. The logical complexity along the critical RPC path has been reduced by a factor of nine. Local RPC, doing normal marshaling, has sped up by factors of 1.7–3.4. We conclude that a migrating-thread model is superior to a static model, that kernel-visible RPC is a prerequisite for this improvement, and that it is feasible to improve existing operating systems in this manner.[1]

1 Introduction and Overview

We begin by defining and explaining four concepts that are key to this paper. They are kernel and user *threads*, *remote procedure call*, *static threads*, and *migrating threads*. We explain how kernel threads interact in implementing RPC, and the difference between implementing RPC with static and migrating threads.

Threads As the term is used in most operating systems and thread packages, conceptually a *thread* is a *sequential flow of control*[4]. In traditional Unix, a single process contains only a single kernel-provided thread. Mach and many other modern operating systems support multiple threads per process (per *task* in Mach terminology), called *kernel threads*. They are distinguished from *user threads*, provided by user-level thread packages, which implement multiple threads of control atop kernel-provided threads, by manipulation of the program counter and stack from user-space. In the rest of this paper, we use the term "thread" to refer to a kernel thread, unless qualified.

In most operating systems, a thread includes much more than the flow of control. For example, in Mach 3.0[24] a thread also (i) is the *schedulable entity*, with priority and scheduling policy attributes; (ii) contains *resource accounting statistics* such as accumulated CPU time; (iii) contains the *execution context* of a computation—the state of the registers, program counter, stack pointer, and references to the containing task and designated exception handler; (iv) provides the *point of thread control*, visible to user programs through a *thread control port*.[2]

[1] This research was sponsored in part by the Hewlett-Packard Research Grants Program and by the OSF Research Institute.

[2] In Mach, a *port* is a kernel entity that is a capability, a communication channel, and a name. If one has the name of a port,

RPC *Remote procedure call*, as the name suggests, models the procedure call abstraction, but is implemented between different tasks. The flow of control is temporarily moved to another location (the "procedure" being called) and later returned to the original point and continued. RPC can be used between *remote* computing nodes, but is often used between tasks on the same node: *local RPC*. This paper focuses only on the local case; in the rest of the paper, for brevity, we use the unqualified term "RPC" to refer only to local RPC (more restricted than the usual use of the term). When a thread in a *client* task needs service provided by another task, such as opening a file, it bundles up a packet of data containing everything the service provider (the *server*, or file system in this case) needs to process the request. This is known as *marshaling* a *message*. The client thread then invokes the kernel to copy the message into the server's address space and allow the server to start processing it. Some time later, the server returns its results to the client in another message, allowing the client to continue processing.

It is important to note that although virtually all modern operating systems provide an RPC abstraction at some level, there is a continuum in that support. Some OS's support RPC as a fully kernel-visible entity (Amoeba[29]), some provide a kernel interface and special optimizations for the combined message send and receive involved in RPC, but fundamentally understand only one-way message passing (Mach), while some support RPC only through libraries layered on other inter-process communication (IPC) facilities (Unix).

Static Threads The difference between static and migrating threads lies in the way the control transfer between client processing and server processing is implemented. In RPC based on static threads, two entirely separate threads are involved: one confined to (or *static* in) the client task, and the other confined to the server task. When the client invokes the kernel to start an RPC, the kernel not only copies the client's message into the server, but also puts the client's thread to sleep and wakes up a thread in the server to process the message. This thread, known as a *service thread*, was created previously by the server for the sole purpose of waiting around for RPC requests and processing them. When the service thread is finished with the request, it invokes the kernel, which puts the server thread back to sleep and wakes up the client thread again. In switching control from one thread to another, a full *context switch* is involved—a change of address mappings, task, thread, stack, registers, priority, etc., including an invocation of the kernel scheduler.[3]

With RPC based on static threads, the server's *computational resources* (the right to use the CPU) are used to provide service to the client. In the object-oriented world, this is known as an "active object" model[9], because a server "object" contains threads that actively provide service.

Migrating Threads If RPC is fully visible to the kernel, an alternate model of control transfer can be implemented. *Migrating threads* allows threads to "move" from one task to another as part of their normal functioning. In this model, during an RPC the kernel does not block the client thread upon its IPC kernel call, but instead arranges for it to continue executing in the server's code. No service thread needs to be awakened by the kernel—instead, for the purposes of RPC, the server is merely a passive repository for code to be executed by client threads. It is for this reason that in the object-based world, this is called the "passive object" model. Only a partial context switch is involved—the kernel switches address space and some subset of the CPU registers such as the user stack pointer, but does not switch threads or priorities, and never involves the scheduler. There is no server thread state (registers, stack) to restore. The client's own computational resources (rights to the CPU) are used to provide services to itself. Note that *binding* in this model can be very similar to that in a thread switching model and will be detailed in Section 6.2.

Although most operating systems support RPC using the static thread model, whether kernel-visible or not, it is important to note that this is not the case for *all* system services. All systems using the "process model"[14][4] for execution of their own kernel code (e.g., Unix, Mach, Chorus, Amoeba), actually "migrate" the user's thread into the kernel address space during a kernel call. No context switch takes place—only the stack and privilege level are changed—and the user thread's resources are used to provide services to itself.

one can perform operations on the object it represents.

[3] Sometimes this invocation of the scheduler is heavily optimized and inlined into the IPC path, but it is still there.

[4] The "process model" is in contrast to the "interrupt model," as exemplified by the V operating system[8], in which kernel code must explicitly save state before potentially blocking.

Static vs. Migrating Threads In actuality, there is a continuum between these two models. For example, in some systems such as QNX[20], certain client thread attributes, such as priority, can be passed along to ("inherited by") the server's thread. Or a service thread may retain no state between client invocations, only providing resources for execution, as in the Peregrine RPC system[21]. Thus it can become impossible to precisely classify every thread and RPC implementation. However, most systems clearly lie towards one end of the spectrum or the other.

1.1 Providing Migrating Threads on Mach

Mach uses a static thread model, and a thread contains all of the attributes outlined in the "Threads" paragraph above. In our work, we decoupled these semantic aspects of the thread abstraction into two groups, and added a new abstraction, the *activation stack*, which records the client-server relationships resulting from RPCs. A *thread* is now only: (i) the logical flow of control, represented by a stack of *activations* in tasks; and (ii) the schedulable entity, with priority and resource accounting attributes. An *activation* represents: (i) the execution context of a computation, including the task whose code it is executing, its exception handler, program counter, registers, and stack pointer; and (ii) the user-visible point of control.

The abstraction exported to user code that corresponds to the old "thread" abstraction is now what we internally call the "activation." This is not only what makes sense for the needs of user programs, but also provides compatibility with original Mach 3.0.

The real thread as defined above, the schedulable entity, is no longer subordinate to a task. By making RPC a single identifiable entity to the kernel, and explicitly recording the relationship between individual activations in the activation stack (which result from RPC), we have elevated RPC to an entity fully visible to, and supported by, the kernel, instead of a sequence of message passing operations. Our thread abstraction now more closely models the original *conceptual* basis of a thread: a logical flow of control. It turns out that elevating the thread and RPC abstractions also enhances *controllability*, because the kernel can now take more elaborate and precise actions on a single activation or on the entire thread. For example, it can propagate information ("alerts") along the chain of activations. Another benefit of introducing to the kernel the notion of an inter-task RPC, is that a number of aggressive IPC optimizations become possible. This was one of our original motivations, but many other benefits have since surfaced.

1.2 Outline of Goals and Benefits

Our original goals in this project were several: (i) change Mach 3.0's thread model to a migrating one, (ii) retain backwards compatibility, and (iii) enable performance improvements via RPC optimizations not possible with static threads. During the design and implementation, we discovered we could achieve much more: (iv) ordinary message-based (fully marshaled) RPC became much faster, (v) thread controllability was enhanced, (vi) kernel code became much simpler, (vii) an apples–apples comparison of static vs. migrating threads was achieved, and (viii) several other advantages, discussed below, became evident.

In the rest of this paper we describe this work in detail. We first discuss related work, then cover the advantages of a migrating thread model, describe our kernel implementation and interface, including discussion of the thread controllability issues, examine how RPC works in the new system, and mention how the Unix server could be changed to better leverage migrating threads. Finally, we present the implementation status and preliminary results, outline future work, and the conclusions we draw from this work.

2 Related Work

Most operating systems use a static thread model, but there are a number of exceptions. Sun's Spring[19] operating system supports a migrating thread model very similar to ours, although it uses different terminology. Spring's "shuttle" corresponds to our "thread," and their "thread" corresponds to our "activation." Spring addresses the controllability issues but did not have to be concerned with backwards compatibility. Alpha[11] was probably the first system to fully adopt migrating threads. It is oriented to real-time constraints, and its migrating thread abstraction is especially important for carrying along scheduling, exception-handling, and resource attributes. In both of these systems a thread can migrate across nodes in a distributed environment, and indeed Alpha's term for a migrating thread is a "distributed thread." Psyche[27] is a single-address-space system that supports migrating threads. The Lightweight RPC system[3] on Taos exploited

migrating threads (control transfer) as a critical part of its design, but focused on high-performance local RPC, and included additional data transfer optimizations. This makes it difficult to isolate the benefits of the improved control transfer. Object-oriented systems have traditionally distinguished between "active" and "passive" objects, corresponding to static and migrating thread models[9]. Clouds[16] exemplifies a passive object (migrating thread) model, while Emerald[5], as we do, provides both active and passive objects—support for both styles of execution. Chorus[26] can use only thread-switching between user-level tasks, but between tasks running in the kernel's protection domain it has "message handlers" which operate in a migrating thread model.

We believe that many of these migrating threads systems did not fully address the attendant controllability issues—did not fully support debugging or need to provide Unix signal semantics, for example.

"Scheduler activations"[1] are kernel threads with special support for user-level scheduling. Scheduler activations are concerned primarily with the behavior of kernel threads *within* a protection domain, while our work deals with thread behavior *across* protection domains. As such, these works are largely orthogonal and theoretically could be combined in the same system, but we do not deal with this issue here.

Except for LRPC on Taos, all of the existing systems supporting migrating threads were designed this way from the start. They are all different from traditional operating systems in many ways other than thread model. To our knowledge, heretofore the thread model issue itself has not been separated out and examined, comparing all of performance, functionality, and simplification. Our goal is to do this by comparing the two thread models in the same operating system, providing information focused on the thread model. By implementing migrating threads on Mach 3.0, we also demonstrate how an existing operating system with static threads can be adapted to migrating threads.

3 Motivation

A migrating thread approach has several advantages which are outlined in this section. The majority of the benefits are linked to use with RPC and are described first. But there are also controllability advantages for threads during *all* kernel interaction, and these are outlined in section 3.2. In the context of the Alpha OS, [11] also discusses many advantages offered by migrating threads.

3.1 Remote Procedure Call

Many of the advantages of migrating threads stem from their use in conjunction with RPC. Migrating threads provide a more appropriate underlying abstraction on which to build RPC interfaces than do static threads. Many of the problems with static threads stem from the semantic gap between the control model—a procedure call abstraction within a single thread of control, and the mechanism used to implement the model—two threads executing in different tasks. Using migrating threads for RPC provides benefits in performance, functionality, and in ease of implementation. Since RPC is very frequently used[3], especially in newer microkernel-based operating systems where most internal system interactions are based on RPC, this aspect of the system can be of great importance in determining the performance and functionality of the system as a whole.

Invocation Efficiency

For RPCs to be performed in the static thread model, two threads, one in each task, must synchronize in the kernel. Two thread-to-thread context switches are required during the operation: one on call and one on return. However, in the migrating thread model, the entire RPC can be performed by just one thread that temporarily moves into the server task, performs the requested operation, and then returns to the client task with the results. No synchronization, rescheduling, or full context switch need be done.

Thread migration also permits optimizations such as those done in LRPC[3] and in other flexibly structured or shared address space systems, e.g., Lipto[15], FLEX[7], and Mach In-Kernel Servers[22, 17]. In these systems there is some degree of inter-domain memory sharing or protection relaxation, thus blurring domain boundaries. RPC implemented by threads that migrate from one domain to another can take advantage of this boundary blurring, providing many optimizations in argument passing and stack handling. At the limit, RPC implemented in a migrating thread model could be specialized to a simple procedure call.

These advantages apply in general to any *service invocation* mechanism, not just large servers invoked through RPC. For example, in an object-based environment, invocation of relatively fine-grained objects is prohibitively inefficient if all objects must be active. With passive objects, it is more feasible to apply similar invocation abstractions to both medium and course-grained objects.

While the existence of fast, efficient microkernels based on static threads demonstrates that high performance is possible in that model, such systems often impose semantic restrictions that distort their implementation towards a migrating thread model. For example, QNX[20], a commercial real-time operating system, supports only unqueued, synchronous, direct process-to-process message passing with priority inheritance; this design makes it a *de facto* migrating threads system. Other microkernels, such as L3[23], retain the full semantics of static threads, but to achieve high performance must impose severe restrictions on the flexibility of scheduling and other aspects of the system not directly related to RPC.

Thread Attributes and Real-time Service

In the static thread model, when a client task performs an RPC, control is transferred to an entirely different thread that has its own scheduling parameters such as execution priority, as well as other attributes such as resource limits. Unless specific actions are taken, the attributes of the thread in the server will be completely unrelated to those of the client thread. This can cause the classic problems of *starvation* and *priority inversion*[13], when a high-priority client is unfairly made to compete with low-priority clients that are accessing the same server. On the other hand, if the client thread migrates into the server to perform the operation, all such attributes can be properly maintained with no extra effort. Obviously, this issue is of particular importance to systems providing real-time service.

A related advantage is in resource accounting, which can be made more accurate since the work done in a server on behalf of a client can automatically be so attributed.

Interruptions during RPC

Often, due to asynchronous conditions, it is desired to interrupt an RPC in which a client is blocked, either temporarily or permanently. To do this cleanly in the static thread model, it is not enough merely to abort the message send/receive operation, because the server will continue processing the request without any indication that the client no longer desires its completion. If some entity wants to abort an RPC in which a thread is blocked, it must find the server to which the RPC is directed, know how to interact with that server enough to send it a request to abort an RPC operation, and provide the server with some kind of identification specifying which RPC is to be aborted. This usually proves to be a complex and difficult process. In addition, every server that may be accessed must support these abort operations. This can be difficult to guarantee in practice, especially if any user-mode task can set itself up as a "server" and allow other user threads to make RPCs to it, as Mach 3.0 allows. Migrating threads, on the other hand, provide a channel through which standardized requests for interruption can be propagated.

Server Simplification

In the case of "personality servers" that emulate monolithic operating systems such as Unix or OS/2, we expect migrating threads to simplify the server, because the original operating system on which the server is based is likely to have used a limited migrating thread model, in which threads "migrate" into the monolithic kernel for system calls. Maintaining this model in the personality server should achieve greater code re-use and simplify the handling of system call interruptions, thread management, and control mechanisms such as Unix signals.

We also expect migrating threads to simplify RPC service in servers, due to the anticipated simpler management of activation pools than thread pools, as described in Section 6.2.

Kernel RPC Path Simplification

As later shown by our results in Section 8.5, migrating threads greatly simplify the kernel RPC path as well. RPC paths based on migrating threads tend to be short and flow naturally, while optimized RPC paths based on static threads are often long, convoluted, and contain innumerable tests.

3.2 Thread Controllability and Kernel Simplification

A migrating threads implementation gives other controllability and simplification benefits unrelated to RPC. In a static thread model, threads are often intended to be *completely controllable resources*. Ideally, in this model, any entity with appropriate privilege, such as a program holding a "thread control port" in Mach 3.0, is able arbitrarily to stop a thread and modify its state, at any time. Conceptually, threads execute only user-mode instructions, and therefore there is never a time when system integrity could be violated by manipulation of the thread.

Unfortunately, this model in its purest form does not work in real operating systems. Threads must be able to invoke kernel-level code in order to communicate with other entities in the system, if they are to do anything more than pure computation. Since a thread executing unknown kernel code may *not* be arbitrarily manipulated, the model of complete controllability must break down somewhat: it must be possible to defer or reject thread control operations when necessary.

Traditional operating systems have various ways of working around this problem which usually work, but are often complex, inconsistent, and unpredictable. For example, Mach 3.0 provides a thread control operation which aborts a system call in which the target thread is blocked, so that the thread can be manipulated. However, many kernel operations cannot be aborted in a transparent, restartable way, so the entity trying to control the thread may have to wait an arbitrary length of time, or retry an arbitrary number of times, before it can safely do so. If this is the case, who is really being controlled—the target thread, or the thread trying to control it?

Since the complete controllability model is not realistic anyway, reducing the ambitiousness of the model, to allow for migrating threads, provides more precisely defined semantics for thread manipulation. In fact, by forcing the boundaries of controllability to be explicitly defined, and recording the flow of control across tasks, additional thread control mechanisms such as cross-domain "alerts" can be provided by the kernel. Defining the boundaries of control also makes all control mechanisms much simpler to implement, as shown in Section 8.5.

4 Kernel Implementation

In this section we describe the underlying structure of our implementation of migrating threads in the Mach 3.0 microkernel. Many of the techniques we used could be similarly applied to other traditional multithreaded operating systems such as monolithic Unix kernels.

4.1 Thread Implementation

Conceptually, a traditional Mach 3.0 user thread started executing in a particular task, and occasionally trapped into the kernel to communicate with "outside" entities. The kernel later returned from the system call and resumed the user code. The initial and normal location of a thread was in user space, and threads only "visited" the kernel occasionally, to request services.

In our migrating thread implementation, the situation is in a sense reversed. A thread starts executing as a purely kernel-mode entity, and later makes an upcall[10] into user space to run user code. Conceptually, the kernel is "home base" for all threads: the only time user-level code is executed is during "temporary excursions" into a task. A thread executing in user mode is associated with the task in which it is currently running, but a thread running in the kernel is not tightly associated with *any* user-level task.

While a thread in the kernel can now make upcalls into user space, the traditional kernel/user interface is still preserved. Once a thread is executing in user space, it can make calls *back* to the kernel in the form of traps and exceptions. Alternatively, the kernel can make further upcalls into the same or a different user task. This redefinition of the kernel/user interface is the primary mechanism supporting migrating threads in our implementation.

A distinction should be made between the "kernel" and what we refer to as "glue" code. The kernel is conceptually a protection domain much like a user-level task, in which threads can execute, wait, migrate in and out, and so on; its primary distinction is that it is specially privileged and provides basic system control services. Glue code is the low-level, highly system-dependent code that enacts the *transitions* between all protection domains, both user and kernel. The distinction between the kernel and glue code is often

overlooked because both types of code usually execute in supervisor mode and are often linked together in a single binary image. However, this does not necessarily have to be the case; for example, in QNX[20], the 7K "microkernel" consists of essentially nothing but glue code, while the "kernel proper" is placed in a specially privileged but otherwise ordinary process. It will become clear in later sections that even though the kernel and glue code may still be lumped together, in the presence of migrating threads the distinction between them becomes extremely important.

4.2 Control Abstractions and Mechanism

Even in the static thread model, in practice the goal of complete controllability of threads cannot be fully realized. While the case of a thread being in the kernel can to some extent be worked around as a special case, with the addition of migrating threads the controllability issue must be more carefully considered. Now, not only is the kernel "out of bounds" for thread control, but in order to maintain protection between tasks, threads that have migrated to other protection domains may also be uncontrollable. For example, if one thread in a client migrates into a server for an RPC, it would not be permissible for another thread in the same client to stop or manipulate the CPU state of the first thread while executing server code.

To provide controllability and protection at the same time, we split the concept of a "thread" into two parts: the part used by the scheduler, and the part providing explicit control. The first, which we still refer to as the "thread," migrates between tasks and enters and leaves the kernel. The second, a user-mode invocation or *activation*, remains permanently fixed to a particular task. Arbitrary control is permitted *only on a specific activation*, not on the thread as a whole.

Whenever a thread migrates into a task (including the initial upcall from the kernel on thread creation), an activation is added to the top of the thread's "activation stack." When a thread returns from a migration, the corresponding activation is popped off the activation stack. This new kernel-visible abstraction, the stack or chain of activations, helps provide controllability.

Activations are created either implicitly during thread creation, or explicitly by servers expecting to receive incoming migrating threads. An explicitly created activation is *unoccupied* until a thread migrates into the task and "activates" it.

Within the kernel, control of activations is implemented primarily with *asynchronous procedure calls*, or APCs, similar to asynchronous traps (ASTs) in monolithic kernels. When returning from the kernel into an activation, glue code checks for APCs attached to the activation and if present, calls them. For example, to suspend an activation, an APC is attached to that activation which will block until resumed. Previously, Mach dealt with thread suspension as part of the scheduler, adding more complexity to its already-complex state machine; now the scheduler knows nothing about one thread suspending another. Instead, the kernel's ordinary blocking mechanism is used, in which a thread only "suspends" itself.

4.3 Kernel Stack Management

Since the activation chain can be broken at any point, all linkage information between activations is stored in the activations themselves, and a single kernel stack is sufficient for the entire thread. This is also required for it to be possible to do task migration across nodes in a distributed system, because state held on a kernel stack cannot be easily encapsulated for transport. This explicit saving of state is generically known as using a *continuation*, although our implementation is very different from the way continuations have been used in Mach in the past[14]. In particular, we confine continuations purely to "glue" (transition) code; all high-level kernel code uses an ordinary process model.

5 Controllability: Semantics, Interface, and Implementation

In this section we describe the semantics of thread control operations, the interface to those operations, and some aspects of the implementation. We believe our approach could be similarly applied to other traditional multithreaded operating systems.

5.1 Thread Control Interface

In the original Mach kernel, threads were exported to user-mode programs in the form of *thread control ports*, through which control operations could be invoked. In our system, while threads still exist, the control

abstraction presented to user-level code is instead the *activation control port*. This can work because the old thread execution abstraction exported to the user was bound to a single task, like activations are now. We maintain compatibility with existing Mach code by making activation control ports direct replacements for thread ports at the binary level—all system calls which previously expected or returned thread ports now use activation ports instead. For compatibility at the source level, appropriate synonyms are provided.

Alerts

In our migrating threads implementation, we have provided the functionality of Mach 3.0's `thread_abort` call, which aborts an in-progress kernel operation and returns control to user code. However, we have provided it in a cleaner and more general form. An *alert* is a form of asynchronous message passed from a client to the kernel or a server it is calling, asking the callee to abort the requested operation and return control to the client as soon as possible. Alerts are primarily an information-passing mechanism supported by the kernel in a uniform way. They do not in themselves provide control over threads, because they have no forcefulness: alerts are merely "requests," not "demands." We currently implement only a polling interface for a server to discover an alert, but probably will also provide an exception interface in the future. Alerts are much like those in Spring[19], The "Alert" and "TestAlert" facilities of Taos[4] are analogous, but apparently do not operate cross-domain.

By default, new activations added to a thread's stack have alerts blocked, to prevent interference with an unwary server's functioning. The kernel is already capable of honoring most alerts, and new servers written to work with migrating threads can be designed to honor them too. In effect, we have provided a generic interruption request mechanism which works uniformly for both migrating RPCs and kernel calls.

We also provide another operation which first generates an alert at the target activation, then breaks the chain, returning control to the client immediately. This works much like termination, discussed below.

Suspension

In Mach 3.0 thread semantics, the basic purpose of suspending a thread is to *prevent it from executing any more user-mode instructions* until it is resumed. Therefore, suspending a task's threads turns that task into a "passive entity," allowing its address space and other state to be examined or modified without interference from its threads. It is not required that all of the thread's computation be immediately stopped, as long as that computation does not implicitly reference the thread's task. For example, explicit device I/O could be allowed to proceed while the thread is suspended, but kernel `copyin` and `copyout` operations, which implicitly affect the task's address space, could not.

Our current implementation allows such kernel activity to proceed. We do not expect this to be a problem in practice, but in any case it should be solved as a side-effect of related work. In that work we are further separating kernel code from "glue" code (described earlier). When an activation is suspended, the kernel ensures that neither user-mode or glue instructions will be executed *in that activation*. If the thread is executing elsewhere, it will not be affected until it attempts to return to the suspended activation.

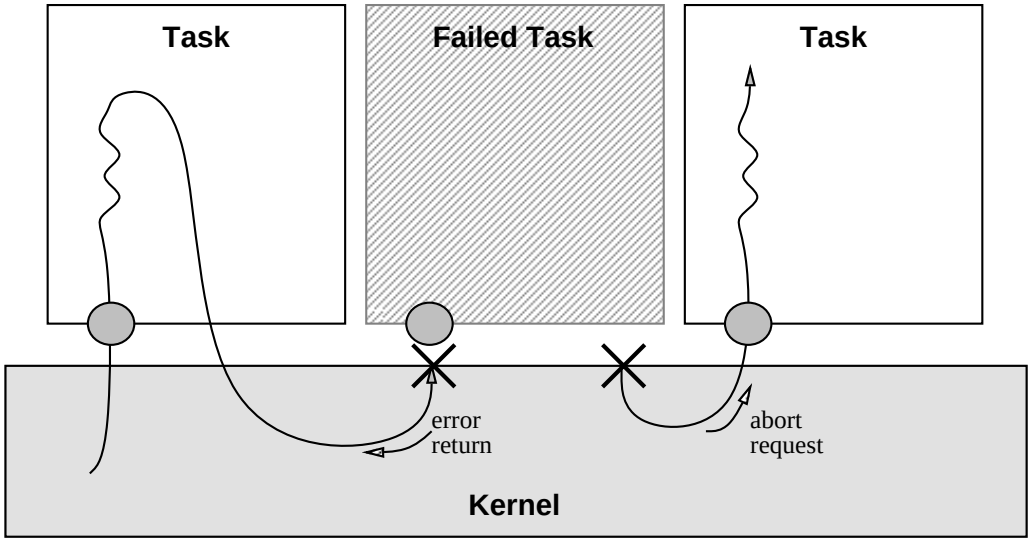
To maintain correct suspension semantics in this model, implicit references made by kernel system calls to the caller's task must be confined to glue code. This is relatively easy to do in Mach because most kernel calls are implemented as generic RPCs to kernel objects: only the low-level RPC code needs to obey the control semantics, not the actual code implementing the kernel calls.[5]

Termination

The termination mechanism in our implementation is illustrated in Figure 1. If an activation not at the top of a thread's activation stack is terminated, or if the thread is in a kernel call at the time, then the thread splits into two separate threads with identical scheduling parameters. One thread is left with the top part of the activation stack, above the terminated activation, and the other thread is given the bottom part. The thread with the upper segment continues executing in the topmost server uninterrupted, ensuring that thread termination does not violate protection. An alert is automatically propagated upward through

[5] Note that traditional Unix suspension semantics are stronger than Mach suspension semantics: they imply that kernel operations are actually completed or aborted before the thread is suspended. On Mach, both with static and migrating threads, implementing Unix suspension semantics involves other Mach control operations in addition to simple Mach suspension.

Figure 1: Activation Termination



this thread, providing a hint that the work being done is probably no longer of value. The thread given the bottom part of the activation stack returns to its now-topmost activation with an appropriate error code. This is essentially the same as the termination mechanism used in Spring.[6]

In our system, not only can a thread be split when running in user mode in a more recent activation than the terminated activation, but also when the thread is executing in the kernel, on behalf of the terminated activation. Previously, attempting to terminate a thread could potentially block for some time, while the caller waited for the victim to leave the kernel or otherwise get to a "clean point." In the new model, terminating an activation is *always* an immediate operation: if the terminated activation happens to be calling the kernel, then it is left behind to finish whatever operation it was performing and quietly self-destruct afterwards. The issue of glue and kernel operations implicitly affecting the thread's task is dealt with as described above under "Suspension."

Our termination mechanism required careful planning of kernel data structures and locking mechanisms; in particular, the line between the kernel and glue code had to be defined precisely. Once worked out, however, this technique not only added additional controllability, but considerably *simplified* the implementation of control mechanisms in the kernel, as we show in Section 8.5.

CPU State

The original Mach 3.0 design provided thread operations which, in the "ideal" complete controllability model, would allow a thread's entire CPU state to be saved, restored, examined, and modified at any time. All CPU state operations were provided by two primitives, `thread_get_state` and `thread_set_state`, defined to produce consistent results only while the target thread was suspended. However, because of the problems with the complete controllability model, many of the things for which the CPU state control mechanisms are commonly thought to be useful, in fact cannot be reliably implemented in bounded time in Mach 3.0[18]. For example, encapsulation of a task's state for checkpointing or transportation to another node either may require the controlling thread to wait an arbitrarily long time, or else requires aborting kernel operations, yielding potentially inaccurate state.

Since the existing CPU state control operations are already problematic, and it would be difficult to achieve complete backward compatibility with them, we chose to structure these operations in our migrating threads implementation to fit *current uses* of these operations: in particular, those made by the Unix server

[6] We are considering providing alternate termination semantics which fully preserve the synchronous nature of RPC. The terminated activation would be merely spliced out, so control is returned to the earlier activations only after those later in the chain have exited. It may not be possible to guarantee these semantics over a partitionable network, however.

and emulator, and by application that create their own threads and control them in straightforward ways.

Mach 3.0 requires `thread_abort` to be called on a thread just before examining or setting its state, unless the thread has just been created. Otherwise, the state operation could work with "stale" information, producing useless results. Under migrating threads, aborting an activation before manipulating its state is not strictly required. If not done, the CPU state operations wait patiently until the thread is in the target activation, without interfering with its functioning. Thus we have loosened the restrictions on these operations while maintaining backwards compatibility with their only valid usage in original Mach 3.0.

Scheduling Parameters

The final Mach 3.0 thread control operations that must be mapped to activation operations are those managing thread scheduling parameters such as priority, scheduling policy, and CPU usage statistics. Unlike the operations described above, these operations are still conceptually performed on threads rather than activations. However, the original Mach 3.0 thread control ports have become activation ports, raising the question of how the interface for these operations should be handled.

Since every active activation is attached to exactly one thread, in our current implementation we export thread operations as operations on activations, and, in the kernel, redirect the operations to the attached thread. However, this raises a protection problem, since *any* activation in the thread can modify the global scheduling state. For example, a server could lower a thread's maximum priority (which cannot be raised without special privileges) while processing an RPC, leaving the client with a "crippled" thread upon return. In our initial implementation, this is not a problem in practice because the Unix server is trusted by all clients. A better solution would be to provide an activation operation which forbids future activations higher in the stack from changing global thread state. Thread state could still be manipulated from that activation or lower (assuming it has not also been forbidden at a lower level).

5.2 Task Control Interface

Most task control operations work the same way under migrating threads as in the original Mach design. Others were modified in straightforward ways to match the new thread model. In particular, the `task_threads` call now returns a list of the activation ports of the task, instead of thread ports. When a task is suspended, resumed, or terminated, all of the activations within it (instead of threads) are similarly suspended, resumed, or terminated.

6 Migrating RPC

Once the basic kernel mechanism for supporting migrating threads was in place, it remained to demonstrate its effect on RPC performance and complexity. Because our focus at this point is primarily on control transfer during RPC, our initial implementation retains the original Mach data transfer interface, based on marshaling and unmarshaling done by user-mode stubs. In this section we describe this RPC system, as well as the changes required to servers to make them support migrating RPC. (No changes were required to make them run with traditional RPC, since the kernel itself is almost completely backward compatible.)

6.1 Client-side

From the client's point of view, RPC semantics, including binding, are unmodified. Existing binaries with normal `mach_msg` calls are supported: the kernel checks the message options to make sure they specify a true RPC, and checks the destination port to ensure that the server is capable of handling migrating RPCs. In practice, almost all MIG-generated `mach_msg` calls meet these requirements, so most clients automatically make use of migrating RPC. Note that the data can still contain port rights and out-of-line memory.

6.2 Server-side

Initializing a server to support migrating RPCs is done in nearly the same way as in servers supporting only thread-switching RPC. In accomplishing the major portion of binding, the server exports send rights to clients, exactly as before. In addition, the server must create one or more unoccupied activations, each containing a pointer to a stack in its own address space, and in the final portion of binding, the entry point

of its normal dispatch function.[7] Providing this information to the kernel can be encapsulated within a function, e.g., in the `cthreads` package.

Traditional static-thread RPC is still supported automatically. A large pool of server threads is no longer needed, but at least one must still exist to process occasional asynchronous messages, because in the current implementation this is used as a fallback mechanism when migrating RPC cannot be used.

When a migrating RPC is made into the server, the kernel allocates an unoccupied activation from the server's pool, copies the incoming message onto the server stack, and makes an upcall into the server task to the dispatch routine. This MIG-generated routine is identical to the one used to dispatch traditional messages, except that it returns through a special kernel entrypoint. On return, the kernel does not need to do any security checks or port manipulation, and the reply port provided by the client in the `mach_msg` call is never used at all.

If a migrating RPC is attempted and the kernel discovers that there are no activations currently available, in our initial implementation the kernel falls back to the normal message path, causing a normal message to be queued to the port. This is not ideal, and we plan to detect when the last available activation is about to be used for a migrating RPC, and instead of immediately making the requested RPC, temporarily "sidetrack" and make a special notification upcall into the server. At this point the server can create more activations if it deems this desirable. If it does, it returns them to the kernel and the original RPC can proceed. Otherwise, it returns immediately and the RPC blocks until a stack is freed.

When this is implemented, server management of an activation pool should be substantially more straightforward than management of a thread pool, for several reasons. The resources will be allocated on demand, by client threads themselves, instead of resource needs having to be predicted in advance, by the server. The server can use some simple decay function to deallocate activations (which are cheap for the kernel to manage since they are simply passive data structures, in contrast to kernel threads). In contrast, with a thread pool, a server is separated by the kernel "wall" from clients' requests—if inadequate numbers of server threads are present, client messages build up in the server's queues without its knowledge. The server has a more complex job as it attempts to keep the number of threads waiting for requests equal to or slightly greater than the number of processors: it has to keep track of the number of threads waiting for messages, running in the server, and blocked on outgoing RPCs or kernel calls. The last aspect is particularly awkward because it requires surrounding every such blocking call with operations to wake up and manage server threads. If it does not, deadlock can result. Finally, substantial complexity is due to multiplexing `cthreads` over kernel threads, which cannot be done with migrating threads[12].[8] However, if it is found necessary to limit the total number of executing threads in a particular server, in order to avoid saturation due to excessive kernel context switching, some of this simplification will not be present.

6.3 User-level Thread Issues

The most important issue with migrating RPC is that the user-level threads and synchronization package most widely used on Mach, `cthreads`, has significant limitations in the presence of migrating RPC.

Server Thread Management The `cthreads` library presents a significant problem to the server of a migrating RPC. Servers use `cthreads` to multiplex user threads on top of kernel threads, replacing kernel-mode context switches with much faster user-level context switches, whenever possible. However, one of the main assumptions made by the user-level threads package is that all of the kernel threads on which it is running its user-level threads are interchangeable—that one kernel thread can be used for an operation just as well as another. This assumption can be satisfied in a static thread model, although in the process it makes real-time monitoring and control of server threads difficult.

In a migrating thread model, however, kernel threads migrating in from clients are *not* interchangeable—they may have different priorities and other attributes. Even ignoring this, the return-to-kernel after an RPC has been processed must be done on the same kernel thread that the RPC came in on. In general, trying to multiplex threads in this manner loses one of the main advantages of our design: providing a kernel

[7] When programming to the RPC stub generator interface, as is usually done, this is not a change in binding semantics.

[8] Skeptics should inspect the OSF/1 server's `ux_server_loop` and related code, where the outlined complexity was found necessary for performance and safety.

entity (the activation stack) which represents a particular piece of work in progress, i.e., an entire logical thread of control. Therefore, multiplexing a server's user-level threads on top of incoming kernel threads is not appropriate. In `cthreads`, multiplexing can easily be avoided by "wiring" the user-level thread.

However, some speed is lost in the elimination of user-level thread multiplexing, because synchronization operations in the server sometimes now require kernel-level context switches instead of user-level context switches. Measuring real applications, including on multiprocessors, will be necessary before we can be sure the gains from better RPC performance are not outweighed by this additional cost. We believe that the speed advantage of user-level context switching is not as significant in typical RPC servers as it is in compute-intensive applications, which are the traditional benchmarks for thread implementations. In well-designed servers providing "system" functions, we suspect that internal contention can be minimized so that the importance of RPC speed outweighs that of context switch speed. We point out that in many commercial microkernel-based systems, including QNX[20], Chorus[26], and KeyKOS[6], OS servers do not generally multiplex user-level threads over multiple kernel threads. Instead, these systems either provide multithreading purely with kernel threads, or their functions are sufficiently decomposed so that each server can be based on a single kernel thread, requiring no internal synchronization. However, until there is more extensive performance analysis of servers using migrating RPC, losing user-level threads when servicing RPCs remains a concern.

Note that it is only for "guest" threads migrating in from other tasks that user-level thread multiplexing is a problem; threads native to the server can still use some kind of user-level thread system, or even a specialized multiplexing mechanism such as scheduler activations.

A More Appropriate Synchronization System Since `cthreads` can no longer multiplex user-level threads on kernel threads in servers, it should be replaced with a synchronization library better optimized to provide synchronization over kernel threads. Also, kernel-visible synchronization will be necessary to fully implement priority inheritance, as we discuss in a longer paper[18]. We are planning a replacement for `cthreads` that provides synchronization primitives in a user-level library, but in cooperation with the kernel.

7 The Unix Server

To function on the new kernel using traditional RPC, no changes were necessary to the OSF/1 single server and emulator, or to the libraries they use. To support migrating RPC, a few changes were required. Initially, we chose ways which have minimal impact on existing code, but better, cleaner mechanisms can be provided in the longer term. The server was modified to invoke the new setup function in the `cthreads` library and to wire incoming `cthreads`. The existing complex management of the server's thread pool, while basically no longer used, was retained. We made no modifications to the emulator. Since we are providing backwards-compatible semantics for thread manipulation, no modifications were needed to the existing complex code for handling Unix signals.

Even when our implementation is tuned, we do not anticipate a large performance improvement in the single server, to a large extent because it was written with the assumption that RPC is very expensive. Therefore, the server avoids RPC as much as possible, instead resorting to other approaches like shared memory pages, whose performance is not enhanced by migrating RPC. However, our initial goal is not primarily to show performance improvement, but to demonstrate the gain in simplicity and cleanliness provided by migrating threads, and how migrating threads can be implemented in a backward-compatible way in an existing operating system.

Desirable Modifications We anticipate that the Unix server could be made simpler with two modifications that take advantage of migrating threads. One is emulating Unix signals under Mach, and is described in [18]. Another is that because Mach 3.0 provides no standard way of propagating abort requests into RPCs, the Unix server must manually handle all Unix system call interruptions such as those caused by pending signals. It ought to be considerably simplified by taking advantage of the propagating abort operations now provided by the kernel. This would also make interruption semantics naturally extend to other servers in the system, such as ones installed by Mach-specific application programs running under Unix.

8 Results

8.1 Status

We have completed the kernel implementation of the system described in this paper. An unmodified emulator-based Unix server runs normally on the new microkernel, using traditional thread-switching RPC. A server modified to provide activations, so that it uses migrating RPC, runs multi-user. Unix signals, including `^C` and `^Z`, are working.

8.2 Experimental Environment

All timings were collected on a single HP9000/730 with 64 MB RAM. This machine has a 67 Mhz PA-RISC 1.1 processor, 128K offchip Icache, 256K offchip Dcache, 96 entry ITLB, and 96 entry DTLB, with a page size of 4K. The caches are direct-mapped and virtually addressed, with a cache miss cost of about 14 cycles. RPC test times were collected by reading the PA's clock register which increments every cycle, and can be read in user mode. Other times were obtained from the Unix server.

The system software is our port of the Mach 3.0 kernel, version NMK14.4, and the emulator-based OSF/1 single server, version 1.0.4b1. The compiler is GCC 2.4.5.u5 with full optimization.

8.3 RPC Path Breakdown and Analysis

To analyze the 3.6 times speedup in null RPC presented in the next section, we counted instructions by hand along the kernel's null RPC path. These counts are broken down by type of processing in Table 1, along with the relative (old/new ratio) and absolute (number of instructions) improvement in each category. 12% of the improvement is due to the inverted server-kernel interface: since the kernel is now "calling" the server rather than vice versa, the kernel no longer needs to save and restore the server's registers on every RPC. 21% results from the kernel's "first-hand" knowledge of RPC: it no longer needs to create, translate, and consume reply ports in order to match a reply to its request. 41% comes directly from the optimized control transfer of migrating threads: switching activations is much simpler than switching threads. 20% of the improvement is due to simpler data management, particularly the elimination of the need to maintain a temporary message buffer in the kernel's address space due to direct copy from source to destination. It is arguable whether this aspect is related to migrating threads.[9]

It is interesting that nearly half of the cost of migrating RPC now resides at the kernel-client boundary (more than half, if measured by memory operations—see below). Therefore, further improvements to other parts of the kernel RPC path will probably lead to only minimal overall speedup. That upcalls are so cheap in comparison, especially in memory operations, may have implications for system structuring in a system supporting migrating threads.

Table 1: Null RPC Path: Breakdown and Improvement

Stage	Instruction Count				Improvement (Instructions)			Improvement (Loads/Stores)	
	Switch		Migrate		$\frac{Sw}{Mg}$	$Sw - Mg$		$Sw - Mg$	
Kernel entry/exit: Client side	146	13%	99	46%	1.5	47	5%	4	1%
Kernel entry/exit: Server side	146	13%	33	15%	4.4	113	12%	58	16%
Port translation	206	18%	12	6%	17.2	194	21%	82	23%
Thread/Activation switch	408	36%	30	14%	13.6	378	41%	152	42%
Message copy	222	20%	39	18%	5.7	183	20%	66	18%
Entire kernel path	1128	100%	213	100%	5.3	915	100%	362	100%

The last two columns of Table 1 show the improvement for each stage, as measured by the number of load/store operations, which should contribute disproportionately to total cycles due to memory subsystem

[9] At first sight, this aspect may seem completely orthogonal. However, on the switching path, the message copyin is at the very beginning, while the copyout into the destination is at the end, separated by hundreds of lines of complex code. Combining these into one direct copy operation would be very difficult. On the much simpler and shorter migrating path, direct copy was easy to support. Note that even with null RPC some copying is involved: the 6 word message header.

costs. We observe that the percentage improvements in instruction count and memory operations are approximately equal for each stage of RPC. This suggests that instruction count is a valid measure of the relative contribution of each stage to the overall performance gain.

Examination of the context switch code in the old optimized RPC path explains much of its cost: the kernel essentially executes a portion of the scheduler specially hand-coded inline. Numerous constraints must be satisfied: both old and new threads must be in just the right states, run and wait queues must be maintained correctly, locks on ports, threads, IPC spaces, and other data structures must be taken and released in the right order to avoid deadlocks; timers are manipulated; interrupt levels are changed; resources acquired along the way must be carefully tracked to ensure that it will be possible to unroll everything if, for some reason, the computation falls off the optimized path.

Table 2 shows the instruction mix for each path, broken into three categories: total instructions, loads/stores, and branches. The migrating path has a somewhat higher percentage of loads and stores (56% vs. 43%), presumably due to the fact that the basic memory-intensive aspects of IPC—register saving and restoring, memory copying, and data structure traversal—are less obscured by computational overhead. The relative incidence of branch instructions is much lower, however (10% vs. 17%). This, along with the ninefold reduction in total number of branch instructions, reflects the lower logical complexity of the migrating path.

Table 2: Null RPC Path: Instruction Mix

	Switch					Migrate				
Stage	All	Load/Store		Branch		All	Load/Store		Branch	
Kernel entry/exit: Client side	146	66	45%	12	8%	99	62	62%	10	10%
Kernel entry/exit: Server side	146	66	45%	12	8%	33	8	24%	3	9%
Port translation	206	90	44%	45	22%	12	8	67%	1	8%
Thread/Activation switch	408	174	43%	78	19%	30	22	73%	3	10%
Message copy	222	86	39%	46	21%	39	20	51%	5	13%
Entire kernel path	1128	482	43%	193	17%	213	120	56%	22	10%

We expect that our results on the RPC path would, in general, extend to other architectures besides the PA-RISC. Although sometimes difficult, most architectures can achieve the single direct copy when the date are contiguous, for example by temporary mapping[23]. Changing the interrupt priority level (IPL) is done four times on the switching path due to scheduler involvement, but not at all on the migrating path; while IPL changes are cheap on the PA-RISC, they are very expensive on some other architectures[28], making migrating threads especially important on them. The unavoidable cost of address space switching is much higher on some architectures, which would lead to a lower improvement ratio, but even then we expect the benefits to be considerable.

8.4 Micro and Macro Benchmark Results

Measurements of the costs of cross-task migrating and traditional switching RPC are presented in Table 3. The columns on the left include only the kernel costs of RPC, while the ones on the right include both kernel and user (marshaling) costs, obtained in another set of runs. On this machine, a null local RPC now spends less than 10 microseconds in the kernel. The speedup from migrating threads varies with parameter size from a factor of 3.4 for null RPC, a factor of 2.0 for 1K of data, to a factor of 1.7 for long in-line marshaled data. This factor of 1.7 comes from the fact that the data is copied three times in the switching path—once during marshaling and twice in the kernel—but only twice on the complete migrating path.

One interesting observation is that the number of cycles per instruction (CPI) is considerably worse on the migrating path (3.0 = 648/213) than on the original path (2.1 = 2318/1128). We believe that some or all of this is due to two factors: the higher percentage of load/store instructions as described above, and the fact that the instructions on the hand-coded migrating path were not carefully scheduled and optimized like the C compiler did to most of the old path. Therefore, more careful coding of the migrating RPC path could somewhat lower CPI. More investigation of the CPI difference is warranted.

Table 3: RPC Times in Cycles

Test	Kernel Time			Kernel Time + User Marshaling		
	Switching	Migrating	Ratio	Switching	Migrating	Ratio
Null RPC	2318	648	3.6	2986	880	3.4
32 `In`	2379	683	3.5	3050	961	3.2
1K `In`	4753	1676	2.8	6470	3210	2.0
32K `In`	137808	$\binom{34703}{84527}$	$\binom{4.0}{1.6}$	183164	109857	1.7

The measurement of the kernel time for 32K migrating RPC showed severe side effects of the HP730's direct-mapped cache. At the top is our original measurement, a suspiciously low time resulting in a rather unbelievable 4.0× speed improvement. Below it is the same measurement taken after shifting the message buffers slightly so that the cache lines would conflict, resulting in an improvement of 1.6×, *below* the factor of two we would expect due to the data being copied once instead of twice. This demonstrates the importance of cache effects in data transfer, and deserves further investigation in the future.

As a preliminary test of overall performance impact, we measured the time for a "make" of the `gas` assembler. Under migrating threads, the elapsed time went from 109 to 107 seconds, an improvement of about 2%. The link phase took about 3 seconds. A link of a larger program (the HP linker itself), improved from 14 seconds to 12 seconds, an improvement of 14%. We believe this greater improvement is due to `ld` having a higher ratio of system calls to computation.

One area where we slow down is in RPCs to the kernel. These do not currently migrate since we haven't changed the kernel to provide activations on its ports. We do not expect doing so to be difficult. When that is done, all messages which originally would have used the optimized path (true RPCs), should be migrating.

We expect a tuned implementation to achieve more overall speedup, and if other RPC optimizations enabled by migrating threads are performed, significantly more speedup.

8.5 Kernel Code Simplification

Confined Controllability Making threads independent of tasks and uncontrollable outside of user mode greatly reduced code complexity in a number of areas. The source file containing most thread control operations was reduced by more than half, from 72K to 32K. In the new 18K source file supporting activations, support for control operations account for only about 10K.[10] This simplification largely resulted from cleaner management of thread suspension, resumption, and termination. The original Mach 3.0 thread control mechanisms had to make numerous tests for special cases, such as a thread manipulating itself, or two threads trying to control each other simultaneously, possibly causing kernel deadlock. Now that controllability is confined within well-defined boundaries, as it must be to support migrating threads, these tricky cases never occur because kernel code is always "out of bounds."

The task management code was reduced from 38K to 20K for similar reasons: the cleaner model simplified locking and eliminated many special-case situations such as the case of a thread terminating its own task.

Migrating RPC On the original switching path, the port translation and context switch code were mostly written in machine-independent C code, while the other categories were PA-specific assembly language. On the migrating RPC path, the machine-independent parts became so trivial that it was easier to inline them into the assembly language path than to go to the trouble of interfacing with a high-level language.[11] The entire 1350 lines of complex C code comprising the optimized RPC path plus about 400 hand-coded assembler instructions, were replaced with about 220 assembler instructions. The resultant simplifications in logical complexity, a factor of nine, are evident from the figures presented in Section 8.3.

[10] The rest is for allocation and freeing of activations and other administration, unrelated to the control operations.

[11] Although some speedup presumably resulted from assembly coding, we believe it was slight, and only feasible due to the simplicity of the migrating path. Those who differ are invited to inspect the original message path and try to hand-code it without changing its semantics.

8.6 Memory Use

In the original microkernel, in general only a few kernel stacks (8K each) were required per processor, due to the continuations mechanism[14]. At the beginning of this project, we disabled continuations in order to simplify our work; this immediately raised kernel memory use to one kernel stack per thread. However, with migrating threads there are now far fewer threads in the system, and kernel stacks are still associated with threads instead of activations. While running our multiuser benchmarks, we observe kernel physical memory use to be at most 300K greater than in the original system. However, we are in the process of reintroducing continuations under the new model, so even this increase should be temporary.

Regarding server virtual memory use, at this writing we statically allocate a large number of activations (40) and the old thread pool remains in the server. Therefore, about three times as much VM is used as before (2.6 MB vs. .8 MB) When we remove the thread pool, server VM use should be about the same as in the old system, because for the most part each server thread/user stack becomes one server activation/user stack. Of course, clients are unaffected because they are unmodified.

9 Future Work

This work *enables* many further improvements to Mach and Mach servers, as well as raising areas for further research. Providing an appropriate replacement for the `cthreads` synchronization primitives is important in order to make a fair evaluation of the impact of relying on kernel-level context switches. Our earlier work on moving trusted servers into the kernel's protection domain and address space (INKS)[22] used ad-hoc thread migration. By re-working the thread abstraction from scratch, our new system solves all of the problems encountered[17].

The "NORMA" (NO Remote Memory Access)[2] version of Mach 3.0 allows IPC between different nodes of a distributed memory multiprocessor, implemented in the microkernel. Extending the migrating thread system to encompass RPC between nodes should be done. The issues involved have already been explored in depth in Alpha[11].

Going in a different direction, our work allows improvements in Mach's support for real-time systems. At the implementation level, we have largely decoupled two portions of the thread abstraction: the schedulable entity (priority, scheduling policies, etc.) from the thread of control (the chain of activations). This makes it feasible to decouple them entirely, enabling a full implementation of priority inheritance.

The Mach message format imposes unnecessary overhead on migrating RPC. The migrating thread model enables other designs which could provide much higher performance, such as LRPC[3]. In cases where protection domains have been merged[22], much of the copying can be avoided.

The migrating RPC mechanism can also be used in thread exception processing. This will allow a no-emulator server, such as OSF/1-MK5[25], to do more efficient argument copying. We believe migrating RPC can also be leveraged by making the Mach pager interface synchronous, with a thread servicing its own page faults. This requires security to be explicitly provided when untrusted pagers are involved.

The OSF Research Institute is adopting our code and is planning to make many of the above improvements, in an Intel 486 base. Our code will also be available to interested parties.

10 Conclusion

We draw three main conclusions from our work. First, by changing the thread model of an existing operating system, and evaluating the two versions, we show that a migrating thread model is superior to a static model. Migrating threads provide superior functionality, performance, and code simplification. In the area of functionality, thread migration (i) provides more precisely defined semantics for thread manipulation and additional control operations, (ii) allows scheduling and other attributes to follow threads, especially important for real-time systems. In performance, thread migration (i) improves the performance of ordinary RPC, and (ii) enables a multitude of aggressive RPC optimizations, especially in systems under current research which provide cross-domain memory or address-space sharing. However, thread migration does have the potential performance disadvantage of not allowing user-level threads that service RPCs to be multiplexed atop multiple kernel threads. In reducing implementation complexity, thread migration simplifies (i) kernel

code, and, we expect, (ii) server code. In each of these areas, our implementation and measurements have demonstrated the first benefit, while potential gains from the second seem evident, but have not yet been realized through full implementation in our system.

Secondly, since migrating threads requires that the kernel treat local RPC as an identifiable semantic entity, we conclude that operating system kernels should directly support the RPC abstraction.

Our third main conclusion is that it is feasible to improve at least some *existing* operating systems, by changing their thread model from static to migrating. Even in the case of Mach 3.0, which has an unusually rich thread-manipulation interface, we show that this far-reaching change can be made while retaining backward compatibility, and with only moderate implementation effort. A key element of that implementation is "basing" threads in the kernel, which temporarily make excursions into tasks via upcalls.

Acknowledgements

We especially thank Mike Hibler for his expert help with the implementation, as well as discussion of controllability and signal issues. We thank Douglas Orr for varied help and Greg Minshall for his careful review of an earlier draft. We had helpful discussions with many members of the OSF Research Institute about many aspects of the system, and with the HP Labs Brevix group about controllability. The anonymous referees, Brian Bershad, Rich Draves, and Alessandro Forin provided many useful comments on earlier drafts, and we thank Mike Jones and Jeff Mogul for all of that, plus their patient shepherding.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] J. S. Barrera. A fast Mach network IPC implementation. In *Proc. of the Second USENIX Mach Symposium*, pages 1–12, 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] Andrew D. Birrell. An introduction to programming with threads. Technical Report SRC-35, DEC Systems Research Center, January 1989.

[5] A. P. Black, N. Huchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans on Software Engineering*, SE-13(1):65–76, 1987.

[6] Alan C. Bomberger and Norman Hardy. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, April 1992.

[7] John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A tool for building efficient and flexible systems. In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993.

[8] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), March 1991.

[10] David D. Clark. The structuring of systems using upcalls. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, WA, December 1985.

[11] Raymond K. Clark, E. Douglas Jensen, and Franklin D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146, Seattle, WA, April 1992.

[12] Michael Condict. Personal communication, November 1993.

[13] Sadegh Davari and Lui Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, 23(2):110–120, April 1992.

[14] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Asilomar, CA, October 1991.

[15] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan, June 1992.

[16] Partha Dasgupta et al. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1), Winter 1990.

[17] Bryan Ford, Mike Hibler, and Jay Lepreau. Notes on thread models in Mach 3.0. Technical Report UUCS-93-012, University of Utah Computer Science Department, April 1993.

[18] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to use migrating threads. Technical Report UUCS-93-022, University of Utah, November 1993.

[19] Graham Hamilton and Panos Kougiouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.

[20] Dan Hildebrand. An architectural overview of QNX. In *Proc. of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[21] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software — Practice and Experience*, 23(2):201–221, February 1993.

[22] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeff Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proc. of the Third USENIX Mach Symposium*, pages 39–55, April 1993.

[23] Jochen Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993.

[24] Open Systems Foundation and Carnegie Mellon Univ. *MACH 3 Kernel Interface*, 1992.

[25] Simon Patience. Redirecting system calls in Mach 3.0: An alternative to the emulator. In *Proc. of the Third USENIX Mach Symposium*, pages 57–73, Santa Fe, NM, April 1993.

[26] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):287–338, December 1989.

[27] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Design rationale for Psyche, a general-purpose multiprocessor operating system. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 255–262, August 1988.

[28] Daniel Stodolsky, J. Bradley Chen, and Brian N. Bershad. Fast interrupt priority management in operating system kernels. In *Proc. of the Second USENIX Workshop on Micro-kernels and Other Kernel Architectures*, San Diego, CA, September 1993.

[29] Vrije Universiteit, Amsterdam, NL. *The Amoeba 5.0 Reference Manual: Programming Guide*, 1992. *rpc* manual page; ftp.cs.vu.nl:amoeba/manuals/pro.ps.Z.

Author Information

Bryan Ford is an undergraduate in Computer Science at the University of Utah. His current major research interest is improving Mach 3.0, but he pursues other interests, including data compression, languages, graphics, and music. He is the author of several widely used packages for the Amiga, including the XPK compression package and the MultiPlayer music program. Bryan is the designer and primary implementor of Mach migrating threads.

Jay Lepreau is Assistant Director of the Center for Software Science, a research group within Utah's Computer Science Department which works in many aspects of systems software. He has worked with Unix since 1979, and has served as co-chair of the 1984 USENIX conference and on numerous other USENIX program committees. His group has made significant contributions to the BSD and GNU software distributions. His current research interests include flexible system structuring, with operating system, language, linking, and runtime components.

The author's addresses are: Center for Software Science, Department of Computer Science, University of Utah, 84112. They can be reached electronically at {baford,lepreau}@cs.utah.edu.