Continuations: Unifying Thread Management and Communication in Operating Systems

Richard P. Draves Brian N. Bershad Richard F. Rashid
Randall W. Dean

March 1, 1991
CMU-CS-91-115 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Thread management and interprocess communication are two key operating system services. Within an operating system, these services can be unified by managing the state of a computation as a continuation, a first class object that can be explicitly examined and manipulated through a well-defined interface. Continuations as first class objects improve the performance of thread management and interprocess communication facilities, and can be used to generalize many optimizations that are common to operating systems. These optimizations can be recast in terms of a single, unifying implementation methodology. We have used continuations to redesign the internals of the Mach operating system at Carnegie Mellon University. On a DECstation 3100, our new system consumes over 85% less space per thread and executes a cross-address space procedure call 14% faster than earlier, heavily optimized versions. This paper describes the application of continuations to the Mach operating system.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035 and in part by the Open Software Foundation (OSF). Draves was supported by a fellowship from the Fannie and John Hertz Foundation.

Keywords: Operating Systems, Communication, Structure, Performance

1 Introduction

We have achieved significant improvements in the performance of the Mach operating system [Accetta et al. 86, Golub et al. 90] by redesigning it to use continuations as the basis for control transfers between execution contexts. A continuation represents the static and dynamic state of a computation as a first class object which can be examined and manipulated through a well-defined interface. Continuations have allowed us to unify the control transfer aspects of thread management and interprocess communication (IPC), significantly reduce the storage requirements of threads, and improve the performance of IPC facilities. On a DECstation 3100, for example, our new system consumes over 85% less space per thread and executes a cross-address space procedure call 14% faster than earlier, heavily optimized versions [Draves 90]. Additionally, continuations enable many common thread and IPC optimizations to be recast in terms of a single implementation methodology. We believe that this first class treatment of control transfer will yield similar results for other operating system kernels which, like Mach, rely on threads and interprocess communication to support distributed computing, multithreaded programming, and multiprocessing [Mullender et al. 90, Cheriton 88, Rozier et al. 88].

1.1 The Problem: Too Many Threads, Too Many Messages

Strictly speaking, continuation management is a fundamental responsibility of any operating system. System calls, synchronization, preemptive scheduling, interrupts, processor exceptions, and interprocess communication represent just a few of the ways in which computations stop and start in an operating system as control transfers between execution contexts. Operating systems have traditionally managed continuations *implicitly* using primitive operations that save and restore processor context in terms of machine-dependent kernel stacks and process control blocks.

The use of implicit continuations (kernel stacks) to manage control transfer can place a serious burden in terms of space and time on system resources. Newer operating systems and applications rely on large numbers of processes or threads, both as a software structuring tool and as a way to manage CPU and I/O parallelism. As the number of threads in a system grows, so too must the number of kernel stacks required to represent those threads' implicit continuations. Implicit continuations also make it difficult to take time-saving "shortcuts" when resuming a blocked thread. New system applications, such as database managers, file systems, and windowing systems, require frequent control transfers both internally (as they tend to be multithreaded) and externally (as they tend to be heavy clients of IPC facilities). The frequency with which threads transfer control to one another increases with the number of threads in the system and with the degree to which the system is decomposed [Bershad 90, Cook 78]. Consequently, overall system performance is directly affected by the high latency of using implicit continuations to transfer control between contexts [Anderson et al. 91].

We encountered these problems in early versions of the Mach kernel. Mach's support for multiple address spaces with multiple threads per address space encouraged the development of complex, multithreaded programs. This resulted in a situation where applications required hundreds of threads to be managed on a single machine. Additionally, Mach's "kernelized" approach, whereby traditional operating system features are implemented as user-level applications, increased the frequency of cross-address space control transfers relative to more monolithic operating systems such as Unix. For example, preliminary measurements of our multi-server operating system environment revealed that the frequency of cross-address space communication was at least three times as great as that of a system where the entire operating system was in a single address space.

These problems forced us to reconsider the use of implicit continuations when managing the transfer of control between execution contexts. Implicit continuations simply provided too little information as to the real control transfer needs of a computation and complicated our attempts to reduce storage costs and improve IPC performance. Dedicating a kernel stack to each thread – even a pageable kernel stack – placed too high a burden on system resources. Improving the performance of control transfer, and therefore IPC, required a better handle on the actual control transfer needs of computations than we had available with implicit continuations.

1.2 The Solution: Explicit Continuations

We have addressed these thread management and IPC performance problems by modifying the kernel to use continuations explicitly when transferring control between contexts. Implicit continuations (saved context on a kernel stack) have been replaced by *explicit* continuations that are implemented as first class kernel objects. These explicit continuations describe what a thread is supposed to do next in terms of a machine-independent interface and compact representation. As a result, explicit continuations may be examined and manipulated by other active computations within the kernel prior to their reactivation.

Explicitly managed continuations have enabled us to make performance optimizations that improve the efficiency of the operating system by reducing the space and time overheads of thread management and interprocess communication. We have identified two important runtime optimizations that specifically address the performance problems described above: *continuation compression*, which reduces the space required by a kernel thread, and *continuation recognition*, which reduces the latency required to resume a blocked thread, thereby improving the performance of IPC facilities. We have also found that many low-level optimizations associated with control transfer in operating systems can be recast in terms of explicit continuations. For example, handoff scheduling [Black 90b, Thacker et al. 88], stackless kernel threads [Thacker et al. 88], asynchronous I/O [Levy & Eckhouse 89], kernel-to-user upcalls [Hutchinson et al. 89, Anderson et al. 90, Scott et al. 89], and Lightweight Remote Procedure Call [Bershad et al. 90] each represent an optimization to IPC and thread management systems that can be described and implemented in terms of continuation management, compression and recognition.

Our experiences with the use of explicit continuations in Mach for both thread management and IPC has led us to conclude that continuations are a unifying mechanism for handling control transfer in an operating system. We have used continuations to handle a diversity of control transfers in the Mach kernel, and have been able to improve system performance in a large number of places by applying a small set of optimizations in a uniform way. Consequently, the improved performance that has resulted from using continuations demonstrates that this unifying approach is also a viable one.

This paper describes the use and performance of continuations in the Mach 3.0 operating system. In Section 2 we define and develop continuations as a model for control transfer in operating systems. In Section 3 we describe the implementation of continuations in Mach. We examine the performance improvements that result from using continuations and the optimizations they allow in Section 4. In Section 5 we show how continuations generalize many optimizations found in other operating systems. In Section 6 we discuss related work. Finally, in Section 7 we summarize and present our conclusions.

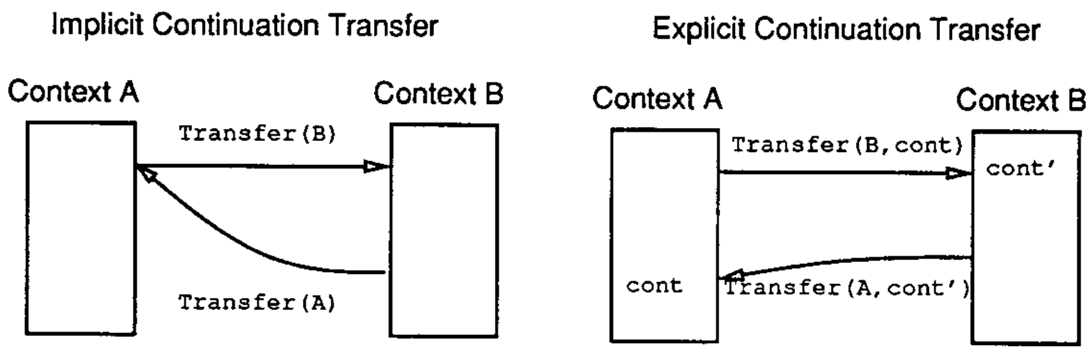
2 Continuations and Control Transfer

A continuation captures the state of a computation and saves it in a form callable as a function. When a continuation is called, the saved computation resumes with a restored environment. Unlike a normal function call, though, control does not return to the continuation's caller. Instead, a continuation call is like a "goto" for which the destination's address and context are both specified.

Continuations can form the basis of thread management in an operating system. Operations that block a computation, such as a system call, a preemption, a message-receive operation, a page-fault, or an exception, are equivalent to creating a continuation that will later be called when the computation resumes.

The language community has identified two types of continuations for managing control transfer within a program, programming language, and compiler [Steele 90]: statement continuations and expression continuations [Milne & Strachey 76]. A *statement continuation* enables a control transfer from a command statement in one execution context to a command statement in another. No information is passed as a result of the control transfer. In contrast, an *expression continuation* captures the evaluation context of an expression. The calling context specifies a value that is passed into the receiving context as the result of the expression.

We can divide continuations into two further subtypes: implicit continuations and explicit continuations. A continuation may be created implicitly by saving the current processor state when transferring control out of a computation; that state will be restored when control transfers back. In contrast, an explicit continuation is one that is specified by a computation as the context in which execution should resume when control returns to that computation. The difference between implicit and explicit continuations is illustrated by Figure 1. For an explicit continuation, the resumption context is specified as an argument to the transfer function. In our terminology, the `Save_Context`, `Restore_Context` operations found in most operating systems generate and call implicit continuations.



```
Transfer(B)
    Save_Context(A)
    Restore_Context(B)

Transfer(A)
    Save_Context(B)
    Restore_Context(A)
```

```
Transfer(B,cont)
    A.continuation = cont
    Transfer B.Continuation

Transfer(A,cont')
    B.continuation = cont'
    Transfer A.continuation
```

Figure 1: Transferring Control With Implicit and Explicit Continuations

Previous operating systems have used implicit continuations to describe the state of a suspended computation. After a call or trap into the kernel, low-level code saves the user's registers on the kernel stack as a kind of implicit continuation. As the system call makes its way through kernel code, the computation pushes call frames onto the kernel stack. If the computation blocks, a `Save_Context` operation saves the kernel's registers onto the kernel stack (just as was done when control transferred into the kernel). When the computation is resumed, a `Restore_Context` operation restores the kernel's computation so that it can eventually unwind its way back up the kernel stack to the low-level trap-handling code. This code restores the user's state in a manner similar to `Restore_Context` and resumes the user-mode computation. The saving and restoring of processor context on the stack correspond to the creation and calling of implicit continuations.

An explicit continuation embodies a much higher level representation of a computation's state, independent of the machine and compiler architecture, than does an implicit continuation. Although an implicit continuation captures the control and environment components of a suspended computation, its representation (registers and a kernel stack) reflects a computation's state at the machine level in terms of return addresses, saved registers and automatic variables. In contrast, an explicit continuation describes what to do next and in which context to do it in a machine-independent form, just as a procedure call represents a machine-independent way to transfer control within a single program. At this level, a continuation becomes a first class object, allowing it to be manipulated by the operating system just as any other first class object such as address space maps and message queues.

2.1 Continuation-Based Optimizations

The use of explicit continuations in an operating system admits two important control transfer optimizations: continuation compression and continuation recognition. These optimizations improve the performance of thread management and IPC operations while preserving the machine independence and internal structure of the operating system kernel (i.e., it is not necessary to expose private interfaces or resort to long assembly language paths to improve performance).

Continuation Compression

Because an explicit continuation specifies the context in which a thread is to be resumed, its use does not assume that the context active when a thread blocks will be restored when it resumes. It follows that there is no need to maintain that context while the thread is blocked. A continuation may, therefore, be replaced with an equivalent but smaller continuation while that thread is not active. We call this kind of continuation replacement *continuation compression.*

Continuation compression can result in savings of both space and time. It allows the kernel to save space by discarding a thread's kernel stack when that thread blocks with an explicit continuation. Moreover, if the next thread ready to run is represented by an explicit continuation, that thread can use the blocking thread's discarded stack to resume execution. We call the transfer of a stack directly from one thread to another a *stack handoff.*

Continuation Recognition

Time can be saved during a control transfer via a continuation call by recognizing when the call can be avoided, either because the work that would be done in the call can be done more simply, or because it need not be done at all. This optimization is called *continuation recognition* and it is made possible by having a machine-independent representation for explicit continuations. This representation can be examined and manipulated before it is called to allow a variety of special cases to be handled.

The stack handoff made possible by compression assists in the task of continuation recognition. A stack handoff from a thread t_1 to a thread t_2 gives t_2 a context in which to execute its own code. More importantly, the handoff allows t_1 to communicate pieces of its own context to t_2. When a thread t_1 blocks on a compressed continuation and hands off its stack to a thread t_2, t_2 can be resumed in the still-active procedure call context left behind by t_1. Within this context, t_2 may complete the control transfer by calling its previously stored explicit continuation, or it may transfer to an alternate, simpler continuation depending on the context inherited from t_1.

3 Using Continuations in Operating Systems

We have applied the two optimizations described above to the Mach operating system kernel with good results. The implementation of explicit continuations in Mach, while straightforward, was an object lesson in the value of having a small kernel to work with, careful interface design, and maintaining a strict division between machine independent and machine dependent code.

3.1 Creating Explicit Continuations

Whenever the operating system is involved in the transfer of control between contexts, a continuation must be generated. Control transfers can occur at the user/kernel boundary, or within the kernel when one thread transfers control to another. Traps, exceptions, and faults at the user level transfer control to entry points inside the kernel, where the kernel creates a continuation which will later be used to resume execution at user level. We distinguish between system calls, which cause a voluntary kernel entry, and exceptions and interrupts, which cause an involuntary entry into the kernel. System calls generate an expression continuation that is invoked with the return code for the system call. Exceptions and interrupts, which do not return values to user programs, generate a statement continuation that is invoked without arguments. The generated continuation, when called from within the kernel, returns control to the user level immediately; control does not return to the caller.

For in-kernel transfers, the kernel uses an explicit continuation whenever the current execution context can be compressed and the kernel stack discarded. A blocking kernel thread specifies its explicit continuation as a function which is passed as an argument to the kernel's thread blocking procedure. The machine independent data structure used to represent threads contains a field for the explicit continuation that can be examined by other threads. When the thread is resumed, it resumes in the specified continuation function. If the blocking thread must preserve any state, it must do so explicitly.[1] In the few cases where it is not possible (or not beneficial) to block with an explicit continuation, a null argument to the blocking function will create an implicit statement continuation (a machine-dependent representation stored on the kernel stack), precluding the compression and recognition optimizations.

3.2 Converting the Kernel to Use Explicit Continuations

Revising the Mach kernel to use explicit continuations internally was a straightforward process. First we identified all kernel procedures which could potentially block. We then separated each

[1]The thread structure contains a small scratch area large enough for 28 bytes of state. For anything larger, the computation must allocate a structure in which to preserve its environment.

procedure into two parts: one before the block and one after.[2] We defined a new procedure that consisted of the post-block, or continuing part of the original procedure, and left only the pre-block part in the original. Next, we manually identified the stack context that was common to the two parts, and modified the pre-block code to store that context in a data structure associated with the blocking thread. Similarly, we modified the post-block procedure so that it used the context in the data structure. In the pre-block procedure, we changed the call to the kernel's implicit blocking function to one that took the post-block procedure as an explicit continuation. Lastly, we changed the post-block procedure to invoke an explicit continuation on exit, rather than returning to its caller off of the stack. Figure 2 illustrates a sample transformation.

Before Explicit Continuations

```
generic_syscall(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block)  {
    /* implicit continuation */
    thread_block();
    P2(arg1);
  } else
    P3();
  /* return control to user */
  return SUCCESS;
}
```

After Explicit Continuations

```
generic_syscall(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block)  {
    /* save context in thread */
    thread_block(syscall_continue);
    /*NOTREACHED*/
  } else
    P3();
  /* return control to user */
  thread_syscall_return(SUCCESS);
}

syscall_continue() {
  /* recover context from thread */

  P2(recovered arg1);

  /* return control to user */
  thread_syscall_return(SUCCESS);
}
```

Figure 2: Transforming a Blocking Kernel Procedure

In most cases, we did not find it difficult to create an explicit continuation for blocking kernel functions. For user threads that trap into the kernel, the primary cases where blocking occurs are on message receives, exceptions and preemptions. Each occurs as a result of a user-to-kernel transfer (system call, exception or interrupt), and each, upon being handled, returns control immediately to the user level by way of the continuation that was created when control transferred into the kernel.

There is no return-to-user-level continuation for threads that run only in the kernel. In practice, though, all of our kernel threads execute an infinite loop, blocking until an event occurs, doing some work, and then blocking again. For these threads, we define the explicit continuation to be a procedure containing the body of the loop, thereby achieving the same result as if we had an infinite loop in a static context.

[2]For procedures that could block multiple times, the separation was repeated.

The Advantage of a Small-Kernel System

The task of changing the kernel to use explicit continuations was simplified by the fact that we started with a small kernel for which synchronization points were few and easily managed. The Mach 3.0 kernel supports only a few simple abstractions, and exports a small interface, so there are only a few potentially blocking calls. In all, there are about 60 places in the entire kernel where a thread can block, but, as we show in Section 4, over 99% of all blocks occur at only six places in the code. We focused our reorganization on those few "hot spots." There are still paths in the kernel where implicit continuations are used, making compression and recognition impossible, but they are traveled so infrequently that they have no effect on system performance.

Had we instead tried to apply continuations to a monolithic operating system kernel such as Mach 2.5, which implements the 4.2 BSD UNIX interface, our job would have been much more difficult. There are over 180 places in the Mach 2.5 kernel where a thread can block, and there are no real hot spots. Many of these blocks occur when a thread is deep within the kernel having made numerous nested procedure calls as a result of a system call, so the blocking calls have very complex continuations. In our estimation, it would have been extraordinarily difficult to have used explicit continuations in Mach 2.5 as generally and as uniformly as we have done in Mach 3.0.

3.3 Using Continuations for Cross-Address Space RPC

Message passing is the dominant kernel operation in Mach, and so it is the primary candidate for the continuation optimizations. Because so much Mach activity depends on message passing, all non-runnable user threads are normally blocked in some type of message receive operation.

To demonstrate how the Mach kernel manages continuations during message passing, we examine the sequence of events that occurs during a remote procedure call between two address spaces on the same machine. Figure 3 shows the fast path through the calling half of an RPC[3]; the fast path is taken when there are no errors or faults and the server's wait precedes the client's call.

A single system call, `mach_msg`, combines the sending and receiving phases of an RPC into one operation. A client thread, the caller, uses `mach_msg` to send an RPC request message to a server's port, and to receive a reply message from a reply port. A server thread, the callee, uses `mach_msg` to send a reply message on the client's reply port, and to receive the next request from its own port. In both cases, the sending thread wakes up a receiving thread and blocks itself, waiting for a message.

The handoff at the heart of the RPC path implements both the continuation compression and recognition optimizations. The RPC path does a scheduling handoff, exchanging the scheduling states of the sending and receiving threads, and a stack handoff. The stack handoff implements continuation compression, leaving the sending thread blocked with an explicit continuation, `mach_msg_continue`, and no kernel stack. The handoff changes the current thread to be the receiving thread, but it does not call the receiver's continuation. Instead, the kernel continues to run in the context of the sender's `mach_msg` system call. This aspect of the handoff enables continuation recognition, because `mach_msg` can check the receiver's continuation. If it is `mach_msg_continue` (because the receiver blocked in this same path), then `mach_msg` continues with the fast RPC path. Otherwise `mach_msg` calls the receiver's general continuation.

Continuation recognition improves performance because it brings together in a single context the sender's and receiver's message processing, allowing the two phases to be optimized together. Checks for exceptional conditions can be combined and redundant work can be eliminated. For

[3]The return phase is symmetric and works in the same way.

example, the fast RPC path avoids queueing and dequeueing the message, redundantly synchronizing (caller locks then unlocks; server locks then unlocks) and updating reference counts on message data structures.

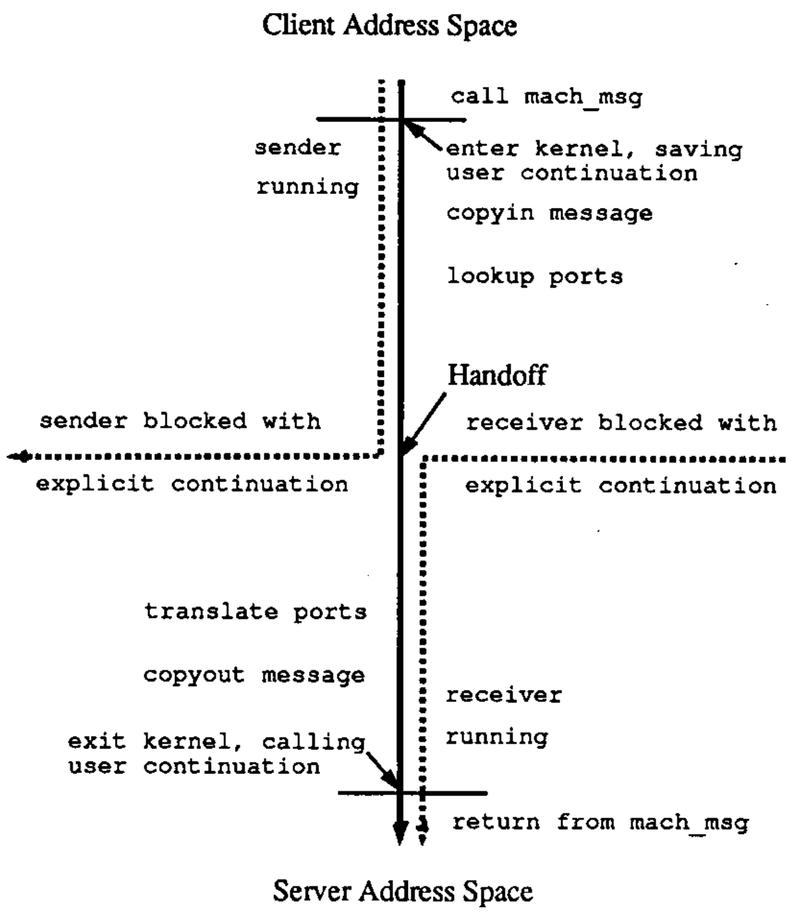


Figure 3: Calling Half of the Fast RPC Path

3.4 Using Continuations for Other Types of Control Transfers

In the Mach kernel, IPC is the most common cause of control transfer, so the use of explicit continuations on the IPC path has the greatest effect on system performance. Several other control transfer paths use explicit continuations as well:

- *Exception Handling*

  A user-level exception that cannot be handled by the Mach kernel, such as a divide-by-zero fault, causes the offending thread to block and the kernel to send a message to an exception handling port associated with that thread [Black et al. 88]. Compression allows the faulting thread and the exception handling thread to share the same stack, in the same manner that occurs with RPC. Unlike user-to-user RPCs though, the source of an exception message is the kernel. Creating a message in kernel space, however, and then copying it into the address space of the exception handler are expensive operations. Continuation recognition allows us to bypass these operations by recognizing an exception handling thread's general purpose receive continuation before invoking it. This allows us to create the message directly in the handler's address space (rather than in the kernel first), and to resume the handler thread in code that simply exits out of the kernel.

- *Preemptive Scheduling*

  The primary reason to use continuations for preemptive thread management is to reduce the rescheduling latency; by blocking with little kernel context, there is little context to be unwound upon unblocking. A user-level thread is preempted in Mach by forcing it into the kernel with an interrupt from a clock or some other external device, setting a low-priority asynchronous trap (AST), and returning to user level. At the kernel boundary, the returning thread catches the AST, discovers that it is being preempted, and relinquishes the processor. At the time a thread relinquishes its processor, its only critical state is that which was saved at the time of the interrupt, and is stored in an explicit continuation that can be called to return control to the user level. The stack context that the thread built up as it was fielding the AST becomes unimportant once the thread blocks, and can therefore be discarded. When rescheduled, the thread's explicit continuation can be called to resume the thread at user level.

Explicit continuations, compression and recognition are also used when threads page fault at the user level, and when threads voluntarily relinquish the processor.

3.5 Implementing Continuations in a Portable Operating System Kernel

The Mach 3.0 kernel runs on a variety of processor architectures. This portability is achieved by dividing the kernel into machine-independent modules, which implement the Mach kernel interfaces, and machine-dependent modules, which manage the hardware. The machine-independent modules manage scheduling, interprocess communication, and virtual-memory. The machine-dependent modules handle the memory management unit, the low-level trap and exception machinery, stack manipulation, and implicit continuations (the traditional context switch).

Using explicit continuations in the kernel required some additions to the internal machine-dependent interface. The interface allows the machine-independent thread management and interprocess communication modules to change address spaces, to manage the relationship of kernel stacks and threads, and to create and call continuations. The new operations are listed in Figure 4. The interface does not include any procedures for reading a thread's continuation; that operation is part of the machine-independent interface. (As mentioned in Section 3.1, a blocked thread's continuation function is part of the machine-independent thread data structure, and can therefore be examined directly by any other thread.)

Although continuations are a unifying model for dealing with control transfer, and should therefore have a uniform invocation method, they are poorly handled by C, the language in which Mach is written. It is difficult to generate and use continuations within the framework of C because the language does not support closures or execution contexts as first class callable objects. Despite the language deficiency, though, the interface and the underlying implementation demonstrate that it is possible to write C code that uses continuations. Appendix A demonstrates how these machine-dependent functions can be combined to implement several machine-independent, low-level thread management functions.

`stack_attach(thread, stack, cont)`
: Attaches the kernel stack to the thread and initializes the stack so that when **`switch_context`** resumes the thread, control transfers to the supplied continuation function with the previously running thread as an argument.

`stack_detach(thread)`
: Detaches and returns the thread's kernel stack.

`stack_handoff(new-thread)`
: Does a stack handoff, moving the current kernel stack from the current thread to the new thread. **`stack_handoff`** changes address spaces if necessary. **`stack_handoff`** returns as the new thread.

`call_continuation(cont)`
: Calls the supplied explicit continuation, resetting the current kernel stack pointer to the base of the stack. This function prevents stack overflow during a long sequence of continuation calls.

`switch_context(cont, new-thread)`
: Resumes the new thread's implicit continuation, switching to its kernel stack. This call changes address spaces if necessary. If a continuation for the current thread is supplied, then its kernel stack is discarded and **`switch_context`** doesn't return. Otherwise, **`switch_context`** saves the current thread's context and stack as an implicit continuation and returns when the calling thread is specified as an argument to a subsequent **`switch_context`**. This function returns the the previously running thread.

`thread_syscall_return(return-value)`
: Calls the current thread's user system call continuation to make the thread return to user space from a system call with the specified return-value. (Low-level machine-dependent trap code creates system call continuations).

`thread_exception_return()`
: Calls the current thread's user exception continuation to make the thread return to user space from an exception or page-fault. (Low-level machine-dependent trap code creates exception continuations.)

Figure 4: Kernel Continuation Interface

3.6 Summary

Continuations represent one more case where leverage and uniformity can be gained by promoting an operating system abstraction to a first class object. In this sense, explicit continuations are similar to Mach's *pmap* abstraction [Rashid et al. 87]. A pmap is a first class object that reflects a sequence of address mappings from virtual to physical memory. By encapsulating the abstraction of memory mapping in a first class object, and by separating the abstraction from its machine-dependent implementation (page tables and segment registers), the pmap can be used and optimized in ways that were not originally obvious or even possible [Young et al. 87]. Continuations as a first class kernel abstraction have yielded similar results. While our approach has resulted in substantial changes to the internals of the system, none of the changes have affected the kernel interface, and therefore none are visible to applications.

4 Performance

To evaluate the performance improvements made possible by the use of explicit continuations, we compare an older version of the Mach 3.0 kernel that doesn't use continuations against one that does. In brief, we show that:

- Almost all control transfers in the kernel take advantage of continuation compression.
- Most control transfers benefit from continuation recognition.
- Because continuation compression effectively changes kernel stacks from a per-thread resource to a per-processor resource, it significantly reduces the space overhead of thread management.
- Continuation recognition and compression reduce the latency of cross-address space communication and user-level exception handling.

Additionally, we describe the effect that each of the optimizations has on overall performance.

4.1 Experimental Environment

We compare two versions of the Mach 3.0 kernel, referred to internally as MK32 and MK40. The MK32 kernel predates the use of explicit continuations. The MK40 kernel uses explicit continuations as described in Section 3.[4]

Both kernels run on a DECstation 3100 and a Toshiba 5200/100. The DS3100 is a MIPS R2000-based workstation with separate 64K direct-mapped instruction and data caches and a four-stage write buffer. It has a 16.67Mhz clock and executes one instruction per cycle, barring cache misses and write stalls. The write buffer takes at least six cycles to process each write. Our DS3100 was configured with 16 megabytes of memory and a 250-megabyte Hitachi disk drive. The Toshiba 5200 is an Intel 80386-based laptop with a 20Mhz clock and a 32K combined instruction and data cache. Our Toshiba 5200 was configured with 8 megabytes of memory and a 100-megabyte Conner disk drive.

We ran the tests in an environment in which Unix system calls are implemented with RPCs to a Unix server running at user level [Golub et al. 90]. We also measured an MS-DOS emulation environment on the Toshiba 5200. The MS-DOS emulation uses the 80386's virtual-8086 mode. It implements privileged operations and MS-DOS system (BIOS) calls with a user-level exception handler that catches the faults resulting from native-mode operations. The exception handling thread runs in the address space of the emulated MS-DOS program.

4.2 Opportunities for Compression and Recognition

The value of the compression and recognition optimizations depend on the frequency with which they can be exercised. To determine this, we counted the number of blocking operations that could take advantage of the optimizations in three tests run on the Toshiba 5200 using the MK40 kernel. The first test measured a short C compilation benchmark. The second test measured a Mach 3.0 kernel build where all the files resided in AFS, the distributed Andrew File System [Satyanaranyanyan et al. 85]. The third test measured the MS-DOS program Wing Commander^{TM}. The short compilation and MS-DOS tests were run with the machine in single-user mode; the

[4]The MK40 kernel also implements optimization unrelated to continuations. We added the unrelated optimizations to the MK32 kernel reported here to ensure a fair comparison.

Toshiba 5200 running MK40 and Unix emulation

Operations Using Compression	Compile Test (22 secs) blocks	%	Kernel Build (4917 secs) blocks	%	DOS Emulation (698 secs) blocks	%
msg receive	3113	83.4	1391769	86.3	200167	55.2
exception	0	0.0	882	0.0	137367	37.9
page fault	34	0.9	3278	0.2	144	0.0
thread switch	0	0.0	114	0.0	4	0.0
preempt	288	7.7	78602	4.9	19101	5.3
internal threads	239	6.4	135756	8.4	5791	1.6
compression total	3674	98.4	1610401	99.9	362574	100.0
no compression	60	1.6	2117	0.1	7	0.0

Table 1: Frequency of Continuation Compression

Toshiba 5200 running MK40 and Unix emulation

	Compile Test count	%	Kernel Build count	%	DOS Emulation count	%
blocks	3734	100.0	1612518	100.0	362851	100.0
recognition	2247	60.2	1166449	72.3	311277	85.9
stack handoff	3614	96.8	1608320	99.7	362567	100.0

Table 2: Frequency of Continuation Recognition and Stack Handoff

kernel build was run in multi-user mode (AFS requires a user-level file cache manager). Table 1 summarizes the results.

The table shows that explicit continuations and compression are used for about 99% of all control transfers. The most frequent operations are message receive and exception handling. The other operations are page-fault handling, voluntary rescheduling [Black 90a], involuntary scheduling preemptions, and blocking by internal kernel threads. The remaining blocking operations (which do not use explicit continuations) include kernel-mode page faults, kernel memory allocation, and places where the kernel does short-term blocks waiting for critical sections to empty, or for a data structure to change state. These infrequent operations are still implemented with implicit continuations in MK40.

Over 60% of the blocking operations in the tests take advantage of continuation recognition, as shown in Table 2. The RPC and exception-handling paths use recognition to "shortcut" a general-purpose continuation call. In addition, nearly all control transfers use stack handoff instead of a traditional context-switch. Although we cannot quantify the effect, we believe that the context sharing that occurs when threads share a stack across a reschedule improves processor performance by reducing cache contention [Mogul & Borg 91].

4.3 The Effect of Continuation Compression

Continuation compression effectively changes the kernel stack into a per-processor, rather than per-thread, resource. For the three test applications, we measured the number of kernel threads and stacks used. Although the number of threads varied from 24 to 43, sampling the count at every clock interrupt revealed that all three tests averaged 2.0 stacks in use. The compile test and DOS emulation never used more than 3 stacks, and the kernel build never used more than 6 stacks. MK40 uses a stack for the currently executing thread and a stack for a special kernel thread that handles global stack allocation and can never discard its kernel stack.

On a DECstation 3100, continuation compression reduces the space overhead of thread management by 85%. Table 3 shows the minimum, average and maximum sizes of the per-thread data structures maintained by the kernel. The data structures potentially allocated for each thread consist of machine-independent state, machine-dependent state, and a kernel stack.[5]

	MK40			MK32		
	min	mean	max	min	mean	max
MI state	484	484	484	452	452	452
MD state	172	206	308	0	0	0
stack	0	0	0	336	3612	4432
total	656	690	792	788	4474	4884

Table 3: Thread Management Overhead on the DS3100 (in bytes)

The machine-independent state includes the thread structure and IPC data structures associated with each thread. The MK40 thread structure contains a 4 byte function pointer and a 28 byte scratch area used to store explicit continuations.

The machine-dependent state includes the user-level's saved register state. Space for floating-point state is allocated only if the thread uses floating-point. The average size calculated here is based on our observation that only about one in four threads relies on floating-point. Table 3 gives machine-dependent sizes for the DS3100; the corresponding sizes for the Toshiba 5200 are 100 bytes of general register state and 112 bytes of floating-point state. Because the MK32 kernel hides the machine-dependent state at the base of the kernel stack, we calculate the size of that state to be 0 bytes.

Pageable Kernel Stacks

The per-thread kernel stack overhead includes the stack pages and their supporting VM data structures. The DS3100 and Toshiba 5200 both use 4K kernel stacks, but an additional 116 bytes of VM data structures are required in MK32 to make the stacks pageable. When the stack of an idle thread is actually paged out, an additional 220 bytes of VM-related data structures per thread are required. A non-resident stack therefore consumes 336 bytes. However, periodic sampling during day-to-day use of the MK32 system revealed that over 90% of the kernel stack pages remain resident, even when the system pages other memory; most threads run often enough that their stacks do not become eligible for pageout.

[5]We only consider the direct cost of resident in-kernel data structures and do not consider the memory usage resulting from user-level activities.

Pageable kernel stacks have other hidden costs as well. Because kernel stacks in MK32 are potentially paged, they must be allocated from virtual memory. The MK40 kernel, when possible, takes advantage of the fact that it is not necessary to page kernel stacks (since there are so few of them), and saves a TLB entry by placing kernel stacks in unmapped physical memory.

4.4 The Effect Of Continuation Recognition

Continuation recognition improves the performance of the RPC and exception paths. We can show this with two simple tests. The RPC test measures the round-trip time for a cross-address space null RPC, which sends a minimal length message in each direction and executes a minimal amount of user code. The exception test measures the time for a user-level exception server thread to handle a client thread's exception. The exception server thread runs in the same address space as the client thread and it does not examine or change the state of the client thread, so the client retakes the exception. The average times for an RPC and an exception are shown in Table 4.

	DS3100			Toshiba 5200		
	MK40	MK32	Mach 2.5	MK40	MK32	Mach 2.5
null RPC	95	110	185	535	510	890
exception	135	425	380	525	1155	1410

Table 4: RPC and Exception Times (in μsecs)

The MK32 RPC path was already highly optimized relative to Mach 2.5 [Draves 90], and yet compression and recognition achieve an additional 15 μsecs reduction in latency on the D53100. The Toshiba 5200's RPC latency increased slightly because the machine-dependent code in MK40 implements stack handoff inefficiently. The low-level trap handler saves user register state on the kernel stack, and the machine-dependent stack handoff procedure must copy the current thread's state from the stack and copy the new thread's state onto the stack. We are fixing this and expect that the Toshiba 5200 times will improve by approximately 50 μsecs.

To understand the source of the improvement in RPC latency, we counted instructions, loads, and stores for each component of the total RPC path, as shown in Table 5. In this case, the performance gain comes from doing a stack handoff instead of a context-switch. Continuation recognition provides only enough performance benefit to offset the cost of saving and restoring state with an explicit continuation.

Although the MK40 path uses 21% fewer instructions, it is only 14% faster. The reason for this discrepancy is that the R2000's write buffer limits the performance of the MK40 path; its 212 stores (at 6 cycles per store) must take at least 1272 cycles.

The use of explicit continuations in MK40 slightly increases the work done at system call entry and exit relative to MK32. The continuation for the user computation contains the callee-saved registers; system call entry saves these registers for blocking system calls and `thread_syscall_return` restores them. The MK32 system call code doesn't save and restore these registers because the normal C calling conventions save and restore them on the kernel stack. Instead, the MK32 context-switch saves and restores these registers. Because the majority of potentially blocking system calls do block, though, little effort is wasted by saving these registers at kernel entry.

The MK40 path also saves an explicit continuation before doing a stack handoff and recovers state after the handoff. As a side-effect of the stack handoff, some of the other work involved

	MK40			MK32		
	instrs	loads	stores	instrs	loads	stores
request path						
syscall entry	64	7	25	67	8	20
msg copyin	41	6	6	41	6	6
sender	180	50	28	185	47	26
handoff or csw	83	22	18	250	52	27
receiver	149	53	20	139	46	15
msg copyout	41	6	6	41	6	6
syscall exit	35	21	1	24	11	1
reply path						
syscall entry	64	7	25	67	8	20
msg copyin	41	6	6	41	6	6
sender	164	41	27	173	41	25
handoff or csw	83	22	18	250	52	27
receiver	105	40	15	96	34	10
msg copyout	41	6	6	41	6	6
syscall exit	35	21	1	24	11	1
user space						
client code	21	3	5	21	3	5
server code	20	4	5	20	4	5
total	1167	315	212	1480	341	206

Table 5: RPC Component Costs on the DS3100

moves from the sending side of the handoff to the receiving side. Because continuation recognition lets the receiving side avoid work, though, the total instruction cost of the sending and receiving components remains roughly unchanged.

5 Generalizing Previous Optimizations with Continuations

Continuations are a good framework with which to realize many of the optimizations found in other operating systems. To illustrate this point, we can compare Mach's continuation-based RPC described in Section 3.3 to the control transfer aspects of Lightweight Remote Procedure Call (LRPC) [Bershad et al. 90].

LRPC is a high-performance interprocess communication facility designed for the common case of cross-address space (same machine) procedure call. Part of LRPC's good performance is due to the fact that threads can cross address space boundaries. A thread in the caller's address space traps into the kernel, but returns to the server's address space where it begins executing the server stub immediately. Upon return, the caller's thread traps back into the kernel from the server's address space and transfers back into the caller's address space at the instruction following the trap. The primary performance advantage of the single thread approach is that message queueing and scheduling can be avoided entirely on the fast LRPC path, since all work is being done in the context of a single thread.

Mach's continuation-based RPC achieves many of the same performance advantages that LRPC does: no queueing, no scheduling, and sharing a kernel stack between the caller and the callee. Further, continuation-based RPC maintains the logical separation between a client's thread and a server's.

A natural extension to the the continuation model (but one which we have not yet implemented) will allow us to completely mimic the LRPC transfer protocol. Presently, when a Mach thread traps into the kernel, it generates an implicit continuation which will transfer control back to the user-level context in which the trap occurred. We are considering extending the IPC interface so that a thread can register an explicit continuation for system call returns. This will allow a server thread to return directly to its stub procedure, bypassing the dispatch machinery that is part of our RPC system [Draves et al. 89].

With the ability to return out of the kernel to a context other than that which was active at the time the kernel was entered, explicit continuations can be used to implement a rich collection of control transfer mechanisms in a general way. For example, the upcalls required by the "x"-kernel [Hutchinson et al. 89] and Scheduler Activations [Anderson et al. 90] can be implemented by keeping a pool of blocked threads in the kernel, each with a default "return-to-user-level" continuation. To perform an upcall, we need only replace the default continuation with one that transfers control out of the kernel to a specific address at user level. Asynchronous I/O [Levy & Eckhouse 89] would behave in a similar fashion; on scheduling an asynchronous I/O, a thread would provide the kernel with a continuation to be called when the I/O completes.

6 Related Work

The language community has been experimenting with continuations for almost two decades. Ward used continuations to define the primitives of a message passing algebra called mu-calculus [Ward & Halstead 80] and showed how all control transfer could be expressed in terms of that algebra. Lampson [Lampson et al. 74] described a generalized control transfer interface based on continuations for an early version of the Mesa programming language [Geschke et al. 77]. A much restricted form of that transfer interface later appeared in the cross-address space RPC implementation for Taos, an operating system designed for the Firefly, DEC SRC's experimental multiprocessor workstation [Thacker et al. 88].

Functional languages that support concurrent execution and explicit continuations have been successful in implementing the former in terms of the latter [Haynes & Friedman 84, Wand 80, Cooper & Morrisett 90]. These efforts, however, have concentrated on control transfer at user level between contexts in the same address space. Additionally, functional languages often use non-contiguous data structures to implement procedure call stacks, partially reducing the incentive to perform compression (a large portion of a kernel thread's discardable state is the unused stack space below the bottom-most active call frame).

Blocking operations that take an explicit continuation argument are an example of continuation-passing-style (CPS) programming [Steele 78, Appel & Jim 89], which was originally developed as a compiler technique. A program written in CPS replaces normal control-flow constructs such as loops and gotos with tail-recursive function calls that take a continuation argument. The compiler converts the program to CPS and then concentrates on optimizing function calls, the sole remaining transfers of control.

7 Future Work and Conclusions

Our work with continuations in Mach is ongoing. We are presently experimenting with continuations at the application level within the context of our user-level threads package [Cooper & Draves 88]. It is our expectation that compression and recognition will result in benefits at the user level analogous to those achieved inside the kernel. Specifically, for applications that do their own user-level scheduling and synchronization [Bershad et al. 88, Anderson et al. 89, Weiser et al. 89], we expect that explicit continuations will reduce the space and time overheads normally associated with large numbers of user-level threads [Dean et al.].

We are not the first to recognize the power and flexibility of continuations as a mechanism for describing and implementing the transfer of control between contexts. The novelty of our work lies in the fact that we have been able to successfully apply continuations as a unifying model of control transfer in a general-purpose operating system kernel. The use of continuations as the basis for control transfer has allowed us to implement new optimizations, and to to recast several optimizations found in in other operating systems in terms of a single abstraction. As a result of this, we have achieved substantial improvements in system performance.

We believe that the methodology and techniques that we have described in this paper can be applied to other operating system kernels to achieve results similar to our own. We invite the reader to examine our system by obtaining the sources for the Mach 3.0 kernel via anonymous ftp from cs.cmu.edu.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Anderson et al. 89] Anderson, T. E., Lazowska, E. D., and Levy, H. M. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[Anderson et al. 90] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy., H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. Technical Report 90-04-02, University of Washington, Department of Computer Science and Engineering, April 1990. Submitted for publication.

[Anderson et al. 91] Anderson, T., Levy, H., Bershad, B., and Lazowska, E. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 1991.

[Appel & Jim 89] Appel, A. W. and Jim, T. Continuation-Passing, Closure-Passing Style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, January 1989.

[Bershad 90] Bershad, B. N. *High Performance Cross-Address Space Communication.* PhD dissertation, University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195, June 1990.

[Bershad et al. 88] Bershad, B. N., Lazowska, E. D., and Levy, H. M. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[Bershad et al. 90] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. Also appeared in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Black 90a] Black, D. L. *Scheduling and Resource Management Techniques for Multiprocessors.* PhD dissertation, School of Computer Science, Carnegie Mellon University, July 1990.

[Black 90b] Black, D. L. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer Magazine*, 23(5):35–43, May 1990.

[Black et al. 88] Black, D. L., Golub, D. B., Rashid, R. F., Avadis Tevanian, J., and Young, M. W. The Mach Exception Handling Facility. Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, April 1988.

[Cheriton 88] Cheriton, D. R. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[Cook 78] Cook, D. *The Evaluation of a Protection System.* PhD dissertation, Cambridge University, Computer Laboratory, April 1978.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C-Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[Cooper & Morrisett 90] Cooper, E. C. and Morrisett, J. G. Adding Threads to Standard ML. Technical Report 186, School of Computer Science, Carnegie Mellon University, December 1990.

[Dean et al.] Dean, R., Bershad, B. N., Draves, R. P., and Rashid, R. Using Continuations to Improve the Performance of User-Level Thread Management. Technical Report *In Preparation.*, School of Computer Science, Carnegie Mellon University.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the the First Mach USENIX Workshop*, pages 101–121, October 1990.

[Draves et al. 89] Draves, R. P., Jones, M. B., and Thompson, M. R. MIG — The MACH Interface Generator. Technical Report Unpublished manuscript available from the School of Computer Science, School of Computer Science, Carnegie Mellon University, July 1989.

[Geschke et al. 77] Geschke, C., Morris, J., and Satterthwaite, E. Early Experiences with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Haynes & Friedman 84] Haynes, C. T. and Friedman, D. P. Engines Build Process Abstractions. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 18–23, August 1984.

[Hutchinson et al. 89] Hutchinson, N. C., Peterson, L. L., Abbott, M. B., and O'Malley, S. RPC in the x-Kernel: Evaluating New Design Techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, December 1989.

[Lampson et al. 74] Lampson, B. W., Mitchell, J. G., and Satterthwaite, E. H. On the Transfer of Control Between Contexts. In *Lecture Notes On Computer Science: Proceedings of the Programming Symposium*, pages 181–203. Springer-Verlag, 1974.

[Levy & Eckhouse 89] Levy, H. M. and Eckhouse, R. H. *Computer Programming and Architecture: The VAX-11 (2nd Edition).* Digital Press, 1989.

[Milne & Strachey 76] Milne, R. and Strachey, C. *A Theory of Programming Language Semantics.* Halsted Press, New York, 1976.

[Mogul & Borg 91] Mogul, J. and Borg, A. The Effect of Context Switches on Cache Performance. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 1991.

[Mullender et al. 90] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44–54, May 1990.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.

[Rozier et al. 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Giend, M., Guillemont, M., Herrmann, F., Leonard, P., Langlois, S., and Neuhauser, W. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

[Satyanaranyanyan et al. 85] Satyanaranyanyan, M., Howard, J., Nichols, D., Sidebotham, R., and Spector, A. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, December 1985.

[Scott et al. 89] Scott, M. L., LeBlanc, T. J., and Marsh, B. D. Evolution of an Operating System for Large-Scale Shared Memory Multiprocessors. Technical Report 309, University of Rochester, School of Computer Science, March 1989.

[Steele 78] Steele, Jr., G. RABBIT: A Compiler for SCHEME. Technical Report TR 474, MIT AI Lab, May 1978.

[Steele 90] Steele, G. L. *Common Lisp. Second Edition.* Digital Press, 1990.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Wand 80] Wand, M. Continuation-Based Multiprocessing. In *Conference Record of the 1980 LISP Conference*, pages 19–28, August 1980.

[Ward & Halstead 80] Ward, S. A. and Halstead, Jr., R. H. A Syntactic Theory of Message Passing. *Journal of the ACM*, 27(2):365–383, April 1980.

[Weiser et al. 89] Weiser, M., Demers, A., and Hauser, C. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.

[Young et al. 87] Young, M., Tevanian, Jr., A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

A Using the Continuation Interface

Figure 5 illustrates how the continuation interface presented in Section 3.5 is used to implement higher level thread management operations. The `thread_handoff` call forms the basis of continuation recognition. It performs scheduling and stack handoff operations, and returns control running as the new thread, but does not call the new thread's continuation. This gives `thread_handoff`'s caller a chance to do continuation recognition by inspecting the continuation and bypassing it with special purpose code. In contrast, the `thread_block` call picks a new thread to execute. If the new thread has an explicit continuation and `thread_block`'s caller has provided an explicit continuation, then `thread_block` can take advantage of the more efficient `stack_handoff` path. Otherwise it must use `switch_context`. This path requires special care because the old thread's stack cannot be freed or the old thread returned to the run queues while still running on the old thread's stack. Therefore `switch_context` returns the previously running thread to the new thread, so the `thread_dispatch` of that previously running thread happens after the `switch_context`.

```
thread_handoff(cont, new_thread) {
    /* new_thread is waiting (not on run queues) */

    old_thread = current_thread();
    stack_handoff(new_thread);
    old_thread->cont = cont;

    /* old_thread left waiting */
}

thread_block(cont) {
    /* stop running the current thread */

    old_thread = current_thread();
    new_thread = pick thread from run queues;

    if (new_thread->cont) {
        if (cont) {
            stack_handoff(new_thread);
            /* now current_thread() == new_thread */

            old_thread->cont = cont;
            if (old_thread is still runnable)
                return old_thread to run queues;

            call_continuation(new_thread->cont);
            /*NOTREACHED*/
        } else {
            allocate stack;
            stack_attach(new_thread, stack, thread_continue);
        }
    }

    thread_dispatch(switch_context(cont, new_thread));
}

thread_continue(old_thread) {
    cur_thread = current_thread();
    thread_dispatch(old_thread);
    (*cur_thread->cont)();
    /*NOTREACHED*/
}

thread_dispatch(thread) {
    if (thread->cont) {
        stack = stack_detach(thread);
        free stack;
    }
    if (thread is still runnable)
        return thread to run queues;
}
```

Figure 5: Using the Continuation Interface