Unifying the User and Kernel Environments

Richard P. Draves
Scott M. Cutshall

March 12, 1997

Technical Report
MSR-TR-97-10

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Unifying the User and Kernel Environments

Richard P. Draves and Scott M. Cutshall

Microsoft Research
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract

Vendors of commercial operating systems today invest resources in two very different environments—one for user-level (application or server) programming and one for kernel-level (device driver or subsystem) programming. The kernel environment is typically more restrictive, with completely different interfaces and programming conventions.

Based on our experience developing and deploying an operating system for an interactive TV system, we believe that it is desirable to unify the user and kernel environments. We structured our operating system to provide common programming and run-time environments for user-level code and kernel-level code. For example, modules loaded into the kernel did not have any restrictions on their use of shared libraries or stack space. We found that unifying the user and kernel environments with common interfaces and run-time requirements, common documentation, and a common debugger produced significant software-engineering benefits. In addition, we could transparently collocate trusted server code in the kernel to reduce memory usage and improve performance.

1. Introduction

We advocate structuring operating systems to unify the user and kernel environments. The operating system should present a single environment, with common interfaces or APIs, common run-time characteristics for scheduling, paging, stack usage, and shared libraries, common debugger and development tools, etc. An operating system that does this automatically supports transparent kernel collocation—the ability to load trusted server binaries into the kernel or run test programs inside the kernel. This unification gives two benefits. From a software-engineering perspective, programmers only have to learn and keep up with one environment and companies only have to develop, maintain, and support one environment. From a performance perspective, one can load trusted code together in the kernel to short-circuit communication and eliminate process overheads.

This paper derives from our experience developing the Rialto kernel and deploying it in an interactive TV system. Microsoft's interactive TV trial with NTT, currently operational in Japan, is running Rialto on Pentium-based set-top boxes manufactured by NEC. Rialto also runs on MIPS R3000 development boards. Device drivers, file systems, network stacks, and other kernel-level subsystems are separately loaded binaries, written using the same interfaces, headers, libraries, and debugger as user-level system code. Even application-level interfaces are available inside the kernel. The kernel environment has no special restrictions on preemptibility or scheduling, the use of shared libraries, or stack space. Device drivers use a few additional kernel-level system calls for managing interrupts, physical addresses, and DMA. Trusted services are developed and debugged in a user process environment and then loaded into the kernel. From a command prompt, test programs can either run in a user process, or load into the kernel, execute, and unload.

We found that this design produced a smaller, simpler system with tangible software-engineering benefits. For example, one technical writer produced a single manual documenting user-level and kernel-level system programming. A two-person test group produced a single test suite for the system. Developers only had to learn one environment. In representative application scenarios, kernel collocation reduced CPU time up to 35% and saved more than 900K bytes, and our dynamic stack management techniques reduced stack memory consumption by a factor of three. The CPU overhead for supporting dynamically growable kernel stacks was 2-4% in our Intel implementation. By implementing a software TLB miss handler for the Pentium and simulating the MIPS virtual memory system, we estimated that our implementation increased the number of MIPS hardware TLB misses by 3%.

To achieve a unified run-time environment for user-level and kernel-level code, we had to solve several technical problems. We invented techniques for safe, efficient, and transparent access to system call and remote procedure call arguments inside the kernel, for sharing library code pages across the user/kernel boundary without using position-independent code, and for providing dynamically growable and pageable kernel stacks.

The rest of the paper is structured as follows. The next section examines related work, particularly the Multics and Chorus systems. Section 3 describes several development examples drawn from other systems to help motivate our approach. Section 4 discusses whether it is reasonable to use common interfaces for user-level and kernel-level system programming. Section 5 examines the technical problems that we solved and Section 6 describes in more detail relevant aspects particular to our implementation. Finally, Section 7 discusses our performance measurements and section 8 concludes.

2. Related Work

Many previous systems have addressed our goals to some degree. For example, the Multics system presented a single environment for application programming and system programming. The Chorus system and some versions of Mach support kernel collocation. “Small kernel” systems move all programming into user-mode, eliminating the kernel programming environment altogether. Some extensible kernels support the safe injection of code into the kernel address space.

The Multics system [Organick 72] was designed to create a single environment for all programs, including device drivers. Multics ran on special-purpose hardware, the GE 645. Processes consisted of a collection of segments, and segments were normally shared between processes with per-segment access control. The segments in a process belonged to different rings of protection, with inner segments protected from outer segments. Supervisor segments, which contained “core” operating system functionality, were part of every process and belonged to ring 0. A very few “master-mode” supervisor segments could execute privileged GE 645 instructions. The segments to which one had access determined the programming environment; there was no binary distinction between user mode and kernel mode. “It is worth reemphasizing that the only differentiation between Multics system programmers and user programmers is embodied in the access control mechanism which determines what on-line information can be referenced; therefore, what are apparently two groups of users can be discussed as one.” [Corbato et al. 72]

Although Multics achieved our software-engineering goal of a unified programming environment, it did so with a hardware base and software organization that is now thirty years old. Our contribution is to revive the idea and demonstrate that it is practical with modern hardware and software.

The Chorus system [Armand 91, Rozier et al. 92] supports the concept of kernel collocation via supervisor tasks. Supervisor tasks execute in privileged mode; they share the kernel’s address space. Chorus can short-circuit IPC between supervisor tasks, although this is not transparent to the code involved. As a result, a program can be built to run as a supervisor task and communicate more efficiently with the kernel and other supervisor tasks. Supervisor tasks also have access to additional APIs for managing interrupts and low-level I/O. Some versions of Mach [Lepreau et al. 93, Condict et al. 94] also support kernel collocation to allow privileged subsystems like the Unix server to run in the Mach kernel.

Our system builds upon the kernel collocation ideas from Chorus and Mach to unify user-level and kernel-level programming. In contrast to the Chorus and Mach systems, in our system any interfaces or APIs available at user level are also automatically and transparently available to components loaded in the kernel. Our transparency allows collocation without recompiling or relinking binaries. We support normal preemption and scheduling inside the kernel, sharing libraries between user mode and kernel mode, and dynamically growable kernel stacks.

Micro-kernel or small message-passing systems solve the software-engineering problem of two programming environments by eliminating the kernel programming environment. Device drivers, file systems, protocol stacks, subsystems, etc. are all implemented in user-level processes. The “small kernel” operating system design has a long history; we only mention two recent examples. L4 [Liedtke 95] takes this approach, motivated by modularity, common interfaces, and flexibility. A device driver process maps I/O ports and the kernel converts device interrupts into IPC messages. The exo-kernel [Engler et al. 95] removes resource management policy and mechanism from the kernel, to the greatest extent possible. The kernel exposes hardware resources in a secure manner and vectors interrupts to user processes. In contrast to these systems, our work allows trusted components to be transparently collocated in the kernel address space for improved performance.

Some extensible operating systems protect against misbehaving or malicious code that has been loaded into the kernel. For example, Spin [Bershad et al. 95] accomplishes this for kernel extensions written in a version of Modula-3. However, this approach increases the gap between the user and kernel programming environments. The extensions are typically small pieces of code that modify the kernel's policies or mechanisms and not complete subsystems. Our system does not provide any protection against misbehaving components that have been loaded into the kernel, but they can be debugged with a familiar debugger or debugged in a user process environment.

3. Software-Engineering Examples

Many other systems provide examples of situations that would have benefited if the user and kernel environments had been unified. For instance, during the development of the Andrew File System (AFS), the Windows NT Win32 subsystem, and the DCE RPC runtime, a unified user/kernel environment would have yielded software-engineering improvements.

There were several reasons for moving AFS's user-level cache manager into the kernel [Kazar 97]. They included improved performance, improved semantics (for example, by using internal kernel interfaces it was possible to check file versions in the read system call path and it was possible to fetch file chunks instead of performing whole-file transfer), and easier porting. Differences in Unix vendor kernel-level APIs, which are not standardized as well as user-level APIs, defeated the goal of easier porting. Development in the kernel was more difficult; for example, the Solaris kernel debugger was an annoyance. The Coda project, an AFS descendant, continued to use a user-level cache manager because of the ease of development and debugging. Coda was able to mitigate the performance impact by putting a small name and symbolic link cache in the in-kernel VFS driver [Steere et al. 90].

In early versions of Windows NT [Custer 92], Microsoft implemented the Win32 GUI subsystem as a user-level process. With version 4.0, Microsoft moved the GUI subsystem into the kernel address space to improve performance. The differences between the two environments, particularly the semantic differences between system calls and IPC, necessitated a partial redesign of the GUI subsystem. It took ten months to stabilize the resulting system, longer than any previous NT release, although many other factors (for example, switching to the Windows 95 user interface) also contributed to this delay [Cutler 97].

The Open Software Foundation's DCE RPC runtime lives in each user process that uses RPC. The OSF source code uses conditional compilation to build a separate version for the OSF kernel. Because of differences in the kernel environment, the kernel implementation does not support TCP and it uses a separate helper program for authentication [Salz 97]. Developing, maintaining, and testing a separate kernel version adds software-engineering costs. Microsoft's DCE RPC is also a user-mode shared library. The Microsoft implementation does not support the NT kernel environment. In contrast, the Rialto port of the Microsoft RPC runtime operates identically in both user mode and kernel mode.

4. Unifying User-Level and Kernel-Level Interfaces

Although a unified user/kernel programming environment might yield software-engineering benefits, perhaps in practice the kernel environment supports performance-critical subsystems whose special needs dictate special interfaces. This was not our experience with Rialto, but to date Rialto has only been used for interactive TV so our experience may not translate to systems like Unix or Windows NT. This section discusses whether it is feasible to unify the user-level and kernel-level systems programming interfaces in a general-purpose operating system.

The kernel environment in the first versions of Unix [Ritchie & Thompson 78] was substantially simpler than one finds in commercial systems today. Binaries could be loaded into new processes but not into the kernel address space; device drivers were compiled into the kernel image. Kernel code and data were not paged or swapped, kernel code was not preemptible, and the kernel did not support multiprocessors or multithreading. The kernel environment then was very different from the user environment, and much simpler. Over time the kernel environment of Unix and its competitors has increased in complexity, and now one finds support for dynamically-loaded drivers and subsystems, threads, preemption, and pageable code and data inside the kernel. As the kernel environment approaches the functionality of the user environment, it begins to make sense for the two environments to use common interfaces for common functionality like synchronization, memory management, thread management, etc.

Today there is a large overlap in the functionality available at user level and kernel level, but the specific interfaces have very little in common across the two environments. For example, Tables 4–1 and 4–2 summarize the situation for heap memory management, synchronization, file I/O, and network I/O in Windows NT and Unix.

Table 4-1: Windows NT Interfaces

User-mode APIs	Kernel-mode APIs
LocalAlloc/Free	ExAllocate/FreePool
Critical sections, events	Nt versions, spinlocks
Open/Read/Write/CloseFile	Devices, IRPs
winsock	AFD, TDI, NDIS

Table 4-2: Unix Interfaces

User-mode APIs	Kernel-mode APIs
malloc/free	kmalloc/kfree
mutex, condition	sleep/wakeup
open/read/write/close	VFS, vnode interface
sockets	mbufs interface

For more complex or data-intensive interfaces, like file and network I/O, can a single interface serve the needs of both user-level code and presumably more demanding kernel-level subsystems, like distributed file systems? The interface between a file system and a network protocol stack might take advantage of the lack of address space boundaries to pass complex arguments like linked lists of buffers, thus avoiding data copies. Two modules in a protocol stack may require an elaborate internal interface for efficient communication.

There are two responses to this line of reasoning. First, it is possible to design interfaces that allow data movement across address space boundaries without copies (for example, [Druschel & Peterson 93]). In an ideal world we would design efficient interfaces that worked well both within an address space and across address space boundaries. Second, in the real world it is feasible to create separate “internal” and “external” interfaces, where the internal interface assumes collocated callers and callees. At binding time performance-sensitive modules inquire if the internal interface is available and otherwise fallback to using the external interface. (The Rialto RPC mechanism supports this type of query.) In the network example, the internal interface would pass mbufs and the external interface would use caller-allocated buffers, but if a file system and a network protocol stack were loaded together—either in the kernel or in a user process—they could take advantage of the more efficient mbufs-style interface. This solution is not ideal, but at least both interfaces are available inside the kernel and inside user processes.

4.1. Device Drivers

A limited amount of support for device drivers, and in particular for interrupt handlers, represents a special case for which we do not advocate unifying the user-level and kernel-level environments. This subsection considers whether it is possible to write efficient device drivers with the addition of a few specialized APIs that are restricted to kernel-mode use. This is certainly true in principle; for example, our system supports device drivers with eight additional APIs that fail when called from user mode. It is not obvious that a device driver model for a general-purpose commercial operating system, with significant additional functionality, could be similarly pared down.

Most device drivers need APIs for installing and synchronizing with first-level interrupt handlers. Because a first-level interrupt handler normally executes in the kernel’s context, it is difficult to make these APIs available to user processes. Furthermore, first-level interrupt handlers typically operate under severe restrictions related to preemptibility and interruptibility; their run-time environment is very different from that of thread-level code.

Device drivers also need to manage physical memory. For example, DMA may require physical addresses or the driver may need to map device memory. The APIs for pinning pages in memory, converting virtual addresses to physical addresses, or mapping device memory can potentially be made available to privileged user processes, but one might reasonably choose to restrict this functionality to kernel-level modules.

That was the extent of the special-purpose device-driver support in our system, but “industrial strength” device driver interfaces like Microsoft’s Windows Driver Model (WDM) [Baker 95] supply important additional functionality that is available only in the kernel environment. The functionality is certainly important, but possibly it could be supplied in a more general fashion. As examples, we discuss several areas of WDM functionality and consider how they might be designed differently.

I/O Request Packets (IRPs) support asynchronous processing in drivers. IRPs and their associated helper functions create conventions that drivers can use to pass work off to each other and signal completion. This support for asynchronous processing should not be restricted to the kernel environment. If it were made available more generally, user-level applications like database or web servers could take advantage of it also.

Deferred Procedure Calls (DPCs) allow first-level interrupt handlers to defer work that need not be performed in an interrupt context. In Windows NT, DPCs are interruptible but they execute in a context that preempts thread-level processing with many of the same restrictions as interrupt handlers. As an alternative, a lightweight context-switch (no need to switch address spaces or restore registers) from an interrupt handler to a kernel worker thread could give DPCs a thread-level execution environment for a small performance penalty.

Plug 'n' play (PNP) support creates a framework for coping with a dynamically changing device environment. This includes a small database that tracks the current hardware environment, software/hardware conventions for device discovery, dynamically loading and unloading drivers, etc. The hardware aspects of this functionality are specific to drivers and should be encapsulated in a driver support library, but much of it could be generalized to assist applications in dealing with dynamic environments.

5. Impact on Operating System Structure

The creation of a common user/kernel run-time environment imposes certain requirements on operating system structure. This section discusses problem areas and possible solutions, identifying the techniques that we used in Rialto. The requirements fall into several areas. For example, the run-time environment in the kernel should not impose any restrictions on the availability of shared libraries. The preemptibility or scheduling of kernel code and user code should be identical. Characteristics like pageability or size limitations of code, data, and stacks should be common as well.

Shared libraries are an important part of the run-time environment because they solve two problems. First, they can reduce the system's requirements for memory and disk space through the elimination of identical code in different executable modules. (Note that the use of shared libraries can actually increase run-time memory requirements in some situations.) This motivation becomes increasingly less important with improvements in memory and disk capacity. Second, shared libraries have important software-engineering benefits. The shared library may be revised to fix bugs or improve functionality without rebuilding client executable modules. Shared libraries can be loaded based on the application's data-dependent run-time requirements, allowing applications to extend their behavior dynamically. The software engineering benefits are the primary reason today that software vendors package functionality in shared libraries.

For example, an examination of 168 Windows NT file systems at our site found 8,604 shared libraries (different file names ending in .dll). Typical single-user machines had an average of 370 shared libraries in their system directory. Excel 97 running a simple spreadsheet loaded 14 code modules and Internet Explorer 3.01 browsing `http://www.yahoo.com` loaded 43 code modules.

The following subsections examine three problems in more detail: providing safe, efficient, and location-transparent access to system call arguments in the kernel, using shared libraries in the kernel, and providing dynamically growable or pageable kernel stacks.

5.1. Location-Transparent System Calls and RPC

The system call and remote procedure call (RPC) mechanisms must be location-transparent to support shared libraries and transparent kernel collocation. In both cases, the modules contain system call and RPC call sites that must function properly whether the module is loaded into an application address space, a server address space, or the kernel address space. This is easy to achieve with a level of indirection in the call linkage. However, efficiently passing and validating arguments in a way that is location-transparent is an interesting problem. Most RPC mechanisms marshal and unmarshal arguments, which is safe and transparent but not efficient. Most system call mechanisms use `copyin`/`copyout` or `try`/`except`, which is safe and efficient but not transparent. We have developed a safe, efficient, and transparent technique based on mapping a *scratch page* in response to invalid argument accesses.

A level of indirection in the system call or RPC linkage provides the hook upon which location-transparent invocation can be built. Several styles of indirection are possible, including object-oriented C++-style virtual functions and shared-library linkages based on jump tables. When caller and callee are in different address spaces, the indirect call goes through a stub or proxy that performs the kernel trap or RPC. When caller and callee are collocated, the indirect call jumps to the callee function without any intervening wrappers, remaining on the caller's stack and using the caller-created stack frame directly. For cross-address space calls, the client stub or proxy

marshals the arguments into a buffer that is transmitted, and a corresponding server stub unmarshals the arguments for the callee function. These are well-known techniques.

This design misses an opportunity to optimize system calls or RPCs when the callee lives in the kernel. For calls into the kernel, most system call mechanisms construct a new stack frame on the kernel stack, but they pass pointer arguments through without marshalling. System services assume the burden of validating and safely dereferencing the user address space pointer. As a consequence, in many cases data copies may be avoided.

For example, most Unix implementations use `copyin` and `copyout` functions for safe argument access. These functions verify that the user's argument points into the user address space (to prevent malicious user processes from supplying valid kernel addresses and so gaining access to the kernel address space) and perform a copy that is protected against exceptions. They return a boolean value that indicates whether the copy succeeded. The `copyin` and `copyout` functions do force a copy, but often the system call can copy data directly to its final destination. However, this technique makes it expensive to examine structure members or buffers piece-meal.

Windows NT uses another solution, based on `try`/`except` exception handling in each system call. Inside a `try` block the system call first uses an inlined helper function to verify that the user argument points into the user address space. The system call can then dereference the pointer safely; any exception causes a branch to recovery code in the `except` block. This allows more flexible access to the user's structure or buffer.

However, this optimization for calls into the kernel conflicts with our requirement of location-transparent invocation. The argument validation must only happen when the callee function is invoked from user mode, because legitimate kernel-mode callers will supply pointer arguments that fail the validation.

Windows NT has a partial solution to this problem. System calls can use a helper function to inquire if they have a user-mode or kernel-mode caller and bypass argument validation for kernel-mode callers. The helper function accesses a previous-mode value saved in per-thread state. Every system call has a corresponding wrapper with a derived name (`Zw` prefix instead of `Nt`) that is used to invoke the system call internally; the wrapper saves the previous-mode value, sets it to indicate a kernel-mode caller, performs the call, and restores the previous-mode value.

5.1.1. Scratch Pages

In Rialto we used a technique that provides transparency and insulates individual system call or RPC method implementations from the concerns of argument validation, while preserving safety and efficiency. There are no marshaling or unmarshaling copies imposed on user-mode callers for common argument types and there is no overhead or wrapper for kernel-mode callers. Our implementation uses a *scratch page* to recover from exceptions due to bad arguments.

In the system call path, arguments that point to scalars, structures, buffers, and strings are range-checked to verify that they point into the user address space, but they are not otherwise validated. The system call path copies the caller's stack frame to the kernel stack. To optimize RPC when the server is located in the kernel address space, the RPC binding automatically creates special stubs that avoid marshalling and use the system call path to trap into the kernel.

The range-check requires some information about the system call arguments. In particular, the system call path must know which arguments are pointers and the size of the data to which they point. For scalars and structures the size is known statically, for buffers the size is passed in another argument, and for null-terminated strings the size doesn't matter because the kernel must access the string sequentially. An invalid page between the user and kernel address spaces simplifies the range-check for scalars, small structures, and strings, so that only one comparison is needed.

When a service inside the kernel dereferences an invalid user pointer, causing a page-fault exception, the virtual memory system maps the invalid user address to a scratch page. The unsuspecting system service then continues execution. The fault handler records the exception in the user thread's state, so that when the system call returns to user-mode the exception can be reraised in the user's context. For an invalid argument that supposedly points to a large buffer, the scratch page may be mapped multiple times. To prevent the scratch page from becoming a back door between processes, at context-switch time any scratch page mappings are removed and the scratch page is zeroed if it was mapped. Multi-processor machines require one scratch page per processor and scratch page mappings should be restricted to the TLB of the processor that experienced the exception. For multi-threaded

processes it is important that scratch page mappings created on behalf of one thread's system call not be visible to other threads.

The scratch page technique does change the semantics of erroneous system calls. For example, a write system call that supplies an invalid buffer pointer writes zeroes in addition to raising an exception in the caller's context. In practice this is not a problem. There is no loss of security for malicious programs; the program could have written zeroes anyway if it had wanted to. There is very little chance of harm from well-intentioned programs, because this type of programming error is typically caught early in development.

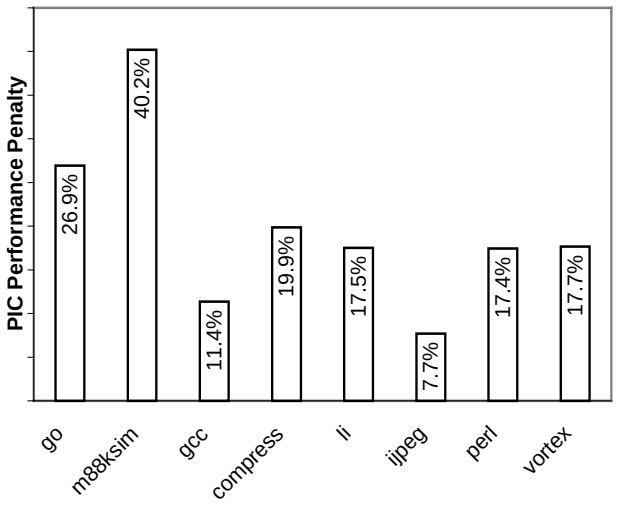
From a performance perspective, the scratch page technique removes exception-handling code and overhead from individual system call or RPC method implementations. It optimizes for the common case, that argument pointers are normally valid. In the system call path it range-checks argument pointers and on the return path there is a quick check for a pending exception. Finally, it adds a check in the context-switch path to see if the scratch page has been mapped.

5.2. Shared Libraries

There are several ways to give user-mode and kernel-mode code equal access to shared libraries. For example, position-independent code solves the problem but at some cost in performance. Alternatively, one can relocate position-dependent code pages for the kernel address space at the expense of duplicating code pages. In any case, the kernel should have its own copy of the shared library's global data in the same way that each user process loading the shared library gets its own copy of global data. Our solution for Rialto overlaps the user and kernel address spaces in a way that lets us use efficient position-dependent code, share code pages, and still have direct access to the entire current user address space from kernel mode.

To share the code pages of a position-dependent library, the library must be loaded at the same virtual address in every address space. For user processes, this can be accomplished by setting aside part of the address space for shared libraries. Preserved relocation information allows the shared library to be relocated when its preferred address is not available, but that fallback modifies code pages and prevents sharing. However, sharing code pages with the kernel is problematic. A typical arrangement splits the hardware address space, with the kernel occupying part of the address space and user processes using the remainder. If the kernel were to execute a shared library located in the current user address space, the kernel would access that process's copy of the library's global data when the kernel should have its own copy of the data, because the code contains references to user-space global data.

Figure 5-1: Position-Independent Code



5.2.1. Position-Independent Code

Some implementations of shared libraries use position-independent code (PIC). PIC allows the shared library to run at a different virtual address in each address space, including the kernel. For example, the Linux [Husain & Parker 96] operating system uses PIC for shared libraries. PIC eliminates any problems with reserving address space regions in different processes or relocating code at run-time.

However, PIC imposes a significant performance penalty on some processor architectures. Figure 5–1 shows our results for the SPECint95 benchmark suite on a 90MHz Pentium with 256K secondary cache, running Linux 2.0.18. Overall the benchmark suite ran 19.5% slower when compiled with "`gcc –O4 –fPIC`" vs. "`gcc -O4`." Although the SPEC benchmarks may not be representative of shared libraries, this does give some indication of PIC's performance impact.

Windows NT and the Microsoft development tools do not use PIC for shared libraries, and we wanted to use standard Microsoft tools. The PIC performance penalty was a concern, but tool compatibility was the primary reason we choose not to implement shared libraries with PIC.

5.2.2. Independent User and Kernel Address Spaces

One possibility for sharing position-dependent libraries with the kernel is to create separate, independent user and kernel address spaces. User processes and the kernel all get a full hardware address space. With this approach shared libraries can be loaded at the same virtual address across all address spaces including the kernel. However, this approach makes it difficult for the kernel to access the current user address space.

Different processor architectures support this technique to varying degrees. The MIPS architecture [Kane 88] restricts user processes to the low half of the address space and encourages the kernel to use the high half. On the other hand, the Motorola 88K architecture [Tucker & Coorpender 91] provides independent user and supervisor address spaces. The 88200 CMMU has two page-table base registers, for user mode and for supervisor mode. Privileged forms of the load and store instructions force the processor to use the user page-table base register. These instructions allow the kernel, running in supervisor mode, to access the current user process's address space.

Segmented architectures like Intel [Intel 95] or Hewlett-Packard's PA-RISC [Kane 96] also make it possible to create independent user and kernel address spaces. A trap into kernel mode changes the active segments and switches to the kernel address space. Inside the kernel hand-crafted assembly-language functions can set up a spare segment register to access the current user address space.

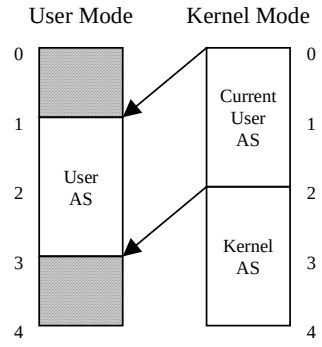
Unfortunately these techniques prohibit efficient, transparent access to system call arguments. The kernel must use special instructions or segments to read and write the current user address space. This means that system call implementations must either use `copyin`/`copyout`-style functions internally, or system call wrappers must copy arguments into and out of the kernel address space. Normal pointer dereferences in a system call implementation can not access user-mode arguments.

5.2.3. Overlapping User and Kernel Address Spaces

The solution we developed for Rialto overlaps the user and kernel address spaces while maintaining the kernel's ability to transparently access the entire user address space. Shared libraries are loaded into the overlap region, so they can run at the same virtual address in all address spaces using efficient position-dependent code. In kernel mode the entire user address space is accessible at an offset from its normal location, so transparent access to system call arguments is possible.

As an example of one possible arrangement for a 32-bit address space, see Figure 5–2. User process address spaces range from 1GB to 3GB and the kernel address space ranges from 2GB to 4GB. This creates a 1GB region, from 2GB to 3GB, in which shared libraries can be loaded. In kernel mode, the current user address space is accessible from 0 to 2GB. (This particular arrangement is hypothetical—our actual implementations differ due to hardware characteristics.)

Figure 5-2: Overlapping User and Kernel Address Spaces



To give system call implementations transparent access to arguments, calls from user mode must go through a path that adjusts or biases pointer arguments (subtracting 1GB in this example). As discussed in Section 5.1.1, our system call path also bounds-checks pointer arguments to ensure that they point into the user address space; it biases them at the same time. Kernel-mode callers invoke system calls directly.

Section 6 describes in more detail our implementation of this idea for the Intel and MIPS architectures. The Intel implementation takes advantage of the architecture's segment registers, while the MIPS implementation uses a pair of TLB contexts (ASIDs) for each user address space. We believe our overlap technique could be reasonably implemented on other architectures that support efficient methods of changing the addressing context (e.g., not flushing the TLB), either through changing segments or changing address space identifiers.

5.3. Kernel Stacks

This subsection considers ways that the operating system can provide functionality for thread stacks inside the kernel comparable to that provided for user-mode threads. Typically the size and pageability of kernel-mode stacks is restricted to prevent problems with recursive page faults on a kernel stack page. For user-mode thread stacks, on the other hand, many operating systems support pageable stacks with on-demand allocation of additional pages to

allow thread stacks to grow dynamically. We developed a method for allowing kernel-mode stacks to be dynamically growable and pageable.

Stack size restrictions significantly constrain the kernel-mode programming environment. To contain kernel stack consumption, kernel-mode programmers must avoid recursive algorithms and stack-allocated buffers or structures, module inter-dependencies must be scrutinized, and circular dependencies must be analyzed. Kernel-mode thread stacks must be put on an equal footing with user-mode thread stacks to achieve a unified programming environment.

One approach to lifting the size restriction on kernel-mode stacks would be to analyze call graphs in the kernel and at appropriate points insert code to grow the stack if necessary. This technique has been demonstrated with whole-program analysis of applications [Grunwald & Neves 96]. However, whole-program analysis is not feasible when the address space contains dynamically loaded modules or shared libraries with composite call graphs that can not be analyzed in advance.

Windows NT successfully uses a variation of this approach to cope with kernel stack growth due to recursive system calls. With version 4.0, Windows NT moved significant parts of the Win32 GUI subsystem into the kernel address space. The GUI subsystem makes callbacks to user code; that user code can in turn perform recursive GUI system calls with multiple levels of recursion possible. Windows NT initially allocates 12K of non-paged kernel stack space for every thread. The first time a thread makes a GUI system call, the kernel reallocates the stack to a 60K region to create room for possible growth. When the GUI subsystem calls back to user mode, the call-back mechanism checks the amount of remaining kernel stack space and allocates additional physical pages to maintain the invariant that a thread running in user mode has 12K of kernel stack space available for system calls, page-fault processing, and interrupts. When an idle thread is swapped out, the kernel deallocates pages beyond the 12K safety margin.

5.3.1. Dynamically Growable Kernel Stacks

The solution we devised for Rialto provides identical semantics for kernel stacks and user stacks; both can grow dynamically as necessary and the kernel can reclaim unused pages from both. The solution comprises two techniques. First, page faults on a missing kernel stack page must be handled without triggering a recursive page fault. Second, deadlocks caused by kernel stack faults while inside certain critical sections must be prevented.

Page-fault processing must happen in a context that can tolerate blocking or preemption. For example, the page-fault handler may block when it tries to enter critical sections in the virtual memory system or when it waits for paging I/O to complete. Continuation techniques [Draves et al. 91] can eliminate the need for a stack in some blocking contexts, but in general the page-fault handler must run on a thread stack or the moral equivalent.

To handle a kernel stack page fault, it suffices to perform initial page-fault processing on a small, fixed-size stack. The initial stack must be per-processor but it can be very small because it only dispatches the page-fault processing. If the first-level page fault handler identifies the fault as a kernel stack fault, then it suspends the current thread and places the thread on a queue. (In other cases, page fault handling can continue on the faulting thread's kernel stack.) A helper thread inside the kernel services the queue, performing the page-fault processing on behalf of the faulting thread. Because the helper thread only performs one specific function, its own stack requirements can be bounded.

This technique implicitly assumes that the kernel can direct page-fault processing to a stack other than the current kernel stack. With modern RISC processors, which do not save state automatically upon encountering an exception, this is easily accomplished. Section 6.1 describes how our implementation deals with the Intel architecture's tendency to push state on the current kernel stack when dispatching a kernel-mode page fault.

Kernel stack page faults that would cause deadlocks must be prevented. This means that all kernel critical sections that the helper thread may need to acquire to perform page-fault processing must be identified, and while those critical sections are held kernel stack page faults must be prevented. (This is similar to the analysis that must be performed to prevent kernel page faults on code or data from deadlocking.)

We use a probing technique to prevent kernel stack page faults. Because only a limited number of internal critical sections must be protected via probing, this mechanism does not impose any burden on normal kernel-mode programming. Before entering a critical section used by the helper thread, a kernel thread must increment a probe count in its thread state record. If the probe count indicates that the stack has not yet been probed, the thread subtracts a safety margin from the current value of the stack pointer and probes that stack location. The probe forces the potential stack fault to occur before the thread enters the critical section. The probe count prevents probing when entering nested critical sections, because that could cause a fault while the outer critical section is held. After exiting

the critical section, the thread decrements its probe count. Because only internal heap-management code and VM system code run inside these critical sections, a 2K stack safety margin suffices.

The thread probe count also allows the kernel to reclaim unused stack pages. For example, the helper thread might wake up periodically (or when memory is tight) and look for threads with unused stack space. If the probe count indicates that it is safe to proceed, the thread can be suspended and its excess stack pages deallocated without risk of deadlock. Our current implementation reclaims unused stack pages every ten seconds.

6. Implementation

Our implementation of the Rialto kernel successfully unified the user and kernel environments. Rialto is a small, soft real-time system designed for disk-less consumer devices [Jones et al. 96, Draves et al. 97]. Rialto supports multiple address spaces and provides a strong separation between user mode and kernel mode operation, but it does not support demand paging or swapping. The implementation segregates machine-dependent code for portability; we have ports for Intel processors and for the MIPS R3000.

Rialto uses single mechanism based on the Component Object Model (COM) [Brockschmidt 95] for both system calls and RPC. In this model, objects like processes or threads export one or more interfaces (C++-style arrays of method pointers). A standard QueryInterface method allows a client to bind dynamically to interfaces. Transparent remote invocation happens through small proxies (stub objects) in the user address space. The mutex/condition synchronization system calls are an exception; they do not use COM and they call into the kernel via a private interface to trigger scheduling actions.

The RPC mechanism maintains a pool of server threads in every process, including the kernel, and it dynamically binds server threads to clients. The kernel automatically creates and destroys server threads. Using this mechanism for system calls (calls into the kernel) simplified our implementation and saved stack memory, because user threads did not have dedicated kernel stacks. However, this did impose a performance penalty on system calls relative to a more traditional system call implementation. Without affecting the unified programming environment but with some increase in complexity, we could have provided user threads with their own kernel stacks.

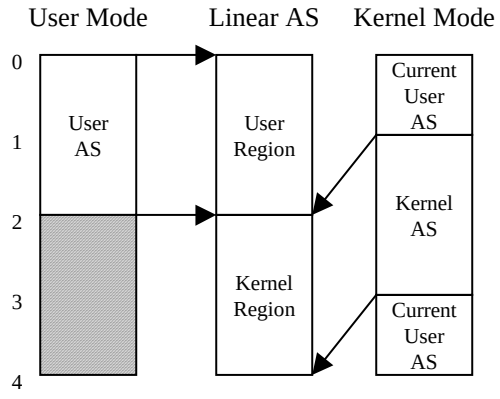
The virtual memory system uses a virtual translation lookaside buffer (VTLB) that combines the pmap approach to portability [Rashid et al. 87] with a software TLB implementation [Huck & Hays 93, Bala et al. 94]. The implementation can replace any mapping in the VTLB, including kernel mappings, so the VTLB can use a small, fixed-size cache instead of page table pages.

6.1. Intel

The Intel architecture presented two challenges for our implementation. First, changing address spaces is relatively expensive because the operation flushes the TLB. Second, the hardware page fault handler pushes state on the current kernel stack, and if the kernel stack pointer is invalid this results in an unrecoverable double-fault. We were able to use the architecture's support for segmentation and protection rings to overcome both of these problems.

We used segmentation to switch efficiently between the user address space and the kernel address space. The Intel architecture translates addresses twice. The processor first converts a virtual address to a "linear" address via a segment register. Each segment register has a base address and a limit. A two-level page table directory then converts the linear address into a physical address. The processor automatically reloads the segment registers when changing between user mode and kernel mode. Changing the page-table base register, on the other hand, is potentially expensive because it flushes the TLB.

Figure 6-1: Intel Address Space



As shown in Figure 6–1, our Intel implementation uses the segment registers to map the overlapping user and kernel address spaces into disjoint regions of the 32-bit linear address space. User segments have a zero base and a 2GB limit; kernel segments have a 1GB base and a 4GB limit. The region from 1GB to 2GB is available for shared libraries in all virtual address spaces. Shared library code pages are mapped twice in the linear address space. (If one implemented shared libraries with position-independent code, they would also be mapped

twice.) Because address arithmetic is modulo 32-bits, the kernel can subtract 1GB before accessing a user pointer and the kernel segment's base adds back the 1GB. The page-table base register does not change across system call boundaries.

We used the architecture's protection rings to handle kernel stack faults safely. The Intel architecture supports four protection levels or rings, although the page-table protection bits lump rings 0, 1, and 2 together as kernel mode. Only ring 0 can execute certain privileged instructions, like changing the page-table base register for context-switches, but any ring can be given the ability to perform I/O instructions. The architecture defines several mechanisms that can change rings. Interrupt gates and call gates reload the segment registers and switch stacks if they are changing to a more privileged ring, but they just push an instruction pointer and segment on the current stack when there is no ring change. Task gates reload the entire machine state, making them significantly more expensive.

To create a safe environment for stack faults, all thread-level or preemptible kernel code runs at ring 1. First level device interrupts and page-faults at ring 1 trap to a small per-processor ring 0 stack. As an optimization most system calls trap directly from ring 3 to ring 1, but synchronization calls trap to ring 0 because they trigger context-switches. Kernel code running at ring 1 must call to ring 0 to flush the TLB and to voluntarily context-switch.

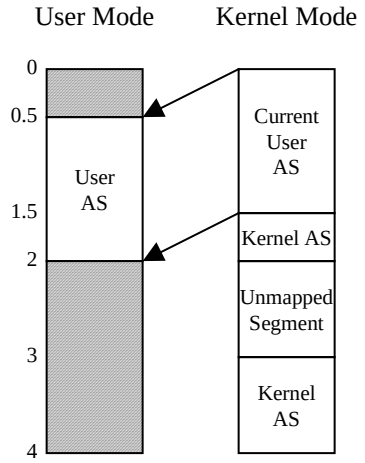
We considered an alternative design, running all kernel-mode code at ring 0 and using task gates for device interrupts and page faults. A task gate can switch to a safe stack in response to a stack fault at ring 0. Section 7.4 examines the performance impact of both the ring 1 design and the task gate design.

6.2. MIPS

The MIPS implementation of Rialto runs on an R3000 processor board. Because the MIPS TLB uses a software miss handler with support for multiple addressing contexts (ASIDs), it provides an interesting example of how to overlap the user and kernel address spaces to support shared libraries.

The MIPS architecture divides the 32-bit hardware address space into several segments. It reserves the high 2GB for kernel mode. The kernel's segment subdivides into a 1GB unmapped segment and a 1GB mapped segment. Memory accesses to the kernel's unmapped segment bypass the TLB; the processor masks off the high bits of the virtual address to form a physical address. TLB entries are tagged with a 6-bit ASID and a "global" bit. If the global bit is set then the ASID is ignored, making the mapping visible in all 64 addressing contexts.

Figure 6-2: MIPS Address Space



Rialto allocates a pair of ASIDs for each user process, with one ASID for the process's own execution and one for kernel-mode access to the user process. Figure 6–2 depicts this arrangement. When a user process makes a system call, the trap handler switches to the process's kernel-mode ASID. (Trap handlers for device interrupts and TLB misses do not change the ASID, because they do not use shared libraries or access the user address space.) The kernel-mode ASID shifts the user address space by 0.5GB, giving threads inside the kernel access to the full current user address space. The 1.5–2GB region is available for loading shared libraries. TLB entries mapping the 3–4GB region have the global bit set.

Because it uses more ASIDs, our design has the potential of increasing the pressure on the TLB. For example, when a user process fills a buffer and then passes the buffer to the kernel with a system call, Rialto uses a second TLB entry to access the buffer in kernel mode. Furthermore, the TLB entries that the kernel uses to execute shared libraries do not have the global bit set. If instead one implemented shared libraries with position-independent code the kernel could map shared libraries above 3GB with global TLB entries, eliminating some TLB misses. Section 7.5 examines both of these performance issues.

6.3. Deficiencies

Our unification of the user and kernel programming environments fell short in two respects. First, our debugger was less effective for the kernel environment. We ported the Microsoft VC++ debugger, using a debug stub that loaded

into the kernel and communicated with the user interface on a host machine. The debugger could handle all user-mode code and shared libraries or servers loaded in kernel mode, but it could not debug device interrupt handlers and it could not step into most internal kernel code. Remedying this would require a lower-level debug stub, perhaps derived from the NT kernel debugger.

Second, the Win32 graphics (GDI) and windowing (USER) services took some shortcuts. GDI and USER used a location-transparent interface called Direct Draw to communicate with the low-level graphics driver, but they also used a few "back-door" interfaces that were not location-transparent. As a result they could not run in a user process context. Furthermore the final version of the USER implementation implicitly assumed that clients were running on a different stack. This meant that modules collocated in the kernel would get incorrect behavior for Win32 USER APIs. Other application-level interfaces did not have these problems.

7. Performance

We measured Rialto's performance using several scenarios drawn from the interactive TV (ITV) system. Our results show that kernel collocation improved CPU times up to 35% and reduced memory usage more than 900K. Our stack management policies cut stack memory consumption by a factor of three, saving 460K for a representative scenario. Our Intel implementation's use of ring 1 increased CPU times 2-4%, and our MIPS implementation's use of two ASIDS per process increased the number of MIPS hardware TLB misses roughly 3%.

7.1. Experimental Setup

For our measurements we used a small test system consisting of a single set-top box (STB) running Rialto connected via a private ethernet to a single head-end server running Windows NT. Although we measured eight different scenarios, because of space limitations we only report numbers here for three representative scenarios.

The set-top box used a PCI bus, a 75Mhz Pentium processor with no secondary cache, and 16MB of main memory. The STB also had three PCI cards, for ethernet, MPEG video, and graphics. The STB displayed 2Mb/s MPEG-I digitized video from the Tiger video file system [Bolosky et al. 96] on the head-end server. The head-end server also hosted many other ITV services, including the channel and program databases, financial transaction services, and a server for program binaries. Users interacted with the STB via a hand-held infra-red remote controller.

The test scenarios used three application programs. The Navigator application controlled channel changing; it was always running throughout the scenarios. Navigator managed "broadcast" channels directly and it started and stopped the applications associated with other channels. The Movies On Demand (MOD) application allowed users to browse and preview movies, purchase movies, and play movies with VCR-like functionality. The Electronic Program Guide (EPG) application displayed a tabular view of the channel and program databases. An alternate view displayed programs sorted by content instead of channel and time. The three scenarios for which we report results were MOD (mostly used the MOD application), EPG (mostly used EPG), and MOD-EPG (used both MOD and EPG).

To achieve reproducible results, we used a "smart monkey" test tool running on the head-end server. The smart monkey read a script of interactive actions, consisting of remote control key presses and delays. It sent key codes in network packets to a driver running on the STB, which simulated real key presses at a low level in the I/O system. A special key code started and stopped data collection for the scenarios.

In addition to being much smaller, our test system differed from the ITV trial system in Japan in several respects. The trial system used ATM over fiber with 6Mb/s MPEG-II video. The trial system used real-time MPEG-II encoders to support broadcast channels, whereas in our test system the broadcast channels just displayed a test pattern. The trial system used a Japanese user interface, substituting Kanji for English. Finally, for the trial system NTT developed several additional applications that were not available for our testing.

7.2. Kernel Collocation

To examine the effects of kernel collocation on CPU and memory usage, we measured the scenarios two ways, first with all system services collocated in the kernel and second with seven services running in separate user processes. Among the seven subsystems were Rialto's real-time resource planner, the RPC endpoint mapper, head-end

services, and the SNMP service. The results showed that kernel collocation saved approximately 230 4K pages in all three scenarios, while the reduction in CPU time varied from 2.6% to 35.0%.

Because the scenarios used interactive and real-time applications driven from a test script, the scenarios' duration did not change significantly. Therefore we recorded cumulative system idle time and from that derived CPU time. The reduction in CPU time demonstrated that kernel collocation improved the system's performance. The MOD scenario showed the smallest improvement in CPU time because it spent the bulk of its time playing video, and that operation used the MPEG driver without involving other system services. The memory savings were relatively constant across the scenarios because they mostly reflected static process overheads: page table pages, copy-on-write shared library data pages, initial stack pages, etc.

Table 7-1: Effect of Kernel Collocation

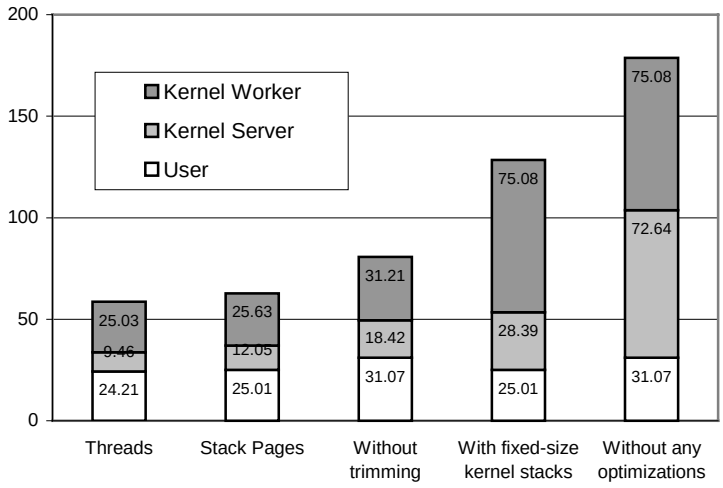
	MOD		EPG		MOD-EPG	
Collocated (K) vs. Not (U)	K	U	K	U	K	U
Duration (seconds)	761.38	761.43	678.08	678.09	625.30	625.31
CPU time (seconds)	302.35	310.35	393.13	604.49	254.68	311.10
CPU Savings	2.6%		35.0%		18.1%	
Max Memory Usage (pages)	2214	2452	1602	1829	2303	2537
Memory Savings	238 pages or 9.7%		227 pages or 12.4%		234 pages or 9.2%	

7.3. Stack Space

Our stack management optimizations collectively reduced total stack memory consumption by approximately a factor of three. Figure 7–2 shows the results for the MOD-EPG scenario. (The other scenarios demonstrated similar results.) For the run shown, we logged every thread creation and termination and every stack page allocation and deallocation. By post-processing the log we could calculate what the stack usage would have been if our optimizations had been disabled. The figure shows time-weighted averages for thread counts and stack pages allocated, averaging from system boot through the end of the scenario.

In this scenario the average number of stack pages, 62.68, barely exceeded the average number of threads, 58.70, because most threads had only one stack page most of the time. User stacks and kernel stacks both had the same maximum size of three pages. (Earlier in development we saw larger user and kernel stacks, but the biggest consumers of stack space were fixed.) Without the periodic trimming or deallocation of unused user and kernel stack pages, the average number of stack pages would have been 80.70. Without growable kernel stacks (assuming fixed 3-page kernel stacks but still trimming user stacks), the average number of stack pages would have been 128.48. Finally, if we had provided each user thread with a dedicated kernel stack instead of using a dynamically allocated pool of kernel server threads, the average number of stack pages would have been 178.79, consuming 460K more memory than was actually used.

Figure 7-2: Stack Memory



7.4. Intel Ring 1 Overhead vs Task Gates

Our Intel implementation's use of ring 1 (to handle kernel stack faults safely) increased CPU times in the scenarios 2-4%. To estimate this, we counted the number of ring transitions of different types in the scenarios and multiplied by the measured cost of individual ring transitions. We also calculated that an alternate implementation, using task gates, would have twice the CPU overhead.

We measured the cost of interrupt and task gate transitions in two ways. First, we used a micro-benchmark loop, yielding a lower-bound on their cost in real code. Second, we measured the cost of individual operations during the MOD scenario, using the Pentium's cycle counter. We inserted measurement code into an internal kernel function, so that every time the function was called it measured the cost of one interrupt or task gate transition in addition to performing its normal operation. This yielded timings of each operation that reflected more realistic TLB, cache, and bus activity. Table 7–3 shows the results of our measurements. In our subsequent calculations we used the mean costs derived from the application scenario.

Table 7-3: Pentium costs (microseconds)

	Scenario		Micro-Benchmark	
	Mean	Std Dev	Mean	Std Dev
Task gate	134.0	47.1	59.2	3.0
Ring 3→0→3	49.5	16.1	32.9	0.6
Ring 1→0→1	46.4	18.5	22.0	1.1
Ring 0→0→0	44.1	26.7	12.4	0.5

To estimate the overhead of using ring 1, we counted the number of interrupt gate ring transitions during the application scenarios. We used the raw counts and the individual cost of each type of ring transition to subtract the ring 1 overhead: removing the cost of ring transitions that were introduced by the design and changing the cost of some ring transitions. (For example, device interrupts at ring 1 were adjusted to be device interrupts at ring 0, which are more efficient.) Table 7–4 shows the results of calculating the "Base CPU" from the measured "Ring 1 CPU."

Similarly, we estimated the overhead of an alternative design using task gates instead of ring 1. "Task Gate CPU" in Table 7–4 shows the resulting calculated overhead of the task gate design. The task gate design would have had roughly twice the overhead of the ring 1 design that we actually implemented.

Table 7-4: Ring 1 and Task Gate CPU Overhead

	MOD		EPG		MOD-EPG	
Base CPU	290.28		385.04		246.31	
Ring 1 CPU	302.35	4.16%	393.13	2.10%	254.68	3.40%
Task gate CPU	313.58	8.03%	414.42	7.63%	264.33	7.31%

7.5. MIPS ASID Pressure

To estimate the performance degradation associated with our dual-ASID implementation of overlapping user and kernel address spaces, we simulated the MIPS hardware TLB on our Pentium set-top box. Although our MIPS implementation works and passes our kernel test suite, we did not have any real applications for the R3000 development board. On-line simulation of the MIPS TLB on the Pentium allowed us to gather data using the ITV applications. In the scenarios we measured, the dual-ASID design increased the number of MIPS hardware TLB misses by roughly 3%. Because MIPS TLB miss handling typically accounts for a small percentage of total program execution, this implies that the overall overhead for the dual-ASID design would likely be very small.

We used a software TLB miss handler on the Pentium to simulate the MIPS TLB. To create a software TLB miss handler, we used page tables that mapped the kernel's own code and static data. The page tables mimicked the effect of the MIPS unmapped memory segment. Whenever the Pentium referenced a "mapped" address that was not in the Pentium TLB, the Pentium would load an invalid page table entry and hence page fault. The page fault handler accessed the simulated MIPS virtual memory system to find a physical address translation. The handler then created a valid PTE for that translation and restarted program execution. At the next page fault, the page fault handler invalidated the previous PTE and created a new valid PTE for the next translation to be entered into the Pentium TLB. Because execution of the page fault handler evicted entries from the Pentium TLB, it was possible for a single instruction that accessed multiple pages to thrash the TLB, preventing forward progress. We solved this problem by taking advantage of the Pentium TLB's set-associativity and its support for 4MB page table entries. (The exact details depended on information from the Intel-confidential Appendix H supplement to the Pentium manual.)

We ensured that the Pentium hardware TLB mapped a subset of the simulated MIPS hardware TLB. Whenever the simulation evicted a translation from the MIPS TLB, it flushed the translation from the Pentium TLB. Therefore the simulation of the MIPS virtual memory system could count MIPS TLB misses. It could not count MIPS TLB hits, because hits in the Pentium TLB were not visible to the simulation. The set-top box was not fast enough to perform two independent on-line simulations (with dual ASIDs and without), so we performed one simulation with dual

ASIDs and counted "extra" TLB misses. Because there were relatively few extra TLB misses we believe this was reasonably accurate.

Because of the simulation overhead, we had to modify the three application scenarios. The set-top box could still display MPEG video, but we had to increase the delays in the "smart monkey" scripts because the applications were noticeably less responsive.

Table 7-5: Increase in MIPS TLB Misses

	MOD	EPG	MOD-EPG
User/kernel	2.25%	2.74%	2.33%
Kernel/kernel	0.32%	0.48%	0.45%
Total	2.58%	3.22%	2.78%

Table 7–5 shows the increase in MIPS TLB miss counts due to the dual-ASID implementation. As described in Section 6.2, there were two sources for the increase: TLB misses for an ASID when a mapping for the dual ASID was already present in the TLB (User/Kernel), and kernel-mode TLB misses for shared library code or static data when a mapping for some other kernel-mode ASID was already present in the TLB (Kernel/Kernel). However, in all cases the increase in the number of simulated MIPS hardware TLB misses was small.

8. Conclusions

We have shown that it is practical to unify the user and kernel programming and run-time environments. Based on our experience developing the Rialto kernel and deploying it in an interactive TV system, we believe a unified user/kernel environment produces significant software-engineering benefits. Transparent kernel collocation is an important benefit of a unified environment. For our system running typical application scenarios, kernel collocation saves 900K bytes of memory and reduces CPU times by up to 35%. Our dynamic stack management techniques save 460K more in stack space. The CPU overhead for our techniques is 2-4% on our Pentium-based set-top box.

We believe our results are applicable to "general-purpose" operating systems, with some caveats. Careful interface design is required to reduce the software-engineering cost of "internal" interfaces within the kernel address space. If this is not done well then our techniques make it easy to move services into the kernel, where subsequent performance optimization traps them through the use of internal interfaces. Furthermore our techniques have a small CPU overhead, although the software-engineering benefits and painless kernel collocation compensate for this.

Acknowledgments

We would like to thank Gilad Odinak for his work on Rialto. We had the privilege of working with many other great people, too numerous to mention, on Rialto and Microsoft's ITV system. Dave Cutler and Butler Lampson provided comments on early drafts. Philippe Bernadat, Michael Condict, Mike Kazar, Jay Lepreau, Mark Lucovsky, Doug Orr, M. Satyanarayanan, and Dan Stodolsky promptly replied to our email questions. Scott Draves and Dave Hanson assisted us with Linux machines. As usual, Pamela Bridgeport and the Microsoft library staff were unfailingly helpful.

References

[Armand 91] F. Armand. Give a Process to your Drivers! In *Proceedings of the EurOpen Autumn 1991 Conference*, Budapest, Hungary, September 16-20, 1991.

[Baker 95] A. Baker. The Windows NT Device Driver Book: A Guide for Programmers. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[Bala et al. 94] K. Bala, M. F. Kaashoek, W. E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 243–253, November 1994.

[Bershad et al. 95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, Colorado, December 3-6, 1995.

[Bolosky et al. 96] W. J. Bolosky, J. S. Barrera III, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, R. F. Rashid. The Tiger Video Fileserver. In *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, IEEE Computer Society, Zushi, Japan, April, 1996.

[Brockschmidt 95] K. Brockschmidt. Inside OLE, Second Edition. Microsoft Press, Redmond, WA, 1995.

[Condict et al. 94] M. Condict, D. Bolinger, D. Mitchell, E. McManus. Microkernel Modularity with Integrated Kernel Performance. http://www.cs.utah.edu/~lepreau/osdi94/condict/abstract.html. Presented at the *Mach-Chorus Workshop at the First Symposium on Operating Systems Design and Implementation*, Monterey, California, November 14-17, 1994.

[Corbato et al. 72] F. J. Corbato, J. H. Saltzer, and C. T. Clingen. Multics—The First Seven Years. In *Proceedings of the American Federation of Information Processing Societies Spring Joint Computer Conference*, pages 571–583, 1972. Reprinted in P. Freeman. *Software Systems Principles*. Science Research Associates, Chicago, 1975.

[Custer 92] H. Custer. Inside Windows NT. Microsoft Press, Redmond, Washington, 1992.

[Cutler 97] D. Cutler, Windows NT Architect. Personal communication, March 1997.

[Draves et al. 91] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, California, October 13-16, 1991.

[Draves et al. 97] R. P. Draves, G. Odinak, S. M. Cutshall. The Rialto Virtual Memory System. Technical Report MSR-TR-97-04, Microsoft Research, February 15, 1997.

[Druschel & Peterson 93] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, Asheville, North Carolina, December 5-8, 1993.

[Engler et al. 95] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 3-6, 1995.

[Grunwald & Neves 96] D. Grunwald and R. Neves. Whole-Program Optimization for Time and Space Efficient Threads. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59, Cambridge, Massachusetts, October 1-5, 1996.

[Huck & Hays 93] J. Huck and J. Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, San Diego, California, May 16-19, 1993.

[Husain & Parker 96] K. Husain and T. Parker. Linux Unleashed, Second Edition. SAMS Publishing, Indianapolis, Indiana, 1996.

[Intel 95] Pentium® Processor Family Developer's Manual, Volume 3. Intel Corporation, Santa Clara, California, 1995.

[Jones et al. 96] M. B. Jones, J. S. Barrera III, A. Forin, P. J. Leach, D. Rosu, M.-C. Rosu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 249–256, Connemara, Ireland, September, 1996.

[Kane 88] G. Kane. MIPS RISC Architecture. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Kane 96] G Kane. PA-RISC 2.0 Architecture. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1996.

[Kazar 97] M. Kazar. Personal communication, February 21, 1997.

[Lepreau et al. 93] J. Lepreau, M. Hibler, B. Ford, J. Law. In-Kernel Servers on Mach 3.0: Implementation and Performance. In *Proceedings of the USENIX Mach III Symposium*, pages 39–55, Sante Fe, New Mexico, April 19-21, 1993.

[Liedtke 95] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain Resort, Colorado, December 3-6, 1995.

[Organick 72] E. I. Organick. The Multics System: An Examination of Its Structure. MIT Press, Cambridge, Massachusetts, 1972.

[Rashid et al. 87] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, Palo Alto, California, October 5-8, 1987.

[Ritchie & Thompson 78] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal*, 57(6): pages 1905–1930, July-August, 1978.

[Salz 97] R. Salz, Open Software Foundation. Personal communication, March 1997.

[Steere et al. 90] D. C. Steere, J. J. Kistler, M. Satyanarayanan. Efficient User-Level Cache Management on the Sun Vnode Interface. In *Proceedings of the Summer Usenix Technical Conference*, Anaheim, California, June 1990.

[Tucker & Coorpender 91] M. Tucker and B. Coorpender. Programming the Motorola 88000. Windcrest Books, Blue Ridge Summit, Pennsylvania, 1991.