The Rialto Virtual Memory System

Richard P. Draves
Gilad Odinak
Scott M. Cutshall

February 15, 1997

Technical Report
MSR-TR-97-04

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

The Rialto Virtual Memory System

Richard P. Draves, Gilad Odinak, Scott M. Cutshall

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

Abstract

This paper describes the design and implementation of virtual memory in the Rialto kernel. The Rialto VM system supports sparse virtual address spaces with mapping between address spaces, copy-on-write faults, and auto-commit (demand-allocation) faults. It does not currently support demand page-in, page-out, or external pagers. Novel aspects of the design include a VTLB architecture that bounds the memory consumed in machine-dependent mapping structures, real-time requirements for preemptibility and interruptibility, provision for efficient sharing between user address spaces and the kernel via a user/kernel address bias, and growable kernel-mode thread stacks.

The Rialto kernel is a light-weight, "soft" real-time kernel. It has been successfully deployed as part of Microsoft's MiTV/OS for the set-top boxes in the Microsoft/NTT interactive TV trial in Yokosuka, Japan.

1. Introduction

Rialto is a light-weight, soft real-time kernel that supports virtual memory, designed primarily for use in consumer appliances like set-top boxes, hand-held or pocket computers, Internet terminals, and CD/DVD game players. These platforms share a few characteristics, notably limited memory but also multimedia or real-time requirements and no requirement for paging.

The following design goals for the kernel affected the virtual memory system:

- Limited memory consumption. We wanted to reduce the VM system's need for machine-dependent mapping structures (like page tables) as much as possible. We also wanted to reduce the per-thread overhead for stacks by making *all* thread stacks growable on demand (and make unused stack pages reclaimable), and we wanted to reduce the system's overall memory consumption by allowing code to be shared between user mode and kernel mode.
- Real-time requirements for preemptibility and interruptibility. We wanted the VM system to be completely interruptible, and be preemptible except for short code sections. In general we wanted the kernel to support preemptive scheduling with sub-millisecond granularity.
- Support for Microsoft's standard APIs. Applications for the system are written to a subset of Win32 [Microsoft 93]. The Win32 memory management APIs required some special support.
- Portable to Intel x86 processors [Intel 95] with hardware page tables and RISC processors like MIPS [Kane 88] with software TLB miss handlers. Rialto first ran on a MIPS R3000 board and was later ported to Intel x86 PCs and set-top boxes.

We had several reasons for supporting virtual memory in an operating system designed for consumer appliances. First, virtual memory allows the operating system to protect against buggy applications. This facilitates development and makes the platform more reliable for the end-user. Furthermore, Internet-connected systems should protect against malicious applications. Alternative protection mechanisms such

as interpreted languages or sand-boxing [Wahbe et al. 93] give up size or performance. Second, virtual memory avoids problems with fragmentation of physical memory, and problems with sizing heaps and stacks. Finally, virtual memory makes our system much more compatible with desk-top operating systems.

On the other hand, consumer appliances have little need for paging or swapping. These platforms do not have paging disks and they do not have the network bandwidth or servers available to support network paging. Furthermore, paging introduces unpredictable latencies; much of the system would need to be locked down or memory resident anyway to support real-time or multimedia programs.

The rest of the paper is organized as follows. The remainder of this introduction presents a short overview of Rialto and Microsoft's interactive TV system. Section 2 discusses Rialto's virtual memory interfaces and APIs. Section 3 presents the implementation, and Section 4 examines some performance measurements. Finally, Section 5 discusses related work and Section 6 presents conclusions.

1.1 Overview

In most respects, the Rialto kernel is just another process whose threads run in the processor's privileged mode. All threads are preemptible and have growable stacks. The kernel implements scheduling and synchronization, inter-process communication, and virtual memory. Device drivers and system services (such as resource planning, file systems, networking, and graphics) are not part of the kernel image but load into the kernel process. Device drivers do most of their work in a kernel thread context, subject to the scheduler. The kernel image also contains some run-time functionality (like the loader and the heap code) that executes in both user mode and kernel mode.

Rialto's "soft" real-time model combines a coarse-grained resource reservation system with fine-grained control over scheduling behavior via scheduling constraints. Constraints allow an application to provide to the scheduler deadlines and estimates of execution time. See [Jones et al. 95] for further details.

Rialto uses Microsoft's Component Object Model (COM) [Brockschmidt 95] as the basis for inter-process communication, including system calls on kernel objects. In this model, objects export one or more interfaces (C++ style vtbls). Every interface inherits from IUnknown, which provides three standard methods: AddRef and Release for reference-counting the interface, and QueryInterface for determining at run-time what other interfaces the object behind an interface pointer supports.[1]

COM provides for transparent remote invocation of interface methods via proxies. In Rialto, most methods use "generic" proxies, which just trap into the kernel. The kernel IPC mechanism selects a server thread from a pool in the server process (creating and destroying server threads as needed), copies arguments, and sets up the stack frame. Interface signatures must be registered with the kernel. Marshalling in the proxy is required when passing "complex" arguments, such as interface pointers or structures with embedded pointers.

Rialto optimizes calls on objects living in the kernel process, with support in the VM system to avoid copying pointer arguments. The kernel call path is similar to a traditional system call, with the addition of server thread selection and object dispatching. The IPC system makes this all location-transparent—depending on where things are loaded at run-time, the same code might perform calls local to a process, calls into the kernel, calls out of the kernel, or calls between two user processes.[2]

The VM system supports dynamically loading libraries (DLLs) into user processes and the kernel, with image pages shared copy-on-write. Each process, including the kernel, using a DLL gets private static data pages via copy-on-write faults. This saves memory because drivers and services (non-shared DLLs)

---

[1] Rialto uses a tiny subset of COM. For example, it does not support class objects, monikers, or a running object table.

[2] We designed the IPC system to support transparent inter-machine method calls, but that hasn't been implemented. Currently inter-machine communication happens via sockets or DCE RPC, with an RPC runtime ported from Windows NT.

loaded into the kernel can take advantage of standard functionality residing in shared DLLs (like the C runtime, DCE RPC runtime, and Win32 APIs) without code duplication.

Any system service that isn't a device driver or statically linked with a driver (for example, file systems and networking services qualify) can be loaded in a user process or be loaded into the kernel. Many of these services were first developed in a user process context, but in normal use we load system services into the kernel process. This avoids the space overhead of having separate server processes and gains the advantage of fast local calls among the system services and the kernel.

1.2 Microsoft Interactive TV

Microsoft's interactive TV system (MiTV) uses the Rialto kernel. MiTV runs applications like Electronic Program Guide and Video on Demand on set-top boxes (STBs) connected via ATM over fiber or hybrid fiber/coax networks to servers running Windows NT and the Tiger multimedia file system [Bolosky et al. 96]. The STBs have Pentium processors, high-performance graphics boards, hardware MPEG2 decoders, and ATM network interface cards. Microsoft and NTT deployed MiTV on March 22, 1996 in a joint trial in Yokosuka, Japan. The trial started with 30 homes and has since grown to about 300 homes. See [Jones 96] for an overview of the MiTV system and the Yokosuka trial.

2. Interfaces

Rialto's virtual memory interfaces consist of two COM interfaces for kernel objects, IAddressSpace and IMemoryObject, plus several APIs for use by drivers and only available inside the kernel process. The virtual memory interfaces have several novel aspects, including control over zero-filling and non-atomic failure semantics. Unlike other memory object/mapping designs, Rialto's Map method uses an abstract IMemoryObject interface but disallows "foreign" (non-kernel) implementations. This simplifies our interfaces and implementation without preventing us from supporting external pagers in the future. Process objects support both the IAddressSpace and IMemoryObject interfaces, so mappings between processes may be established. The Map operation supports asymmetric copy-on-write and no-exception mappings.

2.1 IAddressSpace

The IAddressSpace interface (see Figure 1) provides methods for reserving, committing, protecting, mapping, and deallocating virtual address space. The kernel's process objects support the IAddressSpace interface, and one may QueryInterface from the other process interfaces to obtain an IAddressSpace interface for a specific process. In addition, we have a CurrentAddressSpace API that caches and returns the IAddressSpace interface for the current process.

```
interface IAddressSpace : IUnknown {
        SCODE AllocationInformation([out] ADDR_SIZE *pPageSize,
                        [out] ADDR_SIZE *pAlignment);
        SCODE Allocate([out] ADDRESS *pAddr, ADDR_SIZE Size);
        SCODE Reserve([in, out] ADDRESS *pAddr, ADDR_SIZE Size, ADDR_FLAGS Flags);
        SCODE Commit(ADDRESS Addr, ADDR_SIZE Size, ADDR_FLAGS Flags);
        SCODE Protect(ADDRESS Addr, ADDR_SIZE Size, ADDR_FLAGS Flags);
        SCODE Decommit(ADDRESS Addr, ADDR_SIZE Size);
        SCODE Deallocate(ADDRESS Addr, ADDR_SIZE Size);
        SCODE Map([in, out] ADDRESS *pAddr, ADDR_SIZE Size, ADDR_FLAGS Flags,
                        IMemoryObject *pMemObj, ADDR_SIZE Offset);
        SCODE GetMemoryStatus([out] MEMORY_STATUS *pStatus);
}
```

Figure 1: IAddressSpace Interface

The interface distinguishes between reserving and committing address space. Address space that is reserved is merely set aside for future use; accesses to reserved address space produce an exception. Once a region has been reserved, it may be committed. (The Allocate method reserves and commits in one simple operation.)

Committed address space has protection attributes and may be accessed. Normal committed memory has physical pages allocated, but one may optionally specify that a region should be auto-committed. In that case, physical pages are allocated on demand via auto-commit faults. We use auto-committed memory for thread stacks.

Rialto allows an application to specify whether it needs zero-filled memory, and whether it cares if its deallocated pages are later given to another application without erasure. By default, Rialto erases but does not zero-fill committed memory. The Commit method supports flags for "alloc-zero" and "dealloc-no-erase." In our system only the Win32 VirtualAlloc function specifies alloc-zero, and the loader uses dealloc-no-erase for code pages. In one scenario (see Section 4) we measured a 43% reduction in the number of pages zero-filled or erased as a result of this optimization, and a 13% reduction excluding boot-time startup effects. Changing VirtualAlloc not to specify alloc-zero (in practice our applications don't rely on this feature of VirtualAlloc) improves these percentages to 47% and 15%, respectively.

The IAddressSpace methods that operate on a reserved address space region (Commit, Protect, Decommit, Deallocate) have unusual failure semantics. They are explicitly defined *not* to be failure-atomic; if they return a failure code then their operation may have been performed on part of the region. (This contrasts for example with Mach's VM operations, which were all failure-atomic [Young 89].)

This simplifies the implementation with no real inconvenience for the user. Correct programs will never encounter any of the semantic errors (like trying to Protect a region that is only partially committed). Programs may legitimately encounter resource problems, like running out of physical memory inside Commit, but then their error path clean-up needs to Deallocate the region because it is reserved, and if Commit left the region partially-committed then Deallocate takes care of that too.

Several aspects of the IAddressSpace interface were designed to support the Win32 memory management APIs (VirtualAlloc, VirtualFree, etc.). For example, the Win32 functions distinguish between reserve and commit. The IAddressSpace::GetMemoryStatus method returns the same status structure as the Win32 GlobalMemoryStatus API.

One difference is that Win32 has the concept of sections. An allocation operation produces a section, and the section may only be deallocated in its entirety by specifying the starting address of the section. In contrast, Rialto is more page-oriented and by default does not preserve any boundaries between allocations. However to support the Win32 semantics we added an option to Reserve to create a section and then allow the Size argument for subsequent operations on the section to be zero.

2.2 Mapping and IMemoryObject

Rialto provides some simple support for mapping memory objects into address spaces. The current implementation allows views on an address space to be mapped into another address space; this meets the needs of the loader for sharing image pages and for drivers and servers to map client memory for efficient access. The interface is flexible enough to allow the implementation of more sophisticated memory objects, such as memory-mapped files or external pagers.

```
interface IMemoryObject : IUnknown {
       SCODE RestrictedAccess(ADDR_SIZE Offset, ADDR_SIZE Size, ADDR_FLAGS Prot,
                        [out] IMemoryObject **pMemObj);
}
```

Figure 2: IMemoryObject Interface

The IAddressSpace::Map method takes an IMemoryObject interface pointer as an argument; it maps the memory object into the address space. The caller can either specify an already-reserved region or request that address space be reserved. In addition to protection flags, the caller can also specify copy-on-write and no-exception attributes for the mapping.

The implementation may enforce a machine-dependent mapping alignment granularity, as indicated by the IAddressSpace::ReservationInformation method. Normally the mapping granularity is equal to the page size, but on machines with virtually-indexed, physically-tagged caches, multiple virtual mappings of

a physical page must be restricted to make memory accesses via the different mappings index the cache identically. For most users this is not a concern, because they are either allowing Map to select and reserve the mapped address, or because the source virtual address and mapped virtual address are the same.

The current implementation supports two kinds of memory object, process objects and view objects. One can QueryInterface from any of the process object interfaces, including IAddressSpace, to obtain an IMemoryObject interface that allows that process's address space to be mapped into some other address space. The RestrictedAccess method in IMemoryObject (see Figure 2) creates a view object based on the original object. For example, a view object can be used to hand out an IMemoryObject that allows only read access to a particular read-write region in an address space.

Unlike memory object interfaces in systems such as Mach [Young et al. 87] and Spring [Khalidi and Nelson 93], the IMemoryObject interface does *not* contain the methods necessary to support external IMemoryObject implementations. The Map method fails if the supplied IMemoryObject has a "foreign" implementation. This technique of using an abstract interface but only supporting internal implementations greatly simplified our design.

This still allows the possibility of supporting more sophisticated memory objects with additional functionality and interfaces. For example, our system could be extended to support mapped files, via a method that takes an IFile and returns an IMemoryObject. Similarly, it could support external pagers.

Rialto's copy-on-write mappings are asymmetric. That is, if a region in address space A is mapped into address space B copy-on-write, and A writes to a page in the region, then B sees A's change. If B writes to the mapped region then a copy-on-write fault occurs and A does not see B's change. This contrasts with symmetric copy-on-write, which insulates both parties from each other's changes.

In our experience asymmetric copy-on-write mappings are significantly simpler in implementation. Our loader implements shared images by keeping an original writeable copy of the image in a private address space, and giving user processes copy-on-write mappings. The asymmetric semantics do not make it easy to transparently replace copying with copy-on-write, but this hasn't been a problem: Win32 has CreateProcess instead of fork/exec and we optimize IPC in other ways.

In general, mappings do not hide changes to the underlying memory object. For example, if a region in one process is mapped into a second process, and the first process then changes or deallocates its region, those changes are visible to the second process. Accesses to a mapping may cause an exception if the source address space region is invalid.

No-exception mappings insulate the users of a memory object from bad behavior. For example, if a server maps a user buffer, then it might be vulnerable to the user process deallocating its buffer and rendering the server's mapping invalid. The no-exception attribute protects the server in this scenario; instead of getting an exception the server is given a temporary mapping of a scratch page. (See Section 3.4.1.) No-exception mappings are a simple, low-overhead way of handling this scenario because in the common case there are no exceptions. The alternatives, like requiring the server to access the mapping inside a try/except exception handler, impose overhead on the many users of mappings.

2.3 Device Drivers

Device drivers have some special responsibilities and privileges with respect to virtual memory. This includes restrictions on the memory accessible from device interrupt handlers and several additional APIs available only inside the kernel process. In almost all respects device drivers use the same location-transparent kernel interfaces as other loadable kernel services and user processes. However device drivers can install a device interrupt handler, they can access physical memory, and they can issue IO instructions.

Device interrupt handlers can only access "unmapped" virtual addresses. (Unmapped addresses translate directly to physical addresses; see Section 3.2.1.) Only the kernel address space supports unmapped memory and the implementation limits the location and size of unmapped memory.

The kernel reduces this burden on device drivers in two ways. First, it is possible to mark drivers so that the loader automatically places them in unmapped memory. This takes care of code and static data references in the interrupt handler. Second, the kernel provides a driver heap in addition to the kernel's normal process heap. The driver heap is limited in size but it is located in unmapped memory; this provides for the interrupt handlers' dynamic data references.

```
SCODE MapPhysicalMemory(ADDRESS Phys, ADDR_SIZE Size, ADDR_FLAGS Flags,
                        [out] ADDRESS *pVirt);

SCODE GetPhysicalAddress(ADDRESS Virt, ADDR_SIZE Size,
                        [out] ADDRESS *pPhys, [out] ADDR_SIZE *pContig);

SCODE ReleasePhysicalAddress(ADDRESS Phys, ADDR_SIZE Size);
```

Figure 3: Device Driver Interface

Device drivers can use several functions to access physical memory; see Figure 3. Drivers for memory-mapped devices use MapPhysicalMemory; it takes protection flags and attributes just like IAddressSpace::Map, and IAddressSpace::Deallocate does any necessary cleanup. Drivers that need to convert a virtual address to physical for DMA use the Get/ReleasePhysicalAddress functions. GetPhysicalAddress "locks" as much corresponding physical memory as is contiguous and returns a physical address and size. (If the virtual region was not completely contiguous physically, then multiple GetPhysicalAddress calls are necessary.) ReleasePhysicalAddress releases the lock. The counting lock prevents the physical memory from being reallocated while a DMA is in progress.

3. Implementation

The virtual memory implementation uses a virtual TLB (VTLB) architecture with hybrid mutex/spinlock synchronization, a user/kernel address bias, and fault handling via an interrupt stack and a helper thread. The VTLB architecture bounds the amount of memory consumed by machine-dependent mapping structures such as page tables. The synchronization design meets the kernel's real-time requirements; the VM code is completely interruptible and is non-preemptible only in short code sections. The user/kernel address bias allows code pages to be shared between user and kernel mode, while still giving the kernel efficient access to the user address space. The fault handler protects the kernel from invalid user addresses via a scratch page and it supports auto-commit faults on kernel-mode thread stacks.

3.1 VTLB Architecture

The Rialto kernel uses a virtual TLB (VTLB) to implement virtual memory. The VTLB caches virtual-physical address translations, much like Mach's pmap layer [Rashid et al. 87] but with the distinction that any entry in the VTLB may be flushed at any time. The VTLB approach saves memory because it bounds the size of the machine-dependent mapping data structures. Like the pmap layer, the VTLB promotes portability by insulating the machine-independent VM code from the details of the hardware MMU architecture.

The kernel's machine-independent address-space (AS) data structures contain the only true representation of address spaces, including the kernel address space. Misses in the VTLB search the AS data structures and then enter new mappings into the VTLB. (The following subsection considers the synchronization and recursion issues that arise when handling misses.) The current implementation uses self-balancing binary trees for the machine-independent AS representation.

The VTLB interface (see Figure 4) provides a function VTLBMiss for machine-dependent code to call up to machine-independent code. VTLBMiss has three different outcomes, indicated by the status code. First, the AS data structures might contain a valid mapping for the miss address, in which case VTLBMiss calls VTLBEnter. Second, this might be a copy-on-write or auto-commit fault, in which case the machine-dependent code must initiate fault processing. Third, this might be an invalid memory access, in which case the machine-dependent code must initiate exception processing.

```
SCODE VTLBMiss(AS *pAS, ADDRESS Miss, ADDR_FLAGS Flags);

void VTLBEnter(AS *pAS, ADDRESS Miss, ADDRESS Virt, ADDRESS Phys,
                        ADDR_SIZE Size, ADDR_FLAGS Flags);
void VTLBRemove(AS *pAS, ADDRESS Virt, ADDR_SIZE Size);
void VTLBFlush(AS *pAS, ADDRESS Phys, ADDR_SIZE Size);
void VTLBCleanup(AS *pAS);
void VTLBSynchronize(void);
```

Figure 4: VTLB Interface

The interface also provides five functions for machine-independent code to manipulate the VTLB. VTLBEnter supplies a virtual-to-physical mapping for a contiguous region that contains the miss address. Note that there is no way to "wire" a mapping in the VTLB. The VTLBRemove and VTLBFlush functions may delay flushing the hardware TLB until VTLBSynchronize is called.

The VTLB design allows for either a software TLB cache or a page table page cache. Our preferred VTLB implementation on a machine with software TLB miss handling just extends the hardware TLB, increasing its effective size with a larger software cache. For example, our MIPS port uses a two-way set-associative cache with 512 entries. An alternative implementation could use page tables and recycle page table pages to limit their space overhead. Our Intel x86 port uses this strategy because the processor only supports two-level page tables.

There is a space/time trade-off available in the size and type of the VTLB. First, there is the obvious trade-off that a larger VTLB (bigger cache, or more page table pages) results in fewer VTLB misses. In addition, there is a trade-off between a software TLB cache or cache of page tables. Indexing into a page table is faster than accessing a set-associative cache. Our MIPS implementation takes 26 to 37 instructions to access its two-way set-associative cache when there is a hit, while a linear page table takes 9 instructions if there is a hit. However, page table pages have many unused or underutilized entries. When a page table page is recycled, the cache loses all of its entries even if some of them are frequently accessed. Consequently, a page table cache must consume much more memory to achieve the same hit rate as a software TLB cache organization.

3.2 Synchronization and Recursion

Because the VTLB is a pure cache (any entry may be recycled at any time), it presents several synchronization and recursion problems:

- Infinite recursion of VTLB misses.
- The VM code should be interruptible, yet interrupt handlers must access virtual addresses and hence risk VTLB misses when the VM data structures are in an inconsistent state.
- Similarly, the VM code should be preemptible, yet non-preemptible code must access virtual addresses and risk VTLB misses. A VTLB miss in the non-preemptible code can't wait to acquire a lock held by VM code that was preempted.

Our implementation uses unmapped memory to solve the first two problems and hybrid mutex/spinlock locking of the AS data structures to solve the third problem.

3.2.1 Unmapped Memory

Unmapped memory is kernel virtual memory that one can access without causing a VTLB miss. Unmapped memory prevents infinite recursion of VTLB misses and it prevents VTLB misses in device interrupt handlers. The MIPS architecture implements unmapped memory in hardware; Rialto extends the concept in a machine-independent manner. Unmapped memory may require additional machine-dependent mapping structures beyond the VTLB mapping cache, but these additional data structures are also bounded in size.

The MIPS hardware takes away 1GB of kernel virtual address space to map physical memory twice; 0.5GB for cached accesses and 0.5GB for uncached accesses. Accesses to this 1GB region bypass the MIPS TLB; the hardware directly converts the virtual address to a physical address by clearing the three high bits of the address.

On processors without hardware support for unmapped memory, the software TLB miss handler can implement the direct translation from unmapped addresses to physical addresses. Our Intel x86 implementation initializes a few page table pages (the number depends on the size of physical memory) at boot time to implement unmapped memory. These page table pages may not be recycled; they are a constant overhead for the system.

The VM implementation prevents infinite recursion of VTLB misses by distinguishing between user VTLB misses and kernel VTLB misses. A user VTLB miss may cause a kernel VTLB miss, but the kernel VTLB miss handler only accesses unmapped memory and so it stops the recursion.

The kernel VTLB miss handler makes accesses to three kinds of memory:

- Kernel static code and data. The boot process places the kernel image in unmapped memory.
- Dynamic data. Nodes in the kernel AS data structure are allocated out of a heap that resides in unmapped memory. User AS nodes and other kernel data structures come out of the normal kernel process heap.
- Interrupt stack. The VTLB miss handler does not run on a normal kernel thread stack. Instead, the kernel has a small (one page) stack, the interrupt stack, allocated in unmapped memory for the VTLB miss handler, interrupt handlers, and trap handlers.[3]

The depth of the interrupt stack is bounded because at most four levels execute on it: a trap handler accessing user memory, a user VTLB miss handler accessing kernel memory, a kernel VTLB miss handler accessing unmapped memory, and device interrupt handlers.

Device interrupt handlers are also constrained to avoid VTLB misses. This allows the VM code, including the VTLB miss handler, to be completely interruptible. The Reserve method on the kernel address space supports an attribute indicating that it should return an unmapped address, safe to access in an interrupt handler. Similarly, the MapPhysicalMemory supports the “unmapped” attribute so that the driver’s interrupt handler can access the device memory.

3.2.2 Hybrid Locking

Rialto protects the AS data structures with both a mutex and a spinlock to allow non-preemptible code to take VTLB misses. (The unmapped memory solution doesn’t extend easily to non-preemptible code because it’s harder to restrict the memory accesses performed by non-preemptible code.) Consequently, operations on the AS data structures are preemptible except for short AS-mutation functions.

Most code in the kernel executes preemptively in a thread context, but there is a small core of non-preemptible code. The non-preemptible code consists of scheduling and synchronization code, interrupt handlers, the VTLB functions, the AS-mutation functions, and the IPC path. The bulk of the VM and IPC code (and all other code in the kernel or loaded into the kernel process) executes preemptively.

The non-preemptible code should be regarded as operating outside of any particular thread context. Non-preemptible code operates *on* threads instead of running as a thread. The scheduler may choose to context-switch when leaving a non-preemptible code section to guarantee that the appropriate thread is running at all times.

The kernel supports two forms of exclusive lock, mutexes and spinlocks. Preemptible code uses mutexes to guarantee exclusive access to data structures. The scheduler implements priority inheritance to prevent a

[3] In a multi-processor implementation, each processor would have its own interrupt stack.

high-priority thread from waiting indefinitely for a mutex held by a low-priority thread. Non-preemptible code uses spinlocks to protect data structures. The spinlocks are only really necessary on a multiprocessor, so our uniprocessor spinlock implementation just asserts that it is being used correctly.

The data structure representing an address space has both a mutex and a spinlock. The mutex protects operations on the address space; the IAddressSpace methods acquire the mutex. While the mutex is held the operation may inspect and traverse the data structure. It may allocate and deallocate data structure nodes or physical memory. The spinlock protects mutations of the AS data structure. During a potentially lengthy IAddressSpace method, the implementation makes one or more calls to short AS-mutation functions to actually change the data structure—insert or delete a node. The AS-mutation functions are not preemptible so they can take the spinlock. In the case of the kernel address space, the AS-mutation functions also need to switch to the kernel's interrupt stack. This prevents a kernel VTLB miss from occurring while the kernel AS data structure is in an inconsistent state.

The VTLB miss handler, which is not preemptible, takes the spinlock on an AS when it searches the AS for a mapping. This allows the VTLB miss handler to search an AS even though some other thread is in the middle of an IAddressSpace method, preparing to modify the address space. The spinlock guarantees that the VTLB miss handler always finds the AS in a consistent state.

3.3 User/Kernel Address Bias

The VM system allows server threads in the kernel process to access their client process's memory by adding a bias to the user address to obtain a kernel address. The bias allows the kernel process to share DLLs with user processes. The kernel's IPC mechanism uses this capability to optimize calls into the kernel process while preserving transparency. Efficiently implementing the user/kernel bias on the Intel and MIPS architectures presents some challenges.

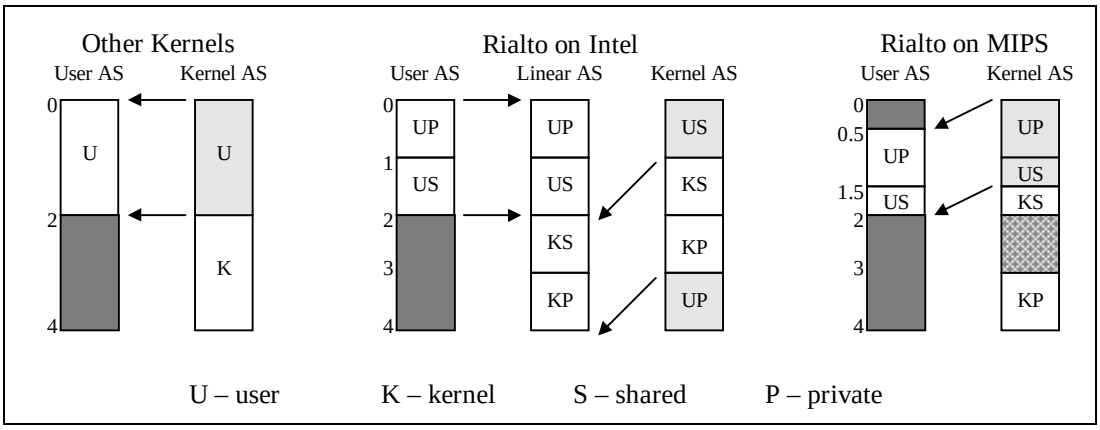


Figure 5: User/Kernel Address Bias

Many operating systems divide the 4GB address space into two disjoint regions. Conventionally the low 2GB is used for user address spaces and the high 2GB for the kernel address space. With this arrangement one can trap into the kernel from a user process, switch to kernel addresses, and still have access to user addresses without having to change hardware MMU contexts.

The conventional approach does not support sharing DLLs between user and kernel processes. A DLL must be linked or fixed-up to make code and static data references to fixed virtual addresses. Because the user and kernel virtual address spaces are disjoint, the DLL's code pages can't be shared between user mode and kernel mode.[4]

[4] The systems in which we were interested don't support position-independent code.

Rialto solves this problem by making the user and kernel address spaces overlap; see Figure 5. The kernel can still address the entire user address space, by adding a bias to user virtual addresses. The overlap region is normally only used for shared DLLs. Although we call the overlap region "shared," this only reflects its conventional use—it is normal per-process memory. Processes are free to put that region of their address space to other uses, but it may inhibit their use of some shared DLLs because DLLs are fixed up to run at a particular virtual address.

The IPC system uses this capability to transparently optimize calls on kernel objects. In general the IPC mechanism must copy arguments from the caller address space to buffers on the callee stack. However, for calls into the kernel it passes pointers through without copying; it just performs a bounds-check and then adds the bias. The bounds-check prevents a malicious user process from tricking the kernel by passing an address that when biased turns into a valid kernel address. (The following subsection explains how the VM system protects the kernel from invalid user addresses.)

Our Intel x86 port takes advantage of the x86 segment registers to implement the user/kernel address bias. The x86 segments translate from virtual addresses to linear addresses, then two-level page tables translate from linear addresses to physical addresses. The user mode segments have a zero base and 2GB limit. The kernel mode segments have a 1GB base and 4GB limit, so they "wrap around." The processor automatically switches segments when changing modes. As in other systems, the high half of the first-level page table page for every user process is identical and points to the kernel's second-level page table pages. This approach avoids changing page tables on user/kernel transitions, which is important because that flushes the TLB on x86 processors.

Our MIPS port implements the user/kernel address bias with paired address space IDs (ASIDs). The MIPS TLB tags entries with an ASID and a global bit which if set overrides the ASID. A processor register contains the currently active ASID. We assign every user address space an odd/even pair of ASIDs for user mode access and kernel mode access. When a trap into the kernel occurs, the handler must change the current ASID before the kernel can correctly access addresses below 2G. Device interrupt handlers and the VTLB miss handler only access addresses above 2GB (the processor's unmapped memory region extends from 2GB to 3GB), so they don't need to change the ASID. As in other kernels, the TLB entries for addresses above 3GB have the global bit set so they are always active. The paired-ASID implementation can suffer extra TLB misses compared to the conventional approach, because user mode and kernel mode accesses to the same user page use two TLB entries instead of one.

3.4 Fault Handling

A fault occurs when the VTLB miss handler fails to find a virtual to physical mapping, either because the accessed memory is invalid or because the memory is copy-on-write or auto-commit. The fault handler has two novel aspects: it transparently protects kernel code from invalid user addresses with a *scratch page* mechanism, and it allows for growing kernel-mode thread stacks on demand via auto-commit faults. The latter feature takes special attention on the Intel x86 architecture.

3.4.1 Scratch Pages

The scratch page mechanism protects the kernel from invalid user addresses, passed into the kernel as an argument to a call on a kernel object. (The same scratch page mechanism also implements no-exception mappings.) The VM system reserves a single page for use as a scratch page. When the kernel accesses an invalid user address, the fault handler enters a mapping for the scratch page. The end result is that the kernel call reads or writes the scratch page instead of getting an exception.

There are two complications. First, the scratch page mappings should not be visible to other threads in the user process. (For example, they should not see a read-only page being replaced with the read/write scratch page.) Therefore at every context-switch all mappings of the scratch page are removed and the scratch page itself is erased if it was used. Second, the user thread that passed the invalid argument to the kernel should get an exception. The fault handler marks the user thread as having a pending exception, and when the thread's kernel call returns the exception is raised in the user thread's context. In some

cases this means that the exception is raised after the kernel call has finished instead of before it starts, changing the semantics of erroneous calls.

Other operating systems typically use copyin/copyout or try/except primitives to access user addresses inside the kernel. The scratch page mechanism is preferable to these alternatives because it imposes very little overhead—quick checks in the context-switch and kernel call return path—in the common case that no faults occur. The alternatives require extra complexity in every kernel method implementation. Putting the pointer validation in each kernel method also makes it difficult to call those methods transparently from inside the kernel process, because the validation must be done only for user clients and the kernel object doesn't know if it's being called directly by a user thread or indirectly from another kernel object implementation.

3.4.2 Growing Kernel Thread Stacks

Growing kernel thread stacks dynamically via auto-commit faults presents several problems. First, when the auto-commit fault happens the thread can not handle the auto-commit fault for itself. Second, care must be taken to avoid an auto-commit fault while holding any locks necessary to resolve the auto-commit fault. The Intel x86 architecture presents a special problem, because the processor wants to push state on the kernel stack when it is not valid.

The kernel handles auto-commit faults with a kernel helper thread. Only preemptible code can take auto-commit faults. When a thread takes an auto-commit fault, the thread is suspended and the helper thread allocates the physical page, updates the thread's address space, and then puts the thread back in the ready queue. In practice we have found that the single helper thread is not a bottleneck but a multiprocessor implementation might benefit from multiple helper threads.

Probing prevents auto-commit stack faults in code that can not tolerate such faults. For example, the thread might hold the mutex on the physical memory pool, or the mutex on the kernel address space, or it might have called into some non-preemptible code. Before taking one of those mutexes, or entering non-preemptible code, the kernel probes 2K below the current stack pointer to force the potential auto-commit fault to happen early. A probe count prevents probing inside nested calls. This works because the code in question uses a bounded amount of stack, and device interrupts and VTLB misses execute on the interrupt stack instead of the current thread stack.

Rialto can recycle unused stack pages. Normally it freely reclaims any stack page below a thread's current stack pointer. For threads with a positive probe count (which are always kernel threads), it avoids reclaiming stack pages within 2K of the current stack pointer.

The Intel x86 architecture makes it difficult to handle stack faults inside the kernel. The x86 has four processor modes, or rings. Most operating systems run user code at ring 3 and kernel code at ring 0 because only ring 0 code can access some processor control registers. In particular, only ring 0 code can change the page table register for a context-switch. The difficulty is that at ring 0, the processor pushes some information on the current stack in response to any faults or interrupts. If the current stack pointer is invalid because this is an auto-commit stack fault, the processor takes an unrecoverable "double-fault."

To avoid this problem, we run kernel threads at ring 1. Ring 0 code only runs on the interrupt stack. The processor automatically switches to the interrupt stack when it transitions to ring 0 for device interrupts or VTLB misses. To avoid extra ring transitions, the IPC code optimizes calls on kernel objects to trap directly from ring 3 to ring 1 and return from ring 1 to ring 3. This works because we don't change the page table register for user/kernel calls. Synchronization calls, which are likely to result in a context-switch, trap to ring 0.

4. Measurements

We measured several aspects of the virtual memory system, using the MiTV system and applications. The support for sharing DLLs between user processes and the kernel saved 480K in our test scenario. The

hybrid mutex/spinlock synchronization design met our sub-millisecond preemptibility requirement. The VTLB architecture bounded the number of page table pages with very little performance penalty.

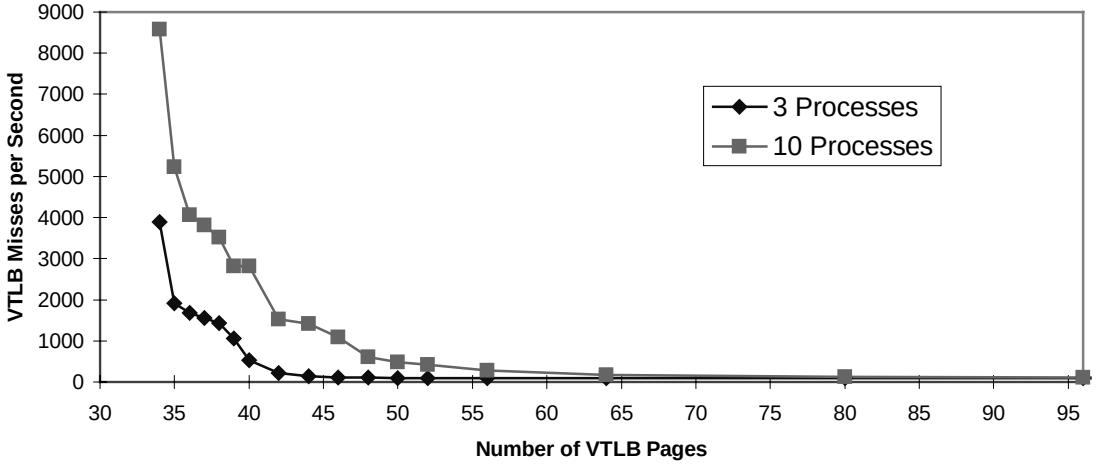
We used a development version of the MiTV system for these measurements. The development system had a single MiTV set-top box connected via private Ethernet to a single head-end server running Tiger and the other MiTV head-end services. The Tiger system delivered 2Mb/s MPEG1 over the Ethernet. The server did not supply broadcast channels and it used a dummy billing system and program data base. In contrast, the production MiTV system used ATM with 6Mb/s MPEG2 and broadcast channels encoded in real-time. We used a standard MiTV set-top box (75MHz Pentium processor, split 8K instruction and data caches on-chip, no external cache, PCI bus) configured with 16M of system memory.

Our test scenario exercised the Navigator, Electronic Program Guide (EPG), and Movies on Demand (MOD) applications for approximately 330 seconds. The scenario changed channels several times, scrolled around in EPG, and purchased and played a short video clip with MOD. Navigator was active throughout the scenario; it controls channel-changing and much of the work of displaying video actually happens in Navigator's address space. A "smart monkey" test application running on the head-end drove our test scenario by supplying hand-controller button events at timed intervals.

Rialto's ability to share DLLs between user processes and the kernel saved 480K in shared code pages in this scenario. The scenario used 42 shared DLLs across all processes. The kernel process loaded 55 DLLs; of these 16 were drivers or statically linked with drivers, 13 were constituents of seven services that could potentially load into user processes, and 26 were DLLs that shared 120 code pages with other processes. Loading the seven services into user processes, the kernel used 13 shared DLLs with 103 code pages.

The hybrid mutex/spinlock synchronization design met our goal of sub-millisecond preemptive scheduling granularity. The test scenario made approximately 30,000 IAddressSpace method calls. The average duration of an IAddressSpace call was 169µs; the maximum duration of any call was 26209µs. These are thread-based times that account for preemption but not device interrupts. In contrast, the non-preemptible AS-mutation functions were called approximately 130,000 times with an average duration of 14µs and maximum duration of 506µs. The non-preemptible VTLB functions were called approximately 50,000 times with an average duration of 33µs and maximum duration of 4729µs. (Only twelve calls took longer than 500µs. We have not yet investigated these outliers.)

Figure 6: VTLB Performance



To quantify the performance penalty of bounding the VTLB size, we measured the number of VTLB misses that occurred in two variations of the test scenario, first with three processes (the kernel, Navigator, and EPG or MOD) and the second with ten processes. We "artificially" created seven more processes by moving all the kernel-loadable services into separate processes. Figure 6 shows the effect of varying the number of VTLB pages on the number of VTLB misses. The VTLB pages include the

VTLB's page table page cache, non-reclaimable page table pages that implement unmapped memory in the kernel address space, and some miscellaneous VTLB data structures. In the three process scenario, 64 VTLB pages sufficed to prevent recycling of page table pages. In the ten process scenario, this required 120 VTLB pages. Bounding the VTLB to 64 pages increased the VTLB miss rate for the ten process scenario from 92/sec to 175/sec. The average VTLB miss latency increased from 41µs to 47µs (due to the cost of recyling page table pages), so the CPU overhead for VTLB misses increased from 0.37% to 0.82%.

5. Related Work

Our VTLB architecture owes a great deal to the Mach pmap layer [Rashid et al. 87]. Like the VTLB, the Mach pmap caches virtual-to-physical mappings and insulates the machine-independent VM code from the details of the hardware MMU. Unlike our VTLB design, the pmap layer does *not* allow kernel mappings to be recycled. This avoids the synchronization and recursion problems that we solved. However, it means that a pmap implementation can not use a simple software TLB as its primary data structure.

[Bala et al. 94] optimized the MIPS R3000 pmap implementation by adding a software TLB as a first-level cache in front of the pmap's page table pages. This achieved a useful performance improvement, but it did not address our goal of eliminating or reducing the number of page table pages.

To our knowledge, no other operating system allows kernel thread stacks to grow on demand. Windows NT 4.0 provides an approximation: it will grow a thread's kernel-mode stack in some circumstances. Windows NT gives all threads an 8K non-pageable kernel-mode stack. Some NT system calls make call-backs to user mode. Because the call-backs may in turn make nested system calls, the call-back operation checks how much kernel stack is left unused, and if necessary allocates and wires down more pages to guarantee subsequent nested calls 8K of kernel stack.

Other operating systems (for example, Chorus [Armand 91]) allow system services to be loaded at run-time into either a user process context or a kernel context. However, we are not aware of any other operating system that allows user process and kernel process components to share dynamically loaded libraries with per-process static data.

Commercial real-time operating systems provide very little virtual memory support. For example, OS-9 [Heilpern 94] has implementations for Intel x86, Motorola 68K, and PowerPC machines. It gives the application programmer malloc and free heap-management APIs. For production use, OS-9 applications can run on a version of the kernel without MMU support.

6. Conclusions

We have designed and implemented a virtual memory system suitable for consumer appliances with limited memory, multimedia and real-time requirements, and no requirement for paging. The Rialto kernel with this VM system has been successfully deployed as part of Microsoft's interactive TV system.

The virtual memory system has several novel attributes. The interfaces provide control over zero-filling of memory and have explicitly non-atomic failure semantics. It uses a memory object abstraction for mapping, but only supports internal memory object implementations, a significant simplification over other approaches. The implementation uses a VTLB architecture that bounds the memory consumed in mapping data structures like page tables and it solves the synchronization and reentrancy problems that arise as a consequence, in a way that satisfies our requirements for preemptibility and interruptibility. It uses a user/kernel address bias to allow sharing of DLLs between user mode and kernel mode while still providing efficient kernel access to user-mode data. Finally, it supports growing kernel-mode thread stacks on demand with auto-commit faults.

Acknowledgments

We thank Alessandro Forin for the initial Intel x86 port and VTLB implementation. Joe Barrera, Mike Jones, and other members of the Rialto group also contributed to our design and implementation.

References

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

[Bala et al. 94] Bala, K., Kaashoek, M. F., Weihl, W. E. 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.

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

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

[Heilpern 94] Heilpern, M. The OS-9 Primer. Microware Systems Corporation, Des Moines, IA, 1994.

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

[Jones et al. 95] Jones, M. B., Leach, P. J., Draves, R. P., Barrera, J. S. Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 12–17. May 1995.

[Jones 96] Jones, M. B. The Microsoft Interactive TV System: An Experience Report. Submitted for publication, 1996.

[Khalidi and Nelson 93] Khalidi, Y. A. and Nelson, M. N. A Flexible External Paging Interface. In *Proceedings of the USENIX conference on Microkernels and Other Architectures*, September 1993.

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

[Microsoft 93] Win32 Programmer's Reference. Microsoft Press, Redmond, WA, 1993.

[Rashid et al. 87] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., Chew, J.. 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, October 1987.

[Wahbe et al. 93] Wahbe, R., Lucco, S., Anderson, T. E., Graham, S. L. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

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

[Young 89] Young, M. W. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, November 1989.