Devirtualizable Virtual Machines
Enabling General, Single-Node, Online Maintenance

David E. Lowell, Yasushi Saito, and Eileen J. Samberg
Internet Systems and Storage Laboratory
Hewlett-Packard Laboratories
Palo Alto, California

david.lowell@hp.com, yasushi.saito@hp.com, eileen.samberg@hp.com

ABSTRACT

Maintenance is the dominant source of downtime at high availability sites. Unfortunately, the dominant mechanism for reducing this downtime, cluster rolling upgrade, has two shortcomings that have prevented its broad acceptance. First, cluster-style maintenance over many nodes is typically performed a few nodes at a time, making maintenance slow and often impractical. Second, cluster-style maintenance does not work on single-node systems, despite the fact that their unavailability during maintenance can be painful for organizations. In this paper, we propose a novel technique for online maintenance that uses virtual machines to provide maintenance on single nodes, allowing parallel maintenance over multiple nodes, and online maintenance for standalone servers. We present the Microvisor, our prototype virtual machine system that is custom tailored to the needs of online maintenance. Unlike general purpose virtual machine environments that induce continual 10-20% overhead, the Microvisor virtualizes the hardware only during periods of active maintenance, letting the guest OS run at full speed most of the time. Unlike past attempts at virtual machine optimization, we do not compromise OS transparency. We instead give up generality and tailor our virtual machine system to the minimum needs of online maintenance, eschewing features, such as I/O and memory virtualization, that it does not strictly require. The result is a very thin virtual machine system that induces only 5.6% CPU overhead when virtualizing the hardware, and zero CPU overhead when *devirtualized*. Using the Microvisor, we demonstrate an online OS upgrade on a live, single-node web server, reducing downtime from one hour to less than one minute.

Categories and Subject Descriptors

K.6.3 [**Management of Computing and Information Systems**]: Software Management - *software maintenance*; D.2.9 [**Software Engineering**] Management - *software configuration management*; D.4.5 [**Operating Systems**]: Reliability

General Terms

Reliability, Management, Performance

Keywords

Planned downtime, online maintenance, availability, virtual machines

ASPLOS'04, October 7-13, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-804-0/04/0010...$5.00.

1. INTRODUCTION

The impact of downtime on customers, employees, and stock prices can be profound. As a result, businesses increasingly construct their computing infrastructure to minimize downtime. Although unplanned downtime from failures is widely addressed through deployment of redundant hardware, *planned downtime* remains a challenging problem with few practical solutions.

Planned downtime results when critical applications must stop to allow for system maintenance, such as adding or replacing hardware, upgrading or patching the operating system, or reconfiguring or upgrading applications. Sites typically deal with planned downtime by batching maintenance events into *maintenance windows* scheduled when they can most afford to stop their systems. However, businesses with high-availability applications typically view maintenance windows not as a solution to planned downtime, but as a cost that must be borne because most forms of maintenance are currently impractical to perform any other way. Furthermore, these companies increasingly demand the highest availability from their infrastructure, leaving fewer convenient windows of time for maintenance.

Although there is little published data on the matter, one study shows that planned downtime has become the dominant source of outage minutes at the higher end of the server market. That 1998 study by HP's NonStop Hardware Division surveyed 426 sites with high-availability applications for 12 months [Non98]. Those sites used servers from IBM, Sun, Compaq, HP, DEC, and Tandem. Of the 5921 outages reported in the survey, 75% were planned. For a subset of sites with detailed data, the survey showed this fraction was increasing over time, and that planned outages typically lasted twice as long as unplanned ones.

Although "hot swap" hardware components have helped with the hardware side, planned downtime for software maintenance is a major problem with few palatable solutions in sight. As things now stand, OS and application vendors must either re-engineer their legacy systems to support online patching and upgrade, or customers must depend on cluster mechanisms that let them perform upgrades slowly over many nodes. For sites with non-clustered servers, there is no way to perform most forms of software maintenance online, at any speed.

In this paper we propose a novel mechanism for performing general OS maintenance *online*, that works on even single nodes. Our technique is based on virtual machines.

Our mechanism lets an administrator run enterprise applications in one virtual machine (VM), while simultaneously upgrading the OS, reconfiguring software, or updating the application in a second VM. When the maintenance is finished, the administrator uses an application migration tool to move the running applications from the first OS instance to the second, or to redirect request traffic to the application already running in the second instance. By working without a spare node, this approach addresses the scalability problems of cluster-style *rolling upgrade*—administrators can upgrade many nodes in parallel. It also provides an online maintenance mechanism for non-cluster systems.

However, general-purpose virtual machine systems generate too much overhead to be useful for the occasional online maintenance. State-of-the art, commercial virtual machine software induces 10-20% overhead for typical enterprise workloads. We believe that enterprise customers will resist paying even a 10% performance penalty at times of peak load to let them perform the infrequent maintenance without downtime.

In this paper, we propose a virtual machine system specially designed for online maintenance. We describe the implementation of such a virtual machine system which we call the *Microvisor.* It allows the administrator to turn off virtualization during long, maintenance-free operation, and eliminate CPU overhead during these times. We call this operation *devirtualization*. At maintenance time, the administrator can turn virtualization on again for the length of the maintenance.

Past efforts to reduce the overhead of virtualization have compromised OS-transparency [Barham03, Bugnion97, Whitaker02]. In contrast, we focus on reducing overhead by eliminating the virtual machine features that are unnecessary for online maintenance. We do away with support for multiplexing more than two OS instances, sharing I/O devices between virtual machines, and oversubscribing memory. We did not however compromise OS-transparency: our virtual machine monitor remains OS-transparent, supporting an unmodified Tru64 Unix 5.1A kernel.

Using our system, we demonstrate a major OS upgrade performed on a web server while it continues to serve web requests. Without our system, this upgrade would have required an outage of around an hour. With our system, we are able to perform the upgrade with three short outages totaling 41 seconds of downtime, without a spare node, and without special upgrade support in the kernel or OS upgrade tools. This online upgrade is a challenging form of maintenance and there currently exists no practical way to perform it without our approach.

Our three main contributions are:

- We propose a novel technique for online maintenance using virtual machines.
- We build a prototype virtual machine monitor that is the first to support devirtualization. We show that tailoring our design to the minimum needs of online maintenance leads to a monitor with just 5.6% CPU overhead, without giving up OS-transparency.
- We demonstrate an online OS upgrade on a web server while it serves web traffic, without the use of a spare node.

A key component of the mechanism we propose for online maintenance is application migration, an area that has been well-studied over the years. In this paper we do not focus on application migration, and instead refer readers to the recent work by Osman et al. on the *Zap* process migration system [Osman02], and past work on cluster-style failover [Huang98, Zhou99].

2. EXISTING ONLINE MAINTENANCE TECHNIQUES

There exist several techniques for performing online maintenance without a virtual machine mechanism. We divide them into two broad classes: *general approaches*, and *subsystem-specific approaches*.

2.1. General approaches

The hallmark of general approaches to online maintenance is their ability to support online a wide variety of software and hardware maintenance, such as hardware addition and replacement, OS patches and upgrades, and system reconfiguration.

There are a number of existing techniques and systems that each provide a general approach to online maintenance, such as spare nodes, server clusters, soft and hard multiprocessor partitions, and loosely-coupled racks of servers with a parallel workload. We will refer to all these techniques as “clusters” because in practice “spare nodes” are often clusters of size two, and multiprocessor partitioning is often used to carve up a multiprocessor into a cluster. In each of these systems, spare compute resources are used to support the application during maintenance. For example, to perform a cluster *rolling upgrade*, the administrator can shift application load from one node onto others, take that node down for almost arbitrary hardware or software maintenance, then shift application activity onto it again and proceed to the next node [Mic00].

Cluster environments have some nice online maintenance properties. From the server vendor’s point of view, the single engineering investment in a cluster mechanism enables online maintenance across many hardware and software subsystems. For cluster customers, they are able to reduce planned downtime using clusters typically composed of inexpensive hardware.

Although clusters are finding acceptance in the marketplace, they are widely used for online maintenance only for trivially parallel applications (such as web servers). Even in multi-node environments with spare nodes, high availability sites with non-trivial applications usually opt for planned maintenance windows instead of rolling upgrade. The main reason is that multi-node maintenance can take a long time. Upgrading all *n* nodes in a cluster with *s* nodes worth of spare capacity takes $\frac{n}{s}$ iterations. To minimize the chance of an accidental, maintenance-induced failure, high-availability sites often mandate that the same crew of administrators finish a maintenance operation that began it. As a result, maintenance events must typically fit within an eight-hour work shift. For clusters of more than just a few nodes, this can be a difficult deadline to hit if *s* is much smaller than *n*. As a result, most high-availability sites do not take advantage of the maintenance properties of their clusters, and instead opt to perform maintenance more quickly in planned maintenance windows, despite the downtime that results. The NonStop availability survey supports this observation: there were 4443 planned outages in the survey despite the fact that many of the surveyed systems were high-availability clusters.

2.2. Subsystem-specific approaches

Server software and hardware vendors can also engineer online maintenance abilities into particular subsystems. For example, dynamic kernel module writers can design their module to be installable and replaceable at runtime. Similarly, object-oriented software systems can be designed so that constituent objects can be upgraded on the fly, as is done in IBM's K42 operating system [Appavoo03], and several user-level systems [Segal93, Hicks96].

Subsystem-specific online maintenance is also increasingly offered for hardware components. For example, on many mid-range and high-end servers, *PCI hot plug* allows PCI cards to be added or replaced without halting the operating system and applications. There exists similar support for disk hot swap in modern storage systems, and CPU and memory hot swap on some multiprocessors.

For software and hardware systems built from online-updatable components, performing maintenance by selectively updating a component or two is an easy and elegant approach. However, modern operating systems and commercial applications are large, and engineering teams are strapped. Redesigning these legacy systems from the ground up to provide this style of maintenance is simply infeasible. Even retrofitting support for hot swap of individual components is proving difficult. Furthermore, the engineering investment made to make one system component upgradable online often does not benefit other components.

3. ONLINE MAINTENANCE USING VIRTUAL MACHINES

Virtual machine systems operate at the powerful boundary between the operating system and hardware. From that privileged perch, the *Virtual Machine Monitor* software (VMM) can circumscribe the OS's view of hardware, encapsulate machine state, and allow multiple OS instances to run on a single node. The VMM monitors and modifies operating systems' interaction with hardware to provide each OS the illusion that it alone controls the hardware on which it runs [Goldberg74].

By letting two operating system instances run simultaneously on the same hardware, virtual machine systems can provide general online maintenance, in a manner analogous to the cluster rolling upgrade discussed in Section 2.1—essentially, administrators can *simulate* a spare node on demand. Using this technique, administrators can perform online OS and application upgrades, patches, and reconfiguration. However, unlike rolling upgrade, online maintenance using virtual machines works on even a single node, allowing maintenance of multi-node environments in fewer than $\frac{n}{s}$ iterations, or even in a single iteration. This maintenance mechanism can also enable online maintenance in single-node compute systems, such as workgroup or department servers. Even though they are not traditionally considered high-availability, planned downtime is still painful and disruptive in these standalone environments.

Figure 1 depicts an online maintenance session using virtual machines. The administrator would likely schedule this maintenance at an off-peak time to ensure enough idle cycles to allow for running multiple OS instances on the server. In this example, the server ordinarily runs a virtual machine monitor, an operating system, and an enterprise application. At the appointed hour, the administrator first starts up a second OS instance. As the original OS and application run, he or she brings the second OS to the upgraded state, perhaps by installing an upgrade to the OS, applying a patch, setting up a new version of the application, or reconfiguring other software. The administrator is free to reboot the second OS as necessary, as well as thoroughly test the new configuration. When the administrator is satisfied with the state of the software on the second OS, he or she can migrate the application's service from the first OS to the second in the manner employed by server clusters [Mic00, Huang98], or in the case of application maintenance, redirect request traffic to the upgraded application already running in the second instance. This migration can also be supplied by a transparent process migration system [Osman02].

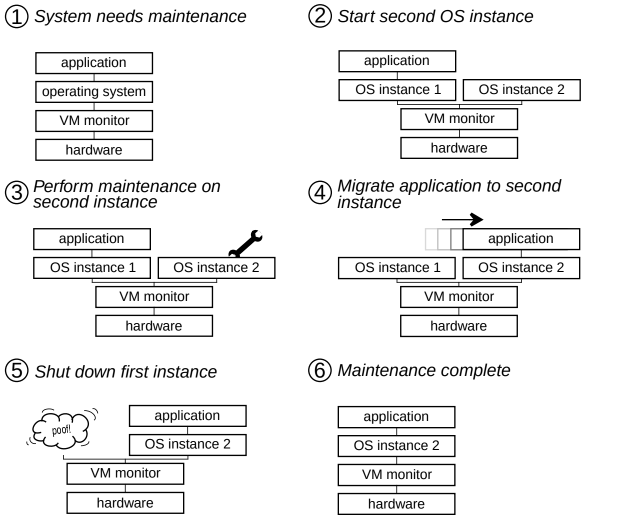

Figure 1: Online maintenance using virtual machines. The virtual machine monitor lets administrators simultaneously boot two OS instances: one supports the application, maintenance takes place in the other.

With the application safely running in the upgraded environment, the administrator can shut down the original OS and virtual machine. The only time application service stopped was during migration. Although the duration of this outage will depend on the application and the migration method employed, migration from one OS to another on the same physical machine can be made quite fast. For example, a migration mechanism could checkpoint to memory the application in the first OS instance, then with help from the VMM, remap the checkpoint pages into the second OS instance without copying. Approximating the page-table-rewriting rate as the 1 GB/s copy bandwidth of a modern system, we estimate that migrating the checkpoint state of a 512 MB process in this fashion would take only 1 millisecond (assuming 4 KB pages and 8 byte PTEs). The further details of how to migrate enterprise applications quickly from one virtual machine to another on the same hardware is an important research topic, but beyond the scope of this paper.

Current virtual machine monitors take over hardware control at first boot, and keep that control until the machine is reset. As the above example indicates, to allow for the possibility of later online maintenance, the administrator must run a virtual machine monitor on their system at all times. The monitor generates overhead, and with current virtual machine systems there is no way to avoid that overhead during maintenance-free execution.

To get a sense of the overhead imposed by a state-of-the-art, virtual machine system, we measured the slowdown induced by a commercial, non-hosted VMM, *VMware ESX Server* v2.1.0, running on a current HP Proliant DL360-G3 with 2.8 GHz Pentium 4-Xeon. Among the variety of benchmarks we ran, we found for non-math workloads, ESX Server generally slows execution by 10-20%. For many problem domains where the virtual machine layer's services are needed continually, this overhead is a small price to pay for the benefit it confers.

However, for online maintenance the virtual machine system is only needed from time to time. We believe customers will resist paying a 10-20% performance hit during periods of peak load to enable the occasional online maintenance.

As we will show, by paring down virtual machine functionality to the essence required for online maintenance, we can get this CPU overhead to zero and remove this impediment to adoption. We will describe our design for such a virtual machine system in the next section.

Overhead is not the only reason to forego using a general-purpose virtual machine system for online maintenance. Many of the features this generality implies, such as network and disk traffic shaping, quality-of-service guarantees, and sophisticated CPU schedulers, are not necessary for online maintenance. However, those features add development cost and complexity (and therefore licensing cost), increase exposure to bugs and security holes, and create more software to manage and update. In contrast, a simple, thin VMM tailored to the minimum needs of online maintenance can be more throroughly debugged, less expensively licensed, more efficiently managed, and less frequently updated, increasing its attractiveness for online maintenance.

4. DESIGN OF THE MICROVISOR

When beginning this project, we wondered whether we could build a virtual machine system that was useful for online maintenance, but that did not force the user to pay 10-20% overhead during long maintenance-free periods. We began by making the following three observations about the online maintenance using virtual machines depicted in Figure 1:

- In the common case, only one OS is needed
- The VMM supports at most 2 OS instances
- The two OS instances are part of the same administrative domain

Informed by these observations, we then designed a virtual machine system based on a very thin hypervisor. We have dubbed our system the *Microvisor.* At each step of the design and implementation, we adhered to the principle that our monitor should be as simple as possible, avoiding features and complexities not strictly necessary for online maintenance. Our design also reflects the necessity of keeping the implementation tractable for a small team.

The Microvisor virtualizes the CPU in order to run either one or two OS instances, but does not support running *n* instances. Supporting a small, fixed number of OS instances greatly simplifies our system's scheduling, memory management, internal data structures, and interrupt handling. We refer to the two OS instances as *OS-0* and *OS-1*. In our system, we treat OS-0 as the "production" OS instance, and we use OS-1 only during maintenance (see Section 5.2).

The Microvisor requires CPU virtualization when an OS boots, and when two OS instances run simultaneously. In the common case of running with a single OS, our system can cease its virtualization efforts entirely, then resume virtualization at maintenance time. Devirtualization of the CPU is possible because the Microvisor deceives the OS very little about the state of the machine, and in practice, these deceptions are only triggered during hardware discovery and when two OS instances share the CPU.

Traditional virtual machine monitors have to virtualize I/O devices to allow them to be shared by multiple OS instances. For online maintenance, I/O virtualization is less necessary since two OS instances rarely share the hardware. Thus, the Microvisor does not virtualize I/O resources. Instead, it partitions devices and dedicates them to each OS instance. Doing so simplifies devirtualization, avoiding the challenge of dynamically interposing device drivers under the OS.

Gigabit ethernet card	$20
4-port Gigabit ethernet card	$450
Gigabit ethernet switch port	$20
Fiber channel adapter	$175

Table 1: Typical retail commodity hardware prices found on pricegrabber.com. July 2004.

Consider a server with 16 PCI slots that is normally provisioned with 12 gigabit ethernet cards used for storage and network access. To allow for online maintenance using the Microvisor, the administrator would have to provision the server with some additional cards, and allocate additional ports on network switches. However, because the additional cards will be dedicated to a virtual machine that is only employed during maintenance, and maintenance is typically performed during off-peak times, those additional cards can be in a lower-performance configuration. So the administrator could fill three of the remaining slots with 12 network ports, using three, 4-port gigabit ethernet cards. The administrator may also be able to get away with fewer cards/ports by multiplexing network or disk traffic over a smaller number of network links during maintenance.

Similarly, the Microvisor also requires a boot volume for use during maintenance by OS-1. This additional boot volume can be a logical volume carved from a storage array. With sufficient support from the storage server, it could also be a copy-on-write replica of OS-0's boot volume. In our prototype system, we use a dedicated, direct-attached boot disk for each virtual machine. To minimize the management overhead of maintaining OS-1's boot volume, we clone OS-0's boot volume onto OS-1's disk on demand.

We believe the one-time cost of this extra I/O hardware is quite reasonable compared to the cost of adding a spare node to a standalone system (see Table 1). Adding just these devices to an existing node costs less, and takes less power, cooling, and rack space [Bose03].

Even when this extra hardware has to be added to many nodes in a cluster or data center, it still represents a small percentage increase in initial hardware investment (roughly 10-20%). This one-time cost should be offset by ongoing savings resulting from reduced downtime, which most surveys show costs from many thousands to millions of dollars per hour[1] [Patterson02b]. By allowing maintenance to proceed in parallel over many nodes, this extra hardware cost should also yield ongoing savings of administrator time, the bulk of which today is spent doing maintenance [Humpreys03]. Recent studies have shown that the hardware and software purchase price is a small (and shrinking) component of server total cost of ownership, and that exactly the costs addressed by the Microvisor—downtime and administrator time—have come to dominate [Levin02, Patterson02a].

[1] Most availability surveys report the cost of unplanned downtime, which likely costs more than a planned outage scheduled at a non-peak time. However, we have yet to find a study that distinguishes the cost of unplanned and planned downtime.

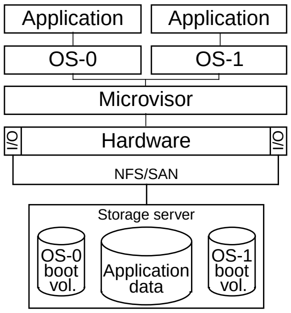

Figure 2: Preferred storage configuration. Each OS instance has a dedicated boot volume. A single application data store is accessible from either OS-0 or OS-1 over NFS or SAN.

Note that it is our intention that users provide both OS-0 and OS-1 with a duplicate *path* to application disk storage, but not to replicate the application data itself (see Figure 2). When maintenance takes place, the application migration mechanism can unmount the NFS or SAN storage in the source OS instance, and remount it for application use in the destination instance, as is done by Zap [Osman02]. Since in this scenario the application runs in only one place at a time, there is no sharing of application data between application instances.

Like I/O devices, the Microvisor also does not virtualize physical memory, since for online maintenance memory will not be shared in the common case. However, we deemed it excessively wasteful to partition memory as we do for I/O devices. Instead, our system allows the first OS to boot to discover all of the available memory in the system. When it comes time to boot the second OS, we employ an OS-level trick similar to VMware's "ballooning" device driver to let us borrow memory from the first OS for use by the second [Waldspurger02].

Since we do not virtualize physical memory, we cannot use page-table-based memory protection to isolate the physical address spaces of OS-0 and OS-1. We employ instead a form of software fault isolation in which we check all virtual-to-physical memory mappings before installing them in the TLB [Wahbe93].

5. IMPLEMENTATION

We have implemented a working prototype of the Microvisor for the HP Alphaserver DS20.

Our DS20 system contains one 500 MHz Alpha 21264 CPUs, 3 GB of SDRAM, and two PCI buses, each with a dedicated PCI host bridge. It runs an unmodified Tru64 Unix, version 5.1A.

5.1. Alpha PALcode

The Alpha 21264 microprocessor runs in the presence of the *Privileged Architecture Library,* or *PALcode.* The PALcode is a set of highly privileged software routines responsible for handling among other things, reset/initialization, TLB fills, interrupt dispatch, and context save and restore for processes. The PALcode is implemented as ordinary machine instructions that are executed in a special mode on the processor. The PALcode gets access to machine state that the operating system and user code cannot access. The Microvisor is implemented as an extension to the 21264 PALcode.

5.2. Assumptions

To ease the development of the Microvisor, we made two simplifying assumptions. First, our monitor is written for a single server platform and chipset which saved us from having to provide a hardware abstraction layer. Second, we assume OS-0 will always be the "production" OS instance, and OS-1 will only be used during maintenance. This assumption simplified devirtualization, allowing us to keep only OS-0's view of hardware completely aligned with reality. However, this also implies applications will be migrated twice during maintenance with the Microvisor: once onto OS-1 at the start of maintenance, and once to get them back onto OS-0 when maintenance on it is complete.

We made one complicating assumption: we assumed that we could consult the kernel sources to understand how it used the physical machine, but that no kernel modifications were allowed. The Microvisor works with a stock Tru64 kernel.

5.3. Microvisor implementation

We will next describe in some detail how the Microvisor is implemented. We will break our description down into the key components. In each section, we will highlight the key challenge we had to solve.

5.3.1. *CPU virtualization*

- Key challenge: schedule two OS instances on the same CPU, intercept physical memory access

The Alpha architecture is designed to be *virtualizable*—all accesses to privileged state by unprivileged code trap to a privileged routine [Goldberg74]. On Alpha the OS kernel is relatively unprivileged: the only right guaranteed the kernel by the architecture is to invoke privileged PALcode routines. All other special machine access the kernel may require is at the discretion of the PALcode. In the language of protection rings, the PALcode runs in ring 0, while the kernel runs in ring 1, and user code runs in ring 4. This is the normal protection arrangement for Unix on Alpha, even without a virtual machine monitor.

Thus, to virtualize our Alpha CPU, we did not need to run the kernel at a reduced privileged level as is required for most architectures [Goldberg74]. Tru64 already performs most privileged operations through explicit calls to the PALcode. The only additional privilege allowed Tru64 is the direct addressability of physical memory. PALcode can disable this access without running the kernel at a reduced privilege level.

When running two OS instances, the monitor's main task is to periodically save to memory the state of the currently scheduled virtual machine, flush the TLB, restore the saved state of the other virtual machine, and let its OS continue execution. We use a very simple round-robin scheduler that switches virtual machines every 10 milliseconds. Since the common case for our system will be to only run a single OS, we do not make any attempt to deschedule an idle OS instance. Therefore, when two OS instances run, each gets 50% of the CPU. We selected this simple scheduling scheme out of deference to our principle of providing the minimum functionality needed for online maintenance: since two OS instances run only rarely, a more sophisticated scheduler seems unnecessary. Other possible scheduling approaches are well-known [Chapin96].

Tru64 Unix keeps some data structures in physical memory. It accesses these structures as well as I/O space via *KSEG addresses*. The kernel can issue loads and stores to a KSEG address just like a normal virtual address, except a bit pattern in bits <42:41> tells the processor that bits <40:0> are a valid physical address and should bypass the TLB.

Our monitor must virtualize KSEG accesses in order to partition I/O devices, and trap and emulate accesses to certain hardware registers. During initialization, it configures the CPU to disable KSEG addressing which causes the kernel's KSEG accesses to be treated by the processor as virtual accesses. Those accesses (at first) miss in the TLB and trap to the Microvisor. Our monitor inspects each faulting address. If it is to physical memory, it first checks that the page being accessed is assigned to the currently executing OS (see Section 5.3.3), then creates a page table entry (PTE) for that page mapping it to the physical page frame in bits <40:13>. It then puts that PTE in the TLB and lets the faulting instruction restart. As long as that translation remains in the TLB, subsequent KSEG accesses to the same physical page do not trap to the monitor.

For KSEG accesses to I/O space, the monitor checks if the faulting address is one it must handle specially, such as an access to one of the server chipset's control status registers (CSRs), or to PCI space (see Section 5.3.2). In most cases, the monitor puts a translation into the TLB for these accesses as well.

5.3.2. I/O partitioning

- Key challenge: allow OS to discover only the I/O devices dedicated to it

As mentioned, the DS20 has two PCI buses, PCI-0 and PCI-1, and each bus has a dedicated PCI host bridge. Each host bridge maps the control registers for the devices on its bus into a dedicated 8 GB range of I/O address space.

Although there are many ways one could dedicate attached I/O devices to a particular OS instance, for simplicity we chose to partition bus-wise and to dedicate all the devices on PCI-0 to OS-0, and all the devices on PCI-1 to OS-1. By partitioning in this fashion, we avoid having to emulate virtual PCI host bridges for each virtual machine. Mid-range and high-end servers often have more than two PCI buses (for example, in the HP Superdome multiprocessor, each PCI slot is on its own PCI bus). In these systems, partitioning bus-wise need not require half of the server's I/O capacity.

Tru64 discovers what PCI hardware is attached by first figuring out how many PCI buses it has, then probing those buses one at a time. It reads a particular CSR on one of the platform's support chips to get the number of attached buses. To dedicate one of the two PCI buses to an OS instance, the Microvisor must trap accesses to this CSR and return a value stating there is only one PCI bus in the system. Tru64 assumes that if there is only one bus in the system, it must be PCI-0. Thus, to dedicate PCI-0 to OS-0, emulating this one CSR during OS-0's boot is sufficient.

However, when OS-1 boots and reads this CSR, it will find it has one PCI bus, and it too will assume its bus is PCI-0. Unless the monitor intervenes, OS-1 will probe PCI-0 and configure into its device list all of OS-0's devices. To force OS-1 to use PCI-1, the Microvisor traps OS-1's accesses to KSEG addresses in the PCI-0 address range, and installs in the TLB a translation from the PCI-0 address to the corresponding address in the PCI-1 address range.

Although the DS20 has two independent PCI buses, the two buses share one interrupt controller. To keep OS-1's devices from interrupting OS-0 when it runs, and vice versa, the Microvisor keeps the unscheduled OS's device interrupts masked for the scheduled OS's entire quantum. The Microvisor emulates the device interrupt masking register so that an OS cannot unmask interrupts from the other OS's devices. As with our scheduler, a more complicated interrupt demultiplexing scheme seemed unwarranted for online maintenance, since multiple operating system instances will rarely share the hardware. However, it would not have been difficult to reschedule an OS in order to deliver an interrupt to it, as most virtual machine monitors do [Sugerman01], although doing so would complicate virtual time (see Section 5.3.6).

In addition to its two PCI buses, the DS20 has an ISA bus containing several non-performance-critical devices, including: two serial ports, one parallel port, a floppy controller, and two IDE controllers. The ISA bus presents a challenge. We wanted each OS to have access to these ISA devices, mainly so that each OS could have a serial port for its console terminal. But the PCI-ISA bridge was hard wired onto PCI-0, and we were unable to find duplicate hardware on a card we could stick in PCI-1. So, we opted to map the region of I/O space used by these ISA devices into both OS-0 and OS-1, and we take steps to make sure both OS instances do not claim the same device. By rewriting the ISA device table set up in memory by the firmware, we keep OS-1 from discovering any ISA device but a single serial port. OS-0 is allowed to discover all remaining ISA devices. The Microvisor must also trap and emulate OS-1's probes of PCI configuration space to cause OS-1 to discover the PCI-ISA bridge attached to PCI-0.

Two aspects of our method of partitioning I/O devices weaken isolation between OS-0 and OS-1. First, mapping the same ISA I/O region into both OS instances allows one OS to address the other OS's ISA devices. Second, although we dedicate a PCI host bridge to each OS instance, there is one shared chip in the chipset ahead of the host bridges that demultiplexes accesses to I/O and memory, and translates the CPU's I/O space addresses into PCI and CSR addresses. Although these shared elements could allow one OS to accidentally or deliberately damage the other OS, for online maintenance this less of an issue. Since both OS instances share the same administrative domain, we do not have to worry about malicious attacks. And since the two OS instances only rarely share the hardware, accidental fault propagation is very unlikely.

5.3.3. Physical memory

- Key challenge: provide memory for OS-1 to use, without setting all of it aside when OS-0 boots

Modern operating systems use virtual memory, in which virtual addresses are translated by the OS and hardware into physical addresses. Traditional virtual machine monitors gain control over memory and allow resource sharing by imposing a second level of translation from physical addresses into "machine" addresses [Bugnion97]. This approach to memory sharing seemed like excessive engineering for the Microvisor, since multiple OS instances would rarely share the machine. Furthermore, having a second level of address translation would have forced us to virtualize DMA, and would have complicated devirtualization.

However, neither did we want to statically set aside half of memory for each OS at first boot, which would be too wasteful considering only one OS will run on the Microvisor most of the time.

Instead, we employ a dynamic sharing scheme that involves neither full memory virtualization, nor static partitioning. The Microvisor lets OS-0 discover all of the memory in the system. When it comes time to boot OS-1, we use a dynamic kernel module in OS-0 to borrow back half of the memory in the system, a technique similar to the “ballooning” kernel module pioneered by VMware [Waldspurger02].

The Microvisor’s ballooning module takes advantage of Tru64 and the firmware’s support for removing from circulation pages of memory that fail at runtime (similar support exists in HP-UX, AIX, and Solaris). To borrow memory for OS-1’s use, our ballooning module allocates a page using OS-0’s internal allocator. To prevent OS-0 from using the page and preserve it for OS-1 over reboots, our module marks the page “bad” in OS-0’s per-page data structures, then frees it onto a list of bad pages.

In addition to marking the page bad in OS-0’s memory management structures, our module also marks it bad in the Hardware Restart Parameter Block (HWRPB), a firmware data structure that survives reboots. When OS-0 reboots, it builds up a new copy of the bad page list based on the bad page bitmaps of the HWRPB, keeping it from using pages previously allocated to OS-1. Similarly, when a page is allocated in OS-0 for use by OS-1, it gets marked “good” in the version of the HWRPB exposed to OS-1 when it boots.

Once maintenance completes and OS-1 halts, our ballooning module returns OS-1’s memory to OS-0 by directly removing pages from its bad page list, marking them “good” both in the kernel’s structures and HWRPB, and placing them on OS-0’s free list.

Note that this mechanism for memory sharing can coexist with pages that really do fail at runtime. Our ballooning module would simply need to update OS-0’s bad page list and bitmaps with those pages that fail while assigned to OS-1. It would be straightforward to add this support to our prototype.

This approach to memory sharing does have one problem. Tru64 needs a certain minimum amount of contiguous memory to be able to boot (around 100 MB on our system). Since maintenance takes place infrequently, we expect OS-0 will run for quite some time before an administrator needs to boot OS-1. Therefore, without taking special steps, it is unlikely there will be sufficient contiguous physical memory available in OS-0 for OS-1’s needs. Traditional virtual machine monitors do not have this problem because they are able to make fragmented machine memory look physically contiguous through their extra level of memory translation. In our current implementation, we simply earmark 128 MB of contiguous physical memory for OS-1 when OS-0 first boots.

5.3.4. *Address space isolation*

- Key challenge: prevent one OS instance from modifying memory of another

It is important to prevent one OS instance from accidentally trashing memory owned by the other OS instance (deliberate attacks are unlikely, since for online maintenance, both OS instances will belong to the same administrative domain). Most VMMs provide this protection by virtualizing physical memory, providing each guest OS a completely separate physical address space. However, since we do not virtualize physical memory in the Microvisor, we protect each OS’s physical address space using an approach similar to “sandboxing” [Wahbe93]. Every time an OS’s virtual or KSEG memory access misses in the TLB, the TLB fill routine in the Microvisor first loads the PTE for that page from the OS’s page tables (or cooks up a PTE for KSEG accesses), then checks the validity of the physical page in that mapping. If that physical page is marked “good” in the HWRPB’s bad page bitmaps, it allows the access and installs the PTE in the TLB. Otherwise it delivers a fault to the currently-scheduled OS.

The TLB fill handler is the most performance-critical routine in the PALcode. The basic TLB fill routine performs only a single memory load on its critical path. Checking each virtual mapping adds to that path 8 memory loads/stores to 3 cache lines. As a result, address space isolation accounts for the majority of the Microvisor’s runtime overhead (see Section 6.1).

5.3.5. *Interaction with the firmware*

- Key challenge: keep firmware in the dark about multiple OS instances

The DS20 contains firmware that provides a user interface for the machine when no operating system runs, and the I/O support needed by booting operating systems before their own device drivers kick in. The DS20 firmware supports only a single running operating system at a time.

The firmware presented a challenge. Typical virtual machine systems have their own I/O device drivers, which let them emulate firmware calls without invoking the real firmware. However, we wanted to avoid shoehorning a disk device driver into the PALcode (a typical SCSI device driver would exceed the codesize of the entire PALcode by an order of magnitude).

Another option would have been to run the real firmware within the virtual machine abstraction created by the Microvisor. However, this would have necessitated virtualizing most of the platform, and implementing the Microvisor in two versions of the PALcode (the firmware’s and Tru64’s).

Instead, we opted to virtualize above the firmware. Since we intend the Microvisor to be used for online maintenance, we observed that two OS instances will not need to boot at once. Therefore, we reasoned our monitor could let each OS instance use the real, unvirtualized firmware without inherent conflict. However, the Microvisor does need to manage the OS-firmware interaction so that the firmware’s un-virtualized, un-partitioned view of the attached I/O hardware does not conflict with an OS instance’s virtualized view.

The Microvisor intercepts all calls by an OS to the firmware by rewriting the firmware’s callback table to point to our own routines. Subsequently, whenever the OS thinks it is invoking a particular routine in the firmware, it actually invokes a routine in the monitor.

Not all firmware calls need to be changed. For those that do not, the monitor simply invokes the original firmware routine. However, for some routines the parameters the OS passes to the firmware need to be modified. For example, as explained in Section 5.3.2, OS-1 thinks its PCI bus is PCI-0, and it therefore assumes its boot disk will be on that bus. However, the firmware knows its boot disk is really on PCI-1. Thus, when OS-1 calls a routine in the firmware to open its boot disk, the Microvisor must trap that call and modify the request to open the disk on PCI-1. No modifications are necessary for OS-0 because the firmware and OS-0 are in agreement about the location of its boot disk.

Virtualizing above the firmware had several advantages. For example, because Tru64 uses a much smaller portion of the chipset’s registers than does the firmware, our monitor had to virtualize less.

However, there was downside to virtualizing above the firmware. First, whenever the booting OS invokes the firmware, our

PALcode has to yield the CPU to the firmware's PALcode. As a result, our monitor is unable to schedule the second OS while the firmware completes the OS's request, and during that time the firmware has carte blanche over the hardware. Although the firmware is generally well behaved and holds the CPU for only short periods of time, yielding the CPU in this manner while an OS boots creates an 8-15 second outage in the OS already running. We could address this outage by modifying the firmware to yield the CPU back to the Microvisor PALcode during I/Os.

Even worse, when the firmware gets a request to open a booting OS's boot device, it first resets both PCI-0 and PCI-1. This I/O reset does not crash the OS instance that is already running, but it does cause it to have an approximately two minute outage while it recovers. We addressed this latter problem by patching in memory the instructions of the firmware PALcode's TLB fill handler, causing it to shield PCI-0 from reset when OS-1 boots, and PCI-1 from reset when OS-0 boots.

Some of the most challenging aspects of the Microvisor to get working were those that dealt with lying to the firmware. If we had this implementation to do over again, we would reconsider virtualizing over the firmware without the ability to modify it. Source access would not only have simplified the task of understanding how to spoof the firmware, but would also have let us eliminate the only source of downtime present in our system.

5.3.6. *Virtual time*

- Key challenge: keep each OS's notion of time correct despite sharing the CPU

In most virtual machine systems, maintaining an accurate notion of passing time in operating system instances is challenging because each OS gets a changing portion of the CPU, and may miss timer interrupts.

Because the Microvisor gives half of the CPU to each of two OS instances, and because we do not deschedule idle OS instances, virtual time turned out not to be difficult. When a second OS boots on the Microvisor, we simply reprogram the real time clock generator in the DS20 chipset to generate timer interrupts at twice the usual rate. So that the clock interrupt ending an OS's scheduler quantum is not lost, we also immediately post a timer interrupt to an OS when it gets rescheduled on the CPU. These two steps ensure that an OS gets exactly the same number of timer interrupts when sharing the CPU as when it runs alone.

5.3.7. *Use of multiple PALcode images*

- Key challenge: optimize code paths

The Alpha microprocessor invokes the PALcode routines via well-known offsets from a base address that is fixed in the PAL_BASE internal processor register. By changing the contents of the PAL_BASE register, the PALcode can cause the processor to use a different PALcode image the next time PAL is entered.

We elected to use this PALcode switching support to optimize the Microvisor. As mentioned in Sections 5.3.2 and 5.3.5, the monitor's virtualization activities underneath OS-0 differ somewhat from those under for OS-1. We actually use two separate PALcode images, one statically compiled with just the code paths and support needed for OS-0, and one with the necessary code and support for OS-1. This optimization saves having a single PALcode image with many code paths conditionalized on whether OS-0 or OS-1 is currently running. The additional copy of the Microvisor PALcode takes up approximately 64 KB of memory.

Thus, when the running OS instance's scheduling quantum expires, its dedicated copy of the Microvisor PALcode saves its state to memory, and jumps to a special entry point in the copy of the Microvisor PALcode for the other OS instance. That copy then resets the PAL_BASE register to point to its base address, and restores the previously saved state of its OS instance.

5.3.8. *Devirtualization and revirtualization of hardware*

- Key challenge: get off of Microvisor PALcode, without losing the ability to get back on

We observe that the addresses of the physical memory and I/O resources OS-0 uses match their true physical locations. This is in contrast to OS-1 which has its accesses to virtual PCI-0 translated by the Microvisor and TLB into accesses to physical PCI-1. Furthermore, although the Microvisor traps and emulates OS-0's accesses to several CSRs, if OS-0 is not booting that emulation is not triggered. Similarly, once OS-0 is booted, it ceases to invoke the firmware. At that point, the Microvisor's presence between OS-0 and the firmware is superfluous.

Therefore, we reasoned OS-0 could run without the active participation of the Microvisor after it boots and before OS-1 boots. The Microvisor takes two steps to achieve this hardware *devirtualization*. First, it re-enables KSEG accesses. Subsequently, all physical memory, I/O, and CSR accesses get issued directly by the CPU without being trapped by the Microvisor, and without being translated by the TLB.

Second, it switches the CPU to an almost entirely stock version of the PALcode. The Microvisor PALcode normally checks for KSEG addresses during a TLB fill, and its scheduler normally looks for certain conditions during an interrupt. Although these checks are unnecessary when devirtualized, the code paths needed to carry them out still pollute the TLB fill and interrupt routines. By switching the CPU to a stock PALcode with unadulterated TLB fill and interrupt hanldlers, we avoid even this tiny overhead.

Eventually, the system administrator will want to perform online maintenance. At that time, the hardware must be *revirtualized* in order to multiplex OS-0 and OS-1 on the same physical machine.

To support revirtualization, we have added a small routine to the stock PALcode that switches the CPU back to using OS-0's Microvisor PALcode. These modifications total 25 machine instructions and are not on any code path accessed during normal execution. The Microvisor PALcode then disables KSEG addressing, resumes its prior CSR emulation, and again translates KSEG accesses to physical memory and I/O.

In addition to bootup and periods of online maintenance, there is one other time when OS-0 must run with the hardware virtualized. Tru64 Unix has a hardware manager system (`hwmgr`) that can dynamically probe for attached peripherals and configure them into the running system. To ensure `hwmgr` only finds the I/O devices dedicated to an OS, the Microvisor must virtualize the hardware during its probing activities. The OS can easily be configured so that the administrator's invocation of `hwmgr` automatically revirtualizes the hardware, if necessary.

5.3.9. Bootstrapping

- Key challenge: load kernel images without forcing Microvisor to walk file system

Ordinarily, the PALcode image used by Tru64 is loaded from ROM at reset. We did not want to rewrite ROM with our PALcodes, so we instead modified the Tru64 boot loader to load the Microvisor PALcodes off disk. Our boot loader loads three different PALcode images into physical memory: *pal.0*, *pal.1*. and *pal.2*. The pal.0 and pal.1 images are the Microvisor PALcodes for OS-0 and OS-1, respectively. The pal.2 image is the stock palcode used by OS-0 when devirtualized.

After loading the Microvisor PALcodes, the boot loader sets aside 16MB of memory to hold OS-1's executable image along with OS-1's 128 MB contiguous memory region (described in Section 5.3.3). Statically allocating the memory for OS-1's executable image simplified the Microvisor's memory management. The boot loader then loads OS-0 into memory, and jumps to its start routine.

When OS-1 is started, its executable image is loaded off disk by OS-0. This means that OS-0 must have on its boot disk a copy of any kernel binary that OS-1 is to use. Similarly, when OS-0 reboots, OS-1 handles the reloading of its image. Each OS uses several special routines in their PALcode to let them set up the executable image of the other OS, an approach similar to Xen's method of domain creation [Barham03]. Having one OS load the other kept the Microvisor from loading the boot loader off disk, which would have added a great deal of complexity to a PALcode that knows nothing of file systems.

6. PERFORMANCE EVALUATION

We have a working prototype of the Microvisor that can boot and schedule two instances of an unmodified Tru64 5.1A kernel. The Microvisor source code consists of 6000 lines of assembly added to the Alpha 21264 PALcode, 700 lines of C added to the Tru64 boot loader, and about 600 lines of C in the balloon module.

The overhead of a virtual machine system can be an important determinant of whether that system gets used for online maintenance. To understand the performance penalty of the Microvisor for different classes of applications, we ran a variety of workloads on the stock system, and compared their performance when run on the Microvisor.

For this study, we selected the following applications:

- *TCP-xput*, *TCP-rtt*: two network microbenchmarks from the *Netperf 2.2* suite. The TCP-xput workload measures sustained throughput achieved through a 100 Mbit switched ethernet using the TCP protocol and large messages. The TCP-rtt load measures round trip times of 1-byte TCP messages.
- *int-gap*, *fp-swim*, *fp-equake*: three integer and floating point math codes from the SpecCPU2000 suite.
- *int-gzip*, *int-gcc*: two non-math benchmarks from SpecCPU2000. The gzip workload compresses a file to memory. The gcc benchmark compiles a large source tree.
- *specweb*, *OSDB*: these applications model enterprise web serving and transaction processing applications respectively. SpecWeb99 bombards an Apache 1.3.28 web server with a mix of dynamic and static html requests. A single linux client opens 30 simultaneous connections (30 is the maximum number of connections the raw server can maintain that conform to SpecWeb's throughput requirement). OSDB is an open source transaction processing workload based on the AS3AP database benchmark [OSD03]. It performs a variety of queries on a 40 MB database. We run version 0.14 of OSDB using *PostgreSQL* version 7.2 as the database system. For both SpecWeb and OSDB, the html or database data is stored in an NFS file system.

6.1. CPU overhead

To measure the slowdown induced by the Microvisor's hardware virtualization, we first measure the baseline "performance" of each application on the raw Alphaserver DS20, where performance is a metric appropriate for that application (e.g. running time, throughput, iterations, etc.). We then measure each application's performance when running on OS-0 with the Microvisor virtualizing the hardware underneath with and without physical memory isolation compiled in. Since we boot only a single OS instance on the hardware, there is no overhead from context switching two OS instances in these first measurements. Overhead is computed as the percent slowdown in performance. We present the results of this study in Figure 3. Each data point represents the average of five runs. For all measurements, the 95% confidence interval is less than ±2% of the mean.

As shown in Figure 3, we find that the Microvisor's virtualization overhead is between 0.1% and 5.6% when it affords full address space isolation between OS instances. When the Microvisor does not protect the memory of one OS instance from the other, its overhead is between 0.04% and 2.2%. This remaining overhead comes from the TLB pressure induced when the Microvisor installs translations for KSEG addresses.

As shown in Figure 4, we also measured these same applications on OS-0, with the hardware devirtualized. We find that all our benchmarks are within measurement error of zero CPU overhead.

We next wanted to understand the overhead of state saving, restoring, and TLB flushing when context switching the two OS instances on the CPU. As shown in Figure 5, we measured the percentage of baseline performance achieved by each application running in OS-0, when OS-1 shares the CPU. For the baseline application performance, we use both the performance on virtualized hardware (isolating the pure context switching costs), and the raw hardware (revealing the overhead of both context switching and virtualization). The virtualized runs have full memory protection enabled. Our simple round-robin scheduler gives each OS 50% of the CPU. Our measurements indicate that CPU-bound applications perform quite close to half their baseline speed. Therefore, the impact of the saving and restoring state to schedule these two OS instances is small, around 0.1%-1.5%.

TCP-xput performs somewhat better than 50% of baseline. This application is not CPU bound, so taking away half its CPU does not diminish its performance by half.

We conclude that we are successful in getting CPU overhead to zero when OS-0 runs devirtualized. Coupled with the Alpha's intrinsic virtualizability, many of our labors to facilitate devirtualization, such as partitioning I/O devices and not virtualizing physical memory, resulted in a virtual machine monitor with very low overhead, even when OS-0 runs virtualized.

The majority of the Microvisor's overhead comes from the extra checks it performs in order to isolate the physical address spaces of each OS instance. For the problem of online maintenance, this protection is not buying the user much. In this problem domain, the user does not have to worry about one OS instance being used to attack another since both share the same administrative domain.

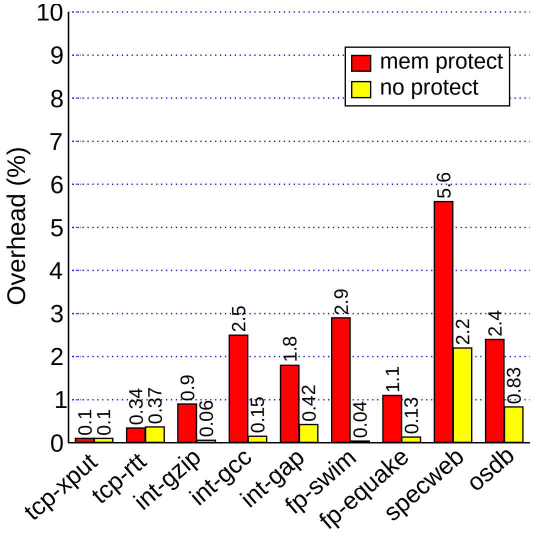

Figure 3: Microvisor virtualization overhead. Overhead is between 0.1% and 5.6% when physical memory protection is enabled, and 0.04% to 2.2% when it is not.

Furthermore, since both OS instances share the hardware only rarely, there is a relatively brief window for accidental fault propagation. Therefore, it is possible to imagine running the Microvisor with no isolation between OS instances, reducing virtualization overhead to just 2.2%, and making devirtualization less necessary. However, we expect most system administrators to feel more confident using the Microvisor with its protection mechanism enabled when two OS instances share the hardware, and to eliminate overhead by devirtualizing the rest of the time.

6.2. Memory overhead

The Microvisor takes up very little memory. Our three PALcode images occupy approximately 216 KB. For working memory, our PALcodes use only memory already set aside for PAL by the firmware.

The dominant source of memory overhead is the 144 MB of physical memory set aside for OS-1 at first boot (128 MB of contiguous memory plus 16MB for its executable image).

7. ONLINE MAINTENANCE STUDY

To demonstrate the use of our prototype for online maintenance, as well as to measure the cost in downtime of our design decisions, we used the Microvisor to perform an online OS upgrade on a web server while it serves web traffic. We chose this application for our study because web traffic is easy to migrate. Complex applications will likely require a more complete process migration or fail-over mechanism [Huang98, Osman02, Zhou99].

Our web server starts out running Tru64 Unix version 5.1A. We upgrade it on the fly to version 5.1B. Despite the relatively small change in version number, 5.1B is a major upgrade to 5.1A (5.1A was shipped in September 2001. 5.1B was shipped in November 2002). To perform our upgrade we use the off-the-shelf 5.1B upgrade CD-ROM, and unmodified Tru64 upgrade tools.

We use SpecWeb99 to generate http traffic served by Apache 1.3.28. To mimic a period of reduced server demand suitable for performing maintenance, we configure the SpecWeb client to open 15 simultaneous connections. From the client end, we measure the total http request throughput over all connections during the maintenance on the web server. From our throughput log we are able to calculate downtime, which we define to be the accumulated time during which throughput from the server is zero.

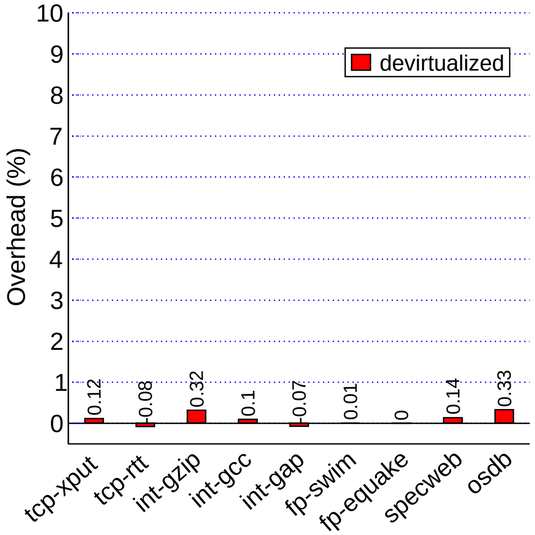

Figure 4: Overhead when OS-0 is devirtualized.

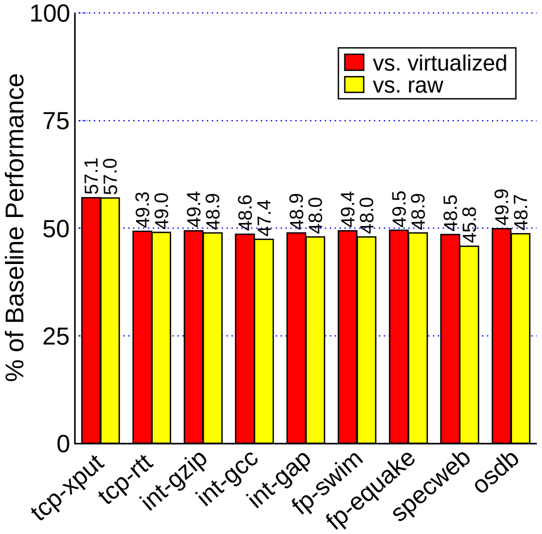

Figure 5: Performance when a second OS instance shares the CPU. Graph shows percentage of baseline application performance. In baseline, a single OS instance runs on either raw or virtualized hardware. TCP-xput is I/O bound, and so we see less than a 50% hit when it gets only half the CPU.

To migrate web service from one OS instance to another, we start Apache in the destination instance then simply reassign the source instance's IP address to the destination instance using *ifconfig*. OS-0 and OS-1 each have a dedicated boot disk. The SpecWeb html file set is accessible from either OS instance over NFS.

At time zero, OS-0 is running devirtualized, serving web traffic generated by SpecWeb running on a separate node. We then perform the OS upgrade as follows. First, we revirtualize the hardware. Next, we instruct the balloon module in OS-0 to "inflate", grabbing half of physical memory for OS-1. We then boot OS-1, and migrate

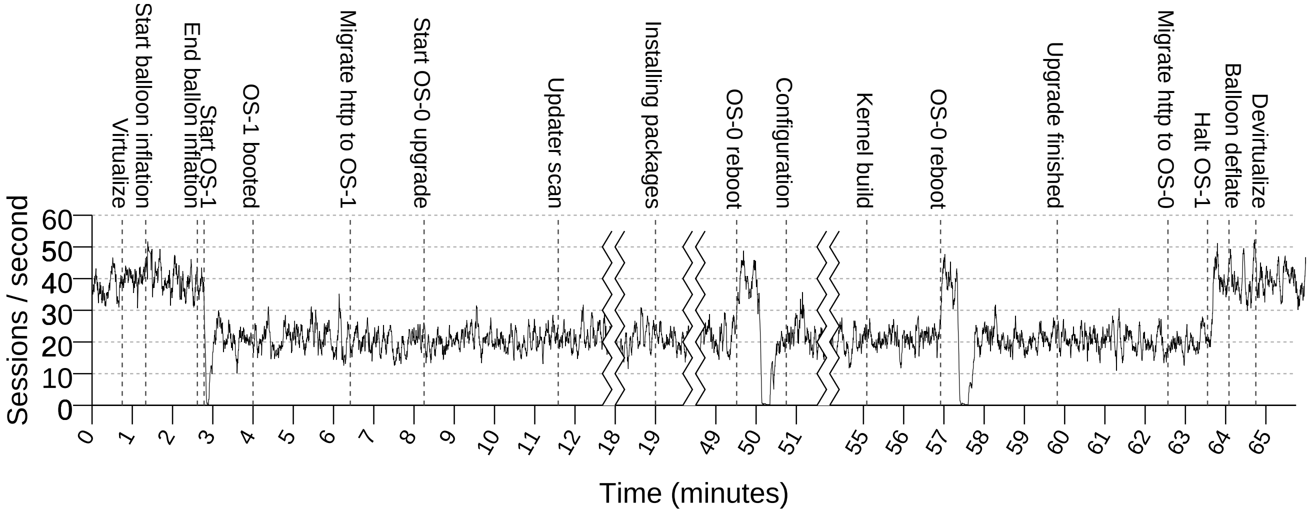

Figure 6: Online OS upgrade. The plot shows SpecWeb throughput (http sessions/sec) over time as viewed from the web client while the web server undergoes an OS upgrade from Tru64 Unix version 5.1A to 5.1B. The graph has three time-warps during lengthy upgrade operations, marked by vertical zig-zag lines. Each time we boot (or reboot) an OS, the Microvisor temporarily cedes the CPU to the firmware. These periods show up as outages during which throughput drops to zero. We accumulate 41 seconds of downtime during this online OS upgrade. Performing it in an offline manner requires about an hour of downtime.

the SpecWeb request traffic to it. Once OS-0 is idle, we drop OS-0 to single-user mode, mount the upgrade CD-ROM, and start the upgrade utility. The upgrade software first scans the boot file system for available space and potential configuration problems. It then installs approximately 100 software packages, and initiates a reboot. Next, the upgrade utility goes through a configuration phase, builds a custom kernel, and triggers a final reboot. Once rebooted, we migrate the SpecWeb request traffic back to OS-0, and halt OS-1. The balloon module then “deflates”, returning OS-1’s memory to OS-0, and the Microvisor devirtualizes the hardware.

Figure 6 shows the results of this experiment. We plot http sessions/second on a timeline labeled with the various actions performed during the maintenance. The instantaneous SpecWeb throughput is quite noisy, so each data point represents a moving average five data points wide. The graph has three time-warps during lengthy upgrade operations, marked by vertical zig-zag lines.

Several features stand out on the graph. First, the drop in throughput by roughly one half when OS-1 starts sharing the CPU is clearly visible, as is the return to full throughput after OS-1 halts. During OS-0’s two reboots, OS-1 briefly runs alone on the hardware, resulting in two spikes to full throughput.

There are three short outages resulting from the Microvisor temporarily ceding control of the CPU to the firmware while it performs boot-time I/O (see Section 5.3.5), bringing the total downtime for this online OS upgrade to 41 seconds. Had this maintenance been performed in an offline manner, it would have required around one hour of downtime.

This 41 seconds of downtime is an artifact of our implementation, and not fundamental to our approach to online maintenance. As discussed in Section 5.3.5, we could reduce this downtime to zero either by modifying the firmware to yield the CPU during long I/Os, or by endowing our monitor with a disk device driver, allowing it to perform boot I/O without invoking the firmware.

Despite the fact that we designed and tested the Microvisor exclusively with Tru64 5.1A, version 5.1B worked out of the box with the Microvisor VMM. However, we did have to modify the balloon module for compatibility with 5.1B.

8. LESSONS FOR MAINSTREAM ARCHITECTURES

We chose to prototype our system on Alpha because we felt writing for it an OS-transparent virtual machine monitor was plausible with a team our size. However, the Alpha architecture is less mainstream than others, such as Intel x86. What lessons can our study impart for these architectures?

We believe building a system like the Microvisor is possible on x86. The main attributes of online maintenance that we exploit—one OS in common case, two OS instances at most, both in the same administrative domain—are just as valid on x86 as they are on Alpha.

Although x86 is not a virtualizable architecture in the strict sense, binary code rewriting techniques used for virtualizing it are well-known [Robin00, Devine02]. Although we have not studied the problem in detail, we see no reason why devirtualizing and revirtualizing an x86 CPU would not be possible.

Assuming the x86 VMM manages memory and I/O devices to allow for devirtualization, we believe the following steps would be involved in devirtualizing such a system.

To devirtualize, the x86 VMM first configures the CPU and chipset’s internal registers to match the state of any registers the VMM was emulating. If employing code modification, the monitor restores any modified instructions. It then returns control to the operating system leaving the processor in kernel mode. The kernel continues to execute, but its operations on privileged state no longer trap to the VMM. These steps are very similar to those performed by VMware when it performs a “world switch” between the VMM’s context and the host OS’s context in their hosted virtual machine products [Sugerman01].

To revirtualize the processor, a kernel module can pass control back to the VMM, who takes over the CPU by returning to kernel code at a reduced privilege level.

We believe the challenge of implementing the Microvisor for a particular platform/OS combination lays not mainly in the architecture, but in the nature of the platform, and the way the OS uses it. For example, we were able to take advantage of Tru64's support for versions of our platform with differing numbers of PCI buses, as well as its support for walling off failed pages of memory. This native support in Tru64 freed us from virtualizing the PCI host bridge, and physical memory. On a different platform, we would simply have had to virtualize more of the machine.

9. RELATED WORK

Virtual machines have been around for almost 40 years. During that time their use has been studied for such problems as OS debugging [Keefe68, Winett71], system security [Popek74, Robin00, Dunlap02], fault tolerance [Bressoud95, Govil99], retrofitting legacy systems [Buzen74, Bugnion97], migrating environments [Sapuntzakis02], aiding the deployment of large-scale server systems [Barham03, Whitaker02], and many more. Our work is the first we know of to examine the use of virtual machines for online maintenance.

There are several soft partitioning schemes that are related to the Microvisor, the closest of these is IBM's *Logical Partitions* or *LPARs* [Borden89]. LPARs allows administrators to slice up a multiprocessor into several partitions, each with its own operating system instance, and dedicated I/O resources and memory. LPARs is fairly unique among soft partitioning schemes in that, like the Microvisor, it virtualizes CPUs so that a partition can receive only a fraction of a processor. LPARs depends on special virtualization support in both the operating system, and the microprocessor.

Denali and *Xen* are lightweight virtual machine systems designed to multiplex many OS instances on x86 hardware [Whitaker02, Barham03]. Both VMMs export a variation of the machine interface—operating systems must be ported to run in one of their virtual machines. For operating systems thus ported, Denali and Xen can enable online maintenance.

In contrast to Denali, Xen, and LPARs, the Microvisor requires no special virtual machine support from the OS. Both Denali and LPARS provide more general virtualization capabilities than does the Microvisor. In exchange for giving up this generality, the Microvisor reduces common case CPU overhead to zero, while retaining the functionality needed to enable online maintenance.

Microkernel operating systems split OS functionality between a very small kernel running close to the hardware, and a larger OS server running on top [Golub90, Hartig97]. This arrangement of software is architecturally similar to Unix running on a virtual machine, although the interface exported by the microkernel abstracts more of the machine than the straight hardware interface exported by a virtual machine layer. Microkernels can allow for the online upgrade of the OS server by letting an administrator run both the old OS server and the upgraded one simultaneously, and migrating applications to use the new server.

Sapuntzakis et al. have looked at taming the problems associated with OS, application, and network configuration by bundling networks of virtual machines with their operating systems and applications into *virtual appliances* [Sapuntzakis03]. Their system makes it easy to deploy software updates by bundling those changes into new versions of the appliances, then causing existing appliance instances to restart using the new versions. They do not focus on deploying these updates without downtime.

10. CONCLUSION

Businesses depend on being able to run their enterprise applications around the clock. As a result, server customers typically harden their infrastructure against failures by deploying their services on redundant hardware. However, no similarly practical approach exists for avoiding most outages from planned maintenance, despite the fact that maintenance accounts for 90% of downtime.

In this paper, we have proposed a novel technique for reducing planned downtime by performing maintenance online using virtual machines. This technique enables a broad class of software maintenance.

Although our technique will work with commodity virtual machine software, we believe potential customers will balk at paying the 10-20% overhead of these systems during periods of peak load, to enable infrequent maintenance to be performed online.

Rather than force users to trade off peak server performance versus downtime, we have proposed and built a virtual machine system that provides online maintenance, but that lets servers run at full speed at peak times. The Microvisor is the first system to provide devirtualization, allowing sites to pay the CPU cost of virtualization just during periods of maintenance. We find that stripping away features not necessary for online maintenance also yields low CPU overhead when virtualized—just 5.6%—without compromising OS-transparency.

We have used the Microvisor to demonstrate a challenging form of single-node, online maintenance, performing a major OS upgrade on a web server while it serves web traffic. The only existing way to do this upgrade online is to use a spare node. Our approach makes even multi-node maintenance practical, eliminating the serial deployment of updates over many nodes required by current techniques.

During our OS upgrade, we accumulate just 41 seconds of downtime rather than the hour that would be required normally. That 41 seconds of downtime is not fundamental to online maintenance using virtual machines—we could eliminate it with a small change to our prototype's firmware.

The main cost of using the Microvisor is the additional I/O and networking hardware it requires to be set aside for use during maintenance. In exchange for this modest, one-time hardware cost, sites can reduce the two dominant ongoing expenses of their budgets: administrator time and downtime.

11. ACKNOWLEDGEMENTS

Our VMware ESX Server benchmark results are published with prior written consent from VMware, Inc. Thank you to Caroline Tice who helped with the early stages of Microvisor development. We are grateful also to David Bernardo and Greg Bellows of the Tru64 virtual memory team for suggesting the bad page trick. Many thanks to Luiz Barroso, Wendy Bartlett, Mike Burrows, Jeff Chase, Alan Eustace, Kourosh Gharachorloo, Dan Magenheimer, Jeff Mogul, Partha Ranganathan, Eric Rassmussen, John Sontag, Mustafa Uysal, and Jeff Wiedemeier for various guidance, support, and editing of this work over the years. Finally, thank you to Anne Marie Lowell for her patient support.

12. REFERENCES

[Appavoo03] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[Barham03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, October 2003.

[Borden89] Terry L. Borden, James P. Hennessy, and James W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, 1989.

[Bose03] Pradip Bose. Keynote address: Power-Efficient Microarchitectural Choices at the Early Definition Stage. In *2003 IEEE/ACM Workshop on Power-Aware Computing Systems (PACS-2003 at Micro-36)*, December 2003.

[Bressoud95] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault-tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[Bugnion97] Eduard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, October 1997.

[Buzen74] Jeffrey P. Buzen and Robert P. Goldberg. Virutal Machine Techniques for Introducing Peripherals into Computer Systems. In *Proceedings of COMPCON 1974*, February 1974.

[Chapin96] Steve Chapin. Distributed and Multiprocessor Scheduling. *ACM Computing Surveys*, 28(1), March 1996.

[Devine02] Scott W. Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent 6,397,242, May 2002.

[Dunlap02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 2002 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[Goldberg74] Robert P. Goldberg. Survey of Virtual Machine Research. *Computer*, pages 34–45, June 1974.

[Golub90] David Golub, Randall Dean, Allessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference*, 1990.

[Govil99] Kinshuk Govil, Dan Teodosiu, and Mendel Rosenblum. Cellular Disco: Resource Management Using Virutal Clusters on Shared-Memory Multiprocessors. In *Proceedings of the 1999 ACM Symposium on Operating Systems Principles*, December 1999.

[Hartig97] Hermann Hartig, Michael Hohmuth, Jochen Liedtke, Sebastian Schonberg, and Jean Wolter. The Performance of Micro-Kernel-Based Systems. In *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, October 1997.

[Hicks96] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic Software Updating. In *Proceedings of the 2001 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 258–266, May 1996.

[Huang98] Y. Huang, P. Y. Chung, C. M. R. Kintala, D. Liang, and C. Wang. NT-SwiFT: Software-implemented Fault Tolerance for Windows-NT. In *Proceedings of the 1998 USENIX WindowsNT Symposium*, August 1998.

[Humpreys03] John Humpreys, Mark Melenovsky, and Vernon Turner. Service-Centric Computing: An Infrastructure Perspective, Outlook and Analysis. Technical Report 28934, IDC Corporation, March 2003.

[Keefe68] D. D. Keefe. Hierarchical Control Programs for Systems Evaluation. *IBM Systems Journal*, 7(2):123–133, 1968.

[Levin02] Mark M. Levin. Data Center Trends and Best Practices. Technical report, META Group, 2002.

[Mic00] Windows 2000 Clustering: Performing a Rolling Upgrade. Technical Report http://www.microsoft.com/windows2000/techinfo/planning/incremental/rollupgr.asp, Microsoft Corporation, 2000.

[Non98] HP NonStop Group. Personal communication. 1998.

[OSD03] Open Source Database Benchmark. http://osdb.sourceforge.net/, 2003.

[Osman02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[Patterson02a] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, UC Berkeley, March 2002.

[Patterson02b] David A. Patterson. A Simple Way to Estimate the Cost of Downtime. In *Proceedings of the 16th USENIX Large Installation Systems Administration Conference (LISA '02)*, pages 185–188, November 2002.

[Popek74] Gerald J. Popek and Charles S. Kline. Verifiable Secure Operating System Software. In *Proceedings of 1974 AFIPS National Computer Conference*, 1974.

[Robin00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, August 2000.

[Sapuntzakis02] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[Sapuntzakis03] Constantine P. Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 17th USENIX Large Installation Systems Administration Conference (LISA '03)*, October 2003.

[Segal93] Mark E. Segal and Ophir Frieder. On-the-fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, pages 53–65, March 1993.

[Sugerman01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[Wahbe93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[Waldspurger02] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[Whitaker02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 2002 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[Winett71] Joel M. Winett. Virtual Machines for Developing Systems Software. In *Proceedings of 1971 IEEE Computer Society Conference*, September 1971.

[Zhou99] Yuanyuan Zhou, Peter M. Chen, and Kai Li. Fast Cluster Failover Using Virtual Memory-Mapped Communication. In *Proceedings of the 1999 ACM Conference on Supercomputing*, June 1999.