101

Virtual Machine Monitors

101.1 Introduction

Years ago, IBM sold expensive mainframes to large organizations, and a problem arose: what if the organization wanted to run different operating systems on the machine at the same time? Some applications had been developed on one OS, and some on others, and thus the problem. As a solution, IBM introduced yet another level of indirection in the form of a **virtual machine monitor** (**VMM**) (also called a **hypervisor**) [G74].

Specifically, the monitor sits between one or more operating systems and the hardware and gives the illusion to each running OS that it controls the machine. Behind the scenes, however, the monitor actually is in control of the hardware, and must multiplex running OSes across the physical resources of the machine. Indeed, the VMM serves as an operating system for operating systems, but at a much lower level; the OS must still think it is interacting with the physical hardware. Thus, **transparency** is a major goal of VMMs.

Thus, we find ourselves in a funny position: the OS has thus far served as the master illusionist, tricking unsuspecting applications into thinking they have their own private CPU and a large virtual memory, while secretly switching between applications and sharing memory as well. Now, we have to do it again, but this time underneath the OS, who is used to being in charge. How can the VMM create this illusion for each OS running on top of it?

THE CRUX:
HOW TO VIRTUALIZE THE MACHINE UNDERNEATH THE OS
The virtual machine monitor must transparently virtualize the machine underneath the OS; what are the techniques required to do so?

101.2 Motivation: Why VMMs?

Today, VMMs have become popular again for a multitude of reasons. Server consolidation is one such reason. In many settings, people run services on different machines which run different operating systems (or even OS versions), and yet each machine is lightly utilized. In this case, virtualization enables an administrator to **consolidate** multiple OSes onto fewer hardware platforms, and thus lower costs and ease administration.

Virtualization has also become popular on desktops, as many users wish to run one operating system (say Linux or Mac OS X) but still have access to native applications on a different platform (say Windows). This type of improvement in **functionality** is also a good reason.

Another reason is testing and debugging. While developers write code on one main platform, they often want to debug and test it on the many different platforms that they deploy the software to in the field. Thus, virtualization makes it easy to do so, by enabling a developer to run many operating system types and versions on just one machine.

This resurgence in virtualization began in earnest the mid-to-late 1990's, and was led by a group of researchers at Stanford headed by Professor Mendel Rosenblum. His group's work on Disco [B+97], a virtual machine monitor for the MIPS processor, was an early effort that revived VMMs and eventually led that group to the founding of VMware [V98], now a market leader in virtualization technology. In this chapter, we will discuss the primary technology underlying Disco and through that window try to understand how virtualization works.

101.3 Virtualizing the CPU

To run a **virtual machine** (e.g., an OS and its applications) on top of a virtual machine monitor, the basic technique that is used is **limited direct execution**, a technique we saw before when discussing how the OS virtualizes the CPU. Thus, when we wish to "boot" a new OS on top of the VMM, we simply jump to the address of the first instruction and let the OS begin running. It is as simple as that (well, almost).

Assume we are running on a single processor, and that we wish to multiplex between two virtual machines, that is, between two OSes and their respective applications. In a manner quite similar to an operating system switching between running processes (a **context switch**), a virtual machine monitor must perform a **machine switch** between running virtual machines. Thus, when performing such a switch, the VMM must save the entire machine state of one OS (including registers, PC, and unlike in a context switch, any privileged hardware state), restore the machine state of the to-be-run VM, and then jump to the PC of the to-be-run VM and thus complete the switch. Note that the to-be-run VM's PC may be within the OS itself (i.e., the system was executing a system call) or it may simply be within a process that is running on that OS (i.e., a user-mode application).

We get into some slightly trickier issues when a running application or OS tries to perform some kind of **privileged operation**. For example, on a system with a software-managed TLB, the OS will use special privileged instructions to update the TLB with a translation before restarting an instruction that suffered a TLB miss. In a virtualized environment, the OS cannot be allowed to perform privileged instructions, because then it controls the machine rather than the VMM beneath it. Thus, the VMM must somehow intercept attempts to perform privileged operations and thus retain control of the machine.

A simple example of how a VMM must interpose on certain operations arises when a running process on a given OS tries to make a system call. For example, the process may be trying to call `open()` on a file, or may be calling `read()` to get data from it, or may be calling `fork()` to create a new process. In a system without virtualization, a system call is achieved with a special instruction; on MIPS, it is a **trap** instruction, and on x86, it is the `int` (an interrupt) in-

struction with the argument `0x80`. Here is the open library call on FreeBSD [B00] (recall that your C code first makes a library call into the C library, which then executes the proper assembly sequence to actually issue the trap instruction and thus make a system call):

```
open:
    push    dword mode
    push    dword flags
    push    dword path
    mov     eax, 5
    push    eax
    int     80h
```

On UNIX-based systems, `open()` takes just three arguments: `int open(char *path, int flags, mode_t mode)`. You can see in the code above how the `open()` library call is implemented: first, the arguments get pushed onto the stack (`mode, flags, path`), then a 5 gets pushed onto the stack, and then `int 80h` is called, which transfers control to the kernel. The 5, if you were wondering, is the pre-agreed upon convention between user-mode applications and the kernel for the `open()` system call in FreeBSD; different system calls would place different numbers onto the stack (in the same position) before calling the trap instruction `int` and thus making the system call[1].

Process	Hardware	Operating System
1. Execute instructions (add, load, etc.)		
2. System call: Trap to OS		
	3. Switch to kernel mode; Jump to trap handler	
		4. In kernel mode; Handle system call; Return from trap
	5. Switch to user mode; Return to user code	
6. Resume execution (@PC after trap)		

Table 101.1: Executing a System Call

When a trap instruction is executed, as we've discussed before, it usually does a number of interesting things. Most important in our

[1]Just to make things confusing, the Intel folks use the term "interrupt" for what almost any sane person would call a trap instruction. As Patterson said about the Intel instruction set: "It's an ISA only a mother could love."

example here is that it first transfers control (i.e., changes the PC) to a well-defined **trap handler** within the operating system. The OS, when it is first starting up, establishes the address of such a routine with the hardware (also a privileged operation) and thus upon subsequent traps, the hardware knows where to start running code to handle the trap. At the same time of the trap, the hardware also does one other crucial thing: it changes the mode of the processor from **user mode** to **kernel mode**. In user mode, operations are restricted, and attempts to perform privileged operations will lead to a trap and likely the termination of the offending process; in kernel mode, on the other hand, the full power of the machine is available, and thus all privileged operations can be executed. Thus, in a traditional setting (again, without virtualization), the flow of control would be like what you see in Table 101.1.

Process	Operating System
1. System call: Trap to OS	
	2. OS trap handler: Decode trap and execute appropriate syscall routine; When done: return from trap
3. Resume execution (@PC after trap)	

Table 101.2: System Call Flow Without Virtualization

On a virtualized platform, things are a little more interesting. When an application running on an OS wishes to perform a system call, it does the exact same thing: executes a trap instruction with the arguments carefully placed on the stack (or in registers). However, it is the VMM that controls the machine, and thus the VMM who has installed a trap handler that will first get executed in kernel mode.

So what should the VMM do to handle this system call? The VMM doesn't really know **how** to handle the call; after all, it does not know the details of each OS that is running and therefore does not know what each call should do. What the VMM does know, however, is **where** the OS's trap handler is. It knows this because when the OS booted up, it tried to install its own trap handlers; when the OS did so, it was trying to do something privileged, and therefore trapped into the VMM; at that time, the VMM recorded the necessary information (i.e., where this OS's trap handlers are in memory). Now, when the VMM receives a trap from a user process running on

the given OS, it knows exactly what to do: it jumps to the OS's trap handler and lets the OS handle the system call as it should. When the OS is finished, it executes some kind of privileged instruction to return from the trap (**rett** on MIPS, `iret` on x86), which again bounces into the VMM, which then realizes that the OS is trying to return from the trap and thus performs a real return-from-trap and thus returns control to the user and puts the machine back in user mode. The entire process is depicted in Tables 101.2 and 101.3, both for the normal case without virtualization and the case with virtualization (we leave out the exact hardware operations from above to save space).

Process	Operating System	VMM
1. System call: Trap to OS		
		2. Process trapped: Call OS trap handler (at reduced privilege)
	3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap	
		4. OS tried return from trap: Do real return from trap
5. Resume execution (@PC after trap)		

Table 101.3: System Call Flow with Virtualization

As you can see from the figures, a lot more has to happen when virtualization is going on. Certainly, because of the extra jumping around, virtualization might indeed slow down system calls and thus could hurt performance.

You might also notice that we have one remaining question: what mode should the OS run in? It can't run in kernel mode, because then it would have unrestricted access to the hardware. Thus, it must run in some less privileged mode than before, be able to access its own data structures, and simultaneously prevent access to its data structures from user processes.

In the Disco work, Rosenblum and colleagues handled this problem quite neatly by taking advantage of a special mode provided by the MIPS hardware known as supervisor mode. When running in this mode, one still doesn't have access to privileged instructions, but one can access a little more memory than when in user mode;

the OS can use this extra memory for its data structures and all is well. On hardware that doesn't have such a mode, one has to run the OS in user mode and use memory protection (page tables and TLBs) to protect OS data structures appropriately. In other words, when switching into the OS, the monitor would have to make the memory of the OS data structures available to the OS via page-table protections; when switching back to the running application, the ability to read and write the kernel would have to be removed.

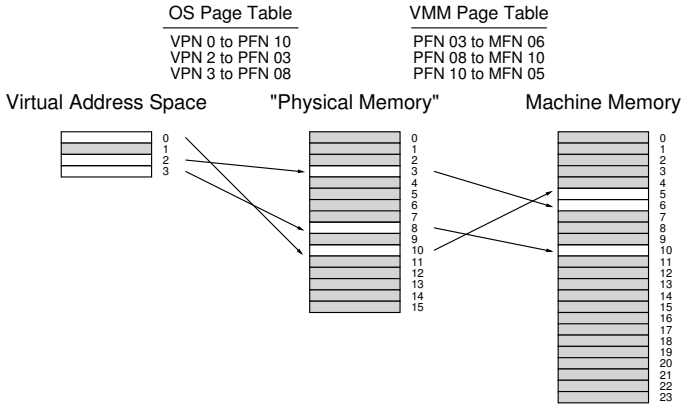

Figure 101.1: VMM Memory Virtualization

101.4 Virtualizing Memory

You should now have a basic idea of how the processor is virtualized: the VMM acts like an OS and schedules different virtual machines to run, and some interesting interactions occur when privilege levels change. But we have left out a big part of the equation: how does the VMM virtualize memory?

Each OS normally thinks of physical memory as a linear array of pages, and assigns each page to itself or user processes. The OS itself, of course, already virtualizes memory for its running processes, such that each process has the illusion of its own private address space. Now we must add another layer of virtualization, so that multiple

OSes can share the actual physical memory of the machine, and we must do so transparently.

This extra layer of virtualization makes "physical" memory a virtualization on top of what the VMM refers to as **machine memory**, which is the real physical memory of the system. Thus, we now have an additional layer of indirection: each OS maps virtual-to-physical addresses via its per-process page tables; the VMM maps the resulting physical mappings to underlying machine addresses via its per-OS page tables. Figure 101.1 depicts this extra level of indirection.

In the figure, there is just a single virtual address space with four pages, three of which are valid (0, 2, and 3). The OS uses its page table to map these pages to three underlying physical frames (10, 3, and 8, respectively). Underneath the OS, the VMM performs a further level of indirection, mapping PFNs 3, 8, and 10 to machine frames 6, 10, and 5 respectively. Of course, this picture simplifies things quite a bit; on a real system, there would be V operating systems running (with V likely greater than one), and thus V VMM page tables; further, on top of each running operating system OS_i, there would be a number of processes P_i running (P_i likely in the tens or hundreds), and hence P_i (per-process) page tables within OS_i.

To understand how this works a little better, let's recall how **address translation** works in a modern paged system. Specifically, let's discuss what happens on a system with a software-managed TLB during address translation. Assume a user process generates an address (for an instruction fetch or an explicit load or store); by definition, the process generates a **virtual address**, as its address space has been virtualized by the OS. As you know by now, it is the role of the OS, with help from the hardware, to turn this into a **physical address** and thus be able to fetch the desired contents from physical memory.

Assume we have a 32-bit virtual address space and a 4-KB page size. Thus, our 32-bit address is chopped into two parts: a 20-bit virtual page number (VPN), and a 12-bit offset. The role of the OS, with help from the hardware TLB, is to translate the VPN into a valid physical page frame number (PFN) and thus produce a fully-formed physical address which can be sent to physical memory to fetch the proper data. In the common case, we expect the TLB to handle the translation in hardware, thus making the translation fast. When a TLB miss occurs (at least, on a system with a software-managed TLB), the OS must get involved to service the miss, as depicted here in Table 101.4.

Process	Operating System
1. Load from memory: TLB miss: Trap	
	2. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid: get PFN, update TLB; Return from trap
3. Resume execution (@PC of trapping instruction); Instruction is retried; Results in TLB hit	

Table 101.4: TLB Miss Flow without Virtualization

As you can see, a TLB miss causes a trap into the OS, which handles the fault by looking up the VPN in the page table and installing the translation in the TLB.

With a virtual machine monitor underneath the OS, however, things again get a little more interesting. Let's examine the flow of a TLB miss again (see Table 101.5 for a summary). When a process makes a virtual memory reference and misses in the TLB, it is not the OS TLB miss handler that runs; rather, it is the VMM TLB miss handler, as the VMM is the true privileged owner of the machine. However, in the normal case, the VMM TLB handler doesn't know how to handle the TLB miss, so it immediately jumps into the OS TLB miss handler; the VMM knows the location of this handler because the OS, during "boot", tried to install its own trap handlers. The OS TLB miss handler then runs, does a page table lookup for the VPN in question, and tries to install the VPN-to-PFN mapping in the TLB. However, doing so is a privileged operation, and thus causes another trap into the VMM (the VMM gets notified when any non-privileged code tries to do something that is privileged, of course). At this point, the VMM plays its trick: instead of installing the OS's VPN-to-PFN mapping, the VMM installs its desired VPN-to-MFN mapping. After doing so, the system eventually gets back to the user-level code, which retries the instruction, and results in a TLB hit, fetching the data from the machine frame where the data resides.

This set of actions also hints at how a VMM must manage the virtualization of physical memory for each running OS; just like the OS has a page table for each process, the VMM must track the physical-to-machine mappings for each virtual machine it is running. These per-machine page tables need to be consulted in the VMM TLB miss

Process	Operating System	Virtual Machine Monitor
1. Load from memory TLB miss: Trap		
		2. VMM TLB miss handler: Call into OS TLB handler (reducing privilege)
	3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB	
		4. Trap handler: Unprivileged code trying to update the TLB; OS is trying to install VPN-to-PFN mapping; Update TLB instead with VPN-to-MFN (privileged); Jump back to OS (reducing privilege)
	5. Return from trap	
		6. Trap handler: Unprivileged code trying to return from a trap; Return from trap
7. Resume execution (@PC of instruction); Instruction is retried; Results in TLB hit		

Table 101.5: TLB Miss Flow with Virtualization

handler in order to determine which machine page a particular "physical" page maps to, and even, for example, if it is present in machine memory at the current time (i.e., the VMM could have swapped it to disk to reduce memory pressure).

ASIDE: **HYPERVISORS AND HARDWARE-MANAGED TLBS**

Our discussion has centered around software-managed TLBs and the work that needs to be done when a miss occurs. But you might be wondering: how does the virtual machine monitor get involved with a hardware-managed TLB? In those systems, the hardware walks the page table on each TLB miss and updates the TLB as need be, and thus the VMM doesn't have a chance to run on each TLB miss

to sneak its translation into the system. Instead, the VMM must closely monitor changes the OS makes to each page table (which, in a hardware-managed system, is pointed to by a page-table base register of some kind), and keep a **shadow page table** that instead maps the virtual addresses of each process to the VMM's desired machine pages [AA06]. The VMM installs a process's shadow page table whenever the OS tries to install the process's OS-level page table, and thus the hardware chugs along, translating virtual addresses to machine addresses using the shadow table, without the OS even noticing.

Finally, as you might notice from this sequence of operations, TLB misses on a virtualized system become quite a bit more expensive than in a non-virtualized system. To reduce this cost, the designers of Disco added a VMM-level "software TLB". The idea behind this data structure is simple. The VMM records every virtual-to-physical mapping that it sees the OS try to install; then, on a TLB miss, the VMM first consults its software TLB to see if it has seen this virtual-to-physical mapping before, and what the VMM's desired virtual-to-machine mapping should be. If the VMM finds the translation in its software TLB, it simply installs the virtual-to-machine mapping directly into the hardware TLB, and thus skips all the back and forth in the control flow above [B+97].

101.5 The Information Gap

Just like the OS doesn't know too much about what application programs really want, and thus must often make general policies that hopefully work for all programs, the VMM often doesn't know too much about what the OS is doing or wanting; this lack of knowledge, sometimes called the **information gap** between the VMM and the OS, can lead to various inefficiencies [B+97]. For example, an OS, when it has nothing else to run, will sometimes go into an **idle loop** just spinning and waiting for the next interrupt to occur:

```
while (1)
  ; // the idle loop
```

ASIDE: **PARA-VIRTUALIZATION**
In many situations, it is good to assume that the OS cannot be modified in order to work better with virtual machine monitors (for example, because you are running your VMM under an unfriendly competitor's operating system). However, this is not always the case, and when the OS can be modified (as we saw in the example with demand-zeroing of pages), it may run more efficiently on top of a VMM. Running a modified OS to run on a VMM is generally called **para-virtualization** [WSG02], as the virtualization provided by the VMM isn't a complete one, but rather a partial one requiring OS changes to operate effectively. Research shows that a properly-designed para-virtualized system, with just the right OS changes, can be made to be nearly as efficient a system without a VMM [BD+03].

It makes sense to spin like this if the OS in charge of the entire machine and thus knows there is nothing else that needs to run. However, when a VMM is running underneath two different OSes, one in the idle loop and one usefully running user processes, it would be useful for the VMM to know that one OS is idle so it can give more CPU time to the OS doing useful work.

Another example arises with demand zeroing of pages. Most operating systems zero a physical frame before mapping it into a process's address space. The reason for doing so is simple: security. If the OS gave one process a page that another had been using *without* zeroing it, an information leak across processes could occur, thus potentially leaking sensitive information. Unfortunately, the VMM must zero pages that it gives to each OS, for the same reason, and thus many times a page will be zeroed twice, once by the VMM when assigning it to an OS, and once by the OS when assigning it to a process. The authors of Disco had no great solution to this problem: they simply changed the OS (IRIX) to not zero pages that it knew had been zeroed by the underlying VMM [B+97].

There are many other similar problems to these described here. One solution is for the VMM to use inference (a form of **implicit information**) to overcome the problem. For example, a VMM can detect the idle loop by noticing that the OS switched to low-power mode. A different approach, seen in **para-virtualized** systems, requires the OS to be changed. This more explicit approach, while harder to deploy, can be quite effective.

TIP: USE IMPLICIT INFORMATION

Implicit information can be a powerful tool in layered systems where it is hard to change the interfaces between systems, but more information about a different layer of the system is needed. For example, a block-based disk device might like to know more about how a file system above it is using it; Similarly, an application might want to know what pages are currently in the file-system page cache, but the OS provides no API to access this information. In both these cases, researchers have developed powerful inferencing techniques to gather the needed information implicitly, *without* requiring an explicit interface between layers [AD+01,S+03]. Such techniques are quite useful in a virtual machine monitor, which would like to learn more about the OSes running above it without requiring an explicit API between the two layers.

101.6 Summary

Virtualization is in a renaissance. For a multitude of reasons, users and administrators want to run multiple OSes on the same machine at the same time. The key is that VMMs generally provide this service **transparently**; the OS above has little clue that it is not actually controlling the hardware of the machine. The key method that VMMs use to do so is to extend the notion of limited direct execution; by setting up the hardware to enable the VMM to interpose on key events (such as traps), the VMM can completely control how machine resources are allocated while preserving the illusion that the OS requires.

You might have noticed some similarities between what the OS does for processes and what the VMM does for OSes. They both virtualize the hardware after all, and hence do some of the same things. However, there is one key difference: with the OS virtualization, a number of new abstractions and nice interfaces are provided; with VMM-level virtualization, the abstraction is identical to the hardware (and thus not very nice). While both the OS and VMM virtualize hardware, they do so by providing completely different interfaces; VMMs, unlike the OS, are not particularly meant to make the hardware easier to use.

There are many other topics to study if you wish to learn more about virtualization. For example, we didn't even discuss what hap-

pens with I/O, a topic that has its own new and interesting issues when it comes to virtualized platforms. We also didn't discuss how virtualization works when running "on the side" with your OS in what is sometimes called a "hosted" configuration. Read more about both of these topics if you're interested [SVL01]. Finally, we didn't discuss what happens when a collection of operating systems running on a VMM uses too much memory. As it turns out, memory overload in virtualized systems can be quite difficult to handle, and thus it is the topic of our next chapter.

References

[AA06] "A Comparison of Software and Hardware Techniques
for x86 Virtualization"
Keith Adams and Ole Agesen
ASPLOS '06, San Jose, California
A terrific paper from two VMware engineers about the surprisingly small benefits of having hardware support for virtualization. Also an excellent general discussion about virtualization in VMware, including the crazy binary-translation tricks they have to play in order to virtualize the difficult-to-virtualize x86 platform.

[AD+01] "Information and Control in Gray-box Systems"
Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
SOSP '01, Banff, Canada
Our own work on how to infer information and even exert control over the OS from application level, without any change to the OS. The best example therein: determining which file blocks are cached in the OS using a probabilistic probe-based technique; doing so allows applications to better utilize the cache, by first scheduling work that will result in hits.

[B00] "FreeBSD Developers' Handbook:
Chapter 11 x86 Assembly Language Programming"
http://www.freebsd.org/doc/en/books/developers-handbook/
A nice tutorial on system calls and such in the BSD developers handbook.

[BD+03] "Xen and the Art of Virtualization"
Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield
SOSP '03, Bolton Landing, New York
The paper that shows that with para-virtualized systems, the overheads of virtualized systems can be made to be incredibly low. So successful was this paper on the Xen virtual machine monitor that it launched a company.

[B+97] "Disco: Running Commodity Operating Systems
on Scalable Multiprocessors"
Edouard Bugnion, Scott Devine, Kinshuk Govil, Mendel Rosenblum
SOSP '97
The paper that reintroduced the systems community to virtual machine research; well, perhaps this is unfair as Bressoud and Schneider [BS95] also did, but here we began to understand why virtualization was going to come back. What made it even clearer, however, is when this group of excellent researchers started VMware and made some billions of dollars.

[BS95] "Hypervisor-based Fault-tolerance"
Thomas C. Bressoud, Fred B. Schneider
SOSP '95
*One the earliest papers to bring back the **hypervisor**, which is just another term for a virtual machine monitor. In this work, however, such hypervisors are used to improve system tolerance of hardware faults, which is perhaps less useful than some of the more practical scenarios discussed in this chapter; however, still quite an intriguing paper in its own right.*

[G74] "Survey of Virtual Machine Research"
R.P. Goldberg
IEEE Computer, Volume 7, Number 6
A terrific survey of a lot of old virtual machine research.

[SVL01] "Virtualizing I/O Devices on VMware Workstation's
Hosted Virtual Machine Monitor"
Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim
USENIX '01, Boston, Massachusetts
Provides a good overview of how I/O works in VMware using a hosted architecture which exploits many native OS features to avoid reimplementing them within the VMM.

[V98] VMware corporation.
Available: http://www.vmware.com/
This may be the most useless reference in this book, as you can clearly look this up yourself. Anyhow, the company was founded in 1998 and is a leader in the field of virtualization.

[S+03] "Semantically-Smart Disk Systems"
Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '03, San Francisco, California, March 2003
Our work again, this time showing how a dumb block-based device can infer much about what the file system above it is doing, such as deleting a file. The technology used therein enables interesting new functionality within a block device, such as secure delete, or more reliable storage.

[WSG02] "Scale and Performance in the Denali Isolation Kernel"
Andrew Whitaker, Marianne Shaw, and Steven D. Gribble
OSDI '02, Boston, Massachusetts
The paper that introduces the term para-virtualization. Although one can argue that Bugnion et al. [B+97] introduce the idea of para-virtualization in the Disco paper, Whitaker et al. take it further and show how the idea can be more general than what was thought before.