VMM ((*86)Architecture support, challenges and solutions)

INTRODUCTION

The concept of using virtual machines was popular in the 1960s and 1970s in both the computing industry and academic research. In these early days of computing, virtual machine monitors (VMMs) allowed multiple users, each running their own single-user operating system instance, to share the same costly mainframe hardware [Goldberg 1974]. Virtual machines lost popularity with the increased sophistication of multi-user operating systems, the rapid drop in hardware cost, and the corresponding proliferation of computers. By the 1980s, the industry had lost interest in virtualization and new computer architectures developed in the 1980s and 1990s did not include the necessary architectural support for virtualization.

VMware is a virtualization layer could be useful on commodity platforms built from x86 CPUs and primarily running the Microsoft Windows operating systems (a.k.a. the WinTel platform). The benefits of virtualization could help address some of the known limitations of the WinTel platform, such as application interoperability, operating system migration, reliability, and security. In addition, virtualization could easily enable the co-existence of operating system alternatives, in particular Linux. Although there existed decades' worth of research and commercial development of virtualization technology on mainframes, the x86 computing environment was sufficiently different that new approaches were necessary. Unlike the vertical integration of mainframes where the processor, platform, VMM, operating systems, and often the key applications were all developed by the same

vendor as part of a single architecture [Creasy 1981], the x86 industry had a disaggregated structure. Different companies independently developed x86 processors, computers, operating systems, and applications. For the x86 platform, virtualization would need to be inserted without changing either the existing hardware or the existing software of the platform.

2. CHALLENGES IN BRINGING VIRTUALIZATION TO THE (X86) ARCHITECTURE

A virtual machine is taken to be an efficient, isolated duplicate of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM). As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs that is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

1. The x86 architecture was not virtualizable----

It contained virtualization-sensitive, unprivileged instructions, which violated the Popek and Goldberg [1974] criteria for strict virtualization. This ruled out the traditional trap-and-emulate approach to virtualization. Indeed,

engineers from Intel Corporation were convinced their processors could not be virtualized in any practical sense [Gelsinger 1998].

2. The x86 architecture was of daunting complexity---

The x86 architecture was a notoriously big CISC architecture, including legacy support for multiple decades of backwards compatibility. Over the years, it had introduced four main modes of operations (real, protected, v8086, and system management), each of which enabled in different ways the hardware's segmentation model, paging mechanisms, protection rings, and security features (such as call gates).

3. (x86) Machines had diverse peripherals -----

Although there were only two major x86 processor vendors, the personal computers of the time could contain an enormous variety of add-in cards and devices, each with their own vendor-specific device drivers. Virtualizing all these peripherals was intractable. This had dual implications: it applied to both the front-end (the virtual hardware exposed in the virtual machines) and the back-end (the real hardware the VMM needed to be able to control) of peripherals.

4. Need for a simple user experience -----

Classic VMMs were installed in the factory. We needed to add our VMM to existing systems, which forced us to consider software delivery options and a user experience that encouraged simple user adoption.

SUPPORT AND SOLUTIONS

1.Virtualizing the(x86) Architecture---

A VMM built for a virtualizable architecture uses a technique known as trap-andemulate to execute the virtual machine's instruction sequence directly, but safely, on the hardware. When this is not possible, one approach, which we used in Disco, is to specify a virtualizable subset of the processor architecture, and port the guest operating systems to that newly defined platform. This technique is known as paravirtualization but since paravirtualization is infeasible .The solution to this problem combined two key insights. First, although trap-andemulate direct execution could not be used to virtualize the entire x86 architecture, it could actually be used some of the time. And in particular, it could be used during the execution of application programs, which accounted for most of the execution time on relevant workloads. As a fallback, dynamic binary translation would be used to execute just the system software. The second key insight was that by properly configuring the hardware, particularly using the x86 segment protection mechanisms carefully, system code under dynamic binary translation could also run at near-native speeds.

Algorithm for when to use binary translation or intrepretation…..

INPUT- Current state of the virtual CPU.
OUTPUT-True if the direct execution subsystem may be used; FALSE if binary translation must be used instead.

```
if !cr0.pe then
    RETURN false
```

```
if eflags.v8086 then
RETURN true
if (e f lags.iopl ≥ cpl)||(!e f lags.if) then
RETURN false;
foreach seg ← (cs, ds, ss, es, f s, gs) do
if “seg is not shadowed” then
RETURN false;
END
RETURN true
```

2. A Guest Operating System-Centric Strategy---

The idea behind virtualization is to make the virtual machine interface identical to the hardware interface so that all software that runs on the hardware will also run in a virtual machine. Unfortunately, the description of the x86 architecture, publicly available as the Intel Architecture Manual [Intel Corporation 2010], was at once baroquely detailed and woefully imprecise for our purpose.For example, the formal specification of a single instruction could easily exceed 8 pages of pseudocode while omitting crucial details necessary for correct virtualization. We quickly realized that attempting to implement the entire processor manual was not the appropriate bootstrapping strategy. Instead, we chose a list of key guest operating systems to support and worked through them, initially one at a time. We started with a minimal set of features and progressively enhanced the completeness of the solution, while always preserving the correctness of the supported feature set. Practically speaking, we made very restrictive assumptions on how the processor’s privileged state could be configured by the guest .

3. The role of HOST operating system---

VMware Hosted Architecture was developed to allow virtualization to be inserted into existing systems. It consisted of packaging VMware Workstation to feel like a normal application to a user, and yet still have direct access to the hardware to multiplex CPU and memory resources. Like any application, the VMware Workstation installer simply writes its component files onto an existing host file system, without perturbing the hardware ACM Transactions on Computer System. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation 12:9 configuration (no reformatting of a disk, creating of a disk partition, or changing of BIOS settings). In fact, VMware Workstation could be installed and start running virtual machines without requiring even rebooting the host operating system, at least on Linux hosts. Running on top of a host operating system provided a key solution to the back-end aspect of the I/O device diversity challenge. Whereas there was no practical way to build a VMM that could talk to every I/O devices in a system, an existing host operating system could already, using its own device drivers. Rather than accessing physical devices directly, VMware Workstation backed its emulated devices with standard system calls to the host operating system. For example, it would read or write a file in the host file system to emulate a virtual disk device, or draw in a window of the host's desktop to emulate a video card. As long as the host operating system had the appropriate drivers, VMware Workstation could run virtual machines on top of it.