CHAPTER 5

第5章 硬件辅助虚拟化

5.1 概述

通过前面几章，读者已经熟悉了 VMM 的结构和功能，也了解了如何通过软件虚拟化技术来实现 VMM。本章以 IA32 架构为主，介绍如何通过硬件辅助虚拟化技术来实现 VMM。很多硬件体系结构都有类似技术，如 IA64 的 VT-i[15]。

硬件辅助虚拟化技术，顾名思义，就是在 CPU、芯片组及 I/O 设备等硬件中加入专门针对虚拟化的支持，使得系统软件可以更加容易、高效地实现虚拟化功能。

之所以需要在硬件中加入虚拟化的支持，原因是多方面的。首先，由于原有的硬件体系结构在虚拟化方面存在缺陷，例如虚拟化漏洞，导致单纯的软件虚拟化方法存在种种问题，如降优先级(deprivilege)的方法存在 Ring Compression 问题。BT 技术存在难以处理自修改代码及自参考代码的问题。PV 技术存在需要修改源码的总问题；其次，由于硬件架构的限制，某些虚拟化功能尽管可以用软件方法来实现，但是实现起来非常复杂，一个典型的例子是内存虚拟化的“影子页表”；最后，某些通过软件方法实现的虚拟化功能性能不佳，例如 I/O 设备的虚拟化。这些问题，都只有通过在 CPU 体系架构上增加相应的硬件支持，才能彻底解决。

这里以 Intel Virtualization Technology (Intel VT) 为例具体说明硬件辅助虚拟化技术所提供的支持。Intel VT 是 Intel 平台上硬件虚拟化技术的总称，包含了对 CPU、内存和 I/O 设备等各方面的虚拟化支持。图 5-1 中的 Physical Platform Resource 列举了 Intel VT 涵盖的内容。在 CPU 虚拟化方面，Inte VT 提供了 VT-x (Intel Virtualization technology for x86) 技术；在内存虚拟化方面，Intel VT 提供了 EPT(Extended Page Table)技术；在 I/O 设备虚拟化方面，Intel VT 提供了 VT-d (Intel Virtualization Technology for Direct I/O) 等技术。

图 5-1 展示了使用 Intel VT 技术实现的 VMM 的典型结构。上层是通用功能，如资源管理、系统调度等。下层是平台相关的部分，即使用 Intel VT 实现的处理器虚拟化、内存虚拟化和 I/O 虚拟化。

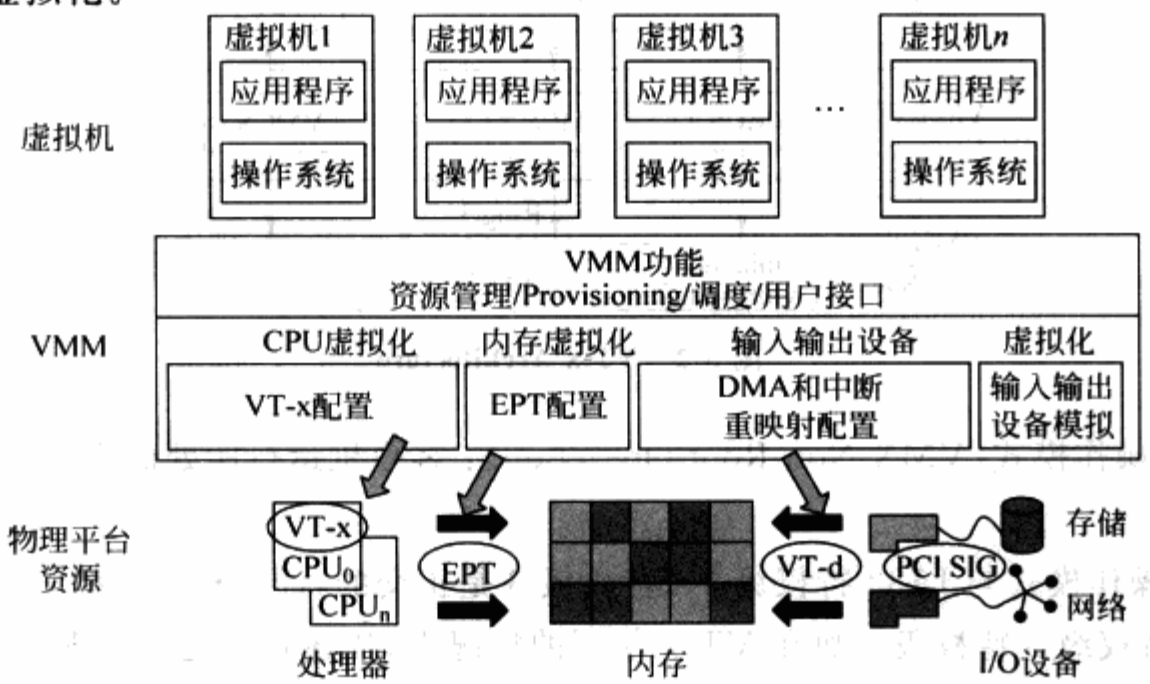

图 5-1 Intel 虚拟技术

本章下面的章节是对图 5-1 的展开，以 Intel VT 为例阐述硬件辅助虚拟化技术，并进一步描述如何在 VMM 中运用这些技术。5.2 节和 5.3 节是 CPU 虚拟化的相关内容，包括 VT-x 的原理介绍，以及如何利用 VT-x 实现 CPU 虚拟化；5.4 节介绍中断虚拟化的相关内容；5.5 节介绍 EPT 的原理和基于该技术的内存虚拟化实现；5.6 节和 5.7 节介绍 VT-d 技术以及基于该技术的 I/O 虚拟化的相关内容。最后，5.8 节对虚拟化技术中的难点——时间虚拟化加以介绍。

此外，AMD 平台也提供了类似的技术，即 AMD Virtualization（AMD-V），其原理和使用方式与 Intel VT 类似，有兴趣的读者可以进一步阅读 AMD-V 的相关文档。

5.2 CPU 虚拟化的硬件支持

5.2.1 概述

Intel VT 中的 VT-x 技术扩展了传统的 IA32 处理器架构，为 IA32 架构的处理器虚拟化提供了硬件支持。[17]手册包含了 VT-x 具体的硬件规范，本节会依据该硬件规范展开描述，读者也可以阅读该手册进一步了解技术细节。

VT-x 的基本思想可以概括为图 5-2 所示。

首先，VT-x 引入了两种操作模式，统称为 VMX 操作模式。

- 根操作模式（VMX Root Operation）：VMM 运行所处的模式，以下简称根模式。

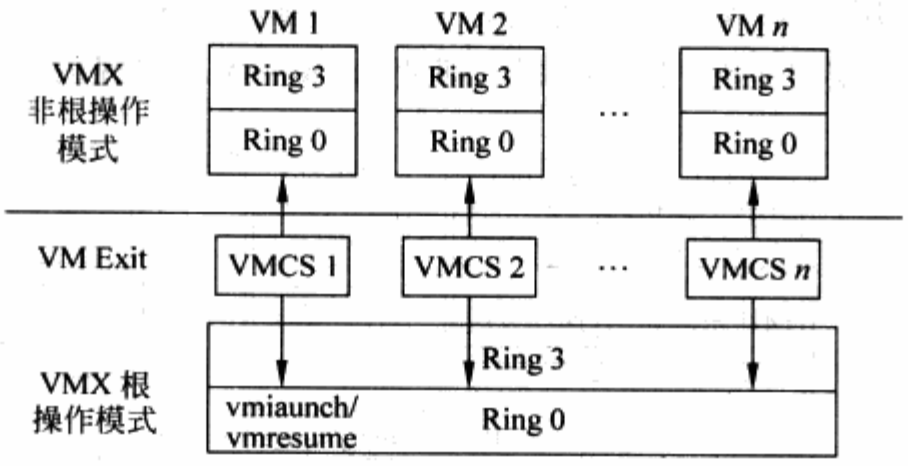

图 5-2　VT-x Architecture

- 非根操作模式(VMX Non-Root Operation)：客户机运行所处的模式，以下简称非根模式。

这两种操作模式与 IA32 特权级 0～特权级 3 是正交的，即每种操作模式下都有相应的特权级 0～特权级 3 特权级。故在 VT-x 使用的情况下，描述程序运行在某个特权级，例如特权级 0，还必须指出当前是处于根模式还是非根模式。

引入两种操作模式的理由很明显。回顾第 4 章 4.1 节关于虚拟化漏洞的阐述，我们知道指令的虚拟化是通过“陷入再模拟”的方式实现的，而 IA32 架构有 19 条敏感指令不能通过这种方法处理，导致了虚拟化漏洞。最直观的解决办法，是使得这些敏感指令能够触发异常。可惜这种方法会改变这些指令的语义，导致与原有软件不兼容，这是不可接受的。引入新的模式可以很好地解决问题。非根模式下所有敏感指令(包括 19 条不能被虚拟化的敏感指令)的行为都被重新定义，使得它们能不经虚拟化就直接运行或通过“陷入再模拟”的方式来处理；在根模式下，所有指令的行为和传统 IA32 一样，没有改变，因此原有的软件都能正常运行。

VT-x 中，非根模式下敏感指令引起的“陷入”被称为 VM-Exit。VM-Exit 发生时，CPU 自动从非根模式切换成为根模式。相应地，VT-x 也定义了 VM-Entry，该操作由 VMM 发起，通常是调度某个客户机运行，此时 CPU 从根模式切换成为非根模式。

其次，为了更好地支持 CPU 虚拟化，VT-x 引入了 VMCS(Virtual-Machine Control Structure，虚拟机控制结构)。VMCS 保存虚拟 CPU 需要的相关状态，例如 CPU 在根模式和非根模式下的特权寄存器的值。VMCS 主要供 CPU 使用，CPU 在发生 VM-Exit 和 VM-Entry 时都会自动查询和更新 VMCS。VMM 可以通过指令来配置 VMCS，进而影响 CPU 的行为。

最后，VT-x 还引入了一组新的指令，包括 VMLAUCH/VMRESUME 用于发起 VM-Entry，VMREAD/VMWRITE 用于配置 VMCS 等。

下面的章节中，会分别对 VT-x 引入的 VMCS、VMX 操作模式、虚拟化相关的新指令分别进行介绍。

5.2.2 VMCS

VMCS的概念与第4章4.2节阐述的虚拟寄存器的概念类似，可以看作是虚拟寄存器概念在硬件上的应用。虚拟寄存器的操作和更改完全由软件执行，但VMCS却主要由CPU操作。VMCS是保存在内存中的数据结构，包含了虚拟CPU的相关寄存器的内容和虚拟CPU相关的控制信息，每个VMCS对应一个虚拟CPU。

VMCS在使用时需要与物理CPU绑定。在任意给定时刻，VMCS与物理CPU是一对一的绑定关系，即一个物理CPU只能绑定一个VMCS，一个VMCS也只能与一个物理CPU绑定。VMCS在不同的时刻可以绑定到不同的物理CPU，例如在某个VMCS先和物理CPU1绑定，并在某个时刻解除绑定关系，并重新绑定到物理CPU2。这种绑定关系的变化称为VMCS的“迁移(Migration)”。

VT-x提供了两条指令用于VMCS的绑定与解除绑定。

- VMPTRLD <VMCS地址>：将指定的VMCS与执行该指令的物理CPU绑定。
- VMCLEAR：将执行该指令的物理CPU与它的VMCS解除绑定。该指令会将物理CPU缓存中的VMCS结构同步到内存中去，从而保证VMCS和新的物理CPU绑定时，内存中的值是最新的。

VMCS的一次迁移过程如下。

(1) 在CPU1上执行VMCLEAR，解除绑定。

(2) 在CPU2上执行VMPTRLD，进行新的绑定。

VT-x定义了VMCS的具体格式和内容。规定它是一个最大不超过4KB的内存块，并要求是4KB对齐。描述了VMCS的格式，各域描述如下。

(1) 偏移0处是VMCS版本标识，表示VMCS数据格式的版本号。

(2) 偏移4处是VMX中止指示，VM-Exit执行不成功时产生VMX中止，CPU会在此处存入VMX中止的原因，以方便调试。

(3) 偏移8处是VMCS数据域，该域的格式是CPU相关的，不同型号的CPU可能使用不同格式，具体使用哪种格式由VMCS版本标识确定。

VMCS块格式如表5-1所示。

表5-1 VMCS块格式

字节偏移	描　述
0	VMCS revision identifier
4	VMX-abort indicator
8	VMCS data(implementation-specific format)

VMCS主要的信息存放在“VMCS数据域”，VT-x提供了两条指令用于访问VMCS。

- VMREAD <索引>：读VMCS中“索引”指定的域。

• VMWRITE <索引> <数据>：写 VMCS 中“索引”指定的域。

VT-x 为 VMCS 数据域的每个字段也定义了相应的“索引”，故通过上述两条指令也可以直接访问 VMCS 数据域中的各个域。

具体而言，VMCS 数据域包括下列 6 大类信息。

(1) 客户机状态域：保存客户机运行时，即非根模式时的 CPU 状态。当 VM-Exit 发生时，CPU 把当前状态存入客户机状态域；当 VM-Entry 发生时，CPU 从客户机状态域恢复状态。

(2) 宿主机状态域：保存 VMM 运行时，即根模式时的 CPU 状态。当 VM-Exit 发生时，CPU 从该域恢复 CPU 状态。

(3) VM-Entry 控制域：控制 VM-Entry 的过程。

(4) VM-Execution 控制域：控制处理器在 VMX 非根模式下的行为。

(5) VM-Exit 控制域：控制 VM-Exit 的过程。

(6) VM-Exit 信息域：提供 VM-Exit 原因和其他的信息。VM-Exit 信息域是只读的。

本节先介绍客户机状态域和宿主机状态域，其他域会在 5.2.4 节中介绍。

1. 客户机状态域

客户机状态域用于保存 CPU 在非根模式下运行时的状态。当发生 VM-Entry 时，CPU 自动将客户机状态域保存的状态加载到 CPU 中；当发生 VM-Exit 时，CPU 自动将 CPU 的状态保存回客户机状态域。

客户机状态域中首先包含了一些寄存器的值，这些寄存器是必须由 CPU 进行切换的，如段寄存器、CR3、IDTR 和 GDTR。CPU 通过这些寄存器的切换来实现客户机地址空间和 VMM 地址空间的切换。客户机状态域中并不包括通用寄存器和浮点寄存器，它们的保存和恢复由 VMM 决定，可提高效率和增强灵活性(见第 6 章 6.3.3 节关于上下文切换的三个例子)。客户机状态域包含的寄存器如下。

(1) 控制寄存器 CR0、CR3 和 CR4。

(2) 调试寄存器 DR7。

(3) RSP、RIP 和 RFLAGS。

(4) CS、SS、DS、ES、FS、GS、LDTR、TR 及影子段描述符寄存器。

(5) GDTR、IDTR 及影子段描述符寄存器。

除了上述寄存器外，客户机状态域中还包含了一些 MSR 的内容。这些 MSR 既可以由处理器进行切换，也可以由 VMM 进行切换。由谁切换，可以通过 VMCS 的一些控制域设定。这些 MSR 包括 IA32_SYSENTER_CS、IA32_SYSENTER_ESP 和 IA32_SYSENTER_EIP 等。

除此之外，客户机状态域还包含了一些非寄存器内容，主要用于精确模拟虚拟 CPU，例如中断状态域等。

2. 宿主机状态域

宿主机状态域用于保存 CPU 在根模式下运行时的 CPU 状态。宿主机状态域只在 VM-Exit 时被恢复，在 VM-Entry 时不用保存。这是因为宿主机状态域的内容通常几乎不

需要改变，例如 VM-Exit 的入口 RIP 在 VMM 整个运行期间都是不变的。当需要改变时，VMM 可以直接对该域进行修改，别忘了，VMCS 是保存在内存中的。

宿主机状态域只包含寄存器值，具体内容如下。

(1) 控制寄存器 CR0、CR3 和 CR4。

(2) 调试寄存器 DR7。

(3) RSP、RIP 和 RFLAGS。

(4) CS、SS、DS、ES、FS、GS、TR 及影子段描述符寄存器。

(5) GDTR、IDTR 及影子段描述符寄存器。

(6) IA32_SYSENTER_CS。

(7) IA32_SYSENTER_ESP。

(8) IA32_SYSENTER_EIP。

与客户机状态域相比，宿主机状态域没有 LDTR，正如操作系统内核通常不使用 LDT 一样，VMM 只需要使用 GDT 就足够了。

此外，当 VM-Exit 发生时，宿主机状态域中的 CS：RIP 指定了 VM-Exit 的入口地址，SS、RSP 指定了 VMM 的栈地址。

5.2.3 VMX 操作模式

5.2.1 节介绍了 VMX 操作模式的基本概念，本节及下节进行详细论述。

作为传统 IA32 架构的扩展，VMX 操作模式这个功能在默认情况下是关闭的，因为传统的操作系统并不需要使用这个功能。当 VMM 需要使用这个功能时，可以使用 VT-x 提供的新指令来打开与关闭这个功能，参见图 5-3。

- VMXON：打开 VMX 操作模式。
- VMXOFF：关闭 VMX 操作模式。

描述了开启/关闭 VMX 的过程，以及 VMX 开启情况下，VMM 和客户软件的交互操作。

(1) VMM 执行 VMXON 指令进入到 VMX 操作模式，CPU 处于 VMX 根操作模式，VMM 软件开始执行。

(2) VMM 执行 VMLAUNCH 或 VMRESUME 指令产生 VM-Entry，客户机软件开始执行，此时 CPU 进入非根模式。

(3) 当客户机执行特权指令，或者当客户机运行时发生了中断或异常，VM-Exit 被触发而陷入到 VMM，CPU 切换到根模式。VMM 根据 VM-Exit 的原因做相应处理，然后转到步骤(2)继续运行客户机。

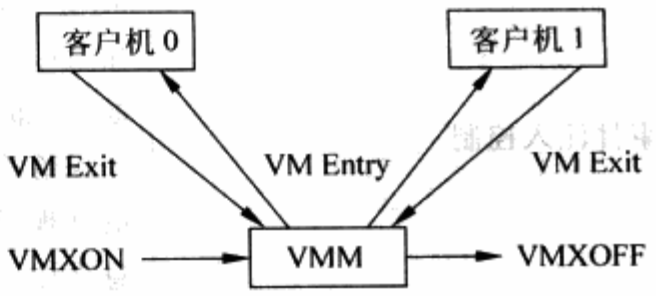

图 5-3 VMX 操作模式

(4) 如果 VMM 决定退出,则执行 VMXOFF 关闭 VMX 操作模式。

下面根据上述流程,介绍 VMX 操作模式的具体细节。

5.2.4 VM-Entry/VM-Exit

VMM 在机器加电引导后,会进行类似操作系统一样的初始化工作,并在准备就绪时通过 VMXON 指令进入根模式。在创建客户机时,VMM 会通过 VMLAUNCH 或 VMRESUME 指令切换到非根模式运行客户机,客户机引起 VM-Exit 后又切换回根模式运行 VMM。本节介绍这两种 VMX 操作模式切换的细节。

1. VM-Entry

VM-Entry 是指 CPU 由根模式切换到非根模式,从软件角度看,是指 CPU 从 VMM 切换到客户机执行。这个操作通常由 VMM 主动发起。在发起之前,VMM 会设置好 VMCS 相关域的内容,例如客户机状态域、宿主机状态域等,然后执行 VM-Entry 指令。

VT-x 为 VM-Entry 提供了两条指令。

- VMLAUNCH:用于刚执行过 VMCLEAER 的 VMCS 的第一次 VM-Entry。
- VMRESUME:用于执行过 VMLAUNCH 的 VMCS 的后续 VM-Entry。

VM-Entry 的具体行为由 VM-Entry 控制域规定,该域的具体定义如表 5-2 所示。

表 5-2 VMCS VM-Entry 控制域

域	描 述
IA-32e mode guest	在支持 Intel 64 架构的处理器上,此位决定了在 VM-Entry 后处理器是否处于 64 位模式。当客户机处于 64 位模式时,需要打开这一位
MSR VM-Entry 控制	在 VM-Entry 时,VMM 可以指示 CPU 在切换到客户机环境之前正确地装载 MSR 的值,这可以通过两个 VMCS 寄存器来指定:VM-Entry MSR-load count(指定了要装载的 MSR 的数目)和 VM-Entry MSR-load address(指定了要装载的 MSR 区域的物理地址)
事件注入控制	在 VM-Entry 时,如果需要,VMM 可以操作 VMCS 中相关的字段(如 VM-Entry Interruption-Information 字段,VM-Entry exception error code 等)来向客户机虚拟 CPU 注入一个事件(Event Injection)。这里的"事件"包括同步的异常(Exception)和异步的中断(外部中断和 NMI)。例如,当客户机虚拟 CPU 意外地试图访问一个保留的虚拟 MSR 寄存器从而导致 VM-Exit 后,就可以在 VM-Exit 的处理函数中向 VM-Entry Interruption-Information 写入需要的信息来注入 #GP(General Protecton fault)。再如,我们模拟的虚拟设备在一个 DMA 操作结束后需要向客户机注入虚拟中断时,也是通过写 VMCS 中这个字段来实现的

VM-Entry 控制域中的“事件注入控制”用到了 VM-Entry Interruption-Information 字段，表 5-3 进一步列出了该字段的格式。每次 VM-Entry 时，在切换到客户机环境后即将执行客户机指令前，CPU 会检查这个 32 位字段的最高位(即 bit31)。如果为 1，则根据 bit10：8 指定的中断类型和 bit7：0 指定的向量号在当前的客户机中引发一个异常、中断或 NMI。此外，如果 bit11 为 1，表示要注入的事件有一个错误码(如 Page Fault 事件)，错误码由另一个 VMCS 的寄存器 VM-Entry exception error code 指定。注入的事件最终是用客户机自己的 IDT 里面指定的处理函数来处理的。这样，在客户机虚拟 CPU 看来，这些事件就和没有虚拟化的环境里面对应的事件没有任何区别。

表 5-3 VM-Entry Interruption-Information 字段的格式

位	描 述
7：0	中断或异常向量(Vector of interrupt or exception)
10：8	中断类型(Interruption type) 0：外部中断(External interrupt) 1：保留(Reserved) 2：非屏蔽中断(Non-maskable interrupt，NMI) 3：硬件异常(Hardware exception) 4：软件中断(Software interrupt) 5：特权软件异常(Privileged software exception) 6：软件异常(Software exception) 7：保留(Reserved)
11	错误代码传递(0 =不传递，1 =传递)
30：12	保留(Reserved)
31	合法(Valid)

2. VM-Entry 的过程

当 CPU 执行 VMLAUNCH/VMRESUME 进行 VM-Entry 时，处理器要进行下面的步骤。

(1) 执行基本的检查来确保 VM-Entry 能开始。

(2) 对 VMCS 中的宿主机状态域的有效性进行检查，以确保下一次 VM-Exit 发生时可以正确地从客户机环境切换到 VMM 环境。

(3) 检查 VMCS 中客户机状态域的有效性；根据 VMCS 中客户机状态域区域来装载处理器的状态。

(4) 根据 VMCS 中 VM-Entry MSR-load 区域装载 MSR 寄存器。

(5) 根据 VMCS 中 VM-Entry 事件注入控制的配置，可能需要注入一个事件到客户机中。

第(1)～(4)步的检查如果没有通过，CPU 会报告 VM-Entry 失败，这通常意味着 VMCS 中某些字段的设置有错误。如果所有这些步骤都正常通过了，处理器就会把执行环

境从 VMM 切换到客户机环境，开始执行客户机指令。

5.2.5 VM-Exit

VM-Exit 是指 CPU 从非根模式切换到根模式，从客户机切换到 VMM 的操作。引发 VM-Exit 的原因很多，例如在非根模式执行了敏感指令、发生了中断等。处理 VM-Exit 事件是 VMM 模拟指令、虚拟特权资源的一大任务。在本节中，介绍 VM-Exit 相关的域以及 VM-Exit 的具体流程。

1. 非根模式下的敏感指令

当成功执行 VM-Entry 之后，CPU 就进入了非根模式。前面提到，敏感指令如果运行在 VMX 非根操作模式，其行为可能会发生变化。具体来说有如下三种可能。

(1) 行为不变化，但不引起 VM-Exit：这意味着虽然是敏感指令，但它不需要被 VMM 截获和模拟，例如 SYSENTER 指令。

(2) 行为变化，产生 VM-Exit：这就是典型需要截获并模拟的敏感指令。

(3) 行为变化，产生 VM-Exit 可控：这类敏感指令是否产生 VM-Exit，可以通过 VM-Execution 域控制(见下节)。出于优化的目的，VMM 可以让某些敏感指令不产生 VM-Exit，以减小模式切换带来的上下文开销。

由此可见，使用 VT-x 技术实现的 VMM，并不需要对所有敏感指令进行模拟，这大大减小了 VMM 实现的复杂性。VM-Execution 域的存在又为 VMM 的实现带来了灵活性，下一节将对该域进行介绍。

2. VM-Execution 控制域

VM-Execution 控制域用来控制 CPU 在非根模式运行时的行为，根据虚拟机的实际应用，VMM 可以通过配置 VM-Execution 控制域达到性能优化等目的。VM-Execution 控制域主要控制三个方面。

(1) 控制某条敏感指令是否产生 VM-Exit，如果产生 VM-Exit，则由 VMM 模拟该指令。

(2) 在某些敏感指令不产生 VM-Exit 时，控制该指令的行为。

(3) 异常和中断是否产生 VM-Exit。

表 5-4 列举了一些典型的 VM-Execution 控制域，在 5.3 节读者可以看到该域是如何被使用到 CPU 的虚拟化实现中的。

表 5-4 VM-Execution 控制域

字　　段	描　　述
External-interrupt exiting	控制外部中断是否产生 VM-Exit： 1：外部中断触发 VM-Exit。 0：CPU 查找客户机 IDT 表直接把中断递交给客户操作系统

续表

字　段	描　述
HLT exiting	控制 HLT 指令是否产生 VM-Exit： 1：客户机执行 HLT 指令会触发 VM-Exit。客户机执行 HLT 指令通常意味着客户操作系统处于空闲状态，此时 VMM 可以暂时挂起客户机，去运行其他的客户机，直到某种条件（如虚拟中断）唤醒客户机为止。 0：不引起 VM-Exit
INVLPG exiting	控制 INVLPG 指令是否产生 VM-Exit： 1：客户机执行 INVLPG 指令产生 VM-Exit。 0：不产生 VM-Exit
WBINVD exiting	控制 WBINVD 指令是否产生 VM-Exit： 1：客户机执行 WBINVD 指令产生 VM-Exit。该位和 INVLPG exiting 一起用于帮助实现 MMU 的虚拟化。 0：不产生 VM-Exit
RDPMC exiting	控制 RDPMC 指令是否产生 VM-Exit： 1：客户机执行 RDPMC 指令产生 VM-Exit。此位用来帮助实现 performance monitor 虚拟化。 0：不产生 VM-Exit
RDTSC exiting	控制 RDTSC 指令是否产生 VM-Exit： 1：客户机执行 RDTSC 指令产生 VM-Exit。此位用来帮助实现 TSC 虚拟化。 0：不产生 VM-Exit
CR8-load exiting	控制 CR8 LOAD 指令是否产生 VM-Exit： 1：客户机装载 CR8 产生 VM-Exit。 0：不产生 VM-Exit
CR8-store exiting	控制 CR8 STORE 指令是否产生 VM-Exit： 1：客户机读取 CR8 产生 VM-Exit。 0：不产生 VM-Exit
MOV-DR exiting	控制 MOV DR 指令是否产生 VM-Exit： 1：客户机访问调试寄存器产生 VM-Exit。此位用来帮助实现调试寄存器的虚拟化。 0：不产生 VM-Exit
Unconditional /O exiting	控制端口 I/O 访问是否产生 VM-Exit： 1：客户机所有访问端口 I/O 指令触发 VM-Exit，如 IN、INS、INSB、INSW、INSD、OUT、OUTS、OUTSB、OUTSW 和 OUTSD。此位用来帮助实现输入输出设备虚拟化。 当 Use I/O bitmaps 为 1 时，此位被忽略

续表

字　　段	描　　述
Use I/O bitmaps	是否使用 I/O 位图来控制 I/O 指令： 1：客户机访问 I/O 端口时，只有当该端口在 I/O 位图对应位的值为 1 时才发生 VM-Exit。 0：不使用 I/O 位图
Use MSR bitmaps	是否使用 MSR 位图来控制 MSR 的访问： 1：客户机访问 MSR 时，只有当该 MSR 在 MSR 位图对应位的值为 1 时才发生 VM-Exit。 0：不使用 MSR 位图
Use TSC offset	当 RDTSC 为 1 时，提供客户机 TSC 和物理 CPU TSC 之间的偏移。见后面 5.3.3 节“VCPU 的硬件优化”
Exception bitmap	当客户机产生异常时，是否产生 VM-Exit。该字段为 32 位，某位置 1 表示该异常发生时引发一个 VM-Exit 陷入到 VMM。否则，直接由客户操作系统处理。缺页异常有特殊处理

3. VM-Exit 控制域

VM-Exit 控制域规定了 VM-Exit 发生时 CPU 的行为，表 5-5 描述了该域的内容。

表 5-5　VM-Exit 控制域

字　　段	含　　义
Host Address Space	在支持 Intel 64 架构的处理器上，此位决定了在下一次 VM-Exit 后处理器是否处于 64 位模式。64 位的 VMM 通常需要打开这一位
Acknowledge interrupt on exit	该位控制当一个外部中断引起 VM-Exit 时，是否应答中断控制器。 1：应答。 0：不应答
VM-Exit MSR-store count	指定 VM-Exit 发生时，CPU 要保存的 MSR 数目
VM-Exit MSR-store address	指定了要保存的 MSR 区域的物理地址
VM-Exit MSR-load count	指定 VM-Exit 发生时，CPU 要装载的 MSR 数目
VM-Exit MSR-load address	指定了要装载的 MSR 区域的物理地址

4. VM-Exit 信息域

VMM 除了要通过 VM-Exit 控制域来控制 VM-Exit 的行为外，还需要知道 VM-Exit 的相关信息(如退出原因)。VM-Exit 信息域满足了这个要求，其提供的信息可以分为如下 4 类。

(1) 基本的 VM-Exit 信息，包括如下内容。

① Exit Reason：提供了 VM-Exit 的基本原因(如表 5-6 所示)。

表 5-6 Exit Reason 字段格式

字 段	描 述
Basic exit reason	VM-Exit 的基本原因，如果 VM-Entry failure 为 1，该字段表示 VM-Entry 失败的原因
VM-Exit from VMX root operation	该位为 1，表示一次 VM-Exit 发生在 CPU 处于根模式时
VM-Entry failure	该位为 1，表示一次 VM-Entry 失败了

② Exit qualification：提供 VM-Exit 的进一步原因。这个字段的值根据 VM-Exit 基本退出原因的不同而不同。例如，对于因为访问 CR 寄存器导致的 VM-Exit，Exit qualification 提供的信息包括：是哪个 CR 寄存器、访问类型是读还是写、访问的内容等。同样的，VT-x 规范也完整地定义了所有 VM-Exit 退出原因所对应的 Exit qualification。对于某些不需要额外信息的退出原因，没有相应的 Exit qualification 的定义。

(2) 事件触发导致的 VM-Exit 的信息。事件是指外部中断、异常(包括 INT3/INTO/BOUND/UD2 导致的异常)和 NMI。对于此类 VM-Exit，VMM 可以通过 VM-Exit interruption information 字段和 VM-Exit interruption error code 字段获取额外信息，例如事件类型、事件相关的向量号等。

(3) 事件注入导致的 VM-Exit 的信息。一个事件在注入客户机时，可能由于某种原因暂时不能成功，而触发 VM-Exit。此时，VMM 可以从 IDT-vectoring information 字段和 IDT-vectoring error code 中获取此类 VM-Exit 的额外信息，例如事件类型、事件向量号等。

(4) 执行指令导致的 VM-Exit 的信息。除了第一类中列出的信息外，客户机在执行敏感指令导致 VM-Exit 时，VMCS 中还有三个字段可以提供额外的信息。Guest linear address 字段给出了导致 VM-Exit 指令的客户机线性地址，VM-Exit instruction length 字段给出了该指令的长度，VM-Exit instruction information 字段给出了当该指令为 VMX 指令时的额外信息。

5. VM-Exit 的具体过程

了解 VM-Exit 需要的各种数据域、控制域后，介绍一下整个 VM-Exit 的大致流程。当一个 VM-Exit 发生时，依次执行下列步骤。

(1) CPU 首先将此次 VM-Exit 的原因信息记录到 VMCS 相应的信息域中，VM-Entry interruption-information 字段的有效位(bit31)被清零。

(2) CPU 状态被保存到 VMCS 客户机状态域。根据设置，CPU 也可能将客户机的 MSR 保存到 VM-Exit MSR-store 区域。

(3) 根据 VMCS 中宿主机状态域和 VM-Exit 控制域中的设置，将宿主机状态加载到 CPU 相应寄存器。CPU 也可能根据 VM-Exit MSR-store 区域来加载 VMM 的 MSR。

(4) CPU 由非根模式切换到了根模式，从宿主机状态域中 CS：RIP 指定的 VM-Exit 入

口函数开始执行。

在 VMM 处理完 VM-Exit 后，会通过 VMLAUNCH/VMRESUME 指令发起 VM-Entry 进而重新运行客户机。当下一次 VM-Exit 发生后，又会重复上述处理流程。虚拟化的所有内容就在 VMM→客户机→VMM→…… 的不断切换中完成。

5.3 CPU 虚拟化的实现

5.3.1 概述

通过前面章节的学习，读者应该了解了处理器虚拟化的概念。与软件虚拟化技术不同的是，使用 Intel VT-x 的 VMM 在处理器虚拟化的实现上更加简单和高效。

和软件虚拟技术用“CPU 执行环境处理器”来描述虚拟 CPU 类似，硬件虚拟化使用 VCPU(Virtual CPU)描述符来描述虚拟 CPU。VCPU 描述符类似操作系统中进程描述符(或进程控制块)，本质是一个结构体，通常由下列几部分构成。

① VCPU 标识信息：用于标识 VCPU 的一些属性，例如 VCPU 的 ID 号，VCPU 属于哪个客户机等。

② 虚拟寄存器信息：虚拟的寄存器资源，在使用 Intel VT-x 的情况下，这些内容包含在 VMCS 中，例如客户机状态域保存的内容。

③ VCPU 状态信息：类似于进程的状态信息，标识该 VCPU 当前所处的状态，例如睡眠、运行等，主要供调度器使用。

④ 额外寄存器/部件信息：主要指未包含在 VMCS 中的一些寄存器或 CPU 部件。例如浮点寄存器和虚拟的 LAPIC 等。

⑤ 其他信息：用于 VMM 进行优化或存储额外信息的字段，例如存放该 VCPU 私有数据的指针等。

由此可见，Intel VT-x 情况下的 VCPU 可以划分成两个部分，一个是以 VMCS 为主由硬件使用和更新的部分，这主要是虚拟寄存器；一个是除 VMCS 之外，由 VMM 使用和更新的部分，主要指 VMCS 以外的部分。图 5-4 展示了 VCPU 的构成。

当 VMM 创建客户机时，首先要为客户机创建 VCPU，整个客户机的运行实际上可以看作是 VMM 调度不同的 VCPU 运行。下面就以 VCPU 的创建—运行—退出为主线，介绍使用 Intel VT-x 技术的 CPU 虚拟化的实现。

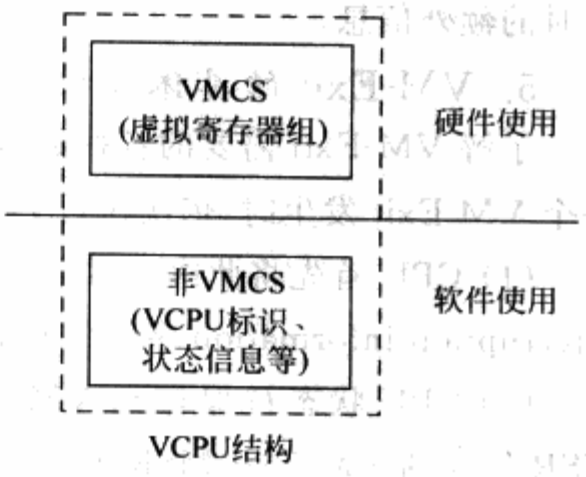

图 5-4 VCPU 结构

5.3.2 VCPU 的创建

创建 VCPU 实际上是创建 VCPU 描述符，由于本质上 VCPU 描述符是一个结构体，因此创建 VCPU 描述符简单来说就是分配相应大小的内存。VCPU 描述符包含的内容很多，通常会被组织成多级结构，例如第一级结构体可以是各个平台通用的内容，中间包含一个指针指向第二级结构体，包含平台相关的内容，如图 5-4 所示。对于这样的多级结构，需要为每一级结构体相应地分配内存。

VCPU 描述符在创建之后，需要进一步初始化才能使用。物理 CPU 在上电之后，硬件会自动将 CPU 初始化为特定的状态。VCPU 的初始化也是一个类似的过程，将 VCPU 描述符的各个部分置成可用的状态。通常初始化包含如下内容。

(1) 分配 VCPU 标识：首先要标识该 VCPU 属于哪个客户机，再为该 VCPU 分配一个在客户机范围内唯一的标识。

(2) 初始化虚拟寄存器组：主要指初始化 VMCS 相关域。这些寄存器的初始化值通常是根据物理 CPU 上电后各寄存器的值设定的。

(3) 初始化 VCPU 状态信息：设置 VCPU 在被调度前需要配置的必要标志。具体情况依据调度器的实现决定。

(4) 初始化额外部件：将未被 VMCS 包含的虚拟寄存器初始化为物理 CPU 上电后的值，并配置虚拟 LAPIC 等部件。

(5) 初始化其他信息：根据 VMM 的实现初始化 VCPU 的私有数据。

VMCS 的创建与初始化

VMCS 的创建与初始化是支持 VT-x 的 VCPU 创建的重要组成部分，这里再详细说明一下。

VMCS 在分配时，只需要分配一块 4KB 大小，并对齐到 4KB 边界的内存即可。初始化则需要根据 VT-x 的定义，对于前面所列的 VMCS 相关内容进行初始化，基本思想是根据物理 CPU 初始化的定义，提供一个和物理 CPU 初始化后类似的状态。此外，根据 VMM 的 CPU 虚拟化策略，设置相应的 VMCS 控制位。

(1) 客户机状态域：这个状态域描述了 VCPU 运行时的状态，因此，初始化的取值基本上是参考物理 CPU 初始化后的状态。例如，物理 CPU 加电后会通过复位地址跳转到 BIOS 执行，那么 Guest RIP 字段可直接设置为虚拟机 Guest BIOS 的起始指令地址。

(2) 宿主机状态域：这个状态域描述了发生 VM-Exit 时，CPU 切换到 VMM 时的寄存器的值，因此，初始化的取值是参考 VMM 运行时的 CPU 的状态。例如，HOST CS、HOST DS 等字段的取值是 VMM 运行时的段寄存器的值；HOST CR0、HOST CR3 等字段的取值是 VMM 运行时控制寄存器的值，通常是保护模式的、开页的。此外，HOST RIP 字段通常被设置为 VMM 中 VMX Exit 处理函数(VMX Exit Handler)的入口。

(3) VM-Execution 控制域：这个域控制 VCPU 运行时的一些行为，如执行某些敏感指

令是否发生 VM-Exit。因此，这个域的取值主要取决于 VMM 对于相应敏感指令的虚拟化策略。举例来说，对于 IN/OUT 指令，如果 VMM 允许客户机软件直接访问某些 I/O 端口，那么 VMM 就会将 Use I/O bitmaps 位置为 1，并且在 I/O bitmap 中将相应的 I/O 端口所对应的位置为 0，这样，客户机软件访问这些 I/O 端口时就不会发生 VM-Exit。VM-Execution 控制域给 VMM 带来了很大的灵活度，允许 VMM 做很多优化，后面还会做进一步的介绍。

(4) VM-Entry 控制域：这个状态域主要在每次 VM-Entry 之前设置，因此在 VCPU 初始化时不需要特别设置。

(5) VM-Exit 控制域：这个状态域有两个字段 VMM 通常有兴趣去设置，一个是 Acknowledge interrupt on exit，有助于更快地响应外部中断；另一个是 Host Address Space，用于支持 IA32e 模式。

(6) VM-Exit 信息域：这个域的值由硬件自动更新，因此不需要初始化。

5.3.3 VCPU 的运行

VCPU 创建并初始化好之后，就可以通过调度程序被调度运行。调度程序会根据一定的策略算法来选择 VCPU 运行。具体的调度策略内容超出了本节的描述范围，本节主要描述在选定 VCPU 之后，如何将 VCPU 切换到物理 CPU 上运行。

1. 上下文切换

从第 2 章已经知道了上下文实际是一个寄存器的集合。这里的寄存器包括通用寄存器、浮点寄存器、段寄存器、控制寄存器以及 MSR 等。前面已经提到，在 Intel VT-x 的支持下，VCPU 的上下文可以分为两部分。故上下文的切换也分为由硬件自动切换(VMCS 部分)和 VMM 软件切换(非 VMCS 部分)两个部分。其中，硬件切换部分可以更好地保证 VMM 与客户机的隔离，但缺乏灵活性。软件切换部分则可以由 VMM 自己选择性地切换需要的上下文(例如，浮点寄存器就无须每次都切换)，从而有更大的灵活性并节省切换的开销。

图 5-5 描述了 VT-x 支持的 CPU 上下文切换的过程。可以归纳为下列几个步骤。

(1) VMM 保存自己的上下文，主要是保存 VMCS 不保存的寄存器，即宿主机状态域以外的部分。

(2) VMM 将保存在 VCPU 中的由软件切换的上下文加载到物理 CPU 中。

(3) VMM 执行 VMRESUME/VMLAUNCH 指令，触发 VM-Entry，此时 CPU 自动将 VCPU 上下文中 VMCS 部分加载到物理 CPU，CPU 切换到非根模式。

此时，物理 CPU 已经处于客户机的运行环境了，rip/eip 也指向了客户机的指令，这样 VCPU 就被成功调度并运行了。

上下文切换次数频繁会带来不小的切换开销，因此对上下文切换进行优化是很有必要的。和操作系统一样，VMM 也使用“惰性保存/恢复(Lazy Save/Restore)”的方法进行优化，其基本思想是尽量将寄存器的保存/恢复延迟到最后一刻，即其他 VCPU 或 VMM 需要

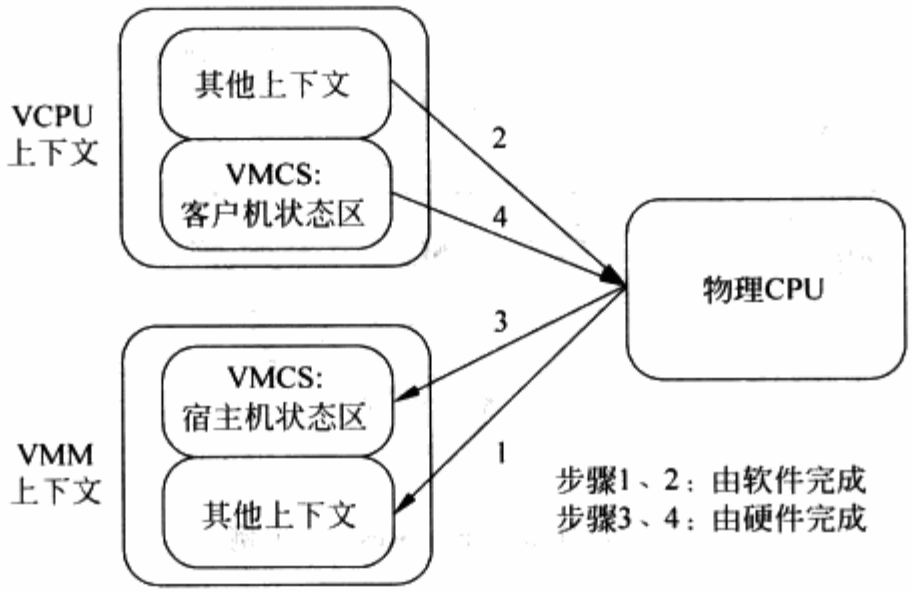

图 5-5 VM-Entry 中的上下文切换

用该寄存器的时候再保存/恢复。这种方法能够减少很多不必要的寄存器保存/恢复，提高上下文切换的效率。具体来说，VMM 通过考察资源的使用情况来实现“惰性保存/恢复”。

（1）对于 VMM 需要使用的寄存器，每次 VCPU 和 VMM 切换时都要保存/恢复。

（2）对于 VMM 没有使用的寄存器，如果 VMM 无法知道 VCPU 是否在最近的执行中曾经修改了这个寄存器（如扩展通用寄存器 DR6），那么在 VCPU 和 VMM 切换时，不需要对这个寄存器进行保存和恢复。但是，当 VMM 进行不同的 VCPU 切换时，例如使一个 VCPU 睡眠并调度另一个 VCPU 运行，需要每次都保存和恢复这个寄存器。

（3）对于 VMM 没有使用的寄存器，如果 VMM 可以知道客户机是否在最近的执行中修改了这个寄存器（如浮点寄存器），还可以做进一步的优化。不仅在 VCPU 和 VMM 切换时，不需要对这个寄存器进行保存和恢复，即使切换不同的 VCPU，也不需要每次都保存/恢复，而是根据需要进行。

举一个简单的例子来说明这种情况。如图 5-6 所示，VCPU1、VCPU2 和 VCPU3 按照顺序调度到物理 CPU 上执行，即 VCPU1 先执行，其次 VCPU2，最后 VCPU3。其中，VCPU1 和 VCPU3 在执行过程中会使用浮点寄存器，而 VCPU2 不会。VMM 了解到这种情况后，在从 VCPU1 调度到 VCPU2 时，只需保存 VCPU1 的浮点寄存器而无须加载 VCPU2 的；从 VCPU2 调度到 VCPU3 时，只需加载 VCPU3 的浮点寄存器而无须保存 VCPU2 的。这样就将原本两次保存/加载的工作减少为一次（保存 VCPU1 半次，加载 VCPU3 半次）。

2. VCPU 的硬件优化

相对于软件虚拟技术实现的 CPU 虚拟化，使用 Intel VT-x 技术的 VMM 可以采用多种方式对 VCPU 的实现进行优化。优化的目的，是尽可能少地在客户机和 VMM 之间切换，从而减少上下文切换的开销。Intel VT-x 提供的优化方法可以分为如下两种。

（1）无条件优化：指以往在软件虚拟技术下必须陷入到 VMM 中的敏感指令，通过 Intel

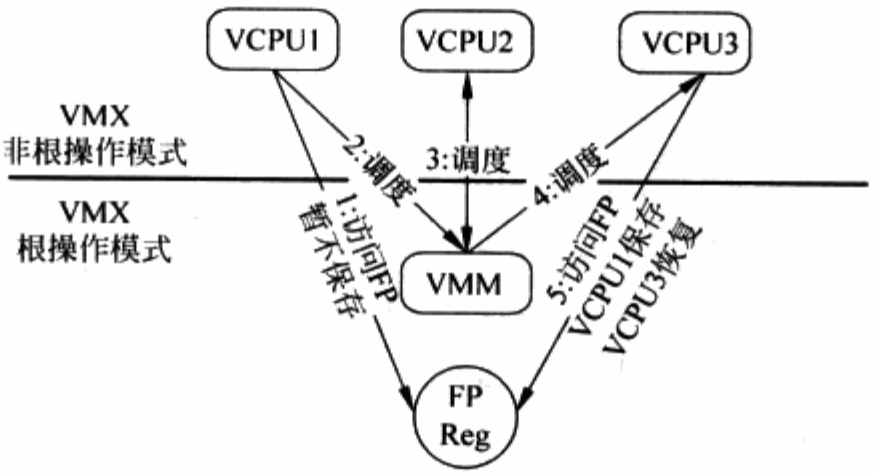

图 5-6　Lazy Save/Restore 示例

VT-x 已可以在客户机中直接执行。如后面将看到的 CR2 访问、SYSENTRY/SYSEXIT 指令。

(2) 条件优化：指通过 VMCS 的 VM-Execution 控制域，可以配置某些敏感指令是否产生 VM-Exit 而陷入到 VMM 中。如 CR0、TSC 的访问。

下面举几个例子来说明 Intel VT-x 带来的优化技术。

1. 访问 CR0

CR0 是一个控制寄存器，控制处理器的状态，如启动保护模式、打开分页机制。操作 CR0 的指令有 MOV TO CR0、MOV FROM CR0、CLTS 和 LMSW，这些指令必须在特权级 0 执行，否则产生保护异常。

在纯软件虚拟机中，客户操作系统是特权级 1、特权级 2 上执行 CR0 读写指令，因此所有的指令都产生保护异常，然后 VMM 模拟操作 CR0 指令的执行。

在硬件辅助的虚拟机中，虽然 CR0 的访问同样需要 VMM 模拟处理，但是 VT-x 提供了加速方法，能够减少因访问 CR0 所引起的 VM-Exit 的次数。首先，VMCS 的“VM-Execution 控制域”中的 CR0 read shadow 字段用来加速客户机读 CR0 的指令。每次客户机试图写 CR0 时，该字段都会自动得到更新，保存客户机要写的值。这样，客户机所有读 CR0 的指令都不用产生 VM-Exit，CPU 只要返回 CR0 read shadow 的值即可。其次，VMCS 的“VM-Execution 控制域”的 CR0 guest/host Mask 字段提供了客户机写 CR0 指令的加速。该字段每一位和 CR0 的每一位对应，表示 CR0 对应的位是否可以被客户软件修改。若为 0，表示 CR0 中对应的位可以被客户软件修改，不产生 VM-Exit；若为 1，表示 CR0 中对应的位不能被客户软件修改，如果客户软件修改该位，则产生 VM-Exit。

同样的机制被用于加速 CR4 的访问。该优化属于条件优化。

2. 访问 TSC

在纯虚拟机软件中，因为读取 TSC 可以在任何特权级别执行，VMM 必须想办法截获 TSC 读取指令。

在硬件辅助的虚拟机中，当“VM-Execution 控制域”中 RDTSC exiting 字段为 1 时，客

户软件执行 RDTSC 产生 VM-Exit，由 VMM 模拟该指令。客户读取 TSC 在某些操作系统里是一个非常频繁的操作，为了提高效率，VT-x 提供了下面的硬件加速。当 VMCS 中 RDTSC exiting 为 1 并且 Use TSC offset 为 1 时，硬件加速有效。VMCS 中 TSC 偏移量表示该 VMCS 所代表的虚拟 CPU TSC 相对于物理 CPU TSC 的偏移，即虚拟 TSC＝物理 TSC＋TSC 偏移量。当客户软件执行 RDTSC 时，处理器直接返回虚拟 TSC，不产生 VM-Exit。这样，对 TSC 的虚拟化只需在适时更新 VMCS 中 TSC 偏移量即可，不需要每次 TSC 访问都产生 VM-Exit，大大提高了 TSC 访问的效率。该优化属于条件优化。

3. GDTR/LDTR/IDTR/TR 的访问

在纯软件虚拟机中，客户操作系统是运行在特权级 1、特权级 2 上，执行 LGDT、LIDT、LLDT 和 LTR 指令，会产生保护异常，需要 VMM 模拟这些指令的执行。在模拟的过程中，对于不同的情况，还有很多复杂的处理。例如，客户操作系统在 GDT 中，为自身内核段设置的描述符的 DPL 是 0。由于它本身运行在非特权级 0 上，所以 VMM 要通过截获 LGDT 指令，对 GDT 中的描述符进行修改。同时，像 SGDT 这样的指令可以在任何特权级下执行，客户操作系统中的程序只需要读取 GDT 并判断描述符的 DPL 就知道自身运行在虚拟机环境下，这也是一个虚拟化的漏洞。

使用 Intel VT-x 技术，VMCS 为客户机和 VMM 都提供了一套 GDTR、IDTR、LDTR 和 TR，分别保存在客户机状态域和宿主机状态域中（宿主机状态域不包括 LDTR，前面说过，VMM 不需要使用它），由硬件切换。而客户机运行在非根模式的特权级 0，所以也无须对 GDT 表等做任何更改，客户机执行 LGDT 等指令也无须产生 VM-Exit。这样的优化大大降低了 VMM 的复杂度，使实现一个 VMM 变得简单。该优化属于无条件优化。

4. 读 CR2

在发生缺页异常时，CR2 保存产生缺页错误的虚拟地址。缺页错误处理程序通常会读取 CR2 获得产生该错误的虚拟地址。缺页错误是一个发生频率较高的异常，这决定了读取 CR2 是一个高频率的操作。读取 CR2 必须在特权级 0 上执行，否则产生保护错误。

在纯软件虚拟机中，客户操作系统是在特权级 1、特权级 2 上执行读取 CR2 指令，产生保护错误，需要 VMM 模拟该指令。

使用 Intel VT-x 技术，VM-Entry/VM-Exit 时会切换 CR2。并且，客户操作系统是在非根模式的特权级 0 执行读取 CR2 指令，不产生保护错误，故无须 VMM 模拟该指令。此外，如果客户机在特权级 0 以外的级别执行读 CR2 指令，会产生保护错误，该错误是否引发 VM-Exit 由 Exception bitmap 控制。该优化属于无条件优化。

5. SYSENTER/SYSEXIT

早期的系统调用是通过 INT 指令和 IRET 指令实现的。在当前主流的 IA32 CPU 中，Intel 推出了经过优化的 SYSENTER/SYSEXIT 指令以提高效率。现代操作系统都倾向于使用 SYSENTER/SYSEXIT 实现系统调用。

SYSENTER 指令要求跳转的目标代码段运行在特权级 0，否则产生保护错误。在软件

虚拟技术中，客户操作系统运行在特权级 1、特权级 2，当客户应用程序执行 SYSENTER 会产生保护错误，需要由 VMM 模拟 SYSENTER 指令。SYSEXIT 指令必须在特权级 0 执行，否则产生保护错误。和 SYSENTER 一样，SYSEXIT 在软件虚拟技术中必须由 VMM 模拟。

使用 Intel VT-x 技术，客户操作系统运行在非根模式的特权级 0，SYSENTER/SYSEXIT 都不会引起 VM-Exit，即客户操作系统的系统调用无须 VMM 干预而直接执行。该优化属于无条件优化。

6. APIC 访问控制

对于现代主流的支持 SMP 的操作系统来说，LAPIC 在中断的递交中扮演着一个非常重要的角色。LAPIC 里面有很多寄存器，通常操作系统会以 MMIO 方式来访问它们。在这些寄存器里，操作系统使用其中的 TPR(Task Priority Register)来屏蔽中断优先级小于或等于 TPR 的外部中断。

通过虚拟化客户机的 MMU，当客户机试图访问 LAPIC 时，会发生一个缺页异常类型的 VM-Exit，从而被 VMM 拦截到。VMM 经过分析，知道客户机正在试图访问 LAPIC 后，就会模拟客户机对 LAPIC 的访问。通常，对于客户机的每一个虚拟 CPU，VMM 都会分配一个虚拟 LAPIC 结构与之对应，客户机的 MMIO 操作不会真的影响物理的 LAPIC，而只是反映到相应的虚拟 LAPIC 结构里面。VMM 的这种模拟有相当大的开销，如果客户机的每一个 LAPIC 访问都导致一次缺页异常类型的 VM-Exit 并由 VMM 模拟的话，会严重影响到客户机的性能。

针对这种情况，VT-x 提供了硬件加速支持。可以设置 VMCS 中的 Use TPR shadow=1，Virtualize APIC accesses=1，设置 Virtual APIC page 为虚拟 LAPIC 结构的地址，同时修改 VCPU 页表，使得客户机访问 LAPIC 时不发生 Page Fault(这需要相应地设置 VMCS 中的 Virtual-APIC address 寄存器)。同时，对于那些暂时不能注入客户机的中断(如果有的话)，还需要挑出优先级最高的那个(就是向量号最大的那个)，将其优先级填入 VMCS 中的 TPR threshold 寄存器。

这样设置后，对于除了 TPR 以外的 LAPIC 寄存器的访问，客户机会直接发生 APIC-Access 类型的 VM-Exit。此时，CPU 可告知 VMM 客户机正试图访问哪个 LAPIC 寄存器，这可降低 VMM 对客户机此次访问的模拟开销；而客户机对 TPR 的读操作则可以直接从虚拟 LAPIC 结构中的相应偏移处读取而无须发生任何 VM-Exit。最后，客户机对 TPR 的写操作只在必要的时候(客户机把 TPR 减小到比 TPR threshold 还要小的时候)才发生 TPR-Below-Threshold 类型的 VM-Exit，这种情况下 VMM 可检测是否有虚拟中断可以注入客户机。

上面谈到 TPR 寄存器时，说是用 MMIO 方式来访问的，其实对于 64 位的 x86 平台，专门有一个特别的系统控制寄存器 CR8 被映射到了 TPR(读写 CR8 就等效于读写 TPR)，64 位的客户机通常通过 CR8 寄存器来访问 TPR。当客户机试图访问 CR8 时，会发生一个

Control-Register-Accesses 类型的 VM-Exit。为了更快地模拟客户机对 CR8 的访问，除了上面提到的设置外，可以设置 VMCS 中的 CR8-load exiting＝0 和 CR8-store exiting＝0。这样，客户机读 CR8 时，CPU 可以从虚拟 LAPIC 结构中相应的偏移处直接返回正确的值而不会发生任何 VM-Exit；当客户机写 CR8 时，只在必要的时候才发生 TPR-Below-Threshold 类型的 VM-Exit。

7. 异常控制

在软件虚拟化技术中，客户机产生的异常都会被 VMM 截获，由 VMM 决定如何处理，通常是注入给客户机操作系统。

使用 Intel VT-x 技术，可以用 Exception bitmap 配置哪些异常需要由 VMM 截获。对于不需要 VMM 截获的异常，可以将 Exception bitmap 中对应的位置 1，则异常发生时直接由客户机操作系统处理。这样的优化可以大大减少由客户机异常引起的 VM-Exit。该优化属于条件优化。

8. I/O 控制

在软件虚拟技术中，VMM 需要截获 I/O 指令来实现 I/O 虚拟化。但由于 I/O 指令通过设置可以在特权级 3 执行，截获 I/O 指令需要额外的处理。

使用 Intel VT-x 技术，可以通过 VMCS 的 Unconditional I/O exiting、Use I/O bitmaps、I/O bitmap 进行配置，选择性地让 I/O 访问产生 VM-Exit 而陷入 VMM 中。这样，对于不需要模拟的 I/O 端口，可以让客户机直接访问。该优化属于条件优化。

9. MSR 位图

x86 包括很多 MSR 寄存器，使用 Intel VT-x 和 I/O 控制一样，可以通过 use MSR bitmaps、MSR bitmap 来控制对 MSR 的访问是否触发 VM-Exit。该优化属于条件优化。

5.3.4 VCPU 的退出

上一节了解了 VCPU 如何被调度运行，也了解了针对 VCPU 运行时的一些优化。和进程一样，VCPU 作为调度单位不可能永远运行，总会因为各种原因退出，例如执行了特权指令、发生了物理中断等。这种退出在 VT-x 中表现为发生 VM-Exit。

对 VCPU 退出的处理是 VMM 进行 CPU 虚拟化的核心，例如模拟各种特权指令。本节介绍在使用 Intel VT-x 的情况下，VMM 是如何处理 VCPU 退出的。

图 5-7 描述了 VMM 处理 VCPU 退出的典型流程，可以归纳为下列几个步骤。

(1) 发生 VM-Exit，CPU 自动进行一部分上下文的切换。见 6.2.7 节。

(2) 当 CPU 切换到根模式开始执行 VM-Exit 的处理函数后，进行另一部分上下文的切换工作(见 6.3.3 节)。

根据 VM-Exit 信息域获得发生 VM-Exit 的原因，并分发到对应的处理模块处理。例如，原因是执行了特权指令，则调用相应指令的模拟函数进行模拟。

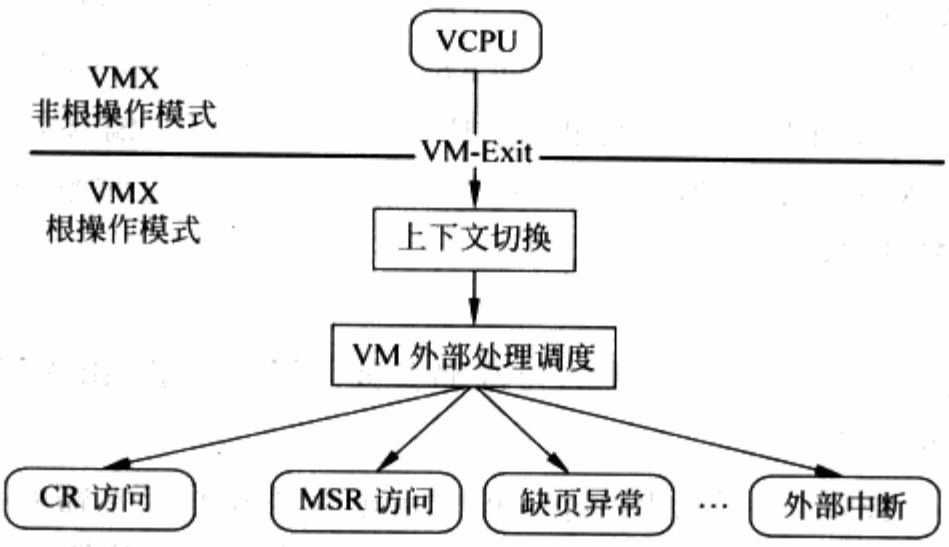

图 5-7　VM-Exit 的处理

图 5-7 列举了一些较为典型的 VCPU 退出的原因，总结起来，VCPU 退出的原因大体上有三类。

(1) 访问了特权资源，对 CR 和 MSR 寄存器的访问都属于这一类。

对于此类 VM-Exit，VMM 通过特权资源的虚拟化来解决。特权资源虚拟化的要点在于解决客户机与 VMM 在特权资源控制权的矛盾。即客户机认为自己完全拥有特权资源，可以自由读写，而特权资源的实际拥有者是 VMM，不能允许客户机自由读写。VMM 通过引入“虚拟特权资源”和“影子特权资源”来解决这个矛盾。“虚拟特权资源”是客户机所看到的特权资源，VMM 允许客户机自由地读写。“影子特权资源”是客户机运行时特权资源真正的值，通常是 VMM 在“虚拟特权资源”的基础上经过处理得到的，因此称其为“影子”。

图 5-8 以特权寄存器为例，展示了特权寄存器的虚拟化过程。当 VCPU 读特权寄存器时，VMM 将“虚拟寄存器”的值返回。例如，对于 MOV EAX，CR0 指令，VMM 将 Virtual CR0 的值赋给 EAX，然后 VM-Entry 返回。当 VCPU 写特权寄存器时，VMM 首先将值写入“虚拟寄存器”，然后根据“虚拟寄存器”的值以及虚拟化策略来更新“影子寄存器”，最后将“影子寄存器”的值应用到 VCPU 上，将值写入 VMCS“客户机状态域”的对应字段并且 VM-Entry 返回。这里的虚拟化策略是因特权虚拟器而异的，例如对于下面指令：

```
MOV EAX, 0x00000001
MOV CR0, EAX
```

假设原来 Virtual CR0＝0x80000001，VMM 比较之后会发现客户机试图将 CR0 的第 31 位(CR0. PG：页模式)清掉，即关掉 CPU 的页模式。为了实现内存的隔离，VT-x 是不允许客户机的页模式关掉的。因此，VMM 会将 Virtual CR0 按照客户机要求设置为 0x00000001，但是影子 CR0 依然设置为 0x8000001(相应 VMCS 中的 Guest CR0 字段也会被设置)。此外，VMM 会通知内存虚拟化模块有关客户机页模式的变化，内存虚拟化模块会做相关处理，如不再使用客户机的页表等。

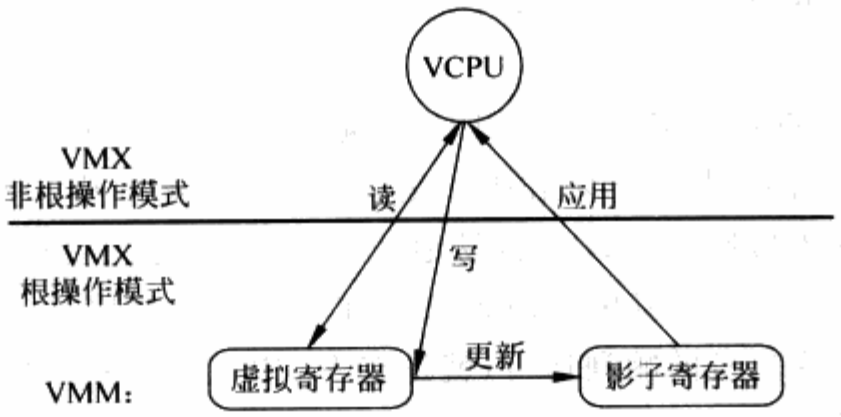

图 5-8 特权寄存器的虚拟化

(2) 客户机执行的指令引发了异常,例如缺页错误。

客户机指令导致的异常,很多是不需要虚拟化的,可以直接交由客户操作系统处理,例如“除 0 错误”、“溢出错误”和“非法指令”等异常。这些都可以通过前面提到的 Exception bitmap 设置。

对于需要虚拟化的异常,没有一个通用的方法,VMM 会对不同的异常做不同的处理。以缺页错误为例,VMM 会首先分析产生错误的原因:如果是因为访问 MMIO 地址导致的异常,则可以知道客户机在做 I/O 操作,VMM 会调用 I/O 虚拟化模块处理;如果是软件虚拟化章节提到的影子页导致的异常,VMM 会调用内存虚拟化模块处理;如果所有的原因都不是,那么就是客户机正常的缺页错误(即不需要 VMM 处理的缺页错误),该异常会被注入给客户机,由客户操作系统自己处理。

(3) 发生了中断。这可以分成两种情况,一种是发生了真正的物理中断;一种是客户机的虚拟设备发生了虚拟中断,并通过 VMM 提供的接口使客户机发生 VM-Exit。对于前者,VMM 首先读取 VMCS 的 VM-Exit interruption information 字段来获得中断向量号,然后调用 VMM 中对应的中断处理函数。对于后者,VMM 在感知到虚拟中断发生时,会用某种方法把该虚拟中断的目标 VCPU 拖到 VMM 中,常用的方法是发一个 IPI 给运行该 VCPU 的物理 CPU。然后,VMM 在 IPI 的处理函数中将该虚拟中断注入给客户机,由客户操作系统处理。

5.3.5 VCPU 的再运行

VMM 在处理完 VCPU 的退出后,会负责将 VCPU 投入再运行。从 VT-x 的角度来看,有几点需要额外考虑。

(1) 如果 VCPU 继续在相同的物理 CPU 上运行,可以用 VMRESUME 来实现 VM-Entry。VMRESUME 比 VMLAUNCH 更轻量级,执行效率更高。因此,作为优化,VMM 调度程序通常会尽量将 VCPU 调度在同一个物理 CPU 上。

(2) 如果由于某种原因(如负载均衡),VCPU 被调度程序迁移到了另外一个物理 CPU 上,那么 VMM 需要做如下几件事情。

① 将 VCPU 对应的 VMCS 迁移到另一个物理 CPU(见 6.2.2 节),这通常可以由一个 IPI 中断实现。

② 迁移完成后,在重新绑定的物理 CPU 上执行 VMLAUNCH 发起 VM-Entry。

此外,在上一节中看到,某些异常和中断是需要注入给客户机,这也是在 VCPU 运行时进行的。通过"事件注入机制",可以很容易地让 VCPU 在运行后直接进入到相应的中断/异常处理函数中执行。整个虚拟化的内容就是在 VMM→客户机→VMM→…… 中完成的。这里再细化一下,客户机的顺利运行,就是在 VCPU 运行→VCPU 退出→ VCPU 再运行→…… 的过程中完成的。

5.3.6 进阶

前面几节介绍了 CPU 虚拟化的基本知识和流程,本节引入一些高阶知识,让读者对 CPU 虚拟化中一些较为复杂的部分有一个感性的认识。

1. CPU 模式的虚拟化

在第 2 章介绍了 CPU 的几种运行模式,除此之外,还有 64 位 CPU 用的 IA-32e 模式。其中,保护模式又包括分页打开和分页关闭两种情况。在一个物理机器上,可能会同时运行一个实模式的虚拟机、一个保护模式的虚拟机和一个 IA-32e 模式的虚拟机,以及其他组合。为此,VMM 必须有模拟各种 CPU 运行模式的能力。

然而,由于客户机物理地址空间和机器真实的物理地址空间并不相同,且客户机物理地址空间占用的真实物理页面通常是不连续的,因此,目前 VT-x 技术要求物理 CPU 处于非根模式时,分页机制必须是开启的,而不考虑客户机当前运行的模式。也就是说,即使客户机运行在实模式,其所在物理 CPU 的分页机制也是开启的。由于实模式使用的内存访问模式和保护模式不同,VMM 需要大量的工作对客户机的实模式进行模拟。随着硬件技术的发展,硬件很可能会直接支持客户机的实模式内存访问,从而大大简化对客户机运行模式的虚拟化。

客户机看到 CPU 模式实际是 VCPU 中设置的模式,即 VCPU 结构中 CR0 寄存器值反映的模式,该模式和物理 CPU 的真正模式可能不同,所以 CPU 模式的虚拟化包括如下两个方面。

(1) 对标志 CPU 模式的控制寄存器(如 CR0 的 PE/PG 位)的虚拟化。

(2) 对 CPU 运行环境的虚拟化,如指令的操作数长度等。

当客户机运行在实模式时,由于物理 CPU 运行在保护模式,两者在指令、运行环境方面都不同,通常 VMM 是对客户机的指令进行模拟执行。

当客户机运行在分页关闭的保护模式时,同样,物理 CPU 实际运行在分页开启的保护模式下。此时,VMM 要负责模拟客户机实模式的内存访问机制。

当客户机运行在分页开启的保护模式时,和物理 CPU 的运行模式一样,此时不需要就模式的虚拟化做额外的工作。

表 5-7 总结了客户机运行模式和物理 CPU 的实际模式之间的对应关系，并总结了模拟客户机运行模式的方法。需要指出的是，客户机运行在何种模式和 VMM 在何种模式运行没有必然关系。例如，基于对大内存支持的需求，可以将 VMM 运行在 IA32-e 模式下。在该 VMM 上，可以运行一个 IA-32 保护模式的虚拟机。当发生 VM-Exit 和 VM-Entry 引起客户机和 VMM 切换时，硬件会通过装载“客户机状态域”/“宿主机状态域”，而自动完成 CPU 模式的转换。

表 5-7 CPU 模式虚拟化总结

虚拟机认为的 CPU 模式	物理 CPU 实际运行模式	CPU 模式虚拟化手段
实模式	分页打开的保护模式	指令模拟
分页关闭的保护模式	分页打开的保护模式	VMM 提供额外的页表
分页打开的保护模式	分页打开的保护模式	不需要
IA-32e 模式	IA-32e 模式	不需要

2. 多处理器虚拟机

随着多核技术的发展，今天大部分的计算机都具备了多个 CPU。可以通过将客户机配置为多 CPU 的虚拟平台，来提高客户机的计算能力。所谓配置多 CPU 的虚拟平台，本质上就是给客户机分配多个 VCPU，并通过调度器让它们共享一个物理 CPU 分时执行或分散到多个物理 CPU 同时执行。这和操作系统中的多线程任务采用的是同样的思想。

与单 VCPU 客户机相比，多 VCPU 客户机在实现上还有几点需要注意。

(1) 多 VCPU 发现的问题。在物理机器上，操作系统需要知道平台所有 CPU 的信息，同样，需要让客户操作系统知道它所拥有的 VCPU 的信息，例如 VCPU 的个数、每个 VCPU 的 ID 号等。这些信息在物理平台上是通过 BIOS 提供的，例如 [18]或者[19]中的 MADT Table。在虚拟环境下，客户机的虚拟 BIOS 负责这项工作。

(2) 多个 VCPU 初始化的问题。在物理平台上，多处理器的初始化通常会遵循一个规范，例如 IA32 手册[16]的 7.5 节 MULTIPLE-PROCESSOR (MP) INITIALIZATION 就定义了 IA32 平台上多处理器的初始化流程。硬件通常会选择一个 CPU 作为主 CPU，称为 BSP(Boot Strap Processor)，来执行 BIOS，其他的从 CPU，称为 AP(Application Processor)，处于 Wait-for-SIPI 状态，等待 BSP 发 SIPI(Start-up IPI)唤醒它们之后再开始执行。客户机同样需要遵循这个规范，包括挑选一个 VCPU 作为 BSP 执行 BIOS，将其他 VCPU 置于 Wait-for-SIPI 状态等待 BSP 发 SIPI，只是这个挑选工作是由 VMM 来完成的。

(3) VCPU 的同步问题。在物理多 CPU 系统中，各个 CPU 都是同时在运行的。但是，对于拥有多个 VCPU 的客户机，在某一个时刻，可能一部分 VCPU 正在运行，一部分则处于睡眠或者阻塞的状态。这对于 VCPU 之间的通信和同步都造成了一些延时问题。例如，当客户机的两个 VCPU 都运行着竞争自旋锁的代码时，考虑下面这段两个 VCPU 通过自旋锁进行同步的代码：

```
acquire_spinlock(lock)
        critical section
release_spinlock(lock)
```

当 VCPU0 上运行的代码通过 acquire_spinlock(lock)取得自旋锁，进入临界区后，VCPU0 可能会被调度出去。其后，当另一段运行在 VCPU1 上的代码尝试获取这个自旋锁时，它将因为得不到自旋锁而不得不进行等待，直到 VCPU0 被重新调度执行，完成临界区并执行 release_spinlock(lock)释放自旋锁。这样，VCPU1 因为 VCPU0 被调度出去的缘故，额外增加了同步的时延。

为了解决这些 VCPU 间的通信和同步的延迟问题，一种解决方案是对一个多 VCPU 客户机的多个 VCPU 进行群体调度(Gang Scheduling)，即它们要么同时在多个物理 CPU 上同时运行，要么同时不运行。群体调度带来的限制是一个多 VCPU 客户机的 VCPU 个数不可以超过物理平台的物理 CPU 个数。

5.4 中断虚拟化

5.4.1 概述

在第 2 章，已经介绍了现代计算机架构的中断系统，对于今天拥有众多五花八门外设的计算机而言，中断系统的作用是至关紧要的。与此对应，在虚拟的环境下，虚拟机有诸多的设备，包括 VMM 模拟的虚拟设备和直接分配给客户机的物理设备，这些设备都需要发送中断给 VCPU，以便得到处理。因此，VMM 需要提供中断虚拟化的支持。

在介绍中断虚拟化之前，再来回顾一下物理平台上的中断架构。通过第 2 章 2.4 节的介绍，我们知道外部中断的流程如图 5-9 所示。首先，I/O 设备通过中断控制器(I/O APIC 或者 PIC)发出中断请求，中断请求经由 PCI 总线发送到系统总线上，最后目标 CPU 的 Local APIC 部件接收中断，CPU 开始处理中断。

在虚拟机的环境中，VMM 也需要为客户机操作系统展现一个与物理中断架构类似的虚拟中断架构。图 5-10 展现了虚拟机的中断架构。和物理平台一样，每个 VCPU 都对应一个虚拟 Local APIC 用于接收中断。虚拟平台也包含了虚拟 I/O APIC 或者虚拟 PIC 用于发送中断。和 VCPU 一样，虚拟 Local APIC、虚拟 I/O APIC 和虚拟 PIC 都是 VMM 维护的软件实体。当虚拟设备需要发送中断时，虚拟设备会调用虚拟 I/O APIC 的接口发送中断。虚拟 I/O APIC 根据中断请求，挑选出相应的虚拟 Local APIC，调用其接口发出中断请求。虚拟 Local APIC 进一步利用 VT-x 的事件注入机制将中断注入到相应的 VCPU(见 5.3.5 节)。

由此可以看出，中断虚拟化的主要任务是实现图 5-10 描述的虚拟机中断架构，具体来说包括虚拟 PIC、虚拟 I/O APIC 和虚拟 Local APIC，并且实现中断的生成、采集和注入的整个过程。

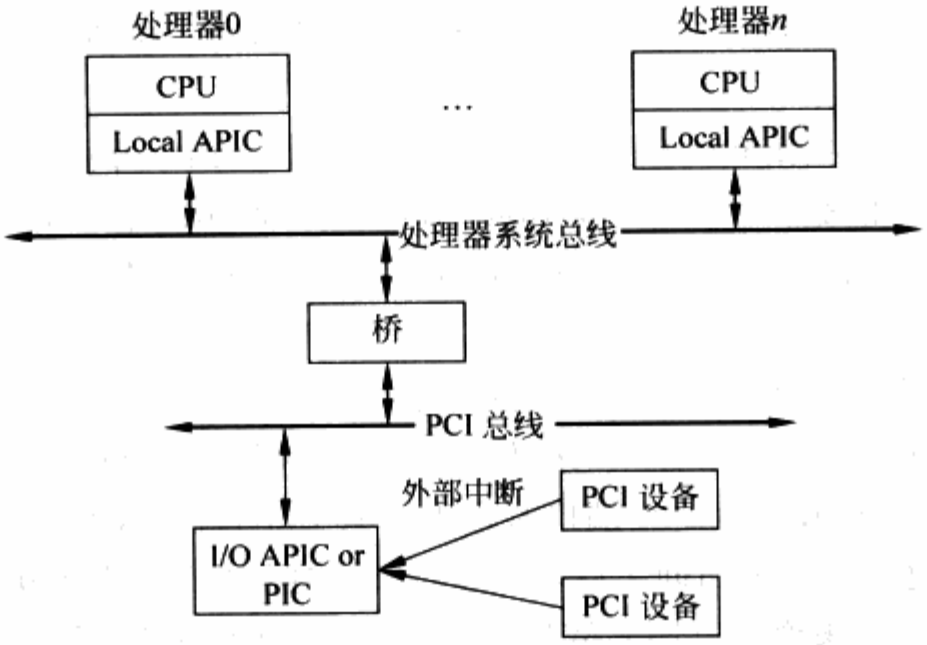

图 5-9 物理平台的中断架构

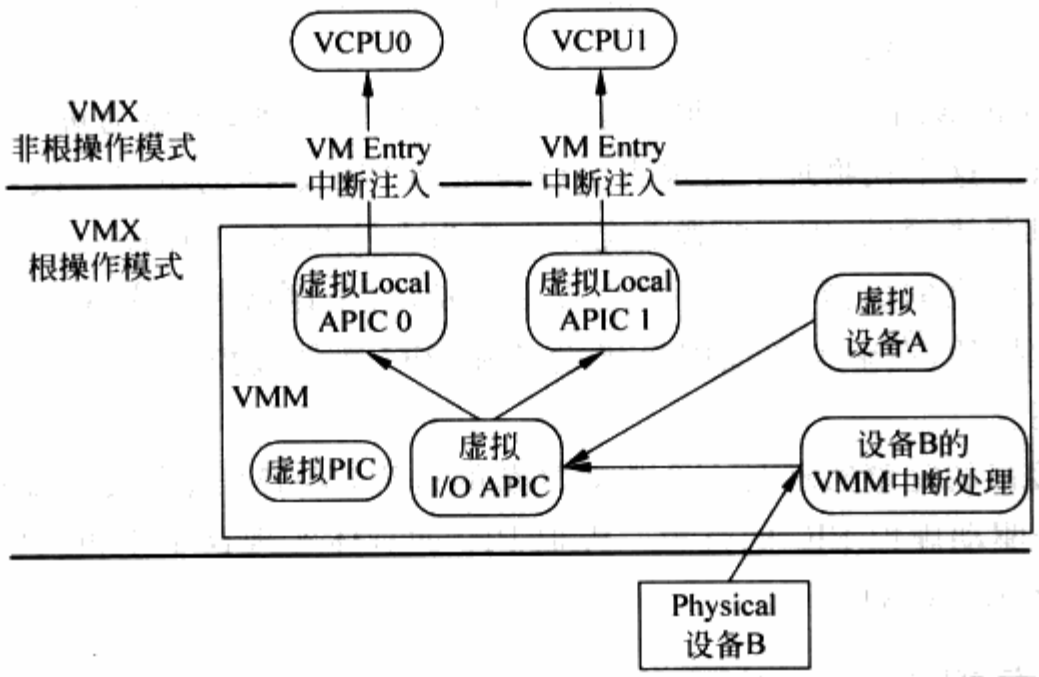

图 5-10 虚拟机中断架构

此外，PCI/PCIe 设备还支持另外一种中断方式 MSI，MSI 可以允许设备直接发送中断到 Local APIC，不需要通过中断控制器(I/O APIC)。MSI 虚拟化的原理和本节描述的原理是类似的，因此，MSI 虚拟化此处不再赘述。读者感兴趣的话，可以进一步参考 PCI/PCIe 规范中关于 MSI 的描述，以及 IA32 手册[16] 9.11 节 MESSAGE SIGNALLED INTERRUPTS 的相关内容。

5.4.2 虚拟 PIC

通过第 2 章的学习，了解了物理 PIC 的功能。PIC 本质上是芯片组的一个设备，参考文献[20]描述了 PIC 完整的规范。因此，虚拟 PIC 的实现，实际上和第 5 章中的设备模拟是

一样的，即根据 PIC 硬件规范，在软件上模拟出虚拟 PIC，为虚拟机提供和物理 PIC 一样的接口。

虚拟 PIC 首先要虚拟出和物理 PIC 一样的软件接口。PIC 为软件提供了如下接口用于操作 PIC。4 个初始化命令字(Initialization Command Words)：ICW1～4，用于初始化操作；3 个操作命令字(Operation Command Words)：OCW1～3，用于操作 PIC。

在 IA32 平台上，PIC 的 ICW1～4 和 OCW1～3 都是通过 I/O 端口访问的。因此，在 VT-x 的帮助下，VMM 很容易就可以实现这些接口的虚拟化。具体而言，这些接口是通过 I/O 端口 0x20/0x21 以及 0xA0/0xA1 来访问，因此，VMM 可以设置 VMCS 的 I/O bitmap 中的相应位，使得客户机在访问这些端口时发生 VM-Exit，便于 VMM 截获。

VMM 在截获这些接口的访问之后，下一步就是按照 PIC 硬件规范对这些接口的定义，实现相应的逻辑。举例来说，接口 OCW1 的功能是用于操作 IMR 寄存器，控制指定中断是否被屏蔽。因此，VMM 会分析客户机的 OCW1 命令，判断出是对哪个中断进行屏蔽或者解除屏蔽，VMM 继而在内部逻辑中记录指定中断是否被屏蔽。如果指定中断被屏蔽了，相应的虚拟中断就不会被提交。

此外，虚拟 PIC 除了为客户机提供正确的虚拟接口以外，还要为虚拟设备提供接口用于发送中断请求。这个在物理上表现为 I/O 设备和 PIC 之间的电气连线，在虚拟环境中由于设备和 PIC 都是虚拟的，因而两者的交互表现为直接的函数调用。

虚拟 PIC 最终会向虚拟 Local APIC 提交中断，这个在物理上表现为 PIC 和 CPU 之间的电气连线。同样的，在虚拟环境中由于设备和 PIC 都是虚拟的，因而两者的交互表现为直接的函数调用。

虚拟 PIC 接口的完整实现是一个相对复杂的过程，VMM 通常会为虚拟 PIC 维护一个内部的状态机来驱动虚拟 PIC 的行为。虚拟 PIC 的具体实现这里不再赘述，读者有兴趣可以参考 Xen 或者 KVM 的实现。

5.4.3 虚拟 I/O APIC

正如第 2 章的 2.4 节指出的那样，PIC 只适用于单 CPU 系统，对于多 CPU，必须通过 I/O APIC 来发送中断。因此，对于多 CPU 虚拟平台，必须实现虚拟 I/O APIC。

和虚拟 PIC 的实现类似，虚拟 I/O APIC 在 VMM 中也是一个虚拟设备。VMM 根据其硬件规范来实现虚拟设备。Intel ICH 系列 Datasheet 详细定义了 I/O APIC 的软件接口。

和 PIC 类似，虚拟 I/O APIC 也会根据硬件规范实现相应的接口内部逻辑，也会为虚拟设备提供接口用于发送中断请求。虚拟 I/O APIC 最后也是通过调用虚拟 Local APIC 的接口来提交中断。

操作系统通过 MMIO 的方式访问 I/O APIC，因此，VMM 的实现和虚拟 PIC 有所不同。VMM 会将虚拟 PIC 的 MMIO 地址对应的页表项置为“该页不存在”，因此，当客户机访问对应的 MMIO 寄存器时，就会发生原因为 Page Fault 的 VM-Exit。这样，VMM 就能

截获客户机对虚拟 I/O APIC 的访问，进而正确地虚拟化。

虚拟 I/O APIC 的具体实现这里不再赘述，读者可以参考 Xen 或者 KVM 的实现。

5.4.4 虚拟 Local APIC

Local APIC 是 CPU 上一个内部部件，负责接受中断。此外，还提供了产生中断的功能，例如 Local APIC Timer Interrupt 和处理器间中断 IPI。Intel 手册 [16] 的第 9 章 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER（APIC）详细定义了 Local APIC 的规范。

和虚拟 PIC、虚拟 I/O APIC 一样，虚拟 Local APIC 在 VMM 中被实现为一个模拟设备。和 I/O APIC 一样，Local APIC 提供给软件的接口是 MMIO 寄存器，因此，VMM 也通过 Page Fault 来截获客户机对 Local APIC 的访问，进一步实现内部逻辑。

虚拟 Local APIC 的最主要功能是向 VCPU 注入中断。在 VT-x 的帮助下，虚拟 Local APIC 可以借助 VM-Entry 事件注入机制简单地实现这个功能。当然，在具体的实现中，还有几个情况需要额外处理，这个在后面的内容中会进一步描述。

5.4.5 中断采集

上面介绍了虚拟机中断架构的各个组件，下面介绍这些组件如何一起工作为客户机产生虚拟中断。这个过程包括两个部分：中断的采集和中断的注入。

中断的采集是指如何将虚拟机的设备中断请求送入虚拟中断控制器。在虚拟机环境里，客户机中断有两种可能的来源。

（1）来自于软件模拟的虚拟设备，例如一个模拟出来的串口，可以产生一个虚拟中断。

（2）来自于直接分配给客户机的物理设备的中断，例如一块物理网卡，可以产生一个真正的物理中断。

采集这两种中断的方法大不相同，前者比较简单，后者则相对复杂。

对于虚拟设备而言，它们是一个软件模块。当虚拟设备需要发出中断请求时，可以通过虚拟中断控制器提供的接口函数发出中断请求，例如使用虚拟 PIC 或者虚拟 I/O APIC 提供的接口。

采集直接分配给客户机的物理设备发出的中断请求要复杂得多。一个物理设备被直接分配给一个客户机，意味着当设备发生中断时，该物理中断的处理函数位于客户操作系统中。而在虚拟化环境中，物理中断控制器由 VMM 控制，且中断发生时 CPU 的 IDT 表通常不是客户机的 IDT 表，因此，物理中断需要首先由 VMM 的中断处理函数接收，再注入给客户机。

下面通过一个例子概要地介绍物理中断采集过程。

（1）物理设备发生中断，假定设备的 IRQ 号为 14，对应的中断向量号为 0x41。

（2）CPU 收到中断，执行标准的中断处理流程，例如应答 PIC、过中断门中断自动屏蔽等。最后，CPU 跳转到 IDT 表中 0x41 表项所指定的处理函数。注意，该处理函数是 VMM

提供的，其目的是将物理中断注入给客户机。

（3）VMM 的中断处理函数对中断进行检查，发现该中断是分配给客户机的设备产生的，因此，VMM 调用虚拟中断控制器的接口函数，将中断发送给虚拟 Local APIC。之后，虚拟 Local APIC 就会在适当的时机将该中断注入给客户机，由客户操作系统的处理函数处理。

（4）在将中断事件通知客户机以后，VMM 会进行后续处理，例如开中断等。

在上述过程中，有两点信息是需要在创建客户机的过程中提供的。

（5）设备的分配信息。在第（3）步中，VMM 必须了解，中断 0x41 所对应的设备是否被分配给了某个客户机以及是哪个客户机。通常，这是在创建客户机的时候由用户决定，用户通过管理工具通知 VMM 相关的绑定信息。

（6）设备在客户机平台上的管脚信息。在第（3）步中，VMM 在调用虚拟中断控制器的接口函数时，需要提供 IRQ 号，即管脚号。需要注意的是，虚拟中断控制器的输入管脚与物理中断控制器的输入管脚并不一定相同。物理的输入管脚是由物理平台决定，而虚拟中断控制器的输入管脚是由 VMM 所提供的虚拟平台决定，通常在创建客户机的时候就已经确定了。VMM 负责在两者之间做转换。

5.4.6 中断注入

中断注入负责将虚拟中断控制器采集到的中断请求按照其优先级，逐一注入客户机虚拟处理器。这里有两个问题需要解决，首先是如何取得需要注入的最高优先级中断的相关信息，其次是如何才能将一个中断注入客户机 VCPU。

对于第一个问题，虚拟中断控制器会负责将中断按照优先级排序，VMM 只需要调用虚拟中断控制器提供的接口函数，就可以获得当前最高优先级中断的信息。

对于第二个问题，虚拟 Local APIC 提供了将中断注入客户机 VCPU 最基本的功能。VMM 可以调用虚拟 Local APIC 的接口来实现中断注入。在这里，VMM 的虚拟中断注入逻辑需要考虑下面的几个问题。

（1）如果目标 VCPU 正在物理 CPU 上运行，如何注入中断？如前所述，只能在 VM-Entry 的时候将中断注入客户机，因此，为了保证中断的及时注入，需要强迫 VCPU 发生 VM-Exit，这时就可以在 VM-Entry 返回客户机的时候注入中断。常用的使客户机发生 VM-Exit 的方法是向 VCPU 所在的物理 CPU 发送 IPI 中断。

（2）如果目标 VCPU 目前无法中断，例如 VCPU 目前正处于关中断的状态（客户机的 EFLAGS.IF 为 0），如何注入中断？Intel 的 VT-x 技术对这种情况提供了一个解决机制，即使用前面章节描述的中断窗口（Interrupt Windows）。该机制通过设置 VMCS 的一个特定字段，告诉物理 CPU，其当前运行的客户机 VCPU 有一个中断需要注入。一旦客户机 VCPU 开始可以接受中断，例如进入开中断状态，物理 CPU 会主动触发 VM-Exit，从客户机陷入到 VMM 中，虚拟中断注入模块就可以注入等待的中断了。

（3）什么时候来触发中断注入？通常的方法是，当中断采集逻辑调用虚拟中断控制器

接口请求发出中断后，虚拟中断控制器会根据内部的状态，例如虚拟 IMR/ISR 寄存器的值，来决定是否需要注入中断给客户机。其判断过程和物理中断控制器判断是否提交中断给 CPU 一样。

图 5-11 给出了中断注入逻辑的基本过程。

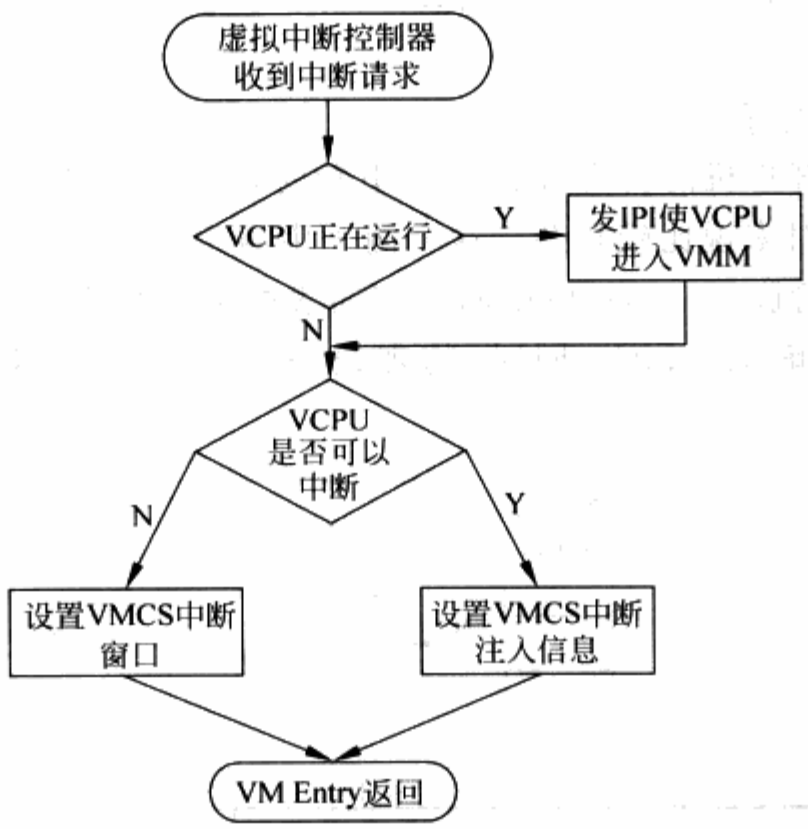

图 5-11 中断注入的过程

5.4.7 案例分析

最后分别举两个例子论述虚拟中断从产生到注入的全过程。

1. 一个简单的例子

在 2.7 节提到，物理平台上的可编程中断时钟 PIT 会定期产生时钟中断。VMM 模拟的虚拟 PIT 也一样，可以定时为客户机产生虚拟时钟中断。本节就以虚拟 PIT 为例，说明虚拟可编程时钟中断是如何被注入客户机的。

(1) 虚拟 PIT 产生中断请求时，调用虚拟中断控制器提供的接口通知虚拟 PIC/IOAPIC 自己有一个中断需要处理。

(2) 虚拟 PIC/IOAPIC 记录下这个中断请求，并检查内部寄存器，如 IMR、IRR 和 ISR 等，以决定是否需要将中断注入给客户机。如果不是，将这个中断请求保存在虚拟 PIC/IOAPIC 的内部逻辑中。当虚拟 PIC 内部状态改变以后(通常是客户机写 PIC/IOAPIC 的寄存器时)，虚拟 PIC/IOAPIC 会检查内部是否有等待处理的中断请求并重复这个过程。

(3) 如果此时需要注入中断，调用中断注入逻辑。

(4) VMM 检查客户机 VCPU 是否正在运行，如果是，则发一个 IPI 强制其进入 VMM

上下文。

(5) 在客户机 VCPU 再运行前，VMM 会检查发现该 VCPU 有中断需要注入，接着 VMM 会检查当前客户机 VCPU 是否能够被注入中断。如果能，使用虚拟中断控制器提供的接口，获取最高优先级中断的信息，设置好 VMCS 中的相应字段，使得当 VCPU 投入运行时自动去执行相应矢量号的中断处理函数。否则，设置中断窗口，等待下次 VM-Exit 之后再注入。

2. 一个复杂的例子

当被直接分配给客户机的物理设备发生中断时，VMM 需要将该中断注入给对应的客户机。这种情况相对来说比较复杂。这个例子中假定物理设备产生中断，设备中断管脚连接到 IOAPIC 的管脚 0x12，对应中断重定向表的矢量号为 0x41，并假定系统使用物理 APIC，客户机使用虚拟 APIC。同时，物理设备中断发生时，物理 CPU 的中断是开启的。

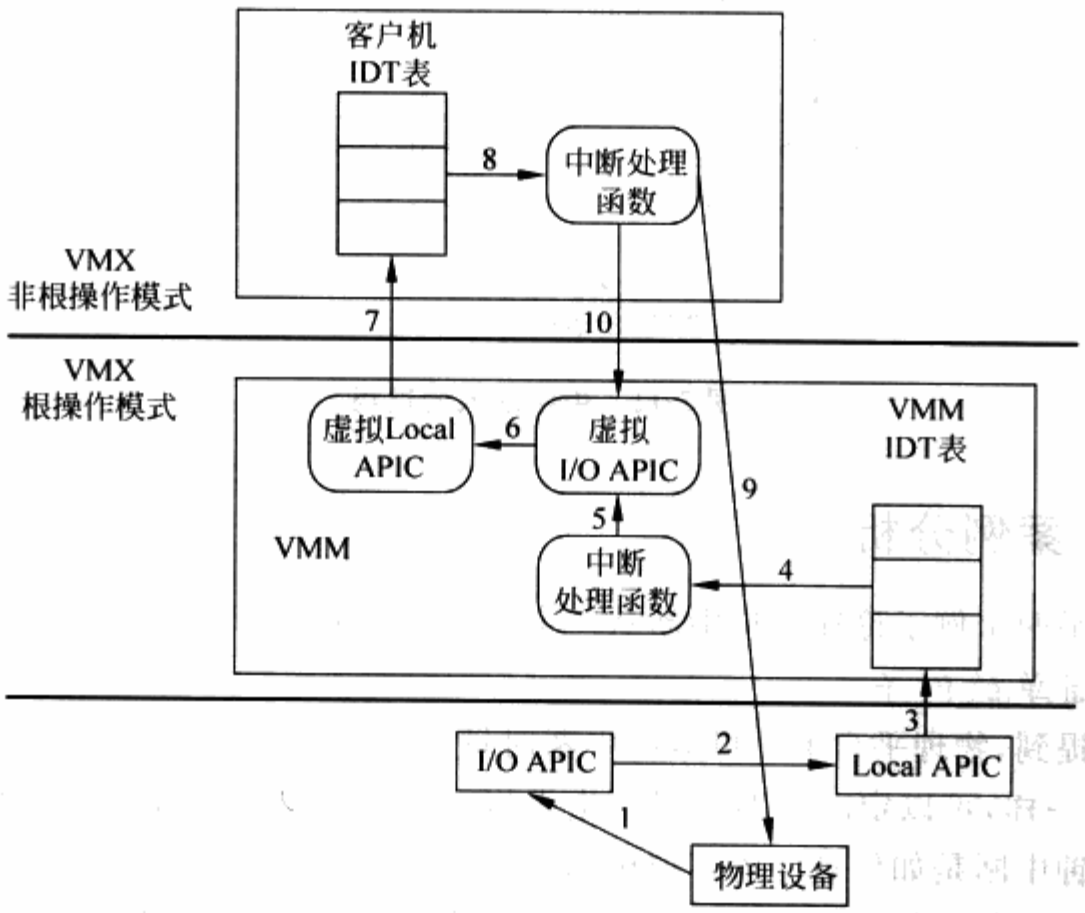

图 5-12　例子：直接分配设备的中断虚拟化

图 5-12 给出了该例子的流程。

(1) 物理设备发生中断，将中断发送给物理 I/O APIC 的管脚 0x12。

(2) 物理 I/O APIC 收到后，将管脚 0x12 转化为中断向量 0x41 发送给 Local APIC。

(3) Local APIC 将中断 0x41 注入到 CPU，CPU 跳转到 IDT 表中 0x41 表项所指定的处理函数。同时，Local APIC 的 ISR 寄存器对应 0x41 的位被置上，后面的等于或低于 0x41 的中断被屏蔽。

(4) VMM 的相关中断处理函数被执行。

(5) 中断处理函数检查发现这个中断是属于客户机的设备产生的中断,故调用虚拟中断控制器的接口函数。在将中断事件注入客户机以后,VMM 通过置一物理 IOAPIC 第 0x12 个 RTE 的屏蔽位,屏蔽后续的物理中断。VMM 向物理 Local APIC 写入 EOI,以清掉 Local APIC 的 ISR 寄存器 0x41 位,从而其他的中断也可以被接受。

VMM 中断处理函数对物理 APIC 的操作到此结束,下面是虚拟 APIC 工作的流程。

(1) 虚拟的 IOAPIC 中断控制器调用虚拟的 Local APIC 的接口函数,并将虚拟 IOAPIC 中相应的矢量号传入。虚拟 Local APIC 的 ISR 相关 bit 位被置上。

(2) 虚拟的 Local APIC 通过中断注入逻辑模块将中断注入到客户机。关于中断注入逻辑模块,参见前面的介绍。

(3) 客户机执行相关中断处理函数。

(4) 客户机中断处理函数处理物理设备的中断。

(5) 客户机向虚拟 Local APIC 写入 EOI, EOI 操作被 VMM 截获。虚拟 Local APIC 的 ISR 位被清掉,同时,虚拟 Local APIC 通知 VMM 客户已经完成对中断 0x21 的处理,VMM 清除物理 IOAPIC 第 0x12 个 RTE 的屏蔽位。

5.5 内存虚拟化

5.5.1 概述

5.2 节已经介绍了如何通过软件实现内存虚拟化,本节介绍硬件辅助的内存虚拟化。

内存虚拟化的主要任务是实现地址空间的虚拟化,内存虚拟化通过两次地址转换来支持地址空间的虚拟化,即客户机虚拟地址 GVA→客户机物理地址 GPA→宿主机物理地址 HPA 的转换。其中,GVA→GPA 的转换是由客户机软件决定的,通常是客户机操作系统通过 VMCS 中客户机状态域 CR3 指向的页表来指定; GPA→HPA 的转换是由 VMM 决定的,VMM 在将物理内存分配给客户机时就确定了 GPA→HPA 的转换,VMM 通常会用内部数据结构来记录这个映射关系。

传统的 IA32 架构只支持一次地址转换,即通过 CR3 指定的页表来实现“虚拟地址”→“物理地址”的转换。这和内存虚拟化所要求的两次地址转换产生了矛盾。可以通过将两次转换合并为一次转换来解决这个问题,即 VMM 根据 GVA→GPA→HPA 的映射关系,计算出 GVA→HPA 的映射关系,并将其写入“影子页表”。类似于“影子页表”这样的软件方法尽管能够解决问题,但是缺点也很明显。首先是实现非常复杂,例如需要考虑各种各样页表同步情况等,这样导致开发、调试和维护都比较困难。读者有兴趣可以参考一下 Xen/KVM 中影子页表的实现。此外,“影子页表”的内存开销也很大,因为需要为每个客户机进程对应的页表都维护一个“影子页表”。

为了解决这个问题，VT-x 提供了 Extended Page Table(EPT)技术，直接在硬件上支持 GVA→GPA→HPA 的两次地址转换，大大降低了内存虚拟化的难度，也进一步提高了内存虚拟化的性能。

此外，为了进一步提高 TLB 的使用效率，VT-x 还引入了 Virtual Processor ID(VPID)功能，进一步增加了内存虚拟化的性能。

5.5.2 EPT

1. EPT 原理

图 5-13 描述了 EPT 的基本原理。在原有的 CR3 页表地址映射的基础上，EPT 引入了 EPT 页表来实现另一次映射。这样，GVA→GPA→HPA 两次地址转换都由 CPU 硬件自动完成。

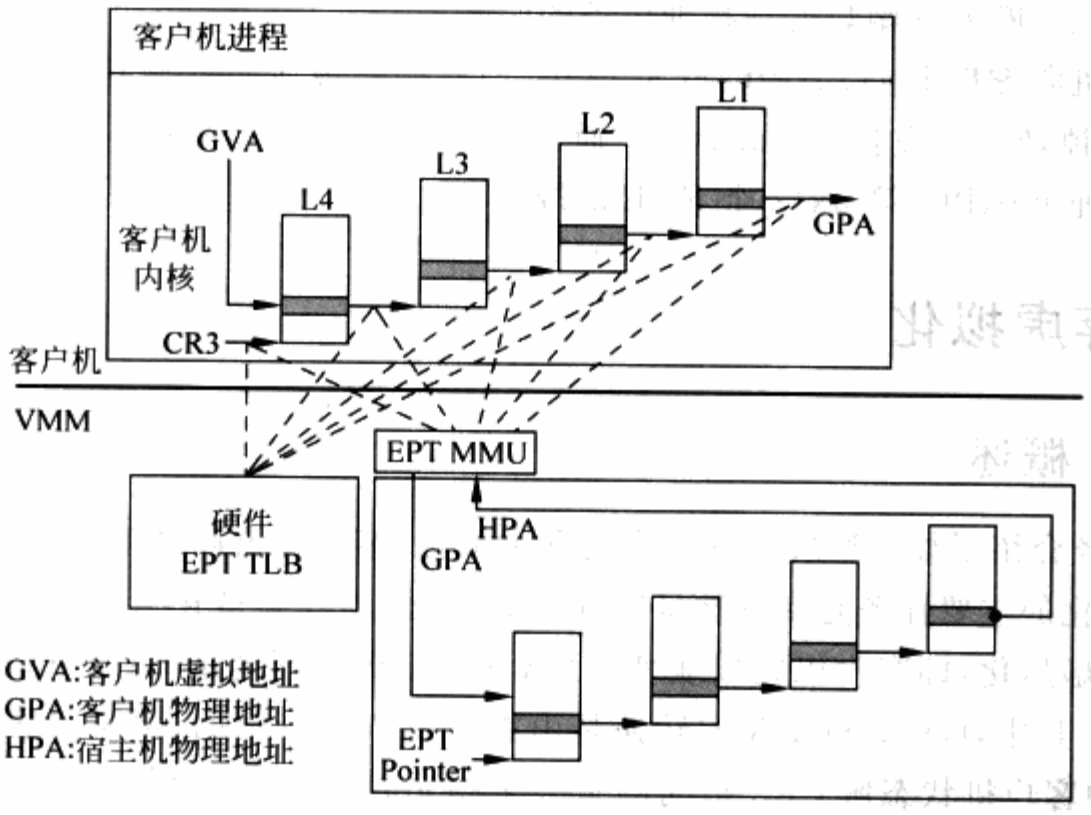

图 5-13 EPT 原理图

这里假设客户机页表和 EPT 页表都是 4 级页表，CPU 完成一次地址转换的基本过程如下。

CPU 首先会查找 Guest CR3 指向的 L4 页表。由于 Guest CR3 给出的是 GPA，因此 CPU 需要通过 EPT 页表来实现 Guest CR3 GPA→HPA 的转换。CPU 首先会查看硬件的 EPT TLB，如果没有对应的转换，CPU 会进一步查找 EPT 页表，如果还没有，CPU 则抛出 EPT Violation 异常由 VMM 来处理。

获得 L4 页表地址后，CPU 根据 GVA 和 L4 页表项的内容，来获取 L3 页表项的 GPA。如果 L4 页表中 GVA 对应的表项显示为“缺页”，那么 CPU 产生 Page Fault，直接交由

Guest Kernel 处理。注意，这里不会产生 VM-Exit。获得 L3 页表项的 GPA 后，CPU 同样要通过查询 EPT 页表来实现 L3 GPA→HPA 的转换，过程和上面一样。

同样的，CPU 会依次查找 L2、L1 页表，最后获得 GVA 对应的 GPA，然后通过查询 EPT 页表获得 HPA。从上面的过程可以看出，CPU 需要 5 次查询 EPT 页表，每次查询都需要 4 次内存访问，因此最坏情况下总共需要 20 次内存访问。EPT 硬件通过增大 EPT TLB 来尽量减少内存访问。

2. EPT 的硬件支持

为了支持 EPT，VT-x 规范在 VMCS 的“VM-Execution 控制域”中提供了 Enable EPT 字段。如果在 VM-Entry 的时候该位被置上，EPT 功能就会被启用，CPU 会使用 EPT 功能进行两次转换。

EPT 页表的基地址是由 VMCS“VM-Execution 控制域”的 Extended page table pointer 字段来指定的，它包含了 EPT 页表的宿主机物理地址。

EPT 是一个多级页表，每级页表的表项格式是相同的，如表 5-8 所示。

表 5-8　EPT 页表的表项格式

字段名称	描　　述
ADDR	下一级页表的物理地址。如果已经是最后一级页表，那么就是 GPA 对应页的物理地址
SP	超级页(super page)：所指向的页是大小超过 4KB 的超级页。CPU 在遇到 SP=1 时，就会停止继续往下查询。对于最后一级页表，这一位可以供软件使用
X	可执行。X=1 表示该页是可执行的
R	可读。R=1 表示该页是可读的
W	可写。W=1 表示该页是可写的

EPT 页表转换过程和 CR3 页表转换是类似的。图 5-14 展现了 CPU 使用 EPT 页表进行地址转换的过程。

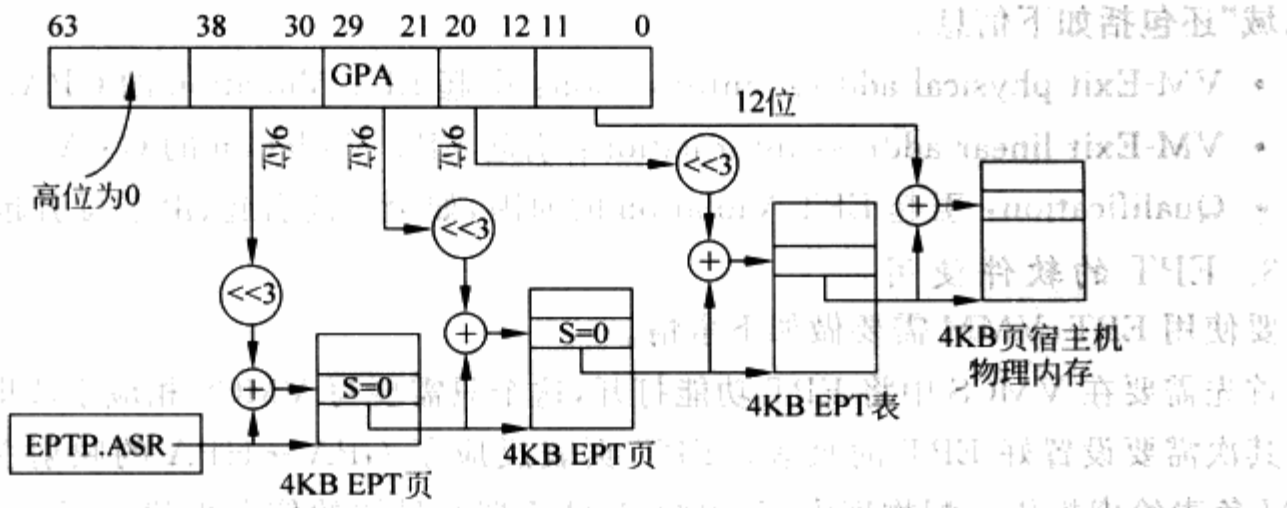

图 5-14　EPT 页表转换

EPT 通过 EPT 页表中的 SP 字段支持大小为 2MB 或者 1GB 的超级页。图 5-15 给出了 2MB 超级页的地址转换过程。和图 5-14 的不同点在于，当 CPU 发现 SP 字段为 1 时，就停止继续向下遍历页表，而是直接转换了。

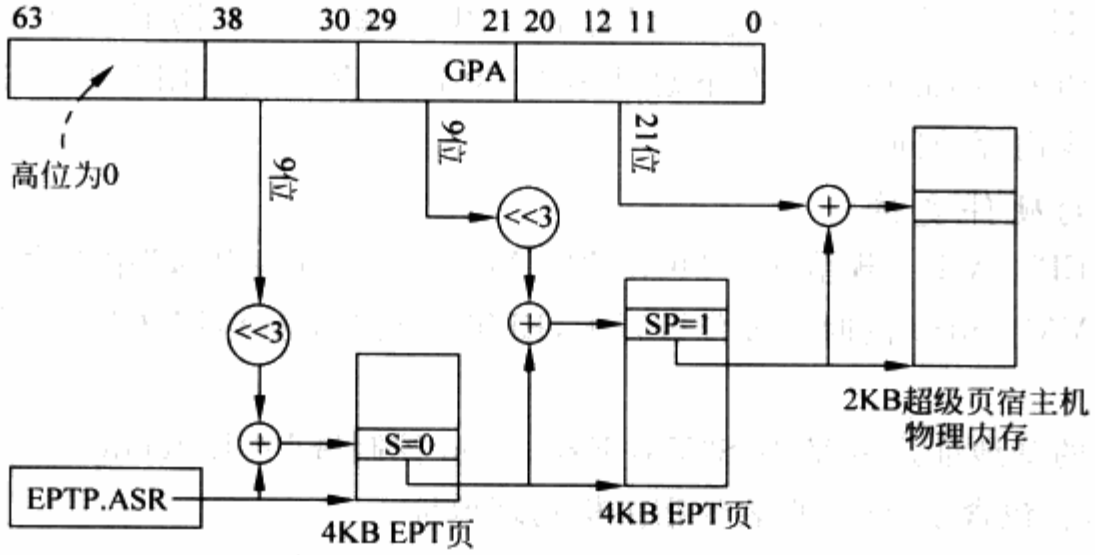

图 5-15　EPT 页表转换：超级页

EPT 同样会使用 TLB 缓冲来加速页表的查找过程。因此，VT-x 还提供了一条新的指令 INVEPT，可以使 EPT 的 TLB 项失效。这样，当 EPT 页表有更新时，CPU 可以执行 INVEPT 使旧的 TLB 失效，使 CPU 使用新的 EPT 表项。

和 CR3 页表会导致 Page Fault 一样，使用 EPT 之后，如果 CPU 在遍历 EPT 页表进行 GPA→HPA 转换时，也会发生异常。

(1) GPA 的地址位数大于 GAW。

(2) 客户机试图读一个不可读的页(R=0)。

(3) 客户机试图写一个不可写的页(W=0)。

(4) 客户机试图执行一个不可执行的页(X=0)。

发生异常时，CPU 会产生 VM-Exit，退出原因为 EPT Violation。VMCS 的“VM-Exit 信息域”还包括如下信息。

- VM-Exit physical-address information：引起 EPT Violation 的 GPA。
- VM-Exit linear-address information：引起 EPT Violation 的 GVA。
- Qualification：引起 EPT Violation 的原因，如由于读引起、由于写引起等。

3. EPT 的软件使用

要使用 EPT，VMM 需要做如下事情。

首先需要在 VMCS 中将 EPT 功能打开，这个只需要写 VMCS 相应字段即可。

其次需要设置好 EPT 的页表。EPT 页表反应了 GPA→HPA 的映射关系。由于是 VMM 负责给虚拟机分配物理内存，因此，VMM 拥有足够的信息来建立 EPT 页表。此外，如果 VMM 给虚拟机分配的物理内存足够连续的话，VMM 可以在 EPT 页表中尽量使用超

级页，这样有利于提高 TLB 的性能。

当 CPU 开始使用 EPT 时，VMM 还需要处理 EPT Violation。通常来说，EPT Violation 的来源有如下几种。

(1) 客户机访问 MMIO 地址。这种情况下，VMM 需要将请求转给 I/O 虚拟化模块。

(2) EPT 页表的动态创建。有些 VMM 采用懒惰方法，一开始 EPT 页表为空，当第一次使用发生 EPT Violation 时再建立映射。

由此可以看出，EPT 相对于传统的“影子页表”方法，其实现大大地简化了。而且，由于客户机内部的 Page Fault 不用发生 VM-Exit，也大大减少了 VM-Exit 的个数，提高了性能。此外，EPT 只需要维护一张 EPT 页表，不像“影子页表”那样需要为每个客户机进程的页表维护一张影子页表，也减少了内存的开销。

5.5.3 VPID

TLB 是页表项的缓存，对地址转换的效率至关重要。TLB 需要和对应的页表一起工作才有效。因此，当页表发生切换时，TLB 原有的内容也就失效了，CPU 需要使用 INVLPG 指令使其所有项失效，这样才不会影响之后页表的工作。例如，我们知道进程切换时，需要切换进程地址空间(通过切换页表的起始物理地址 CR3)，使前一个进程的 TLB 项全部失效。

类似地，在每次 VM-Entry 和 VM-Exit 时，CPU 会强制 TLB 内容全部失效，以避免 VMM 以及不同虚拟机虚拟处理器之间 TLB 项的混用，因为硬件无法区分一个 TLB 项是属于 VMM 还是某一特定的虚拟机虚拟处理器。

VPID 是一种硬件级的对 TLB 资源管理的优化。通过在硬件上为每个 TLB 项增加一个标志，来标识不同的虚拟处理器地址空间，从而区分开 VMM 以及不同虚拟机的不同虚拟处理器的 TLB。换而言之，硬件具备了区分不同的 TLB 项属于不同虚拟处理器地址空间(对应于不同的虚拟处理器)的能力。这样，硬件可以避免在每次 VM-Entry 和 VM-Exit 时，使全部 TLB 失效，提高了 VM 切换的效率。并且，由于这些继续存在的 TLB 项，硬件也避免了 VM 切换之后的一些不必要的页表遍历，减少了内存访问，提高了 VMM 以及虚拟机的运行速度。

VT-x 通过在 VMCS 中增加两个域来支持 VPID。第一个是 VMCS 中的 Enable VPID 域，当该域被置上时，VT-x 硬件会启用 VPID 功能。第二个是 VMCS 中的 VPID 域，用于标识该 VMCS 对应的 TLB。VMM 本身也需要一个 VPID，VT-x 规定虚拟处理器标志 0 被指定用于 VMM 自身，其他虚拟机虚拟处理器不得使用。

因此，在软件上使用 VPID 非常简单，主要做两件事情。首先是为 VMCS 分配一个 VPID，这个 VPID 只要是非 0 的，且和其他 VMCS 的 VPID 不同就可以了；其次是在 VMCS 中将 Enable VPID 置上，剩下的事情硬件会自动处理。

5.6 I/O 虚拟化的硬件支持

5.6.1 概述

在软件虚拟化章节已经阐述过如何用软件的方式实现 I/O 虚拟化，目前流行的“设备模拟”和“类虚拟化”都有各自的优点，以及与生俱来的缺点。前者通用性强，但性能不理想；后者性能不错，却又缺乏通用性。为此，英特尔公司发布了 VT-d 技术（Intel（R）Virtualization Technology for Directed I/O），以帮助虚拟软件开发者实现通用性强、性能高的新型 I/O 虚拟化技术。

在介绍 VT-d 技术前，先量化一下评价 I/O 虚拟技术的两个指标——性能和通用性。性能不必说，越接近无虚拟机环境下的 I/O 性能越好；通用性主要是和全虚拟化挂钩的，使用的 I/O 虚拟化技术对客户操作系统越透明，则通用性越强。通过 VT-d 技术，可以很好地实现这两个指标，无须像“设备模拟”和“类虚拟化”两种技术一样，为了提高某个指标而使另一个指标打折。

如何实现这两个指标呢？对于高性能，最直接的方法就是让客户机直接使用真实的硬件设备，这样客户机的 I/O 操作路径几乎和无虚拟机环境下的 I/O 路径相同，获得高性能是理所当然的；对于通用性，就要用全虚拟化的方法，让客户机操作系统能够使用自带的驱动程序发现设备、操作设备。下面先来看看实现这些目标所面临的挑战。

客户机直接操作设备面临如下两个问题。

（1）如何让客户机直接访问到设备真实的 I/O 地址空间（包括端口 I/O 和 MMIO）。

（2）如何让设备的 DMA 操作直接访问到客户机的内存空间？要知道，设备可不管系统中运行的是虚拟机还是真实操作系统，它只管用驱动提供给它的物理地址做 DMA。

通用性面临的问题和（1）是类似的，要有一种方法把设备的 I/O 地址空间告诉给客户操作系统，并能让驱动通过这些地址访问到设备真实的 I/O 地址空间。VT-x 技术已经能够解决第一个问题，可以允许客户机直接访问物理的 I/O 空间。Intel 的 VT-d 技术则让第二个问题的解决成为可能，它提供了 DMA 重映射技术，以帮助 VMM 的实现者达到目标。

VT-d 技术通过在北桥（MCH）引入 DMA 重映射硬件，以提供设备重映射和设备直接分配的功能。在启用 VT-d 的平台上，设备所有的 DMA 传输都会被 DMA 重映射硬件截获。根据设备对应的 I/O 页表，硬件可以对 DMA 中的地址进行转换，使设备只能访问到规定的内存。使用 VT-d 后，设备访问内存的架构如图 5-16 所示。

图 5-16(a)中是没有 VT-d 的平台，此时设备的 DMA 可以访问整个物理内存。图 5-16(b)是启用 VT-d 的情况，此时，设备只能访问指定的物理内存。相信读者有似曾相识的感觉，是的，这和使用页表将进程的线性地址空间映射到指定物理内存区域的思想一样，只不过对

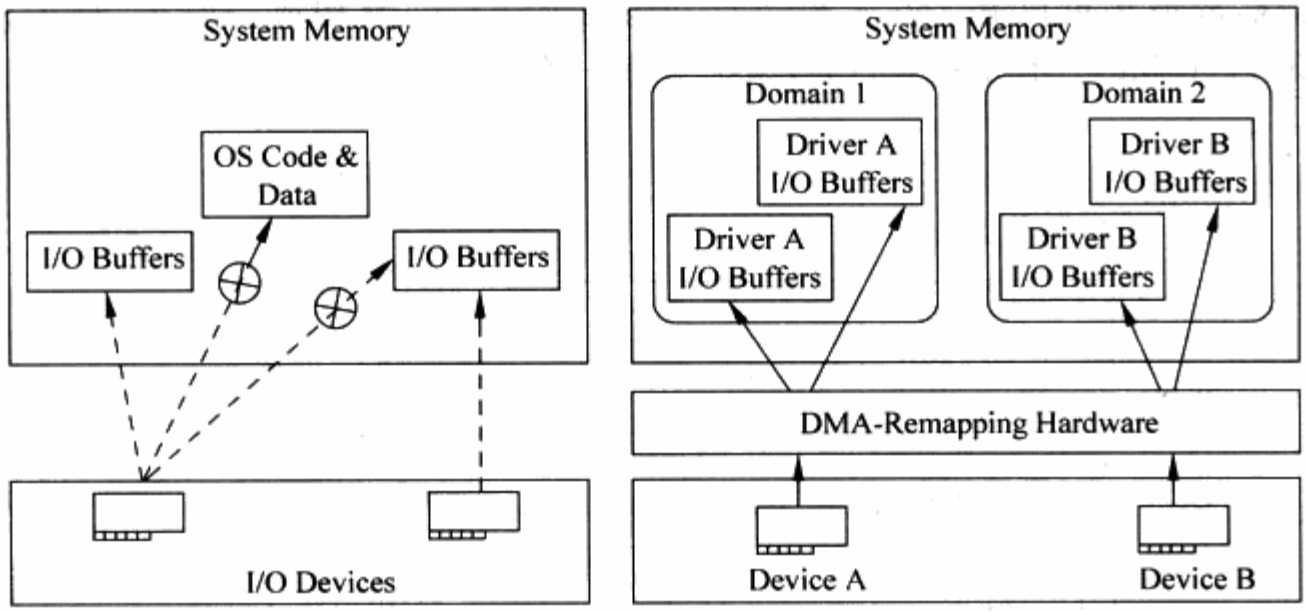

图 5-16 使用 VT-d 后访问内存架构

象换成了设备。下面介绍 VT-d 中核心的 DMA 重映射技术以及如何探测 DMA 重映射硬件，设备分配的内容在 6.7 节中介绍。VT-d 技术较为复杂，限于篇幅，只能对主要技术进行介绍，完整的论述请参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”。

5.6.2 VT-d 技术

在上一节提到的诸多难点中，最主要的难题是 DMA 问题。设备对系统中运行的软件是一无所知的，在进行 DMA 时，设备唯一做的是向(从)驱动程序告知的“物理地址”复制(读取)数据。在内存虚拟化相关章节已经知道，虚拟机环境下客户机使用的是 GPA，则客户机的驱动直接操作设备时也是用 GPA。而设备进行 DMA，需要用 MPA，如何在 DMA 时将 GPA 转换成 MPA 就成了关键问题。要知道，通常无法通过软件的方法截获设备的 DMA 操作，VT-的技术提供的 DMA 重映射就是为解决这个问题而提出的。

1. DMA 重映射(DMA Remapping)

先来回忆一下第 2 章中提到的 PCI 总线结构，通过 BDF 可以索引到任何一条总线上的任何一个设备。同样，DMA 的总线传输中包含一个 BDF 以标识该 DMA 传输是由哪个设备发起的。在 VT-d 技术中，标识 DMA 操作发起者的结构称为源标识符(Source Identifier)。对于 PCI 总线，VT-d 使用 BDF 作为源标识符，在下面的内容中提到 BDF 均代表源标识符。其格式在第 2 章已经给出，读者也可以参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.2.1 节获取详细内容。

除了 BDF 外，VT-d 还提供了两种数据结构来描述 PCI 架构，分别是根条目(Root Entry)和上下文条目(Context Entry)。

(1) 根条目：用于描述 PCI 总线，每条总线对应一个根条目。由于 PCI 架构支持最多

256 条总线，故最多可以有 256 个根条目。这些根条目一起构成一张表，称为根条目表(Root Entry Table)。有了根条目表，系统中每一条总线都会被描述到。图 5-17 是根条目的结构。

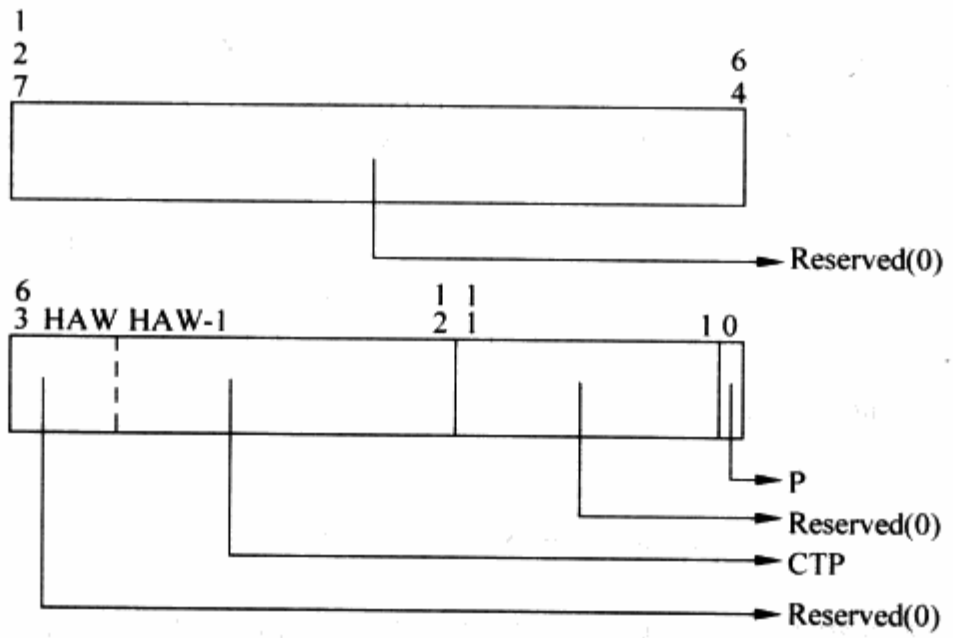

图 5-17　根条目结构

主要字段如下。

- P：存在位。为 0 时条目无效，来自该条目所代表总线的所有 DMA 传输被屏蔽。为 1 时，该条目有效。
- CTP(Context Table Pointer，上下文表指针)：指向上下文条目表。

(2) 上下文条目：用于描述某个具体的 PCI 设备，这里的 PCI 设备是指逻辑设备(见第 2 章关于 BDF 中 function 字段的阐述)。一条 PCI 总线上最多有 256 个设备，故有 256 个上下文条目，它们一起组成上下文条目表(Context Entry Table)。通过上下文条目表，可描述某条 PCI 总线上的所有设备。图 5-18 是上下文条目的结构。

主要字段如下。

- P：存在位。为 0 时条目无效，来自该条目所代表设备的所有 DMA 传输被屏蔽。为 1 时，表示该条目有效。
- T：类型，表示 ASR 字段所指数据结构的类型。目前，VT-d 技术中该字段为 0，表示多级页表。
- ASR(Address Space Root，地址空间根)：实际是一个指针，指向 T 字段所代表的数据结构，目前该字段指向一个 I/O 页表(见后面的内容)。
- DID(Domain ID，域标识符)：VT-d 技术中 Domain 的具体含义请参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.1 节的定义。在此，读者可以理解为本文中的客户机，DID 可以看作用于唯一标识该客户机的标识符，例如 Guest ID。

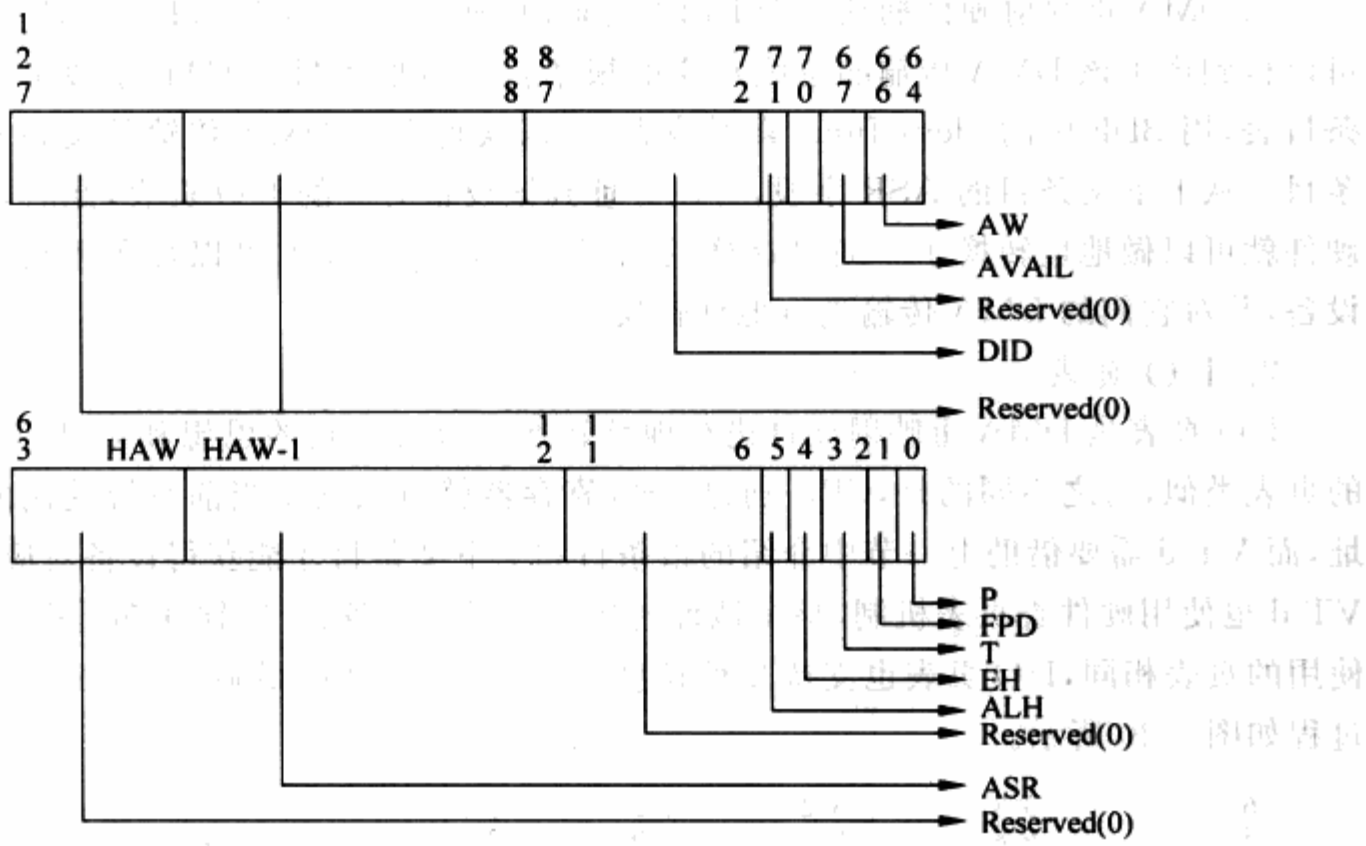

图 5-18 上下文条目结构

其余字段的具体解释请参见“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.2.3 节。

根条目表和上下文条目表一起构成了图 5-19 所示的两级结构。

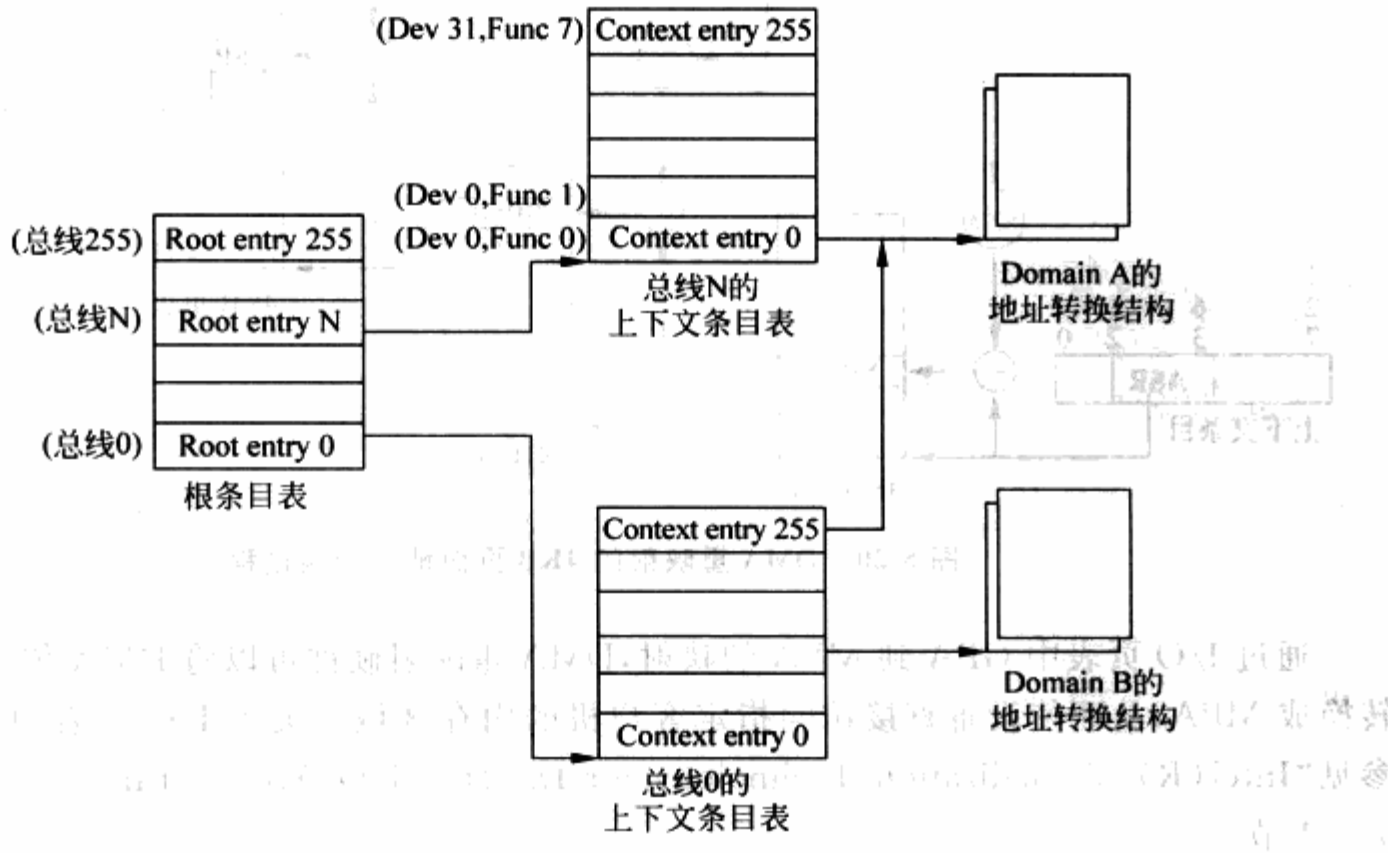

图 5-19 根条目表和上下文条目表构成的两级结构

当 DMA 重映射硬件捕获一个 DMA 传输时，通过其中 BDF 的 bus 字段索引根条目表，可以得到产生该 DMA 传输的总线对应的根条目。由根条目的 CTP 字段可以获得上下文条目表，用 BDF 中的{dev：func}索引该表，可以获得发起 DMA 传输的设备对应的上下文条目。从上下文条目的 ASR 字段，可以寻址到该设备对应的 I/O 页表，此时，DMA 重映射硬件就可以做地址转换了。通过这样的两级结构，VT-d 技术可以覆盖平台上所有的 PCI 设备，并对它们的 DMA 传输进行地址转换。

2. I/O 页表

I/O 页表是 DMA 重映射硬件进行地址转换的核心。它的思想和 CPU 中 paging 机制的页表类似，与之不同的是，CPU 通过 CR3 寄存器就可以获得当前系统使用的页表的基地址，而 VT-d 需要借助上一节中介绍的根条目和上下文条目才能获得设备对应的 I/O 页表。VT-d 也使用硬件查页表机制，整个地址转换过程对于设备、上层软件都是透明的。与 CPU 使用的页表相同，I/O 页表也支持几种粒度的页面大小，其中最典型的 4KB 页面地址转换过程如图 5-20 所示。

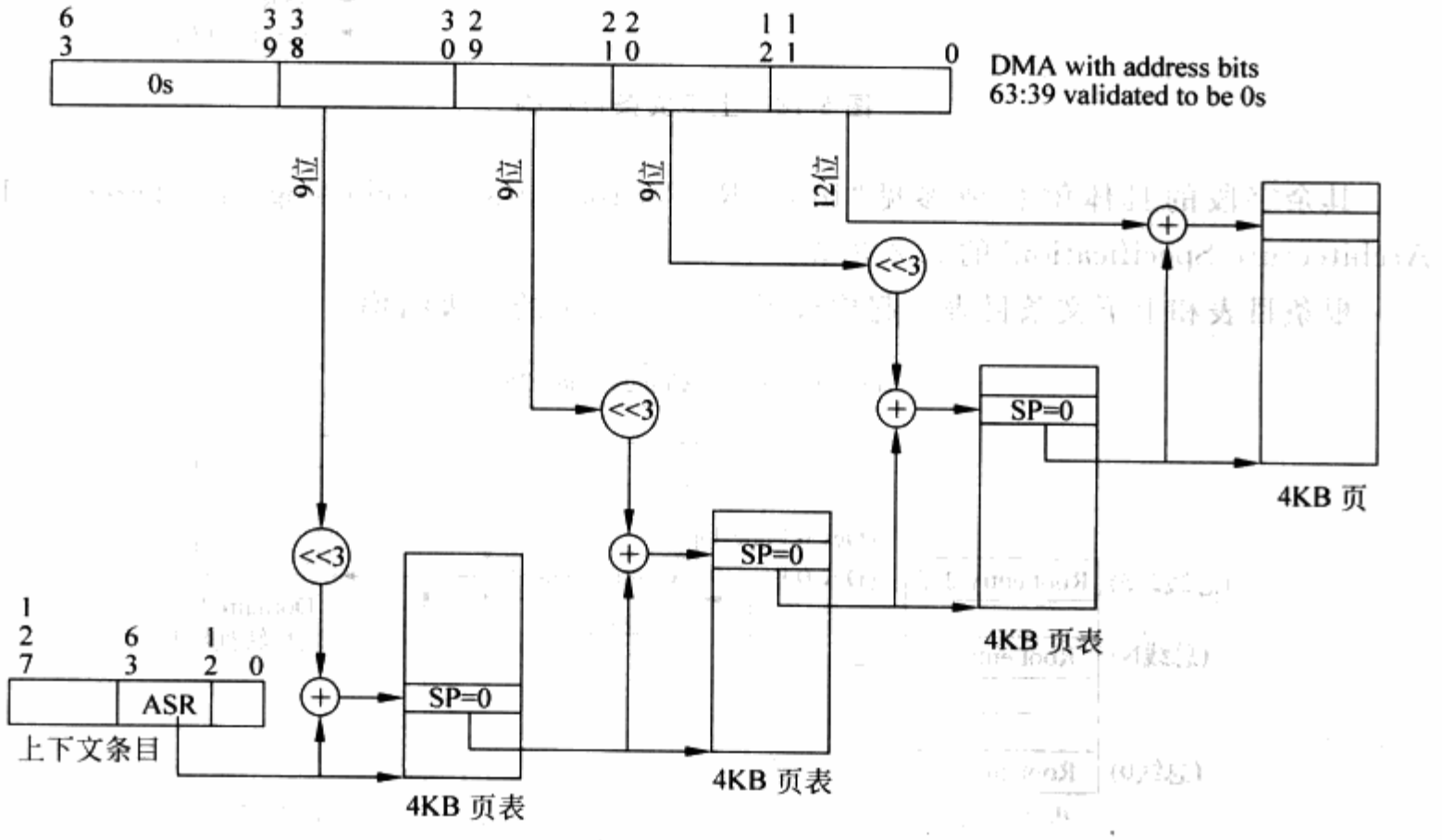

图 5-20　DMA 重映射的 4KB 页面地址转换过程

通过 I/O 页表中 GPA 到 MPA 的映射，DMA 重映射硬件可以将 DMA 传输中的 GPA 转换成 MPA，从而使设备直接访问指定客户机的内存区域。关于 I/O 页表的详细内容请参见“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.3.1 节。

3. VT-d 硬件缓存

VT-d 硬件使用了大量的缓存以提高效率。其中，和地址转换相关的缓存称为 IOTLB，它和 CPU 中的 TLB 功能一样。此外，对于上下文条目，VT-d 硬件提供了上下文条目表。当软件修改了 I/O 页表、上下文条目表之后，要负责对这些缓存进行刷新。

VT-d 对两种缓存分别提供三种粒度的刷新操作。

(1) 全局刷新(Global Invalidation)：整个 IOTLB 或上下文条目表中所有条目无效。

(2) 客户机粒度刷新(Domain-Selective Invalidation)：IOTLB 中或上下文条目表中和指定客户机相关的地址条目或上下文条目无效。

(3) 局部刷新：对于 IOTLB，称为 Domain vPage-Selective Invalidation，指定客户机某一地址范围内的页面映射条目无效。对于上下文条目表，称为 Device Selective Invalidation，和某个指定设备相关的上下文条目无效。

硬件可以实现上述三种刷新操作的一种或多种。对于系统软件来说，它并不知道自己发起的刷新操作被硬件使用哪一种粒度的刷新操作完成。具体内容请参考“Intel(R) Virtualization Technology for Directed I/O Architecture Specification”的 3.2.3.1 和 3.3.1.3 节。

4. VT-d 硬件的探测

和所有的硬件一样，在使用 DMA 重映射硬件之前需要对它进行探测。VT-d 通过 BIOS 的 ACPI 表向上层软件汇报平台 DMA 重映射硬件的情况，硬件由三个主要数据结构描述。

- DMAR(DMA Remapping Reporting)：该结构汇报平台 VT-d 相关硬件的整体情况，可以看作一个总表。其主要字段如表 5-9 所示。

表 5-9 DMAR 结构

字 段 名	描 述
Length	以字节数表示 DMAR 表占用的内存大小
HAW	该平台支持的 DMA 操作可寻址的最大物理地址空间
DMA Remapping Structures	指向下一级硬件描述数据结构，包括 DHRD 和 RMRR 两种

本文中，只介绍 DMA Remapping Structures 字段为 DHRD 的情况。

- DHRD(DMA Remapping Hardware Unit Definition)：用于描述 DMA 重映射硬件，一个 DHRD 结构对应一个 DMA 重映射硬件。典型的实现是平台只有一个 DMA 重映射硬件并管辖所有设备，但 VT-d 技术也支持一个平台多个 DMA 重映射硬件。DHRD 的主要字段如表 5-10 所示。关于 INCLUDE_ALL 和非 INCLUDE_ALL 模式，可以换一种方式理解。对于前者，表示该 DHRD 管辖所有设备(平台只有一个 DMA 重映射硬件的情况)；对于后者，该 DHRD 只管辖 Device Scope 字段描述的设备。

表 5-10　DHRD 结构

字　段　名	描　　述
Flag	指明该 DMA 重映射硬件截获哪些设备的 DMA 传输，它有两种模式。 (1) INCLUDE_ALL 模式：在此模式下，截获所有未被其他 DMA 重映射硬件截获的 DMA 传输；对于平台只有一个 DMA 重映射硬件的情况下，该模式截获所有设备的 DMA 传输。 (2) 非 INCLUDE_ALL 模式：在此模式下，只截获来自 Device Scope 字段所描述设备的 DMA 传输
Register Base Address	指向 DMA 重映射硬件寄存器的基地址，系统软件通过读写这些寄存器操作 DMA 重映射硬件
Device Scope	设备域，数组结构，成员为 DSS。当 DMA 重映射硬件工作在非 INCLUDE_ALL 模式时，该数组所包含设备的 DMA 传输被当前 DMA 重映射硬件截获

- DSS(Device Scope Structure)：描述 DHRD 所管辖的设备。DHRD 的 Device Scope 指向的数组中的每个元素以 DSS 结构表示。该结构可以代表两种类型的设备，一种是 PCI 终端设备，一种是 PCI 桥设备。该结构有三个重要字段，如表 5-11 所示。

表 5-11　DSS 结构

字　段　名	描　　述
Type	指明该 DSS 描述的设备类型，1 代表 PCI 终端设备，2 代表 PCI 桥设备
Starting Bus Number	当 DSS 描述的设备位于某特定 PCI 桥下时，该 PCI 桥下第一条总线的总线号
PCI Path	当 Type 字段为 PCI 桥时，指明该桥所属设备的路径

三种数据结构构成了图 5-21 所示的层次。

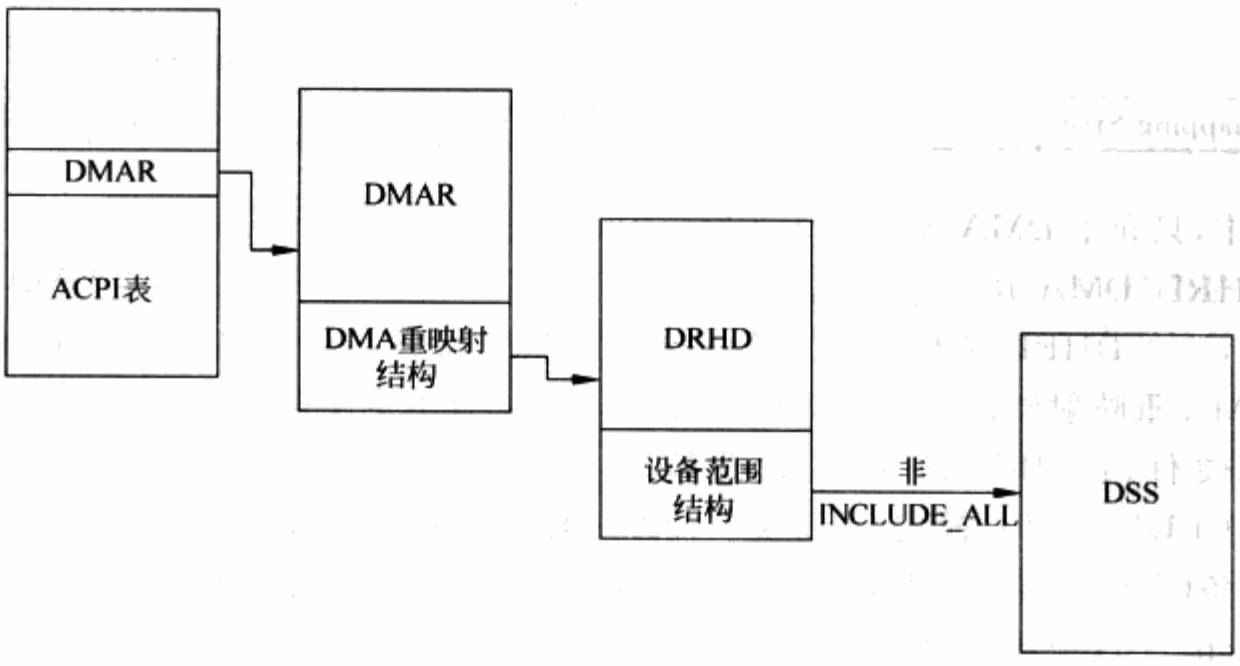

图 5-21　DMAR、DRHD 和 DSS 的层次

其中,第一级是 ACPI 表,从中获得 DMAR,然后依据前面描述的各个结构的各字段,可以解析出平台每个 DMA 重映射硬件的所有信息,例如该硬件的寄存器地址、该硬件管辖的设备等。

本节只介绍了探测 VT-d 硬件用到的主要数据结构以及它们的主要字段,详细信息请参考"Intel(R) Virtualization Technology for Directed I/O Architecture Specification"的第 5 章。

5.7 I/O 虚拟化的实现

5.7.1 概述

在上一节,介绍了英特尔公司的 VT-d 技术为 I/O 虚拟化带来了怎样的便利。本节中,会介绍如何将 VT-d 技术运用到 VMM 的 I/O 虚拟化实现中,并会对上节提及的设备直接分配技术进行介绍。

5.7.2 设备直接分配

在"设备模拟"和"类虚拟化"两种 I/O 虚拟化技术中,所有客户机都共享平台硬件设备。考虑这样一种情况,当 VMM 运行在一台拥有 10 块网卡的服务器上时,前两种技术完全可能只使用一块网卡来满足所有客户机的网络 I/O 需求,这必然导致了低性能和资源的浪费。设备直接分配技术很好地解决了这个问题。可以把某个设备直接分配给一个客户机,让客户机的 I/O 访问直接访问到设备的 I/O 地址空间,从宏观上看,这个客户机直接操作了平台的硬件设备。还记得第 2 章 I/O 架构中的内容吗? I/O 只有如下三个方面。

(1) 驱动程序通过 I/O 地址空间操作设备,即设备直接分配技术解决的问题。

(2) 设备通过 DMA 读取/复制数据。

(3) 已经由 VT-d 的 DMA 重映射技术解决了,(3)中断。在 2.4 节已经介绍过。

设备直接分配的一个问题是如何阻止来自未分配到该设备的客户机的 I/O 访问。例如,系统中有三个客户机(0、1、2),其中客户机 1 分配到了网卡 A,则要阻止客户机 0、客户机 2 对网卡 A 的访问,一个最直接的方法是隐藏,让客户机 0、客户机 1 认为网卡 A 根本就不存在。实际上,无论是"设备模拟"还是"类虚拟化"技术,平台的硬件对于客户机都是透明的,所谓隐藏主要是针对运行"设备模拟器"的客户机/宿主机,或拥有"类虚拟化"后端驱动的客户机/宿主机而言的。隐藏的方式视具体的情况而定,例如可以在拥有硬件设备的客户机/宿主机加载驱动程序前,先给要分配出去的设备加载一个伪驱动作为占位符。由于没有真正的驱动程序,该设备就不会被访问到。

设备直接分配的另一个问题是如何让客户机的 I/O 操作直接访问到设备真实的 I/O 地址空间,该问题在下一节介绍。至此,读者应该理解可以通过设备直接分配将某一设备直

接分配给某个客户机，并让客户机直接操作该设备。

5.7.3 设备 I/O 地址空间的访问

在本节中，只讨论 PCI 设备在设备直接分配的情况下，客户机如何直接访问设备的真实 I/O 地址空间。

在第 2 章已经介绍了，PCI 设备的 I/O 地址空间通过 PCI BAR（Base Address Register）报告给操作系统。为此，有两种选择供设备直接分配技术使用。

（1）将设备的真实 PCI BAR 报告给客户机，并通过 VMCS 的 I/O bitmap 和 EPT 使客户机的端口 I/O 和 MMIO 都不引起 VM-Exit，则客户操作系统的驱动程序可以直接访问设备的 I/O 地址空间。

（2）建立转换表，报告虚拟的 PCI BAR 给客户机，当客户机访问到虚拟的 I/O 地址空间时，VMM 负责截获操作，并通过转换表把 I/O 请求转发到设备的 I/O 地址空间。

两种方法中，方法（1）是高效的（不引起 VM-Exit）和简单的（直接报告真实的 PCI BAR 给客户机），但在实际运用中会存在一些问题。通常，VMM 产品会使用多种 I/O 虚拟化技术，客户机的 I/O 请求，可能一部分由“设备模拟”技术满足，一部分由设备直接分配技术满足。例如一个客户机，它的显卡是由“设备模拟器”模拟的，但网卡又是操作的真实设备。在第 2 章 2.6 节提到过，设备的 PCI BAR 通常是 BIOS 配置并由操作系统直接使用的。那在上述情况中，由“设备模拟器”提供的设备的 PCI BAR 由虚拟 BIOS 配置，而真实设备的 PCI BAR 由平台的 BIOS 配置，两者之间就可能产生冲突。当这种情况发生时，在操作系统看来就是资源冲突，很可能停用其中一个设备而满足另一个设备。此外，操作系统是有权利修改设备的 PCI BAR 的，但应该阻止客户机直接修改真实设备的 PCI BAR，这是为了防止真实设备之间的 PCI BAR 冲突，以及在客户机销毁时把设备分配给其他客户机使用。由于这些原因，在实现设备直接分配技术时，通常采用的是方法（2），即建立转换表。根据 I/O 地址空间的划分，转换表分为 Port I/O 转换表和 MMIO 转换表。

对于端口 I/O，在介绍 VMCS 时已经提到过，可以通过 I/O bitmap 来控制客户机访问某个端口是否引起 VM-Exit。这样，完全可以使用“设备模拟器”的虚拟 BIOS（或是其他手段，取决于 VMM 使用的 I/O 虚拟技术）为分配给客户机的真实设备生成虚拟的 PCI BAR，将它报告给客户操作系统，并修改 I/O bitmap 使客户机在访问这些 I/O 端口时产生 VM-Exit。同时，VMM 维护一张虚拟 PCI BAR 到真实 PCI BAR 的映射表。当客户机通过虚拟的 PCI BAR 发起 I/O 操作时，会因为 VM-Exit 陷入到 VMM 中，VMM 即可以通过转换表获得真实设备的 I/O 端口，帮客户机将请求转发给真实硬件。

对于 MMIO，其访问方式和内存访问无异。完全可以使用内存虚拟技术来解决这个问题。在虚拟 BIOS 产生虚拟 PCI BAR 之后，只需将虚拟的 MMIO 地址空间映射到设备真实的 MMIO 地址空间上，当客户机通过虚拟的 MMIO 地址空间访问设备时，内存虚拟机制会处理一切。举个例子，如果当前 VMM 使用 EPT，则客户机在第一次访问虚拟 MMIO 地

址空间时会陷入到 VMM。此时，可以修改 EPT 页表建立起虚拟 MMIO 地址空间到设备真实 MMIO 地址空间的映射，则在以后的访问中，客户机对该虚拟 MMIO 地址空间的访问就不会再陷入到 VMM。

除了解决客户机直接访问设备 I/O 地址空间的问题外，转换表还可以满足客户机修改设备 PCI BAR 的情况。此时，只需要修改虚拟的 PCI BAR 并维护修改后的值到真实 PCI BAR 的映射即可。至于如何截获修改 PCI BAR 的操作，请参照 2.6 节中提到的两个 I/O 端口 0xCF8～0xCFF。只需要截获客户机对这两个端口的操作即可。

5.7.4 设备发现

前面的内容解决了客户机直接访问真实设备的问题，但如何让客户机中的操作系统发现真实的设备呢？

在上一节也提到，VMM 通常会同时使用多种 I/O 虚拟化技术，其中一项必然会虚拟 PCI 总线（一般来说，这是“设备模拟器”的工作），所以只需将真实设备“挂接”到这条虚拟的 PCI 总线上，客户操作系统枚举 PCI 设备时必然会发现分配它。从第 2 章关于 PCI 设备的介绍中可以知道，PCI 设备暴露给操作系统的接口是 PCI 配置空间，一个很自然的想法是将真实设备的 PCI 配置空间暴露给客户操作系统。前面也提到，对于 PCI 配置空间中的 PCI BAR 通常使用的是虚拟 BIOS 生成的。那么更进一步，可以为设备生成整个虚拟的 PCI 配置空间。

为了让客户操作系统正确地识别分配给它的设备，这个虚拟的 PCI 配置空间中，表示设备标识的前 16 个字节（见 2.6 节关于 PCI）需要使用真实的信息，这是没有关系的，这些信息不会被客户操作系统修改，也不会引起冲突。将生成的 PCI 配置空间以一个虚拟设备的形式挂接在虚拟 PCI 总线上，当客户操作系统枚举总线时即可发现该设备并加载正确的驱动程序。

5.7.5 配置 DMA 重映射数据结构

在上一节介绍 VT-d 技术时，已经介绍 DMA 重映射进行地址转换的过程。对于 VMM 的实现者来说，使用该技术的关键是为所分配的设备正确设置根条目和上下文条目，以及建立 I/O 页表。每个客户机都有一张 I/O 页表，通常在客户机创建初期根据客户机的内存大小、VT-d 硬件支持的页表级数、页大小创建。下面用一个例子说明如何配置设备对应的根条目、上下文条目。

例如，假设 Guest ID 为 1，I/O 页表已经创建好位于地址 A，根条目表和上下文条目表已经在 VMM 加载初期创建好，要分配设备的 BDF 为{00：03：00}。

（1）找到设备对应的 DHRD 结构。

（2）获得该结构的根条目表，通过 BDF 的 bus 字段获得设备对应根条目（以下称 Root Entry0）。

(3) 通过 Root Entry0 的 CTP 字段获得上下文条目表,用 BDF 的 dev: func 字段索引该表,获得设备对应的上下文条目。如果该上下文条目不存在,分配一个并将地址填入 Root Entry0 的 CTP 字段。

(4) 将 I/O 页表的地址 A 填入上下文条目的 ASR 字段,在 DID 字段中填入 Guest ID 1,在 p 字段中填入 1。

(5) 刷新上下文条目的缓存。

上述步骤中,只介绍了主要字段的配置,其余字段也要根据格式正确配置。在刷新操作之后,该设备的 DMA 请求就会被 DMA 重映射硬件截获并进行地址转换。

5.7.6 设备中断虚拟化

DMA 最后一个步骤往往是设备用中断报告驱动程序操作完成。在设备直接分配给客户机,以及 DMA 重映射到客户机内存的情况下,设备的中断也需要注入给客户机。此部分内容已经在 5.4 节中介绍过了,在此不再累述。

5.7.7 案例分析:网卡的直接分配在 Xen 里面的实现

上面的内容介绍了设备直接分配的种种技术,下面以网卡为例,介绍如何应用这些技术将一个物理网卡直接分配给客户机。为了方便起见,这里以 Xen3.2 为例,当然,其他 VMM 原理也是类似的。

首先,VMM 需要知道要直接分配的设备的标识,这里通常使用 PCI 设备的 BDF 号(Bus/Device/Function)来标识设备。设备标识是由用户指定的,因为是用户决定哪些设备被直接分配。用户可以通过 lspci 等工具获得 PCI 设备的 BDF 号,然后通过指定的方式告诉 VMM。这里假设网卡的 BDF 是 03: 00.1。

知道设备的标识后,下一步就是隐藏该设备。如上所述,隐藏设备的方法有很多,Xen 的方法是在 GRUB 启动选项中指定设备的 BDF,然后在启动过程中不去使用这些指定的设备。以网卡为例,用户可以增加 Dom 0 的启动选项 pciback. hide= (03: 00.0),这样,Dom 0 就不会为网卡(03: 00.0)加载驱动,该网卡在系统启动之后就处在未使用状态。

设备被隐藏之后就可以被直接分配给客户机。同样的,用户需要在客户机的配置文件中指定该设备的 BDF 号,告诉 VMM 要将该设备直接分配。在 Xen 里面,用户可以在 HVM 的配置文件中增加 pci = ['03: 00.1']。

Xen 获得直接分配的请求之后,首先需要将该设备呈现给客户机,这是通过在设备模型中虚拟一个 PCI 网卡来实现的。这个虚拟 PCI 网卡可以看作是该物理网卡在客户机中的代言人,客户机操作系统对该虚拟 PCI 网卡的操作请求都会被转交到物理网卡上来,物理网卡处理完后的结果又会通过虚拟 PCI 网卡返回给客户机操作系统。

Xen 负责虚拟 PCI 网卡和物理网卡之间的交互。具体来说,Xen 主要需要做如下的工作。首先,由于虚拟 PCI 网卡的 BDF 和物理网卡的 BDF 不一定相同,因此 Xen 负责进行两

者的转换。其次，虚拟 PCI 网卡的 I/O 空间（MMIO 和 I/O Port）和物理网卡的不一定相同，因此 Xen 也需要为两者建立映射。再次，虚拟网卡的中断和物理网卡也是不一样的，因此 Xen 负责进行中断的转发。最后，Xen 需要为物理网卡设置 VT-d 页表，确保客户机操作系统发出 DMA 请求时，硬件能够正确地处理 DMA 地址。这些工作的具体实现前面章节已经详细阐述过了，这里不再赘述。

5.7.8 进阶

在一个客户机被销毁后，它所拥有的真实硬件设备应该能够被重新利用，分配给其他客户机使用，这就要求设备能够进行再分配。通过前面的内容介绍，读者应该很快能够想到再分配的方法，只需要重新为设备生成新的 PCI 配置空间，建立映射，并挂接到新客户机的虚拟 PCI 总线上，并修改设备对应的上下文条目，就可以将设备分配给新的客户机使用了。其中涉及的步骤前面已经介绍，在此不再累述。

5.8 时间虚拟化

5.8.1 操作系统的时间概念

时间管理是操作系统的一个重要模块。在第 2 章已经介绍了操作系统使用时钟的两种方式，并指出时钟的虚拟化关键是正确的模拟时钟中断。在开始本节内容之前，先来介绍两个操作系统的时间概念。

- 绝对时间（Wall Time）：又称墙上时间。即操作系统启动后到目前为止的总运行时间，它是个单调递增的值。
- 相对时间：指两个时间之间的间隔。例如，两次时钟中断的间隔、两次使用 RDTSC 指令读取 TSC 间的间隔。

正如第 2 章介绍的，硬件定时器如 RTC、PIT 或者 HPET 等都能够以某种频率触发时钟中断，触发的频率可以由软件编程控制。通常，操作系统会将频率设定为一个给定的值（例如 10ms），从而可以知道两次时钟中断之间的时间差（在本例中是 10ms）。同时，硬件定时器也会提供计数器（Counter）的功能，操作系统可以知道两次读取计数器之间的时间差，从而得到相对时间概念。

操作系统在启动的时候会读取 CMOS 的实时时钟，或通过 NPT 协议，得到系统启动时的绝对时间。同时，系统通过维护相对时间，可以知道系统总共运行的时间，从而操作系统可以得到任意时刻点的绝对时间，如下面的公式所示：

当前绝对时间＝系统启动时的时间＋系统启动后运行的时间

图 5-22 描述了操作系统的时钟概念。系统在时间 $t0$ 的时候启动，当实际时间到达 $t1$ 的时候，系统内部维护的运行时间为 $t1-t0$，而系统内部的绝对时间为 $t0+(t1-t0)$，从而

保证了内部时间与实际时间的一致性。每次时钟中断发生时(*t*1、*t*2、*t*3 等),操作系统都会更新内部的时间概念。而在 *t*1～*t*2 时间内,操作系统可以通过读时间设备的计数器得到相对时间。

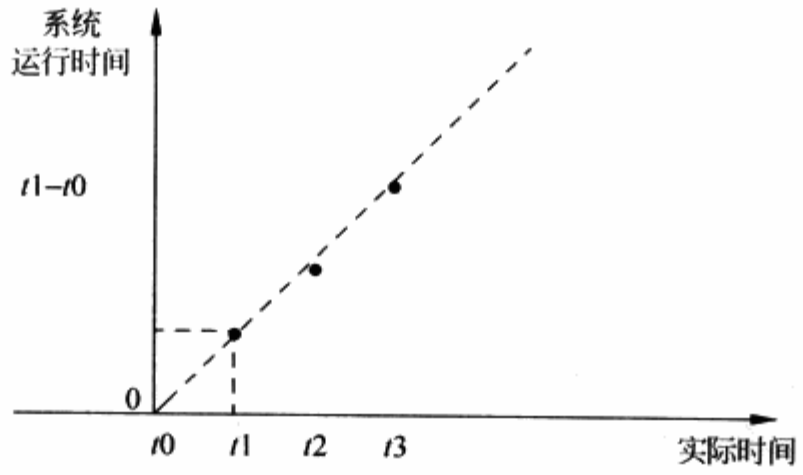

图 5-22　操作系统的时间概念

5.8.2　客户机的时间概念

在硬件辅助的虚拟环境下,客户机操作系统仍然需要维护正确的时间概念,包括相对时间和绝对时间。这意味着 VMM 需要为客户机提供系统硬件时钟设备的仿真,包括 PIT、HPET 和 TSC 等。下面将讨论 VMM 如何虚拟化这些时间设备。

在虚拟环境下,由于客户机是和其他的客户机以及 VMM 共享物理平台,客户机只能得到部分的处理器时间(即使当前只有一个客户机,VMM 本身运行也需要占用 CPU 时间)。在这种情况下,“正确的时间概念”是随着应用的不同有着不同的含义,如何维护正确的时间概念存在着诸多的问题。

图 5-23 和图 5-24 给定了不同的客户机时间概念的实现。假定客户机在 *t*1 时刻处于运行状态,*t*2 的时候被调度进入睡眠状态,*t*3 的时候重新被调度执行,直到 *t*5 的时候被再次调度出去。如果客户机内的某一个程序(操作系统内核或者应用程序)希望得到 *t*4 和 *t*1 之间的相对时间,那么返回值应该是多少呢?

显然,对于不同的情况,返回值应该是不相同的,考虑下面两种应用。

(1) 进程记账:主要用于统计某一个进程的执行时间。假定客户机内的某个进程在 t1 被调度执行,在 *t*4 的时候被调度出去(请注意不要与客户机本身的调度混淆)。显然,进程记账程序希望得到的时间是进程真正运行的时间,也就是(*t*4－*t*3)＋(*t*2－*t*1)。在这种情况下,客户机的时间和实际时间的关系如图 5-23 所示。

(2) 网络速度检测程序:网络速度检测程序通过向远端服务器发送数据包,远端服务器收到数据包以后,会发送一个应答包回来,通过计算发送数据包和收到应答包的时间,就可以大概了解网络速度。假定程序在 *t*1 时向远程的服务器发送数据包,并在 *t*4 的时刻得到服务器的反馈。显然,网络速度检测程序希望得到的是真正的时间,也就是 *t*4－*t*1。在这

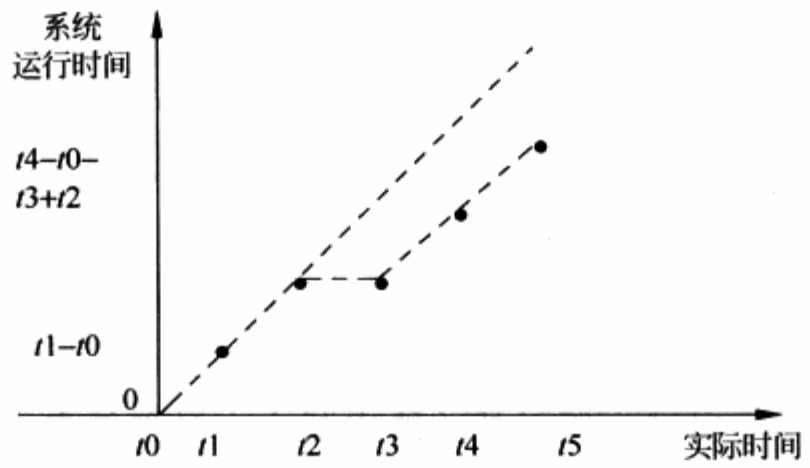

图 5-23　客户机的时间概念(客户机时间！＝实际时间)

种情况下,客户机的时间和实际时间的关系如图 5-24 所示。

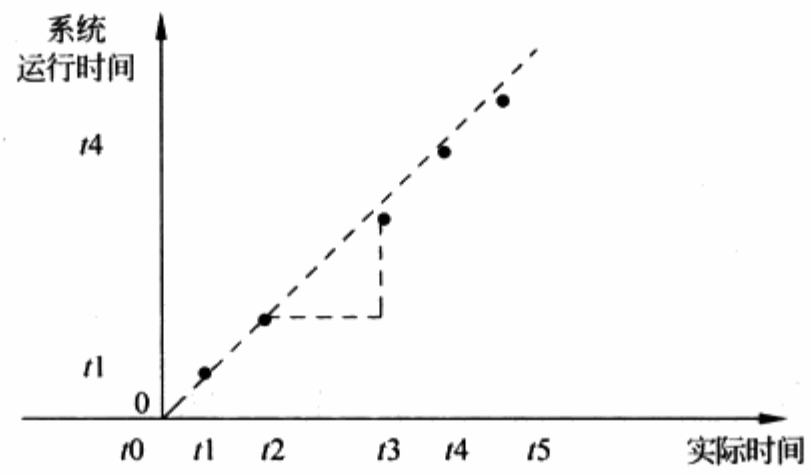

图 5-24　客户机时间概念(客户机时间＝＝实际时间)

在实际需求中,大多数的应用都是像网络速度检测这样的应用,因此,通常时间虚拟化的策略都是给客户机呈现与实际时间相同的时间概念。后面的讨论均是基于客户机时间与实际时间相等的情况。对于需要客户机时间与实际时间不同的情况,会在最后进行简单的介绍。

如前所述,操作系统通过系统中的时钟设备(包括 PIT、HPET 和 TSC 等)得到自己的绝对时间或相对时间,因此首先介绍时钟设备虚拟化的实现方法,然后再讨论如何给客户机提供与实际时间相等的时间。

5.8.3　时钟设备仿真

x86 系统中的时间设备包括 PIT、HPET、ACPI PM Timer 和 TSC 等。本节介绍客户机不会被调度出去的情况,PIT 设备如何虚拟化。然后会简单介绍 HPET、ACPI Timer 等其他时间设备的虚拟化。客户机被调度出去的情况在下一节介绍。

第 2 章介绍了 PIT 的基本概念。其功能主要是为操作系统提供定时的时钟中断和时钟计数器。操作系统通过对 PIT 设备的 I/O 端口读写,设定时钟中断的触发频率,设置和读取时钟计数器。

为了实现时间设备的虚拟化，VMM 必须要提供软件定时器机制，使得程序可以设定在某个未来的时间执行一段代码(参见第 2 章关于操作系统定时器的说明)，同时还需要提供接口，使得程序可以了解当前的实际时间。

假定客户机操作系统设定 PIT 时钟中断频率为 10ms，VMM 截获这一设定(如何截获的方法请参考前面对 I/O 设备虚拟化的讨论)，并通知 PIT 设备模型。PIT 设备模型会向 VMM 注册一个间隔为 10ms 的软件定时器，并提供回调函数，这个回调函数的功能就是向客户机注入一个时钟中断。如果客户机不被调度出去，每隔 10ms，VMM 都会调用这个回调函数，向客户机注入一个时钟中断。具体的中断注入过程请参考前面中断虚拟化相关章节。

当客户机读取 PIT 的 Counter 寄存器时，PIT 设备模型通过 VMM 了解当前的实际时间，并减去 PIT 的时间计数器被初始化时的实际时间，以得到这之间所流逝的时间，经过 PIT 频率的转换后返回给客户机。

对于 HPET 和 ACPI PM Timer，其基本的实现方法相同，不同点在于，客户机读取 PIT 设备是通过 IO 实现，而对于 HPET 和 ACPI PM Timer 是通过 MMIO 截获实现的。同时，各个时间设备的中断号并不相同。

由于操作系统可以依赖于多个时钟设备实现内部时间的维护，因此，当 VMM 提供多个时钟设备的仿真时，需要保证各个设备模型之间的时间一致性。

5.8.4 实现客户机时间概念的一种方法

下面讨论当客户机被调度出去的情况下，如何通过设备仿真实现客户机的时间概念，以使得客户机时间等于实际的时间。如图 5-25 所示，假定客户机在 $t2$ 的时候被调度出去，并在 $t5$ 的时候被调度进来。在这个过程中，根据客户机操作系统对虚拟 PIT 的设置，在 $t3$ 和 $t4$ 的时候需要插入时钟中断，以使得客户机操作系统能够维持内部的时间计数。然而，在 $t3$ 和 $t4$ 的时候客户机并没有运行，因此，VMM 没有机会把中断注入给客户机。

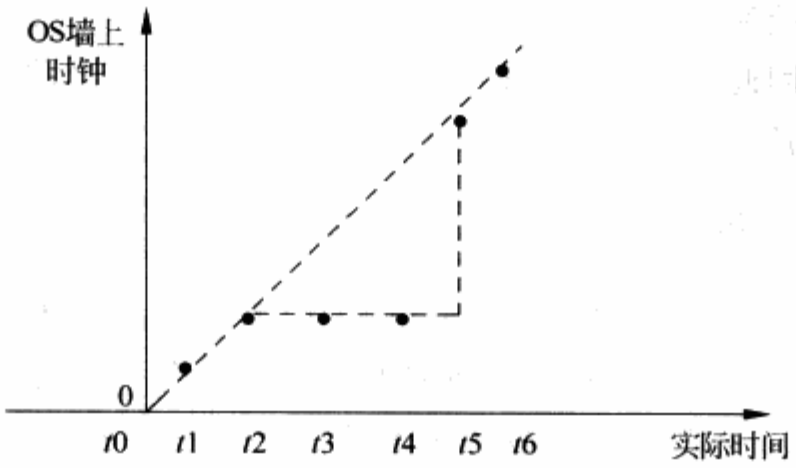

图 5-25 客户机被调度情况下的时钟实现

通常的做法是，当客户机在 $t5$ 时刻被调度回来以后，VMM 连续把 $t3$、$t4$ 时刻丢掉的两个时钟中断连续地注入给客户机。“连续”的意思就是，当客户机处理完 $t3$ 的时钟中断后，立刻把 $t4$ 时刻的时钟中断注入给客户机，在客户操作系统看来就是在一个时钟中断处理完后另一个时钟中断紧接着就发生了。由于这个过程中没有其他的程序被运行，因此，当应用程序或者内核中需要时间服务的程序运行的时候，$t3$ 和 $t4$ 丢失的时钟中断都已经补偿给客户机了。这样，VMM 保证了客户机内部的时间与实际时间一致。当然，如果在操作系统的时钟中断处理函数中，有需要时钟服务的代码运行，那么这些代码仍然会得到不正确的时间。但是，操作系统为了保证中断的快速反应，通常并不会出现这种情况。

图 5-26 给出了实现这一过程的微观示意。当客户机在 $t5$ 时刻被调度运行时，VMM 立刻注入 $t3'$时刻的时钟中断给客户机，使得客户机的时钟概念跳变到了 $t3'$，客户机在 $t5'$的时候执行完 $t3'$中断的中断处理函数后，VMM 立刻注入 $t4'$时刻的时钟中断，使得客户机的时钟跳变到了 $t4'$。由于客户机的时钟处理函数执行速度都很快，因此 $t5'-t5$ 远远小于 $t6-t5$。

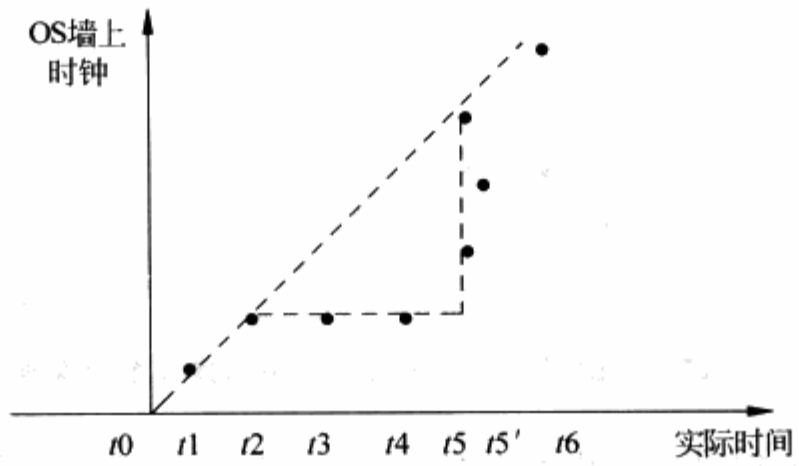

图 5-26　客户被调度出去的情况下中断注入的微观示意图

下面考虑这种情况下，设备模型中计数器的实现方法。在图 5-26 中，在 $t5'$后，由于所有的中断都已经被注入到客户机内，客户机的时钟已经调整到和实际时间一致，设备模型中计数器的值与实际时间也是一致的。而在 $t5$ 到 $t5'$的过程中，由于客户机的时钟概念仍然停留在 $t3$ 的时刻，因此计数器的返回值也应该在 $t3$ 和 $t4$ 之间。由于 $t5$ 到 $t5'$的时间非常短，因此，具体在 $t3$ 和 $t4$ 之间的那个点依赖于具体的实现。

5.8.5　实现客户机时间概念的另一种方法

上面描述了实现时间虚拟化的一种常见的方法。但是，这种实现方法也存在着一些问题。例如，由于需要把客户机被调度出去时的时钟中断补偿回客户机，因此它对系统的性能会有一定的影响，特别是在运行的客户机数量比较多的情况下，时钟中断的补偿会消耗很多的时间。其次，在 SMP 的情况下，各个 VCPU 之间 tsc 的同步存在着一些问题。

正如前面提到的，时钟虚拟化的主要目的是保证客户机内部的时间概念的正确性，因此，可以针对操作系统特定的时间概念维护机制，修改 VMM 时间虚拟化的方法。本节给出一个其他的实现方法。

在硬件平台上，虽然时钟中断丢失的情况非常少见，但还是存在这种可能性，通常的例子是因为操作系统关闭中断的时间过长。然而，有些操作系统已经考虑了时钟中断丢失的情况，它们在收到时钟中断后，会读取时钟设备中的计数器，并根据计数器的值进行修正。如图 5-27 所示，假定操作系统没有收到 $t3$ 时刻的时钟中断，当 $t4$ 时刻时钟中断注入的时候（注意，这是在物理机器上，因此 $t3$ 时刻的中断并没有机会补偿给操作系统），中断处理函数会读取时钟设备中的计数器，发现和上一次时钟中断的发生已经超过了 10ms，因此操作系统会认为自己错过了时钟中断，并根据时钟设备中的计数器来修改自己的时钟概念。

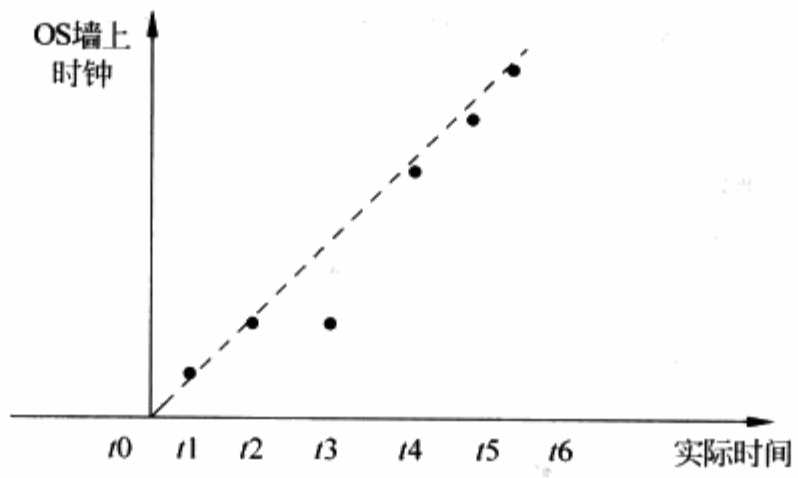

图 5-27　考虑时钟中断丢失情况下的操作系统实现

针对这种操作系统，时钟虚拟化不再需要将错过的中断补偿给客户机。仍然以图 5-25 为例子进行说明。在 $t5$ 时刻客户机被重新调度运行的时候，VMM 并不需要将 $t3/t4$ 时刻的时钟中断都注入客户机，而只需要注入一次。当客户机的时钟中断处理函数读取时钟设备的计数器时，直接返回实际时间。客户机操作系统通过检查计数器返回值，就可以知道错过了 $t4/t3$ 的时钟中断，并将自己的时间概念更新到 $t5$。

5.8.6　如何满足客户机时间不等于实际时间的需求

前面讨论了如何通过设备仿真，使得客户机内部的时间概念与实际的时间相等。然而，正如在前面所说，有些应用需要客户机时间等于客户机实际运行的时间，例如进程记账软件。对于这种需求，一种做法是在客户机内部引入 PV 的时间模块，使得客户机被调度出去的时间（也就是图 5-25 中的 $t2 \sim t5$）被计算在 PV 的时间模块，而不是当前进程上。更详细的资料请参考 VMware 的相关文献[21,22]。

5.9 思考题

(1) Intel 处理器引入哪种新的运行模式来支持虚拟机？这个运行模式有哪些特性？

(2) 对于旧的用 2 级页表并运行在 4 级 EPT 的 32 位客户机操作系统，要完成指令 mov [%esp],%eax 需要多少次物理内存访问？如果是用影子页表，又需要多少次物理内存访问？

CHAPTER 6

第6章 类虚拟化技术

除了利用第4章的陷入与模拟技术外，还有一个方法能够在存在虚拟化漏洞的体系结构上实现系统虚拟化。在计算机硬件不能改变的情况下，VMM提供给虚拟机的硬件抽象是可以修改的。这里所说的硬件抽象，是指硬件平台掩去了内部的具体实现，暴露给软件的抽象接口。完全虚拟化的系统需要给虚拟机提供与硬件平台完全一致的硬件抽象，这样操作系统以及上层软件才能够不做修改地在虚拟机中运行。类虚拟化(par-Virtualization)技术的主要思想就是通过修改暴露给虚拟机的硬件抽象以及上层操作系统，使得操作系统与VMM配合工作，避开虚拟化漏洞，从而实现系统虚拟化。修改后的操作系统能够意识到虚拟环境的存在。

虚拟硬件抽象能够被修改成各种形式，因而对应的虚拟机也各种各样。例如，虚拟硬件抽象可以不包括虚拟内存的支持，这样，在虚拟机中只能运行实模式的系统和应用软件。每个类虚拟化的系统都可以有各自的抽象层设计。与完全虚拟化的系统不同，为了实现不同的目的，两个类虚拟化系统的设计可以有很大的区别。

当然，虚拟硬件抽象与实际硬件的差别越大，客户机操作系统和应用程序需要做的修改也越大。为了尽可地保持上层操作系统和应用软件的兼容性，Xen系统提供了一种实现方式。在x86、安腾等体系结构上，只需要对客户机操作系统作少量的改动，无须对应用程序作任何改动，常用的操作系统如Linux、NetBSD和Windows等就能够运行在Xen系统上。

本章后面的内容将以Xen为主线介绍类虚拟化的主要原理。需要说明的一点是，本章内容不是介绍Xen虚拟化系统本身，而是以Xen为例来介绍类虚拟化的原理。6.1节对类虚拟化技术从宏观上作一个概述。在6.2节，将具体介绍Xen的虚拟硬件抽象。Xen的内部机制将在6.3节中展开介绍。在6.4节把前面小节的技术合在一起，介绍Xen的虚拟机是运行的整个流程。

6.1　概述

6.1.1　类虚拟化的由来

类虚拟化技术通过暴露给客户机操作系统一个修改过的硬件抽象，修改部分操作系统的代码，使得操作系统与 VMM 配合，实现系统虚拟化。

“类虚拟化”这个名词的翻译比较有争议。与完全虚拟化对应，Para-Virtualization 在一些文章中被译为“部分虚拟化”或“半虚拟化”。由于 para 前缀在英文中有“类似”和“辅助”的意思，因此也有文章将其译作“泛虚拟化”或“协同虚拟化”。这些翻译都反映了 para 词缀具有多重含义。最直接的类虚拟化技术需要修改现有的操作系统，因而打破了传统的虚拟化三个要求中的第一条，即同质性。类虚拟化的虚拟层暴露给上层操作系统的不是一个现实存在的硬件抽象层，而是一个更改过的形式。因而，上层操作系统需要经过修改才能运行于这个新的虚拟硬件抽象上。从这个意义上看，类虚拟化不是传统意义上的虚拟化。“泛虚拟化”中带有超越传统的更广义的虚拟化的意思，由于这种翻译偏抽象晦涩，本书中将不采用。另一方面，作为一种新的虚拟化形式，类虚拟化技术也同样能支持在一个物理机器上运行多个虚拟机。并且，通过客户机操作系统与底层的 VMM 配合工作，类虚拟化技术将虚拟化的开销进一步降低。从这个意义上看，“协同虚拟化”表达了上下协作工作的一层含义。这些翻译都有其优劣，本书经过权衡，在后面章节将以“类虚拟化”来指代这种独特的虚拟化技术。

类虚拟化技术代表了一个方向，即改变现有的为真机设计的操作系统，使操作系统意识到虚拟环境的存在，更好地配合 VMM，是更适合运行在虚拟环境中的操作系统。这种意识也被称为受启发或点化的(Enlightened)。

类虚拟化的优势大致有如下三点。

(1) 类虚拟化系统能够为降低虚拟化技术带来的性能开销作最大限度的优化，其主要途径包括消减冗余代码、减少地址空间切换和跨特权级切换、减少内存复制等。操作系统与 VMM 都会带有 CPU 调度、内存管理以及外设驱动等代码，其功能逻辑很多是重复的。在完全虚拟化的系统中，一个磁盘读写请求要完整地经过虚拟机操作系统中的磁盘驱动和 VMM 的真实磁盘驱动两层代码逻辑。在类虚拟化系统中，操作系统里冗余的设备驱动就可以被替换成精简的调用服务的钩子(Stub)，从而缩短一个系统服务的代码路径。

(2) 类虚拟化系统在一定程度上消除了虚拟层和上层操作系统的语义鸿沟(Semantic Gap)，使得整个系统的管理更为方便有效。在这里，语义鸿沟指客户机操作系统内部的运行状态不能够被 VMM 所获得，从而无法最优地调配资源。例如，操作系统需要使用的内存总量是动态变化的，传统上一个虚拟机启动时会被分配到固定大小的内存，VMM 并不知道一个虚拟机在某一时刻具体需要多少内存。而如果这一语义鸿沟能够被打破的话，

VMM 能够在多个或繁忙或空闲的虚拟机之间平衡内存的使用，提高内存使用的效率。

（3）类虚拟化技术还能够用于其他探索和应用。由于能够修改操作系统，类虚拟化方法能够尝试在不同抽象高度提供硬件抽象，甚至提供语义更强大的硬件抽象接口，来优化性能，提供新的功能。

当然，类虚拟化的一个很大的问题是需要对操作系统进行修改。第一，类虚拟化增加了操作系统开发与调试的工作量。每一种操作系统都需要移植后才能运行在类虚拟化 VMM 上。对于同一个操作系统的不同版本，也需要对类虚拟化的支持代码进行相应的修改或维护。第二，对于非开源的操作系统，其修改和维护会有较大的障碍。另外，对于已经停止开发维护的经典操作系统，在其中加入类虚拟化的支持也比较困难。

对于第一个问题，一个缓解的方法是将类虚拟化的支持代码划分出来，封装成一个方法集合，操作系统中的其他代码通过接口来调用这些方法。这样，在操作系统版本衍化的过程中，类虚拟化的支持代码和其他部分的代码能够各自开发，从而最大程度地减少维护工作。

在 x86 等平台的硬件虚拟化辅助技术日趋成熟的情况下，系统虚拟化的实现可以大大地简化，但类虚拟化技术依然有其存在的意义。首先，硬件的支持无法解决所有虚拟化的问题，例如语义鸿沟和冗余代码是不能通过硬件支持来解决的。其次，类虚拟化为虚拟化系统的深度优化提供了空间。

6.1.2 类虚拟化的系统实现

类虚拟化技术修改了硬件抽象，操作系统也需要作相应的修改。然而，如何修改硬件抽象和操作系统并没有一个统一的标准。事实上，不同的类虚拟化的系统为了实现不同的目的，对于硬件抽象层和操作系统的修改都是不同的。最著名的 Denali 系统和 Xen 实现的方式就有很大的差异。Denali 系统的设计目的是在一台物理机器上运行 1000 台或以上的虚拟机，这些虚拟机之间差异很小，能够很大程度上共享资源，并且大多数虚拟机不会同时运行。Xen 的设计目的是在一个 x86 平台上高效运行 100 台以下的虚拟机，并且为了兼容性，客户机操作系统向应用程序提供的应用程序二进制接口（Application Binary Interface，ABI）不变。这样，应用程序不需要修改即能运行在虚拟机操作系统上。Xen 支持的类虚拟化客户机操作系统包括 Linux、NetBSD 等，其对应的移植版本叫做 XenLinux、XenNetBSD。下面分别简要介绍一下 Denali 和 Xen 系统。

1. Denali

Denali 系统在 2002 年由美国华盛顿大学的 Andrew Witaker 等设计实现，旨在将不受信任的应用程序隔离在单独的执行环境中运行的内核。其论文发表在操作系统研究领域最高级的两大会议之一的操作系统设计与实现大会（OSDI）上。

Denali 的虚拟机是轻量级的，其允许 1000 台以上的虚拟机在同一硬件平台上同时运行，以达到高可扩展性（Scalability）。为此，Denali 不像传统虚拟化技术中的 VMM 一样精确、完整地模拟整个真实硬件体系结构，而是为客户机呈现了一个简化了的虚拟硬件抽象。

在 x86 体系结构上运行的操作系统，需要经过移植才能在 Denali 的虚拟硬件体系结构上作为客户机操作系统运行。

Denali 为客户机提供的虚拟指令集是 x86 指令集的一个子集。大多数客户机的指令可以直接在物理 CPU 上运行。与第 4 章提到的用扫描和修补技术或 VMM 模拟执行敏感指令和特权指令不同的是，Denali 不支持模拟执行这些指令。经过修改，操作系统避免了直接使用这些未被 Denali 支持的指令。另一方面，VMM 为客户机操作系统提供了基于虚拟机调度而非阻塞式的 idle 指令，提高了 CPU 资源利用的效率。

由于 Denali 的轻量级虚拟机运行的是小型的应用程序，Denali 中的每个虚拟机只有一个内存地址空间。客户机和 Denali 内核共享这个地址空间，这样避免了虚拟机内应用程序间切换时的 TLB 刷新和虚拟机与 VMM 间切换时的 TLB 刷新。

由于 Denali 上同时运行了许多个虚拟机，当属于某个虚拟机的中断发生时，这个虚拟机很有可能不再运行。这时，Denali 并不立即唤醒这个虚拟机以处理中断，而是将属于这个虚拟机的待处理的中断批量集中起来。等到这个虚拟机下次被轮循调度到时，只需要经过一次 VMM 到虚拟机的切换，客户机就能批量处理所有分发给它的中断。

2. Xen

Xen 类虚拟化系统最早是在 2003 年由英国剑桥大学的系统实验室开发的，其论文发表在操作系统研究领域最高级的两大会议之一的操作系统原理大会(SOSP)上。由于 Xen 系统是开源的，其流传程度十分广泛，对于后来的虚拟化技术的研究有很大的影响。

图 6-1 是 Xen 系统的整体结构图。虚线框围住的部分表示虚拟化层的范围，它是由一个 VMM 和一个特权虚拟机组成。

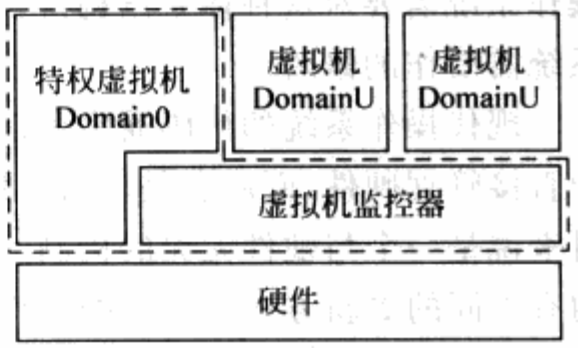

图 6-1　Xen 整体结构

在 Xen 的术语中，每一个系统就是一个域(Domain)。图 6-1 中特权虚拟机被称为 Domain0(或 Dom0)，而其他虚拟机被称为 DomainU(或 DomU)。

由于是类虚拟化，Dom0 和 DomU 的操作系统都是需要移植方能够运行的。

类虚拟化系统的实现形式是可以多种多样的，要进一步介绍类虚拟化技术，需要借助于特定的系统来阐述。目前，x86 体系结构中，Xen 是类虚拟化的代表性系统之一。下面的介绍将主要围绕 Xen 和 XenLinux 展开，但需要强调的是，Xen 只是类虚拟化可能的设计和实现的一种。

6.1.3　类虚拟化接口的标准化

x86 架构上支持类虚拟化 Linux 的系统包括 Xen、lguest、KVM、VMware ESX Server 和微软的 Hyper-V 等。由于每个虚拟化系统对于虚拟硬件抽象的接口各有不同，Linux 中定义了一些标准的类虚拟化接口来覆盖和统一各个类虚拟化解决方案的接口，使得 Linux 操作系统能够在启动时选择使用哪种接口来运行。

VMI(Virtual Machine Interface)是 VMware 公司在 2006 年提出的一种操作系统与 VMM 连接的方法。VMI 的方案中,在操作系统启动时,它会加载一个模块,或称为 ROM。之后,操作系统会调用这个模块中的方法与 VMM 交互。在启动时,操作系统能够根据下层所运行的 VMM 来加载不同的 ROM。这样,同一个操作系统镜像就能够在不同的 VMM 上运行,而不需要重新编译。

由于 VMI ROM 是编译好的二进制模块,它有时不是由内核开发人员编写的。因而,内核开发人员将一部分 ROM 的功能加入了 Linux 内核中,并用了一个统一的接口 Paravirt_ops (pv_ops)来调用。在目前的 x86 平台 Linux 内核,有 Xen、KVM 和 lguest 三个系统虚拟化方案加入了内核开发树。

微软的 Hyper-V 系统中有一层接口转换层,能够兼容 Xen 的类虚拟化接口。

Linux 在启动时能够选择加载真实硬件或者虚拟硬件的 VMI ROM,也可以从各种类虚拟化 Paravirt_ops 集合中选择一个集合。如果直接运行在硬件上,那 Linux 可以选择真实硬件的 ROM 以及 Paravirt_ops 集合。

6.2 类虚拟化体系结构

从操作系统的角度来看,类虚拟化的硬件抽象是一种与 x86 架构有所不同的体系结构。操作系统需要对这种体系结构进行移植。本节将检视一下类虚拟化的硬件抽象,以及操作系统需要作的修改。

现代操作系统的代码通常可以分成硬件相关和硬件无关两种,前者更靠近底层硬件,用于直接管理硬件,而后者是比较高层的软件逻辑。拿内存管理模块来说,如何调配内存,相对来说是一个与硬件无关的机制,而更新页表映射的操作则与具体硬件相关,不同的体系结构有不同的更新方式。高层的硬件无关代码在不同的体系结构间是可以通用的,因而在不同的体系结构间移植一个操作系统,需要改变的通常只是底层与硬件相关的代码。

为了使得操作系统更具有可移植性,现代操作系统通常将与硬件关系紧密的代码集中起来,封装成为一个硬件抽象层。有了硬件抽象层的封装后,操作系统中与硬件相关的代码就不会散落在其他代码中。

XenLinux 是 Linux 修改了硬件抽象层代码以及加入了一些特殊的设备驱动得到的。其修改的部分主要包括指令集、外部中断、内存空间及内存管理方式、I/O 设备驱动、时钟等。下面将分别加以解释。需要强调的是,本节的虚拟硬件抽象是指提供给一个虚拟机的,在 Xen 系统的构成中,Dom0 处于比较特殊的位置,它能够直接管理一部分 I/O 设备,所以 Xen 提供给 Dom0 的硬件抽象有所不同,在这里不作讨论。

6.2.1 指令集

类虚拟化虚拟机能够使用的指令是实际 CPU 指令的一个子集,它不包括特权指令和

敏感指令。对于敏感指令,更准确地说,只是这些指令对于敏感信息的操作不被支持。Xen提供了与这些指令功能相对应的函数,客户机操作系统能够以类似于系统调用的同步通信方式来调用这些函数。与系统调用相对应,这种从操作系统到VMM的调用通常被称为超调用(Hypercall)。

为了防止客户机操作系统执行特权指令,通常的做法是降低客户机操作系统的特权级。在Xen中,VMM运行在特权级0,而客户机操作系统运行在特权级1。如果客户机操作系统使用的指令超出了VMM支持的指令集,由于特权级保护机制,这些指令的执行会发生陷入。对于敏感指令,操作系统虽然能够执行这些指令,但由于这些指令没有VMM的支持,因而可能返回真实硬件的一些执行结果,这样,操作系统就可能无法正确处理。值得注意的是,虽然这些敏感指令能够执行,但由于敏感指令不包括那些写指令,故其不会影响到整个系统的安全性。换句话说,即使操作系统不遵照Xen的规范来使用敏感指令,这个违法的行为只会影响到那个虚拟机本身,而不会影响到系统其他部分的安全与隔离性。

下面以两个x86构架的具体指令来看一下特权指令是如何由超调用所替代的。HLT指令用于在没有工作时停止CPU运行,进入省电模式,直到被下一个外部中断唤醒。类虚拟化操作系统不会使用这条指令,而是发起一个超调用来告知VMM没有需要运行的工作了。在超调用服务函数中,VMM要做的是将客户机的当前VCPU移出等待队列,使得它不再被调度到硬件CPU上运行。由于VMM上可能同时在运行多个虚拟机,因此一个虚拟机或一个VCPU闲置时,其他虚拟机很可能有正在等候的任务。只有在系统中所有虚拟机都闲置时,VMM才会发起HLT指令,让CPU进入节能状态。

再如,LGDT指令用于给GDTR寄存器装载GDT表。Xen允许客户机操作系统装载自定义的GDT表,客户机操作系统发起一个超调用来装载GDT表。在Xen中的超调用服务函数会验证作为GDT表装载的页的类型是GDT数据页,并验证这些GDT描述符项的完整性。Xen同时在VCPU数据结构中保存指向这些GDT页的指针,当虚拟机切换时,Xen会装载被调度执行的客户机的GDT表到硬件寄存器中。

6.2.2 外部中断

在虚拟化环境下,一个虚拟机将不会直接收到来自硬件的外部中断,而只会收到由VMM注入的虚拟中断[①]。

当VMM向虚拟机注入一个虚拟中断时,一个虚拟机可能正在CPU上运行,也可能在调度队列中处于等待状态。如果是后一种情况,只有在虚拟机下一次被调度运行时才能处理这个中断事件。因而,需要一种异步的通信机制将中断传递给客户机操作系统。

根据来源分类,一个虚拟机可能收到的虚拟外部中断有4种:来自物理外部中断、来自VMM、来自同一个虚拟机其他VCPU以及来自其他虚拟机。

---

① 直接由硬件向特定虚拟机发送中断的技术目前尚处于研发中。

最容易理解的虚拟中断是来自物理外部中断的事件，称作物理 IRQ(pIRQ)。例如，一个来自磁盘 A 的 IRQ，经过 VMM 的注入，在虚拟机中就收到一个 pIRQ。这一类虚拟中断来自真实硬件设备，这也就暗示说这个虚拟机拥有对磁盘 A 的设备驱动，磁盘 A 被直接分配给了这个虚拟机。

来自 VMM 的中断事件称为虚拟 IRQ(vIRQ)。vIRQ 不是从硬件收到的 IRQ，而是由 VMM 生成的。例如虚拟时间中断，每个虚拟机可以注册不同时间间隔的虚拟时间中断，VMM 每个指定的时间间隔为虚拟机注入一个时间中断事件。

如果一个虚拟机有超过一个 VCPU，多个 VCPU 之间就可以通过虚拟 IPI 来同步。虚拟 IPI 可以是从一个 VCPU 到其他 VCPU 的中断群发机制，也可以是一个 VCPU 到另一个 VCPU 的点对点中断事件。

一个虚拟机还能够收到来自一个虚拟机的虚拟中断，称为虚拟机间中断(Inter-Domain Interrupt，IDI)[①]。一个虚拟机注册一个 IDI 中断，再通过某种通信方式将 IDI 号码告知另一个虚拟机，这样后者就能够通过这个 IDI 号向前者发送虚拟中断事件了。在 Xen 系统中，虚拟机间中断常用于 DomU 与 Dom0 交互，完成 I/O 请求，这部分内容将在下面的 I/O 子系统部分介绍。

6.2.3 物理内存空间

在虚拟环境中，虚拟机所拥有的物理内存是非连续的。多个虚拟机共享宿主机的物理内存，因此各个虚拟机所拥有的物理页在物理内存中可能是交错分布的，并且可能会动态地发生变化，例如 VMM 有换页机制或者虚拟机迁移等。

如前面章节所介绍的，在虚拟环境中，除了虚拟地址和物理地址外，还有一层地址映射，即实际在 CPU 中使用的地址，在 Xen 的术语中被称为机器地址。

传统的操作系统不能理解非连续的动态变化的内存空间和三层的地址映射。但类虚拟化操作系统可以通过暴露给操作系统更多的信息来使之理解其所处的虚拟环境。Xen 将物理地址到机器地址的翻译表(Physical-to-Machine 表，P2M 表)暴露给操作系统，这样操作系统就能够直接使用机器地址来更新页表映射了。

另一个物理内存相关的问题是操作系统如何侦测可用物理内存。在非虚拟化环境中，操作系统可以通过调用 BIOS 的服务来获得一张内存分布表。例如，PC 兼容机的 BIOS 能够通过中断请求获得一张 E820 表，其中包含了物理内存空间中 ROM 和 RAM 等的分布。在类虚拟化的虚拟机启动时，操作系统不能获取系统真实的 E820 表，而是从 VMM 中获得属于本虚拟机的内存空间信息。

---

① 特权虚拟机还能够注册接收物理 IRQ，从而能够直接驱动外部设备。这种类型的 IRQ 在这里不加以讨论。

6.2.4 虚拟内存空间

x86 架构的另一个问题是 TLB 是由硬件管理的，软件无法很好地管理 TLB。如果 VMM 与虚拟机处于不同的虚拟内存空间，这样在从 VMM 到虚拟机或者从虚拟机到 VMM 都需要进行地址空间切换，而这样的切换都会刷新整个 TLB，从而带来很大的性能损失。另外，VMM 与虚拟机驻留不同虚拟内存空间还有一个问题，即虚拟机中至少需要有一段代码来完成地址空间切换。

对于这个问题，Xen 采用的方法是将 VMM 和虚拟机置于一个虚拟地址空间，这样就解决了 TLB 刷新的问题。为了使 Xen VMM 能够放入 4GB 的虚拟地址空间，Xen 将操作系统可用的 4GB 的虚地址空间压缩了，将最靠近 4GB 顶端的 64MB 虚地址划出来给 Xen VMM 使用。

6.2.5 内存管理

由于 VMM 和客户操作系统共用同一虚拟地址空间，必须有一种机制来保证 VMM 所占据那部分地址空间不被客户机操作系统所访问。通过设定段描述符中的相关标记位，可以限定访问该段的特权级。在 Xen 中，VMM 运行在特权级 0，而客户操作系统运行在特权级 1。因此，通过对段描述符的适当修改，可以限定只有运行在特权级 0 上才能访问 VMM 所在的虚拟地址空间，从而实现对 VMM 地址空间的保护。

在 Xen 的实现中，VMM 和客户机操作系统内核使用不同的 GDT 和 LDT。内核段被设成特权级 1 可以访问，段的界限被设成 VMM 空间的起始地址以下。在客户操作系统启动时，VMM 会向客户机操作系统提供默认的 GDT，该 GDT 并不在分配给客户机操作系统可写内存范围之内。如果客户操作系统想使用默认 GDT 所能提供的特权级 1 和特权级 3 以外的段，就可以在 GDT 表中分配新的 GDT 以及 LDT，并且向 VMM 进行注册。

x86 架构的分页机制可以被用来作虚拟机之间的内存隔离，只要控制了页表的更新，就能控制整个物理内存在各虚拟机之间的分配。在 VMM 中，需要以某种方式记录每个虚拟机所分配到的物理内存页。一个虚拟机不能使用不属于自己的内存页，即不能在页表中建立非法的映射。

对于客户机操作系统，页表页都是只读的，任何对于页表页的修改都会陷入 VMM。这保证了客户操作系统不能随意修改自己的页表，从而保证了一个虚拟机不能随意访问其他虚拟机的内存，起到了虚拟机之间的内存隔离作用。

客户机操作系统可以通过发起超调用来更新页表项。VMM 会对更新的请求进行验证，任何试图映射所辖内存以外物理页的更新都会被禁止。客户机操作系统不能创建到页表页的可写映射。相对普通的内存访问，超调用是开销比较大的操作，所以一个可能的优化是将多个页表项的更新请求缓存起来，在达到一定数目后用一次超调用进行集中更新。

另外，对于单个页表项的更新，VMM 也可以通过陷入与模拟的方式完成。简单地说，

客户机操作系统直接修改只读的页表页，从而触发一个页保护错误。VMM 在页错误处理函数中模拟执行发生页错误的指令，先验证其更新内容的合法性，通过后 VMM 代为完成页表项的更新。这种方式相对于上一种方式实现难度更大，性能也比较差，不能支持批量的更新请求，但能够减少对于操作系统的修改，不需要操作系统显式地发起超调用来修改页表。

6.2.6 I/O 子系统

在完全虚拟化中，I/O 驱动包括客户机操作系统的硬件驱动、设备模拟层，以及 VMM 中真实的硬件驱动。类虚拟化系统不需要如此冗余的三层架构，I/O 交互能够以高于 I/O 指令级别的抽象来实现。

一个类虚拟化的 I/O 子系统至少需要由三个部分组成：一个 I/O 请求发起机制、一个异步的反馈机制和高效的数据交换机制。

在 Xen 中，客户机操作系统不使用任何现有的硬件设备驱动，而是使用一种前后端(Frontend/Backend)交互的设备驱动来发送 I/O 请求和接收 I/O 反馈。处于客户机操作系统内的一端称为前端设备驱动，处于 Dom0 的一端称为后端设备驱动。Xen 的类虚拟化设备驱动区分设备型号，而使每个设备类型使用一种设备驱动，称为类设备驱动(Classed Device Driver)。例如，最常见的磁盘和网卡，这些设备的驱动是通用型的，即虚拟机能够配备网卡设备，但并不是某个具体型号的网卡，如 RTL8139 百兆以太网卡或者 Intel 千兆网卡。

Xen 的 I/O 请求发起机制是用了一个称为环形缓冲区(Ring Buffer)的机制实现的。环形缓冲区是一个虚拟机与 VMM 间共享的页，用于存放 I/O 请求和 I/O 响应的描述符。需要注意的是，这个缓冲区并不存放实际读写的 I/O 数据，而仅仅是 I/O 请求的描述信息，因而一个页能够存放许多个请求。

环形缓冲区是前端驱动通过一个超调用初始化的。为了发送一个 I/O 请求，前端驱动通过超调用将请求描述符写入环形缓冲。与之相应，当一个 I/O 响应到达时，前端驱动通过超调用从环形缓冲中读取 I/O 响应，进行一些清理工作，通知客户机操作系统 I/O 的完成。

前端驱动与后端驱动间通过事件通道机制进行异步通信。前端驱动初始化事件通道，后端驱动将它与这个事件通道绑定。当发送一个 I/O 请求时，前端驱动通过事件通道通知后端 I/O 请求的到达；当 I/O 响应到达前端时，前端驱动会收到一个事件来告知 I/O 响应已到达，进而从环形缓冲中获取 I/O 响应的描述符。

前端驱动通过授权表机制(Grant Table，在 6.3.6 节具体介绍)将 I/O 数据共享给后端，以实现数据传输。前端驱动分配共享页保存 I/O 数据，通过授权表允许后端映射和访问这个 I/O 数据页。通过页共享，后端驱动被允许通过 DMA 直接读写属于虚拟机的 I/O 数据页，这样就节省了 I/O 数据复制所带来的额外的性能开销。

6.2.7 时间与时钟服务

类虚拟化系统的虚拟时间根据不同的时间服务精度的需求，其设计可以是各种各样的。

Xen 通过时间虚拟化保证客户机中各个任务仍能像在非虚拟化环境中一样得到公平的调度，获得相同比例的 CPU 执行时间资源。即客户机中每个进程所分配到的运行时间资源的比例，不受下层 Xen 调度各个客户机行为的影响（当然，虚拟化所带来的额外性能开销会影响各个进程执行的绝对时间）。

为此，Xen 维护了三种时间：实际时间（Real Time）、虚拟时间（Virtual Time）和墙钟时间（Wall Clock Time）。实际时间以纳秒为单位，从计算机启动即开始计时。虚拟时间是每个客户机实际占用 CPU 资源执行所消耗的时间，当客户机不在 CPU 上运行时，虚拟时间的流逝也相应地暂停，直到客户机下次被调度到继续运行。墙钟时间是为每个客户机单独维护的逻辑上消耗的时间，它与真实的时间流逝同步。当一个客户机不在 CPU 上运行时，这个客户机的挂钟时间依然在流逝，并始终与现实世界时间保持同步。

计算虚拟时间的方法显而易见。当客户机被 VMM 调度运行时，它接收来自 VMM 的时间中断，相应地更新虚拟时间。客户机参考虚拟时间来调度各个进程。客户机无须关心真实世界的时间流逝，而只需关心客户机内部虚拟世界的时间流逝。这样，保证了每个进程得到固定长短的虚拟时间，也就保证了每个进程得到固定比例的真实 CPU 执行时间。

为了计算挂钟时间，VMM 从 CPU 的 TSC 计数器（Time-Stamp Counter）换算出从开机起经过的时间。将这个值与初始系统时间相加，即可得到当前系统挂钟时间。TSC 计数器包含在 x86 芯片中，自开机起随每个 CPU 指令时间片单调增加。根据它和 CPU 主频的相对关系，可以换算出自开机器所经过的时间。当每次客户机被调度运行时，VMM 会用以上方法计算一次这个客户机的墙钟时间，保存在与客户机的共享页中，供客户机读取，作为客户机的系统时间。由于挂钟时间计入了客户机不在运行时流逝的时间，因此虚拟化环境下这个客户机的系统时间和非虚拟环境下的系统时间一样，是和真实世界的时间保持同步流逝的客户机系统时间。

Xen 还为客户机操作系统提供闹钟服务。客户机操作系统能够配置一对闹钟，一个是按实际时间计时，一个是按虚拟时间。客户机操作系统还需要自己维护内部的闹钟队列。

6.3 Xen 的原理与实现

6.3.1 超调用

超调用是从客户机操作系统到 VMM 的系统调用。与系统调用类似，Xen 启用 130 号中断向量端口（十六进制的 82H）作为超调用的中断号。这一个中断向量的 DPL 被设置为 1，类型为中断门。这样，超调用能够由处于特权级 1 的客户机操作系统发起，而不能从用户

态发起。

超调用页在虚拟机启动前被初始化，在虚拟机内核启动时，这个页被影射在客户机操作系统的一个固定的虚拟地址上。一个超调用页被划分成 128 块，每块长度为 32 字节。在初始化时，每一块都会写入一段发起超调用的汇编代码。第 i 块的汇编代码会将编号 i 作为参数来发起超调用。在需要发起超调用时，客户机操作系统将超调用页中每块汇编片段当作一个函数，需要调用第 i 号超调用就调用第 i 块代码片段。

6.3.2 虚拟机与 Xen 的信息共享

在客户机操作系统启动时，Xen 会提供一个启动信息页(start_info)给操作系统，提供启动时需要的多种参数。

在运行时，客户机操作系统与 Xen 共享一个信息页，称为共享信息页(shared_info)。

1. 启动信息页

启动信息页中包括的主要信息有内存页的总数、共享信息页的地址、Xenstore 的地址和事件通道号、控制台信息页的地址和事件通道号、页表基地址、P2M 表地址等。

上述这些信息在启动时是必需的。由于这些信息体积比较大，并且在启动时还没有初始化 IDT 表前就会被用到，所以不能通过超调用来获取。

启动信息页在虚拟机启动时由 Xen 填写，被虚拟机读取，在启动过程结束后不再使用。

2. 共享信息页

出于效率的考虑，Xen 将一些常用的共享信息放在共享信息页上，方便与客户机操作的通信。在共享信息页上的主要信息包括 VCPU 相关信息、事件通道相关数据结构、时间以及一些体系结构相关的信息。

在运行时，共享信息页能够被 Xen 和客户机操作系统修改。其中，一些字段由 Xen 更新，例如当前时间等；一些数据由操作系统更新，例如事件通道的事情屏蔽位(Event Mask Bits)等；一些字段由操作系统和 Xen 一起更新，如事件通道的未决事件位(Event Pending Bits)等。

6.3.3 内存管理

在虚拟环境下，VMM 负责对所有的物理页进行管理，负责为各个客户机分配和回收内存，同时保证对于分页和段相关硬件的安全使用。除了 VMM 自己保留的内存以外，其他物理内存都可以以页为粒度进行分配。当前，大部分操作系统都假设所占据的内存是连续的，但是在虚拟环境下，这一假设就不成立了，VMM 需要向客户机操作系统提供一种连续物理内存的假象。

1. 伪物理内存

现代操作系统都支持保护模式，在该模式下，每个应用程序都有独立的地址空间。在应用程序看来，它们对整个内存享有独占权，而不必关心实际分配给它们的是哪些物理内存。

在虚拟环境下，VMM 需要为客户机操作系统做类似的事情。大部分操作系统默认看到的是一块连续的，从地址 0 开始的物理内存。为了实现这类操作系统的虚拟化，就需要提出伪物理内存的概念，以区别于实际的物理内存，即机器内存。机器内存是指物理主机上所安装的所有内存，包括被 VMM 和虚拟域所使用的以及尚未分配的。可以认为，机器内存是由一系列连续的，从地址 0 开始的 4KB 大小的物理页组成的。对 VMM 和客户机操作系统而言，机器页号是相同的。伪物理内存是相对于每个物理域的抽象，它允许客户机操作系统把自己所分配到的内存看作是一系列从地址 0 开始的连续的物理页，而不必担心实际的物理页号是不是连续的问题。

要实现这一抽象，VMM 需要维护一张全局的可读的 Machine-to-Physical 映射表，记录从机器页到伪物理页的映射。同时，需要向每个虚拟域提供一张 Physical-to-Machine 的映射表，来完成相反的映射。很明显的是，Machine-to-Physical 映射表的大小与物理机上实际安装的 RAM 相关，而 Physical-to-Machine 映射表的大小则与分配给相应域的内存大小有关。客户机操作系统中的体系结构相关代码，就可以利用这两张表实现伪物理内存的抽象。概括来讲，只有客户机操作系统的一部分（如页表的管理）需要知道机器地址和伪物理地址的区别。在大部分情况下，操作系统内核并不需要知道物理页的信息，实际需要时则可以通过查询相关映射表来获取信息。

2. 物理页信息的维护

VMM 对于物理页信息的维护主要包括引用计数、所有者（虚拟域号）和页类型。对每个物理页的所有者和用途进行跟踪，使得 VMM 能够对不同客户机进行安全隔离。

在虚拟环境下，物理页按用途可以分为可读写页、页表页、页目录页和描述符表页。这些页的类型是互斥的。也就是说，一张物理页不可能同时为两种类型。对于每种类型，VMM 都维护了一个独立的引用计数，在对页类型进行修改时，首先要求当前页类型的引用计数为 0。

3. 页分配及回收

通过内存自伸缩调节驱动，客户机操作系统可以放弃或者申请额外的内存。在内存使用量较小时，行为良好的客户机会把大量闲置内存返还给 VMM。如果客户机发现自己使用内存过量，它可能尝试把缓冲区数据换出，并且释放一些内存；反之，如果客户机发现还可以再申请一些内存时，它可能尝试扩大自己的块设备缓冲区。所有的这些操作都必须通过显式调用超调用来完成。

6.3.4 页表虚拟化

VMM 往往支持两种模式的内存虚拟化：直接模式和影子模式。这里主要介绍直接模式下的页表虚拟化。

1. 地址翻译

在直接模式下，客户机操作系统中的页表记录的是从虚拟地址到机器地址的映射（页表

项存放的是机器页号),并被直接装载到 MMU 中运行。所以,该模式下的地址翻译与非虚拟环境下没有大的差别。但是,为了保证安全性和隔离性,VMM 必须保证客户机操作系统只能映射给它分配的那些页,并且保证客户机操作系统不能创建到页表页的可写映射(否则前一条规则就被破坏了)。

2. 页表更新

VMM 为每个物理页维护了类型信息。描述表、页目录和页表都有对应的页类型。在客户机操作系统的页表映射中,这几种类型的页必须被映射成只读的。这保证了客户机操作系统不能随意修改自己的页表,从而保证了一个虚拟域不能随意访问其他域的内存。在不能直接修改页表的情况下,客户机操作系统可以通过调用相关超调用来更新页表。在进行实际更新之前,VMM 会对更新的请求进行验证,任何试图映射所分配内存以外物理页的更新都会被禁止。进行验证时,VMM 所维护的页类型和引用计数就起作用了。例如,客户机操作系统不能创建到页表页的可写映射,因为这要求相关页既为页表页,又同时为可读写页,违反了页类型的互斥原则。相对普通的内存访问,超调用是开销比较大的操作,所以在一次超调用中可以进行多次页表的更新。客户机操作系统可以更新普通页表,也可以更新 Machine-to-Physical 映射表。当然,这些更新都必须首先通过 VMM 的验证。

3. 缺页异常的处理机制

客户操作启动时,通过超调用的方式向 VMM 注册一系列陷阱处理函数,这其中就包括缺页异常的处理函数。同时,VMM 允许客户机操作系统装载自己的 IDT。发生缺页异常时,VMM 会产生一个陷阱,并且调用自己的缺页异常处理函数。和正常情况下处理缺页异常一样,VMM 首先根据 CR2 寄存器内容确定导致发生缺页异常的地址。因为缺页异常处理函数本身也会访问内存,可能导致额外的缺页异常。这种情况一旦发生,就会导致 CR2 寄存器的内容被覆盖,所以 VMM 首先需要把 CR2 寄存器的内容复制到一个安全的地方。缺页异常的地址被确定之后,VMM 进一步确定该地址的范围,如果发现是在 VMM 的地址空间范围之内,就做相应的处理;如果该地址在客户机操作系统的地址范围之内,VMM 就在做完相关准备工作之后把缺页异常传递给客户机操作系统。VMM 做的准备工作主要是在客户机操作系统的栈上(1-特权级的栈)创建发生缺页异常的栈帧,客户机操作系统发生缺页异常时的现场也被做相应恢复。客户操作通过调用自己的缺页异常处理函数来完成对缺页异常的处理。页表的更新就是在此时通过超调用的方式完成的。

可写页表(Writable Page Table)提供一种自动批量更新页表的方式。在发生缺页异常而需要对页表进行更新时,VMM 首先对该页表页的内容做一份备份,再清空上一层页表指向该页表项的存在位,然后把该页表页标记为可读写页,最后再把该缺页异常传递给客户机操作系统处理。由于要修改的页表页已经被修改为可读写页,客户机操作系统可以直接对其进行修改,不需要调用超调用。当下一次地址翻译需要用到该页表页时,VMM 首先根据之前保存的页表备份确定客户机操作系统对该页表页所作的所有修改,并进行验证。在所有的更新都验证通过后,VMM 重新把该页表页挂载到上一层页表上去。为了找到对应该

页表页的上一层页表的表项，VMM 需要对所有的页表页维护一个反向影射。

4. 模拟对客户机页表的修改

在 1.4.4 节中已经提到，VMM 写保护客户机的页表页。因此，客户机操作系统无法直接修改其页表页，客户机操作系统可以通过显式调用超调用来更新页表。并且为了降低每次超调用带来的性能开销，一个超调用可以以批处理的方式处理多个修改页表页的操作。

然而，在应用程序进程的创建和销毁等情况下，页表页被频繁地修改。除了批处理的超调用之外，VMM 还提供了模拟对客户机页表的修改的方式，来高效地处理大批量的修改客户机页表页的操作。

由于 VMM 写保护了客户机的页表页，当客户机操作系统试图以修改普通数据页的指令来写入页表页时，发生缺少写页表页权限的缺页异常。VMM 截获这个异常，代替客户机操作系统进行修改页表的操作。这样既能通过写保护截获对客户机页表的修改，防止客户机操作系统随意修改客户机页表，又在逻辑上模拟了修改页表页的操作。

具体地说，若客户机操作系统试图直接写入被 VMM 写保护的页表页而发生缺页异常，VMM 首先根据 EIP 得到引起异常的修改页表页的二进制指令的地址。VMM 解析这条二进制指令，通过与预先存储的指令类型表比对，解析出指令的类型、操作数的类型和操作数的值，从而分析出需要写入的新的页表项的值。同时，CR2 里保存了发生缺页异常的地址，就是被修改的页表项的地址。

有了需要被修改的页表项的地址和新的页表项的值，VMM 就能代替客户机操作系统，修改被写保护的页表项。根据页表项修改前后的值，VMM 施行和通过显式调用超调用修改页表页一样的相关验证，保证宿主机物理页类型的正确性、不同虚拟机间宿主机物理内存的隔离性等。通过了相关验证后，VMM 建立映射客户机页表页的临时的可写映射，通过这个映射将新的页表项的值写入，然后取消这个临时映射。至此，完成了模拟客户机操作系统修改其页表页的过程。

6.3.5 事件通道

在 Xen 中，事件是包括中断在内的异步消息触发的抽象。Xen 中的中断主要包括硬件中断和虚拟中断。前者是物理硬件触发的中断，例如网络包的到达。后者是 VMM 为客户机创造的，模拟的客户机虚拟硬件设备的中断。例如，对应于客户机特定虚拟时间的时间中断。

在 VMM 和每个客户机之间，有一张双方都可以访问的页用于数据共享。VMM 在这张共享页中维护了一个事件向量。此向量的每一位都对应一个事件。当一个事件发生时，VMM 就将向量中对应这一事件的位置上。基于事件向量，Xen 实现了事件通道(Event Channel)这一抽象，作为 VMM 通知客户机某一事件发生的通道。VMM 可以为一个事件分配和绑定一个特定的事件通道，在事件发生时通过这一通道通知通道另一端的客户机。

中断处理是 Xen 事件通道机制的一种应用。硬件的 IDT 表中转载的中断处理函数通

常是 Xen 的特殊处理函数。当中断发生时，硬件根据 IDT 表调用相应的由 Xen 注册的中断处理函数。Xen 的中断处理函数通过事件通道将中断事件通知给相应的客户机的中断函数处理。对每一个需要处理这个中断的客户机，Xen 的中断处理函数创建一个事件通道，并将这个事件通道在事件向量中对应的未决位(Pending)置上，从而实现了异步通知客户机这个中断的发生。如果需要处理中断的客户机正在执行，Xen 立即打断它的正常控制流以处理中断；否则，Xen 为客户机安排一次调度，客户机被调度执行时，Xen 通过 upcall 进入客户机的控制流，调用客户机相应的中断处理函数正确地处理这个中断。在这里，客户机的中断处理函数通常是客户机启动时通过一个超调用向 Xen 注册的。事件通道的端点通过端口号唯一标示。

除了处理中断，Xen 还利用事件通道进行跨客户机通信。所不同的是，这时触发事件的不是 Xen，而是其中一个客户机；并且这种事件通道是双向的。通道任何一端的客户机都可以通过它通知另一端的客户机事件的发生。

6.3.6 授权表

客户机之间的大数据块传输通过共享内存的方式实现。VMM 为客户机提供了一套共享内存的接口。一个客户机可以申明它所拥有的某些内存页可以被其他客户机共享；另一个客户机可以将这些内存页映射到自己的地址空间中。这样，两个客户机都能够访问这些共享的内存页，从而实现数据交换。

每个虚拟机都有自己的授权表，来控制其他虚拟机访问这个虚拟机所拥有的共享页的权限。授权表的每个表项定义了对当前虚拟机的某一内存页，其他的某个客户机具有哪些访问权限(读、写)。通过 VMM 提供的接口，客户机操作系统初始化它的授权表，在客户机操作系统的内核空间中；客户机操作系统修改授权表，就能定义其他客户机对自己所拥有的内存页的访问权限。

在客户机操作系统修改授权表以创建新的访问权限时，会产生新的授权引用(Grant Reference)。授权引用是用来索引授权表的整数下标。其他的客户机可以通过授权引用指定要访问的当前授权表所在的客户机中的物理页。若客户机通过 VMM 提供的接口试图映射授权引用所指向的其他客户机拥有的某个物理页到自己的地址空间中时，VMM 通过授权引用在拥有共享页的客户机的授权表中找到相应的表项。根据这个表项确认拥有共享页的客户机允许将这个页映射给申请共享的客户机。只有通过了权限检查，这个共享页才能被顺利地映射到申请共享的客户机的地址空间中。

6.3.7 I/O 系统

1. 设备驱动划分

设备驱动用于操作系统内核与硬件设备之间的交互。正如前文提到的，在虚拟化环境中，VMM 为客户机操作系统提供虚构出来的虚拟设备，作为硬件资源的抽象。在完全软件

虚拟化技术中，VMM 用纯软件方式来模拟虚拟设备的行为。操作系统使用本地环境下的设备驱动程序，无须经过修改，就能在虚拟环境下驱动这些虚拟设备。VMM 的设备模型负责接收来自客户机操作系统的请求，调用真正的设备驱动与真实的硬件设备交互，完成 I/O 操作。

模拟设备的方式虽然可以直接使用本地环境下的设备驱动程序，但是不能够满足效率上的需求。客户机操作系统的驱动程序不能直接与硬件设备交互。在类虚拟化技术中，通过允许修改客户机操作系统的驱动程序，使得设备驱动可以通过 VMM 提供的接口直接与硬件设备交互。

Xen 使用了前后端模式实现类虚拟化中的外设虚拟化。Xen 的特权虚拟机 Dom0 负责与真实硬件设备的直接交互，它运行了为 Dom0 扮演代理的后端驱动和直接与真实硬件设备交互的原生设备驱动；后端驱动接收来自 DomU 的请求，调用原生设备驱动访问硬件处理这个请求。每个非特权虚拟机 DomU 中运行的是为 DomU 创建虚拟硬件设备的前端驱动；通过事件通道和环缓冲，它将客户机操作系统的设备访问请求传递给后端驱动，并处理来自后端驱动的反馈。前端驱动和后端驱动之间是一一对应的关系。共享同一个真实硬件的多个 DomU 会拥有独立的前端驱动、后端驱动实例对，维护各自的状态信息。前端驱动和后端驱动间通过事件通道进行异步消息传递；通过基于授权机制的共享内存传递数据；通过环缓冲发出和接收 I/O 请求和反馈的描述符。前端驱动和后端驱动本身由客户机操作系统实现，运行在 DomU 和 Dom0 的客户机操作系统中，所以并不是 VMM 的一部分，只是利用 VMM 提供的机制进行相互间的通信。通常，由前端负责分配用于大块数据传输的共享页并通过授权表允许 Dom0 访问；并在共享页中初始化请求/反馈描述符队列（即下文中的环形队列）和其他私有数据结构；创建事件通道与后端连接。

在默认情况下，只有 Dom0 直接与硬件交互。然而，也可以用专门的外设虚拟机（Driver Domain）驱动某些特定的设备。使用外设虚拟机的好处有两个：容错性和兼容性。驱动程序是操作系统中出错率最高的部分，一旦它们出错，会导致整个操作系统崩溃。将驱动程序放到一个特殊的外设虚拟机中，就可以避免驱动程序出错时操作系统崩溃的情况，只需要重新启动外设虚拟机就能使对应的硬件重新恢复工作。这样就提高了容错性。在兼容性方面，在外设虚拟机中运行的特定设备的本地设备驱动所依赖的客户机操作系统不必与 Dom0 一致，可以为了这种设备按需安装和配置特定的操作系统。不同 DomU 的客户机操作系统内的前端驱动实现可以对应同一种后端驱动。

2. 控制流与数据流

正如前面章节中提到的，在 Xen 中，非特权虚拟机和特权虚拟机之间通过事件通道机制来通知 I/O 请求或反馈的到达。

I/O 请求和反馈本身以描述符的方式，保存在使用生产—消费者模型存取的环形缓冲中。每一对前端驱动和后端驱动之间共享一个消费者生产者队列实例。如图 6-2 所示，非特权虚拟机的前端驱动调用 Xen 提供的接口将 I/O 请求描述符保存在队列中，更新指向请

求队列尾部的请求生产者(Request Producer)指针；与之对应的特权虚拟机或外设虚拟机中的后端驱动读取第一个 I/O 请求，更新指向请求队列头部的请求消费者(Request Consumer)指针。后端驱动调用本地驱动处理这个请求。当 I/O 请求处理完成后，后端驱动将 I/O 反馈写入到队列中对应的原先 I/O 请求的位置，更新指向 I/O 反馈队列头部的反馈生产者(Response Producer)指针。队列中的每个元素实际上是 I/O 请求描述符和反馈描述符的联合，因此可以保存两种描述符中的任意一种。当客户机操作系统在执行时，其中的前端驱动读取消费者生产者队列中的第一个 I/O 反馈，处理它，更新指向 I/O 反馈队列尾部的反馈消费者(Response Consumer)指针。生产消费者队列由前端驱动初始化，通过授权机制共享给后端驱动。I/O 请求、反馈描述符的具体结构与具体的设备驱动种类相关。

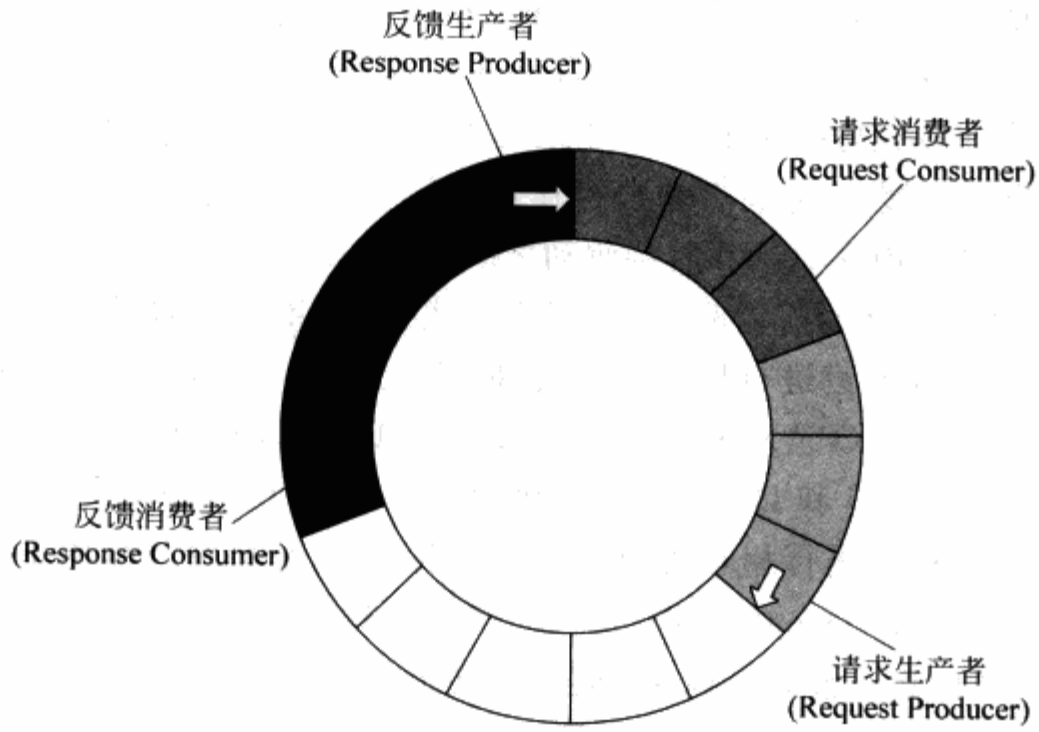

图 6-2　输入输出描述符环

3. 零拷贝技术

在高性能计算中，需要将通信延迟减至最低。通信延迟，除了网络传输延迟外，主要是软件消息处理机制造成的延迟。在传统的操作系统中，消息处理机制包括多次数据复制和变换，造成了时间性能上的开销。例如，在接收网络包时，系统需要将从网卡读取的网络包复制到内核缓存中，再将网络包从内核缓存复制到用户态应用程序的缓存中。这两次复制带来了一定的性能损失。

为了避免不必要的复制，零拷贝技术(Zero-Copy)应运而生。零拷贝技术的设计目的，就是为了避免 I/O 过程中发生任何的数据复制，使得用户空间的数据和外部设备之间能进行直接的数据交互，从而提升性能。

零拷贝技术的实现依赖于 DMA 技术和内存映射技术。下面以网络收包为例加以说明，发包的情况是与之类似的。首先，零拷贝技术运用 DMA 来实现把网络数据包从网卡直接复制到内核缓存中。由于使用了 DMA 技术，这一步完全不需要 CPU 的参与，因此有效

地节省了CPU资源。接着,包含网络包数据的内核缓存被重新映射到了用户空间中。这样,用户应用程序就可以直接访问保存在内核缓存中的网络包数据。这不仅避免了从内核空间向用户空间复制数据,也节省了用户应用程序通过系统调用获取内核缓存数据所带来的运行级别切换的开销。

在虚拟化技术中应用零拷贝技术,面临着新的难题。除了虚拟网卡与客户机内核空间、客户机内核空间与用户空间之间的数据复制需要优化之外,虚拟网卡与物理网卡之间存在的网络包的复制也应该避免。通过使用授权表机制,需要保存网络包数据的虚拟网卡缓存所对应的客户机物理页,通过VMM修改虚实地址映射表的方式,被直接映射到保存物理网卡所接收到的网络包的宿主机物理页上。这样,就避免了从物理网卡缓存的宿主机物理页复制网络包数据到虚拟网卡缓存的宿主机物理页。客户机可以直接、透明地通过虚拟网卡缓存的客户机物理页映射,访问保存了网络包数据的物理网卡缓存的宿主机物理页。

6.3.8 实例分析:块设备虚拟化

原生的虚拟磁盘驱动为上层的操作系统提供了抽象的虚拟设备接口,读写一个磁盘块;将读写请求翻译成符合硬件手册规定的操作下层硬件设备的对应请求,发送给磁盘。Xen的块设备虚拟化(Virtual Block Device,VBD)实现了基于前后端驱动模型的虚拟磁盘。前端驱动为客户机操作系统提供虚拟磁盘设备抽象,后端驱动调用原生磁盘驱动与磁盘交互,相互间利用事件通道、共享内存进行异步通信和数据传输。

在初始化时,前端分配共享页通过授权表供后端共享、初始化基于共享页的环形缓冲和私有数据结构、创建事件通道;与之相对地,后端映射共享页,将自己与前端的事件通道绑定。

在这里,概述VBD处理一个块设备请求的流程。当接收到请求时,前端驱动将请求描述符加入到和Dom0共享环形缓冲中;通过事件通道通知后端驱动请求的到达;将I/O读写数据写入到共享页中,并通过授权表机制允许Dom0来读取这个页。

在这里,请求描述符中包含了操作类型(读或写)、段的数量、读写句柄、ID、第一个数据块相对于磁盘的块号以及一个段数组。段数组中的每个元素都包含了指向I/O数据页的授权引用,以及这个I/O数据段在数据页中的起始和结束位置。值得注意的是,这里逻辑上由连续块组成的段,在磁盘上的实际排列仍然可能是不连续的。物理磁盘的原生驱动本身提供了这一层包含逻辑上连续块的段抽象。

当后端驱动被调度执行(或者正在执行,被VMM打断后),从环形缓冲中读取请求描述符;将包含I/O数据的共享页在DomU的授权下映射到自己的地址空间中。Dom0通过授权机制直接访问DomU的数据页,节省了额外的数据拷贝的开销,使得Dom0可以直接通过DMA读写数据页中的I/O数据到磁盘中,从而提高了性能。

根据请求描述符和I/O数据,后端请求生成一个原生磁盘访问请求,调用原生磁盘驱动访问磁盘。在这里,后端可能合并来自前端的多个访问相邻数据块的请求。原生磁盘驱

动处理完 I/O 请求后，后端驱动生成反馈描述符，保存在环形缓冲中，并通过事件通道通知前端驱动反馈的到达。

前端驱动阻塞式等待，直到收到反馈，完成一些收尾工作，通知客户机操作系统 I/O 完成。为了提高磁盘的吞吐量，VBD 允许乱序处理来自不同虚拟机的 I/O 请求。因此，反馈抵达客户机操作系统的顺序也可能是乱序的。这时，客户机操作系统通过标示请求—反馈配对的唯一标示符(ID)来查找与这个反馈对应的请求。

6.4 XenLinux 的运行

DomainU 虚拟机的创建启动都是由 Dom0 负责。由 Dom0 代替 Xen VMM 进行这项工作可以减少 Xen VMM 的实现复杂度，增加整个系统的健壮性。

那么，首先就需要 Dom0 能够通过一套适当的机制把 DomU 引导起来。尤其是如何能够使 Dom0 读到 DomU 的内核镜像以及 ramdisk 文件，使得在新的虚拟机被创建后，控制流能够顺利地转移到该虚拟机内核入口点。

从 Dom0 文件系统对 DomU 文件系统的访问通常会有很大的困难，而在如今 Xen 的实现中已经能够支持启动加载程序来代劳完成这些工作，它的作用和工作流程就和普通 Linux 启动加载程序工作的机制类似。启动加载程序运行在用户空间，在目前的 Xen 视线中这部分组件通过一个 python 的脚本来完成，它把启动一个新的虚拟机需要的内核以及 ramdisk 文件从它的文件系统中复制出来并存放于 Dom0 的一个临时文件夹中。完成这一步骤后，由虚拟机构建器(Domain Builder)来完成新虚拟机的创建。在 Xen 的实现中，启动加载程序同时会配有一个用来模拟 GRUB 系统的脚本工具，它就如同 Linux 中的 GROB 菜单。进入这个模拟 GRUB 菜单之后，用户可以看到一个或多个启动选项，当用户选择了其中一个条目后，GRUB 系统会找到相应的内核等文件并复制出来交给虚拟机构建器。

对每个虚拟机内存的初始化分配都是在该机创建的时候完成的，各个不同的 DomU 之间的内存分配都是静态完成的，而不是动态调整，这一措施很好地保证了不同虚拟机之间内存空间的隔离，有效地防止了一些非法的跨域的内存访问。而一个虚拟机对内存大小的需求可能会变大，在这个时刻该域可以从 Xen VMM 处请求更多的内存空间。当然，每个虚拟机都有它所能要求的内存空间的最大值，这个值在初始化的时候就会被设定。

至于磁盘设备的分配，对于真实的设备只有 Dom0 才有权访问，Xen 也提供了一套有效的分离设备驱动的机制来保证其他非特权虚拟机对设备的访问。因此，所有的 DomU 将在初始化的时候根据其配置文件由 Dom0 来分配管理属于它的那些虚拟块设备。

同样，根据启动一个新的虚拟机的配置文件，Dom0 也管理分配 DomU 对网络资源的访问。每个分配给 DomU 的虚拟网络接口都会在 Dom0 有对应的网络防火墙路由器(Firewall-Router)，一个虚拟的网络接口就模拟了一个真实的网络适配器。

6.5 思考题

(1) 在 Xen 系统中,假设平台上只运行了一个使用类虚拟化技术的 DomU 客户机。在 DomU 运行 CPU 密集型的应用程序时,其可能的性能损失来自于哪里?

(2) Xen 的 I/O 虚拟化技术中,I/O 环相对于 I/O 指令提高了一个抽象层。I/O 环相对于 I/O 指令级的模拟优势在哪里? 是否在更高的抽象层上能够找到效率更好的 I/O 虚拟化方法?

(3) 在 x86 平台上加入了硬件对虚拟化的支持后,Xen 或其他的类虚拟化技术是否还有存在的必要?

(4) Paravirt_ops 和 VMI 对于 Linux 的开发具有什么帮助与阻碍?

(5) 如果要开发一个通用操作系统专门用于虚拟化环境,现有的类虚拟化能够做哪些深度优化? 操作系统与 VMM 的功能分界线可以做哪些调整?

CHAPTER 7

第 7 章 虚拟环境性能和优化

对于大多数计算机用户来说，应用程序是离他们最近也是最熟悉的软件，应用程序性能直接关系到用户体验。以往，应用程序是面向物理机器开发和优化的，现在，应用程序却可以运行在一个虚拟机上。从一个应用程序的角度来看，整个虚拟环境对它来说是透明的，因此它运行在虚拟机上时逻辑本身不需要也没必要做特别的改变。

由于 VMM 实现方法的多样性，让应用程序面向不同 VMM 进行优化变得格外困难。因此，目前虚拟环境性能优化主要还是集中在 VMM 本身，以便让运行在客户机操作系统上的应用程序具有更好的性能。

到目前为止，读者已经足够了解虚拟化技术的基本原理和实现方法，甚至能搭建并运行一个简单的虚拟环境。现在的问题是：如何衡量该虚拟环境性能的好与坏？如何改进和提高虚拟环境的性能？

接下来，首先从 VMM 的性能评测指标出发，介绍用于 VMM 的性能评测工具；然后再介绍用于在其性能不理想的情况下 VMM 的性能分析工具，借以找到问题所在，并归纳一些用于优化系统的方法；最后，还将介绍 VMM 的可扩展性和影响其可扩展性的因素。

7.1 性能评测指标

通常，评价一个系统的性能好坏可以用吞吐量(Throughput)、延迟(Latency)和资源利用率(Utilization)三个方面来衡量。

所谓吞吐量，是指单位时间内系统的处理能力。处理能力对于不同的评测目标，可能所指的含义不同。例如，在评价一个数据库系统时，所指的吞吐量可以是指单位时间里交易完成的个数；在评价一个网络系统时，吞吐量可以是指单位时间里传输字节数等。

所谓延迟，是指完成一个指定任务所需要花费的时间。例如，在评价一个数据库系统时，可以考察它完成一个查询，或完成一个数据处理所需要的时间；在评价一个网络系统时，可以考察发送一个网络包所需要的时间等。

所谓资源利用率，是指完成一个任务所需要花费的系统资源。例如完成一个数据处理、所占用处理器的时间、占用内存的大小或占用网络带宽大小等。

一般情况下，吞吐量越高、延迟越少、资源利用率越低则表示系统的性能越好；反之，则越差。在某些实际情况下，这三项指标不会同时都朝性能好的方向发展，即该系统既吞吐量高又延迟小又资源利用率低。这时，就要统筹分析考虑，综合评价性能的好坏。例如网络传送，如果系统 A 的网络包传输延迟是 1ms，但是它一次只能传送 512 字节，系统 B 的网络包传输延迟是 2ms，但是它一次能传送 1536 字节，在这种情况下，系统 A 的延迟是 1ms，优于系统 B 的 2ms。但是，系统 B 的吞吐量是每秒 768 千字节，却优于系统 A 的吞吐量每秒 512 千字节。除此之外，如果系统 B 要额外使用处理器资源，如 20%的 CPU 时间，而系统 A 只有 10%，这表示系统 A 在资源利用率方面要好于系统 B。

为了综合评价一个虚拟环境的性能，就要有不同的工具从不同的角度考量它各方面的性能，如处理器、内存、网络和磁盘等。

7.2 性能评测工具

针对不同评测目的，人们积累了一些专门的评测工具，用于测试目标系统的性能。例如，来自于交易处理性能委员会(TPC，http://www.tpc.org)的 TPC-C 就是专门针对联机事务处理的基准测试项目，而来自于标准性能评测公司(SPEC，http://www.spec.org)的 CPU2000/CPU2006、Web2005、JBB2005 和 HPC2002/MPI2006 就分别是专门针对处理器性能、Web 服务器、Java 应用和高性能计算的基准测试项目。

另外，在上节介绍的评测性能指标里，所谓的“一个任务”也不是随意挑选的，在实际的评测工具中，包括了一系列的负载(workload)，用于测试目标系统各方面的性能，借以充分说明问题。例如，TPC-C 就针对订单录入和销售环境事务吞吐量和延迟，包含了若干个负载，专门测试系统在同时执行订货、支付、发货、订单查询和库存查询的性能情况。

针对虚拟环境性能的评测，由于 VMM 作为新生事物，专门针对它的性能评测工具开发尚在早期阶段，目前仅有少数几种工具，如英特尔公司的 vConsolidate 和 VMware 公司的 VMmark(http://www.VMware.com/products/VMmark/)，这些工具主要模拟 VMM 在服务器领域的应用，即为模拟服务器整合应用环境而开发。从另一个角度，人们可以对客户机操作系统的性能进行评测，从而间接地评测 VMM 的性能，显然这不是真正意义上对 VMM 的评测方法，但在一定程度上仍有参考意义，而且这种方法运用起来也十分简便。

7.2.1 重用操作系统的性能评测工具

基于传统操作系统的性能评测方法，即通过在虚拟机上运行传统操作系统的性能评测工具，尽管并不能完全反映 VMM 的性能，但是通常情况下，它在一定程度上还是反映了 VMM 的某方面性能。而且，基于传统操作系统的性能评测工具普遍被大家所熟悉，运用起

来简单方便。因此,综合运用各种传统操作系统上的性能评测工具仍然是衡量 VMM 性能的主要方法。

对 VMM 进行性能评测和优化,包含以下三个方面。

(1) 要评测和优化的是运行在客户机操作系统上的处理器性能指标,常用的有 SPEC 公司的 CPU2000 或 CPU2006 以及 sysbench(http:// sysbench. sourceforge. net)中的 CPU 工具,它们一般包含对处理器整型处理能力和浮点处理能力的评测。除此之外,在实际应用中,cyclesoak(http://www. zip. com. au/~akpm/linux/)因为特别简单方便也被经常使用。

(2) 要评测和优化的是内存虚拟化的性能,对 VMM 的内存虚拟化性能评测通常使用内核编译(Kernel Build)或 sysbench 中的内存评测方法。值得一提的是,内核编译是一个比较综合的性能评测工具,它广为 Linux 开发者和开源的 VMM 开发者采用。这种方法既对处理器敏感也对内存和硬盘读写敏感,因此,在实践中,内核编译既被用来做内存虚拟化的性能评测工具,也被用作硬盘虚拟化的性能评测工具。

(3) 对虚拟外设的性能评测和优化,例如对硬盘虚拟化性能评测工具有 hdparm (http://www. sourceforge. net/projects/hdparm/),对网络设备虚拟化性能评测工具有 netperf(http://www. netperf. org/netperf/)。

正如上节所言,吞吐量是计算单位时间内完成的任务数来衡量,延迟是计算完成一个指定任务所用的时间来衡量,而不管是用哪种性能评测工具,一般都直接利用客户机操作系统提供的时间作为性能评测的衡量指标。因此,如何获取准确的时间就成为了一个关键问题。虚拟环境下的时间虚拟化,尤其是对完全虚拟化下客户机的时间虚拟化本身就存在种种挑战。利用以上这些工具进行性能评测时,必须注意到可能有客户机时间虚拟化不准确而导致的误差。所以,必要时采用物理时间如网络时间或秒表进行比对校验,这一点对于基于 SMP 的客户机性能评测时尤其重要。

另外,由于虚拟环境下的客户机处理器事实上靠分时共享物理处理器资源实现,因此对于评测工具中甚短(例如几个或几十个毫秒)任务所花费时间的测量具有很大的颠簸性,如果在这段甚短任务的执行中客户机处理器被调度出去,它被测量到的延迟就会很大,反之可能很小。如果一个评测工具主要依靠测量甚短任务的吞吐量或延迟来做衡量,这个评测工具对虚拟环境的性能评测具有很大的随机性,因而也是不可靠的。

7.2.2 面向虚拟环境的性能评测工具

由于用传统操作系统上的性能评测方法间接地对 VMM 进行性能评测存在一些问题,开发面向虚拟化方案的性能评测工具对于虚拟化技术的发展至关重要。它能为各种 VMM 开发商提供一个标准的性能评测工具,以方便开发商对他们自己 VMM 产品的优化。今天各大公司和虚拟化方案提供商以及标准性能评测公司都在从事相关的研究和开发,目前来说这方面的工具不是很多,但至少已经有 Intel 的 vConsolidate 和 VMware 的 VMmark 性

能评测工具用来模拟服务器整合环境下的应用，将多种不同种类的负载连同运行这些负载的操作系统，加载在不同的客户机上，通过对不同负载性能的测量来综合评测不同虚拟化方案的性能差异。

vConsolidate 是一个面向虚拟化方案的服务器整合性能评测工具，它的设计原理如图 7-1 所示，不同的负载运行在不同的客户机上，最上面由包装层归纳。今天的 vConsolidate 由 4 个同时运行的独立的性能评测工具组成，它们分别模拟数据库服务器 Sysbench、Web 服务器 WebBench（http://www.pcmag.com/）、Java 应用 JBB 2005 和邮件服务器 LoadSim（http://www.microsoft.com/）。另外 vConsolidate 还同时运行一个空闲客户机，也就是不运行任何负载的客户机，如同传统操作系统上的空闲进程一样。

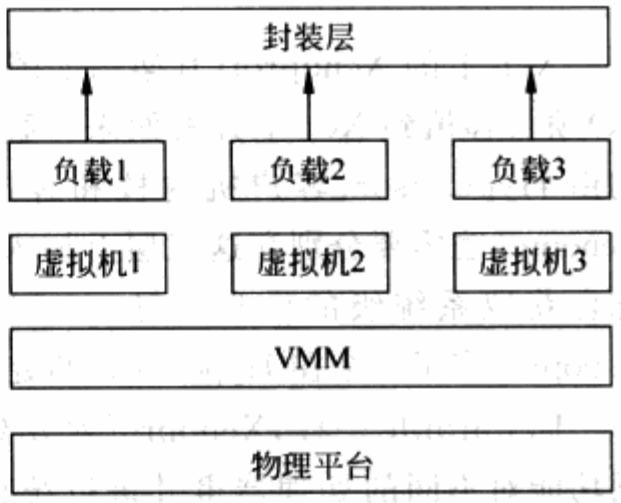

图 7-1　vConsolidate 结构

VMmark 的原理同 vConsolidate 相似，不同之处在于 VMmark 在数据库服务器、Web 服务器、Java 应用和邮件服务器外，还加入了文件服务器。当然，VMmark 也有一个空闲客户机。

7.3　性能分析工具

有了评测工具，就像田径比赛有了秒表和米尺一样，可以用它来测量每一个运动员的成绩。但是，秒表和米尺本身并不能帮助运动员发现影响成绩的因素，一个好的教练还需要借助其他各种仪器（如录像、各种电子仪器等）分析该运动员的各种动作和姿势，进而发现问题，解决问题以提高运动成绩。

对 VMM 性能的调试也是一样，有了各种性能评测工具，也就是有了田径比赛的秒表和米尺，现在还需要一些对性能瓶颈进行分析的工具，以便发现问题，解决问题。今天，基于传统操作系统开发的各种性能分析工具，只要它们在虚拟环境中还可以执行，这些性能分析工具仍然可以应用于对客户机上程序的分析。但是，需要说明的是，这些分析工具是无法发现因为虚拟化而花费在 Hypervisor 中的开销的。

从整个物理系统层面对运行在它上面的 Hypervisor 和客户机一起进行性能分析，这显然要比单一地分析运行在客户机上程序的性能更全面和准确。能对整个系统（客户机和 Hypervisor）进行性能分析的工具由此而生，如 Xen 上的 Xenoprof，KVM 上运行在主机操作系统 Linux 上的 oprofile 等。但是，这些分析工具也有局限性，即都是跟具体的 VMM 绑定的。例如，基于 Xen 的系统性能分析工具 Xenoprof 并不能运行在 KVM 上，而主机操作系统 Linux 上的 oprofile 也不能应用在 Xen 系统性能分析上。下面以 Xen 为例来说明这些

工具的调试原理。

7.3.1 Xenoprof

Xen 上的 Xenoprof(其架构如图 7-3 所示)是将 Linux 上的 oprofile(其架构如图 7-2 所示)项目移植到 Xen 上开发而来。我们知道,Xen 虚拟机的结构由 Hypervisor、客户机内核和客户机应用程序三层组成。Xenoprof 也需要分别在这三层上加入相应的支持,使它们互相配合获取系统级的信息。与 oprofile 相比,最明显的区别是 Xenoprof 多了一个在 Hypervisor 的处理层。

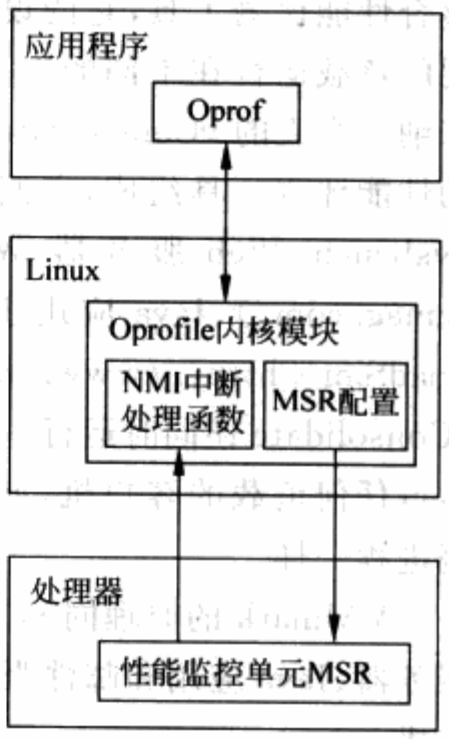

图 7-2 oprofile 结构

同 oprofile 一样,Xenoprof 利用处理器内含的性能监控单元功能对不同的处理器事件进行统计,当这些监控事件的计数达到一定数目时就会触发一个中断。同其他的系统性能分析工具一样,Xenoprof 采用 NMI 中断以避免由于 VMM 本身关中断引起的对监控事件处理的滞后,以及由此导致的对事件计数的不准确性。Hypervisor 捕获 NMI 后可以从被中断处理器的上下文得知原先运行的指令地址(cs: ip)以及执行该指令的当前客户机或 Hypervisor,并通过 Xenoprof 中继获取客户机进程上下文,就可以把来自客户机的 cs: ip 值与程序源函数对应起来,从而实现函数级别的分析。NMI 可以发生在 Hypervisor、客户机内核和客户机应用程序执行的任何时候,对于来自不同层次的 NMI 中断,Xenoprof 的处理也是不一样的。

(1) 对于来自 Hypervisor 的中断,Xenoprof 采样引擎可以直接把中断信息传送给 Dom 0,由位于 Dom 0 内核的 Xenoprof 中继告知同样位于 Dom 0 的应用程序 oprofile 读取和保存。

(2) 对于来自 Dom 1 的中断,Xenoprof 采样引擎直接把中断信息存放到与 Dom 1 共享的内存上,并用异步通知方式让运行在 Dom 1 内核的 Xenoprof 中继传送给运行在 Dom 1 上的 oprofile 应用程序读取和保存。这里,Xenoprof 中继还会额外获取和传送被中断应用程序的进程信息。

这种运行有 Xenoprof 中继和 oprofile 的客户机也叫 Xenoprof 主动客户机,例如 Dom 0 和 Dom 1 就是主动客户机。还有另外一类客户机,在 Xenoprof 采样过程中并不运行 Xenoprof 中继驱动程序,也没有 oprofile 应用程序,因此本身不能获取采样数据而必须依赖 Dom 0 来获取采样数据。这类客户机也被称做 Xenoprof 被动客户机,通常为完全虚拟化的客户机,例如 Dom 2,它的缺点是对于来自客户机用户空间的采样找不到相对应的函数。但对于来自客户机内核的采样数据,如果客户机内核格式是 oprofile 能够识别的 ELF 可执行程序格式,它可以找到 cs: ip 与函数的对应。

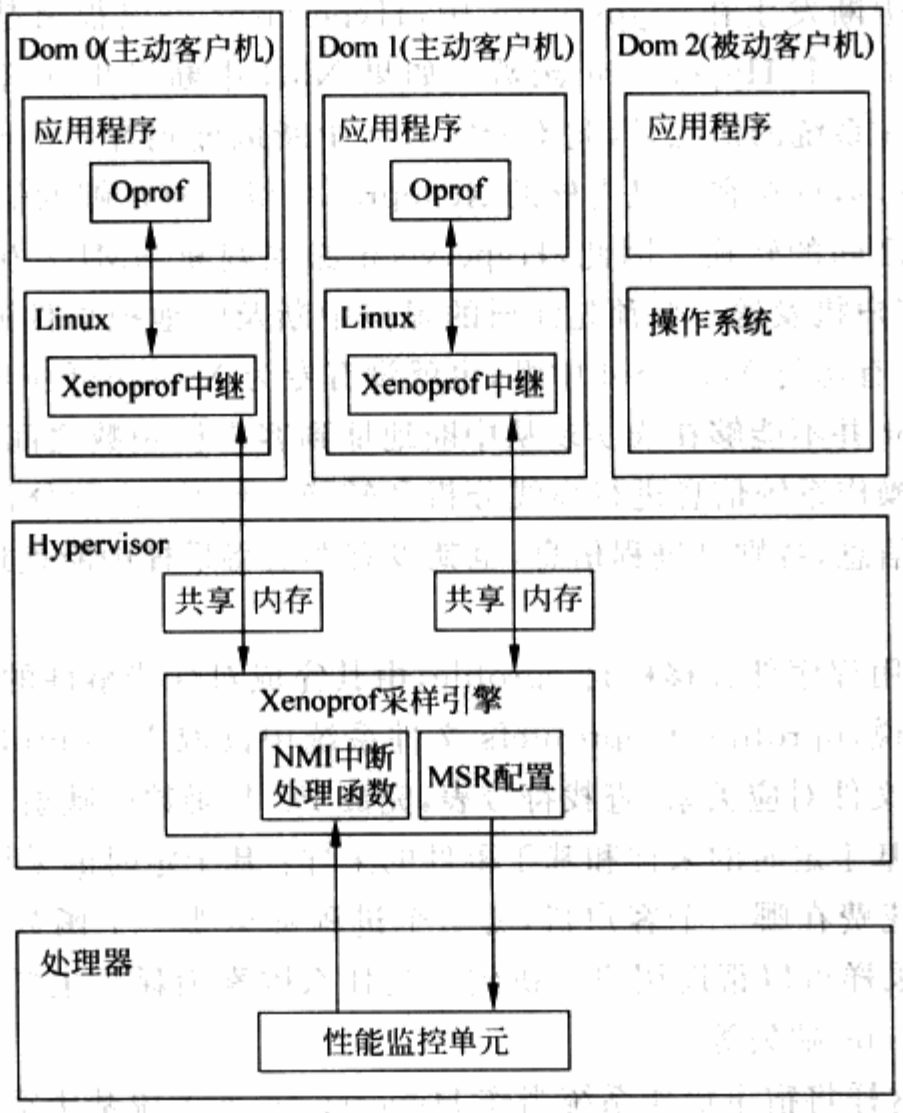

图 7-3 Xenoprof 结构

在图 7-3 中，Xenoprof 包含了三个主要组件：Xenoprof 中继、Xenoprof 采样引擎和应用程序 Oprof。其中，Xenoprof 中继运行在客户机内核中，主要完成以下工作。

(1) 在初始化时，把 Hypervisor 中存放性能事件记录的缓冲区映射到自己的内存空间中，以便将来访问。另外，它分配一个更大的缓冲区，用于存放给 oprofile 应用程序的数据。

(2) 注册虚拟中断 VIRQ 处理函数，该函数完成把 Hypervisor 缓冲区中的 PC 采样与具体映像文件对应，并将对应关系存储在新分配的缓冲区中。

(3) 实现一个基于内存的文件系统 oprofilefs，一方面提供给应用程序一个方便使用的配置性能监控 MSR 的控制接口；另一方面，它又提供了一个读取数据缓冲区的数据通道。

Xenoprof 采样引擎的功能主要包括如下几点。

① 实现对处理器的各种性能监控单元 MSR 的配置，并且通过 Hypercall 接口的方式向来自特权客户机(通常为 Dom0)提供服务。

② 注册 NMI 处理函数，实时处理因为性能监控 MSR 计数器溢出产生的事件。

③ 为每个处理器分配一小块内存，用于临时存放 NMI 处理函数生成的关于该处理器上发生的性能事件的信息。

④ 如果 NMI 中断发生在 Hypervisor 中，Hypervisor 可以很清楚地从发生处的地址 cs：ip 知道是来自哪一个 Hypervisor 函数。如果 NMI 中断发生在客户机上，Hypervisor 并不知道客户机操作系统的信息，因此不能完成从中断地址到客户机函数之间的转换。但是，对于主动客户机，运行在客户机内核的 Xenoprof 中继具有足够的信息，可以完成从中断地址到客户机函数之间的转换。因此，Hypervisor 必须判断 NMI 发生时处理器所在的客户机，并且向这个客户机发送一个预先注册的虚拟中断及时地把事件通知该客户机。对于被动客户机，由于没有运行 Xenoprof 中继，也就没有专为 Xenoprof 中继预先注册的虚拟中断，因此，Hypervisor 并不能够在线实现从中断地址到客户机函数之间的转换，只能在事后利用有限的客户机操作系统信息进行离线分析和转换。但是，这种分析和转换已经丢失了客户机运行现场的信息，特别是进程信息，也就没有办法将采样中断地址准确对应到客户机应用程序地址。

Xenoprof 的应用程序部分移植自 oprofile，由其完成对性能事件的采样分析，只做了少量的扩展。具体来说，oprofile 从 oprofilefs 文件系统中读取 Xenoprof 内核模块的缓存内容，通过 PC 与映像文件对应关系，查找符号表，完成 PC 与函数的映射。

Xenoprof 支持基于定时的采样和基于事件的采样：基于定时的采样可以帮助开发者发现处理器时间主要花费在哪一个客户机，哪一个进程甚至哪一个函数，尤其是 Hypervisor 函数；基于事件的采样可以帮助用户分析到底是什么因素消耗了主要的处理器资源，例如是 TLB 缺失还是 cache 缺失等。

对于像 KVM 这样将宿主操作系统当作 Hypervisor，或者说基于宿主的 VMM 来说，宿主操作系统上的性能分析工具如 oprofile 可以直接用作系统性能分析工具。当然，也可以像 Xenoprof 一样将这些工具扩充使其更好地获取客户机信息。

7.3.2 Xentrace

另外一类系统分析工具通过软件预先设定的探针，来记录 VMM 执行过程中感兴趣的事件，如 Xentrace 和 Kvmtrace。例如，对于一个有 Intel VT-x 支持的完全虚拟机，如果有一个工具可以记录下客户机运行过程中各种VM Exit的发生次数和处理时间，这对分析虚拟机性能将提供非常大的帮助。Xentrace 就是这样一个在 Xen 上的工具，其结构如图 7-4 所示。

Xentrace 本身设计得很通用，通过在 Hypervisor 中与性能相关的关键路径上（例如客户机调度、Hypercall 操作）插入探针记录点，当关键路径被执行到的时候就新产生一条记录，包括发生的时间戳及其他详细状态。分析人员可以事先根据自己感兴趣的记录类型设置过滤器，也就是说，只有通过过滤器筛选之后的

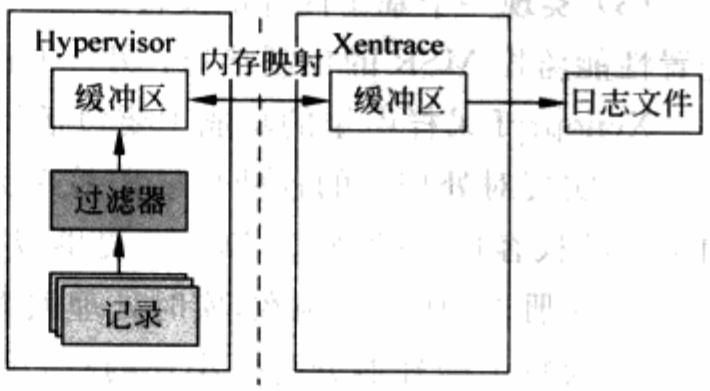

图 7-4 Xentrace 结构

记录才会被写到一个固定的环形缓冲区里面，其余的将被丢弃。该缓冲区通过一个特殊接口同时被映射到 Dom 0 中的 Xentrace 应用程序的进程空间，Xentrace 读出并保存缓冲区中的记录，以供事后统计和分析。

例如，为了统计 VM Exit 的开销，可以在处理 VM Exit 开始和结束的地方分别设置一个记录点，通过这两条记录之间的时间戳差值，可以获得尽可能准确的各种 VM Exit 发生频率和处理开销。

7.3.3　Xentop

我们知道，当某类资源耗尽而其他资源还有很多的时候，该资源就变成了阻碍性能进一步提升的瓶颈。因此，还有一类工具用来统计系统中的资源利用率。

Xentop 就是基于 Xen 的资源利用率统计工具。它运行在 Dom 0 的应用程序空间，使用一套特权 Hypercall 接口来获得当前系统中的各个客户机对不同资源使用情况的历史数据，通过固定间隔的轮循和比较历史数据之间的变化，就可以算出各资源的利用率。

7.4　性能优化方法

不论采用何种虚拟化技术，相对于物理机而言，处理器在 Hypervisor 上执行开销都将构成额外的虚拟化开销。处理器在 Hypervisor 上执行的时间越长，虚拟化开销就越高，虚拟环境的吞吐量和性能也就越差，反之则虚拟环境的性能越好。因此，优化的重点就是应该考虑如何降低这种额外的开销。

虚拟环境性能优化的方法有很多种，而且是具体问题具体分析。下面归纳一些以往在优化过程中常用到的方法，以供读者参考。

7.4.1　降低客户机退出事件发生频率

降低客户机退出事件的频率有助于提高虚拟化的效率，但是客户机退出事件又是实现系统资源共享所必不可少的，如何降低客户机退出事件频率也就成了虚拟化性能优化的一个关键。下面给出一些常见的降低客户机退出事件频率的优化方法。

1. 硬件加速

硬件虚拟化技术本身提供了对某些高频虚拟化事件的硬件加速支持，充分利用这些硬件功能可以降低客户机退出事件频率。如打开影子 TPR/CR 8 寄存器技术可以减少甚至避免客户机 TPR/CR 8 操作导致的客户机退出事件；通过置位客户机异常位图可以直接由硬件将客户机异常注入到客户机 IDT 中，等等。

2. 共享内存

客户机对于虚拟化资源的访问一般需要 VMM 的介入，以便 VMM 以合理的虚拟化策

略共享物理资源。客户机和 VMM 间的共享内存可以使客户机在不采用 Hypercall 服务请求的同时，直接得到由 Hypervisor 预先存放在共享内存中的客户机虚拟化内容。这种方法尤其适合客户机操作系统对虚拟资源的读取，如 VCPU 的状态等，这可以直接降低客户机退出事件发生的频率。

3. 影子页表

对于采用影子页表方法的客户机，如何有效地构建影子页表以降低由影子页表引起的主机上页缺失是一个影响整体虚拟化性能至关重要的因素，也是一个研究的热点。例如，对客户机页表的猜测性预读取和影子页表的预构建，将由于客户机页表缺失引起的异常直接由硬件注入到客户机页缺失处理程序中，等等。当然，采用硬件技术 EPT 可以大大降低客户机退出事件发生的频率。

4. 直接分配 I/O

客户机 I/O 操作引起的客户机退出是实现 I/O 资源共享的基本方法，但是对于有直接分配设备的客户机，则可以直接访问该设备资源，包括 I/O。因此，对于直接分配设备的客户机，可以通过修改该客户机 VCPU 的 VMCS 控制结构中的 io_bitmap，使得这些设备的 I/O 操作引起的客户机退出被完全避免。

5. 批 Hypercall

Hypercall 是类虚拟化操作系统向 Hypervisor 申请服务的基本方法，它本身不可完全避免。在源程序逻辑级别降低 Hypercall 的过度使用是类虚拟化操作系统设计性能好坏的关键。一个常用的降低 Hypercall 使用频率的方法是将多个服务请求合并在一起，通过单一的 Hypercall 向 Hypervisor 申请服务，例如 Xen 中的批处理 Hypercall；对于类虚拟化的设备驱动程序，不管它是用 Hypercall(如 Xen 采用的 PV 设备驱动程序)或者虚拟 I/O 端口访问(如 KVM 和 lguest 采用的 virtio)来向 Hypervisor 请求服务，降低 I/O 或 Hypercall 导致的客户机退出事件频率始终是类虚拟化设备驱动程序设计的准则。通常，这可以结合共享内存通过将多个 I/O 操作合并在一起而向 Hypervisor 发出单一的服务请求的方法实现。Xen 或者 KVM 的 PV 设备驱动程序都依此原理设计。

7.4.2 降低客户机退出事件处理时间

加快对每一个客户机退出事件的处理是另一个提高虚拟化性能的重要方法。但是，如何降低处理时间则对每一个开发者来说都是一个挑战，这同时也依赖开发者系统编程的经验。一些常用的编程技巧在这里也都适用，但同时也还是有一些跟虚拟化相关的通用的方法，如 I/O 的并行处理。

在采用设备模型仿真客户机设备的情况下，如 IDE，一个触发客户机 IDE 进行磁盘读写操作的端口访问可能需要等待很长的时间直至设备模型完成对物理设备的操作(读或写)，这导致该 VCPU 的端口操作响应时间非常慢，有时甚至是几十毫秒。这样的处理方式

在虚拟化情况下会导致很严重的延迟。如果将这种同步处理的方式改为异步方式，就可以避免 VCPU 的等待时间，利用这段时间处理其他操作。具体流程如下：VCPU 发送的端口访问命令送给设备模型后；设备模型启动后台服务进程去响应对该虚拟段口的操作；前端设备模型则直接向 VCPU 返回操作完成的信号；请求完成；之后，设备模型向 VCPU 发出中断请求表示数据准备妥当。如图 7-5 所示。

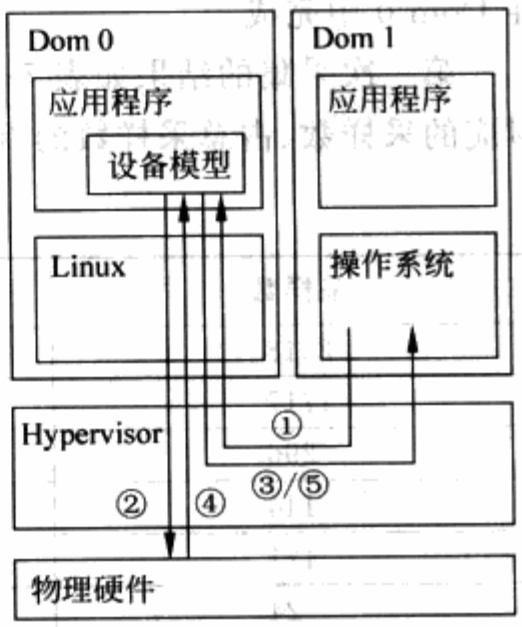

图 7-5 异步方式流程

除了 I/O 并行处理外，还有非主动 FP、非主动宿主机状态保存/恢复等，可以用于降低客户机退出事件处理时间。

7.4.3 降低处理器利用率

除了降低系统延迟外，VMM 还必须尽可能地降低对资源主要是处理器资源的利用。否则，当一定情况下，特别是在高负载情况下，系统的资源如处理器就可能饱和从而限制吞吐量的提高。一个典型的例子是高速网络设备。在 Xen 的类虚拟化网络驱动程序中，提供网络服务的后端程序需要将物理网卡收到的网络包复制到客户机接收缓冲区中，这需要利用大量的处理器资源，并且最终会使得物理处理器饱和而限制 Xen 客户机的网络性能。实验发现，Xen 类虚拟机的网络性能在 10G 网卡/网络上只能达到大约 2～3Gbps，因为此时处理器利用率已经达到 100%。对于 I/O 设备的处理器资源利用，可以通过直接利用分配设备的方法避免处理器在数据传输过程中的介入，从而降低对处理器资源的利用。在 IOMMU 的帮助下，一个直接分配 10G 网卡可以达到大约 6～8Gbps 的性能。

7.5 性能分析案例

下面以 Xen 为例子，运用 Xenoprof 和 Xentrace 发现存在虚拟机中的症结，并解决这些问题。通过实际的例子，使读者对虚拟环境性能优化有一个感性的认识和方向性的参考。

7.5.1 案例分析：Xenoprof

在一台 CPU 主频为 3GHz、前端总线频率为 1333MHz、一级缓存为 64KB、二级缓存为 6MB 以及内存为 2GB 的双核平台上，安装了 Xen 和 Windows XP 并运行 SPEC CPU 2000 性能评测工具后发现，虚拟环境下处理器消耗比较大，达到 90%多，而此时对于物理机仅仅 80%多，所以希望研究一下导致处理器消耗的问题在哪，并优化完全虚拟化的客户机性能。在这个案例中，采用 Xenoprof 调试完全虚拟机的性能。

把 Dom 0 视为唯一的主动客户机，启动一个 Windows XP 完全虚拟机 Dom 1，并把它

作为唯一的被动客户机。在被动客户机中不需要做额外的配置，主要的配置和收集工作都在 Dom 0 中完成。

第一次采集的结果如表 7-1 所示，它是基于文件级别的映射，各列分别是每个映像文件对应的采样数、占总采样数的百分比和采样点所来自的映像文件名。

表 7-1　Xenoprof 文件级采集结果

采样数	(采样数)百分比	映像文件
2018	46.4549	dom 1-kernel
1717	39.5258	dom 1-xen
296	6.814	dom 1-apps
144	3.3149	dom 0-Vmlinux
121	2.7855	Xen-syms
24	0.5525	dom 0-Libc-2.5.so
9	0.2072	dom 0-Qemu-dm
8	0.1842	dom 0-Bash
2	0.046	dom 0-Gawk
2	0.046	dom 0-ld-2.5.so
2	0.046	dom 0-Libxenctrl.so.3.0.0
1	0.023	dom 0-Oprofiled

在表 7-1 中可以看出，消耗处理器时间最多的是 Dom 1-kernel，即 Dom 1 的内核；第二位的是 Dom 1-xen，它表示采样点来自在 Dom 1 完全虚拟机上下文下的 Xen；第三位的是 Dom 1-app，也就是 Dom 1 用户空间的应用程序。

接下来对函数级数据进行采集，如表 7-2 所示。

表 7-2　Xenoprof 函数级采集结果

采样数	百分比	映像文件	函数名
2018	46.4549	dom 1-kernel	(nosymbols)
330	7.5967	dom 1-xen	Number
296	6.814	dom 1-apps	(nosymbols)
276	6.3536	dom 1-xen	vsnprintf
194	4.4659	dom 1-xen	init_vtd_hw
176	4.0516	dom 1-xen	printk
52	1.1971	dom 1-xen	vmx_vmexit_handler
49	1.128	dom 1-xen	trace
29	0.6676	dom 1-xen	vlapic_get_ppr
29	0.6676	dom 1-xen	vmx_get_segment_register
28	0.6446	xen-syms	compat_pg_table_l2
27	0.6215	dom 1-xen	sh_x86_emulate_write_shadow_3_guest_3

续表

采样数	百分比	映像文件	函数名
24	0.5525	dom 1-xen	hvm_set_cr3
24	0.5525	xen-syms	do_suspend_lowlevel
22	0.5064	dom 1-xen	sh_page_fault_shadow_3_guest_3
20	0.4604	dom 1-xen	mmio_decode
19	0.4374	dom 1-xen	shadow_domctl
19	0.4374	dom 1-xen	vmx_set_segment_register
18	0.4144	dom 1-xen	Strnlen
16	0.3683	dom 1-xen	vlapic_write
14	0.3223	dom 1-xen	_flush_tlb_mask
14	0.3223	dom 1-xen	handle_mmio

其中，各列分别是每个函数对应的采样数、占总采样数的百分比、来自哪个映像文件和函数名。基于篇幅考虑，该表中只在此列出了一些最主要的项目。由于 dom 1 是被动客户机，dom 1 中函数级别的采样无法列出。

从表 7-2 中可以看出，vmx_vmexit_handler()消耗处理器时间比较高，这是因为发生 VM Exit 后，对不同的 VM Exit 原因虽调用不同的处理函数，但这些都是在 vmx_vmexit_handler()中完成的，这是符合预期的。

另外，很多消耗处理器时间比较高的函数都来自内存虚拟，包括 sh_x86_emulate_write_shadow_3_guest_3()、sh_page_fault_shadow_3_guest_3()等，可见，这部分的开销还是不小。现在的内存虚拟实现已经加入了不少优化措施，例如对页表项存在位为 0 或者内存映射访问这两类特殊的页错误的快速处理路径。但在像客户机内核需要频繁修改页表这样的情况下，大量的 VM Exit 还是没办法避免。最终的解决方法只有依靠硬件加速的帮助，即 Intel 的 EPT。

这个例子虽然简单，但很好地说明了 Xenoprof 在发现、分析系统的热点和瓶颈方面是相当简单有效的。

7.5.2 案例分析：Xentrace

我们知道，完全虚拟机的主要延迟来自 VM Exit 到 VM Entry 这期间的处理，而对不同的 VM Exit 处理的差异又有很大的不同，如对 CPUID 指令的虚拟化只需要 Hypervisor 的几行代码即可完成，而对 I/O 的虚拟化则需要借助 dom 0 中设备模型的帮助，处理过程中间需要经历一条复杂的路径，因此 I/O 的延迟也就更大。所以，在完全虚拟机模型中，I/O 性能必将成为影响虚拟机性能的主要因素。本节就是一个用 Xentrace 优化完全虚拟机 I/O 性能的案例。

在该案例里，所运行的硬件平台与上例相同，所不同的是安装 Linux 作为客户机操作系统。

与 Xenoprof 类似，Xentrace 的配置、采样和统计过程主要在 dom 0 中完成，完全虚拟机唯一要做的就是运行一个要被调试的工作负载。

表 7-3 就是用 Xentrace 采集到的一组数据，正如前面提到的，I/O 所消耗处理器时间是最大的，占到整个消耗的近 60%，由于它的处理路径复杂，平均处理时间也十分长。

表 7-3 Xentrace 关于 VM Exit 种类统计结果

VM Exit 类型	发生次数	次数比率/%	时间比率/%	平均处理时间（TSC 周期数）
EXCEPTION_NMI	1834902	77.19	38.83	24757.73
IO_INSTRUCTION	335004	14.09	57.93	202286.01
CR_ACCESS	73537	3.09	1.32	21064.35
Invlpg	29229	1.23	0.97	38684.64
EXTERNAL_INTERRUPT	90877	3.82	0.88	11382.69
PENDING_INTERRUPT	13611	0.57	0.06	5401.97

下面再对每个 I/O 端口访问情况做统计，其结果如表 7-4 所示。

表 7-4 Xentrace 关于 I/O 端口统计结果

IO 端口号	发生次数	次数比率/%	时间比率/%	平均处理时间（TSC 周期数）
0x00000020	73827	22.04	7.43	68235.31
0x00000021	217325	64.87	64.33	200599.85
0x00000040	2792	0.83	0.02	4746.62
0x00000043	1396	0.42	0.02	9440.93
0x00000064	1424	0.43	0.12	56975.49
0x000000a0	1385	0.41	0.10	47350.26
0x000000a1	5117	1.53	0.49	65210.24
0x000001f1	1376	0.41	0.13	62497.02
0x000001f2	1376	0.41	0.10	47072.65
0x000001f3	1376	0.41	0.11	52348.20
0x000001f4	1376	0.41	0.11	53086.07
0x000001f5	1376	0.41	0.11	53117.57
0x000001f6	2752	0.82	0.19	47919.80
0x000001f7	5504	1.64	0.45	55527.22
0x000003f6	1376	0.41	0.11	54876.02
0x0000c000	6880	2.05	25.51	2512883.90
0x0000c002	6880	2.05	0.55	54484.96
0x0000c004	1376	0.41	0.11	55264.09
0x0000c030	36	0.01	0.00	55797.19
0x0000c032	54	0.02	0.00	61755.24

从表 7-4 中可以看出，0x20、0x21 和 0xc000 端口的表现非常突出。0x20、0x21 是主 PIC 的访问端口，访问次数分别占到了 I/O 总访问次数的 22.04%和 64.87%，而且每次访问的延迟也相当大，分别占到了 I/O 总延迟的 7.43%和 64.33%。

主、从 PIC 的频繁发生跟客户机中 PIT 中断处理函数有很大的关系。Linux 2.6.9 中默认设置 PIT 中断的发生频率是 1000Hz，而每次中断时，处理函数 do_IRQ 都要分别访问 0x20 端口 1 次用于发送 EOI，访问 0x21 端口 3 次用于置位和重置相应 IRQ 屏蔽位。这样，中断处理函数处理过程中才不会再发生同一中断而使该函数重入。

显然，要降低 PIC 访问频度最直接的办法是修改 Linux 内核代码，以降低产生 PIT 中断的频率。但完全虚拟化实现的设计原则是不能改变客户机操作系统的代码，这样，只能在降低 PIC 的平均访问时间上作文章了。

其实，完全虚拟化在设计之初就考虑过 PIC 的频繁访问对虚拟机的性能影响，所以对属于它的端口(0x40-0x5f)的访问提供了一条捷径，使其能在 Hypervisor 中直接处理，即 Hypervisor 也提供了一部分 I/O 虚拟化的功能，而不需要经过设备模型。而 0x40、0x43 的平均访问时间也有力地说明了这样做的好处——比其他通过设备模型处理的端口的平均访问时间少了整整一个数量级。以前，在设计这个虚拟机模型时把 I/O 虚拟化放到了 dom 0 的应用程序设备模型中，是为了提供更简洁的 Hypervisor 和避免由设备虚拟化导致的 Hypervisor 故障。但当清楚地看到这样做带来的性能开销时，决定把关键的硬件——PIC 的虚拟化也放到 Hypervisor 中。

图 7-6 和图 7-7 显示出优化后的效果。在图 7-6 中，I/O 消耗，特别是 0x20 和 0x21 端口访问时间，比优化前都得到很大的减少。在图 7-7 中，用 SPEC CPU2000、cyclesoak 和内核编译进行测试，结果发现 CPU 吞吐量与优化前相比，也得到显著的提高。

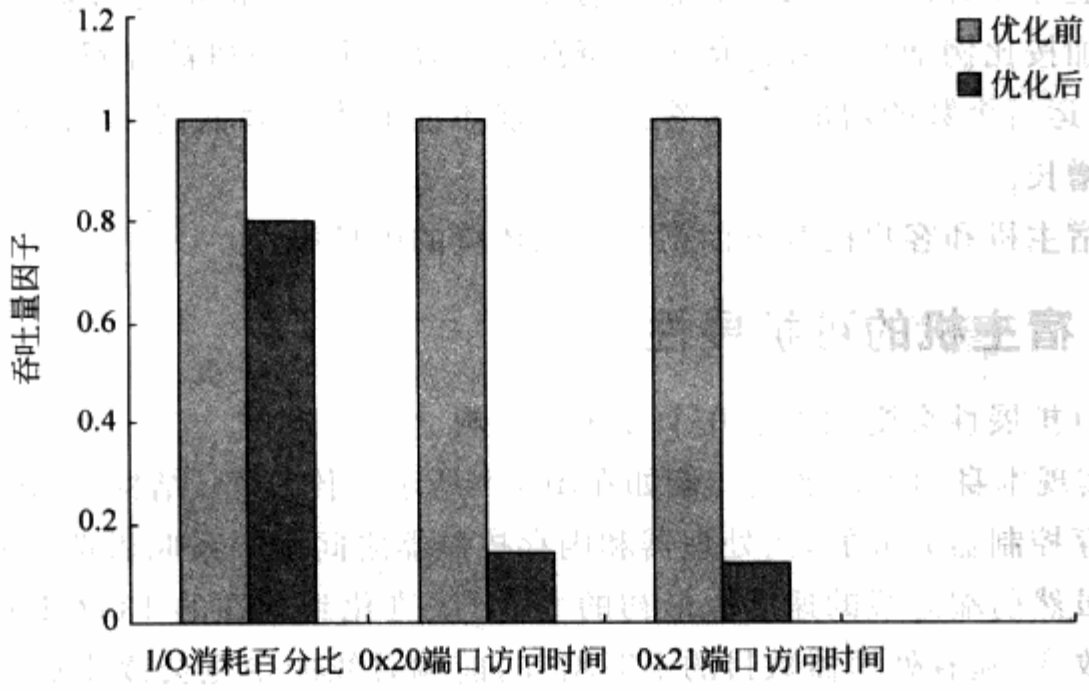

图 7-6 优化前后 I/O 延迟

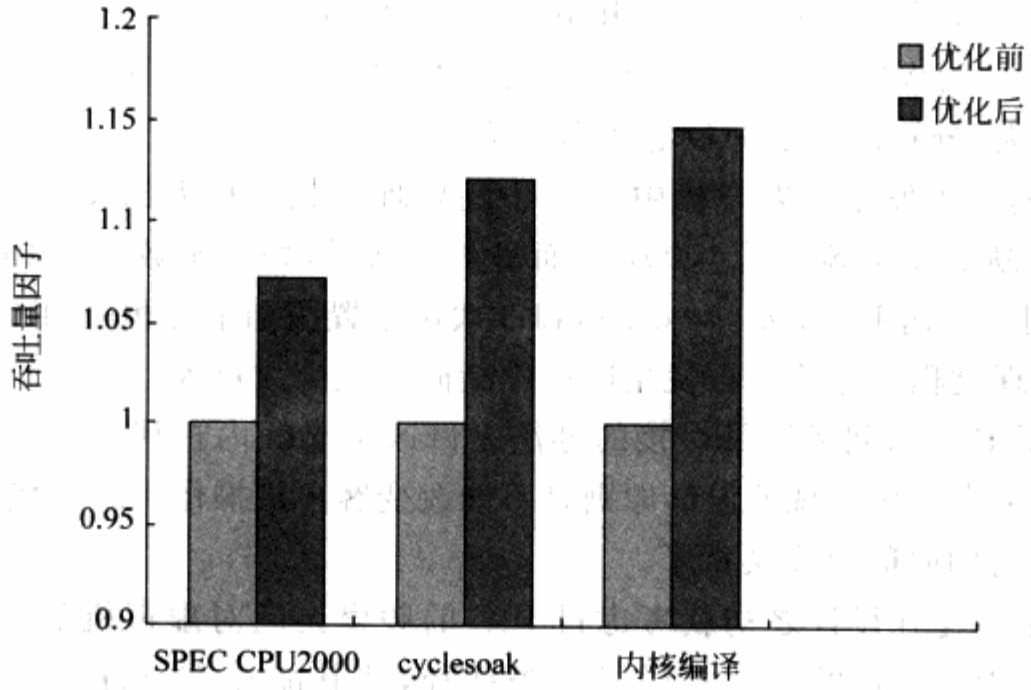

图 7-7　优化前后 CPU 吞吐量

7.6　可扩展性

在此之前，讨论的前提都是在 VMM 上运行一个客户机，以此为评测对象，阐述 VMM 性能评测及其优化的方法。接下来，改变运行虚拟环境的硬件资源（包括增加处理器个数、升级处理器主频、增加内存和磁盘空间等），在虚拟平台上改变运行客户机的总个数，观察和分析由此带给 VMM 性能方面的影响。

一个可扩展性良好的 VMM 应该能够提供更好的性能，虚拟系统的总体能力应该随着硬件资源的增加按比例增长。在总体硬件资源不变且每个客户机被分配的资源不变的情况下，随着客户机运行个数的增加，每个客户机性能不能有大的降低，整个虚拟系统的总体能力应该按比例增长。

下面将从宿主机和客户机两方面衡量虚拟环境的可扩展性。

7.6.1　宿主机的可扩展性

宿主机的可扩展性会受到以下几个方面的影响。

（1）硬件实现本身的可扩展性。例如在图 7-8 所示的传统 PC 结构中，访问内存是通过在北桥上的内存控制器完成的，而处理器和内存控制器之间由一条叫做前端总线 FSB 的系统总线相连。虽然前端总线的速度由最初的 66MHz 进化到现在的 1333MHz，但基本的拓扑逻辑并没有改变，随着处理器数目的不断增加，前端总线的带宽会逐步成为瓶颈。因此，首先可以想象的是，增加处理器的缓存就可以在提高处理器性能的同时增加系统的可扩展性。另外，随着硬件技术的发展，架构的变化也会带来性能的提高并增加系统的可扩展性，

例如新一代基于 Nehalem 的平台(如图 7-8 所示)可以进一步提高处理器的可扩展性。

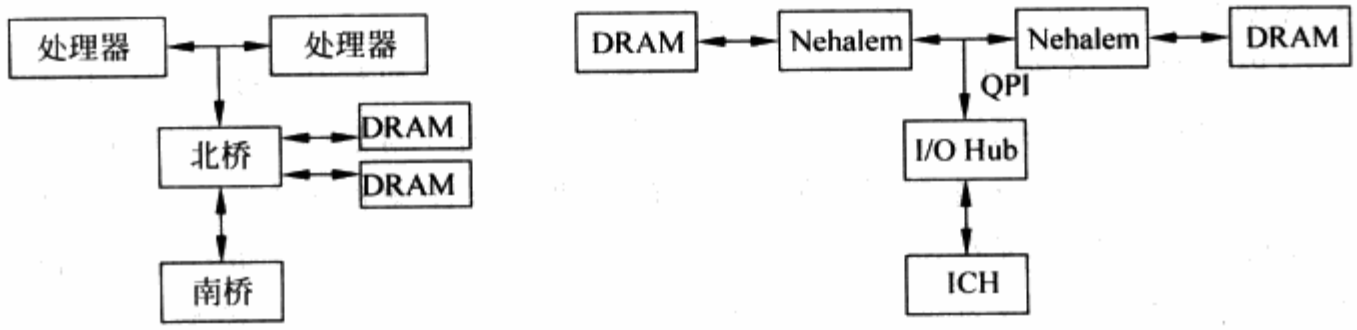

图 7-8　传统 PC 结构和新一代 Nehalem 结构

Nehalem 内部集成了内存控制器,通过独立的内存通道分别将本地 DDR3 内存与远程处理器内存相连,这样对每个处理器就可以主要访问本地内存,避免了访问远程内存而导致的瓶颈。使用新的内存技术 DDR3 和多条内存通道,内存带宽比传统的双处理器 PC 结构提高了数倍。处理器之间由快速路径互连(Quick Path Interconnect,QPI)总线相连,共同构成一个 NUMA(Non Uniform Memory Access)系统。

(2) 系统软件的可扩展性。仍然以上面的为例,虽然 NUMA 系统带来了可扩张性的优势,但由于其对本地和远程内存访问速度的不一致特性,系统程序包括操作系统和 VMM 需要特殊处理以使处理器尽可能地使用本地内存,才能最大程度地发挥其性能。例如,当分配空闲内存时,系统程序应该尽可能地从处理器本地内存中获取,操作系统或 VMM 的调度算法尽可能不将应用程序或 VCPU 调度到非本节点的处理器上。

另外,相对于内存,单个存储设备的速度是非常有限的,而且一个宿主机可连接的存储设备的数量也有限。当宿主机要提供大量存储密集型的服务时,存储系统很容易变成瓶颈。所以,现在在大型的企业级应用中,更好的解决方案是存储区域网络(SAN),即把服务器和专门的存储系统独立开,中间通过高速网络相连。因为最后真正存放数据的地方对于请求服务器是透明的,所以存储空间很容易扩展,存储性能也可以通过负载平衡变得更好。

最后,一个需要考虑的宿主机的可扩张性问题是 VMM 或操作系统中对于中断处理的开销。当一个系统的设备增多的时候,中断服务开销占处理器的处理时间的比例也会上升。多个频繁产生中断的高速设备(如 10G 网卡)的中断开销可能让处理器的处理能力饱和。

7.6.2　客户机的可扩展性

客户机的可扩展性会受到以下几个方面的影响。

1. 硬件资源限制

例如,目前在只有一个处理器的机器上运行很多个客户机很可能让每个客户机的性能低得无法接受。

2. VMM 实现模型

一般来说,在资源一定的情况下,Hypervisor 型的实现是可扩展性最好的;宿主型的实

现是可扩展性最差的，宿主机系统成为所有客户机请求汇聚的地方；对于混杂型实现来说，因为其实现相对灵活，所有可扩展性也可以做得很好。

值得一提的是，混杂型实现中需要一个设备模型，为客户机提供虚拟外设，并提供服务。不同客户机的设备模型可以运行在一个 I/O 服务客户机中，例如，目前 Xen 在 dom 0 中为每个完全虚拟机运行一个设备模型。但是，当客户机越来越多的时候，集中的 I/O 服务客户机可能变成一个瓶颈。现在，Xen 正试图对这个问题进行改进，即让每个客户机的设备模型运行在这个客户机内部或者给这个客户机生成一个专门的运行设备模型的轻量级客户机来提供 I/O 服务。

3. 时间中断

一般的操作系统会编程一个或多个时间源，周期性产生时间中断用于计时、调度等目的。为了让客户机操作系统准确计时，Hypervisor 需要为每个客户机编程物理的时钟源（如 LAPIC）产生同样多的中断，并在中断处理函数中为客户机插入时间中断。随着客户机数量增加，过于频繁的时间中断会消耗越来越多的处理器时间，使性能变得不可接受。对这个问题的一个解决办法是当客户机的 VCPU 被调动出去的时候，停止它在物理时钟源上所设置的定时中断，直到它下次被调度进来时再重新设置并计算中间可能丢失的客户机时钟中断并作出弥补。还有一个更好的解决办法是，不补充丢失的时钟中断，让客户机自己通过读取虚拟平台时钟来修正客户机的正确系统时间，或者采用类虚拟化的时钟驱动程序。对于操作系统来说，一个趋势是避免周期性的时间中断而按需动态地产生。

除了以上几种影响客户机可扩展性因素之外，多 VCPU 的客户机可扩展性很大程度上还受到来自实现过程中对整个虚拟机范围内共享资源的竞争。以完全虚拟机上的内存虚拟为例，现在常用的影子页表如果是整个虚拟机共享的，如 Xen，当 VCPU 数量增加时，对修改影子页表前获得的竞争将严重影响虚拟机的性能。解决办法可以是 VCPU 只维护一份影子页表，但这并不容易，最好是能够借助硬件的支持。在 EPT 推出以后，这个问题有望得到彻底解决。

7.7 思考题

（1）结合影子页表的实现，请思考一下对影子页表的性能优化的关键点在哪里？如何用已有的工具去证实？

（2）请描述一下在哪一种情况下，VMM 需要向客户机提供虚拟的 PMU。

CHAPTER 8

第 8 章 虚拟化技术的应用模式

通过前面的介绍，大家对虚拟化技术已经有了比较深入的了解。这一章会进一步介绍虚拟化技术的各种应用技术，使大家能够了解虚拟化技术是如何被应用来解决实际问题的。

8.1 常用技术介绍

这节主要介绍常用的虚拟化技术，包括虚拟机的动态迁移、快照和克隆。在后面具体的应用方向中，这些技术将会经常被引用到。

8.1.1 虚拟机的动态迁移

1. 什么是虚拟机的动态迁移

迁移，分为系统的迁移和工作的迁移。系统的迁移是指把整个系统的软件，包括操作系统，完全地复制到另一台机器上；工作的迁移是指把某台机器上的一些工作转移到别的机器上。在讲虚拟机的动态迁移之前，先来看看传统的无虚拟机和有虚拟机的静态迁移是什么样子的。无虚拟机下的系统迁移是指在没有虚拟机的环境下把一台机器的所有软件，包括所有状态都复制到另一台机器上。这种迁移可以通过先做出系统镜像，然后复制到其他机器上，或者通过硬盘互相复制的方式来实现迁移。无虚拟机下的工作迁移则需要特殊的系统支持，如哥伦比亚大学的 Zap 系统，通过在操作系统上实现一层薄的虚拟层，实现工作的实时迁移。虚拟机的静态迁移是指在关闭虚拟机的情况下，使用文件传输软件将虚拟机的镜像传输到另外的机器上，然后在相应的机器上启动虚拟机。此种情况下，不能实现工作的迁移。最后再来看看有虚拟机的动态迁移，所谓虚拟机的动态迁移，是指在虚拟机运行的情况下，实时地将虚拟机 1 迁移到另一个虚拟机 2。具体来讲，就是将物理服务器 A 上的虚拟机 1 内的内存里的所有关于虚拟机 1 的信息全部封包通过网络移交到物理服务器 B 上，从而形成了新的虚拟机 2。这是一个内存信息的移交与转换的过程，速度快，其唯一限制就是网络的带宽与突然停电之类的突发情况。在这种模式下，如果 VMM 只把某些工作的信

息传送到其他虚拟机中，则可以实现工作的实时迁移。

2. 虚拟机动态迁移的特点

虚拟机的动态迁移具有如下的优点。

(1) 在非停机的情况下进行迁移，可以实现在线的系统维护，提高系统的可维护性，优化系统中的资源分配，以及通过将不正常工作的宿主机的工作迁移到正常工作的宿主机中实现高可用性。

(2) 在前面的章节中已经知道，虚拟化去掉了硬件的相关性，从而可以实现不同硬件上的虚拟机之间的动态迁移，极大地扩大了可迁移的面，使虚拟机的动态迁移在实用性方面较之以前的方法有了相当大的进步。

(3) 虚拟机通常将硬件资源抽象为一个或一些文件，极大地简化了虚拟机之间迁移的配置过程，更进一步，虚拟机之间的迁移还可以通过配制管理工具自行进行管理，提高了系统的可维护性。

(4) 通过迁移管理工具，可以详细记录迁移，保持追踪，从而在出问题的时候有依据可寻。

还可以指定某一时刻进行迁移工作，无须管理员在场，能有效降低人的使用时间从而降低成本。虚拟机动态迁移也有一些局限性，主要受限于带宽的影响，特别是对于整个磁盘的迁移。目前，对此种局限性已经有了比较好的解决方案，如 VMware 的 Storage VMotion，可以通过磁盘的重定向实现零停机的存储动态迁移。

8.1.2 虚拟机快照

1. 什么是虚拟机快照

顾名思义，虚拟机快照就是把在某一时刻虚拟机的状态像照片一样保存下来。通常，快照将要保存所有的硬盘信息、内存信息和 CPU 信息，VMware 的虚拟机快照还将保存 BIOS 的信息。

2. 为什么要使用虚拟机快照

虚拟机快照被用在很多的地方，包括测试、备份以及安全等。下面就举几个常见的使用虚拟机快照的例子。首先是测试，当想要测试一个新的软件时，并不知道这个软件将会对你的系统产生什么影响，也许这个软件里面有恶意代码，会把你的系统弄崩溃；也许没这么严重，但是会让你的系统变得不稳定或者让其他的软件不能正常工作。如果你在测试软件之前做过快照的话，一切将变得非常简单，只需要回滚到快照之前，干净如初的系统就又回来了。其次是备份，可以定期做虚拟机快照并把快照产生的文件保存到备份盘上面。这听起来很像传统的 Ghost 备份，但使用虚拟机的快照具有以下优点。

(1) 备份可以由虚拟机管理工具来实现，可以在人不在的时候安排备份，可以定时备份，使备份工作简单化。

(2) 通常 Ghost 备份将复制所有的文件，所以生成的备份文件非常大(以 GigaByte 为

数量级),而虚拟机有其自身的文件格式,可以生成增量的文件备份,使备份文件显著减小(以 MegaByte 为数量级)。

(3) 因为虚拟机快照很好地保存了当前虚拟机的运行状态,这些运行的状态可以为程序员的调试提供很好的帮助信息。

8.1.3　虚拟机的克隆

1. 什么是虚拟机的克隆

克隆,是指把一个系统的状态完全不变地复制到另外一个系统上,形成两个完全相同的系统,这里的相同是指操作系统及应用程序的相同。由于 VMM 中维护的信息可能是有所不同的,并且如果从物理机到虚拟机的克隆也可能会有设备上的改动,两个系统的运行环境也就可能不同。虚拟机的克隆主要分为两种:虚拟机到虚拟机的克隆和物理机到虚拟机的克隆。虚拟机到虚拟机的克隆,分为静态克隆和动态克隆,静态克隆即把虚拟机的状态用快照技术保存下来,把保存下来的镜像用文件传输软件复制到其他的机器上。动态克隆即通过网络,同步地把所有的状态迁移到其他的虚拟机上。此种方法的优点在于可以同时对 N 台虚拟机进行克隆操作;缺点在于此间如果断电,所有被克隆的机器将进入不可预计状态,可能将造成比较大的损失。物理机到虚拟机的克隆,此种克隆只能使用静态克隆,因为物理机不具备动态迁移的能力。当需要从一台物理机迁移到虚拟机的时候,虚拟机将会首先虚拟出和此物理机相同的硬件(同样的 CPU、内存和硬盘等),然后通过迁移工具把物理机上的状态全部克隆到虚拟机上。这个功能非常重要,原因主要有以下两点。

(1) 此种克隆不用重装操作系统,不用重装任何软件,传统的系统可以非常方便地移植到现有的虚拟机上,从而可以把以前旧的服务器与新的服务器一并管理,降低管理成本。

(2) 当需要在虚拟环境下测试物理机上已经安装好的软件时,可以把此系统克隆到虚拟机中进行测试。

2. 为什么需要虚拟机克隆

如今的数据中心都是通过将数以万计的机器组成一个整体来进行工作的。部署数以万计的机器需要耗费大量的时间和精力,这显然是不现实的。利用虚拟机的克隆技术,只需要先安装并且配制好一台虚拟机,然后克隆到其他数万台机器上,从而大大降低了整个数据中心的安装和配制时间。以上说的只是一个比较典型的应用,其实任何需要有两台一模一样软件配置的机器时都可以使用克隆。

8.1.4　案例分析:VMware VMotion 和 VMware 快照

1. VMware VMotion

VMware VMotion 是 VMware VirtualCenter 产品的一部分,主要用于 VMware ESX Server 的动态迁移。由于虚拟机已经很好地把虚拟硬件以及运行在虚拟机上的软件的状态都包装在了虚拟环境下,很多状态可以不需要处理或者简单处理。在动态迁移中就只需要

着重处理以下三个类的状态。

- 虚拟设备的状态，例如 CPU、主板、网络、储存设备和显卡适配器等。
- 外部设备的连接，例如网络、USB 设备和 SCSI 储存设备等。
- 虚拟机内存。

在实际的迁移过程中，总共包括了以下 5 个步骤。

(1) 初始化动态迁移，即选择虚拟机簇里面哪个虚拟机需要被迁移，其目标机器是哪一个。

(2) 虚拟机继续在源机器上运行，与此同时，该虚拟机的内存被提前传输到目标机器中缓存起来。该步骤可能执行多次，在第一次循环以后，后面的几次循环只需要把被改变的内存传输过去，以减少时间的损耗。

(3) 停止虚拟机的运行，把当前虚拟机的非内存状态，例如 CPU 状态等，传送到目标机器。

(4) 把虚拟机的控制权移交给目标机器并重新启动虚拟机运行。

(5) 把在停止虚拟机运行时没有传输到目标机器的内存传输过去，并在源机器端销毁虚拟机相关的信息，释放资源。

虚拟机动态迁移的过程中有几个地方需要特别处理，包括网络、磁盘设备还有物理内存。

1) 网络

在虚拟机动态迁移的过程中，由于虚拟机从一台物理机迁移到了另一台，其网络设置将面临一个挑战——原来打开的网络连接必须在迁移后也是连接着的。VMware ESX Server 使用虚拟网络架构技术(Virtual Networking Architecture)解决了这个问题。

在这个架构中，每一个虚拟机都有自己的虚拟网络网卡(Virtual Ethernet Network Interface Card，VNIC)，每个 VNIC 都有自己的 MAC 地址来标志本地网络。而每个 VNIC 都被连接到由虚拟机监控器管理的物理网卡上(NIC)。由于每个 VNIC 拥有自己独立的 MAC 地址，虚拟机就可以随意地在同一个子网中的各物理机器间穿梭，而不用担心网络连接会断开。

其他类似的产品，例如 Xen，也是使用类似的虚拟网络网卡的技术。

2) 磁盘设备

在虚拟机动态迁移前后，如何保证所访问的硬盘资源的一致性和网络一样也是一个挑战，幸好可以将动态迁移时所有相关的资源都储存在网络磁盘服务器上。VMware 就采用了 SAN 或 NAS 服务器来为所有动态迁移所涉及的物理机群提供网络存储服务。

3) 物理内存

在虚拟机动态迁移中，物理内存是所有状态中内容最庞大的一块。先将虚拟机挂起然后再传递所有的页面显然是严重影响执行效率的，会造成虚拟机在动态迁移时大段的服务停顿时间。VMware 采用了影子页的方法来为各个虚拟机提供虚拟的物理内存(影子页的

技术在前面的章节已经有详细叙述，这里不再赘述）。动态迁移的时候，大部分的页面会被预复制（Pre-Copy）到目标物理机上。第一次复制会将所有的页面复制过去。在每一张页面被复制前，它都会被标记为只读，一旦在预复制期间这个页被执行写操作，虚拟机监控器可以立刻发现。在第一次预复制循环后的后续预复制循环中，只有被修改的页面会被再次发送过去。当达到某个条件时，例如预复制循环次数达到预设的最大次数，上一个预复制循环中被修改页的数量小于预设值等，预复制过程会终止，进入下一个迁移步骤。这样的处理方式可以大大降低虚拟机在动态迁移时的服务停顿时间。

2. VMware 快照

最为人们所熟知的虚拟机快照用例就是 VMware workstation 所提供的虚拟机快照功能。最简单的使用方法就是按下快照 Snapshot 功能键，然后给快照取一个名字，加一段描述，VMware 的快照工具就会把当前运行的虚拟的快照储存起来。在虚拟机运行遇到问题时，用户只需要单击“回退（Revert）”按钮就可以回退到以前的快照点。VMware 采用了增量的方法储存快照以减少空间开销。如图 8-1 所示。

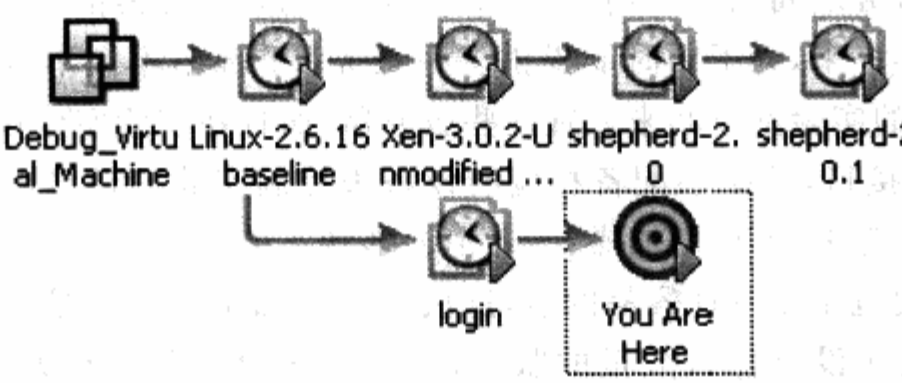

图 8-1　VMware 快照

第一个虚拟机快照 Linux-2.6.16 baseline 的大小将近 1.4GB，而后的快照如 Xen-3.0.2-Unmodified、shepherd-2.0 和 login 等都只有十几 MB 的空间开销。增量方法对于节约空间的贡献是非常大的。当然，由于使用了增量的方法，各个快照就不再是独立的了，例如 Linux-2.6.16 baseline 就同时是 xen-3.0.2-Unmodified 和 login 两个虚拟机快照的父快照。子快照往往可以依据父快照来建立，并且只需储存必要的信息，如两个快照之间的差异等。删除某个快照的代价是昂贵的，特别是多个子快照的父快照。其间会遇到两个问题：重新确定快照之间的关系，主要是父子关系；填补差异，因为增量的原因，当某个快照被删除时，需要处理好其父快照和子快照之间的差异关系。VMware 的快照工具能够帮助用户处理这些烦琐而又复杂的问题。

Xen 也提供了类似的虚拟机快照功能。Dom0 可以通过使用逻辑盘卷管理（Logical Volume Manager，LVM）来做快照，但是其功能支持并不是很完善。和 VMware 一样，挑战最大的是文件系统的快照。为了节约空间，但又要同时保证内存快照和文件系统快照的一致性，以及虚拟机的可持续运行，文件系统快照的实现有很多挑战。当前，已经有研究人员

从事这类问题研究并进行尝试性的开发。增量式的文件系统快照是一种合理高效的解决方案，但是实现一个高效的、正确的不会产生一致性问题的方案并不是那么简单。使用写时复制(Copy-on-Write)的方法来避免冲突和实现增量是当前所尝试的一种方法。Xen 快照功能的设计目的是提供虚拟机备份和回退功能，目前还在完善当中。

8.2 服务器整合

8.2.1 服务器整合技术

随着技术和应用的发展，提供给人们服务的种类越来越多，例如传统的网站服务、邮箱服务，还有新型的搜索服务、地图服务等。而要支持这些服务和应用的不间断运行，提供这些服务的 IT 部门就必须购买相当多的服务器，在它们上面实施这些服务，并且在日常运行中进行维护。

为了避免不同服务之间相互影响、相互干扰，传统的解决方案都是把一个服务装在一个物理服务器上。这样一来，为了同时能提供多种服务，IT 部门不仅不得不购买更多的服务器，而且还得准备大的实验室空间来放置这些服务器，而且在实验室建设方面还需要安装更多的网络接口、电源接口，还必须配备相应的设备来控制实验室环境，例如气压、气温和湿度。这些服务器运行起来，不仅增加噪音，还消耗能源。除此之外，安装和维护这些服务方面，对于 IT 人员来说也不是一件容易的事，这极大地增加了管理和维护开销。更为重要的是，在一般情况下，由于没有什么大量的服务请求，大多数服务器在大部分时间里其 CPU 计算能力、网络带宽和存储空间都处于空闲状态，硬件利用率低下。

有些解决方案出于节省成本的考虑，采取共享的策略来共享硬件资源。例如一些 ISP 提供商会在一台服务器上安装 Web 服务但提供多个账号，并将存储空间分隔成独立的空间以供每个账号共享。这样，每个账号对应的服务就相对独立且不会相互影响，这是一般 ISP 提供租用服务器的解决方案。虽然这种类型的共享可以增加硬件的利用率，但是由于它是建立在同一个操作系统之上的，其所谓的相对独立也只是相对的，服务的鲁棒性、安全性和可扩展性都会下降。

有些解决方案出于增加鲁棒性、安全性和响应速度的考虑，甚至会将一个服务装在多台物理服务器上，例如 Web 服务、搜索引擎。那么，当一台服务器出现问题时，不至于影响到整个服务。这种类型的方案在增加鲁棒性、安全性和响应速度的同时，却大大降低了硬件利用率。

服务器整合的概念是利用虚拟化技术，在一台物理服务器或一套硬件资源上虚拟出多个虚拟机，让不同的服务运行在不同的虚拟机上，不降低系统鲁棒性、安全性和可扩展性的同时，提高硬件利用率和减少成本开销，参见图 8-2。

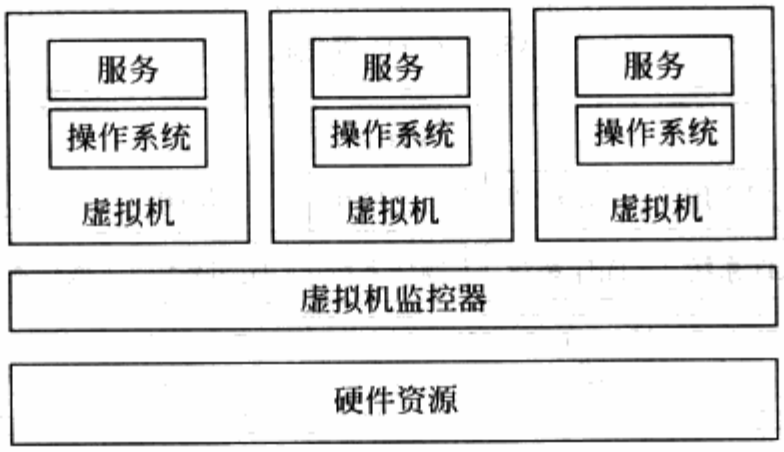

图 8-2　服务器整合

8.2.2　案例分析：VMware Infrastructure 3

在商业系统中，VMware 公司的 VMware Infrastructure 3 就是一个典型的应用虚拟化技术，把诸多物理设备资源虚拟化成多个虚拟机，实现服务器整合的例子。图 8-3 所示为 VMware Infrastructure 3 单个虚拟监控器上的应用架构。

VMware Infrastructure 3 采用了三层管理架构的设计，其中，底层运行的是 VMM（ESX Server）；中间层提供了针对虚拟机集群的基础服务（例如动态负载均衡、虚拟机迁移等）；而最上层则提供了大量企业级应用的管理工具。这种管理架构（如图 8-4 所示）极大地降低了管理的复杂度，提高了虚拟机监控器的工作效率。

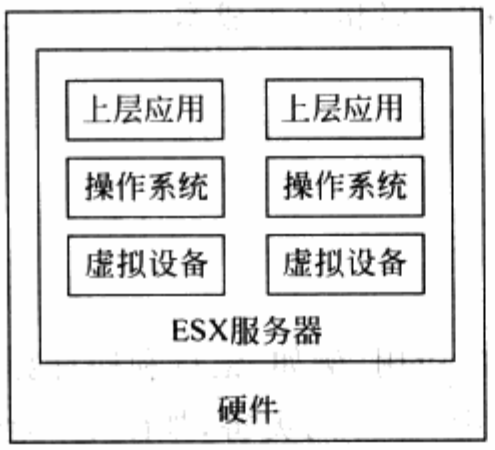

图 8-3　VMware Infrastructure 3 应用架构

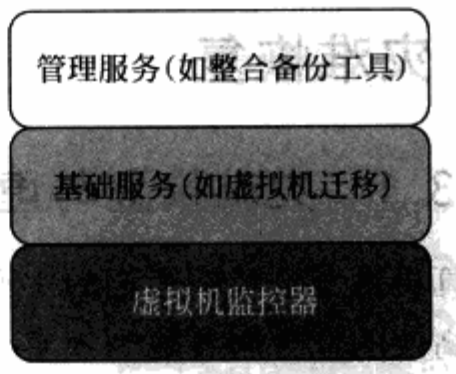

图 8-4　VMware Virtual Infrastructure 架构

VMware Infrastructure 不仅把诸多物理设备资源虚拟化成多个虚拟机，实现服务器整合（如图 8-5 所示），还实现了高效的分布资源调度功能。也就是说，它可以把网络上分布式服务器的硬件资源组织成资源池，虚拟机建立在资源池上而不是某个特定的服务器上。这样，突破基于单机的虚拟机瓶颈，最大可能地平衡分布式资源负载，提高分布式服务器的利用率。

VMware 公司的总结数据表明，其服务器整合解决方案能：

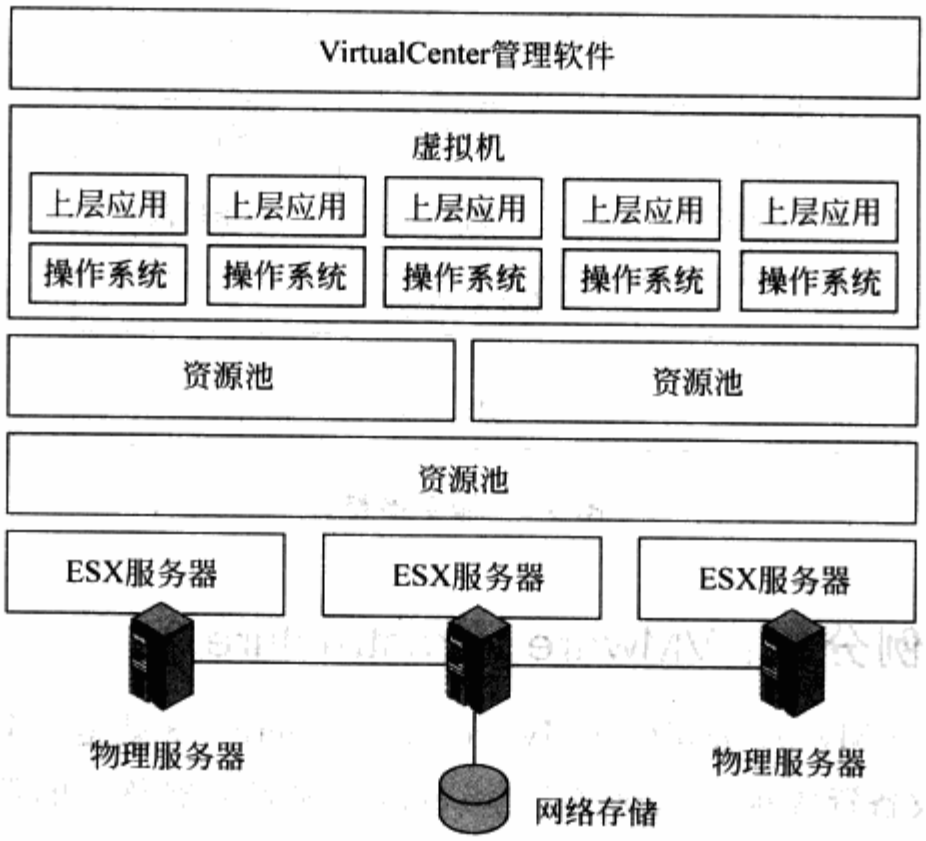

图 8-5　基于 VMware Infrastructure 的服务器整合

(1) 提高硬件利用率从 10%～15%到 80%不等，提高 IT 部门工作效率。

(2) 大大增加系统的可管理性。

(3) 简化服务器的安装过程，节约时间达 50%～70%。

(4) 减少 10 倍或更多的硬件购买需求，节约一半的购买和维护成本。

8.3　灾难恢复

8.3.1　灾难恢复与虚拟化技术

计算机系统已经在人们的生产生活中得到了广泛的应用，例如，银行利用计算机与因特网使得高效的业务服务成为现实。在这些大型的应用中，意外的灾害，如断电、操作失误和硬件损坏，如果处理不当，将造成巨大的经济损失。因此，保证用户数据在灾难发生后的完整性与一致性是学术界与工业界关注的重要问题。但是，由于大型系统往往存储着海量的数据，而系统逻辑往往十分复杂，因此传统的灾难恢复技术存在着如下三方面的缺点。

(1) 缓慢。我们知道，备份往往是灾难恢复的必需手段，而通过克隆原本的计算环境到备用磁盘，我们期望能够在灾难发生时高效地恢复到原来的环境，但一般而言，基于传统技术的克隆十分复杂，以操作系统为例，克隆意味着需要重新安装，配置一个操作系统，并完全克隆其上的应用程序，这往往需要耗费大量的时间与精力。

(2) 不可靠。由于任何一个在原计算环境上的操作都必须备份到备份服务器，任何的

不一致都将导致之后的灾难恢复失败。更进一步，灾难恢复的复杂性使得恢复计划难以测试，而没有经过测试的灾难恢复计划显然是极其不可靠的。

(3) 昂贵。从前面可以看到，为了应对灾难的发生，往往需要克隆原本的硬件与软件环境，因此带来了两倍于以前的成本。

近年来，随着虚拟化技术的进步，越来越多的研究与开发人员将注意力集中到了如何利用虚拟化技术来解决传统灾难恢复技术的局限性，这主要是因为虚拟化技术将整个硬件环境，包括其上的操作系统，应用程序抽象为了一个或几个的文件，从而极大地简化了灾难恢复所需要进行的备份工作。在此基础上，虚拟化技术使得在三方面降低了灾难恢复的难度。

(1) 加快了灾难恢复的速度。由于虚拟机状态被存储在一个或几个文件中，备份和恢复这些文件都相对容易，这极大地缩短了灾难恢复的时间（例如重新安装、重新配置操作系统的时间）。

(2) 增强了灾难恢复的可靠性。由于虚拟机具有硬件无关的特点(VMM 去掉了这一层耦合)。这样的抽象使得虚拟机可以在任何硬件上运行，而无须克隆相同的硬件环境，从而大大提高了灾难恢复的可靠性。

(3) 降低了灾难恢复所需的成本。虚拟化技术使得服务整合成为可能，通过在一套硬件上运行多个虚拟机，提高了硬件的利用率。一方面，这样的整合减少了数据中心的开销；另一方面，由于虚拟机的硬件无关性，我们无须克隆硬件环境，从而减少了灾难恢复的开销。

8.3.2 案例分析：VMware Infrastructure 3

目前，以 VMware 为代表的虚拟化技术厂商已经研发了一定的产品来支持高效、可靠、廉价的灾难恢复。这其中最具代表性的当属 VMware Infrastructure 3(基本架构参见上节)。在上层的管理工具中，VMware Infrastucture 提供了 VMware Consolidated Backup 服务，通过一台 Windows 2003 代理服务器实现对虚拟机备份的集中管理，极大地简化了备份与恢复工作。另一方面，VMware Infrastructure 3 也在相当程度上简化了灾难恢复计划的测试。虚拟机的使用使我们能迅速地开展灾难恢复的测试，可以在独立的网络环境中通过虚拟机镜像来搭建灾难恢复的场景，甚至同时测试多个灾难恢复的场景。一个测试良好的灾难恢复计划将更大程度地保护我们的数据和应用。

8.4 改善系统可用性

8.4.1 可用性的含义

传统的可用性包括易用性和实用性两个方面，易用性即衡量使用一种产品来执行指定任务的难易程度的尺度，实用性则包括了功能的有用性和多样性。一个产品，所拥有的功能越多，并且这些功能会被使用，则实用性就越高。而在计算机领域，我们所谈的可用性是指

通过尽量缩短因日常维护(计划)或突发系统崩溃(非计划)所导致的停机,使系统不间断地向外界提供服务,其理想状态是零停机,即没有停机时间。所以,请读者特别注意这里可用性概念与传统可用性概念的区别。

8.4.2 虚拟化技术如何提高可用性

系统停机的原因不外乎两种情况,第一种是软件出错,例如系统程序崩溃和应用程序崩溃,崩溃的原因主要分为程序自生的 bug 和恶意的攻击者进行攻击使其崩溃,不管是哪一种,都是很难避免的。换句话说,软件出错是客观存在,也是必然存在的,而且在可以预见的未来这种出错也将不可能完全避免。第二种是硬件出错,例如硬件过热烧毁、突然断电以及自然灾害(闪电、洪水)等。同样的,硬件出错也是客观存在且不可避免的。传统的高可用性解决方案如 Microsoft Cluster Service (MSCS) 和 Veritas Cluster Server 致力于在系统发生不可避免的停机时,能在最短的时间内实现服务的恢复。为了达到此目标,其 IT 基础构造需要保证每一台计算机必须有一台镜像,使用群集软件将计算机设置为互相镜像。一般情况下,由主虚拟机向镜像发送心跳信号,当主虚拟机发生故障,镜像接收不到心跳信号时,将由镜像接管主虚拟机的工作。此种方法确实可以保证系统崩溃时的快速恢复能力,而且此方法在软件出错和硬件出错时同样有效。但是,这种方法有着一个显而易见的缺点,每一个计算机都需要一个镜像,这是对资源的一种极大的浪费,而且镜像的管理将会有大量人力的参与,即此方法是资源密集型和劳动密集型的,会带来高昂的成本。而且,此方法要求镜像机与主虚拟机在硬件配制上要完全一样,故只有少数公司在实际生产中会使用这种方法。

8.4.3 虚拟化技术带来的新契机

利用虚拟机技术,可以将多个物理资源群集抽象成和硬件无关的虚拟资源,而硬件资源则像放在池子中的物件一样,由 VMM 分配虚拟资源,故有共享资源池的名称。虚拟机则运行在这些虚拟资源上,VMM 动态监测虚拟机内部的操作系统和应用程序的运行状况,如果操作系统或者应用程序崩溃,则在相应的物理节点重启虚拟机,从而实现软件出错的快速恢复能力。同时,VMM 会动态监测物理节点的可用性,如果某个物理节点失效,则在另外可用的物理节点上重新启动所有受到此物理节点影响的虚拟机,从而实现硬件出错的快速恢复能力。因为物理资源被抽象为和硬件无关的虚拟资源,故使用基于虚拟化技术的恢复方法不受硬件不同的限制,并且在没有出现需要重启某个虚拟机的情况下,所有的资源都可以被利用起来,保证资源的利用率。当需要重启某个虚拟机的时候,VMM 动态地减小已经存在的虚拟机的资源,在有空闲资源的物理节点上重启虚拟机。并且在重新启动新的虚拟机后,可以使用虚拟机动态迁移技术重新分配虚拟机,以达到工作的负载平衡。总的来说,使用基于虚拟机的恢复技术将有以下优点。

(1) 资源的利用得到了保证,没有传统的资源浪费的情况发生,因此种方法不是资源密集型的。

(2) 一切都可以交由 VMM 来管理,无须人的干预,只需要在最初设置一下,因此此种方法不是劳动密集型的。

(3) 利用虚拟机动态迁移技术在故障恢复后重新分配资源,能够获得当前状态下最优的资源分配策略。

(4) 当出错的硬件主机得以修复需要重新工作,或者有新的主机想加入此资源池时,可以很方便地加入,再通过虚拟机迁移技术重新分配资源,使新加入的物理机能很快地得到有效的利用。

此外,通过虚拟化技术,可以实现在线的硬件资源升级,使得日常维护(计划)不需要停机,具体做法如下。

(1) 把需要更新的物理机上的虚拟机使用虚拟机动态迁移技术迁移到其他物理机上。

(2) 此物理机退出,然后更新硬件并重新加入群集。

(3) 使用虚拟机动态迁移技术动态分配虚拟机,以达到资源负载平衡。

总之,通过虚拟化技术,为系统的高可用性带来了非常良好的前景。

8.4.4　案例分析：VMware HA 和 LUCOS

1. VMware HA(VMware Infrastructure 3)

在 VMware Infrastructure 3 的基础架构中,将一组 ESX Server 合并为一个具有共享资源池的群集,使用 VMware HA 监控所有的主机,一旦主机发生故障,VMware HA 会在另外的主机上重启受影响的虚拟机。VMware HA 有一个缺点,它不会动态监控操作系统和应用程序的运行状况,故没有对软件错误的恢复能力,但它可以和传统技术相结合,实现软件错误的恢复。

2. LUCOS(复旦大学并行处理研究所研发)

LUCOS 能够使用虚拟机对操作系统进行动态更新,使用虚拟机主要基于如下几个原因。

(1) 操作系统运行在虚拟机上,虚拟机对操作系统有完全的访问权限。

(2) 因为所有从操作系统到 VMM 的陷阱都是同步的和阻塞的,这个保证了动态更新的原子性。

(3) 在虚拟机中对操作系统进行更新不需要在特定的系统状态,即任何时候都可以更新,由虚拟机来保证更新的一致性。

(4) 因为更新在虚拟机中进行,不会对操作系统的状态带来改变。

LUCOS 具有操作系统的透明性、安全、可靠,并且可以与复旦大学并行处理研究所研发的另一成果自虚拟化(Self-Virtualnation)相结合,实现对运行在物理机上的操作系统的动态更新。具体做法如下：首先使用自虚拟化使操作系统从物理机转到虚拟机里运行；然后使用 LUCOS 对操作系统进行更新；最后再使用自虚拟化使操作系统从虚拟机转回物理机运行。

8.5 动态负载均衡

8.5.1 动态负载均衡的含义

负载均衡是指即通过负载均衡器的协调，将计算任务分摊到多个资源中(资源可以是服务器，也可以是磁盘、CPU)执行，从而在整体上更好地利用计算资源，加快响应时间，提高服务质量。以网络服务为例，目前大型的网站往往通过多台服务器支持某一项服务，为了避免一个服务器节点因分配不均而导致过多的计算任务在该节点执行，通常的做法是在一台单独的服务器上实现一个负载均衡器，扮演网络请求分配器的角色。具体而言，当监听到来自网络的访问请求后，根据当前各个服务器的负载信息和一定的分配策略来决定由哪一台服务器处理该请求。最终，在指定的服务器完成了请求处理后，将处理的结果返回给远端的用户。虚拟化技术的进步为实现高效的动态负载均衡提供了更有力的保证，这主要表现在虚拟化技术简化了计算任务的迁移。在虚拟化技术成熟之前，迁移计算任务因为其较高的复杂性而不能得到广泛的部署，可以设想迁移一个或几个服务进程到更空闲的服务器上去，然后，由于两台服务器计算环境(如操作系统、硬件配置)的不同，迁移工作变得十分复杂。虚拟化技术的优势正是在于其将操作系统及其上的应用程序抽象为了一个或几个文件，从而把真实的硬件资源与软件服务分离开来。因此，计算任务的迁移转化为了虚拟机的迁移。由于 VMM 掩盖掉了硬件的差异性，而上层的软件都包含在虚拟机中，从而操作系统等软件的差异也被掩盖起来。因此，利用虚拟机实现动态负载均衡将更加可靠与简单。

如果把一台服务器视作一个计算节点，那么利用虚拟机技术实现的动态负载均衡主要包含了两方面内容。

(1) 计算节点间的负载均衡。具体而言，即当一个节点负载过重时，把其上的计算任务迁移到另一个节点。一般而言，节点间的负载均衡通常需要负载均衡器和虚拟机迁移技术合作实现。

(2) 节点内的负载均衡。由于虚拟化技术使得可以在一套硬件上运行多个虚拟机，因此为不同的虚拟机合理地分配系统资源 (CPU、磁盘、内存)同样至关重要。在该方面，动态负载均衡通过在 VMM 中实现资源分配算法来调配不同虚拟机之间的资源分配，从而提高整体性能。

8.5.2 案例分析：VMware DRS

VMware DRS 是整合在 VMware Virtual Infrastructure 中的基础服务软件。其主要目的是进行动态负载均衡，更好地为节点内的虚拟机分配计算资源 (CPU、内存、网络和磁盘等)；平衡节点间的计算负载。其主要特点包括以下几个方面。

(1) 利用虚拟化技术，将硬件资源抽象为逻辑资源池。因此，分布式应用程序以虚拟机

的形式运行在逻辑资源池中。

(2) VMware DRS 允许管理员通过 VirtualCenter Management Server 制定资源分配策略,以及虚拟机使用计算资源的优先级,从而按照用户的需求,更大程度地合理分配计算资源。

(3) 即时监控逻辑资源池的使用情况,按照预先设定的策略,在一定情形下(如服务器已经过载)将虚拟机迁移到(通过 VMotion 技术)另一台合适的服务器,从而提高整体服务的性能。

(4) 允许管理人员设置某台服务器的模式,如果服务器进入维护模式,则 VMware DRS 能自动地将服务器上的虚拟机迁移到其他服务器,从而保证高可用性。

(5) 允许方便地加入新的服务器,VMware DRS 将自动重新分配资源给虚拟机,从而有效地利用新的计算资源。

(6) 新的 VMware DRS 中加入了分布式电源管理服务(DPM),通过动态调度来降低数据中心或计算机集群的能源开销。

8.6 增强系统可维护性

8.6.1 可维护性的含义

计算机系统的可维护性是一个相当广泛的概念,在不同的层面上都存在着可维护性的问题。例如,数据中心大量数据和服务器的可维护性,VMM 实现的可维护性(利用 VT 技术实现的 VMM 改善了 VMM 的可维护性)。在本节中,主要关注前者,即数据中心中资源、设备和服务等的可维护性。我们知道,高可用性、可靠、安全是对商务应用的必备要求。例如,考虑一个网络服务的分布式应用程序,我们希望它的停机时间越短越好,但是设备的更新和维护是数据中心运转中必不可少的一个环节。因此,如何提高数据中心的可维护性,简化维护的工作,缩短停机时间就显得很重要。良好的可维护性关键在于提供集中、方便、强大的管理平台。在这个方面,虚拟化技术为提高系统的可维护性带来了一些新的契机,主要表现在以下方面。

(1) 虚拟化技术简化了管理员的维护工作。如在本章的前面部分提到的那样,虚拟化技术成功地分离了硬件资源与软件应用,将硬件、操作系统和应用抽象为一个或几个文件,因此备份与管理工作都变得容易。例如,需要对某台服务器 S 进行维护,利用虚拟机迁移技术,可以很容易地将服务器 S 上的虚拟机迁移到其他服务器,而不影响整个数据中心或者分布式应用的可用性。

(2) 通过整合在虚拟监控器中的一套专门的软件来集中式地管理数据中心中运行的虚拟机。由于 VMM 运行在软件系统的最底层,可以容易地监控硬件资源的使用情况,合理地为其上运行的虚拟机分配计算资源。

8.6.2 案例分析：VMware VirtualCenter

在增强可维护性方面，具有代表性工作的是 VMware VirtualCenter。该软件是集成在 VMware Virtual Infrastructure 管理层的软件，通过一台集中式的服务器（运行 Windows 2000、2003 Server、XP）来管理，配置整个计算环境。VMware VirtualCenter 中包含了数个管理软件，其中，Virtual Infrastructure Client 为 ESX 服务器、虚拟机和 VirtualCenter 所在服务器提供了统一的管理接口，从而使管理人员能方便地完成服务器群的管理工作。在此基础上，VMware VirtualCenter 提供了 CPU、网络和磁盘等重要系统资源的监控功能，从而协助管理员分析整个硬件资源的利用效率。在调配系统资源方面，VirtualCenter 允许管理员使用其中的 VMware DRS 为虚拟机动态地分配计算资源，从而达到负载均衡。在系统备份方面，VMware VirtualCenter 整合了 Storage VMotion，允许管理员将活动的虚拟机磁盘迁移到另一个物理磁盘阵列，从而支持不停机的磁盘维护工作，极大地提高了服务的可持续性。在更新系统方面，VMware VirtualCenter 整合了 Update Manager，支持对 ESX 服务器和某些操作系统（虚拟机）的自动化更新，简化了管理员的维护工作。具体而言，Update Manager 允许管理员定制更新策略（例如，ESX 服务器的更新频率），并根据用户的设定自动地为 ESX 服务器或选定的操作系统（虚拟机）进行更新。更进一步，对于虚拟机的更新，Update Manager 通过快照系统在更新之前自动地备份了虚拟机之前的版本，如果更新失败，管理员可以通过虚拟机快照回滚到上一个虚拟机版本，因此极大地减小了因更新失败带来的风险。最后，Update Manager 与 VMware DRS 合作，在简化维护工作的同时保证服务质量。具体做法是，当有多个 ESX 服务器需要更新时，由 Update Manage 协调，一次将一台机器标记为维护模式，这时 VMware DRS 将运行在该服务器上的虚拟机通过 VMotion 迁移到其他服务器，然后 Update Manager 开始对标记为维护模式的 ESX 服务器进行更新。更新完成后，启用更新的 ESX 服务器，再更新下一台 ESX 服务器。

8.7 增强系统安全与可信任性

8.7.1 安全与可信任性的含义

1. 安全性

安全性包括两个方面：私密性和完整性。私密性是指在没有得到授权的情况下无权访问信息，如果无授权的信息泄露了，则是破坏了私密性。完整性即程序或数据要保持完整，非法的修改程序、对程序或数据内容进行不适当的增删，则破坏了完整性。

2. 可信任性与可信计算

可信任性是指一个实体可以被信任当且仅当这个实体的表现在预期范围内。可信计算是指在计算机中添加硬件设备从而允许软件利用某些特殊的、独立于操作系统的安全策略

构建安全的、互信的计算机环境。

8.7.2　虚拟化技术如何提高系统安全

安全性的提高主要得益于客户机的高分离性和资源的高可控制性。高分离性在这里主要包括客户机与客户机的分离性以及客户机与宿主机的分离性。资源的高可控制性主要包括资源的完全控制权、资源的分离性和资源的易管理性。在传统的安全策略中，一般由操作系统来担任安全的检查与防护。此方法和客户机担任安全的检查相比，有以下缺点。

(1) 操作系统是一个很大且复杂的系统，相对于客户机来说更加容易被攻破，一旦操作系统被攻破，就谈不上什么检查与防护了。

(2) 在大型的企业环境中，不同主机上的操作系统很难检查与保护主机间的安全，而客户机则可以统一管理，具有更强大的安全检查能力和易用性。

(3) 系统在已经被攻破的情况下检查和保护仍然可以得以进行，客户机可以安全地重启被攻破的系统，对于很多的攻击来说，重启就意味着攻击的失效(如 DoS)。

1. 保护系统私密性

首先可以利用客户机的高分离性来保护你的私密数据，把不同的私密数据放在不同的客户机中，即客户机与客户机的分离，或者把不重要的数据放在客户机中，私密数据则放在宿主机中，即客户机与宿主机的分离，使得即使客户机被攻破，黑客也无法拿到你真正关心的私密信息。两种典型的用法如下。客户机与客户机分离：在自己的机器上运行两个客户机，一个用来做高风险，高价值的活动，如银行、投资；另一个客户机则用来进行玩游戏、看电影和上网等娱乐活动。即使上网的时候被黑客远程攻破，他所能做的只是拿到你用来娱乐的客户机上的信息，无法拿到另一台客户机上的信息，如银行账号及密码。客户机与宿主机分离：使用虚拟浏览器进行上网活动，所谓虚拟浏览器即是把浏览器运行在客户机上，再通过运行在客户机上的浏览器进行上网，即使上网的时候被黑客攻破，他所能做的只是把你的客户机弄崩溃，并不能窃取到你宿主机上任何的私密信息，诸如家人的照片、视频等。此外，VMM 运行在操作系统这一层，对整个系统有完全的访问能力，可以动态地监控私密数据的使用，当发生私密数据使用不适当的时候，客户机有能力阻止私密数据的使用，从而有效地防止私密信息的泄露。

2. 保证系统完整性

VMM 运行在操作系统之上，对资源的分配和管理有着完全的能力，VMM 可以分离数据和程序代码，动态地监控代码的使用和数据的使用。用户可以通过客户机提供的接口来配置策略，当策略不被满足时，则客户机拒绝执行相应的代码或者数据的访问，从而保证代码和数据的完整性。策略可以是多种多样的，一种典型的策略是数字签名，从用户空间进入内核空间的代码只有当其拥有用户许可(数字签名)的情况下才能得到执行，从而很好地防止代码注入类的攻击，提高系统的安全性。

3. 利用虚拟机进行病毒测试

杀毒软件的制作者每天都需要测试大量的病毒样本，传统的方式是在物理机上测试，因为病毒会给系统带来不可知的影响，甚至系统崩溃，所以每测试一个病毒都需要备份，系统级别的备份与恢复是非常花时间的。当需要测试多种病毒同时在系统中的影响，更是如此，每新加入一种病毒，则需要另做一个备份。而使用虚拟机的快照技术，能非常好地解决这些问题，快照可以方便地记录下每一特定时间的系统镜像，为这种备份之后不久就需要恢复的场景提供了非常方便的方法，大大地加快了病毒测试的进度。同时，虚拟机可以完全地记录下病毒在虚拟机中所做的一切活动，为病毒的完整行为表现提供了非常好的参照，这是传统方法所做不到的。

在一些反病毒软件中，也使用虚拟化技术来检测病毒，通过将可疑的程序安全地运行在虚拟机中进行监控来确定是否为病毒。

8.7.3 虚拟化技术如何提高可信任性

可信计算这个概念被应用到软件领域，通过使用 VMM 提供的独立的、确定的、抗干扰的虚拟机环境，隔离运行那些非可信应用和要求高度安全性的应用。同时，利用 VMM 独立于操作系统的特点完成包括加密解密、身份验证、计算机日志记录和入侵程序检查等安全策略。

8.7.4 案例分析：sHyper、VMware Infrastructure 3 和 CoVirt

1. sHyper(IBM & Xen)

由 IBM 公司研究开发，通过多级的实现 Hypervisor，与 IBM 系统进行合作，提供一个安全的虚拟架构，主要致力于基础系统服务的安全性保证，具有以下的特点：高分离性、虚拟机之间通过仲裁机制进行文件共享、虚拟机之间拥有良好的沟通能力、可以由用户制定安全策略。虚拟机通过使用虚拟可信平台模块(Virtual Trusted Platform Module，VTPM)保证其自身和相应的操作系统的完整性，有关 VTPM 这里不再详细介绍。资源控制和用户正确性验证以及安全服务、分散服务，使得服务更加容易管理及安全，并且提高了服务的利用率。目前，IBM 和 Xen 公司合作开发 rHyper，把这些功能做到开源的 Hypervisor 中，并且可以在 Linux 上运行，而不仅仅限于 IBM 的操作系统。

2. VMware 安全架构(VMware Infrastructure 3)

VMware Infrastructure 3 中提供了一个安全的虚拟架构，致力于解决最新的 IT 工业界中的安全问题，主要有以下特点：用户分组制度，为不同角色的用户提供不同的资源访问；资源池资源控制和授权，资源分为不同的等级，由最高权限的管理员分配资源给部门管理员；跟踪审查，对于指定的特别操作，虚拟机会记录下操作的管理员和操作时间，提供事件的跟踪记录，方便在出问题的时候找到问题及责任人；会话管理，管理员可以监控任何的会话，并且在必要的时候停止会话。

3. CoVirt（Michigan 大学）

Michigan 大学的 CoVirt 系统是一个入侵检测的虚拟机应用，它在硬件之上构建虚拟平台，将操作系统从原来的硬件平台上移到虚拟平台上。这样使得 CoVirt 能够监视整个操作系统，即使在入侵程序修改了操作系统的情况下仍然能够记录下入侵的信息，并能依照安全策略在入侵发生的前后对于入侵程序做出响应。CoVirt 由两部分组成：ReVirt 负责日志记录；BackTracer 负责入侵检测。CoVirt 是在 Linux 的基础上开发的，基于 FAUmachine(源于 UMLinux)操作系统，每个独立的虚拟机上能运行修改过的 Linux 操作系统，但整个系统会带来 14%～35%的性能损失。随后的研究中，Michigan 大学又基于使用漏洞相关的谓词来探测已知的入侵，此策略较日志记录方法有针对性强、开销小(小于 10%)、无误判的优点。但缺点是需要人们手工写漏洞相关的谓词，不过谓词的编写比补丁的编写要容易很多，故在补丁没有出来之前可以先使用谓词进行暂时的防护。

8.8　Virtual Appliance

以往，我们开发一个应用，都是需要考虑其最终要运行的平台配置的。例如，一个 Windows 应用程序就无法在 Linux 系统上运行。对于软件开发人员来说，在其开发过程中首先考虑到目标系统平台(包括硬件配置和操作系统)的种种特性，尽可能地在与目标系统一致的平台上进行开发。而软件测试人员，也需要搭建与目标系统一致的测试平台，然后在该平台上进行各种各样的测试。由于硬件体系结构和硬件厂商众多，操作系统及其各种版本的特点也不尽相同，这样排列组合下来，导致了目标系统平台的种类也是十分庞大的。为了能做到软件功能强大、能跨平台和可通用性强等，软件开发和测试人员就不得不从软件开发阶段开始做大量的工作，如交叉测试。在后期，实施人员也要根据不同客户的实际平台做有针对性的实施工作，而在维护阶段，维护人员在做技术支持工作和维护性工作时，也需要了解和参考客户的平台配置，才能找到问题原因和解决方案。另一方面，对于硬件提供商来说，其生产和发布新硬件，也需要考虑应用和平台的兼容性，也就是生产出来的新硬件需要在各种主流软件上测试通过才能发布。

基于虚拟化技术，上面的过程能够大大地被简化。Virtual Appliance(VA)，顾名思义，就是将预先配置好的应用程序和操作系统一起打包进一个虚拟机，然后一起发布的一种应用，如图 8-6 所示。

通过这种应用可以做到：

(1) 简化软件的开发和测试。对于软件开发和测试人员来说，因为最终产品会将操作系统打包进来，他们无须关心目标系统平台与其开发测试平台的差异性，而将精力专注于软件功能和性能优化上面。关注和消除平台差异性的工作这时就交由 VMM 完成。

(2) 简化软件的部署和维护。由于 VA 产品内应用程序都已部署配置完毕，基本上可以做到拿来就可以直接运行。另外，可以不用考虑在不同平台上运行，对于技术支持和维护

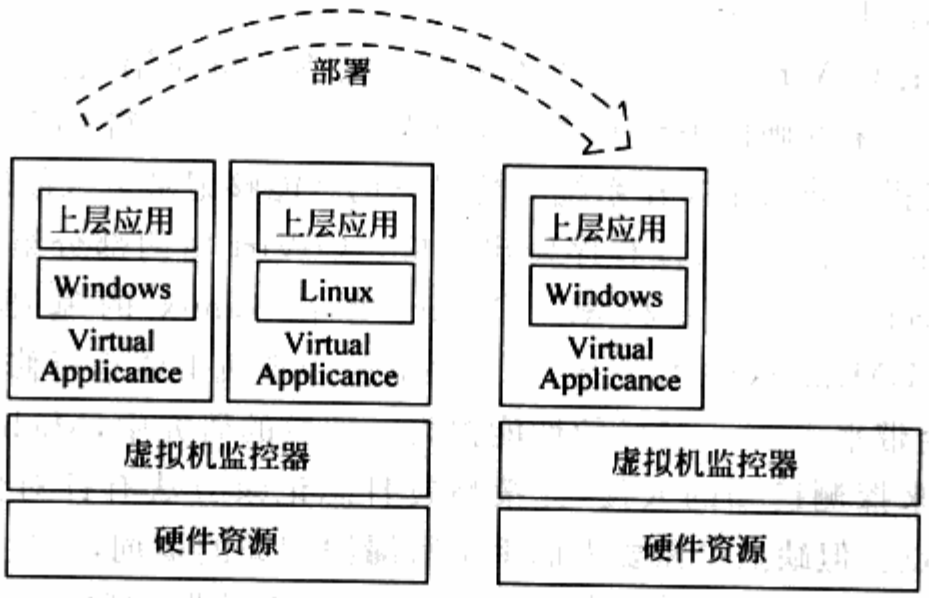

图 8-6　Virtual Applicance

人员来说，更容易找到问题原因和解决方案。流程的简化，自然降低了各种成本，包括开发成本和维护成本。

（3）增强了安全性和独立性。VA 产品中的应用程序都是运行在不同的虚拟机和操作系统里，这样增强了应用程序之间的独立性，避免它们之间相互影响，增强了安全性。

CHAPTER 9

第9章 前沿虚拟化技术

从20世纪60年代诞生至今，虚拟化技术在经历过一段低潮后在20世纪90年代末开始再一次备受学术界和工业界的关注。在本书的前面章节已经涵盖了包括基于软件的完全虚拟化、硬件辅助虚拟化和类虚拟化等在内的当前虚拟化领域的最新技术和产品。然而，虚拟化技术远没有停下它前进的步伐，在诸如容器模型、安全性、系统标准化、电源管理和智能设备等诸多方面都发挥了越来越重要的作用。本章后面部分将着重介绍虚拟化技术在上述前沿方面的最新发展成果，并对虚拟化技术在更多领域的发展趋势作一些展望。

9.1 基于容器的虚拟化技术

操作系统领域一直以来面临的一个主要挑战来自于应用程序间存在的相互独立性和资源互操作性之间的矛盾，即每个应用程序都希望能运行在一个相对独立的系统环境下，不受到其他程序的干扰，同时又能以方便快捷的方式与其他程序交换和共享系统资源。当前面向个人计算机的通用操作系统更强调程序间的互操作性，而缺乏对程序间相对独立性的有效支持。

虚拟化技术因其具有同时运行多个相对独立操作系统的能力而被用来克服上述挑战。VMware和Xen等虚拟化主流产品均采用Hypervisor模型。该模型通过将应用程序运行在多个不同虚拟机内，实现对上层应用程序的隔离。但由于Hypervisor模型更倾向于每个虚拟机都拥有一份相对独立的系统资源，提供相对完全的独立性支持，这种策略造成处于不同虚拟机内的应用程序间实现互操作非常困难。例如，即使是运行在同一台物理机器上，如果处于不同虚拟机内，那么应用程序间仍然只能通过网络进行数据交换，而非共享内存或者文件。

Hypervisor模型这种强独立性保障策略在一定程度上牺牲系统的执行效率。对于高性能计算、Web服务、数据库、游戏平台和分布式系统等许多应用领域，提供高效的资源互操作性同保持程序间的相对独立性具有同等重要的意义。针对这样的需求，学术界提出了

一种基于资源与安全容器概念的虚拟化技术。应用容器模型的虚拟化产品 Solaris 10[1] 和 Linux-VServer[2]，能够在满足基本的独立性需求的同时提供高效的系统资源共享支持。下面将依次介绍资源容器和安全容器的基本概念和发展背景，以及基于容器的虚拟化技术。

9.1.1 容器技术的基本概念和发展背景

容器技术主要涉及两个方面：资源容器和安全容器。下面将分别介绍它们的基本概念、发展背景以及对基于容器的虚拟化技术的影响。

1. 资源容器(Resource Container)

当前比较流行的高性能服务器程序通常是一个资源主体对应多个资源消费者的模式，如事件触发模式(Event-Driven)，或者是多个资源主题对应多个资源消费者的模式，如 CGI 程序。这样造成准确估算出单个资源消费者所使用的资源量变得非常困难，从而无法很好地进行资源管理和控制。例如，在多线程服务器上，一个应用实例对应于一个可以执行多种独立行为的进程，这个进程拥有所有属于它的资源。但在使用线程完成单个任务时，其所使用到的资源往往只是这个进程所属资源的一个子集。由于对资源的控制粒度只能细化到进程级别，因此不可能对单个线程进行独立的资源控制。而对需要多线程协作完成的任务进行资源的统计和控制就更加的难以实现。其他主要的限制来自于资源控制范围、线程调度策略和任务涉及线程差异等在内的多个方面。例如，系统很少对网络资源的使用进行控制，必然造成对涉及网络使用的进程的资源统计数据的变差，进而造成资源调度的不准确。同样的，线程调度中的中断抢占机制会导致计算线程运行时间的偏差，也会引起类似的问题。

学术界为此提出了延时处理机制(Lazy Receiver Processing)(http://en.wikipedia.org/wiki/Comparison_of_virtual_machines。[3] S. Nanda, T. Chiueh: A Survey on Virtualization Technologies)，将进程视为完成某个任务的核心，将网络等资源的使用和进程相联系起来，从而提高了资源统计的精确度。但该机制仍然无法彻底解决线程不是资源控制和统计单元(unit of resource)的问题。

资源容器的概念最早出现在 1999 年 G. Banga 等发表的论文[32]中，主要是为了提高应用程序对系统资源的控制和管理能力。基于资源容器的系统可以把系统中的资源主体(resource principal)从运行主体(即进程)中剥离(process abstraction)出来，从而达到对系统资源进行更精确和高效控制的目的。作为抽象的操作系统载体，资源容器可以拥有一个或多个进程在完成某个任务过程中所使用的所有资源。因此，资源主体不再静态地绑定到进程。进程及其产生的线程与资源主体的关系是动态的、可调整的。甚至来自多个不同进程的线程也可以同时属于一个资源容器。资源容器与任务相对应，记录下任务在执行过程中所消耗的所有资源，包括 CPU、内存和网络等。这样就可以根据这些信息实现对资源的合理调度和控制。资源容器通过引入子资源容器概念可以将任务分解并归类以容器组进行管理，子资源依照一定的规则实现资源的共享，提高灵活性。

基于容器的虚拟化技术通过借鉴资源容器的核心思想，将系统中的资源主体从运行主

体(即虚拟机)中剥离出来,从而达到在虚拟化技术上对系统资源进行精确和高效控制的目的。虚拟机监控器负责对系统中所有的资源容器进行管理和控制,根据用户配置以及系统资源使用的实际情况进行合理分配和回收,实现跨虚拟机的进程间资源共享。但这样的资源控制机制仍然存在安全隐患。

2. 安全容器(Security Container)

克服资源共享带来的安全隐患的有效方法,是采用一套可靠的访问控制机制来防止非法的资源共享。强制访问控制(Mandatory Access Control,MAC)是早期较有影响的访问控制机制,它通过在所有的系统对象上添加有管理员制定的安全策略来限制正在执行的程序的访问权限,从而阻止恶意程序破坏的传播。该方法存在许多限制,首先由于采用了基于安全分级的安全机制,因此只能实现一些普遍的安全策略,无法针对单个程序提出不同的安全策略。其次,它对数据和程序的完整性,以及程序的职能范围无法进行有效的控制。在此基础上,研究者提出了一种名为 Flask[33] 的 MAC 架构,实现了将安全策略逻辑与安全机制的分离,从而能够提供更加便捷的安全策略设置和调整,以满足不同程序对安全策略的要求。SELinux 则是 MAC 机制在 Linux 操作系统上的实现,它涵盖了对进程、文件和套接字在内的多种系统资源的访问控制。通过引入域、角色和类型等概念实现对安全策略的细化和精确定制。虚拟化技术因其卓越的进程和系统资源隔离能力在发展的初期就被用来实现对应用程序的访问控制。将应用程序置于相对独立的运行环境中,由虚拟机监控器依照管理员制定的策略实现程序间的访问控制。

上述这些访问控制机制的实现,往往都依赖于操作系统本身提供的基于保障机密性和完整性的信息隔离机制。但是,这种信息隔离机制却可能被不法程序通过特殊手段绕过,使得访问控制机制形同虚设,而直接对上层应用程序实施篡改和攻击。造成这些潜在威胁的根源在于访问控制机制未能从操作系统中剥离出来。

通过对资源容器的借鉴,学术界提出安全容器概念。其核心思想在于将访问控制机制从操作系统中剥离出来独立于运行环境,依据不同的安全策略形成虚拟的安全容器。在资源容器的基础上,对包括系统进程、文件系统、网络和进程间通信等在内的系统资源进行访问控制。例如,在 Sun 公司较新版本的 Solaris 系统中,就提供了这样的访问控制支持(Solaris Zones[34])。每个应用程序被置于一个独立的运行环境中,各自拥有自己独立的文件系统(整个文件系统的一个子集),网络和设备则根据需要虚拟出来,进程间的通信也被严格地控制。各个程序就像运行在一个安全的容器里面,具有较强的访问控制能力。

基于容器的虚拟化技术借鉴了安全容器的思想,在使用资源容器实现资源共享的基础上通过安全容器技术实现对共享资源的有效访问控制。按照不同的安全对象,诸如域、应用程序和虚拟机等分配不同的可访问资源形成虚拟的安全容器,防止其他对象对其资源进行恶意的、未授权的访问。

下面将详细介绍基于容器的虚拟化技术是如何提高系统独立性和互操作的。

9.1.2 基于容器的虚拟化技术

基于容器的系统需要一个共享的虚拟操作系统镜像。镜像中包括一个唯一的根文件系统，一系列可执行的系统文件和库文件，以及其他建立虚拟机所需的资源。任意一个虚拟机都可以像单机的操作系统一样进行重启、关机等操作。

如图 9-1 所示，基于容器的系统架构由两个平台组成：宿主平台和虚拟平台。宿主平台主要由一个共享的操作系统镜像和一个特权级虚拟机组成。管理员通过特权级虚拟机对客户虚拟机进行管理。虚拟平台由若干个客户虚拟机组成，在客户虚拟机平台上运行的程序与直接运行在物理机上的程序在行为上没有本质差别。上面所述的一些基于容器的系统特性和 Hypervisor 模型下的虚拟机的系统特性类似，而它们的主要区别在于实现程序独立性的方法。图 9-2 给出了两者在安全独立性和资源独立性上的不同结构。基于容器的系统在实现安全独立性时直接使用了操作系统的内部对象（如 PID、UID 等）。如何安全地使用这些对象需要遵循以下两个要点：命名空间的分开；控制访问（如使用过滤器）。在实现“命名空间分开”上，全局的标识符保存在完全不同的空间内，并且各自的空间都不存在指针引用其他空间的对象。这样一来，全局的标识符就变成了各自虚拟机空间独立的标识符。对于“控制访问”的实现，主要依靠对虚拟机访问内核对象权限进行动态检查。在 Hypervisor 系统中安全独立性的实现也是通过命名空间和访问控制来实现的，但更多的是基于包括虚拟内存空间、PCI 总线地址和特权指令等在内的硬件层。基于容器的系统和基于 Hypervisor 模型的系统在资源独立性的实现上大致一致，都需要将诸如 CPU 周期、I/O 带宽等物理资源进行虚拟产生多份虚拟资源。Hypervisor 系统比较有代表性的 Xen 虚拟机监控器系统和基于容器系统的 Linux-VServer[35] 系统都是通过宿主虚拟机来管理网络和硬盘 I/O 的，两个系统的差别仅仅在于它们如何映射资源。

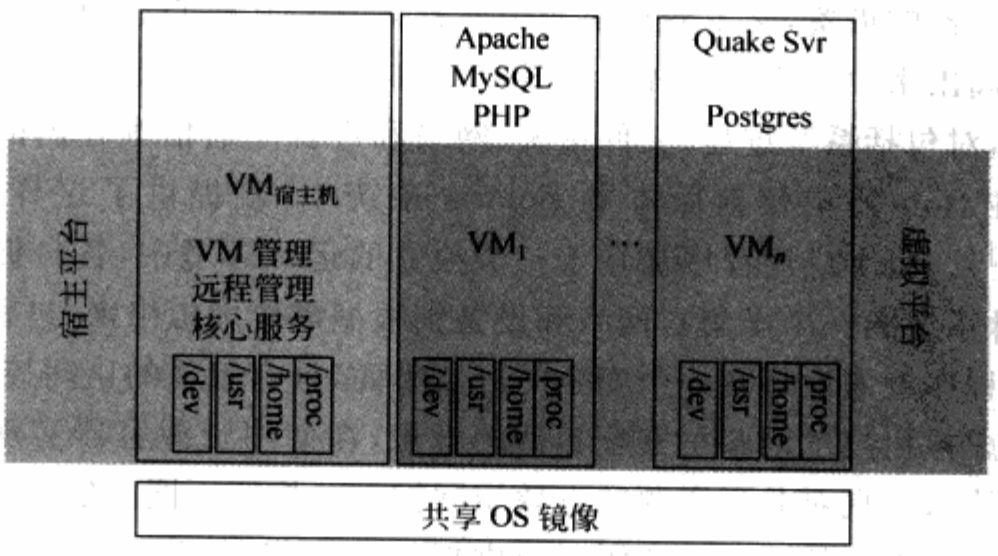

图 9-1 基于容器的系统架构

下面将以 Linux-VServer 系统为例，具体介绍资源独立性和安全独立性等问题。

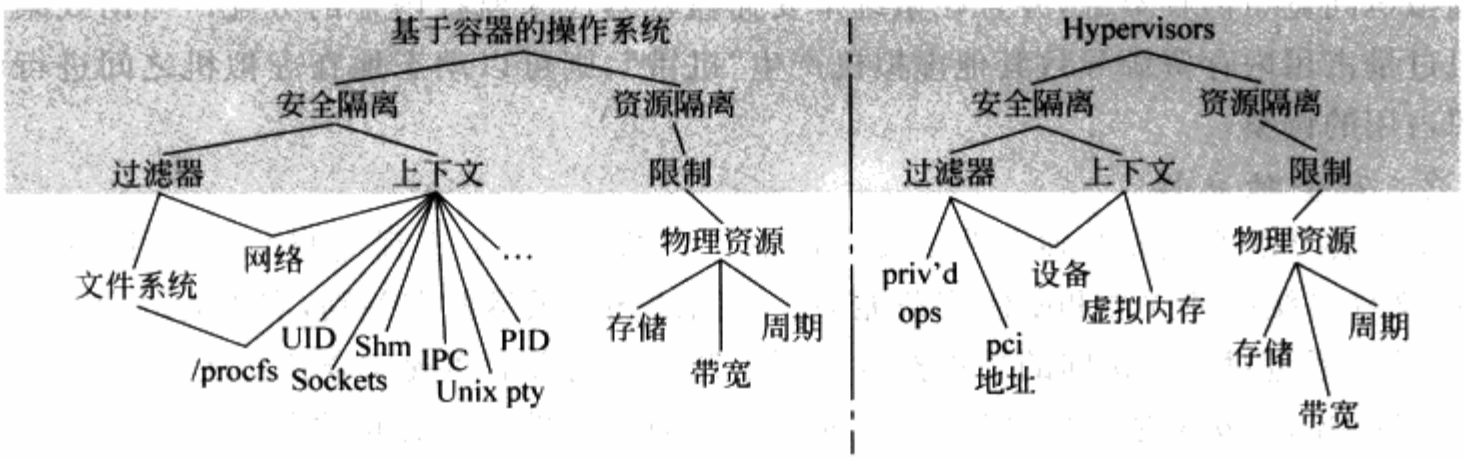

图 9-2　容器与 Hypervisor 对比

1. 资源独立性

资源的独立性主要由 CPU 分配、I/O 分配和储存分配三部分组成。无论哪种资源的分配，公平性总是首先被考虑到的，这是资源独立性的核心。当然，如何高效地利用资源以免造成资源浪费也是一个需要考虑的因素。不同的系统会有不同的实现，但是其遵循的原则是类似的。Linux-VServer 的 CPU 分配、I/O 分配和储存分配都是根据以下原则实现的。

Linux-VServer 的 CPU 分配是在优先级的基础上公平分配 CPU 资源。具体方法就是在标准 Linux 调度器上添加标记过滤器(Token Bucket Filter，TBF)，然后为每一个虚拟机设定运行级别来限定其占用的时间，即标记(Token)，级别高的标记值就相对高，级别低的标记值就相对低。当虚拟机占用 CPU 运行该虚拟机上的进程时，它所拥有的标记值就会持续减少。当某个虚拟机的标记值降为 0 时，它的进程就会从可执行队列里面去除，直到它再次获得足够的标记量以重返可执行队列。每个虚拟机获取标记的速度取决于这个虚拟机是否设置“预订(Reservation)”，或是否和其他虚拟机共同占用“剩余时间资源”。例如，一个虚拟机有 10%的“预定”，那么它每秒会增加 100 个标记值，每个标记可以让其执行它的进程 1ms，即该虚拟机获得了 10%的 CPU 执行时间。而如果该虚拟机同时设定了占用“剩余时间资源”，那么它的进程将会在最低级别的空闲进程被调度执行前获得占用 CPU 的权利，即在所有有预定的虚拟机的执行请求被满足后，有权限使用剩余的 CPU 资源。

Linux-VServer 同样也用基于标记的方法来分配 I/O 资源。在网络 I/O 资源的使用上，每个虚拟机的标记同样需要设定“预定”和共享“剩余 I/O 资源”两个属性。遵循类似于 CPU 分配的机制进行 I/O 资源的分配，在满足按配置分配和公平分配的前提下，实现合理利用剩余资源。硬盘 I/O 资源也采用类似方式在修改 Linux 原有磁盘 I/O 分配器的基础上实现。

Linux-VServer 同时实现了对硬盘和内存资源使用的限制机制。对于硬盘资源，管理员可以指定虚拟机最多可以占用的硬盘节点数。对于内存资源，管理员可以设置如下限制参数：内存最大使用数；可用匿名页表数；可被锁定页表数。这样就可以对硬盘和内存资源的占用进行控制。为了更高效地使用内存资源，VServer 还支持一些其他分配策略，例如

让虚拟机间竞争内存资源，并且在系统里安插监视程序来监控内存的分配。当出现某个虚拟机过量占用内存资源导致其他虚拟机产生“饥饿”，则可以动态地在虚拟机之间进行内存资源占用的调整。

2. 安全独立性

基于容器的系统安全独立性主要由以下几个部分组成：进程空间的独立性、网络的独立性、文件系统的独立性和限制容载能力。不同的部分虽然各有差别，实现的取舍也各不相同，但是都通过对资源访问的严格控制达到了针对不同方面实现安全独立性的目的。

PID 空间的安全独立性，即虚拟机进程空间的独立性是十分重要的。如果没有提供对进程空间的严格控制，将可能造成恶意的虚拟机利用控制机制中的漏洞攻击位于其他虚拟机内的进程的情况出现。因此，对于每个虚拟机来说，它能够看到的和操作的进程空间仅限于它所处的虚拟机空间。VServer 的进程过滤机制能够把所有的虚拟机进程都限制在独立的虚拟机空间里面，防止在两个不同虚拟机空间的进程产生没有经过授权的交互。

网络的安全独立性同样重要，数据包的发送和接收需要绑定到虚拟机并使用严格的认证机制，否则恶意虚拟机和进程就能够轻易地通过伪装监听或者截取数据包，来非法获得未经许可的数据。VServer 并没有完全虚拟化网络系统，它在所有的虚拟机内共享网络系统。但是，如果虚拟机想要改变其绑定的套接字，只有在虚拟机建立之初或者借助宿主虚拟机完成。同时，VServer 还使用了标识符和适当的过滤器来解决类似 localhost 这种特殊地址造成的安全问题。

对文件系统的独立性造成的威胁首先是来自于 chroot 系统调用，它可以用来改变目录系统的信息。恶意的虚拟机往往通过 chroot 系统调用将自己的目录设置成未授权的目录，从而盗取或篡改处于同一文件系统中但属于其他虚拟机的文件。VServer 为了限制虚拟机对 chroot 系统调用的任意使用，引入名为“chroot 壁垒”的特殊文件属性，防止未经授权的根目录路径更变。另一方面，存储在磁盘中分属于不同虚拟机的文件也需要保证只能被所属虚拟机所访问。简单的方法是为每个文件添加一个与文件一起存储到磁盘上的特殊标识符，标记此文件的所有权。该方法的缺点是需要对文件系统做一定的修改。VServer 通过利用部分原有属性，如 UID、GID 等的高位空间来储存标志符，避免对文件系统的修改。当文件被访问时，文件的标志符将被检查以防止非授权访问。

除了盗取和破坏信息外，抢占和恶意消耗系统资源也是需要考虑的安全隐患。VServer 对每个虚拟机的容载提供了限制的方法，管理员通过设置资源的最小限制、软限制和硬限制（即最大限制），达到控制虚拟机容载避免可能消耗系统资源的情况出现。最小限制表示分配给某个虚拟机的最小资源量，硬限制是能够分配给某个虚拟机的最大资源量，软限制则是由虚拟机自行决定的资源限制，这个值范围必须在最小限制和最大限制之间。虚拟机使用的资源的上限是软限制，该值可以根据实际需求动态调整。目前支持的资源限制包括进程数限制、内存空间限制、磁盘空间限制、用户/组的配额限制和网络流量限制等。

3. 其他因素

基于容器的系统的实现还需要考虑一些其他问题，例如节约空间、系统时钟和资源使用统计等，下面将对这些问题以及处理的方法做简单的介绍。

1）节约空间

在基于容器的系统中，由于所有虚拟机使用相同或者类似的系统镜像，大部分文件如类库等都是可以共享，从而减小整个系统空间资源的开销。但是，这样的实现会破坏系统隔离性并且增加系统管理的难度，从而造成安全隐患。基于共享文件如类库、二进制码等一般都很少会发生变化这一特性，VServer 将这些共享文件的属性设置成只读。恶意的未授权篡改操作都能够被系统截获，只有获得授权的对共享文件的修改和删除操作才被允许执行。

2）系统时钟

由于系统中存在多个虚拟系统，因此基于容器的系统与虚拟化系统面临同样的问题——时钟统计得不精确。问题的根源是虚拟系统只在自己占用的 CPU 时间执行，因此基于容器的系统也需要使用时间修正函数来避免系统时间的偏差和由此而来的安全隐患。

3）资源使用统计

系统资源使用的统计信息是十分有用的，例如 CPU 使用时间、进程数、套接字数和网络包收发数等。这些数据分为两类：一类是表示系统当前状态的，如 CPU 使用时间、内存使用量等；另一类表示资源的累积使用量，如网络总流量等。管理员可以依据资源使用的统计信息进行 QoS 控制和资源使用限制等。基于容器的系统不但需要给出整个系统的资源统计信息，同时还需要给出每个虚拟机单独的资源统计信息。

9.2 系统安全

9.2.1 基于虚拟化技术的恶意软件

现代计算机系统往往使用层次化的结构来构造整个系统。较低的层次对较高的层次拥有控制权，因为较低的层次实现了较高层次所依赖的系统抽象。例如，操作系统充当了根据进程虚拟地址空间访问实际物理内存的桥梁，因而对上层应用程序的内存视图拥有完全控制权。为了能够获得系统的控制权，攻击者和防御者都在向系统中较低的层次迁移。如果防御者的安全服务能够相应地驻留在更低的层，就能够检查、隔离并移除运行在较高层的恶意软件。反之，如果恶意软件比安全服务占据了更低的层次，恶意软件就能够躲避甚至是控制安全服务。

随着虚拟化技术的出现，一种模仿虚拟机行为的恶意软件（Virtual Machine Based Rootkit，VMBR）也应运而生。这种恶意软件位于操作系统之下，处于系统的最底层，因此很难被检测和防御。下面介绍两种基于虚拟化技术的恶意软件：Subvert 和 Blue Pill。

1. Subvert

现有的 rootkit 面临如下两个主要问题。

(1) 不能完全控制整个系统,因为 Rootkit 和入侵检查工具往往会同时占据系统中最低的层次,即同时具有最高优先级。

(2) 现有的 Rootkit 面临着功能性和隐蔽性的取舍。功能越是强大的 Rootkit 越容易留下痕迹,越容易被入侵监测工具发现。

Subvert 是由密歇根大学和微软的研究人员合作提出的,一种基于虚拟化技术的新型 Rootkit,解决了现有恶意软件和 Rootkit 无法运行在更低层次上的局限性。它通过在现有的操作系统下安装一个虚拟机监控器(如 Virtual PC),然后将恶意服务运行在一个隔离的虚拟机中,使得目标主机很难检测到。

图 9-3 描述了新型 VMBR 的整体架构,VMBR 运行在操作系统和应用程序的下层。为了实现这一点,VMBR 必须把自己插到目标操作系统与物理硬件之间,并将目标操作系统作为客户操作系统来运行。VMBR 通过操纵系统的启动顺序确保自己在目标操作系统和应用程序之前被装载。在被成功装载之后,VMBR 利用虚拟机监控器启动目标操作系统。虽然目标操作系统仍能正常启动并运行,但 VMBR 运行在更低的层次上并获得了更高的特权级,对目标操作系统能够具有完全的控制权。

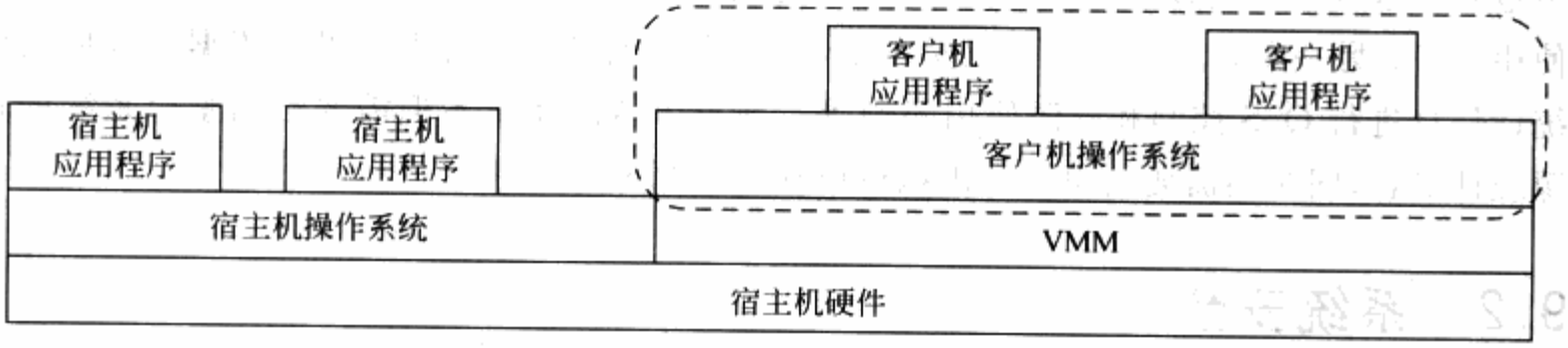

图 9-3 VMBR 架构

安装和启动 VMBR 需要具有足够的权限修改系统的启动顺序,攻击者可以通过多种手段获得该权限。例如,攻击者可以利用远程访问漏洞或者是诱骗用户安装恶意软件等。在获得权限后,攻击者必须把 VMBR 安装到持久存储器上,如磁盘。如果目标系统是 Windows XP,那么可以将 VMBR 安装在第一个活动分区的开始;而如果目标系统是 Linux,则可以禁用交换分区,然后利用交换分区来存储 VMBR。然后,VMBR 修改系统的启动顺序确保其在目标操作系统之前被装载,最方便的方法就是修改系统主分区上的启动记录。

在 VMBR 被成功安装之后,它可以运行多种恶意程序。传统的恶意软件需要在功能性和隐蔽性之间作出权衡,而 VMBR 则不需要考虑这个问题。因为 VMBR 使用一个独立的攻击操作系统来部署恶意软件,对目标操作系统来说,攻击操作系统的所有状态都是不可见的。如此一来,攻击者就可以随意攻击目标操作系统中的程序,而不必担心被检测到。

VMBR 是一种很难被检测到的恶意软件，因为它们虚拟出了目标操作系统的可见状态。一个理想的 VMBR 不会直接修改目标操作系统的内部状态，这使得对其进行检测变得更加困难。但是，入侵检测系统仍然能够觉察到一些 VMBR 存在的迹象。

根据检测系统在系统中驻留的层次，研究人员把检测 VMBR 的方法分为两类，第一类检测方法是在 VMBR 下面运行检测软件来直接获得正确的物理硬件信息，例如借助于安全硬件或者是使用一个安全虚拟机等。通过查看正确的物理内存和磁盘信息，检查软件可以发现由于 VMBR 的存在而引起的异常情况，例如发现启动顺序遭到了修改等，从而判断是否存在 VMBR 攻击。

第二类检测方法是通过运行在 VMBR 之上的软件进行检测。VMBR 的存在必然会对系统造成影响，即使通过虚拟系统状态可以隐藏其中的大部分，但仍然有蛛丝马迹可寻。其中一个就是它需要占用系统资源，包括 CPU 时间、内存和磁盘空间，以及网络带宽等。而指令处理时间上等的延迟是无法隐藏的，因此检测软件可以通过某条指令在执行时间上的差异来确定操作系统是否运行在 VMBR 之上。

2. Blue Pill

Blue Pill 是 Invisible Things 公司的研究员 Joanna Rutkowska 研发的一款恶意软件。它利用 AMD64 的 SVM 扩展将直接运行在硬件上的操作系统动态地搬移到 Hypervisor 上，由 Hypervisor 获得对操作系统的完全控制。SVM 扩展的实质就是对 AMD64 指令集在虚拟化技术方面的指令集扩展。SVM 和 Intel VT-x 类似，能够高效地运行多个客户虚拟机，同时保证安全性和资源的隔离性。

Blue Pill 的执行需要操作系统显式调用其代码。Blue Pill 代码首先开启 SVM 支持，准备好相关结构，然后把程序计数器置为操作系统调用 Blue Pill 代码之后的下一条指令，最后恢复目标操作系统的正常执行。此时，目标操作系统已经作为客户操作系统运行，完全受到 Blue Pill 的控制。Blue Pill 不需要修改 BIOS，启动扇区和系统文件，因此检测非常困难，但是简单的系统重启会导致前一次的攻击失效。

与 Subvert 相同，Blue Pill 也是借助虚拟化技术实现的恶意软件。但是，两者仍然存在一些差异：首先，Subvert 是永久性的，不会因系统重启而失效，但同时也容易被离线检测机制检测到；Blue Pill 可以在系统运行时动态载入，不需要重启来使攻击生效，并且离线检测无法检测到，但会因系统重启而失效。其次，Subvert 在普通的 x86 体系结构上就可以实现；而 Blue Pill 需要基于 AMD64 体系机构的 SVM 扩展支持。最后，Subvert 使用基于商业的 VMM，创建和模拟出虚拟硬件，容易被检查；而 Blue Pill 则使用精简的 Hypervisor 并利用硬件特性，因此性能开销非常小，难以被检测。

9.2.2 虚拟机监控器的安全性

随着虚拟化技术的使用越来越广泛，作为虚拟化技术核心的虚拟机监控器 VMM 的安全性也成为研究的重要课题。

1. 缩小可信计算基础(Trusted Computing Base,TCB)

系统虚拟化技术有着众多的应用场景,其中就包括利用 VMM 来提高系统的安全性,因为 VMM 本身一般被认为是安全的。VMM 之所以可以作为可信计算基础,一个重要原因就是 VMM 的代码量相比通用操作系统要少得多,因此比较容易被验证其安全性。但是,随着 VMM 功能的不断增强,VMM 的代码也变得越来越复杂。有些 VMM,例如 Xen,除了最底层的 Hypervisor 以外,还需要一个特权虚拟机来协助完成虚拟化,这些都导致了可信计算基础的膨胀。在这样的情况下,VMM 本身的安全性也成为需要考虑的问题。

缩小可信计算基础是提高安全性的一个重要手段。在基于 Xen 的虚拟化技术中,除了 VMM 本身,可信计算基础还包括一个完整的操作系统(一般称为 Dom0)和运行在其中的用户态管理工具。管理工具主要用来完成一些需要特权功能才能完成的工作,例如创建虚拟机。因为在可信计算基础上包含了用户态的工具,这就导致可信计算基础被无限扩大了,因为它可以包含所有可能在该物理主机上运行的软件。把一些功能转移到一个独立的安全虚拟机中可以极大地缩小可信计算基础。通过把用户态的工具从 Dom0 中移出,就可以把 Dom0 的整个用户态从可信计算基础中移出。

2. 可信虚拟化环境

在很多应用中都将 VMM 作为信任基础,这就首先要求 VMM 是可信的。VMM 的可信启动是构建可信 VMM 乃至整个可信虚拟化环境的基础。由于 VMM 是和硬件接触最紧密的软件层,直接依赖于系统固件和硬件,可以利用硬件技术验证 VMM 的启动。

Intel 公司开发的可信任执行技术(Trusted Execution Technology,TXT 技术),利用硬件扩展提供一种建立动态可信基础(或称为动态可信根)的机制,从而增强平台的安全性,为建立可信环境及可信链提供必要的支持。

当“可信”这个词被使用时,首先想到的问题就是“谁能保证可信”,“谁又可被别人所相信”。可信平台通过这样一种认证机制,把在其上面运行的软件按启动顺序逐一进行检测,为其他后续软件的执行营造一个可信环境,从而保证整个平台的可信。通过可信平台的认证机制,整个系统从启动开始便自然地建立了一条可信链,在这条可信链上执行的软件是可以被相信的。通常情况下,检测都是从平台启动开始的。最先是 BIOS,接着是启动加载程序,然后再是内核。也就是说,平台启动这一时刻是后面建立起来的可信链的唯一可信基础,这也是通常所说的“静态可信根”(如图 9-4 所示)。维护基于静态可信根开始的可信链,其工作是十分困难和低效的。因为系统经过一段时间的运行后,已经有无数的未知软件在平台上得到执行,所以无法再确定是否当前的执行环境仍然是可信的。由此可见,“静态可信根”对建立可信执行环境而言显得过于庞大,无法适应当前的安全需要。针对上述问题,TXT 技术通过使用“动态可信根”来缩小可信计算基础,克服构建链式可信环境的根本问题,建立动态可信根机制。通过动态可信根机制,检测工作就可以不必从平台启动开始,相应的可信链的构建也没有必要从平台启动开始。可以在系统执行的任何时刻,利用 TXT

技术动态地切入到一个完全独立的安全隔离环境，系统只需要信任在隔离环境中执行的代码，动态加载的这些被隔离执行的代码就是“动态可信根”（如图 9-4 所示）。

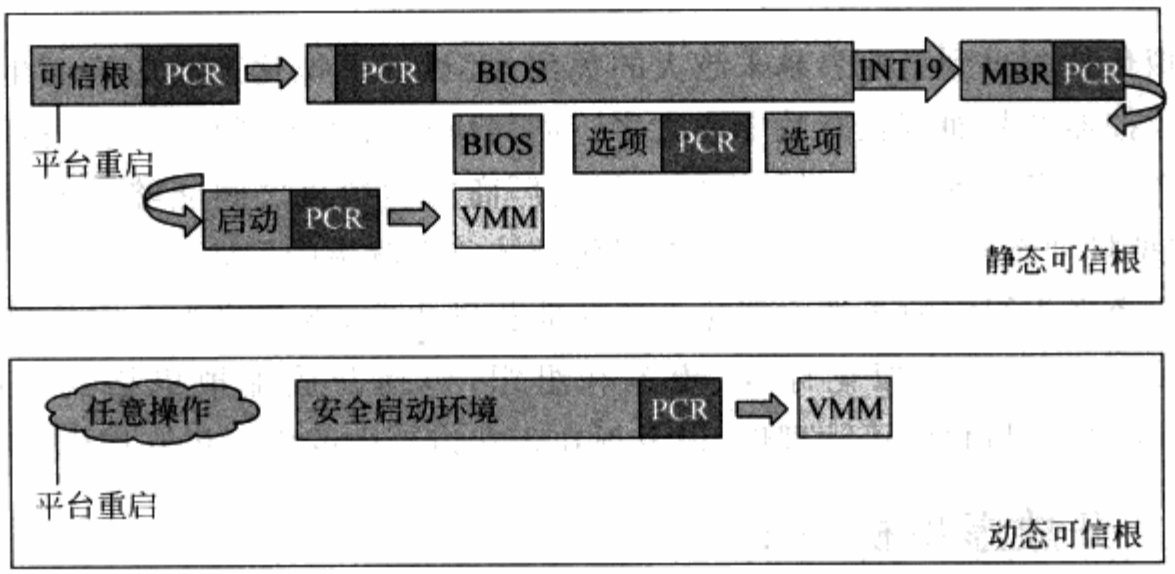

图 9-4　静态可信根与动态可信根

基于 TXT 建立的可信平台，由以下几部分组成：安全模式指令扩展（Safer Mode Extensions，SMX）、认证代码模块（Authenticated Code Module，AC Module）和检测过的安全启动环境（Measured Launched Environment，MLE）。“安全模式指令扩展”对现有指令集进行了扩展，引入了一些与安全技术密切相关的指令，通过执行这些指令，系统能够切入和退出安全隔离环境。认证代码模块是由芯片厂商提供的经过数字签名认证的，与芯片完全绑定的一段认证代码。当系统切入安全隔离环境时，最先被执行的便是这段认证代码，它被认为是可信的，它的作用就是用于检测后续安全启动环境的可信性。安全启动环境则是用于后续检测内核以及启动内核的软件，它不仅需要对即将启动的内核进行检测，同时还需要保证这种检测过程是受保护的，以及它自身所运行的代码不会被篡改。

基于 TXT 技术，英特尔公司开发出了能对 VMM 和操作系统内核启动进行检测的安全启动环境 Tboot（http://tboot.sourceforge.net）。它利用 TXT 技术来构建“动态可信根”，在切入安全隔离环境时进行一系列的可信检测，检测中用于比较的各项检测目标值都被保存在可信平台模块（Trusted Platform Module，TPM）的存储空间里，利用 TPM 自身的安全性使得这些值不会被恶意代码所获取和篡改。Tboot 能够使用 TPM 的平台配置寄存器（Platform Configuration Registers，PCR）来保护和控制启动顺序，使得非法应用不能跳过这些检测步骤直接进入到可信链的后续环节。同时，Tboot 还能在系统关闭或睡眠前清除敏感信息以防泄露，使得可信环境在安全可控制的过程中退出。除此之外，Tboot 还利用支持 TXT 技术的芯片中自带的 DMA 受保护区域（DMA Protected Range，DPR）和 VT-d 的受保护内存区域（Protected Memory Regions，PMRs）来防止恶意代码通过 DMA 跳过检测直接访问内存中的敏感区域。

9.3 系统标准化

随着虚拟化技术得到工业界越来越大的关注，大量的虚拟化产品应运而生。单就最核心的虚拟机监控器产品而言就有数十家之多。如何协调产生虚拟化领域的相关标准，使得各类虚拟化产品的发布等有规可循、不同虚拟化厂商的虚拟机之间能够更方便地进行虚拟镜像共享、虚拟机动态迁移等协作，成为虚拟化领域急需解决的问题。

虚拟化厂商牵头制定的各类虚拟化系统标准相继出现，本节将为大家简单地介绍虚拟化领域当前主要的一些虚拟化标准、规范和组织，包括开放虚拟机格式(Open Virtual Machine Format)、虚拟化的可管理性，以及现在正在讨论中的虚拟机互操作性标准等。

9.3.1 开放虚拟机格式

随着虚拟化技术的快速发展，虚拟机和操作系统造成的开销变得相对较小，而虚拟机提供的对资源的有效隔离使得“在一台物理机器上运行多个虚拟机，而在一台虚拟机上运行一个应用程序或者服务”的模式比“在同一个操作系统上运行多个应用程序和服务”的模式要有效并且安全得多。随着这种趋势的不断发展和深入，服务提供商趋向于把服务或者应用程序连同操作系统一起打包，作为一个完整服务套件(Appliance)来发布。对服务提供商而言，只需要关注虚拟化和操作系统产品的选择，大大降低了测试和维护的成本，同时又能缩短开发周期。而对于产品用户，同样只需关注虚拟化和操作系统产品的选择，不用再担心安装和运行新应用程序带来的兼容性问题。然而，当前不同的虚拟机平台上运行的虚拟机格式相差很大，服务提供商仍然需要为不同虚拟机平台开发不同版本的套件和产品。用户也只能根据虚拟机平台来选择套件。为了进一步统一套件发布格式，实现虚拟机平台无关性，目前主要虚拟机厂商协商制订了“开放虚拟机格式(OVF)”标准。

开放虚拟机格式是一种用来发布运行在虚拟化环境中的应用程序和操作系统套件的格式标准。它由戴尔、微软、IBM、VMware、XenSource 和惠普共同提出，并已被 DMTF(Distributed Management Task Force)组织接受。

在介绍 OVF 前，必须要提到另一个概念——虚拟磁盘。现在有几种开放的虚拟磁盘格式，VMware 的 VMDK、微软的 VHD 和开源的 QCOW。虚拟磁盘常常被用来作为发布虚拟机的载体，但是虚拟磁盘格式和虚拟机格式有着本质的不同。虚拟磁盘描述了这个磁盘上的虚拟机操作系统，当虚拟机监控器打开磁盘的时候才会发现上面带了一个操作系统。虚拟磁盘无法描述这台虚拟机应该被如何安装，也无法描述有多个虚拟磁盘的虚拟机，更加无法描述多台虚拟机组成的集群。而这些问题都会成为虚拟化应用普及后的必然需求。

OVF 的一个主要特点是能够让虚拟机的发布具有更强的可描述性。引用 OVF 白皮书

上的一个例子：一个三层架构的系统，网络层负责显示逻辑，应用层负责业务逻辑，还有一个数据库层。一种简单的实现就是把三层分别放到三台虚拟机中。这样，可以在1～3台物理机器上部署这个系统。但是一般来说，网络层往往需要提供大量的服务，业务层少一些，而数据库则更少。这种情况下，可以把每一层都做成一个虚拟机集群服务，每一层都可以按需求部署到单台或者多台物理机器上，每台物理机器上都可以运行多个虚拟机服务。这个例子也正体现了OVF在规模化上的能力。

OVF也提升了虚拟机在不同平台间可移植性方面的能力，使得虚拟机能更容易地同时支持不同的虚拟机平台。OVF将可移植性能力设定了三个级别。

(1) 虚拟机被限制在一种特定的虚拟化产品、CPU架构或者虚拟硬件上运行，也可能同时受到上述几个条件的共同限制。

(2) 虚拟机能够运行在一个系列的虚拟硬件环境上。例如，VMware公司的第四代虚拟硬件产品，同时得到VMware公司的虚拟平台和Xen公司的3.1版本虚拟平台创建的虚拟硬件环境支持。

(3) 虚拟机能够运行在多个系列的虚拟硬件环境上。例如一个套件能够在Xen、KVM、微软和VMware等主流虚拟化提供商的虚拟化平台上运行。

三个级别的可移植性有各自不同的用处。如果只是给一个特定的机构或者组织作开发，第一级和第二级就能够胜任，因为它们通常会采用比较一致的虚拟化平台。如果要作为商业软件发布，则应该提供第三级的可移植性，这样才能满足广大用户的不同需求。

除此之外，OVF还提供了很多其他的功能，例如完整性验证、授权验证等，读者可以进一步参考OVF规范。

9.3.2 虚拟化的可管理性

虚拟化技术中的可管理性不仅仅指如何去操作虚拟机，而且还包括如何把虚拟机的设计加入到整个计算机体系中去，作为计算机体系的一部分。因此，可管理性是指虚拟化技术从体系结构到虚拟机软件这个完整体系上的设计标准。这里将主要介绍一些相关的机构和相关规范。

DMTF是一个非盈利性质的工业组织，专注于企业和系统的管理的互用性。它的成员包括IBM、英特尔、微软、戴尔和Sun等公司。它发布的标准涉及计算机系统的很多重要方面，例如电源管理、启动控制、系统内存和以太网端口等。现在，虚拟化技术也被融入到这个计算机系统。

目前，这个组织已经发布了如下几份虚拟化相关文档：系统虚拟化模型(System Virtualization Profile)、虚拟系统模型(Virtual System Profile)、分配能力模型(Allocation Capabilities Profile)、资源分配模型(Resource Allocation Profile)和通用设备资源虚拟化模型(Generic Device Resource Virtualization Profile)。同时，还有大量文档正在讨论中。

这些文档涉及了虚拟机设计的各个方面：虚拟化功能主机的发现、虚拟资源的发现和

检查、主机上的虚拟机的发现、主机实体和虚拟实体间关系的检查、如何使用主机上的配置来创建和操作虚拟系统，以及虚拟快照功能的使用等各个方面。

以“系统虚拟化模型”为例，它定义了 CIM_System 类，用来表示主机系统；CIM_HostedDependency 类用来描述主机系统和虚拟机系统之间的部署关系；CIM_VirtualSystemManagementService 类用来描述主机系统提供的用来创建和修改虚拟机系统和它们组件的管理服务；CIM_HostedService 关联了主机系统和它提供的服务之间的关系。不仅如此，还定义了这些类的方法，以及它们的接口和功能。其他文档内容就不在此赘述，有兴趣的读者可以直接访问 DMTF 的官方网站获取更多相关信息。

9.3.3 虚拟机互操作性标准

虚拟机间互操作性包含两方面：在一种虚拟机监控器上识别和运行另一种虚拟机（OVF 标准的内容之一）；用一种虚拟机管理程序来管理不同的虚拟机和虚拟机监控器。下面简单介绍目前在虚拟机互操作性上正在进行的工作。

1. 跨虚拟化平台运行虚拟机

要运行来自其他平台的虚拟机，需要解决两个主要问题：识别虚拟磁盘和提供平台相关的 API。

当前主要有三种虚拟磁盘格式，分别是 VMware 公司的 VMDK 格式、微软公司的 VHD 格式和开源软件 QEMU 的 QCOW 格式。三种实现都开放了各自的设计规范，虚拟化平台可以选择支持其中一种或者几种虚拟磁盘。例如，Xen 虚拟化平台也能支持微软公司的 VHD 磁盘格式。

平台相关的 API 和类虚拟化内核有关。目前，最受关注的类虚拟化内核就是 Xen 提供的 Xen-Linux 类虚拟化内核。类虚拟化内核能够获得较好的性能，降低虚拟化带来的性能损失。但其缺点在于必须对运行在虚拟机中的操作系统内核进行修改。因为第三方内核既不利于 Linux 的发展，也不利于虚拟化厂商维护，因此，Linux 设计了标准的类虚拟化接口——Paravirt-ops 接口。VMware 公司提供了这个接口的第一个实现，称作 VMI。借助 VMI 接口实现，同一个 Linux 二进制内核既可以在 VMware 的虚拟机中使用 VMware 提供的类虚拟化接口高效虚拟运行，也可以在真实硬件上直接运行。由于 Xen-Linux 对内核的改动较大，XenSource 公司的 Paravirt-ops 接口仍在开发中。据 XenSource 公司预期，当 Xen-Linux 完成向 Paravirt-ops 接口过渡后，同一个 Xen-Linux 内核可以在 VMware、Xen 和真实硬件三种平台上运行。

虚拟化平台也给出了支持多种不同类虚拟化内核的解决方案。微软在 Windows Server 2008 中提供了虚拟化整合组件，其中包含用来整合 Xen 虚拟化平台上 Hypercall 接口的适配器，可以将 Xen-Linux 内核中的类虚拟化 Hypercall 调用指令转换成 Windows 虚拟化平台的接口调用。Linux 下的虚拟化标准平台 KVM 也同样在开发类似的用于支持 Xen-Linux 的转换组件。

由于 Windows 产品的源代码并不公开，因此长时间以来，Windows 产品的类虚拟化实现难以出现。但是，微软公司近期发布了 Windows Server 2008 的类虚拟化版本。Novell 公司最新版的 SUSE10 能够在改动后的 Xen 虚拟机监控器上以类虚拟化的方式运行 Windows Server 2008。

2. 虚拟化平台管理工具

随着虚拟化技术的普及和大型数据中心开始使用虚拟化技术，对于不同虚拟化平台的整合管理工具的需求越来越强烈，因为大型数据中心或者企业中开始出现多种不同的虚拟化平台需要进行整合管理。微软已经发布了“系统中心虚拟机管理员”(System Center Virtual Machine Manager 2008)软件。该软件能够在同一控制台下管理包括 Virtual Server 2005R2、Hyper-V 和 VMware ESX 在内的虚拟化平台产品。微软通过和 XenSource 的合作，也在开发能用于 Xen 虚拟化平台产品的管理工具。

为了更方便地实现不同虚拟化平台产品在管理层面的整合，虚拟化厂商相继开放了自己虚拟化平台产品的管理接口，例如 VMware 公司提供的 VMware SDK、XenSource 公司提供的 Web Service API 等。

9.4 电源管理

电源管理是系统软件的一个重要功能。虚拟化技术给电源管理的实现带来了新的挑战。本节将首先介绍电源管理的概念，然后就虚拟化环境下的电源管理做简单的介绍。

节约能源、提高能源使用效率是目前社会发展的趋势，在 IT 的各个领域，尤其是服务器和移动设备领域，电源管理的重要性日趋显著。据统计，一个 3000m^2 耗电 10MW 的数据中心每年单用于冷却方面的费用就高达 400～800 万美元，急需通过有效的电源管理技术和手段来降低能耗和制冷等方面的成本。而对于笔记本等移动设备，由于电池容量的限制，更是要求通过电源管理来尽可能地提高电池的续航时间。

计算机系统中的电源管理通过硬件和软件相结合的方法实现[36]。在硬件上，各个功能模块都已经提供电源管理的功能，例如 CPU 可以通过 DVFS(Dynamic Voltage and Frequency Scaling，动态可变电压和频率)功能根据 CPU 实际使用率调节频率降低功耗；内存可以提供接口将某些暂时不用的内存置于低功耗状态；I/O 模块同样可以提供接口将某些设备或功能关闭。在这些硬件提供的功能中，有些是不需要软件干预的，例如对于某些 I/O 链路，硬件会自动监控上面的活动，当链路处于空闲状态达到一定时间，硬件会自动将该链路置于低功耗状态。另一些则需要通过软件调用来发挥作用，例如 CPU 的 DVFS 功能，需要系统软件根据当前系统的繁忙程度来决定让 CPU 处于 DVFS 中的哪一功耗级别。软件的电源管理目前主要由操作系统来完成，操作系统主导的电源管理技术已经比较成熟，在工业界得到广泛应用。

虚拟化技术的出现，给电源管理带来了新的挑战。首先，作为介于操作系统和硬件之间

新的软件层，虚拟机监控器阻断了操作系统对硬件的直接访问，使得传统的操作系统主导的电源管理模式不再适用。其次，当前的虚拟机监控器本身的电源管理功能相对于操作系统来说还比较缺乏，因此如何利用运行在虚拟机之上的操作系统来帮助虚拟机监控器更好地实现电源管理也是一个挑战。

虚拟环境下的电源管理研究在学术界和工业界都还处在起步阶段。当前最新的一些虚拟化环境下电源管理的方法和工具如下。佐治亚理工学院的 Ripal Nathuji 在[37]中提出的名为 Virtual Power 的虚拟环境下电源管理框架，其基本思想是充分利用操作系统的电源管理功能，结合虚拟机监控器的协调功能完成整个虚拟环境的电源管理。具体来说，虚拟机监控器首先为虚拟机提供一组虚拟的电源管理状态(Virtual PM State)，使得运行在虚拟机上的操作系统能够根据这些状态进行电源管理；其次，当操作系统操作要求进入虚拟电源管理状态时，虚拟机监控器截获这些操作并分析操作系统的电源管理策略。最后，再由虚拟机监控器根据可配置的策略来协调各个虚拟机的电源管理操作，实现对整体物理机器的电源管理。

VMware 公司虚拟化环境下的电源管理产品 DPM(Distributed Power Management)[38]实现了粗粒度的电源管理。它的基本思想是在一个大的计算机集群中，根据实际的工作负载情况，通过动态迁移的方式将虚拟机集中到部分节点上，然后将剩余的空闲节点通过休眠等手段进入低功耗模式，从而达到在这个计算中心范围内降低功耗的目的。XenSource 公司也已经在它们的虚拟化产品中实现了对处理器 DVFS 基本功能的支持。而类似 KVM 这样的操作系统级的虚拟化实现，则更多地依靠宿主操作系统本身的电源管理功能来实现整机的电源管理。

总体来说，虚拟化技术给电源管理带来了挑战也带来了机遇，充分利用虚拟化技术的特点更好地实现电源管理将是今后的一个重要发展方向。

9.5 智能设备

I/O 设备的性能是影响虚拟化方案性能的一个重要因素，尤其是网络设备和存储设备。由于数据中心中的存储设备通常也都是通过网络共享的，如 SAN 和 iSCSI 等，因此 I/O 设备的性能问题也就主要是网络设备的性能问题，智能设备也主要指网络智能设备。

传统的基于软件的设备共享方案由虚拟机监控器为上层虚拟机提供虚拟的 I/O 设备，如基于设备模型的完全虚拟化仿真设备和采用前后端分离设备模型的类虚拟化。基于软件的设备模型都是通过虚拟机监控器对客户机 I/O 操作的介入来实现将物理资源在多台客户机间实现共享，如对仿真设备 I/O 读写的捕捉和仿真，以及对分离式设备中 I/O 读写的递推等。由于存在虚拟机监控器的介入，基于软件的设备共享必定会对系统的整体性能产生负面的影响，例如 Xen 虚拟化平台上基于仿真的 Intel E1000 网卡的传输带宽性能大概只有真实硬件能力的十几分之一；类虚拟化下的虚拟网络设备的性能通常较仿真设备要

好，但对于万兆网卡也只能最多到达 1/3，并且这是在牺牲大量 CPU 资源的情况下取得的。

随着万兆网设备的普及以及数据中心对高速网络的需求，基于软件的设备共享方法面临难以满足用户对设备性能需求的问题。基于 IOMMU 技术的完全硬件设备共享方案，如 Intel VT-d 技术，通过直接在物理硬件层面进行设备虚拟，并将虚拟产生的 PCI 设备直接赋给客户机从而获得高性能的方法得到业界的重视，但是一台计算机中可以插入的 PCI 设备数目有限，而支持的虚拟机数目可能动态增加。这也造成基于完全硬件的设备共享方案面临系统的可扩展性问题，而且使用的物理硬件设备数量也直接关系到数据中心的建设和运转成本。

智能设备就是这样一种介于基于软件的设备共享和基于完全硬件的设备共享方案之间的混合型解决方案，其具体形式可以有多种多样，但它们都具有利用在硬件功能减少虚拟机监控器介入的同时，又不完全摒弃虚拟机监控器辅助的共同特点。智能设备可以是基于公开的工业规范实现，也可以是一家公司独享的私有方案。其中，比较有代表性的实现有使用多队列(Multi Queue)的网卡、PCI SIG 的 Single Root IO Virtualization(SR-IOV)和 SolarFlare 公司的 10G 网卡等。下面将一一给予简单的介绍。

9.5.1 多队列网卡

多队列技术已经在今天的原生操作系统中得到广泛应用，如 Linux 操作系统在 2.6.24 版本的内核中就引入了该项技术。它利用硬件提供的多个网络包发送队列和接收队列特性，将重要进程的网络操作直接绑定到一个或多个网络包处理队列上，从而由硬件保证网络带宽。

如图 9-5 所示，一个典型的多队列网卡具有硬件网络包分发功能，可以基于不同的策略将收到的包分发给不同的队列，每一个队列可以有独立的 DMA 引擎，以及独立的中断处理资源。这意味着拥有独立队列的客户机在网络包处理过程中可以减少虚拟机监控器的介入，从而实现网络 I/O 的高性能，例如英特尔公司的 VMDq 网卡[39]。

9.5.2 SR-IOV

随着设备虚拟化需求的推进，PCI SIG 制定了由 PCIe 设备硬件支持的 I/O 虚拟化方案(I/O Virtualization，IOV)。IOV 分成单根 PCI 桥 IOV(Single Root IOV，SR-IOV)和多根 PCI 桥 IOV(Multi-Root IOV，MR-IOV)。SR-IOV 的 1.0 标准已经正式发布。

支持 SR-IOV 的设备硬件具有一个或多个物理功能模块(Physical Function，PF)如图 9-6 所示，每一个功能模块可以有一个或多个虚拟功能模块(Virtual Function，VF)与之相关联。在 SR-IOV 中，每一个 VF 都具有独立的运行所需要的资源，如中断寄存器、DMA 引擎和 I/O 寄存器等，并且可以被绑定到指定的客户机。一个虚拟功能模块对客户机来说就如同一个传统的 PCIe 设备，具有一个 PCI 总线中唯一的 BDF 识别号，这也是虚拟功能模

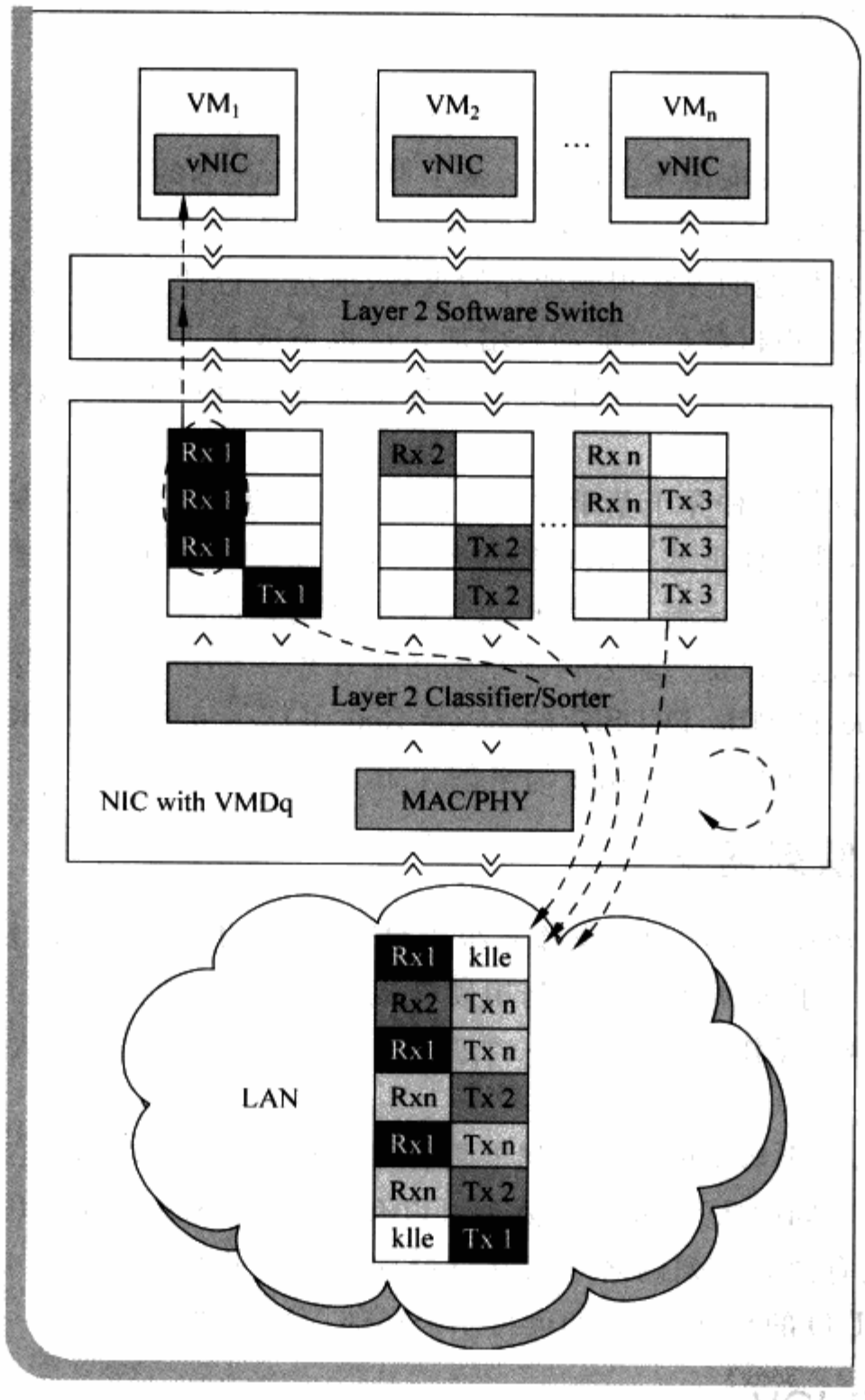

图 9-5　多列队网卡

块上 DMA 操作时硬件用来作为向 PCIe 总线发送访问请求时的识别号。客户机可以访问虚拟功能模块的 PCI 配置空间，但是不同于基于完全硬件的 PCI 设备共享，对虚拟功能模块的 PCI 配置空间访问必须存在虚拟机监控器的介入。这是因为虚拟功能模块的 PCI 配置空间中的寄存器一部分可以直接来自物理功能模块，且为防止虚拟功能模块间在设置上的相互干扰，这些寄存器一般被设置为只读；而另一部分寄存器可以完全由虚拟机监控器辅助仿真实现。SR-IOV 中物理功能模块用来控制跟物理设备相关的全局操作，如链路控制、复位等。

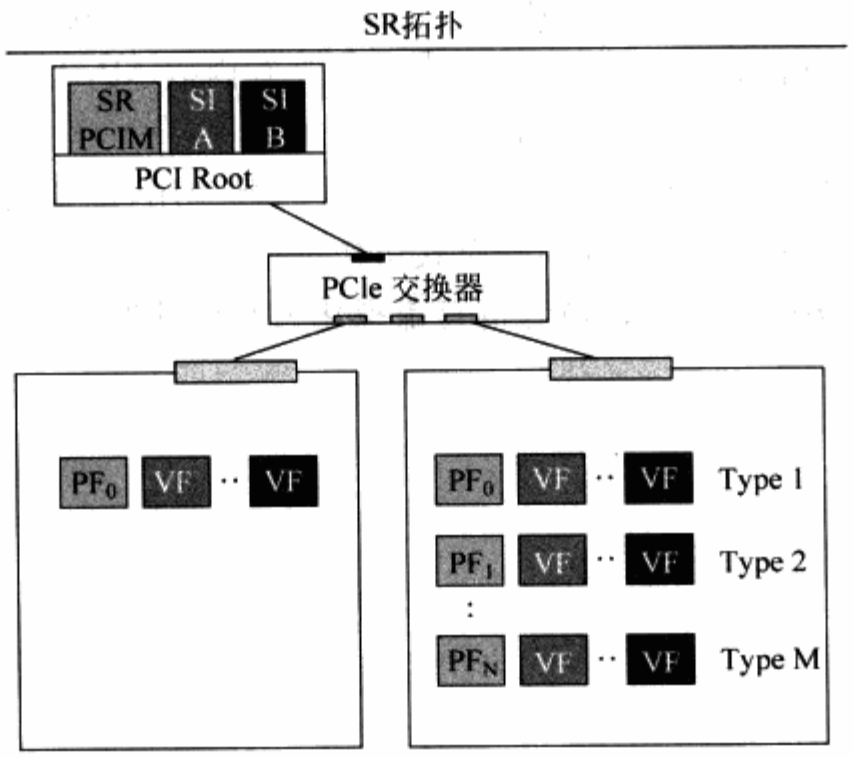

图 9-6　SR-IOV

为了减少 DMA 运行时虚拟机监控器的介入，SR-IOV 设备需要进行地址转换，将客户机驱动程序设置的客户机物理地址转换成主机物理地址，这一点与基于完全硬件的设备共享方案相同，使用 IOMMU 完成。为了更好地支持 SR-IOV 设备的地址转换，PCIe 规范还增加了由设备本身进行地址转换的支持，这可以提高虚拟功能模块与虚拟功能模块间，或者虚拟功能模块与其他 PCIe 设备间访问的效率，使得设备之间的访问不需要由 IOMMU 完成。另外，这种在 PCIe 设备上的内嵌 TLB 地址转换，可以减少对 IOMMU 上 TLB 的压力，从而进一步提高效率。

不同于多队列设备，SR-IOV 设备的 DMA 操作可以由 IOMMU 或者设备内嵌 TLB 实现直接内存访问而不需要虚拟机监控器的介入。因此，SR-IOV 设备具有比多队列设备更好的性能。绑定到 SR-IOV 设备上虚拟功能模块的客户机可以获得与绑定到基于完全硬件的共享设备的客户机在运行时接近的性能，因为它们在运行时都不需要虚拟机监控器的介入。两种方案间的不同主要表现在初始化及控制方式上，绑定到虚拟功能模块的客户机不能直接访问 PCI 配置空间，需要虚拟机监控器和物理功能模块的介入。一个 SR-IOV 的设备，可以具有成百上千个虚拟功能模块，因此比起基于完全硬件的方法具有更好的可扩展性。

9.5.3　其他

当前，除了上述解决方案外，同时存在许多由公司持有的私有解决方案，例如 SolarFlare 公司的 SFC4000 网络控制器。SFC4000 网路监控器包含一个内嵌的 MMU 和多个虚拟网卡，虚拟机监控器可以将虚拟网卡绑定到不同的客户机，每一个虚拟网卡具有独立的 I/O、

中断以及 DMA 资源。跟 SR-IOV 不同的是，它没有 PCI 配置空间，也没有 BDF 号，因为它本身并不是一个 PCIe 设备。这也造成 SFC4000 需要有内嵌 MMU 单元来进行地址转换，实现客户机物理地址到主机物理地址的转换。

随着虚拟化技术的不断发展，以及在更多领域中得到应用，虚拟化技术必将变得越发重要。在代码调试、错误诊断、灾难恢复和分布式协作等诸多方面已经能够看到虚拟化技术的应用。但是，仍然有更多的计算机领域等待着虚拟化为它们的发展做出贡献。

索　引

参 考 文 献

[1] H. McGhan: The gHOST in the Machine

[2] http://en. wikipedia. org/wiki/Comparison_of_virtual_machines

[3] S. Nanda, T. Chiueh: A Survey on Virtualization Technologies

[4] R. P. Goldberg, Survey of Virtual Machine Research, IEEE Computer (June), pp. 34～45, 1974

[5] J. S. Robin and C. E. Irvine, Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, Proc. 9th USENIX Security Symposium, 2000

[6] G. J. Popek, R. P. Goldberg, Formal Requirements for Virtualizable Third Generation Architectures, Communications of the ACM, vol. 17, no. 7, pp. 412～421, 1974

[7] James E. Smith, Ravi Nair, Virtual Machines: Versatile Platforms for Systems and Processes, Morgan Kaufmann, 2005

[8] R. J. Creasy, The Origin of the VM/370 Time-Sharing System, IBM Journal of Research & Development, Vol. 25, No. 5, pp. 483～490, 1981

[9] K. Adams and O. Agesen, A comparison of software and hardware techniques for x86 virtualization, Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, 2006

[10] VirtualBox Architecture, Sun Microsystems, Inc., http://www. virtualbox. org/wiki/VirtualBox_architecture

[11] QEMU

[12] Fedora

[13] A comparison of software and hardware techniques for x86 virtualization PPT http://www. cs. wisc. edu/areas/os/schedules/archive/vmware_osseminar. ppt

[14] PCI Local Bus Specification Revision 3. 0

[15] Intel Corporation, Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i), 2005

[16] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual, Vol3A, System Programming Guide Part 1

[17] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual, Vol3B, System Programming Guide Part 2

[18] Intel Corporation, MultiProcessor Specification v1. 4, 1997

[19] HP, Intel, Microsoft, Phoenix, Toshiba, Advanced Configuration and Power Interface Rev 3. 0b, 2006

[20] Intel, 8259A PROGRAMMABLE INTERRUPT CONTROLLER (8259A/8259A-2) Datasheet, 1988

[21] VMware Inc Technincal Note. Improving Guest Operating System Accounting for Descheduled Virtual Machines in ESX Server 3. x Systems, 2006

[22] VMware Inc. Timekeeping in VMware Virtual Machines

[23] Amsden, Z., Arai, D., Hecht, D., Holler, A., Subrahmanyam, P. VMI: An Interface for Paravirtualization. Ottawa Linux Symposium. 2006

[24] Andrew Whitaker, M. S., and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. OSDI. 2002

[25] Chisnall, D. The Definitive Guide to the Xen Hypervisor, Prentice Hall. 2007

[26] Paul Barham, B. D., Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt and Andrew Warfield. Xen and the Art of Virtualization. SOSP. 2003

[27] Russel, R. lguest: Implementing the little Linux hypervisor. Proceedings of the 2007 Ottawa Linux Symposium. 2007

[28] Kivity, A. and Kamay, Y. and Laor, D. and Lublin, U. and Liguori, A. kvm: the Linux Virtual Machine Monitor. Proceedings of the 2007 Ottawa Linux Symposium. 2007

[29] http://www.sun.com/software/solaris/index.jsp

[30] http://linux-vserver.org/

[31] P. Druschel and G. Banga, Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. OSDI 1996

[32] G. Banga and P. Druschel, Resource Container: A New Facility for Resource Management in Server Systems. OSDI 1999

[33] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen and Jay Lepreau, The flask security architecture: System Support for Diverse Security Policy. The 8th USENIX Security Symposium, 1999

[34] Daniel Price and Andrew Tucker, Solaris Zones: Operating System Support for Consolidating Commercial Workloads. The USENIX 18th Large Installation System Administration Conference, 2004

[35] Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors

[36] V Venkatachalam, M. Franz. Power Reduction Techniques for Microprocessor Systems. ACM Computing Surveys, vol. 37, pp. 195～237, 2005

[37] Ripal Nathuji, Karsten Schwan, Virtual Power: Coordinated Power Management in Virtualized Enterprise Systems, in Proc. 2007 Symposium on Operating Systems Principles

[38] VMware Inc. VMware DRS: Dynamic balancing and allocation of resources for virtual machines

[39] Intel Inc. White Paper: Virtual Machine Device Queues: An Integral Part of Intel Virtualization Technology for Connectivity that Delivers Enhanced Network Performance