中国科学: 信息科学 2010 年 第 40 卷 第 5 期: 692～705
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS

论 文

面向操作系统透明的动态内存半虚拟化技术

汪小林①, 孙逸峰①, 罗英伟①*, 王振林②, 李宇①, 张彬彬①,
陈昊罡①, 李晓明①

① 北京大学计算机科学技术系, 北京 100871
② Deptment of Computer Science, Michigan Technological University, Houghton, MI 49931, USA
* 通信作者. E-mail: lyw@pku.edu.cn

收稿日期: 2009-01-13; 接受日期: 2009-04-10
国家重点基础研究发展计划 (批准号: 2007CB310900)、国家自然科学基金 (批准号: 90718028, 60873052)、国家高技术研究发展计划 (批准号: 2008AA01Z112)、教育部 – 英特尔信息技术专项科研基金 (批准号: MOE-INTEL-08-09) 和美国国家自然科学基金 (批准号: CCF0643664) 资助项目

摘要 文中提出了动态半虚拟化方法 (dynamic paravirtualization), 它借用半虚拟化的方式, 降低带硬件辅助支持的全虚拟化系统中的虚拟机陷出数量. 在有硬件辅助的全虚拟化虚拟机管理器 (VMM) 下, 由 VMM 在二进制代码层对客户操作系统 (Guest OS) 中频繁引起虚拟机陷出的那些热点指令进行替换、消除或合并虚拟机陷出. 这种替换对 Guest OS 而言是透明的, Guest OS 并不会感知到这些变动. 文中重点关注降低内存虚拟化开销的动态内存半虚拟化方法 (DMP). 在 VMM 中实现了一套新的内存虚拟化管理机制, 避免使用影子页表进行虚拟内存地址转换, 并采用代码植入方法消除内存管理引起的虚拟机陷出. 在采用 Intel VT 的 KVM-54 版本中实现了动态内存半虚拟化原型系统. 实验表明, 这种技术可以显著地减少页面中断导致的虚拟机陷出, 从而降低其带来的系统开销. 动态内存半虚拟化能够不修改 Guest OS 源代码也可达到半虚拟化的效果.

关键词
虚拟化
虚拟机管理器
半虚拟化
动态半虚拟化
热点指令
虚拟机陷出
代码植入

1 引言

半虚拟化 (paravirtualization)[1,2] 是一种区别于全虚拟化 (full virtualization)[3] 的虚拟化技术. 全虚拟化技术通过虚拟机管理器 (virtual machine monitor, VMM) 捕获虚拟机 (virtual machine, VM) 上运行的客户操作系统 (Guest operating system, Guest OS) 中所有敏感指令 (包括特权指令以及访问特权数据, 如页表的非特权指令), 并模拟其执行效果 [4]. 半虚拟化则选择修改 Guest OS 源代码, 把产生敏感指令的源代码替换为对 VMM 中相应的超级调用 (hypercall), 主动地让 VMM 接管敏感操作, 避免捕获机制带来的性能开销, 并可在实现上采用一些优化措施 (如合并几个超级调用为一个超级调用) 减少 VM 到 VMM 间的切换开销.

Intel 和 AMD 在 X86 体系架构的 CPU 中都增加对硬件虚拟化技术的支持 (Intel VT、AMD SVM)[5~8]. 以 Intel VT 为例, 它增加了在虚拟状态下的两种处理器工作模式: Root 操作模式和 Non-

引用格式: 汪小林, 孙逸峰, 罗英伟, 等. 面向操作系统透明的动态内存半虚拟化技术. 中国科学: 信息科学, 2010, 40: 692–705

root 操作模式. VMM 运行在 Root 操作模式下, Guest OS 运行在 Non-root 操作模式下. 开始时通过执行 VMXON 指令进入虚拟化状态, 处理器运行在 Root 操作模式下, 执行 VMM 的代码, 当 VMM 要运行一个 VM 时, 执行 VMLAUNCH 指令使处理器发生 VM entry, 进入 Non-root 操作模式, 运行 Guest OS 指令. 此时, 若遇到敏感指令, 处理器会发生虚拟机陷出 (VM exit), 回到 Root 操作模式, 由 VMM 进行处理. 处理完成后, 通过 VMRESUME 指令, 再次发生 VM entry, 进入 Non-root 操作模式继续运行 Guest OS 的指令.

硬件辅助虚拟化技术很好地支持了全虚拟化 (不修改 Guest OS 源码) 并降低了 VMM 的实现难度 (不需要动态二进制翻译). 但目前的硬件辅助虚拟化技术还不完善, 尤其是对 MMU(memory management unit) 以及各种 I/O 设备, 还很难达到高效的虚拟化支持. 单纯基于 Intel 的 VT 技术或 AMD 的 SVM 技术实现的 VMM, 仍然有较大的性能问题 [9]. 一个主要的原因就是过多的 VM exit 导致虚拟化开销较大. 我们对 Guest OS 的陷出点 (trap point, 执行到敏感指令产生 VM exit 的指令位置) 的情况进行了一些统计. 表 1 是我们在 KVM-54[10] 和 Xen-3.2.1[1] 全虚拟化平台上分别统计了一些典型应用程序产生 VM exit 原因的数据. 可以看出, 由于内存管理和 I/O 操作产生的 VM exits 占了绝大多数.

同时, 通过在 KVM-54 上对不同陷出点上的陷出频率进行统计发现, 虽然 Guest OS 中导致 VM exit 的陷出点有上千个, 但陷出频率最高的少数几个陷出点处所产生的 VM exit 次数占总体陷出次数的绝大部分. 表 2 给出了 Guest OS 运行一些典型应用程序时陷出频率最高的前 10 个陷出点的分布情况. 可以看出, 虽然 Guest OS 在运行不同的应用时, 陷出点的陷出频率分布互相之间有差别, 但都相对集中. 前 10 个陷出点处发生的陷出次数, 均占总陷出次数的 60%以上, 有的甚至超过了 80%. 我们将那些陷出频率高、陷出处理总时间长的陷出点称为热点 (hot instructions).

由于陷出处理的时间代价较大, 带 VT 支持的 Intel CPU 一次陷出处理 VMCS(VM control data structure) 结构就需要花费几百乃至上千个时钟周期, 因此如果能够减少 Guest OS 运行时的陷出次数, 将能有效提高虚拟机执行效率.

半虚拟化技术虽然能够很好地提高虚拟化的性能, 但它要求修改 Guest OS 的源代码, 这给其广泛应用带来了很大的局限性. 考虑到热点的集中性, 本文提出了动态半虚拟化 (dynamic paravirtualization) 方法: 结合全虚拟化和半虚拟化这两种技术的各自优点, 在有硬件辅助的全虚拟化 VMM 中, 由 VMM 动态地对 Guest OS 中的热点进行监测并做替换, 消除或合并 VM exit. 与半虚拟化相似, 动

表 1 VM 陷出的原因和计数

		Page fault		I/O		Total
		count	%	count	%	
Kernel Compile	KVM	7459199	83.36	768433	8.59	8948235
	XEN	11219395	81.23	1247207	9.03	13812089
SpecJBB	KVM	329286	2.99	6560472	59.57	11012466
	XEN	352520	4.22	5000529	59.89	8350197
SpecCPU	KVM	34215800	18.69	102468771	55.98	183041637
	XEB	113793259	50.41	56215828	24.90	225727994
SpecWeb	KVM	5668822	15.50	14921351	40.79	36584589
	XEN	11835408	31.46	12428086	33.04	37618375

表 2　KVM-54 中陷出频率最高的 10 个陷出点 [a)]

	Kernel Compile	SpecJBB	SpecCPU	SpecWeb
1	21.21% (pf)	32.48% (io)	12.54% (pf)	4.65% (io)
2	26.07% (pf)	32.47% (io)	9.02% (pf)	7.46% (io)
3	3.71% (pf)	32.47% (io)	8.47% (pf)	7.46% (io)
4	2.44% (rd cr)	0.31% (io)	8.17% (ot)	7.46% (io)
5	2.44% (rd cr)	0.31% (pf)	8.01% (io)	6.08% (rd cr)
6	2.16% (pf)	0.31% (ot)	7.44% (io)	5.77% (io)
7	1.91% (pf)	0.31% (io)	7.44% (io)	5.71% (ot)
8	1.14% (pf)	0.31% (io)	7.44% (io)	5.67% (hlt)
9	0.99% (io)	0.31% (io)	4.54% (io)	5.53% (clts)
10	0.99% (io)	0.18% (ot)	2.07% (ot)	5.50% (rd cr)
Total	68.06%	99.46%	75.14%	64.30%

a) io: I/O 操作; pf: 页面中断; rd cr: 访问控制寄存器; clts and hlt: x86 指令; ot: 其他

态半虚拟化也是通过修改 Guest OS 来提高虚拟化效率. 但与修改 Guest OS 源代码不同, 这里的修改是在对 Guest OS 热点分析的基础上, 通过引入新的机制, 使得可以对热点处的指令片断进行 “去热点” 改造, 然后再由 VMM 在运行时将热点处指令片断替换为改造后的指令片断. 这种修改对 Guest OS 来说是透明的, Guest OS 本身并不能意识到这些修改. 在动态半虚拟化方法中, 并不一定要把 Guest OS 中所有的热点都做替换. 只要能对多数热点进行改造, 大量消除或合并在热点处的陷出, 那么虚拟化的总体效率就可能得到显著的提升.

我们以内存的动态半虚拟化为例验证了动态半虚拟化方法的有效性. 在接下来的章节中, 第 2 节阐述了动态半虚拟化方法, 第 3 节详细介绍内存的动态半虚拟化方法及其在 KVM 中的实现, 第 4 节给出了内存的动态半虚拟化方法的实验结果及其分析, 第 5 节是相关工作介绍, 最后, 第 6 节总结了本文工作并对进一步的工作进行了展望.

2　动态半虚拟化

动态半虚拟化是针对基于硬件辅助虚拟化技术的 VMM, 在 Guest OS 执行状态下, 对 Guest OS 中部分引起大量 VM exits 的热点动态地替换为不会或较少导致 VM exits 新指令片段, 从而达到降低 VM exits 频率, 提高虚拟化性能的一种虚拟化性能优化方法. 热点探测分析、热点替换代码生成和替换代码植入 Guest OS 是动态半虚拟化中所涉及的 3 个关键问题.

2.1　热点探测分析

一种简单准确的热点探测方法是对每一个陷出点都做陷出计数, 这样客户操作系统经过一段时间的运行后, 就可以通过陷出计数找出热点. 但这种方法存在一些效率问题, 特别是无法估算陷出点的数量, 也就无法估计这种方法带来的空间开销.

另一种可行的方法是对陷出点的指令地址进行简单求模运算, 根据余数找到对应入口, 每个入口下面独立地通过陷出计数竞争出陷出次数最多的两个陷出点. 实验表明, 当取散列入口总数为 32 时,

竞争获得的 64 个陷出点的陷出次数之和, 在绝大多数情况下可占到总陷出次数的 97% 以上. 由此可见这种方式以有限的时间空间消耗获得相当准确的热点探测结果.

准确定位热点之后, 可以通过二进制代码解析 (反汇编) 得到热点处执行的敏感指令信息, 进而可以对热点所在函数及热点陷出时的函数调用关系进行捕捉和分析.

2.2 热点替换代码生成

生成热点替换代码的主要目标是为频繁陷出的热点寻找等价的可消除或合并陷出的替代指令片段. 替代指令片段不仅满足不产生 VM exits 或减少 VM exits, 还需要保证其执行效率高, 明显优于"陷出 – 模拟"所产生开销. 但是, 热点替换代码并不能在现有的机制下直接生成, 而是需要在 VMM 中引入新的甚至是复杂的辅助机制才能完成. 第 3 节将详细介绍对内存访问指令的替代及其引入的新的内存管理机制.

在获得热点处敏感指令信息后, 还需要在 VMM 中对敏感指令的模拟过程进行分析, 这能够保证我们对当前热点所应完成的具体效果有清楚的了解. 特别是对模拟过程的分析, 热点替换代码的生成是依据这个分析结果来进行的.

指令替代多数情况下是对单一热点的指令替换, 但对一些特殊的热点, 其前后热点之间存在相互的关系, 这时可以通过对包含多个热点的指令片段做统一的分析后产生等价的指令片段来替换.

2.3 替换代码植入 Guest OS

用于替代热点的新指令片段由 VMM 向正在运行的 Guest OS 中插入, 使得 Guest OS 的执行过程链接到新插入的指令片段上, 我们把这个过程称为代码植入. 代码植入过程是可逆的, 在需要的时候把修改后的指令片段植入到 Guest OS; 而当需要取消植入时, 可以使 Guest OS 的指令代码完全恢复到植入前的状态.

代码植入中需要考虑两个问题: 一是植入的二进制代码放在 Guest OS 中的什么地方; 二是如何保证 Guest OS 能在植入和非植入这两种状态之间方便地切换.

对第一个问题, 主要的解决方法是在 Guest OS 中预留出一段空闲的内核地址空间, 植入的代码放在内核地址空间内. 比较可靠的预留方法是修改 Guest OS 内核代码, 或在 Guest OS 中安装专门的模块或驱动程序. 但这样做就需要在 Guest OS 中做修改, 无法做到对 Guest OS 的透明. 我们采取的方法是, 根据具体 Guest OS 的特点, 在 Guest OS 的内核地址空间中找一块 Guest OS 不经常访问的空间, 由 VMM 监控以保证 Guest OS 只会按我们的安排访问这段空间. 当 VMM 发现 Guest OS 违背我们的安排时, 由 VMM 取消代码植入, 并尝试寻找新的空闲空间做再次植入.

对于第 2 个问题, 我们的解决思路是实现函数级别的代码植入, 而不是代码片段级别的代码植入. 当我们发现热点并得到了替代代码片段时, 我们会把热点所在的函数体整体取出, 用替代指令片段替换热点指令, 得到一个新的函数体. 这个新的函数体插入到 Guest OS 中后, 我们修改所有已经探知的对原函数体的调用, 使之调用插入的新函数. 这样, 如果要取消植入, 只需要把所有修改的函数调用点恢复, 然后在确定 Guest OS 不再访问植入代码时, 把植入代码从 Guest OS 中清除. 由此可见, 代码植入和取消植入都包含两个阶段, 这种两阶段的工作模式确保了 Guest OS 可以在植入状态和非植入状态之间自由切换而不会有任何冲突发生.

由于在热点探测分析过程中已经得到的热点所在函数以及热点陷出时的函数调用关系等信息, 可以方便地实现函数级别的代码植入.

3 动态内存半虚拟化

本节首先回顾半虚拟化和全虚拟化中常采用的两种内存虚拟化方法, 然后介绍结合了这两种方法优点的动态内存虚拟化 (dynamic memory paravirtualization, DMP) 方法.

3.1 传统内存虚拟化方法

Xen 采用半虚拟化方法 [1], 通过修改客户操作系统源码, 避免虚拟化带来的开销. 在内存虚拟化方面, Xen 通过修改客户操作系统中的页表访问相关的源码, 实现了在页表项 (page table entry, PTE) 中直接存放实际的机器内存页面号, 方便 CPU 的内存管理单元 (memory management unit, MMU) 直接把虚拟内存地址转换为机器内存地址. 在操作系统进行页面管理时, 其任何对 PTE 的读取操作, 其修改后的代码都会通过访问机器内存页面号到物理内存页面号的转换表, 返回包含物理页面号的页表项；任何对页表项的写操作, 由于所有的页表页都被 VMM 标记为只读, 因而可通过页面中断 (page fault) 进入 VMM 中, 由 VMM 进行页面管理. 因为半虚拟化方法仅在有限的读写页表页操作上引入了非常有限的开销, 所以半虚拟化方法能达到较高的性能. 但半虚拟化方法需要修改客户操作系统的源码, 不适用于无法得到源码的操作系统.

只有全虚拟化方法才能让诸如 Windows 这类无法得到源码的操作系统作为客户操作系统使用. 在全虚拟化方法中, 通常采用影子页表 (shadow page table) 机制实现内存虚拟化 [3]. VMM 通过为客户操作系统中常被使用的页表页在 VMM 中维护一套对应的影子页表, 影子页表的页表项中存放对应的机器内存页面号, CPU 的内存管理单元通过影子页表完成从虚拟内存地址到机器内存地址的转换. 客户操作系统中的页表仅被其自身访问用于在客户物理地址空间上管理内存. VMM 通过把客户操作系统中的页表页标记为只读, 从而可以捕获所有对页表项的写操作, 进而可以同步地更新影子页表中的页表项. 同时, VMM 对影子页表项的标志位进行特殊的设置, 以实现当 CPU 需要修改影子页表的访问 (access) 标志位和写 (dirty) 标志位时, 能够即时对客户操作系统中原始页表项 (原始 PTE) 也做同样的修改. 这种频繁的同步操作使得采用全虚拟化方法的系统性能明显低于采用半虚拟化方法的系统.

3.2 动态内存半虚拟化

3.2.1 新内存虚拟化机制

动态内存半虚拟化的设计目标是避免全虚拟化中影子页表同步的开销. 为此我们引入了一种新的内存虚拟化机制, 它允许在不修改客户操作系统的源代码的情况下, 把客户操作系统页表项中的客户物理内存页面号替换成机器内存页面号, 实现从虚拟内存地址到机器内存地址的映射. 在 VMM 中可不再为客户操作系统维护相应的影子页表, 避免了影子页表与客户操作系统页表间同步的开销. 我们把在客户操作系统中那些存放着机器页面号的页表称为直接页表 (direct page table), 把使用直接页表的进程称为半虚拟化进程 (para-virtualized process). 在半虚拟化进程运行时, MMU 通过直接页表即可完成虚拟地址到机器地址的转换. 图 1 描述了典型的直接页表结构.

这里的难点问题在于, 如何让客户操作系统在管理半虚拟化进程的直接页表时, 仍然使用客户物理地址读写页表项. 为此, 我们把直接页表结构中所有指向页表页或页目录页的页表项转换为具有特殊格式的受保护的页表项 (protected PTE, P-PTE), 其中 P-PTE 的存在位 (present bit) 被清零. 因此访问页表页或页目录页的访问操作, 因为 MMU 转换虚拟地址到机器地址总会使用某个 P-PTE, 因而

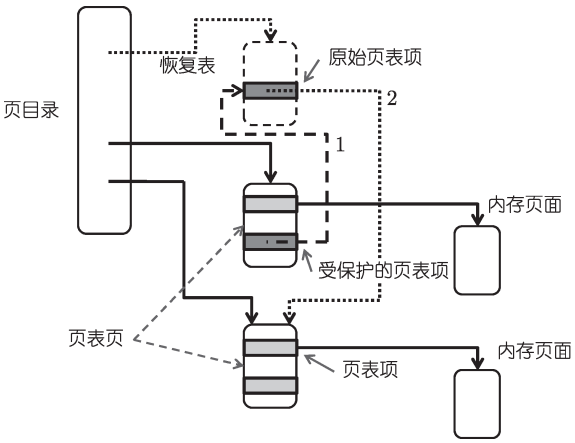

图 1 直接页表结构

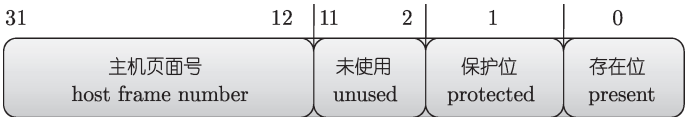

图 2 G2H 表的表项结构

会发生页面中断 (page fault) 进而陷入到 VMM 中来. P-PTE 的状态位具有特殊的模式, 以便能够把 P-PTE 和正常的 PTE 区分开来. 在 3.2.2 小节中, 我们将介绍代码植入机制, 以避免发生页面中断时陷入到 VMM 中.

客户操作系统访问页表项, 还需要知道客户物理地址到主机物理地址之间的转换关系. 在虚拟机启动时, 我们在 VMM 内部建立起从客户到主机 (guest to host, G2H) 和从主机到客户 (host to guest, H2G) 间的内存页面号映射表. G2H 负责把客户内存页面号映射到主机内存页面号, 而 H2G 则做相反的映射. 这两个映射表基本上是静态的, 除非 VMM 需要调整分配给虚拟机的主机物理内存时才可能发生变化. 植入到客户操作系统中的代码可以通过访问链入到客户操作系统地址空间中的 G2H 和 H2G 表, 完成客户和主机间的双向内存地址转换, 从而避免陷人到 VMM 中. 关于植入的具体细节将在 3.2.2 小节中介绍.

对于 32 位的操作系统, 我们简化了 G2H 和 H2G 表的设计, 采用一对一的映射方式, 每个方向的映射最多需要 1 M 个 4 字节表项. 对于 64 位操作系统, 我们将考虑设计一种多级的页表结构, 以减少两个映射表所占用的内存. 图 2 描述了对 32 位操作系统设计的 G2H 的表项结构 (G2H table entry). G2H 表项可通过客户页面号索引访问, G2H 页面项中高 20 位记录的是主机页面号 (host frame number); 第 0 位作为存在位 (present), 表明客户页面是否对应着一个主机页面, 如果没有相对应的主机页面, 存在位就清零; 第 1 位作为保护位 (protected), 说明对应的页面是否被保护起来, 不允许客户操作系统中被未修改的代码直接访问. 如果保护位为 1, 则表明对应的页面被用作页表页或页目录页, 对受保护页面的访问需要通过 P-PTE 完成.

图 3 描述了 H2G 表的表项结构 (H2G table entry). 其高 20 位记录的是主机页面号对应的客户页面号 (guest frame number). H2G 表也包含有保护位和存在位, 保护位用于说明一个主机页面被用作客户操作系统中的页面页或页目录页, 因而被保护起来不会被客户操作系统直接访问; 存在位则表

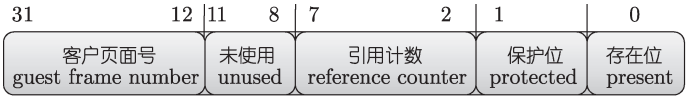

图 3　H2G 表的表项结构

明主机页面是否对应着客户页面. H2G 表项中的第 2~7 位用于记录页面引用计数, 我们将在本节的后边介绍引用计数的作用.

在读取直接页表中的页目录项 (page directory entry, PDE) 和普通页表项 (PTE) 时, VMM 通过查找 H2G 表返回客操作系统可直接使用的 PDE 和 PTE 值. 在读取受保护的页表项 (P-PTE) 时, 需要多通过一级转换, 这将在 3.2.3 小节中详细介绍. 更新直接页表的操作要更复杂. 当客户操作系统要写入新 PTE 值时, 如果写入的 PTE 实际上指向的是一个直接页表页, 则需要把新 PTE 的值转换为对应的 P-PTE 的值后写入. 当客户操作系统写入新 PDE 时, 根据新旧 PDE 是否存在, 分为四种情形处理. 如果旧 PDE 存在位为 0, 则新 PDE 的值可直接写入. 但如果旧 PDE 存在位为 1, 则写入新 PDE 值前需要先把旧 PDE 对应的页表页的在 H2G 中的引用计数减 1. 如果页面的引用计数为 0, 也就是说不存在任何半虚拟化进程使用该页面作为页表页时, 则对应的页面将不再需要保护. H2G 中的引用计数表明了在所有半虚拟化进程的直接页表中, 共有多少个 PDE 引用到该页表作为页表页. 页表页在多个进程间共享一般是在进程创建子进程时, 子进程的页表从父进程中通过写时复制 (copy on write, CoW) 的方式直接继承过来而形成的.

- 如果旧 PDE 所指页面的引用计数值为 1, 则当前进程是最后一个在直接页表中使用该页表的进程. 在用新 PDE 替换了旧 PDE 后, 这个页面就不再是需要保护的页表页, 因而需要把它恢复为普通页面以便于正常访问. 页面中所记录的主机页面号必须被转换为原来的客户页面号, 因为这个页面还可能被其他非半虚拟化进程用作页表页. 指向该页面的 P-PTE 也同时被还原为普通 PTE, 具体策略见后面 3.2.2 小节中的讨论.
- 如果引用计数大于 1, 也就是说还存在半虚拟化进程在其直接页表结构中使用着该页表, 我们可简单地把其引用计数减 1 即可.

如果新 PDE 指向一个新的还未被任何半虚拟化进程用作页表页的页面, 即还未被保护的页面, 那么就需要首先对该页面做保护: 先在 G2H 和 H2G 表中标记该页面被保护, 同时还要把任何引用该页面的页表项 (PTE) 转换为相应的 P-PTE. 在把 PTE 转换为 P-PTE 时, PTE 的原始内容被保存在一个称为恢复表 (recovery table, RT) 的特殊的页表页中, 而转换得到的 P-PTE 中记录了在恢复表中找到原始 PTE 表项的指针. 我们将在 3.2.3 小节中介绍恢复表的作用. 如果新 PDE 指向一个已经被保护的页表页, 说明这个页表页已经被其他进程用作其页表页了, 则只需要把 H2G 表中的页面引用计数增加 1.

当客户操作系统写入 PTE 时, 我们首先检查新旧 PTE 是否包含相同的主机页面号. 如果都包含, 则只需要修改状态位即可. 要注意, 旧的 PTE 可能就是一个 P-PTE! 在这种情况下, 主机页面号是存放在恢复表的原始 PTE 中, 对其修改也是发生在原始 PTE 中. 如果新旧 PTE 的主机页面号不同, 或者旧 PTE 是一个普通的 PTE, 再或者旧 PTE 不存在, 则旧 PTE 可被直接覆盖. 即使旧 PTE 是一个 P-PTE, 因它所指向的受保护页已经被 PDE 所维护 (包括相应的引用计数), 所以也可被直接覆盖, 而无需任何附加操作. 如果新的 PTE 指向一个普通的页 (不是一个直接页表页), 新 PTE 的内容可用于直接覆盖旧 PTE. 否则的话, 我们就需要保护新 PTE 所指向的页 (如果它还没有被保护). 我们用相应的 P-PTE 覆盖旧 PTE, 并将新 PTE 的内容保存在恢复表中.

3.2.2 可适配的代码植入

由于直接页表的保护机制, 在客户操作系统中任何对页表页的访问都会导致其陷出到 VMM, 频繁的陷出到 VMM 会降低虚拟化的性能, 甚至于不及全虚拟化模式的性能. 我们引入可适配的代码植入 (adaptive code implantation) 以避免由于引入新的内存虚拟化机制而导致的过多的陷出, 从而达到高性能. 通过动态检测导致频繁陷出到 VMM 的指令, 并用植入的代码替换这些指令, 就可避免频繁的陷出.

由于传统的内存全虚拟化机制 (影子页表) 在面对内存密集型应用时, 常常需要频繁地同步影子页表, 而同步必须陷入到 VMM 中来完成, 在这种情形下, 无法通过生成替换代码来减少 VM 陷出. 但是, 在动态内存半虚拟化中, 在新的内存虚拟化机制 (直接页表) 下, 我们特别关注那些导致缺页或导致页保护错误的热点指令, 用于替换这些热点指令的植入代码可以利用 G2H 和 H2G 表以及新的内存管理机制, 在客户操作系统中, 不经陷出到 VMM, 就可完成对页表的透明管理. 在这里, 我们对 Guest OS 透明的修改体现在两个方面: 一是引入了新的内存虚拟化机制 (直接页表), 二是植入了新的替换热点的代码.

因为导致 VM 陷出的热点指令的多样性, 用户替换热点指令的代码也大小不一. 如果直接对代码做本地替换, 原来代码空间内一般无法容下替换后的代码, 并且一旦做了代码替换, 还可能对原代码所在函数内所有代码的布局产生影响, 从而引起原函数代码复杂的变化. 为了避免本地代码替换带来的复杂性, 我们采用了一种以函数为基本替换单元的代码替换方法. 首先, 根据热点代码的有限性, 我们生成一个替换代码库, 用于客户操作系统中按新的模式模拟被替换的热点的执行. 然后, 在找到一个热点指令后, 根据调用栈分析找到这个代码所在的函数及其被调用关系, 并基于这个函数的全部代码和热点替换代码生成一个新的函数. 新的函数被放入到客户操作系统的地址空间中, 然后在把所有识别出来的对热点所在原始函数的调用点都替换为对新植入函数的调用. 将来需要取消替换时, 可以简单地把修改的调用点还原即可 (一般来说, 修改调用点不会引起函数调用指令代码长度的变化).

3.2.3 恢复表

我们为每个进程维护一个恢复表 (recovery table, RT). 恢复表也是进程空间中的一个页表. 在 P-PTE 中保存有记录其原始 PTE 的恢复表的序号和表项编号, 因而在知道 P-PTE 的值时, 就可以构造出通过恢复表中的原始 PTE 来访问页表页的虚拟地址, 即可绕过 P-PTE 完成对页表页的读写操作. 当我们植入到客户操作系统中的代码来完成此操作时, 就可不必陷出到 VMM, 并且读写操作会通过 MMU 完成对原始 PTE 的 Dirty 位和 Access 位的更新, 而植入的代码也知道如何根据 G2H 写入正确的值到页表页, 根据 H2G 从页表页读出正确的值.

恢复表允许我们进行进程级的恢复 (process-level recovery) 或虚拟机级的恢复 (OS-level recovery). 在我们的设计中, 每个恢复表都需要占用一个内核空间的 PDE, 而我们可以从客户操作系统中借用的 PDE 是有限的. 因此, 只有有限数目的进程可同时被半虚拟化. 当需要半虚拟化的进程数量超过限制时, 我们就需要做进程级别的恢复, 把通过 LRU 方式探测出来的最不常用的进程恢复为普通进程. 恢复的过程涉及到根据 G2H 和 H2G 表把直接页表页中的主机页面号还原回客户页面号, 并对 P-PTE 通过恢复表中的原始 PTE 进行还原, 从而释放出其占用的恢复表给新的半虚拟化进程使用.

我们保留了客户操作系统的部分地址空间用于维护恢复表、G2H 表、H2G 表以及植入代码地址空间. 尽管客户操作系统几乎不会使用这部分地址空间, 但这部分地址空间仍然可能被客户操作系统访问到. 一旦如此, 就需要做虚拟机级别的完全恢复. 恢复包括把每个半虚拟化进程进行恢复和恢复

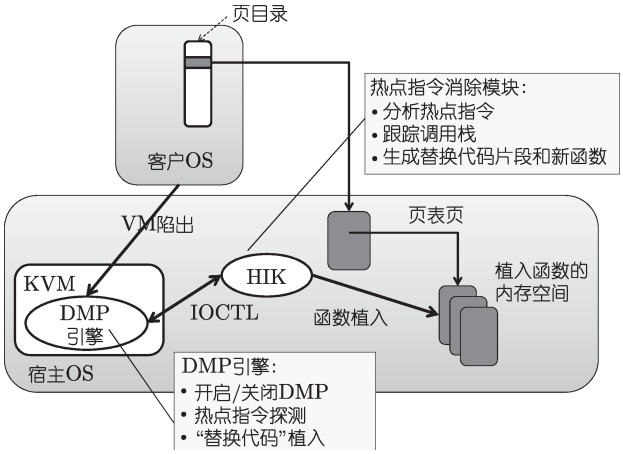

图 4　DMP 实现的结构

所有被修改的调用点.

3.3　DMP 在 KVM 中的实现

3.3.1　DMP 实现的结构

以 32 位 Linux 为 Guest OS, 我们在 KVM-54 的支持 VT 的全虚拟化版本上实现了一个内存动态半虚拟化原型系统. 我们在 Host OS(宿主 OS) 中加入了一个用户态模块 HIK(hot instructions killer, 热点指令消除模块), 在 KVM(kernel based virtual machine) 中加入了一个内核模块 DMP Engine(DMP 引擎, 如图 4 所示). HIK 负责发起热点探测、分析热点函数及其函数调用关系、生成热点替代代码、生成并保存替换函数、发起代码植入及取消等工作; DMP Engine 则实现了直接页表机制、热点探测和代码植入等功能.

HIK 和 DMP Engine 之间通过 IOCTL 进行交互, 它们互相互配合完成内存的动态半虚拟化, 具体的流程为:

最初, 当 KVM 的 VM 启动并进入页保护模式后, HIK 通知 DMP Engine 开启直接页表访问模式, 并开始探测热点. 此时 DMP Engine 会对陷出点进行统计, 找出热点. DMP Engine 并不处理这些热点, 而是由 HIK 主动向 DMP Engine 询问当前热点统计状况. 当 HIK 从 DMP Engine 获得热点列表后, 会根据热点陷出原因找出与缺页或导致页保护错误相关的热点, 对于这些热点, HIK 会要求 DMP Engine 对其函数调用栈进行捕捉. 通过函数调用栈, HIK 可以分析出热点所在的函数及对该函数的调用点, 并将该分析结果返回 DMP Engine.

随后, HIK 从 DMP Engine 处获得热点函数的二进制代码, 根据热点消除模板库生成热点替换代码, 并对热点函数进行去热点改造, 生成一个不会产生或减少了 VM 陷出的新函数. HIK 会在自己的进程空间中分配 4 MB 对齐的地址空间, 并把新函数写入这个空间. HIK 会通过 DMP Engine 把这 4 MB 对齐的地址空间对应的内存页面挂接到 Guest OS 的地址空间中, 使得 Guest OS 可以访问该新函数.

新函数可以被 Guest OS 访问后, HIK 会通知 DMP Engine 对相关已探知的调用点进行修改, 把新函数的地址应用到这些调用点. 并且在 DMP Engine 中通过一个散列表记录下来所有被替换过的调用点及其原来的函数地址, 从而在需要恢复时能一次性取消所有对调用点的修改.

在一个热点经过上面的步骤消除后, DMP Engine 对热点的统计需要重新开始, 以便寻找新的热点. 根据消除热点程序的需求, 这个过程可往复多遍, 以消除绝大多数可被消除的热点.

进一步, 我们还可以实现对热点消除过程的记录: 针对一个特定的虚拟机应用实例, 其热点消除过程可以被记录并保存到文件中, 在下次运行虚拟机时, 可统一应用已经保存下来的热点消除动作, 从而避免系统预热过程.

3.3.2 针对代码植入的特殊内存管理方式

为了简化原型系统的实现, 我们实现了一些特定的简化和优化机制.

在 VM 启动时, 我们会遍历分配给 VM 的内存, 并通过增加页表计数的方式把分配给 VM 的页面固定下来, 这样, G2H 和 H2G 可在内存分配完成后创建出来并在整个 VM 的生命期内不再变化.

针对 32 位 Linux 操作系统, 我们限定分配给虚拟机的内存不超过 768 MB, 这样用于直接内存地址映射的 PDE 不会填满 PD 中供内核使用的 PDE, 我们总可以在内核使用的 PDE 区间中找到多个空闲的 PDE, 这些 PDE 被用来把 G2H、H2G、植入函数地址空间、恢复表和 loop-PDE 链接到当前进程中. loop-PDE 是一个特殊的 PDE, 植入函数可以通过它去访问 P-PTE 和恢复表.

为了简化实现, 我们让每个进程都有一个恢复表, 而每个恢复表都会消耗一个 PDE, 因此当前同时存在的半虚拟化进程的数目就非常有限, 一般只有 10 到 20 个. 但在实际应用中, 比较活跃的进程一般都不超过 10 个. 在我们的实验中, DMP 的工作情况良好.

我们在 DMP Engine 中使用了一个 LRU(least recently used) 队列来维护当前的半虚拟化进程. 当有一个新的进程被调度执行, 而没有空闲的恢复表时, 通过 LRU 机制, 我们将淘汰一个或多个不活跃的半虚拟化进程, 进而空闲出恢复表给新的进程使用.

由于 Linux 内核空间是被所有进程共享的, 当一个进程被从半虚拟化状态恢复到原始状态时, 其用于内核地址空间的页表页是不需要被恢复的. 我们通过给内核空间分配了一个特殊的恢复表, 用于辅助植入函数访问内核页表页, 以及当需要做虚拟机级恢复时的恢复.

我们在 DMP engine 中还使用了一个全局变量来控制启用或禁止直接页表机制. 当外部希望开启直接页表机制时, 就把全局变量置 1, 反之则置 0.

当启动直接页表机制后, Guest OS 会因读写页表页而导致缺页或页保护错误, 并因此陷出到 KVM, 由我们提供的直接页表访问的模拟方式进行模拟执行. 由于直接页表访问模式不仅在 Guest OS 写页表时会陷出, 在读取页表时也会发生陷出, 因此, 直接页表模式下的陷出会更多, 产生更多的热点. 但这些热点可以在随后被有效地消除, 从而提高虚拟化系统的性能, 这正是内存动态半虚拟化的效果所在.

3.3.3 植入代码片断示例

以一条写 PTE 的指令 “mov %ebx, (%ebp)” 为例. 这条指令的含义为: 将 ebx 寄存器的值写入 ebp 寄存器指向的 PTE. 我们用下面的代码片段替换该指令, 替换代码的执行效果和被替换的指令执行完后完全相同, 但不再会产生 VM 陷出.

1. push %eax
2. push %edx
3. mov %ebx, %edx
4. lea (%ebp), %eax

5. call rel32 0xfa0011e0; 调用植入的函数写 PTE
6. cmp $0x0, %eax
7. pop %edx
8. pop %eax
9. jcc rel32 0xfa010227; 跳到代码片段尾部
10. mov %ebx, (%ebp)

首先把 eax 和 edx 寄存器压栈 (第 1, 2 条指令), 并把 ebx 寄存器的值和 ebp 寄存器指向的 PTE 的地址值分别赋值给 edx 和 eax (第 3, 4 条指令), edx 和 eax 是函数 0xfa0011e0 的传入参数, 被调用的函数 0xfa0011e0 将根据前面所讲的机制完成可在 GuestOS 中直接完成的写操作 (第 5 条指令). 函数返回值通过 eax 寄存器传出, 通过比较返回值是否为 0 (第 6 条指令), 可以判断函数内是否完成了对指令的模拟. 在通过栈先恢复了 edx 和 eax 寄存器的值后 (第 7, 8 条指令), 根据前面判断结果, 如果函数体内已经完成的对指令的模拟, 则整个模拟过程结束, 直接跳到片段结束位置 (第 9 条指令); 否则, 执行原始指令以便通过 PageFault 陷出到 VMM 中去模拟写 PTE.

4 评测

我们在 KVM-54 上实现了一个 DMP 原型. 我们的实验环境为: Intel PC 工作站, 2 GB 内存, CPU 为支持 VT-x 的 Intel 酷睿 2 双核 1.86 GHz. 客户操作系统和裸机 (native) 操作系统都选用 Red Hat Enterprise Linux 5 (内核版本 2.6.18-8.el5). 加载 KVM 模块的主机操作系统为 Red Hat Enterprise Linux 5 (内核版本 2.6.24.3). 如果没有特别说明, 参与对比测试的客户操作系统和裸机操作系统都配置为使用 512 MB 物理内存.

我们选用了 SpecJBB 2005, SpecInt 2006, SpecWeb 2005, Kernel Compile 和 WorstCase(一个我们自己编写的 micro program) 测试程序作为测试基准. Kernel Compile 为编译最小化的 Linux2.6.18 内核. WorstCase 是测试一个极端的内存访问情况, 它顺序访问 100 K 个页面, 对每个页面中仅写入一个 4 字节的整数. WorstCase 用于测试裸机和虚拟机在处理缺页中断上的性能差别, 因为 WorstCase 程序平均每次访问内存操作都会引起一个缺页中断. 我们测试了所有 SpecCPU 2006 中整数测试基准程序 (除了 mcf, 因为 mcf 在 KVM-54 上无法完成). 在 SpecCPU 2006 的测试中, 一个 VM 配置的物理内存大小为 752 MB, 这是我们目前所实现的 DMP 版本对 32 位操作系统所能支持的最大物理内存. SpecCPU 2006 一般要求配置 1 GB 内存, 我们尽可能的提高 VM 内存配置以减少页面交换对测试结果的影响. SpecWeb 在完成初始化后, 只会引起很少的页面中断, DMP 对 SpecWeb 的影响是微乎其微的. 因此在后面的实验结果中, 我们省略了对 SpecWeb 测试的结果介绍. 对所有的基准测试, 我们都消除了因页面中断导致陷出最多的前 18 个热点指令.

4.1 总体性能评测

图 5 以规格化的方式对比了所有基准测试程序的运行时间 (其中 JBB 是个例外, 因为 JBB 是通过 Spec 评分来衡量性能). 图中显示, DMP 能够显著降低有大量缺页中断的程序的虚拟化开销, 提高其运行速度, 这与表 1 中关于 Page Fault 所占比例关系的结论是一致的. 对于 Kernel Compile, DMP 比 KVM 提高了 32% 的性能, 基本上达到了裸机 (native) 性能的 91%. 对于 SpecJBB, 正如表 2 所示, 多数热点指令都是 I/O 指令, 因而 DMP 对 SpecJBB 的性能提升不显著, 但也有 3.2% 的性能提升.

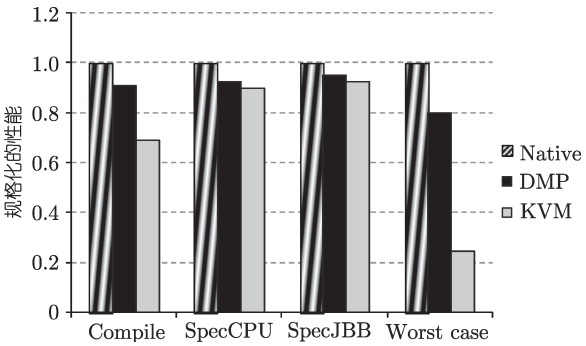

图 5 总体性能评测

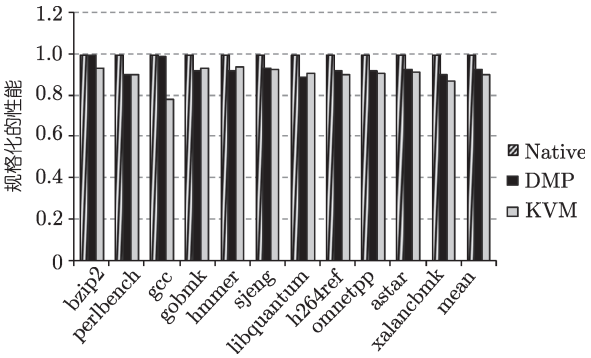

图 6 SpecINT 2006 的性能评测

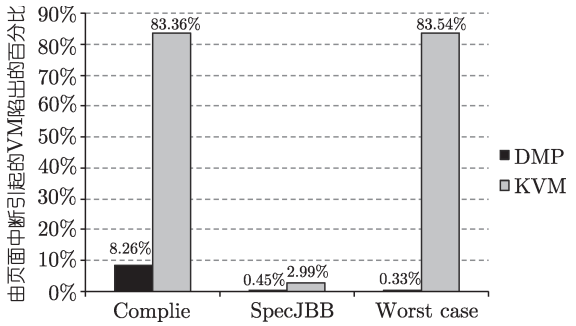

图 7 由页面中断引起的 VM 陷出的百分比

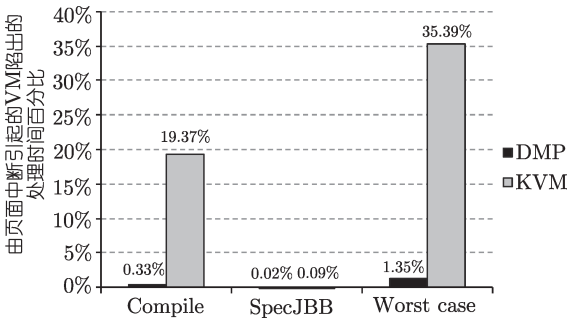

图 8 由页面中断引起的 VM 陷出的处理时间百分比

对于 WorstCase, DMP 的性能达到了裸机性能的 80%, 并且比 KVM 的性能提升了 1 倍还多. 与裸机 20% 的性能差距主要是 VMM 处理时钟中断而带来的性能开销.

平均来说, DMP 相对 KVM 对 SpecCPU 整数测试基准程序有 3.1% 的性能提升. 除了 3 个特别的测试基准程序外, DMP 提升了所有其他测试基准程序的性能. 性能提升最显著的为 gcc 和 bzip2, 分别有 26.2% 和 6.8% 的性能提升, 如图 6 所示. 对于 gobmk、hmmer 和 libquantum, DMP 比 KVM 性能要差 1.5%, 这主要是因为这 3 个基准测试程序在运行过程中很少产生缺页中断. 1.5% 的性能下降主要来自于 DMP 的自身的性能开销.

4.2 内存虚拟化开销

本节分析 KVM 内存虚拟化的开销, 并与 DMP 做对比. 图 7 表示的是各组基准测试程序在 VM 中运行时产生的缺页中断的占总陷出次数的百分比. 这里没有考虑与 SpecInt 中的 gcc 做对比, 因为它本质上和 Kernel Compile 是类似的. DMP 可以把缺页中断导致的陷出次数降到占总陷出次数的 10% 以下, 消除的页面中断陷出率大于 78%, 甚至接近于 100%. 剩余的页面中断陷出, 一部分来自于我们未替换的调用点, 另一部分来自于对页目录页的访问 (这部分访问对于目前的设计而言是不能消除的).

进一步, 我们统计了在 KVM 中处理页面中断陷出所用时间的总和. 这里的时间总和不包括陷出或重入虚拟机的 VM 与 VMM 之间切换所用的时间. 图 8 对比了陷出处理时间占总运行时间的比例. 在 KVM 中, 对 Kernel Compile 而言, 处理页面中断陷出大约消耗了 20% 的总执行时间, 而 WorstCase 则占约 35%. DMP 可以把相应的开销减少到只占总执行时间的 1.5% 以下.

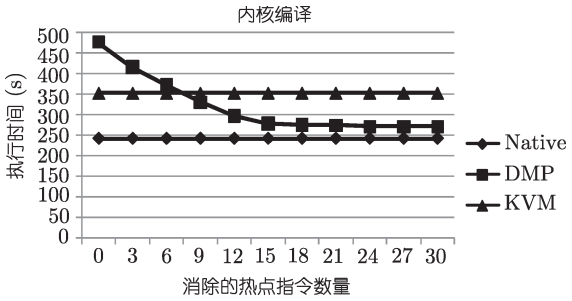

图 9　编译内核的性能随消除热点数的变化

4.3　热点的消除及其性能变化

图 9 通过编译内核所用时间, 说明了消除热点数量与 DMP 性能优化之间的关系. 在 DMP 还未消除任何热点时, 其性能远不如 KVM 和裸机好. 但是随着热点被逐一消除, DMP 的性能迅速接近并超过 KVM 的性能, 当消除的热点数为 8 个时, DMP 的性能与 KVM 相当；当消除到 15 个热点时, DMP 的性能已接近裸机性能. 在更多热点指令被消除时, DMP 的性能已不再有显著的提升. 这个实验表明, 热点指令中的很少一部分对虚拟化的性能产生了主要的影响, 消除这些热点即可明显提升虚拟化性能.

5　相关工作

我们提出的内存动态半虚拟化机制与 Xen 所采用的内存半虚拟化机制 [1] 有很多相似之处. Xen 一般采用直接页表方式, 通过修改客户操作系统完成对页面的管理, 直接维护页表中的机器地址, 从而避免影子页表机制开销较高的问题 [3,10]. Xen 上的客户操作系统在读取页表项时, 通过机器页面号到物理页面号的映射表, 把机器页面号转换回物理页面号; 在写页表项时, 通过页面写保护, 使其陷入到 VMM 中, 在 VMM 的帮助下完成写操作. 我们的方式则是在全虚拟化模式下把影子页表机制动态地转换为半虚拟化直接页表机制.

在静态编译优化的指令选择阶段, 这种用一个或一组开销小的指令替换一个开销大的指令的方法是一种较通用的方法 [11]. 在二进制代码翻译和即时编译中也经常采用这种方法 [9,12,13]. VMWare 的产品采用适应性的二进制代码翻译 (adaptive binary translation) 避免特权和非特权指令导致的陷出 [9]. 在 Java 虚拟机的适应性即时编译技术 (adaptive just-in-time compilation) 中, 也是通过检测热点方法 (detect hot methods) 和选择性重编译这些方法 (selectively recompile them) 来提高 Java 应用的性能 [12].

近年来, Intel 提出了扩展页表 (extended page tables, EPT) 技术 [8], AMD 也提出了嵌入页表 (nested page tables, NPT) 技术 [6], 这两种硬件辅助虚拟化技术都能简化软件实现内存虚拟化管理的复杂性. 它们通过引入从客户机物理地址到主机物理间的转换, 使客户操作系统维护的从虚拟地址到物理地址的映射可以经过这层额外的地址转换而映射到主机物理地址. 虽然这种扩展能够显著的降低 VMM 在地址映射中的干预程度, 但这种双重页表机制使得遍历页表结构需要更多的访存操作, 进而会影响总体性能. Bhargava 等人在文献 [6] 中探讨了一些降低双重页表机制开销的方法. 我们提出的方法可以在没有 EPT 和 NPT 的支持下工作, 这种技术保持了原有的单重页表机制, 因而比 EPT 和 NPT 在遍历页表结构过程中访存操作更少.

6 总结及进一步的工作

降低虚拟化的性能开销仍然需要更有效的硬件辅助机制的出现. 我们相信将来的硬件辅助虚拟化技术会更多的借鉴半虚拟化技术中成功的方法. 本文提出的动态半虚拟化方法, 使得一些半虚拟化的方法可以应用于全虚拟化的环境中. 我们可以在保持全虚拟化环境以支持遗产操作系统的同时, 获得半虚拟化技术带来的性能提升.

当然, 代码植入也会带来一定的安全隐患, 因为植入的代码被允许直接修改页表. 恶意代码可能会在入侵操作系统内核后通过这个修改页表的能力破坏整个系统. 在代码植入和热点消除技术的进一步成熟后, 我们将进一步研究如何避免引入这种安全隐患而达到相近的性能提升. 而目前, 我们假定 VMM 与虚拟机中的客户操作系统之间都是可信的. 在 VMM 或客户操作系统需要更高安全级别时, 我们可以利用恢复机制关闭动态半虚拟化.

虽然本文重点探讨了内存的动态半虚拟化问题, 但我们的动态半虚拟化方法同样也对 I/O 操作适用. CPU 通过 I/O 指令和 MMIO 访问进行的 I/O 操作同样会带来大量的 VM 陷出, 进一步, 我们计划采用动态代码植入的方法来消除这些陷出, 提高 I/O 虚拟化的性能.

参考文献

1 Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP '03. New York: ACM, 2003. 164–177

2 Whitaker A, Shaw M, Gribble S D. Scale and performance in the Denali isolation kernel. SIGOPS Oper Syst, 2002, 36(SI): 195–209

3 Devine S, Bugnion E, Rosenblum M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent, 6397242, 1998-10

4 Smith J E, Nair R. Virtual Machines: Versatile Platforms for Systems and Processes. San Francisco: Morgan Kaufmann Publishers, 2005. 382–396

5 AMD. AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual. 2005

6 Bhargava R, Serebrin B, Spadini F, et al. Accelerating two-dimensional page walks for virtualized systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII. New York: ACM, 2008. 26–35

7 Intel Corp. Intel Virtualization Technology Specification for the IA-32 Intel Architecture. 2005

8 Neiger G, Santoni A, Leung F, et al. Intel virtualization technology: hardware support for efficient processor virtualization. Intel Technol J, 2006, 10: 167–177

9 Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII. New York: ACM, 2006. 2–13

10 Habib I. Virtualization with KVM. Linux J, 2008, 166: 8

11 Cooper K, Torczon L. Engineering a Compiler. San Francisco: Morgan Kaufmann, 2003. 382–432

12 Arnold M, Fink S, Grove D, et al. Adaptive optimization in the Jalapeno JVM. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00. New York: ACM, 2000. 47–65

13 Cramer T, Friedman R, Miller T, et al. Compiling java just in time. In: IEEE Micro 17. Los Alamitos: IEEE Computer Society Press, 1997. 36–43