中国科学: 信息科学 2010 年 第 40 卷 第 12 期: 1543 ~ 1558
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS

论 文

DMM: 虚拟机的动态内存映射模型

陈昊罡[①], 汪小林[①*], 王振林[②], 张彬彬[①], 罗英伟[①*], 李晓明[①]

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

收稿日期: 2009–02–01; 接受日期: 2010–01–15
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) 和教育部 - 英特尔信息技术专项科研基金 (批准号: MOE-INTEL-08-09, MOE-INTEL-10-06) 资助项目

摘要 内存虚拟化方法一直是虚拟机管理器设计中最重要的部分. 文中提出了 VMM 进行内存管理的一种机制: 虚拟机 (VM) 的动态内存映射模型, 它允许 VMM 在虚拟机运行时, 动态地改变它的物理内存与机器物理内存的映射关系. 利用 DMM, VMM 向上能够方便地实现按需取页、页面交换、Ballooning、内存共享、copy-on-write 等虚拟机高级内存管理技术, 向下能够兼容多种虚拟化架构. 它所提供的一种模块化的分层体系结构, 能有效地将上层的内存管理策略和底层的内存虚拟化实现很好地融合起来, 为实现特征可调的内存管理提供了可能. 文中给出了动态内存映射模型的基本原理, 并阐述了利用该模型, 实现各种虚拟机内存管理技术的相应机制和步骤. 同时, 在一个开源的虚拟机管理器 (KVM) 上实现了动态内存映射机制. 测试表明, 该机制具有良好的灵活性和可扩展性, 能够在充分保证虚拟机访问内存的性能的前提下, 实现虚拟机内存的动态管理和调配.

关键词
虚拟机管理器
Xen
虚拟机
内存虚拟化
动态内存映射

1 相关工作及问题的提出

内存是虚拟机最频繁访问的设备之一, 内存虚拟化的效率将对虚拟机的性能产生重大影响. 现代计算机通常都采用段页式存储管理、多级页表等复杂的存储体系结构, 这又给虚拟机管理器 (virtual machine monitor, VMM) 的高性能内存虚拟化设计带来了很大挑战.

当需要在同一物理主机上同时部署多个虚拟机时, VMM 能否提供可伸缩的内存管理功能就显得尤为重要. 这是因为 VMM 需要实现物理内存在虚拟机之间的分割复用, 如果这种分割是静态的, 则一台物理主机上所能并发执行的虚拟机数量必然受到实际硬件的机器内存大小的限制. 同时, 由于虚拟机上运行的软件对内存的需求各不相同, 而且是动态变化的, 基于静态分割的内存管理机制必然会造成内存资源的不合理分配, 从而大大影响虚拟机执行的性能. 为了使 VMM 系统具有更好的伸缩性和可扩展性, 我们希望 VMM 能够提供一种机制, 使得能够在充分保证虚拟机访问内存的性能的前提下, 实现虚拟机内存的动态管理和调配. 例如, 理想的 VMM 应该提供以下一些内存管理功能 [1]:

引用格式: 陈昊罡, 汪小林, 王振林, 等. DMM: 虚拟机的动态内存映射模型. 中国科学: 信息科学, 2010, 40: 1543–1558

(1) 按需取页. 只有当虚拟机真正需要的时候, VMM 才将物理内存分配给它, 而不是简单地将固定大小的内存空间划分给虚拟机. 按需取页能够提高内存资源的利用率.

(2) 虚拟存储. VMM 应该能够利用交换等技术, 给虚拟机提供超过实际机器内存大小的内存空间. 虚拟机上的 Guest OS 能够象运行在裸机上一样, 透明地使用 VMM 提供的整个 “物理内存”.

(3) 内存共享. VMM 应该允许虚拟机之间只读地共享完全相同的内存区域, 从而缓解大量虚拟机并发运行时的内存资源紧缺. 内存共享是内存 copy-on-write 机制的重要基础. 当前的 VMM 分别实现了以上部分管理功能. 如开源虚拟机管理器 Xen 实现了基于 Ballooning 的虚拟存储技术 [2~4], 即允许 VMM 从其他虚拟机窃取一些未使用机器内存页面, 给急需内存的虚拟机使用; VMWare 公司的 VMWare Workstation 实现了基于交换的虚拟存储技术 [5], 即允许将虚拟机的部分物理内存页面交换到宿主操作系统 (host OS) 的交换磁盘分区上; 而 VMWare ESX Server 还实现了基于页面内容比较的虚拟机间内存共享技术 [1].

但是, 由于各种内存管理功能的底层的实现机制是相互独立的, 现有 VMM 中的这些内存管理机制具有以下缺点:

第一, 机制的可扩展性不强. 受到开发周期的影响, 上述 VMM 在实现不同的内存管理功能时, 分别引入了不同的底层支持机制, 导致内存管理模块日益复杂而难以管理, 限制了系统的可扩展性. 例如, 为了实现 Ballooning 功能, Xen 引入了 Grant Table 机制 [6], 但是该机制并不适用于按需取页、内存交换和内存共享, 因而难以继续加入这些内存管理功能.

第二, 完整性和耦合度不高. 现有 VMM 大都未能实现上述所有的内存管理功能. 而即使是已实现的功能, 也都分别采用独立的模块设计. 由于模块所用到的底层机制是相互独立的, 这些模块之间难以有效地协同工作, 甚至是相互冲突的. 例如在 VMWare ESX Server 中, 内存共享机制是不能与交换机制同时启用的 [1]. 而在实际应用中, 我们往往希望 VMM 能够综合地、并发地利用所有可能的内存管理技术, 实现资源利用率的最大化.

第三, 兼容性和可维护性差. 现有 VMM 中许多内存管理功能的实现都依赖于某些特定的系统体系结构, 如半虚拟化、影子页表或硬件辅助虚拟化. 这种平台相关性使得我们在增改某种内存管理功能、或将现有功能移植到新的硬件及操作系统平台上时, 常常牵一发而动全身. 不但耗时费力, 还容易留下错误及安全隐患. 为了解决现有 VMM 系统中的内存管理机制相互独立所带来的代码可维护性差、可扩展性差、耦合度低等难题, 需要有一套通用的、高效的、可扩展的虚拟机内存管理机制. 为此, 本文提出了虚拟机的动态内存映射 (dynamic memory mapping, DMM) 模型. DMM 是 VMM 进行内存管理的一种底层机制, 它允许 VMM 在虚拟机运行时, 动态地改变虚拟机的物理内存与真实硬件的机器内存的对应关系, 从而能够向上很好地支持 VMM 实现按需取页、虚拟存储和内存共享等内存管理功能, 并向下兼容多种内存虚拟化的系统结构.

图 1 显示了 DMM 所提供的模块化的分层体系结构. 动态内存映射模型与底层的硬件和实现无关, 但为实现上层的管理功能提供了必要的、统一的机制. DMM 为实现特征可调的内存管理提供了可能: 一方面, 上层的内存管理策略可调; 另一方面, 底层的实现机制也可调. 也就是说, DMM 将上层的内存管理策略和底层的内存虚拟化实现很好地融合起来了.

在接下来的章节中, 第 2 节首先回顾虚拟机技术的理论基础, 以及当前进行内存虚拟化的基本思路和主要实现方法, 然后提出了 DMM 的理论模型, 并从理论上阐述了该模型对于按需取页、虚拟存储、内存共享等虚拟机高级内存管理技术的适用性, 形式化地给出了相应的方法步骤. 第 3 节介绍了 DMM 在 KVM 上的实现, 设计并实现了对该模型至关重要的页面池机制、违例陷入机制等; 并通过测

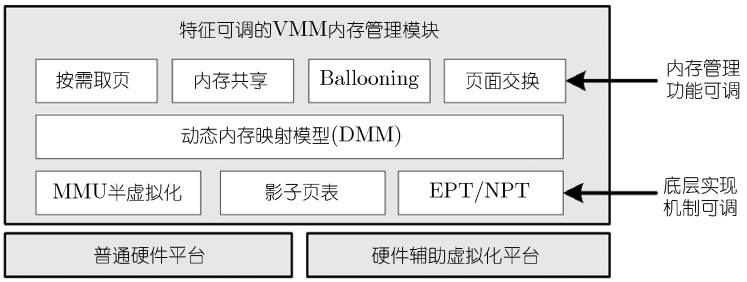

图 1 DMM 提供的模块化分层体系结构

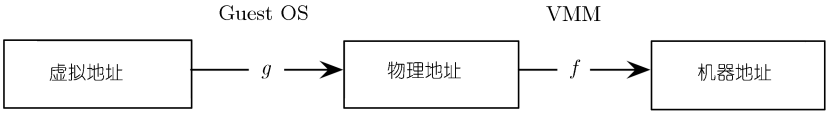

图 2 机器地址、物理地址和虚拟地址

试和分析得到了该机制的主要时间开销和空间开销. 第 4 节是总结.

2 动态内存映射的理论模型

VMM 通常采用分割复用的思想来虚拟计算机的物理内存. 也就是说, VMM 需要将机器的内存分配给各个虚拟机, 并维护机器内存和虚拟机所见到的 "物理内存" 的映射关系 (f-map), 使得这些内存在虚拟机看来是一段从地址 0 开始的、连续的物理地址空间. 本节我们将首先阐释一些基本概念, 然后讨论内存虚拟化中 f-map 的具体内容, 并给出动态内存映射模型的形式化定义, 最后我们将阐释动态内存映射模型对于第 1 节所提出的各种虚拟机内存管理功能的适用性.

2.1 研究基础

现代计算机通常都具备内存分页保护机制, 这给 VMM 进行内存虚拟化提供了必要硬件支持, 因为 VMM 能够以页面为单位建立 f-map , 并利用页面权限设置实现不同虚拟机间内存的隔离和保护. 但是, 由于 Guest OS 本身也会进行页式内存管理, 虚拟机系统中实际上存在着 3 个地址概念: (1) 机器地址 (machine address), 指真实硬件的物理地址, 即地址总线上应该出现的地址信号; (2) 物理地址 (guest physical address), 指经过 VMM 抽象的、虚拟机所看到的伪物理地址; (3) 虚拟地址 (guest virtual address), 指 Guest OS 提供给其应用程序使用的线性地址空间.

显然, VMM 通过 f-map 实现了 "机器地址" 到 "物理地址" 的映射, 我们将这个映射记为 f; 同时, Guest OS 的内存管理模块要完成 "虚拟地址" 到 "物理地址" 的映射, 我们将这个映射记为 g, 则机器地址、物理地址和虚拟地址的关系如图 2 所示.

可见在虚拟机环境下, 虚拟地址必须经过两次映射方能得到总线上使用的机器地址. 为了实现 "虚拟地址" 到 "机器地址" 的高效转换, 现在普遍采用的思想是: 由 VMM 根据映射 f 和 g 生成复合的映射 $f \cdot g$, 并直接把这个映射关系交给处理器中的内存管理单元 (memory management unit, MMU). 这种思想的可行性在于: (1)VMM 维护着映射 f; (2)VMM 能够访问 Guest OS 的内存, 因此能够模拟

MMU 来查询 Guest OS 的页表, 从而能够获得映射 g; (3) 计算复合映射 $f \cdot g$ 能够在恰当的时候高效地进行. 尽管基本思路是一样的, 但现有的 VMM 针对这个思路分别采取了不同的实现方法.

2.1.1 MMU 半虚拟化 (MMU para-virtualization)

半虚拟化指的是经过 VMM 抽象的虚拟机体系结构与实际硬件略有不同, 并以此为代价降低 VMM 本身的复杂性, 同时获得更好的性能 [7]. MMU 的半虚拟化方法实现简单, VMM 将映射关系 $f \cdot g$ 直接写入 Guest OS 的页表中, 并替换原来的映射 g. 为了保证替换后 Guest OS 仍然能够正常工作, 半虚拟化要求修改 Guest OS 的少量源代码. 剑桥大学的 Xen 系统, 是采用半虚拟化技术的 VMM 的典型代表 [2~4].

2.1.2 影子页表 (shadow page table)

与半虚拟化不同, 影子页表技术为 Guest OS 的每个页表维护一个 "影子页面", 并将合成后的映射关系 $f \cdot g$ 写入到 "影子" 中, 而保持 Guest OS 的页表内容不变. 最后, VMM 将影子页表交给 MMU 进行地址转换. 由于影子页表的分配和维护完全是在 Guest OS 之外的 VMM 中进行的, 因而 Guest OS 是完全透明的, 能够适用于无法获得源代码的操作系统 (如 Microsoft Windows) 的虚拟化. VMWare Workstation、VMWare ESX Server 以及 KVM 都使用了影子页表技术 [1,5,8].

2.1.3 硬件辅助虚拟化 (EPT 或 NPT)

近年来, Intel 和 AMD 分别提出了扩展页表 (extended page tables, EPT)[9] 和嵌入页表 (nested page tables, NPT) 技术 [10], 它们允许 VMM 直接将映射关系 f 和 g 分别交由 MMU, 并由硬件自动完成两级的地址转换, 从而大大简化了 VMM 的设计. 但由于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. 为了降低地址转换开销, 处理器通常也要将部分合成后的映射关系 $f \cdot g$ 缓存在快表 (translation look-aside buffer, TLB) 中.

MMU 半虚拟化、影子页表和硬件辅助虚拟化的比较如图 3 所示. 由于本节中讨论的关于动态内存映射的理论模型与具体实现无关, 因此能够适用于上述 3 种内存虚拟化的方法.

2.2 虚拟机的简单内存映射

为虚拟机内存构造 f-map 最简单的方法就是建立物理地址到机器地址的一一映射关系. 形式化地说, 我们定义如下概念:

集合 P: 虚拟机所见到的物理内存;

集合 Z: 真实计算机上机器内存总和;

集合 M: VMM 提供给该虚拟机的机器内存. M 是 Z 的一个真子集.

单射 f: 从 P 到 M 的一个全函数, 即对于任意的 $p \in P$, 都存在唯一的 $m \in M$ 与之对应, 且 f 在虚拟机的生存周期内是不变的.

虚拟机的简单内存映射如图 4 所示. 在这里, 单射 f 就是 VMM 为进行内存虚拟化所维护的 f-map . 实现这种内存管理模式是非常简单的: 对于每个虚拟机, 我们可以用一个数组 (通常以页面为单位) 来保存映射 f, 这个数组在虚拟机创建时被初始化并赋值, 使得数组的每一元素都指向不同的机器页面. 需要查询 f-map 时, 直接用 Guest OS 的物理页面号 (guest frame number, GFN) 作为下标读取数组元素, 便可获得对应的机器页面号 (machine frame number, MFN), 其时间开销为常数. 当然, 这种数据结构和算法的可行性是建立在单射、全函数、生存周期内不变等假设之上的.

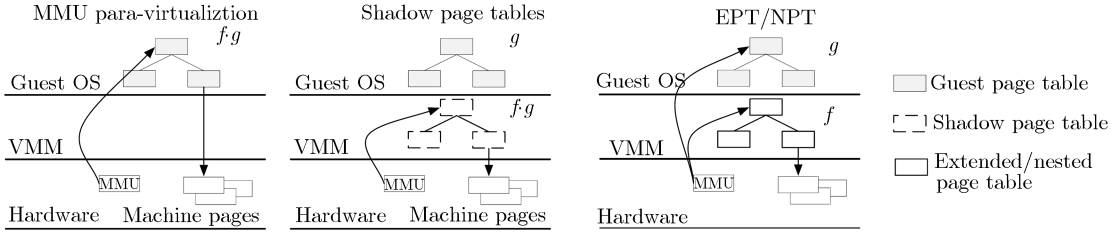

图 3 半虚拟化、影子页表以及硬件辅助虚拟化的比较

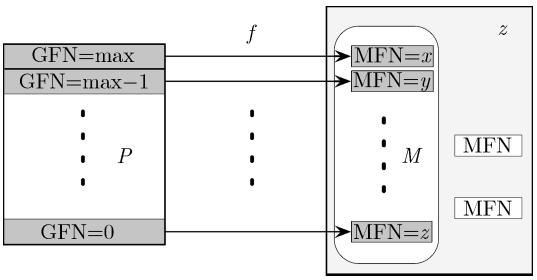

图 4 虚拟机的简单内存映射

但是, 简单内存映射模式无法实现第 1 节所提到的按需取页、虚拟存储、内存共享等高级内存管理功能. 这是因为: 第一, 这些功能要求集合 M 和映射 f 在虚拟机执行过程中是可变的; 第二, 这些功能要求某些物理内存可以没有机器内存与之对应, 即 f 不一定是全函数; 第三, 内存共享要求 f 不一定是单射.

尽管存在上述局限性, 由于这种模式的简单和高效性, 它仍然被一些 VMM 所采用, 例如 KVM [8].

2.3 动态内存映射的定义

我们仍然记虚拟机所见到的物理内存为集合 P, 虚拟机的生存周期为序列 $t_0, t_1, \ldots, t_h$, 假设在时刻 t_i 时, VMM 提供给该虚拟机的机器内存为集合 M_i , 映射 f_i 是从 P 到 M_i 的一个部分函数, 即对于任意的 $p \in P$, $f_i(p) = \emptyset$ 或 $f_i(p) = \{m\}, m \in M_i$, 且满足下列性质:

(1) 存在某个 $i \in \{0, 1, \ldots, h-1\}$, 使得 $f_i \neq f_{i+1}$, 或者 $M_i \neq M_{i+1}$;

(2) 若 $f_i(p) = \emptyset$, 则虚拟机对内存 p 的读写访问都将陷入 VMM;

(3)f_i 可能不是单射, 且对任意两个不同的 $p_1, p_2 \in P$, 若 $f_i(p_1) = f_i(p_2)$, 则虚拟机对内存 p_1 或者 p_2 的写访问都将陷入 VMM;

我们就称函数序列 $F = f_0, f_1, \ldots, f_h$ 是虚拟机内存 P 的一个动态内存映射.

在动态内存映射的定义中, 性质 (1) 保证了作为 f-map 的映射 f_i 和其值域 M_i 都是能够随着时间变化的, 因此我们允许 VMM 动态调整虚拟机使用的机器内存集合, 或者动态调整虚拟机的物理内存到机器内存的对应关系; 性质 (2) 允许虚拟机的某些物理内存没有机器内存与之对应, 而且保证 VMM 能够介入虚拟机对这些内存的访问 (包括读和写), 并进行模拟; 性质 (3) 允许虚拟机的某些物理内存共享同一机器内存, 而且保证 VMM 能够介入虚拟机对这些内存的写访问, 并进行模拟. 图 5 显示了某个时刻时动态内存映射的工作情况. 要注意到在上述定义中, 虚拟机所见到的物理内存集合 P 在虚拟机的生存周期中是不可变化的. 其原因是当前多数的硬件及操作系统都不支持内存的热插拔, 因此

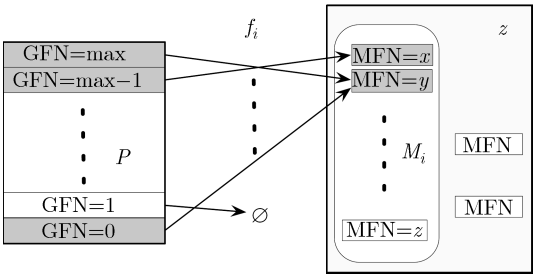

图 5　虚拟机动态内存映射

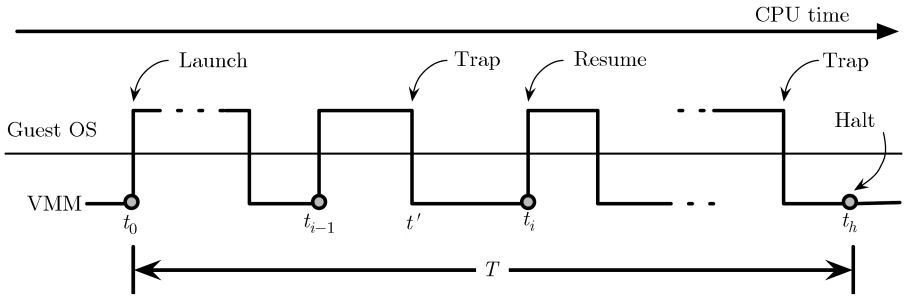

图 6　虚拟机的生存周期序列

Guest OS 所见到的物理内存空间大小通常是恒定的, 我们的 VMM 也要提供这种硬件抽象, 以保证虚拟环境的兼容性.

在动态内存映射的定义中, 需要进一步明确的是生存周期序列的概念. 我们将虚拟机的生存周期抽象为一系列离散的时间点是有根据的. 如图 6 所示, VMM 在 t_0 时刻启动虚拟机, 并在 t_h 时刻销毁虚拟机, 这两个时间点之间的连续时间区间称为虚拟机的生存周期 T. 而虚拟机的执行实际上是 CPU 控制权在 VMM 和 Guest OS(包括其上的应用程序) 之间交替的过程. 不失一般性地, 这里只考虑 VMM 上运行 1 台虚拟机的情况.

一方面, 当 CPU 控制权在 VMM 中时, 如时间段 $[t', t_i)$, VMM 才能够修改和维护 f-map , 但是由于 Guest OS 不在运行中, 这期间 f-map 的变化对 Guest OS 是没有影响的; 另一方面, 当 CPU 控制权在 Guest OS 中时, 如时间段 $(t_{i-1}, t']$, 由于 f-map 对 Guest OS 是不可见的, 可以保证这期间 f-map 一定不会发生变化. 因此, f-map 的变化对虚拟机的执行能够产生影响的关键时刻就是 CPU 控制权即将切换到 Guest OS 之时, 而不用考虑 (t_{i-1}, t_i) 之间的时间. 所以, 我们用离散的时间点序列 $t_0, t_1, \ldots, t_h$ 来考察映射 f 及集合 M 的变化是完全合理的.

2.4　动态内存映射的适用性

本节将根据 2.3 小节给出的动态内存映射的定义, 从理论上阐释这种模型对于按需取页、虚拟存储、内存共享等内存管理技术的适用性. 而且, 我们将给出利用动态内存映射模型实现这些技术的具体步骤.

2.4.1　按需取页

按需取页 (demand paging) 指的是只有当虚拟机真正需要的时候, VMM 才将机器内存分配给它,

并动态地建立该物理内存与机器内存的映射关系. 在动态内存映射模型下, 按需取页可以用如下步骤实现:

(1)VMM 初始化 $M_0 = \emptyset$, 且对于任意的 $p \in P$, 设置 $f_0(p) = \emptyset$, 并启动虚拟机;

(2) 在时刻 t_i, 若虚拟机访问了内存 p, 且 $f_i(p) = \emptyset$, 则根据性质 (2), 虚拟机的执行将中断, 控制转入 VMM;

(3)VMM 分配一块新的机器内存 m, 并设置 $M_{i+1} = M_i \cup \{m\}$, 设置 $f_{i+1}(p) = \{m\}$, 然后恢复虚拟机执行;

(4) 重复步骤 (2) 和 (3), 直到虚拟机停机.

2.4.2 虚拟存储

虚拟机系统中的虚拟存储 (virtual memory) 技术与操作系统中类似, 即允许虚拟机使用超过实际机器内存大小的内存空间 [11]. VMM 实现虚拟存储的方法主要有两种: 页面交换和 Ballooning, 下面我们将分别讨论.

2.4.2.1 页面交换

页面交换技术 (swapping) 是现代操作系统中广泛使用的虚拟内存技术 [11], 它允许操作系统将某些非活动的内存页面保存到内存以外的 "交换空间" 上 (通常是磁盘). 虚拟机的页面交换技术和操作系统的页面交换技术是十分类似的 [12], 它必须建立在按需取页机制的基础之上. 在动态内存映射模型下, 页面交换可以用如下步骤实现:

(1)VMM 初始化 $M_0 = \emptyset$, 且对于任意的 $p \in P$, 设置 $f_0(p) = \emptyset$, 并启动虚拟机;

(2) 在时刻 t_i , 若虚拟机访问了内存 p, 且 $f_i(p) = \emptyset$, 则根据性质 (2), 虚拟机的执行将中断, 控制转入 VMM;

(3)VMM 判断当前机器内存是否充裕, 若是, 则 VMM 分配一块新的机器内存 m, 并设置 $M_{i+1} = M_i \cup \{m\}$, 设置 $f_{i+1}(p) = \{m\}$, 然后转到步骤 (5);

(4) 否则, VMM 需要将某些页面换出. VMM 根据某种策略 (如 LRU) 从页面集合 M_i 中选择一个非空子集 S, 对于任意的页面 $s \in S$, 设置 $M_{i+1} = M_i - \{s\}$, 同时, 对于所有的 $x \in P$ 且 $f_i(x) = s$, 我们设置 $f_{i+1}(x) = \emptyset$, 并将 s 的内容写入交换空间. 然后回到步骤 (3);

(5)VMM 检查 p 的内容是否在交换空间中, 若是, 则从交换空间中将 p 的内容读入 m;

(6) 恢复虚拟机的执行, 转到步骤 (2), 直到虚拟机停机.

2.4.2.2 Ballooning

Ballooning(气球技术) 是虚拟机系统中所特有的虚拟存储技术 [1]. 它的基本思想是: 从其他虚拟机窃取一些未使用机器内存页面, 给急需内存的虚拟机使用. 为了实现内存的窃取, VMM 需要在 Guest OS 的内核中安装一个用于窃取内存的模块, 称作 "balloon driver". Ballooning 通常包括两个过程: VMM 通过 "气球膨胀 (balloon inflating)" 过程从虚拟机回收空闲的内存资源, 而通过 "气球收缩 (balloon deflating)" 过程将先前回收的内存返回给虚拟机. 在动态内存映射模型下, Ballooning 可以用如下步骤实现, 为了简单起见, 我们只考虑两台虚拟机的情况.

1. 气球膨胀 (balloon inflating)

(1)VMM 为两台虚拟机 VM 和 VM′ 分别初始化 M_0 和 M_0', f_0 和 f_0' , 并启动这两台虚拟机. 我们假设 M_0 和 M_0' 非空, 且存在 $p \in P$, $f_0(p) = \emptyset$;

(2) 在时刻 t_i , 若虚拟机 VM 访问了内存 p, 且 $f_i(p)=\emptyset$, 则根据性质 (2), 虚拟机的执行将中断, 控制转入 VMM;

(3)VMM 判断 M_i 中是否还有未被 f_i 映射的元素 m, 即 $m \in M_i - f_i(P)$. 若是, 则 VMM 设置 $f_{i+1}(p)=\{m\}$, 设置 $M_{i+1}=M_i$, 然后转到步骤 (6);

(4) 否则, VMM 将向另一台虚拟机 VM′ 的 balloon driver 发出 "气球膨胀" 请求, balloon driver 将从它的页面集合 M'_i 中获得一个非空子集 S' , 对于任意的页面 $s' \in S'$, 设置 $M'_{i+1}=M'_i-\{s'\}$, 同时, 对于所有的 $x' \in P'$ 且 $f'_i(x')=s'$, 我们设置 $f'_{i+1}(x')=\emptyset$;

(5)VMM 在虚拟机 VM 上设置 $M_i=M_i \cup S'$. 然后回到步骤 (3);

(6) 恢复虚拟机的执行, 转到步骤 (2), 直到虚拟机停机.

2. 气球收缩 (balloon deflating)

(1) 由于 "气球膨胀" 过程的步骤 (4), 存在时刻 t_i 以后的某个时刻 t_j, 存在 $p' \in P'$, 且 $f'_j(p')=\emptyset$;

(2) 在时刻 $t_j(j>i)$, 若虚拟机 VM′ 访问了内存 p′, 且 $f'_j(p')=\emptyset$, 则根据性质 (2), 虚拟机的执行将中断, 控制转入 VMM;

(3)VMM 判断 M'_j 中是否还有未被 f'_j 映射的元素 m', 即 $m' \in M'_j - f'_j(P')$. 若是, 则 VMM 设置 $f'_{j+1}(p')=\{m'\}$, 设置 $M'_{j+1}=M'_j$, 然后转到步骤 (6);

(4) 否则, VMM 将向 VM′ 的 balloon driver 发出 "气球收缩" 请求, 由于 balloon driver 仍然保存着 "气球膨胀" 过程中获得的非空子集 S', 对于任意的页面 $s' \in S'$, VMM 将为虚拟机 VM 分配一个空闲的机器页面 m, 将 s' 的内容复制到 m 中, 并设置 $M_{j+1}=M_j \cup \{m\}-\{s'\}$, 同时, 对于所有的 $x \in P$ 且 $f_j(x)=s'$, 我们设置 $f_{j+1}(x)=m$;

(5)VMM 在虚拟机 VM′ 上设置 $M'_j=M'_j \cup S'$. 然后回到步骤 (3);

(6) 恢复虚拟机的执行, 转到步骤 (2), 直到虚拟机停机. 上述步骤 (4) 中, 若 VMM 无法为虚拟机 VM 分配一个空闲的机器页面 m, 则 VMM 可能发起另一个页面交换或气球膨胀过程, 在此不再赘述.

2.4.3 内存共享和 copy-on-write

内存共享指的是 VMM 允许虚拟机之间只读地共享完全相同的内存区域. 内存共享能够缓解大量虚拟机并发运行时的内存资源紧缺. 例如, 同一台物理主机上的多个虚拟机可能运行着同一类 Guest OS 的多个实例, 或者运行着相同的应用程序、组件代码, 或者载入了相同的数据集. 在这种情况下, 允许虚拟机之间共享相同的内存区域将大大提高内存资源的利用率 [13].

为了保证虚拟机之间的隔离性, VMM 通常不允许不同的虚拟机对同一页面进行可写共享. 因此, 如果某台虚拟机对一个共享的页面执行了写操作, 我们就必须将共享的页面复制一份, 并将修改反映在副本上, 以保证该页面的内容对其他虚拟机是不变的. 这个过程就是 copy-on-write(COW). COW 是类 Unix 操作系统创建新进程时采用的主要技术, 我们认为, 在虚拟机系统中, 它同样可以被用于虚拟执行环境的快速复制 [14]. copy-on-wirte 的工作原理如图 7 所示.

内存共享和 copy-on-write 机制也可以用动态内存映射模型实现. 由于多虚拟机共享内存和单虚拟机共享内存的实现步骤从本质上是一样的, 为了叙述方便, 我们将动态内存映射的定义扩展如下:

记 VMM 上运行的所有虚拟机为集合 V, 其中某个虚拟机 $v \in V$ 所见到的物理内存为集合 P_v . 定义 $P=\{\langle v,p\rangle | v \in V \wedge p \in P_v\}$, 它表示所有虚拟机的所有物理内存. 假设在时刻 t_i 时, VMM 提供给所有虚拟机的机器内存为集合 M_i . 映射 f_i 是从 P 到 M_i 的一个部分函数, 且满足 2.3 小节给出的 3 个性质.

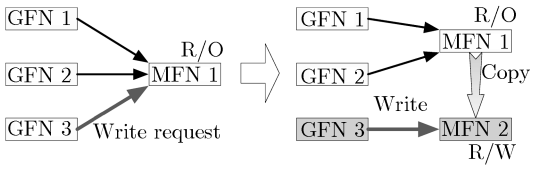

图 7 内存 copy-on-write 的工作原理

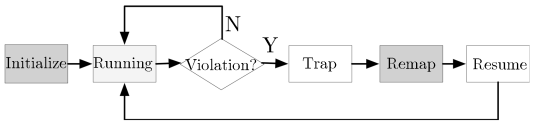

图 8 应用动态内存映射模型的基本流程

在上述定义下, 内存共享和 copy-on-write 的步骤可以描述如下:

(1)VMM 为各个虚拟机初始化 $M_0 = \{s\}$, 并对于任意的 $\langle v, p\rangle \in P$, 设置 $f_0(\langle v, p\rangle) = \{s\}$, 然后依次启动各个虚拟机;

(2) 在时刻 t_i, 虚拟机 v 试图写入内存 $\langle v, p\rangle$. 若存在 $\langle v', p'\rangle \in P, \langle v', p'\rangle \neq \langle v, p\rangle$, 但 $f_i(\langle v, p\rangle) = f_i(\langle v', p'\rangle)$, 则根据性质 (3), 虚拟机的执行将中断, 控制转入 VMM;

(3)VMM 分配一个新的机器页面 m, 将页面 $f_i(\langle v, p\rangle)$ 的内容复制到 m 中. 然后, VMM 设置 $M_{i+1} = M_i \cup \{m\}$, 设置 $f_{i+1}(\langle v, p\rangle) = \{m\}$, 并恢复虚拟机的执行;

(4) 重复步骤 (2) 和 (3), 直到所有虚拟机停机.

在上述步骤中, 所有虚拟机的内存一开始都映射到同一个机器页面, 并随着它们运行的过程利用 copy-on-write 不断分裂. 这从理论上是可行的, 但是考虑到性能问题, 实际的 VMM 通常不会采取如此激进的策略.

从以上内容我们可以看到, 动态内存映射模型能够很好地支持按需取页、虚拟存储、内存共享等内存管理功能, 因此该模型具有很强的适用性. 尽管实现这些功能的方法各不相同, 但其思想都是一样的: 利用 f-map 的违例陷入机制, 由 VMM 介入并模拟违例的行为, 或者对资源进行修复和再分配. 例如, 按需取页、虚拟存储机制是利用动态内存映射模型的性质 (2) 来实现违例陷入, 而内存共享和 COW 则是利用该模型的性质 (3) 来实现违例陷入. 图 8 显示了实现这些功能的基本流程.

3 动态内存映射机制的实现及评估

本节将探讨在开源的 VMM——KVM(kernel-based virtual machine)[8] 上具体实现该机制的必要步骤和主要技术难点. 我们的实现主要用于验证动态内存映射模型是可行实用的, 同时给出实现过程中一些关键技术问题的解决方法.

从程序设计的角度看, 实现动态内存映射机制将涉及以下问题:

(1) 如何维护用于虚拟化的物理资源集合 M, 并且使得 M 是可变的?

(2) 如何表示和维护从 P 到 M 的映射关系 f, 并且使得 f 是可变的?

(3) 如何保证当发生 f-map 违例时, VMM 能够获得计算机的控制权?

为了解决问题 (1), 我们提出了 "页面池" 的概念, 它是虚拟机物理资源集合 M 的一个具体实现; 为了解决问题 (2), 我们设计了用于完成映射的主要数据结构, 并描述了该数据结构是如何与页面池协同工作的; 为了解决问题 (3), 我们详细地考察了 KVM 中影子页表和页故障的处理机制, 并用逆映射 (reverse mapping) 的方法实现了一个安全的页面回收机制, 以保证在发生 f-map 违例时, Guest OS 会由于页故障而陷入 VMM.

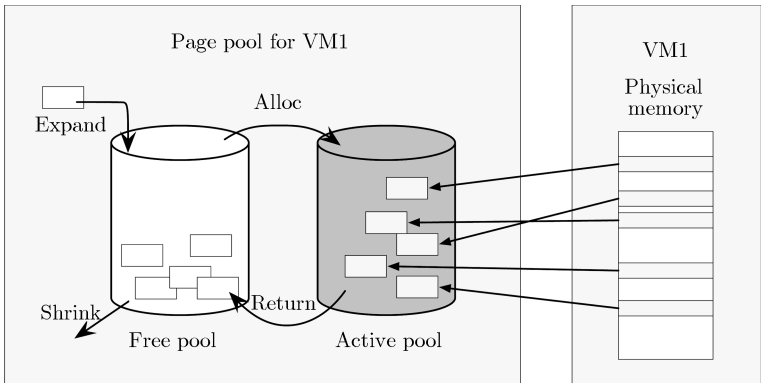

图 9　页面池的整体设计

3.1　页面池设计与实现

页面池 (page pool) 是由 VMM 维护的, 提供给某个虚拟机使用的机器页面的集合, 它的大小可以在虚拟机执行过程中动态地改变. 如前所述, 页面池实际上是集合 M 的具体实现, 根据 2.4 小节的讨论, 集合 M 必须至少支持以下两个操作:

(1)$M_{i+1} = M_i \cup \{m\}$;

(2)$M_{i+1} = M_i - \{m\}$.

因此页面池也必须支持动态地添加和删除某个页面.

3.1.1　页面池的设计

一个页面池实际上由两个部分构成: 空闲池 (free pool) 和活动池 (active pool). 图 9 给出了页面池的整体设计.

空闲池包含了当前分配给该虚拟机、但尚未被虚拟机使用的所有机器页面, 而活动池则包含了所有已被虚拟机映射的机器页面. 严格地说, 活动池中的页面对应于集合 $f(P)$, 而空闲池中的页面则对应于集合 $M - f(P)$. 我们规定:

(1) 虚拟机永远只能和活动池中的页面建立 f-map;

(2)VMM 要改变页面池的大小时, 永远只能向空闲池中添加新的机器页面, 或者从空闲池中取出机器页面.

根据以上设计, 页面池将包含以下操作:

• 分配 (alloc): 将一个页面从空闲池移入活动池, 并且保证为该页面建立 f-map.

• 返还 (return): 将一个页面从活动池放回空闲池, 之前必须保证删除了与该页面有关的所有 f-map.

• 扩大 (expand): 将某些机器页面加入页面池的空闲池中, 其结果是增大了虚拟机所能使用的机器内存 (集合 M) 的大小.

• 收缩 (shrink): 将某些机器页面从页面池的空闲池中删除, 其结果是减小了虚拟机所能使用的机器内存 (集合 M) 的大小.

将页面池划分为两个部分的方法, 使得 VMM 能够快速地得到集合 $f(P)$ 和集合 $M - f(P)$, 并且只需要 O(1) 的时间就能够判断某个页面是否正在被 Guest OS 的 f-map 所映射. 同时, 由于操作

$M_{i+1} = M_i \cup \{m\}$ 和操作 $M_{i+1} = M_i - \{m\}$ 都只在空闲池中进行, 而这不涉及任何有关 f-map 的操作, 因而大大简化了实现的复杂度. 我们将在 3.2 小节中, 再具体讨论建立和删除 f-map 的方法.

3.1.2 页面池的实现

我们使用双链表来实现页面池. 我们为每个虚拟机定义两个链表: free_list 和 active_list, 分别对应于 3.1.1 小节中提到的空闲池和活动池; 类似地, 我们再为每个虚拟机设置两个计数器: free_count 和 active_count, 分别用于表示集合 $f(P)$ 和集合 $M - f(P)$ 的大小. 链表中存放的内容为 "机器页面描述符", 每个描述符对应页面池中的一个页面. 它需要维护的内容包括: (1) 该页面被 f-map 引用的次数, 即页面原像集的大小, 这个计数值对于实现页面共享有重要意义; (2) 页面的标志, 用于表明该页面是否属于活动池等; (3) 对应的页面指针, 我们沿用 KVM 的设计, 使其指向 host Linux 内核的 page 结构; (4) 链表的钩子.

上述数据结构是非常高效的, 每个描述符在 32 位计算机上占用 20 字节, 在 64 位计算机上占用 40 字节. 如果页面大小为 4 KB, 则该描述符的相对开销仅为 0.005(32 位)~0.010(64 位). 由于页面描述符的数量不会多于实际机器的总内存页面数, 总的空间开销是可控的. 例如, 在拥有 2 GB 内存的 32 位计算机上, 总的机器页面管理开销最大为 10 MB.

3.1.3 映射的表示

我们用一个大小为 $|P|$ 的数组 phys_mem[ ](P 为虚拟机所见的物理内存集合) 来表示映射 f, 并对数组 phys_mem[ ] 的元素定义如下:

对于虚拟机的每个物理页面 $p \in P$,

(1) 若 $f_i(p) = \emptyset$, 则 phys_mem[p] = NULL;

(2) 若 $f_i(p) = \{m\}$, 则 phys_mem[p] = page_desc(m), 其中, page_desc(m) 表示机器页面 m 所对应的机器页面描述符指针.

这样, 如果 $f_i(p) \neq \emptyset$, VMM 便能够用 phys_mem[p]→page 来获得对应的 page 结构, 其时间开销为常数. 而当 phys_mem[p] 为 NULL 时, VMM 便可以知道 Guest OS 对物理内存 p 的访问发生了 f-map 违例. 由于我们在 phys_mem[] 数组中只保存页面描述符的指针, 这与 KVM 原有的实现中用数组直接存放 Host Linux 的页面指针相比, 其空间开销是完全相同的.

3.2 违例陷入机制

如 2.3 小节所述, 动态内存映射机制能够工作的关键在于: 当 Guest OS 发生 f-map 违例时会陷入 VMM, VMM 获得计算机的控制权后, 才能模拟或修复 f-map. 本节将具体讨论如何保证这种陷入, 并在 KVM 中实现这个陷入机制.

3.2.1 基本工作原理

在 3.1.3 小节中我们知道, VMM 可以通过查询数组 phys_mem[ ] 来判断 Guest OS 是否发生了 f-map 违例. 但是, 正在运行之中的 Guest OS 是不会去查询 phys_mem[ ] 的 (事实上这是不允许的), 那么如何保证运行之中的 Guest OS 遇到违例时会陷入 VMM 呢?

解决问题的关键在于影子页表. 在采用影子页表技术的虚拟机系统中, 影子页表是虚拟机访问机器内存的唯一途径, 因此, 只要保证对 f-map 的更新能够反映到影子页表中, 就能够利用 MMU 的硬件保护机制, 保证虚拟机的访问违例会陷入 VMM.

由 2.1 小节可以知道, 影子页表的实质就是保存了映射 $f \cdot g$. 因此, 形式化地说, VMM 必须保证如下两点:

(1) 若 $f(p) = \emptyset$, 则对于任意的虚拟地址 v, 若 $g(v) = p$ 则 $f \cdot g(v) = \emptyset$;

(2) 对任意两个不同的 $p_1, p_2 \in P$, 若 $f(p_1) = f(p_2)$, 则对于任意的虚拟地址 v, 若 $g(v) = p_1$ 或 $g(v) = p_2$, 则 $f \cdot g(v)$ 是只读映射.

这两条性质实际上是 2.3 小节的定义在影子页表上的传递性. 在 x86 体系结构下, 上述的 (1) 可以通过清除影子页表项的 P(present) 位来实现, 而 (2) 则可以通过清除影子页表项的 W(writable) 位来实现.

3.2.2 逆映射

2.4 小节中的讨论表明, 在交换、Ballooning、copy-on-write 等应用中, 我们需要根据一个给定的机器页面 m, 找到映射了 m 的所有物理页面的集合 X, 并对 X 中的每个元素 x 更新 f-map . 这里的集合 X 实际上就是 m 关于 f 的原像 $X = f^{-1}(m)$. 逆映射 (reverse mapping) 就是一种能够快速获得原像的数据结构. 由传递性可知, 在违例陷入机制的实现中, VMM 希望能够直接获得的是所有映射了某个机器页面 m 的影子页表项, 即 $V = (f \cdot g)^{-1}(m)$, 而这也是可以通过 reverse mapping 实现的.

KVM 中已经为可写的页面实现了 reverse mapping 机制 [8], 并利用这个机制实现了捕获 Guest OS 对页表的写入, 介入 Guest OS 对页表的写入对于 VMM 维护影子页表是至关重要的. 但是, 为了实现动态内存映射, 我们必须为所有的页面维护 reverse mapping, 而不仅仅是可写的页面, 这就需要修改 KVM 现有逆映射机制的实现. 毫无疑问, 这个改变会大大增加 KVM 内存管理的时间和空间开销, 构成了动态内存映射机制带来的最大的性能代价. 我们将在 3.3 小节对这个性能开销进行评估.

3.2.3 页故障的处理流程

VMM 进行内存虚拟化的最重要方法就是陷入并处理虚拟机的 f-map 违例. 这项工作是由页故障处理函数 (page fault handler) 来完成的. VMM 必须在页故障处理函数中实现 f-map 维护、影子页表维护、逆映射维护, 以及上述提到的页面池、按需取页、虚拟存储、copy-on-write 等全部功能. 因此, 页故障处理函数的工作流程是 VMM 关于内存虚拟化最复杂的一个部分, VMM 必须对导致页故障的原因进行的深入分析, 并针对不同的情况采取不同的处理方法. 本小节中, 我们将看到上述的页面池、逆映射等是如何被 VMM 使用的.

在 KVM 中, 导致虚拟机陷入页故障处理函数的原因只有一个: Guest OS 对内存的使用违反了影子页表定义的权限规则. 因此在陷入时, VMM 能知道的信息包括: (1) 导致页故障的虚拟地址 v; (2) 页故障的性质 (缺页、权限违例、读写违例). 但是, VMM 能够根据其掌握的其他数据结构, 分析出更详细的故障来源. 这些信息包括:

(1) 导致页故障的虚拟机. 这可以通过查询当前 CPU 的调度信息得到.

(2) 与导致页故障的虚拟地址 v 相对应的物理地址 p 及相应的访问权限. 这可以通过模拟 MMU 查询 Guest OS 的多级页表来实现.

(3) 和虚拟地址 v 有关的任何一级影子页表. 这可通过查询 KVM 中维护的存放 Guest OS 页表与影子页表对应关系的 Hash 表来实现.

(4) 与物理地址 p 对应的机器页面描述符 page_desc(m). 如 3.1.3 小节所述, 这可以通过查询数组 phys_mem[ ] 得到.

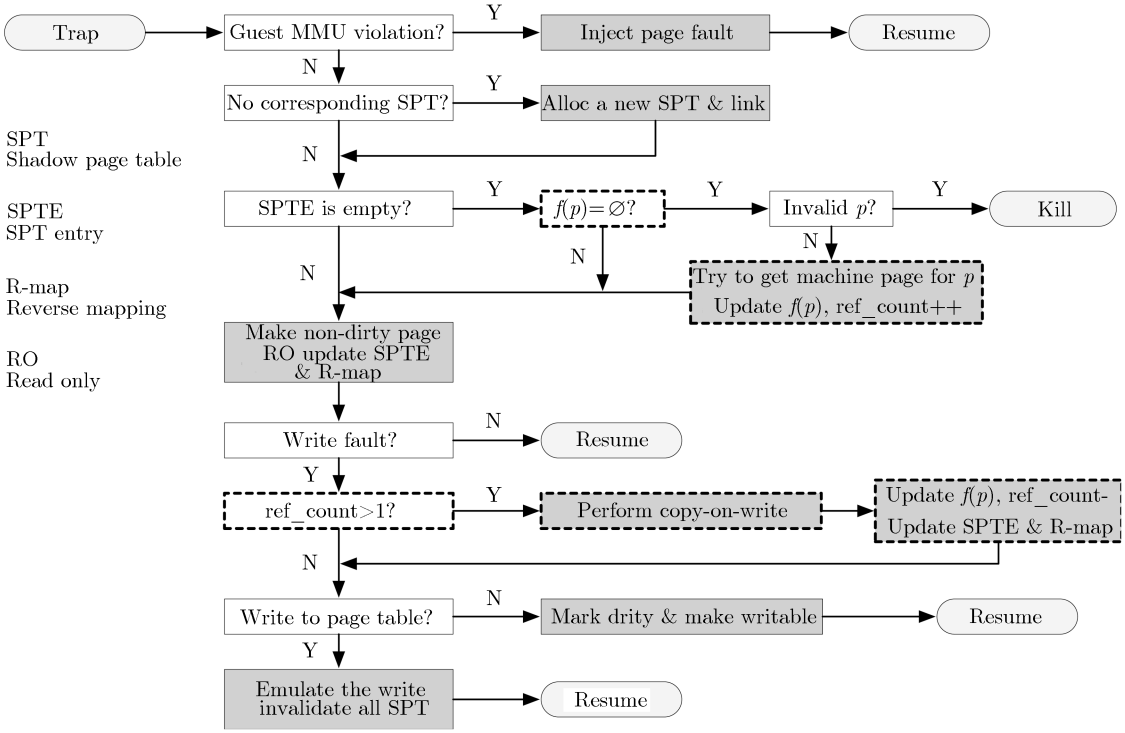

图 10 页故障的主要处理流程

考虑了动态内存映射之后, 页故障处理的主要工作流程如图 10 所示. 图中虚框部分是实现动态内存映射的关键步骤, 下面我们将重点讨论.

首先, KVM 会根据导致页故障的虚拟地址 v, 逐级周游 Guest OS 的页表系统. 在周游的同时, KVM 还会查找各级页表对应的影子页表是否存在, 如果不存在则为其新分配一个, 并建立它与上层影子页表的链接关系. 这样, 如果页故障是由于影子页表缺失而导致的, 则经过此番周游, 各级的影子页表都已补齐了.

当周游到最后一级页表时, KVM 便可以知道这个页面对于 Guest OS 是否存在. 如果该页面根本没有被 Guest OS 映射, 或者访问权限不对, 则 Guest OS 必须对这个页故障负责. KVM 将通过 VT 的中断注入机制, 把这个页故障转交给 Guest OS 处理.

否则, 则表明 VMM 必须对这个页故障负责. 此时导致故障的主要原因包括:

(1) 映射在 f-map 中存在, 但对应的影子页表项没有初始化.

(2) $f(p) = \emptyset$, 这种情况又包括: 物理页面号 p 是非法的、尚未为该物理页面分配机器页面、该页面已被换出、该页面已被 balloon driver 交给其他虚拟机使用等等.

(3) Guest OS 试图写入一个共享页, 而这个页已经在影子页表中标为只读的. 是否共享页可以利用机器页面描述符的 ref_count 域来判断.

(4) Guest OS 试图写入一个用作页表的页面, 而且该页表对应的影子页表已经建立. 一个页面是否被用作页表, 可以通过查询影子页表的 Hash 表获知.

对于原因 (1), VMM 只需根据 $f(p)$ 和 $g(v)$ 设置该页表项即可.

对于原因 (2), VMM 要先建立物理页面到机器页面的映射关系, 然后再更新导致页故障的影子页表项. 为了建立映射关系, 我们需要先得到一个有效的机器页面 m. m 可以直接从空闲池中取出, 但

如果空闲池中没有可用页面, 则需要发起页面淘汰算法或 “气球收缩” 过程, 重新填充空闲池. 这个过程中, 我们需要减少被回收页面的引用计数 (ref_count), 并根据逆映射, 确保放回空闲池的页面不被任何影子页表项映射. 一旦得到了机器页面 m, 我们就可以更新映射 $f(p)$, 增加 m 的引用计数, 然后再根据 $f(p)$ 和 $g(v)$ 设置好该页的影子页表项.

对于原因 (3), VMM 将发起 copy-on-write 过程, 为该机器页面建立一个副本, 然后把导致异常的物理页面重新映射到这个机器页面, 并且设置该映射为可写的. 而对于原来的机器页面, 我们要将它的引用计数减 1, 只要计数值仍然大于 1, 其他的物理页面仍然只能只读共享该页面. 而如果计数值降为 1, 则需要根据逆映射, 将影子页表中所有对该页面的映射置为可写的.

对于原因 (4), VMM 将模拟这个写操作, 然后删除该页表对应的影子页表, 这个影子页表将在后来的访问中重新被建立. 此步骤确保对页表的更新能及时反映到影子页表上. 完成了页故障修复后, VMM 就可以恢复 Guest OS 的执行了. 只要确保进入 Guest OS 之前, 所有的映射关系都已按照真实硬件的逻辑建立好, 整个内存虚拟化过程对 Guest OS 就是完全透明的.

当然, 页故障处理函数还需要把影子页表项的 Accessed 位、Dirty 位等硬件自动设置的标志位向 Guest OS 的页表传递, 需要处理 4 MB 页、物理地址扩展 (PAE) 等特殊的分页模式, 需要在更新影子页表项的同时维护它与逆映射的一致性, 等等. 这些工作是很琐碎的, 但与本文的主体内容无关, 因此不再赘述.

3.3 性能开销评估

本节中, 我们将通过测试, 评估在加入了 DMM 机制后, 给 KVM 的内存虚拟化带来的额外开销. 本文的重点在于 DMM 的理论模型, 以及其通用型、高耦合度、平台无关性等特点. DMM 是对现有的各种内存虚拟化技术的一个很好的补充, 它并不与现有的各种技术冲突, 同时也不在于 KVM 上的实现本身. 我们进行性能开销测试的主要目的是进一步验证动态内存映射机制的可行性和高效性. 通过比较引入该机制前后虚拟机的性能, 我们将证明动态内存映模型在实际的 VMM 设计与实现中具有较强的实用性.

测试中, 我们在同一台物理计算机上部署了两台完全一样的虚拟机, 其上都安装了 Cent OS Linux, 内核版本为 2.6.9. 我们分别用修改前和修改后的 KVM 运行该虚拟机, 并在虚拟机上编译 2.6.20 版本的 Linux 内核. 二者的执行时间差异如图 11(a) 所示.

可以看出, 修改前后的编译时间相差约 138 s, 性能下降了 2.9%. 为了确定性能开销的来源, 我们分别统计了页面池维护、映射计算、逆映射维护这 3 个部分的额外执行时间. 结果发现, 逆映射的维护占了其中的 86%, 对性能的降低起了决定性作用 (图 11(b)). 而其他性能开销的来源包括: 获取机器页面时对空指针的判断、获取 page 结构时的间接寻址、页面池链表的维护等.

逆映射的维护消耗时间的原因在于: 在原来 KVM 的设计中, 它只为可写的页面维护逆映射, 以确保该页面被 Guest OS 用作页表时, 能够去除所有影子页表项中的写权限, 从而介入 Guest OS 页表的写操作. 但是如 3.2 小节所述, 引入了动态内存映射后, 我们必须为包括只读页面在内的所有页面维护逆映射, 以确保映射关系变化时, 能够将变化传递到所有影子页表项中. 这就大大增加了逆映射需要维护的页面数.

为了研究修改前后逆映射数量的变化, 我们对修改前后的逆映射数量展开了测试. 图 11(c) 显示了内核编译过程中逆映射数量的动态变化过程, 图中的离散点为逆映射数量的实时采样, 而连续线是以 100 秒为区间的均值变化. 可以看出, 修改后的 KVM 在平均情况下, 大约多维护了 1600 个页面的

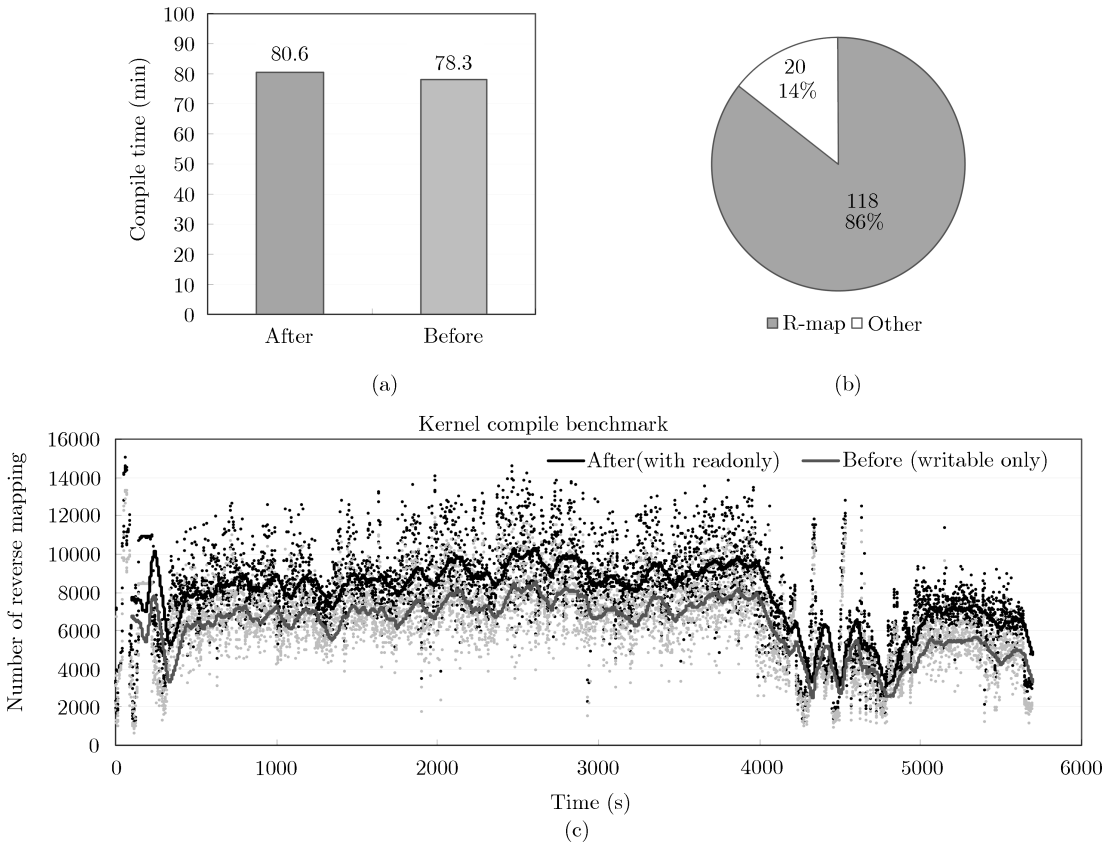

图 11 动态内存映射的性能开销分析

逆映射, 比原先增加了 1/4, 显然, 这将带来较大的开销.

但是, 逆映射是虚拟机动态内存映射机制所必需维护的数据结构, 因此这个开销是不可避免的. 而相对于动态内存映射机制带来的丰富内存管理功能, 2.9% 的性能下降是完全可以接受的. 综上, 我们的测试表明, 虚拟机的动态内存映射机制是非常高效的.

4 总结

本文提出了动态内存映射模型 DMM, 同现有的虚拟机内存管理机制相比, DMM 具有如下特点:

• 平台无关性. DMM 模型是用一种抽象的方法描述的, 并不局限于某种特殊的实现方案, 也不局限于特定体系结构的计算机.

• 通用性. 利用 DMM, VMM 能够很容易地实现虚拟机的按需取页、虚拟内存、Ballooning、内存共享、copy-on-write、虚拟机快速复制. 可以通过 DMM 模型的定义得到相应的机制步骤.

• 高效性. 实验结果表明, DMM 模型给 VMM 的内存虚拟化带来的额外时间开销能够控制在 5% 以内, 而其中的绝大多数开销来源于逆映射的维护.

• 高耦合度. 利用 DMM 实现的虚拟机高级内存管理功能, 由于其底层采用的是同一个高效的机制, 因此具有很好的耦合性, 相互之间的兼容性很好, 而且易于协同工作. 这给 VMM 设计高效的上层内存管理策略带来的极大的方便.

• 安全性. 由于 DMM 具有代码复用等优点, 能够很好地控制整个 VMM 系统的复杂性, 从而提高整个计算系统的安全性和可信度. 同时, 由于虚拟机的所有内存访问都是通过同一机制实现的, 因此易于 VMM 进行集中的隔离控制和权限检查.

综上, 在整个虚拟化系统 VMM 中, 上层 (面向 Guest OS) 对内存虚拟化需求多种多样, 这包括按需取页、虚拟存储 (交换、Ballooning)、内存共享以及 copy-on-write 等; 而底层的内存虚拟化实现也是各不相同, 包括半虚拟化 (Xen)、影子页表 (VMware, KVM)、硬件辅助虚拟化 (NPT, EPT) 等. 在现有的 VMM 中, 上层的不同内存管理功能, 需要依托底层特定的内存虚拟化实现来完成, 这使得 VMM 无论从功能的提供, 还是实现的复杂性、灵活性等方面, 都存在很大的局限性. 而 DMM 提供了一个模块化的分层体系结构, 它与底层的硬件和实现无关, 但为实现上层的管理功能提供了必要的、统一的机制. DMM 为实现特征可调的内存管理提供了可能: 一方面, 上层的内存管理策略可调; 另一方面, 底层的实现机制也可调. 换句话说, DMM 将上层的内存管理策略和底层的内存虚拟化实现很好地融合起来了.

参考文献

1 Waldspurger C A. Memory resource management in VMware ESX server. In: Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI). New York: ACM, 2002. 181–194

2 GoldBerg R P. Survey of virtual machine research. IEEE Comput Mag, 1974, 7: 34–45

3 Pratt I, Fraser K, Hand S, et al. Xen 3.0 and the art of virtualization. In: Proceedings of the Linux Symposium 2005. Linux Symposium, 2005. 65–77

4 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. New York: ACM, 2003. 164–177

5 Sugerman J, Venkitachalam G, Lim B H. Virtualizing I/O device on VMware workstation's hosted virtual machine monitor. In: Proceddings of the 2001 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2001. 1–14

6 Fraser K, Hand S, Neugebauer R, et al. Safe hardware access with the Xen virtual machine monitor. In: Proceedings of the 1st Workshop on Operating System and Architectural Support for the on Demand IT InfraStructure (OASIS). 2004. 1–10

7 Whitaker A, Shaw M, Gribble S D. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01. 2002

8 Kivity A, Kamay Y, Laor D, et al. KVM: the Linux virtual machine monitor. In: Proceedings of the Linux Symposium 2007. Linux Symposium, 2007. 255–230

9 Neiger G, Santoni A, Leung F, et al. Intel virtualization technology: hardware support for efficient processor virtualization. Intel Technol J, 2006, 10: 167–178

10 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

11 Tanenbaum A S. Modern Operating Systems (in Chinese). Beijing: Prentice Hall and China Machine Press, 1999. 52–100

12 Govil K, Teodosiu D, Huang Y, et al. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. ACM Trans Comput Syst, 2000, 18: 229–262

13 Bugnion E, Devine S, Govil K, et al. Disco: running commodity operationg systems on scalable multiprocessors. ACM Trans Comput Syst, 1997, 15: 412–447

14 Ford B, Hibler M, Lepreau J, et al. Microkernels meet recursive virtual machines. In: Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96). New York: ACM, 1996. 137–151