软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(9):3970−3984 [doi: 10.13328/j.cnki.jos.007359] [CSTR: 32375.14.jos.007359]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

RISC-V 架构下的懒惰影子页表模型*

李传东[1,2,3], 衣 然[1,2], 罗英伟[1,2,3], 汪小林[1,2,3], 王振林[4]

[1](北京大学 计算机学院, 北京 100871)

[2](多媒体信息处理全国重点实验室 (北京大学), 北京 100871)

[3](中关村实验室, 北京 100094)

[4](Michigan Technological University, Houghton 49931-1295, USA)

通信作者: 罗英伟, E-mail: lyw@pku.edu.cn

摘 要: 内存虚拟化作为虚拟化技术的核心组成部分, 直接影响虚拟机的整体性能. 目前, 主流的内存虚拟化方法在两维地址翻译开销与页表同步开销之间面临权衡. 传统的影子页表模型通过一套由软件维护的页表, 实现了与原生 (native) 环境相当的地址翻译性能. 然而, 由于影子页表的同步依赖于基于写保护的机制, 频繁的虚拟机退出 (VM-exit) 会严重影响系统性能. 相对而言, 嵌套页表模型依靠硬件辅助虚拟化, 允许虚拟机的客户页表直接加载到内存管理单元中, 从而避免了页表同步的开销. 然而, 这种方法的两维页表遍历却显著降低了地址翻译效率. 基于 RISC-V 架构下的特权级模型和虚拟化硬件特性, 提出了一种懒惰影子页表模型 LSP (lazy shadow paging), 在保留影子页表的地址翻译高效性的同时降低了页表同步开销. 懒惰影子页表模型深入分析了客户机对页表页的访问模式, 将页表同步与转址旁路缓存 (translation lookaside buffer, TLB) 刷新操作绑定以降低虚拟机退出的数量. 然后, 利用 RISC-V 架构中对 TLB 的细粒度刷新且可拦截的特性, 无效化需同步的影子页表项, 将页表同步的软件开销推迟到了首次访问该页面的时刻. 此外, 懒惰影子页表模型利用 RISC-V 架构中全新的特权级模型, 设计了 TLB 拦截的快速路径, 进一步减少了虚拟机退出带来的软件开销. 实验结果表明, 在基础 RISC-V 架构下, 懒惰影子页表相对于传统影子页表在微基准测试中降低了最多 50% 的虚拟机退出数量. 在支持 RISC-V 的虚拟化扩展架构下, 懒惰影子页表对 SPEC2006 基准测试中的典型应用相较于传统影子页表降低了最多 25% 的虚拟机退出数量, 相较于嵌套页表每次 TLB 缺失降低了 12 次访存.

关键词: RISC-V; 虚拟化扩展; 内存虚拟化; 影子页表; 嵌套页表

中图法分类号: TP316

中文引用格式: 李传东, 衣然, 罗英伟, 汪小林, 王振林. RISC-V架构下的懒惰影子页表模型. 软件学报, 2025, 36(9): 3970–3984. http://www.jos.org.cn/1000-9825/7359.htm

英文引用格式: Li CD, Yi R, Luo YW, Wang XL, Wang ZL. Lazy Shadow Paging Under the RISC-V Architecture. Ruan Jian Xue Bao/Journal of Software, 2025, 36(9): 3970–3984 (in Chinese). http://www.jos.org.cn/1000-9825/7359.htm

Lazy Shadow Paging Under the RISC-V Architecture

LI Chuan-Dong[1,2,3], YI Ran[1,2], LUO Ying-Wei[1,2,3], WANG Xiao-Lin[1,2,3], WANG Zhen-Lin[4]

[1](School of Computer Science, Peking University, Beijing 100871, China)

[2](National Key Laboratory for Multimedia Information Processing (Peking University), Beijing 100871, China)

[3](Zhongguancun Laboratory, Beijing 100094, China)

* 基金项目: 国家重点研发计划 (2022YFB4500701); 国家自然科学基金 (62032008, 62032001, 62372011)
本文由“RISC-V 系统软件及软硬协同技术”专题特约编辑武延军研究员、谢涛教授、侯锐研究员、张科正高级工程师、宋威副研究员、邢明杰高级工程师推荐.
收稿时间: 2024-08-25; 修改时间: 2024-10-15; 采用时间: 2024-11-26; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-06-11

[4](Michigan Technological University, Houghton 49931-1295, USA)

Abstract: Memory virtualization, a core component of virtualization technology, directly impacts the overall performance of virtual machines. Current memory virtualization approaches often involve a tradeoff between the overhead of two-dimensional address translation and page table synchronization. Traditional shadow paging employs an additional software-maintained page table to achieve address translation performance comparable to native systems. However, synchronization of shadow page tables relies on write protection, frequently causing VM-exits that significantly degrade system performance. In contrast, the nested paging approach leverages hardware-assisted virtualization, allowing the guest page table and nested page table to be directly loaded into the MMU. While this eliminates page table synchronization, the two-dimensional page table traversal will seriously degrade the address translation performance. Two-dimensional page table traversal incurs substantial performance penalties for address translation due to privilege overhead. This study proposes lazy shadow paging (LSP), which reduces page table synchronization overhead while retaining the high efficiency of shadow page tables. Leveraging the privilege model and hardware features of the RISC-V architecture, LSP analyzes the access patterns of guest OS page tables and binds synchronization with translation lookaside buffer (TLB) flushes, reducing the software overhead associated with page table updates by deferring costs until the first access to a relevant page to minimize VM-exits. In addition, it introduces a fast path for handling VM-exits, exploiting the fine-grained TLB interception and privilege-level features of RISC-V to further optimize performance. Experimental results demonstrate that under the baseline RISC-V architecture, LSP reduces VM-exits by up to 50% compared to traditional shadow paging in micro-benchmark tests. For typical applications in the SPEC2006 benchmark suite, LSP reduces VM-exits by up to 25% compared to traditional shadow paging and decreases memory accesses per TLB miss by 12 compared to nested paging.

Key words: RISC-V; hypervisor-extension; memory virtualization; shadow page table; nested page table

虚拟化技术作为云计算的核心支柱, 被广泛应用于云服务提供商, 用于管理云资源, 确保资源的共享、安全性和隔离性[1−3], 为服务器无感知计算 (serverless)[4,5]等新兴应用提供了基础设施支持和安全保证. 在虚拟化技术中, 内存虚拟化是最为关键的组成部分之一, 直接影响着虚拟机的访存性能和安全性. 一种高效的内存虚拟化方法, 能够提供与原生 (native) 环境相媲美的内存访问性能, 并且不产生额外的虚拟化开销, 最终提高虚拟机的性能. 然而, 尽管存在各种软件和硬件加速方案[6−8], 内存虚拟化的开销仍然不可忽视[9].

内存虚拟化技术使用两维地址映射为虚拟机提供隔离和完备的内存空间: 1) 从客户机虚拟地址 (guest virtual address, GVA) 到客户机物理地址 (guest physical address, GPA) 的映射; 2) 从 GPA 到宿主机物理地址 (host physical address, HPA) 的映射. 通常第 1 维地址映射由进程页表完成, 而对第 2 维地址的映射管理各不相同. 当前主流的内存虚拟化方法分为硬件辅助虚拟化和软件虚拟化. 首先, 最常用的硬件辅助虚拟化方法被称为嵌套页表模型 (nested paging)[2], 其直接使用页表结构维护第 2 维地址映射, 称为嵌套页表 (nested page table, NPT). 在嵌套页表模型下, 进程页表和 NPT 能够直接被装载到内存管理单元 (memory management unit, MMU) 中. 当发生转址旁路缓存 (translation lookaside buffer, TLB) 缺失时, MMU 将在这两套页表中进行高延迟两维页表遍历. 而在没有硬件辅助虚拟化的场景下, 最常用的内存虚拟化方法被称为影子页表模型 (shadow paging)[10]. 该方法采用物理地址翻译表维护第 2 维地址映射, 并且从进程页表和物理地址翻译表中构建一个映射 GVA 和 HPA 的影子页表 (shadow page table). MMU 直接装载影子页表进行地址翻译, 因此 TLB 缺失后将进行与原生场景相同的一维页表遍历.

嵌套页表模型和影子页表模型需要在地址翻译开销和页表同步开销中进行权衡, 但它们总是无法在两者间达到最优[11]. 如图 1 所示, 当启用嵌套页表模型时, 在 TLB 缺失后, GVA 需要首先通过遍历进程页表转化为 GPA, 然后再遍历嵌套页表转化为 HPA 进行访存. 虽然该二维页表遍历由硬件实现, 但是其仍然会造成不可忽视的开销. 例如, 在 RISC-V 架构中最常用的 Sv48 页表模型和 Sv39 页表模型下, TLB 缺失导致的页表遍历将分别造成 24 次和 15 次访存, 而在原生场景仅需要 4 次和 3 次访存. 不过, 嵌套页表模型受益于进程页表的直接更新, 无需虚拟机监视器 (hypervisor 或者 virtual machine monitor, VMM) 的介入. 具体而言, 虚拟机操作系统完全持有被装载到页表基地址中的进程页表的所有权, 对于页表页的修改和页表基地址的装载都与原生环境一致. 而第 2 维页表, 即嵌套页表, 保证了虚拟机的任意地址访存不会在宿主机物理地址空间中越界, 从而维护了安全性[3]. 当使用影子页表模型时, GVA 能够直接通过影子页表转化为 HPA, 产生的地址翻译开销和原生环境一致. 然而, 为了保证影子页表与进程页表和嵌套页表映射的一致性, 当虚拟机更新进程页表时需要 hypervisor 介入以进行复杂的页表同

步操作[6]. 具体而言, hypervisor 需要感知虚拟机对进程页表的修改, 这通常采用写保护机制完成 (write protection), 即在影子页表中将任意进程页表页的权限修改为不可写. 那么虚拟机中任何对于进程页表的修改都会产生虚拟机退出 (VM-exit), 从而造成严重的性能开销. 有研究表明, 在页表更新频繁的场景下, 嵌套页表模型的性能能够高于影子页表模型近 15%, 而在 TLB 缺失频繁的场景下, 影子页表模型又能够比嵌套页表模型性能提升近 32%[9].

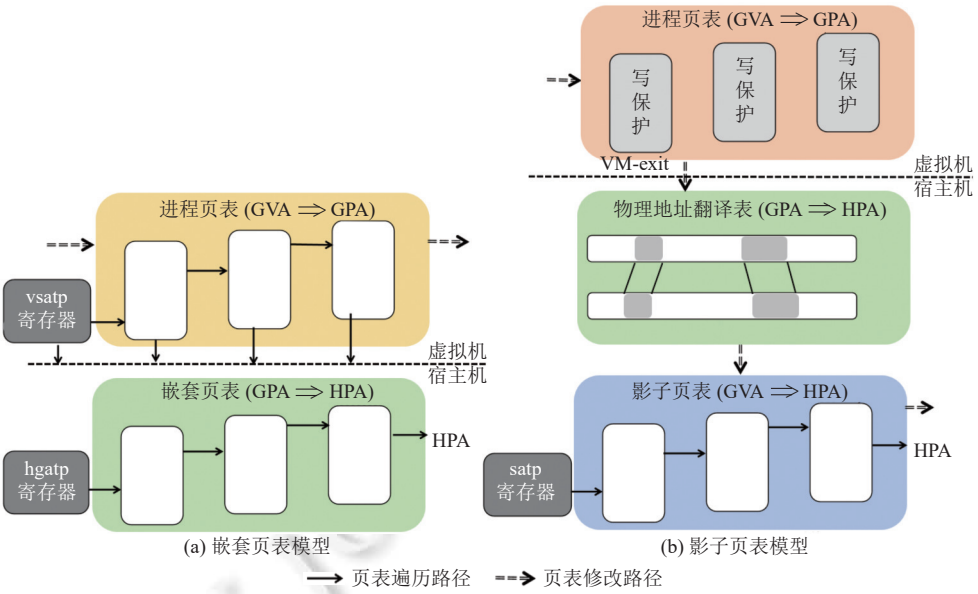


图 1 影子页表模型和嵌套页表模型架构图

对于 RISC-V 架构下的内存虚拟化研究, 该问题的主要难点有: 1) 高地址翻译效率和高页表同步效率: 当前通用方案无法在地址翻译开销和页表同步开销间达到最佳的权衡, 内存虚拟化技术仍然存在性能瓶颈. 2) 用户无感: 云服务提供商不应该对上层客户机操作系统做出任何假设和修改. 3) RISC-V 架构定制: RISC-V 架构与传统架构 (如 x86) 的特权模型和硬件支持各不相同, 应该充分利用 RISC-V 架构特征. 4) 支持主流 RISC-V 架构而无需硬件修改.

针对上述问题, 本文提出了一种基于 RISC-V 架构的懒惰影子页表模型 LSP (lazy shadow paging), 在保留影子页表的地址翻译高效性的同时降低了页表同步开销. 该工作首先对虚拟机操作系统中的页表修改行为进行总结和分析, 发现进程页表的修改操作可以分为两个子集, 其中一个子集中的进程页表修改操作总是与 TLB 刷新对应, 而另一个子集中的操作总是会与一个页面错误 (page fault) 对应, 因此懒惰影子页表模型通过增强页面错误处理程序功能和 TLB 刷新时进行页表同步来降低 VM-exit 的数量. 然后, 利用 RISC-V 架构中对 TLB 的细粒度刷新且可拦截的特性, 将被刷新页表项对应的影子页表项进行无效化的方式, 将页表同步操作的开销推迟到页面的访问时刻. 这种延迟同步的设计能够将对于同一个页表项的修改进行批量同步, 降低了虚拟机退出的软件开销. 最后, 利用 RISC-V 架构中全新的特权级模型, 将影子页表项无效化的操作卸载到最高特权级, 降低了上下文切换的开销. 多组对比实验验证了本文所提方法在地址翻译效率和页表同步效率上的有效性.

本文第 1 节介绍 RISC-V 下虚拟化技术的相关方法和研究现状. 第 2 节介绍本文所需的基础知识, 包括内存虚拟化技术和 RISC-V 架构特性. 第 3 节介绍本文构建的基于 RISC-V 架构的懒惰影子页表技术. 第 4 节通过对比实验验证了所提技术的有效性. 最后总结全文.

1 RISC-V 下的内存虚拟化相关工作

现有的内存虚拟化方法大多都基于传统体系结构进行探究. 然而, 对于 RISC-V 架构而言, 针对于内存虚拟化

的优化方法探究是非常少的. 而随着 RISC-V 架构在全球范围内的广泛应用和快速发展[12], 如何在该架构下实现高效的虚拟机内存虚拟化已成为当前的一个重要研究方向. 当前 RISC-V 架构下的内存虚拟化实现仍然沿用传统架构方案, 研究者只进行了一些功能性的探究. 因此利用 RISC-V 的硬件特性实现高效的内存虚拟化是一个值得探究的方向.

在内存虚拟化的相关优化工作中, 当前最前沿的解决方案通常考虑以修改硬件或者利用架构特性来结合嵌套页表模型和影子页表模型. Wang 等人[9]指出影子页表模型和嵌套页表模型都无法达到最优, 其根据实时监控的 TLB 缺失和页表错误数量动态切换硬件和软件页表模型以达到最佳性能, 然而切换页表模式总是需要重建整个影子页表, 这对于 GB 甚至 TB 级别的负载会造成严重的性能开销. Gandhi 等人[11]观察到虚拟机对于进程页表的修改总是倾斜分布在末级页表页, 并且对于页表页的修改满足双峰分布, 即页表页要么非常频繁地被虚拟机操作系统修改, 要么在相当一段时间内保持稳定. 因此他们提出灵活页表模型 (agile paging), 对于频繁修改的页表页将局部地切换为嵌套页表模式, 而保持稳定的页表页仍然使用影子页表模式. 但是这种局部嵌套页表的方法需要硬件修改, 难以应用在实际云服务环境中. Sha 等人[13,14]在国产申威架构下提出了敏捷影子页表模型 (swift shadow paging), 其利用了申威架构软件管理 TLB 的特性, 在硬件模式维护 TLB 表项的同时维护影子页表. 这种方法适用于支持软件管理 TLB 的架构, 而 RISC-V 架构仍然沿用硬件管理 TLB. Hill 等人[8]提出直接映射模式 (direct mode), 使用段式直接映射的方式维护以加速虚拟机地址翻译过程, 即从虚拟机物理地址到宿主机物理地址的映射由一个段基址加偏移完成, 避免了对嵌套页表的访问. 但是这种方法要求 hypervisor 分配给虚拟机的物理内存是连续的, 这会导致虚拟机内存容量无法动态调节, 也与一些虚拟机管理技术不兼容, 比如虚拟机页面共享, 虚拟机热迁移等, 实用性较低. Ahn 等人[15]提出了推测式扁平嵌套/影子页表模型 (speculative flat nested/shadow paging) 将嵌套页表和影子页表的扁平化设计, 即页表只保留一级, 地址翻译首先通过一次扁平影子页表的访问进行推测执行, 再由扁平嵌套页表进行验证. 这种方法将极大地增大页表的存储开销, 在大工作集场景并不适用. Sha 等人[16]也在国产申威架构上完成了扁平嵌套页表的实现. Park 等人[17]将嵌套页表的表项以缓存行粒度进行放置, 提高了页表遍历的缓存命中. 这些方案大多在传统架构下实现, 很少有对于 RISC-V 架构下内存虚拟化的优化探究. 表 1 展示了当前内存虚拟化学界前沿的页表模型在不同维度的权衡, 目前还没有针对 RISC-V 架构的内存虚拟化页表模型探究, 并且当前已有的解决方案要么需要激进的硬件修改, 要么也是针对其他架构的定制设计. 当前 RISC-V 下的 hypervisor 实现仍然沿用传统的通用影子页表或者嵌套页表模型.

表 1 内存虚拟化中页表模型的不同权衡

页表模型	硬件修改	架构特性	地址翻译效率	页表同步开销	页表存储开销
灵活页表[11]	是	无	较高	较低	较低
敏捷页表[13]	否	申威架构	高	低	低
直接映射模式[8]	是	无	极高	无	无
推测式扁平嵌套/影子页表[15]	是	无	极高	较低	较低
懒惰影子页表	否	RISC-V架构	高	低	低

在 RISC-V 架构下的虚拟化技术研究中, 当前解决方案通常聚焦在虚拟化技术整体的功能完备性和安全隔离性等方面, 内存虚拟化方法仍然沿用传统架构的影子页表模型或者嵌套页表模型. Behrens 等人[18]在无虚拟化扩展的 RISC-V 架构下实现了 RVirt, 其通过陷入-模拟的方式在 U-mode 运行虚拟机, 采用传统影子页表模型实现了内存虚拟化. Martins 等人[19]针对嵌入式系统, 实现了一套满足实时性, 安全性和可靠性的 bao-hypervisor, 其中对于中断虚拟化进行了定制优化, 达到较高的性能, 但是其内存虚拟化仍然使用传统模型. Patel 等人[20]实现了支持全虚拟化和半虚拟化的 Xvisor, 已支持在 RISC-V 架构下以较为轻量级的方式在嵌入式场景运行虚拟机, 但是仍然使用传统影子页表模型和嵌套页表模型进行内存虚拟化.

2 基础知识

本文工作主要是关于 RISC-V 架构下的内存虚拟化实现, 下面就相关概念和基本知识予以介绍.

2.1 RISC-V 架构硬件特性

近年来, RISC-V 架构迅速发展, 从最初的学术研究指令集逐渐演变为在工业界被广泛应用的流行架构[12]. 随着 RISC-V 的普及, 业界对将其他架构中的先进技术适配到 RISC-V 的兴趣也在不断增加. 例如, Arm64[21]和 x86[22]架构早已实现了特定的硬件来辅助虚拟化. 在近年来, RISC-V 的虚拟化扩展 (hypervisor extension, H-extension) 也正式通过并纳入了特权架构规范[23]. 这一扩展定义了可以在处理器核心中实现的硬件功能, 以减少虚拟化的开销并简化 hypervisor 的实现. RISC-V 在初期并没有类似的硬件支持, 因此, 像 RVirt[18]这样的 hypervisor 只能依赖于陷入-模拟机制, 通过特权模式来实现虚拟化分离, 这意味着虚拟化的所有处理都必须由软件独立完成. 与传统架构相比, RISC-V 架构拥有全新的特权级模型和硬件特性, 因此在虚拟化技术的研究和应用中, 应当充分利用这些特性, 以实现 RISC-V 定制的技术实现.

2.1.1 特权级模型

RISC-V 的基础架构定义了一种 3 层的特权模式, 用于管理不同的执行环境, 如图 2 所示, 分别是机器模式 (M mode)、监督模式 (S mode) 和用户模式 (U mode). 每个硬件线程 (Hart) 都在它自己的特权模式下运行. 每个特权级别都有自己的控制和状态寄存器 (control status register, CSR) 来控制和检测中断、异常委托、地址转换等内容的状态. 所有 CSR 都有一个具有它们属于的特权级别的前缀, 例如 M mode 的 CSR 有前缀“m”, S mode 有“s”, 等.

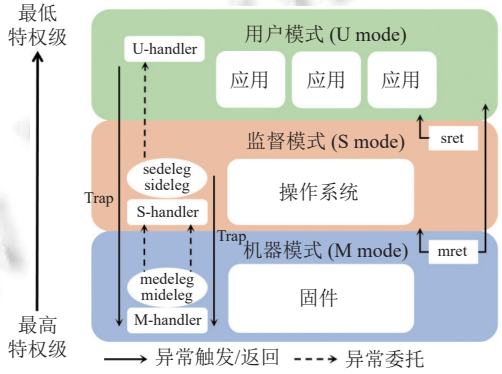


图 2 RISC-V 基础架构定义的特权模型

M mode 是 RISC-V 架构中最高级别的特权级, 负责控制关键资源, 其允许直接访问所有硬件资源, 包括配置和管理其他特权级别的运行环境, 以及软件完成一些在硬件中实现起来太困难或太昂贵的功能. 在 M mode 下运行的代码应当是可信的, 因为它可以访问系统中的所有内容. 硬件模式通常会预留给具有多个特权级别的系统中的低级固件实现, 为上层提供与 M mode 专有资源交互的接口. 当前 RISC-V 架构制定了标准的监督模式二进制值接口 (supervisor binary interface, SBI), 向上提供了一套统一的执行环境[24]. U mode 和 S mode 通常用于应用程序和操作系统. 相比之下, 传统的 x86 架构采用了环形特权级别设计, 通常包括 4 个环 (Ring 0 到 Ring 3), 其中 Ring 0 具有最高权限, 操作系统直接运行在最高特权级下[22].

M mode 下的程序对系统拥有完全的控制权. 当 S mode 或者 U mode 触发了陷阱 (trap) 时, 在默认配置下, 硬件会直接陷入 M mode 进行处理. 而若配置了 M mode 中断和异常授权寄存器 (medeleg 和 mideleg 寄存器), 该陷阱才会被代理到 S mode 进行处理. 进一步的, 若还配置了监督模式中断和异常授权寄存器 (sedeleg 和 sideleg 寄存器), 该陷阱还会被代理到 U mode 进行处理.

RISC-V 架构使用地址翻译和保护寄存器 (satp 寄存器) 控制和管理地址翻译. 在 S mode 或者 U mode 下, 该寄存器保存了页表基地址、地址空间描述符和页表模式. 当访问虚拟地址时, MMU 会通过该寄存器把虚拟地址翻译成物理地址. 而在开启内核页表隔离或者虚拟机退出的时候, 总是需要进行页表切换并且刷新 TLB, 从而导致严重的性能下降. 而在 M mode 下, 访存直接通过物理地址完成, 无需通过页表遍历和地址翻译的过程.

在 RISC-V 架构下, 操作系统所在的 S mode 可以使用 SFENCE.VMA 指令刷新 TLB, 例如在页表更新或者切

换页表基地址的时候. 该指令功能非常强大, 即可以刷新全局所有 TLB 表项, 也可以细粒度地指定地址空间标识符或者要刷新的虚拟地址.

2.1.2 虚拟化扩展

RISC-V 架构中已经正式通过并纳入了一个虚拟化扩展[23], 该扩展详细描述了 RISC-V 如何处理和实现与虚拟化相关的寄存器和特权模式, 以提升虚拟化效率和简化 hypervisor 的实现. 具体而言, 在特权级模式方面, 虚拟化扩展引入了虚拟监督模式 (VS mode) 和虚拟用户模式 (VU mode). 同时, 传统的 S mode 下新增了一套关于该扩展的寄存器和指令集, 且更名为虚拟化扩展监督模式 (HS mode). 通常而言, VS mode 下将运行虚拟机的操作系统. 而在 VU mode 下, 虚拟机进程的行为与常规用户模式类似, 不同之处在于陷阱 (traps) 和系统调用 (syscalls) 将被发送到 VS mode, 而不是 HS mode. 该虚拟化扩展的主要优势在于, 它能够自动处理 VS mode 中控制状态寄存器的读写操作 (带有“s”前缀的寄存器), 将这些操作自动映射到扩展的 VS mode 寄存器上 (带有“vs”前缀的寄存器). 这使得虚拟化的实现更加简单, hypervisor 无需再拦截虚拟机中任何对敏感寄存器的访问, 图 3 对比了是否具有虚拟化扩展的 RISC-V 架构下实现的虚拟化技术.

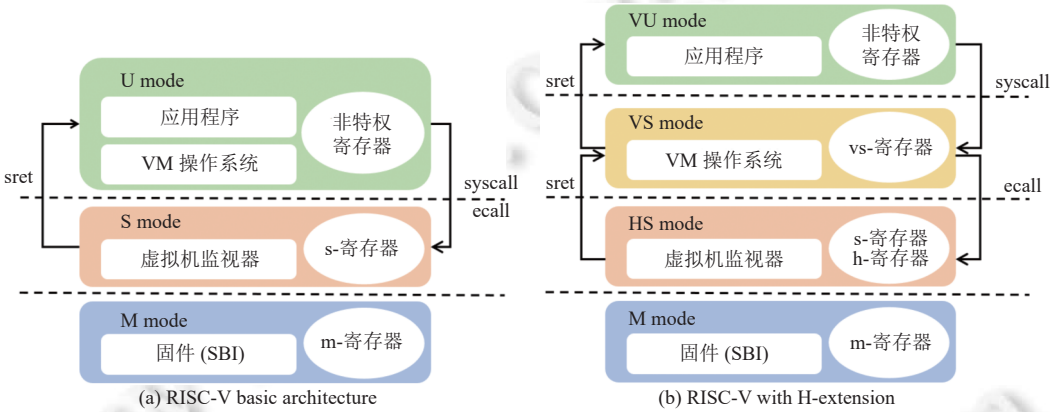


图 3 虚拟化技术在虚拟化扩展 RISC-V 和基础 RISC-V 架构下的实现

尽管扩展了多个特权级, M mode 仍然是具有最高权限的特权级. 当在 VS 或者 VU 模式触发陷阱时, 硬件仍然会陷入 M mode, 若配置了机器模式中断和异常授权寄存器 (medeleg 和 mideleg 寄存器), 会被委托到 HS 模式处理. 进一步地, 若还配置了虚拟机中断和异常授权寄存器 (hedeleg 和 hideleg), 还会被委托到 VS 模式进行处理.

虚拟化扩展中对嵌套页表模型进行了硬件支持. 与传统 x86 的扩展页表机制 (extended page table, EPT) 类似, RISC-V 虚拟化扩展中也使用两个基地址寄存器 (vsatp 和 hgatp 寄存器) 分别保存虚拟机中的进程页表基地址和 hypervisor 中的嵌套页表基地址. 当一个硬件线程处于 VS 或者 VU mode 时, 嵌套页表模型总是被开启, 即 GVA 需要通过两维页表遍历才能够转化为 HPA. 不过, 虽然当前不支持关闭嵌套页表模型, 但是可以将任意基地址寄存器中的模式位置零来禁用任意一级的页表遍历以达到关闭嵌套页表模型的效果.

虚拟化扩展通过引入虚拟指令异常 (virtual instruction exceptions) 来控制 VS 模式下的一些敏感指令的执行. 虚拟指令异常是与传统的非法指令异常 (illegal instruction exceptions) 进行区分. 在虚拟机状态寄存器 (hstatus 寄存器) 中可以细粒度地调整产生虚拟指令异常的行为. 例如该寄存器中的虚拟机内存陷入字段 (virtual trap virtual memory, VTVM) 将控制在 VS 模式下刷新 TLB 和访问 satp 寄存器的行为是否会产生虚拟指令异常. 这增强了 hypervisor 对于虚拟机细粒度管理能力.

2.2 内存虚拟化技术

虚拟化技术通过引入一个新的系统软件抽象层 (即 hypervisor), 以控制虚拟机对物理资源的访问. 每个虚拟机都拥有独立的虚拟硬件抽象 (如 CPU、内存等), 从而构建出一个完整且隔离的执行环境. 为了给虚拟机提供隔离且连续的物理地址空间, hypervisor 引入了客户机物理地址空间的概念. 客户机物理地址空间使得虚拟机能够获得

类似于原生物理地址空间的功能, 虚拟机中的进程页表将客户机虚拟地址 (GVA) 映射到客户机物理地址 (GPA)[11,13,14,25]. 而从 hypervisor 的角度来看, 每个虚拟机都是一个在宿主机中运行的进程, 因此拥有一个对应的宿主机进程页表, 用于维护宿主机虚拟地址 (HVA) 到宿主机物理地址 (HPA) 的映射 (例如, 在 QEMU/KVM 作为 hypervisor 的场景下, QEMU 的进程页表). 通常, GPA 和 HVA 是线性映射的, 因此 hypervisor 通常会直接维护从 GPA 到 HPA 的页表映射. 内存虚拟化在不同的硬件环境下通常采用不同的实现方案, 主要包括基于纯软件的影子页表模型[10]和基于硬件辅助的嵌套页表模型[2]. 表 2 总结并对比了在 RISC-V SV39 页表下的内存虚拟化技术, 包括本文实现的懒惰影子页表模型.

表 2　SV39 页表下的内存虚拟化解决方案对比

解决方案	TLB命中	TLB缺失后的访存次数	页表同步	硬件支持
原生场景	快 (HVA ⇒ HPA)	3	快 (直接修改页表)	MMU
影子页表模型	快 (GVA ⇒ HPA)	3	慢 (由hypervisor介入)	MMU
嵌套页表模型	快 (GVA ⇒ HPA)	15	快 (直接修改页表)	MMU+H-extension
懒惰影子页表模型	快 (GVA ⇒ HPA)	3	中 (批量修改, 快速路径)	MMU

嵌套页表模型是一种广泛应用的硬件辅助内存虚拟化技术. 当前 RISC-V 的虚拟化扩展已经支持了该方法. 当启用嵌套页表模型后, MMU 使用两个页表基地址寄存器来完成地址转换: 一个指向应用的进程页表 (vsatp), 另一个指向每虚拟机的嵌套页表 (hgatp), 从而实现了从 GVA 到 GPA 再到 HPA 的地址翻译过程.

在理想情况下, 地址转换可以通过 TLB 命中直接从 GVA 转换为 HPA, 从而不会产生额外的开销. 然而, 在最坏的情况下, 如果 TLB 未命中, 将触发一个二维的页表遍历, 这相比于原生地址翻译会显著增加开销, 因为每次访问客户机进程页表都需要通过嵌套页表进行额外的遍历. 在 SV39 页表模型下, 地址翻译的内存引用次数将从原生环境下的 3 次增加到虚拟化环境下的 15 次. 具体而言, 翻译 vsatp 寄存器中的基地址需要 3 次访问 (因为每个 GPA 都需要访问嵌套页表), 再加上 3 级虚拟机进程页表的每一级访问, 最终得到 HPA, 共计 3×4+3=15 次内存引用. 当前, 现代商用处理器中的各种缓存机制, 如数据缓存[26]、MMU 缓存[27,28]以及翻译缓存[2,3], 可以减少 TLB 未命中时所需的内存引用次数, 不过本文并不讨论它们的影响. 虽然在虚拟化环境下 TLB 未命中的开销比原生环境更大, 但是嵌套页表模型允许虚拟机操作系统直接更新进程页表, 而无需 hypervisor 的干预.

影子页表模型是一种纯软件内存虚拟化技术, hypervisor 通过将虚拟机中的进程页表和嵌套页表合并按需创建一个影子页表, 直接保存从 GVA 到 HPA 的完整翻译.

在理想情况下, 虚拟化地址转换可以通过命中 TLB 直接从 GVA 翻译到 HPA, 而不会产生额外的开销. 然而, 如果 TLB 未命中, MMU 只需要对影子页表执行一维的页表遍历. 此时, 页表基地址寄存器将直接指向影子页表, 所以页表遍历所需的内存引用次数与原生场景下页表遍历相同. 尽管 TLB 未命中的开销与原生执行相同, 但是由于虚拟机操作系统不允许直接更新影子页表, 其对进程页表的操作无法直接影响到影子页表[6]. 因此, 为了保证进程页表和影子页表的一致性, 虚拟机每次对进程页表的更新都需要进行一次高开销的虚拟机退出以更新影子页表的条目. 最常用的页表同步方法称为写保护 (write-protection). 具体而言, hypervisor 将虚拟机中的进程页表的页表页在影子页表中都标记为只读, 这样在虚拟机中每次尝试对进程页表的修改都会触发一次写权限异常, 从而被 hypervisor 感知到并进行同步操作. 通过这种方法对影子页表的更新需要两次 VM-exit, 在传统架构下会消耗数千个周期: 一次触发写权限异常进行页表同步, 另一次用于刷新 TLB[11].

3　RISC-V 架构下的懒惰影子页表模型

鉴于全新的特权级模型和硬件特性, 本文在 RISC-V 架构下提出了懒惰影子页表模型, 以解决虚拟化技术在内存虚拟化方面的性能瓶颈.

懒惰影子页表模型的设计目标如下: 1) 在地址翻译效率和页表同步开销之间获得更好的权衡. 传统软硬件解决方案无法在这二者间达到最佳权衡, 虚拟化技术的性能严重受限于内存虚拟化, 应该基于 RISC-V 架构的全新特性寻求更佳的权衡; 2) 适配主流 RISC-V 架构规范而无需硬件修改. 当前的 RISC-V 架构基本已经形成较为成熟和稳定的规范, 懒惰影子页表模型应当完全适配当前的硬件规范而无需任何的硬件修改. 同时, 考虑到一些硬件受限场景可能并不会实现虚拟化扩展架构[29], 该方案应该能够在无虚拟化扩展的场景使用并获得性能提升; 3) 用户透明. 云服务提供商对于方案的落地考虑最重要的因素就是是否对用户友好, 因此该方案不应该对客户机操作系统做出任何限制. 懒惰影子页表模型是关于内存虚拟化的优化方案, 因此对于 IO 虚拟化、CPU 虚拟化和中断虚拟化的实现, 我们不做任何限制和优化, 也并非我们的设计目标.

懒惰影子页表模型为了达到最佳的地址翻译效率, 与传统影子页表模型相同, 在 TLB 缺失后并不使用两级页表遍历, 而是只启用一个页表基地址寄存器装载影子页表进行地址翻译. 当然, 为了保证虚拟机权限可控, hypervisor 需要拦截虚拟机内部任意对于页表基地址的访问, 任意读写页表基地址的指令都应该被视为敏感指令. 由于直接使用影子页表, 虚拟机中任何对进程页表的更新都需要被同步到影子页表中.

传统影子页表模型在页表同步期间会导致严重的开销, 而懒惰影子页表模型通过 3 个部分降低页表同步造成的开销, 如图 4 所示. 首先, 基于用户和操作系统对于页表的访问, 总结出页表同步以及 TLB 刷新之间的关系特征并且对页表同步时机与 TLB 刷新进行绑定. 然后, 基于页面访问的局部性原理, 懒惰影子页表在页表同步的时候仅无效化影子页表项, 真正同步操作推迟到具体访问时的页面错误进行. 最后, 基于 RISC-V 的特权级模型, 懒惰影子页表对 TLB 刷新操作的拦截将使用快速路径, 直接在最高特权级进行页表项无效化.

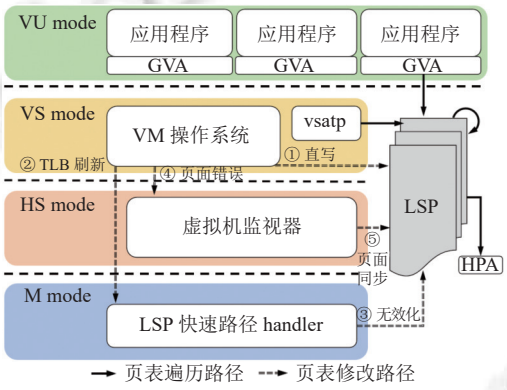


图 4 基于 RISC-V 架构的懒惰影子页表模型框架图

3.1 页表同步时机绑定

对于传统影子页表的同步机制而言, 通常采用将虚拟机中的进程页表页进行写保护的方式完成同步. 当虚拟机操作系统每次尝试更新进程页表的时候, 都将触发一次由写权限异常导致的 VM-exit 并在本次异常处理程序中完成对相应影子页表的修改. 而在修改完成后, 虚拟机还会执行一次 TLB 刷新操作以保证正确性, 该操作会再触发一次由敏感指令错误导致的 VM-exit.

而我们通过对页表修改行为的分析发现, 任何对于页表项的修改都会有与之绑定的后续操作, 即 TLB 刷新或者页面错误. 如表 3 所示, 客户机操作系统对于进程页表的修改主要分为两大类, 一类是进行映射修改, 一类是进行权限位修改. 映射修改操作又分为新增映射 (即 map 操作) 和删除/修改映射 (即 unmap/remap 操作), 权限位修改操作又分为提升权限 (如从不可写修改为可写) 和降低权限 (如从可写修改为不可写). 若允许影子页表和进程页表存在不一致, 即虚拟机可以随意修改进程页表, 但是这些修改不被同步到影子页表中, 那么新增映射操作和提升权限操作实际上并不会导致虚拟机崩溃, 因为这些进程页表项对应的影子页表项总是无效或者具有更低权限的. 当以高权限进行访问时, 会产生一次权限问题导致的虚拟机退出, hypervisor 可以在这里进行介入. 但是, 如果虚拟机

删除或修改映射和降低权限, 由于影子页表中仍然保留的是原来映射或者权限的表项, 虚拟机就会访问错误的内存地址, 从而造成虚拟机崩溃. 然而, 在任何会导致虚拟机崩溃的操作发生时, 客户机操作系统总是会刷新 TLB 以保证 TLB 和页表的一致性.

表 3 客户机操作系统页表修改行为分析

修改分类	页表行为	是否需要刷新TLB	页表不一致后果
映射修改	新增映射	否	产生由未映射导致的虚拟机退出
	删除/修改映射	是	错误, 虚拟机直接访问无权限内存区域
权限位修改	提升权限	否	产生由无权限访问导致的虚拟机退出
	降低权限	是	错误, 虚拟机直接访问无权限内存区域

因此, 懒惰影子页表摒弃了传统写保护的同步方式, 而是通过拦截虚拟机进行页表修改后的绑定操作进行页表同步. 这样一来, 无需执行 TLB 刷新的页表更新操作 (即新增映射和提升权限) 总是会在后续的虚拟机退出中重新进行页表同步, 而需要执行 TLB 刷新的页表更新操作 (即删除/修改映射和降低权限) 就可以在执行 TLB 刷新产生的虚拟机退出处理程序中进行页表同步, 保证了虚拟机执行的安全性. 由此, 该方法可以将传统页表同步产生的两次虚拟机退出减少至一次虚拟机退出.

客户机操作系统对进程页表进行修改后, 总是需要将行为同步到影子页表中以保证地址转换的正确性. 而在这之外, MMU 也会主动对页表进行修改, 例如置位访问位或者脏位. 该修改场景下的页表同步方向与客户机操作系统修改场景下的页表同步方向相反, 因为 MMU 将直接读写影子页表, 软件需要将硬件对于影子页表的修改同步到进程页表以保证客户机操作系统的稳定性. 传统影子页表已经具有较为成熟的解决方案对访问位和脏位进行同步. 当客户机操作系统开始监控访问位时, hypervisor 将被监控页面的影子页表项的存在位 (present bit) 标记为 0, 这样客户机任何对于该页面的访问都将触发一次 VM-exit 进行访问位同步. 当客户机操作系统开始监控脏位时, hypervisor 将被监控页面的影子页表项的写权限位 (writable bit) 标记为 0, 这样客户机任何对于该页面的写都将触发一次 VM-exit 进行脏位同步. 这类从影子页表向进程页表的同步过程通常并不会被频繁触发, 并且信息位的不一致后果通常不会导致严重的虚拟机崩溃. 因此对于该类同步我们仍然沿用传统的同步方式, 而懒惰影子页表将针对更加关键的进程页表向影子页表同步过程进行优化.

3.2 页表同步行为推迟

在传统影子页表模型下, 进程页表和影子页表之间不存在不一致的时刻, 在虚拟机准备修改进程页表之前, 就已经发生写错误异常从而同步更新了影子页表. 即使使用页表同步机制绑定, 二者之间的同步也是同时更新并且一一映射的. 在页表同步的实现中, hypervisor 需要软件遍历进程页表, 影子页表和第 2 级页表进行同步, 该操作会导致高额的软件开销. 特别是相比于传统架构, RISC-V 的虚拟机退出更加轻量级, 而执行高开销的页表同步操作就更成为了性能瓶颈. 在传统方法下, 每次页表更新操作都一定会对应一次该高开销的多维页表遍历进行页表同步.

页面访问具有局部性, 通常 80% 的时间会聚集在访问 20% 的页面上, 而分析操作系统对于页表的修改可以发现, 在一些页表修改的场景并不存在该性质 (比如系统在监控页面时发生的周期性地全局读写访问位和脏位, 进程 fork 时发生的全局写时复制). 同时, 如果虚拟机并不访问发生修改的页表项, 该不一致性实际上并不会导致任何访存错误和虚拟机崩溃. 因此懒惰影子页表将该高开销的同步操作延迟到页面发生访问时进行. 具体而言, 当拦截到 TLB 刷新时, 懒惰影子页表只将对应的影子页表项无效化, 该操作不需要遍历进程页表和第 2 级映射页表, 更加轻量级.

当虚拟机使用这些无效化的页表项进行地址翻译时, 会触发一次虚拟机退出, 在此刻进行页表同步操作. 由此, 懒惰影子页表将真正的页表同步操作从修改时刻推迟到了使用时刻, 该时刻已经是页表同步操作的最晚时刻, 在修改时刻和使用时刻之间, 由于该页表项并不会被访问, 所以也不存在安全问题. 这样做的好处在于如果对于某一个进程页表项存在频繁的更新操作, 而对于该表项映射的内存访问并不频繁的话, 这些更新操作可以完全被批

量处理, 都会推迟到页表项使用时刻使用最新的进程页表项进行影子页表同步.

3.3 页表同步快速路径

在使用页表同步行为推迟后, 一次页表同步仍然会发生两次虚拟机退出, 第 1 次用于无效化影子页表项, 第 2 次则是发生页面错误后的页表同步. 虽然该方法可以批量化一些页表同步操作的发生, 但是却增加了一次虚拟机退出的开销. 虽然在 RISC-V 架构下的虚拟机退出相较传统架构更加轻量级, 但是仍然需要在退出后保存虚拟机上下文. 而懒惰影子页表中页表同步的首次虚拟机退出只需要进行轻量级的页表无效化操作, 上下文切换的开销就成为了瓶颈.

因此, 懒惰影子页表模型针对页表项修改发生的第 1 次虚拟机退出设计了页表同步快速路径. 具体而言, 懒惰影子页表模型在拦截 TLB 刷新操作后直接在最高特权级中处理影子页表项的无效化操作. 首先, 在 M mode 可以直接使用物理地址访存而无需页表遍历和查询 TLB, 避免了 TLB 污染. 然后, 由于页表同步中首次虚拟机退出只需要进行简单的页表无效化操作, 退出到 M mode 后仍然保留虚拟机上下文, 只保存少数需要直接使用的通用寄存器即可完成处理. 由此, 页表同步操作主要分为: (1) 轻量级的快速路径对影子页表项进行无效化; (2) 虚拟机退出对页表进行批量同步. 这极大地提升了传统影子页表中对页表同步的效率.

3.4 懒惰影子页表模型实现

在 RISC-V 基础架构下, 懒惰影子页表模型的实现只需要将传统影子页表模型的页表同步机制进行修改即可. 具体而言, 由于在该架构下虚拟机操作系统也运行在 U mode, TLB 刷新操作本身就会产生一次指令错误的 trap, 在其对应的处理函数中即可进行影子页表无效化的实现. 而在发生页面错误时, 虚拟机同样会直接退出到 hypervisor 中, 在页面错误处理程序中即可检查发生错误的 GVA 对应的进程页表项和影子页表项与第 2 级映射页表, 检查是否发生了权限错误或者映射修改, 进行对应的同步即可. 由于在该配置下, 直接将指令错误的 trap 陷入到 M mode 会大范围影响其他类型指令错误的处理, 当前也无法精确地将 TLB 刷新操作陷入到 M mode, 我们并没有在该配置下实现快速路径.

在具有 H-extension 的 RISC-V 架构下, 懒惰影子页表模型选择将 hypervisor 中的 hgatp 寄存器置零来禁用两级页表遍历中对于嵌套页表的遍历过程, 同时在 vsatp 寄存器中直接装载从 GVA 映射到 HPA 的影子页表基地址. 那么, 当 TLB 缺失后, 硬件只会使用 vsatp 寄存器中的基地址进行一维的影子页表遍历. 同时, 为了保证 vsatp 寄存器中的页表基地址对虚拟机不可见, 我们通过设置 hstatus 寄存器中的 VTVM bit 直接拦截虚拟机中任何对该寄存器的读写而返回其请求的进程页表的基地址, 同时该位的设置也会拦截 TLB 刷新的操作, 这与我们的设计不谋而合. 而对于懒惰影子页表快速路径的实现, 我们通过 medeleg 寄存器直接将虚拟指令错误导致的异常保留在 M mode 进行处理, 由于只在 M mode 中进行简单的页表无效化操作, 所以快速路径直接由高性能汇编代码实现, 同时保留虚拟机的上下文, 仅类似于进行一次函数调用.

在 RISC-V 架构下, 可以选择刷新指定的 TLB 条目或者全局刷新, 理想情况下, 对于页表项修改后, 客户机操作系统应当仅刷新对应的表项而非全局刷新, 因此懒惰影子页表也能够获得该次刷新对应的页表项的 GVA. 而一旦直接进行全局 TLB, 懒惰影子页表则认为是进行了一次页表的大范围调整, 因此不再进行同步行为延迟, 而是直接陷入 hypervisor 在此时进行页表同步操作.

4 实验分析

4.1 测试环境

我们使用 QEMU 7.1.0[30]模拟器构建 RISC-V 硬件环境, 该版本模拟器已经提供 H-extension 支持. 实现了懒惰影子页表模型的 hypervisor 直接运行在该模拟硬件环境下以验证有效性. Hypervisor 总共管理 16 GB 物理内存. 运行在 hypervisor 上的虚拟机配置为使用 8 GB 物理内存. 我们通过微基准测试在 RISC-V 基础架构上测试和对比传统影子页表模型和懒惰影子页表模型; 选取 SPEC2006 测试集[31]中的典型应用: 403.gcc, 429.mcf, 433.milc,

473.astar 在具有 H-extension 扩展的 RISC-V 架构下进行测试. 所选取的典型应用都是内存密集型程序. 而由于 hypervisor 在模拟器上运行, 程序的运行时间指标无法展示实际的性能情况, 因此对于每个测试, 我们使用 TLB 缺失数量或 VM-exit 数量来进行对比.

为了评估懒惰影子页表模型的有效性, 我们研究了以下几个问题:

- 在 RISC-V 基础架构下懒惰影子页表模型较传统影子页表模型的表现.
- 在 H-extension 扩展下懒惰影子页表较传统影子页表模型和嵌套页表模型的表现.

4.2 基础架构下的性能分析

在 RISC-V 基础架构下, hypervisor 通过陷入-模拟的方式运行功能受限的虚拟机. 因此, 我们在此实现下运行微基准测试. 为了测试懒惰影子页表模型对页表同步操作的性能提升效率, 虚拟机将对 1 024 个页面进行顺序操作, 共计 10 万次. 对这些页面的操作分为直接访存和修改对应页表项, 我们在不同的操作占比下观察懒惰影子页表模型相较于传统影子页表模型所产生的访存相关的 VM-exit 数量变化.

如图 5 所示, 随着页表修改操作占比的增加, 传统影子页表模型与修改操作的数量呈线性相关, 当全部操作都是页表修改时, 传统影子页表模型会产生 20 万次的 VM-exit, 包括写保护和 TLB 刷新操作导致的 VM-exit. 而懒惰影子页表模型将页表同步操作与 TLB 刷新操作绑定, 并且将页表同步的软件开销推迟到首次访问时产生, 因此当页表操作占比远大于页面访问操作时, 懒惰影子页表模型避免了对未被访问页面的页表同步操作, 最高能降低 50% 的 VM-exit 数量.

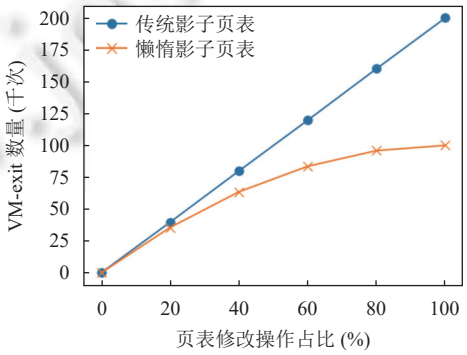


图 5 不同页表修改操作占比下的 VM-exit 对比

我们还模拟了真实场景下虚拟机操作系统周期性地对页表项进行修改的行为. 虚拟机操作系统将对 1 024 个页面对应的页表项中的访问位和脏位进行周期性地写零, 同时还会对这些页面进行满足二八分布 (即 80% 的时间访问 20% 的页面) 的访问. 我们调整周期性全局写页表项的时间窗口大小, 比较懒惰影子页表模型相较于传统影子页表模型所产生的 VM-exit 数量变化. 时间窗口大小代表着数倍于页面数量的访存次数 (例如时间窗口为 2 表示经过 1024×2 次访存后全局写一次页表项), 统计 10 个时间窗口内产生的结果.

如后文图 6 所示, 传统影子页表模型始终会产生 2 万次的 VM-exit 数量, 这是因为每个时间窗口周期后的每次页表项修改都将产生 2 次 VM-exit. 而懒惰影子页表模型与访问的页面数量有关. 在时间窗口较小时, 懒惰影子页表模型能够降低近 32% 的 VM-exit 数量.

4.3 虚拟化扩展架构下的性能分析

在具有虚拟化扩展的架构下, 我们实现的 hypervisor 更加完备, 能够直接运行 Linux 内核. 因此, 我们直接运行 SPEC2006 测试集中的典型的内存密集型程序: 403.gcc, 429.mcf, 433.milc, 473.astar, 对比懒惰影子页表模型、传统影子页表模型和嵌套页表模型的表现. 由于测试环境是在 x86 架构下进行模拟的, 因此程序的运行时间没有参考性. 我们仅对比关键的访存指标: TLB 缺失数量和访存相关的 VM-exit 数量.

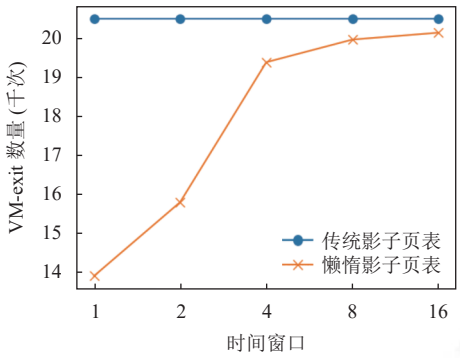


图 6　不同时间周期下的全局页表读写产生的 VM-exit 对比

如图 7 所示, 使用懒惰影子页表模型相较于嵌套页表会导致 14%–32% 的 TLB 缺失, 这是由更多的上下文切换导致的. 然而, 在嵌套页表模型下, 每次 TLB 缺失将导致额外的 15 次访存, 而懒惰影子页表模型只会产生额外 3 次访存. 因此, 虽然懒惰影子页表模型导致的 TLB 缺失的数量增多了, 但是实际导致的惩罚远远小于嵌套页表模型. 使用懒惰影子页表模型和传统影子页表模型会导致更多的 VM-exit, 而在嵌套页表模型下几乎不存在访存相关的 VM-exit. 这是因为虚拟机只会在构造嵌套页表的时候才会产生 VM-exit, 而每个虚拟机只维护一个嵌套页表, 当虚拟机预热后几乎不会再产生相关 VM-exit. 而虚拟机将为每个进程维护不同的影子页表, 进程每次运行都需要重新建立影子页表, 并且虚拟机对影子页表的访问都会被拦截. 因此懒惰影子页表造成的 VM-exit 会高于嵌套页表. 与传统影子页表相比, 懒惰影子页表能够降低最多 25% 的 VM-exit. 这得益于懒惰影子页表提供的页表同步绑定和推迟机制.

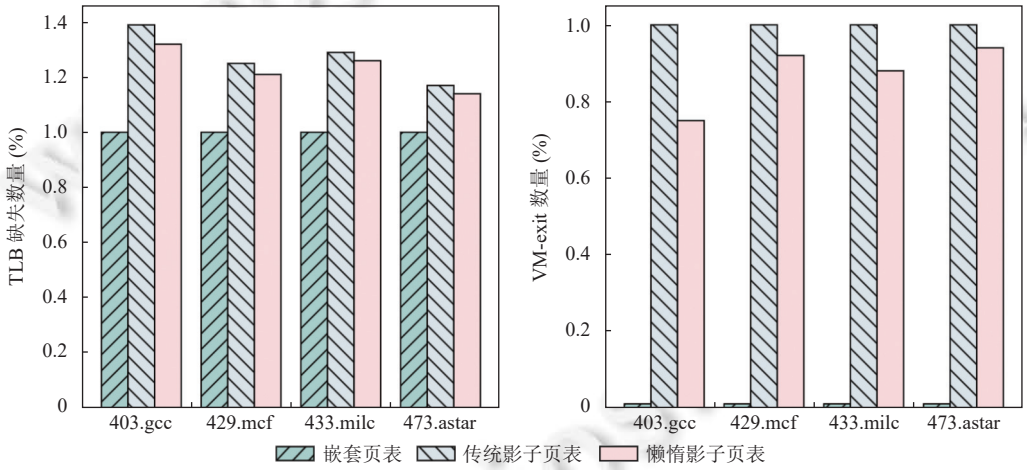


图 7　不同页表模型在访存密集型程序下的 TLB 缺失和虚拟机退出数量

5 讨　论

5.1 懒惰影子页表安全性分析

懒惰影子页表通过分析页表同步行为推迟了进程页表向影子页表的同步时刻, 并且利用 RISC-V 架构下的最高特权级加速页表同步的上下文切换开销. 这两个核心设计并没有引入额外的安全问题.

对于推迟同步机制, 恶意用户可能在虚拟机中运行恶意操作系统, 通过修改进程页表后不主动刷新 TLB 而继续通过过时的影子页表项访问未被映射的内存. 但是这里仅发生了虚拟机内用户态越权访问虚拟机内核态内存,

而对于 hypervisor 而言, 该段内存仍然被该虚拟机占有, 并不会用于保存其他数据. 而被 hypervisor 回收的页面一定会通过其引用计数保证其不会存在于影子页表映射中. 因此虚拟机并不会越权访问宿主机内存.

对于快速路径机制, 在最高特权级引入的代码都是属于 hypervisor 的实现, 虚拟机仅能通过刷新 TLB 的指令进入该段代码. 该段代码的输入仅包含客户机虚拟地址, 然后遍历由 hypervisor 维护的影子页表, 无效化该地址对应的影子页表项. 通过简单的边界检查即可保证虚拟地址没有越界, 并且该段代码不会执行虚拟机提供的任何内存地址, 仅作为一段插入最高特权级的只访问可信数据的快速代码片段. 当然, 若在该段代码中添加其他功能, 该代码明确的功能也能够快速地完成形式化验证, 只需要验证对于所有虚拟地址的输入都能够产生合法的结果即可.

5.2 懒惰影子页表限制

懒惰影子页表是针对 RISC-V 架构定制的内存虚拟化解决方案, 需要利用 RISC-V 架构中的最高特权级, 细粒度刷新 TLB, 异常委托等特性来加速页表同步的上下文切换开销. 而对于没有该特性的架构, 例如 x86 或者 ARM 架构, 懒惰影子页表无法发挥出最大的效率. 同时, 懒惰影子页表要求 TLB 刷新指令需要明确指出刷新的虚拟地址以便无效化影子页表项, 而对于刷新全局 TLB 的指令将触发全局主动同步, 造成更高的开销. 因此, 懒惰影子页表的高效需要客户机操作系统保持细粒度刷新 TLB, 尽量避免全局刷新 TLB. 因此懒惰影子页表更适用于一些有性能要求的定制化的场景, 客户机操作系统以该模式刷新 TLB 能够获得最高的地址翻译效率.

6 总 结

懒惰影子页表模型尝试在 RISC-V 架构下解决虚拟化技术中的内存虚拟化导致的性能瓶颈. 本文利用 RISC-V 架构中的特权级模型和硬件特性在地址翻译开销和页表同步开销之间获得了更好的权衡. 懒惰影子页表模型通过分析虚拟机操作系统对页表项的修改行为, 将通过写保护触发的同步绑定到了 TLB 刷新时刻. 然后, 懒惰影子页表模型通过将页表同步操作延迟到使用时刻来避免复杂的软件开销, 在修改时刻仅进行轻量级的影子页表项无效化操作. 懒惰影子页表模型还对修改时刻产生的虚拟机退出实现了快速路径, 进一步降低了上下文切换的开销. 最终, 通过在 RISC-V 基础架构和虚拟化扩展下的实验表明, 本文所提出的懒惰影子页表模型在保留高效地址翻译效率的同时, 降低了页表同步开销, 提升了虚拟机的性能.

References:

[1] Bhardwaj S, Jain L, Jain S. Cloud computing: A study of infrastructure as a service (IAAS). Int'l Journal of Engineering and Information Technology, 2010, 2(1): 60–63.

[2] Bhargava R, Serebrin B, Spadini F, Manne S. Accelerating two-dimensional page walks for virtualized systems. In: Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Seattle: Association for Computing Machinery, 2008. 26–35. [doi: 10.1145/1346281.1346286]

[3] Cervone HF. An overview of virtual and cloud computing. OCLC Systems & Services: Int'l Digital Library Perspectives, 2010, 26(3): 162–165. [doi: 10.1108/10650751011073607]

[4] Li YK, Lin YY, Wang Y, Ye KJ, Xu CZ. Serverless computing: State-of-the-art, challenges and opportunities. IEEE Trans. on Services Computing, 2023, 16(2): 1522–1539. [doi: 10.1109/TSC.2022.3166553]

[5] Li ZJ, Guo LS, Cheng JG, Chen Q, He BS, Guo MY. The serverless computing survey: A technical primer for design architecture. ACM Computing Surveys (CSUR), 2022, 54(10S): 220. [doi: 10.1145/3508360]

[6] Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. ACM SIGPLAN Notices, 2006, 41(11): 2–13. [doi: 10.1145/1168918.1168860]

[7] Ryoo JH, Gulur N, Song S, John LK. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. ACM SIGARCH Computer Architecture News, 2017, 45(2): 469–480. [doi: 10.1145/3140659.3080210]

[8] Gandhi J, Basu A, Hill MD, Swift MM. Efficient memory virtualization: Reducing dimensionality of nested page walks. In: Proc. of the 47th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Cambridge: IEEE, 2014. 178–189. [doi: 10.1109/MICRO.2014.37]

[9] Wang XL, Zang JR, Wang ZL, Luo YW, Li XM. Selective hardware/software memory virtualization. ACM SIGPLAN Notices, 2011,

46(7): 217–226. [doi: 10.1145/2007477.1952710]

[10] Waldspurger CA. Memory resource management in VMware ESX server. ACM SIGOPS Operating Systems Review, 2002, 36(SI): 181–194. [doi: 10.1145/844128.844146]

[11] Gandhi J, Hill MD, Swift MM. Agile paging: Exceeding the best of nested and shadow paging. ACM SIGARCH Computer Architecture News, 2016, 44(3): 707–718. [doi: 10.1145/3007787.3001212]

[12] Dörflinger A, Albers M, Kleinbeck B, Guan YJ, Michalik H, Klink R, Blochwitz C, Nechi A, Berekovic M. A comparative survey of open-source application-class RISC-V processor implementations. In: Proc. of the 18th ACM Int'l Conf. on Computing Frontiers. Virtual Event: Association for Computing Machinery, 2021. 12–20. [doi: 10.1145/3457388.3458657]

[13] Sha S, Zhang Y, Luo YW, Wang XL, Wang ZL. Swift shadow paging (SSP): No write-protection but following TLB flushing. In: Proc. of the 17th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. Association for Computing Machinery, 2021. 29–42. [doi: 10.1145/3453933.3454012]

[14] Sha S, Zhang Y, Luo YW, Wang XL, Wang ZL. Accelerating address translation for virtualization by leveraging hardware mode. IEEE Trans. on Computers, 2022, 71(11): 3047–3060. [doi: 10.1109/TC.2022.3145671]

[15] Ahn J, Jin S, Huh J. Revisiting hardware-assisted page walks for virtualized systems. ACM SIGARCH Computer Architecture News, 2012, 40(3): 476–487. [doi: 10.1145/2366231.2337214]

[16] Sha S, Du HL, Luo YW, Wang XL, Wang ZL. Software-based flat nested page table in sunway architecture. Journal of Computer Research and Development, 2022, 59(4): 737–746 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.20210140]

[17] Park CH, Vougioukas I, Sandberg A, Black-Schaffer D. Every walk's a hit: Making page walks single-access cache hits. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: Association for Computing Machinery, 2022. 128–141. [doi: 10.1145/3503222.3507718]

[18] Liang ZY, Li TZ, Cui EF. RISC-V virtualization: Exploring virtualization in an open instruction set architecture. In: Proc. of the 5th Int'l Conf. on Computing, Networks and Internet of Things. Tokyo: Association for Computing Machinery, 2024. 473–477. [doi: 10.1145/3670105.3670188]

[19] Sá B, Martins J, Pinto SES. A first look at RISC-V virtualization from an embedded systems perspective. IEEE Trans. on Computers, 2022, 71(9): 2177–2190. [doi: 10.1109/TC.2021.3124320]

[20] Patel A, Daftedar M, Shalan M, El-Kharashi MW. Embedded hypervisor Xvisor: A comparative analysis. In: Proc. of the 23rd Euromicro Int'l Conf. on Parallel, Distributed, and Network-based Processing. Turku: IEEE, 2015. 682–691. [doi: 10.1109/PDP.2015.108]

[21] Lim JT, Dall C, Li SW, Nieh J, Zyngier M. NEVE: Nested virtualization extensions for ARM. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: Association for Computing Machinery, 2017. 201–217. [doi: 10.1145/3132747.3132754]

[22] Merrifield T, Taheri HR. Performance implications of extended page tables on virtualized x86 processors. In: Proc. of the 12th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. Atlanta: Association for Computing Machinery, 2016. 25–35. [doi: 10.1145/2892242.2892258]

[23] Cui EF, Li TZ, Wei Q. RISC-V instruction set architecture extensions: A survey. IEEE Access, 2023, 11: 24696–24711. [doi: 10.1109/ACCESS.2023.3246491]

[24] Domingos JM, Rocha T, Neves N, Roma N, Tomás P, Sousa L. Supporting RISC-V performance counters through Linux performance analysis tools. In: Proc. of the 34th Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP). Porto: IEEE, 2023. 94–101. [doi: 10.1109/ASAP57973.2023.00027]

[25] Pham B, Veselý J, Loh GH, Bhattacharjee A. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In: Proc. of the 48th Int'l Symp. on Microarchitecture. Waikiki: Association for Computing Machinery, 2015. 1–12. [doi: 10.1145/2830772.2830773]

[26] Kwon O, Lee Y, Hong S. Virtual PTE storage: Repurposing last-level cache to accelerate address translation for big data workloads. In: Proc. of the 2022 IEEE Int'l Conf. on Consumer Electronics-Asia (ICCE-Asia). Yeosu: IEEE, 2022. 1–5. [doi: 10.1109/ICCE-Asia57006.2022.9954665]

[27] Barr TW, Cox AL, Rixner S. Translation caching: Skip, don't walk (the page table). ACM SIGARCH Computer Architecture News, 2010, 38(3): 48–59. [doi: 10.1145/1816038.1815970]

[28] Bhattacharjee A. Large-reach memory management unit caches. In: Proc. of the 46th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Davis: Association for Computing Machinery, 2013. 383–394. [doi: 10.1145/2540708.2540741]

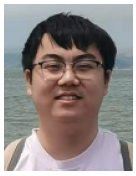
[29] Brown N, Jamieson M. Performance characterisation of the 64-core SG2042 RISC-V CPU for HPC. arXiv:2406.12394, 2024.

[30] Bellard F. QEMU, a fast and portable dynamic translator. In: Proc. of the 2005 Annual Conf. on USENIX Annual Technical Conf. Anaheim: USENIX Association, 2005. 41.

[31] Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1–7. [doi: 10.1145/1186736.1186737]

附中文参考文献:

[16] 沙赛, 杜翰霖, 罗英伟, 汪小林, 王振林. 申威架构下的软件平滑嵌套页表. 计算机研究与发展, 2022, 59(4): 737–746. [doi: 10.7544/issn1000-1239.20210140]

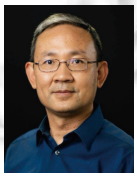


李传东(1999－), 男, 博士生, CCF 学生会员, 主要研究领域为虚拟化, 内存管理, 操作系统.

衣然(2002－), 男, 博士生, 主要研究领域为操作系统, 内存管理.

罗英伟(1971－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为系统软件, 操作系统, 虚拟化, 云计算.

汪小林(1972－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为系统软件, 操作系统, 虚拟化, 云计算.



王振林(1970－), 男, 博士, 教授, 博士生导师, 主要研究领域为编译, 操作系统, 系统虚拟化.