计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(9):15−23 [doi: 10.15888/j.cnki.csa.008843]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于地址空间标识符的 QEMU 动态跳转优化①

位金弈, 梁洪亮

(北京邮电大学 计算机学院, 北京 100876)
通信作者: 梁洪亮, E-mail: hliang@bupt.edu.cn

摘　要: 随着硬件技术的不断演进和软件需求的持续增长, 人们对以 QEMU 为代表的指令集架构仿真平台的执行性能提出了更高的要求. 本文针对目标架构支持虚拟内存的场景, 分析了 QEMU 现有动态跳转处理机制及其存在的问题, 根据常见虚拟内存系统的特点提出了基于地址空间标识符的动态跳转优化方案, 并以 RISC-V 为目标架构在 QEMU 主线 6.2.0 版本上实现了该方案. 实验表明, 相较于原生 QEMU, 基于地址空间标识符的动态跳转方案提升了约 12% 的运行性能.

关键词: QEMU; 动态跳转; 地址空间标识符; RISC-V

引用格式: 位金弈,梁洪亮.基于地址空间标识符的 QEMU 动态跳转优化.计算机系统应用,2022,31(9):15–23. http://www.c-s-a.org.cn/1003-3254/8843.html

Optimization of Dynamic Jump Handling in QEMU Based on Address Space Identifier

WEI Jin-Yi, LIANG Hong-Liang

(School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China)

Abstract: The continuous evolution of hardware and software technology demands higher execution performance from instruction set architecture emulators represented by QEMU. This study analyzes the limitations of QEMU's existing dynamic jump handling mechanism in the scenario where the emulated architecture supports virtual memory, and proposes an optimized scheme based on address space identifiers suitable for common virtual memory systems. The proposed scheme is implemented for the RISC-V frontend in QEMU mainline 6.2.0 version. Evaluation results show that the dynamic jump scheme based on the address space identifier achieves an average performance improvement of 12% compared to the native QEMU.

Key words: QEMU; dynamic jump; address space identifier; RISC-V

1　引言

指令集架构仿真技术能够在缺少对应硬件设备的情况下辅助完成相关软件的开发和测试工作, 以及方便硬件设计者更早地对设计方案进行评估. 以近年来新兴的 RISC-V[1] 指令集架构和开源领域主流的指令集架构仿真器 QEMU[2] 为例, RISC-V 指令集中对应向量指令的 V 扩展[3] 和对应硬件加速虚拟化的 H 扩展[4] 等指令扩展模块均是先在 QEMU 中以 0.x 版本进行实现, 经过软硬件磨合迭代到 1.0 版本后才由 RISC-V 基金会批准并正式写入指令集规范.

随着硬件技术的不断演进和软件需求的持续增长, 人们对仿真器的执行性能也提出了更高的要求. 间接跳转的处理方式是决定仿真器执行效率的关键因素[5], 特别是在全系统仿真的场景下, 目标架构处理器的虚拟内存功能允许跳转目标的物理地址在运行时动态变化, 这一特性为仿真器正确和高效地实现间接跳转和跨页直接跳转 (后统称为动态跳转) 的语义带来了许多挑战.

① 本文由"RISC-V 技术及生态"专题特约编辑武延军研究员、宋威副研究员、张科高级工程师以及邢明杰高级工程师推荐.
收稿时间: 2022-03-30; 修改时间: 2022-04-25; 采用时间: 2022-05-16; csa 在线出版时间: 2022-07-22

本文分析了仿真目标架构支持虚拟内存的场景中 QEMU 翻译块间跳转机制的不足, 并结合常见虚拟内存模型的特点提出了基于地址空间标识符的动态跳转改进方案. 该方案将地址翻译从动态跳转执行的关键路径中移除, 在保证正确解析跳转目标的前提下提升仿真执行性能. 本文在 QEMU 主线 6.2.0 版本中实现了这一方案, 并以 RISC-V 架构的操作系统 xv6-riscv[6] 为例对方案的有效性进行评估, 实验结果表明, 改进后的 QEMU 在 8 个测试程序中实现了约 12% 的平均运行性能提升.

2 QEMU 的现有翻译块间跳转机制

为了避免重复的取指译码开销, QEMU 使用动态二进制翻译技术, 以基本块为单位将目标架构的汇编指令翻译成对应的宿主机指令并进行缓存, 由此得到的仿真执行单元被称为翻译块 (translation block). 系统仿真模式下, 执行流在翻译块间的转移分为 3 类: 页内直接跳转、间接跳转和跨页直接跳转.

2.1 页内直接跳转

对通过直接跳转连接的目标架构基本块而言, 其对应的翻译块拥有固定的前驱后继关系. 作为优化, QEMU 通过翻译块直接链接的方式将前驱翻译块的退出操作重定向到后继翻译块的入口处, 以消除后续执行中动态查找后继翻译块的运行时开销.

该机制借助 QEMU 的中间表示 (intermediate representation, IR) 指令 goto_tb 和 exit_tb 完成. 以 64 位 RISC-V 架构为例, 当允许翻译块直接链接时, QEMU 的翻译前端为直接跳转生成的 IR 指令和对应的 x86_64 宿主机代码示例如图 1 所示.

翻译块 tb 首次执行时, 执行流将通过 goto_tb 对应的空跳转指令, 更新程序计数器 (PC) 后经 exit_tb 返回 QEMU 查找或生成后继翻译块, 并将 tb 中对应分支的空跳转指令重定向到后继翻译块的宿主机指令入口处. 后续执行翻译块 tb 时将在 goto_tb 处完成直接跳转, 不再更新 PC 或进行翻译块查找.

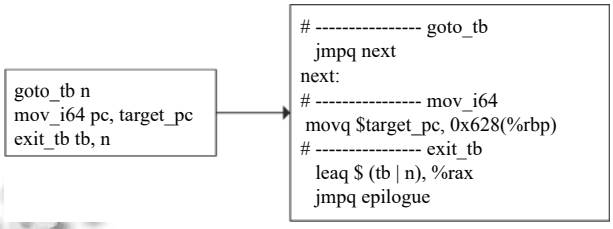

图 1 直接跳转 IR 序列在 x86_64 后端生成代码举例

2.2 间接跳转

间接跳转的目标地址不能在 QEMU 进行指令翻译时确定, 且在运行时有多种可能 (如相同函数的不同返回地址和面向对象模式中相同基类的不同虚方法等), 故无法使用翻译块直接链接的方式进行优化, 需要借助运行时的翻译块查找机制确定当次跳转的目标. 这一查找过程涉及两级翻译块缓存, 即: 1) 以 PC 地址通过哈希直接映射方式索引的 vCPU 线程私有缓存数组 tb_jmp_cache; 2) 以翻译块的虚拟地址和物理地址等信息作为键的全局共享哈希表 QHT[7,8], QHT 缓存了由 QEMU 翻译得到且依然有效的所有翻译块. 为了后续叙述的方便, 本文以 H0 和 H2 分别指代 QEMU 现有的两级翻译块缓存, 前述查找过程的具体流程如图 2 所示, 其中虚线框代表对应环节的输入.

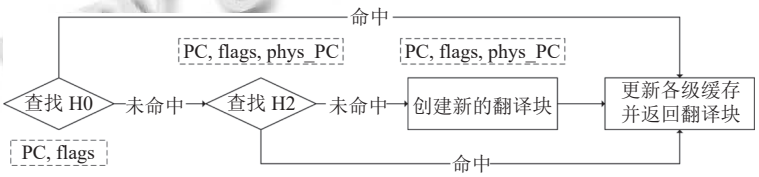

图 2 QEMU 的翻译块查找流程

翻译块执行间接跳转时, 需要更新 PC 后再通过上述翻译块查找机制获取跳转目标翻译块的入口地址 (调用辅助函数 helper_lookup_tb_ptr), 并通过 goto_ptr 对应的宿主机间接跳转指令完成跳转. 以 64 位 RISC-V 架构为例, QEMU 的翻译前端为间接跳转生成的 IR 指令和对应的 x86_64 宿主机代码示例如图 3 所示.

2.3 跨页直接跳转

跨页直接跳转拥有翻译时可获知的目标地址, 但其跳转目标与当前基本块位于目标架构的不同虚拟内存页面上. 在全系统仿真模式下, 运行在 QEMU 中的特权软件可以通过修改页表等方式决定任意内存页的映射物理地址和访问权限, 导致运行时跳转目标发生

变更, 这使得跨页直接跳转拥有了与间接跳转类似的动态特性. 因此, QEMU 保守地禁止对跨页直接跳转使用翻译块直接链接的优化, 转而使用与间接跳转完全相同的动态查找机制完成跳转.

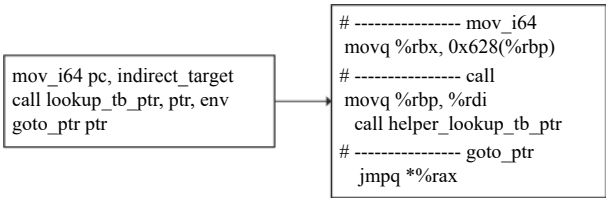

图 3 间接跳转 IR 序列在 x86_64 后端生成代码举例

2.4 现有跳转机制存在的不足

H0 本身的查找开销较低, 但其基于直接映射的组织方式使得缓存冲突的概率较大; 由于 H0 仅使用虚拟地址进行索引, QEMU 必须在内存映射发生改变时刷新 H0 以避免歧义. 上述因素导致翻译块查找的流程需要相对频繁地访问 H2, 并使用物理地址确定跳转的目标翻译块. 然而, QEMU 中目标架构地址翻译过程本身所依赖的 softTLB 同样不能在内存映射改变后保持有效, 往往需要重新访问 softMMU 获取 PC 对应的物理地址. 因为 QEMU 不区分数据 TLB 与指令 TLB, 仿真运行过程中由翻译块查找需求导致的地址翻译还将与翻译块本身的数据读写操作争用 TLB, 带来反复的页表查询开销, 降低仿真执行的效率.

另一方面, QEMU 保守地使用运行时查找翻译块的方式确保跨页直接跳转的正确性. 单次翻译块查找的运行时开销为数十条至数千条宿主机汇编指令 (取决于 H0 和 TLB 是否命中), 且需要占用相应的缓存空间, 故代价远高于页内直接跳转 (链接后通过单条宿主机跳转指令即可完成). 而且, 对于任何稍具规模的软件而言, 绝大多数由某一模块发起, 以位于其他模块中函数为目标的调用行为都会是跨越虚拟内存页面直接跳转, 且相关内存映射在软件的整个执行过程中不会改变. 因此, 我们认为, 基于简化程序执行常见路径的原则, 这一不足是有改进空间的.

3 观察与讨论

虚拟内存是现代通用处理器的一项关键特性, 通过对内存访问进行抽象, 使特权软件可以在运行时配置任意虚拟地址的映射内存位置和访问权限. 虚拟内存机制的运作依赖软硬件的配合, 包含以基数树形式存在于内存中的页目录、用于在页目录中查找虚拟地址对应映射描述条目的硬件单元 MMU 和用于缓存 MMU 查找结果的地址翻译缓存 TLB. 针对这一虚拟内存的主流实现模型 (单页目录、MMU、TLB), 关于虚拟内存存在场景下内存映射的不确定性对动态跳转带来的影响, 本文有以下 3 点观察.

3.1 翻译块查找无需使用完整物理地址

为了区分虚拟地址相同而物理地址不同的翻译块, H2 使用翻译块的物理地址辅助翻译块的索引和比对, 而这也正是目前翻译块查找需要物理地址的原因. 对于前文提到的虚拟内存的主流实现模型, 内存映射完全由 vCPU 所使用的页目录 (包含其下各级页表) 所决定, 因此我们可以尝试为“页目录整体”建立一个地址空间标识符, 并将其作为额外的索引替代物理地址对翻译块进行标识, 免去查找时的地址翻译开销.

3.2 TLB 刷新本身不涉及内存映射内部的改变

一般而言, TLB 刷新只是为了迫使 MMU 重新访问页表, 使用更新后的页目录进行地址翻译, 对页目录本身的修改或切换操作并不是由 TLB 刷新导致的. 但 QEMU 目前并不维护页表页相关的状态, 因而在 TLB 刷新后默认内存映射可能发生任何变化, 无法区分地址空间的整体切换和局部修改. 一种改进方法是: 在仿真运行时监测对页目录的写入操作, 并据此修正已经建立的地址空间标识符. 如果决定内存映射的页目录在切换间隙未发生变化, 则之前运行过程中为其建立的地址空间标识符亦不需改变.

3.3 对页目录的监测只需关注被使用过的部分

QEMU 翻译基本块时, 需要首先根据当前 PC 进行地址翻译以获取取指使用的物理地址, 该过程会通过 softMMU 访问当前页目录中的相关页表页. 另一方面, 页目录中那些从未因指令地址翻译的需求而被访问的页表页可以被视为尚未加入当前地址空间, 对这些页面的修改并不会影响该地址空间下翻译块查找的结果, 以及任何跨页直接跳转的有效性. 由此, 我们可以将监测范围局限于已生效的页表页, 从而消除主动探测页目录完整结构所带来的开销. 当已生效页表页被修改时, 先前建立的地址空间标识符不再有效, 需要为其分配新的标识符以指代经过内部修改后得到的新地址空间.

4 基于地址空间标识符的动态跳转机制

基于上述观察与讨论, 我们为 QEMU 引入地址空

间标识符的概念, 并对其动态跳转处理机制进行优化.

4.1 地址空间标识符的定义与维护

4.1.1 地址空间标识符的定义

本文使用页目录的物理页号和一个单调递增的版本号共同作为由该页目录所决定内存映射的地址空间标识符. 版本号的值初始化为零, 并在每次检测到页目录变化时增一以示区别. 在页目录部分相同的情况下, 若 vCPU 线程记录的地址空间标识符版本号高于先前存储于某处的标识符版本号, 则说明对应地址空间的内存映射在仿真运行期间发生了改变.

由此, 地址空间标识符提供了可用于区分不同内存映射和检测地址空间内存映射是否发生改变的简单机制. 在后续描述中, 将以 QASID 代指此处引入的地址空间标识符以简化行文, 并区别于一般意义上的硬件 ASID.

4.1.2 地址空间标识符的维护

与 QEMU 中代码页写保护的建立过程类似, 我们采用如下方式对页表页进行写保护: 维护新的物理页状态位图 DIRTY_MEMORY_PTE, 并在 softMMU 因为指令地址翻译的需要而访问各级页表页时标记此位图中代表相应页面的比特. 当 QEMU 为 softTLB 添加新的地址映射条目时, 将根据页面在前述位图中的状态对应设置 TLB 表项的 TLB_NOTDIRTY 标志位, 从而截获后续所有对受保护页表页的写操作.

页表页修改导致的版本号更新代表着对相应旧 QASID 的弃用, 在一些情况下, 修改并不会影响对动态跳转目标的解析, 作为这类场景中写保护触发时的剪枝优化, 我们进一步做如下 3 点设计:

1) 若被修改位置并非目标架构中所规定的有效的页表项, 则不需要更新版本号. 该方式可以过滤部分由动态内存分配或进程按需调页引起的页表页修改, 减少无谓的 QASID 弃用.

2) 保护页表页时记录页目录的版本号, 若写保护发生时, 页目录版本号已经大于先前记录的版本号, 则不需要继续作更新. 版本号不同代表先前内存映射对应的 QASID 已被丢弃, 而新的内存映射尚未包含该页表页.

3) 目标架构的有效页表项被修改时, 立即取消对页面的保护. 单个页表项变化导致的内存映射改变足以使原本的地址空间失效, 故其余的映射条目也不需要继续被保护. 在原物理页面不再作为页表使用后, 该方法可以减少清空页面导致的重复写保护开销.

4.1.3 vCPU 线程对地址空间标识符的获取

我们把 vCPU 当前使用的 QASID 作为 vCPU 线程私有数据存储在 vCPU 的控制结构中. 其中, 页目录物理页号在目标软件设置页目录 (如写入 RISC-V 架构中的 SATP 寄存器) 时同步作更新. 版本号在 TLB 刷新时设为非法值, 并按需从页目录的页描述符 PageDesc 中获取, 无需随页表页修改而更新. 其原因在于 TLB 与页表页内存的数据一致性是由软件手动维护的, 而 QASID 作为对内存映射的整体描述应具有相同的特性.

4.2 地址空间标识符在动态跳转处理中的应用

4.2.1 间接跳转

QASID 唯一决定了当前的内存映射, 因而可以在翻译块的查找过程中替换原本使用的物理地址, 以省去地址翻译的开销. 为此, 本文修改 QEMU 现有的两级翻译块缓存结构, 引入另外一个借助 QASID 进行索引的共享翻译块缓存 H1, 并在查找间接跳转目标时优先查询该缓存.

H1 亦基于 QHT 实现, 其与 H2 的区别主要在于键值对的选取上. 对于键, 我们将 H2 中原本使用的物理地址替换为 QASID 的页目录物理页号; 对于值, 我们把原本 H2 中指向翻译块的指针改为指向翻译块包装的指针. 翻译块包装 (translation block wrapper) 由翻译块指针和对应 QASID 构成. 不在 H1 中直接插入翻译块的原因是翻译块可能被多个地址空间所共享, 此时单一的翻译块数据结构无法存储多个 QASID, 因而基于哈希表的查找也就无从谈起. 引入修改后 QEMU 对翻译块间接跳转目标的查找流程如图 4 所示.

查找 H1 的过程不涉及版本号信息, 因而在命中后还需要与 vCPU 中记录的 QASID 版本号进行比对. 版本号相等代表内存映射未发生改变, 视为翻译块查找命中, 而版本号不相等的场景可以分为 3 类, 其中 (1) 和 (2) 对应图 4 中检查版本号不通过的情形, 需要进一步根据物理地址查询 H2 以判断跳转目标是否发生改变.

(1) 翻译块包装版本号小于 vCPU 版本号, 但 H2 中使用物理地址查找到相同的翻译块——说明地址空间的内存映射发生改变, 但并未影响当前跳转的目标地址, 因此我们只需更新翻译块包装中的版本号信息.

(2) 翻译块包装版本号小于 vCPU 版本号, 且 H2 中使用物理地址未查找到或查找到不同的翻译块——说明地址空间发生改变且影响了当前跳转的目标地址,

因此我们需要创建新的翻译块包装替换 H1 中的原有元素.

(3) 翻译块包装版本号大于 vCPU 版本号——说明内存映射发生改变但当前 vCPU 尚未刷新 TLB, 较高的版本号来自并行执行的其他 vCPU 线程, 因此我们视为翻译块查找命中.

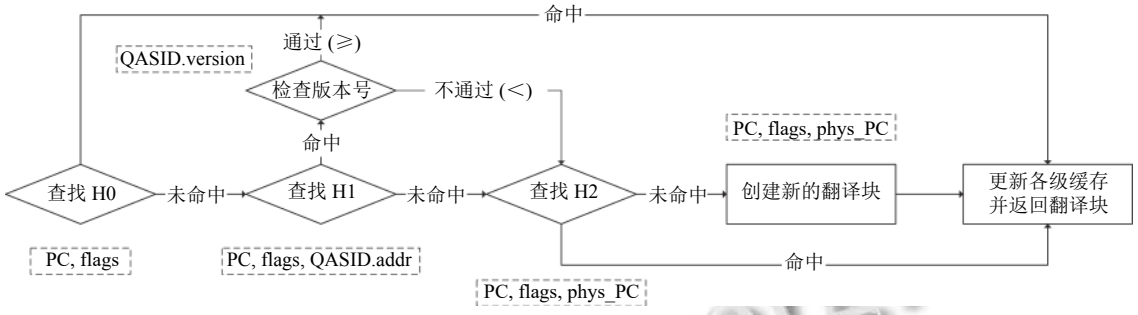

图 4 引入哈希表 H1 后的间接跳转目标查找流程

4.2.2 跨页直接跳转

目前 QEMU 跨页直接跳转的主要困境在于缺乏高效的手段在运行时验证跳转目标的正确性, 因而其现有实现方案放弃对翻译块进行链接, 转向使用彻底的动态查找. 我们提出的地址空间标识符 (QASID) 提供了检验跳转目标正确性的轻量级方法, 基于 QASID 和第 4.2.1 节中引入的哈希表 H1, 可以实现更加高效且语义相同的跨页直接跳转.

新方案为每个包含跨页直接跳转的翻译块分配用于存储 QASID 的额外缓存空间, 并引入一条用于检查内存映射是否改变的 IR 指令 asid_check, 配合页内翻译块直接链接机制中原有的 goto_tb 和 exit_tb 完成跨页场景中的翻译块直接链接, 跨页直接跳转对应的 IR 指令序列如图 5 所示.

```
asid_check tb, n, target_pc
goto_tb n
mov_i64 pc, target_pc
exit_tb tb, n
```

图 5 跨页直接跳转对应的 IR 指令序列

asid_check 读取当前 vCPU 的 QASID 并与翻译块内缓存的 QASID 进行比较以检测内存映射是否发生改变. 二者相等时执行后续 goto_tb 和 exit_tb 对应的指令, 流程与页内直接跳转相同; 二者不等时代表内存映射可能发生变化, 执行流进入 slow path, 调用 asid_check 对应的辅助函数 asid_check_helper 做进一步处理. 以 x86_64 后端为例, asid_check 对应的宿主机代码如图 6.

我们设计 QASID 的宽度为 8 字节以便在 64 位宿主机上通过单条汇编指令完成 QASID 的比较操作, 其中页目录物理页号和版本号分别占据 4 字节, 可以处理至多 16 TB 的物理地址空间. QASID 缓存分配在翻译块尾部, 其基地址可由 %rip 作偏移得到. 由于多线程场景下可能出现多个 vCPU 使用不同 QASID 并行执行相同翻译块的场景, 需要避免缓存争用导致的 slow path 开销, 因而缓存的大小由 vCPU 线程的数量决定, 在运行时使用 vCPU 编号进行索引. 另一方面, vCPU 的编号和当前 QASID 可以从位于 %rbp 低地址的 CPUNegativeOffsetState 结构体中读取, 这一结构体用于存储仿真运行时需要频繁使用的架构无关数据 (如 softTLB 和外设中断请求等), 我们在其中增加 vCPU 编号和 QASID 两个字段以简化 asid_check 的实现.

```
asid_check tb, n, target_pc
# 读取 vCPU 的编号
movl -0xc (%rbp), %edi
# 读取 vCPU 的地址空间标识符
movq -0x8 (%rbp), %rsi
# 与翻译块缓存的地址空间
标识符比较
leaq 0x..(%rip), %rdx
cmpq (%rdx, %rdi, 8), %rsi
# 标识符不匹配则跳转
jne slow_path
# 后续指令
next_instr:

slow_path:
# 传递参数并跳转至辅助函数
movq %rbp, %rdi
leaq $(tb | n), %rsi
movq $target_pc, %rdx
leaq $next_instr, %rax
push %rax
jmpq asid_check_helper
# 尾调用直接返回
```

图 6 IR 指令 asid_check 在 x86_64 后端生成代码示例

当 asid_check 的执行流进入 slow path 时, 辅助函数使用 vCPU 当前的 QASID 和传入的跳转目标虚拟地址查找 H1, 根据查找 H1 的结果和当前翻译块的链

接状态, 有以下两种处理方式:

(1) H1 查找命中, 且翻译块包装中记录的版本号匹配, 且后继 goto_tb 已经链接了 H1 查找所返回的翻译块——此时可以看作翻译块共享导致的缓存冲突, 因此我们只需更新翻译块的 QASID 缓存并直接返回.

(2) 其余情况——表明内存映射发生变化, 或翻译块被多个地址空间所共享且在其中拥有若干个不同的后继翻译块, 或翻译块尚未链接, 此时后继跳转的合法性有待进一步确认, 因此我们需要将翻译块中的 goto_tb 重置后返回.

辅助函数 asid_check_helper 结束后, slow path 将返回与其相邻的 goto_tb 指令处, 根据辅助函数是否断开了记录在 goto_tb 中的直接链接, 执行流将会在此处完成跳转, 或通过 exit_tb 回到 QEMU 查找后继翻译块并尝试重新链接.

跨页直接跳转的重链接过程比较复杂, 因为 vCPU 线程共享翻译块, 我们需要保证各线程对共享翻译块的跳转目标达成一致. 为此本文对翻译块链接的过程增加如下同步机制:

(1) 使用翻译块本身的 jmp_lock 保护翻译块的链接过程. 原本的翻译块链接机制只需要获取跳转目标翻译块的 jmp_lock, 并结合了一些无锁编程技巧. 我们将其修改为一并获取目标翻译块和跳转翻译块本身的 jmp_lock, 为避免可能导致的死锁问题, 加锁通过 trylock 的方式进行, 在失败时释放已有锁并作重试.

(2) 引入新翻译块字段 link_resolving 并使用 jmp_lock 保护. 具体而言, 我们在断开链接时设置 link_resolving 字段中对应当前线程的比特位, 以表明 vCPU 线程对当前翻译块的链接意图, 尝试对翻译块进行链接时清除 link_resolving 中对应当前线程的比特位, 且仅当这一字段为零时才真正进行链接.

在上述同步机制的保护下, 翻译块的重链接过程可以对以下两种场景进行甄别.

(1) 内存映射发生改变导致翻译块跳转目标变化;

(2) 跳转目标同时存在多个对应物理地址.

其中第 2 种情况无法由被共享翻译块末尾的直接跳转语义处理, 需要将其丢弃, 并使用原本基于动态查找完成跳转的方式重新翻译. 我们通过维护翻译块的跳转候选对象 jmp_candidate 完成对多跳转目标物理地址的检测, 具体流程如伪代码 1 所示.

伪代码 1. 多跳转目标物理地址的检测

输入: 翻译块 tb, 跳转方向 n, 目标翻译块 tb_next
输出: 翻译块跳转是否存在多个目标物理地址

```
if (tb→asid_cache[n] 仅包含单一地址空间)
    tb→jmp_candidate[n] = RAM_ADDR_INVALID
endif
if (tb→jmp_candidate[n] == RAM_ADDR_INVALID)
    tb→jmp_candidate[n] = tb_next→phys_pc
else if (tb→jmp_candidate[n] != tb_next→phys_pc)
    return TRUE
endif
return FALSE
```

5 实验评估

本节以 64 位 RISC-V 架构的教学用 UNIX 操作系统 xv6-riscv 及其用户程序为测试对象, 对本文设计并实现的 QEMU 动态跳转优化算法进行评估. 我们的实现基于 QEMU 主线 6.2.0 版本, 测试操作系统内核与用户程序均使用前缀为 riscv64-elf-的交叉编译器和二进制工具进行编译, 其中 GCC 版本为 11.1.0, binutils 版本为 2.36.1. 编译 C 语言文件时所使用的代码生成相关编译选项为-O -fno-omit-frame-pointer -mcmodel=medany -ffreestanding -fno-common -fno-pie -no-pie -nostdlib -fno-stack-protector, 需要说明的是, 我们去除了-mno-relax 选项以启用链接器的松弛化 (linker relaxation) 操作, 以便将 RISC-V 编译器生成的函数调用指令序列 auipc+jalr 按照链接时地址尽可能转换成对应直接跳转的 jal 指令.

测试使用的用户程序为 xv6-riscv 自带的测试用例集 usertests, 该测试集共包含 61 个测试程序, 涵盖了用户进程管理、虚拟内存管理、文件系统等诸多方面, 其中运行时间较长的 8 个测试程序如表 1 所示.

我们以连续模式 (对应命令行选项-c) 执行 usertests 测试集两次, 并将收集到的执行时长累加作为各测试用例的最终运行时间. 为了记录单个测试用例的执行时长, 我们在 usertests 的代码中插桩, 使用 xv6 的 uptime 系统调用获取每个测试测试用例执行前后的时间戳, 取其差值作为程序的运行时间, 另外将 xv6 内核中的时钟中断周期调整为 10 ms 以获取相对精确的计时. 前述设定下 usertests 测试集中测试程序的运行时间节选如图 7 所示.

图 7 中 QEMU 代表使用主线 6.2.0 版本运行时记

录的各测试用例的运行时间, QEMU-PTE 代表加入页表页保护时各测试用例的运行时间, QEMU-H1 表示实现了完整的动态跳转优化算法时各测试用例的运行时间, 所有数据均保留一位小数. 图 8 是将上述数据以原生 QEMU 运行时间做正则化后得到的相对比例.

分析 QEMU-PTE 的运行时间可以得出, 引入页表页写保护并未对测试程序的整体执行时间造成很大影响——最差情况下在 execout 这一测试用例中引入了 6% 的执行开销. 进一步分析可以发现, execout 在执行过程中需要反复使用 sbrk 系统调用将系统拥有的虚拟内存全部分配完毕, 而 OS 为应用程序分配内存的过程即是将物理页地址写入应用程序页表页的过程, 因而 execout 的运行过程涉及到较多对已受保护页表页的写操作, 引入了一定的开销. 本测试中仿真环境使用 RISC-V 的 Sv39 虚拟内存模型, 共包含三级页表页, 由于保护代码映射同时需要保护涉及到的所有非叶节点高层页表页, 因此, 当页表级数随着虚拟地址空间的扩大而增加时, 对高层页表页的保护会引入更多的开销.

表 1 测试集 usertests 中部分程序的名称与逻辑描述

名称	程序逻辑描述
bigdir	在当前目录下创建文件, 并为其创建500个硬链接, 随后删除所有硬链接.
execout	创建子进程以较小粒度分配系统主要内存后加载新程序, 测试exec的错误恢复功能.
manywrites	创建若干子进程并发创建文件并执行写入操作.
concreate	创建若干子进程对相同文件并发执行创建、删除和链接操作.
createdelete	创建若干子进程在相同目录中并发创建和删除文件.
reparent2	创建若干子进程, 子进程执行fork后不经wait直接退出.
twochildren	创建两个子进程并使其同时退出, 循环1 000次.
sbrkfail	创建若干子进程以较大粒度分配系统全部内存, 测试失败后的清理机制.

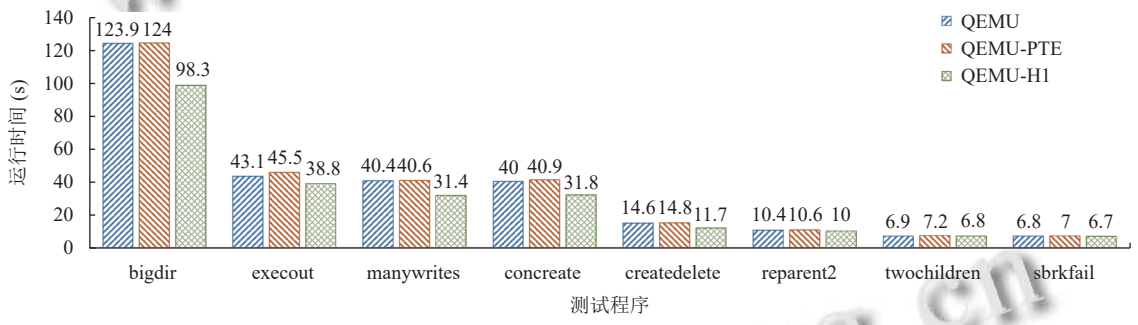

图 7 usertests 测试程序运行时间节选

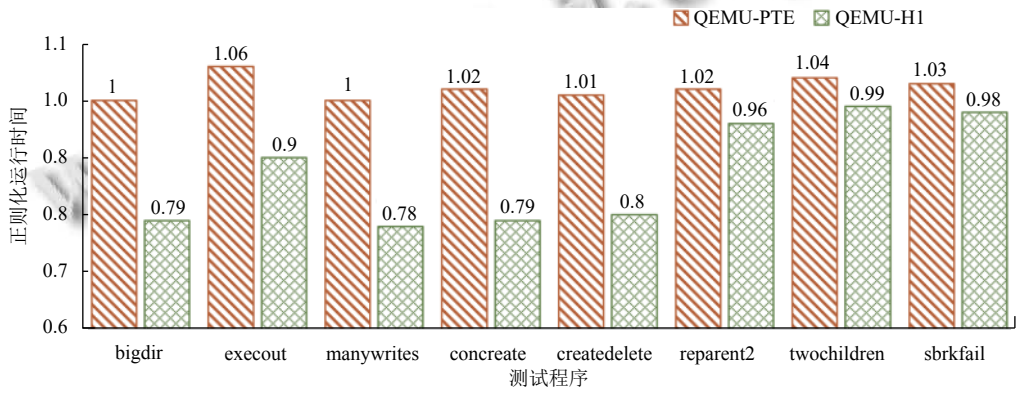

图 8 usertests 测试程序正则化运行时间

另一方面, 图 8 表明 QEMU-H1 在选取的 8 个测试用例中实现了约 12% 的平均运行性能提升, 在涉及大量文件操作的 bigdir, manywrites, concreate 和 createdelete 四个测试用例上表现尤为明显, 而在 reparent2, twochildren 和 sbrkfail 上带来的变化则并不显著. 我们通过分析 xv6-riscv 相关内核功能实现, 认为上

述性能变化的原因如下: 文件操作的完成过程涉及较多的对代码规模较小函数的调用, 包含文件系统日志、inode 管理与查找、用户空间数据交换和磁盘块读写等, 相关函数的实现分散在内核的不同模块中, 因而产生了数量较多的函数返回与跨页直接跳转; 与此相对地, 其余测试用例中大量使用的 fork, wait, exit, sbrk 系统调用的实现相对独立, 仅使用少量接口与外部模块交互. 我们反汇编内核二进制文件发现, 相关接口函数的代码规模较大且数量少, 对应跳转目标及返回地址可以被很好地缓存在 H0 中, 故 QEMU-H1 中额外引入的动态跳转优化没有对其产生明显的影响.

6 相关工作

目标架构虚拟内存特性的仿真对模拟器的整体设计和运行效率有深刻影响. Tong 等人[9] 在特定工作负载上定量研究了 QEMU 中内存仿真模块的运行开销, 并探索了 softTLB 的若干改进设计方案, 包含自适应扩容的动态大小 TLB 和为 TLB 额外添加全相联后备缓冲等. Cota 等人[10] 改进了 QEMU 中 TLB 的扩容策略, 根据时间窗口内的 TLB 占用率选择合适的 TLB 大小以平衡 TLB 的命中率与刷新开销. Hong 等人[11] 提出使用内联缓存的方式加速 QEMU的动态跳转, 对应目前 QEMU 中使用的 lookup_and_goto_ptr 逻辑.

以上工作旨在优化 QEMU 仿真过程中与地址翻译相关的缓存结构以增加命中概率, 但由软件实现的缓存查询操作本身亦导致了一定的开销. 对此, Chang 等人[12] 提出了嵌入式影子页表 (ESPT) 的方案, 将目标架构的页表嵌入 QEMU 进程自身的页表中, 从而利用 x86_64 宿主机的地址翻译硬件完成单指令的目标架构内存访问. 在 ESPT 的基础上, Wang 等人[13] 通过 Linux 系统的 mmap 系统调用完成对 QEMU 自身页表的操作, 去除了对内核模块的依赖. Huang 等人[14] 借助龙芯 3A4000 处理器的对偶硬件 TLB 实现了更加通用的跨指令集架构内存虚拟化, 解决了 ESPT 方案中目标架构地址空间需要严格小于宿主架构地址空间的局限, 且可以处理目标架构与宿主架构内存页面大小不匹配的情形.

7 结语

本文介绍了 QEMU 中翻译块间跳转机制的原理, 详细分析了仿真目标架构支持虚拟内存的场景下现有跳转机制的不足之处, 并针对常见的单页目录虚拟内存模型提出和实现了基于地址空间标识符的动态跳转改进方案. 该方案通过监视对页表页的写操作维护 QEMU 内部的地址空间标识符, 并利用该标识符优化动态跳转目标的查找过程. 最后, 本文使用运行在 RISC-V 架构上的教学用 UNIX 操作系统 xv6-riscv 对提出的方案进行了性能评估, 验证了本文工作的有效性: 引入页表页写保护对程序的执行时间开销影响较小, 动态跳转优化方案实现了约 12% 的平均运行性能提升.

本文方案的局限在于 QASID 由页目录唯一决定, 因而不能高效处理内存映射受页目录以外因素影响的场景, 如 RISC-V 中可能开启的物理内存保护机制 PMP 和由 H-扩展引入的两阶段地址翻译等. 后续可以考虑设计综合各项因素的 QASID 分配算法, 以提高方案的泛用性.

参考文献

1 Waterman A, Lee Y, Avizienis R, *et al*. The RISC-V instruction set. 2013 IEEE Hot Chips 25 Symposium (HCS). Stanford: IEEE, 2013. 1. [doi: 10.1109/HOTCHIPS.2013.7478332]

2 Bellard F. QEMU, a fast and portable dynamic translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference. Anaheim: USENIX Association, 2005. 41–46.

3 RISC-V 向量扩展规范. https://github.com/riscv/riscv-v-spec. [2022-03-20].

4 Waterman A, Asanović K, Hauser J. The RISC-V instruction set manual: Volume II: Privileged architecture, document version 20211203. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf. [2022-03-20].

5 Hiser JD, Williams D, Hu W, *et al*. Evaluating indirect branch handling mechanisms in software dynamic translation systems. International Symposium on Code Generation and Optimization (CGO). San Jose: IEEE, 2007. 61–73. [doi: 10.1109/CGO.2007.10]

6 xv6-riscv 项目主页. https://github.com/mit-pdos/xv6-riscv. [2022-03-20].

7 Cota EG, Bonzini P, Bennée A, *et al*. Cross-ISA machine emulation for multicores. IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Austin: IEEE, 2017. 210–220. [doi: 10.1109/CGO.2017.7863741]

8 Cota EG. Scalable emulation of heterogeneous systems [Ph.D. Thesis]. New York: Columbia University, 2019.
9 Tong X, Koju T, Kawahito M, *et al*. Optimizing memory translation emulation in full system emulators. ACM Transactions on Architecture and Code Optimization, 2015, 11(4): 60. [doi: 10.1145/2686034]
10 Cota EG, Carloni LP. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). Providence: ACM, 2019. 74–87. [doi: 10.1145/3313808.3313811]
11 Hong DY, Hsu CC, Chou CY, *et al*. Optimizing control transfer and memory virtualization in full system emulators. ACM Transactions on Architecture and Code Optimization, 2016, 12(4): 47. [doi: 10.1145/2837027]
12 Chang CJ, Wu JJ, Hsu WC, *et al*. Efficient memory virtualization for cross-ISA system mode emulation. Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). Salt Lake City: ACM, 2014. 117–128. [doi: 10.1145/2576195.2576201]
13 Wang Z, Li JJ, Wu CG, *et al*. HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). Istanbul: ACM, 2015. 53–64. [doi: 10.1145/2731186.2731188]
14 Huang KL, Zhang FX, Li C, *et al*. BTMMU: An efficient and versatile cross-ISA memory virtualization. Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). Virtual: ACM, 2021. 71–83. [doi: 10.1145/3453933.3454015]

(校对责编: 孙君艳)