计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
62(8): 1931−1947, 2025

本期推荐

基于细粒度状态标签的代码缓存优化方案

牛 根 张福新

(处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

(中国科学院大学 北京 100049)

(niugen18z@ict.ac.cn)

Code Cache Optimization Schemes Based on Fine-Grained State Label

Niu Gen and Zhang Fuxin

(*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

(*University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract Software code cache is widely used in dynamic binary translators to manage the dynamically generated code blocks. The translation, refresh, and memory occupancy of code blocks are key metrics for software code cache. There has been little research on software code cache for system-level dynamic binary translators. Existing system-level dynamic binary translators use state label scheme to achieve correct and efficient instruction semantic simulation, but this scheme introduces additional problems for software code cache management. Through in-depth analysis of the state label scheme, two types of problems are summarized: conflicts and redundancies. To address these two problems, two code cache optimization schemes based on fine-grained state label are proposed, including multi-state code cache scheme and weak state label scheme. These two schemes are implemented in LATX-SYS and evaluated with Ubuntu/x86 16.04 and Windows XP/x86 system booting on LoongArch platform. The evaluation results show that the code block refresh and translation are reduced by 43% and 18% respectively. The code block similarity ratio is decreased from 59.63% to 5.06%. The translation overhead and memory occupancy are both reduced. Overall, the system boot time is reduced by 20%. Finally, testing of the weak state label scheme on SPEC CPU2000 shows that the number of code blocks is reduced by an average of 13%, with only 2%−3% performance overhead introduced.

Key words software code cache; binary translation; virtualization; code block; dynamic optimization

摘 要 动态二进制翻译器中广泛使用软件代码缓存来管理翻译生成的代码块.代码块的翻译、刷新和内存占用是软件代码缓存的重要指标.目前仅有少量的针对系统级动态二进制翻译器中代码缓存的研究.已有的系统级动态二进制翻译器为实现正确且高效的指令语义模拟,均使用了状态标签方案,但该方案会对软件代码缓存管理带来额外的问题.通过深入分析状态标签方案,总结了其给代码缓存管理带来的2类问题:冲突和冗余.针对这2类问题,提出了基于细粒度状态标签的代码缓存优化方法,包括多状态代码缓存和弱状态标签.这2种方案在LATX-SYS中实现并在龙芯LoongArch平台上使用Ubuntu/x86 16.04和Windows XP/x86客户机操作系统进行了测试.结果表明,代码块刷新次数和翻译次数分别降低了43%和18%,代码块相似率从59.63%降至5.06%,翻译开销和内存占用均得到降低.总的来说,系统启动时间降低了20%.最后,针对弱状态标签方案进一步测试了SPEC CPU2000,结果表明代码块数量平均减少了

收稿日期:2023-10-31;**修回日期**:2025-01-07

基金项目:国家重点研发计划项目(2022YFB3105103)

This work was supported by the National Key Research and Development Program of China (2022YFB3105103).

通信作者:张福新(fxzhang@ict.ac.cn)

13%, 且仅带来 2%~3% 的性能开销.

关键词 软件代码缓存; 二进制翻译; 虚拟化; 代码块; 动态优化

中图法分类号 TP314

DOI: 10.7544/issn1000-1239.202330856 **CSTR:** 32373.14.issn1000-1239.202330856

动态二进制翻译技术可以实现程序的模拟执行. 被模拟执行的客户机程序通过动态二进制翻译器(dynamic binary translator, DBT)运行在宿主机之上. 当客户机和宿主机的指令集架构不同时, 动态二进制翻译技术可以实现跨架构的程序模拟执行. 二进制翻译的基本原理是将客户机代码翻译成语义等价的宿主机代码, 通常以代码块为单位进行翻译. 代码块由一段地址连续、顺序执行的代码组成, 通常以控制转移指令为结尾. 一个客户机代码块通过二进制翻译生成一个语义等价的宿主机代码块, 本文称之为翻译代码块. 软件代码缓存负责存储翻译代码块, 以便后续重复利用, 降低动态翻译开销.

根据程序模拟的层次, DBT 可分为用户级和系统级 2 类, 其代码缓存的实现相近, 都是存储翻译代码块以便复用, 不同的是翻译时指令语义模拟的实现. 如表 1 所示, 用户级 DBT 可以假设客户机永远处于用户态, 只需支持用户态指令. 系统级 DBT 则需要考虑客户机的各种状态, 需要支持所有指令. 为保证系统正确运行, 客户机指令语义模拟需要正确地实现. 以 1 条特权指令为例, 若客户机处于低特权级, 则需要翻译生成触发客户机异常的操作, 而不能直接生成该条特权指令的执行语义, 否则不仅会运行出错, 还可能造成安全问题.

Table 1 Factors in Instruction Semantics Simulation

表 1 指令语义模拟的因素

DBT	客户机状态	指令范围
用户级	单一	用户态
系统级	复杂	全部

系统级 DBT 中语义模拟方案有 2 种: 一种是动态检测方案, 该方案在翻译代码块中生成判断当前客户机状态的宿主机代码, 从而在运行时进行检测; 另一种是状态标签方案, 该方案在翻译时仅依据当前客户机状态, 并将生成的翻译代码块的状态标记为当前客户机状态. 在翻译代码块执行前需进行对比, 确保待执行的翻译代码块的状态与客户机状态一致. 详细介绍见本文第 2 节.

出于对性能的考虑, 目前的系统级 DBT 均采用了状态标签方案以避免运行时开销. 但状态标签方案也会给软件代码缓存引入额外的问题. 本文详细分析了状态标签方案, 总结了 2 类问题: 一类是冲突问题, 源于不同状态的翻译代码块存储在同一个代码缓存空间, 导致空间利用率的下降和不必要的刷新; 另一类是冗余问题, 源于不同状态的翻译代码块可能包含相同的宿主机代码, 意味着多余的翻译开销和额外的空间占用. 详细分析见本文第 4 节.

本文的主要贡献和创新有 3 点:

1)深入分析了系统级动态二进制翻译器中已有的指令语义模拟方案给软件代码缓存管理带来的问题, 并将其总结为冲突问题和冗余问题.

2)针对 1)中 2 类问题, 提出了基于细粒度状态标签的代码缓存优化方案. 其中多状态代码缓存方案可以解决代码块冲突问题, 减少重复的动态翻译开销; 弱状态标签方案可以解决代码块冗余问题, 减少冗余的动态翻译开销.

3)在真实的系统级动态二进制翻译器 LATX-SYS 中实现了 2)中 2 个方案, 并进行了详细的测试与分析. LATX-SYS 基于 QEMU[1] 开发, 可以在 LoongArch[2] 宿主机上运行 x86 客户机操作系统.

1 相关工作

软件代码缓存的研究如表 2 所示, 大多集中于用户级 DBT, 仅有多线程共享机制是针对系统级 DBT.

Table 2 Relative Research on Software Code Cache

表 2 软件代码缓存相关研究

研究内容	相关工作
缓存替换策略	替换策略对比[3]、粗粒度刷新[4-8]、选择性刷新[9]
持久性缓存	代码复用率[10]、基于 PIN[11-12]、基于 DynamoRIO[13]、基于 QEMU[14]、基于 Box64[15]
工作集和热代码	抢先式刷新[16-17]、动态容量调整[17-18]、热代码缓存[4,19]
硬件辅助机制	指令解码缓存[20-21]、利用片上便签存储器[22-23]、代码块查找加速[24-25]
优化内存占用	优化辅助代码[26-28]、热路径选择[29]、代码块重叠[30]
多线程共享机制	线程私有[31-32]、线程共享[32-33]、并行翻译[34]

1.1 缓存替换策略

文献 [3] 对软件代码缓存的替换策略进行了详

细的对比分析，认为全刷新策略和先进先出（FIFO）策略都是最优的，二者的缺失率都很低并且没有碎片化问题. 文献 [4−8] 均提出了粗粒度的刷新方式，其与 FIFO 策略的结合被认为是各方面权衡之下最优方案. 文献 [9] 提出了选择性刷新策略，平衡了内存占用和性能，但又引入了碎片化问题.

1.2 持久性缓存

持久性缓存可以在相同程序的多次执行或不同程序的执行之间进行代码缓存的共享，以降低翻译开销和内存占用. 文献 [10] 最早对此问题进行了分析，统计了相同程序多次运行和不同程序之间的代码复用率. 此后出现若干持久性缓存的具体实现，文献 [11−12] 是基于 PIN 的实现，主要针对相同程序多次运行. 文献 [13] 是基于 DynamoRIO 的实现，进一步研究了特权代码共享的安全问题. 文献 [14] 是基于 QEMU 的实现，总结了此前方案的不足，实现了范围更大的持久性缓存. 文献 [15] 进一步提出了累积式持久性缓存，在多次运行中不断增加缓存的代码块，从而不断提高持久性缓存的命中率.

1.3 工作集和热代码

根据局部性原理，程序运行期间通常会呈现出阶段性的特征，即某时间段内的工作集是有限的，这种特征可以被二进制翻译器识别. Dynamo[16] 提出了抢先式刷新策略，当发现程序可能出现工作集变迁时，便主动进行一次缓存刷新. 文献 [17] 在 FIFO 策略的基础上进行工作集变迁的检测，并动态地调整软件代码缓存容量. 文献 [18] 结合了文献 [16−17] 方案，提出了静态缓存和动态缓存. 静态缓存容量固定，在工作集变迁时提前进行刷新；动态缓存容量可变，在工作集变迁时调整容量. 文献 [4, 19] 依据代码块的执行热度，将热代码存储在一个独立的空间中，从而自适应地识别工作集的变迁，但统计执行热度需要付出额外开销.

1.4 硬件辅助机制

软硬件结合是提高性能的方式之一. 文献 [20−21] 提出了相应的硬件缓存机制来加速指令解码. 文献 [22−23] 利用了嵌入式系统中的片上便签存储器来加速代码缓存的访问. 针对代码块查找过程的加速，文献 [24] 提出了基于 CAM 和 RAM 的硬件查找表. 文献 [25] 则利用了数据缓存的部分空间来存储代码块的映射关系，并使用特殊的指令来加速查找.

1.5 优化内存占用

软件代码缓存会占用内存空间，而内存资源是有限的，尤其在嵌入式环境中. 文献 [26] 优化了控制转移的翻译，减少了翻译产生的代码量. 文献 [27] 发现出口代码占很大空间，并提出若干方案进行优化. 文献 [28] 针对二进制翻译器的各类辅助代码进行优化，包括跳转、间接跳转查找、上下文切换等，减少内存占用. 文献 [29] 则针对超级块的构建，提出了可以平衡内存占用和整体性能的路径选择算法. 文献 [30] 针对代码块重叠问题进行了研究，提出了相应的解决方案来降低内存占用.

1.6 多线程共享机制

系统级 DBT 中软件代码缓存的研究集中于多线程共享机制. 在多核客户机的场景下，每个客户机核心可以使用 1 个宿主机线程进行模拟. 文献 [31] 使用线程私有代码缓存来保证性能. 文献 [32] 则对比了线程私有代码缓存和线程共享代码缓存，研究了内存和性能之间的平衡. 在线程共享代码缓存基础上，文献 [33] 提出了一个更高效的共享哈希表来加速访问，文献 [34] 将共享代码缓存进行分区，来实现多线程并行的代码块翻译. 不同架构的存储一致性模型也不同[35]，文献 [33] 研究了客户机和宿主机的存储一致性模型不同时的正确性问题.

1.7 本文工作

本文的贡献体现在缓存替换策略和优化内存占用 2 个方面：一方面，多状态代码缓存方案区分了不同状态的代码块，各自独立进行刷新，避免了互相的冲突，减少了重复的动态翻译开销；另一方面，弱状态标签方案消除了部分状态带来的冗余代码块，降低了内存占用，减少了冗余的动态翻译开销.

2 二进制翻译和指令语义模拟

2.1 二进制翻译

根据程序模拟的层次，可将 DBT 分为用户级和系统级. 图 1 展示了 DBT 所在的软硬件层次结构. 最

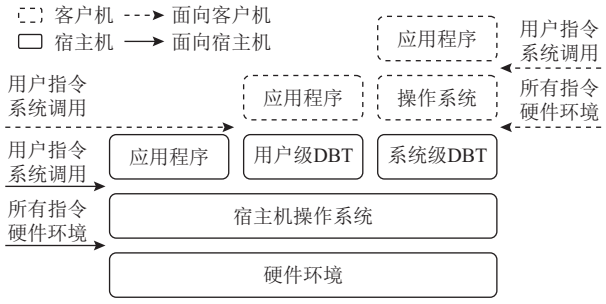


Fig. 1 Software/hardware hierarchy structure of binary translators

图 1 二进制翻译器的软硬件层次结构

底层为处理器等真实硬件环境,其上是宿主机操作系统,再上面是宿主机应用程序. DBT 作为宿主机应用程序,向上提供客户机所需的模拟环境. 用户级 DBT 可以模拟客户机用户态指令,并提供客户机系统调用,实现客户机应用程序的模拟执行. 系统级 DBT 可以模拟所有客户机指令,并提供客户机硬件环境的模拟,实现客户机操作系统的模拟执行.

动态二进制翻译的基本执行流程如图 2 所示. 代码块查找阶段会搜索软件代码缓存,寻找是否已经存在所需的翻译代码块. 若找到,则进入代码块执行阶段,执行这个翻译代码块. 若未找到,则进入代码块翻译阶段,读取客户机二进制,翻译产生对应的翻译代码块,并存储到软件代码缓存中,随后回到代码块查找阶段,便可以找到刚刚产生的翻译代码块,从而继续执行. 代码块执行完成后,回到代码块查找阶段,继续寻找下一个需要执行的翻译代码块.

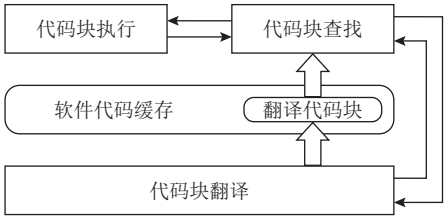


Fig. 2 Execution flow of dynamic binary translation

图 2 动态二进制翻译的执行流程

软件代码缓存是一块用于存储翻译代码块的内存空间. 随着程序的持续运行,翻译代码块会不断产生,而软件代码缓存的空间有限,因此需要替换策略. 表 3 对比了 3 种常见的替换策略. 全刷新策略在代码缓存空间用尽时会清空整个空间,然后继续存储新的翻译代码块. 全刷新的优势在于实现简单、刷新操作更快,劣势是复用率较低、动态翻译开销更大. FIFO 策略会优先把最老的翻译代码块无效掉,腾出空间给新的翻译代码块. FIFO 的优势在于可以尽可能复用已有的翻译代码块;劣势在于无效翻译代码块的操作较为复杂,且由于代码块长度不一,可能需要多次刷新才能获取足够的空间. 全刷新和 FIFO 这 2 种方案共同的优势是没有碎片化问题,有利于简化逻辑、降低刷新开销. 这 2 种方案结合,也被认为是目前最好的方案,即粗粒度的 FIFO 策略. 该策略的一次刷新操作会无效掉某个地址范围内的所有翻译代码块.

Table 3 Comparison of Replace Strategy

表 3 替换策略的比较

替换策略	复用率	碎片化	刷新操作
全刷新	低	无	简单
FIFO	高	无	复杂
粗粒度的 FIFO	较高	无	适中

代码块执行阶段中一个典型的优化是代码块链接. 如果每次执行完 1 个翻译代码块,都需要进行 1 次代码块查找,那么整体性能就会很低. 而客户机基本块的结尾常常是直接跳转指令,其跳转目标是确定的,意味着在当前翻译代码块(记为 *A*)执行完后,下一个待执行的翻译代码块(记为 *B*)也是确定的. 代码块链接会修改 *A* 的结尾处代码,使 *A* 执行完后不再退出执行,而是跳转到 *B* 继续执行. 若客户机基本块的结尾是间接跳转指令,此时跳转目标是不确定的,只有在每次执行后才可知,此时代码块查找是必要的.

2.2 指令语义模拟

为保证正确性,DBT 需要模拟客户机指令的正确语义. 指令执行语义常常与执行时的处理器状态有关,若状态不同,则指令的行为可能完全不同. 在用户级 DBT 中,通常认为客户机一直处于用户态,此时用户态指令的行为一般是固定的. 而在系统级 DBT 中,则需要考虑客户机所有可能的状态,且需要模拟所有的客户机指令,包括用户态指令和特权态指令.

系统级 DBT 在模拟客户机指令语义时有 2 种方案,即动态检测方案和状态标签方案. 动态检测方案在翻译代码块运行时对客户机状态进行动态检测. 具体来说,代码块翻译阶段负责生成读取并判断当前客户机状态的宿主机代码,以及依据判断结果执行对应宿主机代码的选择逻辑. 代码块执行阶段负责执行翻译代码块,从而动态地检测客户机状态并执行对应的指令行为. 状态标签方案在进入执行阶段之前进行检测. 具体来说,在代码块翻译阶段中,依据当前客户机状态,在翻译代码块中仅生成当前状态下客户机指令的执行行为,并将翻译代码块的状态标记为当前客户机状态. 在代码块查找阶段中,将翻译代码块的状态与客户机状态进行对比,只有相同的情况下才能进入代码块执行阶段,否则便是查找失败,进入代码块翻译阶段,生成当前客户机状态下的翻译代码块.

本文以 x86 客户机指令 FADD 和客户机状态 *EM* 为例,描述 2 种方案的基本原理. 当客户机状态 *EM*=1 时,FADD 触发浮点异常;当客户机状态 *EM*=0 时,FADD 执行浮点加法操作. 为了便于说明和理解,本例中客户机基本块仅包含 1 条客户机指令 FADD,且仅考虑客户机的 *EM* 状态.

图 3 展示了指令语义模拟的动态检测方案. 首次执行时, 在代码块查找阶段查询失败, 进入代码块翻译阶段, 随后生成的翻译代码块的开头位置包含了读取和判断客户机状态的代码, 以及 2 种不同的执行行为. 在代码块执行阶段, 根据客户机状态不同, 该翻译代码块的执行行为不同. 当客户机状态 *EM*=1 时, 读取和判断当前客户机状态后, 执行触发客户机异常的操作. 相应地, 当客户机状态 *EM*=0 时, 执行浮点加法操作. 由此可见翻译代码块中的宿主机代码模拟了该指令在不同客户机状态下的不同行为, 保证了正确的客户机指令语义.

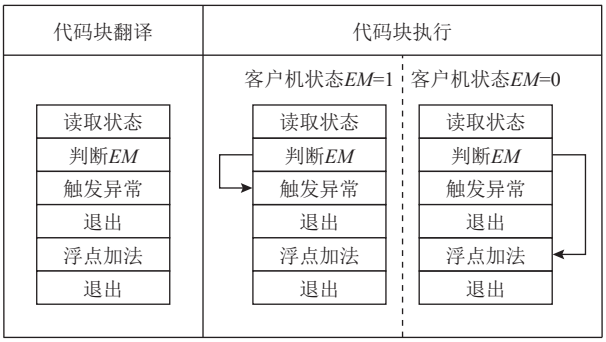


Fig. 3 Dynamic checking scheme

图 3 动态检测方案

图 4 展示了指令语义模拟的状态标签方案. 首次执行时代码块查找失败, 进入代码块翻译阶段, 此时客户机状态 *EM*=1, 翻译代码块 *A* 中仅包含触发客户机异常的宿主机代码, 且翻译代码块 *A* 的状态标签中 *EM* 状态会被标记为 1. 在代码块查找阶段, 会依据当前客户机状态进行判断. 对于翻译代码块 *A*, 当客户机状态 *EM*=1 时, 二者状态一致, 认为查询成功, 可以执行; 当客户机状态 *EM*=0 时, 二者状态不同, 认为查询失败, 则进入代码块翻译阶段. 此时在客户机状态 *EM*=0 时会翻译生成仅包含执行浮点加法操作的翻译代码块, 且其状态标签中 *EM* 会被标记为 0. 由此可见, 翻译代码块的状态标签描述了其所需的执行条件, 从而保证正确的客户机指令语义.

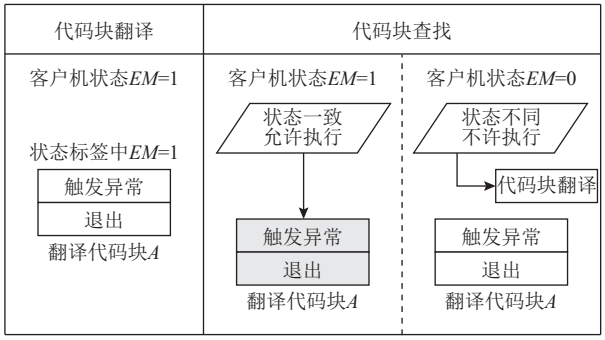


Fig. 4 State label scheme

图 4 状态标签方案

动态检测方案和状态标签方案均能实现指令语义的正确模拟, 但也都存在不足之处. 在代码块翻译阶段, 动态检测方案总是会生成该指令的所有行为, 使得代码膨胀率较高, 增加软件代码缓存和硬件指令缓存的压力. 状态标签方案仅在需要执行时才会生成相应的指令行为, 消除了不必要代码, 但代码块的数量会更多. 在代码块执行阶段, 动态检测方案下, 每次执行时都会至少进行 1 次读取和判断操作, 意味着更多的运行时开销, 从而降低整体的性能表现. 状态标签方案需要在运行前进行 1 次额外的逻辑检查, 但运行时无需额外的检测, 避免了翻译代码块的运行时开销, 有利于提高整体性能.

二进制翻译器所面临的一个重要问题是性能问题, 尤其是系统级 DBT. 目前已有的系统级 DBT 均采用了状态标签方案.

3 软件代码缓存问题分析

状态标签方案通过对翻译代码块添加对应的客户机状态标签, 来避免动态检测的开销. 此时对于同一个客户机代码块, 软件代码缓存中可能会存在多个状态标签互不相同的翻译代码块. 本文对该现象进行深入的研究分析, 总结出了 2 个问题: 一个是冲突问题, 表现为不同状态的翻译代码块存储在同一个软件代码缓存空间中, 在存储、链接和刷新时会互相冲突, 降低空间利用率; 一个是冗余问题. 对于来自同一个客户机代码块的多个翻译代码块, 尽管它们的状态标签不同, 但其内部的宿主机代码可能是相同的, 意味着存在额外的翻译开销和冗余的内存占用.

3.1 代码块冲突问题

冲突问题源于不同状态标签的翻译代码块存储于同一个代码缓存空间中. 典型的客户机状态是特权级. 对于一个包含特权指令的客户机基本块, 在客户机状态处于用户态时, 生成的翻译代码块中应当包含触发客户机异常的行为. 当客户机处于特权态时, 生成的翻译代码块中应当包含该特权指令的正常执行行为. 图 5 展示了 Ubuntu/x86 16.04 系统在启动阶段代码缓存中 2 种状态的翻译代码块的分布. 横坐标是软件代码缓存 FIFO 区域的编号, 每个 FIFO 区域容量为 4 MB, 共有 32 个, 总容量 128 MB. 纵坐标是翻译代码块数量. 系统启动阶段最初执行的是引导程序和内核初始化程序, 客户机处于特权态, 因此仅包含特权级翻译代码块. 当第一个用户程序开

始运行后, 逐渐出现用户级翻译代码块. 此时 2 种翻译代码块交错地存储在软件代码缓存中. 图 6 展示了系统启动阶段完成后, 软件代码缓存中代码块的分布情况. 随着用户程序的执行和软件代码缓存的刷新, 用户级翻译代码块和特权级翻译代码块交错地分布在整个软件代码缓存空间中.

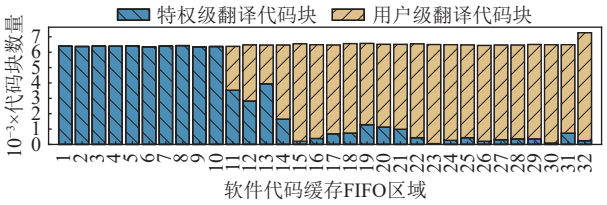


Fig. 5 Code blocks distribution: system startup stage

图 5 代码块分布: 系统启动阶段

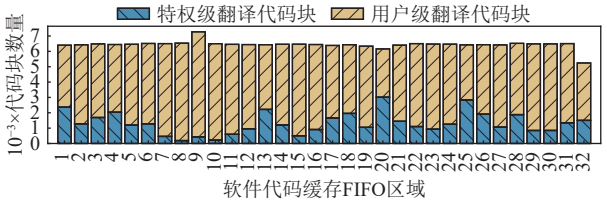


Fig. 6 Code blocks distribution: system running stage

图 6 代码块分布: 系统运行阶段

从代码块分布的角度来看, 同一状态的代码块是比较分散的. 而当客户机处于某个状态下时, 只能执行该状态下的翻译代码块, 但这些代码块分散地存储在整个软件代码缓存空间中, 程序执行的局部性不够好, 增加了硬件指令缓存的压力.

从软件代码缓存的容量限制来看, 同一空间中存储不同状态的翻译代码块是低效的. 在部分架构下, 受限于直接跳转指令的范围, 如 MIPS 架构中直接跳转指令的范围是−128~+128 MB, 为了支持代码块链接机制, 软件代码缓存的容量被限制在 128 MB. 然而不同状态的翻译代码块之间并不能直接跳转, 因此将它们存储在相同的空间中反而是不必要的.

从软件代码缓存的替换策略来看, 对于粗粒度的刷新策略, 1 次刷新操作会无效掉若干翻译代码块, 其中包含多种状态的翻译代码块. 如图 6 所示, 一个 FIFO 区域内既有特权级翻译代码块, 也有用户级翻译代码块. 2 种翻译代码块的执行情况并不相同, 共同进行刷新是不合理的. 实际上, 客户机操作系统在运行中, 执行的用户程序数量多、种类多, 而操作系统始终只有 1 个, 且部分系统代码会经常运行如异常和中断处理程序.

我们将翻译代码块被无效后又重新翻译生成的行为称为代码块的重翻译, 并在 128 MB 软件代码缓存中, 基于 4 MB 粒度的 FIFO 刷新策略, 统计了 Ubuntu/x86 16.04 系统启动过程中的重翻译情况, 如图 7 所示. 总共收集了 80 次 FIFO 刷新时的数据, 每 10 次进行合并统计, 同时也对全部 80 次进行了合计. 我们进一步定义了重翻译距离. 对于一个翻译代码块的重翻译行为, 重翻译距离表示为当它所在的 FIFO 区域刷新后, 又经过了 N 次 FIFO 刷新, 然后该翻译代码块被重新生成. 重翻译距离的范围是 0~31, 大于等于 32 意味着整个软件代码缓存都被刷新过, 因此不予统计. 特别地, 重翻译距离为 0 意味着该翻译代码块被刷新后, 立刻在原 FIFO 区域内被重新翻译生成.

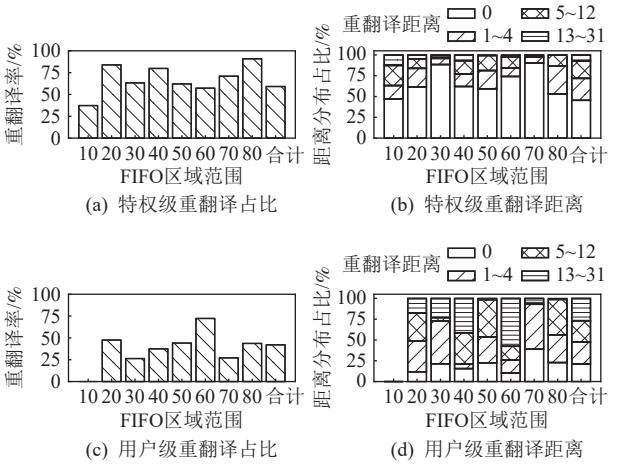


Fig. 7 Retranslation of translation code blocks

图 7 翻译代码块的重翻译

图 7(a)(b)展示了特权级翻译代码块的重翻译情况. 图 7(a)为代码块的重翻译率. 总的来看, 对于被刷新的特权级翻译代码块, 高达 60% 会在后续运行中重新翻译生成. 图 7(b)展示了重翻译距离的分布. 总的来看, 重翻译距离为 0 的情况占比高达 46%, 也就是说有将近一半的特权级翻译代码块在被刷新后立刻就重新生成了. 重翻译距离为 1~4 的情况占比为 24%, 重翻译距离 5~12 的情况占比为 13%. 总的来说, 重翻译距离在 12 以内的情况占比高达 83%.

图 7(c)(d)展示了用户级翻译代码块的重翻译情况. 由于系统启动最初阶段均是系统代码, 因此第 1 列并没有用户态的数据. 图 7(c)展示了重翻译率. 对于被刷新的用户级翻译代码块, 总共有 45% 会重翻译, 明显低于特权级翻译代码块的重翻译率. 图 7(d)展示了重翻译距离的分布. 对于被刷新的用户级翻译代码块, 重翻译距离为 0 的情况仅占 22%, 而大于 12 的情况占比高达 41%, 也就是说用户级翻译代码块被刷新后往往不会立刻被重翻译, 而是运行一段时间后才可能会被重翻译.

3.2 代码块冗余问题

冗余问题源于同一客户机基本块对应的多个状态标签不同的翻译代码块可能包含相同的宿主机指令. 完整的状态标签中包含了多种客户机状态信息，可能会导致代码块冗余问题.

我们发现了导致 x86 客户机出现大量代码块冗余的客户机状态. 在 x86 架构的处理器中有 3 个与浮点和向量指令相关的标志位，分别是 *MP*, *EM*, *TS*. 为了便于描述，我们将这 3 个标志统称为浮点向量控制状态. 当该状态开启时，浮点和向量指令的执行会触发异常；当该状态关闭时，浮点和向量指令可以正常执行. 如表 4 所示.

Table 4 Control Status and Directives of Floating Vector

表 4 浮点向量控制状态及指令行为

浮点向量控制状态	浮点向量指令行为
开启	触发异常
关闭	正常执行

x86 架构通过这个机制来提供针对浮点和向量上下文的延迟处理，当客户机使用该机制时，同一段客户机程序既可能会运行在该状态启动的情况下，也可能会运行该状态关闭的情况下. 而当一个客户机代码块不包含浮点和向量指令时，其执行行为与浮点向量控制状态无关. 在状态标签方案下，同一个客户机代码块会对应多个浮点向量控制状态不同的翻译代码块，而它们的宿主机指令可能是相同的，因此造成了代码块的冗余.

我们将来自同一个客户机基本块的多个状态标签不同的翻译代码块定义为一个相似组. 更进一步地，我们将一个相似组中宿主机代码也相同的多个翻译代码块定义为一个冗余组. 一个相似组的相似数定义为该组内翻译代码块的数量，而一个冗余组的冗余数定义为该组内翻译代码块的数量减 1，即认为至少有 1 个翻译代码块是必要的，其余的则是冗余的. 对于某个时刻下的软件代码缓存，定义相似率为所有相似组的相似数的和与代码块总数的比值，定义冗余率为所有冗余组的冗余数的和与代码块总数的比值. 根据定义可知，冗余率的上限是相似率的一半. 相应地，非冗余率即为 1 减去冗余率，非相似率即为 1 减去相似率. 同时，相似非冗余率即为相似率减去冗余率.

表 5 展示了一个代码缓存的示例. 客户机代码块 A 产生了 3 个翻译代码块 A_1, A_2, A_3, 对应一个相似组，其相似数为 3. 客户机代码块 B 和 C 均仅产生了 1 个翻译代码块，因此不构成相似组. 翻译代码块 A_1 和 A_2 的宿主机指令完全相同，构成一个冗余组，且冗余数为 1. 总的来说，代码缓存的冗余率为 20%，非冗余率为 80%，相似率为 60%，相似非冗余率为 40%.

Table 5 Example of Redundancy Ratio

表 5 冗余率示例

客户机代码块	翻译代码块	状态标签	宿主机指令
A	A_1	*ST*1	*CS*1
	A_2	*ST*2	*CS*1
	A_3	*ST*3	*CS*3
B	B_1	*ST*4	*CS*4
C	C_1	*ST*5	*CS*5

图 8 展示了 Windows XP/x86 系统在启动过程中某些时刻下，软件代码缓存中的相似和冗余情况. 纵坐标使用 FIFO 刷新次数表示系统运行时间，横坐标为代码块数量. 平均而言，软件代码缓存的相似率达到 60%，冗余率则高达 28%. 此处只统计了浮点向量控制状态带来的冗余，详细介绍见 4.2 节.

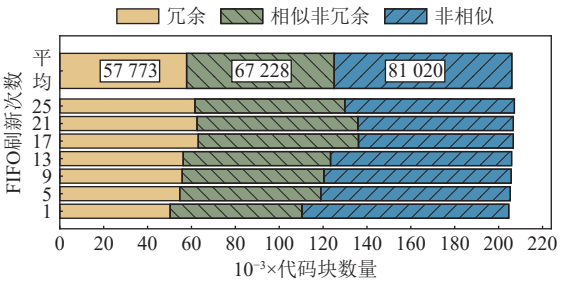


Fig. 8 Redundancy ratio of code block

图 8 代码块冗余率

4 设计与实现

我们总结了状态标签方案造成的 2 类问题：一类是冲突问题；另一类是冗余问题. 本文针对这 2 类问题，分别提出了 2 种基于细粒度状态标签的解决方案. 这 2 种方案并不冲突，可以共存来同时解决这 2 类问题. 针对冲突问题，本文主要解决了由特权级状态带来的代码块冲突情况. 针对冗余问题，本文主要解决了由浮点向量控制状态带来的代码块冗余情况. 本质上来说，本文提出的方案并不局限于特权级状态和浮点向量控制状态，对客户机也不局限于 x86 架构. 对于任何带来代码块冲突和冗余的客户机状态，本文的方案均被使用来解决相应的问题.

4.1 多状态代码缓存

代码块冲突问题的根源在于不同状态的翻译代码块存储在同一个软件代码缓存中，因此解决的方式是将不同状态的翻译代码块存储于不同的软件代码缓存空间以避免互相冲突. 本文提出了多状态代码缓存，支持将特权级代码块和用户级代码块存储于不同的代码块缓存空间中. 区别于其他将代码缓存空间分割的方案，该方案设计的多个代码缓存空间考虑的是系统级 DBT 特有的代码块状态标签，依据状态的不同来分别存储. 因此该方案与其他代码缓存空间的方案是兼容的，例如文献 [2] 提出的根据代码块的执行热度来分别存储生命周期较短的代码块和生命周期较长的代码块，但该方案需要付出额外的开销来统计执行热度.

图 9 展示了针对特权级的多状态代码缓存方案. 针对 x86 的 0 特权级和 3 特权级，分别设计了特权级缓存和用户级缓存. 特权级缓存用于存储特权级翻译代码块，对应客户机的操作系统代码. 用户级缓存用于存储用户级翻译代码块，对应客户机的用户程序代码. 在多状态代码缓存的基础实现中，2 个代码缓存空间的容量都是 128 MB，且均采用 4 MB 粒度的 FIFO 刷新策略. 当客户机运行操作系统时，代码块的执行、查找和翻译均发生在特权级缓存中. 当客户机运行用户程序时，代码块的执行、查找和翻译均发生在用户级缓存中.

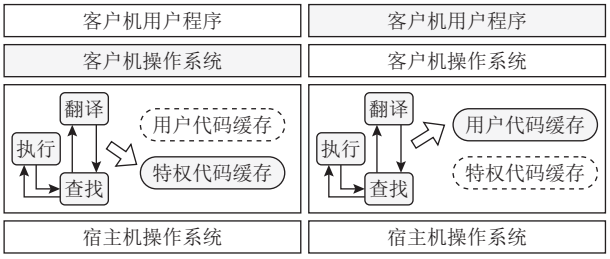


Fig. 9 Multi-state code cache scheme

图 9 多状态代码缓存方案

需要注意的是，x86 的特权级并不只有 0 和 3，还有 1 和 2. 若客户机操作系统使用了 1 特权级和 2 特权级，也可以进一步增加相应特权级的代码缓存空间. 通常来说，x86 操作系统只使用了 3 特权级和 0 特权级，分别对应特权态和用户态，因此本文只处理了 0 和 3 这 2 个特权级. 更进一步，支持 Intel VT-X 扩展的 x86 处理器还拥有一个特殊的 non-root 模式，属于 x86 架构的硬件虚拟化特性. 若 x86 架构的客户机操作系统中运行了使用硬件虚拟化的程序，此时也可能会存在状态为 non-root 的翻译代码块. 在多状态代码缓存方案下，可以将此类翻译代码块存储于独立的空间，与 root 状态的翻译代码块区分开.

从代码块执行的角度来看，处于某个特权级状态下的客户机，例如用户态，通常只会运行用户程序代码，即运行在用户级缓存中，此时缓存空间中全部都是用户级翻译代码块，因此有着更好的指令局部性，有利于硬件的指令预取. 此外，用户级缓存中不再负责存储特权级翻译代码块，也就有了更多的空间来存储用户级翻译代码块，有利于提高代码块查找命中率，降低动态翻译开销. 相应地，多状态代码缓存方案需要更多的内存空间来支持多个软件代码缓存空间. 若受限于内存容量，或设计了过多的软件代码缓存空间，使得单个代码缓存空间的容量过低，反而可能会造成大量的代码块查找缺失，进而产生更多的动态翻译开销，总体上反而会降低性能. 因此缓存空间的数量和各自的容量需要有合理的设计.

4.2 弱状态标签

代码块冗余问题表现为不同状态的翻译代码块有着相同的宿主机代码. 该问题的根源在于客户机状态的不同并不一定会导致客户机指令行为的不同. 本文提出了弱状态标签方案，并针对 x86 架构的浮点向量控制状态进行了实现，消除了由该状态导致的冗余代码块.

图 10 展示了浮点向量控制状态导致冗余代码块的示意图. 客户机基本块 A 包含 2 条客户机指令，分别是定点指令和浮点指令. 在浮点向量控制状态关闭时，基本块 A 会生成翻译代码块 A_1 并将状态标签中浮点向量控制状态记录为关闭. A_1 中的宿主机指令负责完成对应的定点和浮点操作. 在浮点向量控制状态开启时，基本块 A 会生成翻译代码块 A_0 并将状态标签中浮点向量控制状态记录为开启. A_0 中的

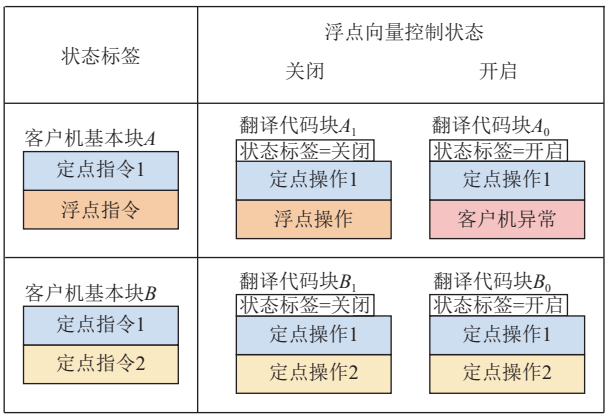


Fig. 10 Example of redundant code block generation

图 10 冗余代码块产生示例

宿主机指令首先完成对应的定点操作，因为定点指令的行为不受影响，而浮点指令对应的则是触发客户机异常. 可见代码块状态标签发挥了作用，正确地处理了浮点指令在不同客户机状态下的不同行为，此时翻译代码块 A_1 和 A_0 都是必要的.

客户机基本块 B 包含 2 条客户机定点指令. 因为定点指令不受浮点向量控制状态的影响，因此无论客户机的浮点向量控制状态是开启还是关闭，定点指令的行为是不变的，因此对应的翻译代码块 B_1 和 B_0 中的宿主机指令便是相同的. 此时翻译代码块 B_1 和 B_0 状态标签不同，但却有着相同的宿主机指令，因此是存在冗余的. 对于客户机的某种状态 S，若一个基本块中不包含受该状态影响的指令，我们将其称之为 S 无关基本块，反之则称为 S 相关基本块. 客户机基本块 B 属于浮点向量控制状态无关基本块，客户机基本块 A 则属于浮点向量控制状态相关基本块.

针对代码块冗余问题，我们提出了弱状态标签方案，在保证客户机指令语义模拟的正确性的前提下，可以消除由特定状态导致的冗余代码块. 弱状态标签方案的基本思想是将导致冗余的状态 S^* 在翻译代码块的状态标签中进行弱化，从而消除客户机状态 S^* 对 S^* 无关基本块的影响，进而避免冗余代码块的产生. 为保证客户机指令语义模拟的正确性，弱状态标签方案引入了额外的动态检测机制来避免出错. 根据检测的粒度不同，可以分为 2 种实现方案：一种是基于代码块的弱状态标签方案；另一种是基于指令的弱状态标签方案. 本文实现了针对浮点向量控制状态进行弱化处理的弱状态标签方案.

图 11 描述了基于代码块的弱状态标签方案. 仍然以基本块 A 和基本块 B 为例. 对于基本块 A 的翻译，仍会生成翻译代码块 A_1 和 A_0 且状态标签中仍然

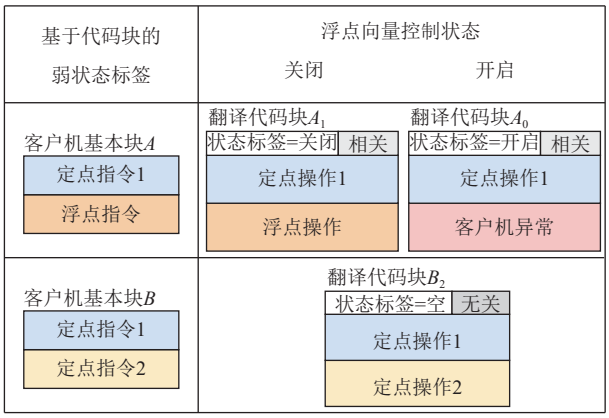


Fig. 11 Code block-based weak state label scheme

图 11 基于代码块的弱状态标签方案

包含客户机当前的浮点向量控制状态信息. 对于基本块 B 的翻译，则只会生成翻译代码块 B_2，其状态标签中不再包含浮点向量控制状态信息. 由此实现了冗余代码块的消除.

为保证正确性，此时翻译代码块会包含一个额外的标记，用来记录对应的基本块与浮点向量控制状态是相关的还是无关的. 基本块 A 是浮点向量控制状态相关的基本块，因此源于基本块 A 的翻译代码块，即翻译代码块 A_1 和 A_0，都会被标记为相关. 相应地，源于基本块 B 的翻译代码块 B_2 会被标记为无关. 在代码块查找阶段，根据此标记来决定是否需要检查浮点向量控制状态. 若相关，则需将翻译代码块的状态标签中记录的浮点向量控制状态与客户机当前的浮点向量控制状态进行比较，不一致时认为查询失败. 若无关，则不必进行该检查.

代码块链接允许执行流从一个翻译代码块直接跳转到另一个翻译代码块，可以减少上下文切换和代码块查找的开销，是二进制翻译器提高性能的关键. 在原本的代码块状态标签方案中，为保证正确性，只允许相同状态标签的翻译代码块进行链接. 而在基于代码块的弱状态标签方案中，部分翻译代码块被消除，状态标签的含义也有所改变，因此需要新的规则来指导代码块链接.

我们以 A_0^*，A_1^*，B_2^* 表示 3 类翻译代码块，类比于图 11 中的翻译代码块 A_0，A_1，B_2，这 3 类分别是：1）A_0^* 表示标记相关且状态标签为开启；2）A_1^* 表示标记相关且状态标签为关闭；3）B_2^* 表示标记无关的 3 类翻译代码块. 表 6 展示了所有的代码链接情况. 从 A_0^* 直接跳转到其他翻译代码块的情况是无需关心的，因为 A_0^* 会触发客户机异常，进而退出代码块执行. A_1^* 不能直接跳转到 A_0^* 的原因是二者的状态标签不同，此规则和原状态标签方案保持一致. 值得注意的是从 B_2^* 直接跳转到 A_0^*，A_1^* 是不允许的，因为当 B_2^* 正在运行时，当前客户机的浮点向量控制状态是不确定的. 若代码块链接使 B_2^* 直接跳转到 A_0^*，那么在客户机浮点向量控制状态关闭时会出错，A_0^* 对应的浮点指令本应正常被执行却错误地触发了客户机异常. 相应地，

Table 6 Analysis of Code Block Link Cases

表 6 代码块链接情况分析

直接跳转	跳转情况
无需关心	$A_0^*\rightarrow A_0^*$，$A_0^*\rightarrow A_1^*$，$A_0^*\rightarrow B_2^*$
允许	$A_1^*\rightarrow A_1^*$，$B_2^*\rightarrow B_2^*$，$A_1^*\rightarrow B_2^*$
不允许	$A_1^*\rightarrow A_0^*$，$B_2^*\rightarrow A_0^*$，$B_2^*\rightarrow A_1^*$

若允许B_2^*直接跳转到A_1^*，则可能使A_1^*对应的浮点指令在本应触发客户机异常的时候正常执行. 反之，A_1^*和B_2^*都是可以直接跳转到B_2^*，因为B_2^*的行为与浮点向量控制状态无关.

为在代码块链接的情况下保证正确性，我们设计了基础方案和优化方案. 基础方案只是简单地不允许可能出错的代码块链接，尽管这可能会带来较高的性能损失. 优化方案如图 12 所示，我们引入了一定程度的动态检测机制，其基本思想是在翻译代码块的起始位置生成一段检查客户机浮点向量控制状态的操作，若状态检测通过则继续执行，若失败则退出执行. 实际上，只有A_1^*和A_0^*这 2 类翻译代码块需要动态检测，B_2^*不需要，而且B_2^*无需在翻译代码块中生成检测代码，从而减少内存占用. 进一步地，我们只需要针对B_2^*到A_0^*，A_1^*进行动态检查，从而尽可能地降低动态开销，为此每个翻译代码块设置了 2 个入口地址，以检测入口对应状态检测代码，执行入口则跳过了状态检测.

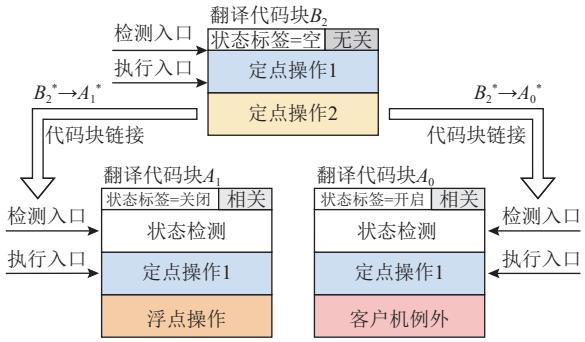


Fig. 12 Optimized scheme for code block-based weak state label

图 12 基于代码块的弱状态标签优化方案

图 13 描述了基于指令的弱状态标签方案. 此方案下所有翻译代码块的状态标签中都不再包含浮点向量控制状态的信息，从根本上消除了冗余代码块. 为确保正确性，我们需要修改受影响的浮点指令和向量指令的翻译过程. 对于客户机基本块 *A*，浮点指令的翻译结果中会首先进行客户机状态检测，此时只需检测浮点向量控制状态即可. 根据检测结果，选择正常执行浮点操作，或者触发客户机异常，从而正确地完成指令语义模拟. 在代码块链接过程中，不再关心浮点向量控制状态，只需检查其他状态是否相同即可.

尽管翻译代码块中的状态标签不再包含浮点向量控制状态的信息，但对于当前客户机状态的维护上还是需要处理该状态，以便进行动态检测. 基础方案下，我们针对每条浮点和向量指令都生成了状态检测代码，实现简单，但会带来较大的内存占用和性能开销. 通过观察指令的行为，我们进一步实现了优化方案. 对于浮点向量控制状态，其只会导致 2 种指令，要么正常执行，要么触发客户机异常. 而在触发客户机异常时，执行流会直接退出代码块执行阶段. 对于同一个基本块中的多条指令，若前面的指令已经进行了状态检测，那么后续指令可以不必再进行检测，从而降低代码量和动态开销. 需要说明的是，实际上本文所述的浮点向量控制状态对应 x86 架构中 *MP*, *EM*, *TS* 这 3 个标记位，不同指令的状态检测行为也有所不同，具体实现中只能针对相同的状态检测行为进行处理，才能保证正确性.

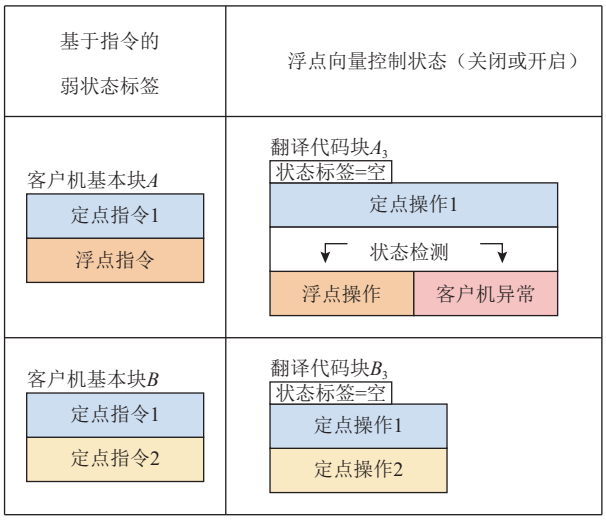


Fig. 13 Instruction-based weak state label scheme

图 13 基于指令的弱状态标签方案

从本质上讲，弱状态标签方案是原有的 2 种指令语义模拟方案的结合，即在动态检测与状态标签之间寻找平衡，引入一定的动态开销的同时，极大地降低由部分特殊的状态标签所导致的冗余代码块，从而减少不必要的动态翻译开销，减少内存占用，缓解软件代码缓存的压力. 我们发现 x86 架构下的浮点向量控制状态是非常适合该方案的一种场景. 一方面，用户程序的同一段代码既可以运行在该状态开启时，也可以运行在该状态关闭时，意味着同一个客户机基本块会对应多个不同状态的翻译代码块，其中可能存在冗余情况. 另一方面，很多用户程序没有或仅有少量的浮点和向量指令，且包含浮点和向量指令的代码所在的位置通常比较接近，这就有利于我们尽可能地降低所需的动态检测开销. 相对来说，客户机的特权级状态并不适合该方案，因为不同特权级下执行的通常是不同的程序，也就不存在冗余问题.

5 实验测试

我们在系统级动态二进制翻译器 LATX-SYS 中实现并测试了本文提出的 2 种方案. LATX-SYS 基于 QEMU 6.0.0 开发，自研了效率更高的 LATX 翻译引擎，能够在 LoongArch[34] 宿主机平台实现完整 x86 架构操作系统的模拟运行. 表 7 展示了实验配置信息.

Table 7 Experiment Configuration

表 7 实验配置

系统层次结构	实验配置
客户机测试程序	系统启动、SPEC CPU2000
客户机操作系统/x86	Windows XP，Ubuntu 16.04
系统级 DBT	LATX-SYS
宿主机操作系统	Loongnix（Linux 4.19 内核）
宿主机硬件平台	3A5000 LoongArch 处理器

所有测试的基准配置是 LATX-SYS 配备单个容量为 128 MB 的软件代码缓存. 基准的替换策略是 4 MB 粒度的 FIFO 刷新策略，优于 QEMU 自身所使用的简单的全刷新方案. 基准的指令语义模拟采用状态标签方案，实现上与 QEMU 保持一致.

针对代码块冲突问题，我们主要观察了代码块分布情况，并对关键数据进行了统计. 针对代码块冗余问题，我们主要分析了软件代码缓存的冗余率. 我们选择了 Windows XP/x86 和 Ubuntu/x86 16.04 这 2 种客户机操作系统，主要测试了系统启动阶段，因为启动阶段的代码量较大，会产生大量的翻译代码块，从而对软件代码缓存管理带来较大的压力. 此外，为了进一步测试弱状态标签方案带来的动态开销，我们基于 SPEC CPU2000 性能测试集进行了测试.

不同的客户机操作系统在是否使用浮点向量控制状态上有所区别. 在 Linux 系统上我们选择了 3.10 和 4.14 这 2 个版本. Linux 3.10 版本使用了浮点向量控制状态来实现延迟上下文切换机制. 而 Ubuntu 16.04 默认的 Linux 4.14 版本则不再支持延迟上下文切换机制，即不再使用浮点向量控制状态. 而 Windows XP 系统则同样使用浮点向量控制状态来实现延迟上下文切换.

5.1 代码块分布

我们在基准配置的基础上，实现了针对特权级的多状态代码缓存，其包含 2 个软件代码缓存空间，分别是用户级代码缓存和特权级代码缓存，容量均为 128 MB 且均采用 4 MB 粒度的 FIFO 刷新策略. 我们测试了 Ubuntu 16.04 在使用 Linux 4.14 内核的启动过程，代码块的分布情况见图 14 和图 15.

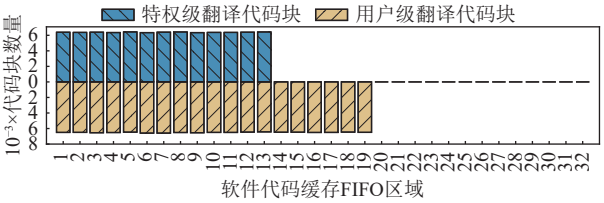


Fig. 14 Code blocks distribution with multi-state cache: system startup stage

图 14 多状态缓存下的代码块分布：系统启动阶段

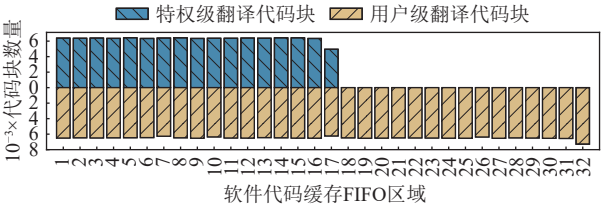


Fig. 15 Code block distribution with multi-state cache: system running stage

图 15 多状态缓存下的代码块分布：系统运行阶段

相比于图 5 和图 6 所示的基准配置下代码块分布情况，多状态代码缓存中特权级翻译代码块和用户级翻译代码块互不干扰，各自拥有独立的空间. 随着系统运行，软件代码缓存的 FIFO 刷新也在各自的空间中进行. 当客户机执行用户程序时，软件代码缓存管理的刷新操作只会作用在用户级代码缓存上，也就只会无效化用户级翻译代码块，而不会无效化特权级代码块. 根据前文的统计，特权级代码块具有更高的重翻译率，将其无效化反而会带来额外的开销.

我们进一步统计了软件代码缓存的一些关键操作，包括刷新次数和翻译次数，详细数据见表 8. 替换测试同为 4 MB 粒度的 FIFO 刷新策略，多状态代码缓存下总 FIFO 刷新次数降低了 44%，刷新开销得到极大降低. 翻译代码块的刷新次数下降了 43%，符合 FIFO 刷新次数降低的情况. 进一步看，用户级翻译代码块的刷新次数降低了 20%，而特权级翻译代码块的刷新次数降低了 100%. 因为多状态代码缓存下，用户级程序执行时只会操作用户级代码缓存，不会干扰特权级代码缓存，且特权级翻译代码块总量不多，足以完全容纳在独立的 128 MB 空间中. 在代码块翻译次数上，总次数降低了 18%. 得益于独立的空间，特权级翻译次数降低了 48%. 用户级翻译次数只降低了 6%，低于用户级刷新导致的下降 20% 的翻译次数，这是因为系统启动阶段有大量用户程序执行，因此

更倾向于执行新的用户代码, 被刷新的用户级翻译代码块更倾向于不被重翻译. 该结果也与图 7 中展示的翻译代码块的重翻译率相吻合, 即特权级翻译代码块有更高的重翻译率, 对其减少刷新会极大降低翻译次数, 而用户级翻译代码块的重翻译率更低, 对其减少刷新所带来的翻译次数的下降也会相对更少.

Table 8 Comparison of Code Cache Operations

表 8 代码缓存操作的对比

统计类别		基准	多状态代码缓存	降低比例/%
FIFO 刷新次数		87	49	44
代码块刷新	总次数	563 491	319 041	43
	用户级	399 310	319 041	20
	特权级	164 181	0	100
代码块翻译	总次数	769 473	633 962	18
	用户级	563 021	526 799	6
	特权级	206 452	107 163	48

软件代码缓存的容量是一个重要因素, 而多状态代码缓存相当于将总容量扩大了, 从而极大地减少了刷新和翻译次数. 为了更好地体现不同状态代码块之间冲突降低带来的提升, 我们进一步测试了在总容量相同的情况下, 多状态代码缓存的具体情况. 总容量设置为 128 MB, 与基准相同, 特权代码缓存与用户代码缓存的容量比例是一个重要参数, 容量调整的粒度是 4 MB, 与 FIFO 刷新的粒度一致.

图 16 展示了不同容量比例下, 多状态代码缓存的代码块翻译次数. 横坐标为特权代码缓存的容量占比, 纵坐标为代码块翻译次数. 最好的情况下, 总翻译次数为 728 979, 此时特权代码缓存容量为 40 MB, 用户代码缓存容量为 88 MB. 图 16 中 2 条横线来自表 8, 实线表示基准方案的总翻译次数 769 473, 虚线表示多状态代码缓存的总翻译次数 633 962. 由此可见在总容量相同的情况下, 合适比例的多状态代码缓存也会优于基准方案. 然而, 如果任意一个代码缓存容量过低, 都会造成严重的空间不足, 进而导致代码块的刷新次数和翻译次数都很高.

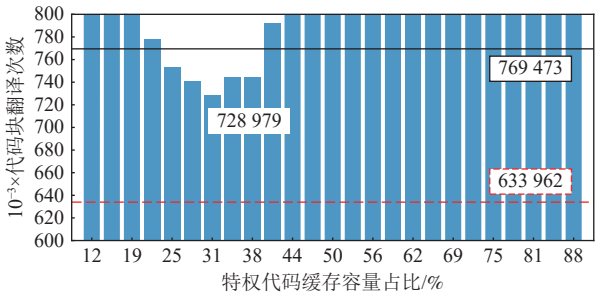


Fig. 16 Translation number of code blocks with different capacity percentage of privileged code cache

图 16 特权代码缓存不同容量占比下代码块翻译次数

5.2 代码块冗余

我们在基准配置的基础上实现了弱代码标签方案, 针对浮点向量控制状态进行冗余代码块的消除. 在实际测试中, 我们进一步区分了特权级翻译代码块和用户级翻译代码块, 如表 9 所示. 在基准配置的基础上, 仅针对用户级翻译代码块进行冗余消除的记为弱状态标签方案(W1), 而针对所有的翻译代码块进行冗余消除的记为弱状态标签方案(W2).

Table 9 Testing Schemes for Code Block Redundancy

表 9 代码块冗余测试方案

测试方案	冗余消除范围
基准	未消除
W1	针对用户级代码块
W2	针对所有的代码块

图 17 展示了 Windows XP 系统启动过程中代码缓存的相似率, 启动过程包括了 3 个运行阶段. 图 17(a)初始阶段中, 基准的相似率高达 55%, 其中特权级相似率达到了 38%, 因为初始阶段有大量系统代码. W1 下相似率降低至 34%, W2 进一步降低到 13.73%. 因为具体实现中只处理了浮点向量控制状态, 其他状态仍可能导致相似代码块, 进而可能产生一定程度的冗余. 图 17(b)启动桌面过程中, 基准的相似率达到了 60%, 用户级和特权级的相似率都较高. W1 降低至 18%, W2 进一步降低至 5%, 此时用户级和系统级的相似率都很低, 意味着大量的冗余是浮点向量控制状态所导致的. 图 17(c)进入桌面后, 用户进行的操作是点击开始菜单, 此时基准的相似率达到 30%, 用户级相似率高达 19%, 意味着此时主要执行用户程序. W1 降低至 16%, 用户级相似率降至 1%. W2 进一步降低至 3%, 绝大部分冗余代码块被消除,

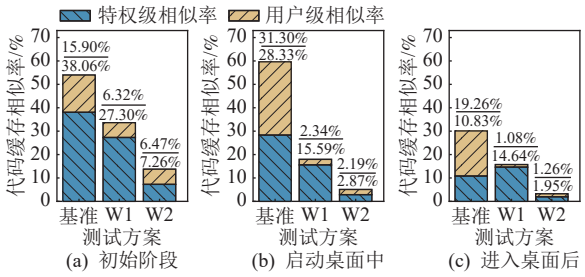


Fig. 17 Code cache similarity ratio in Windows XP system startup process

图 17 Windows XP 系统启动过程代码缓存相似率

只剩下少量由其他状态导致的相似代码块，其中存在些许冗余.

图 18 展示了 Windows XP 启动过程中代码缓存相似率的变化情况，记录了从系统启动到进入桌面的整个过程. 横坐标为运行时间，表现为 FIFO 刷新次数，从而体现软件代码缓存的变化. 特别地，横坐标为 0 指的是软件代码缓存首次存满的时候.

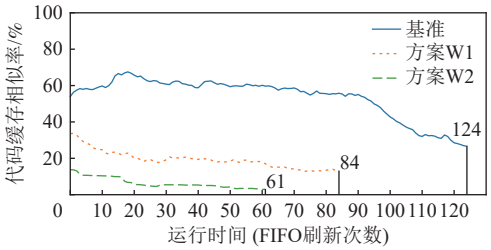


Fig. 18 Change curve of code cache similarity ratio in Windows XP startup process

图 18 Windows XP 启动过程代码缓存相似率变化曲线

根据图 18 可知，基准方案共经历了 124 次 FIFO 刷新，代码缓存相似率明显很高，最高处接近 70%. W1 的代码缓存相似率已有大幅下降，共经历 84 次 FIFO 刷新，代码缓存相似率在 30% 左右，最终降至 20% 以下. W2 的代码缓存相似率只在系统启动初始阶段较高，这是因为初始阶段有固件代码和初始化的系统代码，客户机状态更加复杂，而我们的实现只处理了浮点向量控制状态，其他状态仍可能导致代码块相似. 最终弱状态标签 B 的代码缓存相似率降低至不到 5%，消除了绝大部分冗余代码块.

5.3 综合测试

多状态代码缓存可以避免不同状态代码块之间的冲突，减少不必要的刷新，从而减少了重复的翻译. 弱状态标签可以消除特定状态带来的冗余代码块，从而减少了冗余的翻译. 二者都可以减少动态翻译次数，降低软件代码缓存管理的压力，具体表现是在相同的客户机程序下，总的代码块翻译次数会有所下降. 我们对这 2 种方案测试了单独和组合的情况下，系统启动过程中总的代码块翻译次数.

同时我们也统计了理想的最优情况，此时软件代码缓存的容量设定为无限大，不存在刷新，并且也不存在冗余代码块. 理想情况数据的获取方式是记录所有翻译代码块的相关操作，然后经过离线分析，从中去除因缓存刷新导致的重翻译记录，去除所有的冗余代码块的记录. 结果如图 19 所示.

图 19(a)展示了 Ubuntu 16.04 系统启动过程中，不同方案的代码块翻译次数. 为了体现弱状态标签

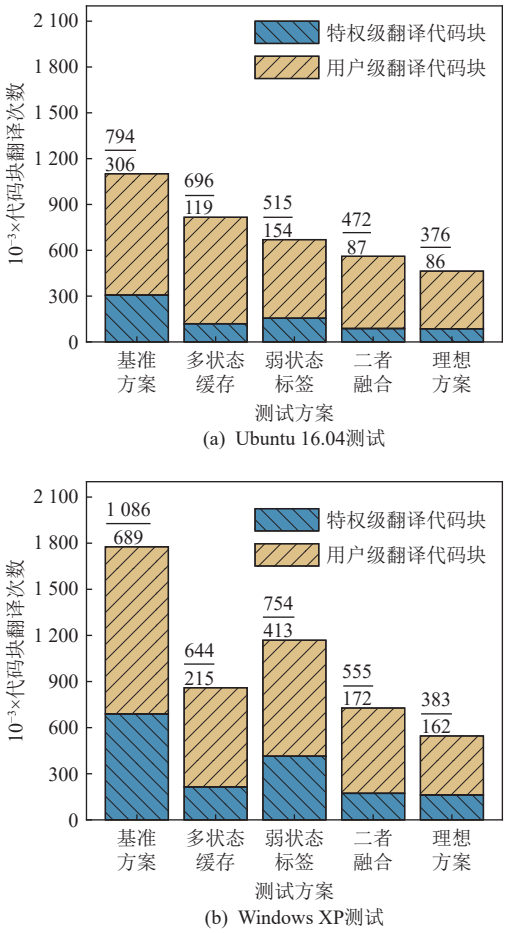


Fig. 19 Comprehensive testing of system startup

图 19 系统启动的综合测试

方案，我们使用了 Linux 3.10 内核并通过内核参数的配置来确保该客户机使用了浮点向量控制状态，从而对本文的优化方案进行测试. 基准方案总翻译次数达到 1 100 000. 仅采用多状态代码缓存时，总翻译次数下降至 815 000，其中特权级翻译次数下降程度更高，源于独立空间带来的优势. 仅采用弱状态标签时，总翻译次数下降至 669 000，用户级翻译次数有较大幅度下降，意味着大量冗余被消除了. 2 种方案相结合时可以达到更好的效果，总翻译次数下降至 559 000，比较接近理想的 462 000. 此时系统级翻译次数已非常接近理想情况，相差的部分源于浮点向量控制状态之外的状态带来的冗余代码块. 离用户级翻译次数还有一定距离，尽管消除了冗余代码块，但仍受限于缓存容量.

图 19(b)展示了 Windows XP 系统启动过程的测试结果，统计了从系统启动到关机的整个过程. 基准方案总翻译次数高达 1 775 000，有大量的系统代码和用户代码，从而对缓存容量更为敏感. 因此，仅采用多状态代码缓存时总翻译次数为 859 000，低于仅采

用弱状态标签时的 1 167 000. 这 2 种方案的结合下总次数进一步降低至 717 000, 但与理想的 545 000 仍有较大距离, 同样源于代码缓存的容量不足.

5.4 性能测试

本文所提出的 2 种方案均可以降低软件代码缓存管理的压力, 表现为代码块刷新、翻译次数降低、减少内存占用. 多状态缓存方案具有更好的指令局部性, 可以减少系统运行过程中的无效刷新, 弱状态标签方案能够减少冗余代码块的翻译开销, 从而提高系统的整体运行效率.

我们测试了 Ubuntu 16.04 系统的启动时间、内核, 并通过内核参数的配置确保该客户机使用了浮点向量控制状态, 从而对本文的优化方案进行测试. 图 20 展示了 LATX-SYS 在不同配置下相对于基准方案的运行时间. 横坐标为多状态代码缓存占总容量 128 MB 的容量比例. 图 20(a)为原始状态标签方案, 图 20(b)为弱状态标签方案. 图 20 总体趋势与图 16 相近, 在合适的容量占例下, 多状态代码缓存的运行时间更短. 图 20(a)中, 实线为基准方案, 因此是 100%. 虚线为使用了 2 个 128 MB 空间的多状态代码缓存方案, 此时整体运行时间减少了约 10%. 图 20(b)中, 实线为弱状态标签方案, 此时大量的冗余代码块被消除, 使得翻译开销和刷新开销得到下降, 整体运行时间降低了 17%. 虚线为多状态代码缓存和弱状态标签的融合方案, 代码块的刷新、翻译次数大大减少, 使得整体运行时间降低了 20%.

弱状态标签方案引入的动态检测机制, 尽管可以降低冗余代码块的翻译和内存占用, 但会带来更

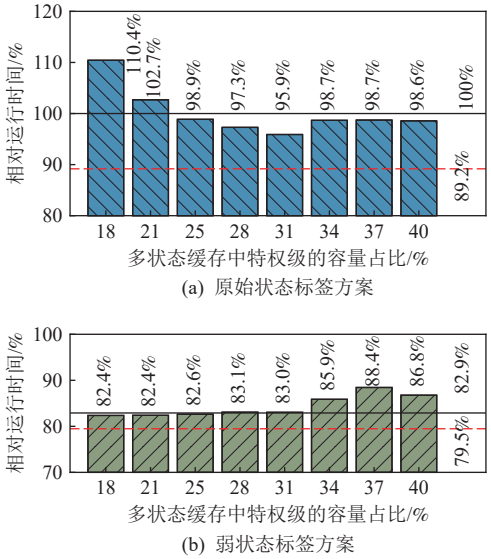


Fig. 20 Time test for system booting

图 20 系统启动时间测试

多的运行时开销, 因此我们针对弱状态标签方案测试并分析了其动态开销. 如表 10 所示, 根据动态检测的粒度不同, 弱状态标签方案的具体实现可以分为 2 类: 一类是基于代码块检测的方式; 另一类是基于指令检测的方式. 每种方式的实现又分为基础方案和优化方案, 因此共有 4 种弱状态标签方案的具体实现. 我们在基准配置的基础上实现了这 4 种方案, 针对客户机 x86 架构中的浮点向量控制状态进行冗余消除, 而且都是针对所有的翻译代码块, 包括特权级和用户级. 我们使用了 SPEC CPU2000 基准测试程序, 客户机是 Ubuntu 16.04 系统, Linux 3.10 内核版本使能了浮点向量控制状态. 我们将基准方案的性能表现作为基准, 测试与其相对的性能表现.

Table 10 Implementation of Four Weak State Label Schemes

表 10 4 种弱状态标签方案的实现

弱状态标签	基于代码块	基于指令
基础方案	BN	IC
优化方案	BC	ICF

注: BN 表示代码块无链接(block no-linking)方案; BC 表示代码块检测(block checking)方案; IC 表示指令检测(instruction checking)方案; ICF 表示指令融合检测(instruction checking fusion)方案.

图 21 展示了基于代码块的弱状态标签方案的性能表现. 基础方案禁止了某些情况下的代码块链接, 因此造成了严重的性能损失, 根据测试结果, 基础方案整数性能下降为 78%, 而浮点性能下降为 55%. 因为浮点测试含有更多的浮点向量指令, 所以有更多的代码块链接情况会被禁止, 导致性能损失严重. 而大部分整数测试没有浮点向量指令, 因此性能几乎不受影响. 优化方案整数性能 97%, 浮点性能 99%, 性能损失非常低.

图 22 展示了基于指令的弱状态标签方案的性能表现. 基础方案整数性能为 91%, 可见翻译代码块执行中的动态检测开销远低于代码块链接被禁止所带来的开销. 以整数测试 175 为例, 其浮点向量指令并不多, 因此动态检测开销不算高, 但在禁止部分, 代码块链接时性能损失非常严重. 此外, 浮点性能仍降低为 55%, 因为浮点测试含有大量的浮点向量指令, 所以有很高的动态检测开销. 而在优化方案下, 整数性能为 99%, 浮点性能为 97%, 性能损失也是非常低.

针对浮点向量控制状态的弱状态标签方案, 我们也在相同的环境配置下测试了该方案对 SPEC CPU2000 基准测试程序的冗余代码块消除情况. 由于测试的均是用户程序, 因此我们只统计了用户级翻译代码

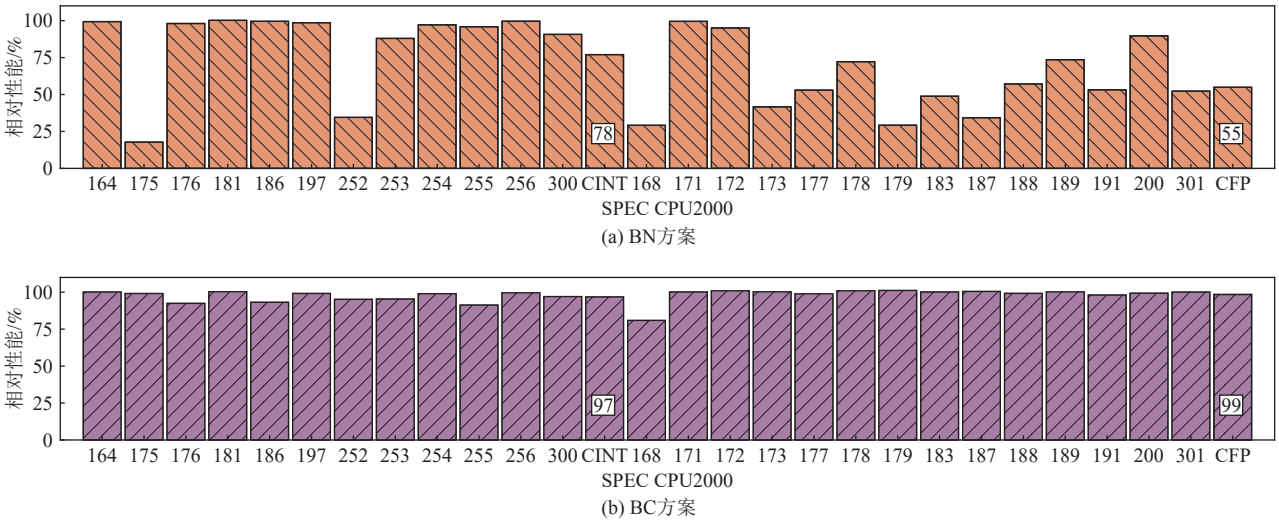


Fig. 21 Performance testing of code block-based weak state label scheme

图 21 基于代码块的弱状态标签方案性能测试

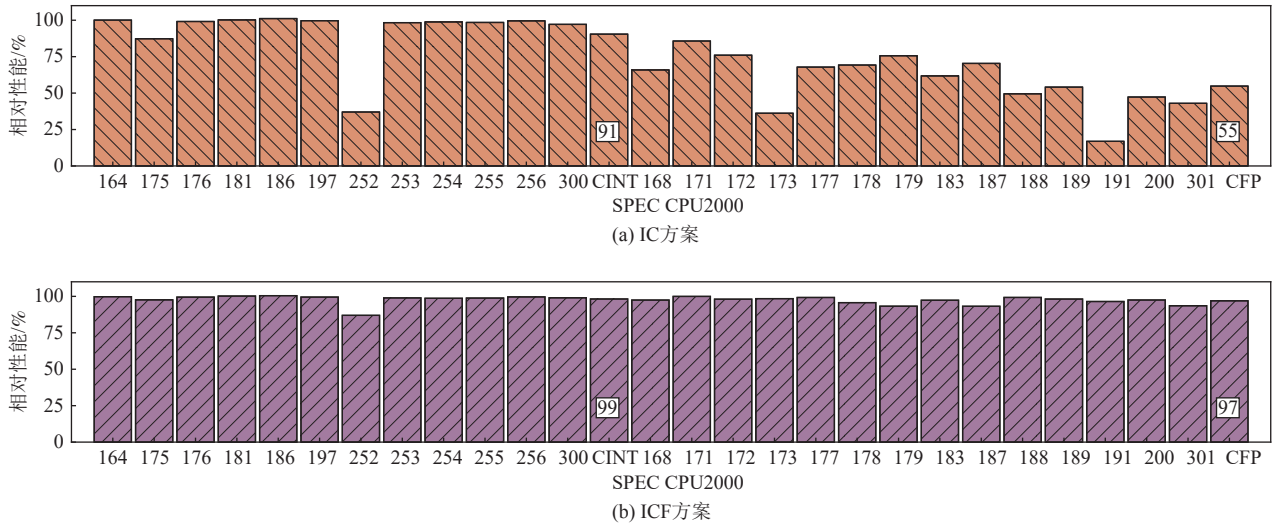


Fig. 22 Performance testing of instruction-based weak state label scheme

图 22 基于指令的弱状态标签方案性能测试

块的冗余消除情况. SPEC CPU2000 通过控制输入数据集的大小来控制运行时间, 但不同数据集使用的是同一个测试程序代码, 因此从运行代码量的视角看来, 小数据集与大数据集的差异并不大, 因此我们基于 test 数据集进行测试和分析.

图 23 展示了弱状态标签方案的冗余消除效果. 图 23 0 线上半部分为基准方案下冗余代码块和非冗余代码块的比例, 平均来说, 整数程序的冗余率为 14%, 略高于浮点程序的冗余率 13%. 整数程序中的最高冗余率为 30%, 而浮点程序的最高冗余率为 23%,

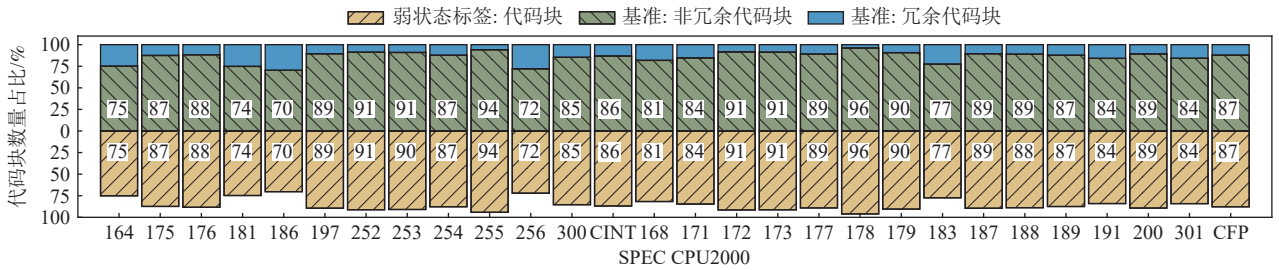


Fig. 23 Code block redundancy of SPEC CPU2000

图 23 SPEC CPU2000 代码块冗余

这是因为只有包含定点指令的基本块才会产生冗余代码块,浮点测试包含大量的浮点向量指令,相对来说更不容易产生冗余.图 23 0 线下半部分为弱状态标签方案下总翻译代码块数与基准方案的总翻译代码块数的比值,平均来说,整数程序的代码块数量下降为 86%,浮点测试的代码块数量下降为 87%,这与基准方案中的非冗余代码块的比例基本一致,可见本文提出的弱状态标签方案充分发挥了作用,将冗余代码块全部消除.

总的来说,SPEC CPU2000 这种处理器密集型程序在二进制翻译的运行环境中对代码块的翻译开销并不敏感,更容易受到动态运行时开销的影响.测试结果表明,优化后的弱状态标签方案引入的运行时开销很低,仅有 2%~3%.

6 总结与展望

本文研究了系统级动态二进制翻译器的软件代码缓存,分析了指令语义模拟的状态标签方案,总结出状态标签对软件代码缓存管理带来的 2 类问题,即冲突问题和冗余问题.冲突问题会导致重复的动态翻译开销,冗余问题会导致冗余的动态翻译开销.针对这 2 类问题,提出了 2 种细粒度状态标签的代码缓存优化方法,分别是多状态代码缓存方案和弱状态标签方案,对于特定的客户机状态,解决由该状态导致的冲突问题和冗余问题.测试结果表明,本文提出的方案可以减少代码块的翻译和刷新开销,有效降低了软件代码缓存管理的压力.

未来的工作包括 2 个方面:一方面是优化本文提出的方案.对于多状态代码缓存,总容量一定时 2 类代码缓存的容量比例是一个关键参数,后续可以设计一个自适应算法进行动态调整.对于弱状态标签,在不同的客户机架构和客户机系统下,定位哪种状态带来了冗余问题是十分关键的,后续可以设计一个自动化分析框架来进行冗余分析.另一方面是进一步深入研究系统级动态二进制翻译器的软件代码缓存管理.例如,相同的客户机可能会被多次启动和运行,后续可以研究一种软件代码缓存持久化方案,从而加快客户机系统的启动速度,提高系统运行效率.

作者贡献声明:牛根负责全文的设计和实现以及论文的撰写和修改;张福新提出实验设计和论文撰写的指导意见.

参 考 文 献

[1] Fabrice B. QEMU: A fast and portable dynamic translator[C]//Proc of the 2005 USENIX Annual Technical Conf (USENIX ATC 05). Anaheim, CA: USENIX Association, 2005: 41−46

[2] Hu Weiwu, Wang Wenxiang, Wu Ruiyang, et al. Loongson instruction set architecture technology[J]. Journal of Computer Research and Development, 2023, 60(1): 2−16 (in Chinese)
(胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术[J]. 计算机研究与发展, 2023, 60(1): 2−16)

[3] Kim H, Michael D S. Code cache management schemes for dynamic optimizers[C]//Proc of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures. Piscataway, NJ: IEEE, 2002: 92−100

[4] Kim H, Michael D S. Managing bounded code caches in dynamic binary optimization systems[J]. ACM Transactions on Architecture and Code Optimization, 2006, 3(3): 263−294

[5] Ma Shulan. TCache replacement algorithm for dynamic binary translation[J]. Computer Applications and Software, 2008, 25(4): 273−275 (in Chinese)
(马舒兰. 动态二进制翻译中的 TCache 替换算法[J]. 计算机应用与软件, 2008, 25(4): 273−275)

[6] Wang Nan, Shan Zheng, Yue Feng. Optimization in code cache management from I386 to Alpha in dynamic binary translation[J]. Journal of Information Engineering University, 2010, 11(6): 688−691 (in Chinese)
(王楠, 单征, 岳峰. I386 到 Alpha 动态二进制翻译中的代码缓存管理优化[J]. 信息工程大学学报, 2010, 11(6): 688−691)

[7] Yin Jinbiao, Song Qiang. TCache management schemes for dynamic binary translator QEMU[J]. Computer Applications and Software, 2012, 29(9): 98−100 (in Chinese)
(殷金彪, 宋强. 动态二进制翻译器 QEMU 的 Tcache 管理策略[J]. 计算机应用与软件, 2012, 29(9): 98−100)

[8] Xu Jinlong, Jiang Liehui, Dong Weiyu, et al. Research on division management mechanism of dynamic binary translation cache[J]. Computer Engineering, 2012, 38(2): 60−62 (in Chinese)
(徐金龙, 蒋烈辉, 董卫宇, 等. 动态二进制翻译缓存的分区管理机制研究[J]. 计算机工程, 2012, 38(2): 60−62)

[9] Apala G, Kim H, Mary S. Balancing memory and performance through selective flushing of software code caches[C/OL]//Proc of the 2010 Int Conf on Compilers, Architectures and Synthesis for Embedded Systems. New York: ACM, 2010[2024-11-11]. https://dl.acm.org/doi/10.1145/1878921.1878923

[10] Kim H, Michael D S. Characterizing inter-execution and inter-application optimization persistence[C]//Proc of the 17th Annual Int Conf on Supercomputing Workshop on Exploring the Trace Space for Dynamic Optimization Techniques. New York: ACM, 2003: 51−58

[11] Vijay J R, Dan C, Robert S C. Persistence in dynamic code transformation systems[J]. ACM SIGARCH Computer Architecture News, 2005, 33(5): 69−74

[12] Vijay J R, Dan C, Robert C, et al. Persistent code caching: Exploiting code reuse across executions and applications[C]//Prof of the 5th Int

Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2007: 74−88

[13] Derek B, Vladimir K. Process-shared and persistent code caches[C]// Proc of the 4th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2008: 61−70

[14] Wang Wenwen, Yew P C, Zhai A, et al. A general persistent code caching framework for dynamic binary translation (DBT)[C]//Proc of the 2016 USENIX Conf on USENIX Annual Technical Conf (USENIX ATC'16). Berkeley, CA: USENIX Association, 2016: 591−603

[15] Lin Haoming, Dong Yong, Chi Wanqing, et al. Efficient dynamic binary translation with accumulative persistent code caching[C]//Proc of the 28th Int Conf on Parallel and Distributed Systems (ICPADS). Piscataway, NJ: IEEE, 2023: 458−466

[16] Vasanth B, Evelyn D, Sanjeev B. Dynamo: A transparent dynamic optimization system[J]. ACM SIGPLAN Notices, 2000, 35(5): 1−12

[17] Derek B, Saman A. Maintaining consistency and bounding capacity of software code caches[C]//Proc of the 3rd Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2005: 74−85

[18] Ma Ruhui, Guan Haibing, Zhu Erzhou, et al. Code cache management based on working set in dynamic binary translator[J]. Computer Science and Information Systems, 2011, 8(3): 653−671

[19] Deng Fei, Gao Feng, Yan Yuanqiang, et al. Research on code cache management strategy based on code heat in dynamic binary translator[C]//Proc of the 18th Int Conf on Software Quality, Reliability and Security Companion (QRS-C). Piscataway, NJ: IEEE, 2018: 647−651

[20] Chen Wei, Shen Li, Lu Hongyi, et al. A light-weight code cache design for dynamic binary translation[C]//Proc of the 15th Int Conf on Parallel and Distributed Systems. Piscataway, NJ: IEEE, 2009: 120−125

[21] Chen Wei, Wang Zhiying, Lu Hongyi, et al. A hardware approach for reducing interpretation overhead[C]//Proc of the 9th Int Conf on Computer and Information Technology. Piscataway, NJ: IEEE, 2009: 98−103

[22] Jose A B, Bruce R C. Heterogeneous code cache: Using scratchpad and main memory in dynamic binary translators[C]//Proc of the 46th ACM/IEEE Design Automation Conf. Piscataway, NJ: IEEE, 2009: 744−749

[23] Hsieh A C, Liu C C, Hwang T T. Enhanced heterogeneous code cache management scheme for dynamic binary translation[C]//Proc of the 16th Asia and South Pacific Design Automation Conf (ASP-DAC 2011). Piscataway, NJ: IEEE, 2011: 231−236

[24] Filipe S, Tiago G, Adriano T, et al. A hardware-assisted translation cache for dynamic binary translation in embedded systems[C]//Proc of the 23rd Int Conf on Emerging Technologies and Factory Automation (ETFA). Piscataway, NJ: IEEE, 2018: 307−312

[25] Li Zhanhui, Liu Chang, Meng Jianyi, et al. Cache load balancing oriented dynamic binary translation[J]. Journal of Computer Research and Development, 2015, 52(9): 2105−2113 (in Chinese)
(李战辉, 刘畅, 孟建熠, 等. 基于高速缓存负荷均衡的动态二进制翻译研究[J]. 计算机研究与发展, 2015, 52(9): 2105−2113)

[26] Jose A B, Bruce R C, Jack W D, et al. Fragment cache management for dynamic binary translators in embedded systems with scratchpad[C]//Proc of the 2007 Int Conf on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07). New York: ACM, 2007: 75−84

[27] Apala G, Kim H, Mary L S. Reducing exit stub memory consumption in code caches[C]//Proc of the 2nd Int Conf on High Performance Embedded Architectures and Compilers (HiPEAC'07). Berlin: Springer, 2007: 87−101

[28] José A B, Bruce R C, Jack W D, et al. Reducing pressure in bounded DBT code caches[C]//Proc of the 2008 Int Conf on Compilers, Architectures and Synthesis for Embedded Systems (CASES'08). New York: ACM, 2008: 109−118

[29] Apala G, Kim H, Mary L S. DBT path selection for holistic memory efficiency and performance[C]//Proc of the 6th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments (VEE'10). New York: ACM, 2010: 145−156

[30] Liu Chang, Chen Zhijian, Meng Jianyi, et al. Compress DBT code cache using control flow identification[J]. Journal of Computer-Aided Design & Computer Graphics, 2014, 26(6): 999−1006 (in Chinese)
(刘畅, 陈志坚, 孟建熠, 等. 利用控制流识别进行二进制翻译代码缓存压缩[J]. 计算机辅助设计与图形学学报, 2014, 26(6): 999−1006)

[31] Wang Zhaoguo, Liu Ran, Chen Yufei, et al. COREMU: A scalable and portable parallel full-system emulator[J]. ACM SIGPLAN Notices, 2011, 46(8): 213−222

[32] Ding Jiun-Hung, Chang Po-Chun, Hsu Wei-Chung, et al. PQEMU: A parallel system emulator based on QEMU[C]//Proc of the 17th Int Conf on Parallel and Distributed Systems. Piscataway, NJ: IEEE, 2011: 276−283

[33] Emilio G C, Paolo B, Alex B, et al. Cross-ISA machine emulation for multicores[C]//Proc of the 15th Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2017: 210−220

[34] Emilio G C, Luca P C. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation[C]//Proc of the 15th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments (VEE'2019). New York: ACM, 2019: 74−87

[35] Hu Weiwu. Shared Memory Architecture[M]. Beijing: Higher Education Press, 2001 (in Chinese)
(胡伟武. 共享存储系统结构[M]. 北京: 高等教育出版社, 2001)



Niu Gen, born in 1996. PhD candidate. Student member of CCF. His research interests include virtualization and binary translation.

牛　根, 1996 年生. 博士研究生. CCF 学生会员. 主要研究方向为虚拟化、二进制翻译.



Zhang Fuxin, born in 1976. PhD, professor. Senior member of CCF. His main research interests include computer architecture, binary translation, and operation system.

张福新, 1976 年生. 博士, 正高级工程师. CCF 高级会员. 主要研究方向为计算机体系结构、二进制翻译、操作系统.