计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(11):3−10 [doi: 10.15888/j.cnki.csa.009330]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 QEMU 的高效指令追踪技术①

王　涛, 秦宵宵, 徐学政, 王　璐, 方　健

(军事科学院 国防科技创新研究院, 北京 100071)
通信作者: 徐学政, E-mail: xuezhengxu@126.com

摘　要: 系统模拟器通过模拟处理器、内存、外设等硬件资源创建一个完整的虚拟计算机环境, 支持运行和调试不同架构的软件, 可大大缩短跨架构的软件开发周期. 模拟器的调试模块通常具有指令追踪功能, 可记录程序运行的指令序列以用于进一步分析, 如程序运行时间评估、程序行为模式分析、软硬件联合仿真等. 支持 RISC-V 架构的主流模拟器 QEMU 和 Spike 均具有指令追踪功能, 但其时间和空间开销过大, 在应对规模较大的应用时效率低下. 本文提出了一种基于 QEMU 的指令追踪技术, 将程序中的基本块、控制流图等静态信息与分支选择等动态信息解耦, 在保证指令序列不失真的同时高效追踪执行序列. 相比 QEMU 原生实现的指令追踪, 本文提出的指令追踪技术的时间开销平均降低了 80% 以上, 空间开销平均降低了 95% 以上. 此外, 本文面向 RISC-V 架构, 实现了多种场景下的指令序列离线分析, 包括指令分类统计、程序热点标记、行为模式分析等.

关键词: QEMU; RISC-V; 指令追踪; 模拟器; 处理器

引用格式: 王涛,秦宵宵,徐学政,王璐,方健.基于 QEMU 的高效指令追踪技术.计算机系统应用,2023,32(11):3–10. http://www.c-s-a.org.cn/1003-3254/9330.html

Efficient Instruction Tracing Based on QEMU

WANG Tao, QIN Xiao-Xiao, XU Xue-Zheng, WANG Lu, FANG Jian

(Defense Innovation Institute, Academy of Military Sciences, Beijing 100071, China)

Abstract: The system emulator creates a virtual environment by emulating hardware resources such as processor, memory, and peripherals, which can support software running and debugging of different architectures and greatly shorten the cross-architecture software development cycle. The emulator usually supports instruction tracing and can be employed for analysis by recording the instruction sequence of program running, such as running time evaluation and behavior pattern analysis related to the program, and joint emulation of software and hardware. As the mainstream emulators supporting RISC-V architecture, both QEMU and Spike support instruction tracing. However, they are time- and space-expensive and inefficient when dealing with large-scale applications. Thus, this study proposes an instruction tracing technology with QEMU. When instructions are traced without distortion, static information such as basic blocks and control flow charts in the program is decoupled from branch selection and other dynamic information. Compared with the native instruction tracing implemented by QEMU, the proposed technology reduces the time overhead by more than 80% and the space overhead by more than 95%. Additionally, based on RISC-V architecture, this study realizes off-line analysis of instruction sequences in various scenarios, such as instruction classification statistics, program hotspot marking, and program behavior analysis.

Key words: QEMU; RISC-V; instruction tracing; emulator; processor

① 基金项目: 国家自然科学基金 (62102439)
本文由“RISC-V 技术及生态”专题特约编辑邢明杰高级工程师、宋威副研究员、张科正高级工程师以及易秋萍副教授推荐.
收稿时间: 2023-03-29; 修改时间: 2023-06-27; 采用时间: 2023-07-21; csa 在线出版时间: 2023-09-15
CNKI 网络首发时间: 2023-09-18

© 中国科学院软件研究所 http://www.c-s-a.org.cn

1 引言

1.1 研究背景

以 QEMU[1] 为代表的系统模拟器通过模拟 CPU、内存、外设等硬件资源创建一个完整的虚拟计算机环境, 支持运行和调试不同架构的软件. 模拟器在芯片设计和验证阶段被广泛使用, 相比各类硬件仿真平台具有明显的速度优势, 这主要得益于其高度抽象的实现方式: 在模拟一个目标设备时, 模拟器可以根据需要对其行为进行抽象建模并实现. 例如, QEMU 忽略了 CPU 的部分微架构实现, 仅对指令集的语义进行建模, 大大加快模拟速度的同时保证了上层软件的跨架构运行, 能够显著缩短跨架构的软件开发周期[2,3].

模拟器的调试模块通常具有指令追踪功能, 可记录程序运行的指令序列以用于进一步分析. 例如: (1) 通过统计指令序列中各类型指令的数目粗略评估程序的实际运行时间; (2) 通过程序各时间段的基本块采样分析程序的行为模式[4,5]; (3) 通过对程序热点片段的标记和提取进行软硬件的联合仿真等. 利用模拟器准确和高效地对目标程序进行指令追踪, 是进行各类分析的基础. 在硬件层面, 芯片通常会提供硬件性能计数器、追踪和调试模块等机制帮助用户进行性能分析[6], 模拟器的调试和追踪模块是对相应硬件抽象模拟, 从而实现跨架构的性能分析. 在软件层面, 可利用插桩技术 (如 Intel Pin[7] 和 DynamoRIO[8,9] 等) 实现指令追踪和性能分析, 相比之下, 模拟器的指令追踪无需修改程序, 不受硬件平台的限制, 可在芯片设计初期实现跨指令集架构的模拟运行. 本文拟研究基于模拟器的程序指令级的追踪和分析. 在系统层面, 用户可利用 OpenTracing[10], OpenTelemetry[11] 等技术对分布式系统中的各组件进行追踪和性能监测.

近年来, RISC-V 架构[12] 凭借其精简、开放、模块化的设计和高可定制的特点在工业界和教育界广受欢迎. 面向 RISC-V 的处理器接连问世, 支持 RISC-V 的系统模拟器也被广泛使用, 如 Spike[13], QEMU[1] 等. 本研究实现了基于 QEMU 的高效指令追踪, 并面向 RISC-V 架构实现了多种基于指令序列的分析.

1.2 研究现状

支持 RISC-V 架构的模拟器 QEMU 和 Spike 均具备指令追踪功能. Spike 是专门面向 RISC-V 架构的模拟器, 能够模拟一个或多个硬件线程. 相较于其他模拟器, Spike 更专注于对 RISC-V 指令集的准确模拟, 而非对多架构以及各类设备的虚拟化, 常作为参考模型用于芯片的测试和验证. Spike 支持追踪程序执行的指令信息, 它会将模拟的硬件线程号、指令地址、指令编码及反汇编信息按照指令执行的顺序记录下来, 以供用户分析.

QEMU 是一个面向多种架构的开源模拟器和虚拟机管理工具, 通过动态二进制翻译机制模拟不同的指令集架构. QEMU 会按块翻译、缓存和执行目标架构的代码, 并支持将翻译和执行过程中处理的指令地址、编码和反汇编信息以日志的形式输出.

Spike 受限于自身的模拟速度和日志记录方式, 其指令追踪功能相比 QEMU 的要慢很多. 图 1 对比了在模拟 NPB 基准套件中的 bt.S.x 时二者的时间 (详细实验配置见第 5.1 节). 可以看出, 在该目标应用下 Spike 的模拟效率仅为 QEMU 的 30%, 开启指令追踪后效率更是不足 QEMU 的 3%.

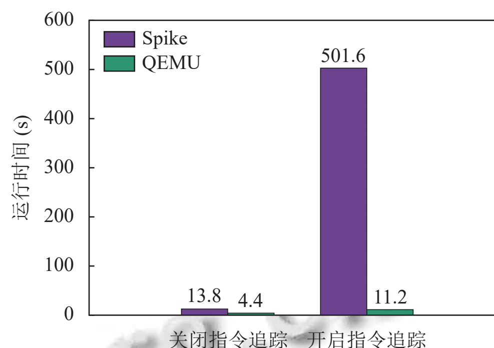

图 1　比较 QEMU 和 Spike 的指令追踪效率

虽然 QEMU 的指令追踪效率优于 Spike, 但仍然无法应对较长的指令序列, 如某些大型软件或循环次数较多的基准测试程序. 仍以 bt.S.x 为例, 在数据规模仅为 *S* (数组大小仅为 12×12×12) 的情况下, QEMU 在开启指令追踪后的模拟时间变为 2.5 倍, 记录的日志文件高达 300 MB, 这并不利于后续基于指令序列的各类分析.

1.3 需求与挑战

通过分析和实践, 我们认为用户对于模拟器的指令追踪功能有至少以下 3 点需求: (1) 序列完整. 指令序列的正确性和完整性是后续分析的基础, 指令追踪应支持记录程序从开始到结束的完整指令序列; (2) 信息多元. 用户对于指令序列分析的需求是多样的, 但所需的信息各不相同. 例如, 指令分类统计需要记录指令

© 中国科学院软件研究所 http://www.c-s-a.org.cn

的类型和频次, 程序热点分析需要记录指令的地址和频次, 而程序的行为分析需要记录不同执行阶段的指令序列; (3) 执行高效. 指令追踪的时间开销和空间开销要在可接受的范围内, 尤其是空间开销, 如果指令序列占用空间过大将不利于后续的分析.

开发满足需求的指令追踪功能面临的主要挑战是: 难以在保证指令序列完整性和信息多元性的同时降低时间和空间开销. 按需进行指令追踪能够一定程度降低开销, 例如, 当用户只需统计指令数目时, 模拟器无需为其提供完整的指令序列, 只需要统计指令频次, 这将大大降低空间开销. 然而, 指令追踪应按需提供多种信息而无需为各类应用单独开发不同的指令追踪程序. 为特定的应用场景开发特定的指令追踪方法难以适应用户多样化的分析需求.

1.4 基于 QEMU 的高效指令追踪技术

本研究提出了一种基于 QEMU 模拟器的指令追踪技术, 将程序中的基本块、控制流图等静态信息与分支选择等动态信息解耦, 在保证指令序列不失真的同时高效追踪执行序列. 相比 QEMU 原生实现的指令追踪, 本文提出的指令追踪技术的时间开销平均降低了 80% 以上, 空间开销平均降低了 95% 以上. 此外, 本文面向 RISC-V 架构, 实现了多种场景下的指令序列离线分析, 包括指令分类统计、程序热点标记、行为模式分析等. 对于不同的序列分析, 可按需还原指令序列, 降低分析开销.

本文第 2 节简要介绍了 QEMU 原生的指令追踪实现, 包括块翻译机制、日志系统和插件机制. 第 3 节介绍了一种基于信息解耦的高效指令追踪技术. 第 4 节介绍了技术的具体实现. 第 5 节介绍了实验设计、结果以及案例分析. 第 6 节进行了总结.

2 QEMU 指令追踪

QEMU 原生的指令追踪[14] 可以基于块翻译和日志系统实现. 例如, 通过开启“in_asm, exec, nochain”调试选项, 可以翻译块为单位记录指令信息以及目标程序的 PC 序列, 经过处理可还原出完整的指令序列.

2.1 块翻译机制

QEMU 的高效模拟得益于其块翻译机制, 它以程序的基本块为单位, 通过在两种架构之间建立 TCG (tiny code generator) 翻译层, 将目标架构的指令翻译成宿主机指令. TCG 可分为前端和后端, 前端将目标架构的指令翻译成中间表示 (TCG IR), 后端再将 TCG IR 翻译成宿主机的指令. 图 2 展示了 TCG 翻译的流程示意. 与编译器类似, 将翻译过程分成前后端的优势在于: (1) 服务于目标架构的前端翻译和面向宿主机架构的后端翻译可以独立实现, 避免前后架构的组合爆炸; (2) TCG IR 不与任何指令集架构相关, 将通用的优化技术应用于 TCG IR, 各个架构都将从中受益.

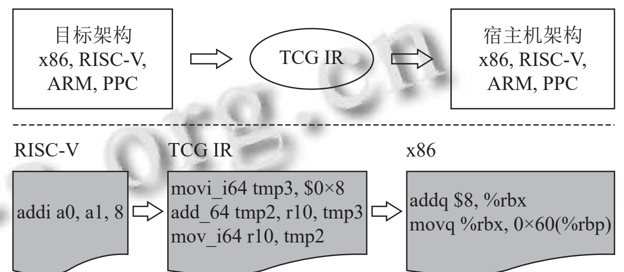

图 2 TCG 代码翻译示意

QEMU 以基本块为单位翻译代码. 当 QEMU 第 1 次翻译一段代码时, 会将连续的目标指令翻译成宿主机指令, 直到遇见跳转指令才停止翻译. 翻译得到的代码块被称作翻译块 (translation block, TB).

QEMU 将翻译好的 TB 缓存并用哈希表进行维护. 在每次执行完一个 TB 时, QEMU 根据需要执行的下一个 PC 值在哈希表中查找对应的 TB 并加载运行. 为了降低查表的开销, QEMU 在 TB 之间建立链接, 使得一个 TB 执行后, 直接跳转并执行下一个 TB, 而无需查找 TB. 这些链接在一起的 TB 被称为扩展的翻译块. TB 的缓存技术和链接技术大大提高了 QEMU 的模拟性能.

2.2 日志系统

QEMU 内部具有功能强大的日志系统, 支持记录 TB 的执行顺序、指令反汇编、中断等. 通过分析 QEMU 产生的日志信息, 开发人员可以了解应用程序的运行行为, 快速定位缺陷. 表 1 给出了 QEMU 支持的常用日志类型. QEMU 的日志信息由各自独自的开关管理, 在启动时通过调试选项开启一个或多个日志类型.

2.3 插件机制

QEMU 提供了插件机制以观察记录目标程序的运行, 可在启动时通过动态链接库的形式加载运行一个或多个插件. 通过插件, 用户可在 QEMU 模拟运行目标程序的多个阶段 (如 CPU 实例化、执行、TB 翻译、系统调用等) 注册回调函数, 分析提取感兴趣的信息. 例如, 可在 TB 执行阶段注册插件函数, 根据作为参数

© 中国科学院软件研究所 http://www.c-s-a.org.cn

传入的 TB 信息分析记录 TB 包含的指令数目、指令类型、TB 的执行次数等. 该函数会在 TB 的执行阶段被调用, 伴随 TB 的每次执行而执行. 利用 QEMU 的插件机制可以实现指令级别的观测记录.

表 1　QEMU 常用日志类型

选项	日志类型
out_asm	TB对应的宿主机汇编指令
in_asm	TB对应的目标程序汇编指令
int	中断和异常
exec	执行的每个TB信息
cpu	进入TB时的CPU寄存器
fpu	进入TB时的FPU寄存器
mmu	MMU相关状态
plugin	插件信息
nochain	禁用TB链接 (打印完整TB序列)
strace	用户态系统调用

插件的实现独立于 QEMU 内部的模拟执行, 并且只能观测、分析和记录 QEMU 暴露的信息 (如 TB), 无法修改机器状态, 保证了程序的模拟执行不受影响.

3　基于信息解耦的指令追踪方法

3.1　解耦静态基本块信息和动态指令序列

程序中的基本块[15] 指的是一段连续的代码, 程序的执行只能从基本块的第 1 条语句进入, 从基本块的最后一条语句离开. 图 3 展示了一个简单的基于基本块的控制流图. 如果基本块的第 1 条指令被执行, 则后续的指令必然被执行, 换言之, 程序运行结束后, 一个基本块内所有指令的执行次数是相同的. 基于以上性质, 指令追踪只需记录基本块的 ID, 后续根据每个基本块的指令信息即可还原完整的指令序列. 例如, 通过记录序列“B0-B1-B4”并通过后续解析这 3 个基本块内部的指令序列即可按需还原完整的指令序列. 该方法的核心思想是: 解耦静态的基本块信息和动态的指令序列, 加快指令追踪的过程, 后续按需还原指令序列, 可大大降低指令追踪的时间和空间开销.

3.2　解耦静态控制流图和动态分支选择

通过分析发现, 在控制流图[15] 中存在部分基本块仅有一个出边 (即指向其他基本块的边), 例如图 3 中的 B1, B2 和 B3. 对于仅有一个出边的基本块, 当其被执行时, 其唯一后继的基本块一定被执行, 记录其后继的基本块信息是冗余的. 例如, 对于执行序列“B0-B2-B3-B4”, B2 的执行必然伴随着 B3 和 B4 的执行. 此外, B0 作为入口基本块也必然被执行. 因此, 序列“B0-B2-B3-B4”可压缩为“B2”.

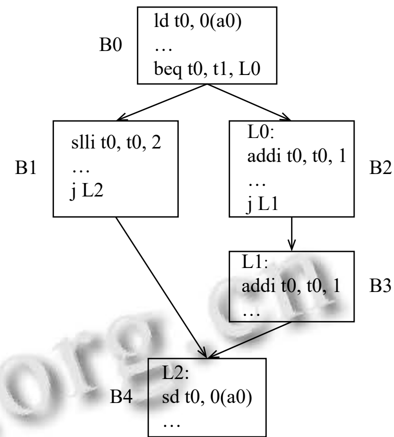

图 3　基于基本块的控制流图示意

上述方法的核心思想是: 将静态的控制流图与动态的分支选择解耦, 仅记录程序在运行至分支点时的选择 (即具有一个以上出边的基本块), 后续通过遍历控制流图对序列进行还原.

3.3　面向需求的指令序列还原

通过前文介绍的两种信息解耦技术, 指令序列分为静态和动态两部分信息分别记录. 静态信息包括程序的控制流图、各个基本块的指令信息等; 动态信息包括程序执行的分支选择. 通过遍历控制流图并根据记录的分支选择序列恢复完整的基本块序列, 并可进一步通过检索基本块信息还原完整的指令序列. 然而, 并非所有的后续分析都需要完整的指令序列, 为了提高序列分析的效率, 可按需对指令序列进行还原. 图 4 展示了上述流程的示意图.

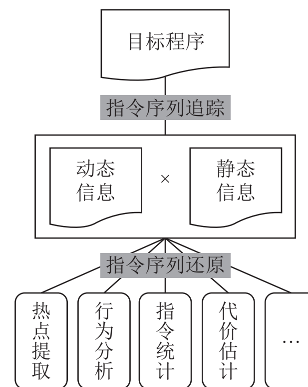

图 4　指令序列的追踪与分析流程示意图

以热点片段提取为例, 用户希望将程序中执行频率较高的片段提取出来, 图 5 展示了如何通过记录的序列

© 中国科学院软件研究所 http://www.c-s-a.org.cn

还原提取需要的信息. 首先, 根据记录的仅包含分支信息的序列“B1-B1”以及左侧的控制流图对基本块序列进行还原. 自起始基本块 B0 开始, 沿着图中的边依次搜索, 当遇到具有一个以上出边的基本块时 (如 B3), 在记录的分支选择序列中依次选择对应的基本块 (如 B1), 直至还原出基本块序列. 实际上, 该应用无需维护基本块的执行顺序, 仅记录执行频率即可, 尤其无需对基本块包含的指令信息进行展开. 经过简单统计, 可以得出各基本块的执行频次, 之后可根据需要筛选出热点片段.

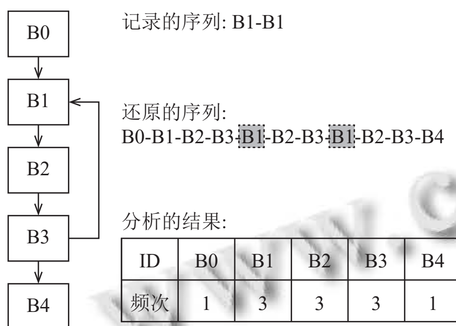

图 5　热点片段提取分析流程示意

可见, 该过程是按需对指令序列进行还原, 一次指令追踪的结果可支持多种离线的指令序列分析.

4　基于 QEMU 的指令追踪实现

实现上述指令追踪技术有两个关键点: (1) 控制流图和基本块等静态信息的生成; (2) 动态分支选择序列的记录. 在本研究中, 上述操作均基于 QEMU 的插件机制实现.

4.1　控制流图生成

QEMU 的块翻译和块链接机制自然构成了程序的控制流图. 依托于 QEMU 的插件机制, 在 TB 翻译时注册函数记录控制图的节点信息, 在 TB 执行时注册函数记录 TB 的跳转信息作为控制流图的边信息. 当程序运行结束时, 通过记录的节点信息和边信息构造出程序的控制流图. 此外, 为了方便频次统计, 在输出控制流图时将各个基本块的执行频次也一并输出.

与传统的控制流图的不同之处在于: (1) QEMU 建立的 TB 及其链接是动态建立的, 相当于在静态控制流图的基础上进行了动态切片. 由于用户对于未被运行的代码并不感兴趣, 该特性并不妨碍指令序列的跟踪; (2) TB 有别于传统基本块的定义. 传统的基本块是静态建立的, 能够确定程序不会跳转到基本块中间, 而 TB 的建立是动态的, 一般是从第 1 条语句一直到分支或跳转语句, 此刻无法确定程序之后不会跳转到基本块中间. 图 6 展示一个简单的循环体建立 TB 的过程, 涉及的 3 条指令中第 1 条为变量初始化 (PC 为 0x10), 后两条为循环的主体 (PC 为 0x14 和 0x18). 当第 1 次块翻译时, TB1 被建立, 3 条指令均被包含在内. 当执行到 0x18 时, 程序产生分支跳转, 目标地址为 0x14, 此时并不存在起始位置为 0x14 的基本块, QEMU 新建了 TB2, 包含 0x14 和 0x18 两条指令, 之后再次跳转到 0x14 时, 存在起始位置为 0x14 的基本块 TB2, 并不会新建 TB. 可见, TB 的概念与基本块并不完全一致, TB 之间可能存在重叠的区域 (如 TB1 和 TB2). 然而, 含有重叠的 TB 并不会影响控制流图的建立和指令追踪的正确性. 例如, 序列“TB0-TB1-TB2”记录的是真实的指令序列. 需要注意的是, 当统计程序热点区域时, 需要对重叠的部分重复计数, 例如 0x14–0x18 这一区域的执行频次应该由 TB1 和 TB2 的频次相加.

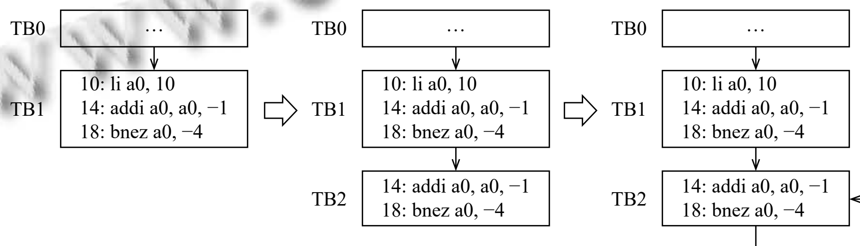

图 6　循环体建立 TB 的过程示意

4.2　基本块信息提取

通过在 TB 翻译阶段注册插件函数, 可记录 TB 的 ID、PC、指令条数、指令反汇编等信息. 该过程仅在 TB 的首次翻译时进行, 每个 TB 只执行一次, 不受动态执行次数的影响.

4.3　序列记录

程序的执行序列指的是 TB 的执行序列. 虽然静态控制流图已经包含了所有的 TB 节点和 TB 间的跳转信

© 中国科学院软件研究所 http://www.c-s-a.org.cn

息, 但是缺少了程序执行的动态分支选择. 当 TB 可跳转至多个 TB 时, 需要按顺序记录其每一次的分支选择. 由于控制流图是动态建立的, 在程序运行结束之前无法确定 TB 是否存在一个以上的跳转目标, 因此需要暂时记录完整的 TB 序列, 待程序运行完毕后, 只需线性扫描控制流图中的 TB, 输出分支处选择的 TB 而忽略其他 TB.

5 实验与结果

5.1 实验环境

本研究中的代码基于 QEMU v7.2, 采用插件的形式实现. 实验基于 RISC-V 用户态 QEMU 开展, 实验平台采用 Intel Core i7-9750H 处理器, 内存 16 GB, 运行 Ubuntu 20.04. 图表绘制基于 gnuplot[16] 实现.

实验测试程序选取自常用的基准测试, 包括 Dhrystone[17]、whetstone[18] 以及 NPB[19]. 其中, Dhrystone 的循环次数为 10^6, whetstone 循环次数为 10^5, NPB 采取串行版本中的 *S* 和 *W* 规模. 程序均由 RISC-V 的 gcc 工具链编译, 目标指令为 RV64GC.

5.2 实验结果

表 2 给出了基于常用基准测试的指令追踪实验结果. 其中, 本研究所生成的日志文件包括静态和动态信息两部分, 其中静态控制流图输出为 JSON 格式, 动态指令序列输出为二进制文件, 每个基本块 ID 由 32 位整型表示, 表 2 中所列日志大小是二者相加的结果.

表 2　基于常用基准测试的指令追踪实验结果

测试程序	执行指令数	时间 (s)	开启原生的指令追踪		使用本研究的指令追踪		时间开销比率 (%)	空间开销比率 (%)
			时间 (s)	日志 (MB)	时间 (s)	日志 (MB)		
Dhrystone	3.3×10^8	0.7	137.5	5900	6.8	99.0	5.0	1.7
whetstone	1.7×10^9	17.7	379.2	16000	37.6	343.0	9.9	2.1
bt.S.x	4.0×10^8	4.4	11.2	313	5.0	10.4	44.0	3.3
cg.S.x	4.0×10^8	1.1	94.2	4100	8.8	174.2	9.3	4.2
dc.S.x	2.0×10^8	0.4	78.6	3600	5.4	86.9	6.9	2.4
ep.S.x	1.8×10^9	27.3	243.8	9800	44.6	309.2	18.3	3.2
ft.S.x	5.5×10^8	4.5	64.4	3000	9.9	112.2	15.4	3.7
is.S.x	3.1×10^7	0.2	6.3	263	0.8	12.4	12.7	4.7
lu.S.x	1.5×10^8	1.5	5.2	151	2.0	6.0	38.4	4.0
mg.S.x	3.8×10^7	0.3	4.7	181	0.8	8.8	17.0	4.9
sp.S.x	1.8×10^8	1.6	10.9	417	2.6	14.5	23.9	3.5
ua.S.x	2.7×10^9	16.0	323.9	14000	42.6	591.0	13.2	4.2
bt.W.x	1.3×10^{10}	138.1	365.0	9000	147.4	269.4	40.4	3.0
cg.W.x	2.6×10^9	6.4	670.7	29000	23.9	1163.0	3.6	4.0
dc.W.x	2.0×10^{11}	503.8	—	—	2262.1	160000.0	—	—
ep.W.x	3.7×10^9	55.3	511.6	20000	69.8	616.2	13.6	3.1
ft.W.x	1.2×10^9	9.3	160.0	7400	14.2	277.2	8.9	3.7
is.W.x	5.1×10^8	2.9	101.7	4100	6.2	179.0	6.1	4.4
lu.W.x	2.5×10^{10}	241.5	710.4	19000	251.1	738.4	35.3	3.9
mg.W.x	2.2×10^9	14.9	223.4	8800	20.0	410.3	8.9	4.7
sp.W.x	2.5×10^{10}	211.8	1431.0	53000	247.0	2050.0	17.3	3.9
ua.W.x	1.6×10^{10}	95.9	1857.0	79000	148.4	3400.0	8.0	4.3

对比 QEMU 原生的指令追踪实现, 本研究提出的指令追踪技术在运行时间和日志空间占用上均有明显的提升, 其中运行时间平均仅为 QEMU 原生实现的 17.0% (3.6%–44.0%), 日志空间占用平均仅为 3.7% (1.7%–4.9%). 值得注意的是, 在本实验环境下程序 dc.W.x 未在可接受的时间内基于原生实现完成指令追踪, 手动结束运行时其日志大小超过 2 TB, 运行时间超过 10 h.

为了预估在给定时间/空间限制下指令追踪的最大长度, 实验选取了部分耗时较长的测试统计了时间/空间开销和指令序列长度的关系. 图 7 给出了指令序列长度与运行时间的关系. 整体上, 二者为线性正相关, 各用例的时间增长率因其热点片段的指令特点不同而有所差异. 例如, 对于 ep.S.x 每记录 10^8 条指令耗时约

© 中国科学院软件研究所 http://www.c-s-a.org.cn

2 s, 而 ua.S.x 耗时约 1 s. 图 8 给出了指令序列长度与日志空间的关系. 类似地, 二者为线性正相关, 各用例的日志空间增长率因其热点片段的基本块大小等因素而有所差异 (见第 3.1 节). 例如, 对于 ep.S.x 每记录 10^8 条指令占用 25 MB, 而 cg.S.x 需要 50 MB. 此外, 由于程序起始和结束阶段的行为区别于热点区域 (见第 5.5 节), 时间和空间开销的规律可能也有所不同.

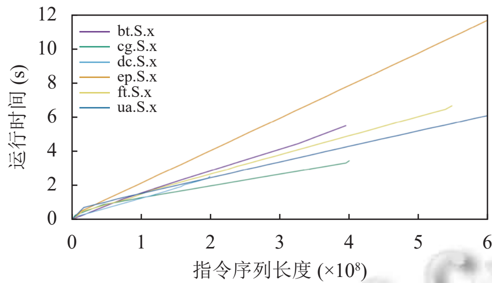

图 7　指令序列长度与运行时间的关系

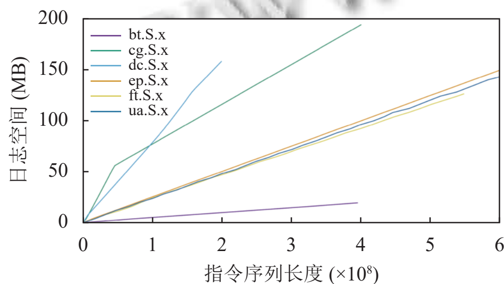

图 8　指令序列长度与日志空间的关系

5.3　热点标记

基于 bt.S.x 的指令追踪结果进行热点基本块的提取 (方法参见图 5), 可根据基本块执行频率迅速定位程序的热点区域. 经分析, 函数 binvchr 包含了执行频次最高的几个基本块 (执行频次 20 000 以上). 图 9 展示了 bt.S.x 的热点区域的汇编代码. 由于该应用无需对完整的指令序列进行分析, 仅统计基本块频次即可, 分析过程仅耗时 1 s.

5.4　指令统计

基于 bt.S.x 的指令追踪结果进行指令统计并绘制直方图 (图 10). 其中, 横轴是指令的频次, 纵轴是高频指令的名称 (由 RV64GC 组成). 经统计, 该程序以浮点运算为主, 使用频次最高的指令是双精度乘减取反指令 (fnmsub.d).

该应用与热点片段提取类似 (参见图 5), 无需考虑指令的相对顺序, 可通过先统计基本块频次和基本块中各指令的频次, 二者相乘即可得到指令统计信息. 由于该应用可按需恢复指令序列, 无需对完整的指令序列进行分析, 分析和绘制过程仅耗时 1 s.

```
0000000000018ecc<binvcrhs>:
18ecc:   00068797   auipc       a5, 0×68
…
18eda:   02853e07   fld         ft8, 40(a0)
18ede:   05053807   fld         fa6, 80(a0)
18ee2:   1af777d3   fdiv.d      fa5, fa4, fa5
18ee6:   00853087   fld         ft1, 8(a0)
…
18fdc:   421cf44b   fnmsub.d    fs0, fs9, ft1, fs0
18fe0:   a21c7a4b   fnmsub.d    fs4, fs8, ft1, fs4
18fe4:   9a1bf9cb   fnmsub.d    fs3, fs7, ft1, fs3
18fe8:   921b794b   fnmsub.d    fs2, fs6, ft1, fs2
18fec:   4a1af4cb   fnmsub.d    fs1, fs5, ft1, fs1
18ff0:   d217f0cb   fnmsub.d    ft1, fa5, ft1, fs10
…
```

图 9　bt.S.x 热点区域汇编代码

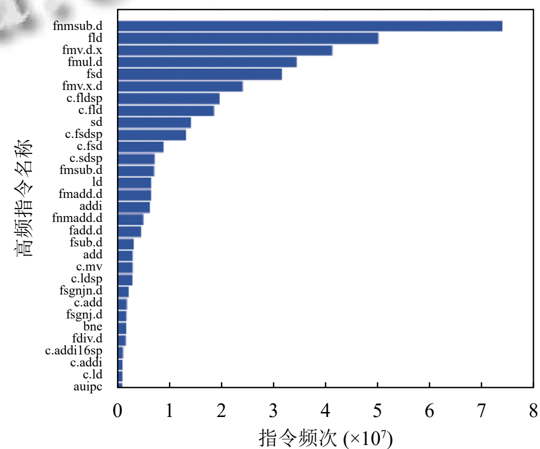

图 10　基于 bt.S.x 指令序列的指令统计

5.5　行为分析

按照指令序列的先后顺序, 标记执行的基本块, 有助于分析程序运行过程中的行为模式. 该过程可通过区间采样和聚类算法进一步分析[4,5]. 进行基本块序列恢复过程仅耗时 2 s, 图表绘制需要额外 40 s.

图 11 给出了程序在执行各个阶段的基本块分布, 其中横轴为程序执行的指令数 (截取了前 3.5×10^7 条指令), 纵轴为基本块 ID (按照频次选取前 300 个基本块). 经过分析可见, 程序运行起始阶段的行为与后续有所不同, 起始阶段可能在进行程序的初始化, 后续阶段是有规律的循环运行某一任务.

5.6　问题与不足

本研究存在 4 点问题需要进一步改进: (1) 虽然本研究所提技术相比 QEMU 原生实现在时间和空间效率上均有明显提升, 但对于超大型应用程序依旧无能为力, 未来可通过限定跟踪范围 (通过目标程序前后插入全局符号或限定 PC 值) 对部分程序片段进行指令跟

© 中国科学院软件研究所 http://www.c-s-a.org.cn

踪; (2) 暂未实现多核处理器的指令序列跟踪, 该功能未来可通过增加线程 ID 将指令序列按线程记录; (3) 暂未考虑指令动态更新. 如果在程序运行中代码段被修改 (如 Linux Kernel 中的动态指令注入), 目前的实现无法对前后的 TB 进行区分. 未来可通过保存更新前后的 TB 进行优化; (4) 实验基于 QEMU 用户态模拟器, 并未跟踪系统调用的指令序列. 通常, 用户仅对系统调用号感兴趣, 未来可通过补充系统调用号进一步完善指令序列追踪. 追踪系统调用的具体指令序列, 可使用 QEMU 全系统模拟器.

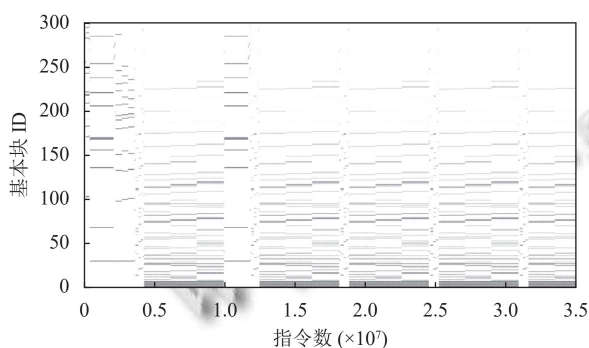

图 11　基于 bt.S.x 指令序列的程序行为分析

6　总结

本研究提出了一种基于 QEMU 的高效指令追踪技术, 通过动态和静态信息解耦的方式对指令序列进行高效的记录. 实验表明, 本研究提出的指令追踪技术相比 QEMU 的原生实现, 时间开销平均降低了 80% 以上, 空间开销平均降低了 95% 以上. 此外, 本研究面向 RISC-V 实现了多种指令序列分析应用. 实验表明, 基于生成的静态和动态信息可高效地按需还原指令序列满足不同分析的需求.

参考文献

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

2 徐学政, 王涛, 方健, 等. 面向 RISC-V 的汇编程序语义等价性自动化测试系统. 计算机系统应用, 2021, 30(11): 33–40. [doi: 10.15888/j.cnki.csa.008348]

3 刘阳, 汪丹, 方林伟, 等. 基于 RISC-V 的数据安全指令. 计算机系统应用, 2023, 32(1): 392–398. [doi: 10.15888/j.cnki.csa.008896]

4 Weaver VM, McKee SA. Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy. Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers. Göteborg: Springer, 2008. 305–319.

5 Sherwood T, Perelman E, Hamerly G, *et al*. Automatically characterizing large scale program behavior. ACM SIGPLAN Notices, 2002, 37(10): 45–57. [doi: 10.1145/605432.605403]

6 Matraszek M, Banaszek M, Ciszewski W, *et al*. FrankenTrace: Low-cost, cycle-level, widely applicable program execution tracing for ARM cortex-M SoC. Proceedings of the 2023 Cyber-physical Systems and Internet of Things Week. San Antonio: Association for Computing Machinery, 2023. 72–77.

7 Patil H, Cohn R, Charney M, *et al*. Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation. Proceedings of the 37th International Symposium on Microarchitecture. Portland: IEEE, 2004. 81–92.

8 Bruening DL, Amarasinghe S. Efficient, transparent, and comprehensive runtime code manipulation [Ph.D. Thesis]. Cambridge: Massachusetts Institute of Technology, 2004.

9 Bruening D, Amarasinghe S. Maintaining consistency and bounding capacity of software code caches. Proceedings of the 2005 International Symposium on Code Generation and Optimization. San Jose: IEEE, 2005. 74–85.

10 OpenTracing. https://opentracing.io/. (2023-03-23)[2023-03-29].

11 Boten A, Majors C. Cloud-native Observability with OpenTelemetry. Birmingham: Packt Publishing, 2022. 42–45.

12 RISC-V 规范. https://github.com/riscv/riscv-isa-manual. (2023-03-21)[2023-03-29].

13 Spike simulator. https://github.com/riscv/riscv-isa-sim. (2023-03-16)[2023-03-29].

14 Velea R, Togan M. Instruction tracing and application profiling with QEMU. MTA Review, 2017, 27(2): 73–76.

15 Aho AV, Lam MS, Sethi R, *et al*. Compilers: Principles, Techniques, and Tools. 2nd ed., Addison-Wesley Pub., 2006.

16 Williams T, Kelley C. gnuplot: An interactive plotting program. http://www.gnuplot.info/docs_4.0/gnuplot.html. (1998-12-03).

17 Weicker RP. Dhrystone: A synthetic systems programming benchmark. Communications of the ACM, 1984, 27(10): 1013–1030. [doi: 10.1145/358274.358283]

18 Curnow HJ, Wichmann BA. A synthetic benchmark. The Computer Journal, 1976, 19(1): 43–49. [doi: 10.1093/comjnl/19.1.43]

19 Bailey DH, Barszcz E, Barton JT, *et al*. The NAS parallel benchmarks. Technical Report, Moffett Field: NASA, 1991.

(校对责编: 孙君艳)

© 中国科学院软件研究所 http://www.c-s-a.org.cn