电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER THESIS

论文题目 **基于 QEMU 的热点代码探测与动态优化模型的研究与实现**

学 科 专 业	**计算机软件与理论**
学　　号	**201021060230**
作 者 姓 名	**张世宜**
指 导 教 师	**杨国武　教 授**

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

基于 QEMU 的热点代码探测与动态优化模型的研究与实现

（题名和副题名）

张世宜

（作者姓名）

指导教师 杨国武 教 授

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 学科专业 计算机软件与理论

提交论文日期 2013.03 论文答辩日期 2013.05

学位授予单位和日期 电子科技大学 2013 年 06 月 29 日

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

RESEARCH AND IMPLEMENTATION OF QEMU-BASED HOTSPOT CODE DETECTION AND DYNAMIC OPTIMIZATION MODEL

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Major: Computer Software and Theory

Author: Zhang Shiyi

Advisor: Yang Guowu

School: School of Computer Science & Engineering

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名：_____ 日期： 年 月 日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名： 导师签名：

日期： 年 月 日

摘 要

二进制翻译技术是实现软件跨平台移植和硬件仿真的核心技术，采用二进制翻译技术能使不同处理器体系架构的可执行目标文件能够跨硬件平台以及跨操作系统执行，同时还能实现硬件逻辑的验证，系统软件的调试等。动态二进制翻译技术是目前最热门的二进制翻译技术。它采用边翻译边执行的策略实现对目标架构的指令向本地架构的指令转化，具有实时翻译、迅速响应、多源多目标等特性。然而，动态二进制翻译技术还处于不断发展的阶段，有许多尚待完善之处，例如，将动态二进制翻译技术与硬件技术结合，以加速仿真器的执行；引入新型的优化算法来提高代码翻译的质量等。如何实现快速翻译和优化一直是动态二进制翻译技术的研究热点。

QEMU 是一个典型的动态二进制翻译系统，它能实现对目标架构指令的实时翻译和执行，并具有多源多目标、快速翻译、支持自引用/自修改代码等特性。然而，较之于快速翻译，在其生成的主机代码中，有许多不必要的内存存取操作以及寄存器移动操作，这些指令的执行将导致极大的开销；另外，QEMU 采用串行化的方式来实现目标指令翻译、优化和执行，这种方式将造成许多优化无法进行，这是由于优化操作需要进行耗时的分析，造成极大的优化开销，常常得不偿失。

文章对 QEMU 的 TCG 翻译引擎进行了详细研究，并在此基础上提出了一种热点代码探测与动态优化模型：结合处理器多核心以及多线程技术，通过代码插装及 NET 算法实现 QEMU 的热点代码探测，将探测到的热点代码块进行合并，生成一个超级块，并对超级块进行深度优化，以产生精简的主机代码。文章采用多线程技术，让合并和优化例程在不同的线程中执行，这使得核心仿真线程不会考虑优化算法带来的开销。多个线程同时在不同的处理器核心中并行执行，使得原始 QEMU 的串行化执行转换为并行化执行，有效地提升处理器的利用率，从而改进 QEMU 的性能。此外，文章还提出一种新颖的优化方法——委托机制，该机制能够有效消除代码中存在的内存加载操作以及寄存器移动操作，以提升翻译后代码的质量，从而达到代码优化的目的。以 QEMU-ARM 仿真平台进行测试，实现结果表明，该模型能够有效提升 QEMU 的平均执行性能约 10%。

关键词：二进制翻译，QEMU，热点探测，委托机制

ABSTRACT

Binary translation technology is the core technology of software portable across platforms and hardware simulation, which enables the executable object file of different hardware/software platforms to be executed on another one, and is convenient for hardware logic verification and system software debugging as well.

Dynamic binary translation technology is currently the most popular binary translation technology. It takes the strategy of real-time translation and execution to transform the binary code of the object CPU architecture into the local one, and equips with characteristics of real-time translation, rapid response, multi-sources and multi-targets and so on. Dynamic binary translation, however, is still on the stage of continuous development and needed to be perfected. For example, combines with different hardware technologies to accelerate the execution speed of the emulator, introduces novel and efficient optimization algorithm to improve quality of translated code etc. How to translate and optimize it fast has been the hot research point for a long time.

QEMU is a typical dynamic binary translation system, it uses dynamic binary translation technology to achieve real-time translation and execution on the target binary code, and have multi-sources and multi-targets, rapid translation, support self-referencing/self-modifying code and other characteristics. Compared to the fast translation, however, the host code generated by QEMU often exist many unnecessary memory access and register move operations, these will lead to significant execution overhead. Besides that, QEMU translates, optimizes and executes the object binary code in a serialized way, which leads many advanced optimization algorithms to be hardly used, because of these algorithms do optimization needs for time-consuming optimization operation, and optimization overhead often outweighs the benefits.

The paper studies the TCG translation engine in details, and put forwards, based on the research, a model of hotspot code detection and dynamic optimization: combination with multi-cores of processes and multi-threads technologies, implementation to detect the hotspot code of QEMU through code instrumentation and NET algorithm, and

merge them into a super block. When a super block is generated, the depth optimization will be processed for generating simplified host codes. The paper makes consolidation and optimization routines to be executed in different threads on different CPU cores by using multi-threads technology, which makes the core simulation thread will not be considered the overhead of the optimization algorithm. Threads can be executed paralleled at the same time in different processor cores makes the serialized execution of original QEMU converted into parallelized execution, it effectively improves the utilization of the processor, so as to the performance of QEMU. In addition, the paper proposes a novel optimization method called Delegate Mechanism, which can effectively eliminates the memory load operation code and a register move operation for enhancing the quality of the translated code, so as to achieve the purpose of code optimization. Experiment based on QEMU-ARM indicates that the model can effectively improve the QEMU average execution performance by about 10%.

Key words: Binary Translation, QEMU, Hotspot Detection, Delegate Mechanism

目 录

第一章 绪论 **1**
1.1 课题背景 1
1.2 研究现状 2
1.3 研究内容 3
1.4 论文组织 5
第二章 二进制翻译技术介绍 6
2.1 二进制翻译技术的结构 6
2.2 二进制翻译技术的分类 7
2.2.1 解释执行 7
2.2.2 静态二进制翻译 8
2.2.3 动态二进制翻译 9
2.3 二进制翻译技术需解决的关键问题 10
2.3.1 处理器体系架构相关问题 11
2.3.2 存储映射问题 11
2.3.3 代码挖掘问题 12
2.3.4 执行效率问题 12
2.3.5 实时性问题 12
2.3.6 运行环境的仿真问题 12
2.4 二进制翻译技术的应用 13
2.4.1 固定源和目标的二进制翻译系统 13
2.4.2 可变源和目标的二进制翻译系统 14
2.5 本章小结 15
第三章 QEMU 动态二进制翻译系统 16
3.1 QEMU 系统框架 16
3.1.1 仿真控制核心 17
3.1.2 TCG 核心翻译引擎 18
3.2 QEMU 仿真模式 19
3.2.1 用户级仿真 19
3.2.3 系统级仿真 20
3.3 QEMU 翻译单位 21
3.3.1 基本块 22
3.3.2 基本块管理 23
3.3.3 直接块链 24
3.4 QEMU 优化策略 25

3.4.1 TCG 中间码的优化 25
3.4.2 主机代码的优化 27
3.5 QEMU 执行流程 28
3.5.1 仿真环境初始化阶段 28
3.5.2 代码翻译和执行阶段 28
3.6 本章总结 29
第四章 QEMU 热点代码探测与动态优化模型设计 31
4.1 总体设计 31
4.2 热点代码块的探测 33
4.2.1 代码插桩设计 33
4.2.2 热路径算法设计 37
4.2.3 热路径缓存池设计 39
4.3 热点代码块的合并 40
4.3.1 中间码缓存区设计 40
4.3.2 中间码的合并 41
4.4 热点代码块的优化 43
4.4.1 块间优化 43
4.4.2 块内优化 44
4.4.3 热点块重定位 49
4.5 本章总结 50
第五章 性能评估 51
5.1 实验准备 51
5.2 DQEMU 性能评估 52
5.2.1 DQEMU 的主机寄存器映射 52
5.2.2 DQEMU 翻译后的主机代码生成量 53
5.2.3 DQEMU 的执行性能 54
5.2.4 DQEMU 结果分析 54
5.3 CQEMU 性能评估 55
5.3.1 SPEC CPU2006 测试套件部署 55
5.3.2 CQEMU 性能测试 56
5.3.3 CQEMU 结果分析 57
5.4 本章总结 57
第六章 总结与展望 59
6.1 工作总结 59
6.2 未来展望 60
参考文献 61
致谢 64
攻硕期间取得的研究成果 65

第一章 绪论

随着现代电子信息技术的发展，各种先进的处理器体系架构不断的问世，这些处理器除了具备更加快速的处理性能外，还具备一些自身独有的特性。然而，新型处理器的普及需要得到相应软件的支撑。开发新式的处理器可能会因得不到软件的支持而影响其推广应用以及市场前景，没有足够的场份额反过来又会导致得不到丰富的软件支撑。这种处理器架构和软件支持的相互制约关系，使得新式的处理器必须考虑向后兼容，更无法充分发挥新式处理器的先进特性。另一方面，不同的处理器体系架构具备不同的指令集以及自身的特性，这种处理器体系架构的差异性导致很难实现软件的移植。对于一些开源软件，必须针对特定目标处理器平台重新编译后，才能够转移到目标平台上运行。而对于一些不开源或只有可执行目标文件的软件来说，跨平台运行几乎无法实现。此外，跨平台编写软件常常需要对软件进行及时编译和调试，以快速发现问题。但是，软件本身无法在开发平台上直接运行和调试，需要装载在目标平台上进行，这种软件验证过程非常繁琐和耗时，极大地阻碍了软件的开发进度。

为了克服上述难题，二进制翻译技术[1,2]便应运而生。它能够自动将目标处理器体系架构的指令集动态地翻译成本地处理器体系架构的指令集，并在本地处理器上边翻译边执行。这种动态的翻译和执行特性使得它能够很好的实现不同处理器架构之间的可执行目标文件的相互移植，还能够充分利用现有处理器自身独有的特性，加速代码的执行效率，也能够很好地实现对跨平台软件的及时调试。

1.1 课题背景

二进制翻译是指采用软件技术将一种处理器体系架构的指令集翻译成功能对等的另一种处理器体系架构指令集，通过指令集翻译以及目标平台架构的仿真来实现软件跨平台运行的目的。大多数采用二进制翻译技术的系统都用于实现指令集仿真，仿真器通过虚拟化一个目标体系架构环境，采用二进制翻译技术来实现可执行目标文件的本地执行，运行在虚拟仿真环境的程序就好像运行在其真实的目标体系架构环境一样。因此，二进制翻译技术很好地消除了软硬件之间的兼容性问题，有效地推动了软硬件技术的更新。

二进制翻译技术本质上也属于编译技术。较之于传统的编译技术，其编译处理的对象不再是某种形式的高级语言，而是某种处理器体系架构的目标二进制镜像。该目标二进制镜像经过传统编译器产生，经由二进制翻译器处理后生成另一种处理器体系架构的可执行目标代码；而传统编译器则是将某一种形式的高级语言进行处理，生成某种特定处理器体系架构的可执行目标代码。

在现代处理器体系架构中，多核心和多处理机技术已发展得相当成熟，它能让多个控制进程（线程）在多个处理器核心中并行地执行，从而有效增强处理器并行处理事务的能力，以此改进整个硬件系统的处理性能。伴随着硬件技术的发展，多核心和多处理器技术不仅广泛地应用于高端服务器，还逐步进入了家用电脑领域。虽然二进制翻译系统能够较好地仿真多处理机和多核处理器，但是二进制翻译系统本身却采用单线程模式来实现虚拟环境的仿真以及动态代码翻译，即对目标代码的翻译、优化和执行通过串行化方式来进行。这种串行化工作模式将导致一个问题：如果采用各种优化技术对目标代码进行深度优化，虽然能够保证优质的代码翻译质量，但其优化过程所产生的开销可能会很大，从而增加整个二进制翻译系统的开销；如果对目标代码不进行深度优化，虽然不存在优化开销，但其产生的翻译代码质量将很难得到保证，在翻译代码中可能存在许多冗余的指令，这些冗余指令的执行将会增加翻译后的目标代码的执行开销，从而也会增加整个二进制翻译系统的开销。因此，如何让二进制翻译系统既能够快速翻译和执行，又能够产生优质的代码质量已成为当前二进制翻译技术的研究热点。

1.2 研究现状

自上世纪 80 年度起，二进制翻译技术就广泛应用以解决软件兼容以及软件移植问题。DEC 公司开发的 FX!32 仿真器[3]，它使得 IA-32 可执行目标文件[4]可在一个运行于 Windows 操作系统的 Alpha 处理器体系架构上透明地执行。昆士兰大学曾开发过一个开源的就静态二进制翻译系统 UQBT[5-7]，以及后续改进的动态二进制翻译系统 UQDBT[8,9]。UQBT 采用静态二进制翻译技术实现目标二进制代码的翻译执行，并具有多源多目标的特性。UQDBT 对前者做了相应改进，应用了动态二进制翻译技术来实现目标二进制代码的动态翻译和执行，并提供了很高的实时性，解决静态二进制翻译技术来带的自修改代码、无法实时获取运行时的动态信息等缺陷。Fabrice Bellard 开发的开源仿真器 QEMU[10]被广泛应用于仿真各种目标处理器架构，它提出了一种中间描述语言——TCG（Tiny Generator Coder），是一

种具有快速翻译、支持多源多目标等特性的动态二进制翻译系统。

目前 QEMU 衍生出了许多不同的版本，有些版本实现了对 QEMU 功能的扩充和定制，以满足特定的需求。例如，目前流行的 Android 模拟器，它基于 QEMU 翻译引擎，增加了对 ARM 处理器架构设备的模拟，并定制了皮肤和键盘映射。还有些版本采用结合主机架构自身的特性，实现了对 QEMU 的相关优化[11-13]。例如，PQEMU[14]对 QEMU 仿真多核心处理器架构的技术进行改进，采用多个实例来模拟仿真处理器核心；COQEMU[15]结合多核心和多线程技术来仿真多个目标处理器核心，多个线程在多个主机处理器核心上并行执行，以达到仿真多核环境真正的并行。

在国内，很多高校和研究所也对二进制翻译技术进行深入的研究，并取得丰硕的研究成果。由中国航空计算技术研究所自主研发的 BTASUP[16]二进制翻译系统，能够实现对 1750 处理器体系架构的可执行镜像在 PowerPC 处理器体系架构上透明地执行。由中国科学院自主研发的 Digital-Bridge[17]动态二进制翻译系统，可实现在 MIPS 处理器体系架构上运行 x86 体系架构的可执行镜像。由清华大学陈渝博士主研的 Skyeye[18]模拟器，实现了多种嵌入式开发板的硬件模拟，让各种嵌入式软件可直接在主机平台上进行及时调试和验证。较新版本的 Skyeye 模拟器还增加了 LLVM[19]优化引擎模块，使得其具备优质的代码质量，并具备较高的执行性能。

1.3 研究内容

QEMU 是一款非常流行的多源多目标二进制翻译系统，通过采用动态二进制翻译技术，实现对目标体系架构的二进制代码进行动态的翻译和执行。它可将各种目标体系架构(x86，ARM，SPARC 和 PowerPC)的二进制目标代码通过 TCG 技术动态翻译成主机体系架构（x86，ARM，SPARC，PowerPC 和 MIPS）的二进制目标代码，具有多源和多目标特性。此外，QEMU 还有效地解决了动态二进制翻译技术中自引用代码、自修改代码和精确异常等问题。

TCG 技术是 QEMU 二进制翻译系统的核心，它可将目标体系架构的二进制代码反汇编成等价的 TCG 中间码，然后再将 TCG 中间码转换成等价的主机体系架构的代码。由于 QEMU 不对代码做任何优化，因此其翻译过程非常迅速。另外，QEMU 还提供了一个缓存区用于缓存翻译后的代码块，并采用直接块链技术实现代码块内部的直接跳转，降低代码块与主循环的切换次数，能够有效地增强 QEMU

的执行性能。

然而，较之于快速翻译，QEMU 翻译的代码质量不高。由于 QEMU 在翻译过程中不对翻译代码进行优化，导致在翻译后的代码中存在许多冗余指令，特别是有许多内存存取操作，每次执行翻译代码时都需要执行这些冗余指令，这将极大地增加 QEMU 的执行开销。由于 QEMU 采用单线程的串行工作模式，如果在 QEMU 翻译过程中增加各种优化机制来改进翻译代码的质量，势必会增加翻译开销，这种翻译开销将抵消由执行优化后翻译代码时所提升的执行性能，因此不能有效地改进 QEMU 的整体性能。

如果能在不影响 QEMU 正常翻译执行的情况下，又能够对其翻译的代码进行深度优化，这将能够有效改进 QEMU 的性能。多核心技术和多线程技术使之成为可能：将 QEMU 由单线程工作模式转换为多线程工作模式，一个线程（核心线程）负责整体的翻译和执行，其它线程（优化线程）负责对翻译代码进行深度优化，各个线程间在多个处理器核心上并行地执行，当优化线程对翻译代码进行优化后，转而执行优化后的代码。通过这种协同工作方式，既能够保证 QEMU 执行主体不被打扰，又能够有效地改进 QEMU 的执行性能。

另外，QEMU 按照基本块来执行代码，各基本块之间相互独立。为了保证代码的正确性，每个基本块在执行时必须先从内存读取仿真环境数据，在执行结束时还需要将其执行结果写回目标仿真环境，这种仿真环境的装载和存储将造成极大地执行开销。如果能将多个基本块合并成一个较大的超级块，既能够减少仿真环境的装载和存储次数，又能够对代码块进行更多的优化，从而进一步提升 QEMU 的执行性能。

因此，针对 QEMU 存在的问题，文章设计和实现了一种热点代码探测和动态优化模型：

（1）利用代码插桩技术，在 QEMU 动态翻译过程中插入热点收集代码以探测出热点代码块。通过代码插桩，QEMU 翻译过的每个基本块将包含插桩代码，当基本块被执行时，热点探测代码也将被执行。当基本块的执行次数超过规定阈值时，热点插桩代码将调用热点块探测例程实现热点代码块的搜集。

（2）结合 QEMU 翻译引擎自身的特性，提出一种热点块探测算法。包括定位热点块首部、热点阈值的确定、热点块缓冲区的管理等。由于热点块探测算法的实现代码将通过代码插桩技术插入每个基本块中且频繁执行，因此热点块探测算法的好坏将直接影响 QEMU 的执行性能。设计一个代码量少且命中率高的热点块探测算法将变得非常关键。

（3）运用相关技术实现对探测到的热点代码块进行合并，以产生一个超级块。超级块中包含所有参与合并的基本块所蕴含的功能，并具有一个入口多个出口。热点块的合并采用单独的线程实现，并充分利用处理器多核心的特点，实现 QEMU 核心仿真线程与合并例程在不同的处理器核心中并行执行。

（4）提出一种新颖的优化技术——委托机制，实现对合并后的超级块代码进行深度优化，消除代码中存在的冗余指令，提高翻译后代码的质量，以此提升仿真器的代码执行性能。超级块的优化采用单独的线程实现，在不妨碍 QEMU 主体线程正常执行的前提下，对超级块中的代码进行分析和优化。

通过上述实现，文章旨在运用多核心技术和多线程技术，在 QEMU 翻译执行时进行热点代码探测，并对热点块进行合并优化，改进 QEMU 翻译的代码质量，进一步提升 QEMU 的执行性能。

1.4 论文组织

第一章，文章的绪论，概述二进制翻译技术的起源和工作原理，国内外二进制翻译技术的研究现状和应用情况，以及课题的研究内容和研究意义。

第二章，介绍二进制翻译技术的框架结构，详细阐述解释执行、静态二进制翻译、动态二进制翻译三种不同翻译策略的实现原理和应用情况以及设计二进制翻译系统时需要考虑的相关问题。

第三章，介绍 QEMU 二进制翻译系统的系统框架组成、两种仿真模式、QEMU 的翻译单位、不同阶段的优化策略及其主要执行流程。

第四章，阐述 QEMU 的热点代码探测与动态优化模型的设计与实现。包括：代码插桩、热路径探测算法、热点块合并、超级块的优化与重定位等。

第五章，对实现的多线程热点代码探测与动态优化模型进行实验验证，以评估其对 QEMU 的性能改进。实验测试包含改进型 QEMU 与原始 QEMU 的执行性能对比，以体现改进型 QEMU 性能的提升。

第六章，对本文工作进行归纳总结，并提出后续的改进方向。

第二章 二进制翻译技术介绍

二进制翻译是不同处理器体系架构间代码移植和硬件仿真的重要手段，在硬件虚拟化以及软件移植领域得到了广泛地应用。自上世纪 80 年代起，二进制翻译技术获得了许多举世瞩目的研究成果，并相继地研发了许多具有重要研究性质以及商业性质的系统。

2.1 二进制翻译技术的结构

二进制翻译技术的翻译过程和传统编译技术[20]有点类似，都需要经过前端分析、中端优化和后端代码生成三个阶段。按照二进制翻译技术在不同阶段所做地不同任务，可将其划分为三个部分：前端解码器、中端优化分析器以及后端翻译器。图 2-1 描述了二进制翻译器的组成框架以及指令生成流。

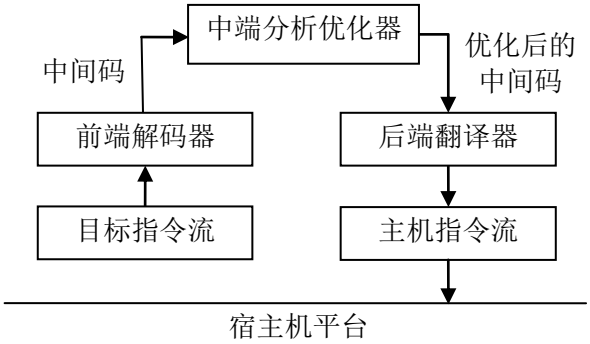


图 2-1 二进制翻译器组成框架

前端解码器根据目标体系架构指令结构的特征，以及可执行目标文件的格式（例如 x86 采用 PE Format，Linux 采用 ELF Format），通过对指令模式配对进行二进制指令的编码格式分析，相当于完成二进制指令的反汇编操作。前端解码器对二进制指令解码后通常生成中间描述语言（IDL），称之为中间码。中间码一般采用精炼、功能简单的描述字来表示目标指令的功能。对于一条复杂的目标指令，前端解码器可能会产生多条中间码来表述目标指令的实际操作。

前端解码器要求对每条目标指令按照其格式精准地解码，并处理过程调用/程序跳转、自修改/自引用、指令的操作码/操作数的识别和分析等。该阶段的解码功能与传统编译技术的前端操作相类似，都是负责对处理目标进行分析解码操作，

但两者的解码对象不同，因此两者的解码技术以及关键难点也不相同。

中端分析优化器负责对前端产生的中间码进行优化分析，以消除代码中目标体系架构的一些特性，实现到主机体系架构的转化。由于不同体系架构之间的差异性，经前端解码器得到的中间码一般含有目标体系架构的一些特性，这些特性不利于实现到主机体系架构的翻译。

因此，中端分析优化器需要屏蔽这种不同体系架构间带来的差异性，从而为后端翻译器的翻译工作提供支持。屏蔽体系架构的差异性是二进制翻译可重定位能力的核心技术，它可将多种目标体系架构的指令集翻译成与相同的机器无关的中间表示，这种方式使得增加仿真目标体系架构只需实现前端解码，后端翻译器可不做任何改变便可实现一个适用于新目标体系架构的二进制翻译系统。此外，中端分析优化器还负责对中间码进行相关优化工作，例如：指令生存期分析[21]、线性扫描[22]、无效指令移除[23-25]等。

后端翻译器与传统编译器相类似，用于实现中间码到主机体系架构机器码的转化。它依照目标体系架构以及操作系统平台的特点，按照规定的格式将中间码翻译成功能等同的、可在主机体系架构上直接执行的二进制指令序列。同时，后端翻译器也采用相关代码优化策略，消除冗余的二进制码，以期产生精简的主机指令序列。

2.2 二进制翻译技术的分类

二进制翻译技术根据翻译策略以及翻译时机[26]的不同，可将其分为解释执行、静态二进制翻译以及动态二进制翻译。

2.2.1 解释执行

解释执行针对目标体系架构的每条指令按照其功能进行解释执行，在整个翻译过程中，系统不会缓存已翻译过的代码。因此，当下一次遇到相同指令时，仍然按照前面的过程进行重复翻译。图 2-2 给出了采用解释执行的二进制翻译器的指令解析过程。在解释执行过程中，各条指令之间相互独立，因而无法实现对目标指令集的优化，在对每条指令进行翻译后，都需要更新仿真目标环境的状态。解释执行的开发过程较简单，很容易实现与老的体系架构兼容，但其翻译过程效率极其低下，仿真环境的响应延迟让用户难以承受[26]。目前采用解释执行的仿真器有 IBM/360 Emulator Of The IBM 1401 以及 VAX Emulation Of The PDP-11。

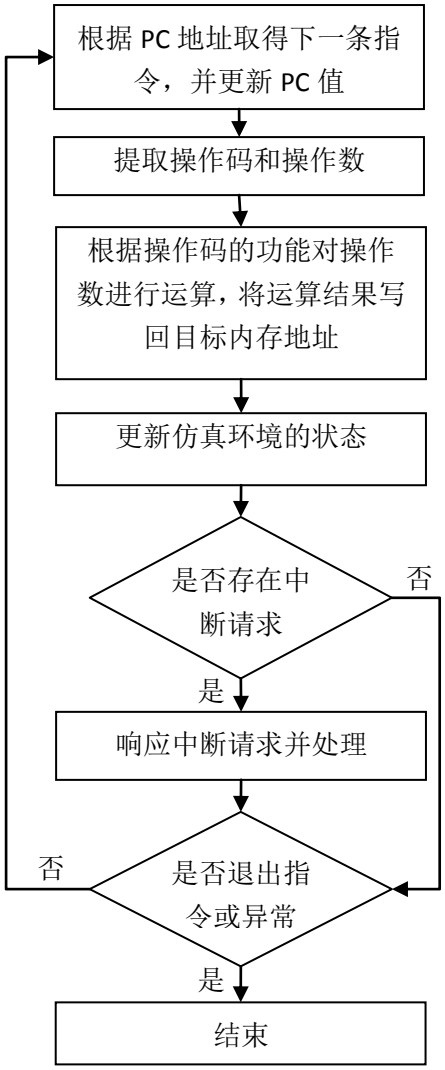


图 2-2 解释执行的指令解析过程

2.2.2 静态二进制翻译

静态二进制翻译[27]采用了完全不同的技术来实现目标二进制代码翻译，它在可执行目标镜像运行之前，采用离线翻译的方式对其整个可执行镜像全部翻译成主机体系架构上的可执行二进制指令序列，并将翻译后的代码缓存起来。这样使得一次翻译得结果可多次使用，如果后续的代码执行相同的指令，则无需再次重新进行翻译。

图 2-3 给出了静态二进制翻译系统的基本结构，其详细描述了目标体系架构的可执行目标镜像静态翻译到主机体系架构上的翻译流程。这种翻译机制能有效地改善了翻译代码可重用性问题，提升仿真器的性能。但是，静态二进制翻译也存

在其固有的缺点：它不能有效获取可执行镜像运行时的动态信息，例如当前寄存器的内容，设置断点等，因而不便于进行动态调试；如果在可执行镜像中存在自修改代码，将会导致代码无法识别以致翻译器崩溃；另外，静态二进制翻译对用户和当前的软硬件环境有较强的依赖，无法满足实时性的需求。目前采用静态二进制翻译技术的仿真器有 FX!32 和 UQBT。随着翻译技术的发展，静态二进制翻译技术已逐步被动态翻译二进制技术取代。

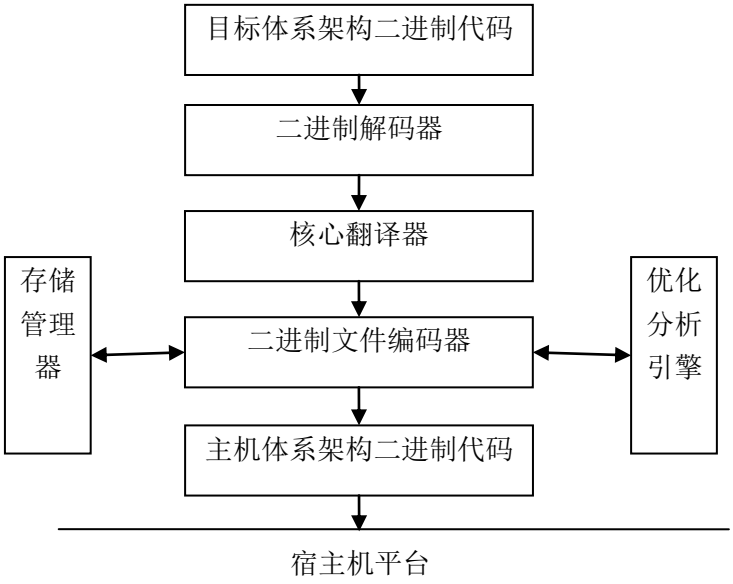


图 2-3 静态二进制翻译系统的基本结构

2.2.3 动态二进制翻译

动态二进制翻译技术[28-30]汲取了解释执行和静态二进制翻译技术的特点，它采用边翻译边执行的方式来实现二进制目标代码的执行。当二进制目标代码首次被执行时，以基本块（Basic Block）为单位进行翻译，一个基本块包含以控制转移（例如分支、直接跳转或过程调用）结束的多条目标指令序列。

图 2-4 给出了动态二进制翻译系统的动态翻译流程。在执行一条指令时，首先查看以该指令地址为起始地址的基本块是否存在。如果存在，则直接进入该基本块，执行该基本块的指令序列；如果不存在，则从该条指令开始翻译一块目标二进制代码，翻译后转入到该基本块内执行指令序列，同时缓存已翻译的代码块，并将该基本块放入缓存队列以便下次查找。

动态二进制翻译既能够提供良好的实时性和执行性能，又能够很好地解决自修改代码问题，且对用户透明。这使得动态二进制翻译技术得到广泛的应用，目

前常用的仿真器如 QEMU[10]、JVMs[31]、BOA[32,33]等都采用了动态二进制翻译技术。由于动态二进制翻译器需要缓存翻译后的代码，因此需要占用较大的内存空间；另外，它以基本块为单位来执行代码，在块内代码执行过程中，无法响应中断，只有当一块代码执行完毕后返回主循环才能够处理中断。因此，中断响应具有一定的延迟，对于一些实时性要求较高的场合其提供的实时性能仍不能完全满足。

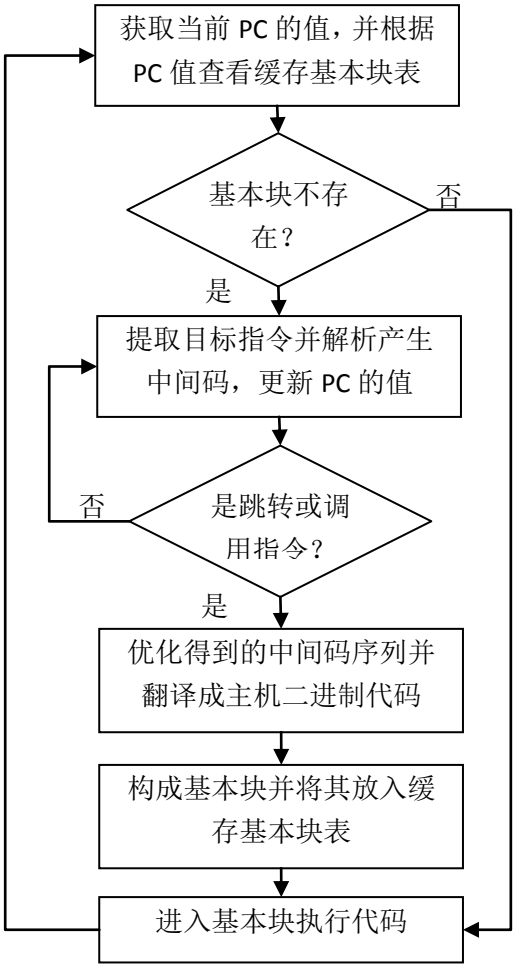


图 2-4 动态二进制翻译系统的动态翻译流程

二进制翻译技术经历了由解释执行到静态二进制翻译，再到动态二进制翻译的发展历程，每一阶段都在前一阶段的基础上进行了改进和优化，以满足应用不断增强的需求。

2.3 二进制翻译技术需解决的关键问题

虽然各种处理器体系架构间存在较大的差异，例如：指令集不同、寄存器数

量和作用不同以及各种架构独有的特征，但是它们都基于冯·诺依曼结构进行设计，因此不同处理器架构上所做的任何计算都能进行相互转化。设计有效的二进制翻译系统本身是一项非常艰巨的挑战，它不仅仅是仿真目标体系架构，还希望在仿真的同时，充分考虑本地处理器平台的特征，提供良好的执行效率。以下给出了二进制翻译技术涉及的几个关键问题，有效的解决这些问题将能够很好的提升二进制翻译系统的效率，以满足实时性的需求。

2.3.1 处理器体系架构相关问题

不同的处理器体系架构具有不同的指令特性，其指令处理方式的实现也各有不同。经典的处理器指令解析模式如 RISC 和 CISC，两者采用完全不同的实现方式来解析指令。某些处理器体系架构中包含的原子操作指令，在其它处理器体系架构上实现其语义也相当困难。

不可中断性指令是造成仿真处理器体系架构的另一难题，特别是主机体系架构上没有相类似的指令时，其几乎无法精确实现。较之不可中断性，更棘手的问题是如何处理目标体系架构中的精确异常问题。如果主机处理器体系架构提供指令重排序执行，而在翻译执行过程中出现了异常，这将导致很难定位异常点。解释执行对每条指令解释执行，因此可精确定位异常点。而静态二进制翻译在执行之前对目标二进制代码完全翻译后在执行，当异常发生时，由于其执行不依赖其目标二进制码，当异常发生时，根本无法定位异常点，动态二进制翻译中按基本块来执行代码，能够定位异常发生时所处在的代码块，可采用相关技术来定位异常点，但实现较复杂。

2.3.2 存储映射问题

在仿真目标体系架构时，需要保留其目标寄存器的内容，这可以采用目标寄存器到主机寄存器的映射实现。但由于体系架构之间的差异性，主机平台寄存器数量可能少于目标寄存，这使得目标寄存器的内容只能存放于主存中。而对主存的存取周期比较长，这将是翻译执行效率低下。某些体系架构还存在 I/O 端口，对 I/O 端口的仿真也将对翻译执行造成困难[34]。

另外，各种体系架构之间的数据表示方式也存在差异。如 x86 平台采用小尾端的方式存储数据，而 ARM 平台采用大尾端的方式存储数据，MIPS 平台还同时支持两种存储方式。在对目标二进制指令进行翻译时，就需要考虑数据的表示方

式，进行大/小尾端的转换。

2.3.3 代码挖掘问题

代码挖掘问题涉及两个关键问题：自引用问题 (self-reference)和自修改问题[35]（self-modifying）。自引用是指代码引用自身进行后续操作，例如求校验和。自修改问题是指代码对其自身进行修改。解决这两个问题需要能够发现代码正在引用/修改自身，解释执行能够很好的解决这两个问题，但对纯静态二进制翻译器来说不可能实现，动态二进制翻译器也需要借助一定的技巧[36]来监视二进制目标代码。

2.3.4 执行效率问题

二进制翻译器的执行效率是评价其优劣的标准之一，提供快速的翻译和执行能够提供较高的实时性。很多二进制翻译器具备强大的功能，但其执行效率却不尽如人意。如何改进其执行效率也是目前研究热点，通常从两方面着手：一是通过优化分析，提高翻译生成的目标二进制代码的质量，以降低其执行开销；而是改进翻译器本身，降低翻译自身的开销，以实现快速的翻译。此外，提高代码的可重用性也能很好的改善二进制翻译器的性能。由于代码缓存空间的限制，不能实现对翻译的目标代码的全部缓存，因此，需要借助一定的算法来进行缓存管理，以提高翻译后代码的命中率。

2.3.5 实时性问题

某些系统平台的二进制代码在执行时有严格的时间限制，而二进制翻译进程的快慢取决于执行的代码是否已被翻译过，以及当前主机平台的处理器性能。因此，对于实时性的二进制码的翻译需要一种技术来提前得知，以便在其正式执行之前进行翻译[37]，满足实时性的需求。

2.3.6 运行环境的仿真问题

如果目标二进制镜像运行的操作系统与当前系统平台相同，只是运行的体系架构不同，那么可以直接调用本地的操作系统 API 来完成目标镜像的请求。由于目标镜像的应用二进制接口与本地主机平台存在较大差异，在调用本地系统服务之前需要进行一定的格式转换处理。如果两者的系统平台也不相同，那么它们的系统调用将无法对应，翻译过程将更加复杂。此外，对于系统级仿真的二进制翻

译器，还需要妥善地处理处理器的特权指令。

2.4 二进制翻译技术的应用

经过二十多年的发展，二进制翻译技术得到了广泛的应用，并产生了许多具有商业价值的二进制翻译系统。按照仿真目标体系架构和运行环境的可重定向特征，可将其分成两大类：第一类为固定源和目标的二进制翻译系统，其固化了仿真目标和运行环境，不可更改；第二类是可变源和目标的二进制翻译系统，它能支持多种源体系架构的二进制代码，并能够在多种主机平台上运行。本节将通过这两类二进制翻译系统来展开讨论，阐述二进制翻译技术的具体应用情况。

2.4.1 固定源和目标的二进制翻译系统

较早的二进制翻译技术的实现都基于固定源和目标，它通过源和目标一一对应是方式实现源体系架构二进制指令的翻译和执行。如 DEC 公司研发的 FX!32 系统[3]、HP 公司研发的 Aries 系统[38]以及 Intel 公司研发的 IA32 EL 系统[4]。

FX!32 系统是一个基于剖析信息指导的二进制翻译系统，它能够在 x86 体系架构上 Windows NT 操作系统平台运行 Alpha 体系架构的二进制代码。FX!32 结合静态翻译和动态翻译技术，具有高效、精确且透明执行的特点。

当 FX!32 首次执行时，采用解释执行的方式来翻译和执行代码，同时保存执行路径等信息，另一个后台进程根据保存的路径信息进行静态翻译和优化。当下次执行时，就是用已翻译后的本地代码。代码的重复执行将不断加快其执行速度。FX!32 系统能正确翻译执行 Windows NT 操作系统平台的实用程序，如 Excel、PowerPoint、Word 等。由于 FX!32 首次执行采用解释执行方式，刚开始执行时速度会很慢，且只能通过静态分析来优化代码，无法利用动态优化。

Aries 系统是由 HP 公司于 1999 年研发的一款二进制翻译系统，它结合解释执行和动态翻译两种方式来实现代码的翻译和执行。Aries 系统仿真实现了 PA-RISC 具备的所有指令及其硬件特性，且在翻译过程中不需要人为监测。

较之于 FX!32 系统，Aries 只对频繁执行的指令块进行动态二进制翻译，系统运行结束后撤销全部翻译代码，而不必改变原始的目标程序。固动态的翻译策略具备快速翻译的特性，也不会违例原始目标程序的完整性。这使得 Aries 系统能够在惠普 HX 系统平台的安腾 64 位处理器体系架构上高效地执行 PA-RISC 二进制目标程序。

IA32 EL 是 Intel 公司在 2003 年研发的一款二进制翻译系统，其目的是通过软件模拟实现在最新的安腾处理器上执行 32 位 Intel 处理器的可执行程序，以兼容老式的处理器上的程序，减少硬件设计带来复杂度问题。IA-32 EL 系统主要面向用户编写的可执行程序，代码的翻译和执行分为两个阶段：第一阶段对代码不做深度的剖析，以确保较低的翻译开销，并为下一阶段的翻译搜集热点信息。第二阶段对热点块进行动态翻译并做深度优化。通过验证，IA32 EL 执行 IA32 程序的效率比 IA32 程序直接在主机上执行的执行效率更高。

2.4.2 可变源和目标的二进制翻译系统

随着二进制翻译技术的不断发展，二进制翻译迈入了新的研究阶段。由于固定源和目标的二进制翻译系统只能在特定平台上仿真固定的目标平台，这一缺陷使得其无法得到广泛的应用，特别对该类系统的移植，需要重新设计源和目标的对应关系。因此，可变源和目标的二进制翻译系统应运而生，它采用 CPU 描述字来表征各种处理器的特性，并自动化构造出指令解析器、代码产生器以及翻译引擎，从而降低二进制翻译系统的可移植性问题。具有代表性的系统有昆士兰大学研发的 UQBT 和 UQDBT 系统[7-9]，奥地利维也纳技术大学研究开发的 Bintran 系统[39]，以及 Transitive 公司研制的 Dynamite 系统[40]。

UQBT 和 UQDBT 是昆士兰大学先后研发的多源和目标二进制翻译系统，前者采用静态二进制翻译技术，后者引入了动态二进制翻译技术。UQBT 框架可依照二进制镜像的格式自动产生文件解码器、指令解码器、以及语义抽象转换器。这种机制可将与机器无关的部分抽离出来，为二进制翻译后端提供一种统一的中间表示。如果需要对翻译系统进行移植，只需编写解码器后端而不需要修改解码前端，有效的改善了编码效率，提高了系统的可重用性。

另外，UQBT 采用了两种中间格式来实现目标二进制代码的翻译：寄存器传输列表（RTLs）以及高层寄存器传输语言（HRTL）。它首先对指令进行格式和语义分析，以产生 RTLs 格式，然后再经过语义抽象到 HRTL 表示，最后再将 HRTL 表示转化成本地主机架构的二进制代码。UQDBT 系统采用了动态二进制翻译技术来实现目标二进制代码的动态翻译和执行，与 UQBT 相比，它能够很好的解决静态二进制翻译技术中存在的自引用、自修改代码问题。

Bintran 系统是一款机器自适应的动态二进制翻译系统，其主要通过 CPU 描述字实现复杂指令集、精简指令集和超长指令字三种处理器架构之间的转换。Bintran

的核心模块是代码生成器，它可根据目标处理器架构和本地主机架构的特性来产生一个翻译单元，并结合调度器实现目标代码的翻译及代码生成。此外，Bintran 还包含一些与体系架构相关的部分，如进行翻译引擎和调度器间切换的汇编码、解决变长指令和寄存器映射等情况的特殊代码。和 UQDBT 不同的是，Bintran 只采用一种中间表示语言来描述源体系架构二进制代码的语义，同时支持 PowerPC、x86 到 Alpha 体系架构的翻译。

Dynamite 系统是由英国曼彻斯特大学提出并研发的一款动态二进制翻译系统，它能够实现本地二进制目标代码和非本地二进制代码透明地执行。Dynamite 系统由三个主要模块构成：指令解码模块、优化分析模块以及代码生成模块。该系统是动态二进制翻译技术的标准实现方案，通过前端解码器将目标二进制代码翻译成功能等价的中间码，当一块中间码翻译结束后，通过中端优化分析器对其进行优化分析，然后再将优化后的中间码翻译成功能等价的本地主机码。中端优化器基于中间码进行分析，负责实现僵死代码删除、代码重排、分支预测等，使得能够产生较高质量的主机码，因此使系统具备较高的翻译执行性能。

2.5 本章小结

本章通过详述二进制翻译技术的结构、三种不同翻译策略的实现方式、需要解决的关键问题以及二进制翻译技术的应用。理解二进制翻译技术的基本结构，是理解和设计二进制翻译系统的基础；三种二进制翻译策略突出了二进制翻译技术的发展历程；解决二进制翻译技术的关键问题是实现快速二进制翻译的重要目标；最后还详细阐述了两类不同的二进制翻译系统，并给出两类系统的实际应用，展现二进制翻译技术的应用价值。本章通过对二进制翻译技术的详细讨论，为后续章节理解 QEMU 二进制翻译系统的原理和实现做出重要铺垫。

第三章 QEMU 动态二进制翻译系统

QEMU 是一款开源的动态二进制翻译系统，它提供的虚拟仿真环境能够运行未经修改的目标操作系统（例如 Windows 或 Linux）及其系统平台上的应用程序。QEMU 自身也能够运行于各种主机操作系统平台，例如 Linux、Windows 以及 Mac OS。同时，QEMU 还具备多源多目标特性，它可在多种主机处理器架构（x86，PowerPC，ARM 和 SPARC）上仿真多种目标处理器架构(x86，PowerPC，ARM，SPARC，Alpha 和 MIPS)。QEMU 的主要应用于硬件仿真、系统级调试以及硬件逻辑验证。硬件仿真通过建立一个虚拟硬件环境，目标操作系统运行在仿真环境中就好像运行在真实的物理环境一样，它使得软件跨平台运行成为可能，是软件移植和实现虚拟系统的重要手段。系统级调试是系统软件开发的难点，在真实的物理环境中运行的系统软件无法设置断点、探测 CPU 状态以及异常捕获。然而，QEMU 能够方便地实现单步调试，获取、保存和复位 CPU 状态，克服了系统级调试存在的难题。另外，QEMU 通过增加新的机器描述字和新的仿真设备，能够快速的验证新设备的执行逻辑和硬件兼容性。

3.1 QEMU 系统框架

QEMU 系统主要由仿真控制核心和 TCG 核心翻译引擎两部分组成，其系统基本框架如图 3-1 所示。

仿真控制核心部分负责调度整个系统的运行，确保系统的正常运转。在系统运行期间，仿真控制核心需要维护仿真目标环境 CPU 的状态、响应硬件中断并处理、维护仿真的硬件设备、加载目标可执行文件以及管理代码缓存空间。

TCG 核心翻译引擎负责对目标二进制代码进行动态地翻译。当仿真控制核心发现当前需要执行的目标二进制代码还未翻译，将翻译请求提交至 TCG 翻译引擎。TCG 根据提交的仿真控制核心目标代码地址代码进行翻译，翻译过程以基本块为单位，遇到跳转指令或代码跨页结束。当本次翻译结束后，TCG 将翻译好的代码存放至主机代码缓存区，仿真控制核心将跳转到主机代码缓存区当前基本块代码对应的起始地址以继续执行。

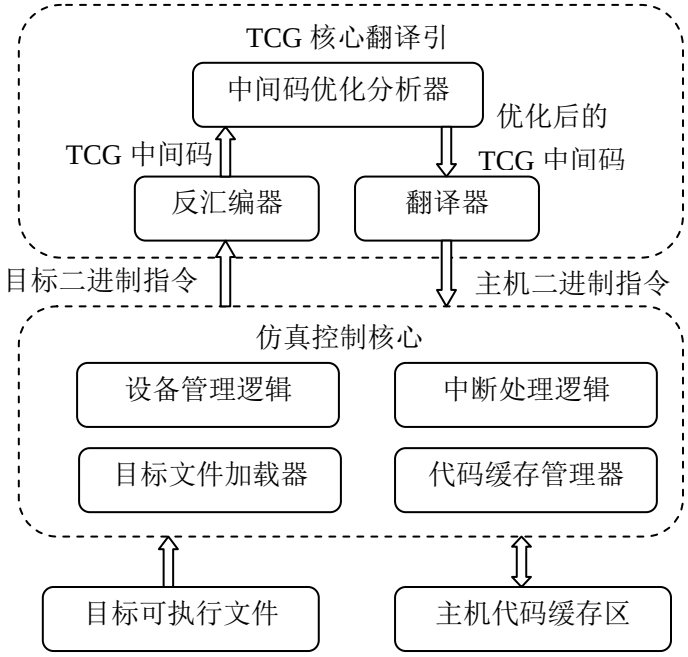


图 3-1 QEMU 系统基本框架

3.1.1 仿真控制核心

仿真控制核心负责协调整个系统的运行，包括：设备管理逻辑、中断处理逻辑、目标文件加载器和代码缓存管理器四个部分。

设备管理逻辑负责管理仿真的外部设备，例如 SD Card、NandFlash 以及 NorFlash 等，外部设备的读写请求、设备控制逻辑以及中断请求都由仿真控制核心实现。当执行逻辑由基本块代码返回到核心主循环时，仿真控制核心将对每个设备的状态进行更新，并处理设备的读写请求。

中断处理逻辑负责响应硬件发出的中断。当基本块代码执行结束时，仿真控制核心将检查目标仿真环境的中断位，如果被置位，则响应硬件发出的中断。

目标文件加载器用于加载目标可执行文件，加载时分析目标文件的格式（Windows 采用 WinPE，Linux 采用 ELF），并按照指定的格式解析出文件的代码段，数据段，BSS 段等。代码缓存管理器负责管理主机代码缓存区。

主机代码缓存区用于存放翻译后的主机代码，由于指令执行存在着字对齐限制，在每次进行代码翻译之前，代码缓存管理需要对当前产生代码的缓存区起始地址进行字对齐，以保证指令能够顺利执行；当缓存区满时，代码缓存管理器还需要清空缓存区以便容纳新翻译的主机代码块。

此外，仿真控制核心还维护了一个目标处理器的状态，并采用 CPUState 结构

体描述，它包含目标处理器的通用寄存器、中断寄存器、段寄存器以及各种标志寄存器等。系统的整个运行中都需要该结构体的参与，并根据该结构体中寄存器的内容来决定下一步的执行动作。

3.1.2 TCG 核心翻译引擎

TCG 核心翻译引擎是动态二进制翻译技术的典型应用，它将目标 CPU 架构的二进制代码转换成主机 CPU 架构的二进制代码。例如，TCG 能将 ARM 体系架构的二进制代码转换成在 x86 体系架构上本地可执行的二进制代码。

TCG 由反汇编器、中间码分析优化器以及翻译器三部分组成。反汇编器接收目标二进制指令，对目标二进制指令进行解码，按照目标架构平台的指令格式取出操作码和操作数，并转换成功能等价的中间码序列；中间码分析优化器对反汇编器产生的中间码进行分析优化，去除中间码中存在的冗余指令；翻译器对优化后的中间码与主机指令集进行一一映射，分配主机寄存器，并生成对应的主机二进制指令序列。

TCG 翻译过程可分为两阶段：首先将目标架构的二进制代码转换为功能等价的中间码，然后将中间码转换成本地可执行二进制代码。图 3-2 给出了 TCG 二进制代码翻译流程。

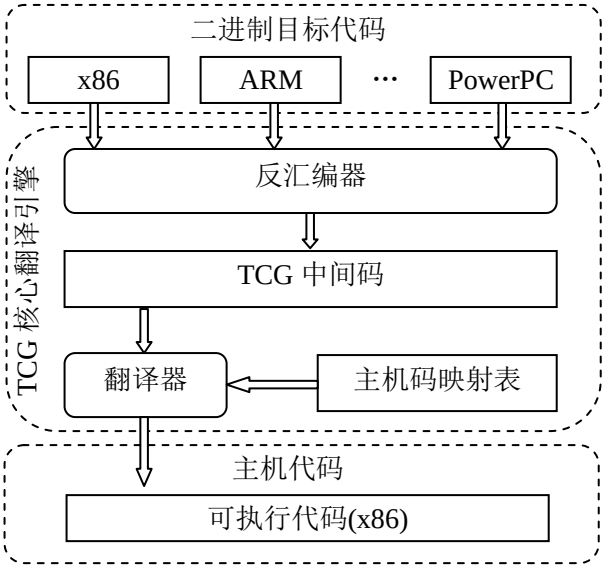


图 3-2 TCG 二进制目标代码的翻译流程

第一阶段，TCG 将目标二进制代码转换成等价的中间码序列，转换过程以基本块为单位。TCG 以仿真目标 CPU 的当前程序计数器 PC 的值为参考，从物理内存加载目标二进制指令进行转换。TCG 将反汇编以 PC 值指向的所有后续目标指

令序列，直到遇到控制流转移指令（如 call，jump，return 等指令）结束。针对每条目标指令，反汇编器可能会产生多条中间码序列，由多条中间码序列功能描述目标指令的功能。如图 3-3 所示，一条 ARM 加法指令经反汇编后产生了四条中间码序列：前两条分别将目标寄存器 r2 和 r3 的值加载到临时寄存器 tmp8 和 tmp9 中；第三条对 tmp8 和 tmp9 进行加法操作，相加结果放入临时寄存器 tmp10 中，第四条将 tmp10 的值写回至目标寄存器 r1。

第二阶段，TCG 将中间码序列翻译成本地可执行的主机指令序列。TCG 维护了一个中间码映射表，映射表中包含中间码与主机体系架构二进制指令序列的映射关系。这些二进制执行序列在本地处理器上执行，以完成中间码描述的功能。TCG 通过映射表将一个基本块中的所有中间码映射成功能等价的主机指令序列，执行产生的主机代码块等价于目标代码执行的功能。图 3-3 给出了一条 ARM 加法指令经过 TCG 翻译后产生的 x86 主机指令序列。

ARM 目标码	
add r1, r2, r3	*# r1 <- r2 + r3*
TCG 中间码	
mov_i32 tmp8, r1	*# tmp8 <- r2*
mov_i32 tmp9, r2	*# tmp9 <- r3*
add_i32 tmp8, tmp8, tmp9	*# tmp8 <- tmp8 + tmp9*
mov_i32 r1, tmp8	*# r1 <- tmp8*
x86 主机码	
mov 0x8(%ebp), %ebx	*# %ebx <- (r2)*
mov 0xc(%ebp), %esi	*# %esi <- (r3)*
add %esi, %ebx	*# %ebx <- %ebx + %esi*
mov %ebx, 0x4(%ebp)	*# (r1) <- %ebx*

图 3-3 ARM 目标码对应的 TCG 中间码以及 x86 主机码

3.2 QEMU 仿真模式

QEMU 是一个多功能的可移植动态二进制翻译系统，按照应用层次的不同，可分为用户级仿真和系统级仿真。

3.2.1 用户级仿真

用户级仿真为相同操作系统不同处理器架构的应用程序提供一个虚拟运行环

境， QEMU 只对应用程序能够感知的部分进行仿真，包括：逻辑内存空间布局、用户级指令集以及目标 CPU 寄存器。图 3-4 给出了 QEMU 用户级仿真的系统结构。

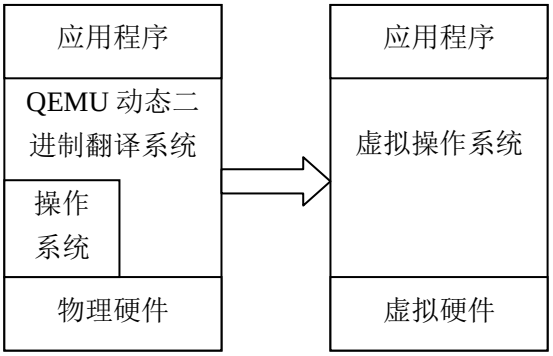


图 3-4 QEMU 用户级仿真的系统结构

在用户级仿真模式下，应用程序的目标操作系统平台与主机操作系统平台必须一致，应用程序的系统调用可直接转换为主机操作系统的系统调用进行操作。此外，QEMU 随应用程序运行而创建，翻译并运行应用程序的指令集，当应用程序执行结束时，QEMU 也随之退出。由于不同处理器架构操作系统平台之间的差异性，主机操作系统平台使用的共享库（DLL）可能与目标操作系统使用的共享库存在差别，因此，仿真的应用程序时需要提供相应的共享库，以便在运行时能够正确的处理函数库之间的调用关系。

3.2.3 系统级仿真

系统级仿真为运行目标操作系统而提供了一个完整的目标架构虚拟运行环境，包括：虚拟的硬件资源、网络接口以及图形化窗口。图 3-5 给出了 QEMU 系统级仿真的系统结构。

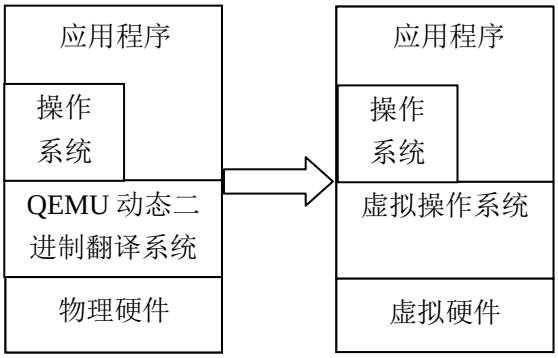


图 3-5 QEMU 系统级仿真的系统结构

在系统级仿真模式下，QEMU 不仅要进行目标指令集翻译，还需要管理目标

架构仿真的各种设备，响应设备发出的中断等。因此，系统级仿真提供了健全的虚拟运行环境，可运行不经任何修改的目标操作系统，目标操作系统运行在虚拟环境中就好像运行在真实的物理平台一样。系统级仿真需要分离出非特权指令和特权指令。对于非特权指令，可直接进行翻译生成功能对等的主机指令，翻译过程不需要进行额外检查。而特权指令需要经过特殊处理，这是因为 QEMU 运行在主机操作系统的用户模式下，不具备特权指令的执行权限，特权指令只能够陷入内核才能运行。因此，QEMU 对特权指令采用主机平台上一些特定的指令序列来实现，这些指令序列通常由主机平台提供的原语组成。

用户级仿真和系统级仿真各有利弊。用户级仿真不必仿真整个硬件环境，只需要提供应用程序运行必不可少的虚拟运行环境，诸如系统调用、I/O 等部分由主机操作系统平台提供。因此，用户级仿真实现简单，能够快速搭建仿真环境，并能够让应用程序快速执行。系统级仿真提供了一个完整的虚拟硬件环境，能够在主机平台上运行不同处理器架构的操作系统。因此，对于系统级应用，系统级仿真是必不可少的实现手段。

虽然两种仿真模式应用场合不同，但它们都采用相同的核心翻译引擎。系统级仿真较之于用户级仿真，只增加了设备管理和硬件中断模块，目标指令集的翻译过程完全相同。文章提出的热点代码探测和动态优化模型都适合两种仿真模式，为了方便阐述以及实验结果比较，文章主要讨论用户级仿真模式。

3.3 QEMU 翻译单位

区分解释执行、静态二进制翻译系统和动态二进制翻译系统最主要的标志在于翻译单位的选取，即每次翻译目标代码的长度。解释执行对单条目标指令进行解析，按照目标指令的功能进行操作，每次只解析一条目标指令；静态二进制翻译系统对整个目标代码进行一次性翻译，其后的执行不再进行翻译工作；而动态二进制翻译系统每次只翻译一定长度的目标代码块，称之为基本块，并在运行时动态地进行翻译和执行。

QEMU 是一款动态二进制翻译系统，它对目标代码的翻译按照基本块进行，当执行到未翻译的目标代码时，启动 TCG 核心翻译引擎翻译一块目标代码。为了降低基本块与仿真控制核心来回切换的次数，QEMU 引入了直接块链技术，它通过在每个基本块中增加两条跳转指令，指定下一个基本块的执行首地址，在当前基本块代码执行结束后直接跳转到下一个基本块执行，省去了函数调用和堆栈维

护的开销。

3.3.1 基本块

QEMU 在翻译目标代码时，每次翻译以基本块为单位。在每个基本块中，包含以当前目标 PC 地址开始到遇到分支转移指令（jmp/jne/bl）结束的一段目标代码，图 3-6 给出了一块 ARM 目标码解析所得的 TCG 中间码以及主机代码。QEMU 从当前目标 PC 地址开始，对每条目标指令进行解析，产生 TCG 中间码，当遇到分支转移语句时，将前面目标代码段生成的 TCG 中间码序列进行分析优化，并逐条翻译成主机指令序列，此时所得的主机指令序列构成一个基本块。当分支转移指令为无条件跳转指令(jmp/bl)时，基本块包含一个入口和一个出口，基本块内部无分支指令；当分支转移指令为条件跳转指令（jne/je）时，基本块包含一个入口和两个出口，在基本块代码执行结束时，根据测试条件来判断下一个将要执行的基本块位置（或 PC 地址）。

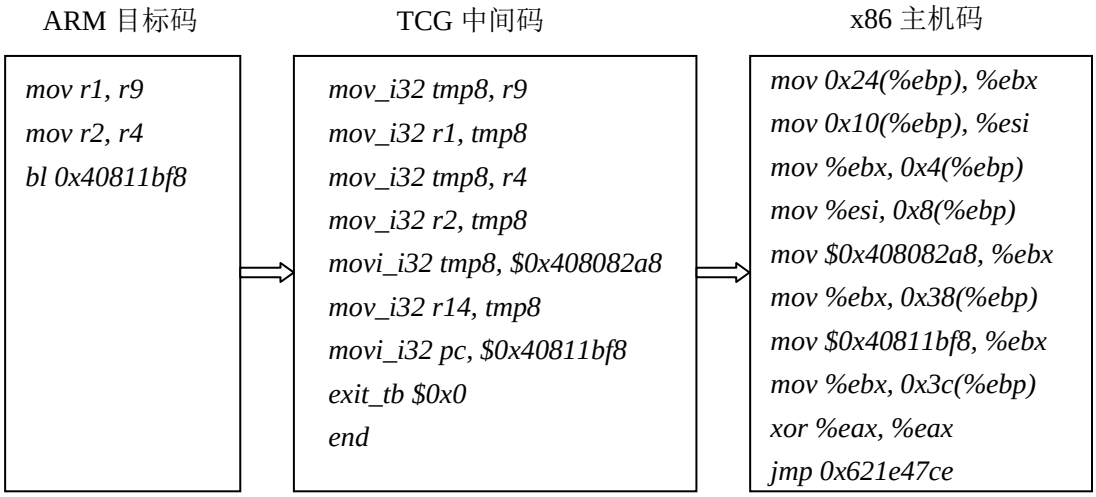


图 3-6 基本块代码示意图

基本块中的主机代码可分为两部分：功能执行和环境保存。功能执行完成目标代码段中每条指令的功能，保证执行此主机代码段和目标代码段在真实物理环境执行的结果一样；环境保存主要用于更新目标 CPU 各寄存器的内容，保证在当前基本块退出时，目标 CPU 环境已被更新。在图 3-3 产生的 x86 主机码中，前六条指令用于完成对应目标代码段的功能，后四条指令用于保存目标 CPU 的环境。

3.3.2 基本块管理

基本块是动态二进制翻译系统的基本执行单元，一旦进入基本块执行，必须执行基本块内部的所有代码。每个基本块代码都对应唯一目标代码段，因此，需要有相应的数据结构来建立目标代码段与产生的主机代码段之间联系。另外，当执行完基本块代码返回主循环时，需要一种快速的方法来定位下一个需要执行的代码是否已翻译，如果已翻译，对应的基本块代码所在的起始地址。QEMU 采用 TranslationBlock 结构体来管理基本块，其主要成员如下所示：

```
struct TranslationBlock {
    target_ulong pc;
    target_ulong cs_base;
    uint16_t size;
    uint8_t *tc_ptr;
    struct TranslationBlock *phys_hash_next;
    uint16_t tb_jmp_offset[2];
    struct TranslationBlock *jmp_next[2];
    struct TranslationBlock *jmp_first;
    [……]
};
```

在该结构中，pc 为目标 CPU 的 PC 起始地址；cs_base 为段基址地址；size 为翻译目标代码段的大小；tc_ptr 指向翻译的主机代码所在缓存空间的起始地址；phys_hash_next 将所有具有相同的哈希值的基本块结构体指针构成一个单链表，该链表用于查找特定 pc 值的基本块；tb_jmp_offset 存放两个跳转位置所在基本块内部的偏移位置，直接块链将用到此成员来重新设置目标跳转位置；jmp_next/jmp_first 共同描述可直接跳转到该基本块的其它基本块。

QEMU 翻译的每块目标代码段都对应一个 TranslationBlock 结构体，用该结构体来建立目标代码段与主机代码段之间的对应关系。所有已翻译的基本块都保存在 tb_phys_hash[CODE_GEN_PHYS_HASH_SIZE]哈希表中，tb_find_slow()函数将使用它来查询特定 PC 地址的基本块。该函数对 PC 地址进行哈希，用得到的哈希值来定位基本块所在哈希表的偏移位置，然后再以 phys_hash_next 成员字段来遍历所有具有相同的哈希位置的基本块，从而实现基本块的查找。此外，QEMU 还提供一个快速哈希表 tb_jmp_cache[TB_JMP_CACHE_SIZE]，该哈希表用于保存最

近使用的基本块，以便快速定位，tb_find_fast()函数将使用它来定位基本块。在查询特定 PC 地址的基本块时，tb_find_fast()函数将首先被调用，如果找到对应 PC 地址的基本块，则直接执行基本块代码，否则将调用 tb_find_slow()函数进行深度搜索。如果对应 PC 地址的基本块代码未找到，说明当前位置的目标代码段还未翻译，QEMU 将启动 TCG 核心翻译引擎对当前 PC 地址的代码进行翻译，翻译后将得到的基本块同时放入 tb_phys_hash 和 tb_jmp_cache 两个哈希表中。

3.3.3 直接块链

QEMU 各基本块之间相互独立，当每个基本块代码执行结束时，目标 CPU 仿真环境各寄存器的值都将被更新，因此，主循环可根据当前 CPU 的 PC 地址来查找下一个待执行的基本块。当基本块返回主循环时，需要进行堆栈恢复操作以及块查询操作，如果每次在执行一块代码后都返回主循环，那么进行堆栈恢复以及块查询将非常耗时，这将降低基本块代码的执行性能。如果当前基本块代码执行结束时，已知下一块将要执行的基本块代码的位置，直接跳转到下一块的基本块代码而不返回主循环，这将极大地改善性能。因此，QEMU 引入了直接块链技术。

QEMU 的直接块链技术在每一个基本块内部都增加了一条（无条件转移）或两条（条件转移）直接跳转指令（jmp）。初始时，跳转指令的偏移为零，当执行到该指令时，不产生任何操作，基本块代码执行结束后直接返回主循环，并将其对应的 TranslationBlock 结构体地址以及分支编号保存至 next_tb 中。当 QEMU 找到下一块基本块时，如果满足一定条件，则通过 next_tb 找到该基本块代码内部的跳转指令，并将其跳转偏移设置为下一个基本块代码的起始地址。通过设置后，当下次再次执行到该基本块代码时，将直接跳转到下一块代码的起始地址开始执行，而不必再次返回到主循环。

图 3-7 给出了引入直接块链后基本块的执行路径。其中，cpu_exec()函数为仿真控制核心主循环；TB 为基本块代码区；Prologue 用于主循环进入 TB 块代码时保存主循环堆栈；在 TB 块代码执行结束返回主循环时，由 Epilogue 来恢复主循环堆栈。

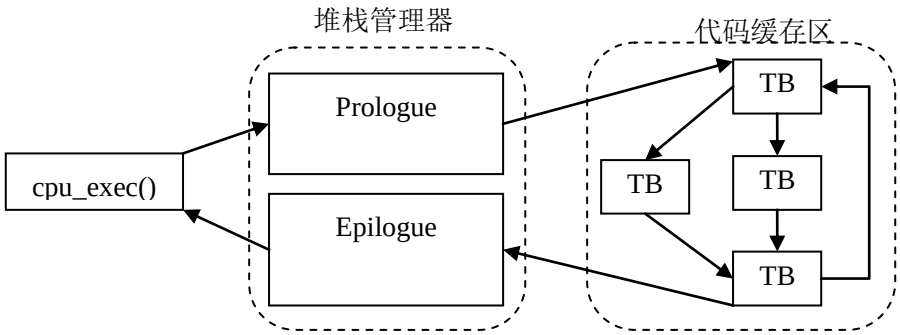


图 3-7 引入直接块链后基本块的执行路径

3.4 QEMU 优化策略

一个动态二进制翻译系统的性能好坏取决于优化策略的选择。一个具有优秀的优化策略的动态二进制翻译系统，将产生较精简的主机代码，能够有效改善主机代码的质量，提高翻译后主机代码的执行速度。如果优化策略选择不当，不仅主机代码质量得不到改善，还会造成很大的优化开销，反而降低系统的执行性能。优化策略的选择需要考虑两个关键因素：优化开销和主机代码质量。

然而，这两个因素存在相互制约关系：如果在翻译代码时进行深度优化，虽然能够产生高质量的主机代码，降低主机代码的执行开销。但是，深度优化是一个非常耗时的过程，优化开销可能抵消高质量代码改善的执行开销；如果优化后的主机代码将多次执行，这无疑会提升整个系统的性能；如果优化后的代码只会被执行一次，那么优化开销将得不到补偿，反而造成降低整个系统的性能。因此，一个优秀的优化策略，不仅要考虑代码优化程度，还需要考虑优化过程的开销。

QEMU 采用的 TCG 核心翻译引擎以其快速翻译而闻名，它能快速响应翻译请求，并进行实时翻译。在 TCG 代码翻译过程中，将以基本块为单位对代码进行优化，其优化过程相对简单，主要分为两部分：TCG 中间码的优化以及主机代码的优化。

3.4.1 TCG 中间码的优化

QEMU 对 TCG 中间码的优化由 TCG 中端优化分析器完成。当反汇编器产生一块 TCG 中间码，将送入 TCG 中端优化分析器，TCG 中端优化分析器对该块 TCG 中间码进行 TCG 生存期分析，以移除中间码中存在的一些僵死操作。TCG 生存期分析逆序扫描整个中间码块，在扫描过程中，标志每条中间码各变量生存周期。

初始时，中间码中所有变量的生存状态都置为真；如果中间码对某变量进行赋值操作，该变量的生存状态将置为假；如果中间码对某变量进行引用操作，则再次将该变量生存状态置为真。当遇到中间码对某变量进行赋值操作时，探测到该变量生存状态已为假，即该条中间码的赋值结果将被后续中间码的赋值结果替换，当前中间码的操作无效，将该条中间码所引用的所有变量的生存状态都置为假。并将操作码置为 NOP，表示该条中间码为冗余操作，在进行代码翻译器不产生任何主机代码。

考虑如下两条 ARM 目标指令：

```
add r1, r2, r3          #r1 <- r2 + r3
add r1, r3, r4          #r1 <- r3 + r4
```

经过反汇编后，其产生的 TCG 中间码如下：

```
mov_i32 tmp8, r2        #tmp8 <- r2
mov_i32 tmp9, r3        #tmp9 <- r3
add_i32 tmp8, tmp8, tmp9    # tmp8 <- tmp8 + tmp9
mov_i32 r1, tmp8        # r1 <- tmp8
mov_i32 tmp8, r3        #tmp8 <- r3
mov_i32 tmp9, r4        #tmp9 <- r4
add_i32 tmp8, tmp8, tmp9    # tmp8 <- tmp8 + tmp9
mov_i32 r1, tmp8        # r1 <- tmp8
```

通过分析发现，以上两条 ARM 目标指令都对寄存器 r1 进行赋值操作，第一条指令的操作结果将被第二条指令覆盖。TCG 中端优化分析器通过 TCG 生存期分析，能够有效地删除这种冗余操作，它逆序对每条中间码各变量的生存状态进行扫描并更新。经过 TCG 生存期分析后，其优化后的 TCG 中间码如下：

```
NOP $0x2, $0x2          #tmp8(0) <- r2(0)
NOP $0x2, $0x2          #tmp9(0) <- r3(0)
NOP $0x3, $0x3, $0x3    # tmp8(0) <- tmp8(0) + tmp9(0)
NOP $0x2, $0x2          # r1(0) <- tmp8(0)
mov_i32 tmp8, r3        #tmp8(0) <- r3(1)
mov_i32 tmp9, r4        #tmp9(0) <- r4(1)
add_i32 tmp8, tmp8, tmp9    # tmp8(0) <- tmp8(1) + tmp9(1)
mov_i32 r1, tmp8        # r1(0) <- tmp8(1)
```

TCG 生存期分析只对整个代码逆序扫描一遍，便可删除一些冗余的操作，因

此，其优化开销较小，不会造成整个系统性能的瓶颈。然而，其优化也具备一定的局限性，它只能消除一些重复的变量赋值操作，不能进行更加深度的优化，在产生的主机代码中还常常存在许多其它的冗余操作。

3.4.2 主机代码的优化

QEMU 对主机代码的优化由后端翻译器完成。后端翻译器在翻译代码过程中，对中间变量采用延迟写策略：一旦中间变量持有主机寄存器，则让该变量一直持有；如果主机寄存器不足，则优先释放优先级高的主机寄存器对应的临时变量，并将该值写入对应的内存区域。当整个缓存块代码执行结束时，将对所有还未写入内存的临时变量全部写入，以保证仿真环境的正确性。

在 QEMU 中，主机寄存器的分配按照一定的优先级分配和释放。例如，在 x86 平台，其可分配的主机寄存器按优先级高低分别为 EBX、ECX、EDX、ESP、EBP、ESI、EDI。其中，ESP 用于存放堆栈指针，EBP 用于存放基址地址。可被中间变量使用的主机寄存器其余六个。当对中间变量进行赋值操作时，为其映射主机寄存器，当临时变量再次被使用时，不再从内存加载该临时变量的值，而直接引用已映射的主机寄存器中保存的值进行操作。

延迟写策略能够减少主机代码中的内存存取指令，减少主机寄存器的分配和释放开销，能够有效改善系统性能。但是，当主机寄存器不足时，该策略按照优先级顺序释放主机寄存器，这会导致一个问题：加入所有寄存器已被分配，当中间变量 T1 请求主机寄存器，QEMU 会将寄存器 EAX 与其对应的中间变量解除映射，并为 T1 映射 EAX，如果此时 T2 也请求主机寄存器，按照优先级原则，会将 T1 持有的 EAX 释放掉，再次分配给 T2，而 T1 在随后的操作中很可能会被使用，又不得不再次请求寄存器分配，并从内存加载数据。导致这个问题的因素在于主机寄存器按照优先级释放原则不当，其具体体现在产生的主机代码中存在许多内存加载指令。

如果在释放主机寄存器时不按照优先级顺序，而是优先释放最近最久未使用（LRU）的中间变量持有的寄存器，这无疑会减少许多内存加载。因此，文章后续章节将提出一种改进的优化方法，以提高主机寄存器的命中率，减少主机代码中的内存存取操作，从而加快系统的执行性能。

3.5 QEMU 执行流程

QEMU 的执行流程可分为两个阶段：仿真环境初始化和代码翻译和执行。仿真环境初始化阶段负责解析命令行参数、构建目标仿真环境、加载并解析目标文件等；代码翻译和执行阶段负责对目标代码进行动态的翻译和执行，并动态的更新目标代码对目标仿真环境操作，以完成目标代码的功能。

3.5.1 仿真环境初始化阶段

仿真环境初始化阶段用于初始化 QEMU 系统运行所需的信息，包括：解析命令行参数、构架目标仿真环境、加载并分析目标文件等。

QEMU 采用命令行的方式启动系统，在启动时，可提供多个参数选项来控制 QEMU 的执行动作和执行目标。例如，-d 选项用于打印目标指令的反汇编(in_asm)、TCG 中间码(op)、优化后的 TCG 中间码(op_opt)以及生成的主机代码(out_asm)；-L 选项用于指定目标文件需要的共享库文件的所在的位置。QEMU 运行的第一步便是解析命令行参数，按照用户的参数来设置系统的特征。

在解析命令行参数后，QEMU 将构建目标仿真环境。构架目标仿真环境主要由 env 结构体表示，该结构体中存放目标仿真环境的所有信息。env 结构体中各目标寄存器的值将被设置为一个初始的执行状态，包括通用寄存器、堆栈寄存器（SP）、基址寄存器（BP）以及程序计数器（PC）等。另外，QEMU 还将构建仿真目标环境的内存布局、页表、系统调用与信号处理例程。

在仿真目标环境构建后，QEMU 将根据命令行参数传递的目标文件名，对其进行加载，加载解析过程由 loader_exec()函数完成。该函数完成对目标文件格式的分析，并按照指定的格式将目标文件的代码段、数据段和堆栈段加载到目标仿真环境的内存布局中。目前，QEMU 支持两种 Linux 平台的目标文件格式：ELF 和 bFLT。ELF 格式是 Linux 系统可执行文件的标准格式，大多数 Linux 可执行文件都采用此格式，bFLT 格式使用并不多，文章采用的测试用例均为 ELF 格式文件。

3.5.2 代码翻译和执行阶段

代码翻译和执行阶段是 QEMU 执行的核心单元。QEMU 初始化仿真环境后，会一直处于此阶段执行代码，直到目标文件执行结束才退出，其翻译和执行过程由 cpu_loop()函数完成。该函数是 QEMU 的翻译系统的核心主循环，它囊括了基

本块的管理与查找、基本块代码的执行、页表管理、缓存空间管理、中断响应和设备管理等。

图 3-8 给出了 cpu_loop()涉及的主要翻译和执行流程。其核心过程如下：

(1) 获取仿真目标 CPU 的 PC 值，执行步骤(2)。

(2) 调用 tb_find_fast()从 tb_jmp_cache[]哈希表中快速定位需要执行的 TB 块，如果找到 TB 块则执行步骤(5)，否则执行步骤(3)。

(3) 调用 tb_find_slow()从 tb_phys_hash[]哈希表中深度搜索对应 PC 值的 TB 块，如果找到 TB 块则执行步骤(5)，否则执行步骤(4)。

(4) 调用 tcg_gen_code()翻译对应 PC 值的一块目标代码，翻译后返回 TB 块，并更新页表、代码缓存区、tb_jmp_cache[]以及 tb_phys_hash[]，执行步骤(5)。

(5) 调用 tcg_qemu_tb_exec ()从 TB 块代码的起始地址执行主机代码，执行完毕返回步骤(1)。

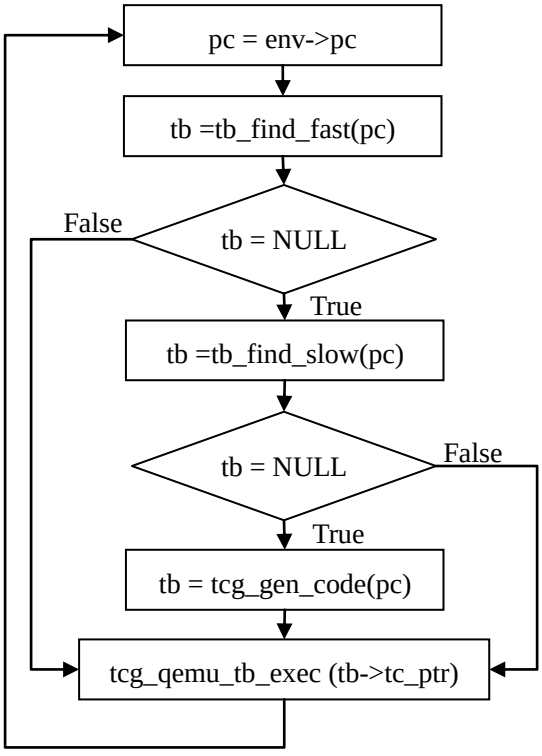


图 3-8 cpu_loop()翻译和执行流程

3.6 本章总结

本章通过对 QEMU 系统框架、仿真模式、翻译单位、优化策略以及执行流程

五个部分的详细介绍，洞悉了 QEMU 系统的实现原理、采用的关键技术、以及动态二进制翻译流程。通过对本章的介绍，为后续章节提出的 QEMU 热点探测与动态优化模型提供了完整的系统模型。同时，本章还介绍 QEMU 的优化策略的不足之处，在本文即将阐述的优化模型中，将针对其存在的不足之处进行详细介绍，并提出一种高效的优化策略进行弥补。

第四章 QEMU 热点代码探测与动态优化模型设计

动态二进制翻译系统能够仿真不同处理器体系架构的指令集以及硬件，它不仅是虚拟化系统的核心技术，还为云计算以及移动计算提供系统支持。因此，二进制翻译系统的性能好坏对相关领域产生重要的影响。然而，以下因素将阻碍动态二进制翻译系统的性能：翻译时的仿真开销、翻译及优化开销以及翻译后主机代码的质量。在虚拟化系统中，动态二进制翻译系统的可重定位能力也是动态二进制翻译系统的重要需求。虚拟化系统要求运行不同处理器架构的可执行程序在不同的主机体系架构上，这种需求将对动态二进制翻译系统在进行设计时增加额外的限制条件，从而导致额外的开销。由于动态二进制翻译系统在运行的同时执行目标可执行文件，翻译系统整体的性能将对翻译后的主机代码的执行性能非常敏感，因此，动态二进制翻译系统很难实现精确优化以产生高质量主机代码。然而，随着多核以及多线程的引入，翻译系统的开销可由多个处理器核心承担。这种翻译系统能够充分利用处理器多核心资源，将自身由单线程变为多线程，这使得翻译系统能够利用单独的优化线程来实现更加精确的优化，并且不影响整个系统的执行开销。

本章以 QEMU 为原型，设计一个热点代码探测与动态优化模型。该模型充分利用多核心以及多线程技术，对 QEMU 产生的主机代码进行热点探测，并对热点代码块进行合并及优化，以产生高质量主机代码，从而提升 QEMU 的性能。

4.1 总体设计

QEMU 是一个可重定位的动态二进制翻译系统，具有翻译速度快，执行效率高等特点。然而，较之于快速翻译，其产生的主机代码质量却不能得到保证。归结其原因，主要在于以下两个方面：

（1）QEMU 的 TCG 翻译引擎对目标代码不做过多的翻译优化，在翻译后的主机代码中，常常存在许多冗余的代码，执行这些冗余代码将造成一定的执行开销。在这些冗余代码主要表现为大量的内存存取操作以及寄存器移动操作，执行这些操作将非常耗时。

（2）QEMU 翻译后的每块主机代码之间相互独立，在每次进入一块代码开始

执行时，需要从内存加载当前的执行环境，主要表现为寄存器数据的加载；在执行一块主机代码后，还需要将所有的数据都写回内存。这种因基本块切换而造成的环境加载已存储将造成极大地执行开销。

针对问题（1），一种有效的解决方法是在翻译代码时对代码进行深度优化，以产生高质量主机代码。然而，QEMU 采用单线程实现，仿真可执行代码与翻译引擎处于同一个运行环境。如果进行深度优化，虽然主机代码的质量得到了提高，但是却增加了翻译时的优化开销，特别是当一些主机代码只运行一次，这种优化工作将得不偿失，反而增加了整个系统的执行开销。

针对问题（2），可适当增大基本块的大小，让每个基本块容纳更多的主机代码。这样将减少基本块的切换次数，相应环境的加载与存储随之降低，从而提升 QEMU 的性能。但是，动态翻译系统设计基本块时，采用以控制转移指令而结束，每个基本块都只有单个入口单个出口。当增加基本块的大小时，一个基本块内部不再呈现这种规律，会存在多个入口和多个出口，不但整个系统的设计将变得非常复杂，而且在翻译时无法确定翻译块的结束位置，在基本块进行跳转时，还无法定位跳转的目标位置。

随着多核心和多线程技术的引入，上述问题便迎刃而解：将 QEMU 单线程变为多线程，QEMU 主线程负责仿真环境的管理以及热点代码探测，当探测出热点代码块时，交由优化线程对热点块进行合并及优化。当下次再次执行到这些基本块时，直接跳转到优化后的主机代码块中执行。QEMU 主线程与优化线程在 CPU 的不同核心并行执行，既充分利用了处理器多核心的架构，又能够在执行时对代码进行动态优化，产生高质量主机代码。图 4-1 给出了多线程热点代码探测与动态优化模型框架。

其中，仿真核心线程为 QEMU 主线程，负责仿真环境的管理、TCG 引擎代码翻译和执行，即原始 QEMU 的执行过程。动态优化线程包括中间码管理器、热点块合并器与热点块优化器三个部分。中间码管理器负责管理基本块与其中间码之间的映射关系；热点块合并器根据获取的热点块结构，找到该结构对应的中间码，与后续的热点块进行合并，以生成一个超级块；热点块优化器负责对产生的超级块进行优化分析，产生高质量的主机代码，并将其存放于热点块代码缓存区。

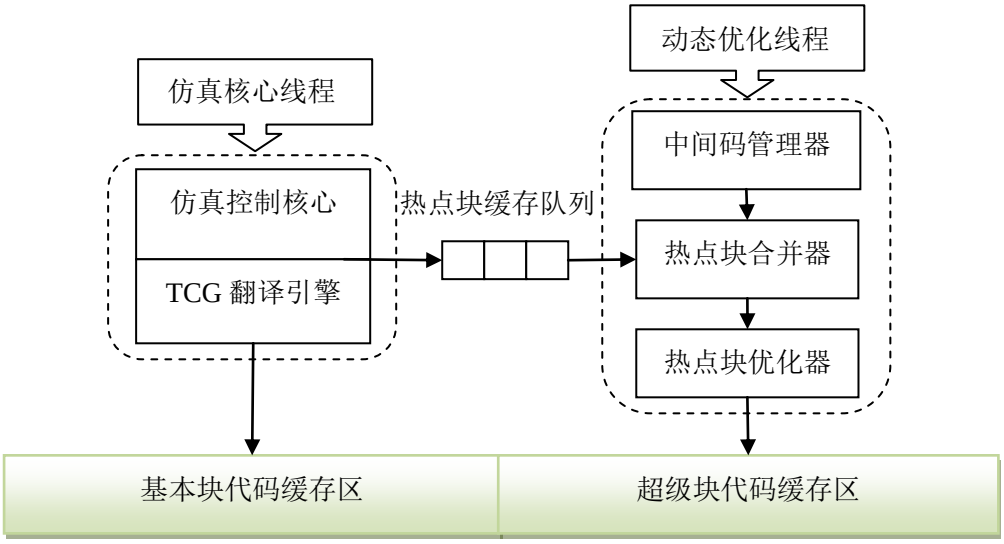


图 4-1 多线程热点代码探测与动态优化模型框架

在一个超级块生成后，当热点块继续执行时，将直接跳转到超级块中执行优化后的主机代码，而不再执行 TCG 翻译的原始主机代码。通过该模型实现，优化开销不再由 QEMU 主线程承担，而是由优化线程在不同的核心中分担优化开销。在优化后的超级块中，既去除了基本块之间的环境装载与存储开销，又对超级块内部代码进行了深度的优化。因此，执行这些超级块将提升主机代码的执行性能，从而提升整个 QEMU 的性能。

4.2 热点代码块的探测

热点代码块是指被频繁执行的一连串基本块代码，即一条热路径。在执行翻译后基本块代码执行时，会存在许多热路径，这些热路径被频繁执行多次。对于动态二进制翻译系统而言，优化每个基本块代码是不明智的，许多基本块代码在执行完一次后就不再重复执行，其进行优化后的代码也不会被执行，对这种基本块的优化非但不能提升效率，还会造成优化线程的开销。相对于优化每个基本块代码，一种更好的策略是只优化经常被执行的基本块，这些块代码执行频率高，通过执行优化后的代码，能够改善翻译后代码的执行效率。因此，如何有效地探测出热路径并对其进行合并优化是提升翻译系统的关键。

4.2.1 代码插桩设计

QEMU 执行翻译后代码时，以基本块作为基本的执行单位，各基本块之间相

互独立，每一个基本块代码都对应一段唯一且连续目标代码。因此，对 QEMU 进行热点代码探测即是探测出这些被频繁执行的基本块。由于 QEMU 引入了直接块链技术，在一个基本块代码执行结束后，直接跳转到下一个基本块代码执行而不返回主循环，这使得不能通过主循环来剖析基本块的执行路径。而探测热路径必须搜集基本块的执行路径信息，通过对执行路径进行分析才能找出频繁执行的路径。

一种可行的解决方案是取消直接块链技术，在每个基本块代码执行结束时都直接返回主循环，在主循环中搜集基本块的执行路径信息。但是，取消直接块链使得 QEMU 仿真核心的执行速度慢了一个数量级，这反而降低了系统的性能。另一种可行的方案是进行代码插桩，每个基本块代码内部都插入一块热点探测代码，在基本块代码执行时同时进行热点代码探测。

代码插桩是指在原始的代码中插入一段自定义的代码，在代码执行时动态搜集代码的执行信息。为实现热路径探测，在 QEMU 的每个基本块首部，插入了一块插桩代码，插桩代码与反汇编后的目标代码一起翻译成主机代码。当每个基本块代码执行时，插桩代码首先被执行，由它来搜集当前基本块代码的执行信息。由于每个基本块都需要进行代码插桩，因此，当 QEMU 对目标代码进行翻译之前，通过 TCG 翻译引擎产生插桩的 TCG 中间码，待插桩中间码生成后，才开始反汇编目标代码。当一块 TCG 中间码反汇编结束，连同插桩代码一并送入后端翻译器进行翻译，以生成一个基本块代码。此时，每个基本块都顺利插入探测代码。

TCG 翻译引擎提供了一套中间描述语言(IDL)，在对目标指令进行反汇编时，根据对目标代码的功能进行解析，采用 IDL 生成对应的中间码。对 QEMU 基本块的代码插桩充分利用了翻译引擎的 IDL 特性。在翻译引擎对目标代码进行反汇编之前，对需要插桩的代码采用 IDL 编写，编写完成后将会产生插桩代码的中间码。插桩中间码和反汇编目标码生成的中间码一样，都能被后端翻译器识别和翻译。

图 4-2 给出了采用 IDL 编写的插桩代码。其中，插桩代码分为两部分：跟踪存根和预测存根。跟踪存根用于定位头块，即频繁执行的基本块序列中第一个基本块。如果基本块是头块（tb 结构体中 enable_profile 使能位置 1），则对该基本块进行引用计数，否则将不执行跟踪存根代码，其主要减少插桩代码的执行频率，降低执行开销，后续章节介绍热点块探测算法是将对其进行详细说明。预测存根负责热路径搜集，当热路径使能位(trace_predict_enable)置 1 时，则将当前基本块指针记录到热路径缓存中。

```
/* trace_stub */
addr = tcg_const_ptr((tcg_target_long)(&tb->enable_profile));
ret = tcg_temp_new_i32();
tcg_gen_ld_i32(ret, addr, 0);
tcg_temp_free_i32(addr);
label = gen_new_label();
tcg_gen_brcondi_i32(TCG_COND_EQ, ret, 0, label);
tcg_temp_free_i32(ret);
tmp = tcg_const_ptr((tcg_target_long)tb);
gen_helper_trace_profile(tmp);
tcg_temp_free_i32(tmp);
gen_set_label(label);
/* predict_stub */
addr = tcg_const_ptr((tcg_target_long)(&trace_predict_enable));
ret = tcg_temp_new_i32();
tcg_gen_ld_i32(ret, addr, 0);
tcg_temp_free_i32(addr);
label = gen_new_label();
tcg_gen_brcondi_i32(TCG_COND_EQ, ret, 0, label);
tcg_temp_free_i32(ret);
tmp = tcg_const_ptr((tcg_target_long)tb);
gen_helper_trace_predict(tmp);
tcg_temp_free_i32(tmp);
```

图 4-2 插桩代码示例

跟踪存根和预测存根的具体操作通过 QEMU 提供的 helper 函数机制实现。由于 QEMU 对复杂的目标指令功能不产生对应的主机代码，而是通过调用 helper 函数来完成其具体功能。因此，设计跟踪存根增加引用计数以及预测存根热路径搜集也采用该技术。gen_helper_trace_profile()函数用于增加当前基本块的引用计数，如果当前基本块为头块，则会进而执行后续的代码，跳转到该函数执行，否则直接跳转到后续预测存根执行。gen_helper_trace_predict()函数完成热路径的搜集，一旦热路径使能位置 1，该函数将被调用，将当前基本块放置于热路径缓存中，否则跳转到基本块后续代码进行执行。图 4-3 给出了一块 ARM 目标代码经代码插桩后产生的 TCG 中间码以及 x86 主机码示例。

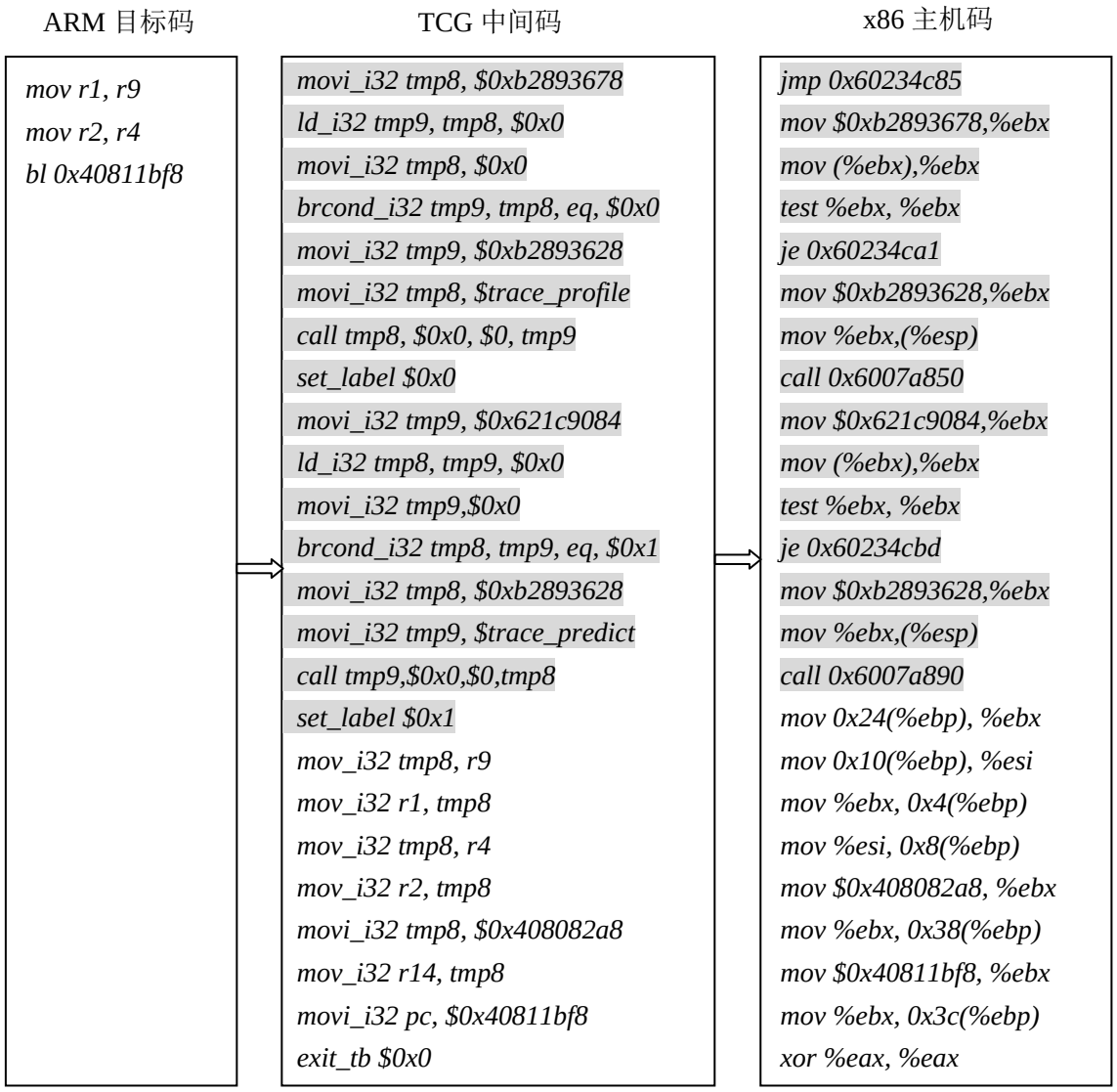


图 4-3 ARM 目标码经代码插桩后对应的 TCG 中间码以及 x86 主机码

图 4-3 灰色部分表示插桩的 TCG 中间码以及 x86 主机代码。在主机代码中存在两条 call 指令，两条 call 指令分别对应跟踪存根和预测存根的两个 helper 函数的位置。当条件满足时，由 call 指令调用对应的 helper 函数来完成具体的功能。

经过代码插桩后，每个基本块代码包含插桩代码和正常翻译的主机代码两部分。当进入基本块代码执行时，插桩代码将首先被执行，完成当前基本块的热路径探测。由于插桩代码存在于每个基本块内部，它不会影响直接块链技术的实现，当通过直接跳转到下一基本块代码执行时，对应基本块的插桩代码也能够正常运行。因此，代码插桩能够很好的兼容 QEMU 翻译机制，并不对其正常执行产生较大的开销。图 4-4 给出了经过代码插桩后基本块的结构以及执行流程。

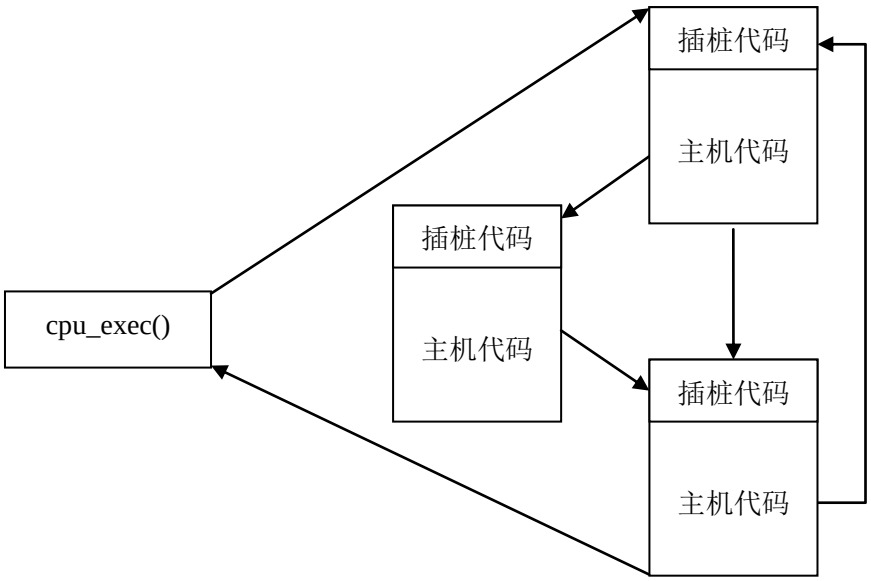


图 4-4 代码插桩后基本块的结构以及执行流程

4.2.2 热路径算法设计

代码插桩有效地解决了直接块链不返回主循环的问题，它对每个基本块都插入一段热点探测代码，在基本块代码执行时进行动态热点探测。但是，代码插桩会增加各基本块代码的指令数量，执行插桩代码会导致执行开销。如果基本块对应的目标代码数量本身较少，翻译后对应的主机条数较少，而插桩的代码可能高于目标码产生的主机码。例如，在图 4-3 中，一块 ARM 目标码生成的指令条数就远远少于插桩代码生成的指令条数。如果插桩代码每次都执行，势必会增加原始仿真核心的执行开销。另一方面，进行代码插桩的目的是用于热点代码块的探测，即找到热路径，一个高效的热点块探测算法显得尤为重要，探测算法的执行开销不仅要低，还需要能够准确找到热路径。基于此，文章设计和实现了一个基于 NET 算法的热点探测算法。

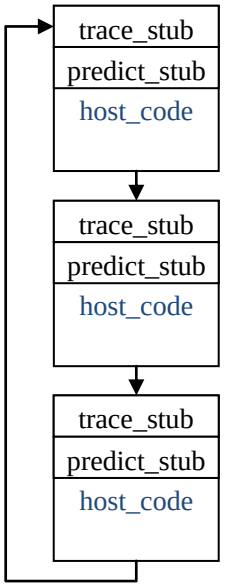
NET 算法是进行热路径探测的实用方法，它能够以较小的代价有效地探测出热点代码块。NET 算法全称为 Next Execute Tail，其核心思想如下：定位基本块的头块，当头块被执行时，增加其引用计数；当头块的引用计数超出给定阈值时，将该头块以及后续的基本块作为一条热路径，直到再次执行到该头块结束。NET 算法定位头块的方式很简单，只要是分支跳转的目标都将其作为头块。NET 算法只将头块进行引用计数累计、阈值判断等，其它非头块的基本块不进行以上操作。由于头块的数量远远少于基本块的数量，通过这种方式，插桩的代码将不会产生

较大的执行开销。另外，非头块的基本块的执行顺序不会出现分支跳转，只通过头块进行计数和判断能够有效地探测出热路径。

文章基于 NET 算法的基本思想，结合 QEMU 自身的特性，设计了一个高效地热路径探测算法。为定位基本块中的头块，在基本块结构体（TranslationBlock）中，引入了一个新的字段 enable_profile，该字段用于标识当前基本块是否为头块，如果是则置 1，否则置 0，初始时默认置 0。

原始的 NET 算法定位头块的方式是采用目标代码翻译时判断，如果当前基本块是分支转移的起始块，则将该块标识为头块。为简化设计，提高头块定位效率，QEMU 采用查询技术实现头块的定位。当通过 tb_find_fast()或 tb_find_slow()查询到基本块时，表明该基本块先前已被生成，且后续跳转基本块将直接跳转到该基本块执行。因此，这两个函数查询出来的基本块即是头块，直接将该基本块对应的 enable_profile 字段置 1。这种设计能够让热点块探测过程开销较小。

虽然 QEMU 每个基本块都插入了热点探测代码，但是，只有头块中的热点探测代码将被执行，其它非头块的热点探测代码只在进行热点路径搜集时才会执行。因此，代码插桩虽然增加了相应基本块的代码长度，但不会增加基本块的执行开销。在 4.2.1 小节曾提及，插桩的代码分为跟踪存根和预测存根。跟踪存根即是判断当前基本块是否为头块，如果是则增加该基本块的引用计数，否则直接跳出跟踪存根代码，进入预测存根；预测存根代码当且仅当热路径使能位置 1 时才执行。图 4-5 给出了插桩代码的基本块结构以及热点探测算法的伪代码。




```
trace_stub:
    jump < target >;
    if enable tb->enable_profile = TRUE then
        tb-> exec_count += 1;
        if tb-> exec_count > THREASHOLD then
            trace_predict_enable = TRUE;
        end if
    end if
predict_stub:
    if trace_predict_enable = TRUE then
        if tb exists in recording list then
            trace_predict_enable = FALSE;
            compile_trace (recording list);
        else
            Add tb to recording list;
        end if
    end if
```

图 4-5 跟踪/预测存根的结构以及热点探测算法伪代码

在图 4-5 描述的伪代码中，exec_count 成员记录每个头块的执行计数，当头块的执行计数超过由 THREASHOLD 规定的阈值时， trace_predict_enable 置 1，将启动预测存根热路径搜集例程。预测存根会将后续所有执行的基本块结构体指针 TB 存放于记录列表（recording list）中，一旦 tb 已存在于纪录列表中，表示热路径环路已形成，将该热路径打包并发送至热路径缓存池。

4.2.3 热路径缓存池设计

热路径缓存池用于保存热点探测代码搜集的热路径块，所有探测出的热路径都将打包发送至缓存池中，等待后续处理。因为热点块的探测与优化合并采用多线程实现，所以需要一种方法来实现线程间的通信和协同工作。文章中采用了缓存池的思想来实现，热路径缓存池接收由热点探测代码发送过来的热路径包，并向合并优化例程提供服务接口，将热路径包送至合并优化例程进行处理。

热路径缓存池由 trace_pool 结构体描述，其结构体定义如下：

```
struct trace_pool {
    uint32_t count;
    struct trace_packet *next;
    struct trace_packet *free;
    pthread_mutex_t lock;
    pthread_cond_t ready;
};
```

其中，count 字段表示当前缓存池中包含的热路径包的数量；next 字段指向下一个可用的热路径包，只要缓存池不空，next 字段将包含下一个可被处理的热路径包；free 字段包含下一个可被分配的热路径包，当需要将热路径放置到热路径缓存池时，需要从 free 字段指向的位置来分配一个可用热路径包；lock 字段用于线程间的互斥访问；ready 字段为一个条件变量，当热路径缓存池为空时，优化合并例程将睡眠在该条件下，一旦缓存池中有新的热路径包到来，将唤醒休眠在该条件下的合并优化例程。

缓存池中存放的基本块以热路径包为单位，一条热路径包含若干个基本块。热路径包由 trace_packet 结构体来描述，其结构体定义如下：

```
struct trace_packet {
    uint32_t count;
```

```
    struct TranslationBlock **packet;
    struct trace_packet *next;
};
```

在该结构体中，count 字段表示该热路径包含基本块的数量；packet 存放具体的热路径基本块；next 字段指向下一个可用的热路径包。

热路径缓存池采用了轻量级的互斥实现，并提供了统一的接口来对缓存池进行管理：对于热点探测例程，当探测到一条热路径时，通过 malloc_trace_packet()函数申请一个可用的热路径包，然后将热路径中包含的基本块指针放入包中；再通过 push_packet(packet)函数将热路径包放入热路径缓存池；对于合并优化例程，当需要处理新的热路径时，通过 pop_packet()函数返回一块热路径包，当包内的热点块处理完毕，再通过调用 free_trace_packet(packet)函数来释放热路径包，以便循环使用。至此，热路径缓存池通过以上接口不断为热点探测例程以及合并优化例程提供服务。

4.3 热点代码块的合并

经过热点代码块探测后，下一步工作便是对热路径中包含的各个基本块代码进行合并，使之构成一个超级块。对热点基本块的合并可通过两种方式实现：其一是通过修改前端反汇编器，根据各基本块中的 PC 地址重新生成中间码；其二是在原始仿真核心翻译时保留各基本块对应的中间码，然后对各热点块的中间码进行合并即可。由于修改前端反汇编器工作量大，且会改动 QEMU 当前的翻译器框架，文章采用了第二种方法，在 QEMU 翻译代码是同时保留中间码。

4.3.1 中间码缓存区设计

原始的 QEMU 并不会保留各基本块的中间码，当翻译完一段目标代码后，其对应的中间码将被释放。为此，文章增加了中间码缓存区，并设计了相应数据结构实现基本块到中间码的映射。

中间码缓存区由 GenOpcBuffer 结构体进行描述，其主要成员定义如下：

```
struct GenOpcBuffer {
    [...]
    uint32_t nr_goe;
    uint32_t cur_goe;
```

```
    GenOpcEmt *goe;
    uint16_t *gen_opc_base;
    TCGArg *gen_opparam_base;
};
```

其中，nr_goe 字段表示可存放中间码块的个数，cur_goe 字段表示当前已存放的中间码块的数量，goe 字段描述中间码块的特征，gen_opc_base 字段为存放中间码的缓存基地址，gen_opparam_base 为存放中间码参数的基地址。

中间码缓存区包含若干个中间码块，一个中间码块对应一个基本块，中间码块由 GenOpcEmt 结构体描述，其包含中间码所在缓存区的位置以及和基本块的映射关系，其结构体定义如下：

```
struct GenOpcEmt {
    uint32_t rtb; /* related TranslationBlock */
    uint16_t *opc_base;
    uint32_t opc_len;
    TCGArg *opparam_base;
    uint32_t opparam_len;
};
```

其中，rtb 字段标志该中间码块对应的基本块地址，opc_base 字段表示中间码块在中间码缓存区的位置，opc_len 字段标志中间码块所占中间码缓存区的大小，opparam_base 字段表示中间码块对应的参数所在缓存区的位置，opparam_len 字段标志中间码块所在参数缓存区的大小。

当 QEMU 翻译目标代码产生基本块时，相应的中间码都会随之保存，以便后续对中间码进行合并。为了快速定位基本块中间码的位置，在基本块结构体中还引入了 opc_offset 字段，该字段标志其对应中间码在中间码缓存区的偏移位置。

4.3.2 中间码的合并

QEMU 翻译产生的中间码都包含一个入口，一个或两个出口。对于无条件分支转移指令，其目的地址明确，只会产生单出口的中间码；而对于有条件分支转移语句，会根据条件真假来选择分支执行，这种中间码将包含两个出口。因此，对中间码的合并需要考虑下一个基本块从当前基本块的哪个分支语句进行跳转。另外，每个基本块对应的中间码都有自身的 label 标号，且都从 0 顺序编号，在合

并后，需要将各个基本块的 label 标号进行映射，实现在超级块中的各个标号的唯一性。经过合并后，产生的超级块将包含一个入口，多个出口。图 4-6 给出了基本块对应的中间码合并前后的结构。

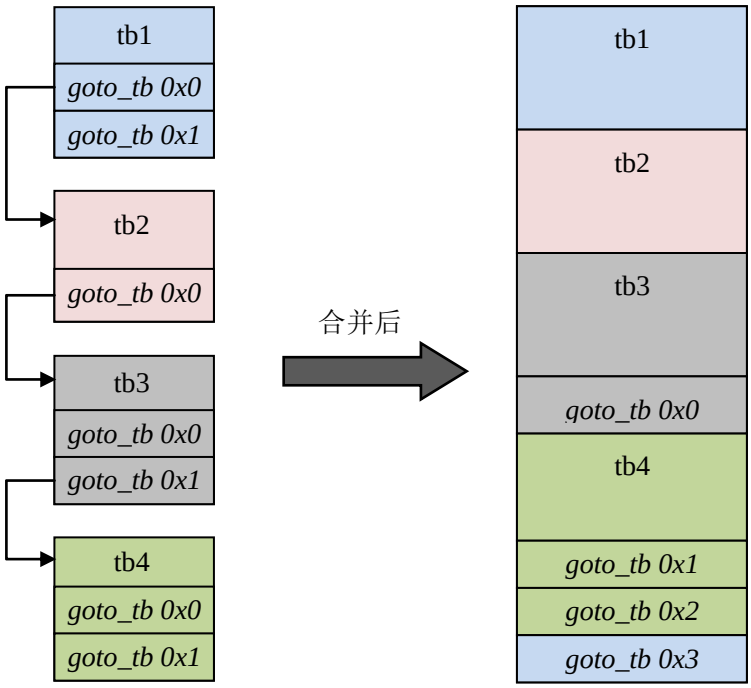


图 4-6 中间码合并示意图

在图 4-6(a)中，tb1~tb4 为待合并的基本块，其基本块之间的跳转关系由图中箭头所示。在合并基本块的中间码时，采用嵌套合并方式。 goto_tb 操作码表示当前基本块结束的跳转指令，每个基本块可能存在两条 goto_tb 操作码，分别对应条件分支转移指令跳转的不同目的位置。合并时，根据 goto_tb 对应的跳转地址与下一基本块的地址进行比较，可得出下一基本块从当前基本块的哪个分支跳转达到。得到跳转分支后，去掉该分支跳转指令，直接将下一基本块对应的中间码放置到后续的合并缓冲中。当前基本块的分支跳转处理完毕，将返回上一层基本块继续进行合并处理，直到顶层的分支转移指令全部处理结束。经过合并后，超级块对应的中间码结构如图 4-6(b)所示。在生成的超级块中，跳转指令由原来的 7 条变为 4 条，各跳转指令对应的标号按顺序递增。

此外，合并中间码还间接地减少了基本块之间的代码量。合并前每一个基本块在执行结束后需要恢复环境，而通过合并后，一些热点分支将不会再产生环境恢复指令。

4.4 热点代码块的优化

经过热点代码块的合并，一条热路径中包含的所有基本块中间码将产生一个超级块，由超级块来描述各基本块需要完成的功能。生成超级块后，需要对超级块进行优化分析，消除块中存在的冗余指令，以产生高质量的主机代码。对超级块的优化包括两部分：块间优化和块内优化。块间优化负责处理基本块之间的环境保存与恢复操作；块内优化用于消除超级块内部存在的一些冗余指令。针对块内优化，文章还提出了一种新颖的优化方案——委托机制。

4.4.1 块间优化

由于 QEMU 以基本块作为代码执行的基本单元，且各基本块之间相互独立，因此，在产生的基本块代码中，每个基本块都存在环境装载与环境存储操作，其主要体现在：在开始执行基本块代码时，所有的目标寄存器都映射为内存，要对目标寄存器进行操作，则需从内存加载各目标寄存器的值，并完成到主机寄存器的映射操作；在基本块代码执行结束，返回主循环或直接跳转下一基本块代码执行之前，需要将所有映射的主机寄存器的值写回目标寄存器，保证目标仿真环境的正确性。每个基本块的执行，都必须执行这些操作，从而导致较大的执行开销。

超级块能有效减少环境转载与存储操作。在一个超级块中，包含多个基本块的代码，各基本块的环境装载将缩减为一次超级块的环境装载，各基本块的缓存存储将缩减为非热点分支的环境存储。在原始的 QEMU 产生的基本块中，每个基本块只包含至多两个出口，在跳出基本块之前，将对所有还未更新的目标寄存器进行更新，更新后再根据分支转移目标跳出基本块。而合并后的超级块将包含多个出口，在执行超级块代码的过程中，可能会出现执行非热点分支目标而直接跳出基本块。非热点分支是指热点基本块的目标跳转分支不再当前热路径中，在执行非热点分支时，需要存储当前的环境，将所有已映射的主机寄存器的值写回目标寄存器中。因此，在超级块跳出非热点分支之前，才进行环境的存储。

超级块中每一个非热点分支跳转都必须有单独的环境存储操作，每个环境存储操作与各基本块的非热点分支跳转一一对应。由于超级块退出的时机不同，其对应的主机寄存器映射以及返回目标都存在差异，因此，不同的退出分支需执行不同的环境存储操作，以保证仿真环境的正确性。图 4-7 给出了超级块的环境装载与存储结构。在图中产生的超级块中，针对每一个非热点跳转分支，都存在相应

的环境存储操作，这些环境存储操作在将在超级块代码进入不同的退出时机时执行。例如，超级块执行 tb1 的主机代码后便直接转入非热点分支，此时将先执行环境存储 4，随后再执行跳转目标 4。超级块中的非热点跳转目标和基本块的跳转目标一样，可使用直接块链技术，直接跳转到目标基本块代码执行。

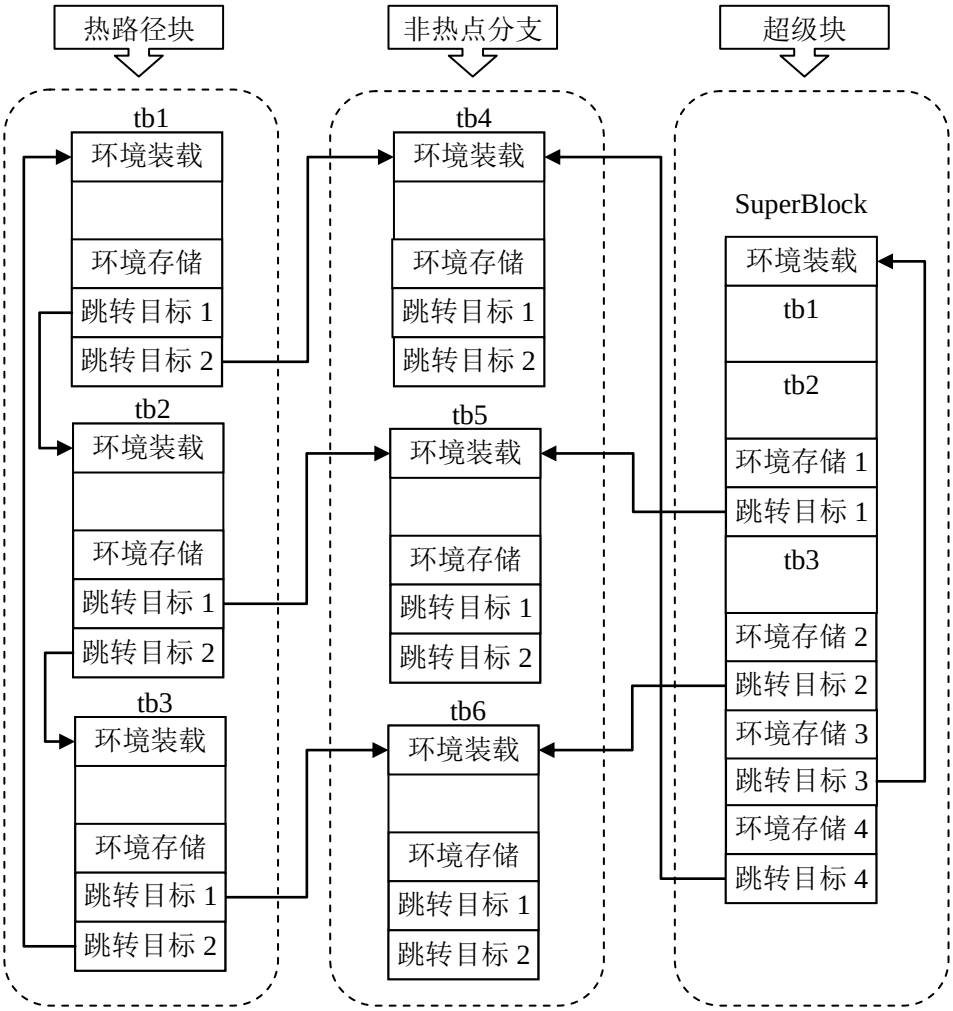


图 4-7 超级块的环境装载与存储结构图

4.4.2 块内优化

块内优化是指对超级块内部的中间码进行分析，消除一些冗余的代码，以产生高质量的主机代码。在原始的 QEMU 中，翻译生成的主机代码的质量主要受以下几个因素影响：

（1）由于不考虑指令之间的依赖关系，当相邻指令需要使用同一操作数时，需再次存取内存，导致频繁的产生内存加载操作。

（2）中间变量之间的赋值操作将产生大量的寄存器移动操作。

（3）基于临时变量的主机寄存器映射策略会频繁地进行寄存器分配和释放，增加翻译时开销。

针对上述问题，文章提出了一种新颖的改进方案：在 TCG 基础之上，为中间变量引入一种新的类型——委托类型。该方案能有效消除冗余的内存加载操作和寄存器移动操作，提升主机码质量；同时，改进的寄存器分配策略能提升主机寄存器的命中率，降低其分配和释放开销。

在大多数体系结构中，由于主机寄存器数量的限制以及主机平台和目标平台之间的差异性，QEMU 很难实现目标寄存器到主机寄存器的直接映射。因此，QEMU 将所有仿真目标寄存器都映射为内存，仅为临时变量分配主机寄存器。临时变量的生命周期非常短，一般只作用于一条目标码。当一条目标码解析完成时，所有的临时变量都将被释放，映射给临时变量的主机寄存器也将被回收以便再次分配。由于目标码之间相互独立，主机寄存器在使用后立即释放，这能有效防止寄存器争用情况的发生。但它不考虑指令之间的相关性，这会产生冗余的内存加载操作。图 4-8 给出两条 ARM 目标码因指令相关性而导致产生重复的内存加载。图中目标寄存器 r2 同时作为两条 ARM 目标码的输入操作数。前面提及，仿真目标寄存器初始时映射为内存，在解析目标码操作之前需将其值从内存加载到主机寄存器中，x86 主机码（1）和（2）用于完成加载 r1 与 r2 的值。当第一条目标码解析完成时，r3 由于赋值操作将持有主机寄存器 EBX，r1 和 r2 均未持有主机寄存器。在解析第二条目标指令时，r2 需再次从内存加载，对应于 x86 主机码（4）。这种冗余的内存加载操作便是由于指令间相关性问题引起的，其体现在当前操作的输入操作数在其后的操作中还将用作输入。如果在解析目标码时对后续操作还将使用的输入操作数预先映射主机寄存器，则可消除这种冗余的内存加载操作。

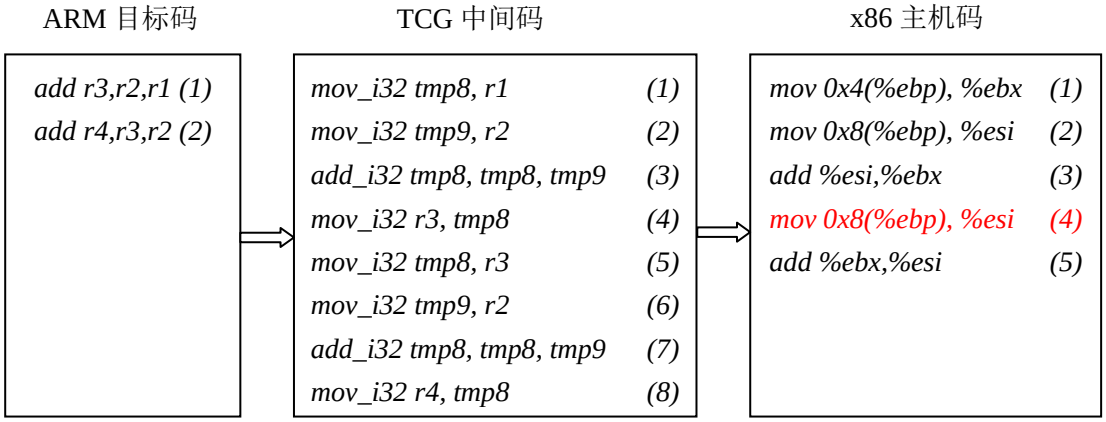


图 4-8 指令相关性问题引发的重复内存加载

除了存在冗余的内存加载操作外，TCG 还会产生一些冗余的寄存器间移动操作。目标码中的赋值操作便会产生寄存器移动操作。例如，一条 ARM 目标码：mov r2, r1，TCG 翻译引擎会产生一条 x86 主机码 mov %ebx, %esi（假定 r1 已映射主机寄存器%ebx），以便让 r1 和 r2 各自持有独立的主机寄存器。通过对 TCG 翻译引擎深入分析发现，一旦中间变量映射主机寄存器，除非该变量显示地解除映射关系，否则该主机寄存器的值就不会改变。因此，针对上述赋值操作，更好的办法是让 r1 和 r2 共享同一主机寄存器，从而避免产生这种寄存器间移动操作。同时，共享主机寄存器也会减少寄存器分配和释放开销，降低主机寄存器的争用率。

在 TCG 翻译引擎中，中间变量（以下简称变量）被描述为一个结构体 TCGTemp，由该结构体中包含的多个字段共同来描述当前变量的状态。下面给出了该结构体包含的主要成员及其相关含义：

```
struct TCGTemp {
    int val_type;
    int reg;
    tcg_target_long val;
    int mem_reg;
    tcg_target_long mem_offset;
    [……]
};
```

字段 val_type 用于描述变量的取值类型，其取值的不同将影响其它字段的释义，下面给出它的几种取值：

① TEMP_VAL_DEAD：变量为僵死变量，其值无意义。

② TEMP_VAL_CONST：变量为常量，val 字段存放当前变量的值。

③ TEMP_VAL_REG：变量已映射主机寄存器，其值保存在 reg 字段所映射的主机寄存器中。

④ TEMP_VAL_MEM：变量映射为内存，其值存放于由 mem_reg 字段和 mem_offset 字段所指向的内存位置。

委托原意为把事情托付给别人代其办理，文章中也遵循这种思想。正如 TCG 翻译策略那样，先将所有的目标码的输入操作数赋给临时变量，由临时变量来完成具体的操作，操作完毕后，再将结果写回给输出操作数。

为实现共享寄存器映射，文章针对 TCGTemp 结构体成员的 val_type 字段，引入了一种新的取值类型——委托类型（TEMP_VAL_DELEGATE）。当 val_type 为

TEMP_VAL_DELEGATE 时，val 字段指向被委托的对象，变量的值由 val 所指的对象决定。当变量为委托类型时，该变量与其所指对象共享同一取值。在对委托类型变量进行操作时，直接引用被委托对象的取值来进行操作。如果变量所指的对象在操作过程中取值发生改变，则先将其取值传递给该变量，再对它进行更新。

在原始的 TCG 翻译过程中，针对赋值操作，不论输入变量为何种取值（立即数除外），输出变量都会映射主机寄存器，这是导致产生寄存器间移动操作的主要原因。引入委托机制后，输入变量在以下两种情况下，输出变量会成为委托类型，此时它既不会映射主机寄存器，也不会产生任何寄存器移动操作。

（1）输入变量为内存类型，且生命周期未结束。

（2）输入变量为寄存器类型，且生命周期未结束。

针对情况（1），需先从内存加载输入变量的值到主机寄存器，并将该主机寄存器分配给它。此后，两种情况都将输出变量置为委托类型，并都指向输入变量。图 4-9 给出了委托机制寄存器预分配伪代码。其中，ts 代表输入变量，ots 代表输出变量，IS_DEAD_ARG 用来判断输入变量是生命周期是否结束，ref_count 字段表示该变量的引用计数。

```
if(ts->val_type = TEMP_VAL_REG) {
    reg = ts->reg;
} else if(ts->val_type = TEMP_VAL_MEM) {
    reg = reg_alloc();
    tcg_out_ld(reg, ts->mem_reg, ts->mem_offset);
}
if(IS_DEAD_ARG(1)){
    /* 输入变量生命周期结束 */
    ots->reg =reg;
    ots->val_type = TEMP_VAL_REG;
    ts->val_type = TEMP_VAL_DEAD;
} else {
    /* 输入变量生命周期未结束 */
    ts->reg = reg;
    ots->val_type= TEMP_VAL_DELEGATE;
    ots->val = get_position(ts);
    ts->ref_count++;
}
```

图 4-9 委托机制寄存器预分配伪代码

当输入变量为委托类型时，赋值操作会产生两种新的情况：

（1）输入变量为委托类型，且生命周期结束。

（2）输入变量为委托类型，且生命周期未结束。

对于上述两种情况，输出变量也将置为委托类型，并让其指向与输入变量所指向的相同对象，此时，多个变量间共享同一取值。 图 4-10 给出了委托机制寄存器移动伪代码。在解析寄存器移动操作时，直接让输出变量指向输入变量，输出变量自身只为委托类型，并增加委托对象的应用计数。此时，多个变量共享主机寄存器映射。当输入变量的生命周期结束时，将该变量从共享主机寄存器链中移除，并减少变量所指对象的引用计数。

```
if(ts->val_type = TEMP_VAL_DELEGATE) {
    dts = variable(ts->val);
    ots->val_type= TEMP_VAL_DELEGATE;
    ots->val = ts->val;
    dts->ref_count++;
    if(IS_DEAD_ARG(1)) {
        /* 输入变量生命周期结束 */
        ts->val_type = TEMP_VAL_DEAD;
        dts->ref_count--;
    }
}
```

图 4-10 委托机制寄存器移动伪代码

针对具有指令相关性的输入变量，委托机制将为输入变量预先映射主机寄存器，并让输出变量成为委托类型。当后续指令使用同一输入变量时，可直接进行操作而无需再次从内存加载数据。例如，在图 4-8 所示的代码中，引入委托机制将消除 x86 主机码（4）。针对寄存器间移动操作，委托机制直接将输出变量置为委托类型，并指向输入变量，此时两个变量共享主机寄存器映射，而不产生任何 x86 主机代码。

此外，在原始 TCG 中，主机寄存器的分配策略按照固定优先级进行分配。如果出现寄存器争用，则释放优先级最高的主机寄存器，然后进行本次分配。然而，这种分配策略会引起寄存器抖动：当前释放的主机寄存器所对应的变量在不久的将来又将重新使用，此时又必须再次为该变量映射主机寄存器并从内存加载其值。委托机制能够消除这种抖动，它对每一个变量引入了委托计数（ref_count），当变量被委托时，其计数增 1。计数值越大，表明引用该变量取值的变量就越多，那么

它在后续操作中的使用频率就越高。当出现主机寄存器争用时，优先释放委托计数值最小的变量。通过这种寄存器分配策略，既能有效消除寄存器抖动，又能显著提高主机寄存器的命中率。

综上所述，委托机制能够有效消除超级块中存在的重复内存存取操作以及寄存器移动操作，通过改进的寄存器分配策略，还能够提升主机寄存器的使用效率。

4.4.3 热点块重定位

通过对超级块的中间码进行合并优化，将产生一块主机代码，生成的主机代码将存放于超级块代码缓存区。当 QEMU 再次执行到已优化热路径基本块时，需要有一种方法让代码的执行路径直接跳转到超级块中执行，而不是再次执行原始的基本块代码。

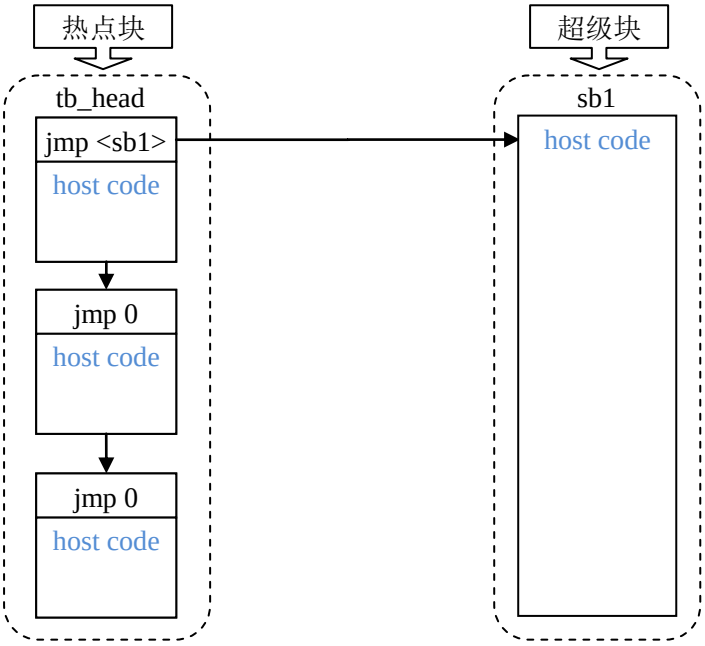


图 4-11 热点块重定位示意图

为实现热点块的重定位，设计时对每个基本块头部都增加了一条直接跳转指令。初始时，直接跳转目标为 0，基本块代码将顺序执行。当超级块代码生成后，将当前热路径的头块基本块内部的直接跳转目标置为超级块代码的首地址。因此，如果热路径代码再次执行，将首先进入热路径头块的代码执行，执行第一条便是跳转指令，通过跳转后，继而转入超级块代码的执行。图 4-11 给出了热点块代码跳与超级块的跳转关系。在图中所示的热路径中，第一个基本块即为头块 tb_head，当生成超级块 sb1 后，头块 tb_head 的第一条跳转指令的跳转目标将被设置为超级

块代码所在超级块缓存区的首地址。通过设置，当 QEMU 再次执行到已优化的热路径时，可直接通过头块的直接跳转指令重定位到超级块代码中执行，而不会再次执行基本块原始的执行路径。

4.5 本章总结

本章对 QEMU 多线程热点代码探测与动态优化模型的总体设计框架、热点探测机制的原理和实现、热路径缓存池的设计、热点代码块的合并和优化进行了详细介绍。

总体设计框架给出了文章设计的核心架构，随后介绍了总体框架各部分的具体实现：通过代码插桩实现热点代码探测，在基本块代码执行时动态进行热点探测；并设计和实现热路径缓存池，保证热点探测例程与热点代码优化例程能够并行执行；设计和实现中间码缓存池，实现热点代码块的中间码合并，以产生超级块；对产生的超级块进行了块间优化和块内优化，块间优化消除基本块之间重复的内存加载和存储操作，块内优化通过委托机制有效消除超级块中的一些重复的内存存取操作以及寄存器移动操作，并有效改善主机寄存器的命中率，以提升主机代码的质量；最后提出热点块重定位技术，使已优化的热点块代码直接跳转到超级块代码中执行，从而达到文章的设计目的。

第五章 性能评估

通过对 QEMU 进行热点代码插桩以及动态优化模型的设计，QEMU 在执行时将进行热点代码探测，并采用多线程对探测的热路径进行合并及优化，以其提升 QEMU 的执行性能。本章将原始 QEMU 与改进型 QEMU 进行比较，来详细阐述改进型 QEMU 的执行性能。另外，委托机制是文章优化算法的核心算法，为了体现委托机制的性能，本章还将原始 QEMU 与只引入委托机制的 QEMU 进行比较，包括：主机寄存器映射次数，翻译生成的主机代码量以及引入委托机制前后 QEMU 的执行性能。

5.1 实验准备

本章测试的处理器体系架构为 Intel Pentium(R) Dual-Core CPU 双核处理器，CPU 主频为 2.00 GHZ，内存大小为 2.00 GB，主机操作系统平台为 32 位 Ubuntu 10.04 UTS。

由于 QEMU 在编译后针对不同的仿真目标架构生成各自独立的可执行文件，本章只针对 ARM 目标仿真环境进行测试分析。QEMU 仿真 ARM 体系架构平台时，可直接在本地处理器系统架构（x86）上执行 ARM 处理器体系架构的可执行目标文件。

本章将采用三类测试用例来评估改进型 QEMU 的性能：

（1）普通 ARM 目标可执行文件，包括：ls、uname、dmesg、which、who 五个可执行目标文件，这五个可执行文件都是 Linux 系统的标准例程，只是该例程对应的指令集为 ARM 目标仿真环境的指令集；

（2）nbench 基准测试程序，该基准测试程序能够测试目标仿真架构的执行性能，该程序经过 arm-linux-gcc 交叉编译工具编译生成 ARM 目标仿真环境的可执行文件；

（3）SPEC CPU2006 基准测试套件，该套件提供多种测试用例，可用来测试 CPU 的处理性能。

为了详细阐述改进型 QEMU 各个优化模块的性能，本章将待测试的 QEMU 可

执行文件分为以下三种：

（1） QEMU： 即原始的 QEMU，版本为 QEMU 0.15.0。该版本 QEMU 的 TCG 翻译引擎未做任何修改与优化。

（2）DQEMU：引入委托机制但未增加热点代码探测与合并优化例程的 QEMU，该版本在原始 QEMU 的基础之上，对 TCG 翻译引擎作了修改，采用委托机制实现主机代码的优化。DQEMU 仍然采用串行化得执行方式，即翻译、优化、执行都在一个核心线程中交替执行。

（3）CQEMU：既引入了委托机制，又增加了热点代码探测与合并优化例程的复合型 QEMU。该版本在原始 QEMU 基础之上，采用代码插桩实现热路径探测与合并，并采用委托机制对合并的超级块进行优化。CQEMU 采用了多线程机制，核心执行线程和合并优化例程采用不同的线程执行，可同时在处理器不同的核心上并行执行。

5.2 DQEMU 性能评估

委托机制是针对 TCG 核心翻译引擎因翻译代码的质量不高而提出的一种优化方案。因此，本节将通过三个方面来阐述引入委托机制前后 QEMU 的执行性能。包括：主机寄存器映射、翻译后主机代码生成量以及 nbench 执行效率。

5.2.1 DQEMU 的主机寄存器映射

QEMU 在翻译目标代码时，每一条目标指令的寄存器都必须映射为主机寄存器才能进行操作。由于仿真目标平台的寄存器与主机平台的寄存器的差异性，可能一一映射所有的目标寄存器。例如，ARM 仿真目标平台的寄存器个数达到 50 个，而 x86 主机平台可用的主机寄存器只有 8 个。因此，在翻译目标代码时，如果主机寄存器不足，则会将已映射的主机寄存器释放，以便为当前的目标仿真寄存器进行映射。由于已映射的主机寄存器包含了某一目标寄存器的数据，释放映射时需要将数据写回内存，因此，释放主机寄存器会产生不必要的内存存储操作。由此可知，主机寄存器的映射次数越少，其产生的内存存储操作就越少。

图 5-1 给出了五个标准 Linux 命令执行后，原始 QEMU 与引入委托机制后 QEMU 的主机寄存器映射次数对比。从图中可知，采用委托机制后，主机寄存器的总映射次数明显减少，即主机寄存器分配与释放次数减少，从而对应的内存存储操作有效降低。 经过试验比较，引入委托机制后 QEMU 的主机寄存器映射次

数较原始 QEMU 的主机映射次数减少约 10%。

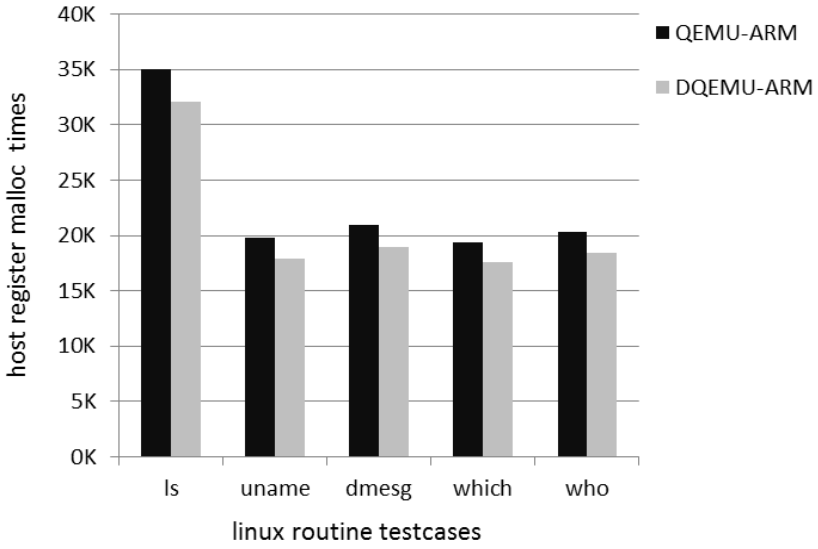


图 5-1 主机寄存器映射次数比较

5.2.2 DQEMU 翻译后的主机代码生成量

引入委托机制的最终目的是减少翻译后主机代码中存在的一些冗余指令，包括内存存取指令以及寄存器移动指令。其主要体现在翻译后生成的主机代码质量更加精简，用更少的指令来完成同等功能的操作。翻译的主机代码量越少，其执行的开销就越低。因此，翻译生成的主机代码量是衡量其执行性能的主要因素。

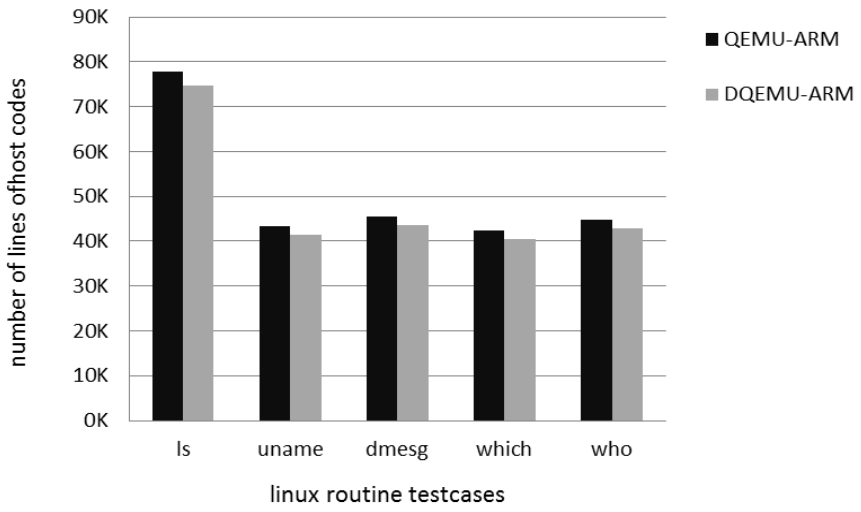


图 5-2 翻译后代码生成量比较

图 5-2 给出了五个标准 Linux 命令执行后，原始 QEMU 与引入委托机制后 QEMU 的翻译生成的主机代码量的对比。从图中可得知，引入委托机制后 QEMU 产生的主机代码量明显减少，其主要减少内存存取操作与寄存器移动操作两类指令，其主机代码量较原始 QEMU 减少约 8%。

5.2.3 DQEMU 的执行性能

nbench 是一款用于测试 CPU 整数、浮点数与内存性能的基准测试程序，由于 QEMU 仿真目标处理器架构，因此，可采用该基准测试程序来测试 QEMU 的执行性能。前两小节已提及，引入委托机制后，QEMU 的主机寄存器映射次数明显减少，翻译生成的主机代码量得到降低，因此，QEMU 执行目标代码的性能应能得到提升。

图 5-3 给出了引入委托机制前后，nbench 的执行性能对比。从图中可得出，引入委托机制的 QEMU 执行性能明显高于原始 QEMU，其平均性能提升约 7%。

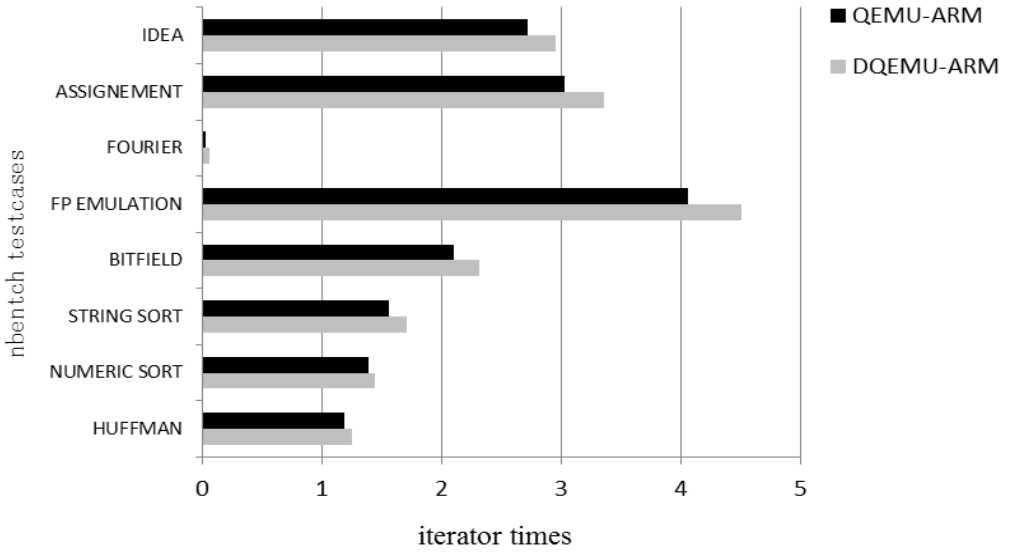


图 5-3 nbench 执行性能比较

5.2.4 DQEMU 结果分析

DQEMU-ARM 在原始 QEMU 中引入了委托机制，该机制对具有指令相关性的寄存器预分配主机寄存器，下次使用时直接从操作主机寄存器而无需内存加载操作，有效降低主机代码中的内存存取操作；其次，该机制还引入了主机寄存器共享策略，多个中间变量和映射相同的主机寄存器，降低了主机寄存器的争用率，

同时降低主机寄存器的分配和释放开销，有效地减小主机寄存器间的移动操作；最后，该机制还改善了 QEMU 主机寄存器分配策略，优先释放引用计数最小的寄存器，提升主机寄存器的命中率，进一步减少内存存取操作。

通过实验结果分析，DQEMU 能够减少主机寄存器分配次数，提升主机代码质量，从而能够有效提升 QEMU 的执行性能。

5.3 CQEMU 性能评估

CQEMU 是指引入多线程热点探测机制和动态优化模型的增强型 QEMU，本节将通过 SPEC CPU2006 测试套件完成对 CQEMU、DQEMU 以及原始 QEMU 三者的性能对比，从而全面展示 CQEMU 的执行性能。

5.3.1 SPEC CPU2006 测试套件部署

SPEC CPU2006 是一套评估计算机系统的标准，可用于测量和对比 CPU 处理整数、浮点数的性能。该标准中包含多个测试用例，例如 gcc、gzip2、hmmer 等，其通过测量各测试用例在待测平台上运行的时间来衡量 CPU 的执行性能。SPEC CPU2006 测试套件以源代码的形式发布，且提供不同软硬件平台（不同的操作系统以及处理器架构）的配置文件，其安装过程可由该套件提供的标准命令来执行。

由于文章以 QEMU 的 ARM 仿真环境为测试对象，测试套件需要编译成 ARM 处理器架构的可执行程序。因此，需要配置测试套件的编译环境，并安装跨编译工具链 arm-linux-gcc。

arm-linux-gcc 是一款跨平台编译工具，它可在 x86 处理器平台上将源文件（C/C++源文件）编译成 arm 处理器平台可执行的目标镜像，其编译过程和本地 GCC 编译器编译过程一样。当对 arm-linux-gcc 跨编译工具链安装完成后，在 SPEC CPU2006 测试套件提供的配置文件中，只需将对应配置文件中的 C/C++编译器的路径设置为 arm-linux-gcc 所在路径即可。

SPEC CPU2006 提供的测试用例非常多，本节选择了六个典型的测试用例来进行性能评估，包括 mcf、bzip2、gcc、hmmer、gobmk、lbm 以及 namd。在这些测试例子运行中，其运行时可通过参数来制定迭代次数，并可输入不同的测试数据来验证性能。针对每个测试用例，本节都采用套件自身提供的测试数据，其存放于测试用例对应的 Ref 目录。

5.3.2 CQEMU 性能测试

图 5-4 给出了 CQEMU、DQEMU 以及原始 QEMU 执行 SPEC CPU2006 测试套件中测试用例的运行时间对比结果。其中，X 轴表示所采用的测试用例，Y 轴表示各测试用例在不同测试目标运行的正常执行时间。

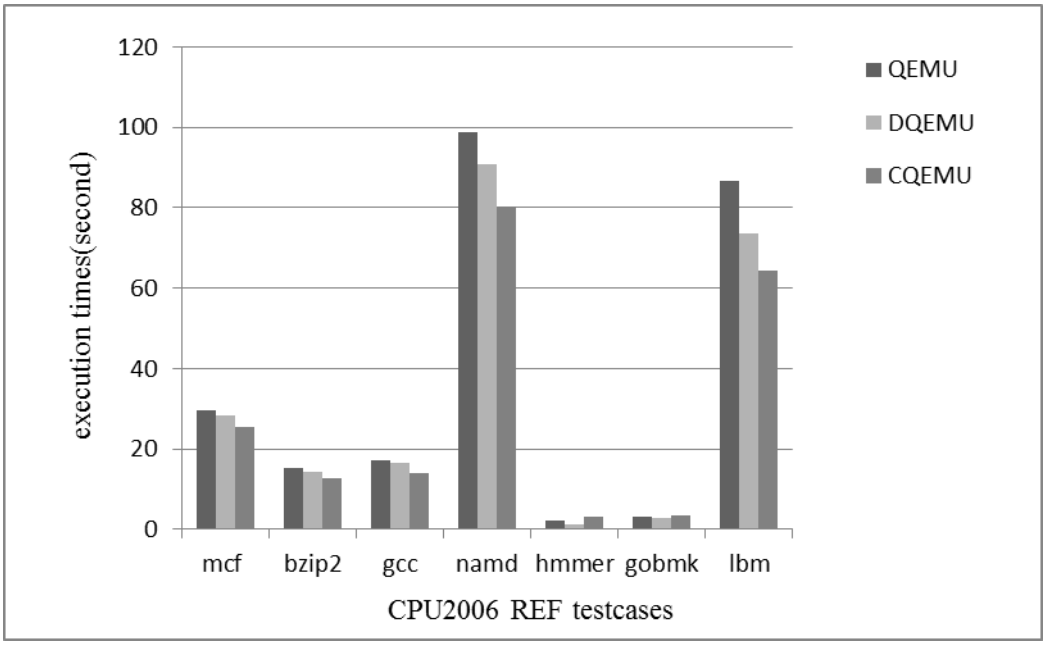


图 5-4 SPEC CPU2006 性能测试

从图 5-4 中可知，只引入委托机制的 DQEMU 在各个测试用例中都运行的很好，且较于原始 QEMU 其执行时间都有所减少，这表明委托机制能够有效地改进产生的主机代码质量，能提升 QEMU 的执行性能。然而，由于 DQEMU 采用单线程实现，即对中间码的优化和翻译后主机代码的执行需要交替进行，因此，委托机制存在一定的优化开销，因此对 QEMU 的执行性能提升具有一定的局限性。另一方面，委托机制无法消除基本块之间的环境装载与存储操作，由于各基本块之间相互独立，基本块之间的跳转位置未知，无法去除这些重复的环境装载与存储。其优化范围具有一定局限性，只能优化基本块内部的一些冗余加载和寄存器移动操作。

在引入热点代码探测和动态优化模型后，CQEMU 的性能较于 DQEMU 有很大的改进。在 CQEMU 中，由于采用了多线程来实现热点代码探测和主机代码优化，核心仿真线程和优化线程相互独立，进行代码优化的开销由运行于不同处理器核心的优化线程来承担，这既不会增加主线程的开销，也不会妨碍主线程的正

常执行。当优化线程对代码优化完毕，将产生的精简的主机代码放入跟踪缓存区，并将原始基本块的块头重定位到优化后的代码块的位置，这样，当热路径再次执行时，将直接跳转到优化后的代码块中执行，从而提升 QEMU 的执行效率。

在图 5-4 中，CQEMU 对于 mcf、bzip2、gcc、namd 和 lbm 五个测试用例的执行时间都有效地缩减，特别是 namd 和 lbm，其正常执行时间较原始 QEMU 缩短约 21%，较之于 DQEMU 缩短约 10%。从图中分析得知， namd 和 lbm 的执行时间最长，对于长期运行的程序，由于其迭代执行的次数最多，那么 CQEMU 执行合并优化后的代码的机会就越多。每一次执行优化后的主机代码，都相应缩减因环境装载和恢复操作以及执行冗余指令带来的开销，从而提升一定的性能。

然而，对于 hammer 和 gobmk 两个测试用例，CQEMU 的执行时间较于 DQEMU 以及原始 QEMU 的执行时间都要长，这是由于这两个测试用例运行时间较短，其执行优化后的主机代码带来的性能提升还无法完全抵消因执行插装代码而带来开销。CQEMU 中对每个基本块都进行了代码插桩，用于探测热点代码，插桩代码的执行会造成一定的执行开销，另外，线程间的通信也会存在一定的开销。

CQEMU 通过长时间运行测试用例，利用执行合并优化的主机代码提升的性能来抵消执行插桩代码以及线程通信的开销。由于 hammer 和 gobmk 两个测试用例运行时间都相对较短，执行优化合并后的主机代码所提升的性能还不足以抵消执行插桩代码和线程通信的开销，从而造成其时间较于前两者都长。

5.3.3 CQEMU 结果分析

CQEMU 通过引入热点代码探测和动态优化例程，在代码执行过程中对热点路径块进行合并优化，通过执行优化合并的超级块代码来提升 QEMU 的执行性能。由于在每个基本块中都插入了插桩代码，插桩代码的执行对导致一定的开销，多线程通信也会造成一定的执行开销，CQEMU 通过执行优化合并的超级块来抵消执行插桩代码和线程通信的开销。因此，CQEMU 适合执行需要长时间运行的程序，程序的运行时间越长，其执行合并优化后的代码的次数就多，相应提升的性能就越高。对于短期运行的程序，其执行速度不一定能够得到改善，反而有可能会有所降低。

5.4 本章总结

本章对引入委托机制的 DQEMU 进行了详细的实验测试，通过主机寄存器映

射次数、主机代码生成量、nbench 的执行效率三个方面来阐述该机制对 QEMU 的性能提升情况，实验测试结果表明 DQEMU 能够有效地改善主机代码质量，提升 QEMU 的执行性能。此外，本章还对采用多线程热点代码探测与动态优化模型的 CQEMU、DQEMU 以及原始 QEMU，采用 SPEC CPU2006 基准测试套件进行了性能评估，评估结果显示 CQEMU 能够进一步提升长期执行的程序的性能。

第六章 总结与展望

6.1 工作总结

QEMU 是一个开源的动态二进制翻译系统，它能够在多种主机平台上仿真多种处理器架构。TCG 是 QEMU 的核心翻译引擎，它采用动态二进制翻译技术对目标二进制代码进行实时的翻译，翻译和执行都以基本块为基本单位。由于 QEMU 采用单线程实现，且 TCG 核心翻译引擎在翻译过程中不做过多的优化，造成在生成的主机代码中，常常存在许多不必要的内存存取和寄存器移动指令，这些指令的执行势必和造成 QEMU 的执行开销。

本文以 QEMU 动态二进制翻译系统为基础，采用热点代码探测和动态优化模型对 QEMU 进行重新设计和实现，将 QEMU 单线程优化和执行变为多线程优化和执行，从而提升 QEMU 的执行效率。总揽全文，文章主要做了以下工作：

（1）利用代码插桩技术对每个基本块插入热点探测代码，以便在基本块代码执行时动态地探测出热路径块，消除因直接块链技术而导致的无法在主循环中进行热点搜集的难题。

（2）基于 NET 算法，结合 QEMU 自身的特性，设计和实现了适合 QEMU 的热路径搜集算法，该算法能够采用较少的代码实现精准的热路径搜集。

（3）实现对热路径缓存池的设计，热路径缓存池负责接收插桩代码搜集的热路径块，并向合并优化例程提供热路径块以便进行热点块的优化和合并。设计中充分考虑了缓存池的可伸缩性以及进程间互斥访问，保证缓存池在各线程之间正常的运转。

（4）实现对热路径的合并，文章通过缓存每个基本块的中间码，对热路径中对应基本块的中间码进行合并，最后合成一个单入口多出口的超级块。

（5）实现超级块的优化，主要包含块间优化和块内优化，块间优化主要消除各基本块之间的环境装载与存储操作，块内优化主要采用委托机制来消除超级块中的冗余指令，产生更加精简的主机代码。

（6）实现超级块重定位，当超级块翻译生成主机代码后，将热路径中的头块对应的第一条指令直接跳转至超级块所在的代码区，以便当再次执行到热路径时，直接跳转到超级块中执行，达到文章设计的目的。

在文章的最后，还对多线程热点代码探测和动态优化模型的 QEMU 进行了性能评估，并通过实验详细测试了委托机制对 QEMU 的性能影响。实验结果表明，委托机制能够有效改善代码地质量，多线程热点探测和动态优化模型能够进一步提升长期运行的程序的性能。

6.2 未来展望

基于动态二进制翻译技术的优化技术研究已经得到了半个世纪的发展，目前仍然是一个非常活跃的研究方向。同样，国内外学者对 QEMU 动态优化技术的研究也逐步深入，并以此带动 QEMU 动态优化技术的发展。随着多核处理器的问世以及多线程技术的发展，将动态二进制翻译技术与多线程结合其实是一个研究热点，而对代码的优化算法也不断得到提升。例如，LLVM 就是一个专门针对代码优化的优化框架，它能够利用多种优化策略来提升代码的质量。

文章对 QEMU 利用了多线程热点代码探测和动态优化，但优化方法略显简单，仅采用了委托机制来实现块内的代码优化，其仅能消除重复的内存存取操作以及寄存器移动操作，而不能实现其它方面的优化，因此具有一定的局限性。在翻译的主机代码中，可能还存在许多其它的冗余代码，消除这些冗余代码需要专门的优化策略来实现。

因此，文章的下一步工作将在当前优化框架的基础上，利用更多的优化算法来进一步提升主机代码的质量。另外，当前模型保留了基本块的中间码，每个基本块中间码都需要保存到内存，以便后期进行合并优化，基本块中间码的存放需要占用大量的内存空间，对应一些大型的应用程序，可能会造成内存使用不足的情况。因此，下一步工作将对这方面进行改进，把基本块对应的中间码存放于特定格式的文件中，以此减少内存的耗费。

参考文献

[1] M. Probst. Dynamic binary translation[C]. In Proceedings of the UKUUG Linux Developers' Conference, Bristol, United Kingdom, 2002

[2] Altman ER. Ebcioglu, K. Gschwind, et al. Advances and future challenges in binary Translation and optimization[C]. In Proceedings of the IEEE, IEEE, USA, Nov 2001, 89(11): 1710-22

[3] A. Chernoff, M. Herdeg, R. Hookway, et al. FX!32: A profile-directed binary translator[J]. IEEE Micro, 1998, 18(2):56–64

[4] B. Leonid, D. Tevi, E. Orn, et al. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium based systems[C]. Proceedings of the 36th Annual IEEE/ACM International Symposium on Micro architecture (MICRO-36), IEEE-CS Press, 2009

[5] C. Cifuentes, M. Van EmmeriK. UQBT: Adaptable Binary Translation at Low Cost[J]. Computer, IEEE Computer Society Press, March 2000, 33(3):60-66

[6] C. Cifuentes, V. Malhotra. Binary Translation: Static, Dynamic, Retargetable[C]. Proceedings International Conference on Software Maintenance, Monterey, CA, IEEE-CS Press, Nov 1996, 340-349

[7] C. Cifuentes, M. Van Emmerik, D. Ung, et al. Preliminary Experiences with the Use of the UQBT Binary Translation Framework[C]. Proceedings of the Workshop on Binary Translation, Newport Beach, Oct 16, 1999. Technical Committee on Computer Architecture Newsletter, IEEE-CS Press, Dec 1999, 12-22

[8] D. Ung, C. Cifuentes. Machine-Adaptable Dynamic Binary Translation[C], Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, Boston, USA, ACM Press, Jan 2000, 30-40

[9] D. Ung, C. Cifuentes. Optimizing Hot Paths in a Dynamic Binary Translator[C], Second Workshop on Binary Translation, Oct 19, 2000

[10] F. Bellard. QEMU, a fast and portable dynamic translator [C]. USENIX Association Proceedings of the FREENIX/Open Source Track, 2005, 41-46

[11] A. Jeffery. Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU[D]. Master’s thesis, University of Adelaide, Australia, 2009

[12] V. Chipounov, G. Candea. Dynamically Translating x86 to LLVM using QEMU[R]. Technical report, 2010

[13] Y. Hu, H. Jin, Z. Yu, et al. An Optimization Approach for QEMU[C]. 2009 1st International Conference on Information Science and Engineering (ICISE 2009), 2009, 129-132

[14] J. H. Ding, Y. C. Chung, P. C. Chang, et al. PQEMU: A parallel system emulator based on QEMU[R]. In 1st International QEMU Users Forum, 2011

[15] Z. Wang, R. Liu, Y. Chen, et al. COREMU: a scalable and portable parallel full-system emulator[R]. In Proc.PPoPP, 2011

[16] 黄英兰, 杨晋兴, 钟珊. 二进制翻译系统 BATSUP 中的动态翻译器的设计与实现[J]. 航空计算技术, 2005, 35(3):50-53

[17] 白童心, 冯晓兵, 武成岗等. 优化动态二进制翻译器DigitalBridge[J]. 计算机工程, 2005, 31(10):103-105

[18] Skyeye. http://www.skyeye.org/index.shtml

[19] C. Lattner, V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In International Symposium on Code Generation and Optimization, 2004, 75–88

[20] V. Aho, S. Ravi, D. Jeffrey. Compilers: Principles, Techniques, and Tools[J]. Addison-Wesley, 1986

[21] M. Probst, A. Krall, B. Scholz. Register liveness analysis for optimizing dynamic binary translation[C]. Ninth Working Conference on Reverse Engineering, Proceedings, 2002, 35-44

[22] M. Poletto, V. Sarkar. Linear scan register allocation[C]. ACM Transactions on Programming Languages and Systems, 1999

[23] 宋 强, 陈香兰, 陈华平. 动态二进制翻译器QEMU中冗余指令消除技术研究[J]. 计算机应用与软件, 2012, 29(5):67-69

[24] 王丽一, 文延华. 动态二进制翻译中的冗余load删除优化技术[J]. 计算机应用与软件, 2008, 25(6):40-43

[25] 郑红阳. 改进qemu的多模式指令解码研究[D]. 湖北: 华中科学技术大学, 1-3

[26] C. Cifuentes, V. Malhotra. Binary Translation: Static, Dynamic, Retargetable?[C]. Proceedings International Conference on Software Maintenance, Monterey, CA, Nov 4-8 1996, IEEE-CS Press, 340-349

[27] C. Howard, H. W. Chung, L. J. Wei, et al. Dynamic trace selection using performance monitoring hardware sampling[C] Proceedings of the international symposium on Code generation and optimization, feedback-directed and runtime optimization, San Francisco, California, 2003, 79-90

[28] D. Ung, C. Cifuentes. Machine-Adaptable Dynamic Binary Translation[C]. Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, 2000, 41-51

[29] C. Cifuentes and D. Ung. Walkabout - a retargetable dynamic binary translation framework[R]. Tec- hnical Report TR-2002-106, Sun Microsytems Laboratories, Palo Alto, CA 94303, 2002

[30] K. Scott, A. J. Davidson. Strata: A software dynamic translation infrastructure[C]. In IEEE Workshop on Binary Translation, 2001

[31] B. S. Yang, S. M. Moon, S. Park, et al. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation[C]. In International Conference on Parallel Architectures and Compilation Techniques, 1999

[32] G. Michael, A. Erik. Inherently Lower Complexity Architectures using Dynamic Optimization[C]. Proc. Workshop on Complexity Effective Design in conjunction with ISCA-2002, Anchorage, AK, 2002

[33] G. Michael. Dynamic and Transparent Binary Translation[J]. Computer, IEEE Computer Society Press, March 2000, 33(3):54-59

[34] R. Altman. Welcome to the Opportunities of Binary Translation[J]. Computer, IEEE Computer Society Press, March 2000, 33(3): 40-45

[35] Y. N. Srikant, P. Shankar. The Compiler Design Handbook, Optimizations and Machine Code Generation[M]. Chapter 19, CRC PRESS

[36] Alexander Klaiber. The Technology behind Crusoe Processor[R]. Transmeta technology report, Jan 2000, 3-12

[37] ER. Altman. Advances and future challenges in binary translation and optimization[M]. In Proceedings of the IEEE, Nov 2001, 89(11):1710-22, Publisher: IEEE, USA

[38] Z. Cindy, T. Carol. PA-RISC to IA-64: Transparent Execution, No Recompilation [J]. IEEE Computer Society Press, March 2000, 33(3) :47-52

[39] M. Probst. Fast machine-adaptable dynamic binary translation[C]. In Proceedings of the Workshop on Binary Translation 2001, September 2001

[40] http://www.transitives.corn/pmssroom.htm#brief

致谢

毕业季，难离舍。

三年研究生生涯，晃眼而过。在这三年里，我有过充实，有过感动，有过幸福，有过快乐！感谢你们，我在电子科技大学遇到的每一位老师、同学以及朋友！谢谢你们陪我一路走来，对我的鼓励和关怀！再次，我向你们表一声真挚的感谢！

在这三年的研究生学习和科研中，我要特别感谢我的导师杨国武教授，谢谢您给予我的悉心指导，让我在学习生涯中收获了很多丰富而又宝贵的知识，让我明白了认真做事，诚挚做人的道理；尝到了锲而不舍、孜孜不倦的回报。感谢您在这三年中对我以及整个团队付出的辛勤培育，无微不至的关怀！

我还要感谢郭文生老师、詹瑾瑜老师，以及张秀平师兄、李坤师兄、李婷师姐，谢谢您们在项目中给予我的指导和帮助，是您们的帮助让我及团队能够快速融入项目，学到了许多技术和专业知识。同时，我要感谢项目组成员邵院华、徐琳、赵莹德，感谢你们在项目中对我的支持和鼓励。感谢我的身边的每一位朋友，特别是王志才、杨刚、罗庆斌、林丽秀，有你们的一路陪伴，让我整个研究生生活变得充实，丰富多彩。

最后，由衷感谢为评论本论文的各位老师，您们辛苦了！

攻硕期间取得的研究成果

[1] **张世宜**, 杨国武, 邵院华. 一种 QEMU 仿真器 TCG 动态翻译框架的优化方法. 电子科技大学计算机科学与工程学院硕士生学术论坛(嵌入式系统分论坛). 2012 年 12 月

[2] 邵院华, 杨国武, **张世宜**. QEMU 动态二进制翻译系统中寄存器分配技术的研究与实现. 电子科技大学计算机科学与工程学院硕士生学术论坛(嵌入式系统分论坛). 2012 年 12 月

[3] 数字电视嵌入式软件平台及产业化. 科技部核高基项目（2009ZX01039）, 2009

[4] Android 模拟器的定制. 电子科技大学计算机科学与工程学院嵌入式实时计算实验室科研项目, 2011 年 9 月

[5] 基于 QEMU 的热点代码探测和优化技术的实现,. 电子科技大学计算机科学与工程学院银杏黄创新资金项目, 2012 年 9 月

[6] QEMU 仿真器的研究与优化. 电子科技大学计算机科学与工程学院嵌入式实时计算实验室科研项目, 2012 年 3 月