中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

指令流分析制导的动态二进制翻译优化技术

作者姓名：胡起

指导教师：张福新 正高级工程师 中国科学院计算技术研究所

学位类别：工学硕士

学科专业：计算机系统结构

培养单位：中国科学院计算技术研究所

2023 年 6 月

Optimization of Dynamic Binary Translation Directed by Instruction Flow Analysis

A thesis submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Science in Engineering

in Computer System Architecture

by

Qi HU

Supervisor: Professor Fuxin ZHANG

Institute of Computing Technology, Chinese Academy of Sciences

June, 2023

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

构建新型指令架构的一个重要方面就是建立软件生态系统，采用二进制翻译技术可以加速软件生态的建立，解决闭源软件无法移植等困难。其中，二进制翻译的优劣对程序的性能有重要的影响，它不仅关乎用户的体验，还会影响程序的正常运行。因此，二进制翻译优化是解决这些问题的关键方案。

本文在龙芯二进制翻译器 LATX（Loongson Architecture Translator from X86）基础上，提出指令流分析制导的二进制翻译优化方案 IFADO（Instruction Flow Analysis Directing Optimization）。该方法在反汇编阶段进行指令流分析，在翻译阶段根据分析结果进行相应的优化。另外，通过在优化框架内加入性能分析系统，分析出翻译器存在的 EFLAGS 运算性能较低，翻译后指令语义冗余，以及 SSE 标量指令高位值保留操作冗余这三个问题。为此提出了三种优化方案：

1. EFLAGS 延迟计算优化会通过分析基本块内的指令，根据每条指令生成和使用 EFLAGS 的情况，删除会被后续 EFLAGS 计算结果覆盖的 EFLAGS 计算指令，从而减少了基本块内冗余 EFLAGS 计算指令的产生。此优化方法成功消除了基本块内大量的 EFLAGS 计算指令，使得 SPEC CPU2000 性能提升超过 6%。

2. 为了解决 EFLAGS 延迟计算优化无法彻底消除基本块内部以及基本块结尾的 EFLAGS 计算指令的问题，本文提出了一种“反馈式”指令语义化翻译方案。该方案通过对多条指令进行语义翻译，实现了指令间 EFLAGS 的“计算-使用-计算”的优化消除。同时，“反馈式”优化能够利用后续基本块的反馈信息，对已翻译基本块进行持续优化，消除基本块结尾的 EFLAGS 计算。SPEC CPU2000 测试结果显示该优化性能提升超过 3%。

3. 针对 SSE 标量运算指令高位冗余的“保存-恢复”操作，本文提出了 SSE 标量指令高位运算消除优化方法（Scalar Higher Bits Remove, SHBR）。该算法通过对基本块进行分类，并根据基本块类型有针对性地消除其内部的“保存-恢复”操作。同时，在基本块链接时处理不同类型基本块链接的情况，确保了优化后的正确性。SHBR 算法使 SPEC CPU2000 浮点性能提高超过 9%。

关键词：动态二进制翻译，二进制翻译优化，指令流分析，龙架构

Abstract

An important aspect of building a new instruction set is to establish a software ecosystem. Adopting binary translation technology can accelerate the establishment of the software ecosystem and overcome difficulties such as the inability to port closed-source software. The quality of binary translator achievement has a significant impact on the performance of the translated program, which not only affects user experience but also the program execution behavior. Therefore, binary translation optimization is a key solution to address these issues.

Our research proposes an instruction flow analysis-directed binary translation optimization, which is called IFADO (Instruction Flow Analysis Directing Optimization), based on the Loongson binary translator LATX (Loongson Architecture Translator from X86). This approach conducts instruction flow analysis during the disassembly phase, obtaining and recording optimization information, and implements corresponding optimizations during the translation phase according to the analysis results. Furthermore, to identify performance bottlenecks in the translator, a performance analysis system is added into this framework. Our research discovers three main issues in the LATX translator through analysis: low performance in EFLAGS operation, redundant instruction semantics after translation, and useless VRs (Vector Registers) high-bits preservation operations in SSE scalar instruction translation. Based on these findings, our research proposes three optimization strategies:

1. The EFLAGS lazy calculation optimization analyzes instructions in each TB (Translation Block). Then, based on the generation and usage of EFLAGS for each instruction, the EFLAGS calculation instructions that will be overwritten by subsequent EFLAGS computation results can be removed. This method reduces the generation of redundant EFLAGS calculation instructions in TBs. The optimization successfully eliminates a substantial number of EFLAGS instructions, resulting in a performance improvement of over 6% for SPEC CPU2000 benchmarks.

2. To address the issue of EFLAGS calculation instructions that cannot be com-

pletely eliminated within or at the end of TB, our research proposes a ”feedback” semantic translation scheme. This approach optimizes the ”compute-use-compute” process of EFLAGS between instructions by semantically translating multiple instructions. Additionally, the ”feedback” optimization utilizes feedback information from subsequent TB which can continuously optimize already translated TB and eliminate EFLAGS computation at the end of TB. Results from SPEC CPU2000 testing show a performance improvement of over 3% with this optimization.

3. In response to the redundant ”save-restore” operations that preserve higher bits of VRs in SSE scalar instructions, our research proposes an optimization called SHBR (Scalar Higher Bits Remove). This algorithm classifies TBs and selectively eliminates the internal ”save-restore” operations according to their type. In addition, in the case of different types of basic block links, the correctness of the optimization needs to be handled during the basic block links. The SHBR algorithm improves the floating-point performance of SPEC CPU2000 benchmarks by more than 9%.

Keywords: Dynamic Binary Translation, Binary Translation Optimization, Control Flow Analysis, LoongArch

目 录

第 1 章 引言 ····· 1
1.1 二进制翻译概述 ····· 2
1.1.1 解释执行 ····· 2
1.1.2 静态二进制翻译 ····· 3
1.1.3 动态二进制翻译 ····· 4
1.1.4 混合式二进制翻译 ····· 4
1.2 二进制翻译性能开销 ····· 5
1.2.1 翻译过程的开销 ····· 5
1.2.2 指令膨胀带来的开销 ····· 6
1.2.3 间接跳转带来的开销 ····· 6
1.2.4 其他开销 ····· 7
1.3 本文的主要工作及贡献 ····· 7
1.4 本文组织结构 ····· 8
第 2 章 相关工作 ····· 11
2.1 二进制翻译技术 ····· 11
2.2 经典二进制翻译系统的优化技术 ····· 11
2.2.1 FX!32 ····· 12
2.2.2 Transmeta Crusoe ····· 13
2.2.3 MAMBO-X64 与 JTLT ····· 14
2.2.4 Dynamo ····· 15
2.2.5 HQEMU ····· 15
2.2.6 Rosetta2 ····· 16
2.2.7 QEMU ····· 16
2.3 龙芯二进制翻译器 LATX ····· 18
2.4 本章小结 ····· 19
第 3 章 指令流分析制导的优化设计和实现 ····· 21
3.1 整体设计 ····· 21
3.1.1 分析探针 ····· 22
3.1.2 优化器 ····· 22
3.1.3 性能分析器 ····· 24
3.2 性能分析框架 ····· 24

3.2.1 性能分析框架设计 ····· 24
3.2.2 性能分析数据 ····· 25
3.3 优化方向 ····· 30
3.4 本章小结 ····· 30
第 4 章 EFLAGS 延迟计算 ····· 33
4.1 概述 ····· 33
4.2 可行性分析 ····· 33
4.3 EFLAGS 延迟计算框架 ····· 38
4.3.1 指令流分析扫描 ····· 39
4.3.2 EFLAGS 指令消除 ····· 41
4.4 性能分析 ····· 42
4.5 本章小结 ····· 44
第 5 章 “反馈式”指令语义化翻译 ····· 47
5.1 引言 ····· 47
5.2 整体架构 ····· 47
5.3 语义化翻译方案 ····· 48
5.3.1 指令序列搜索 ····· 48
5.3.2 语义化翻译 ····· 49
5.4 跨基本块反馈优化 ····· 51
5.4.1 消除式优化 ····· 51
5.4.2 链接式优化 ····· 52
5.5 自修改处理和恢复 ····· 53
5.6 性能分析 ····· 55
5.6.1 EFLAGS 消除分析 ····· 55
5.6.2 各优化阶段性能分析 ····· 57
5.6.3 关闭跳转优化后性能分析 ····· 58
5.6.4 性能分析总结 ····· 59
5.7 本章小结 ····· 59
第 6 章 SSE 标量指令高位运算消除 ····· 61
6.1 引言 ····· 61
6.2 整体架构 ····· 61
6.3 基本块分类 ····· 62
6.4 向量寄存器高位消除优化 ····· 63
6.4.1 向量寄存器高位信息计算 ····· 64

6.4.2 基本块类型识别 ····· 66
6.4.3 高位“保存-恢复”操作消除 ····· 67
6.4.4 基本块链接 ····· 68
6.4.5 链接时优化 ····· 70
6.5 性能分析 ····· 70
6.5.1 整体性能分析 ····· 70
6.5.2 基本块切换和 SHBR 消除分析 ····· 71
6.5.3 链接时优化分析 ····· 74
6.6 本章小结 ····· 75
第 7 章 总结与展望 ····· 77
参考文献 ····· 81
致谢 ····· 85
作者简历及攻读学位期间发表的学术论文与研究成果 ····· 87

图形列表

1.1 静态二进制翻译器工作流程 ····· 3
1.2 动态二进制翻译器工作流程 ····· 4
1.3 每 1 万条指令的间接跳转指令数 ····· 7

2.1 FX!32 架构图 ····· 12
2.2 Transmeta Crusoe 架构图 ····· 13
2.3 Dynamo 核心算法图 ····· 15
2.4 HQEMU 架构图 ····· 16
2.5 QEMU 基本块链接示意图 ····· 17
2.6 QEMU EFLAGS 延迟计算优化 ····· 18

3.1 指令流分析制导优化框架构图 ····· 21
3.2 单指令优化器架构 ····· 22
3.3 跨指令优化器架构 ····· 23
3.4 基本块优化器架构 ····· 23
3.5 性能分析框架 ····· 24
3.6 指令的性能测试方案 ····· 27
3.7 SSE 向量指令的分类 ····· 29
3.8 ADDSD 指令“保存-恢复”操作 ····· 29

4.1 EFLAGS 结构图 ····· 33
4.2 TEST 指令 EFLAGS 模拟翻译实例 ····· 34
4.3 SPEC CPU2000 基本块翻译实例 ····· 35
4.4 基本块 EFLAGS 使用示意图 ····· 36
4.5 EFLAGS 延迟计算框架 ····· 39
4.6 EFLAGS 延迟计算优化指令流分析过程 ····· 40
4.7 EFLAGS 指令消除 ····· 41
4.8 EFLAGS 延迟计算后 CoreMark 性能对比 ····· 42
4.9 EFLAGS 延迟计算优化的 SPEC CPU2000 性能数据 ····· 43
4.10 SPEC CPU2000 各子项 EFLAGS 消除对比 ····· 44

5.1 “反馈式”语义化翻译框架 ····· 48
5.2 指令序列搜索框架 ····· 48
5.3 IDIV 指令翻译对比 ····· 50

5.4 基本块尾语义化翻译实例 ····· 50
5.5 消除式跨基本块反馈优化 ····· 52
5.6 链接式跨基本块反馈优化 ····· 53
5.7 自修改后出现的 EFLAGS 丢失的问题 ····· 54
5.8 优化后的自修改处理和恢复 ····· 55
5.9 “反馈式”语义化翻译 SPEC CPU2000 性能 ····· 56
5.10 语义化翻译 EFLAGS 消除对比 ····· 56
5.11 语义化翻译各阶段性能对比 ····· 57
5.12 关闭跳转优化后语义化翻译性能对比 ····· 58

6.1 SHBR 整体架构 ····· 62
6.2 SHBR 算法对基本块的分类 ····· 63
6.3 向量寄存器高位信息计算实例 ····· 64
6.4 基本块类型识别实例 ····· 67
6.5 基本块链接处理示意图 ····· 68
6.6 SHBR 优化 SPEC CPU2000 性能结果 ····· 71
6.7 不同类型基本块在 SPEC CPU2000 中的切换频率 ····· 72
6.8 SPEC CPU2000 中不同类型基本块切换使用指令类型 ····· 72
6.9 不同类型基本块在 SPEC CPU2000 中的比例 ····· 73
6.10 SPEC CPU2000 中 SHBR 消除比例 ····· 73
6.11 不同类型基本块链接时优化 SPEC CPU2000 性能结果 ····· 75

7.1 IFADO 优化各阶段 SPEC CPU2000 性能结果[1] ····· 78

表格列表

1.1 Arch Linux 中 LoongArch 架构软件支持清单 ····· 2

2.1 已经完成的单指令优化性能比较 ····· 19

3.1 gzip 中指令使用频率 ····· 26
3.2 EFLAGS 运算指令性能结果 ····· 28

4.1 SPEC CPU2000 进行 EFLAGS 延迟计算优化前性能评估结果 ····· 37
4.2 EFLAGS 延迟计算后 CoreMark 性能数据 ····· 43

第 1 章　引言

敏捷芯片开发、开源指令集等技术的出现，大大加快了计算机体系结构的发展，Hennessy 和 Patterson 更是提出“计算机体系结构迎来了新的黄金时代”的观点[1]。但新指令集的产生以及体系结构的高速发展也迎来新的挑战，其中最主要的是软件生态问题。新指令集的产生会导致原有的二进制程序无法在新的平台运行，该问题也极大地阻碍了新平台的生态建设。

2021 年，龙芯中科公司推出了全新的自主指令集龙架构（LoongArch），以及首款使用该架构的处理器 3A5000[2]。与前一代处理器 3A4000 相比，3A5000 处理器同频性能平均提升超过 20%。同时，使用龙架构的 SPEC CPU2006 定点程序的平均动态指令数也比 MIPS 的定点程序约少了 12%，充分展现了龙架构的优势[3]。更为重要的是，龙架构也完成了我国芯片设计中使用自主指令集的最后一块版图，从架构设计到指令功能、应用二进制接口（Application Binary Interface, ABI）标准等都实现了完全自主，不必依赖国外授权。然而，由于采用了新的指令架构，发布时间较短，原生的 LoongArch 应用数量相对较少，影响了用户的使用体验。因此，如何将 X86 平台的程序迁移至龙芯平台，使其能够运行更多应用程序成为当前亟待解决的问题[4]。

解决上述问题主要有两种方案：第一种是使用迁移的方式，通过修改原有程序的代码，在新的架构上重新编译。这种迁移方式最为直接，提供的原生软件不仅拥有可靠的使用体验，还可以根据 LoongArch 架构进行深度优化，获得优异的软件性能。但该方案也存在着诸多问题，一方面，虽然编译后的程序可以在新的平台上直接运行，但该方法需要获取源代码，对于那些遗留软件，以及商业级闭源软件，都无法通过迁移的方式完成。另一方面，迁移的方式需要对每一个软件进行修改和编译，以现有的 Arch Linux 仓库为例，如表 1.1 所示。其中关键库就有 257 个，如果通过人工手动迁移，需要投入巨大的时间，成本较高。

第二种解决方案是使用二进制翻译技术，在 LoongArch 平台上运行 X86 的应用程序。二进制翻译技术提供了自动将不同架构的应用程序转换成目标架构的解决方案，是一种极为重要的跨指令集兼容技术，打破了指令集之间的“壁垒”。二进制翻译允许在不重新编译原程序的情况下，在一个新的指令集上运行

表 1.1 Arch Linux 中 LoongArch 架构软件支持清单

Table 1.1 List of software support for LoongArch architecture in Arch Linux

仓库	二进制包数量	完成度
core	257	100%
extra	2651	88%
community	5691	41%

其他架构的应用程序。

使用二进制翻译的最大优点在于，可以无差别地将无法获得源代码的程序和闭源的商业软件等翻译并在 LoongArch 架构上运行；同时，在二进制翻译的帮助下，可以减轻人工迁移的负担，加快新架构生态软件的建设。但是，二进制翻译也存在着不可避免的缺陷，其中最主要的就是翻译后软件的性能问题。因此，本部分将会在 1.1 中详细介绍二进制翻译器，并且在 1.2 中对二进制翻译的关键问题进行讨论。

1.1 二进制翻译概述

埃里克提出二进制翻译可以分成三大类，分别是解释执行、静态二进制翻译以及动态二进制翻译[5]。解释执行时会对每条指令进行实时的解释，不进行优化和缓存。解释执行过程对用户透明，开发过程也相对容易，但是性能低下。静态二进制翻译采用“离线”翻译方式，在程序不运行的情况下，对原二进制文件进行分析、翻译以及优化，但该方法对于自修改代码等较难处理；动态二进制翻译采用“在线”翻译方式，在程序执行到某个片段时才会对该片段进行翻译，可以有效的解决自修改代码、代码和数据区分困难等静态二进制翻译中的问题，所以主流的二进制翻译器大多采用动态翻译的方案，如 QEMU[6]。

1.1.1 解释执行

执行解释指的是根据原平台程序的二进制指令，按照每一条指令的语义，逐条指令的模拟处理器中各种状态（如寄存器的数据、处理器的状态等），与静态二进制翻译和动态二进制翻译不同，执行解释不会将原平台指令翻译成目标平台指令并运行，而是只对原平台指令进行解释，并模拟指令在处理器上的执行。

由于执行方式的不同，解释执行与静态和动态二进制翻译相比具有很大的性能差距。然而，在处理冷代码时，由于翻译执行的开销远大于解释执行的开销，因此解释执行会有自身的优势。此外，解释执行可以更准确地模拟每条指令的状态，对于不同架构中相差较大的指令，都可以做到精确的模拟。

1.1.2　静态二进制翻译

静态二进制翻译是采用离线翻译的方式，将原二进制文件转换成目标平台的二进制文件。在此过程中，原平台二进制程序不会被直接运行，而是通过数据流分析和其他技术，将指令逐条翻译并优化。如图 1.1 所示，原二进制文件会根据 ELF 头信息，并通过控制流分析进行反汇编操作。反汇编后的指令会逐条转换成目标平台的指令，并进行优化处理。转换后的指令会和新生成的目标平台 ELF 头信息组合在一起，构成翻译后的目标平台的二进制文件。由于静态二进制翻译可以提前完成翻译和优化，不会占用运行时间，因此拥有更多的优化空间。

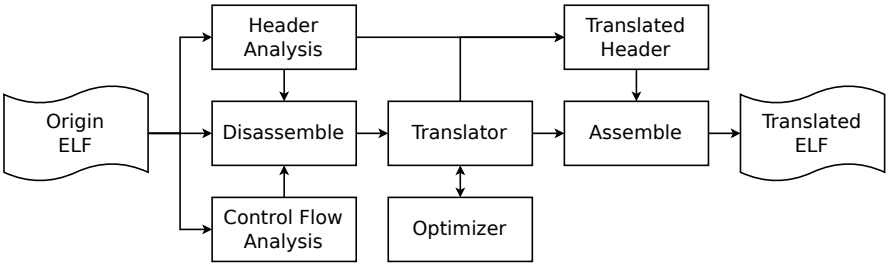

图 1.1 静态二进制翻译器工作流程

Figure 1.1 Static binary translator workflow

但静态二进制翻译存在诸多问题，对于复杂程序支持比较困难。例如，对于存在自修改以及即时编译技术（Just-In-Time, JIT）的程序，静态翻译无法及时将自修改后的代码翻译成目标平台代码，因此无法支持这些程序。

另外，静态二进制翻译对于间接跳转的处理较为困难。一般常用做法是通过识别可能的跳转目标位置，建立跳转映射表，实现间接跳转的地址转换。但由于间接跳转的识别不一定很准确，很可能会导致漏识别一些跳转位置而引发程序崩溃[7,8]。同时，因识别不准确而产生的无效跳转映射也会使程序膨胀过大，影响实际的使用。

此外，静态二进制翻译难以区分代码和数据，尤其是 X86 这类变长指令集，以及 ARM 和 RISC-V 中的压缩指令集[9]。虽然可以采用指令流分析的方案，从

入口处逐条指令分析，并根据跳转的目标，进行下一步的翻译。但是，对于那些采用代码混淆技术的程序，使用异常等改变控制流的情况，静态翻译就无法准确地区分出指令和数据。

1.1.3 动态二进制翻译

与静态二进制翻译不同，动态二进制翻译采用“边翻译边运行”的方式，将原平台架构的指令在运行时翻译成目标平台架构指令并运行。如图 1.2 所示，翻译器读取二进制文件后，通常会按照基本块的粒度执行翻译，并通过上下文切换的方式进入翻译后的代码执行。当执行完一段翻译后的代码之后，同样需要利用上下文切换退出到翻译模式，并对后续指令进行翻译，以此循环，直到程序执行结束。动态二进制翻译有效的解决了静态翻译过程中的一些问题，如对自修改代码的处理，可以采用页权限保护机制，让自修改代码触发例外，在翻译器的帮助下进行重新翻译。

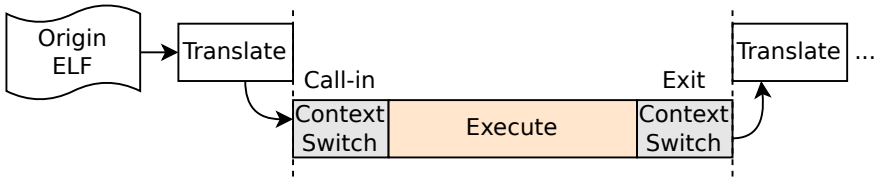

图 1.2 动态二进制翻译器工作流程

Figure 1.2 Dynamic binary translator workflow

然而，动态二进制翻译也存在一定缺陷，最主要的是会引入额外的翻译开销，这导致了动态二进制翻译只能加入轻量级优化，能进行的优化有限。在本文的 1.2 部分，我们将会对动态二进制翻译优化这一个关键性问题进行详细讨论。

1.1.4 混合式二进制翻译

动态与静态二进制翻译各有其局限性。为了更有效地解决这些问题，提出了混合式二进制翻译方法[10,11]。这种混合式二进制翻译大多采用动静结合的翻译方式，结合了动态和静态二进制翻译各自的优势。

执行前，混合式二进制翻译器会使用静态翻译方式，提前完成大多数指令的翻译工作，从而减少动态二进制翻译中额外的翻译开销[12]；同时，静态二进制翻译器可以加入更多的优化步骤，对翻译后的代码进行离线优化。在运行时，翻译器会使用动态翻译方式，解决静态翻译中难以解决复杂程序翻译的问题。

此外，为了应对冷代码启动时较大的开销问题，部分混合式二进制翻译器还会集成解释器模式[13]。对于执行次数较少的代码块，翻译器会采用解释执行的方法。仅当这部分代码块的运行次数达到设定的阈值时，才会切换至翻译模式，对该代码块进行翻译。

1.2　二进制翻译性能开销

二进制翻译是解决在新的指令平台上运行其他架构程序的一把利器，但却也面临着许多挑战。如不同架构存在不同的内存一致性模型，当在弱内存序的架构上翻译运行强内存序的程序时，需要进行额外的处理，带来额外的开销[14]。此外，架构本身的 ABI 不一致也会给翻译带来难题。如原程序平台的页大小小于目标平台使用的页大小，翻译器无法进行更细粒度的页管理，导致出现错误。

当然，二进制翻译中最关心的是翻译后程序性能。如果翻译后程序性能较差，不仅会影响用户的使用体验，还会影响程序执行的正确性。如需要翻译的程序存在超时机制模块，二进制翻译性能较差可能会引发程序的超时错误，使得程序无法正常运行。

在本部分，我们将探讨二进制翻译中各种开销的来源。1.2.1 部分将分析翻译过程中引入的开销；1.2.2 部分将探讨翻译后指令膨胀所带来的性能损失；1.2.3 部分将探讨间接跳转相关的开销；最后，1.2.4 部分将概述可能导致翻译后程序执行速度下降的其他因素。

1.2.1　翻译过程的开销

动态二进制翻译采用“边翻译边运行”的运行模式，其翻译的时间包含在程序运行时间内，所以翻译过程的性能会直接影响程序的性能。对于那些存在有大量冷代码的程序，翻译开销对性能的影响会变得尤为明显。另一方面，程序初次启动含有大量的冷代码，所以翻译开销也影响了程序的响应速度，对于那些实时程序来说并不友好。

为了解决这个问题，可以使用静态翻译器来辅助翻译和优化，将程序的大部分代码静态翻译，并存储在 AOT（Ahead-Of-Time）文件中，在程序运行时刻加载并运行。对于更加复杂的部分，如自修改代码，间接跳转等，则留给动态翻译部分处理，从而减少翻译开销[12]。尽管如此，对于大多数应用以及基准程序来说，翻译过程的开销还不到 2%[15,16]。因此，在评估二进制翻译的性能时，可以

不考虑翻译过程的开销。

1.2.2 指令膨胀带来的开销

由于不同架构的 ISA 指令语义不同，导致二进制翻译过程中指令数量的增加（指令膨胀），而这正是造成程序性能开销的主要原因，而指令之间的差异主要体现在多个方面。

首先，不同架构指令相同操作之间存在着细微的差异。例如 X86 中的 SSE 向量指令中标量运算指令，它们的操作结果仅改变向量寄存器的低位，而高位的数据保持不变，而 AArch（ARMv8）和 RISC-V 中类似操作的向量指令的结果高位通常被 0 扩展。因此，如果要使用 AArch 指令模拟 X86 指令，必须额外添加保存/恢复指令，以保留向量寄存器的高位数据。

其次，不同架构指令可以操作的数据类型也有所区别。例如，X86 中的 X87 浮点运算指令可以使用 80 位宽度的浮点寄存器，而其他的指令架构通常使用 64 位宽度的浮点寄存器。因此高效的翻译 80 位浮点运算也是一个巨大的挑战。

另外，在不同指令架构中，同一含义的指令所支持的立即数范围也不一定一致。例如，MIPS 架构中的加法指令支持 16 位立即数，但在 LoongArch 和 RISC-V 架构中，加法指令只支持 12 位立即数，这使得翻译器需要额外的指令来装载这些立即数。

当然，不同架构寄存器数量不同也会对程序性能有较大影响。当以拥有较少寄存器的指令集模拟拥有较多寄存器的指令集时，如在 X86 架构上翻译运行 AArch 架构程序，由于寄存器数量不足，必须将原程序中的寄存器数据存放到内存中，这会带来额外的内存访问，从而降低了程序的性能。

1.2.3 间接跳转带来的开销

在 SPEC CPU2000 中，间接跳转指令约占指令总数的 1.27%，如图 1.3 所示。虽然间接跳转占比不高，但如果不高效的处理这些指令，会导致多次的上下文切换，产生大量的寄存器保存恢复指令以及查找跳转到目标地址等操作，从而使每条间接跳转指令产生超过 1:100 的性能差距。因此，为了保证翻译后程序的高效运行，对间接跳转的优化是十分重要的。

为了优化间接跳转，通常采取查找表的方式。查找表在内存区域内建立了原地址（Source PC, SPC）和目标地址（Target PC, TPC）之间的映射关系，每次翻

译间接跳转时，通过对该查找表的索引，获取 SPC 到 TPC 的映射，以完成对应的转换[17]。

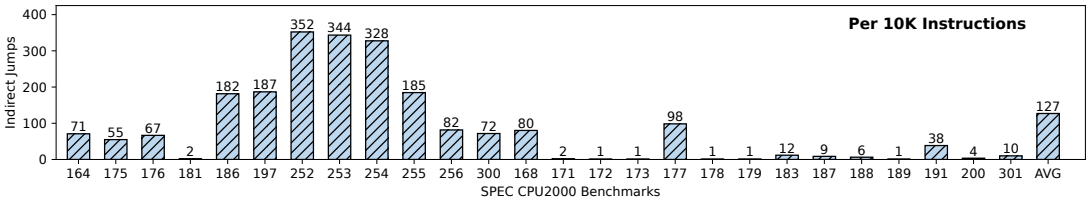

图 1.3 每 1 万条指令的间接跳转指令数

Figure 1.3 Number of indirect jumps per 10K instructions

1.2.4　其他开销

上下文切换、基本块链接和系统调用等因素也会影响翻译后程序的性能。上下文切换通过保存和恢复所有的寄存器值来实现，从而使翻译系统从翻译环境或执行环境切换到另一种环境，这样会导致大量的访存操作从而影响翻译后程序性能。但是，通过链接已翻译的基本块，大多数上下文切换操作可以被消除，从而显著降低上下文切换带来的性能损失。

不过，基本块链接也存在着缺陷。虽然基本块链接增加的直接跳转指令不会导致分支预测失败率升高，但其仍会给处理器的 BTB（Branch Target Buffer）带来不小的挑战，如果增加的跳转指令使 BTB 的容量不足，那处理器对分支指令处理性能将显著下降。为了解决这个问题，可以通过跟踪热点块，并将其合并成更大的基本块，从而消除相关的跳转指令[18]。

此外，系统调用或信号可能会引起翻译程序的性能问题，但它们在实际应用和基准测试程序中出现的频率很低，对系统整体性能的影响很小。

1.3　本文的主要工作及贡献

二进制翻译是解决新架构上应用程序缺失，扩展软件生态的关键方案。为了使应用程序能够在新架构上无缝运行，需要实现一个完善的二进制翻译器。因此，当前的二进制翻译器的核心都是使用动态二进制翻译的方法。有些翻译器为了降低翻译性能，也采用了静态翻译的方式，但最终翻译出来的程序仍需要动态端的辅助。此外，翻译后程序的效率是二进制翻译中的核心问题。所以本文将会以龙芯动态二进制翻译器 LATX（Loongson Architecture Translator from X86）为基础，

提出一套基于指令流分析制导的动态二进制翻译优化方案 IFADO（Instruction Flow Analysis Directing Optimization），以解决性能分析得出的 EFLAGS 计算、指令语义冗余和向量高位无效计算这三个性能瓶颈。

1. 在 EFLAGS 计算冗余消除上，本文提出了 EFLAGS 指令延迟计算（Lazy Calculation）的优化方案。通过分析基本块内各指令的 EFLAGS 产生和使用情况，得到后续 EFLAGS 结果是否被覆盖的信息，我们可以标记指令是否需要产生 EFLAGS 指令，以有效减少 EFLAGS 指令的生成。

2. 在指令语义冗余优化方面，本文提出了“反馈式”指令语义化翻译优化方案。该方法通过分析基本块内指令片段，获取可优化的指令序列，并按语义进行多对多的翻译。同时“反馈式”翻译方法还能利用后续基本块分析信息，通过后续基本块反馈的信息，不断优化基本块内的指令语义翻译。此外，该方案还记录并处理了原有语义与优化后指令语义之间的差异，以确保在出现特殊情况时，能够准确恢复优化前状态。

3. 在向量寄存器高位冗余计算方面，本文提出了标量计算指令高位消除算法（Scalar Higher Bits Remove, SHBR），对每个基本块进行数据流分析和指令类型分析，将其分成标量（Scalar）、向量（Packed）和普通（None）三种类型，根据每种基本块的类型，对向量寄存器高位的计算进行针对性的优化。

1.4 本文组织结构

本文的第一章介绍了二进制翻译在解决软件生态问题方面的重要作用，以及相关的概念和分类，探讨了二进制翻译的关键问题，包括影响翻译后程序性能的几个因素，并概述了本文的研究内容和研究思路。

第二章介绍了二进制翻译技术以及二进制翻译优化的相关工作，选取了多个开源或商用二进制翻译器。他们采用了包括代码优化、跳转优化、热点块分析优化、LLVM 辅助优化、硬件辅助优化等手段，实现了较高效性能的优化效果。最后本章介绍了龙芯二进制翻译器 LATX 及其已经使用的部分优化技术。我们将基于 LATX 进行优化探索工作。

第三章讨论了指令流分析制导的优化（IFADO）整体框架，包括分析探针、优化器和性能分析器三个部分，并对它们的设计进行了详细描述。此外，本章还提供了性能分析器获取的性能分析数据，并根据性能分析数据，提出了 EFLAGS

延迟计算、“反馈式”指令语义化翻译和 SSE 标量指令高位运算消除这三个优化方向。

第四章到第六章详细介绍了三个优化的设计与实现。第四章首先对 EFLAGS 延迟计算的可行性进行了分析，探讨了优化收益与指令动态消除数量之间的关系，并得出了预估收益的计算结果。接着，本章介绍了 EFLAGS 延迟计算的框架，并实现了基于指令流分析扫描的 EFLAGS 计算优化，在翻译阶段实现了优化动作。最后，本章对 EFLAGS 延迟计算优化进行了性能分析和评估，并分析了未能消除 EFLAGS 指令的原因。

第五章探讨了“反馈式”指令语义化翻译的优化实现。本章首先阐述了语义化翻译的实现方法，接着介绍了“反馈式”跨基本块的优化策略，同时为了避免自修改引发的问题，实现了自修改处理与恢复的处理流程。本章最后给出了该优化的性能分析，对于出现部分子项性能低下，以及性能提升不明显等异常情况，进行相应的实验，并给出了合理的解释。

第六章介绍了 SSE 标量指令高位运算消除优化方法。本章首先介绍了基本块分类和基本块类型识别算法，接着给出了各类型基本块优化方案，并介绍了不同类型基本块链接时刻的特殊处理方案。最后，本章对 SSE 标量指令高位运算消除优化方法进行了性能分析和评估，详细探讨了各程序性能变化的原因。

第七章对全文研究工作进行了总结，概括了各种优化方法的优缺点，并对优化过程中出现的问题进行了探讨，给出了后续优化的参考方向。

第 2 章　相关工作

2.1　二进制翻译技术

自 1987 年 Hewlett-Packard 公司推出最早的二进制翻译系统以来，国内外不断出现使用二进制翻译技术来解决程序跨平台的问题的方案。美国数字设备公司研发了 FX!32 翻译器，实现了在 Alpha 平台上动态翻译运行 X86 程序，其 SPEC CPU2000 定点测试性能可达到本地编译执行效率的 60%[13]；IBM 公司自主研制的 DAISY 二进制翻译系统可以在 BOA 架构与 PowerPC 架构处理器的异构系统上运行 PowerPC 架构程序[19,20]；Queenland 大学开发了 UQBT 二进制翻译器，采用静态翻译方式的，可以支持多种原平台和目的平台[21]。

HP 公司[18] 开发的 Dynamo 是一款纯软件实现的二进制翻译和优化器，可以实现本地程序的运行性能收集和优化，经过翻译后的程序平均性能提高了 9%。随着时间的推移，Dynamo 发展成为 DynamoRIO，支持了 ARM 和 X86 架构的二进制翻译和插桩分析[22]。

QEMU 是则是一款开源二进制翻译器，支持 KVM 虚拟化和跨架构的程序运行。使用动态翻译的方式支持系统级和用户级的虚拟化[6]，但是由于其兼顾通用性而牺牲优化，使得其执行效率仅为原生执行的 10%-20%。

近几年各种架构的不断兴起，各大厂商也开始使用二进制翻译技术对原有的软件生态进行迁移。苹果公司为了迁移其 X86 的软件生态，开发了翻译器 Rosseta 2，它采用静态与动态相结合的方式，将 x86 平台的程序翻译到 macOS 操作系统的 ARM 平台上，为 macOS 软件生态带来了更多可能性。龙芯公司针对新的自主指令集龙架构（LoongArch），提出了 LATX 动态二进制翻译器，实现 X86 程序到 LoongArch 程序的翻译运行[3]，翻译后 SPEC CPU2000 效率可达到本地编译执行效率的 50%。

2.2　经典二进制翻译系统的优化技术

通过二进制翻译后的程序常存在着运行效率低，实际应用难以使用等问题，如何提高二进制翻译器翻译后代码执行效率成为了现阶段的研究热点。为此提出了多种二进制翻译的优化技术，如指令优化、缓存管理、并行翻译、跳转优

化、硬件支持、代码生成等。本节将介绍几种经典的二进制翻译器所采用的优化技术。

2.2.1 FX!32

1996 年，Digital 公司推出了 FX!32，这是一款将 32 位 X86 程序移植到 Alpha 架构的二进制翻译器[13,23]，其架构图如图 2.1 所示。

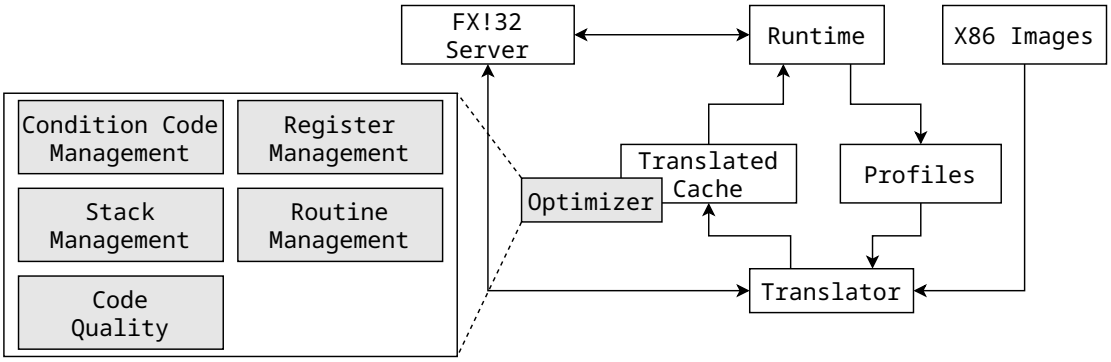

图 2.1 FX!32 架构图

Figure 2.1 Architecture of FX!32

FX!32 在首次执行 X86 程序时会记录程序的运行情况，并生成一个概要文件。根据该概要文件，在后续运行过程中可以优化原有的 Alpha 代码。

1. 条件代码（EFLAGS）优化。由于 X86 会产生 EFLAGS 信息，但这些信息并不是全部被使用，所以 FX!32 根据全局数据流，只计算使用的条件代码。

2. 寄存器映射优化。X86 使用的寄存器少于 Alpha 架构的寄存器，所以可以将 X86 的寄存器映射到 Alpha 架构寄存器上，以减少指令生成。

3. 堆栈优化。由于 X86 常使用堆栈保存临时数据，所以可以根据 X86 堆栈的特性，将堆栈指针上部分的区域数据认为成无效数据，并根据此进行指令的消除，同时减少堆栈指针的更新速度。

4. 运行相关优化。针对存在的 CALL/RET 的操作，其返回地址会存储在栈上，但此地址为原平台的地址，无法直接使用。FX!32 采用影子栈的方式，每次 CALL 时刻将返回地址对应的目标平台地址压入影子栈中，在返回时刻查找影子栈进行匹配，降低了地址查找和转换的开销。

5. 代码优化。FX!32 针对翻译后代码进行优化，实现了死代码消除、常量传播、公共子表达式消除、寄存器重命名、全局寄存器分配、指令调度以及窥视孔优化等方案。

然而，这 FX!32 自身也有一些缺点[24]，如：

1. 非对齐访问。由于 Alpha 处理器的普通访存指令无法支持非对齐访问，出现非对齐访问的时候，会陷入内核并通过仿真的方式来处理这类访问。FX!32 对此并没有进行特别的优化，这可能会因为多次触发非对齐访问而大幅度的影响性能。

2. 模拟执行的。有些 X86 的指令无法使用翻译的方式进行，如 X87 的 80 位更高精度的浮点计算，这些指令会按照模拟执行的方案进行计算。然而模拟执行的性能远远低于翻译执行，即使需要模拟执行的指令较少，但运行次数过多也会对性能造成很大的影响。

3. Cache。由于经过翻译，其指令膨胀会影响 Cache 的命中率，同时由于翻译后的代码并不是按照原程序一样，这也会导致原程序局部性较好的指令片段被打乱，导致 Cache 的局部性变差。

4. 分支预测。与 Cache 的问题类似，由于经过翻译，其存在多次的地址映射，这样会影响处理器分支预测的能力，导致分支性能变差。

2.2.2　Transmeta Crusoe

Transmeta Crusoe 将 VLIW 架构应用于全系统二进制翻译，在硬件辅助下通过二进制翻译，实现了一个 X86 兼容环境[25,26]，其架构图如图 2.2 所示。

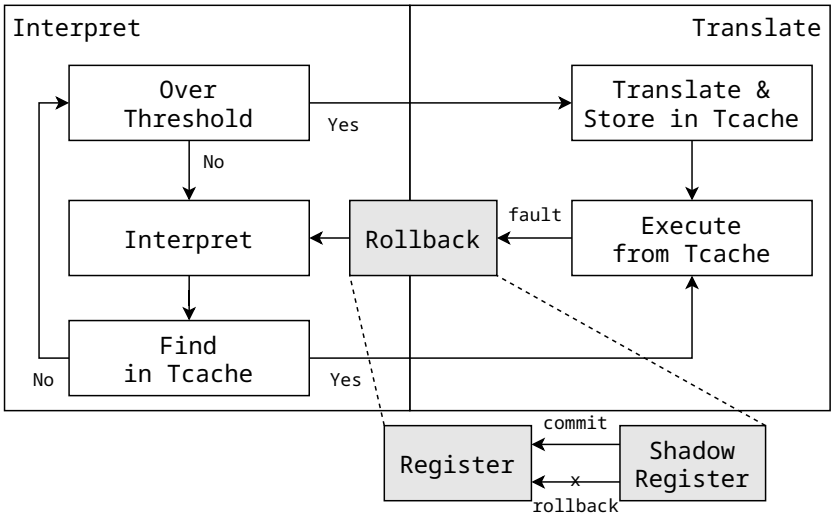

图 2.2 Transmeta Crusoe 架构图

Figure 2.2 Architecture of Transmeta Crusoe

Crusoe 将翻译过程分成解释执行和翻译执行两个部分，对于冷代码采用解

释执行的方式，当运行到达一定阈值后，会启用翻译器进行翻译，并将翻译后代码存储在 Tcache 中，避免再次翻译。

为了提高二进制翻译的效率，Crusoe 还在硬件上增加了影子寄存器（Shadow Register）结构。这些影子寄存器作为 X86 寄存器的复制，翻译后的指令只对这些影子寄存器进行操作。同时，Crusoe 在翻译中加入了检查点（Checkpoint）的机制，只有在检查点的时刻，才会将影子寄存器数据提交到真实的本机的寄存器内。如果发生了异常或例外，影子寄存器可以执行恢复操作，并进入解释执行状态，从上一个检查点重新开始对指令进行解释执行。这样既保证了异常或例外随时可以得到正确的处理，又可以让 Crusoe 对 X86 指令进行更加激进的优化，如指令调度，部分违背指令行为的语义转换和消除等。

2.2.3 MAMBO-X64 与 JTLT

为了降低间接跳转带来的开销，MAMBO-X64 和 JTLT 在硬件中引入一些结构，从而加快了翻译后代码处理间接跳转时查找和跳转的速度。

MAMBO-X64 通过返回优化、跳转表优化以及快速原子哈希表优化来降低间接跳转的开销[17]。其中，返回优化是在压栈时将源指令地址（SPC）和目标指令地址（TPC）压入栈中，返回时弹出这两个指令地址并进行比较，如果跳转目标是弹出的 SPC，则按照 TPC 的地址跳转；跳转表优化则是在翻译时刻直接将 SPC 替换成 TPC，而不再需要进行转换；快速原子哈希表则是把 SPC 和 TPC 打包成 64 位原子操作，提高查找、增加和删除等动作的效率。MAMBO-X64 的设计实现了非常低的性能开销，与本地执行相比，平均仅为 10%，同时这些优化平均还能减少约 40% 的开销。

JTLT 方案则采用了硬件辅助的方式来降低间接跳转的开销，通过添加 JTLT（Jump Target-address Lookup Table）结构，并与 BTB 联合使用，实现了 BTB 预测地址就是对应的 TPC，在硬件上实现了直接转换[27]。为了预测 ret 这种栈结构返回地址，则采用了双地址返回值栈（Dual-address return address stack）方式，即每次做 call 或其他调用时，将对应的 SPC 和 TPC 压入栈中，返回时，先进行栈预测，若预测不成功，则退化成 JTLT 方案。

2.2.4　Dynamo

Dynamo 使用纯软件实现，旨在通过程序运行收集性能信息，以便对原程序进行重新翻译，生成更高效的代码[18]，其核心算法如图 2.3 所示。

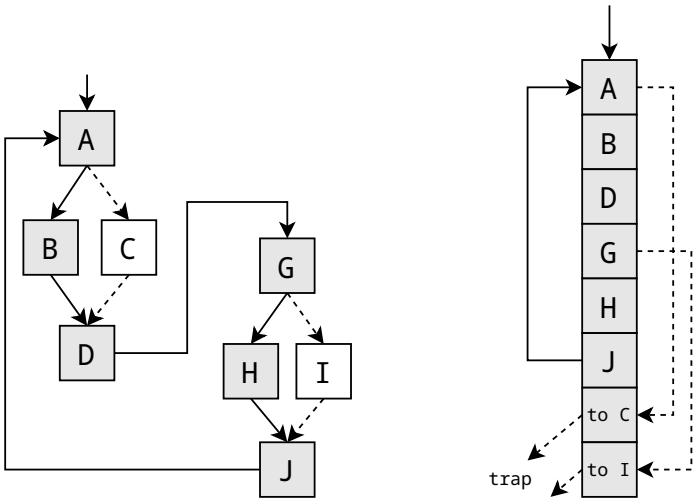

图 2.3 Dynamo 核心算法图

Figure 2.3 Algorithm of Dynamo

Dynamo 优化的核心是识别热点路径并进行优化。Dynamo 使用轻量级的热点追踪方案 MRET（Most Recently Executed Tail），该方案从回边对应的地址开始，在其运行的头部插入统计代码，用于记录运行次数。当该地址统计代码运行次数超过一定阈值，则启用解释执行的方式，在解释执行环境继续运行并记录。MRET 算法认为此时解释执行记录的路径即为热点路径。该方法可以避免翻译代码中插入过多 profile 代码影响运行性能。同时，可以将收集到的热点路径进行优化，对其路径上的基本块进行拼接，可以去除基本块链接中的直接跳转，直接形成一个更大的基本块，并对此进行整体的优化和指令消除。

2.2.5　HQEMU

HQEMU 由国立清华大学开发，该跨平台二进制翻译器以 QEMU 为基础，在其中加入了 LLVM 模块用于编译优化，从而获得更优的代码质量[28,29]，架构图如图 2.4 所示。

HQEMU 采用了类似于 Dynamo 的方案，通过识别并优化运行中的热点路径，以提升整体程序性能。HQEMU 会在 QEMU 翻译出的基本块头部加入统计代码，用于记录每个基本块的运行情况。这些统计代码分为两部分，一部分用于

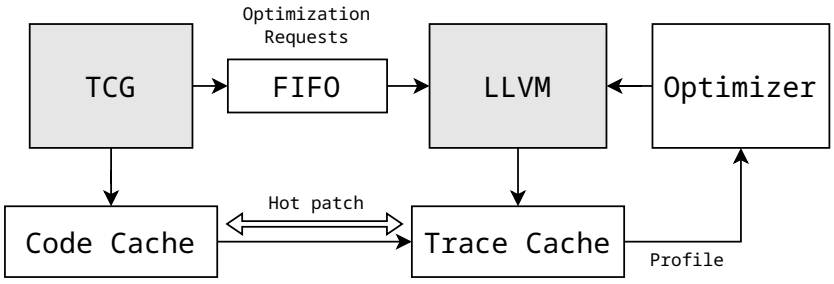

图 2.4 HQEMU 架构图

Figure 2.4 Architecture of HQEMU

记录该基本块运行的次数，另一部分用于记录运行的路径，只有当基本块运行次数超过了阈值，路径记录的统计代码才会被启用。此时会将运行情况记录在数组中，为下一步分析做准备。

HQEMU 在获取到热点路径后，会将其放入优化队列中。优化端会根据优化队列中的热点路径信息，将 QEMU 产生的 TCG 中间代码转换成 LLVM IR，并调用 LLVM 对其进行优化，以生成更高效的本地码。优化后的代码会存入 Trace Cache 中，并在优化器的帮助下动态的替换之前的低效率代码。

2.2.6 Rosetta2

苹果公司在 2020 年发布了 Rosetta2，它可以让英特尔架构的程序在 Apple Silicon 上运行。为此，Rosetta 采用了静态与动态相结合的方式，提前为二进制程序生成 AOT（Ahead-Of-Time）文件。该文件会先被加载到程序的地址空间，然后再结合动态翻译的方式，解决无法由 AOT 文件实现的诸如自修改等问题。此外，由于 AOT 文件是离线生成的，能够在离线状态下进行更加重量级的优化，以获得更加高效的翻译后程序。

2.2.7 QEMU

QEMU 作为一款开源的机器模拟器和虚拟化工具，提供了用户态二进制翻译、系统态二进制翻译和 KVM 的虚拟化支持，其二进制翻译器 TCG（Tiny Code Generator）通过实现类似于 LLVM IR 的一种中间指令 TCG IR，可以实现多种指令集间的翻译执行。原程序指令首先会被 TCG 前端翻译成 TCG IR，并在 TCG 后端的作用下翻译成目标平台的指令。

TCG IR 的中间层提供了原平台与目标平台之间的隔离，使指令间的转换以

及添加新的指令架构变得更加容易，但是中间层也伴随着不少的缺陷，比如它翻译过程中会损失大量的原指令流的语义信息，从而导致一些优化的实现变得更加复杂，翻译出来的效率也只有原生的 20% 左右。

尽管如此，QEMU 仍然进行了大量的优化，如基本块链接等。不仅保留了其支持多种架构的优势，而且翻译后的程序效率也基本达到可接受的水平。

1. 基本块链接。QEMU 通过实现完善的基本块链接优化，有效减少了基本块的退出，从而加速了程序的运行，其实现如图 2.5 所示。QEMU 在每个基本块退出翻译代码前添加一个链接槽位，并将其记录在基本块的信息域内。第一次运行时此基本块还未链接时，程序会退出翻译器并开始查找下一个基本块。在下一个基本块执行前，会根据前一个基本块的信息域内的记录，索引到链接槽位，并通过修改链接槽位的跳转目标来实现基本块的链接。

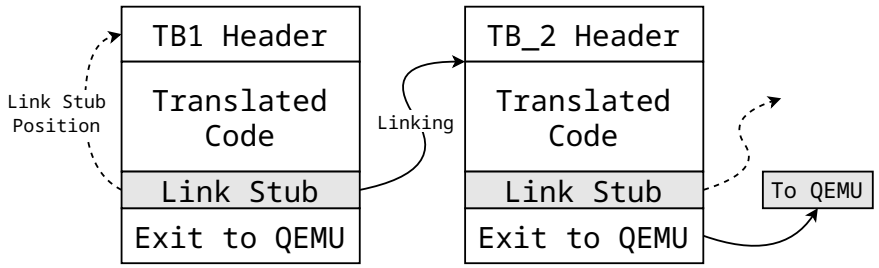

图 2.5 QEMU 基本块链接示意图

Figure 2.5 QEMU Translation Block Link

2. 寄存器重分配。QEMU 为了保证不同平台间的互相翻译，没有使用寄存器直接映射的方式，而是将寄存器的数据保存在内存中。每次使用寄存器时，需要将寄存器的数据从内存中加载，运算完成后将数据存回内存内。访存的开销是巨大的，如果每次都采用“加载-计算-存回”的方式完成一条指令的翻译，将会对其性能产生极大的影响。所以 QEMU 加入了寄存器重分配机制，在中间代码生成后，会扫描整个基本块寄存器使用情况，对寄存器进行活性分析，尽可能的将原平台的寄存器映射到目标平台的寄存器，以消除其中的“加载-存回”指令。

3. 中间代码优化。除了寄存器重分配外，QEMU 还在中间代码层上实现了常见的优化方案，如死代码消除等。QEMU 还对 EFLAGS 相关翻译实行了优化，通过延迟计算的方案，减少 EFLAGS 模拟指令的生成。

如图 2.6 所示，QEMU 在翻译指令时，如果该指令会产生 EFLAGS 信息，则记录下该指令的源操作数以及操作码，此时并不产生 EFLAGS 模拟指令（①）。

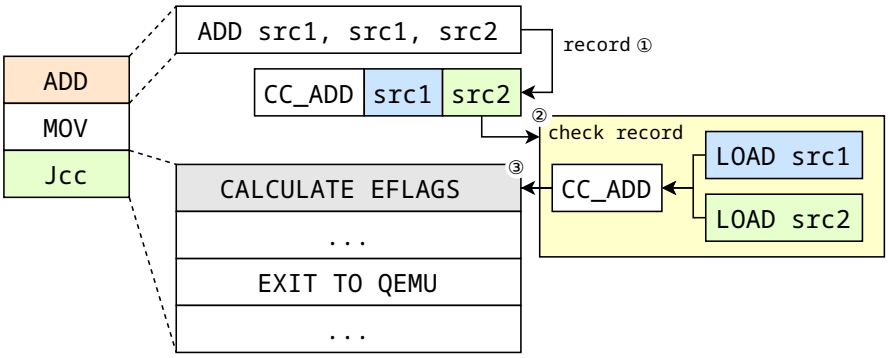

图 2.6 QEMU EFLAGS 延迟计算优化

Figure 2.6 QEMU EFLAGS Delay Calculation Optimization

在后续指令需要使用 EFLAGS 时，检查先前记录下的延迟信息（②），根据相应的源操作数和操作码，计算出此时的 EFLAGS 状态（③）。

2.3 龙芯二进制翻译器 LATX

龙芯二进制翻译器 LATX 采用了动态二进制翻译的方式，将 X86 平台的程序翻译到 LoongArch 架构上。现在，LATX 二进制翻译器可以正确运行众多商业应用，如微信、WPS 等，表现出良好的稳定性。LATX 解决了页大小不匹配、指令翻译验证困难和 EFLAGS 和 X87 浮点栈这几个最为主要的问题。

1. 页大小不匹配。与内存序类似，如何在大页系统（LoongArch 16K）上运行小页系统（X86 4K）的程序是一个复杂的问题，因为相邻区域内的每个页的权限可能不同，这使得在直接映射时无法确定宿主机上每个页的权限。为了解决相邻区域内页权限不同的问题，LATX 提出了影子页机制。该机制会在影子页中记录其原本权限，剥夺映射页的所有权限，并为其在影子页中开辟一个对应的空间。当进行读写该页操作时，会触发页权限异常，在异常处理函数内根据影子页原权限记录，对影子页进行模拟读写操作。

2. 指令翻译验证困难。QEMU 采用单条指令测试案例进行指令翻译的测试，但这种测试方式无法对指令间有相互依赖关系的情况进行测试，覆盖率并不高[30]。所以开发了随机指令测试系统 LATIVE（Loongson Architecture Translator Instruction Verification Enviroment），将指令进行随机组合，结果与真机相比，快速定位出错的翻译。

3. EFLAGS 和 X87 浮点栈。为了解决 X86 中 EFLAGS 指令翻译膨胀度过

高以及 X87 浮点栈复杂的处理，LoongArch 架构中加入了部分 EFLAGS 计算指令以及浮点 TOP 模拟指令。LATX 使用这些计算指令简单高效的解决了 X86 中 EFLAGS 和 X87 浮点栈翻译复杂的问题。

表 2.1 已经完成的单指令优化性能比较

Table 2.1 Performance comparison of completed single instruction optimizations

测试类型	优化前	优化前 $\frac{LATX}{Native}$	优化后	优化后 $\frac{LATX}{Native}$	性能提升
CINT	1220	45.3%	1295	48.1%	6.1%
CFP	1838	49.1%	1912	51.1%	4.0%

LATX 相比原生程序，翻译后的代码性能约为原生程序的 50%。通过单指令翻译分析与优化，对于常用指令，性能得到了 5% 以上的提升，如图表 2.1 所示，基本满足当前软件的性能需求。

2.4　本章小结

本章首先介绍了二进制翻译技术，并分析了主流二进制翻译器所使用的优化技术。Transmeta Crusoe 和 MAMBO-X64 等二进制翻译器虽然能够获得较高的性能，但其需要通过使用或修改硬件结构的方式，实现其中的优化。Dynamo 和 HQEMU 等二进制翻译器则通过热点块的跟踪和优化来提高整体程序性能，但这一操作会带来较大的开销，从而使翻译后程序的启动时间变长，不利于 JIT 等频繁自修改的程序。QEMU 在多架构间的二进制翻译方面的实现相当完善，但由于其通用性，翻译后程序效率也仅仅是达到基本可用状态。

此外本章还介绍了龙芯二进制翻译器 LATX。针对商业应用的需求，LATX 解决了许多问题。同时为了提高性能，它对单条指令进行了优化，实现了翻译后的代码性能接近原生代码的 50% 效率。为了进一步改善翻译后代码的性能，LATX 需要跨越指令的界限，通过指令流分析来实现更多的优化措施。

第 3 章　指令流分析制导的优化设计和实现

3.1　整体设计

为了提高龙芯二进制翻译器 LATX 的性能，我们寻求了一个平衡点，实现了指令流分析制导的动态二进制翻译优化方案 IFADO（Instruction Flow Analysis Directing Optimization）。这样，可以在尽量不增加翻译开销的前提下，实现了最大程度的优化空间。

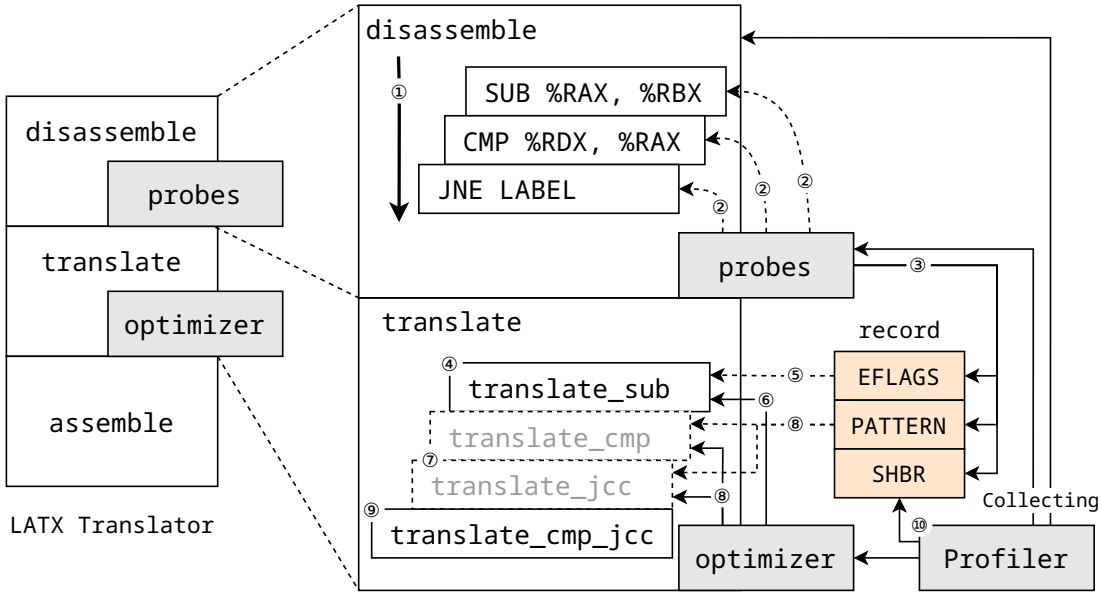

图 3.1 指令流分析制导优化框架构图

Figure 3.1 Framework for Translation Optimization Directed by Instruction Flow Analysis

我们在 LATX 的基础上加入了 IFADO 优化框架，该框架可以分为**分析探针**、**优化器**和**性能分析器**三个部分，整体架构如图 3.1 所示。

在反汇编阶段，分析探针可通过分析反汇编结果，收集指令流信息，识别可能存在的优化机会，并按照不同优化类型进行记录。在翻译阶段，优化器将基于记录的优化数据，指导每一种指令的翻译，避免不必要指令的生成。除此之外，性能分析器作为辅助测量工具，可以收集分析探针的优化数据，进而对翻译优化进行跟踪，从而改善翻译结果。同时性能分析器还提供各阶段优化的统计信息，可以用于指导未来优化方向。

3.1.1 分析探针

分析探针作用于反汇编阶段，通过指令流分析，按照优化类型分别识别存在的优化，具体的分析流程如下：

1. 反汇编器将基本块内的指令进行反汇编，生成指令列表（图 3.1 ①）；

2. 分析探针根据每种优化的识别方式，按照指令流顺序或逆序，逐条分析指令间的依赖关系（图 3.1 ②）；

3. 分析探针完成基本块内的分析后，会将分析数据按照每种优化类型，存入分析记录区域，用于下一步的优化动作（图 3.1 ③）。

分析探针记录的优化数据会被存放在分析记录区域，并按照优化类型分类存储。同时，这些优化数据还提供给性能分析器，用于统计本基本块的优化情况，以及提供跨基本块优化信息，以便进一步改善性能。

3.1.2 优化器

优化器作用于翻译阶段，会根据分析探针识别出的优化信息，完成翻译过程的优化工作。优化器按照不同的优化类型，分成**单指令优化器**、**跨指令优化器**和**基本块优化器**三种类型。

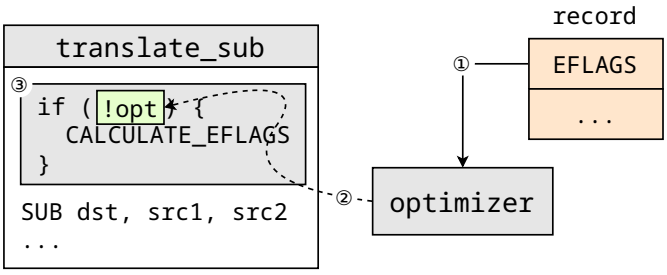

图 3.2 单指令优化器架构

Figure 3.2 Framework for Single-Instruction Optimizer

单指令优化器能够提供针对每条指令的优化，其作用域限定于单条指令，架构如图 3.2 所示。在每条指令翻译时刻，优化器根据优化信息记录中的数据，判断该指令是否可以进行优化（①），并将优化信息传递给翻译函数（②）。指令翻译函数根据获取到的优化信息，选择合适的翻译方式或优化模板来执行翻译，以减少翻译后指令的产生（③）。

跨指令优化器可以通过合并多条指令来实现优化，它的作用范围涵盖整个基本块，架构如图 3.3 所示。在指令翻译时刻，跨指令优化器会根据优化信息，

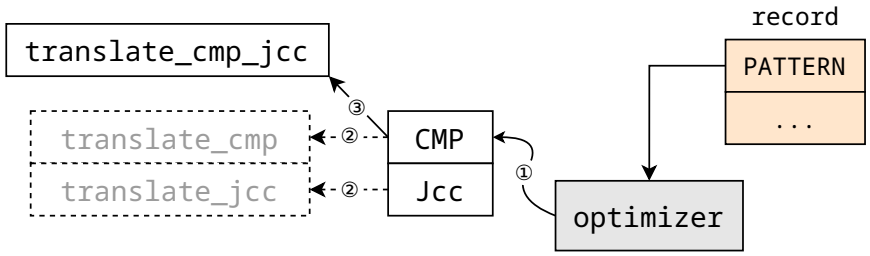

图 3.3 跨指令优化器架构

Figure 3.3 Framework for Multi-Instruction Optimizer

确定跨指令优化的指令序列，并且定位出待优化的几条指令（①）。跨指令优化器会首先将这几条指令无效化，防止它们使用普通翻译函数进行翻译（②）；接着根据优化信息，选择指令语义化翻译优化翻译函数，并将这几条指令的翻译函数重新映射到该优化翻译函数上（③）。在翻译这几条指令时，会调用映射后的优化翻译函数进行翻译。

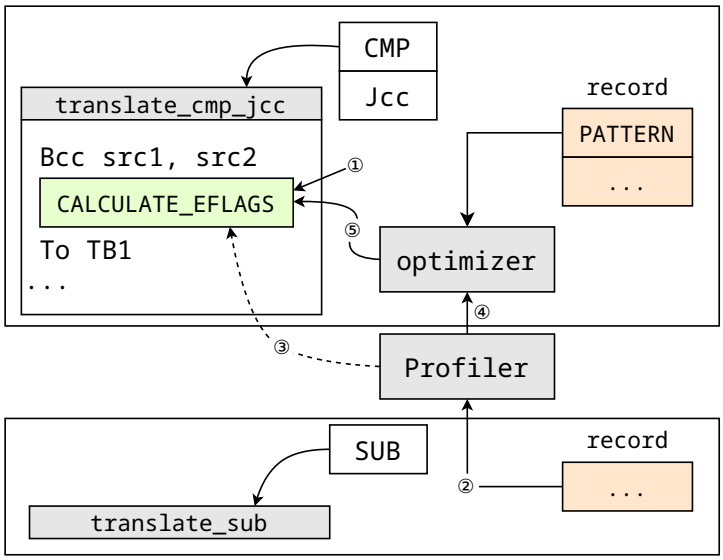

图 3.4 基本块优化器架构

Figure 3.4 Framework for Translation Block Optimizer

基本块优化器通过处理基本块间的优化信息，对基本块内部的指令进行优化，其架构如图 3.4 所示。在翻译时刻，基本块优化器会根据优化信息，标记基本块中以及基本块尾部（如基本块链接部分）需要优化的指令（①）。对于需要后续基本块信息才能确定能否被优化的指令，此时只进行记录，不进行消除；在下一个基本块翻译完成后，该基本块优化信息会被性能分析器收集（②）。性能分析器会根据前一个基本块标记的可优化信息，判断是否可以对其进行进一步

的优化（③）；如果可以继续优化，则调用基本块优化器（④）完成相应的优化动作（⑤）。

3.1.3 性能分析器

性能分析器会收集分析探针记录的优化数据，用于基本块间优化分析。同时性能分析器还会收集翻译器每部分的性能数据，以及优化情况，用于分析翻译器的翻译情况，并指导翻译器后续优化工作。

1. **收集探针记录的优化数据**（3.1.2 节）。通过收集探针记录的数据，进行跨基本块的优化。将收集到的优化数据与基本块记录的可优化记录进行比对，确认能够执行的优化，并调用基本块优化器执行优化动作。

2. **收集性能数据和优化情况**（3.2.1 节）。性能分析器将收集到的翻译器各部分性能数据进行收集分类，并存放在翻译器全局性能信息域中，提供准确的运行时性能统计信息，方便后续的优化工作。

3.2 性能分析框架

3.2.1 性能分析框架设计

性能分析器会收集翻译器各部分的性能数据和优化数据，用于后续的性能分析工作，其性能分析框架如图 3.5 所示。

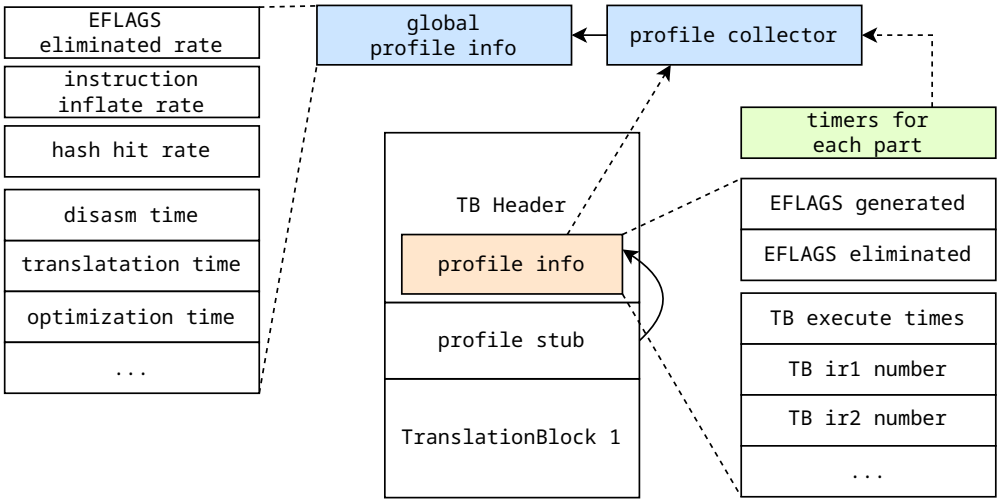

图 3.5 性能分析框架

Figure 3.5 Framework for Profiler

统计信息被区分成三种类型，分别是**运行时刻基本块信息、运行时刻翻译器**

信息和**静态分析信息**。

1. **运行时刻基本块信息**。这类信息需要在运行时，根据每个基本块的运行次数或状态，进行数据汇总，得到对应的统计结果。如 EFLAGS 产生和消除情况统计，基本块执行次数统计等。

由于 LATX 在运行过程中会对基本块进行链接，被链接的基本块无法退出到翻译器中，因此，性能分析系统无法捕捉每个基本块的运行情况。所以为了知晓链接部分基本块的运行次数，追踪链接后基本块的执行顺序，需要在基本块的头部插入统计代码，保证了分析系统可以追踪每一个基本块。

此外，为了方便插入的统计代码记录每个基本块的统计信息，分析系统在基本块的头部设立了统计信息保存区域。在基本块执行时刻，统计代码会使用原子指令，完成对头部统计信息域的操作。在程序运行结束时，每个基本块产生的统计信息会被收集器统一汇总到全局统计信息域，用于最后的数据汇总和分析。

2. **运行时刻翻译器信息**。为了评估各种优化对翻译性能的影响，性能分析系统还会对翻译器各模块实施插桩，并对运行时间进行统计。这部分的统计信息会被性能分析器收集，直接记录在全局统计信息域中。

3. **静态分析信息**。静态分析信息一般指在翻译过程中可以直接获取的性能数据，例如各个基本块翻译后指令的统计以及膨胀度统计等信息。性能分析器可以直接获取翻译过程中的翻译记录，并将其进行处理、保存和汇总，得到全部基本块的静态分析信息。

3.2.2　性能分析数据

我们根据性能分析框架，对二进制翻译器 LATX 产生的指令进行分析。通过插桩技术，获取翻译前后的指令序列以及运行的次数，并根据 SPEC CPU2000 运行数据，统计出各指令使用频率。

1. **指令膨胀统计**。表 3.1 展示了 gzip 中指令使用的频率。我们可以看到除了 MOV 类型的指令外，其他指令的翻译的膨胀率都超过 2 以上。一方面，这些指令尽管通过 “一对一” 的翻译方式可以实现其核心操作，但由于内存操作数的存在，需要多条指令计算访存地址，并通过访存指令加载或存入，导致指令平均膨胀度的提高。另一方面，一些运算类型指令，如 CMP 和 XOR，它们都会产生 EFLAGS 信息，而条件跳转指令，如 JNE 指令需要使用 EFLAGS 信息，这些 EFLAGS 相关的计算虽然有龙芯二进制翻译扩展指令（Loongson Binary Translation, LBT）的

辅助，但其 EFLAGS 信息的计算最少也需要额外的一条 LBT 指令，从而进一步增加了翻译后代码膨胀率。

当然，由于这些需要产生 EFLAGS 计算指令的运算类型指令占总动态指令数的 50% 以上，因此 EFLAGS 计算指令的性能和数量都会对翻译后的程序的性能有着较大影响。

表 3.1 gzip 中指令使用频率

Table 3.1 Frequency of instructions usage in gzip

指令类型	动态运行数量	占比	平均膨胀率
mov	7,718,467,866	16.04%	1.78
cmp	6,987,729,783	14.52%	2.87
movzx	5,350,658,906	11.12%	2.97
jne	3,790,599,144	7.88%	6.74
lea	3,151,611,499	6.55%	2.66
sub	2,744,875,934	5.70%	3.77
and	2,513,651,036	5.22%	3.94
je	2,318,250,121	4.82%	6.93
movsxd	1,995,327,254	4.15%	1.06
jae	1,978,473,851	4.11%	6.95
add	1,850,150,565	3.84%	3.47
xor	1,563,243,497	3.25%	4.16
shr	944,254,944	1.96%	2.58
test	615,277,980	1.28%	1.49
shl	523,026,764	1.09%	3.46
Other	4,082,735,574	8.48%	3.79
总计	48,128,334,718	100.00%	3.52

2. **龙芯二进制翻译扩展指令（LBT 指令）性能测试**。由于 X86 程序中需要生成 EFLAGS 计算的指令在动态指令中所占比例较大，所以 LBT 指令的性能也影响着翻译后程序运行的效率。为了评估这些二进制翻译指令的性能，采用了循环计时方法，性能测试方案如图 3.6 所示。针对每种指令的测试用例，循环执行

多次后计算每次平均用时，将运行时间与空操作或简单定点运行时间进行对比，分析出每种类型指令的性能数据，推测微结构情况[31]。

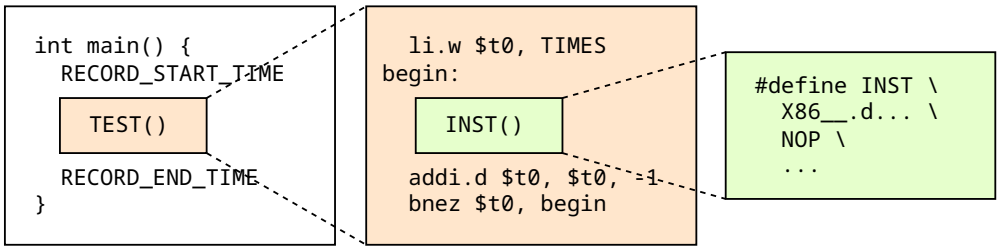

图 3.6 指令的性能测试方案

Figure 3.6 Structure of the Instruction Performance Test

我们采用的测试的数据和方案如下：

(a) **处理器基准性能**。使用 MOVE 类型指令，分别测试指令间存在依赖关系和无依赖关系的运行时间。通过该测试，能够分析出处理器发射部件的宽度，同时用于后续的基准比较。

该项测试得到存在依赖关系的指令运行时间为 6.877s，无依赖关系的指令运行时间为 1.734s，存在依赖关系的执行时间约为无依赖关系的执行时间的 4 倍，所以可以得出处理器发射宽度为 4。

(b) **指令执行周期**。由于 EFLAGS 运算指令只操作 EFLAGS 寄存器，且源操作数是 GPR，所以不存在依赖关系。这里使用 EFLAGS 运算指令与存在相关性的 ADD 指令的指令包，进行循环测试。通过不断增加指令包内 ADD 指令个数，统计其运行时间。通过测试时间之间的对比，分析其时间跃变点，可以得出其指令执行周期。

指令包内 ADD 指令个数为 2 个时为运行时间跃变点，说明此时 ADD 串行执行的速度和单条 EFLAGS 运算指令运行速度相同。此时可以得出 EFLAGS 运算指令执行周期为 2 cycles。

(c) **指令发射宽度**。测试指令发射宽度可以通过指令执行周期和指令带宽之比进行计算。采用 EFLAGS 运算指令运行时间与基准时间相比的方式，获得 EFLAGS 运算指令带宽，通过指令带宽，以及前序测试的指令执行周期，获取指令的发射宽度。

我们可以得到 EFLAGS 计算指令运行时间为 13.750s，与相同无依赖关系的基准时间 1.734s 相比，是其 8 倍，说明指令带宽为 MOVE 指令的 $\frac{1}{8}$，根据指令

执行周期为 2 cycles，可以得出指令发射宽度为 1。

(d) 其他测试的数据和结果如表 3.2 所示。

表 3.2 EFLAGS 运算指令性能结果

Table 3.2 Performance Results of the EFLAGS Instruction

指令类型	执行时间（秒）	发射宽度	运算周期	备注
基准（MOVE）	1.734	4	1	数据无关
基准（MOVE）	6.877	1	1	数据相关
搬运指令 +ADD	13.750	–	–	2 条 ADD
搬运指令 +ADD	20.626	–	–	3 条 ADD
搬运指令[1]	13.750	1	2	带宽为基准 $\frac{1}{8}$
运算指令 +ADD	13.750	–	–	2 条 ADD
运算指令 +ADD	20.625	–	–	3 条 ADD
运算指令[2]	13.750	1	2	带宽为基准 $\frac{1}{8}$
条件指令 +ADD	6.875	–	–	1 条 ADD
条件指令 +ADD	13.751	–	–	2 条 ADD
条件指令[3]	6.875	1	1	带宽为基准 $\frac{1}{4}$

[1] 指 LBT 指令中 EFLAGS 寄存器和通用寄存器数据搬运指令

[2] 指 LBT 指令中 EFLAGS 计算指令

[3] 指 LBT 指令中根据 EFLAGS 状态计算相应条件真假指令

从测试结果可以得到，EFLAGS 计算指令的性能并不优秀，微结构实现不如一般运算指令。在 3A5000 这一个四发射处理器上, 一条 EFLAGS 运算指令的开销等效于八条不存在数据依赖关系的普通运算指令的开销。

3. **SSE 向量指令翻译后性能分析**。由于 X87 浮点运算指令采用栈结构，其指令运算性能较低且扩展性不好，所以 AMD64 程序中普遍使用 SSE 向量指令替代 X87 浮点运算指令。SSE 向量指令可以被分成两大类型，分别是向量运算指令（如 ADDPS）和标量运算指令（如 ADDSS），如图 3.7 所示，这些标量运算指令用于替代 X87 浮点运算指令。

由于指令集的差异，想要完整模拟 SSE 向量指令中的这些标量运算指令往往需要进行更多的操作。X86 这些标量运算指令只对目标向量寄存器的低位数

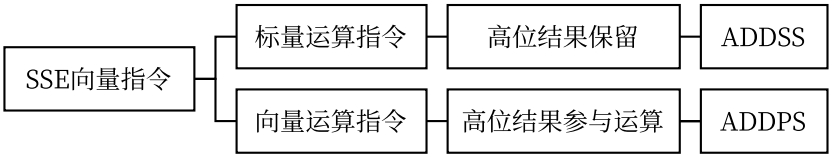

图 3.7 SSE 向量指令的分类

Figure 3.7 Classification of SSE Instructions

据进行运算和操作，而高位的数值保持原值。而对于 LoongArch 指令集来说，对应相似语义的指令只会将低位运算结果计算出并存入目标寄存器，而其高位的数据不保证为原值。所以，为了保证翻译语义的正确性，在翻译每一条标量运算指令时，都需要将向量寄存器内高位的数据进行“保存-恢复”。

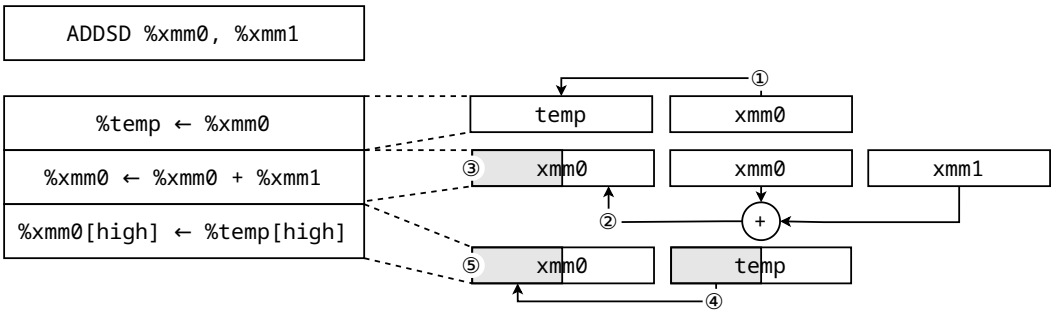

图 3.8 ADDSD 指令“保存-恢复”操作

Figure 3.8 Save/Restore for ADDSD Instruction

高位数据“保存-恢复”动作如图 3.8 所示。对于标量运算指令 ADDSD 指令，其语义为将向量寄存器低 64 位值做加法运算，运算结果存回目标寄存器低 64 位中，目标寄存器高 64 位结果保持原值。由于 LoongArch 指令计算加法会导致目标寄存器高位被破坏，所以为了保护目标寄存器 xmm0 高位数据，需要执行以下操作：

(a) 将 xmm0 的数据保存到临时寄存器 temp 中（①）；

(b) 执行加法运算指令（②），这会导致 xmm0 的高位被破坏（③）；

(c) 使用 temp 中保存的原 xmm0 的高位数据恢复 xmm0 的高位（④），从而实现 xmm0 的高位数据保持原值（⑤）。

实际上，如果 ADDSD 指令不关心高位的数据，其最简化翻译方案是只保留②过程。如果能够通过一些分析，将高位数据“保存-恢复”动作（① 和 ④）消除，其浮点性能将会有较大的提升。

3.3 优化方向

根据 3.2.2 部分的性能分析数据，我们可以得出 EFLAGS 指令的消除，以及标量运算高位“保存-恢复”动作消除是优化的重要方面。所以本文将会主要围绕 **EFLAGS 指令消除**、**指令语义化翻译**和**标量运算高位计算消除**这三个方面进行优化。

1. **EFLAGS 延迟计算**。为了减少 EFLAGS 指令的产生，采用 EFLAGS 延迟计算（Lazy Calculation）机制，即对于所有的 EFLAGS 计算指令，只有在会被使用时才会被计算。

2. **“反馈式”指令语义化翻译**。通过在反汇编阶段识别指令序列，可以获取可被优化的指令片段，在翻译阶段按照统一的翻译模板进行语义化翻译。并且在后续的优化过程中，加入“反馈式”的优化动作，通过后续分析信息，对前序翻译后的代码继续进行优化。

3. **SSE 标量指令高位运算消除**。针对 SSE 向量指令中的标量运算指令高位“保存-恢复”动作带来的较大开销，通过指令流分析，获取每个向量寄存器高位的状态。根据这些高位状态以及基本块内指令类型，将基本块进行分类，根据基本块不同的类型，动态消除这些“保存-恢复”动作，以优化浮点运算。

3.4 本章小结

本章介绍了指令流分析制导的动态二进制翻译优化（IFADO）的设计和实现。通过在反汇编阶段进行指令流分析，分析探针收集优化信息并存储；在翻译阶段，优化器从存储的优化信息中获取优化方案，根据不同类型的优化，使用三种不同类型的优化器进行优化。为了能够进行跨基本块优化，还加入了性能分析器，通过收集基本块内的优化情况，对已经完成翻译的基本块进一步优化；同时性能分析器还会收集翻译器各模块性能数据，用于指导后续优化方向。

通过性能分析器收集到的数据，我们发现了 X86 指令中，产生 EFLAGS 计算指令占比约 50%，其 EFLAGS 指令对指令膨胀的贡献较大；同时龙芯二进制翻译指令中 EFLAGS 计算指令性能较差，也影响了翻译后程序性能。另外，SSE 向量指令中标量运算指令用于替代 X87 浮点指令，这些指令的翻译会产生“保存-恢复”的动作，至少会增加一条向量搬运指令，从而使浮点相关指令膨胀率变高，影响了浮点运算性能。

为了解决这些问题，我们围绕着 EFLAGS 指令消除、指令语义化翻译和标量运算高位计算消除这三个方面，提出了对应的三种优化措施。在第 4 章，我们将会介绍 EFLAGS 延迟计算优化的设计与实现；在第 5 章，我们将会给出“反馈式”指令语义化翻译的细节设计和实现；在第 6 章将会详细介绍 SSE 标量指令高位运算消除算法，并将其与高位完全消除进行对比，分析其消除的效果。

第 4 章　EFLAGS 延迟计算

4.1　概述

在 3.2.2 部分中，我们深入探讨了需要进行 EFLAGS 计算的指令的频度以及优化的重要性。为了加速这些指令的 EFLAGS 计算，龙芯 3A4000 处理器引入了龙芯二进制扩展指令，这些指令只对 EFLAGS 信息进行操作，而不影响结果信息。但由于这些指令的产生，其会对包括 Cache、访存以及处理器内部微结构中取指、发射等造成压力，影响程序整体性能；同时这些二进制扩展指令的性能并不优秀，严重影响程序运行性能。因此，为了优化该问题，我们设计了 EFLAGS 延迟计算优化，以减少 EFLAGS 计算指令的产生，从而提高翻译后的程序性能。

本章节将在 4.2 部分首先对 EFLAGS 延迟计算的可行性进行分析，并结合分析结果概述 EFLAGS 延迟计算框架的设计思想；接着在 4.3 部分详细介绍该框架的设计实现，该框架将在指令分析阶段对 EFLAGS 进行预处理，分析基本块内各指令的 EFLAGS 使用情况，在指令翻译阶段根据分析结果决定是否翻译 EFLAGS 运算指令；最后，在 4.4 部分对优化结果进行分析。

4.2　可行性分析

EFLAGS 是 CPU 的标志寄存器，其结构如图 4.1 所示，它用于记录当前 CPU 的部分状态以及运算的结果状态。多数运算指令会根据运算结果，设置其运算的标志位（EFLAGS 位），这些标志位包括 CF（Carry Flag）、PF（Parity Flag）、AF（Auxiliary Carry Flag）、ZF（Zero Flag）、SF（Sign Flag）和 OF（Overflow Flag）这六种类。在用户态二进制翻译中，只需要关注这几类标志，其他与处理器状态相关的标志位可以忽略。

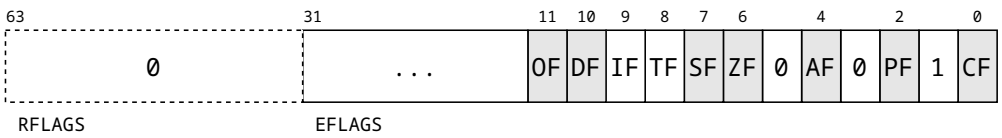

图 4.1 EFLAGS 结构图

Figure 4.1 Structure of EFLAGS

为了处理每条指令所产生的 EFLAGS，传统方案是针对每条运算指令，根据

其原操作数内容和运算后结果，进行模拟计算。由于每种类型的标志所包含的信息是正交的，所以需要对每种类型的标志单独计算。同时，每种类型的模拟计算往往还需要多条指令模拟，所以整体的 EFLAGS 运算需要大量的指令参与。

为了处理每条指令带来的 EFLAGS，传统的做法是对每条运算指令，根据其操作数内容和计算结果，独立计算每种类型的标志，如图 4.2 所示。由于多种类型的标志所包含的信息是正交的，加上每种类型的模拟计算往往需要多条指令进行模拟，这导致了 EFLAGS 运算会产生大量的运算指令。

```
IN:
0x000400e5f: test      rdx, rdx


OUT:
0x1608a49a0: and       t4,s2,s2
0x1608a49a4: ori       t0,zero,0x1
0x1608a49a8: sll.d     t0,t0,16
0x1608a49ac: ori       t0,t0,0x6081
0x1608a49b0: sll.d     t0,t0,16
0x1608a49b4: ori       t0,t0,0x8f00
0x1608a49b8: andi      t1,t4,0xff
0x1608a49bc: add.d     t1,t0,t1
0x1608a49c0: ld.bu     t0,0(t1)
0x1608a49c4: andi      a6,a6,0xfffb
0x1608a49c8: or        a6,a6,t0     // pf
0x1608a49cc: ori       t2,a6,0x40
0x1608a49d0: andi      a6,a6,0xffbf
0x1608a49d4: maskeqz   t1,t2,t4
0x1608a49d8: masknez   t3,a6,t4
0x1608a49dc: or        a6,t1,t3     // zf
0x1608a49e0: srl.d     t3,t4,56
0x1608a49e4: andi      t3,t3,0x80
0x1608a49e8: andi      a6,a6,0xff7f
0x1608a49ec: or        a6,a6,t3     // sf
0x1608a49f0: andi      a6,a6,0xfffe // cf
0x1608a49f4: andi      a6,a6,0xf7ff // of
```

图 4.2 TEST 指令 EFLAGS 模拟翻译实例

Figure 4.2 Example of EFLAGS translation through simulation

所以为了加速 EFLAGS 的翻译，引入了龙芯二进制扩展指令（LBT 指令），在每一条运算指令前，都会紧跟着一条或多条 EFLAGS 运算指令，图 4.3 展示了 SPEC CPU2000 中一个基本块的翻译情况，其中的 SUB，NEG，AND 以及 TEST 指令都会产生对应的 EFLAGS 运算信息，而 JG 指令会使用 EFLAGS 标志位，判断是否进行跳转。

但实际上对于其中大部分的龙芯二进制扩展指令，他们是不需要生成的，如

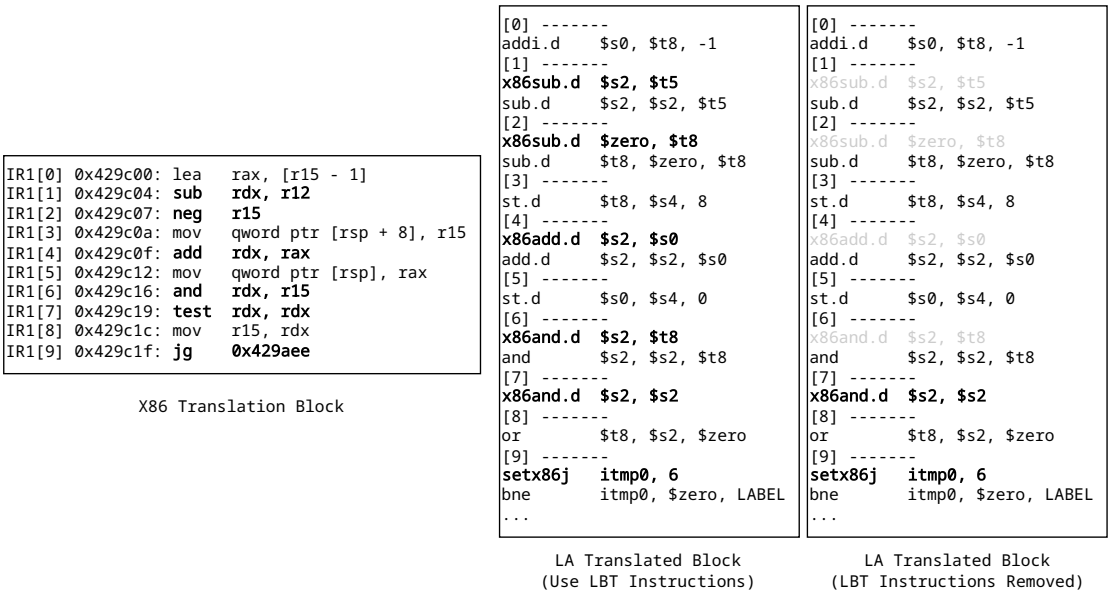

图 4.3 SPEC CPU2000 基本块翻译实例

Figure 4.3 Example of Translation Block Translation in SPEC CPU2000

图 4.3 中展示了基本块翻译中是否进行 EFLAGS 消除的对比。由于除了最后的 JG 指令需要使用 EFLAGS 信息，其余的前序指令产生的 EFLAGS 结果都会被后序运算所覆盖，所以只有最后的 TEST 指令需要产生龙芯二进制扩展指令，其他运算指令的 EFLAGS 计算可以省略。基本块使用的 EFLAGS 情况如图 4.4 所示。

从 EFLAGS 使用图中可以看到，在一个基本块中，运算指令产生的 EFLAGS 很可能会被随后的计算指令覆盖，特别是常用的 ADD、SUB 以及 AND、OR 这类指令，他们只会生成 EFLAGS 信息而不会使用。这样的指令的出现会使得在该指令前的指令无需再进行对 EFLAGS 信息的计算，如该基本块中的 SUB、NEG 以及 ADD 的指令会因为 AND 指令，而无需产生 EFLAGS 计算指令。

另外，大多数时候，基本块的结束是由一条条件分支指令作为结尾，该指令通常伴随 CMP、TEST 指令一起使用，用于产生所有的 EFLAGS 信息，提供给条件分支指令作为判断跳转的条件。所以在大多数情况下，一个基本块只需要在结尾处生成一条 EFLAGS 指令，用于后序条件跳转操作。

为了预先衡量 EFLAGS 延迟计算带来的性能收益，我们只考虑指令消除的情况，它符合一定的趋势，可以定义性能收益量为 Φ。

定义 4.1. 优化收益与指令动态消除数量和指令开销（运行周期数、发射单元数量以及指令间数据依赖等）成正比，与其优化前动态指令数成反比，其公式

	OF	SF	ZF	AF	PF	CF
LEA	--	--	--	--	--	--
SUB	M	M	M	M	M	M
NEG	M	M	M	M	M	M
MOV	--	--	--	--	--	--
ADD	M	M	M	M	M	M
MOV	--	--	--	--	--	--
AND	M	M	M	M	M	M
TEST	0	M	M	U	M	0
MOV	--	--	--	--	--	--
JG	T	T	T	--	--	--
EXIT	0	M	M	U	M	0

M: Modified, T: Tested, U: Undifined, --: No Change

图 4.4 基本块 EFLAGS 使用示意图

Figure 4.4 EFLAGS used in Translation Block

如 (4.1) 所示。

$$\Phi = \sum \gamma_i N_i / T \tag{4.1}$$

其中 T 代表优化前程序的动态指令数，γ_i 代表 i 类型指令的开销，N_i 代表 i 类型指令动态消除的数量。

根据定义 4.1 ，我们对 SPEC CPU2000 的子项进行 EFLAGS 延迟计算优化前性能评估，其分析结果如表 4.1 所示。由于消除的都是后序不再使用 LBT 运算指令，且不同类别的 LBT 指令运行周期数和发射单元数量一致，因此可以假定其不同类型 EFLAGS 指令开销为统一值。

从分析表 4.1 中，我们可以看出，其几何平均性能约能提高 8%，但由于会受到其他的因素影响，如处理器前端压力、Cache 命中率的改变以及对齐等情况，则其性能很可能高于或低于该值。但从分析的结果可以看出，此优化的正向收益一定大于可能存在的负向效果。

表 4.1 SPEC CPU2000 进行 EFLAGS 延迟计算优化前性能评估结果

Table 4.1 Performance Evaluation Results before EFLAGS Optimization in SPEC CPU2000

SPEC 子项	动态指令数 T	消除 EFLAGS 数 N	指令开销 γ	预估收益 Φ
164.gzip	914,249,687,214	45,338,891,817	2	9.9%
175.vpr	409,746,860,040	15,621,156,363	2	7.6%
176.gcc	360,583,576,775	15,816,240,872	2	8.8%
181.mcf	111,805,312,398	5,932,851,087	2	10.6%
186.crafty	401,703,630,087	29,454,626,624	2	14.6%
197.parser	757,242,867,276	30,618,302,500	2	8.1%
252.eon	333,136,091,931	6,536,931,065	2	3.9%
253.perlbmk	838,382,129,804	27,490,746,079	2	6.6%
254.gap	450,713,638,439	36,241,484,935	2	16.1%
255.vortex	739,613,996,786	23,532,841,903	2	6.4%
256.bzip2	739,320,223,986	48,947,087,341	2	13.2%
300.twolf	754,604,174,529	58,669,087,219	2	15.5%
INT AVG	–	–	–	9.4%
168.wupwise	668,723,753,944	31,101,409,408	2	9.3%
171.swim	182,576,402,750	2,963,047,558	2	3.2%
172.mgrid	418,966,797,356	7,932,046,348	2	3.8%
173.applu	477,699,227,494	6,747,833,750	2	2.8%
177.mesa	586,382,961,511	26,661,655,160	2	9.1%
178.galgel	391,694,937,836	20,000,377,108	2	10.2%
179.art	258,354,178,467	11,838,724,345	2	9.2%
183.equake	184,945,729,140	1,605,781,547	2	1.7%
187.facerec	471,258,061,664	47,165,980,016	2	20.0%
188.ammp	669,562,837,877	18,763,469,446	2	5.6%
189.lucas	412,231,389,323	23,631,076,609	2	11.5%
191.fma3d	409,261,861,740	12,348,419,455	2	6.0%
200.sixtrace	1,035,752,234,382	15,591,573,977	2	3.0%
301.apsi	578,463,841,318	30,280,584,068	2	10.5%
FP AVG	–	–	–	6.2%

4.3 EFLAGS 延迟计算框架

为了提高翻译后性能，QEMU 针对 EFLAGS 相关翻译，也提出了一种延迟计算的方案[6]，即在翻译指令时，记录该指令的操作码和源操作数，但不立即生成 EFLAGS 运算指令，而是等到后续指令需要使用 EFLAGS 中任意位信息时，才在后续指令翻译过程中产生前序指令的 EFLAGS 计算指令。Digital Bridge 也提出过类似的计算模式[32]，即对基本块内的 EFLAGS 计算情况进行分析，消除不必要的冗余 EFLAGS 计算。这些方案均存在一些缺陷：

1. 额外开销：由于 QEMU 延迟计算方案在产生 EFLAGS 计算指令时需要获取前序指令的源操作数数据，因此必须要求模拟的寄存器数据在内存中保持最新值，以便在 EFLAGS 计算指令产生时可以获取到最新的源操作数数据，这导致了更多的内存加载指令的生成。

2. 粗粒度：如果任一 EFLAGS 被使用，就会对全部的 EFLAGS 信息进行计算。这导致了某些几乎不被使用的 EFLAGS 位会被重复的计算，产生冗余指令。

3. 可扩展性差：优化大多都只在基本块内实现，对于跨基本块情况需要加入更多的复杂处理，如预翻译、超级块、热路径优化等。

所以为了解决这些问题，我们提出了 EFLAGS 延迟计算方案，该整体框架如图 4.5 所示。其采用在反汇编阶段进行指令流分析，获取每条指令的 EFLAGS 使用信息；在优化阶段，根据分析的 EFLAGS 使用信息，调用单指令优化器完成对应指令的优化。该方案在不引入额外开销的情况下，支持了 LBT 指令的消除操作。同时采用细粒度方式，针对 EFLAGS 中每标志位单独处理，进一步减少 EFLAGS 指令的产生。此外该方案还考虑了相应的扩展性，EFLAGS 分析结果能够被后续优化复用，实现了低开销的跨基本块的 EFLAGS 消除，我们会在第 5 章详细讨论跨基本块的 EFLAGS 消除。

在反汇编阶段，将进行指令流分析，这里采用逆序分析 EFLAGS 使用和产生情况（①）；分析后的结果将会存储在每条指令的分析结果区域内（②）。在指令翻译阶段，优化器将会按照指令流顺序，依次读取每条指令的信息，并从每条指令的分析结果区域内获取 EFLAGS 的使用情况（③），根据其使用情况，确定每条指令翻译时，是否可以消除某些标志位的翻译。

由于 LBT 指令只支持了常见的运算指令的 EFLAGS 计算，所以现在的二进制翻译器 LATX 将 EFLAGS 计算分成了两类，一类是采用 LBT 指令实现，另

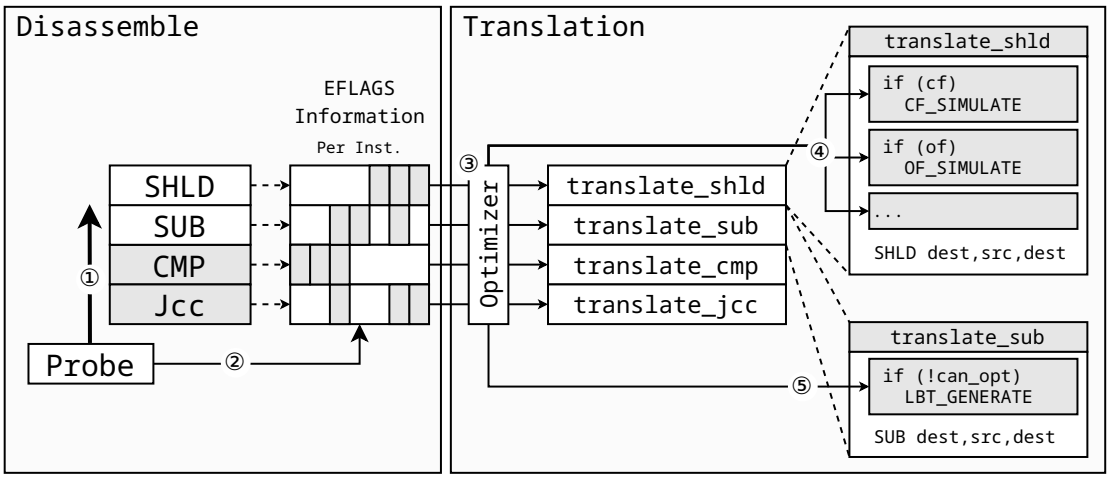

图 4.5 EFLAGS 延迟计算框架

Figure 4.5 Framework of EFLAGS Lazy Calculation Optimization

一类则是使用普通指令进行模拟。对于 LBT 指令的计算，优化器将会判断该指令是否存在需要计算的标志位，如果存在需要计算的标志位，则产生 LBT 指令（⑤）；对其他普通指令模拟的 EFLAGS 计算，则会根据标志位的类型逐一判断是否需要计算，对于不需要计算的标志位，优化器会进行优化和消除（④）。

4.3.1　指令流分析扫描

为了优化分析过程的性能，采用逆序方式对指令流进行扫描，其分析过程如图 4.6 所示。分析的数据按照产生方式被分为两部分，第一部分是指令本身的 EFLAGS 信息，包括指令使用 EFLAGS 的类型以及产生 EFLAGS 的类型信息（②）；另一部分是分析探针扫描时，根据指令本身的 EFLAGS 信息计算出的 EFLAGS 状态（③），包括该指令后序需要使用的 EFLAGS 状态位以及该指令需要产生的 EFLAGS 状态位。

分析探针扫描会分成两个阶段，包括了该指令**后序需要使用的状态位的计算**，以及该指令**需要产生的状态位的计算**。例如在图 4.6 中，探针正在分析 TEST 指令的 EFLAGS 信息（①）。

1. **后序使用的状态位的计算**（④）。在运算 TEST 指令需要产生的状态位前，需要首先获取该指令的后序指令需要使用的 EFLAGS 状态类型。由于扫描采用从后向前的逆序方式，所以该操作只需要获取上一次分析得到的“需要使用的状态位”信息，并基于上一条指令所产生和使用的 EFLAGS 状态类型，得出此时 TEST 指令的“需要使用的状态位”信息，如算法 1 所示。

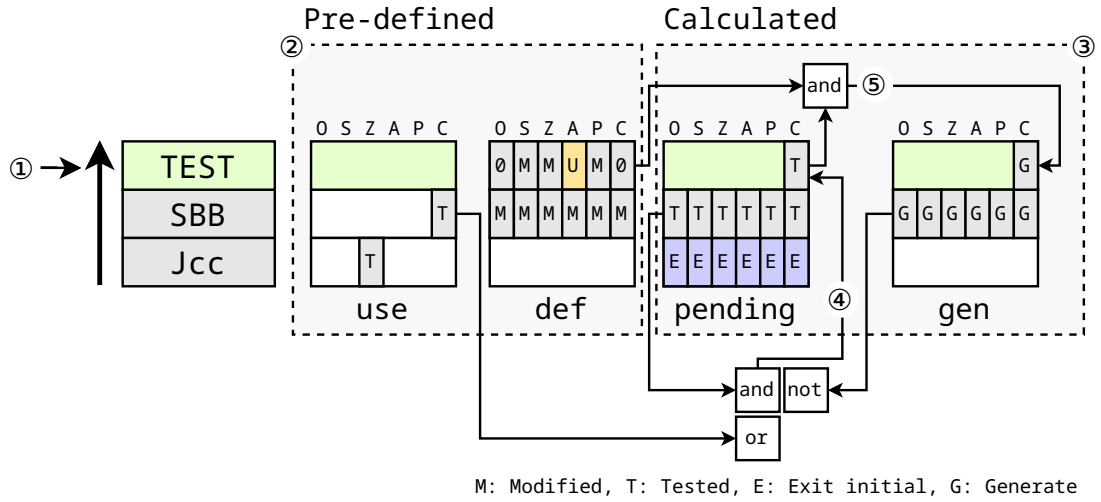

图 4.6 EFLAGS 延迟计算优化指令流分析过程

Figure 4.6 Instruction Flow Analysis of EFLAGS Optimization

该算法首先计算该指令的后一条指令完成 EFLAGS 计算后，剩余需要计算的 EFLAGS 状态类型。这可以通过使用后一条指令的“后序指令需要使用的状态位”信息，并排除后一条指令已经完成计算的 EFLAGS 状态类型获得。由于后一条指令也有可能使用部分的 EFLAGS 状态，所以需要与后一条指令使用的状态位进行或运算，以获取该指令的“后序指令需要使用的状态位”。

算法 1 EFLAGS Probe Scanning Algorithm

```
1: procedure EFLAGSPROBE(TB)                    ▷ input current Translation Block
2:     insts[n].pending ← ALL_EFLAGS
3:     for i = n→1, insts[i] in TB do
4:         pending ← (insts[i + 1].pending) AND (NOT insts[i + 1].gen)
5:         insts[i].pending ← (insts[i + 1].use) OR (pending)
6:         eflagsProbeGenerate(insts[i])
7:     end for
8: end procedure
```

2. **需要产生的状态位的计算**（⑤）。根据“该指令产生的状态位”和“后序需要使用的状态位”两个信息计算产生的状态位，如算法 2 所示。首先，为了满足后序指令的 EFLAGS 状态位的使用需求，如果该指令可以计算出状态位，则必须产生对应的计算指令。如果该指令对于该状态位的情况保持不变，则需要前序指令计算，并将状态位传递给前序指令，即设置该指令的“后续需要使用的状

态位”。

算法 2 EFLAGS Probe Construction Generating Table Algorithm

```
1: procedure EFLAGSPROBEGENERATE(inst)
2:     inst.gen ← (inst.def) AND (inst.pending)
3: end procedure
```

特别的，由于需要保证后续的基本块能够获取正确的 EFLAGS 信息，所以在基本块结尾处会强制设定成“需要全部的 EFLAGS 信息”，如图 4.6 中蓝色标志（E 标志）所示。

4.3.2　EFLAGS 指令消除

在完成探针分析后，优化器将会在指令翻译阶段进行优化动作。由于 LBT 指令只支持常见的 EFLAGS 计算，所以有一些不常用的指令需要采用指令模拟的方式完成 EFLAGS 运算。为了更加细致的管理 EFLAGS 的计算，针对这两种不同的计算方式，使用不同的 EFLAGS 指令消除方案，如图 4.7 所示。

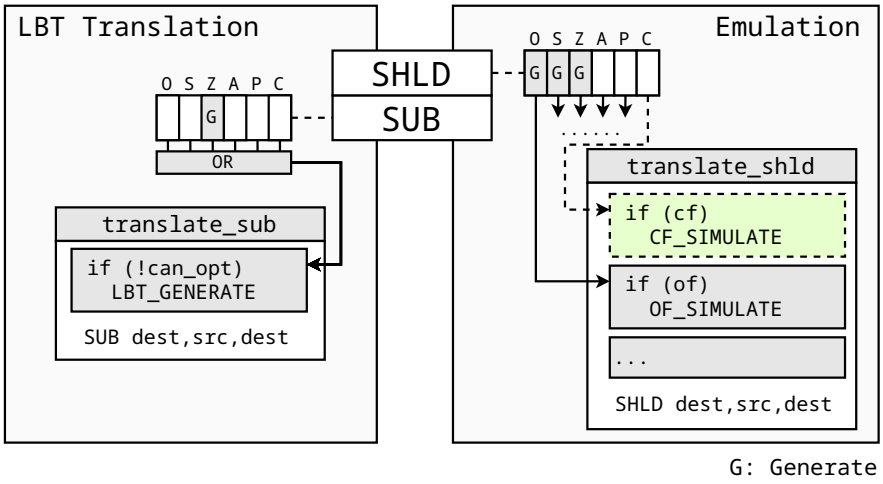

图 4.7 EFLAGS 指令消除

Figure 4.7 EFLAGS elimination

1. **LBT 指令消除**。LBT 指令计算 EFLAGS 的行为与 X86 对应的指令相同。因此，在进行翻译时，可以在运算操作前加入相应的 LBT 指令以完成相应的 EFLAGS 计算。在 EFLAGS 指令消除时，如果该指令的某一类标志需要计算，则对应的 LBT 指令需要保留；反之，则可以将该 LBT 指令进行消除。如图 4.7 中对于 SUB 指令的翻译，虽然探针分析得出该指令只需要计算 ZF 标志位，但由于其使用了 LBT 指令，所以该 LBT 指令仍然需要产生。

2. **模拟 EFLAGS 计算指令消除**。与使用 LBT 指令不同，这类情况需要使用普通指令进行 EFLAGS 的模拟计算。由于每种标志的含义是正交的，所以在翻译时可以针对每种标志，产生单独的翻译指令。因此，在进行 EFLAGS 指令消除时，优化器需要针对每种标志单独判断。如果某一标志需要进行计算，则需要生成相应的模拟计算指令；反之，则将该标志对应的计算指令进行消除。如图 4.7 中对于 SHLD 指令的翻译，探针分析可知该指令需要计算 OF，SF 和 ZF，但不需要计算 CF。因此，翻译该指令时，可以省略相应的 CF 标志位模拟指令，而 OF 标志位则需要产生对应的模拟指令来计算。

4.4 性能分析

为了对 EFLAGS 延迟计算优化的结果进行性能分析和评估，我们使用了 CoreMark 程序和 SPEC CPU2000 测试集分别进行测试。

我们首先使用 CoreMark 进行性能测试。CoreMark 是一款用于嵌入式系统性能评估的程序，其程序能够被完整地装载在 L1 Cache 中，运行时不会受到内存时延等影响。同时该程序核心循环均为运算操作，对内存分配和库的依赖很少。因此，它是一个很好的测试指令翻译优劣的工具。

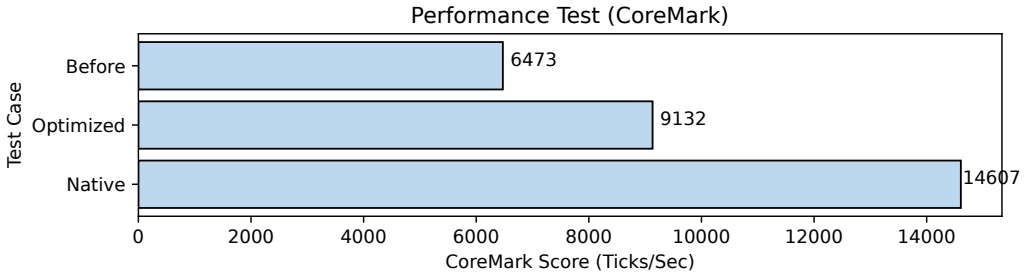

图 4.8 EFLAGS 延迟计算后 CoreMark 性能对比

Figure 4.8 CoreMark performance Comparison

CoreMark 的测试结果如图 4.8 所示，其性能从优化前的 6473 分提高到现在的 9132 分，性能达到原生的 65% 左右。为了探究 EFLAGS 延迟计算对性能影响，我们使用性能分析框架分析 EFLAGS 优化数据。情况如表 4.2 所示。我们根据性能分析框架给出的统计数据，可以观察到 EFLAGS 指令的优化率达到 55%，而优化开销仅占翻译时间的 4% 左右。

为了更加准确的测量 EFLAGS 延迟计算带来的性能提升，我们还进行了

表 4.2 EFLAGS 延迟计算后 CoreMark 性能数据

Table 4.2 Performance Data of CoreMark after EFLAGS Lazy Calculation

动态指令数	原 EFLAGS 数量	优化数量	优化比例	优化开销	开销占比（$\frac{opt.time}{trans.time}$）
11,572,359,694	1,710,998,750	941,230,733	55.0%	2.541ms	4.44%

SPEC CPU2000 的性能测试，测试结果如图 4.9 所示。

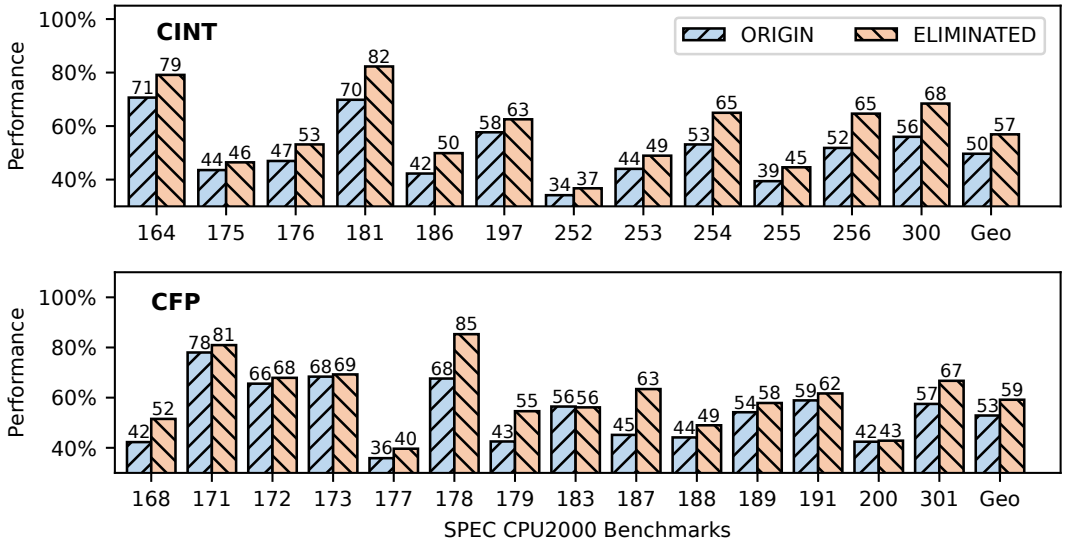

图 4.9 EFLAGS 延迟计算优化的 SPEC CPU2000 性能数据

Figure 4.9 Performance Data of SPEC CPU2000 after EFLAGS Lazy Calculation

从图中我们可以看到，定点分数提高了约 7%，浮点分数提高了约 6%。测试子项中可以发现，有些子项的性能提升非常明显，如定点中的 181.mcf 子项，性能提升了 14%，达到原生的 84% 的性能；浮点中的 178.galgel 子项性能提升了 18%，达到了原生 86% 的性能。当然其中还有一些子项的性能提升不明显，如定点中的 252.eon 子项，其性能提升只有 2%，与原生程序相比其性能仍旧很低；浮点中 183.equake 和 200.sixtrace 子项，性能提升约 1%。为了分析 SPEC CPU2000 中各子项性能提升的差距，我们通过性能分析框架获取其 EFLAGS 产生和消除的信息，如图 4.10 所示。

结合表 4.1 中的预估收益，虽然 181.mcf 子项的 EFLAGS 指令消除数量较 252.eon 子项少，但由于 181.mcf 程序的动态指令数量要远低于 252.eon 程序，所以对于 181.mcf 程序来说，其消除的 EFLAGS 数量占总的动态指令比例很高，从

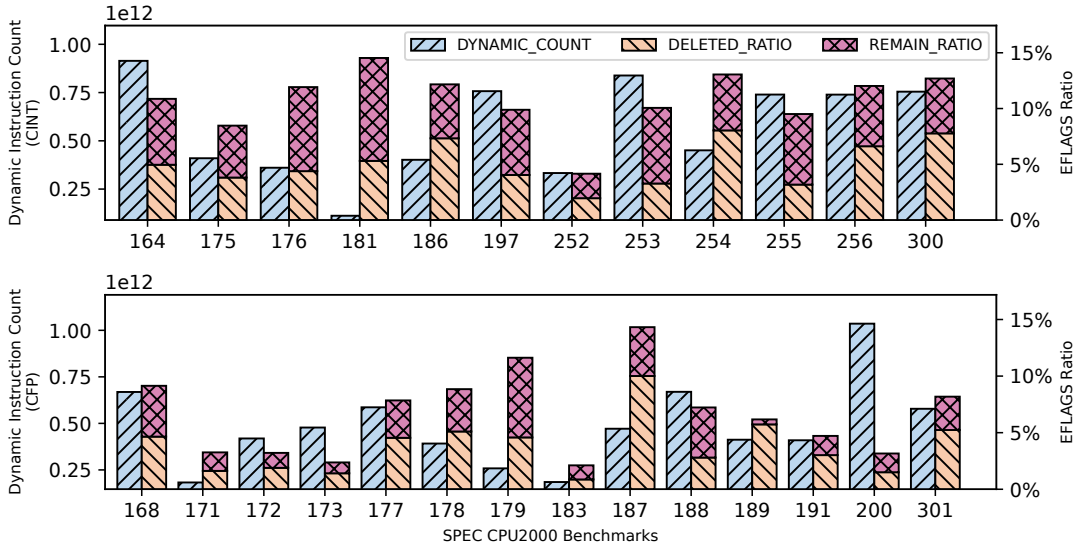

图 4.10 SPEC CPU2000 各子项 EFLAGS 消除对比

Figure 4.10 EFLAGS Elimination for each Test in SPEC CPU2000

而带来了显著的性能提升。而 183.equake 和 200.sixtrace 子项消除的 EFLAGS 比例较其他程序要低，因此性能提升也相对较小。该结果与表 4.1 中的预估收益接近，也说明预估收益的合理性。

大多数情况下，以下两种原因导致了 EFLAGS 运算未被完全消除：

1. **后续指令需要使用 EFLAGS**。条件跳转（Jcc），条件移动（CMOVcc）和条件设置（SETcc）指令，他们需要使用 EFLAGS 中的某几个标志，该标志位需要前序指令进行相应的 EFLAGS 计算，属于无法避免的 EFLAGS 计算。为了解决该问题，将会在第 5 章介绍指令语义化翻译方案，减少这种需要使用 EFLAGS 的情况的产生。

2. **基本块结尾 EFLAGS 需要生成**。在基本块结尾处，需要产生所有 EFLAGS 标志，以保证后续基本块能够获取正确的 EFLAGS 信息。此时如果结尾处使用了直接跳转指令，则会导致前序指令的 EFLAGS 无法消除。该问题需要通过扫描后续基本块，以确定是否能够消除。但持续的扫描会导致翻译开销增大，所以在第 5 章将会介绍“反馈式”的跨基本块的 EFLAGS 消除方案，以缓解该问题。

4.5 本章小结

本章为了解决 EFLAGS 计算指令性能较差问题，设计并实现了 EFLAGS 延迟计算优化。相比 QEMU 等其他二进制翻译软件实现的 EFLAGS 延迟计算，该

方案可以做到低开销，细粒度，高可扩展。

本章 4.2 节对 EFLAGS 延迟计算优化的可行性进行了分析，给出了 EFLAGS 指令消除的实现思想，并提出了优化收益与动态指令消除间的数学模型，并根据该模型进行了收益预估，证明了该优化的有效性。

在 4.3 节介绍了 EFLAGS 延迟计算框架的实现细节。优化分为两个阶段，分别是指令流分析扫描阶段和 EFLAGS 指令消除阶段。指令流分析扫描阶段采用逆指令流顺序扫描，通过分析该指令之后所需要的 EFLAGS 标志位信息，计算出该指令需要产生的 EFLAGS 标志位，并将该信息存储在对应的指令结构内。EFLAGS 指令消除阶段会根据翻译指令的类型，将优化分为 LBT 指令消除和模拟 EFLAGS 计算指令消除两种方式，并进行针对性优化。

在 4.4 节我们进行了性能的测试和分析，得出了与预估收益相似的结果，即消除的 EFLAGS 指令占动态指令数越多，则其优化效果越明显。性能测试数据显示，SPEC CPU2000 的性能定点浮点均有 6% 以上的绝对性能提升。

同时，性能分析结果还显示 EFLAGS 计算指令未被完全消除，这主要是由于后续指令需要使用 EFLAGS，以及基本块结尾 EFLAGS 需要生成这两个原因所致。所以在第 5 章，我们将会针对这个问题，提出相应的优化方案，进一步消除无效的 EFLAGS 指令。

第 5 章　“反馈式”指令语义化翻译

5.1　引言

为了解决存在的后续指令需要使用 EFLAGS 和基本块结尾 EFLAGS 需要生成两个问题，我们提出了“反馈式”指令语义化翻译优化方案。该方案将 EFLAGS 生成的指令与使用的指令相结合，使用语义化翻译的方式减少指令中间 EFLAGS 的生成；同时，在翻译过程中将待优化部分进行标记，并在后续基本块的翻译过程中，通过后续基本块反馈的信息，对该基本块再次进行优化，以达到对结尾处 EFLAGS 指令的消除。相较于**窥孔优化**，指令语义化翻译更为激进。它只关注指令序列的主要语义，并对其进行语义替换翻译。对于不符合原操作的部分，将在异常处理和基本块结尾等环节添加恢复操作，同时翻译器可以继续对恢复操作进行优化，进一步提升性能。

本章将会详细介绍“反馈式”指令语义化翻译的设计与实现，在 5.2 节将会介绍优化的整体设计。该优化分为两个部分，分别为**语义化翻译优化**和**跨基本块反馈优化**，在 5.2 和 5.4 节将会详细介绍这两个优化措施。

此外，本优化也对跨越基本块时可能出现的自修改问题进行了处理。在 5.5 节将会详细讨论跨基本块优化后发生自修改带来的问题，同时介绍我们实现的恢复方案，该方案可以确保跨基本块优化后，即使出现自修改代码等情况，基本块的执行状态也能保持正确。

5.2　整体架构

为了达到对应的优化效果，我们设计的“反馈式”指令语义化翻译框架如图 5.1 所示。优化过程分成了反汇编阶段的**探针搜索优化指令序列**和翻译阶段的**优化器执行语义化翻译**这两个阶段。

在反汇编阶段，探针将会按照指令流顺序，寻找存在优化的指令序列（①）。探针发现相应的指令序列后，会将该序列打包并记录（②），同时将指令包内第一条指令的操作码替换为特殊的操作码（③），并将指令包内其他指令设置成无效指令（④）。在翻译阶段，优化器会根据指令包内的记录，判断指令是否已被无效，如果无效则不进行翻译操作；否则则按照记录的操作码，选用相应的语义

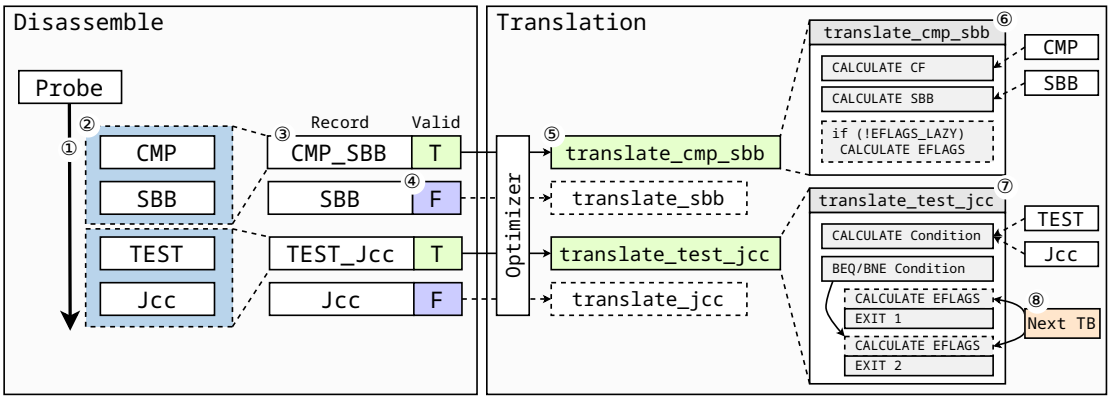

图 5.1 “反馈式”语义化翻译框架

Figure 5.1 Framework for Instruction Semantic Translation

化翻译函数进行翻译（⑤）。语义化翻译函数分成两大类，分别是基本块内语义翻译函数（⑥）和基本块结尾语义翻译函数（⑦），这两种翻译函数采用了不同的优化策略，我们将会在 5.3 节详细的介绍。

另外，对于基本块结尾语义翻译函数，还加入了跨基本块反馈式优化方案，用于消除基本块结尾的 EFLAGS 计算指令，该优化将在 5.4 节详细介绍。

5.3 语义化翻译方案

5.3.1 指令序列搜索

在反汇编阶段，优化探针会对指令流进行分析，寻找可优化的指令序列，并通过修改指令记录，指导优化器进行翻译，其与窥孔优化分析方法类似，如图 5.2 所示。

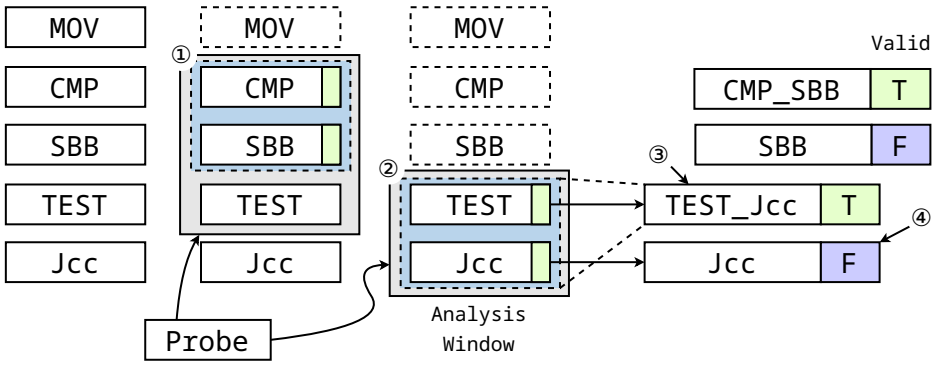

图 5.2 指令序列搜索框架

Figure 5.2 Framework for Instruction Pattern Search

优化探针将会维护一个分析窗口，对该窗口内的指令和数据流进行分析，以

判断是否存在可优化指令序列。如果满足相应条件，则会对该窗口进行标记，并通过不断扩大窗口宽度，寻找更大的优化空间。如果扩大窗口宽度后，不再存在符合的优化指令序列，则进行快速回退，将前序标记中最大的窗口认为是该序列的一个优化指令包（①），并将优化指令包最后一条指令设定为下一次的分析窗口头，进行后续的分析（②）。

在探针完成相应的分析后，会将指令包中的第一条指令的操作码进行替换，并标记该指令包需要完成的融合动作和采用的翻译函数（③），同时，还会将指令包内其他指令进行无效化标记（④），以确保翻译器不会对这些指令进行翻译。如图 5.2 所示的 TEST 和 Jcc 的可优化指令序列。该序列被探针识别后，会将 TEST 指令的操作码替换成 TEST_Jcc 类型（③），翻译阶段优化器会调用 test_jcc 语义翻译器进行翻译；同时会将 Jcc 指令设置为无效（④），翻译阶段不再对该指令进行翻译。

5.3.2　语义化翻译

在分析探针完成分析后，会执行翻译过程。由于分析探针对指令包内指令操作码和有效位进行了操作，所以框架复用了普通函数的翻译逻辑。翻译器会首先根据有效位判断该指令是否需要进行翻译，然后会根据指令操作码查询对应的语义化翻译方案，最后调用对应翻译函数进行翻译。

语义化翻译与窥孔优化技术原理相似，都会根据分析出的可优化指令序列，进行相应的指令消除、替换以及优化。然而，与窥孔优化相比，语义化翻译扩展了优化的范围，不再局限于严格遵循指令序列的所有语义情况。它会将指令在当前不需要使用的语义置于翻译器的异常处理部分或基本块结尾等位置，只有在控制流发生改变（例如发生异常）需要使用这些语义时，才会执行相应的语义恢复操作，加大了优化的力度。

语义化翻译分为**基本块内语义化翻译**和**基本块尾语义化翻译**两种类型。

1. **基本块内语义化翻译**。基本块内语义化翻译用于解决多指令间 EFLAGS 信息传递导致的 EFLAGS 指令无法消除，及单指令翻译膨胀过大的问题。如图 5.3 所示，对于 IDIV 指令的翻译，由于其操作的是 128 位的除法运算，其无法直接使用指令进行翻译，需要调用 C 语言编写的函数进行模拟。所以如果采用单条指令的翻译方案，该指令会产生超过百倍的指令膨胀，严重影响了性能。但如果使用基本块内语义化翻译方案，当识别到 IDIV 指令前存在 CQO 指令，即

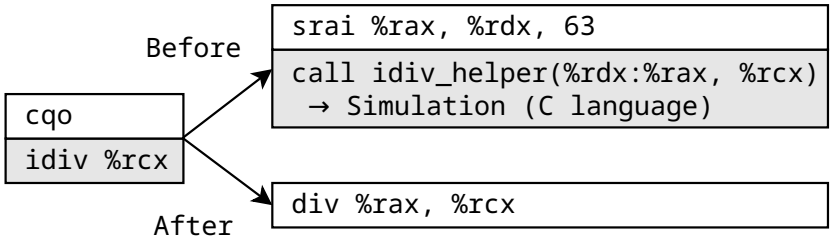

图 5.3 IDIV 指令翻译对比

Figure 5.3 Comparison of IDIV Translations

128 位被除数的高 64 位进行了符号扩展，使得此次运算变成了普通的 64 位除法操作，则可以直接使用对应的除法指令完成翻译。

基本块内语义化翻译如图 5.1 中 ⑥ 所示，其核心思想是识别指令包内指令，并按照特定模板进行翻译。具体流程如下：

(a) 获取指令包内所有指令的操作数，按照翻译模板执行翻译动作；

(b) 检查指令包中的指令是否需要计算 EFLAGS。将指令包当成一条“融合指令”，分析指令包内指令 EFLAGS 产生和使用情况，结合 EFLAGS 延迟计算优化，判断是否需要产生 EFLAGS 计算指令。

2. **基本块尾语义化翻译**。基本块尾的语义化翻译与基本块内的语义化翻译略有不同，其主要体现在结尾处的 EFLAGS 处理上。如图 5.4 所示。

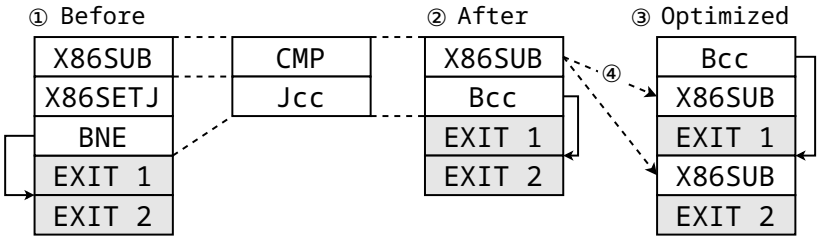

图 5.4 基本块尾语义化翻译实例

Figure 5.4 Example of Semantic Translation at the End of Translation Block

由于在 EFLAGS 延迟计算优化中，为了防止后续基本块需要使用前序基本块的 EFLAGS 信息，基本块结尾处会被强制设置成“EFLAGS 信息全部产生”，所以在退出基本块前需要对 EFLAGS 进行计算。为了进一步消除结尾处的 EFLAGS 指令，实现跨基本块的“反馈式”优化，这里进行了特殊的处理。我们将 EFLAGS 计算放入退出的代码部分，这样当任意一个出口发现后续基本块不再使用该 EFLAGS 计算的结果时，就可以将该出口的 EFLAGS 计算消除。这

样基本块结尾的 EFLAGS 优化只依赖于后续的单个基本块，因此无需基本块的两个出口同时满足消除条件才能执行优化。这样优化更为灵活，可以有效提高 EFLAGS 消除比例。

图 5.4 给出了 CMP 和 Jcc 的指令序列的优化示意图。在执行语义化翻译前，两条指令会单独进行翻译。这会产生 EFLAGS 计算指令 X86SUB[1] 以及 EFLAGS 测试指令 X86SETJ[2]，同时为了实现条件跳转，还需要通过 BNE 指令来测试 X86SETJ 指令的结果（①）。如果采用基本块内语义化翻译，则优化器会根据 CMP+Jcc 的语义，调用相应的条件跳转指令直接翻译，当然为了保证基本块退出时 EFLAGS 的正确性，还需要产生 X86SUB 指令用于计算基本块退出时刻的 EFLAGS 状态（②）。为了进行基本块尾的优化，EFLAGS 指令会被放入退出的代码部分（④），此时的 CMP+Jcc 指令语义化翻译后只剩下条件跳转指令 Bcc（③）。

5.4　跨基本块反馈优化

为了解决 EFLAGS 指令在基本块结尾处无法完全消除的问题，我们提出了一种反馈式优化方案。在完成基本块尾语义化翻译后，会在基本块末尾生成相应的退出代码，并将 EFLAGS 计算指令放入各个出口的退出代码部分。当后续基本块完成分析和翻译后，会将后续基本块的状态反馈到该基本块，并对可以消除的 EFLAGS 指令执行优化操作。

优化分为两种类型，分别为**消除式优化**和**链接式优化**，分别用于解决未被消除情况占优和消除情况占优两种情况。

5.4.1　消除式优化

消除式优化即通过后续基本块的反馈，消除部分无用的指令，通过将这些指令修改为 NOP 指令，以达到优化效果，优化架构如图 5.5 所示。该种类型优化的优势是对于无法消除的情况，不会产生多余的指令；但对于可以消除的情况，消除会导致产生多余的 NOP 指令，所以对于消除为多数的情况下会有一定的优化损失。

以 CMP+Jcc 的基本块尾语义化翻译为例，消除式优化的具体流程如下：

1. 在生成 EFLAGS 运算指令时，优化器会记录下 EFLAGS 运算指令生成情

[1]LBT 指令，该指令按照 X86 中 SUB 指令行为，计算出给定的两个寄存器的 EFLAGS 信息。

[2]LBT 指令，该指令根据给定的条件类型及 EFLAGS 信息，设置目标寄存器值为真或假（1 或 0）。

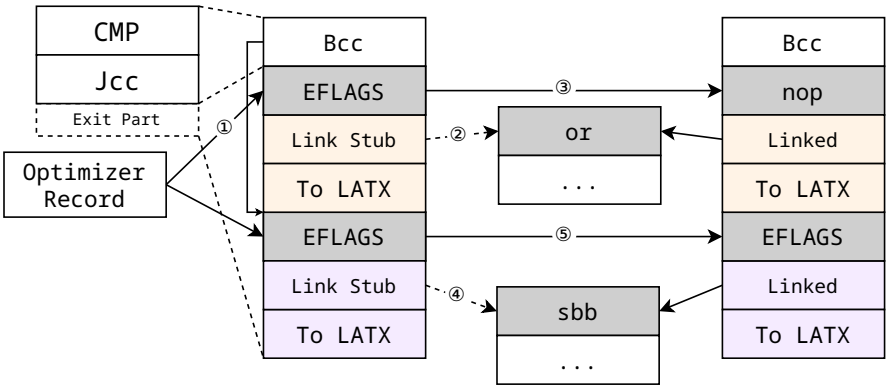

图 5.5 消除式跨基本块反馈优化

Figure 5.5 Eliminative Cross-Translation Block Feedback Optimization

况和生成的位置，方便后续优化过程中对该 EFLAGS 运算指令进行定位（①）；

2. 在后续基本块链接过程中，执行优化动作。如果后续基本块分析得出不需要使用前序基本块的 EFLAGS 信息，如 ② 中，基本块链接时刻发现后续基本块第一条指令为 OR，该指令 EFLAGS 计算会覆盖前序结果，因此可以消除基本块出口处的 EFLAGS 指令。为此，优化器会搜索记录的 EFLAGS 运算指令的生成位置，并将其修改为 NOP 指令（③）；

3. 如果在基本块分析过程中发现后续基本块需要使用前序 EFLAGS 信息，如 ④ 中基本块链接时，分析得出后续基本块第一条指令 SBB 需要使用前序的 CF 信息，那么优化器将不会执行相应的消除操作，EFLAGS 将继续保留（⑤）。

5.4.2 链接式优化

除了消除式优化，我们还针对使用指令模拟 EFLAGS 计算的情况，设计了链接式优化。通过链接式优化，可以在基本块链接时刻改变链接位置，提前跳转到下一个基本块，实现对无用指令的优化，优化架构如图 5.6 所示。该种类型的优化优势是对于可以进行消除的情况，基本块会提前跳转到下一个基本块，不会产生多余的指令；但对于无法消除的情况，会保留下预留给链接点的指令，所以对于消除为少数的情况下有一定性能损失。

以 CMP+Jcc 的基本块尾语义化翻译为例，链接式优化的具体流程如下：

1. 在基本块结尾处，每个出口都会预留两个链接点，在这两个预留的链接中间会加入 EFLAGS 计算指令；

2. 优化器记录每个出口处预留的链接点位置（①），将他们设置为链接的候

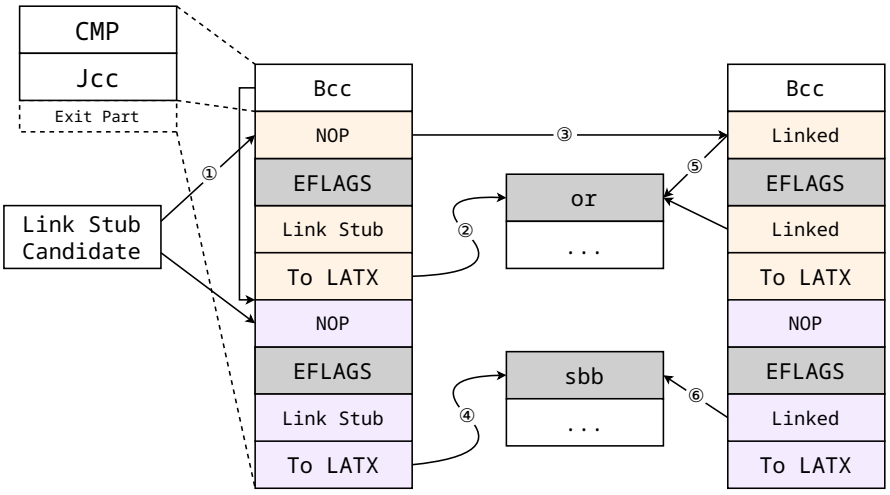

图 5.6 链接式跨基本块反馈优化

Figure 5.6 Linked Cross-Translation Block Feedback Optimization

选点；

3. 在执行基本块链接过程时，判断后续基本块的 EFLAGS 使用情况。如果后续基本块不需要使用前序基本块的 EFLAGS 信息（②），则可以执行优化操作，将 EFLAGS 计算指令前的链接候选点设置为链接点位（③），并执行基本块链接动作。为了保证基本块断链时刻的正确性，还需要将另一个候选点进行链接，但该点位的链接在运行时刻不会被执行到。在基本块运行时，该基本块会从 EFLAGS 指令前的候选点跳转到下一个基本块，此时 EFLAGS 指令不会被执行；

4. 如果后续基本块不需要使用前序基本块的 EFLAGS 信息（④），此时需要将 EFLAGS 计算指令后的链接候选点进行链接，此时 EFLAGS 计算指令前的链接候选点保持为 NOP 指令。

链接式优化不仅友好地处理多消除情况，而且还能有效地处理无法通过 LBT 指令计算的 EFLAGS。相比之下，消除式优化会将模拟指令修改为 NOP，由此带来的大量 NOP 指令会降低程序性能。

5.5　自修改处理和恢复

在实际应用中，常遇到存在自修改代码（Self-Modifying Code, SMC）的情况，如 Java、Javascript 等 JIT 程序。对于这些程序的翻译，LATX 由于存在代码缓存（Code Cache），所以需要捕获自修改情况，并对其翻译后的代码缓存进行无效，以保证自修改后的代码能够被正确执行。

但优化后的代码在出现自修改的情况下，会引发一些不一致的问题，如图 5.7 所示，展示了消除式优化发生自修改发生后，EFLAGS 因为被优化消除，导致后续基本块无法获取到正确 EFLAGS 信息的问题。

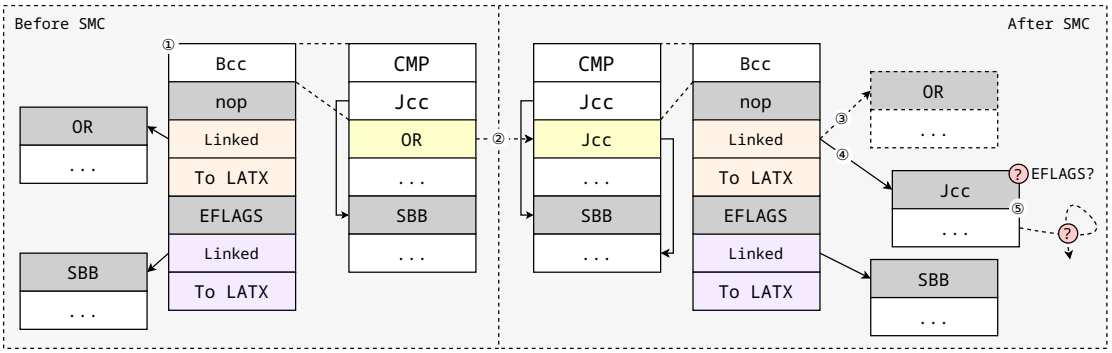

图 5.7 自修改后出现的 EFLAGS 丢失的问题

Figure 5.7 Problem: EFLAGS information lost after SMC

由于优化，原 EFLAGS 指令被替换为 NOP（①），当没有发生自修改时，基本块之间能够正确地计算并传递 EFLAGS 信息。但当发生自修改时，如后续基本块的 OR 指令被改为 Jcc 指令（②），翻译器将会断开原有跳转链接，并将发生自修改的基本块进行无效（③），以确保自修改的代码生效。

在下一次的基本块链接时刻（④），由于原 EFLAGS 指令被替换成 NOP 指令，导致其 EFLAGS 计算的信息丢失，因此，当下一个基本块需要使用 EFLAGS 信息时，就会发生错误（⑤）。例如，图中 Jcc 指令需要使用前序基本块的 EFLAGS 信息，但由于该信息已经丢失，因此无法判断条件跳转的跳转位置，从而导致错误发生。

为了解决这个问题，我们加入了自修改处理和恢复机制，在翻译过程中增加了保存 EFLAGS 信息的步骤，并在解链时进行 EFLAGS 指令的恢复，保证了自修改出现后优化部分不会影响后续执行。图 5.8 展示了优化后的自修改处理和恢复的过程。

1. 在翻译时，将原指令翻译优化时的 EFLAGS 指令保存在基本块最后，作为该基本块异常恢复时的 EFLAGS 的备份信息（⑤）；

2. 自修改产生时，异常处理程序会被调用，首先根据异常发生地址，获取基本块信息以及优化后的指令（①）；

3. 处理程序执行断链操作时，会查找该基本块的 EFLAGS 备份信息（②），

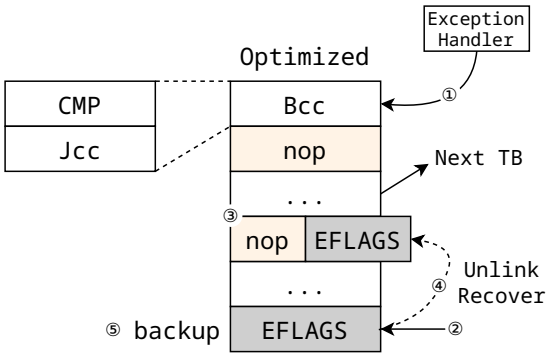

图 5.8 优化后的自修改处理和恢复

Figure 5.8 SMC Handling and Recovery for Optimized Code

并获取断链操作所在的出口的 EFLAGS 优化位置（③）；

4. 将备份的 EFLAGS 信息恢复到 EFLAGS 优化位置（④），即重新产生原 EFLAGS 指令。

在基本块断链后对 EFLAGS 计算指令进行恢复，使得基本块恢复到了未优化前的状态，此时再次进行基本块链接时，就可以继续执行跨基本块反馈优化，从而在保证后续执行正确的情况下，达到优化的目的。

对于链接式优化后自修改情况，由于是采用跳转方式对 EFLAGS 计算实现消除，其并未删除 EFLAGS 计算指令，所以只需按照原基本块解链方案，对两处链接点同时解链即可，处理相对简单。

5.6　性能分析

为了对优化效果进行评估，我们继续使用 SPEC CPU2000 测试程序，在已有的 **EFLAGS 延迟计算优化的基础上**，实现了“反馈式”语义化翻译方案。其性能如图 5.9 所示。

我们可以看到性能在 EFLAGS 延迟计算的基础上，绝对性能约提高了 3%。从图表中，我们可以看到 175.vpr 的性能提升最为明显，但其中也存在着性能下降的子项，如 186.crafty 和 255.vortex，这两个子项都有不同程度的性能下降，其性能下降绝对值约为 3%。

5.6.1　EFLAGS 消除分析

为了进一步分析各子项性能提高或降低的原因，我们对其 EFLAGS 消除情况进行了统计，如图 5.10 所示。

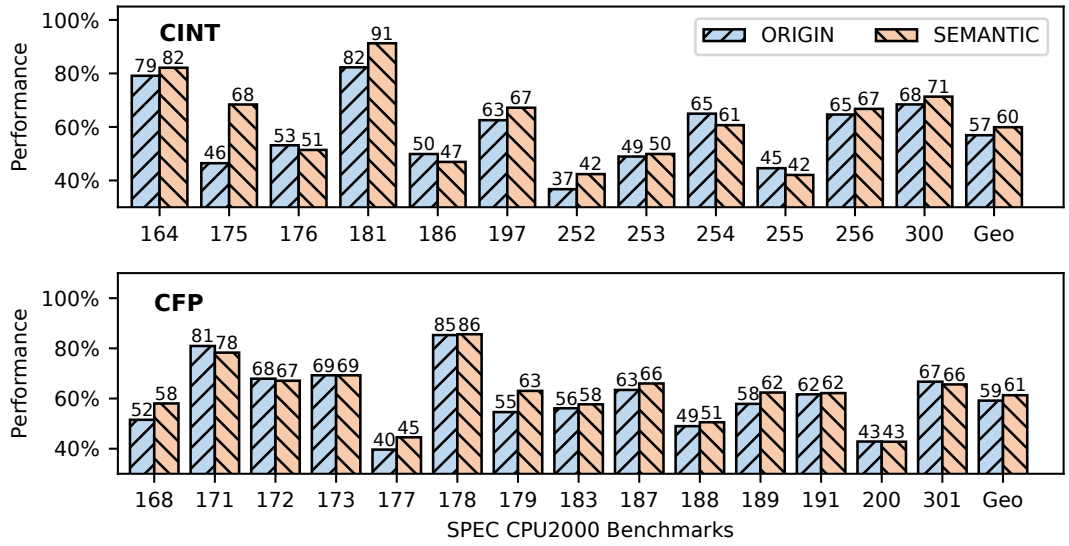

图 5.9 “反馈式”语义化翻译 SPEC CPU2000 性能

Figure 5.9 Performance of SPEC CPU2000 for Feedback Semantic Translation

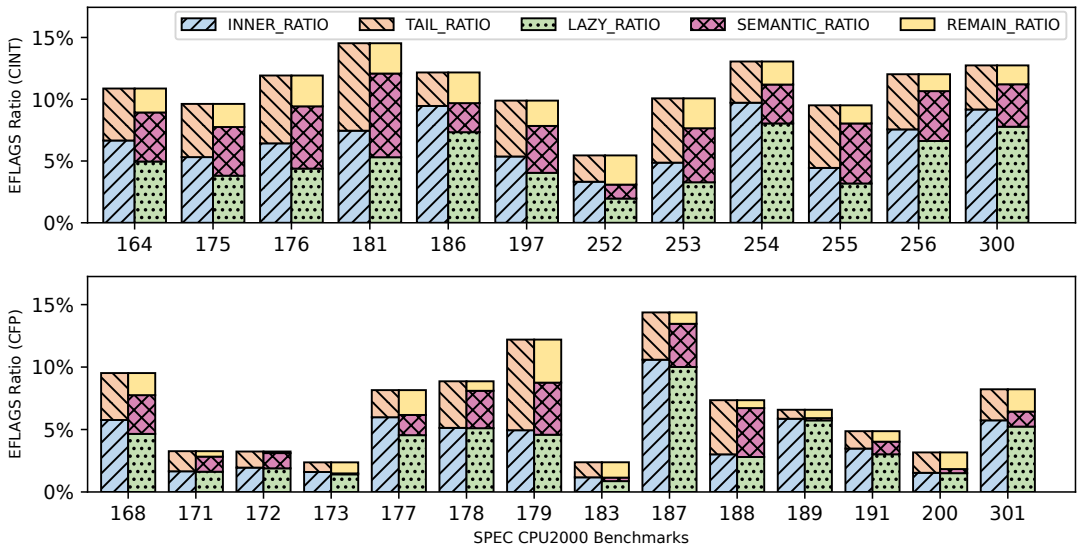

图 5.10 语义化翻译 EFLAGS 消除对比

Figure 5.10 Comparison of EFLAGS Elimination in Feedback Semantic Translation

从 EFLAGS 计算指令分布来看，对于 SPEC CPU2000 中的子项，其基本块内的 EFLAGS 计算数量（Inner Ratio）略多余基本块尾产生的 EFLAGS 计算数量（Tail Ratio）。从 EFLAGS 计算指令消除情况来看，我们可以发现对于 SPEC CPU2000 的子项，EFLAGS 延迟计算消除的 EFLAGS 指令数量（Lazy Ratio）要多于语义化翻译消除的指令数（Semantic Ratio）。这是由于优化前，基本块内的

EFLAGS 计算指令数往往多于基本块结尾需要生成的 EFLAGS 计算指令数量，所以相对来说 EFLAGS 延迟计算优化能够消除更多的 EFLAGS 计算指令。

从总体上来看，语义化翻译起到了对 EFLAGS 延迟计算的一个补充，使得指令消除的比例大约能占原来需要生成的 EFLAGS 指令的 80%，实现了大部分 EFLAGS 指令的消除。

5.6.2　各优化阶段性能分析

对于 175.vpr 和大多数 SPEC CPU2000 的子项来说，其性能升高主要来源于多余的 EFLAGS 消除带来的收益。但对于 186.crafty 和 255.vortex 等性能下降的程序来说，多余的 EFLAGS 消除反而带来了额外的开销，这似乎不是很合理。为了探索语义化翻译的问题，我们对它的优化阶段进行了性能测试，将优化分为普通 EFLAGS 指令优化和指令模拟 EFLAGS 运算优化两个部分。普通 EFLAGS 指令指使用了 LBT 扩展指令的 EFLAGS 计算，由于这种情况下只会产生一条 EFLAGS 计算指令，所以使用消除式优化方案；对于指令模拟 EFLAGS 运算的情况，由于其不支持 LBT 扩展指令，所以会产生多条指令对 EFLAGS 进行计算，主要是浮点比较指令 COMISX 等，其需要使用链接式优化方案。性能测试结果如图 5.11 所示。

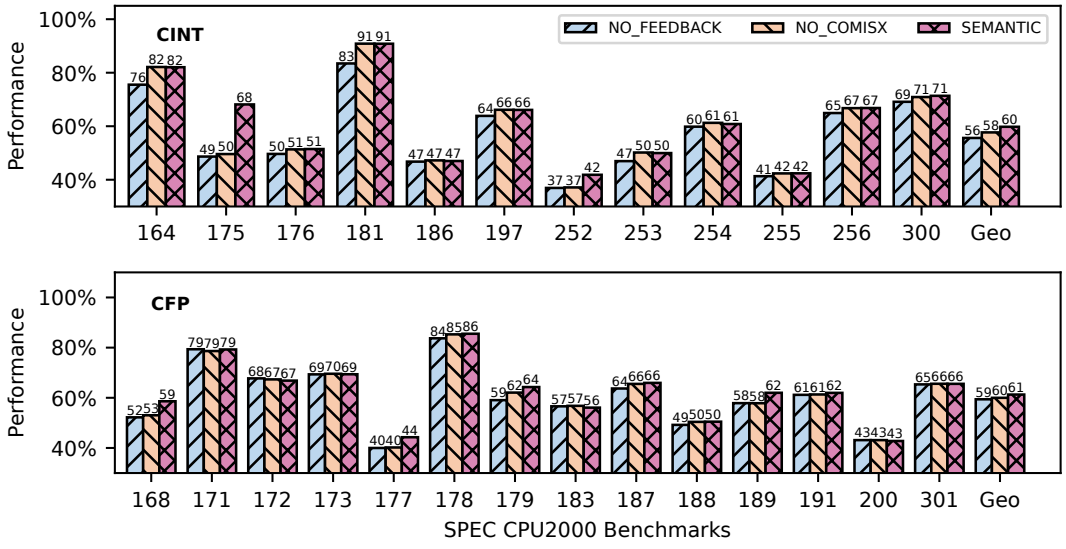

图 5.11 语义化翻译各阶段性能对比

Figure 5.11 Performance Comparison of Feedback Semantic Translation in each Stage

1. 对于多数的程序，其性能提升与 EFLAGS 消除的数量呈现正相关，如

164.gzip，175.vpr，181.mcf 等。从图中可以看出，大多数的程序性能提高来源于 EFLAGS 指令消除；但是，对于一些程序，如 175.vpr 和 252.eon 等，由于其包含大量浮点比较操作，所以指令模拟 EFLAGS 运算（COMISX 类指令）优化可以带来较大的性能提升。

2. 对于存在性能下降的 186.crafty 和 255.vortex 等程序，我们可以发现无论是否进行了 EFLAGS 的消除优化，其性能都无明显提升。一方面是由于其消除的 EFLAGS 指令占比偏少，其对性能的提升不会很明显；另一方面，其消除 EFLAGS 指令采用消除式优化方案，消除的 EFLAGS 使用 NOP 指令替换，其性能收益相比链接式优化方案相比会更低一些。当然，由于需要分别对每个基本块的出口加入 EFLAGS 计算指令，其产生的指令数会大于原有翻译的指令数，对 Cache 也存在一定的压力。

5.6.3 关闭跳转优化后性能分析

因为语义化翻译对基本块结尾的跳转方式有所改变，与翻译器已实现的跳转优化存在一定的冲突，因此在启用语义化翻译优化时，部分情况下无法开启跳转优化。为了更精准地分析语义化翻译方案带来的性能提升，我们在关闭跳转优化的情况下进行了性能优化测试，如图 5.12 所示。

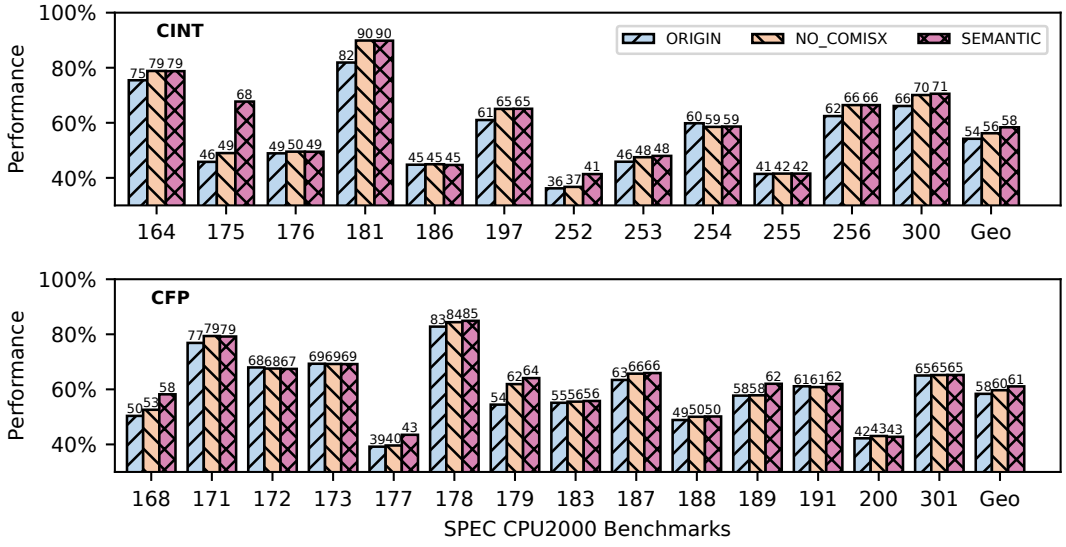

图 5.12 关闭跳转优化后语义化翻译性能对比

Figure 5.12 Performance Comparison after Closing Jump Optimization

我们可以看到，性能下降的 186.crafty 和 255.vortex 等程序，其性能在关闭

跳转优化后，与优化前（Origin）相比几乎没有变化，但相较于开启跳转优化，性能有所降低。对于这几个性能下降的程序，由于语义化翻译优化导致了部分基本块的跳转优化无法开启，影响了其初始的性能，再加上语义化翻译优化对这几个程序影响较小，所以形成了性能下降的趋势。

5.6.4　性能分析总结

我们将窥孔优化与本优化进行比较，从图 5.9，图 5.10 和图 5.11 的对比中，可以得出：

1. 如果只进行语义化翻译，不执行“反馈式”优化，其性能提升不明显。因此，对于窥孔优化而言，由于无法消除基本块结尾的 EFLAGS 计算指令，其优化所带来的性能提升相对较小。

2.“反馈式”语义化翻译中 EFLAGS 消除对于性能的收益要小于 EFLAGS 延迟计算的收益。

由于基本块结尾处存在多条分支指令，导致运行开销大多源于这些分支指令。一方面，语义化翻译仅仅减少了一部分指令间的冗余操作，而没有消除产生的 EFLAGS 计算指令，另一方面，EFLAGS 计算指令的开销要比分支指令小。所以无论是语义化翻译，还是翻译中的 EFLAGS 消除，其受到基本块结尾处分支指令的影响，其优化收益都有所下降。

5.7　本章小结

本章详细介绍了“反馈式”语义化翻译优化方案，该方案为解决翻译过程中，基本块内多指令间 EFLAGS 计算重复，以及基本块结尾 EFLAGS 需要生成两个问题进行优化。与窥孔优化相比，语义化翻译放宽了优化限制，扩大了优化范围，从而实现了更高的优化收益。

语义化翻译解决了基本块内多指令间 EFLAGS 重复计算的问题，通过将多指令融合，按照其语义翻译，可以使得多指令间对 EFLAGS 的处理从“计算-使用-计算”的模式削弱成基本块内的一些简单运算逻辑，不再需要多次的 EFLAGS 计算。同时语义化翻译还对单指令翻译进行了优化，对于那些单指令翻译膨胀率较大的指令，采用多指令的语义化翻译很可能得到更优的收益。

语义化翻译通过将多条指令融合，按照其语义重新翻译，解决了基本块内

EFLAGS 重复计算的问题。它可以使得多指令间对 EFLAGS 的处理从“计算-使用-计算”的模式变为一些简单的运算逻辑，不再需要多次的 EFLAGS 计算。此外，语义化翻译还可以对单条指令翻译进行优化，减少那些具有较大翻译膨胀率的指令的不必要的计算，从而提高系统的性能。

另外，为了克服基本块末尾需要生成 EFLAGS 的难题，在语义化翻译的基础上增加了跨基本块的反馈优化机制，从而动态地去除基本块结尾处 EFLAGS 的计算，缓解了基本块结尾处需要生成 EFLAGS 的问题。

最后，本章对“反馈式”语义化翻译优化方案的性能进行了评估，并对其中子项的性能下降以及相比 EFLAGS 延迟计算性能提升较低等进行了分析和实验。虽然该优化会导致部分基本块无法进行跳转优化，但其总体上，与 EFLAGS 延迟计算相比，绝对性能也有 3% 左右的提高。

第 6 章　SSE 标量指令高位运算消除

6.1　引言

X86 SSE 的浮点标量运算指令被设计用来取代 X87 指令，为了保持向前兼容，它只会对向量寄存器的低位数据进行运算，而高位的数值会保持不变。而在 LoongArch、AArch 等指令集中，类似的指令只会将低位运算结果计算出并存入目标寄存器，但其向量寄存器高位数据无法保证维持原值。因此，为了翻译这些指令，需要对每条指令的高位进行“保存-恢复”操作，以确保向量寄存器的高位数据不会发生变化。

但是，每一条标量运算指令都需要进行的” 保存-恢复” 操作所带来的开销是比较大的。如图 3.8 中 ADDSD 指令的翻译，其语义为对向量寄存器低 64 位的值进行加法运算，并将结果存储回目标寄存器的低 64 位中，同时保持目标寄存器高 64 位的值不变。对于 LoongArch 翻译来说，需要实现以下操作：

1. 读取目标寄存器高位数值，并将其保存在临时寄存器中；
2. 使用浮点加法指令 FADD.D 进行加法运算，结果存目标寄存器；
3. 将 1 中读取的高位寄存器数值，重新插入到目标寄存器的高位。

所以为了消除这些标量运算指令内冗余的“保存-恢复”操作，我们设计了 SSE 标量指令高位运算消除算法（Scalar Higher Bits Remove, SHBR）。在 6.2 节中，我们将会介绍 SHBR 的整体架构设计，并在 6.3 节阐述基本块分类方案，我们将会对基本块进行分类，根据不同的类型采取相应的优化措施。在 6.4 节我们将会对优化算法的各个部分进行详细的介绍。最后，我们将在 6.5 节进行性能分析，以评估优化算法所带来的性能提升情况。

6.2　整体架构

我们设计的 SHBR 优化算法整体架构如图 6.1 所示。为了删除标量浮点运算的“保存-恢复” 操作，同时保证优化后的程序结果与优化前一致，首先要对基本块进行指令流和数据流分析（①）。优化探针将会按照基本块粒度，对基本块内的指令执行数据流分析（②），分析出每一个向量寄存器高位数据的来源和数值，并将其存储在优化记录区域，同时根据分析的结果得出该基本块的类型（③）。

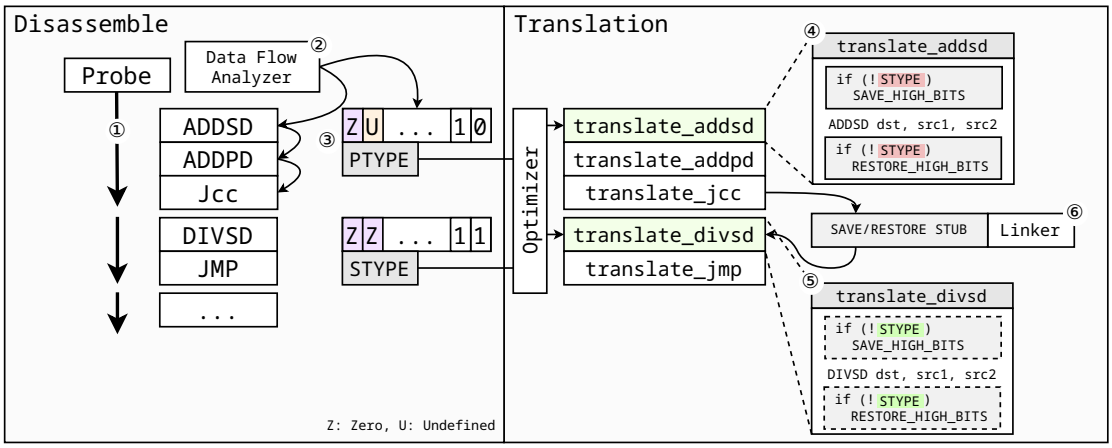

图 6.1 SHBR 整体架构

Figure 6.1 Overall Architecture of SHBR

在翻译阶段，翻译函数会根据基本块类型，进行对应“保存-恢复”操作的消除。对于不能消除的基本块，保留基本块内的所有指令的“保存-恢复”操作（④）；而对于可以消除的基本块，则可以将其内的所有指令的“保存-恢复”操作进行消除（⑤）。

另外，在基本块间链接时，对于不同类型的基本块间，需要插入特殊处理的指令片段，以保证基本块间的正确切换（⑥）。此外，这些用于特殊处理的指令片段会引起额外的开销，可以对这些指令片段继续进行优化，以消除不必要的处理操作，以此来减少开销。

6.3 基本块分类

为了实现相应的优化，第一步就需要对翻译的基本块进行分类，以实现对每种类型的基本块采用不同的优化方案。我们将基本块分成 S（Scalar，标量）、P（Packed，向量）和 N（None，普通）三种类型，其关系如图 6.2 所示。具体的基本块分类算法将会在 6.4.2 节详细介绍。

1. S 类型：基本块内都是标量运算指令，或者存在有一定的向量运算指令，但其向量寄存器数据高位数据不会影响低位结果，且高位结果是确定值（0 值或从其他寄存器搬运）或会被后续运算覆盖；

2. P 类型：基本块内存在向量运算指令，且不符合 S 类型的定义；

3. N 类型：基本块内不存在 SSE 向量指令，包括了标量运算指令和向量运算指令。

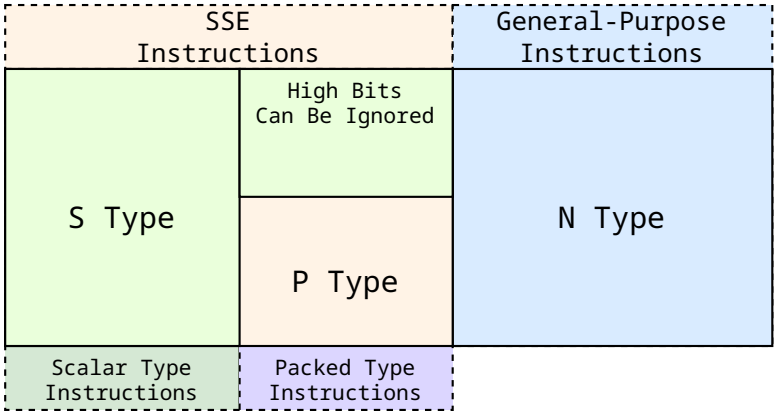

图 6.2 SHBR 算法对基本块的分类

Figure 6.2 Classification of Translation Block in SHBR

在将基本块进行分类后，由于 S 类型基本块对于寄存器高位数据并不关心，所以我们将会对 S 类型基本块执行冗余“保存-恢复”操作的消除优化；而 P 类型基本块需要使用高位的数据并产生相应的计算结果，所以对 P 类型不做任何优化操作。

6.4　向量寄存器高位消除优化

为了实现向量寄存器高位消除，我们按照翻译的流程，将其分为五个阶段：

1. **向量寄存器高位信息计算**。通过指令流和数据流分析，识别基本块内各向量寄存器数据来源，并将其标记在基本块的信息内。同时将本基本块内向量寄存器高位为 0 的信息传递到下一个基本块，作为下一个基本块向量寄存器高位信息的初始值，继续进行向量寄存器数据来源计算。

2. **基本块类型识别**。通过分析基本块内指令类型以及 1 中计算出的向量寄存器高位的信息，将基本块分类成标量基本块（S 类型），向量基本块（P 类型）和普通基本块（N 类型）。

3. **向量寄存器高位“保存-恢复”操作消除**。对所有的基本块按照分类进行优化，消除所有 S 类型基本块的向量寄存器高位“保存-恢复”操作，保留 P 类型基本块的向量寄存器高位的“保存-恢复”操作，N 类型基本块不存在向量指令操作向量寄存器，无需进行处理。

4. **基本块链接**。基本块链接时候需要进行必要的处理以保证优化后正确性。相同类型基本块链接采用直接链接方式，不需要进行特殊处理；S 类型链接到 P

类型需要对向量寄存器高位数据进行加载；P 类型链接到 S 类型需要对向量寄存器高位数据进行保存。P 或 S 类型链接到 N 类型则不需要进行保存或恢复动作，同时 N 类型基本块继承链接者的类型，即根据链接者的类型，转变自身的基本类型。

5. **链接时优化**。在进行基本块的链接时需要加入链接处理动作，该动作需要将向量寄存器高位进行保存或恢复。但实际上根据步骤 1 得到的信息，我们可以进行相应的优化，减少处理动作所需要的操作数量，有以下两种优化场景：

(a) 如果链接基本块是从 S 类型到 P 类型，同时 P 类型基本块初始值存在高位为 0 的向量寄存器，那么这些寄存器在 P 类型的基本块中不需要加载；

(b) 如果链接基本块从 P 类型到 S 类型，同时 S 类型基本块退出时存在高位为 0 的向量寄存器，那么这些寄存器在 S 类型的基本块前不需要保存。

6.4.1 向量寄存器高位信息计算

向量寄存器高位信息计算分成**向量寄存器高位数据分析**和**基本块间向量寄存器信息传递**两个阶段。在分析结束后，会将分析得到的各个向量寄存器高位的信息保存在此基本块的优化信息域内，用于此基本块类型的识别以及后续的优化过程。

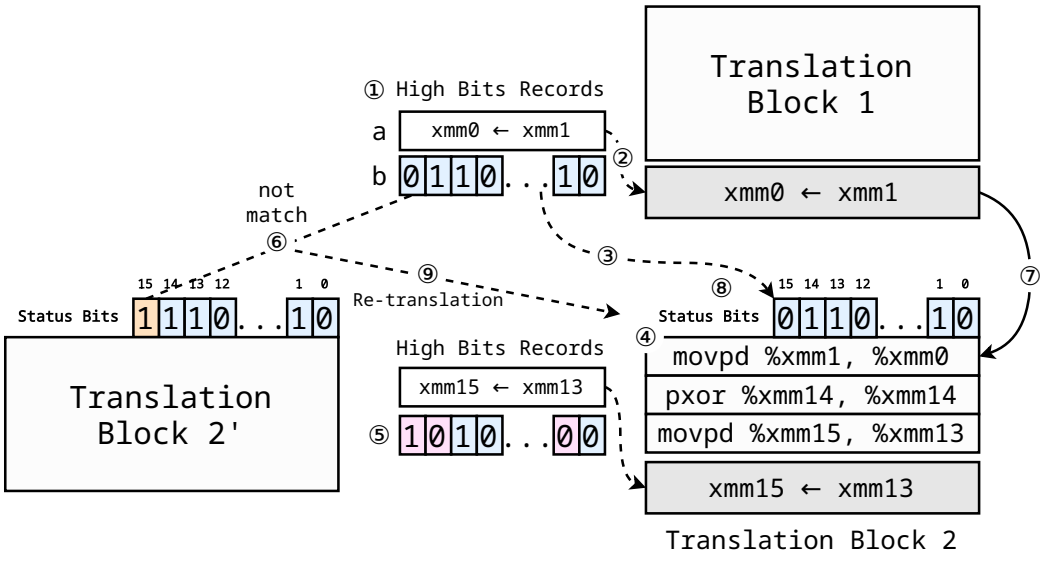

图 6.3 向量寄存器高位信息计算实例

Figure 6.3 Example of Vector Register High Bits Information Calculation

1. **向量寄存器高位数据分析**

向量寄存器高位数据分析与数据流分析类似，按照指令流顺序，根据指令语义，逐一模拟计算每一条指令中目标寄存器的高位数据，并标记来源。为了提高翻译性能，相比于数据流分析，目标寄存器的高位数据可以简化成为三种状态：分别为 0 状态，传递状态（由其他寄存器的高位数据传递得到），以及未定义状态。未定义状态指那些高位数据无法确认数据来源的情况，如从内存中加载数据到向量寄存器中。

如图 6.3 所示，在基本块 2 内（④），xmm1 寄存器的高位会被 movpd 指令设置为 xmm0 寄存器的高位数据，同时，前序分析过程中得出 xmm0 寄存器的高位为 0 状态，所以此时 xmm1 寄存器高位会被记录成 0 状态；xmm14 的高位会因为 pxor 指令而清零，所以 xmm14 寄存器高位会被记录为 0 状态；xmm15 寄存器的高位会被 movpd 指令设置为 xmm13 寄存器高位的值，所以 xmm15 寄存器的高位会被记录传递状态（由 xmm13 高位数据传递得到）。

2. 基本块间向量寄存器信息传递

在完成一个基本块的高位数据分析后，其每一个向量寄存器的高位状态都可以被单独确定，这些信息会被记录到基本块的信息域中，如图 6.3 ①所示，该域记录了每一个向量寄存器高位数据的状态以及来源（非 0 状态时）。①.b 中记录的是状态信息，0 表示 0 状态，1 表示传递状态或者未知状态。①.a 中记录的是传递状态寄存器的数据来源。其中，根据向量寄存器高位是否为 0 状态，形成一个标记向量，也称状态字（①.b），并传递给下一个基本块（③）。寄存器高位数据传递（①.a）记录了除高位状态为 0 状态外的寄存器高位的情况，该域中的记录指明了寄存器的高位数据的传递方向。如果发生了数据传递，则在基本块结尾处形成搬运指令，完成对高位数据的搬运操作（②）。

在进行下一个基本块翻译时，会读取上一个基本块传递来的状态字（③），并将其记录在此基本块的状态字中，作为此基本块的一个状态进行维护（⑧）。在基本块查找时刻，该状态字会作为一个键值参与比较，只有具有相同键值的基本块才能被查找到并执行（⑦），如果出现不匹配的情况（⑥），则按照查找的状态字作为该基本块初始状态，进行重新翻译（⑨）。

此外，在初始化当前基本块的向量寄存器高位时，会使用该状态字进行初始化。如果某个寄存器的高位在上一个基本块退出时状态字记录是 0 状态，则在当前基本块初始化时将该寄存器的高位信息标记为 0 状态，并进行向量寄存器高

位数据分析的工作。例如，⑧中状态字第 0 位标记了 xmm0 寄存器高位状态，其记录 0 说明前序基本块（基本块 1）退出时 xmm0 的高位为 0。所以该基本块初始化时 xmm0 高位记录会被初始化为 0 状态，其指令 movpd 会将 xmm0 高位数据搬运到 xmm1 的高位，所以此刻 xmm1 的高位也会被记录为 0 状态。该基本块完成分析后，可以得到 xmm1 的高位记录为 0 状态（⑤），该记录来源于 xmm0 高位，即初始化记录 0 状态。

6.4.2 基本块类型识别

基本块会按照指令流和数据流分析结果，将基本块分成 S、P 和 N 三种类型。由于标量运算指令只含有算术运算指令，如 ADDSS 等，对于逻辑运算指令，X86 采用了向量运算指令进行代替。所以如果只进行指令类型的分析，大部分基本块都将会是 P 类型，此刻无法起到很好的优化效果。所以为了避免这种情况的发生，我们分析时还会进行数据流分析，根据每一个寄存器高位的状态，以确认更多能够被判定为 S 类型的基本块，分析方法如下：

1. 如果基本块内指令不存在向量指令，则该基本块为普通基本块（N 类型）；

2. 如果基本块内指令都为标量运算指令（SSE 指令中不影响高位的指令），则该基本块为标量基本块（S 类型）；

3. 如果基本块内存在向量运算指令，则进行数据流分析，判断是否能被确定为 S 类型基本块：

(a) 如果基本块内存在向量运算指令使用未定义的高位数据（如从内存中加载数据到向量寄存器），且该高位数据会对目的向量寄存器低位数值产生影响，则认为是 P 类型基本块；

(b) 对高位数据按照逐条指令语义进行运算，如果基本块结束时刻，分析得到的目的向量寄存器高位为未定义值（不为 0 状态或者传递状态），则认为是 P 类型基本块；

(c) 其余的都认为是 S 类型基本块。

数据流分析实例如图 6.4 所示。对于存在向量运算指令（PXOR 以及 ANDPS）的基本块，① 中由于 xmm1 寄存器的高位数据被 PXOR 指令清零，所以此时的 ANDPS 指令中 xmm0 寄存器高位值虽然未知，但按照其指令语义运算后，xmm0 寄存器高位与 xmm1 寄存器高位计算后，结果为确定值 0，所以不符合 3b ，同时其不确定的高位数据（xmm0 高位）并未影响低位数值，所以不符合 3a ，此基

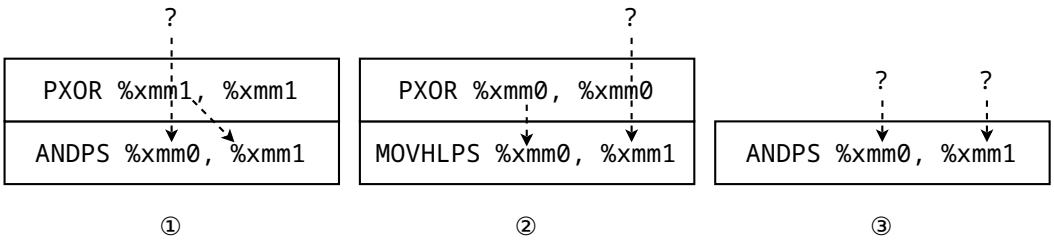

图 6.4 基本块类型识别实例

Figure 6.4 Example of Translation Block Type identification

本块符合 3c，被定义为 S 类型；② 中由于 xmm1 寄存器高位值是不确定的，且会被指令 MOVHLPS 搬运到 xmm0 低位，所以符合 3a，该基本块被定义为 P 类型；③ 中 xmm0 和 xmm1 寄存器高位均不确定，运算结果的高位也为不确定值，所以符合 3b，该基本块被定义为 P 类型。

6.4.3　高位“保存-恢复”操作消除

在 3.2.2 部分，我们介绍了标量运算指令的普通翻译方式，它通过“保存-恢复”操作，保护了向量寄存器高位数据。如图 3.8 所示，其中过程 ① 和 ④ 是进行的“保存-恢复”操作。为了提高运行速度，我们可以通过消除该操作，仅保留核心的加法指令，以减少生成后指令的膨胀度。

在基本块识别完成后，其将被划分为 P/S/N 三类，每类可根据不同的优化方案进行优化：

1. **P 类型基本块**。该类型基本块被定义成需要保证向量寄存器高位的基本块，基本块将不会进行优化动作，对于其中存在的标量运算指令，保留其向量寄存器高位“保存-恢复”的动作。

2. **S 类型基本块**。该类型基本块被定义为无需保证向量寄存器高位的基本块，基本块会执行优化动作，对于其中存在的标量指令，其向量寄存器高位“保存-恢复”动作会被删除；对于其中可能存在的向量运算指令，按照原始的向量运算指令翻译方案进行翻译。虽然这些向量指令会对高位产生影响，但由于前序基本块划分分析中，这种类型的指令的影响只可能存在于一个极小的片段内或者其数据本身就无意义，因此不会对基本块的正确性产生影响，可以无视其高位的情况。

3. **N 类型基本块**。该类型基本块中不存在 SSE 向量指令，所以无需进行向

量寄存器高位“保存-恢复”操作消除。

6.4.4 基本块链接

基本块链接是二进制翻译中最基本的优化操作，通过基本块的链接会将同一个控制流上的基本块连接在一起，减少上下文切换以及查找下一个基本块的开销。由于基本块被分类优化，所以链接时刻需要对不同类型的基本块进行处理，处理流程如图 6.5 所示。

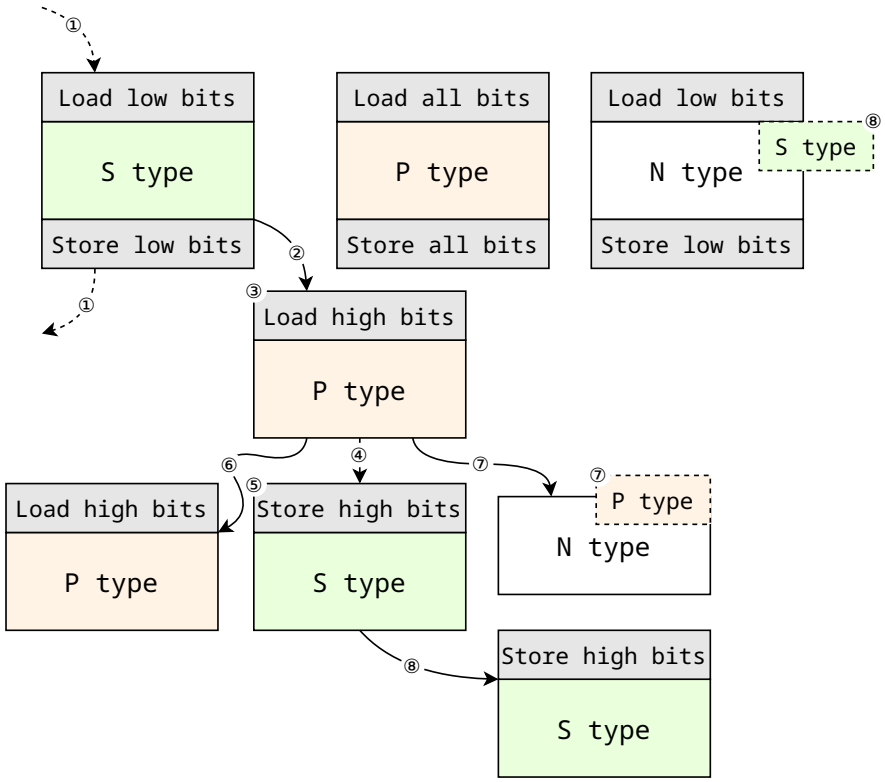

图 6.5 基本块链接处理示意图

Figure 6.5 Translation Blocks Link in SHBR

1. **相同类型基本块链接**。由于基本块类型相同，其对于向量寄存器高位的影响是相同的，所以无需特殊处理，即按照正常基本块链接动作进行，无需加入修正指令（⑥）。

2. **N 类型基本块链接**。N 类型基本块内未使用任何的 SSE 向量指令，所以对于向量寄存器的高位并不关心，所以无论是按照 P 类型进行处理还是按照 S 类型进行处理都不存在任何问题。

为了减少链接时进行特殊处理的开销，N 类型的基本块会继承链接者的基本块类型，即前序的基本块类型。所以 N 类型基本块可以分为继承 S 类型的 N

类型基本块和继承 P 类型的 N 类型基本块两类。例如此时是 S 类型的基本块链接到 N 类型基本块，N 类型基本块会被修改成 S 类型的基本块，并按照相同类型基本块链接 进行链接（⑦），此时基本块类型为继承 S 类型的 N 类型基本块，后续按照 S 类型基本块处理。

3. **P 类型基本块链接到 S 类型基本块**。P 类型基本块会维护向量寄存器的高位数据，而 S 类型基本块对向量寄存器高位的数据并不关心，所以直接链接会导致向量寄存器的高位数据因为运行了 S 类型的基本块而丢失。从 P 类型到 S 类型基本块链接需要对寄存器高位数据进行保存，具体处理操作如下：

(a) S 类型基本块会在翻译后代码前生成向量寄存器高位保存处理头，用于保存所有的向量寄存器高位（⑤）；

(b) 在基本块链接时刻，会动态识别前一个基本块类型，如果发现前一个基本块类型是 S 类型，即与本基本块类型一致，按照相同类型基本块链接的方式，不需要执行特殊处理，此时链接会跳过高位保存处理头，直接链接到基本块的翻译后代码处（⑧）；

(c) 如果前一个基本块类型是 P 类型，则需要对寄存器高位数据进行保存，则链接到向量寄存器高位保存处理头处，运行时刻会先经过向量寄存器高位的保存操作，将向量寄存器中高位的数据保存到内存中的向量寄存器高位后，再运行 S 类型基本块翻译后代码（④）。

4. **S 类型基本块链接到 P 类型基本块**。S 类型基本块链接到 P 类型基本块需要加载向量寄存器高位的数据，其具体操作与P 类型基本块链接到 S 类型基本块类似：

(a) P 类型基本块在翻译后代码前生成向量寄存器高位加载处理头，用于加载所有向量寄存器的高位（③）；

(b) 在基本块链接时刻，动态识别前一个基本块类型，基本块类型一致（P 类型）则直接连接到基本块的翻译后代码处（⑥）；

(c) 如果前一个基本块类型是 S 类型，则链接到向量寄存器高位加载处理头处，运行时刻会先经过向量寄存器高位的加载动作，将内存中的向量寄存器高位的值加载入向量寄存器后，再运行 P 类型基本块翻译后代码（②）。

6.4.5 链接时优化

不同类型基本块链接时刻会对向量寄存器高位数据进行保存或恢复，该部分可以根据向量寄存器的状态进行优化，减少保存恢复指令的生成。

1. 获取本基本块的高位数据初始化信息，即前一个基本块的高位数据分析结果；

2. 获取本基本块的高位数据分析结果，对每一个向量寄存器高位的分析数据，执行：

(a) 对于 P 类型链接到 S 类型的情况，如果分析结果为 0 状态，则说明该向量寄存器的高位在基本块执行完成后结果为 0，保存的高位数据会被该基本块执行结果所覆盖，无需进行保存操作，可以将该向量寄存器高位保存指令消除；

(b) 对于 S 类型链接到 P 类型的情况，如果初始化信息为 0 状态，可以推迟该向量寄存器高位的清零动作，只有在使用时才执行相应的清零操作，此时可以将该向量寄存器高位加载指令消除。

6.5 性能分析

6.5.1 整体性能分析

经过我们的实验，发现在将寄存器高位“保存-恢复”操作完全消除的情况下，SPEC CPU2000 程序仍能够正常运行，可以认为该状态是“保存-恢复”动作消除能够优化到的上限。此外，我们在**“反馈式”语义化翻译的基础上**实现了 SHBR 优化，为了检验其效果，我们将 SHBR 优化后的结果与“保存-恢复”操作完全消除的结果（SHBRX）以及优化前结果（Origin）进行了对比，如图 6.6 所示。

1. **SHBR 优化对浮点运算敏感**。从图中我们可以看出，即使是高位“保存-恢复”操作完全消除，定点的性能并没有明显的提升，对于浮点的测试来说，高位“保存-恢复”操作的消除会带来较大的性能提升。这是因为 SHBR 优化去除的是 SSE 指令标量运算中的冗余操作，而这些指令用于替代 X87 浮点运算，所以在浮点程序中才会有明显的改善。

2. **SHBR 优化开销较大**。对于定点程序以及浮点的 168.wupwise，SHBR 优化会导致其性能下降，主要是由于部分浮点运算需要使用向量运算指令，这会导致不同类型的基本块之间频繁切换。不同类型基本块间切换需要对高位数据进

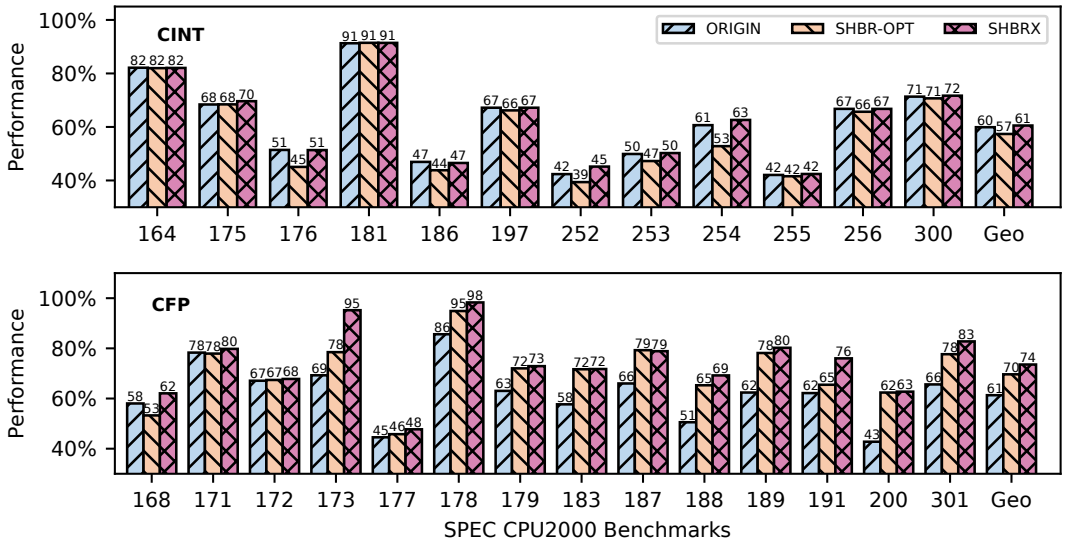

图 6.6 SHBR 优化 SPEC CPU2000 性能结果

Figure 6.6 SPEC CPU2000 Performance Results on SHBR

行保存/恢复，会带来较多的访存操作，导致性能下降。

3. **SHBR 对浮点的优化收益可观**。对于浮点程序，SHBR 优化给出了较好的优化效果，其绝对性能提高了 9%，相比优化前性能提高 15% 左右。与完全消除的情况相比，其性能相差 4% 左右，还有部分的提升空间。这主要是因为不同类型基本块间的切换会带来较大的性能损失；同时，基本块类型分析使用了相对保守的算法，会有部分基本块被识别成向量类型基本块（P 类型），该类型基本块中包含的标量运算指令“保存-恢复”操作不会被消除。这导致了 SHBR 优化暂时还无法达到其上限性能。

6.5.2　基本块切换和 SHBR 消除分析

为了探究性能下降的具体原因，我们对 SHBR 优化进行更详细的分析，统计基本块切换数据。图 6.7 给出了不同类型基本块切换次数占总块间跳转次数的比例，图 6.8 给出了不同类型基本块切换使用的指令占动态指令数的比例。

从图中我们可以看出，从 S 类型基本块切换到 P 类型基本块（S2PTYPE）的次数稍多于从 P 类型切换到 S 类型基本块（P2STYPE）的次数。我们对比性能出现较大下降的子项 252.eon，可以发现其存在较多的基本块类型切换操作，约 15% 的基本块跳转需要处理基本块类型的切换，影响了其翻译后程序。

除了 252.eon 程序之外，在定点程序中，例如 186.crafty 和 197.parser 等，尽

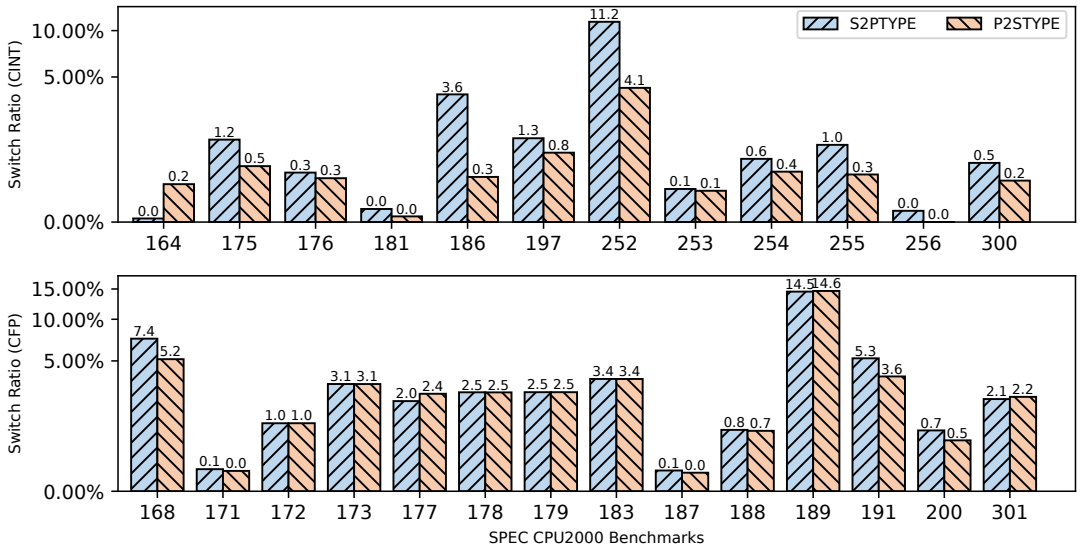

图 6.7 不同类型基本块在 SPEC CPU2000 中的切换频率

Figure 6.7 Switching Frequency of Different Types of TB in SPEC CPU2000

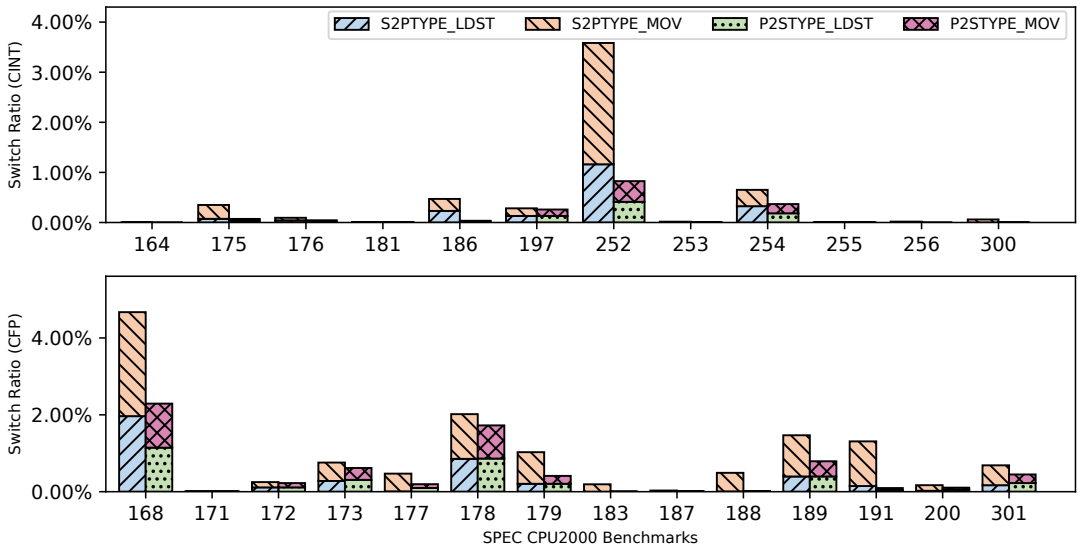

图 6.8 SPEC CPU2000 中不同类型基本块切换使用指令类型

Figure 6.8 Instruction Types Used for Different Types of TB Switching in SPEC CPU2000

管它们存在相当多的切换操作，但是这些基本块切换所使用的指令数在总动态指令数中的比例较低，因此对它们的性能影响相对较小。然而，对于浮点程序中的 189.lucas 等，尽管它们存在较多的基本块类型切换操作，并且切换使用的指令数量相对较高，但性能却没有特别下降；另一方面，定点程序中的 176.gcc 性能下降较为显著，然而与其他子项相比，其基本块类型切换操作的数量并不算很多。所以我们对基本块的类型和 SHBR 指令消除情况进行了分析，如图 6.9 和

图 6.10 所示。我们可以得出以下结论：

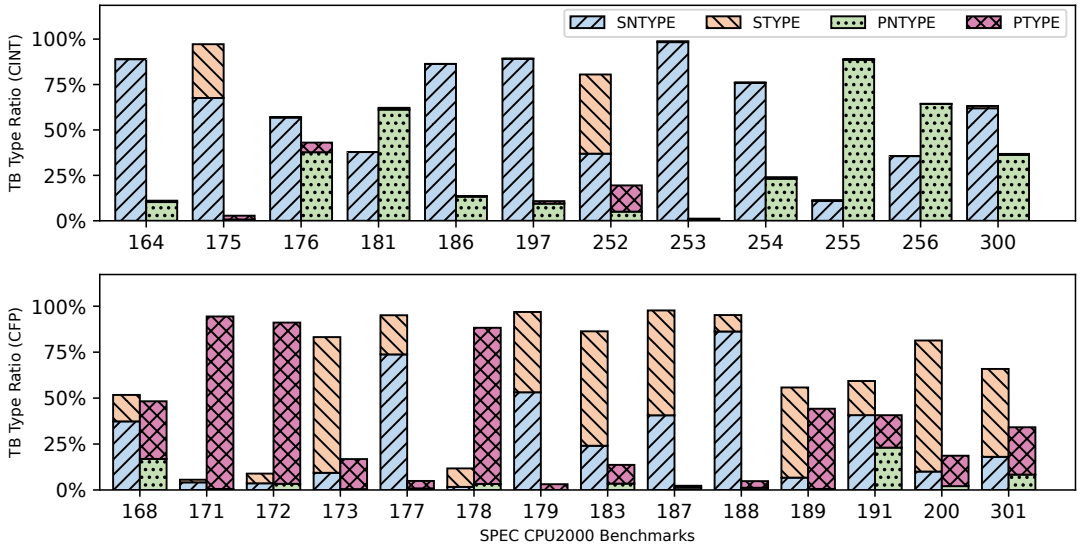

图 6.9 不同类型基本块在 SPEC CPU2000 中的比例

Figure 6.9 Proportion of Different Types of TB in SPEC CPU2000

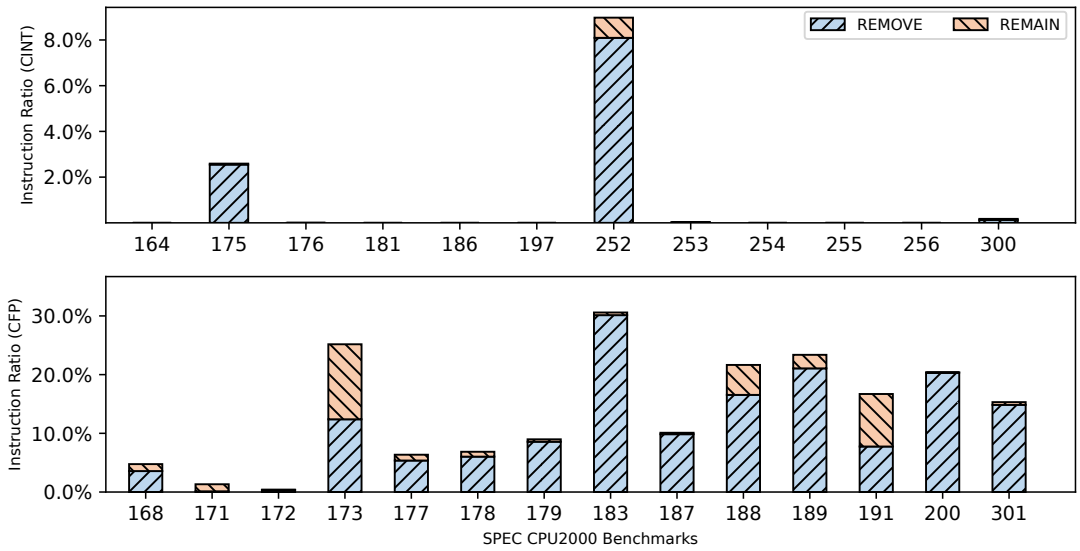

图 6.10 SPEC CPU2000 中 SHBR 消除比例

Figure 6.10 SHBR elimination ratio in SPEC CPU2000

1. 在定点程序中，N 类型基本块占据了大多数，该基本块中不包含任何向量指令。为了深化研究，我们将 N 类型进行分类，按照其继承的前序基本块类型，将其分为继承 S 类型的 N 类型基本块（SNTYPE）和继承 P 类型的 N 类型基本块（PNTYPE）。从图 6.9 和图 6.10 中可以看出，在定点程序中，N 类型的基

本块在 P 和 S 两种类型中占据了大部分，因此能够消除的“保存-恢复”操作相对较少。

2. SHBR 算法对 175.vpr 和 252.eon 程序消除了较多的“保存-恢复”操作。与 175.vpr 的性能相比，176.gcc 的性能降低较高，其中部分原因在于 175.vpr 程序对“保存-恢复”操作进行了消除。对于 252.eon 程序来说，其由于存在过多的基本块类型切换操作，其执行的切换指令过多，所以“保存-恢复”操作消除并不能给其带来性能提升。

3. 浮点程序中 N 类型基本块相对占比减少，S 类型基本块数量增多，特别是 173.applue，183.equake，以及 200.sixtrace，从 SHBR 消除比例看，这几个 S 类型基本块数量较多的程序也消除了较多的“保存-恢复”操作。

4. 浮点程序中大多数的“保存-恢复”操作都能被消除，且性能提升主要由消除贡献。对比图 6.6 中“保存-恢复”操作被完全消除的情况（SHBRX），173.applu 性能差距较大的原因主要是因为“保存-恢复”操作消除的不够彻底。

5. SHBR 消除数量占比较多的程序，如 183.equake，188.ammp，189.lucas 以及 200.sixtrace，301.apsi，这几个程序的性能提升都十分明显。173.applu 和 191.fma3d 由于其“保存-恢复”操作消除比例较低，所以其性能与完全消除情况存在一定差距。

6. 对于拥有较多基本块切换指令的子项，其优化后的性能会有明显的下降。如 252.eon 和 168.wupwise，其性能比优化前分别下降了 4% 左右，由于 168.wupwise 程序消除的“保存-恢复”操作更少，因此其性能下降更为明显。

综上所述，SHBR 算法虽然可以消除大多数的“保存-恢复”操作，但对于 SPEC CPU2000 中一些子项，由于优化分析窗口的限制，导致其部分“保存-恢复”操作消除未能完全消除，再加上存在基本块类型的切换，从而使其性能与完全消除情况还存在差距。虽然算法引入了部分的开销，但其浮点测试的平均性能仍有 9% 的提高，这表明优化算法对于浮点密集型程序有较大的的性能提升。

6.5.3 链接时优化分析

为了探索 SHBR 优化中，对于不同类型基本块链接时优化情况，我们对比了优化前后的数据，如图 6.11 所示。

经过优化，部分子项的性能提升明显，但大多数子项的提升幅度较小。这主要是因为基本块类型切换占总基本块动态运行的比例较小，多数子项的切换比

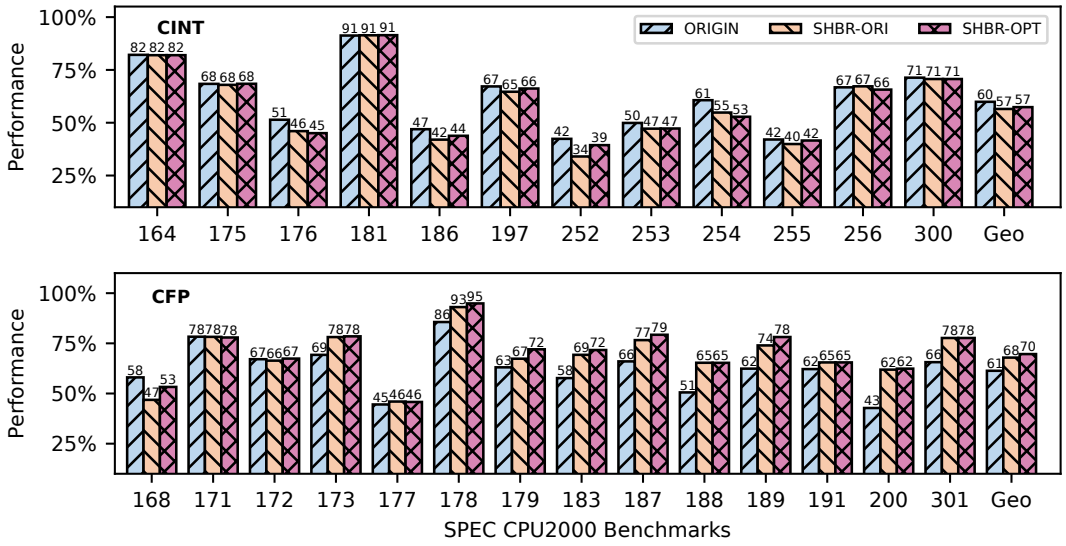

图 6.11 不同类型基本块链接时优化 SPEC CPU2000 性能结果

Figure 6.11 SPEC CPU2000 Performance Results when Linking Different Types of TB

例低于 6%。其中，切换比例最多的 252.eon 以及 168.wupwise 在进行链接优化后，其性能均有一定提升，绝对性能提升量达到 5% 以上。

6.6　本章小结

本章详细介绍了 SSE 标量指令高位运算消除优化算法的设计与实现，解决了 ARM、RISC-V 以及 LoongArch 等指令集中因指令差异导致的需要添加额外“保存-恢复”操作的问题。通过将基本块分类，根据基本块类型实行不同的优化方式，将“保存-恢复”操作尽可能的消除，以达到提高浮点运算的性能。

基本块分类不仅仅通过基本块内指令类型进行判断，而是会进行数据流分析，并根据数据流分析结果得出基本块的类型。这样可以获得更多的标量类型（S 类型）基本块，从而可以去除这些 S 类型基本块内的“保存-恢复”操作，以此可以获得更高的性能收益。对于剩余的向量类型（P 类型）基本块，其含有较多的向量指令，所以高位的“保存-恢复”动作消除会带来更大的性能损失，不进行优化。

为了确保不同类型基本块之间的正确链接和跳转，优化过程中还会处理不同基本块间的相互跳转问题。为此，在不同类型基本块之间加入保存/恢复向量寄存器高位的操作，实现正确的基本块切换。同时，为了保证切换的效率，还根

据数据流分析的结果，对不同类型基本块间的处理操作进行优化，消除不需要的保存/恢复操作。

最后，我们针对 SHBR 算法进行了性能的测试和分析，得出该算法对于浮点的程序提升效果较为明显，绝对性能提升超过 9%。对于定点运算程序，由于其中较多的不同类型基本块的切换操作，导致其性能会有略微下降。由于现阶段优化无法分析全部的基本块信息，所以在未来的优化中，可以通过热点块分析或离线优化等方式，获取更多基本块的信息，对片段内基本块进行代价分析，减少不同类型基本块间频繁切换动作的产生，实现性能的进一步提高。

第 7 章　总结与展望

本文就二进制翻译中性能这一关键问题，以龙芯二进制翻译 LATX 为基础，设计并实现了指令流分析制导的动态二进制翻译优化方案。该方案将优化过程拆分成分析探针识别以及优化器优化两个阶段。分析探针跟随反汇编阶段，进行指令流分析，获取并判断优化类型和方式，优化器则在翻译阶段进行相应的优化操作，以提高翻译后程序性能。

同时为了获取 LATX 翻译器性能瓶颈，加入了性能分析框架并进行性能分析。本文对翻译器的指令膨胀统计、二进制翻译扩展指令性能测试以及 SSE 向量指令翻译性能进行了分析，并针对性的提出了 EFLAGS 延迟计算、“反馈式”指令语义化翻译以及 SSE 标量指令高位运算消除这三种优化方案，其各优化的 SPEC CPU2000 定点和浮点性能如图 7.1 所示。本文优化主要工作如下：

1. **EFLAGS 延迟计算**。该优化的核心是对于每一需要生成 EFLAGS 运算的指令，进行 EFLAGS 的数据流分析，对于前序 EFLAGS 的结果被后序计算覆盖的情况，进行相应的消除。SPEC CPU2000 性能显示该优化可以带来 6% 以上的绝对性能提升。

2. **“反馈式”指令语义化翻译**。该优化解决了基本块内和基本块尾 EFLAGS 无法使用延迟计算消除的情况。对于基本块内，采用了语义化翻译方式，将多指令合并翻译，以减少多指令运算过程中的 EFLAGS 生成；而对于基本块尾，则采用了“反馈式”翻译方式，通过跨基本块的反馈信息，动态消除基本块尾的 EFLAGS 计算指令。在此基础上，本优化还针对自修改情况进行了处理和恢复，保证了优化的正确性。最后我们还对优化的性能提升进行了讨论，并分析了 EFLAGS 消除与性能提升之间的关系。该优化相比窥孔优化有更大的优化空间，其 SPEC CPU2000 程序约有 3% 的绝对性能提升。

3. **SSE 标量指令高位运算消除**。该优化核心思想是将基本块进行分类，根据每种类型基本块进行相应的高位消除优化。首先会根据基本块内指令类型以及数据流分析结果，得出相应的基本块类型；其次按照基本块的类型，进行相应的优化；最后在基本块链接过程中，需要对不同类型基本块间链接进行处理，保证运算的正确性。SPEC CPU2000 性能显示，其浮点性能有 9% 的绝对性能提升。

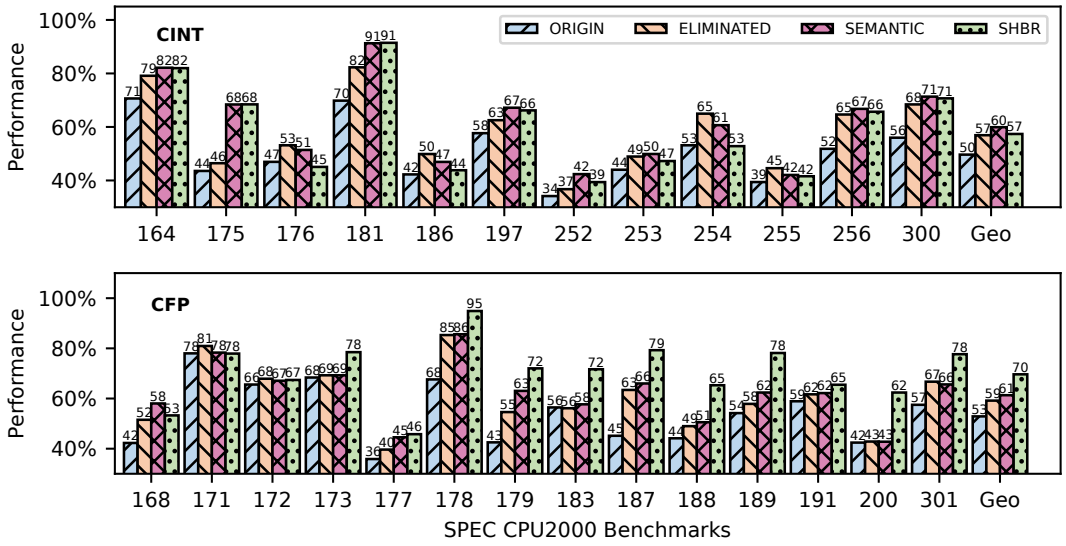

图 7.1 IFADO 优化各阶段 SPEC CPU2000 性能结果[1]

Figure 7.1 SPEC CPU2000 Performance Results for each stage of IFADO[1]

当然，本工作其对于 ARM 等其他含有 EFLAGS 类型的指令集的翻译优化也具有一定的参考价值。另外，本工作尽管已经对二进制翻译进行了系统优化，但仍有诸多不足，因此未来可以从这几个方面进行进一步完善：

1. **“反馈式”优化的探索深度不够**。

目前，“反馈式”优化为了保证翻译器的翻译效率，只进行了基本块之间的反馈，即相邻基本块之间才能进行反馈优化。这意味着对于距离较远的基本块，其反馈信息并未能及时反馈至现有的基本块，导致部分优化的遗失。因此，当后续基本块仅包含少量指令，尤其是只有一条无条件跳转指令时，通过再向后多分析一个基本块，可以获得更多有利于优化的信息。

2. **SSE 标量指令高位运算消除对定点程序不友好**。

从前序分析中，我们可以看到 SHBR 优化对于定点程序不是很友好，主要原因是它增加了不同类型基本块之间的切换开销。为了解决这个问题，我们需要最小化不必要的不同类型基本块切换操作。未来的优化可以从两个方向进行：

(a) 在热点路径上进行 SHBR 优化，将热点路径内的“保存-恢复”完全消除。这样可以通过热点路径获取主要的性能收益，对于其他部分不进行优化，降低了不同类型基本块动态切换的次数。

[1]原始叠加数据，即后续优化均在前序优化基础上完成，在 5.6 节和 6.5.1 节有相应说明。

(b) 加入启发式算法，在“保存-恢复”操作密集的部分实行 SHBR 优化。该做法可以缓解基本块类型上的“乒乓效应”，即在一个区域内频繁发生基本块类型的切换，减少由此带来的负担。

3. **其他优化**。

除了上述几种优化方式，还有更多的可以提升二进制翻译后程序性能的方法，如间接跳转优化和更优代码生成等。尽管目前因为翻译开销的问题，仍然无法加入更加重量级的优化，但未来可以通过引入 AOT[33] 机制和离线优化等技术实现翻译后优化，在编译器[34] 和重型优化器的帮助下，对产生的翻译后文件进行深度优化，以达到更好的性能效果。

参考文献

[1] Hennessy J L, Patterson D A. A new golden age for computer architecture [J/OL]. Commun. ACM, 2019, 62(2): 48–60. https://doi.org/10.1145/3282307.

[2] 姜伟超. 胡伟武为中国造" 芯" [J]. 科技创新与品牌, 2022, No.175(1): 28.

[3] 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 (2-16).

[4] 胡伟武, 高翔, 张戈. 龙芯指令系统架构及其软件生态建设 [J]. 信息通信技术与政策, 2022, No.334(4): 43-48.

[5] Altman E, Kaeli D, Sheffer Y. Welcome to the opportunities of binary translation [J/OL]. Computer, 2000, 33(3): 40-45. DOI: 10.1109/2.825694.

[6] Bellard F. QEMU, a fast and portable dynamic translator [C]//ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference. USA: USENIX Association, 2005: 41.

[7] Di Federico A, Agosta G. A jump-target identification method for multi-architecture static binary translation [C/OL]//CASES '16: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. New York, NY, USA: Association for Computing Machinery, 2016. https://doi.org/10.1145/2968455.2968514.

[8] Shen B Y, Hsu W C, Yang W. A retargetable static binary translator for the ARM architecture [J/OL]. ACM Trans. Archit. Code Optim., 2014, 11(2). https://doi.org/10.1145/2629335.

[9] Chen J Y, Yang W, Hsu W C, et al. On static binary translation of ARM/Thumb mixed isa binaries [J/OL]. ACM Trans. Embed. Comput. Syst., 2017, 16(3). https://doi.org/10.1145/2996458.

[10] Shen B Y, You J Y, Yang W, et al. An llvm-based hybrid binary translation system [J]. 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), 2012: 229-236.

[11] Liu I C, Wu I W, Shann J J J. Instruction emulation and os supports of a hybrid binary translator for x86 instruction set architecture [J]. 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015: 1070-1077.

[12] Wang J, Pang J, Liu X, et al. Dynamic translation optimization method based on static pre-translation [J]. IEEE Access, 2019, 7: 21491-21501.

[13] Hookway R J, Herdeg M A. DIGITAL FX!32: Combining emulation and binary translation [J]. Digital Tech. J., 1997, 9(1): 3–12.

[14] Rocha R C O, Sprokholt D, Fink M, et al. Lasagne: A static binary translator for weak memory model architectures [C/OL]//PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2022: 888–902. https://doi.org/10.1145/3519939.3523719.

[15] Borin E, Wu Y. Characterization of dbt overhead [C/OL]//IISWC '09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC). USA: IEEE Computer Society, 2009: 178–187. https://doi.org/10.1109/IISWC.2009.5306785.

[16] Baraz L, Devor T, Etzion O, et al. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on itanium®-based systems [C]//MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. USA: IEEE Computer Society, 2003: 191.

[17] d'Antras A, Gorgovan C, Garside J, et al. Optimizing indirect branches in dynamic binary translators [J/OL]. ACM Trans. Archit. Code Optim., 2016, 13(1). https://doi.org/10.1145/2866573.

[18] Bala V, Duesterwald E, Banerjia S. Dynamo: A transparent dynamic optimization system [C/OL]//PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2000: 1–12. https://doi.org/10.1145/349299.349303.

[19] Ebcioğlu K, Altman E R. DAISY: Dynamic compilation for 100% architectural compatibility [C/OL]//ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture. New York, NY, USA: Association for Computing Machinery, 1997: 26–37. https://doi.org/10.1145/264107.264126.

[20] Altman E R, Gschwind M K, Sathaye S W, et al. BOA: The architecture of a binary translation processor [C]//1999.

[21] Cifuentes C, Van Emmerik M. UQBT: adaptable binary translation at low cost [J/OL]. Computer, 2000, 33(3): 60-66. DOI: 10.1109/2.825697.

[22] Bruening D, Zhao Q, Kleckner R. DynamoRIO: dynamic instrumentation tool platform [J]. URL http://www.dynamorio.org, 2020.

[23] Chernoff A, Herdeg M, Hookway R, et al. FX!32 a profile-directed binary translator [J/OL]. IEEE Micro, 1998, 18(2): 56-64. DOI: 10.1109/40.671403.

[24] Drongowski P J, Hunter D P, Fayyazi M, et al. Studying the performance of the FX!32 binary translation system [C]//2007.

[25] Klaiber A C. The technology behind crusoe tm processors low-power x86-compatible processors implemented with code morphing [C]//2000.

[26] Dehnert J C, Grant B K, Banning J P, et al. The transmeta code morphing™ software: Using speculation, recovery, and adaptive retranslation to address real-life challenges [C]//CGO '03: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. USA: IEEE Computer Society, 2003: 15–24.

[27] Kim H S, Smith J. Hardware support for control transfers in code caches [C/OL]//Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36. 2003: 253-264. DOI: 10.1109/MICRO.2003.1253200.

[28] Hong D Y, Hsu C C, Yew P C, et al. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores [C/OL]//CGO '12: Proceedings of the Tenth International Symposium on Code Generation and Optimization. New York, NY, USA: Association for Computing Machinery, 2012: 104–113. https://doi.org/10.1145/2259016.2259030.

[29] Fu S Y, Hong D Y, Wu J J, et al. SIMD code translation in an enhanced HQEMU [C/OL]// ICPADS '15: Proceedings of the 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS). USA: IEEE Computer Society, 2015: 507–514. https://doi.org/10.1109/ICPADS.2015.70.

[30] 邹旭. 面向二进制翻译的随机化测试生成研究 [D]. 中国科学院大学, 2021.

[31] Abel A, Reineke J. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems [J]. 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019: 34-46.

[32] 马湘宁, 武成岗, 唐锋, 等. 二进制翻译中的标志位优化技术 [J]. 计算机研究与发展, 2005 (329-337).

[33] Riedel S, Schuiki F, Scheffler P, et al. Banshee: A fast llvm-based RISC-V binary translator [C/OL]//2021: 1-9. DOI: 10.1109/ICCAD51958.2021.9643546.

[34] Engelke A, Okwieka D, Schulz M. Efficient llvm-based dynamic binary translation [C/OL]// VEE 2021: Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2021: 165–171. https://doi.org/10.1145/3453933.3454022.

致 谢

回首在龙芯的这几年，我的内心充满感慨。在这里见证了 3A5000 更换指令集的过程，有机会参与了 LoongArch 生态的建设和发展，亲眼目睹了不久前 3A6000 的初片流回调试的现场。这些事情不仅见证了我的成长，也见证了龙芯的持续进步。

首先，我要感谢我的导师张福新老师。在我的求学过程中，您始终如一地关心和支持我，为我提供了宝贵的指导和帮助。您严谨的治学态度，深厚的专业素养和无私的精神激励着我在学术道路上不断追求卓越。

同时，我要感谢我的同学和朋友们。在这段时光里，我们共同学习、共同进步，相互激励，共同成长。你们的陪伴，让我的求学生涯更加充实和有意义。

另外，我要感谢龙芯团队。在这里，我看到了一个充满活力和创新精神的团队，他们为实现中国芯片事业的崛起而努力拼搏。在参与这一过程中，我收获了宝贵的经验和成长，深感荣幸。

最后，我还要感谢评阅本文以及参加答辩的各位专家和教授。也愿“规格严格，功夫到家”和“博学笃志，格物明德”能成为我不断进步的座右铭，激励我做出更多的贡献。

作者简历及攻读学位期间发表的学术论文与研究成果

作者简历:

胡起，安徽省黄山人，中国科学院计算技术研究所硕士研究生。

2016.9–2020.6 哈尔滨工业大学（威海）计算机科学与技术学院获学士学位

2020.9–2023.6 中国科学院计算技术研究所计算机系统结构专业硕士研究生

申请或已获得的专利:

1. CN115827064A 一种指令控制方法、装置及电子设备

参加的研究项目及获奖情况:

- 龙芯动态二进制翻译器 LATX
- 龙芯静态二进制翻译器 LASTM
- 2022 年研究生国家奖学金
- 2021 年曙光硕士生奖
- 2021 年中国科学院大学三好学生