中国科学院大学

University of Chinese Academy of Sciences

硕士学位论文

面向 RISC-V 的软硬协同二进制翻译器设计

作者姓名：晏悦

指导教师：王剑　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：处理器芯片全国重点实验室

中国科学院计算技术研究所

2024 年 6 月

Design of a Hardware-Software Co-Design Binary Translator for RISC-V

A thesis submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Engineering

in Computer System Architecture

By

YAN Yue

Supervisor: Professor WANG Jian

State Key Lab of Processors

Institute of Computing Technology, CAS

June, 2024

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。承诺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享有著作权的研究成果，未在以往任何学位申请中全部或部分提交。对本论文所涉及的研究工作做出贡献的其他个人或集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

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大学有关收集、保存和使用学位论文的规定，即中国科学院大学有权按照学术研究公开原则和保护知识产权的原则，保留并向国家指定或中国科学院指定机构送交学位论文的电子版和印刷版文件，且电子版与印刷版内容应完全相同，允许该论文被检索、查阅和借阅，公布本学位论文的全部或部分内容，可以采用扫描、影印、缩印等复制手段以及其他法律许可的方式保存、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

近年来，多款国产处理器设计能力迅速接近国际先进水平，但采用不同指令集。这带来了软件适配和生态迁移的困难，造成了生态碎片化的问题，阻碍了国产处理器的发展。二进制翻译技术，能缓解上述问题，但目前高性能二进制翻译器性能只能达到原生性能的 80%，且软件优化方案已经比较成熟，难以进一步提升性能，亟需软硬件协同的优化。

针对生态碎片化和二进制翻译器性能瓶颈的问题，本文在一个软硬协同的二进制翻译系统——x86 微译器中，添加了 RISC-V 架构的支持。该技术的核心目标是在单一硬件平台下实现多指令集的共存，同时实现接近原生的运行效率。

本文的主要工作及贡献如下：

1. 设计并实现了 RISC-V 微译器：通过增加专门的微码指令支持 RISC-V 指令翻译，获得了较高的 RISC-V 程序翻译执行性能。

2. 优化了 RISC-V 微译器的性能：添加了 4 种优化方案，包括行尾放松、分支放松、可变长行、指令压缩，测试显示优化方案降低了翻译缓存的缺失率，提升了微译器的性能。

本文在 Gem5 模拟器中实现了 RISC-V 微译器原型系统，实验显示优化后的 RISC-V 微译器在运行 SPEC 2000 测试的平均性能达到了原生程序的 96.1%，有效缓解了性能瓶颈，实现了接近原生程序的运行效率。微译器同时支持 x86 和 RISC-V 两种指令集，为多架构二进制翻译提供了一种新的解决方案。

关键词：二进制翻译，多架构，软硬协同，RISC-V

Abstract

In recent years, the capabilities of various domestically-produced processors have rapidly approached international advanced levels, but they often utilize different instruction sets. This diversity has created difficulties in software adaptation and ecosystem migration, leading to ecosystem fragmentation which impedes the development of these domestic processors. Binary translation technology offers a means to mitigate these challenges, although the performance of high-performance binary translators currently only achieves up to 80% of native performance, and with software optimization strategies already mature, further performance enhancements are challenging without hardware-software co-optimization.

This thesis addresses ecosystem fragmentation and the performance bottleneck of binary translators by integrating support for the RISC-V architecture into a co-designed binary translator technology—x86 MicroTranslator. The core goal of this technology is to enable the coexistence of multiple instruction sets on a single hardware platform while achieving near-native execution efficiency.

The main work and contributions of this thesis are as follows:

1. Design of the RISC-V MicroTranslator: By adding specialized microcode instructions to support RISC-V instruction translation, high performance in the execution of translated RISC-V programs was achieved.

2. Four optimization strategies were implemented, including the relaxation of cache line endings, conditional jump instructions, support for compressed instructions, and support for variable-length microcode lines. Tests showed that these optimizations reduced the miss rate of the microcode cache and enhanced the performance of MicroTranslator.

A prototype system was implemented in the Gem5 simulator. Experimental results demonstrate that the optimized RISC-V MicroTranslator achieves an average performance of 96.1% of native programs when running the SPEC 2000 benchmarks, effectively alleviating the performance bottleneck and achieving near-native execution efficiency. By supporting both x86 and RISC-V instruction sets, MicroTranslator provides a novel solution for multi-architecture binary translation.

Key Words: Binary translation, multi-architecture, software and hardware collaboration, RISC-V

目 录

图目录

表目录

第 1 章　引言

随着计算机技术的迅速发展，对硬件算力的要求不断提升，对硬件多元化发展的需求也日益增加，推动着各类新型处理器架构的涌现。David Patterson 也宣称我们正处于计算机体系结构的一个 “黄金时代”[1]，例如云计算、虚拟化、边缘计算等新兴领域对处理器的性能、能效、安全等方面提出了更高的要求，这些要求往往需要不同的指令集架构来支持。此外，出于国家信息安全和自主可控的考虑，国产处理器的研发和应用也得到了前所未有的重视，如龙芯、飞腾、兆芯等公司推出了一系列国产处理器产品，采用了不同的指令集架构。

然而，由于历史原因和商业利益，不同的处理器架构之间生态互不兼容，导致了生态碎片化问题，增加了软件开发的成本和复杂度。传统的 x86 和 ARM 指令集构筑出了强大的 “生态壁垒”，大量传统软件只有 x86 或者 ARM 版本，无法直接在其他架构的处理器上运行；而新兴指令集由于出现时间较晚，软件较少，生态不完善，难以吸引开发者和用户，进而存在市场竞争力不足的问题，即便微架构设计和性能优化再出色，也难以在市场上获得成功。因此，当处理器设计能力和性能达到一定水平时，如何打破指令集生态壁垒，丰富新型指令集架构的生态，是当前国产处理器发展的重要课题。

目前对于新型指令集生态建设，主要有两种办法：生态迁移和二进制翻译。生态迁移是指将原有的软件生态迁移到新的指令集架构上，这需要大量的人力和物力资源，主要适用于关键应用和系统级软件，需要有源代码并重新编译并移植。二进制翻译是指通过软硬件技术将原有的二进制程序，不经修改的、直接翻译到新的指令集架构上运行，这种方法可以减少迁移的成本，但是相对于原生运行，性能会有一定的损失，主要应用于指令生态初期建设，适用于对性能要求不高的程序或者古老的没有源代码的程序。

本文主要聚焦于二进制翻译技术，通过软硬协同的方式，提升二进制翻译器的性能，为新型指令集架构的生态建设提供技术支持。

本章节首先介绍分析国产处理器生态碎片化问题，接着介绍二进制翻译技术的概念和分类，然后阐述本文的主要工作及贡献，最后介绍本文各章节的组织结构。

1.1　国产处理器生态碎片化问题

中国国产处理器在多个架构上展现出丰富多彩的发展，其中 x86、ARM、LoongArch[2] 和 RISC-V 等架构代表了不同的技术路线，由各个公司推动。参见表1-1，以下是各架构的特点：

- x86 架构：代表公司为兆芯和海光，采用 x86 架构 IP 内核授权模式，可基于公版 CPU 核进行优化或修改，性能起点高，生态壁垒相对低。但是依赖海外

表 1-1 国产处理器的发展现状

Table 1-1 Development status of domestic CPUs

指令集	代表公司	优势	不足
x86	兆芯，海光	兼容 Windows	授权问题
ARM	华为，飞腾	兼容安卓	授权问题
LoongArch	龙芯	自主可控	生态不足
RISC-V	开芯院，阿里	开源开放	生态不足

企业授权，自主可控风险偏高。

- ARM 架构：代表公司为华为和飞腾，采用 Armv8 永久授权，具有更高的自主化程度，可自行研发设计 CPU 内核和芯片，也可扩充指令集。但是存在长期隐患，这些国产 CPU 厂商一直难以获得 Armv9 的永久授权。
- LoongArch 架构：代表公司为龙芯，采用自研的 LoongArch 指令集，具有相对更高的自主可控程度，已经在党政军工等行业得到了广泛应用。但是自研指令集生态不足，需要大量投入才能建立起完善的软件生态。
- RISC-V 架构：代表公司为开源芯片研究院和阿里平头哥，采用国际开源的 RISC-V 架构，具有相对精简的指令集架构，并遵循开源宽松的 BSD 协议，在国内得到了迅速发展，但同样生态不足，需要长时间生态建设。

其中，x86 属于复杂指令集架构（Complex Instruction Set Computer, CISC），ARM、LoongArch 和 RISC-V 属于精简指令集架构（Reduced Instruction Set Computer, RISC）。

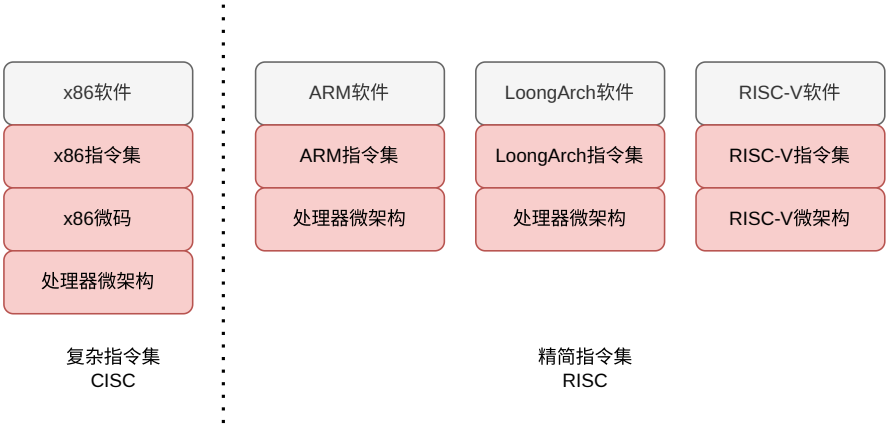

图 1-1 不同指令集 CPU 的架构图

Figure 1-1 Architecture of CPUs with different instruction sets

注: 图中白色为软件层，红色为硬件层，并忽略了操作系统，这不是本文研究重点。对于复杂指令集 x86，内部实现了微码层，而精简指令集架构内部一般没有微码层。

如图1-1所示，同一套软件源代码需要针对不同的指令集进行编译才能在不同架构的 CPU 上运行[1]。而如果软件不能编译，只有二进制程序，就不能在其他

架构处理器上运行，这就是生态碎片化问题。而大量用户态软件处于商业利益、防破解等原因，不会开源，只有二进制程序，这加剧了生态碎片化问题。例如大量 x86 版本的工业软件、游戏、办公软件等，只有二进制程序，无法直接在其他架构的 CPU 上运行。

随着中国国产处理器的指令集多样化发展，出现了生态碎片化问题，这增加了应用程序在不同架构间迁移和适配的复杂性。存在的问题包括：

- **适配和迁移负担：**不同架构间的适配和迁移需要大量人力和物力资源。
- **历史兼容包袱：**不同指令集的历史兼容包袱使得跨架构的兼容性复杂。
- **编译与源代码的限制：**古老软件无源代码，只能通过翻译运行。
- **操作系统支持的挑战：**操作系统厂商需要投入更多资源以支持不同架构。

1.2　二进制翻译器概述

目前，二进制翻译技术是解决生态碎片化问题、兼容性问题的主要方法。

二进制翻译技术能够将源指令集（称为客户指令集，guest ISA）上的二进制程序翻译到目标指令集（称为宿主指令集，host ISA）上执行，借此能把已有的软件从一个平台迁移到另一个平台，实现不同指令集架构之间的兼容性，丰富新型指令集架构的生态。在历史上，Alpha[3]、Transmeta[4]、Apple[5]、Intel[6] 等公司都曾经使用过二进制翻译技术来协助推出新的指令集架构，目前市场上也有一些商业化的二进制翻译器产品。

参考一篇二进制翻译综述的分类方法[7]，二进制翻译器可以分为静态和动态、用户态和系统态、解释型和翻译型、软件实现和软硬件实现等，这些分类方式可以根据不同的需求和应用场景进行组合，并且基本是**正交**的。

下面是对这些分类的简要说明：

1.2.1　静态与动态

静态二进制翻译：在程序执行前，将整个源程序从一种机器指令集转换到另一种。这种转换通常产生可直接运行的预翻译文件（Ahead-of-Time file, AOT 文件），无需在运行时进行额外的转换。静态翻译的优点在于它可以在转换过程中进行深入的代码分析和优化，但它不适用于即时生成代码（Just-In-Time Code, JIT 代码），也难以处理代码数据混淆等复杂的情况，导致其使用场景受限。

动态二进制翻译：在程序运行时，以基本块或者函数体为单位，按需将代码从源指令集转换为目标指令集，一边翻译一边执行，这种方法能处理更广泛的情况。动态翻译还引入了代码缓存（Code Cache），用于存放已经翻译过的代码，以提高性能。动态翻译过程中能获取更多程序运行时的信息，可以进行更多的优化。但同时动态翻译也面临着翻译开销大、代码优化不充分等问题。

[1]这里主要关注 C/C++ 等底层语言，对于 Java 等支持跨平台运行的语言，也需要 Java 虚拟机对不同指令集平台进行编译适配。

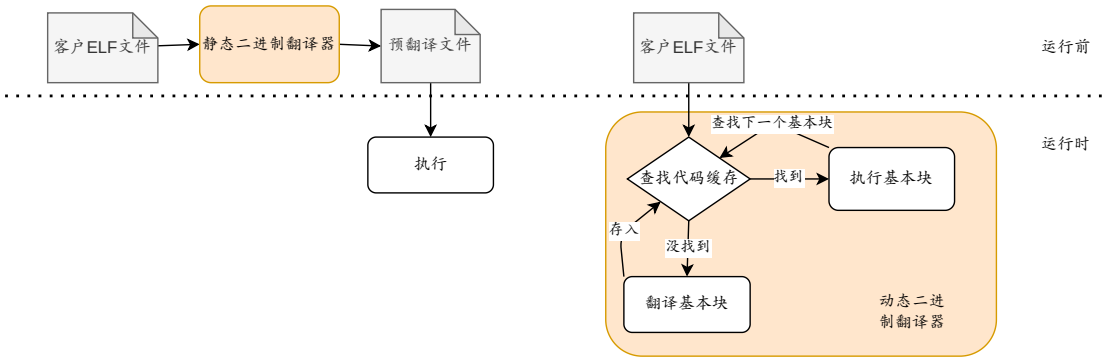

图 1-2　静态与动态二进制翻译器

Figure 1-2　Static and Dynamic Binary Translators

注: 左图为静态二进制翻译器，在运行前提前翻译成预翻译文件，运行时直接执行。右图为动态二进制翻译器，在运行时翻译基本块并存入代码缓存，翻译过的基本块就直接执行。

两者的概念图可以查看1-2。目前也有动静结合的方法，即通过静态翻译尽可能翻译更多的基本块并存储在预翻译文件中，在运行时如果遇到未翻译的基本块再进行动态翻译，这种方法可以减少运行时的翻译开销，提高性能。

1.2.2　用户态与系统态

用户态二进制翻译：仅转换用户空间程序的指令，不涉及操作系统核心态的代码，运行环境可以直接依赖于宿主操作系统。这种翻译对于实现应用程序的跨架构兼容性特别有用，相对于系统态翻译，性能更高。

系统态二进制翻译：模拟整个系统，包括 CPU、内存、设备等，能够转换操作系统内核代码和驱动程序，以实现整个操作系统的兼容。这要求翻译器能够处理更为复杂的系统级别的操作，如中断处理和系统调用，通常性能较低。

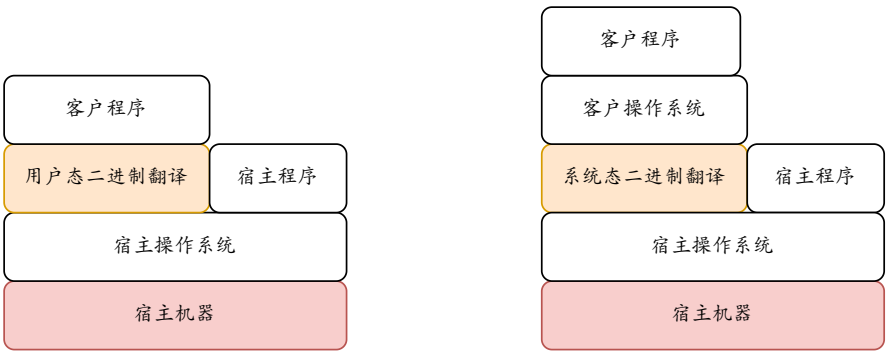

图 1-3　用户态与系统态二进制翻译器

Figure 1-3　User and System Binary Translators

两者的概念图可以查看1-3。

1.2.3　解释型与翻译型

解释型二进制翻译：以单条指令为单位，逐条解释源指令并实时执行。这种方式可以实现较高的兼容性，因为它可以逐步处理源程序的每个指令，但由于不保存已解释的指令，其性能通常较低，比本地原生执行慢 10 到 100 倍。一般用于调试、模拟和验证等场景。

翻译型二进制翻译：以基本块为单位，将源程序的指令翻译成目标机器的指令，存储翻译后的基本块，并进行优化，最后执行翻译后的代码。这种方法在翻译阶段可能会花费更多时间，但运行翻译后的代码的速度更快。

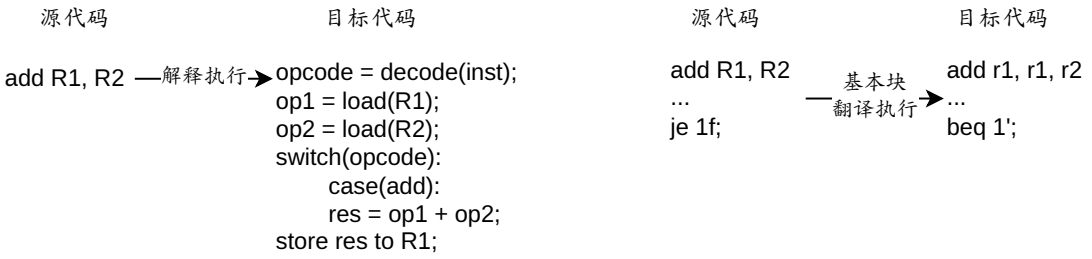

图 1-4　解释型与翻译型二进制翻译器

Figure 1-4　Interpretive and Translational Binary Translators

注: 左图为解释型二进制翻译器，以单条指令为单位，解释指令语义执行。右图为翻译型二进制翻译器，以基本块为单位，翻译整个基本块并进行优化后执行。

两者的概念图可以看1-4。

1.2.4　软件实现与软硬件实现

软件实现：在不修改宿主硬件的情况下，完全在软件层面上实现二进制翻译。这种实现方式的优点在于其灵活性和可移植性，但可能会因为缺乏硬件支持而影响翻译性能。

软硬件实现：结合使用专门的硬件支持和配套的软件技术来提高二进制翻译的效率。例如，通过在处理器中实现特定的硬件间接跳转表（CAM 表）来加速间接跳转的翻译执行[8]，添加特定的指令集扩展来缩小语义差异[9]。这种方法可以显著提高性能，但增加了硬件设计的复杂性和成本。

1.3　二进制翻译器性能问题

表2-1对这些分类进行了总结，各种二进制翻译技术的选择取决于特定应用的需求、目标架构的特性以及性能和兼容性之间的权衡。

目前主流的二进制翻译器主要采用**动态、用户态、翻译型和软件实现**的方式，如 QEMU[10]、Rosetta2[5,11]、ExaGear[12]、LATX[2,13] 等（Rosetta2 也有动静

表 1-2　二进制翻译技术的分类及优劣势比较

Table 1-2　Classification and Pros and Cons of Binary Translation Technologies

分类	说明	优势	不足
静态	提前翻译	深入的代码分析和优化	使用场景受限
动态	运行时翻译和优化	能处理动态生成代码	额外的运行时开销
用户态	只翻译用户态指令	实现简单，性能较高	支持应用有限
系统态	模拟整个系统	能直接跨架构运行操作系统	性能较低
解释型	逐条指令解释执行	方便调试模拟等	性能很低
翻译型	以基本块为单位翻译执行	性能较高	设计更加复杂
软件	纯软件实现	灵活性和可移植性	性能不高
软硬件	特定硬件支持	特定硬件加速提升性能	增加硬件设计复杂性

态结合优化，LATX 有硬件指令集扩展支持），这是由于动态翻译可以在运行时进行优化并能支持 JIT 程序的翻译运行，用户态翻译、翻译型翻译可以实现更高的性能，软件实现可以实现更好的可移植性，对硬件修改较少。

翻译性能是衡量二进制翻译器优劣的重要指标，本文将翻译性能定义如下：

$$\text{翻译性能} = \frac{\text{本地原生代码执行时间}}{\text{基于二进制翻译执行时间}} * 100\% \tag{1-1}$$

对同一份测试程序的源代码用相同编译参数，直接编译到宿主指令集得到原生二进制代码 B_{host}，交叉编译到客户指令集得到客户二进制代码 B_{guest}，在硬件平台直接运行原生程序 B_{host} 的时间记为**本地原生代码执行时间**。使用二进制翻译系统运行客户程序 B_{guest} 的时间记为**基于二进制翻译执行时间**。通常后者会大于前者，两者的比值即为翻译性能，性能越高，越接近 100%，说明翻译器越优秀。

但是由于目前主流的二进制翻译器性能相对较低，例如 QEMU[10] 虽然支持多架构应用，但在翻译运行 SPEC CPU 2017[14] 程序时候，仅有约 10% 的性能。商业二进制翻译器也存在性能损失，如苹果的 Rosetta2[5,11]、华为的 ExaGear[12]、和龙芯的 LATX[2,13]，性能仅达到原生运行的 70% 左右，这直接影响了软件生态迁移的流畅度和成功性，相对于同架构不同操作系统模拟的虚拟机（性能接近 100%），性能不足也限制了二进制翻译器的广泛应用。后文2.2节会详细分析现有二进制翻译器的开销来源。

除了性能问题，二进制翻译也较难实现多架构支持。在虚拟化技术广泛应用的今天，二进制翻译器的多架构支持也是一个重要的研究方向。目前，云计算和移动计算催生出多样化异构系统，计算平台从传统的单一指令系统向多指令系统融合发展，例如 Google 使用多种指令集组成的异构系统，传统的虚拟化只能实现单指令集中多种操作系统的迁移，应用软件难以在不同指令集架构的硬件上迁移运行，多架构二进制翻译器可以解决跨指令集应用迁移问题。目前主流的开源二进制翻译器 QEMU 支持多架构，但性能太低，难以商用化。探索出高性能的多架构二进制翻译器也是当前的研究热点。

除了 CPU 架构指令集差异导致的生态碎片化，GPU 目前也面临相似的问题。英伟达公司的 CUDA 软件生态在 GPU 领域占据绝对优势，但是只支持英伟达的 GPU。AMD、Intel 和很多国内显卡公司开发的加速器无法直接运行 CUDA 程序，这也导致了 GPU 生态碎片化问题。随着更多有竞争力的 GPU 架构的出现，也有更多用户倾向于在不同 GPU 架构上运行 CUDA 程序，目前有两种解决方案：重新编译 CUDA 程序，或者通过二进制兼容技术。重新编译现有的 CUDA 程序完全合法但需要人力投入，例如 AMD 和 Intel 都有工具分别将 CUDA 程序移植到他们的 ROCm 和 oneAPI 平台。而后者的代表是 ZLUDA[15]，通过兼容层允许在非英伟达 GPU 上运行未经修改的 CUDA 程序（兼容层和二进制翻译不同，更接近于 Wine 这样的库兼容技术），并声称提供了“接近原生”的性能。但最新版本的英伟达用户协议（ELUA）中明确禁止使用 ZLUDA 等技术在其他平台上运行 CUDA 代码，但这并不代表兼容技术不可行，也有更多技术被提出用以打破“CUDA 霸权”。但本文的关注点在于 CPU 架构的二进制翻译技术，对于 GPU 架构的二进制兼容技术暂不做更多讨论。

1.4　本文的主要工作及贡献

为了解决 CPU **生态碎片化**问题和**二进制翻译器性能**问题，需要一种多架构软硬协同的二进制翻译技术。这项技术的关键目标是在同一套硬件下实现多指令集的共存，同时实现接近原生的运行效率。

如图1-5所示，本课题组提出了一种多架构软硬协同的二进制翻译技术——**微译器**，按照上述分类属于动静结合、用户态、翻译型、软硬结合，通过在硬件层面支持一套融合微码，在软件层面实现多架构的二进制翻译器，进而能实现多种指令集软件的共存和运行，同时性能接近原生运行性能。

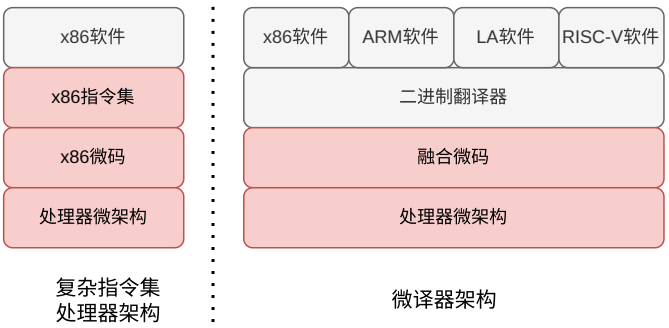

图 1-5　微译器架构简图

Figure 1-5　The architecture of Micro-Translator

注: 硬件仅对外暴露微码，软件二进制翻译器支持多架构，性能接近原生运行性能。

在本团队的工作基础上，**本文**在微译器已有的支持 x86 指令集翻译执行的框架下，添加了 RISC-V 架构的支持，本文的主要工作及贡献如下：

1. 设计并实现了 RISC-V 微译器：设计了寄存器映射方案，累计添加了 51 条微码指令用于支持 272 条 RISC-V 指令集的翻译执行。实现了微译器对于多架构的高效支持，为添加更多指令集提供了技术支持。

2. 优化了 RISC-V 微译器的性能：添加了 4 种优化方案，包括行尾放松、分支放松、可变长行、指令压缩。这些优化方案进一步提升了翻译缓存的利用率，降低了缓存缺失率，提升了微译器整体性能。

本文在 Gem5 模拟器中实现了 RISC-V 微译器原型系统，实验显示优化后的 RISC-V 微译器在运行 SPEC 2000 测试的平均性能达到了原生程序的 96.1%，有效缓解了性能瓶颈，实现了接近原生程序的运行效率。微译器同时支持 x86 和 RISC-V 两种指令集，能有效的解决 CPU 生态碎片化问题和二进制翻译器性能问题，为多架构二进制翻译提供了一种新的解决方案。

1.5 论文的组织结构

本文的组织结构如下：

在第二章介绍二进制翻译的相关工作，包括 4 种主流的软件二进制翻译器，发现其性能难以接近原生性能。接下来对软件二进制翻译器的性能开销进行分析，然后介绍一些已有的软硬协同二进制翻译器是如何缓解性能瓶颈的，为了更进一步降低性能开销，引出了复杂指令集 x86 处理器中微码缓存的概念，最后介绍团队项目——微译器的架构设计和实现，并讲解微译器是如何消除间接跳转的性能开销，进而提升翻译性能。

第三章介绍本文在微译器中添加 RISC-V 架构的设计与实现，首先讲解了软件层二进制翻译器如何重构用于支持多架构翻译，接下来详细介绍融合微码编码设计来更好的提取不同指令集的共性和差异，进而分析 RISC-V 和 x86 的指令集语义差异，如何添加合适的融合微码指令用以支持 RISC-V 指令翻译。最后分析 RISC-V 和 x86 ABI 差异，如何在 RISC-V 微译器中处理系统调用差异、寄存器映射、栈的初始化问题等。

第四章介绍 RISC-V 微译器的优化方案，RISC-V 微译器在引入翻译缓存后会产生额外的硬件性能开销。本章首先分析这部分性能开销的来源，然后提出了几种优化方案，包括行尾放松、分支放松、可变长行、指令压缩，这些优化方案降低了 RISC-V 微译器的性能开销，提升了整体性能。

第五章介绍 RISC-V 微译器的实验数据分析，包括搭建原型系统使用的 Gem5 实验环境，运行性能分析使用的测试程序，进行正确性验证使用的调试环境，然后对 RISC-V 微译器的性能进行测试和分析，并分别统计了各种优化方案的性能提升。

第六章总结全文并展望未来工作。

第 2 章　相关工作

本章首先介绍软件实现二进制翻译的相关工作，包括著名的几款二进制翻译器，如 QEMU、Rosetta2、ExaGear、LATX 等，并对软件二进制翻译器的性能开销进行分析，发现间接跳转的性能开销以及指令集语义差异是其主要瓶颈。接下来介绍软硬协同二进制翻译器的相关工作，包括超长指令字兼容和指令集扩展，能有效缓解性能开销，但仍有一定局限性。为了解决间接跳转和指令集语义差异的问题，引出了复杂指令集 x86 处理器的微码缓存的概念，为 x86 微译器的设计提供了设计思路。最后介绍了本团队的 x86 微译器项目。

2.1　软件二进制翻译器

软件二进制翻译器是一种通过软件实现的二进制翻译器，它对硬件改动较小，能够在不同的硬件平台上运行，因此具有更好的可移植性和灵活性。本节主要介绍一款开源的用户态二进制翻译器 QEMU，以及三款商用级的用户态二进制翻译器，包括苹果公司的 Rosetta2、华为公司的 ExaGear、龙芯公司的 LATX。其中 Rosetta2 和 LATX 也有硬件支持，但主要还是依赖软件实现，因此归类为软件二进制翻译器。

2.1.1　QEMU

QEMU 是一个广泛使用的开源多平台二进制翻译框架，特别在云计算环境中，它常和 KVM 一起使用，提供了一种高效的虚拟化解决方案。QEMU 同时支持系统态翻译和用户态翻译，本文主要关注用户态翻译部分，它允许在一个主机操作系统上模拟另一个操作系统的应用程序。

QEMU 也同时支持多种架构，包括 x86、ARM、RISC-V 等，使其成为开发跨平台应用程序的理想工具，也可用于软件开发、测试以及安全研究。如图2-1，它通过将不同指令集的机器代码翻译成中间语言（IR，Intermediate Representation），再将 IR 翻译成宿主指令集，实现跨架构的兼容性。这种设计使得 QEMU 能够处理多种指令集，为用户提供了一定的灵活性。然而，QEMU 的性能仅能达到原生性能的 10%，这主要归因于双层翻译（客户代码翻译到 IR，IR 翻译到宿主代码）的性能损失以及 Helper 函数模拟浮点和向量指令的性能开销[16]。

基于 QEMU 的优化主要旨在提高其性能和执行效率，例如中间代码优化[17]、基本块链接优化[18]、缓存管理[19]、Helper 函数优化[20] 等。以下给出两个优化示例，包括 HQEMU 的多线程并行翻译和执行，以及寄存器分配的优化。

HQEMU[18] 是一个基于 QEMU 和 LLVM 构建的多线程和可重定向动态二进制翻译器。这个项目的目标是通过利用多核处理器的并行处理能力来减轻 DBT 的开销，同时允许更复杂的优化技术的应用。HQEMU 通过在不同的线程上分别

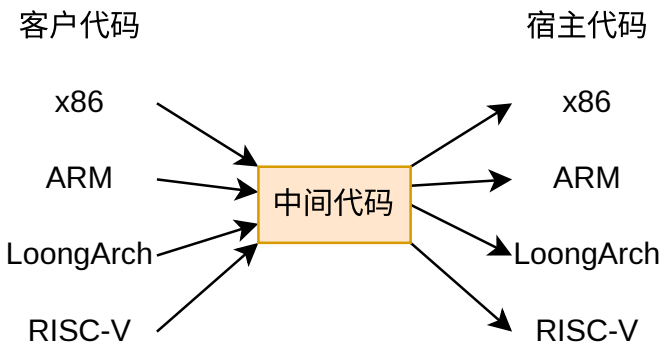

图 2-1　QEMU 二进制翻译器架构图

Figure 2-1　The architecture of QEMU binary translator

注: 通过中间代码，QEMU 能在多种宿主指令集机器上运行多种客户程序。

运行 QEMU 翻译器和 LLVM 优化器，从而实现了这一目标。在一系列基准测试中，比如 SPEC CPU2006，HQEMU 在 x86 到 x86-64 的模拟中，相比于原始的 QEMU 性能提高了 2.4 倍至 4 倍。

在动态二进制翻译中，寄存器分配是提高翻译代码执行效率的关键。原始的 QEMU 在许多宿主机上将所有目标寄存器映射到内存，并仅将少数临时变量存储在宿主寄存器中。这种方法并没有考虑相邻指令之间的依赖关系，导致即使在执行前一个指令时已经将一个客户寄存器的值加载到临时变量中，执行下一个指令时也需要再次从内存中重新加载该值。为了解决这个问题，[18] 提出了一个优化方法，其在两个或更多相邻指令执行中有效利用了临时变量，从而减少了对内存的不必要访问。通过这种优化，基准测试显示可以实现 10% 到 20% 的速度提升。

尽管以上优化示例展示了通过改进 QEMU 的翻译过程和内部机制，可以提升 QEMU 的效率和性能，但是改进后的性能仍然仅相当于原生性能的一小部分，这在一些对性能要求较高的应用场景下显得不够可用。

2.1.2　Rosetta2

Rosetta2 是苹果公司开发的商用二进制翻译系统，通过支持预先编译 (AOT) 和即时编译 (JIT) 两种模式实现动静结合翻译，预先翻译对性能影响大的代码，并把翻译后代码保存在磁盘上，而对于需要动态解析的代码，采用即时编译的方式进行翻译。Rosetta2 主要针对 64 位的 x86_64 指令集进行翻译，以适配苹果 M1 等 ARM64 处理器的架构。

Rosetta2 的核心目标是保证 MacOS 上的 x86 软件能够在 ARM 架构下运行，由于 x86 和 ARM 指令集都采用小端法（little-endian），这简化了指令翻译过程，避免了复杂的字节序反转操作。这一点与苹果过去从 PowerPC（大端法，big-endian）切换到 x86 的过程不同，后者在转换时面临更多的挑战。

然而，x86（强内存序）与 ARM（弱内存序）在内存一致性模型上的差异可能导致多线程软件运行结果出现差异，这是模拟 x86 的一个重要挑战[21]。苹果通过在芯片内部额外实现一个 Intel 版本的强内存 TSO 模型，并通过后门开关在运行 Rosetta2 时切换到该内存模型，解决了这一问题。硬件支持的内存模型有效的降低来并发程序翻译的性能开销，提高了 Rosetta2 的性能。

2.1.3　ExaGear

ExaGear 是华为公司开发的动态二进制翻译软件，专为在基于自研的 ARM 鲲鹏服务器上运行而设计。ARM 服务器由于其能效比优势，逐渐在数据中心中得到广泛应用，但是由于软件生态的限制，一些只有 x86 版本的软件无法在 ARM 服务器上运行。ExaGear 通过将 x86 应用程序翻译为 ARM 指令集，使得这些软件能够在 ARM 服务器上运行。

ExaGear 通过修改 Linux 的 binfmt_misc 组件，使得系统能够识别并使用 ExaGear 作为 x86 应用程序的解析器，实现了在安装过程中的高效集成。ExaGear 主要包括两个关键组件：指令翻译引擎和 x86 运行环境。指令翻译引擎充当 x86 应用程序与 ARM 架构服务器之间的中间件，实现了在 x86 应用程序启动时的实时翻译功能。x86 运行环境为 x86 应用程序提供了必要的标准库、实用程序和配置文件，构建了一个完整的运行时环境。此外 ExaGear 还通过 Trace 优化技术来减少分支数目，进而改善内存布局和局部性，减少间接跳转查找过程[22]。

2.1.4　LATX

龙芯二进制翻译器（LATX）是龙芯公司开发的一款动态二进制翻译软件，用于在龙芯处理器（LoongArch 架构）上运行 x86 架构的应用程序。结合 LATX 和 Wine[23]（一个操作系统 API 翻译软件，它可以用 Linux 的系统调用来模拟实现 Windows 的系统调用，从而实现在 Linux/ x86 上运行 Windows/x86 的应用程序），用户可以在龙芯处理器上运行大量的 x86 应用程序，包括 Windows 应用程序，例如微信、WPS、部分游戏等。

后文2.3.2节会讲解，通过添加龙芯二进制翻译扩展指令（LBT），LATX 能生成出更接近 x86 语义的宿主指令，而无需用多条指令模拟。此外，LATX 也添加了各类优化措施，例如 x86 EFLAGS 延迟计算、基本块链接、跨基本款反馈优化等，更多细节可以参考胡起的硕士论文[24]。

表2-1 对本节提到的 4 个主流的二进制翻译器进行了总结。

表 2-1 主流二进制翻译器

Table 2-1 The mainstream binary translators

二进制翻译器	公司	客户平台	宿主平台
QEMU	开源项目	x86、ARM、RISC-V 等	x86、ARM、RISC-V 等
ExaGear	华为	x86	ARM
Rosetta2	苹果	x86	ARM
LATX	龙芯	x86	LoongArch

2.2 软件二进制翻译的性能开销分析

上一节提到的三款商用级用户级二进制翻译器，包括苹果公司的 Rosetta2、华为公司的 ExaGear 和龙芯公司的 LATX，在运行 SPEC2017 基准测试时，根据1-1公式，得到的翻译性能分别为 67.2%、72.7% 和 60%，距离原生性能 100% 还有较大的差距，为此希望通过性能开销分析找出二进制翻译器的性能瓶颈。

2.2.1 理论分析

在进行实验分析之前，我们可以先对二进制翻译器的性能开销进行理论分析。如图2-2，二进制翻译的性能开销主要分为两类：指令集语义差异和二进制翻译器机制的开销。

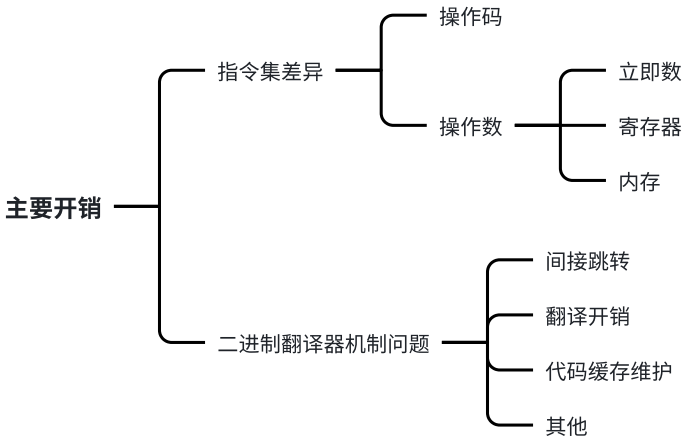

图 2-2 二进制翻译器的性能开销分析

Figure 2-2 The anaylsis of binary translator overhead

指令集是计算机软件和硬件交互的接口，是定义了计算机的操作和数据的格式的一种规范。指令包括操作码和操作数，操作码是指令的功能码，操作数是指令的操作对象。其中操作数按照操作对象可以分为立即数、寄存器、内存等。

- 操作码：操作码不同导致一条客户指令翻译成多条宿主指令，例如乘累加指令翻译为乘法和累加两条指令。

- 立即数：如果客户指令的立即数范围大于宿主指令的立即数范围，那么需要额外的指令来存放或加载立即数。
- 寄存器：如果客户指令的寄存器数量大于宿主指令的寄存器数量，那么需要额外的指令来保存和恢复寄存器，也称为寄存器溢出。
- 内存：如果客户指令的内存地址访问模式复杂，那么需要额外的指令来计算内存地址。

而二进制翻译器机制的开销主要包括：

- 间接跳转的性能开销：如图2-3，间接跳转是指在程序运行时，通过读取寄存器的值来决定跳转到哪个地址。这种跳转方式在静态翻译时无法确定，因此需要在运行时查询哈希表来确定跳转地址，通常需要 10 余拍。
- 翻译开销：翻译开销是指动态二进制翻译器在翻译客户指令时产生的性能开销。这种开销主要包括指令翻译、基本块优化、基本块链接等。
- 代码缓存维护：二进制翻译器需要维护一个软件的代码缓存，用于存储翻译后的宿主指令。当缓存满时，需要进行缓存替换产生开销。

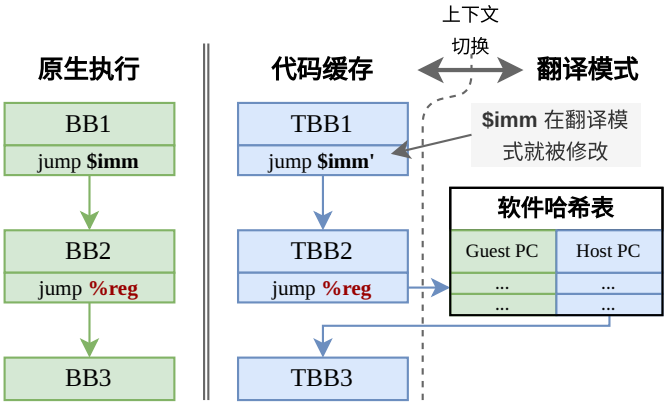

图 2-3　间接跳转开销

Figure 2-3　The overhead of indirect jump

注：由于翻译后的代码地址发生了非线性偏移，间接跳转的目标地址只能在运行时查询哈希表获取，通常需要 10 余拍。

2.2.2　实验分析

在理论分析的基础上，我们可以通过实验来验证二进制翻译器的性能开销。

根据本团队之前完成的一项工作[16]，基于如下两个前提，使用指令膨胀率来分析二进制翻译器的性能开销。

(1) 为了保证客户程序的精确异常，二进制翻译器不会做指令重排这样的激进优化，即保持指令的顺序不变，指令的边界不被打破。

(2) 主流二进制翻译器在运行 SPEC 2017 这类运算密集型的基准测试时，98% 的时间都在执行生成的宿主指令上，翻译开销和维护代码缓存的开销可以

忽略不计。

指令膨胀率是指，每条客户指令平均翻译出的宿主指令数，是一个大于 1 的小数，计算方式参考2-1。举例来说，一条 x86 的 add 指令可能翻译成 LoongArch 的 load 指令和一个 add 指令，那么这条 x86 add 指令的膨胀率就是 2。对所有的客户指令的膨胀率求加权平均值，权重为客户指令的动态执行次数，即得到了某个翻译器在运行某个程序的指令膨胀率（为了说明简单，忽略了基本块内指令的优化）。

$$\text{总体膨胀} = \frac{\text{生成的宿主指令数}}{\text{客户指令数}} = \frac{\sum_i \text{指令频次}_i \times \text{指令膨胀率}_i}{\sum_i \text{指令频次}_i} \tag{2-1}$$

假设所有指令执行时间相同，那么膨胀了多少倍，程序的执行时间就会变长多少倍，指令膨胀率约等于性能增加的值。事实上，不同指令的执行时间是不同的，但可以大致通过指令膨胀率来估计程序的性能开销。指令膨胀率越高，翻译后的程序要执行的指令数越多，执行时间越长，性能越低。即便多发射处理器能在单拍内执行多条指令，缓解更多指令带来的性能开销，但根据测试数据，指令膨胀率和性能下降值保持正相关，相关系数为 0.98[16]。因此，可以用指令膨胀率来衡量性能开销。

如图2-4，测量并模拟了 3 款商用级二进制翻译器（ExaGear、Rosetta2、LATX）在运行 SPEC2017 的指令膨胀率，模拟误差在 5% 以内。注意这三款翻译器都是复杂指令集（x86）到精简指令集（ARM/ LoongArch）的翻译器，这两类指令集语义差异较大，因此指令膨胀率较高。图中2-4蓝色的点表示测量的真实膨胀率，而分解成不同颜色的柱状图表示模拟的膨胀率来源。根据指令膨胀率的来源，主要将开销分成了 5 类：

(1) **操作码差异：**不同指令集的操作码差异引起 Eflags 计算等操作的额外指令翻译，增加了指令膨胀率。此外 LoongArch 对于子寄存器默认符号扩展，x86/ARM 默认零扩展。对应图2-4中**棕色**部分。

(2) **操作数访问模式不同：**复杂指令集（x86）可以直接访问内存，而其他精简指令集只能操作寄存器，导致操作数模式不同。对应图2-4中**橙色**部分。

(3) **内存地址计算不同：**复杂地址计算方式（x86）与其他指令集的简单计算方式导致在翻译时需要额外指令。例如 x86 计算地址 $addr = base + index * scale + disp$; 其他的大多为 $addr = base + offset$。对应图2-4中**绿色**部分。

(4) **立即数加载：** x86 是变长指令集，支持编码 64 位立即数和 32 位地址偏移；而其他指令集均为定长指令集，立即数的编码空间有限。在翻译一条带有长立即数的指令时，需要额外的访存指令（从内存加载长立即数）或者是多条立即数加载指令（多个短立即数拼接成一个长立即数）。对应图2-4中**红色**部分。

(5) **间接跳转：**客户指令地址到宿主指令地址是非线性的，而间接跳转的目标地址在运行时才能知道，需要额外的几条指令查询间接跳转哈希表，导致性能开销。对应图2-4中**紫色**部分。

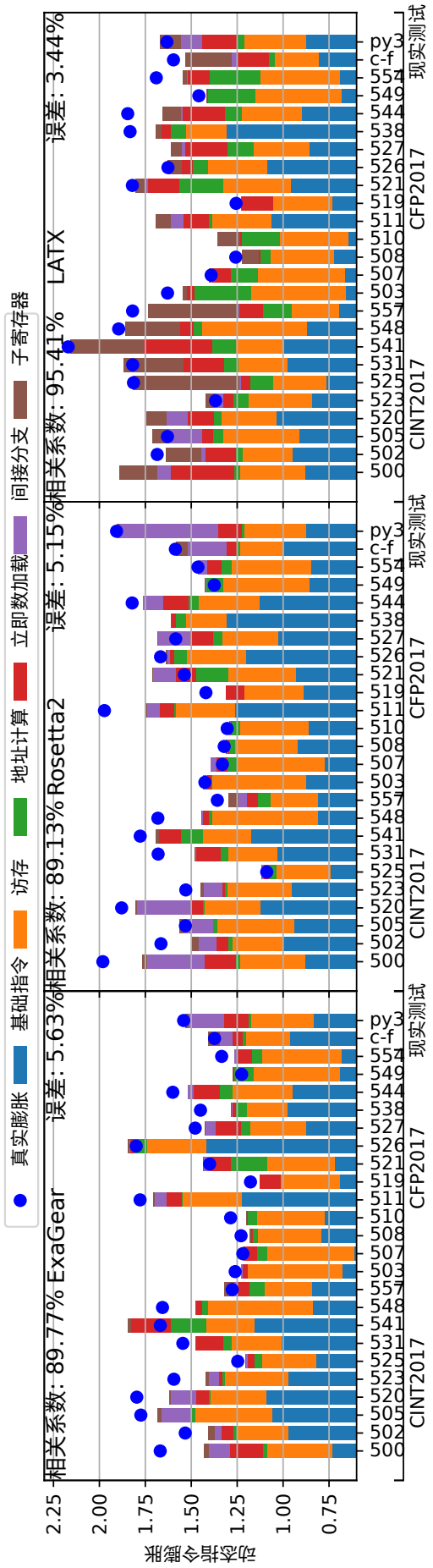

图 2-4　三个主流二进制翻译器指令膨胀来源分析图

Figure 2-4　The breakdown of instruction inflation for three major binary translators

以上五类主要开销也对应于上一节2.2.1理论分析的内容：前 4 个属于指令集语义差异，分别对应操作码、内存、立即数开销（操作数访问模式不同也属于复杂指令集和精简指令集的差异，也可归为操作码不同）；而间接跳转属于二进制翻译机制问题。由于 x86_64 通用寄存器只有 16 个，少于 ARM/LoongArch 的 32 个通用寄存器，所有没有寄存器溢出产生的开销。而翻译开销和代码缓存维护的开销在现代高性能二进制翻译器中也是可以忽略不计的。

以上这 5 类主要开销总结起来，主要包括**指令集语义差异**和**间接跳转开销**这两个方面，其性能开销很难通过软件优化来解决，接下来将介绍软硬协同二进制翻译器的相关工作。

2.3 软硬协同二进制翻译器

指令集作为软件和硬件沟通的桥梁，设计初衷是让软件和硬件设计解耦，使得软件开发者不需要关心硬件的细节，只需要关心软件的逻辑。而二进制翻译器则是一种将不同指令集之间的桥梁，它可以将不同指令集的二进制代码翻译为目标指令集的二进制代码，从而实现不同指令集之间的兼容。从设计层次来看，二进制翻译器相对指令集更加靠近硬件层，因此添加硬件支持的软硬协同二进制翻译器是一个比较自然的选择。

此外，二进制翻译器一般应用于生态迁移，在处理器厂商推出新型处理器时，为了保持对旧软件的兼容性，或者引入其他指令集已有的繁荣生态，需要引入二进制翻译器。处理器厂商也有动力在新型处理器中添加硬件支持，以提高二进制翻译器的性能，减少软件兼容的性能开销。本节主要介绍两个代表性的软硬协同二进制翻译器，分别是 Transmeta 公司的超长指令字兼容和龙芯公司的指令集扩展技术。

2.3.1 超长指令字

在处理器设计早期，由于当时的硬件晶体管资源有限，难以硬件发掘指令集并行性，因此超长指令字（Very Long Instruction Word, VLIW）技术应运而生。超长指令字技术通过软件来发掘指令级并行性，将多条指令打包成一条超长指令字，从而提高指令级并行性，提高处理器的性能。Transmeta 公司推出的 Crusoe 处理器就是一种基于超长指令字的处理器，其宣称在性能和功耗方面都优于同期的 x86 处理器[4]。

但由于初期超长指令字技术的复杂性和软件开发者对于超长指令字的不熟悉，导致超长指令字生态并不繁荣，因此 Transmeta 公司配套推出了软件上的代码转换器（Code Morphing Software, CMS），通过软硬协同的方式实现了对 x86 指令集的兼容。Transmeta 的 CMS 充当了翻译器和优化器的角色，使得基于非 x86 的 VLIW 处理器能够执行 x86 二进制代码。还提供了一个动态优化的执行环境，能够在运行时对代码进行优化，从而提高执行效率和能耗性能[4]。

如图2-5所示，代码转换器将从程序接收到的 x86 汇编代码指令翻译成微处

理器的本机指令（超长指令字）。通过这种方式，Crusoe 也可以模拟其他指令集架构，例如 Crusoe 也能将字节码翻译为其本机指令集中的指令来执行 Java 字节码。

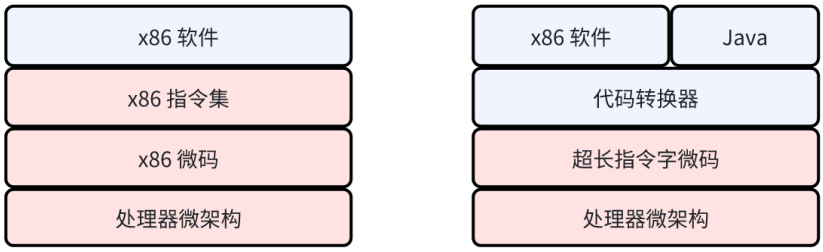

图 2-5　Transmeta 架构图

Figure 2-5　The architecture of Transmeta

注: 在超长指令字处理器上实现兼容 x86 的指令集，并支持 JAVA 程序。

这种架构有如下特点：

- 架构兼容性与灵活性：Transmeta 从初代 128 位 VLIW 处理器切换到第二代的 256 位 VLIW 处理器，都能通过软件翻译器保持软件向前兼容。
- 解释与翻译配合：冷客户代码通过解释执行，热代码通过翻译执行，提高了执行效率。
- 精确异常处理：Transmeta 在硬件中引入了一套提交和回滚机制，当发生异常时，配合软件翻译器回滚，能够保证异常处理的精确性。

尽管 Transmeta 公司提出了许多创新的技术，但并未在市场上取得成功，最终于 2007 年退出处理器市场。随着半导体工艺进步，处理器的晶体管资源变得丰富，基于硬件的乱序执行技术逐渐成熟，已经能够用硬件发掘指令级并行性，超长指令字技术的优势逐渐减弱，超长指令字技术的发展也逐渐停滞。但 Transmeta 公司在多架构兼容性方面的工作提供了宝贵的经验，成为软硬结合二进制翻译器的重要先驱者。

2.3.2　指令集扩展

客户和宿主指令集语义差异是二进制翻译器性能的主要瓶颈之一，为了解决这个问题，在宿主指令集中添加指令集扩展来接近客户指令集的语义是一种常见的解决办法。例如龙芯公司在 LoongArch 指令集中添加的二进制翻译扩展指令（Loongson Binary Translation, LBT）[2]。

龙芯早期使用 MIPS 指令集，并添加了便于 x86 和 ARM 二进制翻译的一系列自定义指令集，例如对 x86 EFLAGS 的支持、对 X87 浮点指令的支持、非对齐访存的支持等，形成了 LoongISA[9]。但由于 MIPS 中用户定义指令（UDI，user defined interface）槽位有限，导致了 LoongISA 的指令集扩展受限，所以在 2020

年龙芯公司发布了全新的、自主可控的、支持更多指令集扩展的 LoongArch 架构[2]。并从 3A5000 开始，龙芯处理器开始支持 LoongArch 架构，并配套使用 LATX 二进制翻译器进行 x86 应用程序的翻译。LoongArch 指令集中包括了对二进制翻译支持的扩展指令集，称为 LBT 指令集，用于支持 x86、ARM、MIPS 的二进制翻译。LBT 扩展指令集与 LATX 软件配合使用，可以实现更高效的二进制翻译，提高了龙芯处理器的软件兼容性。

指令集扩展虽然能够接近客户指令集的语义，但这样会增加宿主指令集的复杂度，更加剧了指令集的历史包袱，同时添加指令集扩展也难以支持多种指令集的兼容。

2.4 x86 处理器微码缓存

间接跳转开销虽然有很多软件优化的方法，例如软件返回地址栈优化函数返回[3]、软件跳转预测优化分支表[25] 等，但是由于客户和宿主地址空间的非线性映射问题，其开销很难通过软件完全消除。微译器可以通过硬件支持来消除间接跳转的性能开销，为此首先需要介绍 x86 处理器的微码缓存，这启发了微译器的设计。

x86 微码缓存（Uop Cache）是为了在 x86 CPU 后端实现超标量乱序执行并降低译码功耗而引入的关键组件[26]，本节对其介绍。

2.4.1 微码缓存介绍

x86 处理器是一种复杂指令集计算机，其指令集架构复杂，指令长度不固定，指令格式多样。为了实现超标量乱序执行，x86 CPU 后端需要将复杂的变长 x86 指令转换成类 RISC 格式的定长微码。微码的引入简化了指令集的关系，降低了 CPU 后端设计复杂度，使得指令能在后端能够更高效地乱序执行。

为了降低译码能耗、提高性能，研究者们引入了微码缓存，用于存储已经译码过的微码。如果这条指令已经译码过，就直接从微码缓存中读取微码，而不需要再次译码。[26] 论文中指出，在标准测试集中，微码缓存能消除 75% 的指令译码，在多媒体应用中，这个比例甚至高达 90%。对于 Intel P6 处理器来说，能节省 10% 左右的整机功耗。

具体而言，在 CPU 的前端，指令缓存和微码缓存是分开的，如图2-6。在前端译码阶段，系统会首先查询微码缓存，检查是否已经缓存了当前指令的微码。如果微码已经在缓存中，CPU 就直接读取微码并发射到后端执行。如果微码未缓存，系统则从指令缓存中取得指令，进行译码，并将译码结果存入微码缓存中，注意一个指令缓存行可能生成多个微码缓存行。

2.4.2 微码缓存组织形式

微码缓存的组织形式与指令缓存有所区别。指令缓存的地址被分为标记、索引和块偏移三部分，而微码缓存的地址被分为标记、索引两部分，其中地址低位

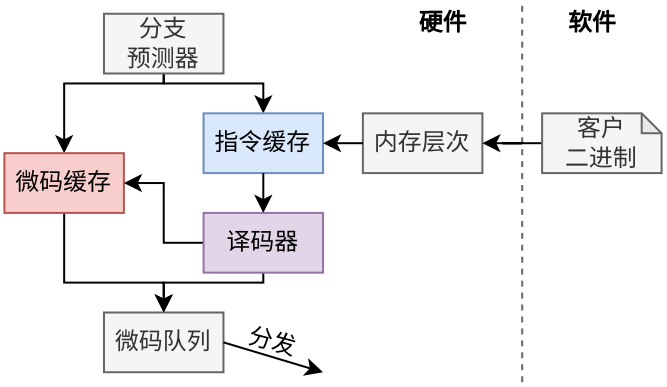

图 2-6　x86 处理器前端架构图

Figure 2-6　The architecture of x86 processor front end

注: 前端包括指令缓存和微码缓存。如果微码已在微码缓存中则直接使用；如果微码未缓存则从指令缓存中取得指令，进行译码并存入微码缓存中。

的块偏移也是标记的一部分。这是因为一条指令会被译码为多条数量不定的微码，原本的块偏移无法唯一标识一条微码。多条指令组成的指令基本块（以控制流指令结尾）译码成的多条微码，会尽可能放在一个微码行中，然后用指令基本块的首地址来索引这个微码行。

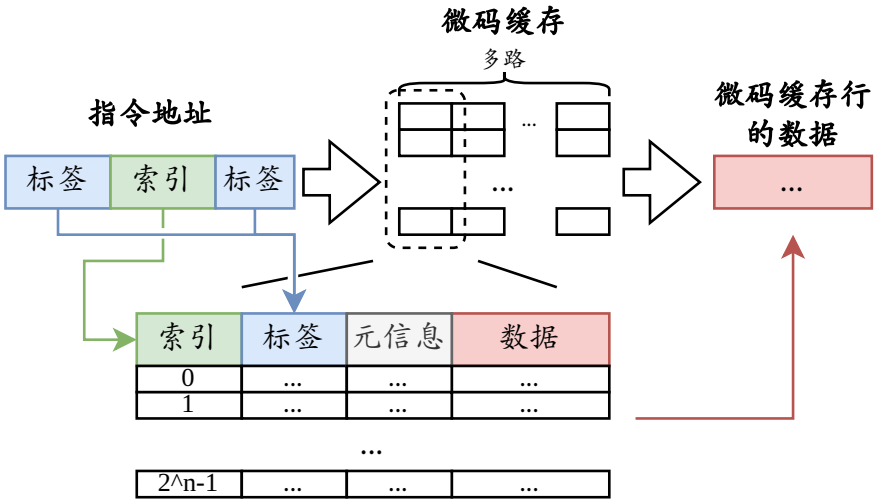

图 2-7　微码缓存的组织形式

Figure 2-7　The organization of uop cache

如图2-7，指令地址首先通过索引找到对应的微码组，然后通过标记找到对应的微码行。微码行开头有元信息存储有关这个微码行的信息，例如微码数量、立即数数量等。接下来读出对应的微码，发射到后端执行。

由于 x86 是变长指令集，微码是定长指令集，如果 x86 指令中有长立即数，没办法存放在定长的微码中，所以长立即数会单独放在微码行的最后。如图2-8，

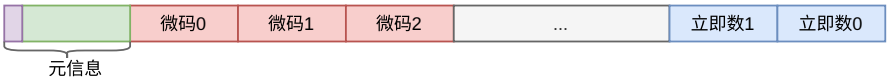

图 2-8　微码缓存一行的内容

Figure 2-8　The content of a uop cache line

微码行中微码和立即数是相向生长的，微码从前往后，立即数从后往前，如果相遇，说明微码行已满，需要分配新的微码行。

2.4.3　微码缓存行的结束条件

和指令缓存不同，指令缓存行是以单条指令为单位，一行会存储多条指令，没有空洞。而微码缓存行是以基本块为单位，一行存储一个基本块的微码，存储的微码数量不固定，当基本块短时，可能会有空洞。

微码指令会尽可能填满一个微码缓存行，但是当满足如下 3 个条件之一时，会结束当前微码缓存行的填充[26]：

(1) 遇到控制流指令并预测会跳转：当遇到控制流指令时，微码缓存会截断这一行，确保每个微码缓存行只包含一个基本块的微码。

(2) 微码缓存行已满：当微码缓存行中微码或者立即数填满一行时，会结束当前微码缓存行的填充。

(3) 遇到指令缓存行的结尾：这样保证一个指令缓存行产生的多个微码缓存行会放在同一个微码缓存组中（不同路），当需要刷新这一行指令缓存时，可以方便的通过索引找到对应的微码缓存组进行刷新，而不必刷新整个微码缓存。

此外，为了保证精确异常处理，需要保证一条宏码对应的多条微码一起提交，在后端重排序缓冲区（ROB）中需要添加对应的标志位，标记这些微码是一起提交的。（例如一条复杂的 x86 指令可能会翻译成多条微码，这些微码在后端原子化提交）。

如图2-9给出了一个指令缓存行对应多个微码缓存行的示例（这里的微码行忽略了立即数的存储）。假设指令缓存行为 8 字节，存储了 A、B、C、D 四条指令，长度分别为 2、1、3、2 字节，微码缓存行最多能存储 4 条微码。4 条指令译码得到的微码分别为 a0、a1、b0、c0、d0。则微码缓存行 1 存储了 a0、a1、b0，微码缓存行 2 存储了 c0、d0。本来 c0 是可以存储在微码缓存行 1 中的，但是由于 B 指令是控制流 ret 指令，满足结束条件 1，所以微码缓存行 1 结束，c0 存储在微码缓存行 2 中。可以看到结束条件导致了微码缓存行中存在空洞。

此外微码缓存行 1 和微码缓存行 2 是存放在同一个微码缓存组中不同路上的，意味着它们的索引相同，但是标记不同。当这一个指令缓存行需要被刷新，可以通过索引找到这这一组的所有微码缓存行一起刷新，而无需刷新整个微码缓存，保证了指令缓存行和微码缓存行的一致性。

x86 微码缓存的引入为 x86 架构的超标量乱序执行提供了重要支持，存储已经译码过的微码而无需再次译码，减少了译码的开销，降低了整体功耗。

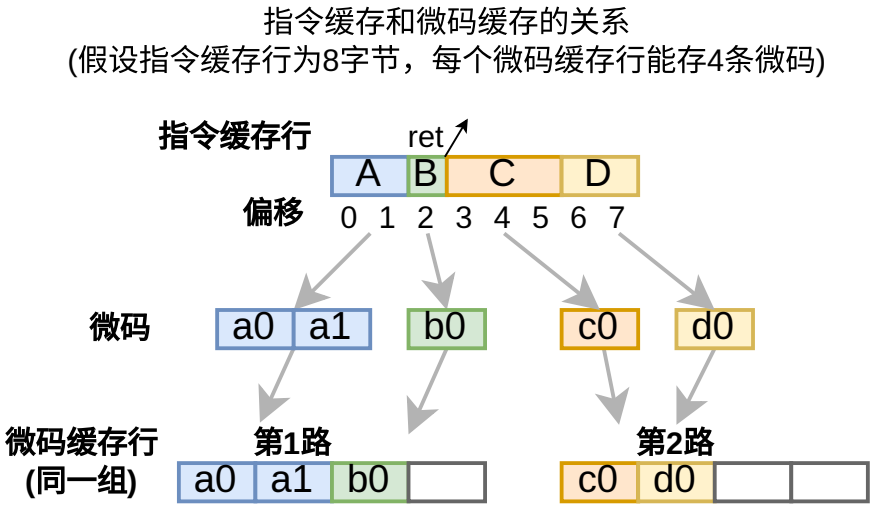

图 2-9　指令缓存和微码缓存关系

Figure 2-9　The relationship between instruction cache and uop cache

2.4.4　微码与二进制翻译相似性

本团队发现，面向 x86 的二进制翻译与 x86 处理器的微码有一定的相似性，具体表现在以下几个方面：

(1) 面向 x86 的二进制翻译会在软件层将复杂的 x86 指令翻译成宿主指令（通常为定长的 RISC 指令），而 x86 处理器前端在硬件层将 x86 指令翻译成类 RISC 的定长微码。

(2) 二进制翻译中使用软件层的代码缓存存储翻译后的宿主指令，而 x86 处理器使用微码缓存存储译码后的微码。

(3) 二进制翻译可以添加指令集扩展来缩小和客户指令语义差异，而 x86 处理器也可以不断修改内部的微码来适应新的 x86 指令集扩展。

此外，硬件实现的微码相对于二进制翻译器还有如下优势：

(1) 语义差异：内部的微码修改不会对外暴露，而指令集扩展技术会增加指令集的历史包袱。

(2) 间接跳转开销：硬件的微码缓存维护好了 x86 宏码地址和微码的映射关系，没有间接跳转开销；而二进制翻译器中处理间接跳转的开销较大。

(3) 精确例外处理：硬件后端 ROB 维护了微码的提交顺序，保证了精确异常处理；而二进制翻译器中需要额外的回退机制来保证精确异常处理。

这些相似性和优势启发了本团队提出一种软硬协同的二进制翻译技术——**x86 微译器**。

2.5　x86 微译器项目

本节首先介绍 x86 微译器的整体架构，然后详细介绍 x86 微译器的各个组成部分的设计与实现，包括 x86 微码指令、预翻译文件、翻译缓存等。最后介绍

x86 微译器是如何消除间接跳转开销的。

2.5.1 x86 微译器整体架构

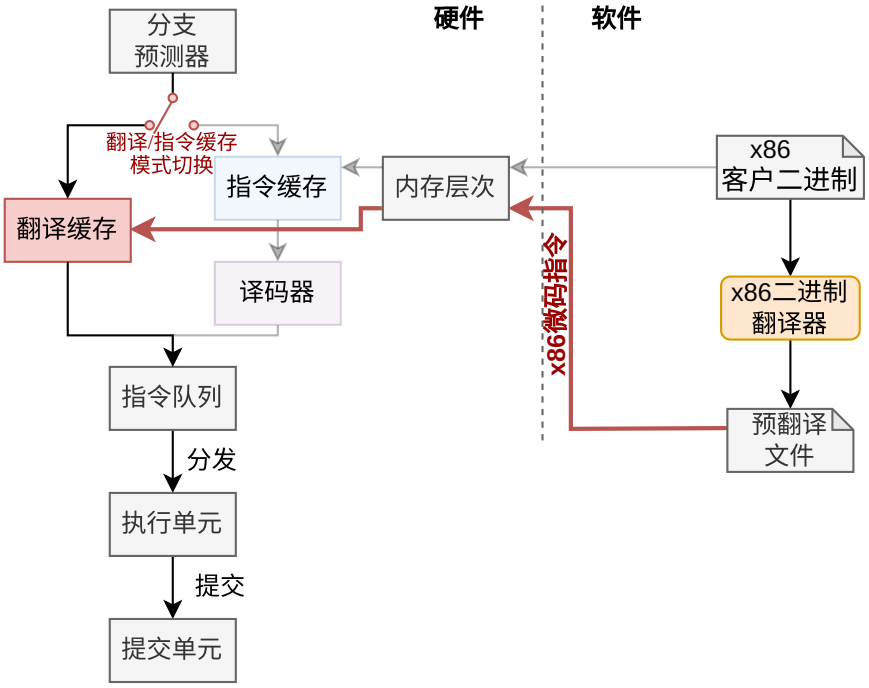

图 2-10 x86 微译器整体架构图

Figure 2-10 The overview of the micro-translator architecture

如图2-10展示了 x86 微译器的整体架构，相对于传统的 x86 处理器前端图2-6，x86 微译器在硬件侧和软件侧都做了一些改动。

在硬件部分，引入了**翻译缓存**（Translation Cache），该缓存作为一级缓存负责存储预翻译的微码指令集，替代了原本的微码缓存。翻译缓存的每行组织形式和微码缓存类似，都是前面部分存放微码指令，后面存放立即数，微码指令和立即数相向生长，中间可能有空洞，产生新的开销，后文会提出对应的优化方案。

此外与传统微码架构不同，微码缓存数据是直接来源于 CPU 和指令缓存，通过硬件译码器进行译码并存入微码缓存中；而翻译缓存通过软件的二进制翻译“译码”，透过内存层次（从内存加载到 L3 Cache, 再到 L2Cache，最后到微码缓存）进行填充，取代了传统的指令缓存和译码器的角色。在传统 x86 架构下，取指部件会同时查询指令缓存和微码缓存；而在 x86 微译器架构下，取指部件仅查询翻译缓存，硬件的译码器被软件的二进制翻译器取代。为了兼容传统的 x86 处理器模式，添加了一个模式切换逻辑，可以切换回使用硬件译码器进行译码，这样可以在不改变原有 x86 处理器的基础上实现 x86 微译器的功能。后文默认使用软件二进制翻译器进行译码，也就是翻译缓存的方式。

在软件部分，引入了静态和动态二进制翻译器。程序首先通过静态二进制翻译器被翻译成 x86 微码指令，并被写入预翻译文件，存储在硬盘中。在客户程序

执行阶段，预翻译文件被加载到内存中，程序计数器被设置为客户程序的入口。取指部件从翻译缓存中取指，若翻译缓存或内存层次命中，则从翻译缓存取指，发送到处理器后端执行，不断取指执行。若翻译缓存和内存层次均未命中（例如存在自修改代码等），说明客户指令还未翻译，此时会调用动态二进制翻译器进行实时翻译，并将翻译结果写入翻译缓存，再取指执行。

接下来详细介绍 x86 微译器各个组成部分的设计与实现，包括 x86 微码指令、预翻译文件、翻译缓存。

2.5.2　x86 微码指令

由于 x86 微码指令是需要存储在磁盘中的，并不像传统 x86 微码只是作为一个“暂时指令”存在于 CPU 运行期间，所以 x86 微码指令需要像普通指令集一样进行**编码**，二进制翻译器把指令**编码**为定长的 x86 微码指令，并存储于磁盘中。CPU 再从磁盘中加载 x86 微码指令，**解码**为 CPU 识别的信息。因此 x86 微码会被编码为 x86 微码指令集，有自己的编码方式。

2.5.3　预翻译文件

本节讲解预翻译文件的设计与实现，预翻译文件是存储预翻译的 x86 微码指令集的文件，是二进制翻译器的输出。

预翻译文件的设计思想在于，尽可能多的存储所有可能用到的 x86 微码指令，减少动态二进制翻译的开销。由于 x86 微码指令和普通指令在一级缓存中寻址模式不同，普通指令能根据行内偏移直接访问到指令，而 x86 微码指令只能根据行内第一条指令的地址访问到整行指令（在2.4小节有提到）。也就是说，x86 微码指令只能以一个微码行（一个基本块）为单位进行访问，而不能以一个微码指令为单位进行访问。如果出现了跳转到一个微码行的中间位置，那么就需要重新翻译这个微码行，这样会增加额外的开销。对于传统的 x86 处理器模式，这个开销是很小的，因为是硬件来译码并填充到微码缓存中，只需要 2 拍左右；而对于 x86 微译器架构，这个开销就会很大，因为需要调用软件的二进制翻译器进行实时翻译，可能需要上百拍的开销，这样的开销是无法接受的。因此提出了**重复存储**的概念，即存储任意一个地址开始的 x86 微码指令，这样就能保证在跳转到任意一个地址时，都能从预翻译文件中加载对应的微码指令。

2.5.4　翻译缓存

本节讲解翻译缓存的设计与实现，翻译缓存是 x86 微译器的核心组件，用于存储预翻译的 x86 微码指令集。

翻译缓存的结构整体和微码缓存类似，但是有一些细节上的差异，在此详细介绍翻译缓存行的组织形式，重点关注元信息和数据部分，其余的标签、有效位等部分和普通的一级缓存类似。一篇关于微码缓存的专利[27] 中提到了微码缓存的一行设置为 74byte。而 x86 微译器的翻译缓存行的大小为 64 字节，结构类似图2-8。

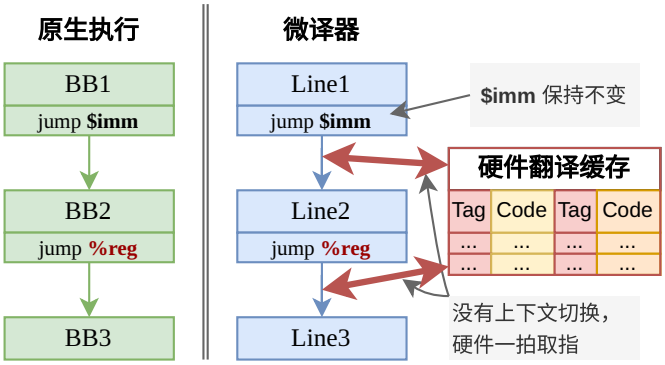

图 2-11　微译器高效处理间接跳转

Figure 2-11　The efficient handling of indirect jumps in the micro-translator

如图2-11，展示了翻译缓存如何高效处理间接跳转。相对于软件二进制翻译器中对间接跳转的处理2-3，硬件上的微码缓存天然维护好了 x86 指令地址和微码指令地址的非线性映射关系，可以一拍查询缓存获取到间接跳转对应的微码指令地址，而不需要软件上复杂的哈希表查询逻辑，这样可以消除间接跳转的开销。

2.6　本章小结

本章介绍了二进制翻译的相关工作，包括 4 种主流的软件二进制翻译器，发现其性能难以接近原生性能。接下来对软件二进制翻译器的性能开销进行分析，发现指令集语义差异和间接跳转的性能开销是二进制翻译器性能的主要瓶颈。然后介绍一些已有的软硬协同二进制翻译器是如何缓解这些性能瓶颈的，例如添加指令集扩展来缩小指令集语义差异等。为了更进一步消除间接跳转开销，引出了复杂指令集 x86 处理器中微码缓存的概念，启发了 x86 微译器的设计。最后介绍团队项目——x86 微译器的架构设计和实现，并讲解其是如何消除间接跳转的性能开销，进而提升翻译性能的。

第 3 章　RISC-V 微译器设计与实现

x86 微译器借鉴 x86 处理器微码缓存的设计思路，实现了一种高效的、软硬协同的、x86-to-x86 的二进制翻译器，并通过翻译缓存消除了间接跳转的开销，达到了较高性能。但目前 x86 微译器只能支持 x86 指令集，无法支持其他指令集，而本文的主要工作是为了支持 RISC-V 指令集，验证其对多指令集的支持能力，同时为未来支持更多指令集打下基础，最终实现在单一硬件平台上实现多指令集的共存，并尽可能地接近原生执行效率。

本节主要讲解为了让 **x86 微译器**支持 RISC-V 架构需要做的工作，主要包括 RISC-V 指令集的支持、RISC-V ABI 的支持等。

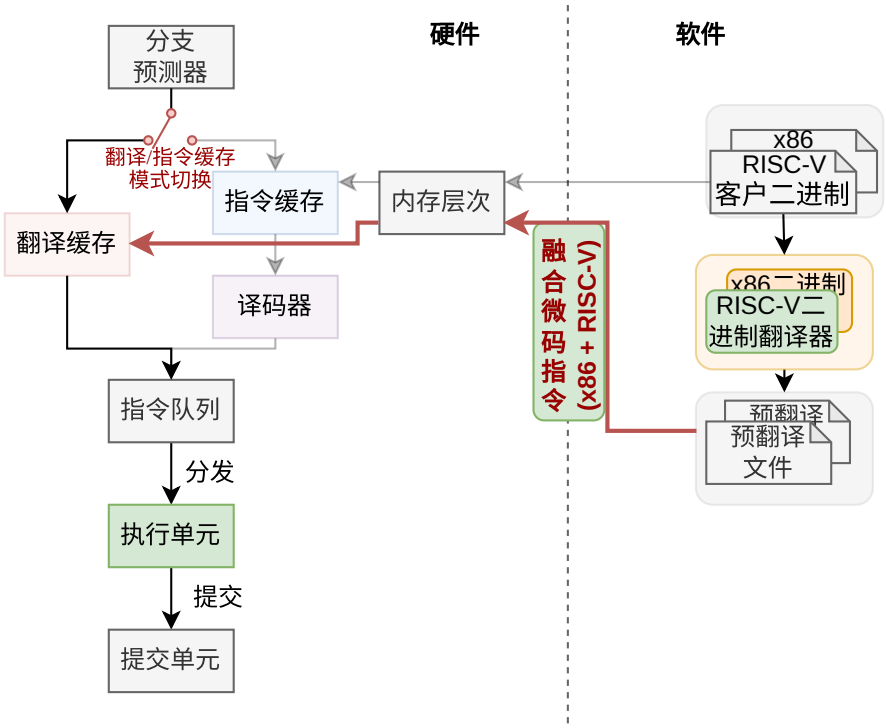

图 3-1　RISC-V 微译器整体架构图

Figure 3-1　The overview of the RISC-V micro-translator architecture

注: 绿色部分为本文重点工作，在 x86 微译器上支持 RISC-V 架构所需要的修改。

如图3-1讲解了在 x86 微译器上添加 RISC-V 架构所需要的修改，其中绿色部分为本文主要工作，包括软件层添加 RISC-V 二进制翻译器，硬件层添加对应的融合微码指令并在后端执行单元添加对应的执行逻辑。

3.1 软件层 RISC-V 二进制翻译器

本文首先重构了微译器中软件层二进制翻译器的代码，使其支持如下 3 个设计目标：

(1) 可扩展性强：能够方便支持多种指令集的翻译。

(2) 代码发现性强：能够发现更多的二进制代码进行翻译，减少动态二进制翻译的介入。

(3) 可维护性强：能够方便的维护和更新翻译器。

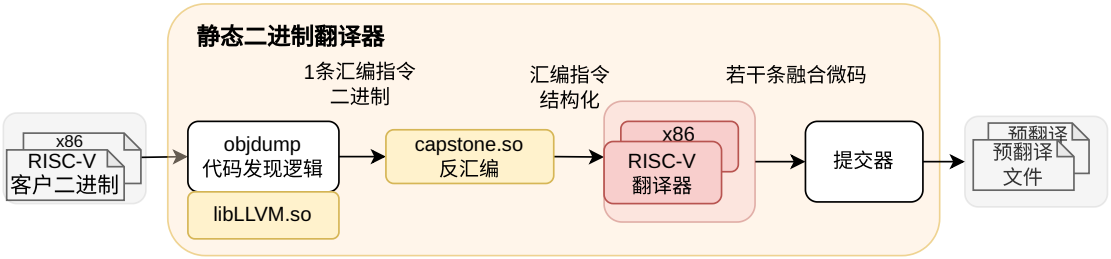

图 3-2 静态二进制翻译器架构图

Figure 3-2 The architecture of the static binary translator

因此在实现上，如图3-2，尽可能的将翻译器的功能模块化，分为代码发现模块、反汇编模块、翻译模块、提交模块。代码发现模块用于尽可能的发现更多的二进制代码，例如通过符号表、字符串表等信息发现更多的代码；反汇编模块用于将 ELF 文件中的二进制代码转换为汇编代码；翻译模块用于将汇编代码翻译为融合微码指令；提交模块用于将翻译后的融合微码指令组织成翻译缓存行的形式，写入预翻译文件。

此外将架构相关的代码和架构无关的代码分开，架构相关的代码只有翻译模块，架构无关的代码主要是代码发现模块、反汇编模块和提交模块。这样只需要添加新的翻译模块，就能够支持新的指令集的翻译，而不需要修改其他模块的代码。

最后会希望尽可能复用现有的工具，例如 LLVM objdump 工具用于代码发现，这样能够减少二进制翻译器的实现难度，提高开发效率；使用 capstone 工具用于反汇编，这样能够减少反汇编的实现难度，提高反汇编的准确性。此外选用的 LLVM objdump 和 capstone 工具都天然支持多种指令集，这样能够方便未来支持多种指令集的翻译。

3.2 融合微码设计

本节详细介绍重要的一个概念——融合微码。“融合”代表它能融合了多种指令集的特征，包括 x86 和 RISC-V，未来还可以支持 ARM，MIPS 等其他指令集，或许叫统一微码也更好理解。但融合并非简单的把所有指令集简单拼接就好，这会造成指令集冗余，挤占有限的编码空间。而是需要对各个指令集的特征

进行分析，找到共性，抽象出一种更加简洁的指令集，这就是融合微码的设计思想。

对于 RISC-V 微译器而言，融合微码是在前文2.5.2节中 x86 微码指令集进行扩充得到的。本节首先讲解**融合微码**的编码方式，通过调整融合微码的编码方式，使得微码能够支持多种指令集的翻译，同时保持简洁性、可扩展性。

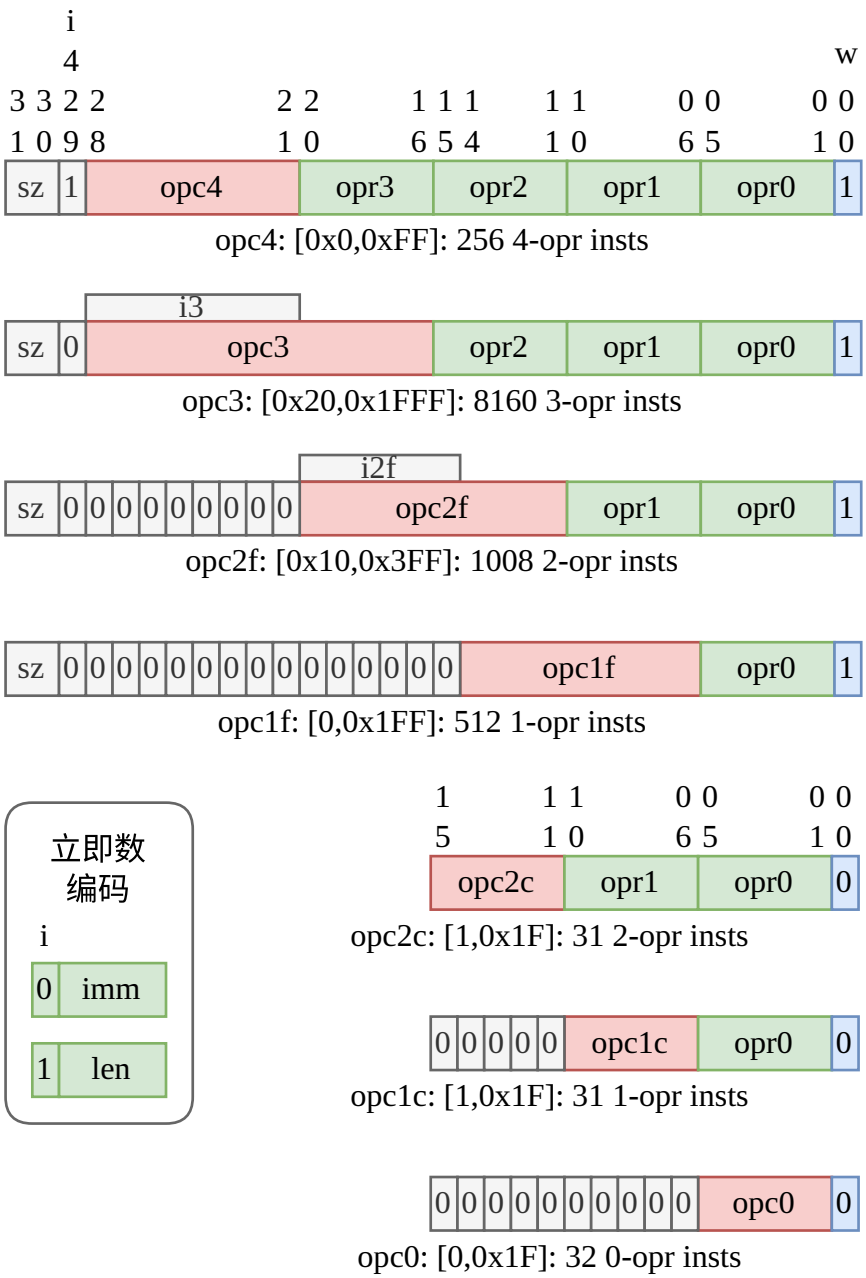

图 3-3　融合微码编码方式

Figure 3-3　The encoding of the fused microcode

融合微码的设计准则是需要高效的支持多种指令集，并且保持精简指令集（RISC）的特点。它的目标是同时支持 x86、RISC-V、ARM 等多种指令集，而这些指令集的编码方式差异较大，例如 x86 是变长指令集，长度可以从 1 字节到 15 字节不等，而 RISC-V、ARM 是固定长度指令集，长度为 4 字节；不同指令集的寄存器数量、立即数长度、指令格式等也有很大差异；对于同一个立即数，可能存储在不同的位置，例如 RISC-V 的 12 位立即数会根据不同的指令格式存储在不同的位置，可能分散存储在指令开头和中间。融合微码希望屏蔽这些差异，尽可能简洁，同时保持高效。

此外，参考 RISC-V 中压缩指令集的特点，本文将融合微码指令尽可能编码为 2 字节，以减少一条融合微码的占用空间，提高一行翻译缓存行中微码数量，提高翻译缓存的命中率，提高性能。因此融合微码指令同时支持 2 字节和 4 字节编码，如图3-3所示，前 4 行为 4 字节编码，后 3 行为 2 字节编码，后文默认把 4 字节编码指令称为**标准指令**，2 字节编码指令称为**压缩指令**，根据指令最低位是否为 0 来区分。

如图3-3，展示了融合微码的编码方式，主要包括两部分：操作码、操作数；其中操作数又分为立即数、寄存器、操作数长度。

3.2.1 操作码

对于操作码，按照操作数的数量分为 0、1、2、3、4 个操作数，对应的操作码（opcode）为 opc0、opc1、opc2、opc3、opc4。其中 opc4、opc3 只能在标准指令中使用，因为压缩指令只有 2 字节，不足以存储 4 个操作数的信息。而 opc0 只能在压缩指令中使用，不需要存储操作数信息。而 opc1、opc2 可以在标准指令和压缩指令中都可以使用，按照后缀区分为 opc1f、opc1c、opc2f、opc2c，f 后缀表示标准指令，c 后缀表示压缩指令。

根据编码槽位可知，对于标准指令：opc4 指令支持 256 个操作码，opc3 指令支持 8160 个操作码，opc2f 指令支持 1008 个操作码，opc1f 指令支持 512 个操作码；对于压缩指令：opc2c 指令支持 31 个操作码，opc1c 指令支持 31 个操作码，opc0 指令支持 32 个操作码。可见标准指令的编码空间更大，支持更多的操作码，而压缩指令的编码空间较小，只支持少量操作码，需要谨慎选择更常用的指令作为压缩指令。

3.2.2 操作数长度

对于操作数长度，按照操作数的长度分为 1、2、4、8 个字节，分别对应 8 位、16 位、32 位、64 位操作数，这需要 2 比特来编码，存放在标准指令的最高位（sz 位，size）。对于压缩指令，由于编码空间很有限，需要的操作数长度只能编码在操作码 (opc) 中。这主要是为了支持 x86 指令，由于历史原因，x86 支持了多种操作数长度，例如 add 指令支持 8 位、16 位、32 位、64 位操作数；而现在的精简指令集，一般只支持固定长度的操作数，例如 RISC-V 的指令集，只支持 32 位和 64 位操作数，分别使用 addw 和 add 指令。

操作数长度本质上也可以看做操作码 (opc) 的一部分，放在指令最高位只是为了方便解码，减少解码的复杂度。但在后续实现中发现可能不同的操作数会使用不同的长度，例如 opc4 指令中的第一个操作数可能是 4 字节，第二个操作数可能是 2 字节，那么这时候的 sz 位就没有作用了，因此本文的设计中，sz 位只是一个辅助位，如果操作数长度不同，需要在操作数的编码中明确指定。

3.2.3　立即数

目前所有指令集都会把立即数编码在指令中，但由于 x86 是变长指令集，x86 最多支持 64 位立即数，32 位的偏移量 (Displacement)；而 RISC-V 最多支持 12 位和 20 位立即数，AArch64 支持 12 位和 16 位立即数，并且都没有偏移量的概念。因此融合微码需要支持多种立即数的编码方式，同时保持简洁性。

对于图3-4展示了 RISC-V 的立即数分布（包含了访存指令的地址偏移），同样发现接近 10% 的指令使用了接近 12 位的立即数。如果直接将 12 位立即数直接编码到融合微码中，会导致编码空间不够，难以支持多种指令集、多种立即数的编码方式。

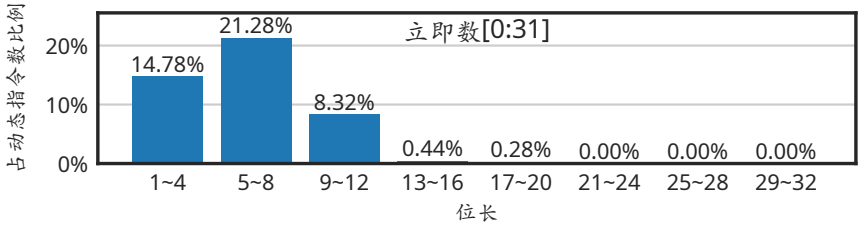

图 3-4　RISC-V SPEC2017 中立即数分布

Figure 3-4　The immediate Distribution in RISC-V SPEC2017

因此参考微码缓存的设计思路，将短的立即数直接编码在操作数中，长的立即数存储在微码缓存行的立即数区域。由于指令编码空间有限，并且出于简洁性的考虑，只把 4 比特以下的立即数编码在操作数中，更长的立即数存储在立即数区域，前者称为**直接立即数**，后者称为**间接立即数**，用 5 比特的操作数 (opr) 最高位来区分；如果为 0，表示直接立即数，把短立即数直接编码在操作数中；如果为 1，表示间接立即数，把长立即数的长度（len）编码在操作数中，而真正的立即数需要根据立即数的索引到立即数区域中查找。存储立即数长度 (len) 是为了支持**变长立即数**，例如 0x1ff，需要 2 字节来存储，而 0x1ff00，需要 3 字节来存储，这样可以根据 len 来到立即数区域中取出对应长度的立即数，并不需要每个立即数都使用 4 字节存储，而是按需存储，节省编码空间。

3.2.4　寄存器

目前主流指令集都支持的通用寄存器数量不同，例如 x86 有 16 个通用寄存器，RISC-V 和 AArch64 有 32 个通用寄存器。过少的通用寄存器数量会导致寄存器分配困难，会溢出到内存中，增加访存开销；过多的通用寄存器数量会导致

更多的寄存器编码，增加编码空间，因此在设计中，只支持 32 个通用寄存器的编码，通过 5 比特的操作数 (opr，operand) 来寻址寄存器。

由于翻译器还需要使用一些临时寄存器，例如用于存储中间结果的寄存器，对于 32 个通用寄存器可能不够用，因此还需要支持**临时寄存器**，临时寄存器类似于 x86 的段寄存器或者 RISC-V 的 CSR 寄存器，只能用于存储临时数据，不能用于通用寄存器的操作。所以还需要添加一些搬运指令，用于临时寄存器和通用寄存器之间的数据搬运。

对于浮点寄存器，目前主流指令集都是使用单独的浮点寄存器（X87 会使用浮点栈，本文并不支持，目前 gcc 编译生成的代码也基本不会使用 X87），例如 x86 SSE 指令集有 16 个 128 位的 XMM 寄存器，RISC-V 有 32 个 64 位的浮点寄存器。综合考虑，在设计中，只支持 32 个 64 位浮点寄存器的编码，同样通过 5 比特的操作数 (opr) 来寻址浮点寄存器。

如图3-5，展示了融合微码的寄存器编码方式，主要包括通用寄存器、浮点寄存器和临时寄存器。

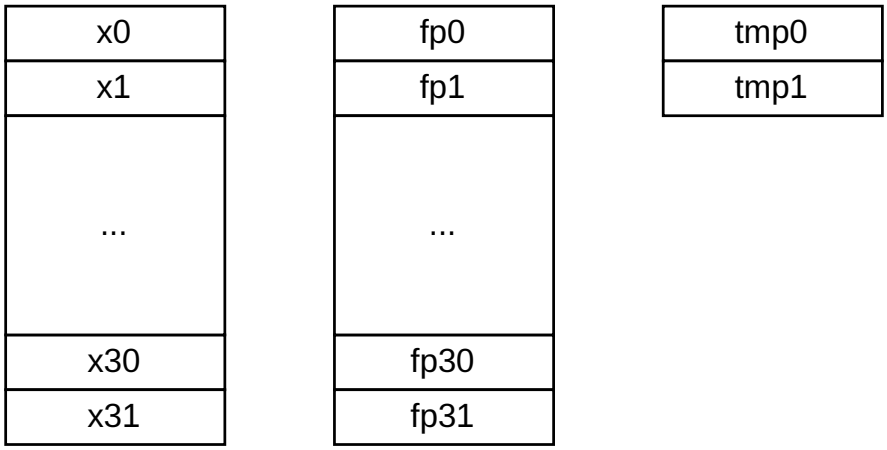

图 3-5　微译器寄存器设计

Figure 3-5　The design of the micro-translator registers

3.3　RISC-V 指令翻译

上节中的融合微码编码设计是支持多种指令集的基础，本节主要介绍 RISC-V 指令和 x86 指令的语义差异，以及如何通过添加微码指令来支持 RISC-V 指令的高效翻译。

设计一套高效完备的融合微码“指令集”是一个长期的工程，这不是本文研究重点。为了快速验证微译器的实际效果，同时减少“指令集设计”产生的额外性能影响，因此目前的融合微码是基于原本的 **Gem5 x86 微码**上**扩充**的，基于这种扩充的方式，所有 x86 指令都可以直接翻译成一条或者多条融合微码指令，而 RISC-V 指令的翻译则需要分析 RISC-V 指令和 x86 指令（x86 微码）的语义差异，然后添加新的融合微码指令。

如图3-6，由于 x86 指令默认符号扩展，而 RISC-V 指令同时支持符号扩展和零扩展，为了尽可能实现一条 RISC-V 指令翻译成一条融合微码指令，所以需要添加适当的“零扩展”微码指令。

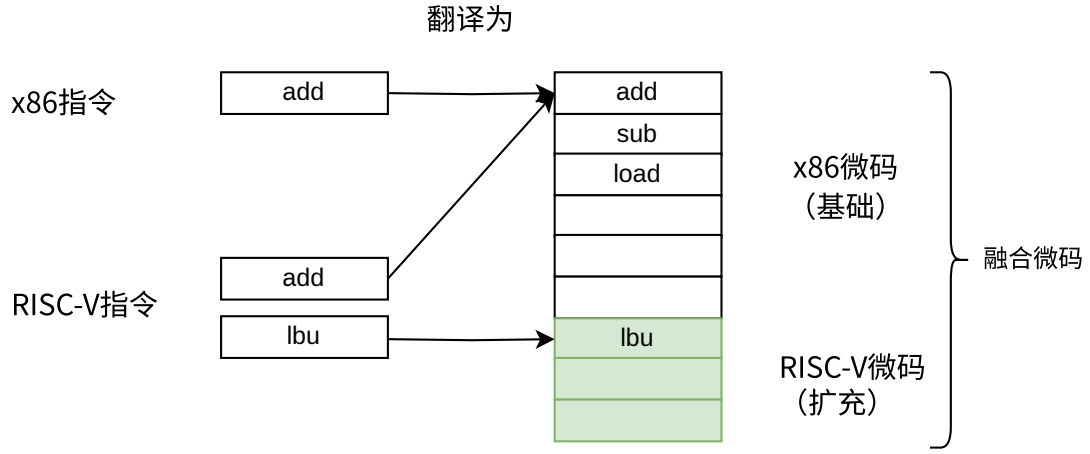

图 3-6　指令翻译到融合微码过程

Figure 3-6　The process of translating instructions to fused microcode

注: x86 add **和** RISC-V add **指令都翻译成融合微码** add **指令，而** lbu **（加载字节并无符合扩展）指令，需要额外添加** lbu **微码指令**。

融合微码添加有两大设计原则：

(1) **一对一原则**：一条 RISC-V 指令尽可能翻译成一条融合微码指令，减少指令翻译的膨胀率，提高性能。

(2) **复用原则**：尽可能复用现有的 x86 微码，减少硬件侧新指令的添加，减少硬件开发的复杂度。

目前已经完成所有的 RISC-V 指令（包括 IMAFDC 指令集，简称 GC 指令集）到融合微码的翻译，完整的翻译表格见附录A-1。

不同类别的指令翻译如下：

3.3.1　整数指令

整数指令（I）主要包括移位指令、算数指令、逻辑指令、比较指令、分支指令、访存指令等。大部分复用 x86 微码，例如加法 add 指令复用 x86 的 addflags 指令（会计算 EFLAGS，但 RISC-V 不会使用它）；其余的无法一对一翻译的指令，需要添加新的微码指令。

例如对于大多以 w 结尾的指令，例如 addw 指令，这是 RISC-V64 位指令集中的 32 位加法指令，默认进行符号扩展并填入 64 位寄存器，而 x86 微码中没有这种指令，所以需要添加一个 addw 微码指令。而对于 auipc 指令，这是 RISC-V 中的一个特有指令，需要添加一个 auipc 微码指令。访存指令中的 lb、lh、lw、lbu、

lhu 指令，这些指令需要额外添加微码指令，用于加载字节、半字、字，并进行符号扩展或者零扩展。

此外，翻译出的指令还需要利用到硬件原有特性，例如 x86 的 call 和 ret 指令会利用硬件返回栈（return stack，RAS），用于把返回地址压栈，并在 ret 指令中弹栈，这样可以减少分支预测错误率。而 RISC-V 会使用 jal、jalr 指令来作为函数调用和返回，所以需要添加 jal_call、jalr_ret 微码指令，并充分利用硬件的返回栈机制。

3.3.2 乘除法指令

乘除法指令（M）主要包括乘法、除法、取余等指令。对于 32 位乘法，RISC-V 指令默认需要 2 条指令才能得到完整的 64 位乘法结果，而 x86 微码中默认只用了 1 条微码，所以对于这类乘除法指令，需要添加新的微码指令。以 w 结尾的指令，例如 mulw 指令，默认进行符号扩展并填入 64 位寄存器，也需要添加新的微码指令。

3.3.3 原子指令

原子指令（A）包括原子加法、原子比较等指令。由于目前尚未实现多线程支持，使用普通的指令替代原子指令，暂时没有考虑原子性问题。如图3-7展示了一条原子加法指令的翻译过程，会翻译成 6 条融合微码指令。这也是唯一出现的一类 RISC-V 指令翻译成了多条融合微码指令，其余指令都是一条指令翻译成一条融合微码指令。但由于原子指令出现的频率极低，在单线程情况下对性能影响不大。

3.3.4 浮点指令

浮点指令（F，D）包括单精度浮点 F 和双精度浮点 D 指令。由于 x86 SSE 模式下的浮点指令和 RISC-V 浮点指令都遵循 IEEE-754 标准，所以很多可以直接复用 x86 的浮点微码指令。出于简洁考虑，对于特殊的舍入模式暂未考虑和 x86 的差异，对于运行 SPEC CPU 2000 中的浮点测试，没有发现问题。

此外，x86 SSE 模式下使用 16 个 128 位的 XMM 寄存器，用于存储浮点和整数，而 RISC-V 使用 32 个专用的 64 位的浮点寄存器，微译器使用 32 个 64 位浮点寄存器并直接复用 x86 的浮点微码指令。而对于 RISC-V 的浮点寄存器和通用寄存器之间的数据搬运，x86 没有这种指令，所以需要添加新的浮点寄存器搬运指令，用于浮点寄存器和通用寄存器之间的数据搬运。对于浮点转换指令，例如 fcvt.w.s 指令，这是 RISC-V 中的一个特有指令，需要添加一个 fcvt.w.s 微码指令。

3.3.5 压缩指令

压缩指令（C）：为了压缩指令长度，把常用的 4 字节指令替换为 2 字节长度的压缩指令。会尽可能把原本就是 2 字节的 RISC-V 压缩指令直接翻译成 2 字

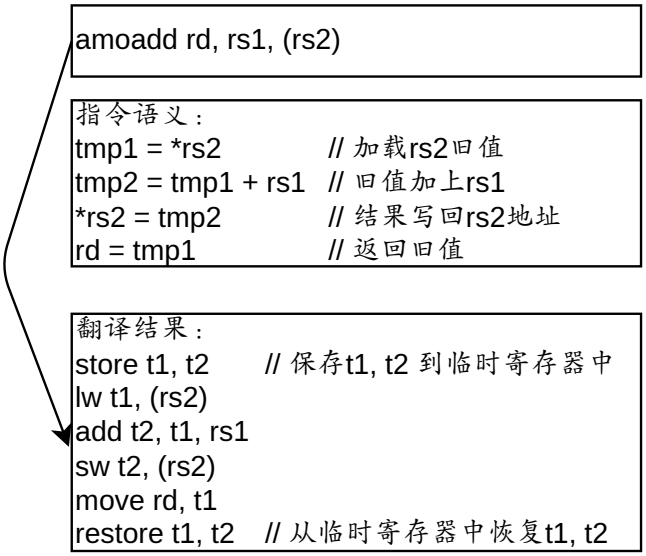

图 3-7　RISC-V 原子加法指令翻译过程

Figure 3-7　The translation process of RISC-V atomic addition instruction

注: 会使用两个临时寄存器，需要前后都进行寄存器保存和寄存器恢复。暂未实现多线程，没有考虑原子性问题。

节的融合微码指令，例如 c.add 指令翻译为 c_add 微码指令。

3.3.6　小结

累积统计数目如下：原本 x86 指令共有数千条，x86 微码也有 500 余条。为了添加 272 条 RISC-V 指令，本文并没有简单添加 272 条微码指令，而只是添加了 41 条整数相关指令，10 条浮点相关指令，累计添加 51 条微码指令。如图3-8展示了指令翻译的数量统计。

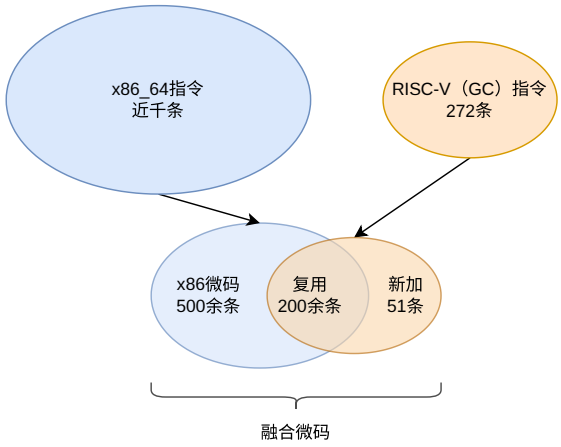

图 3-8　x86、RISC-V 指令翻译到融合微码的数目统计

Figure 3-8　The number of x86 and RISC-V instructions translated to fused microcode

3.4 硬件层执行单元修改

本节主要介绍硬件层的修改，由于添加了新的融合微码指令，需要修改硬件层的微码执行单元，使其能够支持新的融合微码指令。由于本文在 Gem5 模拟器的基础上进行修改，因此只需关注指令的功能实现以及一条指令的执行周期数。

本文添加的融合微码指令和原有的 x86 微码相比，功能上没有太大的区别，主要是添加了一些新的指令，例如 lbu、addw、auipc 等，这些指令都能在硬件层找到对应的功能单元，只需要修改微码执行单元的控制逻辑即可，并且可以在一个周期内完成。而对于乘法、除法指令，和原本 x86 微码一样，默认进入对应的乘法、除法功能单元，分别需要多个周期完成。

3.5 RISC-V ABI 差异处理

ABI 全称为 Aplication Binary Interface（应用二进制接口），为程序提供了一种与操作系统和硬件交互的接口，定义了程序的二进制接口，包括数据类型、函数调用约定、系统调用等内容。不同指令集的 ABI 也是不同的，在二进制翻译系统中需要维护这种差异性，保证程序运行的正确性。ABI 包括内容比较多，其中主要的包括系统调用传参、初始化栈等问题，本节介绍 RISC-V 微译器项目是如何处理 ABI 差异的。

3.5.1 系统调用差异

如表3-1所示，总结了 x86 和 RISC-V 的系统调用号和参数传递方式的差异。x86 的系统调用号存储在 rax 寄存器中，返回值也存储在 rax 寄存器中，参数传递方式为 rdi, rsi, rdx, r10, r8, r9。而 RISC-V 的系统调用号存储在 a7 寄存器中，返回值存储在 a0 寄存器中，参数传递方式为 a0, a1, a2, a3, a4, a5。因此需要在 RISC-V 的二进制翻译器中，将 RISC-V 的系统调用号和参数传递方式转换成 x86 的系统调用号和参数传递方式。

参数传递的差异可以通过把 x86 的参数寄存器和 RISC-V 的参数寄存器映射到相同的微码寄存器即可，如表3-2所示，所有的黄色寄存器就是 6 个参数传递寄存器。系统调用号差异同理也可解决。

指令集	系统调用号	返回值	参数 1	参数 2	参数 3	参数 4	参数 5	参数 6	其他参数
x86	rax	rax	rdi	rsi	rdx	r10	r8	r9	栈传递
RISC-V	a7	a0	a0	a1	a2	a3	a4	a5	栈传递

表 3-1 x86 和 RISC-V 的系统调用差异

Table 3-1 The difference between x86 and RISC-V system calls

3.5.2　寄存器映射

寄存器映射一直是二进制翻译中一个重要的研究课题。如表3-2所示，展示了 x86 和 RISC-V 映射到微码的的寄存器映射表。对于 x86 寄存器映射到微码寄存器较为容易，因为 x86 只有 16 个通用整数寄存器，而微码定义了 32 个通用寄存器，所以只需要把 x86 寄存器固定映射到前 16 个通用寄存器即可，还可以把一些常用的寄存器（例如 AH，BH 等子寄存器和段寄存器）映射到后面的寄存器中，剩余的一些寄存器作为临时寄存器供二进制翻译器使用。

表 3-2　x86 和 RISC-V 到微码的寄存器映射表

Table 3-2　The register mapping table from x86 and RISC-V to microcode

微码	x86	RISC-V	微码	x86	RISC-V	微码	x86	RISC-V	微码	x86	RISC-V
0	RAX	A7	8	R8	A4	16	T0	Zero	24	T8	S8
1	RCX	TP	9	R9	A5	17	T1	RA	25	T9	S9
2	RDX	A2	10	R10	A3	18	T2	S2	26	T10	S10
3	RBX	GP	11	R11	A6	19	T3	S3	27	T11	S11
4	RSP	SP	12	R12	T1	20	T4	S4	28	T12	T3
5	RBP	T0	13	R13	T2	21	T5	S5	29	T13	T4
6	RSI	A1	14	R14	S0	22	T6	S6	30	T14	T5
7	RDI	A0	15	R15	S1	23	T7	S7	31	T15	T6

但是对于 RISC-V 映射到微码方案，由于 RISC-V 本身就有 32 个通用寄存器，固定映射到 32 个微码寄存器后，就没有额外的临时寄存器供二进制翻译器使用了。对于微译器项目，得益于软硬件协同设计的基本原则，本文额外添加了两个微码寄存器作为临时寄存器，这两个寄存器只对二进制翻译器可见，不对 RISC-V 应用程序可见，类似于 x86 “段寄存器”，属于特殊寄存器，具体的使用场景参考图3-7。

此外，相对于传统的软件二进制翻译器，微译器不需要维护源寄存器块、源内存镜像等信息，不需要管理代码缓存，因此可以减少翻译器由于翻译机制占用的寄存器数量，降低寄存器压力。

3.5.3　栈的初始化

由于我们目前关注于用户态二进制翻译器，不太涉及系统态指令的翻译和处理操作系统等概念，但是当加载运行不同指令集的程序时，在 libc 库眼中，操作系统已经准备好了这个程序的初始化栈等信息，例如 argc，argv 参数、环境变量指针等，对于 x86 和 RISC-V 程序，这个初始化栈是不同的，需要不同的处理。

用户态模拟的 Gem5 模拟器，会扮演操作系统的角色，负责加载程序、初始化栈、运行程序等操作。因此需要修改 Gem5 在启动 RISC-V 程序的初始化栈，把 RISC-V 相关的 argc、argv、envp 等信息放到正确的位置。

而对于真实处理器，需要在操作系统层面进行修改，把 RISC-V 的 ABI 转换

成 x86 的 ABI，并修改加载器用以识别不同指令集的程序并初始化栈，这是一个较为复杂的工作，不在本文的研究范围内。

3.6 本章小结

本章主要介绍了在微译器中添加 RISC-V 支持所需要的工作，包括软硬件层的设计和修改。首先把软件层二进制翻译器重构为适合多架构翻译的框架、将体系结构相关和无关的代码分离；接下来讲解了融合微码的设计和实现，包括操作码、操作数长度、立即数、寄存器等的设计，如何更好的支持多种指令集；然后介绍了 RISC-V 指令的翻译，主要是 RISC-V 和 x86 指令的语义差异，如何通过添加尽可能少的微码指令来支持 RISC-V 指令；其次介绍了硬件层的修改，主要是添加新的融合微码指令对具体功能部件的修改；最后介绍了 RISC-V 和 x86 的 ABI 差异，RISC-V 微译器通过寄存器映射来处理系统调用差异、并处理了栈的初始化问题。

第 4 章　RISC-V 微译器优化方案

前文2.5.4中提到了翻译缓存的引入能消除间接跳转的开销，上一节3.3中添加适当的融合微码指令能缩小指令集语义差异，共同提升二进制翻译器的性能。然而，RISC-V 微译器的引入也会带来一些新的问题，产生新的性能瓶颈，本节将首先分析 RISC-V 微译器的开销来源，然后针对这些问题提出一些优化方案，从而减缓这些性能瓶颈，进一步提升性能。

4.1　RISC-V 微译器开销来源

RISC-V 微译器的主要开销来源于磁盘、内存、缓存三个方面，前两者来源于预翻译文件中**重复存储**机制，后者来源于翻译缓存的**存储效率**问题。重复存储机制是为了减少动态二进制翻译的开销，提高性能，但是会增加磁盘和内存的开销；存储效率问题是由于翻译缓存行的固定长度，导致存储效率较低，缓存缺失率较高，进而影响性能。本小节主要分析重复存储机制，下一小节将分析存储效率问题并给出优化方案。

预翻译文件的作用和可执行文件（Linux 中为 ELF 格式文件）类似，是存储程序的二进制指令的文件格式，处理器会从预翻译文件中加载融合微码指令集，进而译码执行。预翻译文件的数据段和 ELF 文件数据段相同，但是由于重复存储机制，代码段相对于 ELF 文件的代码段会膨胀 64 倍（64 为翻译缓存行长度），接下来会举例介绍预翻译文件的代码段格式。

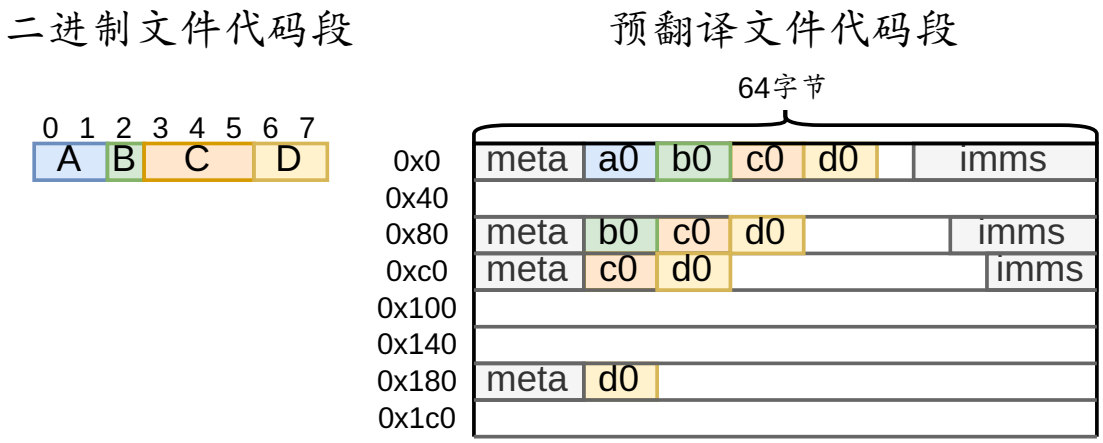

图 4-1　预翻译文件格式

Figure 4-1　The format of the Ahead-of-Time file

注: A、B、C、D **为客户指令**，a0、b0、c0、d0 **为翻译出的融合微码指令，由于重复存储，会存储** a0-b0-c0-d0、b0-c0-d0、c0-d0、d0 **这四行融合微码指令，并填充对齐。**

如图4-1，展示了预翻译文件的代码段格式，为了简单起见，假设图中的指令缓存行大小为 8 字节，包含 4 条客户指令 A、B、C、D，起始地址分别为 0x0、0x2、0x3、0x6，生成的融合微码指令为 a0、b0、c0、d0。对应的预翻译文件会有 8 行，每行长度为 64 字节（相对于 ELF 文件膨胀了 64 倍）。第 0 行（地址 0x0）存储 A 指令开始的所有融合微码指令 a0、b0、c0、d0; 第 2 行（地址 0x80）存储 B 指令开始的所有融合微码指令 b0、c0、d0; 第 3 行（地址 0xC0）存储 C 指令开始的所有融合微码指令 c0、d0; 第 6 行（地址 0x180）存储 D 指令开始的所有融合微码指令 d0。其余的第 1、4、5、7 行都是空行，用于填充对齐。能看出来，ELF 文件中代码地址和预翻译文件中代码地址是线性映射关系，只需要简单的地址映射就能找到对应的融合微码指令，地址关系如下：

$$Addr_{AOT} = Base + Addr_{ELF} \times 64 \tag{4-1}$$

再回顾一下，为何需要**重复存储**融合微码指令呢? 这是因为可能有指令跳转到一行的中间位置（例如跳转到指令 C），这样就需要从预翻译文件中加载 C 指令开始的融合微码指令。如果只存储 A 指令开始的融合微码指令，那么在跳转到 C 指令时，就需要重新翻译 C 指令开始的融合微码指令，这样会增加额外的开销，这个开销包括调用动态二进制翻译器进行实时翻译、翻译缓存的填充等，需要数百拍才能完成，因此为了减少这个开销，需要重复存储融合微码指令。

重复存储办法本质上是一种用空间换时间的策略，通过增加预翻译文件的大小，减少了动态二进制翻译的开销，提高了性能。虽然文件代码段膨胀了 64 倍，但目前主流的 SPEC 2017 程序的代码段大小在几十 MB，这样的膨胀对于现代存储设备来说并不是很大的开销。而对于内存来说，预翻译文件中有大量的空行，可以通过压缩算法进行透明压缩，减少内存占用，Linux 中的 zswap 技术就是这样的一种技术。对于多级缓存来说，只有被取到的缓存行才会被加载到缓存中，未被取到的缓存行不会被加载到缓存中，例如图4-1中可能只有第一行被取到，其他行不会被加载到缓存中。

4.2 翻译缓存优化

本文设计中，翻译缓存是和指令缓存同级的（或者说，是用于**替换**指令缓存的），用于存放融合微码指令。如果翻译缓存大小和指令缓存大小相同，总行数相同，那么一行中存放的指令数量越多，缓存的总指令数量就越多，缓存缺失率就会越低，性能就会越好，因此翻译缓存的存储效率对性能影响较大，存储效率的定义如公式4-2所示。

$$\text{存储效率} = \frac{\text{实际存放的指令数量}}{\text{缓存行能存储的最大指令数量}} \tag{4-2}$$

如前文2.5.4所述，翻译缓存的每行组织形式和微码缓存类似，都是前面部分存放微码指令，后面存放立即数，微码指令和立即数相向生长，中间可能有**空**

洞。此外对于 64 字节长度的翻译缓存行，本文还设计了一个 16 字节长的元信息部分，用于存储一些额外的信息，如指令类型、指令长度等。以定长的 4 字节指令举例，一行指令缓存可以存放 16 条指令，而一行翻译缓存最多存放 12 条指令，存储效率只有 75%。更严峻的是，由于有 3 个结束条件的限制，实际存放的指令数量可能更少，存储效率更低，根据实验，平均一行翻译缓存只能存放 5 条指令，存储效率只有 31.25%。

首先回顾下空洞产生的原因，是由于三个结束条件的限制，导致翻译缓存行中的微码指令提前结束，不能填满整个缓存行。相对于2.4小节中提到的微码缓存的 3 个结束条件，翻译缓存结束条件基本相同：1. 遇到指令缓存行的结尾；2. 遇到控制流指令；3. 翻译缓存行已满。

翻译缓存是从微码缓存设计演化而来的，因此翻译缓存的空洞问题也是从微码缓存继承而来的。[28] 对传统微码缓存的空洞问题进行了分析，放松了前两个结束条件，能够提升微码缓存的存储效率，进而提升了 12% 的整体性能。本文也借鉴了这个思路，对 RISC-V 微译器中翻译缓存进行了优化，放松了前两个结束条件，提升了翻译缓存的存储效率。

4.2.1　行尾放松

前文中提到，遇到指令缓存行的结尾是一个结束条件，这样才能保证指令缓存行和翻译缓存行是一对多的关系。当遇到自修改代码等情况时，需要刷新一行指令缓存行，这样才能方便找到对应的翻译缓存行进行刷新和替换。然而，可以适当放松这个结束条件，允许连续的两行指令缓存行的指令填充到同一行翻译缓存行中。

如图4-2，展示了放松指令缓存行结尾的翻译缓存行组织形式。相对于4-1中的形式，能允许下一行指令缓存行的指令填充到同一组翻译缓存行中，例如指令 E 和指令 F 可以填充到上一个指令缓存行对应的翻译缓存行中。假如有指令跳转到 C 指令开始的基本块，就能直接取出 c0-d0-e0-f0 这四条微码指令（原本由于 D 指令为行结尾的结束条件，这一行只能存 c0-d0 指令），这样就能减少这一行的空洞，提升了存储效率。

而对于自修改代码的处理，由于一行微码缓存行中的指令最多对应两行连续的指令缓存行，例如 c0-d0-e0-f0 这四条微码指令对应的指令缓存行为 A-B-C-D 和 E-F，所以只需要刷新对应的两行指令缓存行即可。自修改代码的处理在实际场景中并不常见，并且对性能影响较小，因此放松这个结束条件是可行的。

4.2.2　分支放松

前文中提到，控制流指令可能改变程序原本指令执行流，将顺序执行的指令分割为一个个基本块，因此控制流指令是一个结束条件，这样能够保证翻译缓存行中的微码指令是一个基本块的连续指令。控制流指令分为条件跳转和无条件跳转，对于无条件跳转，一定会跳转到另一个基本块，作为结束条件是合理的；

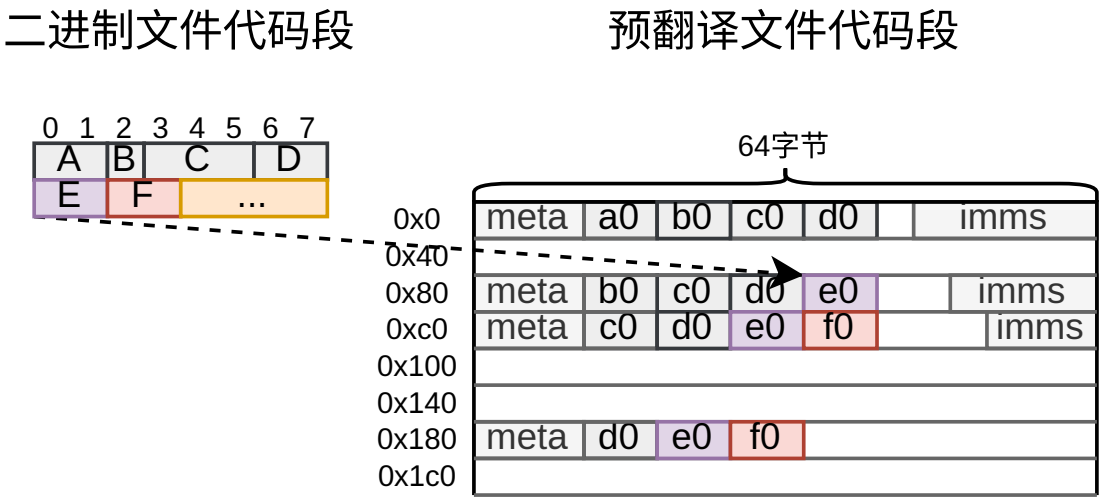

图 4-2　行尾放松

Figure 4-2　Relaxing the end of the instruction cache line

注：允许连续的两行指令缓存行的指令填充到同一组翻译缓存行中，下一行的指令 e0 和指令 f0 可以填充到上一个指令缓存行对应的翻译缓存行中。

但是对于条件跳转，在翻译过程中并不确定这条指令是否会跳转，如果不跳转，那么这条指令后面的指令也是连续的，属于同一个基本块的，可以填充到同一行翻译缓存行中，因此可以适当放松条件跳转指令的结束条件。（对于传统的微码缓存，预测为跳转的控制流指令才是一个结束条件，这是由于硬件译码可以快速判断是否跳转，但是对于 RISC-V 微译器，软件预翻译过程的“译码”并不能判断是否跳转，因此需要放松这个结束条件。）

对于所有的条件跳转指令，都可以放松这个结束条件，将这些指令和后续指令都可以填充到同一行翻译缓存行中。如图4-3，展示了放松条件跳转指令的翻译缓存行组织形式。对于条件跳转 beq 指令，可以将后续的指令也填充到同一行翻译缓存行中。虽然在运行时，beq 指令可能会跳转，后续的指令不会执行，但是对于不跳转的情况，后续的指令会执行，这样就能减少这一行的空洞，提升了存储效率。

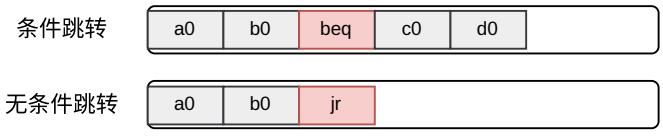

图 4-3　放松条件跳转

Figure 4-3　Relaxing the conditional jump

注：翻译缓存行忽略了元信息和立即数。对于条件跳转 beq 指令，后续还能放微码；对于无条件跳转 jr 指令，后续不能放微码。

4.3　可变长行优化

前文中提到过，每一个翻译缓存行中所有指令对应于一个基本块，根据本文插装分析的结果，每个基本块平均长度为 5 条指令。然而，由于翻译缓存行的大小是固定的，为 64 字节，这意味着即便放松了结束条件，一行中有效的指令大约只有 5 条，占据 20 字节，存储效率平均只有 35% 左右。

为此，本文提出了可变长翻译缓存行的优化方案，即根据基本块的长度动态调整翻译缓存行的大小，从而提升存储效率。如图4-4，展示了可变长翻译缓存行的组织形式，本文实现了两种长度的翻译缓存行，分别为 32 字节和 64 字节，根据基本块的长度动态选择合适的翻译缓存行，对于长度小于等于 6 的基本块，选择 32 字节的翻译缓存行，对于长度大于 6 的基本块，选择 64 字节的翻译缓存行。同时在翻译缓存行的元信息中增加了一个字段，用于存储翻译缓存行的长度，这样就能在加载翻译缓存行时，根据这个字段动态选择合适的翻译缓存行并解析其中的指令。

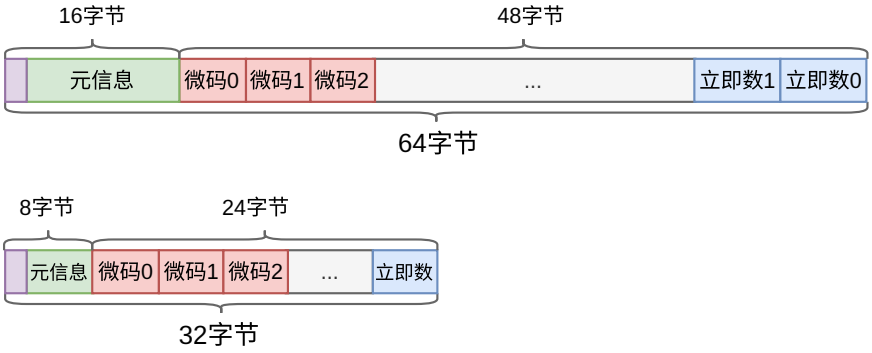

图 4-4　可变长翻译缓存行组织形式

Figure 4-4　The organization form of the variable-length translation cache line

注: 有 32 字节和 64 字节两种长度，根据基本块的长度动态选择合适的翻译缓存行。

如果基本块平均长度为 5 条指令，均使用 32 字节的翻译缓存行，存储效率为 $5 * 4/32 = 62.5\%$，比固定长度的 64 字节翻译缓存行提升了接近一倍的存储效率，这样就能减少缓存缺失率，提升性能。

4.4　指令压缩优化

前文中提到，借鉴 RISC-V 压缩指令集的思想，融合微码指令集也可以进行压缩编码，从而在一行翻译缓存中存放更多的指令，提升存储效率。回顾图3-3，展示了融合微码可以有 2 字节和 4 字节两种长度，分别为压缩指令和标准指令。

由于 x86 微码和 RISC-V 微码共享压缩指令集，因此需要设计一套通用的压缩编码方案，能够同时满足 x86 和 RISC-V 的需求。融合压缩指令的编码空间有限，总共可以支持 31 条 2 操作数、31 条 1 操作数和 32 条 0 操作数的指令，因

表 4-1 x86 和 RISC-V 在 SPEC2017 中前 15 个常用指令

Table 4-1 The top 15 most common instructions in SPEC2017 for x86 and RISC-V

	x86 指令	比例	RISC-V 指令	比例
1	mov_R8_M8	5.64%	c.add_R_R	5.49%
2	movsd_R16_M8	4.28%	fld_R_M	4.64%
3	mulsd_R16_R16	4.22%	c.addi_R_IMM	4.53%
4	addsd_R16_R16	3.66%	fmul.d_R_R_R	3.87%
5	jne_IMM8	3.17%	add_R_R_R	3.61%
6	mov_R8_R8	3.04%	c.ldsp_R_IMM_R	3.27%
7	je_IMM8	2.97%	c.mv_R_R	3.22%
8	add_R8_IMM8	2.81%	c.fld_R_IMM_R	3.13%
9	movapd_R16_R16	2.69%	c.sdsp_R_IMM_R	2.94%
10	mov_R4_M4	2.59%	fmadd.d_R_R_R_R	2.80%
11	movupd_R16_M16	2.17%	fsub.d_R_R_R	2.65%
12	cmp_R8_R8	2.01%	c.ld_R_IMM_R	2.56%
13	mov_R4_R4	1.91%	ld_R_M	2.52%
14	lea_R8_M8	1.83%	bne_R_R_IMM	2.25%
15	subsd_R16_R16	1.67%	fadd.d_R_R_R	2.22%

此需要对指令集进行精心设计，找出最常用的指令，进行压缩编码。本文通过 QEMU 插装工具分析了 SPEC 2017 中 x86 和 RISC-V 指令集的常用指令，发现这些指令占据了大部分的指令数，如表4-1所示。根据 x86 和 RISC-V 指令集的常用指令，本文设计了一套融合指令集的压缩编码。

表4-2中列出了本文设计的压缩指令集，包括 2 操作数、1 操作数和 0 操作数的指令，其中包括了常用的 x86 和 RISC-V 指令所对应的压缩融合微码，如 add、mov、cmp、syscall 等。能从表中发现，目前 2 操作数的压缩指令已经占满了 31 条，1 操作数和 0 操作数的压缩指令还有空余，可以继续添加新的压缩指令。说明压缩指令对 2 操作数指令压力比较大，后续可以优化编码方式，用以支持更多的 2 操作数压缩指令。

表 4-2 压缩指令列表

序号	压缩指令名		
	2 操作数	1 操作数	0 操作数
1	addflags_sz8	dec_sz4	invalid
2	addflagsimm_sz8	dec_sz8	halt
3	addimm_sz8	inc_sz4	ldpop
4	addwimm_sz8	inc_sz8	macroop_movs1
5	addw_sz8	ld_stack_subssz_sz9	macroop_movs2
6	andflags_sz8	rdip	macroop_movs4
7	andflagsimm_sz8	st_stack_subssz_sz9	macroop_movs8
8	auipcimm	wripcalli	macroop_stos1
9	jalrimm	wripcalli_call	macroop_stos2
10	jalrimm_T0	wripi	macroop_stos4
11	jalrimm_ret	wripi_call	macroop_stos8
12	limm_sz4	wripi_ret	mfence
13	limm_sz8		nop
14	mov_sz4		syscall
15	mov_sz8		stcall
16	orflags_sz8		
17	sraflagsimm_sz8		
18	srlflagsimm_sz8		
19	sllflagsimm_sz8		
20	stpp_sz1		
21	stpp_sz2		
22	stpp_sz4		
23	stpp_sz8		
24	subflags_sz8		
25	subw_sz8		
26	subflagsimm_sz8		
27	wripcallflagsi		
28	jalimm		
29	jalimm_call		
30	xorflags_sz4		
31	xorflags_sz8		

4.5 本章小结

本章介绍了 RISC-V 微译器的优化方案，RISC-V 微译器在引入翻译缓存后会产生额外的硬件性能开销。本章首先分析这部分性能开销的来源，发现主要来源于重复存储机制和翻译缓存的存储效率问题，前者会导致磁盘、内存的开销增加，可以通过透明压缩方式解决；后者会导致缓存缺失率增加，性能下降。针对存储效率问题，本章提出了几种优化方案，包括行尾放松、分支放松、可变长行、指令压缩，这些优化方案提升了翻译缓存的存储效率，降低了 RISC-V 微译器的性能开销，提升了整体性能，具体实验数据将在下一章实验章节中展示。

第 5 章　实验数据分析

本章主要介绍 RISC-V 微译器的实验数据分析，包括实验环境、测试程序、调试环境、性能分析等。首先介绍 RISC-V 微译器的实验环境，在添加指令翻译过程的调试环境，然后展示 RISC-V 微译器在运行 SPEC2000 的性能分析，最后分别分析添加 4 个优化方案后对整体性能影响的效果。

5.1　实验环境

本课题在 Gem5 模拟器[29] 上进行了实验，Gem5 是一个广泛使用的开源计算机系统体系结构模拟器，它具有如下特点：

- 开源：Gem5 是一个开源项目，每年都会有大量的开发者参与到 Gem5 的开发中并发布新的版本。过去 12 年间，有超过 250 名开发者参与到 Gem5 的开发中，提交了超过 7500 次的 commit[30]。目前最新的版本是 v23，也是本文使用的版本。
- 广泛使用：Gem5 被广泛用于学术界和工业界，用于研究新的计算机体系结构，新的内存层次结构，新的缓存替换算法等。
- 周期精确：Gem5 是一个周期精确的模拟器，可以模拟 CPU 的每一个周期，论文[31] 表明 Gem5 与实际硬件的性能差距在 5% 以内。
- 多架构支持：Gem5 支持多种指令集架构，包括 x86、RISC-V、ARM 等，可以方便的进行不同架构的模拟，便于本课题的研究。
- 模块化：Gem5 的设计是模块化的，分为 CPU 模块、缓存模块、内存模块等，可以方便的进行模块的替换和扩展。
- 高度可配置：Gem5 在一次编译后，在动态运行时可以通过配置参数进行不同的配置，例如 CPU 类型、缓存大小等。
- 事件驱动：Gem5 是一个事件驱动的模拟器，它将时间的流逝模拟为一系列离散事件，只有在事件发生时才会进行模拟。例如在模拟 CPU 时，只有在取指、译码、执行等事件发生时才会进行慢速模拟，其余时间 Gem5 会快速跳过，这样可以提高模拟器的效率。

Gem5 有全系统模拟器和用户空间模拟器两种模式，全系统模拟器可以模拟整个计算机系统，包括 CPU、内存、外设等，用户空间模拟器只模拟用户空间的部分，不模拟内核和硬件，只模拟用户空间的指令执行[29]。本课题使用**用户空间**模拟器进行实验，因为本课题主要关注用户态二进制翻译器优化，只翻译用户态的指令，不涉及内核态的指令。

为了和真实的硬件环境更加接近，本文使用了和真实硬件校准过的 RISC-V 处理器参数[32] 进行模拟，本文将这个参数作为**基准参数**，然后在这个基准参数上进行 RISC-V 微译器的实验。接下来的性能对比都是相对于这个基准参数进行

的。

为了不引入额外的面积、功耗等开销，本文使用同样大小的翻译缓存**替换**了原有的指令缓存，这样可以保证硬件开销不变。表5-1展示了 Gem5 的硬件参数。

表 5-1 Gem5 硬件参数，RISC-V 微译器模式仅替换了前端的指令缓存，其余参数保持不变。

Table 5-1 Gem5 hardware parameters

	RISC-V 处理器模式	**RISC-V 微译器模式**
处理器核	硬件译码 + RISC-V 指令	软件翻译 + 融合微码
	类型：乱序 6 发射处理器	
	译码：每拍 6 条	
	主频：3.4GHz	
	分发：每拍 6 条微码	
	发射队列：69 条微码	
	分支预测：4096 项的锦标赛算法	
内存层次	指令缓存: 32KB， 8 路组相连，2 拍延迟	**翻译缓存**: 32KB， 8 路组相连，2 拍延迟
	数据缓存: 32KB，8 路组相连，2 拍延迟	
	二级缓存: 256KB，8 路组相连，12 拍延迟	
	三级缓存: 8MB，16 路组相连，36 拍延迟	
	内存: 4GB，DDR3-1600, 30ns 延迟	

5.2 测试程序

本课题使用了 SPEC CPU 2000 测试集作为测试程序。SPEC CPU 2000 测试集是一个通用的测试 CPU 性能的测试集，包括了 12 个整数测试程序和 14 个浮点测试程序，是一个较为完备的测试集，本机运行时间在几分钟量级。

由于 Gem5 模拟器性能开销较大，在使用用户态模拟、乱序处理器模型的配置下，相对于本机运行时间，Gem5 平均慢 1 万倍，这意味着一个在本机上运行 1 秒的程序，在 Gem5 上需要运行 3 小时。因此选择 SPEC 2000 test 测试集来缩短实验时间，所有测试程序在 Gem5 模拟器上运行的时间均控制在几分钟到几小时之间，这样方便本文进行实验。

测试程序使用 riscv64-linux-gnu-gcc 交叉编译器编译到 RISC-V 架构上，gcc 版本为 12.3，编译参数均为-O3 –static 静态编译，SPEC 各子项程序还有其默认的编译参数。

5.3 调试环境

为了保证整个 RISC-V 微译器的正确性，需要好的测试集和调试环境。

对于测试集，本文使用了单元测试、集成测试、性能测试等多种测试手段，保证了翻译器的正确性。单元测试用于测试软件端翻译器的正确性，集成测试用于测试 RISC-V 微译器在运行小型测试程序时的正确性，性能测试用于测试 RISC-V 微译器在运行大型测试程序时的性能。

- 单元测试: 选择了 riscv-tests[33] 和 riscv-arch-tests[34]，两者均使用汇编语言编写，用于测试单条 RISC-V 指令能否正确翻译运行。前者对 RISC-V IMAFD 每条指令均有 10 余个测试用例，后者能生成数百个整数指令的随机测试用例。
- 集成测试: 使用了手写的 C 语言程序，用于测试 RISC-V 微译器在运行小型测试程序时的正确性，例如能否正确的初始化栈信息、能否正确处理系统调用等。
- 性能测试: 使用了 SPEC 2000 test 测试集，用于测试 RISC-V 微译器在运行大型测试程序时的性能。

如果测试程序出现了错误，可以通过调试环境来定位错误。由于 Gem5 模拟的软硬件系统上无法直接运行 GDB 调试器，本课题通过打印指令执行信息、打印寄存器信息等方式来定位错误。对于含有错误的 RISC-V 微译器，需要和标准正确的 Gem5 模拟器进行对比，找出错误的原因。标准正确的 Gem5 模拟器是指没有任何修改的、配置为 RISC-V 架构的 Gem5 模拟器，它能直接硬件译码运行 RISC-V 程序。通过在 Gem5 模拟器上添加了打印所有整数、浮点寄存器信息的功能，当指令执行流比对不一致时，可以通过打印寄存器信息来定位错误，调试过程参考图5-1。

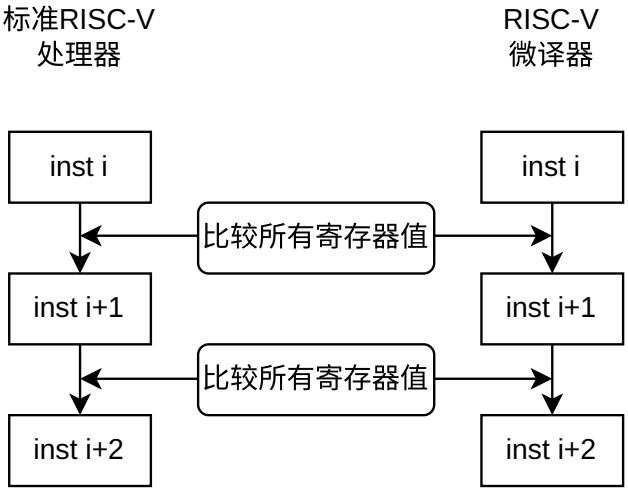

图 5-1　RISC-V 微译器逐指令调试过程

Figure 5-1　The process of debugging RISC-V transutor

5.4　性能分析

由于 Gem5 的模拟器的性能开销较大，本文选择了 SPEC CPU 2000 test 测试集作为测试基准。Gem5 模拟器在运行测试集时，参考表5-1总共有三个测试配

置，分别为：

(1) RISC-V 处理器模式（基准参数）：CPU 没有修改，使用指令缓存，通过内存层次、多级缓存和指令缓存，硬件译码并运行 RISC-V 程序。

(2) RISC-V 微译器模式：使用翻译缓存，软件端二进制翻译器把 RISC-V 指令翻译为融合微码，CPU 通过内存层次、多级缓存和微码缓存，硬件运行融合微码。

(3) 优化版 RISC-V 微译器模式：在上一个配置基础上，开启了所有优化选项，让一行微码行中存储微码指令数目变多了，减少翻译缓存缺失率。

本文以每拍指令数目（IPC，Instructions Per Cycle）作为性能指标，IPC 越大，性能越好。以 RISC-V 处理器模式为基准性能（原生性能），RISC-V 微译器模式的性能**归一化**为基准性能的百分比，并分别测量关闭优化以及开启所有优化的性能。

如图5-2所示，RISC-V 微译器在运行 SPEC2000 整数测试集时，性能表现如下：在不开启任何优化的情况下，性能为原生性能 87.8%，其中主要是 176.gcc 和 186.crafty 两个测试程序性能下降较多，这是由于这两个测试程序代码循环较少，时间局部性较差，指令代码较多，导致了大量的缓存缺失。而对于其他的几个测试程序，性能大多在 90% 以上，说明这些程序的时间局部性较好，性能下降较少，RISC-V 微译器的性能表现较好。

在开启所有优化后，性能为原生性能的 94%，性能提升了 6%，说明优化方案的效果较好。对于 176.gcc 和 186.crafty 两个测试程序，性能提升较多，分别为 20% 和 24%。下一小节将对优化方案进行分析。

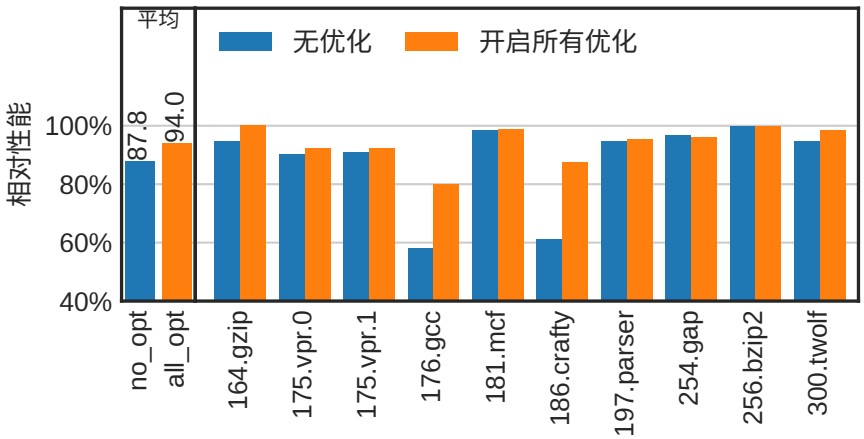

图 5-2 SPEC2000 整数性能对比图

Figure 5-2 The performance comparison of SPEC2000 integer

如图5-3所示，RISC-V 微译器在运行 SPEC2000 整数测试集时，翻译缓存缺失率表现如下：未开启优化时，缓存缺失率平均为 2.3%，其中最高的为 gcc 和 crafty 测试程序，缓存缺失率为 7.7% 和 9.2%；这与性能表现相符，缓存缺失率越高，性能越低。开启所有优化后，缓存缺失率平均为 0.6%，下降了 1.7%，说

明优化方案能有效减少缓存缺失率，提升性能。经过计算，缓存缺失率和性能之间的皮尔森相关系数为-0.91，说明缓存缺失率和性能之间存在很强的负相关性。目前的性能下降主要受到了未命中次数的影响。

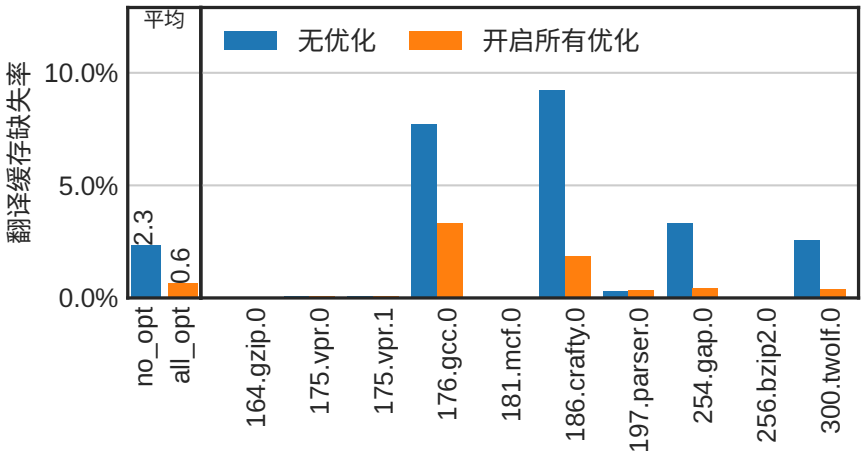

图 5-3　SPEC2000 整数测试下缓存缺失率

Figure 5-3　The cache miss rate of SPEC2000 integer

如图5-4所示，RISC-V 微译器在运行 SPEC2000 浮点测试集时，性能表现如下：在不开启任何优化的情况下，性能表现在加上微码缓存后，性能为原生性能的 96.5%；在开启所有优化后，性能为原生性能 98.2%，性能提升了 1.7%。这说明无论是否开启优化，RISC-V 微译器对于浮点性能表现较好。

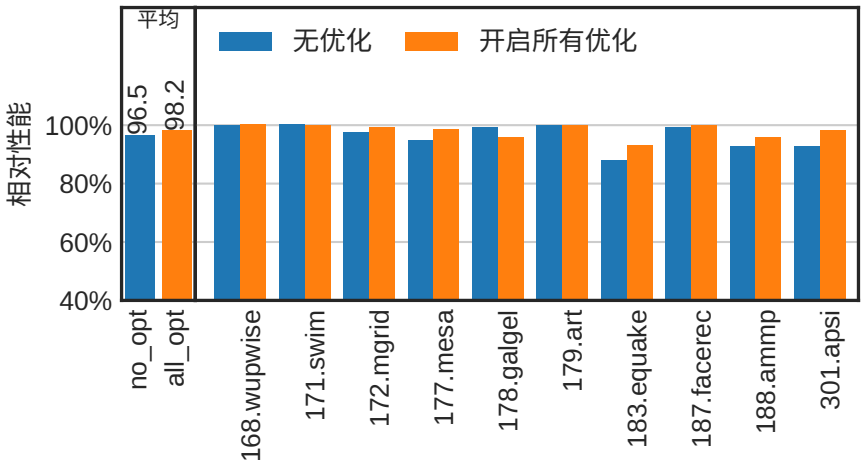

图 5-4　SPEC2000 浮点性能对比图

Figure 5-4　The performance comparison of SPEC2000 floating point

5.5　优化方案分析

获取到缓存缺失是导致性能下降的主要原因，但缓存缺失也有三种可能，分别为强制缺失、容量缺失和冲突缺失（缓存的 3C 定律）。本文需要进一步分析

缓存缺失的类型，以便进一步优化。首先通过关闭处理器中缓存预取算法、增加程序运行时间等，发现缓存缺失率没有明显变化，说明不是强制缺失。接下来修改缓存的组相连度（4 路修改为 8 路、16 路），发现缓存缺失率也没有明显变化，说明不是冲突缺失。最后通过修改缓存大小，发现缓存缺失率有明显变化，说明是容量缺失。

简单修改缓存大小虽然能有效提升性能，但是会导致额外的面积和功耗开销，因此本文需要进一步优化缓存缺失率，在不改变缓存大小的情况下，通过提升翻译缓存行中的指令数量，提升存储效率，进一步减少缓存缺失率，提升性能，也就是第4章中提出的四种优化方案。

本节将对第4章中提出的四种优化方案进行分析，分别是行尾放松、分支放松、可变长行、指令压缩。由于 Spec2000 浮点测试在关闭和打开优化条件情况下，性能都接近原生性能，差异不大，因此这里不再分析。后文将分析 Spec2000 **整数**测试集的优化效果。为了防止不同优化条件相互影响，本文分别对每个优化方案单独进行分析，也就是单独开启其中一个优化方案，其余优化方案关闭。

5.5.1 行尾放松

回顾第4章提出的优化方案 1——行尾放松，也就是允许第二行的指令填充到上一行的翻译缓存行中，这样相当于能提升上一行的翻译缓存行中的指令数量，等价于提升翻译缓存的有效容量，减少缓存缺失率，提升性能。

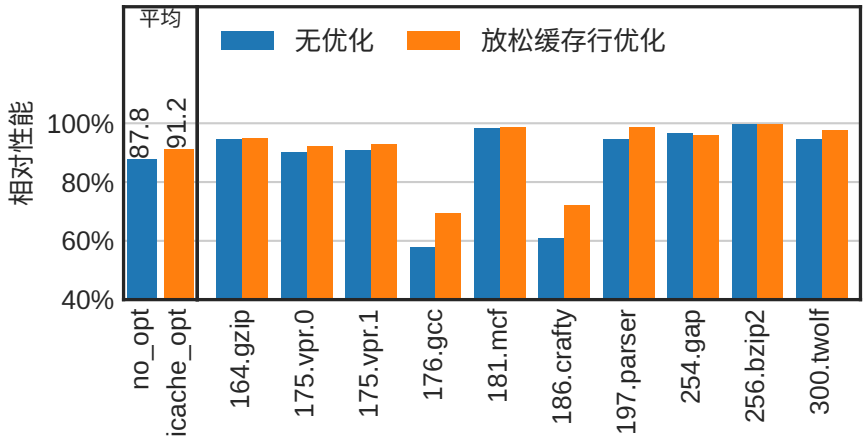

图 5-5 行尾放松后性能对比图

Figure 5-5 The performance comparison of relaxing instruction cache line

如图5-5所示，RISC-V 微译器在运行 SPEC2000 整数测试集时，开启放松指令缓存行优化后，平均性能从 87.8% 提升到了 91.2%，性能提升了 3.4%。

如图5-6所示，开启放松指令缓存行优化后，翻译缓存行平均指令数量从 4 条提升到了 4.5，提升了 0.5 条指令。这说明放松指令缓存行优化方案有效提升了翻译缓存行的指令数量，减少了缓存缺失率，提升了性能。但同样需要注意到，相对于 64 字节长的翻译缓存行，一行极限能存储 12 条指令，目前平均只存

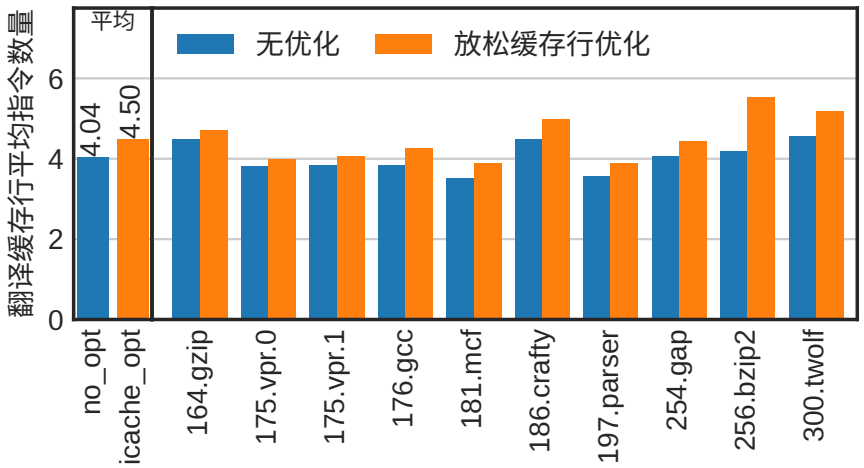

图 5-6　行尾放松后翻译缓存行平均指令数量

Figure 5-6　The average number of instructions in translation cache line after relaxing instruction cache line

储了 4.5 条指令，还有较大的提升空间。

5.5.2　分支放松

回顾第4章提出的优化方案 2——分支放松，也就是允许条件跳转指令后的指令仍然能放到同一行的翻译缓存行中，这样对于不跳转的条件跳转指令，后续的指令仍然在同一行中，能继续执行。

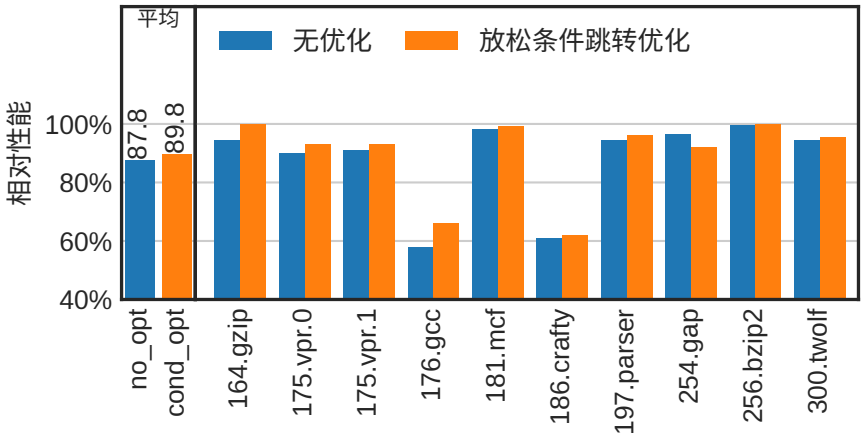

图 5-7　分支放松后性能对比图

Figure 5-7　The performance comparison of relaxing conditional jump optimization

如图5-7所示，RISC-V 微译器在运行 SPEC2000 整数测试集时，开启放松条件跳转优化后，平均性能从 87.8% 提升到了 89.8%，性能提升了 2%。

如图5-8所示，开启放松条件跳转优化后，翻译缓存行平均指令数量从 4 条提升到了 6.8 条，提升了 2.8 条指令。这说明放松条件跳转优化方案有效提升了翻译缓存行的指令数量，减少了缓存缺失率，提升了性能。但平均指令数的提升

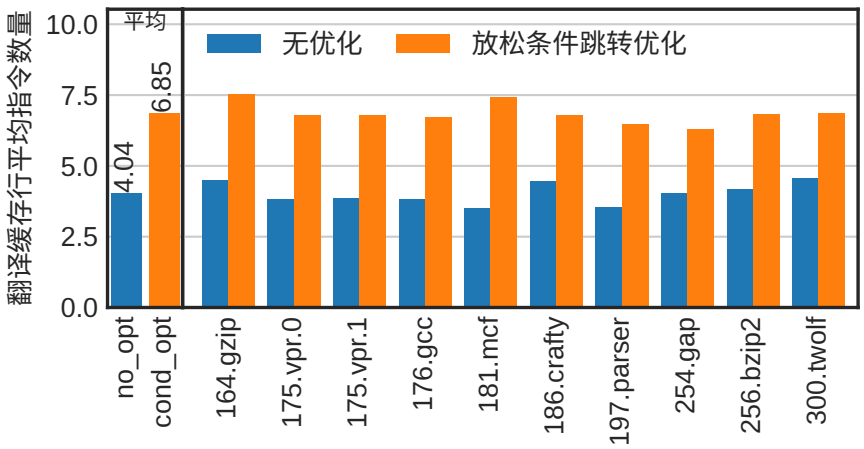

图 5-8　分支放松后翻译缓存行平均指令数量

Figure 5-8　The average number of instructions in translation cache line after relaxing conditional jump

大于性能的提升，说明条件跳转后的指令虽然放在了一行中，相当于分支预测静态预测一定不跳转，但是如果实际跳转了，后续指令没有执行，也同样浪费了翻译缓存行的容量。

5.5.3　可变长行

回顾第4章提出的优化方案 3——可变长行，也就是允许翻译缓存行的长度不固定，根据翻译缓存行的指令数量动态调整长度。当指令数多时候使用 64 字节的翻译缓存行，当指令数少时候使用 32 字节的翻译缓存行。

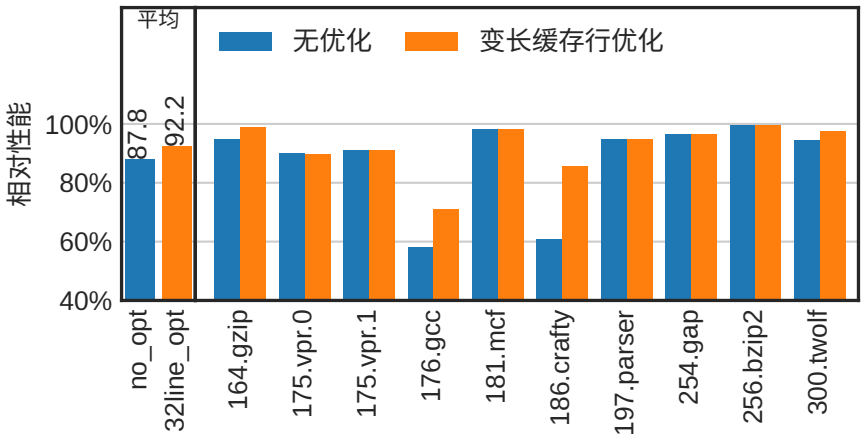

图 5-9　可变长行优化后性能对比图

Figure 5-9　The performance comparison of using variable length microcode cache line optimization

如图5-9所示，RISC-V 微译器在运行 SPEC2000 整数测试集时，开启变长微码缓存行优化后，平均性能从 87.8% 提升到了 92.2%，性能提升了 4.4%。这也是所有优化中效果最好的一个优化方案。根据分析发现缓存行平均指令数为 4.0

条，而 32 字节的翻译缓存行最大能存储 6 条标准融合指令，存储 4 条指令是足够的，大部分情况下不需要使用 64 字节的翻译缓存行。更短的翻译缓存行长度，意味着更高的存储效率，减少了缓存缺失率，提升了性能。

5.5.4　指令压缩

回顾第4章提出的优化方案 4——指令压缩，也就是将融合微码指令压缩为更短的指令存储，减少翻译缓存行的占用。

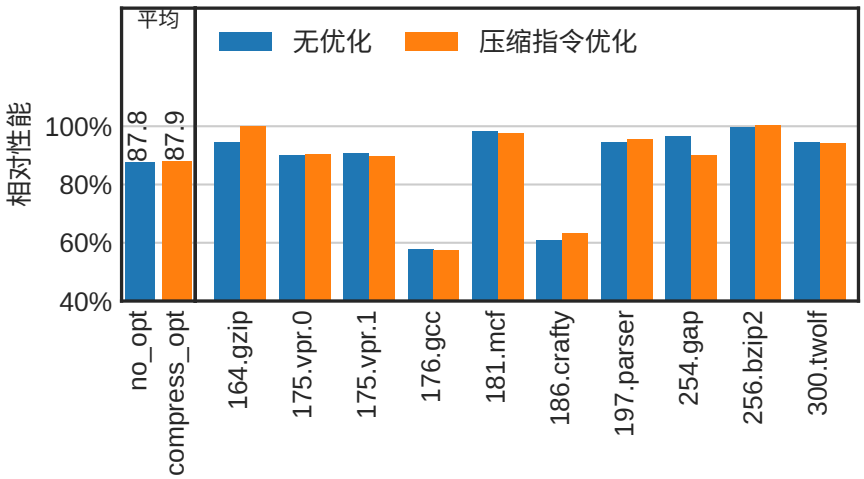

图 5-10　指令压缩后性能对比图

Figure 5-10　The performance comparison of adding compression instruction optimization

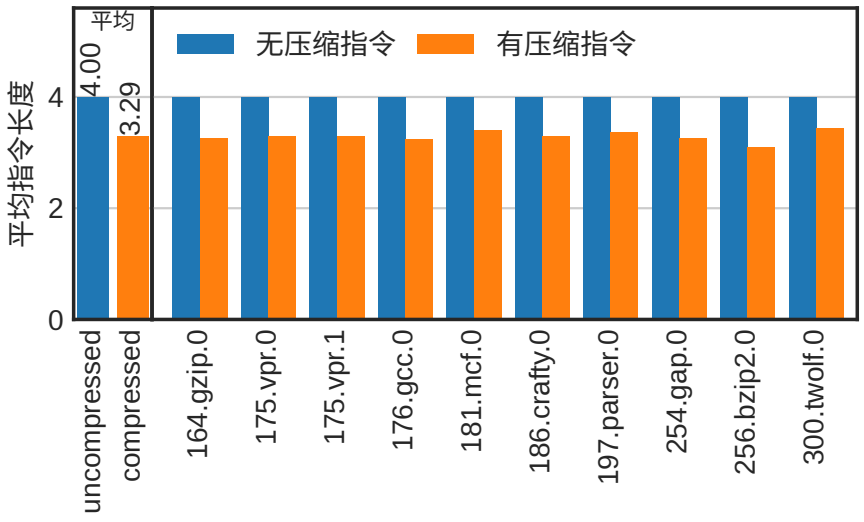

图 5-11　指令压缩后平均指令长度变化

Figure 5-11　The average instruction length change after adding compression instruction optimization

如图5-10所示，RISC-V 微译器在运行 SPEC2000 整数测试集时，开启压缩指令优化后，平均性能没有什么变化，性能为 87.9%。

如图5-11所示，开启压缩指令优化后，平均指令长度从 4 字节减少到了 3.3 字节，减少了 0.7 字节。这说明压缩指令优化方案有效减少了融合指令平均指令长度，减少了翻译缓存行的占用，但是由于翻译缓存行结束条件的限制，导致了翻译缓存行的指令数量没有增加，性能没有提升。3.3 字节的平均指令长度，也接近 RISC-V C 扩展压缩指令的平均长度（3 字节），说明大部分 RISC-V 压缩指令都被翻译为融合微码的压缩指令，考虑到融合微码压缩指令同时支持 x86 指令的翻译，这个长度是可以接受的。

5.6 本章小结

本章首先介绍了 RISC-V 微译器的实验环境，也就是使用 Gem5 模拟器的配置方案; 接下来介绍了测试程序，包括 SPEC CPU 2000 test 测试集; 进一步介绍了调试环境，包括单元测试、集成测试、性能测试等; 接下来对 RISC-V 微译器在 RISC-V 上运行的性能进行了分析，包括性能对比、缓存缺失率分析等，实验结果表明 RISC-V 微译器在运行 SPEC2000 整数测试集时，性能为原生性能的 87.8%，开启所有优化后性能提升到了 94%；在运行 SPEC2000 浮点测试集时，性能为原生性能的 96.5%，开启所有优化后性能提升到了 98.2%。整数和浮点测试集在开启所有优化后 SPEC2000 平均性能为原生性能的 96.1%。

最后对本文提出的 4 个优化方案进行了分析，分别是行尾放松、分支放松、可变长行、指令压缩，性能提升分别为 3.4%、2.0%、4.4%、0.1%。得出结论，行尾放松和分支放松对性能提升效果较好，可变长行优化对性能提升效果最好，指令压缩优化对性能提升效果不显著（但有效降低了平均指令长度）。

第 6 章　总结与展望

6.1　总结

本文发现国产处理器因采用不同指令集而引发的软件适配难题和生态碎片化问题，这阻碍了国产处理器的发展。二进制翻译技术能够缓解这些问题，但目前高性能二进制翻译器的性能瓶颈仍然存在，只能达到原生性能的 80%，且软件优化方案已经比较成熟，难以进一步提升性能，亟需软硬件协同的优化。

本文成功在一种新的二进制翻译系统——x86 微译器中添加了对 RISC-V 架构的支持。此系统的主要目标是在单一硬件平台上实现多指令集的共存，并尽可能地接近原生执行效率。本研究的主要工作和贡献包括：

1. 成功设计并实现了 RISC-V 微译器，包括寄存器映射方案及支持 272 条通用 RISC-V 指令的 51 条微码指令。

2. 对 RISC-V 微译器进行了性能优化，通过行尾放松、分支放松、可变长行、指令压缩等四种优化策略，降低了微码缓存的缺失率，提升了性能。

在 Gem5 模拟器中实施的原型系统的实验结果表明，经过优化的 RISC-V 微译器在执行 SPEC 2000 测试集时的平均性能达到了原生程序的 96.1%，有效地缓解了性能瓶颈，并实现了接近原生程序的运行效率。微译器的同时支持了 x86 和 RISC-V 两种指令集，为多架构二进制翻译提供了一种创新的解决方案。

6.2　未来展望

展望未来，有几个潜在的研究方向可能会进一步提升本文提出的 RISC-V 微译器的功能和性能：

(1) 丰富测试程序：当前的测试主要依赖于 SPEC 2000 测试集，未来可以考虑增加更多的真实测试程序，如数据库、编译器、图像处理程序等，以更全面地评估微译器的性能和适用性。

(2) 扩展指令集支持：探索将 ARM、LoongArch 等其他流行指令集整合进微译器，以增强其对各类国产处理器的支持，进一步提升系统的通用性和适用性。

(3) 微架构优化：进一步研究和优化微译器的内部微架构，如改进翻译缓存的组织形式。通过增加更多的优化策略，如更高效的指令压缩编码、更精简的缓存行元数据信息等，进一步提升缓存存储效率。还可以通过添加翻译缓存行的预取机制，进一步降低缓存缺失率，提升性能。

(4) 硬件实现验证：将研究成果从模拟环境转移到实际硬件上，以验证系统在真实硬件下的性能表现以及面积和功耗等方面的指标。

(5) 融合微码设计：目前的融合微码还是基于 Gem5 x86 微码的扩展，但融合微码的实现会极大影响客户指令的翻译膨胀率，未来可以考虑设计一种更加

通用的融合微码，需要全面考虑多种指令集的特征，兼顾指令集编码空间、硬件实现复杂度等多方面因素。

(6) 系统态支持：当前的微译器主要支持用户态的二进制翻译，融合微码也只考虑各个指令集的用户态指令，未来可以考虑增加对系统态指令的支持，包括如何高效支持各指令集的控制寄存器、系统调用、内存管理等，以更好地支持操作系统和关键应用程序的翻译。

参考文献

[1] Hennessy J L, Patterson D A. A new golden age for computer architecture [J]. 2019, 62(2): 48–60.

[2] 胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术 [J]. 计算机研究与发展, 2023, 60(1): 2-16.

[3] Anton Chernoff, Mark Herdeg, Ray Hookway, et al. FX!32 a profile-directed binary translator [J]. IEEE/ACM International Symposium on Microarchitecture, 1998, 18(2).

[4] James C. Dehnert, Brian K. Grant, John P. Banning, et al. The Transmeta Code Morphing™ Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges [J]. International Symposium on Code Generation and Optimization, 2003: 15-24.

[5] Apple Inc. Running Intel Binaries in Linux VMs with Rosetta [J/OL]. Apple Developer Documentation, 2023[2023-11-23]. https://developer.apple.com/documentation/virtualization/running_intel_binaries_in_linux_vms_with_rosetta.

[6] Intel core processors and intel bridge technology unleash windows 11 [EB/OL]. [2024-04-16]. https://www.intel.com/content/www/us/en/newsroom/news/intel-tech-unleashes-windows-experience.html.

[7] 谢汶兵, 田雪, 漆锋滨, 等. 二进制翻译技术综述 [J]. 软件学报, 2024, 35(6): 1.

[8] Wang J, Gao X, Li G, et al. Godson-3: A scalable multicore risc processor with x86 emulation [J]. IEEE/ACM International Symposium on Microarchitecture, 2009, 29(02): 17-29.

[9] 胡伟武, 靳国杰, 汪文祥, 等. 龙芯指令系统融合技术 [J]. 中国科学：信息科学, 2015(4): 459-479.

[10] Bellard F. QEMU, a Fast and Portable Dynamic Translator [C]//USENIX Annual Technical Conference. 2005: 46.

[11] Apple Inc. About the Rosetta Translation Environment [J/OL]. Apple Developer Documentation, 2023[2023-11-23]. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment.

[12] 华为技术有限公司. 华为动态二进制翻译工具（ExaGear） [EB/OL]. 2023[2023-11-23]. https://www.hikunpeng.com/developer/devkit/exagear.

[13] 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, 48(4): 43.

[14] Standard Performance Evaluation Corporation. SPEC CPU® 2017 [EB/OL]. 2017[2023-11-28]. https://www.spec.org/cpu2017/.

[15] Vosen. Zluda [EB/OL]. 2020[2024-4-9]. https://github.com/vosen/ZLUDA.

[16] Xie B, Yan Y, Yan C, et al. An instruction inflation analyzing framework for dynamic binary translators [J]. ACM Transactions on Architecture and Code Optimization, 2024.

[17] Nan L, Jianmin P. Intermediate code optimization method for binary translation based on intermediate representation rule replacement [J]. Journal of National University of Defense Technology, 2021, 43(4): 156-162.

[18] Hong D Y, Hsu C C, Yew P C, et al. Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores [C]//IEEE/ACM International Symposium on Code Generation and Optimization. 2012.

[19] Yue F, Pang J, Han X, et al. An improved code cache management scheme from i386 to alpha in dynamic binary translation [C]//Proceedings of the 2010 Second International Conference on Computer Modeling and Simulation. 2010: 321-324.

[20] Wang W. Helper function inlining in dynamic binary translation [C]//Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction. 2021: 107-118.

[21] Gouicem R, Sprokholt D, Ruehl J, et al. Risotto: A dynamic binary translator for weak memory model architectures [C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. 2022: 107–122.

[22] Yandong L. Principle and application of dynamic binary translation technology [EB/OL]. 2021 [2024-04-15]. https://ppt.infoq.cn/slide/show?cid=83&pid=3238.

[23] Amstadt B, Johnson M K. Wine [J]. Linux Journal, 1994(4es): 3.

[24] 胡起. 指令流分析指导的动态二进制翻译优化技术 [D]. 中国科学院计算技术研究所, 2023.

[25] d'Antras A, Gorgovan C, Garside J, et al. Optimizing Indirect Branches in Dynamic Binary Translators [J]. ACM Transactions on Architecture and Code Optimization, 2016, 13(1): 1-25.

[26] Baruch Solomon, Avi Mendelson, Doron Orenstien, et al. Micro-operation cache: A power aware frontend for variable instruction length ISA [J]. International Symposium on Low Power Electronics and Design, 2001: 4-9.

[27] David N. Suggs T U, Austin. Operation cache: US 10606599 [P]. 2020.

[28] Jagadish B. Kotra, John Kalamatianos. Improving the Utilization of Micro-operation Caches in x86 Processors [J]. IEEE/ACM International Symposium on Microarchitecture, 2020: 160-172.

[29] Binkert N, Beckmann B, Black G, et al. The gem5 simulator [J]. SIGARCH Computer Architecture News, 2011, 39(2): 1–7.

[30] Lowe-Power J, Ahmad A M, Akram A, et al. The gem5 Simulator: Version 20.0+ [J]. ArXiv, 2020.

[31] Butko A, Garibotti R, Ost L, et al. Accuracy evaluation of gem5 simulator system [J]. 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip, 2012: 1-7.

[32] Akram A, Sawalha L. Validation of the gem5 Simulator for x86 Architectures [J]. IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, 2019: 53-58.

[33] riscv-software-src. riscv-tests [EB/OL]. 2013[2024-4-9]. https://github.com/riscv-software-src/riscv-tests.

[34] riscv-none-isa. riscv-arch-tests [EB/OL]. 2018[2024-4-9]. https://github.com/riscv-non-isa/riscv-arch-test.

附录 一　RISC-V 指令翻译表

下表展示了 RISC-V 指令翻译到微码指令的对应关系，并按照 RISC-V 指令类型进行了分类。其中微码指令分为**已有微码**和**新加微码**两种，已有微码指令是原有的 Gem5 x86 微码指令，新加微码指令是新增的类似 RISC-V 的微码指令。我们希望新加微码尽可能少来减少编码空间的占用，因此我们尽量将 RISC-V 指令翻译成已有微码指令。

能发现大部分以 w 结尾的 RISC-V 指令会翻译成新加微码指令，这是由于 w 结尾的指令为 RISC-V64 相对于 RISC-V32 新加的操作 32 位整数的指令，而 Gem5 x86 微码指令中没有对应的 32 位整数操作指令，因此我们需要新加微码指令来实现这些指令。

表 A-1 RISC-V 指令翻译表

分类	RISC-V 指令	翻译成已有微码	翻译成新加微码
移位	sll	sllflags	
	slli	sllflagsimm	
	sllw		sllw
	slliw		sllwimm
	srl	srlflags	
	srli	srlflagsimm	
	srlw		srlw
	srliw		srlwimm
	sra	sraflags	
	srai	sraflagsimm	
	sraw		sraw
	sraiw		srawimm
算数	add	addflags	
	addi	addflagsimm	
	addw		addw
	addiw		addwimm
	sub	subflags	
	subw		subw
	lui	limm	
	auipc		auipcimm
逻辑	xor	xorflags	
	xori	xorflagsimm	

表 A-1 续表

分类	RISC-V 指令	翻译成已有微码	翻译成新加微码
	or	orflags	
	ori	orflagsimm	
	and	andflags	
	andi	andflagsimm	
比较	slt		slt
	slti		sltimm
	sltu		sltu
	sltiu		sltuimm
分支	beq		
	bne		condjumpimm
	blt		
	bge		
	bltu		condjumpuimm
	bgeu		
访存	lb		lb
	lh		lbu
	lbu		lh
	lhu		lhu
	lw		lw
	lwu		lwu
	ld	ld	
	sb	st	
	sh	st	
	sw	st	
	sd	st	
其他	jal		jalimm/call
	jalr		jalrimm/ret
	fence	mfence	
	fence.i	mfence	
	ecall	syscall	
	ebreak	syscall	
乘除法指令	mul		mul
	mulw		mulw
	mulh		mulh
	mulhsu		mulhsu
	mulhu		mulhu

表 A-1 续表

分类	RISC-V 指令	翻译成已有微码	翻译成新加微码
	div		div
	divu		divu
	divw		divw
	divuw		divuw
	rem		rem
	remu		remu
	remw		remw
	remuw		remuw
浮点移动	fmv.w.x		mov2fpword
	fmv.x.w		mov2intword
	fmv.d.x	mov2fp	
	fmv.x.d	mov2int	
	fcvt.s.w		
	fcvt.d.w		
	fcvt.s.wu		
	fcvt.d.wu		cvtgi2f
	fcvt.s.l		
	fcvt.d.l		
	fcvt.s.lu		
浮点转换	fcvt.d.lu		
	fcvt.w.s		
	fcvt.w.d		
	fcvt.wu.s		
	fcvt.wu.d		cvtf2gi
	fcvt.l.s		
	fcvt.l.d		
	fcvt.lu.s		
	fcvt.lu.d		
浮点访存	flw		lwfp
	fld	ldfp	
	fsw	stfp	
	fsd	stfp	
浮点运算	fadd.s/d	maddf	
	fsub.s/d	msubf	
	fmul.s/d	mmulf	
	fdiv.s/d	mdivf	

表 A-1 续表

分类	RISC-V 指令	翻译成已有微码	翻译成新加微码
	fsqrt.s/d		msqrtf
浮点乘加	fmadd.s/d	mmaddf	
	fmsub.s/d	mmsubf	
	fnmadd.s/d		mnmaddf
	fnmsub.s/d		mnmsubf
浮点符号	fsgnj.s/d		fsgnj
	fsgnjn.s/d		fsgnjn
	fsgnjx.s/d		fsgnjx
浮点比较	fmin.s/d	mminf	
	fmax.s/d	mmaxf	
	feq.s/d		mcmpf.eq
	flt.s/d		mcmpf.lt
	fle.s/d		mcmpf.le
	fclass.s/d		fclass
原子指令	lr.w	lw	
	sc.w	st	
	lr.d	ld	
	sc.d	st	
	amoswap.w/d		
	amoadd.w/d		
	amoxor.w/d		
	amoand.w/d		
	amoor.w/d		翻译为多条指令
	amomin.w/d		
	amomax.w/d		
	amominu.w/d		
	amomaxu.w/d		

致 谢

在论文即将完成之际，回顾研究生三年的生涯，内心许多感慨，在此向所有关心、支持和帮助过我的人表示衷心的感谢！

我要感谢党和国家，我出生农村，家里经济条件受限，在初中阶段一直到本科毕业，一直都获得过国家的贫困生资助，让我能够顺利完成学业。感谢党和国家对基层教育和公平性的重视，让我有机会从农村，走到省会城市成都，再到国家首都北京，接受更好的教育。

我要感谢我的母校中国科学院大学，我在此完成了本科和硕士的学业。本科依托于中国科学院的优质教育资源，我在此奠定了扎实的数学、物理、计算机基础，本科的操作系统课程、组成原理课程和体系结构课程都让我对计算机系统有了更深的理解，促使我选择了计算机系统结构作为硕士的研究方向。硕士期间在中国科学院计算技术研究所和龙芯中科技术有限公司联合成立的龙芯实验室下，我有幸参与到二进制翻译系统的分析与优化工作，我在此获得了很多的知识和经验，也认识了许多优秀的同学和老师。

我要向我的硕士生导师王剑老师表示最诚挚的感谢，王老师在我硕士期间给予了我很多的指导和帮助，并在我研究方向的选择上给予了很多的建议，让我能够顺利完成硕士学业。此外我还想感谢课题组的张福新老师，他作为龙芯实验室主任和二进制翻译领域的专家，在我的研究工作中给予了我很多的建议和指导，让我受益匪浅。

我要向实验室师兄谢本壹博士表示感谢，谢博士在带领我入门二进制翻译领域的过程中给予了我很多的帮助，和谢博士共同完成的两个课题，也让我在代码调试、实验分析制图和论文写作等方面都得到了很多的锻炼。我还要感谢实验室的其他师兄师姐们、同学们、学弟学妹们，特别是李欣宇师兄、燕澄皓同学、张壮壮同学、刘天义师兄、张婷婷师姐、陶思成师弟、吴翔师弟等，我们共同完成的课题让我学到了很多，也感受到了团队合作的重要性。

我还要感谢我的女朋友袁嘉怡，感谢她在我本科和研究生期间对我的支持和鼓励，让我多次走出困境，重新振作，她的陪伴让我感到很幸福！她也让我在学业和生活上都能够做到更好的平衡！

最后我要感谢我的父母，是他们的无私付出和默默陪伴，让我能走到今天，他们是我永远的依靠和榜样！

忠心感谢所有关心、支持和帮助过我的人，谢谢你们！

2024 年 6 月

作者简历及攻读学位期间发表的学术论文与其他相关学术成果

作者简历:

晏悦，四川成都人，1998 年出生。

2017 年 09 月——2021 年 06 月，在中国科学院大学计算机科学与技术学院获得学士学位。

2021 年 09 月——2024 年 06 月，在中国科学院计算技术研究所攻读硕士学位。

已发表（或正式接受）的学术论文:

(1) Xie, Benyi, Xinyu Li, **Yue Yan**, Chenghao Yan, Tianyi Liu, Tingting Zhang, Chao Yang and Fuxin Zhang. "On-Demand Triggered Memory Management Unit in Dynamic Binary Translator." Advanced Parallel Programming Technologies (2023).

(2) Xie, Benyi, **Yue Yan**, Chenghao Yan, Sicheng Tao, Zhuangzhuang Zhang, Xinyu Li, Yanzhi Lan, Xiang Wu, Tianyi Liu, Tingting Zhang and Fuxin Zhang. "An Instruction Inflation Analyzing Framework for Dynamic Binary Translators." ACM Transactions on Architecture and Code Optimization (2024).

参加的研究项目:

(1) 二进制翻译膨胀分析工作 合作参与者
(2) x86 微译器项目 合作参与者
(3) RISC-V 微译器项目 负责人

获奖情况:

(1) 2023 年 中国科学院计算技术研究所 所长优秀奖
(2) 2023 年 中国科学院大学 三好学生
(3) 2023 年 龙芯中科技术有限公司 龙芯实验室优秀学生