计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
2025,34(6):158−167 [doi: 10.15888/j.cnki.csa.009874] [CSTR: 32024.14.csa.009874]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

动态二进制指令集翻译的机器学习优化方法①

王耀华[1], 张真瑜[2], 蔡雨晴[2], 戴鸿君[2]

[1](统信软件技术有限公司, 北京 102600)
[2](山东大学 软件学院, 济南 250101)
通信作者: 戴鸿君, E-mail: dahogn@sdu.edu.cn

摘　要: 动态二进制翻译在跨平台移植和兼容性方面具有重要应用, 但现有方法面临翻译效率和资源开销的挑战. 本文提出一种基于机器学习的动态二进制指令集翻译方法. 该方法的创新点在于将机器学习二进制分析、神经机器翻译技术融合为动态二进制翻译系统, 构建了一个高效精准的指令集翻译框架. 方法包括: 利用预训练模型捕获指令级语义信息、训练 Transformer 架构的神经翻译模型来实现指令映射, 最终通过形式化验证, 将机器学习方法生成映射与动态二进制翻译系统进行集成. SPEC CPU 2006 基准测试结果表明, 该方法在翻译规则覆盖率和运行效率方面优于现有方法, 为动态二进制翻译系统优化提供了新思路.

关键词: 动态二进制翻译; 机器学习; 指令集翻译; 神经机器翻译; 形式化验证; 翻译规则

引用格式: 王耀华,张真瑜,蔡雨晴,戴鸿君.动态二进制指令集翻译的机器学习优化方法.计算机系统应用,2025,34(6):158–167. http://www.c-s-a.org.cn/1003-3254/9874.html

Dynamic Binary Instruction Set Translation Optimization Method Based on Machine Learning

WANG Yao-Hua[1], ZHANG Zhen-Yu[2], CAI Yu-Qing[2], DAI Hong-Jun[2]

[1](UnionTech Software Technology Co. Ltd., Beijing 102600, China)
[2](School of Software, Shandong University, Jinan 250101, China)

Abstract: Dynamic binary translation plays a crucial role in cross-platform portability and compatibility. However, existing methods face challenges in translation efficiency and resource overhead. A machine learning-based dynamic binary instruction set translation method is proposed. The proposed method innovatively integrates machine learning-based binary analysis and neural machine translation into a dynamic binary translation system, constructing an efficient and accurate instruction set translation framework. The method includes leveraging pre-trained models to capture instruction-level semantic information, training a Transformer-based neural translation model for instruction mapping, and integrating machine learning-based mappings with dynamic binary translation systems through formal verification. Experimental results from SPEC CPU 2006 benchmark tests demonstrate that the proposed method achieves superior translation rule coverage and runtime efficiency compared to existing approaches, offering new insights for optimizing dynamic binary translation systems.

Key words: dynamic binary translation; machine learning; instruction set translation; neural machine translation; formal verification; translation rule

随着移动计算和异构计算的快速发展, 跨平台软件迁移和兼容性问题日益凸显. 特别是在移动设备和云计算领域, 将传统 X86 平台的应用程序迁移到 ARM 架构设备上的需求越来越迫切. 这种迁移需求不仅来

① 基金项目: 北京市科技计划 (Z231100005923009)
收稿时间: 2024-10-22; 修改时间: 2024-11-12; 采用时间: 2025-01-07; csa 在线出版时间: 2025-04-30
CNKI 网络首发时间: 2025-05-06

自移动应用开发者, 也来自企业级应用的云端部署需求[1,2]. 动态二进制翻译作为一种有效的解决方案, 能够在不同指令集架构之间实现程序的动态转换和执行, 具有重要理论研究价值和实际应用意义[3–5].

近年来, 学术界和工业界在基于机器学习的二进制分析领域开展了广泛的研究. 一些研究者致力于将二进制代码通过模型进行相似度分析, 以支持程序分析、安全评估和逆向工程等任务[6–8]. 然而, 这些工作主要关注结果的静态分析, 而未将其应用于实际的程序执行和运行中.

为了实现二进制翻译在实际环境中的应用, 明尼苏达大学的研究团队提出了一种自动化生成二进制翻译规则并集成到模拟器中的方法[9]. 该方法通过静态分析和动态分析相结合的方式, 自动提取源二进制和目标二进制之间的翻译规则, 并将其应用于模拟器的动态翻译过程中. 然而该方法仍然存在一些局限性: 首先, 无法实现对 64 位程序的支持, 影响了方法的适用性; 其次, 该方法依赖于程序源代码, 难以对没有源代码的程序进行特定优化. 此外, 该方法未充分利用深度学习技术, 限制了翻译规则的泛化能力和适应性.

针对现有二进制翻译方法的局限性, 提出以下研究问题: 1) 如何利用神经机器翻译技术, 构建高质量的 X86 到 ARM 翻译数据集, 实现高效精准的二进制指令集翻译, 使得在实际应用中能够直接处理无源代码的二进制程序; 2) 如何将机器学习方法与传统动态二进制翻译系统有机结合, 通过翻译规则及优化策略, 在保证翻译正确性的同时提升系统执行效率.

为回答这些问题, 本文提出一种机器学习与动态二进制翻译系统集成的框架, 如图 1 所示. 图 1 主要包括 3 个模块: 预训练模块、翻译模型模块和 DBT 集成模块. 预训练模块通过对开源软件的 X86 和 ARM 二进制文件进行分析, 反汇编后提取控制流和数据流信息组成预训练数据集, 最终在预训练模型中将汇编代码块转化为向量嵌入表示; 翻译模型模块利用源代码级调试信息构建有监督数据集, 结合预训练模型的向量嵌入信息, 学习 X86 到 ARM 汇编的翻译表达形成翻译规则; DBT 集成模块处理经过参数化处理、形式化验证以及筛选后的翻译规则, 最终集成应用于 DBT 系统之中.

综上所述, 本文的主要研究内容和贡献如下: 1) 提出一种基于机器学习的动态二进制指令集翻译方法, 融合机器学习二进制分析、神经机器翻译技术和动态二进制翻译技术; 2) 改进面向二进制分析的预训练模型, 通过在大规模二进制数据上的无监督学习, 获得了高质量的指令级别语义表示; 3) 构建 X86 到 ARM 指令集翻译数据集, 并在此基础上训练了基于 Transformer 架构的神经机器翻译模型, 实现了源指令到目标指令的高精度映射; 4) 提出一种翻译规则自动化生成、筛选和验证方法, 通过形式化方法保证翻译的正确性, 并通过参数化方法提高规则的泛化能力; 5) 基于 X86 指令集到 ARM 指令集实现机器学习方法与动态二进制翻译系统的集成, 提升翻译效率和系统性能. 在 SPEC CPU 2006 基准测试中, 取得优于现有方法的结果.

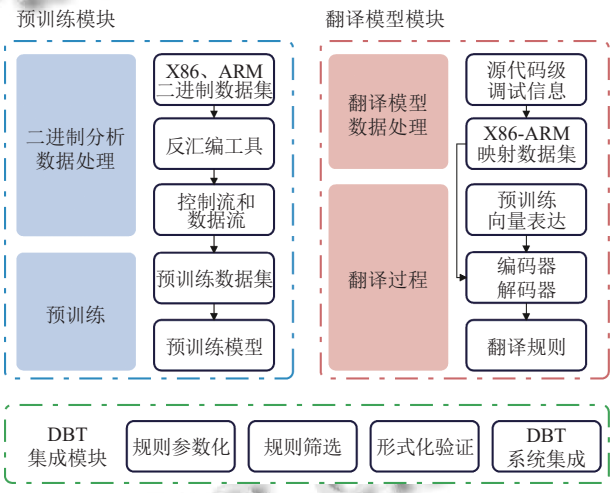

图 1 基于机器学习与动态二进制翻译系统的集成框架

1 背景及相关工作

1.1 动态二进制翻译

动态二进制翻译是一种将源指令集架构 (ISA) 的二进制代码翻译成目标 ISA 的二进制代码的技术, 常用于跨平台虚拟化、程序分析和安全评估等领域[10–12]. 传统的二进制翻译方法主要包括基于解释器的翻译和基于即时编译 (JIT) 的翻译两种类型[13].

基于解释器的翻译通过逐条解释和模拟源 ISA 指令, 实现目标 ISA 转换[14]. 这种方法实现简单, 但翻译效率较低. 基于 JIT 的翻译方法则在程序运行时动态地将源 ISA 代码块编译成目标 ISA 代码块, 并进行缓存和重用[13]. JIT 方法在翻译效率和执行性能方面优于解释器方法, 但翻译过程引入了额外的编译开销.

研究者们针对动态二进制翻译的性能瓶颈问题,

提出了多种优化方法, 如代码缓存[15]、trace 优化[16]、动态代码生成[17]等. 这些方法在一定程度上提升了翻译效率和执行性能, 但仍然面临着翻译精度、资源开销等挑战.

动态二进制翻译模拟器 FEX-Emu 将在后续作为集成系统使用, FEX-Emu 是一个允许在 ARM64 主机上执行 X86 二进制文件的模拟器. 与 Box86 和 Box64 不同的是, FEX-Emu 采取不同的模拟方法, 它使用中间表示 (IR) 而不是直接将指令翻译成目标主机的指令集架构 (ISA). 因此, 大多数操作都是在 IR 级别上执行, 而不是直接在 X86 上执行. 在性能方面, 文献[18]通过密码安全审计和 7zip 压缩算法的测试对比了 FEX-Emu、原生执行、Box64 以及 QEMU 的性能表现. 结果显示, FEX-Emu 的性能在某些算法上优于 Box64, 远优于 QEMU, 平均达到原生执行的 20. 75%.

Wine 是一个允许在 Linux 和 macOS 等 POSIX 兼容操作系统上运行 Windows 应用程序的兼容层[19]. 通过将 FEX-Emu 与 Wine 相结合, 可以在 ARM 架构的设备上运行 X86 版本的 Windows 软件, 这极大地拓宽了 ARM 设备的应用范围. FEX-Emu 能够模拟 X86 指令集, 使 Windows 程序可以在 ARM 处理器上运行, 而 Wine 则提供必要的 Windows API 实现, 使程序能够正常工作. 这种结合不仅让 ARM 设备能够运行更多应用程序, 还为后续在 ARM 设备上运行 X86 Windows 软件提供了一种可行的解决方案.

1.2 机器学习二进制分析

近年来, 机器学习技术在二进制分析领域得到了广泛关注和应用. 研究者尝试将机器学习引入恶意软件检测、漏洞发现等任务, 以期提高分析效率和精度. Yakura 等[20]提出了使用卷积神经网络学习恶意软件的二进制图像表示, 无需手工特征工程, 在多个数据集上展现出优异的检测性能. Tian 等[21]设计了一种基于神经网络的漏洞检测模型, 通过学习程序流图的结构特征, 能够自动发现二进制程序中的多种类型漏洞. Luo 等[7]提出了 VulHawk 方法, 利用分而治之策略和迁移学习来缓解跨平台二进制代码搜索面临的编译环境差异问题. 尽管这些工作取得了进展, 但它们大多还处于研究原型阶段, 离实际工业部署仍有不小差距. 一方面, 模型泛化能力有待进一步提高, 对抗样本存在使得模型的鲁棒性备受挑战. 另一方面, 高质量标注数据的匮乏限制了模型的训练和优化. 此外, 模型判决过程的“黑盒”特性也引发了可解释性和可信度方面的担忧. 这些问题在一定程度上阻碍了机器学习在二进制分析实际生产中的应用. 研究者需要在算法改进、训练数据优化、模型可解释性等方面做进一步探索, 助力机器学习二进制分析技术走向工业界.

2 方法设计与翻译规则生成

近年, 机器学习技术及深度学习技术在各个领域取得了广泛的成功和应用. 受此启发, 本文将深度学习方法引入动态二进制翻译领域, 有望显著提升翻译规则的泛化能力和适用范围.

传统的动态二进制翻译系统主要依赖于人工构建的翻译规则, 将客户机指令序列翻译成语义等价的主机指令序列. 然而, 手工设计翻译规则不仅耗时耗力, 而且很难覆盖所有可能的情况. 文献[9]提出了一种新颖的方法, 其通过机器学习方法自动从源码编译得到的客户机和主机二进制代码中学习翻译规则. 该方法利用编译器调试信息找出源代码和汇编指令的对应关系. 如图 2 所示, 利用编译器调试信息找出一行源代码分别对应的 ARM 和 X86 汇编指令序列, 并通过符号执行验证语义的等价性, 最终自动生成可以将一段客户机指令序列映射为主机指令序列的参数化规则. 在这个过程中跳过传统 DBT 流程中的中间表示 (IR), 直接生成目标代码, 有望进一步提升端到端翻译的效率. 该实验表明, 学习得到的规则可以嵌入到 QEMU 等主流 DBT 系统, 并且能够取得 1.25 倍的平均性能提升.

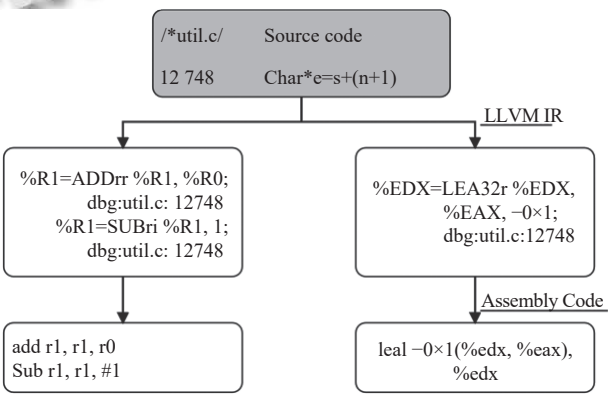

图 2 一行源代码对应的 X86 与 ARM 汇编

在此基础上, 可以进一步利用深度学习技术, 结合大规模源码-汇编对应数据, 端到端地学习源代码到不同架构汇编代码得到翻译模型. 一方面, 借助深度神经

网络强大的表示学习能力, 从海量异构数据中自动提取翻译知识, 摆脱对手工构建规则的依赖; 另一方面, 充分利用源代码蕴含的高层语义信息, 从更高的抽象层次入手学习翻译规则, 有望克服传统方法局限于特定架构和指令的问题, 大幅提高规则的泛化能力. 通过在大规模数据集上训练, 学习得到的翻译模型可以应用到各种不同领域的程序中, 显著扩大规则的适用范围.

与此同时, 端到端的规则学习也面临着不少技术挑战, 例如如何适应不同的编程语言和编译器、如何在确保准确性的同时提高学习和翻译的效率等, 这些都有待在后续研究中进一步探索. 将深度学习方法引入动态二进制翻译领域, 必将开启一个全新的研究方向, 极大地推动二进制翻译技术的发展.

2.1 预训练模块

预训练模块旨在通过大规模二进制数据的自监督学习, 获得高质量的指令级别语义表示, 为后续的翻译任务提供良好的初始化和先验知识. 本节将详细介绍预训练模块的设计, 包括预训练数据准备和预训练模型结构两个部分.

预训练模块基于文献[22]提出的 palmtre-BERT 模块, 基于 BERT 传统掩码语言模型的基础上进行额外的 Code-Word Prediction 预测, 在给定代码片段的空白位置处对最有可能出现的单词或标记进行 Definition-Use 分析, 旨在识别汇编中变量的定义 (Definition) 和使用 (Use) 情况. 原模型只针对一个指令集或一个体系结构进行学习, 未跨多个 CPU 体系结构学习语言模型. 跨体系结构意味着不同体系结构中语义相似的指令可以映射到嵌入空间中的相邻区域. 在标志化进行修改的基础上, 进行跨体系结构的学习, 通过向量空间的相似性为接下来的翻译模型提供优质的向量化表达.

为获得足够多样化和覆盖全面的预训练数据, 收集大量的 X86 和 ARM 二进制程序, 涵盖了各种类型的应用程序、库函数. 这些程序来源于不同编译器 (如 GCC、LLVM), 以确保预训练数据的广泛代表性.

在数据准备的过程中, 首先, 使用 binary ninja 的反汇编组件对二进制程序进行反汇编, 得到对应的汇编代码; 然后, 对汇编代码进行预处理, 包括指令格式化、操作数规范化、伪指令展开等, 将其转换为统一的指令序列格式; 此外, 还通过静态分析和动态插桩等方法, 提取指令序列的上下文信息, 如数据流、控制流、调用关系等, 以丰富预训练数据的语义表示.

经过上述处理, 最终得到一个大规模的指令序列数据集, 包含数亿条 X86 和 ARM 指令, 以及它们的上下文信息. 该数据集为预训练模型的训练提供了充足的数据支持.

综上, token 化的汇编指令首先在嵌入层中进行多维度的编码, 之后通过多层 Transformer 编码器提取高阶的交互特征, 最终输出 token 级别的隐藏状态, 供下游的预训练任务使用. 这种结构使得预训练模块能够充分建模汇编指令内部以及指令之间的语义信息.

2.2 翻译模型模块

翻译模型模块是研究方法的核心组成部分, 其目的是在预训练模块学习到的指令语义表示的基础上, 构建一个端到端的神经机器翻译模型, 实现从 X86 指令序列到 ARM 指令序列的自动转换. 本节将详细介绍翻译模型模块的设计, 包括翻译数据集的构建和翻译模型结构的设计.

为了训练高质量的翻译模型, 需要构建一个大规模的 X86-ARM 指令级别平行语料库. 与传统的机器翻译任务不同, 指令级别的翻译数据集需要满足以下特殊要求.

- 指令粒度: 数据集中的每个样本应由一组 X86 指令序列和 ARM 指令序列组成, 而不是自然语言中的句子或段落.
- 语义等价: 每组指令序列应在语义上等价,即它们在相同输入下应产生相同的执行结果.
- 代码多样性: 为了提高翻译模型的泛化能力,数据集应覆盖各种类型的代码, 如不同的应用程序、库函数、操作系统组件等.

为满足以上要求, 研究采用一种源代码级别对齐的方法来构建翻译数据集, 具体步骤如下.

1) 收集大量的开源项目和基准测试程序, 这些程序包含了 C/C++等多种编程语言, 覆盖各种应用领域. 从 X86 汇编的角度来说覆盖了算术、逻辑、转移、内存、SIMD、分支、浮点等指令.

2) 对每个项目或程序, 使用不同的编译器 (例如 GCC、LLVM) 和编译选项, 分别生成 X86 和 ARM 两个版本的二进制文件.

3) 对每个二进制文件, 使用 binary ninja 的反汇编组件生成对应的汇编代码.

4) 通过源代码级别的调试信息 (如 DWARF) 和符

号表信息, 将 X86 和 ARM 汇编代码中的每组指令与源代码中的语句建立映射关系.

在完成数据集构建后, 为将其应用于 FEX-Emu 的特性中, 需要进行脚本处理以满足 FEX-Emu 的运行要求. 首先, 需要定义一个寄存器映射表, 将 X86-64 的寄存器映射到对应的 ARM64 寄存器.

其次, 为适应 FEX-Emu 的特性, 需要将某些 X86 指令翻译成自定义伪指令, 如将 X86 的跳转指令 (如 jmp) 翻译成 set_jump #Label, 在 FEX-Emu 中会被解释为跳转函数. 这些自定义伪指令将在 FEX-Emu 运行时的环境中被捕获和特殊处理, 以实现正确的控制流和 pc 值管理. 通过上述寄存器映射表和指令翻译规则, 可以将构建好的 X86-ARM64 汇编指令对应数据集转换为适配 FEX-Emu 的形式. 这一预处理步骤借助脚本自动完成, 可以显著减少手工适配的工作量, 提高数据集的可用性.

本研究提出神经机器翻译模型, 用于实现 X86 到 ARM 汇编指令的自动转换. 该模型基于 Transformer 的 Seq2Seq 架构, 并在编码器和解码器部分进行改进, 以适应汇编指令翻译的特点.

在编码器部分, 利用预训练模型作为特征提取器, 将 X86 汇编指令序列映射到一个高维语义空间. 将预训练的输出作为编码器的输入, 可以为翻译模型提供丰富的语义表示, 有助于提高翻译质量. 在输入的过程中进行语义的组合, 将连续的汇编指令序列作为一个段落的整体输入到模型中, 而不是将其拆分成独立的指令, 以包含更多上下文信息.

在解码器部分, 基于 Transformer 标准解码器架构进行了针对性优化. 模型采用 6 层 Transformer decoder 堆叠, 每层配备 8 头注意力机制和 128 维隐藏层. 为增强模型对变长序列的处理能力, 设计了特殊的序列标记策略, 在输入序列首尾分别添加<sos_p>和<eos_p>标记. 这些特殊标记不仅标示序列边界, 还为模型提供了额外的位置和语义信息.

在序列生成过程中, 通过改进的 beam search 算法提升生成质量. 具体而言, 采用 beam size 为 10 的搜索策略, 并结合累积概率评分机制, 在每一步生成时动态更新和调整候选序列. 为防止生成过长序列, 设置最大长度阈值为 200, 同时通过提前终止机制处理已完成生成的序列, 提高解码效率.

针对指令翻译任务的特点, 选择了参数共享机制. 通过将编码器的词嵌入权重与解码器输出层权重绑定, 不仅减少了模型参数量, 也在一定程度上缓解了过拟合问题. 此外, 引入了基于 LogSoftmax 的概率归一化处理, 并配合带有偏置项的线性变换层, 提升了模型的数值稳定性.

为优化训练过程, 模型采用混合目标函数进行训练. 结合交叉熵损失和 BLEU 评分, 其中交叉熵损失通过 mask 机制精确计算有效位置的预测误差, 而 BLEU 评分则从更宏观的角度评估生成序列的质量. 这种多目标训练策略使模型在保持指令语义准确性的同时, 也能生成更符合目标指令集特征的序列.

通过以上设计, 构建一个高效、鲁棒的神经机器翻译模型, 用于实现 X86 到 ARM 指令序列的自动转换. 与传统的基于规则或启发式的翻译方法相比, 该模型能够自动学习复杂的指令映射关系, 降低人工编写规则的成本. 同时, 得益于预训练模块提供的高质量语义表示, 该模型在缺乏大规模平行语料的情况下, 也能够取得良好的翻译性能.

最终得到模型, 输入 X86-64 汇编得到 ARM64 汇编, 如表 1 所示, 将映射关系保存下来用于后续的动态二进制翻译集成过程.

表 1 X86-64 汇编至 ARM64 汇编映射

X86-64汇编	ARM64汇编
mov r10d dword [rax]	ldr w14 [x4]
test r10d r10d	tst x14 x14
je label	b.eq #label

3 系统集成与实现

3.1 翻译规则参数化

为提高翻译规则的泛化能力和适用性, 需要对规则中的操作数进行参数化处理. 这里主要涉及两类操作数: 寄存器和立即数.

对于寄存器, 使用符号 reg0、reg1 等来替代具体的寄存器名称. 这样, 一条规则就可以匹配多个具有相同指令序列但使用不同寄存器的 X86 代码块. 表 2 是一个示例.

表 2 规则参数化

X86-64汇编	ARM64汇编
mov reg0, qword [reg1 +imm0]	Ldr reg0, [reg1, #imm0]
test reg0, reg0	tst reg0, reg0
je $label	b.eq #label

如表 2 所示, 在这个例子中, 原始的 X86 代码使用了 rax、rbx 等具体的寄存器, 但在规则中它们被替换为 reg0、reg1 等符号. 当在 FEX-Emu 中应用该规则时, 需要根据 X86 代码块中的实际寄存器来动态地确定 reg0、reg1 所代表的具体 ARM64 寄存器.

对于立即数, 使用 imm0、imm1 等符号来表示. 与寄存器类似, 在应用规则时, 需要根据 X86 代码块中的实际值来替换这些符号. 以上面的规则为例, 当 X86 代码块中的指令是 mov rax, qword [rbx+16]时, 需要将 imm0 替换为 16 除寄存器和立即数, 还需要对标签进行特殊处理.

通过引入寄存器、立即数和标签的符号化表示, 并在应用规则时动态地确定其具体值, 可以使一条规则匹配更多的 X86 代码块, 从而提高规则的泛化能力.

3.2 重要规则筛选与形式化验证

在完成规则的参数化和初步筛选后, 需要进一步识别出在实际应用中频繁出现、对性能影响较大的重要规则, 并对其进行形式化验证, 以确保规则的正确性和可靠性.

首先, 为找出高频次的 X86 汇编指令序列, 在 FEX-Emu 运行过程中需动态统计每条规则的匹配次数. 具体来说, 在规则匹配成功时, 将对应规则的计数器加 1. 经过一段时间的运行, 可以得到一个按照匹配次数排序的规则列表. 位于列表前部的规则是最常见、最有代表性的 X86 指令序列, 它们对性能的影响也最为显著. 将这些规则标记为“重要规则”, 并将其优先纳入后续的验证过程.

接下来, 使用符号执行引擎 angr 来对重要规则进行形式化验证. 具体步骤如下所示.

1) 将规则中的 X86 和 ARM64 汇编代码分别传入 angr 的符号执行引擎.

2) 对 X86 和 ARM64 的通用寄存器、标志寄存器、内存地址等状态进行符号化, 并为其赋予符号值.

3) 执行 X86 和 ARM64 的符号执行, 获得最终状态.

4) 提取 X86 和 ARM64 最终状态中的寄存器值、内存值, 构建等价性验证公式.

5) 利用 SMT 求解器对验证公式进行求解, 判断在所有可能的符号值下, X86 和 ARM64 的最终状态是否始终等价.

通过形式化验证, 可以从数学和逻辑的角度严格证明重要规则的正确性, 排除可能存在的错误, 从而为规则的实际应用提供可靠的保证. 同时, 基于符号执行的验证方法也具有较高的自动化程度和通用性, 可以较容易地应用到新的规则和新的体系结构之上.

3.3 FEX-Emu 集成改造与系统实现

为了将机器学习得到的翻译规则集成到 FEX-Emu 中, 需要对 FEX-Emu 的代码进行一定的改造和扩展. 将机器学习得到的规则集成到 FEX-Emu 中, 需要在前端实现高效的匹配算法, 在中间层引入新翻译路径, 在后端对 ARM 代码生成器扩展.

最终系统实现的流程图如图 3 所示, 首先是预训练模块, 这一过程涉及准备两种不同编译器的 X86 及 ARM 二进制文件, 如图 3 所示. 完成这一步骤后, 对这些文件进行反编译, 以提取数据流和控制流信息, 并按照不同的架构对它们进行分类处理后送入预训练模型中进行训练. 具体来说, 将对 X86 架构和 ARM 架构的数据流及控制流进行预训练, 从而获得可应用于不同架构的预训练模型.

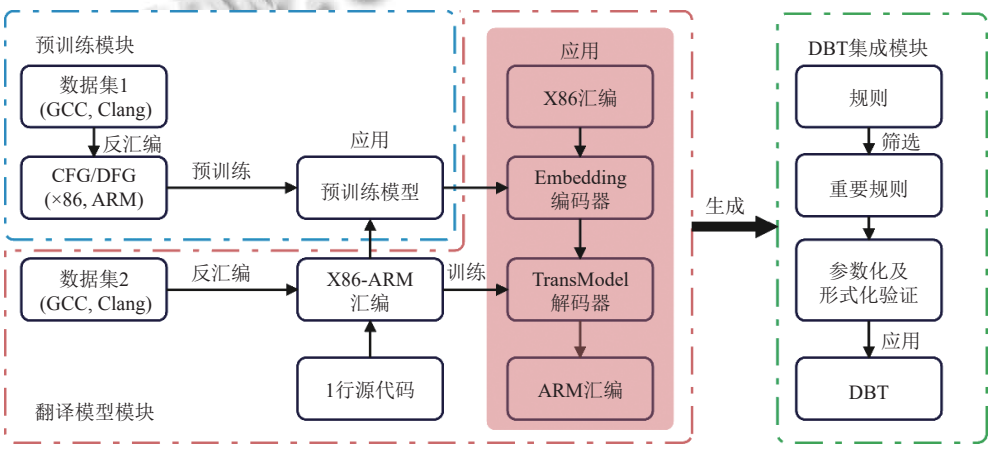

图 3 系统实现流程图

在翻译模型模块中, 将利用预训练模块中获得的预训练模型. 训练过程中, 对二进制文件进行编译优化(设置优化级别为 O1), 之后通过反汇编工具反编译这些二进制文件, 以获取源代码行号与 X86 汇编语言及 ARM 汇编语言之间的映射关系. 这一步骤是为构建翻译模型模块的训练数据集. 接下来, 将训练集输入到预训练模块中, 以获得指令嵌入信息, 并利用这些信息进行翻译模型 Decoder 的训练. 最终模型经训练后, 能够将 X86 汇编通过预训练的 Encoder 和翻译模型 Decoder 转换为对应的 ARM 汇编, 并保存为翻译规则.

最后, 在 DBT 集成模块中, 应用翻译模型模块所得的成果, 即 X86 汇编映射至 ARM 汇编的转换规则, 并将这些规则应用于参数化处理、形式化验证及 DBT 领域.

4 实验分析

4.1 实验环境与数据集

本研究使用两种不同的硬件平台进行实验. 对于深度学习模型的训练, 使用一台配备 Intel 第 10 代 i9 处理器、NVIDIA RTX 3090 显卡、128 GB 内存的高性能工作站, 操作系统为 Ubuntu 22.04.4 LTS. 这样的配置能够提供充足的计算资源, 以支持大规模神经网络模型的训练.

对于 ARM 架构的测试则是在一台搭载鲲鹏 920 处理器的服务器上进行. 该服务器安装同样的 Ubuntu 22.04.4 LTS 操作系统, 内核版本为 5.15.0-107-generic. 通过在实际的 ARM 硬件上运行程序, 可以真实地评估所生成代码的性能和能耗表现.

为构建用于训练和评估的数据集, 选取 binutils、coreutils、diffutils 和 findutils 这 4 个广泛使用的开源软件包. 使用 GCC 和 Clang 两种编译器, 分别将这些软件编译到 X86 和 ARM 两种架构上, 从而得到 4 种不同二进制文件配置: X86-GCC、X86-Clang、ARM-GCC 及 ARM-Clang. 在编译过程中, 通过添加适当的选项来生成带有调试信息的二进制文件, 这使得项目能够提取每条汇编指令与其对应源代码之间的映射关系. 为此开发一系列脚本和工具, 用于自动化地完成编译、反汇编、指令对齐等数据集构建步骤.

经过处理, 从 286 个二进制文件中提取出约 4 亿 2 529 万条匹配的 DFG 及 CFG 指令对. 这些指令对覆盖各种不同的程序功能和代码模式, 进行预训练得到向量表达.

考虑到神经网络模型对数据质量的敏感性, 对原始数据集进行进一步的清洗和过滤. 剔除长度过短或过长的指令序列, 删除出现频率极低的指令类型, 并平衡不同指令类型的数量分布.

最终, 从过滤后的数据集中随机采样 60 万条 X86-ARM 指令组对, 作为训练神经机器翻译模型的数据集. 这 60 万条数据在保留原始数据集主要特征的同时, 大小也更加适中, 能够在可接受的时间内完成模型的训练和调优.

4.2 翻译模型评估

最后在完成翻译模型的训练后, 使用多种定量和定性的方法对其性能进行全面评估.

首先, 关注模型在训练过程中的收敛情况. 通过记录并绘制每一轮迭代的损失函数值, 可以直观地判断模型学习的进度和效果. 如图 4 所示, 经过 150 000 次迭代后, 翻译模型在训练集上的损失函数值稳定在 0.006 6 左右. 这一数值的持续下降和最终收敛, 说明模型已经很好地拟合训练数据, 学习到 X86-ARM 汇编翻译的一般规律.

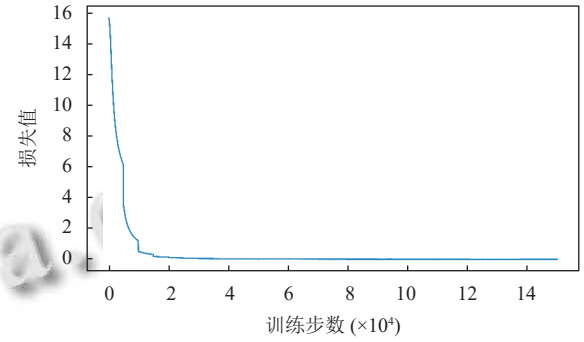

图 4 Loss 迭代图

其次, 使用 BLEU (bilingual evaluation understudy) 指标来衡量模型生成的 ARM 汇编代码与标准答案之间的相似度. BLEU 是一种广泛用于机器翻译和自然语言处理领域的评价指标, 它通过计算生成结果和参考答案之间的 *n*-gram 重合度来评估翻译的流畅性和准确性.

为客观地评测模型的泛化能力, 选取一组未参与训练的测试集, 其中包括 add2line 等常见的 Linux 工具. 在对这些测试样本进行翻译后, 得到理想的 BLEU 评分, 样本的得分都在 99 分. 这表明翻译模型能够准确地将 X86 汇编翻译为语法和语义正确的 ARM 汇编, 即使对于全新的代码也能保持很高的翻译质量.

为进一步验证生成代码的实际可用性, 将翻译后的 ARM 汇编集成到 FEX-Emu 中, 并在真实的 ARM 硬件环境下运行. 选取 SPEC2006 基准测试套件中的 400.perlbench, 并对其中的 abbrev.pl 进行翻译和测试. 结果表明, FEX-Emu 成功执行翻译后的代码, 并得到与原始 X86 版本一致的运行结果. 这充分证实翻译模型的实用价值, 它所生成的 ARM 汇编不仅在语法和语义上正确, 更能在实际的硬件环境中稳定运行, 完成预定的计算任务.

通过损失函数收敛性、BLEU 得分和实际运行结果这 3 个维度的评估, 全面地考察所构建翻译模型的性能表现. 评估结果表明, 该模型能够以极高的精度将 X86 汇编翻译为 ARM 汇编, 生成的代码在保证功能正确性的同时, 也具备很强的可读性和可维护性.

4.3 二进制翻译系统性能评估

最后指令分析为全面评估基于机器学习的二进制翻译系统的性能, 使用 SPEC CPU2006 基准测试套件进行详细测试和分析. 由于 FEX 的特性限制, 只能进行块级别的规则匹配, 因此规则的覆盖率相对于理想情况会有所降低.

本研究重点关注规则匹配对 X86 基本块的覆盖情况, 如图 5 所示, 通过分析测试结果, 发现大部分基准测试的块覆盖率都在 30%–45% 之间. 其中, 433.milc、434.zeusmp、437.leslie3d、444.namd 和 470.lbm 等测试的覆盖率都超过 40%, 说明规则库已经能够较好地适应这些程序的代码模式. 而对于 456.hmmer、458.sjeng、462.libquantum 和 473.astar 等测试, 由于 FEX 单块翻译的限制, 覆盖率低一些.

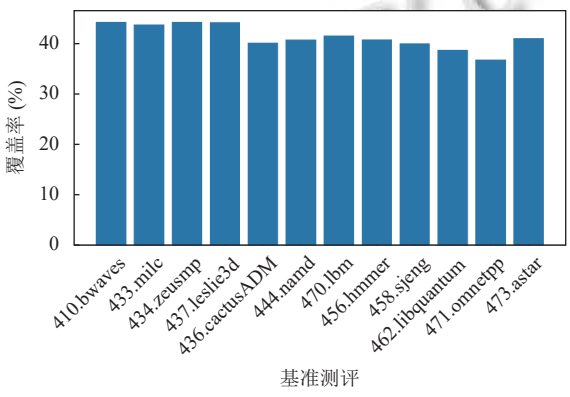

图 5 SPEC 测试覆盖率

图 6 为翻译系统的运行效率, 纵向对比可以观察到大多数基准测试都取得 1%–3% 的性能提升. 其中, 447.dealII、459.GemsFDTD、403.gcc 和 482.sphinx3 的提升最为显著, 分别达到 2.81%、5.04%、8.55% 和 5.98%. 分析这些测试代码特征, 发现性能提高主要归功于匹配到的高质量翻译规则. 这些规则能充分利用目标架构的特性, 生成更高效、简洁的 ARM 代码序列, 从而加速程序运行.

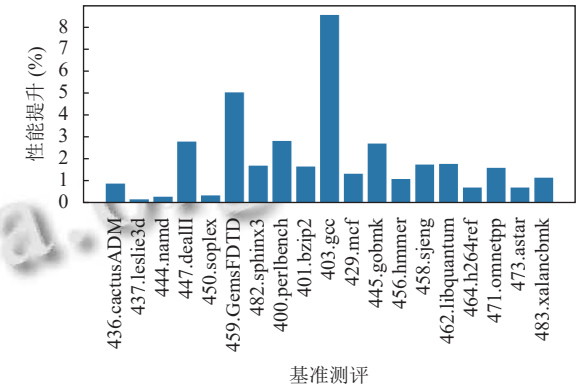

图 6 SPEC 测试性能提升

在翻译系统的横向对比中, 选择 QEMU 作为基准, 因为它被广泛用作优化基础, 可以提供更全面的性能评估. 值得注意的是, 现有的部分优化 QEMU 的研究缺乏 64 位测试数据. 如图 7 所示, 本系统在大多数测试中性能显著优于 QEMU. 特别是在 462.libquantum 和 456.hmmer 等测试中, 性能提升幅度分别达到 52% 和 6 倍. 在 429.mcf 测试中 QEMU 略有优势, 但整体上本系统的优势大幅优于 QEMU.

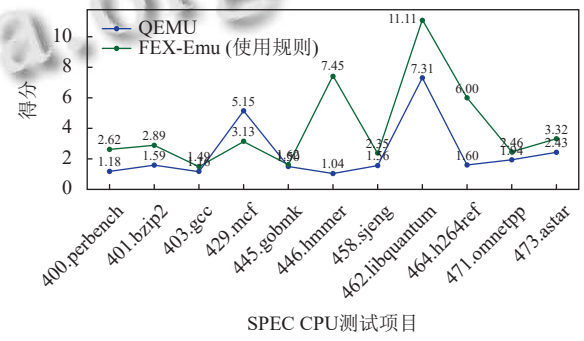

图 7 SPEC 测试性能对比

深入分析加速过程, 在指令流数目中以 434.zeusmp 为例, 当不使用翻译规则时, FEX-Emu 生成的 ARM 指令数量为 2 736 093 条. 而在应用了 250 条机器学习优化的翻译规则后, ARM 指令数降至 2 678 134 条, 减少了 57 959 条指令, 降幅约为 2.12%. 类似地如图 8 所示, 在 433.milc、454.calculix 和 483.xalancbmk 测试中, 使

用优化规则也分别减少了 33 708 条、53 522 条和 55 938 条指令, 优化效果明显.

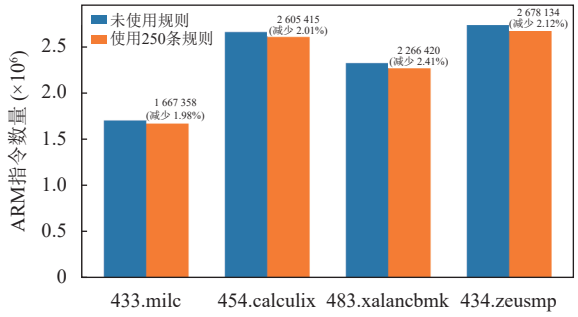

图 8 指令流数量减少

当然, 目前的系统性能仍有较大的改进空间. 一方面, 受 FEX 框架所限, 全局代码优化和寄存器分配等手段还未能充分发挥作用. 另一方面, 对于某些特定领域的程序, 如数值计算或多媒体编解码等, 专用硬件指令的使用也有待加强. 计划在后续工作中, 完善基础设施的同时, 针对重点领域开发更多高质量的翻译规则, 以期获得更大的性能提升.

5 结语

针对动态二进制翻译领域的关键挑战, 论文提出一种基于机器学习的指令集翻译方法. 该方法创新性地将机器学习技术应用于动态二进制翻译领域, 为解决动态二进制翻译中的复杂问题提供了新的思路和方法. 通过融合机器学习二进制分析、神经机器翻译技术和动态二进制翻译技术, 实现了 X86 指令集到 ARM 指令集高效精准的二进制翻译.

本研究提出的翻译框架在架构的各个模块中均具有创新性. 预训练模块在二进制分析任务中引入跨架构的分析方法, 显著提升了任务处理的广泛性和适应性. 翻译模型在确保良好效果的基础上, 能够胜任传统二进制分析任务. 在集成模块方面, 构建的翻译系统经过严谨的实验验证, 已成功与主流动态二进制翻译 (DBT) 框架实现集成, 展现了较强的实用性和鲁棒性. 整体而言, 该框架为二进制翻译领域带来了实质性的性能提升, 且未来仍有优化与扩展的空间.

尽管如此, 本研究仍存在一些不足和改进空间. 在数据方面. 后续研究可以进一步扩大数据集的规模和多样性, 纳入操作系统、驱动程序、加密算法等专业领域的代码, 从更细粒度的层面捕获指令映射的模式. 在模型方面, 未来可以探索模型压缩等技术, 在保证精度的同时提升模型的效率.

参考文献

1 Gouicem R, Sprokholt D, Ruehl J, *et al*. Risotto: A dynamic binary translator for weak memory model. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. Vancouver: ACM, 2022. 107–122. [doi: 10.1145/3567955.3567962]

2 Xing T, Barbalace A, Olivier P, *et al*. H-container: Enabling heterogeneous-ISA container migration in edge computing. ACM Transactions on Computer Systems, 2021, 39(1–4): 1–36. [doi: 10.1145/3524452]

3 Ebcioglu K, Altman E, Gschwind M, *et al*. Dynamic binary translation and optimization. IEEE Transactions on Computers, 2001, 50(6): 529–548. [doi: 10.1109/12.931892]

4 Wu J, Dong J, Fang RL, *et al*. FADATest: Fast and adaptive performance regression testing of dynamic binary translation systems. Proceedings of the 44th International Conference on Software Engineering. Pittsburgh: ACM, 2022. 896–908. [doi: 10.1145/3510003.3510169]

5 Jiang JH, Liang CY, Dong RC, *et al*. A system-level dynamic binary translator using automatically-learned translation rules. Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Edinburgh: IEEE, 2024. 423–434. [doi: 10.1109/CGO57630.2024.10444850]

6 Yu ZP, Cao R, Tang QY, *et al*. Order matters: Semantic-aware neural networks for binary code similarity detection. Proceedings of the 2020 AAAI Conference on Artificial Intelligence. New York: AAAI, 2020. 1145–1152. [doi: 10.1609/aaai.v34i01.5466]

7 Luo ZP, Wang PF, Wang BS, *et al*. VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search. Proceedings of the 2023 Network and Distributed System Security Symposium. San Diego: Internet Society, 2023. 1–18. [doi: 10.14722/ndss.2023.24415]

8 David Y, Alon U, Yahav E. Neural reverse engineering of stripped binaries using augmented control flow graphs. Proceedings of the 2020 ACM on Programming Languages, 2020, 4(OOPSLA): 225. [doi: 10.1145/3428293]

9 Wang WW, McCamant S, Zhai A, *et al*. Enhancing cross-ISA DBT through automatically learned translation rules. Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and

Operating Systems. Williamsburg: ACM, 2018. 84–97. [doi: 10.1145/3173162.3177160]

10 Bellard F. QEMU, a fast and portable dynamic translator. Proceedings of the 2005 Annual Conference on USENIX Annual Technical Conference. Anaheim: USENIX Association, 2005. 41. [doi: 10.5555/1247360.1247401]

11 Altinay A, Nash J, Kroes T, *et al*. BinRec: Dynamic binary lifting and recompilation. Proceedings of the 15th European Conference on Computer Systems. Heraklion: ACM, 2020. 36. [doi: 10.1145/3342195.3387550]

12 Fioraldi A, D'Elia DC, Querzoni L. Fuzzing binaries for memory safety errors with QASan. Proceedings of the 2020 IEEE Secure Development. Atlanta: IEEE, 2020. 23–30. [doi: 10.1109/SecDev45635.2020.00019]

13 Ung D, Cifuentes C. Machine-adaptable dynamic binary translation. ACM SIGPLAN Notices, 2000, 35(7): 41–51. [doi: 10.1145/351403.351414]

14 Souza M, Nicácio D, Araújo G. ISAMAP: Instruction mapping driven by dynamic binary translation. Proceedings of the 2010 International Symposium on Computer Architecture. Saint-Malo: Springer, 2011. 117–138. [doi: 10.1007/978-3-642-24322-6_11]

15 Wang WW, Yew PC, Zhai A, *et al*. A general persistent code caching framework for dynamic binary translation (DBT). Proceedings of the 2016 Conference on USENIX Annual Technical Conference. Denver: USENIX Association, 2016. 591–603. [doi: 10.5555/3026959.3027013]

16 Böhm I, Edler von Koch TJK, Kyle SC, *et al*. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. San Jose: ACM, 2011. 74–85. [doi: 10.1145/1993498.1993508]

17 Hawkins B, Demsky B, Bruening D, *et al*. Optimizing binary translation of dynamically generated code. Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization. San Francisco: IEEE, 2015. 68–78. [doi: 10.1109/CGO.2015.7054188]

18 Lombardi F, Recchia A. On abstract machines security and performance. Procedia Computer Science, 2024, 231: 111–118. [doi: 10.1016/j.procs.2023.12.182]

19 Duncan R, Schreuders ZC. Security implications of running windows software on a Linux system using Wine: A malware analysis study. Journal of Computer Virology and Hacking Techniques, 2019, 15(1): 39–60. [doi: 10.1007/s11416-018-0319-9]

20 Yakura H, Shinozaki S, Nishimura R, *et al*. Malware analysis of imaged binary samples by convolutional neural network with attention mechanism. Proceedings of the 8th ACM Conference on Data and Application Security and Privacy. Tempe: ACM, 2018. 127–134. [doi: 10.1145/3176258.3176335]

21 Tian JF, Xing WJ, Li Z. BVDetector: A program slice-based binary code vulnerability intelligent detection system. Information and Software Technology, 2020, 123: 106289. [doi: 10.1016/j.infsof.2020.106289]

22 Li XZX, Qu Y, Yin H. PalmTree: Learning an assembly language model for instruction embedding. Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2021. 3236–3251. [doi: 10.1145/3460120.3484587]

(校对责编: 王欣欣)