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

研究生学位论文开题报告

报告题目	龙芯平台的 ARM64 动态翻译器的优化研究		
学生姓名	吴翔	学号	2022E8013282025
指导教师	章隆兵	职称	副研究员
学位类别	工学硕士		
学科专业	计算机技术		
研究方向	二进制翻译		
培养单位	中国科学院计算技术研究所		
填表日期	2025 年 03 月		

中国科学院大学制

报告提纲

一　选题的背景及意义

1.1　选题背景

随着龙芯处理器性能的不断提升，尤其是近年来推出的 3A6000 处理器，其在主频、缓存以及多核并行处理能力方面的显著提升，已经能够满足更高性能的需求。3A6000 作为龙芯系列的重要产品，采用了自主研发的 Loongarch 架构，为其提供更强大的计算能力，具备了 4 核心、8 线程的处理器设计和最高 3.0 GHz 的工作频率，提供了更加优越的性能表现。然而，尽管硬件性能不断提升，但龙芯平台的软件生态建设仍然面临着不完善的问题。对于没有源码的应用程序和库函数的兼容上，软件工作者仍面临着巨大的开发压力。

ARM 架构，凭借其低功耗、高效能的特点，已经在移动设备、嵌入式市场取得了显著的市场份额，且在 PC 和 HPC 高性能计算领域也逐渐展现出其强劲的市场增长势头。据 IDC 的数据显示，2023 年 ARM 架构的全球市场份额已突破 50%，在移动市场中几乎占据主导地位，并且随着各种开发平台和生态系统的建设，ARM 架构在 PC 及高性能计算领域的市场份额逐年增加。对于龙芯平台来说，有效地兼容 ARM 架构的软件生态，特别是 Android + ARM 体系，在移动设备、嵌入式系统以及高性能计算领域的拓展至关重要。

二进制翻译作为一种高效的跨架构兼容方案，长期以来受到各大处理器厂商的青睐，尤其是在新的处理器架构推出时，能够有效地解决与现有软件生态之间的兼容性问题。二进制翻译通过将目标架构的指令集转换为源架构的指令集，使得不同架构之间的程序能够无缝运行，这对于提升新处理器的生态建设、加速其市场推广具有至关重要的作用。在过去的研究和开发过程中，龙芯推出了兼容 X86 架构的翻译器 LATX，成功地解决了 x86 架构下的软件兼容性问题。LATX 翻译器通过高效的二进制翻译技术，使得基于 x86 架构的软件能够在龙芯的 Loongarch 架构上流畅运行，为开发者提供了更加丰富的软件支持，极大地扩展了龙芯处理器的应用范围。

1.2　选题意义

当前，龙芯平台缺乏一款高效且完备的动态翻译器，虽然存在 QEMU 等翻译器，但其性能远未达到最佳水平，严重制约了龙芯平台的跨平台应用和高效能表现。因此，优化龙芯平台的 ARM64 动态翻译器，不仅能够提升现有 ARM 架

构应用在龙芯平台的兼容性，还能够显著提高相关应用在龙芯平台的运行效率，进而推动龙芯处理器在多领域的应用。

通过优化 ARM64 动态翻译器，可以帮助龙芯平台高效地运行 ARM 架构的库函数和应用程序。这一优化不仅有助于兼容基于 Android 和 ARM 架构的生态系统，还能够促进龙芯平台在嵌入式设备、移动设备及高性能计算领域的应用扩展。高效的翻译器将为开发者提供一个更加稳定且高性能的开发平台，推动龙芯处理器在国内外市场的竞争力提升，为自主可控的技术发展做出积极贡献。

实践证明，在掌握三大 C 编译器（GCC，LLVM，GoLang）、三大虚拟机（Java，JavaScript，.NET）和三大指令系统（MIPS，x86，ARM）的二进制翻译系统的基础上，可以在较短时间内构建良好的软件生态，而 ARM 的二进制翻译系统就是龙芯软件生态兼容的最后一块拼图。

二　国内外本学科领域的发展现状与趋势

2.1　典型的二进制翻译系统

2.1.1　QEMU

QEMU 是一种广泛使用的可移植二进制翻译系统，支持多种目标机架构（如 x86、PowerPC、ARM、Sparc、MIPS）和宿主机架构。然而，QEMU 在可移植性上做出了性能牺牲，其未经优化的模拟执行速度常常不到本地执行的 10%。这主要是由于 QEMU 为兼容多种架构，采用了体系结构无关的中间代码，无法充分利用宿主机的体系结构特性，从而限制了对翻译代码的优化。

2.1.2　Rosetta2 和 ExaGear

Rosetta2 是苹果公司为其基于 ARM 架构的 M1 和 M2 芯片推出的二进制翻译工具。最初，Apple 推出了 Rosetta1 来支持从 PowerPC 到 Intel 的迁移，而 Rosetta2 则是为支持从 Intel 到 ARM 的迁移。它能够将基于 x86 架构的应用程序无缝地转换为 ARM 架构执行，从而支持过渡期内的应用兼容性。Rosetta 2 在性能优化方面较为突出，通过静态分析和动态翻译相结合的方式，能够提高翻译执行效率。虽然它的执行速度相较于原生 ARM 应用略有下降，但与 QEMU 相比，Rosetta 2 能够充分利用 Apple 自家 ARM 架构的硬件特性，翻译过程更加高效。

ExaGear 是由俄罗斯公司 Eltechs 开发的一款二进制兼容层，旨在使基于 ARM 架构的设备能够运行 x86 架构的应用程序。它主要应用于 Linux 系统，尤其在服务器应用中广泛使用。此外，ExaGear 还可以与 Wine 结合使用，以在 ARM 平台上运行 Windows 应用程序。Wine 作为一个开源的兼容层，允许 Windows 应用在 Linux 平台上运行。当 ExaGear 和 Wine 结合使用时，Wine 负责将 Windows API 调用翻译成 Linux 可理解的系统调用，而 ExaGear 则负责将 x86 指令翻译成 ARM 指令。这种组合方式使得 ARM 设备能够在不依赖 x86 硬件的情况下运行大量的 Windows 应用程序。

2.1.3 LATX 和 LATA

x86 到 LoongArch 的体系结构翻译器 LATX（Loongson architecture translator from x86）、ARM 到 LoongArch 的体系结构翻译器 LATA（Loongson architecture translator from ARM）支持同一操作系统的用户程序翻译（如 Linux/LoongArch 上运行 Linux/x86 的应用）、不同操作系统的应用翻译（Linux/LoongArch 上运行 Windows 或者安卓应用），以及操作系统本身（直接运行 Windows 或者安卓系统），以达到消除单一指令系统壁垒的效果。

LATX 利用了 LBT 扩展的 x86 运算模拟指令、分支模拟指令以及浮点栈模式，处理 EFLAG 计算、浮点栈等问题，大幅降低了对应指令的翻译开销。此外，LATX 通过指令流分析消除不必要的 EFLAG 计算，并针对指令序列中的特定组合按照语义进行二对一的翻译，进一步提高翻译效率。

LATA 在 QEMU 的基础上，剔除了 QEMU 使用的 TCG IR 中间表示，使用龙芯伪指令的设计模式。LATA 目前只支持部分定点指令的翻译，不支持浮点和向量指令的翻译，也不支持 LBT 拓展指令。LATA 仅通过软件模拟的方式计算标志位，即使通过延迟计算的方式消除了部分冗余的标志位运算，相比于 LATX 中通过硬件拓展指令模拟的方式，翻译开销仍比较大。

2.1.4 Berberis 和 Houdini

Intel 的 Houdini 项目是在 x86 架构设备上运行 ARM 应用的一套解决方案，其核心机制也是通过动态二进制翻译技术，将 ARM 指令实时转换为 x86 指令集。相比与传统的二进制翻译，Houdini 不会对整个安卓程序进行翻译，而是采

用一种混合执行的架构。首先，ART（Android Runtime）会将 Java 源码编译生成的字节码文件加载至内存中执行，字节码通过 ART 的抽象层屏蔽硬件差异，实现一次编写，多架构运行；当 Java 代码通过 JNI（Java Native Interface）调用 ARM 架构原生库（.so 文件）时，libhoudini.so 作为动态二进制翻译层介入，实现指令架构层面的转换。为了避免全指令模拟，Houdini 实现了系统调用代理，将 ARM 库中的系统调用（如 open、mmap）映射到宿主系统的系统调用；同时，对于一些高频函数，直接映射为 x86 原生实现，极大地减少翻译开销 。

Berberis 是一款开源的用户空间动态二进制翻译器，这个项目的推出主要为了配合 Android Open Source Project(AOSP) 对 RISC-V 指令集的支持。Berberis 的核心动机是使开发者能够在没有 RISC-V 硬件的情况下，通过动态二进制翻译将 RISC-V 指令 (主要是 RV64GCV_Zba_Zbb_Zbs_Zcb 指令集) 转换为 x86-64 指令，从而解决开发者无法获得 RISC-V 硬件的问题。

2.2 二进制翻译系统的瓶颈分析

2.2.1 DynamoRIO

DynamoRIO 是一个运行在 Windows 和 Linux 系统的二进制翻译平台，常用于动态分析、程序优化、程序保护等领域。DynamoRIO 的性能开销大体上可以分为两类：应用程序代码和 DynamoRIO 代码；应用程序代码是包括被缓存到基本块和 Trace 中的代码；DynamoRIO 代码包括翻译程序的代码以及为了程序正确执行在代码缓存中插入的额外指令，这些指令在原生程序中不会运行。

DynamoRIO 统计了生成代码执行和二进制翻译本身的时间，分别占比 93.8% 和 6.2%。DynamoRIO 将增加的开销主要分为 3 个部分，间接分支，哈希表查找和其他开销；DynamoRIO 对间接分支处理的开销较大，因为翻译程序需要额外的指令来确保分支目标是否仍然在当前 Trace 中。如果分支目标没有保持在追踪内，就需要执行一个哈希表查找来识别目标。即使在追踪中内联了间接分支，每次分支仍然需要检查目标是否在 Trace 内，间接分支大约占执行时间的 2%。哈希表查找大约占了 4% 的执行时间，为了优化间接跳转的查找，DynamoRIO 对查找过程进行了部分内联处理，这是 DynamoRIO 添加的主要性能开销来源；其它开销包括 DynamoRIO 代码的其他部分和查找常规的开销，占了总计不到 1% 的执行时间。

虽然 DynamoRIO 对二进制翻译的执行开销进行了分类和统计，但是仅仅着重于间接分支和对其的哈希查找上，分析的过程和结果还不够全面。

2.2.2 Borin‘s StarDBT

StarDBT 是一个用户级二进制翻译系统，只支持单一架构（x86）的代码翻译。StartDBT 将翻译操作和相关开销分为五大类：1. 初始化开销（Initialization Overhead）：在翻译和执行应用程序之前，DBT 工具需要加载到内存并初始化；2. 冷代码翻译开销（Cold Code Translation Overhead）：StarDBT 遇到新的基本块时，翻译代码并放入代码缓存过程中产生的开销。3. 代码分析开销（Code Profiling Overhead）：为了加速应用程序执行，StarDBT 使用运行时信息来检测热代码并进行优化。这个过程包括两种开销：代码插桩开销和执行插桩指令的开销。4. 热追踪构建开销（Hot Trace Building Overhead）：在识别热块后，StarDBT 构建热代码所产生的开销。5. 翻译代码执行开销（Translated Code Execution Overhead）：为了保持原程序的行为，翻译器模拟翻译的指令产生的开销。

Borin 的测算方法非常巧妙，他通过对二进制翻译器源码进行适当的修改，创建了多个修改版本，每个版本会“开启”或“关闭”其中的一些特定部分。通过测试这些修改版本的执行时间，并结合一系列运算，他能够精确地得到各个部分的耗时。

尽管 Borin 的工作在分析二进制翻译的开销方面较为全面，但由于需要修改二进制翻译器的源码，对于一些商业或者闭源的翻译器并不适用。同时，他的分析方法仅适用于同一指令集的同源二进制翻译器，因此其通用性较差。

2.2.3 Deflater

Deflater 是一个基于指令膨胀率的二进制翻译器分析框架。此前的分析方法在分析代码缓存中的代码时，只能统计宿主代码总体的开销，无法分析开销的具体来源，比如 x86 的 add 指令在 arm 或者 loongarch 架构上会膨胀成多少条指令，这类数据对于分析跨架构的二进制翻译的开销非常关键，因为跨架构二进制翻译器中由于指令集差异带来的指令开销非常明显。指令膨胀率的定义是翻译后的动态指令数和原生程序动态指令数的比值。，膨胀率和翻译器的执行时间成正相关关系；对于相同架构的翻译器，指令的膨胀率越高，翻译器在同一平台上的

执行时间越长。

分析框架主要由三部分组成：数学模型，渺测试集和膨胀模拟器。数学模型论证了总体膨胀可以通过单条指令的膨胀与基本块的优化计算得到，是膨胀率解析器的理论依据。渺测试集是一组黑盒测试，用于在无需依赖于二进制翻译器源码的前提下提取膨胀信息。膨胀率模拟器是一套基于踪迹的模拟器，利用渺测试集提取出的膨胀信息分析二进制翻译的指令膨胀。

Deflater 能够分析翻译器的翻译规则和优化，而且这种分析是黑盒的，对于没有源码的商业二进制翻译器也适用。同时，Deflater 通过量化指令数和执行时间的关系，精准计算出不同架构间指令翻译的膨胀率，从而在指令层面细粒度地分析宿主代码的开销来源。

2.3 二进制翻译系统的常见优化

2.3.1 纯软件的优化

纯软件的优化包括分支跳转优化，代码缓存优化，特殊指令优化等。

分支跳转包括直接分支跳转和间接分支跳转两类。直接分支跳转的目标地址唯一，通过程序运行时地址回填等方法即可确定。间接分支跳转的目标地址在程序执行时确定，跳转目标地址不唯一。对于间接分支跳转目标地址的确定，通常是首先从缓冲区代码中进行查找，如果查找失败则控制流切换到翻译器开展翻译。此过程涉及频繁的控制流切换和翻译查找。间接分支跳转目标地址的计算是影响二进制翻译效率的关键瓶颈。间接分支跳转目标地址查找是确定源程序跳转地址和目标程序跳转地址关系映射的过程，查找十分耗时。D’Antras 等人基于硬件辅助生成一个包含多 case 目标地址的分支跳转表，通过目标地址反推理法获取间接分支跳转指令，引入间接跳转优化后，可以降低 40% 的运行时维护开销。对于未涵盖的其余间接分支使用快速原子哈希表处理，将每个哈希表条目的源和目标地址打包为 64 位指针并原子的从共享缓存中读写，避免在分支查询时加入栏栅同步指令，加速多线程翻译对间接跳转的同步效率.

跨平台指令集的差异性是限制二进制翻译生成高效目标码的一个关键瓶颈，对于体系结构差异性较大的特殊指令可以采用解释执行或者高级语言实现的函数模拟，然而这种翻译方法的翻译效率差强人意。不同平台的浮点和向量指令在精度处理、异常舍入等方面存在差异，为了能精确模拟浮点结果，QEMU 采

用高级语言实现的 Helper 函数模拟，并未考虑目标平台硬件特性，翻译效率较低。对于携带标志位指令的模拟是影响二进制翻译性能的又一个关键瓶颈。x86 和 ARM 等架构采用专用标志位处理条件转移指令，并基于标志位实时反映处理器运行时的各种状态和运行结果。而 MIPS、RISC-V、Alpha 等并没有标志位寄存器，翻译时采用软件模拟标志位运算。标志位的模拟不仅要完成标志寄存器逻辑功能，还要实时更新标志位的状态，引入了大量的内存访问和额外计算开销。为了提高标志位的翻译效率并减少非必要的计算量，FX!32 等引入了延迟计算技术，将标志位的计算向后拖延到执行阶段。延迟计算有效地减少了代码计算量，但是部分标志位的定值并不会被后续指令使用。

缓存已翻译代码可有效避免重复翻译，代码缓存设计是当前二进制翻译系统的普遍做法。在二进制翻译执行过程中，当不确定后续执行代码块时，优先从缓存中查找，如果未命中再切换到翻译器开展翻译。对于缓存的查找也会带来额外的代码查找开销。

2.3.2 软硬结合的优化

针对源平台和目标平台体系结构差异大的问题，LoongArch 在原有龙芯指令系统基础上增加了 MIPS 不具备但 x86 和 ARM 具备的核心功能，在指令功能、运行时环境、核心态功能等方面实现处理器的硬件扩展。测试 Linux 操作系统启动时间，结果表明基于软硬协同加速后龙芯二进制翻译器性能提升了 21 倍。

为了高效保证内存一致性，Rosetta2 在 ARM 硬件上支持了 x86 的强内存模型，当翻译 x86 代码时会自动切换到强内存模型，而运行 ARM 程序时再切回到其原生内存模型，硬件支持的内存模型有效降低了并发程序翻译时内存模型变换带来的开销。

此外，有研究利用硬件加速提升程序的并行度。Transmeta 利用 Crusoe 处理器的 VLIW 指令特性，重组翻译后的二进制代码以实现同步执行。Crusoe 增加了特殊的硬件功能来检测同一个地址的“读-写-读”冲突，支持对访存指令进行激进式的代码调度。

三　课题主要研究内容、预期目标

课题围绕“龙芯平台的 ARM64 动态翻译器的优化研究“展开，在已有的龙芯平台的二进制翻译器 LATA 的基础上，实现一个更加全面，效率更高的动态二进制翻译器 LATA-V。研究工作主要分为两个阶段展开：分别是 LATA-V 框架搭建和 LATA-V 性能分析和优化。

阶段一的主要工作是实现 LATA-V 的主要框架，并进一步完善指令翻译，尤其是支持 ARM 的向量指令和浮点指令到龙架构的翻译；同时在指令翻译的过程中，引入指令随机测试框架，支持单指令的随机测试，有利于在翻译器的初级阶段快速定位错误的指令翻译。之后，为了方便调试翻译器运行大型应用时，设计并实现了基于 QEMU 的调试框架。阶段一的主要目标是在龙芯平台上正确运行架构为 ARM64 的 SPEC CPU2006 基准测试程序，包括所有的定点和浮点子项。

阶段二的主要工作是在阶段一的翻译器的基础上，基于指令膨胀率的性能分析方法，深入分析 LATA-V 的性能瓶颈；之后，在分析结果的基础上，结合 Loongarch 架构和 ARM 架构的异同点，充分利用龙芯平台和龙芯指令集的优势，针对 SPEC CPU2006 的负载，进行针对性的性能调优。阶段二的预期目标是在 SPEC CPU2006 的负载上，LATA-V 的翻译执行的效率能够达到原生效率的 40%。

四　拟采用的研究方法、技术路线、实验方案及其可行性分析

本课题中指令模拟部分是方法分为两部分，其中定点指令通过直接翻译的方式，由 ARM 指令直接翻译成龙架构的指令。浮点和向量指令支持软浮点和硬浮点两种方式，其中软浮点的方式通过插桩实现，效率较低；硬浮点方式支持 ARM 指令到龙芯的浮点和向量指令的直接翻译，效率比较高。

指令翻译的正确性验证由随机指令测试框架和调试框架完成。随机指令测试框架通过对比每条翻译指令执行后的 CPU 状态来测试指令翻译的正确性和完备性，对比的对象是真实的 ARM 机器或者一个正确执行的翻译器，比如 ARM 架构的 QEMU。对于指令随机组合的正确性验证，则由调试框架完成，通过对比不同应用程序在 LATA-V 和 QEMU 上的执行结果，确定翻译器 LATA-V 功能是否正确。

翻译器的性能分析主要和龙芯平台上的原生程序进行对比实验完成。通过

测试程序执行的指令数、运行时间等性能数据，对相应的优化进行定量的分析。

五　已有科研基础与所需的科研条件

本课题中，为了在龙芯平台上运行翻译器 LATA-V 并进行相关的实验，使用的机器配置为：3A5000 处理器芯片，搭载 2*8GB DDR4 3200MHz 内存条，运行 Loongnix 操作系统，使用 GCC 8.3.0 编译器版本，安装调试框架中用来对比的翻译器 QEMU，版本为 qemu-aarch64 version 7.1.0。

以上的科研条件均已具备。

六　研究工作计划与进度安排

2024 年 5 月-2024 年 9 月：确定毕业设计课题及工作方向，着手开展学科领域的研究工作，完成毕业设计初期准备工作。

2024 年 9 月-2023 年 11 月：完善 LATA-V 的指令翻译，能够正确执行 Coremark 负载。

2024 年 11 月-2025 年 1 月：完成 LATA-V 的整体框架，支持 ARM 的向量和浮点指令到龙架构的高效翻译。

2025 年 1 月-2025 年 2 月：完成 LATA-V 的随机指令测试框架和调试框架

2025 年 2 月-2025 年 3 月：完成 LATA-V 的瓶颈分析和性能调优。

2025 年 3 月-2025 年 4 月：整理实验数据，撰写毕业论文。

2025 年 4 月-2025 年 5 月：完成毕业设计论文与毕业。

参考文献

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

[2] 燕澄皓. 二进制翻译中的指令膨胀分析与优化研究 [M]. 中国科学院计算技术研究所, 2024.

[3] 胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术 [J/OL]. 计算机研究与发展, 2023, 60(1): 2-16. https://crad.ict.ac.cn/cn/article/doi/10.7544/issn1000-1239.202220196.

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

[5] 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): 2687-2723.

[6] 邹旭. 面向二进制翻译的随机化测试生成研究 [D]. 中国科学院计算技术研究所, 2021.

[7] Bellard F. Qemu, a fast and portable dynamic translator [C]//USENIX Annual Technical Conference. 2005: 46.

[8] Borin E, Wu Y. Characterization of dbt overhead [C/OL]//2009 IEEE International Symposium on Workload Characterization (IISWC). 2009: 178-187. DOI: 10.1109/IISWC.2009.5306785.

[9] Bruening D, Amarasinghe S. Efficient, transparent, and comprehensive runtime code manipulation [J]. 2004.

[10] d'Antras A, Gorgovan C, Garside J, et al. Optimizing indirect branches in dynamic binary translators [J]. ACM Transactions on Architecture and Code Optimization (TACO), 2016, 13 (1): 1-25.

[11] Guo H, Wang Z, Wu C, et al. Eatbit: Effective automated test for binary translation with high code coverage [C]//2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2014: 1-6.

[12] Hiser J D, Williams D, Hu W, et al. Evaluating indirect branch handling mechanisms in software dynamic translation systems [C]//International Symposium on Code Generation and Optimization (CGO'07). IEEE, 2007: 61-73.

[13] Hookway R. Digital fx! 32 running 32-bit x86 applications on alpha nt [C]//Proceedings IEEE COMPCON 97. Digest of Papers. IEEE, 1997: 37-42.

[14] Inc. A. About the rosetta translation environment [J/OL]. Apple Developer Documentation, 2023. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment.

[15] Shi Q, Zhao R. Floating point optimization based on binary translation system qemu [C]//2016 2nd Workshop on Advanced Research and Technology in Industry Applications (WARTIA-16). Atlantis Press, 2016: 1338-1343.

[16] Xie B, Yan Y, Yan C, et al. An instruction inflation analyzing framework for dynamic binary translators [J/OL]. ACM Trans. Archit. Code Optim., 2024, 21(2). https://doi.org/10.1145/3640813.

[17] Yue F, Pang J, Han X, et al. An improved code cache management scheme from i386 to alpha in dynamic binary translation [C]//2010 Second International Conference on Computer Modeling and Simulation: volume 2. IEEE, 2010: 321-324.