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

# 研究生学位论文中期报告

| | | | |
|---|---|---|---|
| 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　学位论文进展情况，存在的问题，已取得阶段性成果

## 1.1　LATA-V 动态翻译器的设计和实现

在翻译器 LATA 的基础上，我实现了龙芯平台上 ARM64 的动态翻译器 LATA-V；相比于前者，LATA-V 进一步完善指令翻译，支持了 ARM 的向量和浮点指令到龙架构的翻译；在指令翻译的过程中，引入指令随机测试框架，能够根据指令 pattern 和指令数量生成测试案例，支持单指令的随机测试。同时，为了方便翻译器在运行大型应用时的调试，设计并实现了基于 QEMU 的调试框架。目前，LATA-V 的总体框架已经基本完成，实现了约 500 条 ARM 架构指令的翻译，其中浮点和向量指令约为 200 条左右，LATA-V 已能够正确运行 ARM64 的 SPEC CPU2006 基准测试程序，包括所有的定点和浮点子项；随机指令测试框架也已经能够正常运行，能够在指令翻译和性能优化过程中发现错误的翻译；调试框架的总体功能也基本实现，完成了 LATA-V 和 QEMU 的寄存器状态的对齐，支持指令粒度和基本块粒度的检查点的设置，能够定位不同负载中指令随机组合的翻译错误，弥补随机指令测试框架只能进行单指令测试的不足。

### 1.1.1　LATA-V 总体框架

LATA-V 的总体框架主要由 3 个部分组成，包括代码查找，代码执行和指令翻译等模块，主要的流程如图1所示。

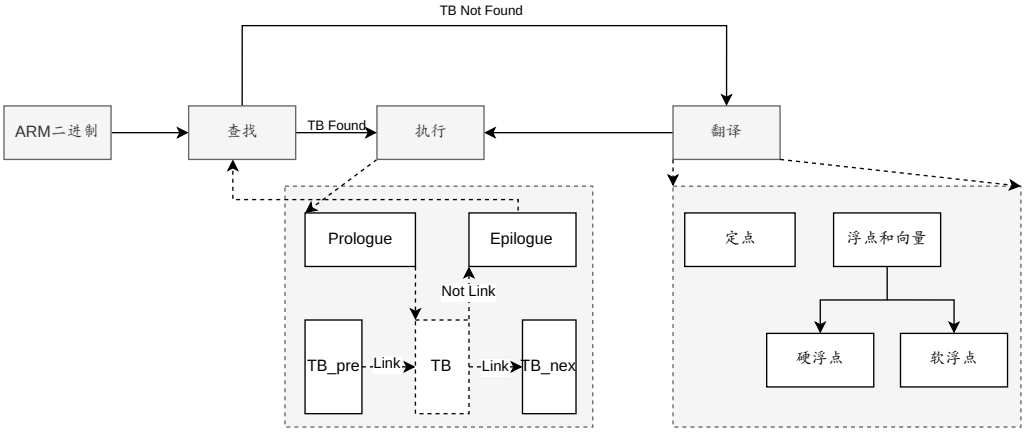


**图 1　LATA-V 动态翻译器**

**Figure 1　LATA-V Dynamic Translator**

**代码查找：**在 LATA-V 翻译器中，大部分代码块会被重复执行；为了减少代

码块的重复翻译，翻译好的代码块通常会被存储到代码缓存中；在执行翻译代码前，翻译器会先根据下一个代码块的 PC 查找代码缓存，如果代码块在代码缓存中命中，翻译器会进入执行阶段；如果没有命中，翻译器则会进入指令翻译阶段。

**指令翻译：**在指令翻译阶段，LATA-V 翻译器会以基本块为粒度进行指令翻译。LATA-V 中基本块的划分受多种条件的约束：一是受基本块内指令的数量的影响，一般来说，基本块中指令的数量是翻译器的可以设置的一个参数，例如可以设置基本块的指令数量为 1，从而实现指令的单步执行，LATA-V 也不例外；同时，由于翻译资源的限制，基本块内的指令数量会有一个上限值，该上限值的大小和翻译器的指令集架构相关，在 LATA-V 中该值为 512。二是基本块的划分受指令类型的影响，在 LATA-V 中，基本块以跳转指令或者系统调用指令为结束的标志，这是因为在二进制翻译中，基本块通常定义为一个具有单一入口和单一出口的连续指令序列，其内部不包含控制流的中断。在划分基本块的过程中，LATA-V 会对二进制代码进行解码，提取出操作码以及其他相关信息（如操作数、寻址模式等），解码后的操作码会作为键，在预先构建的映射表中查找相应的翻译函数，该映射表在 LATA-V 中是一个巨型的 switch-case 结构。查找到对应的翻译函数后，翻译器调用该函数将 ARM 架构的指令直接转换为 Loongarch 架构的伪指令表示，同时进行必要的优化和重定位。重定位阶段，翻译器会将龙架构的伪指令转换为汇编源代码，之后汇编模块将由汇编源码生成的二进制代码装入代码缓存中。如果映射表中未找到匹配的操作码，LATA-V 通常会调用一个默认的错误处理函数或支持的补充模块，以处理特殊或未预见的指令。指令翻译根据指令类型的不同，可以分为定点指令，浮点和向量指令以及特殊指令的翻译。定点指令的翻译是 LATA-V 中比较容易的部分，唯一需要注意的是：由于 ARM 架构和 Loongarch 架构对于高位拓展的处理不同（ARM 架构高位清零，Loongarch 架构高位符号拓展），在使用龙架构的指令模拟 32 位的 ARM 架构指令时，需要对相应的结果进行高位清零操作。浮点和向量指令在翻译阶段有两种处理方式：一是硬浮点的方式，直接通过对应的龙架构的向量和浮点指令进行翻译，包括向量拓展指令（LSX）和高级向量拓展指令（LASX），分别支持 ARM 架构 128 位和 256 位向量和浮点指令的翻译；二是软浮点的方式，软浮点的存在是为了解决硬件浮点支持不足时对浮点运算的需求，LATA-V 会将一些难以通过 Loongarch 浮

点指令拓展的直接翻译的 ARM 指令映射到 helper 函数，这些函数是在 LATA-V 内部实现的 IEEE 754 标准的浮点运算，保证模拟环境中浮点运算的正确性和一致性；软浮点方式翻译的浮点指令的效率通常比较低，所以只针对一些指令频度较低或者实在难以直接翻译的 ARM 浮点指令才使用软浮点方式进行翻译。对于一些特殊指令，例如系统调用指令或者一些非法指令，LATA-V 也是通过 helper 函数通过将模拟的过程通过高级语言实现，减少翻译的难度，同理这些指令的翻译效率也是较低的，具体效率较低的原因在代码执行部分会有涉及。

**代码执行：**对于二进制翻译器来说，代码的执行分成 3 个部分，分别是翻译态，执行态以及翻译器和执行态间的上下文切换。翻译态包括代码查找和指令翻译阶段的代码，执行态则包括代码缓存中的代码以及的 helper 函数中的代码，而上下文切换部分由两个基本块组成，在图1中由 Prologue 和 Epilogue 命表示。Prologure，Epilogue 以及翻译好的基本块 TB 共同组成了代码缓存。翻译好的基本块在代码缓存中只是简单地按顺序放置，如果不做基本块间的链接，每个基本块结束后都需要返回翻译态中查找下一个基本块块的入口地址，这样会带来巨大的上下文切换开销。通过链接，当一个基本块的末尾遇到跳转指令（如无条件跳转、条件跳转、调用等）时，LATA-V 会尝试解析跳转地址：如果目标基本块已经被翻译并存在于代码缓存中，则 LATA-V 会将当前基本块末尾的跳转指令直接修补为跳转到目标块的入口地址。如果目标块尚未翻译，则可以先使用一个跳转桩（trampoline）或者占位符，待目标块翻译完成后，再进行修补。上下文切换的高开销是导致翻译器效率低下的关键因素之一，同时也是 helper 函数性能较低的主要原因。

### 1.1.2 随机指令测试框架

随机指令测试框架主要由 2 部分组成，分别为随机指令生成器以及验证器；随机指令生成器是一个生成测试数据块的 Perl 脚本，可以在任意地方执行。验证器是一个运行在目标架构（例如 ARM 架构）的 Linux 可执行文件。随机指令测试框架的流程如图2所示。首先，测试配置文件中定义了待测试的指令格式，包括指令编码，寻址模式以及相关的限制。生成器脚本根据测试配置文件以及用户输入的指令模板和测试数量生成包含随机指令的二进制指令数据，该数据是一条待测试指令和一条非法指令交替排列的，非法指令的用处之后会解释。随后，

验证器程序分别在主设备（真实硬件或参考平台，这里使用的是 QEMU）和从设备（待测模拟器，这里是 LATA-V）上加载并执行该数据，执行过程中每条指令后的寄存器状态都会进行交叉比对。若出现不一致或异常，则记录到日志并输出错误信息；若无异常则继续执行，直至全部指令完成。最终，主设备端会打印寄存器状态及对比的结果，帮助定位并修复模拟器在指令实现或解码上的问题，从而有效验证指令集的正确性与稳定性。

寄存器检查通过为非法指令信号（SIGILL，Signal Illegal Instruction）注册一个例外处理程序来实现，该处理程序可通过其 sigcontext 参数访问寄存器内容；在 Unix/Linux 系统中，当一个信号被触发时，内核会捕获当时进程的状态信息，并将其保存到一个结构体中，这个结构体通常称为 sigcontext。它包含了当时 CPU 的寄存器值、程序计数器（PC）、栈指针、标志寄存器等关键信息。对于使用 SA_SIGINFO 标志设置的信号处理程序，处理函数通常会收到一个指向 ucontext_t 的指针，其中就包含了一个 sigcontext。这样，信号处理程序就可以通过 sigcontext 获取或修改进程在信号发生时的状态，这在调试、错误恢复或捕捉非法指令（如 SIGILL）的场景中非常有用。

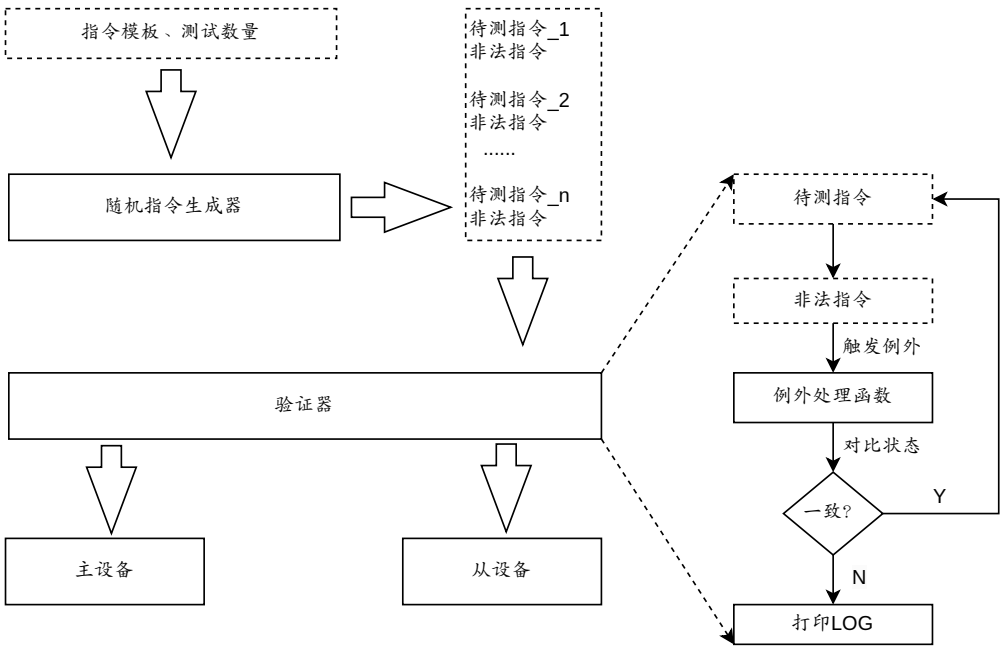


**图 2　随机指令测试框架**

**Figure 2　Random Instruction Testing Framework**

### 1.1.3 基于 QEMU 的调试框架

调试框架分为顺序模式和二分模式，能够实现指令粒度，基本块粒度的调试。顺序模式能够精确定位出错的指令或者基本块；二分模式二分查找程序的出错点，能实现快速定位；

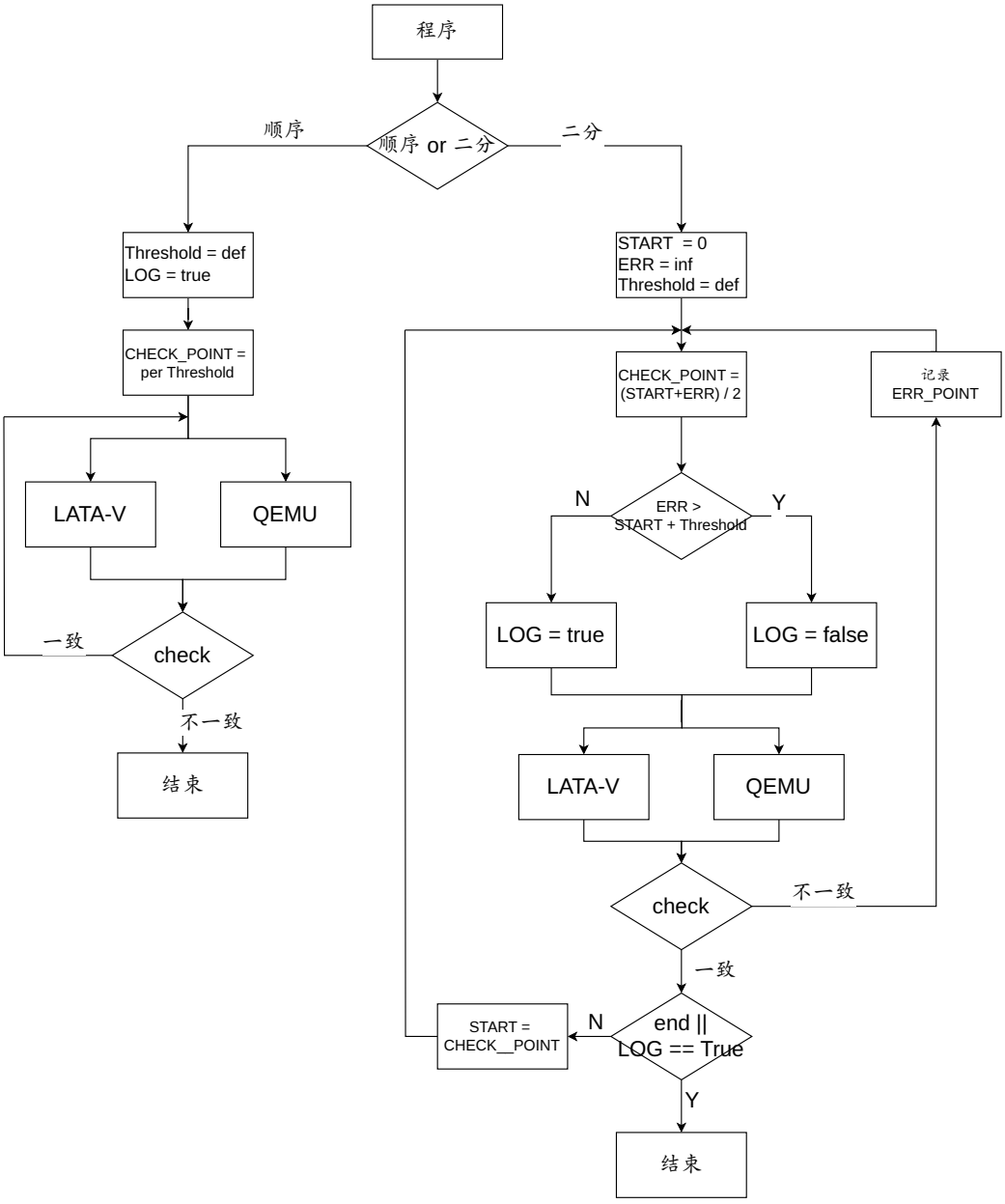


**图 3　基于 QEMU 的调试框架**

**Figure 3　Debugging framework based on QEMU**

## 1.2 LATA-V 的瓶颈分析

LATA-V 的瓶颈分析是在指令膨胀率的基础上完成的。在 SPEC CPU 2006 的负载上，统计 LATA-V 的动态指令数频率，如图4所示。膨胀率分析会在动态指令数占比和单指令的膨胀的基础上，计算得出某类指令在总体膨胀中的占比；单指令膨胀和总体的膨胀数据还未统计完成。

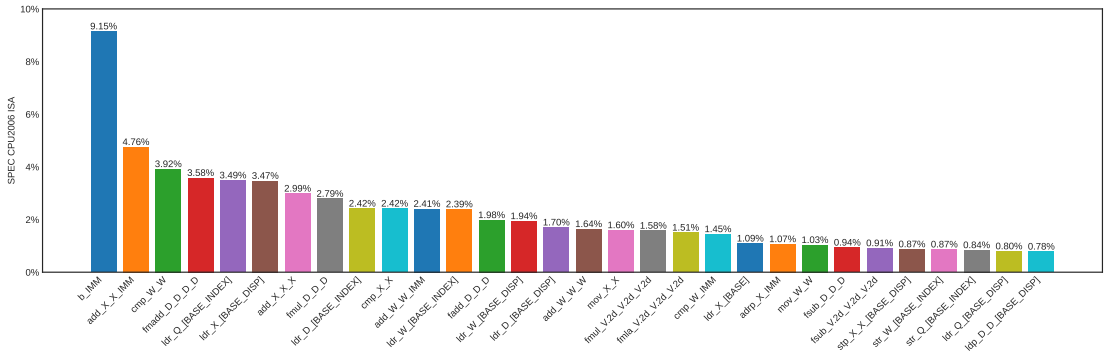


**图 4 SPEC CPU 2006 动态指令数占比**

**Figure 4 Proportion of Dynamic Instruction Count in SPEC CPU 2006**

## 1.3 LATA-V 的性能优化

### 1.3.1 复用 X86-LBT 优化

复用 X86-LBT 优化的主要思想是将龙芯平台已经实现的 X86 硬件拓展指令用在 ARM 标志位运算指令的翻译中，而想要达成指令的复用，要求 X86 和 ARM 平台的标志位和指令对标志位的影响有相似性。

X86 架构有一套丰富的状态标志位，存储在标志寄存器（EFLAGS）中。这些标志位包括进位标志（CF）、奇偶标志（PF）、辅助标志（AF）、零标志（ZF）、符号标志（SF）、溢出标志（OF），在 EFLAGS 寄存器中的位置如图5a 所示；ARM 处理器的状态寄存器（FPSCR）中有 4 位标志位（NZCV），分别对应符号标志（N）、零标志（Z）、进位标志（C）和溢出标志（V），在 FPSCR 寄存器中的位置如图5 b 所示; 其中 ARM 中的标志位 N，Z，C，V 分别对应 x86 中的标志位 SF，ZF，CF 和 OF。分析了 2 种架构标志位的设计不难发现，且相比于 x86 架构，ARM 架构的标志位恰好是 x86 的子集，也就是说只要实现了 X86 的标志位寄存器，就统一了 2 种架构的标志位寄存器，唯一需要调整的是标志位在各自状态寄存器中的位置。

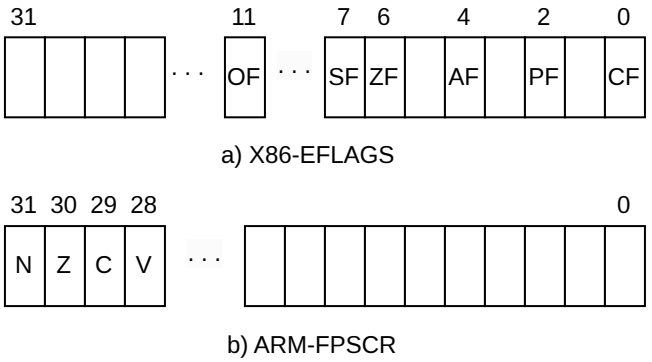


**图 5　x86 和 ARM 运算标志位**

**Figure 5　Arithmetic Flags of x86 and ARM**

指令对标志位的影响如表1所示：其中表1a 定义了 7 种指令对标志位的影响，表1b 和表1c 分别表示 ARM 指令和 x86 指令标志位的影响。从表1中可以看出，x86 架构中几乎所有的算术、逻辑和移位运算指令都会影响 ELFAGS 寄存器，并且 x86 架构存在子寄存器运算的情况，包括 8 位，16 位，32 位和 64 位的寄存器长度；ARM 架构中只有部分的运算和逻辑指令会影响 FPSCR 寄存器，移位指令本身并不影响标志位，同时 ARM 中的寄存器长度只有 32 位和 64 位；所以，从指令种类的角度来说，大部分 ARM 的标志位运算指令能在 x86 中找到对应逻辑的指令。

经过测试可以发现，ARM 中的 ADCS，ADDS，ANDS，EORS，CMP 指令对标志位的修改和 x86 中的 ADC，ADD，AND，XOR 和 SUB 指令的行为一致。BICS 指令本身的语义比较复杂，但是最终会影响标志位的是两个操作数的与操作，所以只需要在执行 BICS 执行之前正确维护操作数的值即可；SBCS 和 CMN 指令的语义都是先将减数按位取反，不同的是 SBCS 保持进位不变，CMN 指令会将进位临时置 1，最终影响标志位的是被减数、减数取反后的数以及进位三者相加的结果，所以 SBCS 和 CMN 指令对标志位的影响可以复用 x86 的 ADC 指令。

**表 1　指令对标志位的影响**

**Table 1　The impact of instructions on flag bits**

**a) 指令影响标志位的说明**

| 缩写 | 含义 |
|---|---|
| T | Tests，使用该标志位 |
| M | Modify，置 0 或者置 1 |
| 0 | Reset，置 0 |
| 1 | Set，置 1 |
| — | Undefined，未定义 |
| R | Restore，恢复之前的值 |
| Blank | 空表示不影响 |

**b) ARM 指令对标志位的影响**

| Instructions | N | Z | C | V |
|---|---|---|---|---|
| 01.ADCS | M | M | M | TM |
| 02.ADDS | M | M | M | M |
| 03.ANDS | M | M | 0 | 0 |
| 04.BICS | M | M | 0 | 0 |
| 05.EORS | M | M | M | M |
| 06.CMP | M | M | M | M |
| 07.CMN | M | M | M | M |
| 08.SBCS | M | M | M | TM |

**c) X86 指令对标志位的影响**

| Instructions | OF | SF | ZF | AF | PF | CF |
|---|---|---|---|---|---|---|
| 01.ADC | M | M | M | M | M | TM |
| 02.ADD | M | M | M | M | M | M |
| 03.AND | 0 | M | M | — | M | 0 |
| 04.CMP | M | M | M | M | M | M |
| 05.DEC | M | M | M | M | M | |
| 06.IMUL | M | — | — | — | — | M |
| 07.MUL | M | — | — | — | — | M |
| 08.NEG | M | M | M | M | M | M |
| 09.OR | 0 | M | M | — | M | 0 |
| 10.ROL/ROR | M | | | | | M |
| 12.SAL/SAR/SHL/SHR | M | M | M | — | M | M |
| 16.SBB | M | M | M | M | M | TM |

### 1.3.2　高位清零消除优化

在 ARM64 中，写 32 位子寄存器会被零拓展到 64 位；在 LoongArch 中，写 32 位子寄存器会被符号拓展成 64 位。在翻译 ARM64 的 32 位子寄存器运算时，需要额外的指令实现零拓展。

优化主要分为两部分：

- **基本块内的优化**：对寄存器的操作可以分为 4 类：Read_w，Write_w，Read_x，Write_x，分别表示对 32 位/64 位寄存器的读写；分析每个寄存器的数据流，只有出现先 Write_w 后 Read_x 的数据流时，才需要在 Read_x 前对高位寄存器清零。同时在每个基本块尾部，需要对数据流中最后存在 Write_w 操作的寄存器进

行高位清零。

- **基本块间的优化**：由于每个基本块只有两个出口，对于存在循环的基本块，在基本块的尾部的清零操作可以移动到基本块出口，减少当前循环的指令数量。

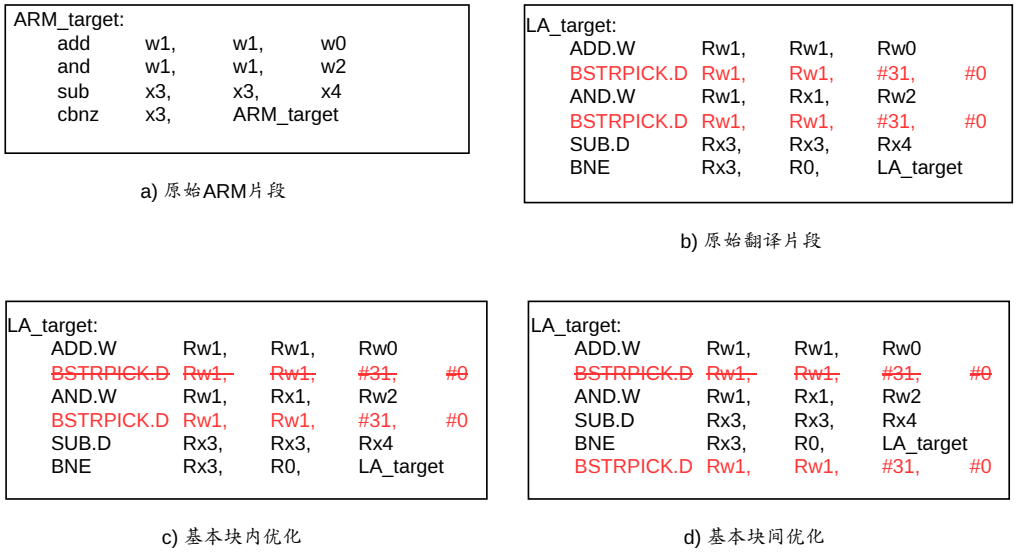


**图 6　高位清零消除优化**

**Figure 6　High-bit Zeroing Elimination Optimization**

## 二　下一步工作计划和内容，预计答辩时间

2024 年 2 月-2024 年 3 月

- 优化 LATA-V 的指令翻译和代码结构；
- 完善 LATA-V 的瓶颈分析和性能优化；
- 提升随机指令测试框架的测试覆盖率，减少调试框架的调试时长。

2024 年 3 月-2024 年 4 月

- 完善 LATA-V 整体框架，提升 LATA-V 在负载 SPEC CPU 2006 上的效率；
- 完善随机指令测试框架和调试框架；
- 撰写毕业论文。

## 三　已取得科研成果列表（已发表、待发表学术论文、专利等）

无

## 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.