中国科学院大学

University of Chinese Academy of Sciences

硕士学位论文

面向二进制翻译的随机化测试生成研究

作者姓名：

指导教师：

学位类别：工学硕士

学科专业：计算机系统结构

培养单位：中国科学院计算技术研究所

2021 年 6 月

Research on Random Test Generation for Binary Translation

A thesis submitted to the

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Engineering

in Computer System Architecture

By

Institute of Computing Technology, Chinese Academy of Sciences

June, 2021

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。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保存和使用学位论文的规定，即中国科学院大学有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　　　　　　　　　导师签名：

日　　期：　　　　　　　　　　日　　期：

摘 要

二进制翻译技术是一种跨指令集架构（Instruction Set Architecture，简称 ISA）软件兼容方案。自二进制翻译技术出现以来，基于各种指令集架构的二进制翻译器陆续被开发出来。这些翻译器采用自动生成或者手工编写的翻译规则，实现目标软件的跨平台运行。然而，开发一个正确（且高效）的二进制翻译器是非常困难的。二进制翻译器中随机出现的错误往往要耗费开发人员大量的时间进行调试。对二进制翻译器进行全面的测试能尽早暴露其错误。目前，面向二进制翻译测试的相关研究非常少，开发者往往只能手写测试或者使用一些已有的测试集。手写测试集扩展性差、测试量少。已有测试集不具针对性、代码覆盖度低。因此，设计一种可不断产生新测试程序的测试生成系统很有必要。

本文以龙芯中科开发的二进制翻译器 XQM（X86 to MIPS Based on QEMU）为测试目标，参考芯片验证系统设计出面向二进制翻译的随机化测试生成器。该生成器具有扩展性好，可灵活面对各种测试场景，生成的代码简短易调试等优势。本文的贡献如下：

1. 设计并实现了针对二进制翻译的随机化测试生成器 BTRTG（Binary Translation Random Test Generator）。与传统的随机化测试不同，本文根据二进制翻译器的翻译特点，设计了一套以测试模板为引导，能灵活面对各类测试场景的测试生成器。该生成器根据测试工程师编写的测试模板自动生成 32 位 X86 汇编程序，既能在开发早期用于验证单指令翻译的正确性，也能在开发中后期用于验证优化翻译的正确性。

2. 选取了四个测试场景并使用 BTRTG 进行了针对性测试。为保证测试具有代表性，本文选取了单指令翻译和优化翻译等典型翻译方式下的测试场景各两个。其中，有三个实验均发现了翻译器中的错误，另一个实验提升了代码的测试覆盖率。

3. 生成了一套轻量但高效的测试集。该测试集包含了部分单指令翻译的测试和优化翻译的测试。本文通过代码覆盖率测试发现，该测试集对翻译器源代码各模块的覆盖率都比较高，总体分布均匀。此外，该测试集测出了传统测试集没有测出的新错误。

关键词： 二进制翻译，随机化测试，测试场景，代码覆盖率

Abstract

Binary translation is one approach to implement cross-ISA(Instruction Set Architecture) software compatibility. Since the proposal of binary translation, binary translators based on various ISAs have been developed one after another. These translators achieve cross-platform emulation by automatically generated or handwritten translation rules. However, it is not easy to develop a correct (and efficient) binary translator. Random errors in binary translators can cost developers a lot of time to debug. A comprehensive test of the binary translator can expose its errors as early as possible. Up to now, there was little research on the testing of binary translation. Developers can only use handwritten tests or some existing test suites. Handwritten tests are of poor expansibility and small quantity. Existing test suites lack pertinency and with low code coverage. Therefore, it is very necessary to design a test system that can generate new test programs continuously.

This paper takes the binary translator XQM(X86 to MIPS Based on QEMU) developed by Loongson as the test target. We referred to the chip verification system and designed a randomized test generator for binary translation. This generator achieves good scalability and flexibility in various test scenarios. The generated code is short and easy to debug. The main contribution is as follows:

1. A randomized test generator BTRTG(Binary Translation Random Test Generator) for binary translation. Unlike traditional randomized tests, this paper summarizes the translation mechanisms of the binary translator. Based on the mechanisms, we design a test generator. This generator is driven by a template and can flexibly deal with various test scenarios. The generator automatically generates 32-bit x86 assembly according to the test template written by the test engineer. The generated tests can be used to verify the correctness of single instruction translation in the early stage of development, and also verify optimization translation in the middle and late stages of development.

2. Four tests based on four test scenarios. To ensure that the tests are represen-

tative, we select two typical test scenarios in single instruction translation and two in optimization translation. New errors were found in three of the tests. The other one improved the code coverage.

3. A lightweight but efficient test suite. The test suite contains both unit tests and tests for optimization translation. It is found that the code coverage of each source code module is relatively balanced. Besides, the test suite detected errors that were not detected in other test suites.

Keywords: Binary Translation, Randomized Test, Test Scenario, Code Coverage

目 录

第 1 章 引言 …… 1
1.1 应用软件的跨架构兼容 …… 1
1.2 二进制翻译技术概述 …… 2
1.3 二进制翻译器的开发与测试 …… 5
1.4 本文主要工作 …… 6
1.5 论文组织结构 …… 7
第 2 章 相关工作 …… 9
2.1 二进制翻译器开发相关工作 …… 9
2.1.1 XQM 简介 …… 10
2.1.2 XQM 的测试重点 …… 13
2.2 二进制翻译器测试相关工作 …… 15
2.2.1 传统测试套装 …… 15
2.2.2 随机化测试相关背景 …… 18
2.3 本章小结 …… 21
第 3 章 BTRTG 的设计与实现 …… 23
3.1 概述 …… 23
3.2 指令生成引擎 …… 24
3.2.1 辅助指令插入机制 …… 26
3.2.2 内存操作数的设计 …… 27
3.2.3 控制转移逻辑 …… 29
3.3 测试知识 …… 32
3.4 模板语言 …… 33
3.5 验证机制 …… 35
3.6 本章小结 …… 37
第 4 章 典型测试场景与测试 …… 39
4.1 浮点运算的计算正确性验证 …… 39
4.1.1 边界值测试 …… 39
4.1.2 精度测试 …… 41
4.2 操作数处理的正确性验证 …… 42
4.3 栈操作局部优化的验证 …… 44
4.4 Flag-Pattern 全局优化翻译的验证 …… 45
4.5 本章小结 …… 48

第 5 章　测试集综合评测 ····· 49
5.1 代码覆盖率测试 ····· 49
5.2 测试方法 ····· 50
5.2.1 硬件配置 ····· 50
5.2.2 测试集 ····· 51
5.2.3 目标程序 ····· 52
5.2.4 编译选项 ····· 53
5.2.5 测试过程 ····· 53
5.3 覆盖率提升 ····· 53
5.4 纠错能力 ····· 55
5.5 本章小结 ····· 55
第 6 章　总结与展望 ····· 57
6.1 本文工作总结 ····· 57
6.2 未来工作展望 ····· 57
参考文献 ····· 59

图形列表

1.1 虚拟机层次结构 · · · 3

2.1 二进制翻译器运行流程 · · · 9
2.2 QEMU 和 TCG · · · 11
2.3 cpu_loop 函数中 QEMU 例外处理 · · · 12
2.4 cpu_exec 中 QEMU 例外捕捉 · · · 12
2.5 QEMU 和 XQM 的函数接口 · · · 13
2.6 XQM 的指令翻译过程 · · · 13
2.7 XQM 代码模块 · · · 14
2.8 DBT5_UT 的内部原理 · · · 15
2.9 GNU C 库测试样例 · · · 16

3.1 BTRTG 概览图 · · · 24
3.2 X86 架构指令格式 · · · 25
3.3 指令生成流程 · · · 25
3.4 FPTAN 指令隐含操作数 ST0 的初始化 · · · 26
3.5 除零的规避 · · · 27
3.6 地址和内存操作数的生成 · · · 29
3.7 JMP 指令跳转地址合法性分析 · · · 30
3.8 JMP、JCC 指令生成方案 · · · 31
3.9 LOOP/LOOPcc、CALL 指令生成方案 · · · 32
3.10 FSIN 指令测试知识决策树 · · · 33
3.11 SUB 指令单元测试片段 · · · 35
3.12 Flag-Pattern 优化翻译测试片段 · · · 35
3.13 内存操作数的验证 · · · 36
3.14 生成的测试程序的编译流程 · · · 37

4.1 FADD 加法规则 · · · 39
4.2 FADD 指令边界值测试知识 · · · 40
4.3 FADD 边界值测试测试结果，× 为失败，√ 为通过 · · · 41
4.4 X86 操作数处理相关代码的覆盖率对比 · · · 44
4.5 TEST_JZ 模式 · · · 46
4.6 BEQ 指令替换 TEST_JZ 模式 · · · 46

4.7 Flag-Pattern 测试模板 · 47
4.8 Flag-Pattern 优化翻译的覆盖率对比 · 48

5.1 某 C 代码片段 · 49
5.2 BTRTG 和 DBT5_UT 的覆盖率对比 · 54
5.3 BTRTG 和 SPEC CPU2000 的覆盖率对比 · 54

表格列表

3.1 X86 地址编址模式 28

4.1 通用型指令错误类型分布情况 44
4.2 标志位延迟计算的模式/尾 47

5.1 DBT5_UT 测试用例分布 51
5.2 SPEC CPU2000 基准测试程序 52
5.3 目标程序分组 52

第 1 章 引言

1.1 应用软件的跨架构兼容

指令集架构（Instruction Set Architecture，简称 ISA）[Hennessy 等, 2011] 是对计算机硬件的抽象。它定义了计算机支持的硬件资源和机器码的行为，是硬件为软件提供的接口。一个可执行的用户程序的行为和它所属的指令集架构紧密联系。当前主流的指令集架构有 X86、ARM 和 MIPS 等。指令集架构的发展常常需要和应用场景的变换和芯片设计工艺的进步相结合，比如手机对低功耗的追求促使了 ARM 的崛起。

新指令集架构被开发出后往往没有配套的软件生态，如何兼容已有应用软件成为了处理器设计厂商需要面对的问题。在历史的发展过程中，全球出现过多家处理器设计厂商，仅面向个人电脑的 X86 架构芯片制造商就有 15 家。除几家巨型 CPU 制造商有能力独立研发一套指令集外，其它制造商大多采用已有的指令集架构设计芯片。采用已有的指令集架构可以直接吸收其相应的软件市场，更容易占据一份市场份额，但需要为此支付高昂的专利费。采用新指令集处理器厂商的制造商则会设法兼容已有软件。比如，Transmeta[Wikipedia contributors, 2021a] 使用二进制翻译器在 VLIW 架构上支持 X86 架构指令；苹果公司在其 PC 端处理器架构从 PowerPC 迁移到 X86，再到目前的 ARM 架构的过程中，先后开发两个版本二进制翻译器 Rosetta[Wikipedia contributors, 2021b] 用于兼容旧版本的软件。

我国 CPU 的发展也同样面临缺少软件生态的困难。过去很长一段时间，CPU 作为我国信息安全产业的核心技术，一直受制于人。国内新兴的软硬件公司一直在推广国产处理器和基于国产处理器的面向各应用场景的国产软件解决方案。处理器的生态建设决定了处理器的生命力。国产处理器起步晚，市场占有率低。如何将大量已有软件运行在国产处理器上，从而占据更多市场份额，是制造商急需解决的问题。

软件的移植直观上有两种方式：1）将已有软件的源码重新编译；2）在虚拟机上运行已有软件。前者涉及的难点包括软件源码难以获得，老旧软件的编译环境难以复原以及重新编译大量软件的人力成本很高等。后者使用一个软件来

模拟应用程序的执行环境似乎是一劳永逸的方案，但是性能的损失会带来软件使用体验的下降。当前国际上的商用二进制翻译器的性能损耗一般不高于 30%。Rosetta 2 的运行效率达到本地的 78%-79%，总体上是用户可以接受的。

在跨架构的虚拟机中，二进制翻译器是跨平台运行程序的最佳软件实现方式。以软件的形式模拟程序运行环境的方式有两种。(1) 解释器，解释器的工作原理是每次读取源程序的一条指令并按照源 ISA 语义解释执行，模拟其对硬件数据读写的效果。这样当源程序的指令模拟执行完毕便相当于源程序执行结束。其缺点是指令膨胀度高，模拟效率很低。(2) 二进制翻译器，二进制翻译器将一块原架构指令序列转换为本地架构指令序列并在本地机器上运行。翻译器以软件的方式模拟了源架构硬件状态，翻译的过程引入了高效的翻译规则使得其模拟性能损失较少。

1.2 二进制翻译技术概述

二进制翻译技术是一种将源架构指令重编译成目标架构指令的技术，源架构和目标架构可以相同。同架构的指令翻译广泛用于代码的调试、检测、分析和性能优化等方面，而不同架构之间的二进制翻译一般用于跨架构的虚拟化中。

在代码分析方面，通过对源二进制程序进行修改、插桩，可以实现功能丰富的二进制码分析工具。这其中具有代表性的有 DBI（Dynamic Binary Instrumentation）框架 Valgrind[Nethercote 等, 2007]，它通过动态二进制插桩的方式来捕获程序的运行情况，与其配套的 DBA（Dynamic Binary Analysis）工具根据这些信息分析程序的行为，如检测内存溢出等。

在代码优化方面，二进制翻译技术更是扮演着软件加速器的作用。JVM[Lindholm 等, 2014] 引入了 JIT（Just-in-Time）编译器将 Java 字节码编译成本地机器码并缓存，优化后的本地码的执行效率比字节码的解释执行更高。Dynamo[Bala 等, 2000] 则是用于特定硬件平台 PA-8000 上加速一般应用软件的程序，它通过搜集并优化热点代码的方式透明地加速软件的运行。抛开特定硬件，在内核中，相关方案 [Kedia 等, 2013] 被提出用于优化系统上应用软件的执行。

二进制翻译在虚拟化中的应用，解决了软件的跨平台兼容。图 1.1为二进制翻译器，源架构，目标架构的层次结构图。其中，被虚拟的平台称为客户机（Guest），本地的平台称为宿主机（Host），而二进制翻译器是运行在宿主机上的

一层软件。例如，在 MIPS 机器上模拟 X86 架构程序的运行，则 MIPS 机器为宿主机，X86 机器为客户机。

图 1.1 虚拟机层次结构

Figure 1.1 The Virtual Machine Hierarchy

二进制翻译器按照模拟的层次分为应用级模拟和系统级模拟。应用级模拟支持一个客户机应用程序在宿主机上运行，它通过翻译用户级指令和模拟系统调用等系统功能来实现。系统级模拟以软件的方式模拟了整个客户机的裸机硬件资源，一个客户机操作系统可以在不做任何修改的情况下运行。系统级模拟往往需要实现一款处理器的所有指令，然而如用供软件于获取处理器配置细节的 CPUID 等特权级指令的指令语义复杂，导致其翻译出的指令膨胀率较高。此外，大量的外设和浮点异常的检查、MMU（Memory Management Unit）的软件实现等相关硬件机制的模拟使得系统级翻译器的代码量非常庞大。进程级态模拟的结构更为简单，但由于在操作系统领域里对于一个程序边界的划分是比较模糊的，这增加了进程级模拟的复杂性。

二进制翻译器根据翻译和执行的时间关系又分为静态翻译和动态翻译。静态翻译是指提前将程序翻译成本地代码，翻译一遍后便可获得一个可永久运行的本地程序。动态翻译则采用翻译和执行交叉进行的方案，每次运行目标程序时都需重新翻译。静态翻译中翻译的过程和运行的过程相互独立，因此可采取一些耗时但高效的优化方法优化代码的生成过程，例如 rev.ng[Di Federico 等, 2017] 使用 LLVM 编译器作为后端，通过编译优化技术提升生成代码的质量。静态翻译的局限性也很明显：无法处理自修改代码；无法处理动态链接；间接跳转的实现非常复杂。因此，纯静态翻译的翻译器数量比较少。动态翻译采用实时翻译的方式很好地避开了静态翻译的局限性，然而其在进行翻译的优化时往往需要在优化时长和优化效果之间进行折中。为结合动、静态翻译两者的优点，一些以动态翻译为执行主逻辑、静态翻译为优化辅助的翻译方案被陆续提出。例如，

HQEMU[Hong 等, 2012] 单独创建一个线程以激进的手段优化热点代码，然后替换掉动态生成的相对低效的代码。这种通过抓起并优化热点路径（Hot Trace）代码的方式提升了程序整体运行速度。

从上世纪八九十年代开始，面向不同场景的二进制翻译器不断被开发出来。其总的发展趋势是由两个架构之间的翻译转向多架构可重定目标的翻译。其中一些特征明显，具有代表性的二进制翻译器按照其相关文献发表的时间顺序列举如下：

1）FX!32[Hookway 等, 1997] 由 Digital 公司开发用于 Alpha 平台支持 32 位的 X86 应用程序。它提出了一些经典的针对 X86 架构特性进行优化翻译的方法，比如：对于状态标志位的计算采用按需计算的惰性计算方案；使用影子栈来加速 RET 指令的执行。

2）DAISY[Ebcioğlu 等, 1997] 由 IBM 开发用于 VLIW 架构支持 PowerPC 的系统级虚拟化。其翻译生成的本地指令具有较好的并行性，同时又能以软件的方式正确处理精确例外。

3）UQBT[Cifuentes 等, 2000] 由昆士兰大学开发，它是少有的静态二进制翻译器，也是早期支持可重定目标的翻译框架。它使用一种规范语言描述 ISA 架构，使用多层中间语言翻译指令。通过规范语言描述 ISA 架构在后来的全系统模拟的可重定目标二进制翻译器框架 Captive[Spink 等, 2020] 中也被使用到。

4）CMS[Dehnert 等, 2003] 由 Transmeta 开发用于 VLIW 架构支持 X86 架构的系统级模拟。它使用到了硬件来辅助软件的翻译。软件翻译代码时进行一些猜测来简化代码的翻译过程，而这种猜测有小概率引发错误。CMS 在硬件上维护一套 X86 寄存器的影子映射，执行时只修改影子寄存器中的值，一旦出现错误，便可恢复到之前状态。影子寄存器中的修改在一次翻译结束后才进行提交，而如果在翻译过程中出错，随时都可以回退到安全状态。

5）QEMU[Bellard, 2005] 是一个开源的可重定目标的二进制翻译器。它由 Fabrice Bellard 个人于 2005 年开发以来，吸引了大量的开发者参与开发。其通用性和在外设、各类调试工具等方面的丰富性使得其成为学术界和工业界进行相关研究的主要选择之一。

1.3 二进制翻译器的开发与测试

功能的正确性是二进制翻译器重要评价标准之一，开发一个正确（且高效）的二进制翻译器并非易事：

1. 翻译器涉及到两种处理器架构之间的功能转换，对开发者的系统知识要求很高。翻译规则的制定，寄存器的分配，内存的布局设计，异常的捕捉和处理等都需要开发者熟悉两种架构的硬件规范和系统特性。

2. 指令众多且指令语义多变，制定正确的翻译规则难度大。根据文献 [Dasgupta 等, 2019]，X86-64 Haswell 指令集架构手册中的用户级指令共包含了 774 种指令助记符和 3155 种指令变种。开发者在设计翻译规则时需要保证每种指令变体的正确性，非常困难。

3. 高效的优化方案涉及到多指令的翻译，指令的翻译之间相互影响，增加了翻译的复杂性。

4. 验证二进制翻译器的正确性非常困难。二进制翻译器的正确性并非是目标程序的执行结果和在原客户机上的结果完全一致，实际上程序的运行结果可能受系统的配置以及外部环境所影响。不过可以断定的是，二进制翻译器的运行结果必须是源架构机器上执行目标程序时可能出现的结果。

为保证翻译器功能的正确性，可以对源码进行形式化验证或测试。翻译器在开发的过程中，迭代速度非常快，形式化验证难以进行。测试作为一种可操作性更强的手段，能在尽可能保证代码的正确性下支持代码的快速迭代。测试与迭代结合在一起形成一套高效的开发方案。

持续集成 [Fowler 等, 2006] 作为一种软件开发实践受到了团队开发者们的广泛使用，它使用迭代与测试相结合的方式在开发流程上确保了软件的正确性。持续集成的主要内容是团队中的成员频繁向项目集成自己的代码，每次集成都有一个后台工具对代码进行自动构建和测试，及时检测错误。该方式帮助开发团队更快更好的合作。持续集成至少有以下三个优点：（1）易于维护代码仓库，开发新模块；（2）暴露错误代码；（3）快速开发。总而言之，其最大的好处是降低开发风险。它能保证项目在一定时间内收敛于一个可靠的点，而不是反复的错误和遥遥无期的调试。

持续集成中重要的一点就是有足够且可依赖的测试，当前已有的测试可依赖性差。可依赖的测试不仅能对二进制翻译器的源代码有很高的覆盖度，而且需

要有执行时间短、易于定位和调试错误等特点。当前，针对二进制翻译器的测试非常少，且大多测试不充分。而一些大型的测试又多是性能相关的测试，调试起来非常困难。因此，开发一条针对二进制翻译的测试方案成为急需解决的问题。

为了实现二进制翻译器的测试方案，首先应当研究二进制翻译器本身的特点，包括其单条指令翻译和指令优化翻译中的薄弱环节。总结出其测试重点后，再针对性提出解决方案，比如一般单指令翻译可以通过对其操作数数值进行随机化生成来验证，优化翻译则可以随机生成相应模式的指令组合进行验证。本文试图从整个系统的翻译正确性出发，设计出能面对各种翻译场景的测试方案。

1.4 本文主要工作

本文主要研究面向二进制翻译的随机化测试方案。本文的测试目的是：提升二进制翻译器稳定性和可靠性；加快开发速度，节省人力资源。本文采用随机化测试的理由是随机化能为持续集成开发提供源源不断的测试样例，生成的测试包含了大量通过人工编写的方式难以覆盖到的特殊程序运行行为。本文的主要贡献如下：

1. 设计并实现了针对二进制翻译的随机化测试生成器 BTRTG（Binary Translation Random Test Generator）。与传统的随机化测试不同，本文根据二进制翻译器的翻译特点，设计了一套以测试模板为引导，能灵活面对各类测试场景的测试生成器。BTRTG 搭配对应的函数库和编译脚本，共同组成一套完整的自动化测试生成工具。该生成器根据测试工程师编写的测试模板自动生成 32 位 X86 汇编程序，既能在开发早期用于验证单指令翻译的正确性，也能在开发中后期用于验证优化翻译的正确性。

2. 选取了四个测试场景并使用 BTRTG 进行了针对性测试。为保证测试具有代表性，本文选取了单指令翻译和优化翻译等典型翻译方式下的测试场景各两个。其中，三个实验均发现了翻译器中的错误，另一个实验提升了代码的测试覆盖率。

3. 生成了一套轻量但高效的测试集。该测试集包含了部分单指令翻译的测试和优化翻译的测试。本文通过代码覆盖率测试发现，该测试集对翻译器源代码各模块的覆盖率都比较高，总体分布均匀。此外，该测试集测出了传统测试集没有测出的新错误。

1.5　论文组织结构

本文共分为六章。第一章为本文的引言部分，主要介绍了二进制翻译技术产生的历史渊源以及测试二进制翻译器的目的和意义。第二章前半部分介绍了二进制翻译器开发的相关工作，分析了二进制翻译器的测试重点；后半部分介绍了二进制翻译的传统测试集和测试方法，并分析了其优缺点。第三章介绍了本文实现的测试生成器 BTRTG 的设计方案，包括它的工作原理、模块组成和各模块的实现方式等。第四章针对四个测试场景进行了四个测试实验，在测试翻译器的同时也阐述了 BTRTG 的使用，并对其功能进行了一个多角度的评测。第五章首先阐述了本文中覆盖率测试的测试方法，然后将本文生成的轻量级测试集和另外两个传统测试集进行了覆盖率的对比，展示了新测试集测试的充分性。第六章总结了全文的工作，并分析了当前工作的不足和下一步工作方向。

第 2 章　相关工作

本章主要介绍应用级二进制翻译器的开发及测试相关工作。开发相关工作基于龙芯中科的 XQM 虚拟机展开介绍，本章主要分析其组成部分，提炼测试要点。测试的工作列举了传统的测试集或测试方案，本章分析了其各自的局限性。本章以已有测试为基础，结合二进制翻译器本身的翻译特征，探索随机化测试生成器的设计思路和设计目标。

2.1　二进制翻译器开发相关工作

图 2.1为应用级二进制翻译器的运行流程简图。二进制翻译系统启动后先对模拟的硬件资源做基本的初始化，之后开启 CPU 的模拟直到退出。它将待翻译的程序装载至自身所在地址空间，然后分析其文件头信息并找到程序入口地址，之后反复进行读指令、反汇编、翻译和执行等流程。

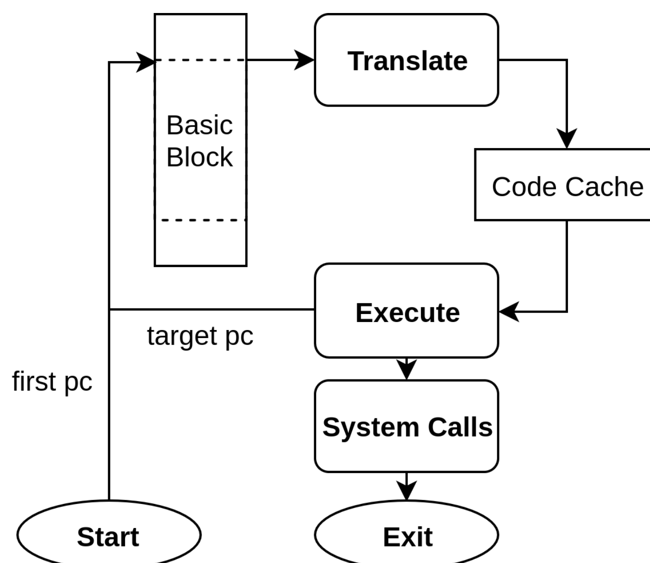

图 2.1 二进制翻译器运行流程

Figure 2.1 The Running Flow of the Binary Translator

虚拟 CPU 的模拟一般通过一个大循环实现，重复翻译和执行生成的代码直至客户程序退出。翻译时一般以一个基本块（Basic Block）为翻译的基本单位。基本块即为以控制传输指令结尾的连续多条指令组成。翻译生成得到的 Host 指令一般需要一个代码缓存区（Code Cache）缓存，后续重复执行该基本块时可省

去翻译的开销。翻译的过程实质上是组合 Host 机器指令实现被翻译指令的语义的过程。翻译和执行过程之间通过保存和恢复客户机上下文来进行切换。

生成的代码在执行的过程中和二进制翻译器进行交互，虚拟硬件资源也在此时被访问。以下列举应用级模拟中的两种交互方式：

1. 处理系统调用。系统调用的接口和硬件架构，操作系统密切相关。不同操作系统的系统调用表一般不同，如 Windows 不支持 Linux 下的 sbrk、brk 系统调用。相同操作系统下，系统调用的实现又由其所在架构的应用程序二进制接口（ABI）决定。因此在模拟系统调用时，翻译器内部会在宿主机平台上使用高级语言实现该系统调用的处理函数，然后生成保存系统调用号等参数的指令和一条跳向系统调用处理入口的跳转指令来模拟整个系统调用处理过程。

2. 检查异常。异常是处理器在执行指令时检查到了预定的条件时所产生的同步事件，如除零异常。一旦检测到例外，翻译器需要将中断控制器相应的状态位置位以触发中断。如在除零异常中，需要在翻译除法指令时生成判断操作数是否为零的检测指令。

二进制翻译器的代码组成十分复杂，将二进制翻译系统分模块分析，抓取测试关键点能提高测试的效率。直观上，指令集的翻译是二进制翻译器的“骨干”，虚拟硬件资源如内存、中断机制、各种外设等是围绕在“骨干”周围的“皮肉”。一般来说，使用一种甚至几种测试手段都无法保证整个系统的正确性。实际上，除一些大型公司或者开源社区有能力从零开始开发一个完整的虚拟机，多数开发者都是基于已有的开源虚拟机做二次开发。这些二次开发的虚拟机，开发者往往只需要保证修改的模块的正确性即可。

本文中的二进制翻译器 XQM 即为基于 QEMU 二次开发的虚拟机。以下以 XQM 为例，分析其运行原理和代码组成。

2.1.1 XQM 简介

XQM（X86 to MIPS Based on QEMU）是龙芯中科为在 MIPS 平台兼容 X86 应用程序而开发的二进制翻译器，目前其模拟效率超过 50%，稳定性需要进一步提升。

XQM 基于开源虚拟机 QEMU，QEMU 的通用性和健壮性使得其成为虚拟机开发者的首选。QEMU 采用二进制翻译的方式，支持在多种宿主机平台上模拟出

多种客户机平台。庞大的开源社区力量不断丰富该系统，当前 QEMU 几乎支持所有常用外设的模拟。学术界基于 QEMU 展开了一系列关于二进制翻译器优化技术的相关研究，这为开发高效的二进制翻译器提供了理论参考。此外，QEMU 的用户量巨大，庞大的用户群体为 QEMU 报告了大量错误，进一步加强了其安全可靠性。

生成出的本地指令的质量影响到整个翻译系统的效率，XQM 重写了 QEMU 中的翻译过程。QEMU 使用了 TCG（Tiny Code Generator）的 IR（Intermediate Representation，中间表示）作为宿主机指令集架构和客户机指令集架构之间翻译的桥梁。一种架构的指令先通过客户机前端翻译成 TCG 中间码，然后再由宿主机后端翻译成另宿主机指令，如图 2.2 所示。以中间语言为媒介的翻译方式利于 QEMU 对多平台支持的扩展。然而，通过两次翻译带来的指令膨胀度非常高，在 MIPS 上运行 X86 应用程序时指令膨胀约为 1：7，性能损失巨大。为了降低翻译过程中指令的膨胀度，XQM 采用了直接将 X86 指令翻译成 MIPS 指令的方式。

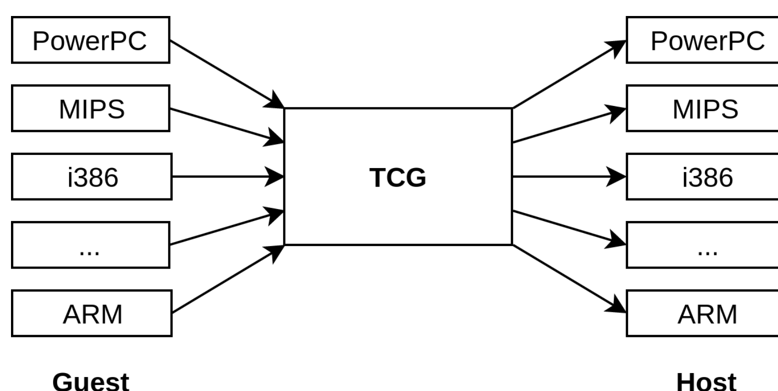

图 2.2 QEMU 和 TCG

Figure 2.2 QEMU and TCG

QEMU 中虚拟 CPU 的主循环逻辑如代码 2.3。cpu_exec 负责代码的翻译执行，当有例外被触发后，其返回例外编号 trapnr，之后虚拟 CPU 调用相应的例外处理程序进行处理。

虚拟 CPU 的执行逻辑 cpu_exec 如图 2.4。它使用跳转点设置函数 sigsetjmp 设置一个长跳转目标地址，此后碰到以下三类情况会导致程序跳至此处：（1）收到了宿主机的信号；（2）指令执行过程中检测到例外，主动陷入；（3）模拟系统调用指令直接跳转到此处。cpu_handle_exception 函数用于判断 CPU 是否需要处

理此例外。如需要处理，返回例外类型，否则依次指令查找基本块（查找过程中翻译未翻译的基本块），执行基本块等操作。

```
for (;;)
    trapnr = cpu_exec(...);
    switch(trapnr)
        case 0x80:
            ret = do_syscall(...);
            break;
        case EXCP_xx:
            ...
    endswitch
endfor
```

图 2.3 cpu_loop 函数中 QEMU 例外处理

Figure 2.3 QEMU Exception Handling in cpu_loop Function

```
/* prepare setjmp context for exception handling */
if (sigsetjmp(cpu->jmp_env, 0) != 0)

while (!cpu_handle_exception(cpu, &ret))
    ...
    tb = tb_find(...);
    cpu_loop_exec_tb(...,tb,...);
endwhile

/* return the trap number */
return ret;
```

图 2.4 cpu_exec 中 QEMU 例外捕捉

Figure 2.4 QEMU Exception Capture in cpu_exec Function

XQM 通过重新实现图 2.5中 QEMU 的三处接口来改变 QEMU 的翻译逻辑。这三个接口的功能如下：

1. 截取代码生成接口，QEMU 在翻译基本块时转到 XQM 的翻译流程进行翻译。

2. 修改基本块的链接逻辑，和 XQM 的基本块退出过程相适配。

3. 截取运行本地代码的接口，当前从客户机切换至本地代码时执行的本地码为 XQM 生成的代码。

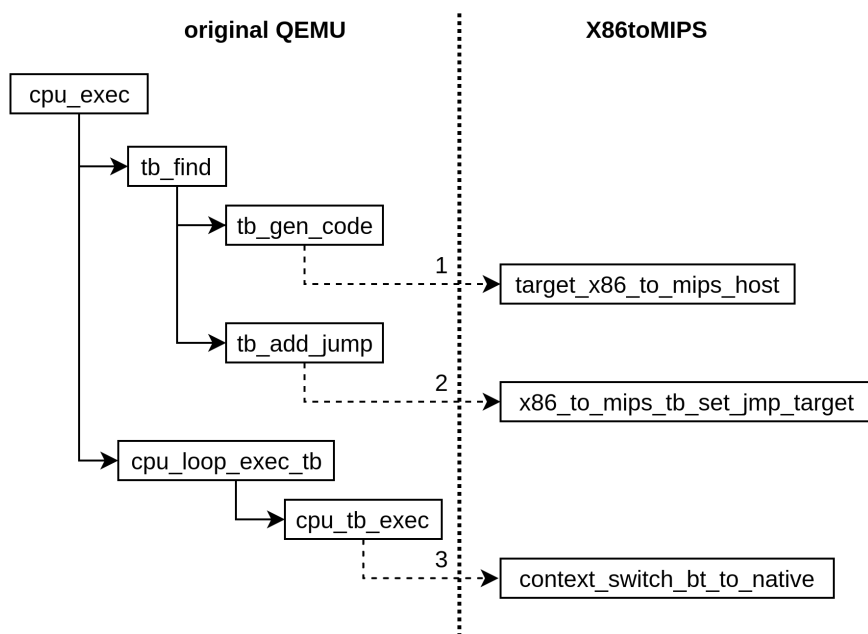

图 2.5 QEMU 和 XQM 的函数接口

Figure 2.5 Interfaces between QEMU and XQM

2.1.2　XQM 的测试重点

XQM 的整个翻译过程如图 2.6。它由源架构反汇编器、翻译程序和目标架构汇编器组成，中间定义了两种指令表示结构缓存中间码。

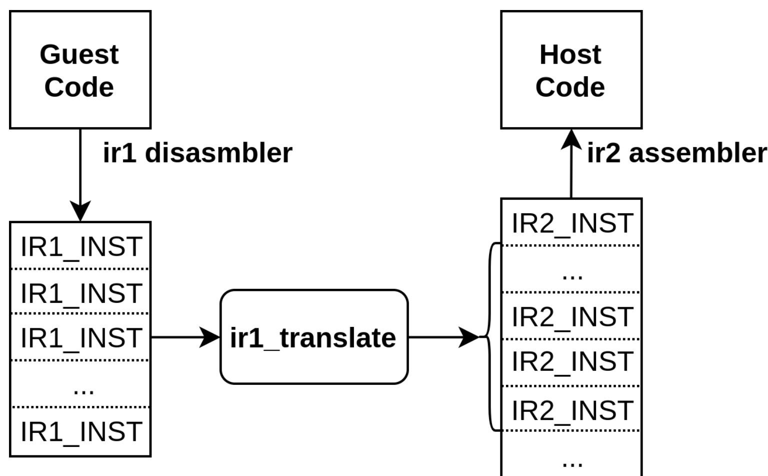

图 2.6 XQM 的指令翻译过程

Figure 2.6 Instruction Translation Process of XQM

XQM 使用 Capstone 作为源架构反汇编器，将二进制格式的 X86 代码反汇编保存在 IR1_INST 结构体中。一个基本块反汇编完便得到一个 IR1_INST 结构

体数组。之后翻译器根据 IR1 的指令类型选用相应的指令翻译函数。翻译函数的翻译规则由开发者人工确定，众多的翻译函数是翻译器的错误高发区。翻译好的 MIPS 指令暂存在 IR2_INST 结构体中，本地指令生成结束后再使用汇编器生成机器码。由于翻译过程中生成的一些程序 Label 对应的地址偏移需要在所有 MIPS 指令生成后才能被计算出，故此处采用一个 IR2_INST 中间结构而不直接生成 MIPS 机器码。

站在测试人员的角度可以将 XQM 划分成图 2.7所示的几个模块。它包含单指令翻译和优化翻译分为两大部分，其内部又细分几个组成成分。这么划分的依据有以下几点：

1. 单指令翻译函数只负责一条指令的翻译，其正确性独立于其它指令翻译函数。优化翻译涉及到多条指令的组合翻译，其中指令之间的翻译过程相互影响。因此，单条指令翻译和优化翻译被分为两大类。

2. 原指令的语义由多个子功能组成。单条指令的翻译是一个使用本地指令组合出目标指令各项语义功能的过程，比如计算 EFLAGS 寄存器中的各状态位、检查指令的例外等。各个子功能的模拟相对独立，因此需要划分成不同模块以分别进行测试。

3. 优化翻译由多个独立的优化算法组合而成。这些算法各自的优化原理不同，需要分开进行针对性测试。

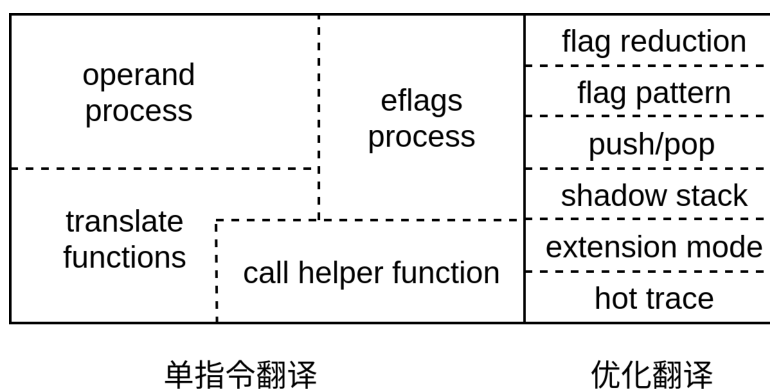

图 2.7 XQM 代码模块

Figure 2.7 The Modules of XQM

2.2　二进制翻译器测试相关工作

二进制翻译的测试工作在二进制翻译器的开发过程中起到了加快开发速度、提升代码质量的重要作用。传统二进制翻译开发使用的测试套装主要来自于开发者手写的测试和其它与测试相关的项目。这些测试套装对二进制翻译器不具有针对性，或多或少存在其局限性，在依赖度或和错误定位能力能方面表现不突出。检测错误的能力是衡量一个测试套装的重要标准。二进制翻译系统的复杂度很高，手写的测试数量有限，难以穷尽各种运行情形，查错能力不足。

2.2.1　传统测试套装

2.2.1.1　单元测试集 DBT5_UT

DBT5_UT 是中科院计算所国家重点实验室二进制翻译组编写的针对单指令翻译函数的一组单元测试集。该测试集共包含 332 个测试程序，其中通用型指令，浮点运算指令，SSE 系列扩展指令各有 115，74，143 个测试。

DBT5_UT 中通过在 C 程序中使用嵌入式汇编的方式测试单条指令，每个文件测试一条指令。它使用数组保存了操作数的边界值，根据指令操作数的个数生成操作数初始值的组合作为嵌入式汇编代码块的输入。嵌入式代码块结尾处将指令的目的操作数读出并和标准输出（即 X86 机器上该位置的正确输出）进行比较，从而判断指令的执行是否正确。图 2.8为 DBT5_UT 的内部结构。

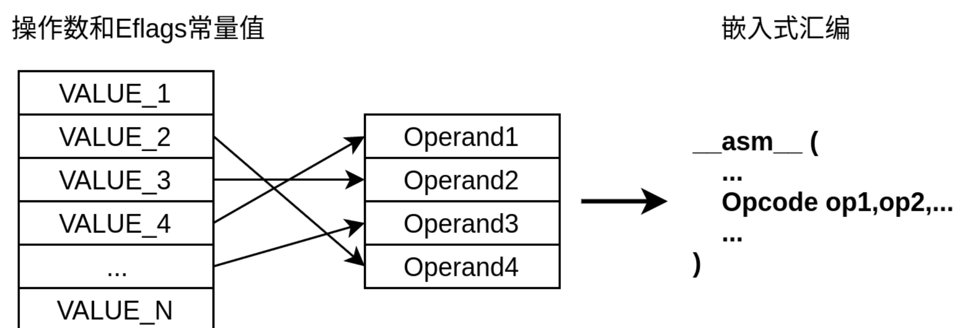

图 2.8 DBT5_UT 的内部原理

Figure 2.8 Internal Principles of DBT5_UT

该单元测试的特征是使用了大量的用宏定义包装的嵌入式汇编来处理不同指令的不同操作数组合格式。每个测试的输出信息比较详细，易于调试。随着单元测试的增加，其对源代码的整体覆盖率比较高。然而，它存在以至少下 4 个缺点：

1）扩展性差。添加一个新的单元测试需要理解整个代码框架。

2）缺少丰富的代码组合。被测试指令封装在一块固定的代码中，没有复杂的前后文环境，这导致其对优化相关代码和衔接相关代码的测试不足。

3）对操作数编码格式的覆盖不足。比如内存操作数：X86 指令中内存地址的组成一共有 6 种格式，在嵌入式汇编中指定变量为内存操作数，编译器往往只使用到一到两种格式。又如受测试的指令没有使用 6 个段寄存器。

4）对操作数数值的覆盖不足。在测试程序中提前设定的边界值数量有限，实际上无法穷尽各种情况，另外没有考虑非法的操作数数值，比如除零、除法溢出之类。

2.2.1.2 GNU C 库测试套装

GNU C 库是 GNU 计划实现的 C 标准库 [Loosemore 等, 1993]。它为 GNU/Linux 系统提供了关键的 API，而 GNU C 库测试套装则用于测试这些 API 的功能正确性。GNU C 库实现了多种接口标准要求的全部或部分功能，如 POSIX、ISO C11 和 BSD 等。其核心是 C 和 C++ 编程语言的应用编程接口。这些接口的功能覆盖输入输出，字符串处理，数学计算，内存分配，系统调用及信号处理等方方面面。

```
/* This is your test case 'main' method.  */
static int
do_test (void)
{
  /* Test goes here.  */
}

#include <support/test-driver.c>
```

图 2.9 GNU C 库测试样例

Figure 2.9 A Test Case of GNU C Library

GNU C 库提供的测试套装由多个单元测试组成，单个测试的结构如图2.9 所示。do_test 函数内部罗列一系列该测试的测试样例。每个测试外部有一个测试驱动程序。驱动程序创立了一个包含时间限制、运行参数等通用信息的测试配置结构，然后根据该结构调用 do_test 主测试函数进行测试。

该测试套装用作二进制翻译的测试的理由有以下几点：（1）高级语言编译成可执行程序都需要链接 C 标准库，库函数运行的正确性是保证整个可执行程序正确性的关键；（2）C 标准库覆盖了应用程序行为的方方面面，从数学计算到进程间的交互等，从工程角度上看是翻译器完备性的有力依据；（3）GNU C 库开源，其测试套装由全球开发者经几十年贡献而成，测试效果好；（4）部分数学运算测试，其测试的函数实际上编译成了对应的浮点运算指令，比如 sin，log 等。

然而，在二进制翻译测试的角度上看，GNU C 库具有以下几点不足：（1）其对单条指令的测试不足，比如：它不包含对定点运算的单独测试；（2）不易定位错误。测试由高级语言经编译器编译而成，在机器码级别上很多测试代码复杂度高，比如 malloc 函数测试等，（3）不同测试的失败可能由相同的错误导致。

2.2.1.3　小型集成测试 Coremark、BusyBox 等

当前还不存在公开的完全面向二进制翻译的集成测试。因此，在开发二进制翻译器的过程中选取常用的应用程序作为集成测试是一个迁就但可行的方案。比如使用 Linux 系统常用命令套装 Busybox，性能测试程序 Coremark 等。这些程序能触发单元测试所不能触发的错误。

Coremark[Gal-On 等, 2012] 是一套用于评估嵌入式系统中 CPU 性能的工业评测标准。它的设计目标是仅使用一个数字基准分数便能反映出 CPU 的性能。该测试集中包含了四个在嵌入式系统中的常用算法：（1）列表处理，包括查找和排序等；（2）矩阵运算，包含一些常用矩阵操作；（3）状态机，判断一个输入流包含的数字是否合法；（4）循环冗余验证算法（CRC）。Coremark 被选择二进制翻译器测试的理由是它开源，轻便，小巧易于调试，且在开发前期能够通过其基准分对翻译器的性能有一个早期的评估。

BusyBox[Wells, 2000] 是一个将常用 Unix 程序封装在一个可执行文件里的软件。这些可执行程序涵盖了 300 多个 Unix 常用的 Shell 命令，比如 clear，ls，cp，cut 等。它的设计目的是在嵌入式系统上以少量内存启动系统。二进制翻译器在通过 BusyBox 的测试之后，便具备了比较基础的使用价值。

2.2.1.4　大型集成测试集 SPEC CPU 系列

SPEC CPU 系列测试基准套装由 SPEC（Standard Performance Evaluation Corporation，标准性能评测组织）开发，用于评测计算密集型情况下计算机系统的

性能。它收集了不同领域的典型应用程序，能较全面地评测计算机三个方面的性能：CPU，内存架构以及编译器。XQM 的宿主机为 32 位 X86 架构，因而本文使用和其时代比较接近的 SPEC CPU2000 进行测试。SPEC CPU2000 选用了 12 个整点测试程序和 14 个浮点测试程序，根据文献 [Phansalkar 等, 2005] 中统计各程序在 Ref 输入集下平均动态执行指令条数约为 2300 亿。其输入集根据大小分 Test，Train，Ref。考虑到 Ref 输入集每次执行时间过长，Test 输入集测试不够充分，本文选用 Train 输入集进行测试。

由于 SPEC CPU2000 注重于性能测试，并不能很好地适用于二进制翻译的测试。在实际使用过程中，其暴露了以下缺陷：

1）指令执行流长，难以定位和调试错误。其测试用例为了充分挖掘系统的性能，运行时间长，因此实际错误发生后可能又经过大量指令流的执行才能被观测到。所以，重现错误以及定位都需要较长时间的跟踪，简单的错误也只能进行缓慢的调试。

2）测试程序的指令种类较少，整体指令覆盖率低。高级语言经编译器编译成可执行程序，其指令规律性较强，操作数格式单一。

2.2.2 随机化测试相关背景

随机化测试属于黑盒测试的一种测试手段，它通过生成随机且独立的输入来测试软件。随机化测试的优势非常明显，它能在短时间内生成大量的测试，覆盖到人工测试难以覆盖的场景。随机化测试也存在局限性，其测试的效率不如白盒测试高，且需要面临较多无法进行随机的情形。

直接针对二进制翻译器的随机化测试相关研究非常有限，目前仅 EATBit[Guo 等, 2014] 一项。而在与二进制翻译测试类似的编译器测试、芯片验证等领域中，相关研究则非常丰富。

（一）编译器测试方向的研究

文献 [Burgess 等, 1996] 针对编译器的代码生成和代码优化阶段的正确性测试做了相关研究。作者选用当时发展比较快的 Fortran77 语言的编译器，该编译器在持续集成新的优化机制。它通过实时计算出变量的值来保证程序不会发生溢出等错误。为了验证编译器的优化是否正确，它针对特定的优化算法设计出代码生成规则生成具有目标优化代码的程序。该生成器生成的测试程序具备自动

验证的功能，能在运行结束自动报告错误。最终作者选用两个主流的编译器进行测试，在不开优化和开优化的情况下编译和执行测试程序，发现了编译器的多处错误。

Quest[Lindig, 2005] 是一个 C 程序生成器，用于检查编译器在不同平台下对函数调用规范实现的正确性。程序的生成由用户使用 Lua 脚本语言程序进行引导，主要是设置好函数，参数的类型。程序的生成是类型驱动的。C 语言的语法是递归的，生成器通过限制递归深度来避免无穷的递归。Quest 生成的程序也带有自检测功能，它主要通过生成断言实现。断言的依据参数在传入函数前后不能发生改变。另外，Quest 生成的测试程序相比于 GCC 测试套装和 SPEC 测试在参数类型的分布上更加均匀。作者通过该方案在开源和商业性的 C 编译器中找出来 13 个新 bug。

Csmith[Yang 等, 2011] 是一个产生合法的随机 C 程序的软件。Csmith 首先生成结构体类型定义，然后从 main 函数开始自上而下生成 C 程序代码。它创建一个概率表用于选择语句，使用过滤函数来保证语句生成的合理性。为保证测试程序对错误报告的准确性，Csmith 生成的 C 程序都有唯一确定的运行结果。对于未定义或不确定的行为，比如整型的边界值运算定义区别、未初始化变量使用等，它在生成代码的过程中对这些行为进行分类并检测以避开生成有歧义的程序语句。Csmith 对 C 语言语句的覆盖范围齐全，包括常用控制流语句和指针、数组、结构体等。Csmith 目前发现了各类编译器中的错误超过 400 个。作者高质量地报告了这些 bug，报告中包含错误的一般外观，在编译器中的相关模块以及在各编译器版本中的分布情况。

（二）芯片验证方面的研究

IBM Research Lab 提出了 Genesys[Lichtenstein 等, 1994] 和 Genesys-Pro[Adir 等, 2004] 用于生成随机二进制指令和数据资源来验证处理器的功能正确性。后者在前者基础上做了改进。在 Genesys 之前，IBM 研发的验证平台只针对其特定的处理器架构。为支持任意架构，基于处理器模型的 Genesys 诞生。在 Genesys 中，处理器架构的描述部分和生成器引擎相互独立，添加新的架构支持只需要增加一份架构描述文件。Geneys-Pro 在基于模型的基础上又提出了三大改进：（1）具有编程语言般表达能力的模板语言；（2）适用于处理器描述的高级块构建语句；（3）将指令的生成过程转化成限制满足问题（CSP）的求解过程。该平台包

含了测试模板，架构相关的模型以及架构无关的生成引擎等三部分，为典型芯片验证系统。模板语言通过丰富的序列控制语句以及约束语句等灵活描述出成各种测试场景。

IBM Haifa Research Lab [Aharoni 等, 2003] 提出了 FPgen，它是一个针对浮点运算测试知识分类生成的生成引擎。浮点运算单元（FPU）的验证是处理器验证中挑战性最大的部分之一。浮点测试的边界条件繁多，浮点数值的表示多样，浮点运算指令功能繁多且运算逻辑复杂。FPgen 中定义了一套高级的指令模型描述语句称作覆盖模型，覆盖模型用于描述一个验证任务。它包括指令格式和相关限制以及输出信息。这些描述语法适用于浮点指令的表达，它能在控制生成器在指定范围内生成数据。在文献 [Aharony 等, 2011] 中，测试人员基于对 FPU 的大量验证经验，设计了一个浮点测试知识包 FPTK。该测试知识包对浮点边界值做了充分的分类。FPgen 根据 FPTK 对浮点操作数进行初始化，更高概率生成测试人员感兴趣的边界值。

（三）二进制翻译的随机化测试

为了弥补传统测试集的不足，郭辉提出了针对二进制翻译中的优化翻译的随机化测试方法 EATBit[Guo 等, 2014]。EATBit 一共包含了三点贡献。（1）提出了对指令分组的策略，每一组指令有具有相似特征的指令类型组成。生成器在组内随机挑选指令，组合不同的组可获取到符合某种模式的指令组合。比如，在 Flag-Pattern 优化中，为减少计算 EFLAGS 状态位，该优化将定义和引用 EFLAGS 位的前后两条指令翻译成一条指令。所以，在 Flag-Pattern 的测试设计中则将定义 EFLAGS 状态位的指令作为一组，引用 EFLAGS 状态位的指令作为另一组，按先后顺序进行生成便可得到 Flag-Pattern 优化翻译相关的测试。（2）提出选择寄存器的滑动窗口算法，避免了对操作数的无效修改。在随机生成的指令中，如果前后多条指令改写了同一个操作数，则仅最后一条指令的执行有效。这导致前序的指令的执行效果被覆盖，实际未被检测。针对该局限性，EATBit 中提出了滑动窗口算法，该算法即将目的操作数使用过的寄存器和其它寄存器分成不同组，每次申请新的源操作数寄存器时都尽量从缓存的目的寄存器中选取。（3）提出了使用指令封装和动态训练的方式来避免产生异常的测试。

该方案的局限性正如文章中所阐述的：内存的模拟使用一个静态的 64 字节的内存；不生成控制转移指令。这两点都和寻址相关，也是随机化中难以解决的

难点。操作数类型的丰富有利于检查翻译器对源平台操作数的处理代码的正确性。控制转移指令是翻译过程中基本块分块的条件，它们的存在可以使得对基本块切换之间的处理代码的检测更加充分。此外，该方案生成的指令种类非常有限，仅包括算术运算指令、逻辑运算指令和部分数据传输指令。

2.3　本章小结

本章介绍了二进制翻译测试的背景。首先，本章阐述了二进制翻译器的二次开发过程中修改核心翻译代码的方法，并按照功能将翻译部分代码进行了分类，提炼出测试重点。然后，本章介绍了传统的测试集及其缺陷，这是本文引入随机化测试的原因。紧接其后，本章介绍了编译器和芯片领域的随机化测试相关工作，并对已有的面向二进制翻译器的随机化测试的局限性进行了分析。

第 3 章　BTRTG 的设计与实现

二进制翻译器中生成的本地指令能正确且完整地保持原有指令的语义是正确模拟原程序功能的关键。单条指令的翻译用于实现指令的完整功能，优化翻译用于提升生成指令的质量。单指令翻译过程相互独立，优化翻译涉及到多条指令的组合翻译。本章兼顾翻译器的这两种特征，参考芯片验证系统的设计思路，开发了随机测试生成器 BTRTG。BTRTG 具有良好的扩展性，能灵活面对各种测试场景。其生成的测试程序简短，自带验证功能，便于调试。

3.1　概述

BTRTG 借鉴了芯片验证系统的设计思路，结合了二进制翻译器的特征，引入以下两种机制：（1）针对单指令模拟的正确性，引入测试知识模块。测试知识为测试工程师根据测试经验保存的一些操作数数值的边界值或具有某种分布特点的随机值。相对完全随机的操作数数值，这些边界值更能提升测试程序的质量。（2）针对优化翻译，引入了测试模板语言。模板语言结合了汇编程序的特性，为测试程序的生成提供一些限制条件。这些限制条件包括：指令的组合方式，指令的操作数类型，验证信息输出的位置等。BTRTG 根据测试模板的引导生成可执行的汇编程序。

在指令生成的合法性保证上，本文抛弃了 Genesys-Pro 中通过求解约束满足问题来生成指令的方法。在 Genesys-Pro 中，指令生成引擎根据指令的限制信息形成一个约束满足问题（CSP），然后通过求解该问题来生成合法指令。如果该问题无解则回退当前指令，重复尝试或选择其它指令。本文采用在生成指令的过程中加入一些特殊限制条件来保证每条生成指令的合理性，指令生成后直接放入队列并最终写入测试程序。本文抛弃 CSP 求解方案有两方面的原因。（1）指令限制信息归结为 CSP 和指令回退机制在设计上比较复杂。这套机制的实现需要一个模拟器来实时模拟当前指令的运行，模拟器的开发本身并不容易。（2）二进制翻译的验证相对芯片验证更为简单，随机性程度低的指令已经具备好的测试效果。单条指令的二进制翻译考虑的最小粒度是宿主机平台的指令，而芯片验证需要考虑的粒度需要精确到指令每个比特位运算的正确性。比如，X86 的 ADD

指令的加功能有对应的 MIPS 指令的加法指令来模拟。其加功能只需要边界值正确即可，其它大量可能的数值运算的正确性已经由宿主机的芯片正确性来保证。

图3.1 为 BTRTG 的概览图。测试工程师根据一个具体的测试场景设计出测试方案，然后使用模板语言规范设计好测试模板。BTRTG 中的解析模块解析该模板并保存在相应数据结构中。同时，解析模块也将解析相应的测试知识并保存。Walk 主引擎遍历模板结构，调用相应的语句处理函数生成 X86 指令。生成指令的方式是先生成指令名，再以树状方式生成指令操作数。每生成一条指令将其插入指令队列相应的位置，最终输出到测试文件。测试文件的头部包含了宏定义语句和符号以及必要的寄存器初始化指令，尾部为程序退出相关指令。测试工程师使用 BTRTG 提供的自动编译库编译该汇编程序得到可执行的测试程序。最后，待测试的二进制翻译器运行该测试，如果测试运行失败，可根据测试输出的错误提示定位错误。

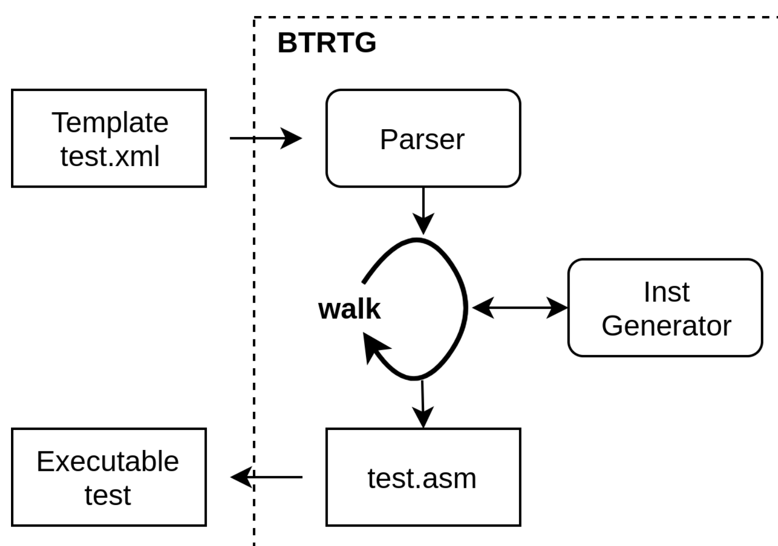

图 3.1 BTRTG 概览图

Figure 3.1 BTRTG Overview

3.2 指令生成引擎

图3.2为 X86 指令编码格式。一条指令由以下几部分组成：前缀，操作码，ModR/M 和 SIB 组成的寻址说明符，地址偏移，立即数。其中，除操作码外其它都是可选项。整个指令的长度变长，编码复杂。

Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
1byte optional	1,2,3 byte	1 byte if required	1 byte if required	0,1,2,4 byte	0,1,2,4 byte

图 3.2 X86 架构指令格式

Figure 3.2 X86 Architecture Instruction Format

图3.3为一条指令的生成过程，该例中生成了一条 ADD 指令和两条操作数初始化指令。指令生成器的输入有两种类型：（1）一个只包含指令类型信息的种子，该种子由模板解析器选定指令后创建；（2）模板中定义的伪指令，伪指令包含了操作数的类型等具体信息，例如 add ecx,mem32。指令生成包括指令名和操作数两部分。操作数在生成指令名后按序生成。在生成操作数的过程中，指令生成器除了确保操作数本身的合理性外，还可能插入一些辅助指令封装当前指令，以保障整条指令的合法性。在保证指令语义合法的基础上，为丰富操作数数值的意义，指令生成器在生成指令的过程中会插入一些辅助性指令初始化操作数。完全随机的初始值是完全没有意义的，测试工程师需要提前准备好测试知识，生成器根据测试知识选取初始化数值。

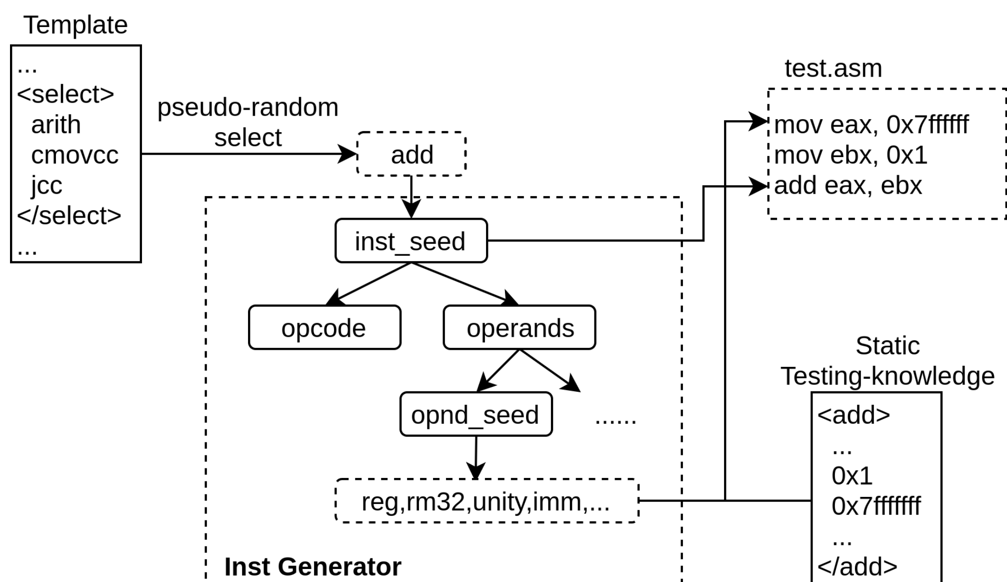

图 3.3 指令生成流程

Figure 3.3 Instruction Generation Process

指令合法性保证包括两个范畴。（1）指令语法的合法性。语法正确性要求生成引擎能枚举所有的指令格式，包括操作数类型和长度，生成引擎在这种硬性

限制下生成指令。例如 LOOP 指令的目标地址只能为 8 位的地址偏移，因此在 [-128,+127] 范围之外的操作数都不合法。（2）指令语义的合法性。语义正确性往往和指令所在上下文密切相关。比如地址的生成一直是指令随机生成中的难点。传统的生成方案布包括每次选择固定的访存地址和低效的非法回退生成机制。前者淡化了访存格式的多样性，后者则需要引入额外的模拟器模拟地址的计算。

后续三小节分三个小部分阐述 BTRTG 中对指令语义合法性的保证机制。

3.2.1 辅助指令插入机制

BTRTG 中的辅助指令即为当前生成指令前后插入的一些封装指令，这些指令的主要功能是改写寄存器、内存的值和修改控制流。指令生成引擎在生成指令的过程中，会自动插入这些辅助性指令，插入的情形包括以下三类：（1）初始化操作数，丰富测试的意义；（2）检查操作数数值，规避会造成程序退出的例外；（3）初始化内存地址，引导指令访存；（4）结合汇编程序中的 label 关键字和跳转指令，修改控制流，规避死循环、无效跳转目的地址等。本小节介绍（1）和（2），（3）和（4）结合其它相关控制逻辑将在后两小节详细介绍。

操作数初始化指令是测试程序和测试知识的接口，这些指令在生成每个操作数之后插入，一般通过 MOV 等数据传输指令来完成。在初始化某个操作数数值时，引擎首先根据操作数类型向测试知识生成接口申请一个合适的常数值，然后根据该操作数类型特有的初始化方法插入初始化指令。除了显示的操作数外，指令的隐含操作数也会被初始化，比如 MUL 指令隐含的 EAX、EDX 寄存器操作数。图3.4中，FPTAN 指令的隐含操作数被初始化为弧度值 0.3，并以单精度浮点的表示方式写入 ST0。

fptan	fstp st0 mov dword [float_data], 0x3e99999a fld dword [float_data] fptan

图 3.4 FPTAN 指令隐含操作数 ST0 的初始化

Figure 3.4 Initialization of FPTAN's Implied Operand ST0

操作数初始化是可选项，其可能性大小由模板中设置的指令初始化概率决

定。初始化可以发生在任何一条指令之前。比如，对于单指令的测试可以将其初始化概率设置为 1.0，而对于基本块级测试，可以仅给块内第一条指令设置一个较小的初始化概率。初始化概率大小的选择需结合测试知识和具体的测试场景。

规避指令的例外的目的是防止程序在运行中途退出。根据 Intel 指令参考手册 [Corporation, 2016b]，每条指令都可能触发多种例外。有些例外实际上已经被避免，有些例外则可能发生，这些未避免的例外可能导致程序的崩溃。因此，要插入该类辅助指令，我们需要深入分析指令这些例外。

以 DIV 指令为例，考虑在保护模式下所有可能触发的例外：DE（除法错误），GP（通用保护），SS（栈段错误），PF（页错误），AC（非对齐访存），UD（无效操作码）。AC 和 UD 在不打开对齐访问检测和使用 LOCK 前缀的情况下不会发生。PF 触发后由 Host 操作系统自动进行页错误处理。GP 和 SS 由越界或者段寄存器值非法造成，可通过下一小节介绍的地址寄存器初始化机制来规避。最后，触发 DE 的原因有除零或溢出，这是我们需要规避的。图 3.5为对除零溢出的规避：检测出被除数为零则跳过除法指令。溢出的规避则各更加麻烦，我们可以通过设计合理的测试知识来减少溢出情形的出现，见第3.3节。

div ax	test ax,$0 jz L div ax L: ...

图 3.5 除零的规避

Figure 3.5 The Avoidance of Division by Zero

例外的规避是可选项。一方面，验证例外的处理是功能验证的一部分。因此，在需要验证例外时，我们可以禁止规避指令的生成。另一方面，规避所有的例外难以实现。在浮点运算中，规避例外意味着要插入大量额外指令，这降低了测试程序的连续性，影响测试效果。

3.2.2　内存操作数的设计

指令访存的随机化是指令生成的难点之一，其难点包含以下两个方面：

1. 随机产生的内存地址可能不具备相应读写权限，这导致了生成的指令一旦含有内存操作数便无法执行。例如，以 EAX 寄存器的值为内存地址进行访

存，其值由前序指令流决定。考虑到指令流的随意性和 EAX 的位宽，其值可以是 $[0,2^{32}-1]$ 范围内任意值，覆盖地址区域长达 4GB。而进程中可访问的地址区域一般比较小，地址空间中大部分是未分配区域，不具有读写权限。

2. 内存操作数的访存格式以及访存地址在地址空间中的分布不可控。在 EATBit 中，内存操作数使用了一个固定的 64 字节数据区域，且访问该地址的编址模式单一。程序数据段的装载地址由链接器和装载器共同决定，程序内部无法决定运行时内存操作数的虚拟地址。

本节提出的操作数生成方案包含以下五部分：（1）全局的数据标签，包含几段初始化好的数据块；（2）地址生成器，在数据段范围生成一个可用于组合地址模式的合法地址；（3）辅助指令，根据生成器生成的合法地址初始化地址操作数中的寄存器；（4）寄存器资源锁，防止包含地址的寄存器被改写；（5）链接器的链接脚本，通过修改数据段加载地址调节操作数的实际内存分布。

表 3.1为本节中对各编址模式和设计的地址生成方案。X86 的地址编址模式由 Base、Index、Scale 和 Displacement 组成，共包含六种组合方式。Base 和 Index 为寄存器，一般用做数组访问的基址和索引。Scale 可取 2，4 或 8，它一般作为数组访问中的数据类型长度。Displacement 为立即数，表示数据地址的偏移部分。

表 3.1 X86 地址编址模式

Table 3.1 X86 Addressing Modes

编址模式	操作数格式	辅助指令
Displacement	[data]	-
Base	[reg]	lea reg,data[1]
Base + Displacement	[reg + imm]	lea reg,data
(Index*Scale) + Displacement	[reg*imm + data]	mov reg,index[2]
Base + Index + Displacement	[regb + regi + imm]	lea regb,data mov regi,index
Base + Index*Scale + Displacement	[regb + regi*imm + imm]	lea regb,data mov regi,index

[1] data 为数据标签地址偏移.

[2] index 为地址生成器生成的索引值.

以图 3.6为例，该例中生成了编址模式为 Base + Index*Scale + Displacement 的一个内存操作数。图中左侧为地址生成器选择的地址，该地址位于 datai 数据块内。图中右侧为生成的指令，访存操作数的基址和索引分别为 EDX 和 ECX 寄存器。在生成该操作数的过程中，首先生成 EDX。之后锁住 EDX，防止表示索引的寄存器也为 EDX。初始化生成的两个不同寄存器，便可安全使用该内存操作数。

```
                        datai:  times 8192 db 0
base ← datai             ...
index ← 0x1d6           lea edx, datai
scale ← 8               mov ecx, 0x1d6
                        mov dword [edx + ecx*8 + 0x10d0], 0x402df854
```

图 3.6 地址和内存操作数的生成

Figure 3.6 Generation of Address and Memory Operands

该方案使用全局的数据标签作为内存操作数的选择范围并不失一般性。在 Linux 进程的地址空间布局中，4GB 连续地址被分成不同功能的几块区域。地址块的功能由操作系统规定，与硬件无关。而在硬件层面，也就是翻译器的实现层面，X86 保护模式的一大特征就是提供一系列硬件标志位标记内存的访问权限。因此在访存地址的生成中，地址块的功能不再是关注点，只需关注地址的读和写权限即可。通过指定数据段的装载地址可使得全局数据块覆盖地址空间的任意位置。

此外，翻译器对内存操作数的翻译方式和内存编址模式有关，该方案覆盖六种编址模式具有一般性。

3.2.3　控制转移逻辑

控制转移指令是翻译基本块分块的标志。生成控制转移指令的意义不仅在于指令本身的测试，还在于测试翻译器在翻译两个基本块之间的所有行为，比如：块与块之间的链接，寄存器的分配算法，上下文切换，代码缓冲区的分配等。

控制指令的生成难点在于其跳转目标地址的选择。跳转目标地址与内存操作数的地址类似，涉及到地址的权限问题。完全随机的目标地址往往不合法，图 3.7 列举了 JMP 指令的随机跳转可能出现的非法地址。此外，为防止程序执行时间

过长，程序中不应包含死循环。因此，合法的跳转地址应满足以下两点要求：(2) 跳转的目标地址处有可执行的指令；(2) 跳转地址不应引发死循环。

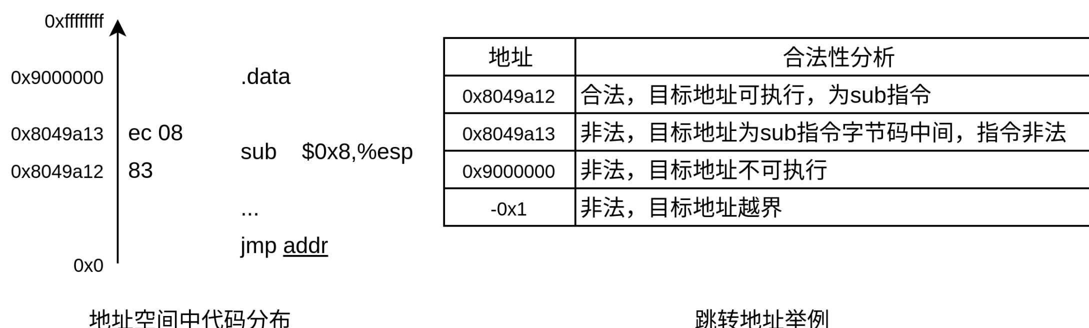

图 3.7 JMP 指令跳转地址合法性分析

Figure 3.7 Legitimacy Analysis of JUMP Instruction's Target Address

本小节提出的控制转移指令生成方案包含以下几部分：(1) 随机在生成的任意指令前生成 Label，这样所有的 Label 处都包含可执行的指令；(2) 跳转目标地址都以 Label 的形式生成；(3) 跳转指令前后添加额外的跳转指令以修正执行流，保证指令的执行流不出现死循环；(4) 针对 CALL 指令和 LOOP/LOOPcc 指令的额外机制。

以 JMP 指令为例，图3.8展示了其生成方案。首先，在整个测试程序的生成过程中，生成器以一定概率（如 0.1）在当前生成的指令前插入 Label。之后在生成跳转指令时，随机选择一个已经存在的 Label，如 L_i。在 L_i 前插入 L_n，L_n 为待生成的 jmp addr 实际跳转地址。此时，原来 L_i 之前的指令的后续执行流已经被改变，可能会造成死循环。为了避免，再在 L_n 前添加 jmp L_i 指令恢复原执行流。Jcc 为条件分支，具有两个跳转目标地址，随意的跳转容易造成死循环。其生成和 JMP 指令类似，如图 3.8。

CALL 和 LOOP/LOOPcc 指令的特点是该指令的目标地址处代码执行完毕后会回到该指令的下一条指令继续执行。CALL 指令用于函数调用，一般和 RET 指令结合使用，生成时在 CALL 的目的地址处生成被调用的指令并在结尾处添加 RET 指令用于返回。LOOP/LOOPcc 指令为循环指令，ECX 寄存器为控制循环次数的计数器，ECX 的值太大可能造成循环时间过久。在循环内部 ECX 的值不应被随意修改。

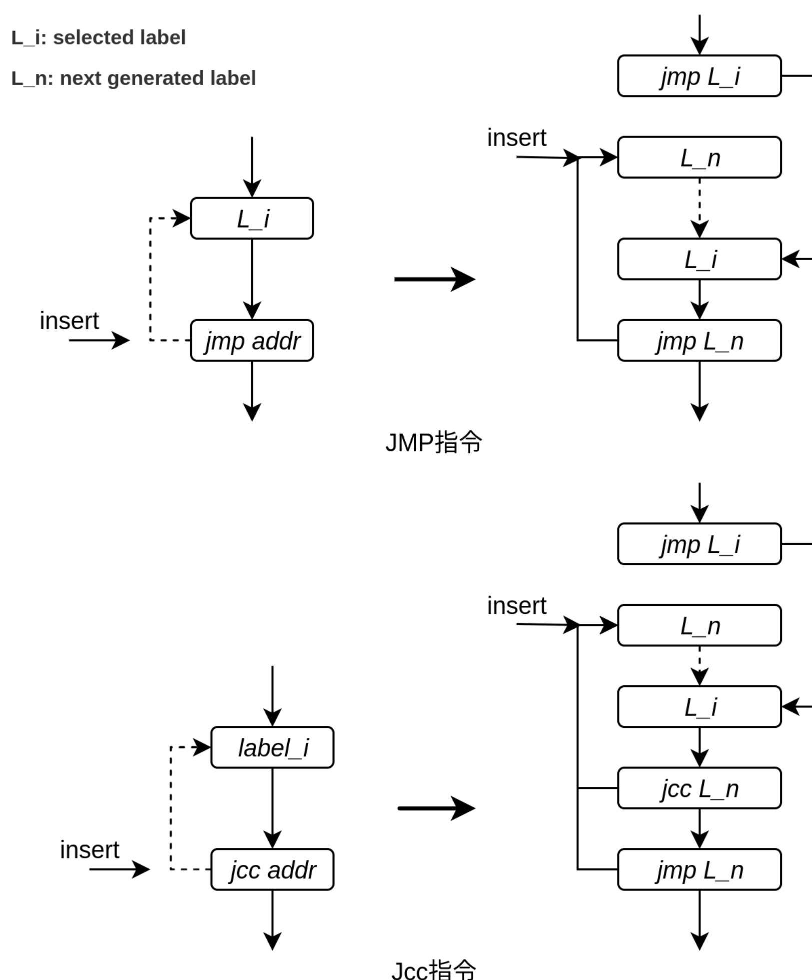

图 3.8 JMP、JCC 指令生成方案

Figure 3.8 Instruction Generation Scheme of JMP, JCC

根据这两类指令的特点，本文采用以下设计来生成指令：（1）引入控制锁 ctrllock，该锁的作用是禁止控制转移指令的生成。它用在跳转指令后，使得跳转的目标地址处指令为一个封闭的块，不包含其它跳转指令从而避免死循环。（2）使用对 ECX 的寄存器锁，在循环内部不再生成包含 ECX 的寄存器值，防止其被修改。（3）添加 Cct（Closed Control Transfer）模板语句，该语句将跳转目的地址处被调用的指令进行封装，被封装指令可看作为跳转操作的内部指令。其具体生成方案如图 3.9所示。

此外，间接跳转可以基于以上方案实现。根据以上分析，指令使用 Label 作为跳转的目标地址，将 Label 地址装载到寄存器中，便可以作为间接跳转的操作数。

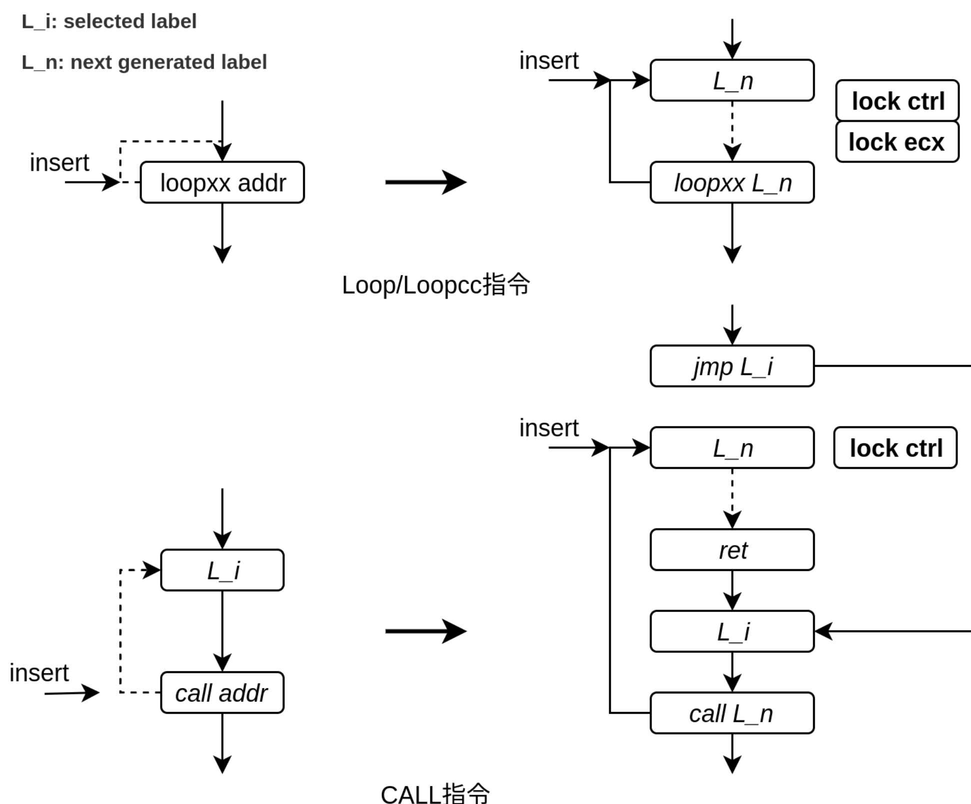

图 3.9 LOOP/LOOPcc、CALL 指令生成方案

Figure 3.9 Instruction Generation Scheme of LOOP/LOOPcc, CALL

3.3 测试知识

在测试领域，测试知识即由测试工程师给测试生成器添加的限制条件，这些限制来自于历史经验或者理论分析，用于引导一个测试的测试倾向。在 BTRTG 中，测试知识的设计目的是为指令操作数提供有意义的初始数据。这些数据可以为历史积累下的指令翻译过程中的边界值，也可以为基于特殊算法的随机数发生器。这些值对指令的验证更具针对性，应该有更高的概率出现在测试程序中。

二进制翻译器中测试场景和翻译过程密切相关，对测试场景有充足的认识是设计出带倾向性指令测试数据的重要技巧。比如，在 ADD 指令的测试中，边界值除 32 位最大、最小值和零等边界值外，还应该包含能修改 AF、PF 等状态位的值。ADD 的翻译过程中对状态位的计算需要独立的一块指令。一般意义上，翻译函数生成的本地指令在被执行时也有其执行流。测试知识的设计除了考虑翻译函数的覆盖率外，还应考虑生成的本质指令的执行情况。本地指令为机器码，其覆盖情况没有明确的评判标准。根据经验，一般在条件分支指令和对原客

户机寄存器置位的指令处可以挑选出边界值。

在本文的设计中，指令测试知识通过带权决策树组织。每条指令的任意一个操作数都对应于一棵带权决策树，该树的每个结点表示了取值的一种倾向类型。在某些指令中，如 FADD，其两个操作数功能对称，因此采用相同的决策树。在另外一些指令中，如 DIV 指令，其被除数和除数之间的大小关系会导致除零和溢出等例外，需采用不同的决策树以区别开。从父节点到子节点的边都有一个权重值，该值代表了在父节点被选择的情况下，子节点被选择的概率。该值由测试知识模板维护，可根据需求进行修改。

图 3.10是 FSIN 指令的一个简单测试知识生成树。它的一级可选类型包含三部分：（1）所有的边界值；（2）1/3 的倍数，分布于整个取值空间，且 1/3 为无限循环小数，可用以验证 SIN 运算的精度；（3）边界值加减 3 倍以内最小浮点精度，用于验证边界处的精度。（1）和（3）都属于特殊情况，故权重都只取 10。（2）为一般情况，故权重取 80。

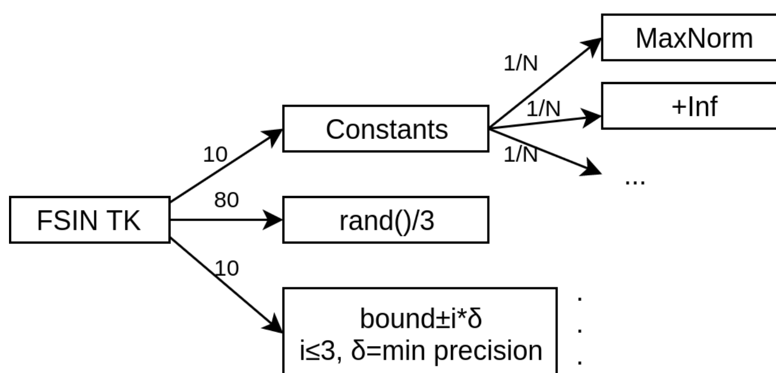

图 3.10 FSIN 指令测试知识决策树

Figure 3.10 FPAdd Instruction’s TK Decision Tree

3.4 模板语言

测试人员使用模板语言来引导生成器按照指定规则生成相对随机的程序。模板语言的引入可以避免生成无意义的随机测试。模板语言最重要的是其表达能力，当前 BTRTG 提供的模板语言已满足基本的测试需求，更多的语法规则有待后续对测试场景的挖掘而相应地补充。该语言目前支持两种语句类型：指令级封装语句和指令块封装语句。前者指导指令生成器生成特定的指令，后者用于控制不同指令的生成顺序、组合方式等。

指令块封装语句目前有以下五种：

- Seq（Sequence）语句，顺序遍历语句生成指令。
- Sel（Select）语句，根据其封装的语句权重选择语句进行生成。
- Cct（Closed Control Transfer）语句，封装一段不包含跳转语句的封闭代码，包括 LOOP/LOOPcc、CALL 等指令。该类语句只能封装指令级封装语句，生成的指令在跳转后执行。
- Rpt（Repeat）语句，根据给定的次数重复生成内部封装好的语句。
- Trv（Traverse）语句，遍历一条指令的测试知识的各种组合，重复遍历被封装的语句。该语句主要针对单元测试设计。

指令级封装语句目前有三种：

- G（Instruction Group）语句，该语句从分类好的指令组中随机选择一条指令，另随机选择一种合理的参数组合生成一条指令。
- C（Check Point）语句，该语句定义一个检查点，参数为一个寄存器名。它会生成一组对检查点函数的参数压栈和函数调用指令。
- I（Instruction）语句，根据提供的指令信息生成一条指令。指令信息是一条伪指令，其操作数信息可以是类型或者真实的寄存器。比如，提供的信息为 mov ebx, mem32，则可能生成指令 mov ebx, [ecx + 0x23]。

另外，模板语言中还添加了变量机制。变量初始时包含名称，类型信息，生成器在第一次使用某变量时更具其类型信息生成随机的操作数。之后，所有使用该变量的位置都被替换成该操作数，该过程类似于宏替换。添加变量机制的目的有两个：（1）方便指定检查点的检查目标为某寄存器。寄存器是随机生成的，使用变量可以使得检查点处的寄存器和检查点之前生成的寄存器为同一个寄存器。（2）方便设置指令操作数之间的依赖情况，比如将前一条指令的源操作数设置为后一条指令的目的操作数。

下面从两个实际测试中选取两个测试模板片段作为具体使用样例。

图 3.11来自于一个减法的单元测试模板，该片段用于测试指令类型 sub al, imm8。该模板片段使用到了 Trv 语句，减法指令的语句处设置了 trv = ”true”，它的功能是指导 Trv 语句遍历 SUB 指令的两个操作数的各种取值组合，取值源自于测试知识。生成的每个小代码片段由一条减法指令和一个检查点组成。该例中，减法指令目标寄存器为 AL，AL 和 8-bit 立即数分别被初始化成不同的值。

每个片段的代码组成不变，这两个取值发生改变。

Template	Instructions

<traverse>
<I type = "sub al, imm8" trv = "true"> </I>
<C type = "al"> </C>
<traverse>
	mov al, 0xff
	sub al,0x81
	check al

图 3.11 SUB 指令单元测试片段

Figure 3.11 Unit Test Fragment of SUB Instruction

图 4.7来自于 Flag-Pattern 优化翻译的测试模板。该例中，Pattern-head 和 CMOVcc 都为指令组，Pattern-head 指令组包含了所有会修改 EFLAGS 状态位的算术指令，CMOVcc 则包含所有条件数据传输指令。他们一前一后按序生成，得到模式尾为 CMOVcc 的指令 Pattern。在两条指令后，故设置一个检查点，由于不明确 CMOVcc 生成的指令的目的操作数，故检查整个 X86 硬件状态。指令块封装语句 reapeat 引导引导生成器重复 1000 遍生成该模式指令（任意一条 EFLAGS 定义指令后跟任意一条引用 EFLAGS 的 CMOVcc 指令）。例如，本例中生成了一条 INC 指令和一条 CMOVL 指令。

Template	Instructions
<repeat times = "1000"> <G type = "pattern-head"> </G> <G type = "cmovcc"> </G> <C type = "x86_state"> </C> </repeat>	inc dh lea esi, data0 mov eax, 0x22a cmovl edx,[esi+eax*8+0xe30] check x86_state

图 3.12 Flag-Pattern 优化翻译测试片段

Figure 3.12 Test Fragment of Flag-Pattern Optimization Translation

3.5　验证机制

判断测试的运行是否正确依赖于可靠的验证机制，而人工的方式筛查指令是否正确执行是非常困难的，因此在过去一些自动化验证工具应运而生。文献 [Jiuye 等, 2015] 提出了一种针对二进制翻译的验证方案。它的思路是，同一个程序在标准 QEMU 上和待测试二进制翻译器上同时运行，两个二进制翻译器都将

各自的架构状态和写数据信息发送给第三个验证器进程，由验证器根据两者的数据来确定程序是否正确执行。在 QEMU 中，Alex Bennée 添加了实现类似对比功能的插件 lockstep。lockstep 在每次执行完一个基本块后对比 PC 值，该方法只能对比程序的执行流，虽然可以较好地辅助错误的排查，但依旧是不完整的验证。开发者可以根据对比需求在 lockstep 中加入需要对比的状态值，然而判断翻译的正确性不能简单通过两者是否相等判断出来。

实际上，待测试虚拟机和源虚拟机在某个执行点架构状态不同可能有以下几点原因：（1）待测试虚拟机本身存在错误；（2）两者的架构状态初始化数据不同，比如系统为了安全性考虑往往给栈基址初始化一个随机值；（3）一些受外界状态影响的系统调用的返回值不同，比如获取当前的时间等。由此可知，后两种情况并非翻译器的错误，容易干扰到验证结果。对于问题（2），可以将寄存器和内存初始化为相同的值，甚至可以让两者使用共享的一块内存。对于问题（3），则需要改写系统调用，将和外界环境相关的返回值改成无关的定值。根据文献 [Jiuye 等, 2015]，这些问题的解决都需要修改原系统，工程量较大。

为了方便验证，本文试图将部分验证功能集成到测试程序本身，集成的关键在于输出与验证相关的信息。BTRTG 中定义了检查点（Check Point），检查点由模板设计者根据需求添加。在测试程序中插入检测点语句（由 <C> tag 封装），可以检测指定寄存器的值与标准机器上的值是否相同。对于内存值的对比，则可以先将内存值取到寄存器中，再进行对比。例如，图 3.13中，生成 inc mem32 指令后，现将该内存值传给 EAX 寄存器，再对比 EAX 寄存器。除了对比寄存器状态外，测试点还可以输出当前 PC 等帮助定位错误的信息。

```
<V var = "mem" type = "mem32"></V>
<I type = "inc @mem"></I>
<I type = "mov eax,@mem"></I>
<C type = "eax"></C>
```

图 3.13 内存操作数的验证

Figure 3.13 Verification of Memory Operand

当前，C 语句支持 EFLAGS 以及其它寄存器的单独检测，也支持如 X86 状态相关的所有寄存器的同时检测。其中，EFLAGS 寄存器的检测是一个难点。在

X86 指令手册中，指令可以只定义 EFLAGS 的部分位。例如，OR 指令不定义 AF 位，这意味着在 OR 指令的执行后 AF 可以是任意值。这些未定义状态位不应该在验证的过程中被对比。因此，在生成指令的过程中，EFLAGS 中的每一个状态位的定义情况都应被记录，之后在 C 语句中不检查当前未定义的 EFLAGS 位。

图 3.14既是测试程序的编译过程，也是保存测试点标准输出的过程。程序编译过程中通过两次链接。第一次链接后在客户机标准机器上运行，检查点函数输出测试点的标准输出并导入文件。第二次链接时将标准输出一并链接，检查点函数的功能为和对应位标准输出进行对比。第二次链接的输出为最终的可执行测试程序。

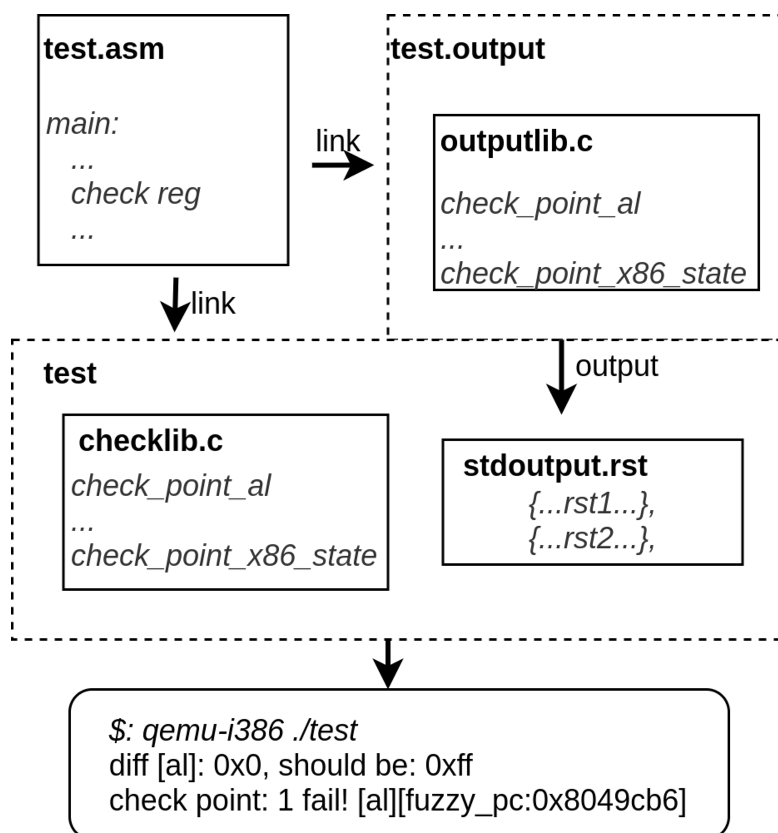

图 3.14 生成的测试程序的编译流程

Figure 3.14 Compiling Process of the Generated Test

3.6　本章小结

本章介绍了 BTRTG 的设计方案及各个模块的设计目的。本章先描述了 BTRTG 的整体框架和工作流程，然后分别介绍其各个功能模块的设计方案。指令生成引擎中有两个难点：内存操作数的生成和控制转移指令的生成。测试知识和模

板语言的设计都是为了提升测试生成的自动化程度和生成的测试的质量。最后，在测试程序中加入自动验证的功能使得测试的使用变得更加友好。

第 4 章　典型测试场景与测试

测试场景的选取是测试过程的第一步。选定测试场景后，分析测试场景并设计测试程序进行测试。测试过程中，测试人员通过观察代码覆盖率等参数，不断提升测试效果从而得到高质量的测试。本章选取了翻译器测试过程中的四个典型场景，分析各个场景下的测试方法，并使用 BTRTG 生成测试程序进行测试。这四个样例中，前两个为指令翻译测试，后两个为优化测试。在这四个测试的设计中，BTRTG 的各个模块的设计意义和功能的健全性也得到充分体现。

4.1　浮点运算的计算正确性验证

X87 浮点运算单元（FPU）在图形处理、科学、工程和商业应用中提供了高性能浮点处理能力 [Corporation, 2016a]。浮点运算单元除了支持浮点数据类型外，还支持整型和 BCD 整型等数据类型。浮点处理单元除了提供了基本算术运算指令外，还提供三角函数和对数运算等复杂运算指令。X87 提供的单精度、双精度和 80 位扩展双精度浮点运算遵循 IEEE 754-1985 标准。浮点数的表示包括符号位，阶码和尾数三部分，相对于整型其边界值非常多。本节以验证浮点运算的计算正确性为目标，展示测试知识的选取方法。

4.1.1　边界值测试

DEST / SRC	-∞	-F	-0	+0	+F	+∞	NaN
-∞	-∞	-∞	-∞	-∞	-∞	#IA	NaN
-F	-∞	-F	SRC	SRC	±F, ±0	+∞	NaN
-0	-∞	DEST	-0	±0	DEST	+∞	NaN
+0	-∞	DEST	±0	+0	DEST	+∞	NaN
+F	-∞	±F, ±0	SRC	SRC	+F	+∞	NaN
+∞	#IA	+∞	+∞	+∞	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

图 4.1 FADD 加法规则

Figure 4.1 FADD Results

以 FADD 指令为例，该指令有四种参数格式。本节以操作数数值为测试重点，不关注操作类型，因此仅针对固定的指令类型 fadd m32fp 进行测试。根据 Intel 指令手册 [Corporation, 2016b]，图4.1为浮点加法在各类值下的计算行为。其中，F 代表有穷数，#IA 为算术操作数无效异常。

根据图 4.1，源操作数和目的操作数分别有 7 种取值情况。本文根据这些数据类型添加离散型测试知识，如图 4.2。其中，有穷数取自然对数的底数 e=2.71828...（单精度表示为 0x402df854）。7 个特殊值封装在常数组 fadd-bnds 中。设置 FADD 指令的操作数测试知识为结点 fadd-bnds，其权重置为 100。而另一项测试知识 normal 并不封装任何常数，它代表了测试知识从随机数产生器获得，在后续精度测试中被使用到。

```
<!-- tks/floatpointConsts.xml -->
<CG name = "fadd-bnds">
    <Float32i> 0xff800000 </Float32i>   −∞
    <Float32i> 0xc02df854 </Float32i>   −e
    <Float32i> 0x80000000 </Float32i>   −0
    <Float32i> 0x00000000 </Float32i>   +0
    <Float32i> 0x402df854 </Float32i>   +e
    <Float32i> 0x7f800000 </Float32i>   +∞
    <Float32i> 0x7fc00001 </Float32i>   nan
</CG>

<!-- tks/floatpointTK.xml -->
<TK inst = "fadd">
    <Node name = "fadd-bnds" weight="100"> </Node>
    <Node name = "normal" weight="0"> </Node>
</TK>
```

图 4.2 FADD 指令边界值测试知识

Figure 4.2 Testing-Knowledge for FADD’s Boundary Values

XQM 经该单元测试测试后，得到测试结果如图 4.3。根据结果可知，加法操作只有在结果为有穷且非零时正确。根据测试输出的调试辅助信息不难发现，XQM 输出的特殊值和 X86 的特殊值不一致但却存在某种固定的对应关系。比如，标准输出 $+\infty$ 应该为 0x7fff8000000000000000，而 XQM 得到的错误输出为 0x43ff8000000000000000。进一步分析源码发现，实际的加法模拟中使用了 MIPS 的双精度浮点加法 ADD.D 指令，运算本身没有错误，错误发生在读取浮点寄存器的过程中。读取浮点寄存器时，需要将 MIPS 中 64 位双精度浮点数据转换为

80 位扩展双精度表示，模拟该过程的转换函数出现了错误。

SRC \ DEST	-∞	-F	-0	+0	+F	+∞	NaN
-∞	×	×	×	×	×	-	×
-F	×	√	√	√	×	×	×
-0	×	√	×	×	√	×	×
+0	×	√	×	×	√	×	×
+F	×	×	√	√	√	×	×
+∞	-	×	×	×	×	×	×
NaN	×	×	×	×	×	×	×

图 4.3 FADD 边界值测试测试结果，× 为失败，√ 为通过

Figure 4.3 Results of FADD’s Boundary Value Testing, × is Fail, √ is Pass

4.1.2　精度测试

在 QEMU 中，复杂的浮点运算指令都是由 C 函数模拟的，这些函数被称为 helper 函数。在 helper 函数的内部，核心运算部分又调用了 C 库中的数学函数。比如，三角函数的正弦，实际上使用的是数学库函数 sin。该方案在一定程度上保证了计算的精确性。而对于通用浮点运算指令以及经手动翻译优化后的浮点运算指令，其精度可能存在问题。

为测试浮点计算的精度，调整图 4.2中的测试知识权重，将 normal 结点权重置为 100，选择浮点随机数生成器提供测试知识。精度测试中生成了大量的随机浮点数组合情况，经过进一步的测试，更多的问题得到暴露。

X87 浮点运算的寄存器为 80 位扩展双精度，而 MIPS 浮点运算使用的寄存器为 64 位双精度，该差异是浮点运算错误的主要原因。不同的表示范围导致了二者在高精度运算下存在精度上的差异。此外，一些在应用中几乎不可能出现的寄存器特殊访问方式会导致特殊的错误。例如，在 MOVD 指令修改了 MMX0 寄存器的低 32 位之后，如果读取 ST0（MMX0 为 ST0 的低 64 位）的值，则 XQM 下使用 64 位寄存器模拟 MMX0 的方式下读取出来的值和宿主机上的值完全不一致。

4.2 操作数处理的正确性验证

X86 架构中，操作数的各种组合构成了指令的大量变种，其翻译规则的复杂性为单指令的测试增加了很多困难。另外，操作数读、写指令中的扩展模式优化进一步增加了操作数处理的复杂性。本节的目的是测试翻译器对各类操作数的处理的正确性。

单指令翻译的翻译流程都是相似的。在设计 X86 指令翻译函数的过程中，根据 X86 指令执行的流程，一般采用以下步骤翻译指令：

1. 取源操作数；
2. 例外检查，如除法溢出等；
3. 指令关键功能的模拟；
4. 计算 EFLAGS 寄存器的 6 个状态位；
5. 保存目的操作数。

在该流程中，步骤 1 和 5 都涉及到 X86 操作数的处理。为了进行 X86 和 MIPS 指令之间的操作数转换，XQM 设计了操作数处理相关接口，专门用于 X86 指令的操作数和 MIPS 操作数之间的读写：（1）将 X86 指令的立即数、地址、地址操作数、寄存器等操作数类型转换为 MIPS 操作数；（2）将 MIPS 的寄存器写入到 X86 的寄存器或内存操作数中。

这个转换过程中，由于 X86 架构为 32 位，MIPS 机器为 64 位，因此需要额外的指令进行数据扩展等特殊处理。而消除这些额外指令的优化机制称为扩展模式优化。下面详细介绍。

在 XQM 中，Guest 中通用寄存器与 Host 中寄存器一一对应，该方案称为寄存器直接映射。Host 包含 32 个通用寄存器，为减少寄存器分配的复杂度以及生成不必要翻译指令，Guest 的寄存器被映射到 Host 的寄存器，比如 EAX 映射到 t3（MIPS 第 15 号寄存器）。在每个基本块执行前，翻译器都会进行翻译器程序和生成的本地代码之间的上下文切换，Host 的寄存器在此时恢复成上一基本块结束时的值。

由于 Host 和 Guest 寄存器位宽的差异，翻译器有时需要对 Host 寄存器高 32 位进行零扩展、符号扩展等处理。假如 Host 和 Guest 都是 32 位机器，则寄存器的读写操作完全一致。而实际上，Host 为 64 位 MIPS 机器，同时支持 32 位和 64 位 MIPS 指令。Guest 使用 32 位 X86 指令，在设计翻译函数时，根据具体情况

使用 32 位或 64 位 MIPS 指令。不同 MIPS 指令使用相同的物理寄存器，只是 32 位 MIPS 指令只用到寄存器低 32 位，因为它在计算中会自动将寄存器的低 32 位做符号扩展到 64 位。而如果寄存器中的值表示的是 32 位地址，它不应该被符号扩展，所以需要增加额外的指令将其高位清零。

生成额外的高位清零指令会带来指令的膨胀，通过扩展模式的传递和分析可以减少该类型指令的生成。比如在 mov reg1,reg2 中，假设 reg1 为地址类型的操作数，则 reg2 也可看做合法的地址操作数，之后如果将 reg2 当做地址操作数使用，比如 add reg3，[reg2] 则不需要在对 reg2 的高位进行清零操作。

扩展模式优化的基本步骤是：

（1）开始翻译前，设置一个表保存所有 Host 寄存器的初始扩展模式，该初始扩展模式由其映射的 Guest 寄存器决定，比如 EAX 为零扩展，ESP 为地址扩展；

（2）在翻译单条指令过程中，翻译器先根据寄存器的当前扩展模式选用合适的方式装载寄存器。然后，翻译器根据使用的本地指令类型以及使用到的本地寄存器的当前扩展状态，重新计算寄存器的扩展状态并更新扩展模式表。

（3）基本块翻译结束后，翻译器根据通用寄存器的当前扩展状态和默认状态插入指令调整相应的 Host 寄存器值。

为测试操作数的处理，本节生成了 XQM 支持的几乎所有指令（特殊指令如 INT 等除外）的所有变种的单元测试程序。而在边界值的选取上，本次测试中使用的测试知识大部分使用 DBT5_UT 中使用的边界值，值添加了少量其他边界值。这样选择的目的是控制测试知识为不变量，而 BTRTG 生成的随机操作数组合格式作为因变量，方便对比效果。

经过测试，我们发现了原程序中的多处错误。浮点以及扩展指令的测试中涉及到精度和表示相关的错误比较多，不方便统计。表4.1为通用型指令中的暴露各错误类型的分布情况，一共 16 处。表中列举的错误案例在实际程序中非常少见。比如 xadd eax, [eax + 0x7b5] 中，EAX 寄存器既作为第一个操作数又作为第二个内存操作数的基址是非常特殊的。在 shrd eax,ebx,0x21 中，移位计数超过 32，在 X86 架构中为合法未定义行为的指令。

图 4.4为本节生成测试集和 DBT5_UT 测试集对 X86 操作数处理代码的覆盖率比较，各项指标均有提升。BTRTG 生成了 DBT5_UT 中未使用到的寄存器，比如 DBT5_UT 只使用到了 XMM 中的 0 号和 1 号寄存器，而 BTRTG 生成了 XMM

表 4.1 通用型指令错误类型分布情况

Table 4.1 Error Type Distribution of General Purpose Instructions

错误类型	数量	例子
Exception Miss	2	除法溢出，除零
Segment Fault	3	xadd eax, [eax + 0x7b5] 中 EAX 寄存器同时作为操作数和地址，扩展模式相覆盖导致出错
Assertion Failed	6	shrd eax, ebx, 0x21 中移位值 0x21 超出 EAX 位宽 32，根据指令规范其为合法指令，不应报错
Different Value	5	xchg dh, [edx + eax + 0x4d6] 中先修改 dh 寄存器导致第二个操作数地址发生变化

中 0-7 号所有寄存器。另外 DBT5_UT 中 CMPXCHG 指令未使用到 8 位和 16 位寄存器。

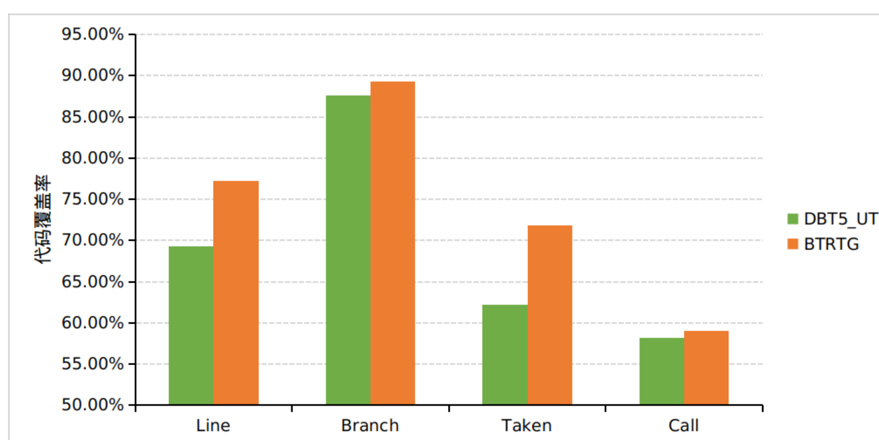

图 4.4 X86 操作数处理相关代码的覆盖率对比

Figure 4.4 Coverage Comparison of X86 Operands's Processing Code

4.3 栈操作局部优化的验证

在 XQM 中，开发者采用一切可行的手段降低指令的膨胀度，这些手段中包含了大量的局部优化。比如，在算术运算指令中，涉及到立即数的运算时，由于 X86 的立即数可以为 32 位，而 MIPS 指令只有 16 位立即数，翻译器往往先将立即数保存在一个寄存器中。在 add eax,0x1ffff 中，翻译器需要先使用 or temp,0x1ffff 将立即数操作数保存在临时寄存器 temp 中，然后再进行加法运算。然而，如果

立即数较小，位宽不超过 16 位，比如 0x1f，则可以直接使用 MIPS 的 ADDIU 指令模拟。因此，在立即数运算中可以先检查立即数的位宽，再选择合适的翻译指令以避免不必要的保存立即数操作。本节以压栈、出栈中对 ESP 寄存器进行加减的指令进行的优化来展示局部优化翻译的测试设计。

X86 架构中，压栈指令有 PUSH、PUSHA、PUSHF，出栈指令的 POP、POPA、POPF。这些指令除了在操作数和栈之间进行数据传输外，还会加减栈顶指针 ESP 寄存器。加减的数值是操作数的位宽，比如，push eax 操作中，ESP 减 4。而 MIPS 架构中不提供 PUSH、POP 操作，因此，翻译器只能通过一段内存来模拟栈，并使用一条 MOV 指令做数据的存取，一条 ADDI 指令做 ESP 的加减。而对于连续的 PUSH 和 POP 操作，翻译器可以先缓存好 ESP 的修改值，在所有栈操作结束后再生成修改 ESP 的指令。这便是栈操作中 ESP 计算的局部优化。

为了验证该优化的正确性，本文生成了随机的压栈和出栈操作。值得注意的是，压栈操作尽量多于出栈操作，这样防止出栈时栈为空。由于该局部优化代码量非常少，统计覆盖率意义不大。通过随机生成的 PUSH 和 POP 操作，发现了一处错误引发程序段错误。该错误的原因是 XQM 中对段寄存器的压栈和出栈在修改 ESP 值时的增、减量为段寄存器的位宽 2，而手册上规定为 4。

4.4　Flag-Pattern 全局优化翻译的验证

在翻译器中，保守的翻译方式是每条 Guest 指令都由一组 Host 指令模拟完成，且其入口和出口之间的行为对模拟硬件的影响完全满足该 Guest 指令的语义。在实际情况中，翻译器往往不需要保证指令粒度级别的语义相等，只需要按在单个基本块甚至多个基本块组成的路径的粒度上保持语义相等即可。因此，很多作用于全局的优化优化方案被提出。例如，在文献 [Song 等, 2019] 中，作者提出了一种基于机器学习技术以基本块为单位的方式翻译代码。它的原理是先将同样的一块高级代码片段使用编译器编译成两个 Host 平台的指令，然后设置好这两块指令之间的寄存器和内存之间的对应关系并作为输入，训练出翻译规则。训练完成后，将该翻译引擎加入到翻译器中，根据是否满足翻译条件选择机器学习翻译方式，不满足的采用传统翻译方式。

XQM 中也使用到了多种全局性的优化翻译方案。测试全局优化翻译的关键是生成符合优化算法特点的指令组合模式。本节以 MIPS 平台的标志位延迟计算

（Delayed Computing）优化翻译技术 [马湘宁 等, 2005] 的测试为例，展示 BTRTG 生成指令组合的能力。该优化翻译的特点是使用一条 MIPS 指令取代定义和使用 EFLAGS 某状态位的两条指令，该优化方案又被称为 Flag-Pattern。

EFLAGS 寄存器是 X86 中用于记录运算指令标志位的特殊寄存器，MIPS 中没有对应的功能部件。它由算术运算、逻辑运算、EFLAGS 位操作指令等一部分指令定义，被条件数据传输、条件跳转、条件置位等另一部分指令引用。EFLAGS 中和运算相关的位有 6 个，它们需要大量 MIPS 指令来分别模拟其计算过程。基于动态反馈的标志位线性分析算法（EfLA）[唐锋 等, 2007] 可以有效减少冗余的 EFLAGS 位计算。

除了减少没有被引用的标志位计算外，标志位的即时计算也可以被优化，Flag-Pattern 即是在使用时进行标志位计算的延迟计算。它的优化关键是：MIPS 中的条件指令自带计算条件的功能，这样的一条指令等价于两条 X86 指令。例如：

```
test ebx, ecx
...
jz label
```

图 4.5 TEST_JZ 模式

Figure 4.5 TEST_JZ Pattern

在该代码片段中，TEST 使用 EBX 和 ECX 做逻辑与操作，修改了 ZF 位。而 JZ 需要根据 ZF 位来判断是否跳转。在 MIPS 代码中，BEQ 条件转移指令可以同时完成计算 ZF 和条件跳转的功能。以上指令翻译成：

```
...
beq rd, rz, label
```

图 4.6 BEQ 指令替换 TEST_JZ 模式

Figure 4.6 Replace TEST_JZ Pattern with BEQ Instruction

其中 rd 寄存器为 TEST 指令的计算结果寄存器，rz 为 0 号寄存器。ZF 的计算需要至少 3 条指，因此，以上操作节约了至少 3 条指令。

标志位延迟计算算法步骤如下：

1. 定义模式记录结构 Pattern，保存模式头、模式尾、两个源操作数和一个目的操作数等信息。

2. 在翻译基本块之前,从后往前扫描指令,记录基本块中包含的所有 Pattern。

3. 在翻译指令时：（1）若该指令定义了标志位，保存其源寄存器操作数和目的操作数的数值到特定寄存器；（2）若该指令引用了某个标志位，则使用前述特定寄存器作为操作数，并用 MIPS 相关条件指令进行翻译。

在该优化中，支持的模式头和模式尾如表 4.2。

表 4.2 标志位延迟计算的模式/尾

Table 4.2 Pattern Head and Pattern Tail of EFLAGS's Lazy Calculation

pattern-head	CMP，SUB，TEST，OR，AND，DEC，ADD，INC
pattern-tail	Jcc，CMOVcc，SETcc

针对该优化方案的测试，可以生成一些定义 EFLAGS 状态位的指令，其后跟随一些引用 EFLAGS 状态位的指令。我们将模式头的指令分为一组，模式尾指令分为一组，然后随机挑选指令进行组合。一个简单的测试模板如图 4.7。该模板中，模式尾为 CMOVcc 类型指令，生成模式头加模式尾指令组合 1000 次，几乎能覆盖所有的模式尾为 CMOVcc 指令的情况。

```
<Template>
    <repeat times = "1000">
        <G type = "pattern-head"> </G>
        <G type = "cmovcc"> </G>
        <C type = "x86_state"> </C>
    </repeat>
</Template>
```

图 4.7 Flag-Pattern 测试模板

Figure 4.7 Flag-Pattern Test Template

由于每个模式头可以最多和三个模式尾进行组合，我们还需要生成多个模式头对多个模式尾组合的情况。此外，为保证在引用 EFLAGS 位前每个 EFLAGS 位都被定义，我们可以对模式头进行更加细致的分类划分，使得模板在生成 EFLAGS 最小定值指令序列后才生成模式尾指令。经测试，本节生成的测试集相

对 DBT5_UT 和 SPEC CPU2000 在覆盖率上均有提升。本节测试集覆盖了所有的优化模式，而 SPEC CPU2000 不包含 ADD_JB 等模式。

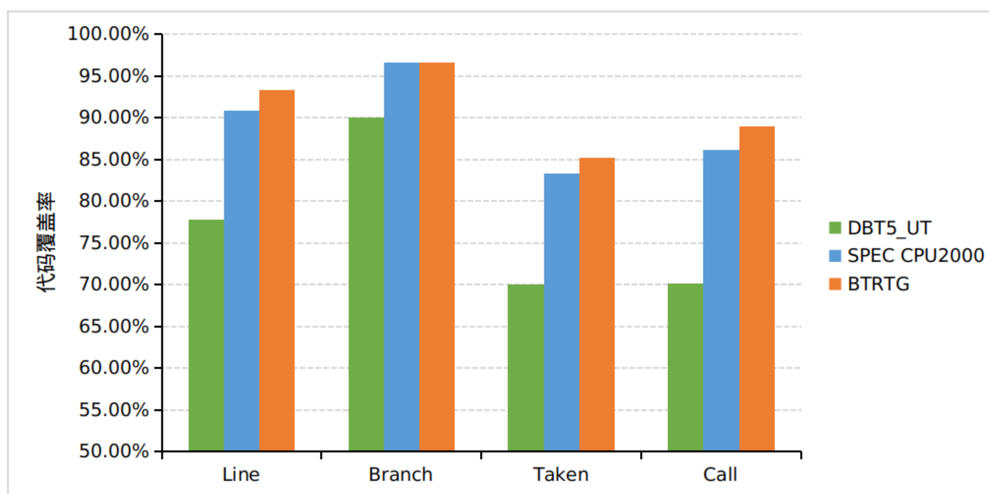

图 4.8 Flag-Pattern 优化翻译的覆盖率对比

Figure 4.8 Coverage Comparison of Flag-Pattern Optimization Translation

4.5 本章小结

本章提取了四个测试场景并使用 BTRTG 进行针对性测试。第一个实验中，浮点运算的计算结果得到检测，多处细节性的错误得到暴露。第二个实验中，操作数的随机生成覆盖到了指令格式的各种组合模式，BTRTG 在使用和 DBT5_UT 相同的边界值进行测试的情况下，新发现了 16 处通用型指令的翻译错误。第三个实验针对翻译器系统中常见的局部优化，发现了 1 处新错误。第四个实验针对 EFLAGS 的延迟计算优化翻译，使用少量模板代码生成的测试程序便得到了对优化代码的较高的覆盖率。本章既测试了二进制翻译器，又展示了 BTRTG 的功能。

第 5 章　测试集综合评测

本文使用 BTRTG 生成了一套轻量的测试集，它包含了第 4章中所有测试程序。为了评测其质量，我们将其和 DBT5_UT 单元测试以及 SPEC CPU2000 标准测试集进行对比。本章以对 XQM 的代码覆盖率为标准，评测新测试集的质量。

5.1　代码覆盖率测试

代码覆盖率是对测试集在源码上运行时执行流的特定角度统计值。它仅反映测试集对被测试软件进行测试的充分程度，与待测试软件的正确性无关。覆盖率 =（执行次数/总统计数）* 100%。以下列举几种常见的覆盖率指标：

（1）行覆盖率

行覆盖率也称作语句覆盖率，它用于报告源码中每个可执行语句行的执行情况。由于代码的错误可能存在于代码中任意位置，行覆盖率能反映潜在的错误分布概率。凡可以生成可执行代码的行都被纳入统计，一些隐式的语句比如省略了的 return 语句不纳入统计范围。执行次数多于 0 次的行计入实际执行行数，行覆盖度 = 实际执行行数/纳入统计总行数 *100%。

比如，在图 5.1中，int x 不纳入统计。如果 condition 为 True，则代码覆盖率为 3/3；如果 condition 为 False，则行覆盖率为 2/3。

```
int *p = NULL;
if (condition)
    p = \&v;
*p = 1;
```

图 5.1 某 C 代码片段

Figure 5.1 A C Program Fragment

（2）分支覆盖率

分支覆盖率也称为判定覆盖率，它用于报告控制语句中布尔表达式的 True 和 False 情况。它只考虑整个布尔表达式的最终取值，和其中使用到的逻辑操作符无关。除了常用的 if、while 语句外，它还统计 switch、例外处理等一切包含代码入口和出口的位置。

它比行覆盖率能更好地发掘程序的潜在错误。比如，在图 5.1中，如果 condition 为 True，则行覆盖率已经是 100% 了。实际上如果 condition 为 False，该程序存在错误。这是分支覆盖率的优越之处。

分支覆盖率又包含两部分：（1）分支语句的执行率；（2）被执行到的分支语句的两条边的选择执行率。

（3）条件覆盖率

条件是指一个逻辑条件表达式中不包含逻辑运算的最小操作数部分。条件覆盖率则是报告条件的 True 或 False 情况。只要一个条件具备 True 或 False 两种取值，则其覆盖度达到 100%，与整个逻辑表达式无关。它和分支覆盖率相似，但对控制流更加敏感。比如，对于表达式 if (a && false)，无论 a 如何取值，if 中条件都不会满足，因此其分支覆盖率只能到 50%，但是条件覆盖率可以有 100%。

（4）函数调用覆盖率

函数调用覆盖率用于报告是否调用了每个函数。它对于保证模块之间的接口正确性非常有益。

代码覆盖率通常被用来进行动态代码分析、软件故障预测、软件可靠性评价、测试集测试充分性衡量等 [Marré 等, 1996]。在我们的测试中，采用 GCOV[Wikipedia contributors, 2021c] 工具对源码进行统计。GCOV 包含的覆盖率指标有行覆盖率、两种分支覆盖率和函数调用覆盖率。测试目标代码为 XQM 中 X86 转 MIPS 的核心翻译代码。

5.2 测试方法

5.2.1 硬件配置

本实验在装配 Loongson-3A4000 处理器的台式机上完成，该处理器以及系统的配置情况如下：

- 主频：1800MHz
- 架构：mips64
- CPU 核数：4
- 单核线程数：1
- 内存：8GB
- L1d 和 L1i Cache：64KB

- L2 Cache：256KB
- L3 Cache：8MB
- 内核版本：Linux 3.10.84

5.2.2　测试集

在评测生成的测试集质量时，我们使用常规测试集进行对比。我们选用的常规测试集包括典型的单元测试集 DBT5_UT 和较大型集成测试集 SPEC CPU2000。

本文生成的测试集包含 352 个针对单指令翻译函数的单元测试，6 个针对栈操作优化翻译的单元测试，14 个针对 EFLAGS 延迟计算的单元测试，以及 23 个针对扩展模式优化翻译的测试程序，合计 395 个。

DBT5_UT 一共包含了 332 个测试，根据算术运算、数据传输等指令类型分类如表 5.3。

表 5.1 DBT5_UT 测试用例分布

Table 5.1 Test Case Distribution of DBT5_UT

指令类别	单元测试文件数	描述
arith	13	通用型算术运算指令
mov	6	通用型数据传输指令
logic	13	通用型逻辑运算指令
string	20	字符串操作指令
eflag	26	状态位操作指令
jcc	37	跳转指令
farith	33	X87 浮点算术运算指令
fldst	10	X87 浮点数据传输指令
fctrl	7	X87 浮点控制指令
mmx	24	MMX 扩展指令集
xmm	143	SSE，SSE2 扩展指令集

SPEC CPU2000 测试集包含了 12 个定点测试和 14 个浮点测试，表 5.2 列举了各测试程序功能描述。

表 5.2 SPEC CPU2000 基准测试程序

Table 5.2 SPEC CPU2000 Benchmarks

名称	描述	名称	描述
164.gzip	数据压缩	168.wupwise	量子色动力学
175.vpr	FPGA 电路布局和布线	171.swim	浅水模型
176.gcc	C 编译器	172.mgrid	三维势场中的多网格求解
181.mcf	最小费用网络流	173.applu	抛物线/椭圆偏微分方程
186.crafty	象棋	177.mesa	3D 图形库
197.parser	自然语言处理	178.galgel	流体动力学
252.eon	光线追踪	179.art	神经网络模拟
253.perlbmk	Perl 语言	183.equake	有限元模拟
254.gap	计算群论	187.facerec	人脸识别
255.vortex	面向对象数据库	188.ammp	计算化学
256.bzip2	数据压缩	189.lucas	质数检测
300.twolf	布局布线模拟	191.fma3d	有限元失效模拟
		200.sixtrack	粒子加速器模拟
		301.apsi	环境气象相关计算

5.2.3 目标程序

本文测试的目标程序为 XQM 的核心代码，其代码分布如下表。

表 5.3 目标程序分组

Table 5.3 Classification of Target Source Program

代码分类	功能描述	有效代码量级（行）
flag_rdtn	EFLAGS 线性分析优化翻译	<100
flag_ptn	EFLAGS 延迟计算优化翻译	300+
extension	扩展模式优化翻译	400+
ir1	X86 操作数处理相关	600+
ir2	MIPS 指令生成相关	1000+
int_translate	通用型指令翻译函数	4000+

fp_translate	浮点指令翻译函数	1000+
translate	翻译引擎	1000+

5.2.4　编译选项

测试程序的编译在 X86 平台上完成, 汇编器选择 NASM-2.16rc0, C 编译器选择 GCC-9.3.0。编译时自定义链接脚本, 指定数据段装载地址为 0x09000000。根据第 3.5 节可知，编译测试程序的过程中经过两次链接，第二次链接加入了标准输出。标准输出的数据量较大导致之后程序装载时数据段地址和第一次执行时不一致，因此需要指定其装载地址在一个固定位置。SPEC CPU2000 编译时编译参数为-m32 -O2 -static -mno-sse -mno-sse2。DBT5_UT 编译时编译参数为-m32 -static。XQM 在配置时添加配置参数-enable-gcov -gcov=/usr/bin/gcov 。

5.2.5　测试过程

测试时，依次添加以下优化参数运行测试集所有程序:（1）tr-bh，单指令手工优化翻译;（2）flag-ptn，EFLAGS 延迟计算优化翻译;（3）flag-rdtn，EFLAGS 线性分析优化翻译；（4）tb-link，基本块链接优化技术；（5）ss，影子栈优化技术；（5）xmm128map，本地向量指令优化翻译；（6）不添加任何优化选项。

本次测试中, 为了避免 SPEC CPU2000 的执行时间过长, 其输入集选择数据量中等的 Train。SPEC CPU2000 和 DBT5_UT 的测试程序都能正确执行。BTRTG 生成的测试集中有部分发生崩溃, 程序崩溃后, 则其相应的覆盖率无法被 GCOV 工具统计。由于本文生成的测试中 SSE 和 SSE2 扩展指令集单元测试崩溃较多, 因此不对比扩展指令翻译代码的覆盖率。

5.3　覆盖率提升

图 5.2为本文测试集相对 DBT5_UT 的覆盖率提升。由图可见，本文测试集在优化翻译、X86 操作数处理、通用型指令翻译函数和翻译器引擎部分的代码覆盖率有明显提升。其中，对 Flag-Pattern 优化部分代码的覆盖率提升了 20% 左右，这主要是因为 DBT5_UT 中程序格式单一，缺乏组合性。本文生成的测试集根据优化算法的特征生成符合优化方式的指令组合从而提升了覆盖率。在通用型指令翻译函数部分的覆盖率提升主要是因为部分新增的指令翻译函数在

DBT5_UT 中没有相应的测试。本文测试集在 MIPS 代码生成部分的覆盖率则不如 DBT5_UT，主要原因是本文测试集中对扩展指令集的测试大量崩溃导致了与其相关的 MIPS 代码生成部分没有被统计到。两者对 Flag-Reduction 优化的覆盖率完全相同，这是因为该优化本身的代码量非常少，不足 100 行。

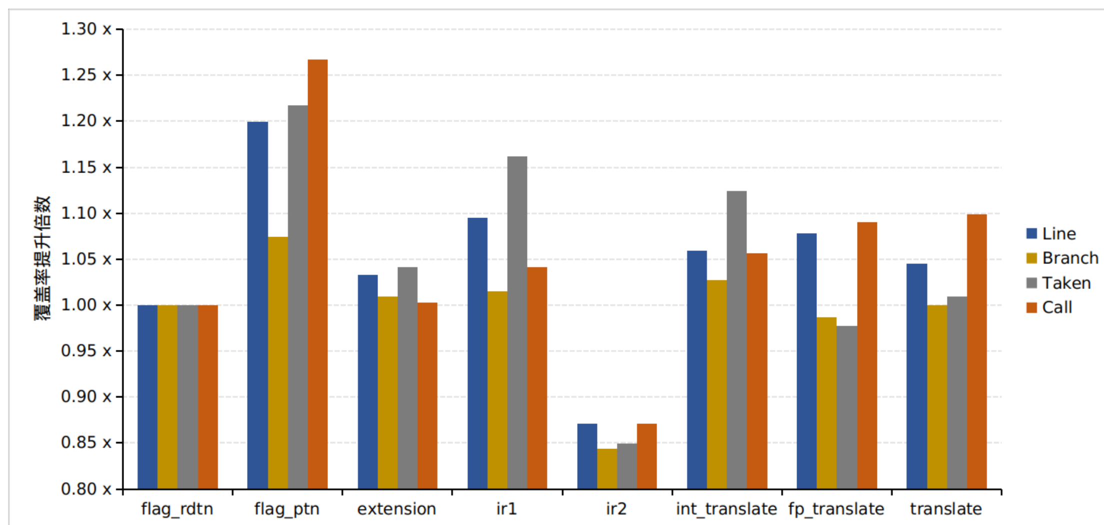

图 5.2 BTRTG 和 DBT5_UT 的覆盖率对比

Figure 5.2 Coverage Comparison of BTRTG and DBT5_UT

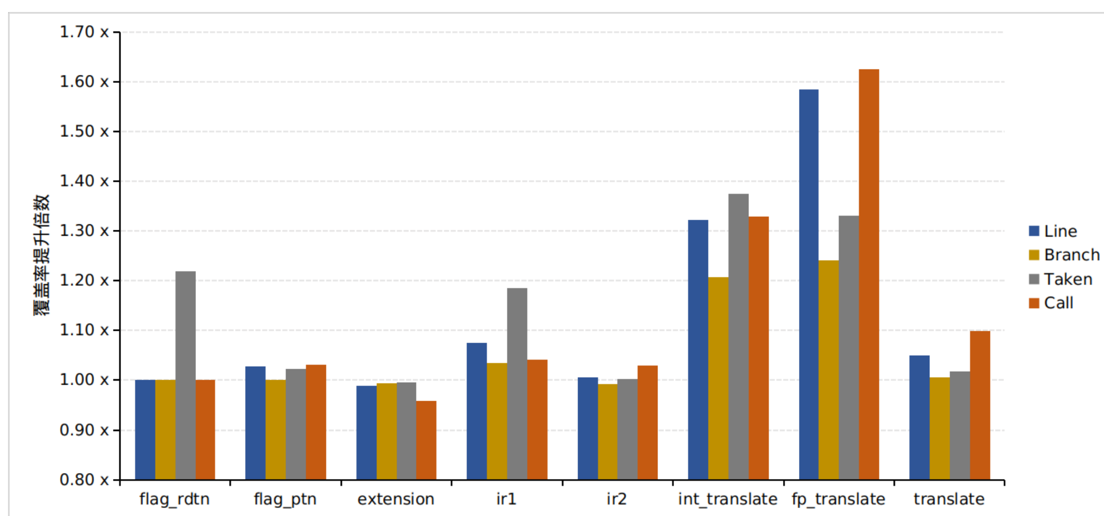

图 5.3 BTRTG 和 SPEC CPU2000 的覆盖率对比

Figure 5.3 Coverage Comparison of BTRTG and SPEC CPU2000

图 5.3为本文测试集相对 SPEC CPU2000 的覆盖率提升。由结果可知，本文测试集在各个模块的覆盖率几乎都超越或者持平 SPEC CPU2000。定点和浮点翻译函数部分更是提升了 30% 左右，这是因为 SPEC CPU2000 代码量虽然多，但

是其包含的指令种类并不多。而在 X86 操作数处理部分，本文测试集也有接近 5% 的覆盖率提升。

综合以上结果，可知 DBT5_UT 在单指令翻译上测试覆盖率较高，而 SPEC CPU2000 在优化翻译上测试覆盖率较高。本文测试集在各种翻译情形下都有较高的测试覆盖度，相对均匀。

5.4　纠错能力

在本文测试集的测试中，新发现了大量二进制翻译器的错误。其中，已修复的有 16 处错误，详情见第 4.2节。浮点运算部分存在 64 位双精度和 80 位扩展双精度的数据表示转换问题以及计算的精度问题。另外还有一些未知错误导致了部分 MMX、SSE、SSE2 指令单元测试程序的崩溃，详情见第 4.1节。

5.5　本章小结

本章使用 BTRTG 生成了一部分单元测试集和优化翻译测试集，并对该测试集进行了评测。本章先介绍了覆盖率测试的各覆盖率指标的概念，然后介绍实验环境以及实验过程。为了对本文测试集有一个大致的评估，本章将其与传统测试集 DBT5_UT 和 SPEC CPU2000 进行了对比。对比发现，本文测试集的对代码的各个模块都有较高的覆盖率。另外，保守的估计，该测试集检测出了翻译器系统中二十处以上新的错误，这充分反映了其高效的纠错能力。

第 6 章　总结与展望

6.1　本文工作总结

二进制翻译器在应用软件的跨平台兼容中发挥了重要作用。开发一个二进制翻译器工程量庞大，其正确性保证非常困难。优质的测试程序能有效发现翻译器中的错误并加快开发进度。过去手写测试集的方式扩展性差、工程量大，而已有的测试集又因为缺少对二进制翻译的针对性而效率低。已有的在随机化测试生成方面的工作存在面向测试场景范围窄、操作数生成不全面等缺点。因此，开发一种能灵活面向各种测试场景的随机化测试生成器尤为重要。

本文参照芯片的验证平台，提出并实现了一个面向二进制翻译的随机化测试生成器。该生成器在测试人员设计的测试模板的引导下生成随机的 X86 汇编级测试程序。生成的测试简洁且具有自我验证的功能，方便于调试。

本文针对二进制翻译器开发中典型的四个测试场景（两个单元测试和两个优化测试）设计了测试模板并进行了测试，这些实验对生成器各项功能进行了展示：（1）生成器能根据指定常数或随机数生成器为指令的操作数提供丰富的输入值，该功能利于发现指令运算中的错误。（2）丰富的操作数生成功能能有效发现翻译过程中操作数转换时的错误。（3）广泛地支持各种指令和灵活的指令组合生成语句能引导生成符合各种优化翻译的代码模式。

最终，本文生成的测试在 XQM 中发现了多处错误，且对 XQM 核心代码有比较高且均衡的覆盖度。

6.2　未来工作展望

随机化测试生成器中模板语言的表达能力的强大与否决定了其在各种测试场景下的适应程度。本文仅测试了几种典型测试场景。未来随着测试面的增加，应当添加更多的模板引导语句，最终让我们的模板语言接近普通编程语言的表达力。

指令生成引擎需要被进一步优化。目前为了保证生成指令的合法性，在指令前加入了一些辅助性指令，比如初始化内存地址相关寄存器等，这在一定程度上减弱了指令组合随机度。未来可以通过辅助性指令和非法指令回退相结合的方

式：设定一个指令反复生成次数阈值，在该阈值以下使用非法指令回退方式，超过该阈值则采用注入辅助指令方式。此外，操作数之间的依赖关系需要更深入的探究。生成器应当能够按照某种关联模式生成操作数。

系统态的测试生成相对用户态测试生成更为复杂，因为很多特权级指令不能进行随机生成。这些不能随机生成的指令就需要通过静态封装的方式嵌入到测试中。系统态的测试程序样例应当是一个个微型操作系统，能写入镜像并在裸机上启动。本人参考 IncludeOS[Bratterud 等, 2015] 对微小镜像的生成进行了探索，认为系统态下的测试需要通过一些手写代码引导系统进入某个特定状态之后再注入随机代码。比如，我们可以参照 Dune[Belay 等, 2012] 中提到的轻量级页表将系统引导进入具有内存管理单元的保护模式，从而测试保护模式下的功能正确性。

参考文献

唐锋, 武成岗, 冯晓兵, 等. 基于动态反馈的标志位线性分析算法 [D]. 2007.

马湘宁, 武成岗, 唐锋, 等. 二进制翻译中的标志位优化技术 [J]. 计算机研究与发展, 2005, 42 (2): 329.

Adir A, Almog E, Fournier L, et al. Genesys-pro: Innovations in test program generation for functional processor verification [J]. IEEE Design & Test of Computers, 2004, 21(2): 84-93.

Aharoni M, Asaf S, Fournier L, et al. Fpgen-a test generation framework for datapath floating-point verification [C]//Eighth IEEE International High-Level Design Validation and Test Workshop. IEEE, 2003: 17-22.

Aharony M, Gofman E, Guralnik E, et al. Injecting floating-point testing knowledge into test generators [C]//Haifa Verification Conference. Springer, 2011: 234-241.

Bala V, Duesterwald E, Banerjia S. Dynamo: A transparent dynamic optimization system [C]// Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. 2000: 1-12.

Belay A, Bittau A, Mashtizadeh A, et al. Dune: Safe user-level access to privileged {CPU} features [C]//10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). 2012: 335-348.

Bellard F. Qemu, a fast and portable dynamic translator. [C]//USENIX annual technical conference, FREENIX Track: volume 41. Califor-nia, USA, 2005: 46.

Bratterud A, Walla A A, Haugerud H, et al. Includeos: A minimal, resource efficient unikernel for cloud services [C]//2015 IEEE 7th international conference on cloud computing technology and science (cloudcom). IEEE, 2015: 250-257.

Burgess C J, Saidi M. The automatic generation of test cases for optimizing fortran compilers [J]. Information and Software Technology, 1996, 38(2): 111-119.

Cifuentes C, Van Emmerik M. Uqbt: Adaptable binary translation at low cost [J]. Computer, 2000, 33(3): 60-66.

Corporation I. Intel® 64 and ia-32 architectures software developer’s manual [J]. Volume 1: Basic Architecture, 2016, 1(253665).

Corporation I. Intel® 64 and ia-32 architectures software developer’s manual [J]. Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference A-Z, 2016, 1(325383).

Dasgupta S, Park D, Kasampalis T, et al. A complete formal semantics of x86-64 user-level instruction set architecture [C]//Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2019: 1133-1148.

Dehnert J C, Grant B K, Banning J P, et al. The transmeta code morphing/spl trade/software: using speculation, recovery, and adaptive retranslation to address real-life challenges [C]//International Symposium on Code Generation and Optimization, 2003. CGO 2003. IEEE, 2003: 15-24.

Di Federico A, Payer M, Agosta G. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries [C]//Proceedings of the 26th International Conference on Compiler Construction. 2017: 131-141.

Ebcioğlu K, Altman E R. Daisy: Dynamic compilation for 100% architectural compatibility [C]// Proceedings of the 24th annual international symposium on Computer architecture. 1997: 26-37.

Fowler M, Foemmel M. Continuous integration [J]. Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf, 2006, 122(14): 1-7.

Gal-On S, Levy M. Exploring coremark a benchmark maximizing simplicity and efficacy [J]. The Embedded Microprocessor Benchmark Consortium, 2012.

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.

Hennessy J L, Patterson D A. Computer architecture: a quantitative approach [M]. Elsevier, 2011.

Hong D Y, Hsu C C, Yew P C, et al. Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores [C]//Proceedings of the Tenth International Symposium on Code Generation and Optimization. 2012: 104-113.

Hookway R J, Herdeg M A. Digital fx! 32: Combining emulation and binary translation [J]. Digital Technical Journal, 1997, 9: 3-12.

Jiuye C, Wu Y, Boye S, et al. Automatic validationforbinarytranslation [J]. Computer Languages Systems& Structures, 2015, 43.

Kedia P, Bansal S. Fast dynamic binary translation for the kernel [C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 101-115.

Lichtenstein Y, Malka Y, Aharon A. Model based test generation for processor verification [J]. 1994.

Lindholm T, Yellin F, Bracha G, et al. The java virtual machine specification [M]. Pearson Education, 2014.

Lindig C. Random testing of c calling conventions [C]//Proceedings of the sixth international symposium on Automated analysis-driven debugging. 2005: 3-12.

Loosemore S, Stallman R M, Oram A, et al. The gnu c library reference manual [J]. 1993.

Marré M, Bertolino A. Reducing and estimating the cost of test coverage criteria [C]//Proceedings of IEEE 18th International Conference on Software Engineering. IEEE, 1996: 486-494.

Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation [J]. ACM Sigplan notices, 2007, 42(6): 89-100.

Phansalkar A, Joshi A, Eeckhout L, et al. Measuring program similarity: Experiments with spec cpu benchmark suites [C]//IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE, 2005: 10-20.

Song C, Wang W, Yew P C, et al. Unleashing the power of learning: An enhanced learning-based approach for dynamic binary translation [C]//2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19). 2019: 77-90.

Spink T, Wagstaff H, Franke B. A retargetable system-level dbt hypervisor [J]. ACM Transactions on Computer Systems (TOCS), 2020, 36(4): 1-24.

Wells N. Busybox: A swiss army knife for linux [J]. Linux Journal, 2000, 2000(78es): 10-es.

Wikipedia contributors. Transmeta — Wikipedia, the free encyclopedia [EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=Transmeta&oldid=1013880468.

Wikipedia contributors. Rosetta (software) — Wikipedia, the free encyclopedia [EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=Rosetta_(software)&oldid=1014563058.

Wikipedia contributors. Gcov — Wikipedia, the free encyclopedia [EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=Gcov&oldid=998214638.

Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in c compilers [C]//Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011: 283-294.