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

硕士学位论文

基于 LLVM 编译优化制导的二进制翻译系统

作者姓名：

指导教师：

学位类别：电子信息硕士

学科专业：计算机技术

培养单位：中国科学院计算技术研究所

2023 年 6 月

Binary translation system based on LLVM compilation optimization guidance

A thesis submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Electronic and Information Engineering

in Computer Technology

By

Supervisor: Professor

Institute of Computing Technology, Chinese Academy of Sciences

June, 2023

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。

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有关保存和使用学位论文的规定，即中国科学院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

一款指令集的成功与否不仅取决于芯片的硬件性能，也严重依赖软件生态的丰富程度。二进制翻译可以将一种源指令集翻译为语义等价的目标指令集，已经成为了跨架构软件兼容的重要手段。然而，主流二进制翻译以动态翻译为主，较难实施复杂的运行时优化，因而效率较低。而动静结合的翻译方式可以进行离线优化，同时兼具动态翻译的完备性，具有较高的研究价值。

本文以 QEMU 为基础，利用 QEMU 完备的运行时逻辑实现了系统的动态端，同时借助 LLVM 完善的分析与优化框架开发了高效的静态端，最终实现了一个以 LLVM 编译优化制导的动静结合二进制翻译系统 COGBT。该系统在翻译性能上相较 QEMU 取得了明显提升：SPEC 2000 CINT 测试提升了 80.9%，CoreMark 测试提升了 92.8%。此外该系统将翻译与优化完全隔离开来，可基于此系统便利地开发各类优化 pass、实验各种编译优化方案 (如 PGO)。本文主要工作及贡献如下：

1. 设计并实现了一个基于 QEMU 和 LLVM 的动静结合二进制翻译系统。该翻译系统利用 LLVM 对翻译代码实施离线优化，相比纯 QEMU 取得了约 80.9% 的性能提升。

2. 提出了一个基于 LLVM 栈模式的翻译方案。该方案利用 LLVM IR 的栈来缓存客户 CPU 的寄存器状态，通过 LLVM 后续的栈提升优化 (mem2reg) 减少了访存指令的数量。

3. 提出了在 LLVM 上实现寄存器映射的方式。该方案利用内联约束将客户 CPU 的部分寄存器绑定到宿主机 CPU 的寄存器降低了翻译代码客户寄存器加载与同步的访存开销。

4. 实现了在 LLVM IR 上的 LBT(LoongArch Binary Translation) 硬件辅助优化以及基本块链接优化。利用 LoongArch 特有的 eflags 辅助计算指令减少了标志位计算的翻译代码，利用基本块链接技术降低了上下文切换开销。

关键词：二进制翻译，LLVM 编译器，编译优化，预先翻译

Abstract

The success of an instruction set depends not only on the hardware performance of the chip, but also on the richness of the software ecosystem. Binary translation, which translates a source instruction set into another semantically equivalent target instruction set, has become an important tool for cross-architectural software compatibility. However, Most of the mainstream binary translators use the dynamic translation scheme, making it difficult to implement complex runtime optimizations and therefore less efficient. The combination of dynamic and static translation allows for offline optimization and has the completeness of dynamic translation, which is of high research value.

This paper is based on QEMU, and makes use of QEMU's complete runtime processing logic to implement the dynamic side of the system, while developing an efficient static side with the help of LLVM's comprehensive analysis and optimization framework. We finally implemented a dynamic and static binary translation system COGBT, which is guided by LLVM compile optimization. The system achieves significant improvement in translation performance compared to QEMU: 80.9% in SPEC 2000 CINT test and 92.8% in CoreMark test. In addition, the system completely isolates translation from optimization, allowing for easy development of various optimization passes and experimentation with various compile optimization schemes(like PGO). The main work and contributions of this paper are as follows:

1. We implemented a hybrid binary translation system based on QEMU and LLVM. The system uses LLVM to implement offline optimization and achieves about 80.9% performance improvement compared to QEMU.

2. We propose a translation scheme based on the LLVM stack model. The scheme uses the stack of LLVM IR to cache the guest register state, and reduces the number of memory access instructions by subsequent stack promotion optimization of LLVM (mem2reg).

3. We propose a way to implement register mapping on LLVM based binary translators. By using inline constraints to bind guest registers to host registers, we can reduce

the overhead of loading and writing back guest register states.

4. We implemented LBT (LoongArch Binary Translation) hardware-assisted optimization and basic block linking optimization on LLVM IR. The translation code for flag bit computation is reduced by using LoongArch-specific LBT instructions, and the context switching overhead is reduced by using the basic block link optimization.

Keywords: Binary Translation, LLVM Compiler, Compile Optimization, AOT

目 录

图形列表

表格列表

第 1 章　引言

1.1　研究背景

随着通用计算的发展逐渐放缓，针对特定应用场景的领域特定架构 (Domain Specific Architecture，DSA) 逐步兴起，计算机体系结构的发展又一次迎来了黄金年代 [1]。以 NPU、DPU 等为代表的新型器件在特定领域取得了以较小功耗提升数倍性能的成果，验证了 DSA 的优越性。在体系结构朝着领域化、专用化的方向发展的趋势下，指令集架构也朝着多样化的方向发展 [2]。

指令集架构 (Instruction Set Architecture，ISA) 是硬件 (通常来说是指 CPU) 实现的一组原语，可直接在硬件上执行。这组原语同时也是软件与硬件之间的接口，约定了两者之间的交互方式。各类 CPU 根据自身设计目标以及实现方式的不同，开发了各种符合自身使用场景的指令集架构。如 X86 指令集指令复杂、功能丰富，适合高性能计算场景；ARM 指令集指令精简、能耗比出众，适合移动计算；LoongArch 指令集自带二进制翻译扩展，适合作为兼容指令集等。除了以上列举的几种指令集，目前存在的指令集尚有二十几种[1]之多。

尽管目前推出的指令集不少，但是软件生态却只分布在少数指令集手中。X86 指令集由于推出时间较早，性能优异，已经占据了桌面端生态。而 ARM 指令集由于能耗比出众，移动端生态优势明显。新兴指令集则由于推出时间较晚，软件适配尚未完成，其发展面临着软件匮乏的窘境。

龙芯中科于 2021 年推出了其自主设计的新一代处理器 3A5000。该处理器采用了最新的 LoongArch 指令集，通过指令的合理设计减少了动态指令数，性能相较上一代 MIPS 处理器提升了 50%[3]，为国家信息化体系的建设和发展注入了动力。尽管在硬件上取得较大突破，龙芯处理器在软件生态方面依然面临软件不足的问题。因此对开发跨架构软件兼容的二进制翻译器提出了以下需求：

1. 二进制翻译器应该足够高效。作为软件的跨架构兼容方案，用户最关心的是经过翻译器翻译后的软件是否能有原生软件接近的性能。性能差异过大将影响用户体验，不利于指令集的推广。

2. 二进制翻译器应该安全完备。正确性是二进制翻译最基本的前提，经过

[1]数据统计自 Linux Arch 目录下支持的指令架构数量

翻译后软件的行为应该与原生软件的行为一致，如此才能实现对用户透明。

3. 二进制翻译器应该易于维护，最好可重定目标。二进制翻译器作为一种较大型底层软件，其开发维护难度较大，给每种指令集都开发一款二进制翻译器需要极大的人力物力投入，如果翻译器可重定目标，就能减轻一部分后端维护负担，加快软件的迭代速度。

1.2 二进制翻译概述

为了解决软件迁移这一难题，二进制翻译技术得到了广泛的应用和发展 [4]。二进制翻译技术是一种指令集虚拟化技术，可以将客户 (Guest) 指令集翻译成语义等价的宿主 (Host) 机指令集 [5]。由于翻译的透明性，对于客户程序而言，其运行在二进制翻译系统上的状态将与在真实客户机上的状态是完全一致，并且这个过程对于客户程序是无感知的。

传统上根据是否对客户程序实施翻译可以将软件模拟分为解释型和翻译型 [6]。翻译根据翻译时间又可细分为动态翻译与静态翻译 [7]。解释器不翻译指令，而是通过宿主机上的函数来模拟客户指令的行为与语义，因而其实现较为简单但是效率有所欠缺，主流模拟器较少采用这种做法，大多用于调试或者辅助运行。翻译器将客户指令翻译成一条或多条宿主机指令，并将翻译结果缓存起来，可以做到一次翻译多次复用，因此具有较高的效率，是主流的实现方式，但是实现难度相对较高。

在二进制翻译器中，动态翻译 [8] 属于运行时翻译，只有当客户程序的指令实际要被执行到时翻译器才会实施翻译，并且由于翻译是运行时进行的，翻译器可以获取运行时信息，因此可以处理自修改、代码数据混淆等难题。然而，因为翻译与优化的开销占据运行时开销，动态翻译器一般难以实施复杂的运行时优化。

静态翻译属于离线翻译，翻译器通过程序分析技术静态地提取客户程序的指令，并离线翻译、优化这些指令，因此可以实施较为复杂的优化，但是在自修改代码以及代码数据混淆问题上，纯静态二进制翻译器也无能为力。

鉴于动态翻译与静态翻译在效率与完备性上互有优劣，针对动静结合的二进制翻译系统的研究已经成为了目前二进制翻译一个比较火热的方向。这种混合翻译系统利用静态端对热点代码做激进优化提升效率，同时利用动态端处理

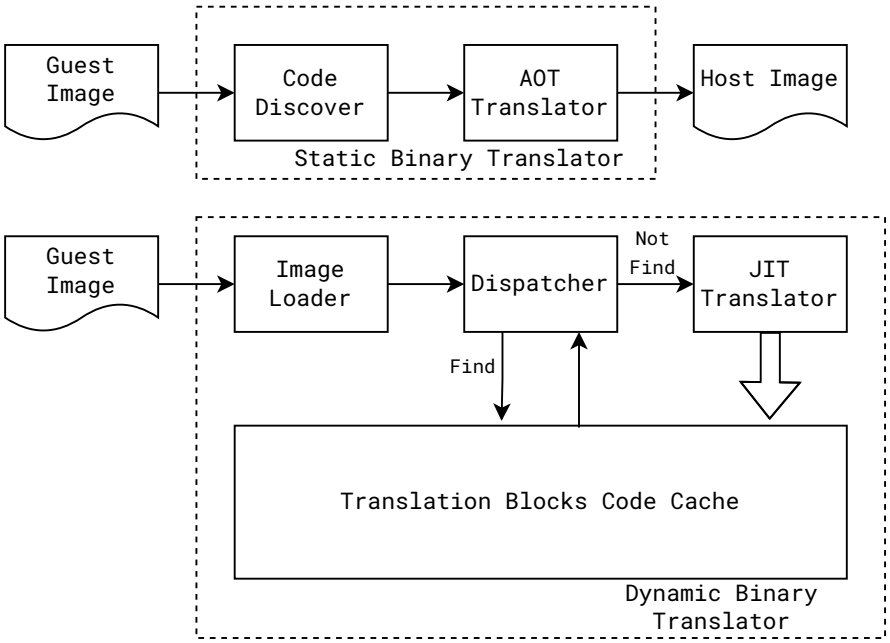


图 1.1 静态翻译与动态翻译基本框架

自修改等问题。一个典型的例子是苹果公司的 Rosetta2 翻译器 [9]，这是一个从 X86 翻译到 ARM 的动静结合翻译器。该翻译器先使用 AOT(Ahead Of Time) 技术将 X86 程序进行离线翻译，将翻译结果缓存到特定的 AOT 文件，之后执行的时候再加载这些预翻译文件，对于静态翻译阶段没有覆盖到的代码，Rosetta2 将运行动态端进行在线翻译，以此来保证正确性。已有的资料显示 Rosetta2 的翻译方式也是使用逐指令直接翻译 [10, 11]，得益于 ARM 架构与 X86 架构在指令集上的相似性，这种做法也可以做到较好的翻译效果。但是对于指令集相差较大的两种架构，如 X86 和 LoongArch，直接翻译的方式将较难做到跨指令优化。

图 1.2 展示了逐指令翻译的不足。图 1.2a 是几条地址接近的 X86 访存指令，翻译的 LoongArch 指令首先计算访存地址，然后实施访存。从单一指令翻译看来，计算地址是必须的且无法消除，但是如果实施跨指令分析可以发现指令 *lu12i.w $t0, 0x6cf* 属于公共子表达式，只需计算一次。图 1.2b 则展示了逐指令翻译在跨指令活动变量分析上的不足。*div r14* 需要将 *rdx/rax* 的商与余数分别存放到 rax 与 rdx 中，逐指令翻译时无法了解商与余数是否会被使用，因此需要翻译商和余数的计算过程。然而通过活动变量分析可知，后续的 *imul rax* 只用到了 rax，并且重新定值了 rdx，因此实际不需要计算第一条除法指令的余数，翻译存在冗余计算。

上述两个例子表明，在实施 X86 到 LoongArch 的翻译过程中，逐指令翻译存

```
X86:
0x0000402fda: mov dword ptr [rip + 0x2cc268], eax
0x0000402fe0: mov dword ptr [rip + 0x2cc266], ebx
0x0000402fe6: mov dword ptr [rip + 0x2cc264], ecx
0x0000402fec: mov dword ptr [rip + 0x2cc262], edx

LoongArch:
0xffd00019d0: lu12i.w   t0,0x6cf  //common subexpression
0xffd00019d4: ori       t0,t0,0x248
0xffd00019d8: st.w      $s0,t0,0
0xffd00019dc: lu12i.w   t0,0x6cf  //common subexpression
0xffd00019e0: ori       t0,t0,0x24c
0xffd00019e4: st.w      $s3,t0,0
0xffd00019e8: lu12i.w   t0,0x6cf  //common subexpression
0xffd00019ec: ori       t0,t0,0x250
0xffd00019f0: st.w      $s1,t0,0
0xffd00019f4: lu12i.w   t0,0x6cf  //common subexpression
0xffd00019f8: ori       t0,t0,0x254
0xffd00019fc: st.w      $s2,t0,0
```

(a)

```
X86:
0x0000403493: div  r14 //rdx:rax/r14->rax, rdx:rax%r14->rdx
0x0000403496: imul rax //rax*rax->rdx:rax
0x000040349a: sub  rdi, rax
0x000040349d: add  rdi, rcx
0x00004034a0: mov  rsi, r13
0x00004034a3: mov  qword ptr [r9 + 0x440], rax
0x00004034aa: mov  rdx, r12 //r12->rdx

LoongArch:
0xffd000d42c: add.d     $t0, $t7, $zero
0xffd000d430: mod.du    $s2, $s0, $t0   // dead define
0xffd000d434: div.du    $s0, $s0, $t0
              ...
0xffd000d498: mulh.du   $s2, $s0, $s0   // dead define
0xffd000d49c: mul.d     $s0, $s0, $s0
0xffd000d4a0: sub.d     $t0, $s7, $s0
0xffd000d4a4: or        $s7, $t0, $zero
0xffd000d4a8: x86add.d  $s7, $s1
0xffd000d4ac: add.d     $s7, $s7, $s1
0xffd000d4b0: or        $s6, $t6, $zero
0xffd000d4b4: st.d      $s0, $a3, 1088
0xffd000d4b8: or        $s2, $t5, $zero
```

(b)

图 1.2 逐指令翻译的不足。(a) 存在跨指令的公共子表达式，(b) 存在冗余计算

在不足，通过跨指令翻译存在进一步优化的可能。然而要实施跨指令翻译优化需要解决两个问题：其一是需要有较强表达能力的 IR(Intermediate Representation) 作为分析优化的基础；其二是需要有比较完善的分析与优化工具。当 IR 表达能力较弱时，翻译器一方面无法携带足够的客户指令信息供后续分析优化，另一方面需要较多 IR 才能表达指令语义，增加了翻译时负担与出错概率，无法满足开发高效二进制翻译器的需求。典型的例子是 QEMU 的 TCG IR[12]。TCG 几乎不支持浮点、向量，在翻译较复杂的定点指令以及浮点向量指令时需要调用较多辅助函数，制约了其效率；此外，缺乏 IR 的分析与优化工具将迫使翻译器进行大量基础设施开发。这将延长翻译器的开发周期，不利于软件的快速迭代。

由于 LLVM 的出现，以上两个问题都可以得到较好的解决。

1.3　LLVM 编译框架概述

LLVM[13] 是由一系列编译工具与分析优化库组成的编译基础设施。通过引入 LLVM IR 将编译器的前后端解耦，并且通过提供大量独立、完善的模块供开发者使用减轻了开发者的负担。图 1.3 展示了其基本架构。前端使用 clang 可将高级语言转化为 LLVM IR 的中间表示，并且可在这阶段实施较复杂的语法分析和语义检查；中端是 LLVM 设计的一种架构无关中间表示，可在其上实施各种架构无关的优化；后端则是架构相关的代码生成部分，经过指令选择、指令调度、寄存器分配以及各种架构相关的优化之后生成宿主机的机器指令。

通过将编译的各个阶段解耦，开发者可以方便地复用 LLVM 的编译库。当开发者需要开发一种新语言之时，只需实现前端的语法解析、语义检查等部分，无需涉及后端架构相关代码；同样地，当添加一种新架构时，开发者也只需要完成后端代码生成。这种良好的架构设计降低了重复开发的工作量，可基于此开发设计所需的功能或优化模块。

LLVM 能实现前后端解耦以及高效优化的关键在于引入了 LLVM IR。LLVM IR 是一种 SSA(Static Single Assignment)[14] 形式的中间表示，这种中间表示只能被定值一次，即不允许对同一个变量多次定值，引入这种限制可以简化 Define-Use 分析关系。假定变量 A 可被多次定值，那么某处对 A 的使用就可能有多个来源，要识别具体的来源就需要较多分析，而 SSA 只有单一来源，进行可达定值 (Reaching Definition) 分析时就十分便利。此外，SSA 在进行活动变量分析来

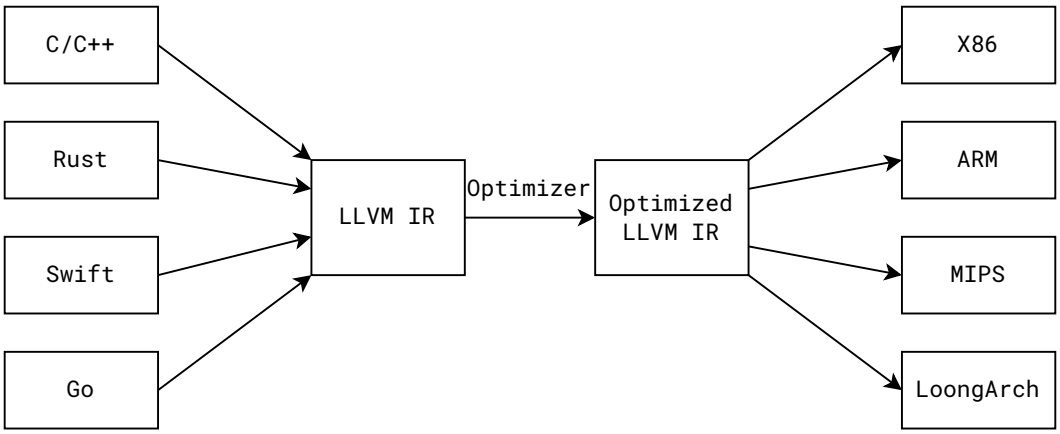


图 1.3 LLVM 基本框架

获取变量的存活范围时具有明显的优势。编译器可以精确地获悉每个变量从定值到最后一次使用之间的存活范围，进行更好的寄存器分配。简而言之，从编译分析与优化的角度看，这种 IR 简化了数据流分析，使得许多优化更容易实现。

在语义表达上，LLVM IR 也可以精确地携带高级语言的语义信息。这主要得益于其精心设计的类型系统与 IR 指令。类型系统在 LLVM IR 的优化与代码生成过程中扮演着重要角色，其强大的表达能力能提供丰富的信息供优化器使用。LLVM IR 含有定点、浮点、向量、指针、结构以及函数等多种类型，基本能以较直接的方式还原高级语言的变量类型。优化器可以利用这些信息实施分析与优化，如指针分析、结构体成员重排等。更为重要的是，前端可以较为方便地将高级语言或者其他代码 (如二进制) 转化为 LLVM IR，减轻了前端的开发负担与出错概率。表 1.1 对比了 QEMU 的 TCG 与 LLVM IR，可以看到 LLVM IR 支持的类型更加丰富。

表 1.1 LLVM IR 与 QEMU TCG 对比

IR	整型	浮点	向量	指针	结构体
LLVM IR	支持任意精度	支持多种精度	支持任意元素数及精度	支持多种类型	支持
QEMU TCG	只支持 32/64/128 位	不支持	只支持 64/128/256 位	只支持定点指针	不支持

除了在类型系统上别出心裁，LLVM IR 指令在设计上也同样考虑周到。除了基本的整型算术、逻辑指令，LLVM IR 也支持浮点与向量有关的运算。值得一提的是，LLVM IR 提供了原子访存类指令，并且提供了 Intrinsic 机制供开发者添加硬件架构相关的特有指令。原子指令可以高效模拟高级语言多线程之间

的同步，而 Intrinsic 机制可方便地添加架构相关的硬件指令实现特有的加速。

从开发者的编程视角上看，LLVM IR 有如下几个优点：

1. IR 有无限虚拟寄存器，无需手动进行寄存器分配。在 LLVM IR 中可以任意使用寄存器，在后续的代码下降过程中，编译器会实施寄存器分配算法，只将最常用的值分配到物理寄存器中，尽可能高效地利用有限寄存器资源。

2. 用户不需要关心底层硬件细节。LLVM IR 作为一种中间表示，其设计初衷就是为了屏蔽底层架构相关的部分，使前端高级语言能方便地映射到 LLVM IR 上。因此，使用 LLVM IR 可以不用考虑繁杂的后端架构，只需将精力集中在前端翻译上。

3. IR 设计简单，使用方便。LLVM IR 中的许多概念与高级语言类似，基本都能和高级语言对应。如 Module 代表高级语言中一个源文件，Function 对应高级语言的函数，Basic Block 对应源代码的基本块等。这种相似性使得开发者很容易上手这种中间表示，降低学习成本。

LLVM IR 的诸多优势使其非常适合做二进制翻译的中间表示，并且基于 LLVM IR 能非常轻易地开发优化 pass。在 LLVM 中，对 IR 的分析或操作均由所谓的 pass 完成，如对变量的存活性分析，对死代码的消除等。要执行某种优化只需编写一个对应的 pass 即可。图 1.4 展示了对 IR 执行优化的基本过程。

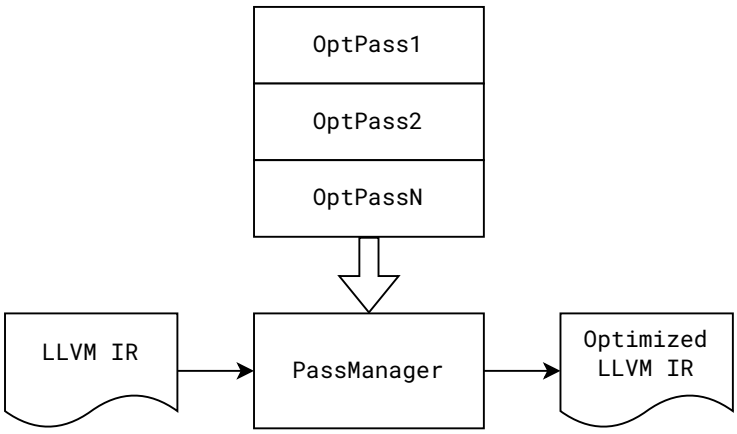


图 1.4 LLVM pass 运行示意图

1. 根据优化对象的大小，选择合适的 pass 类型，并对其进行实现。LLVM 将 pass 分为模块级与函数级，分别作用于 IR 中的模块与函数。

2. 将实现的 pass 加入 PassManager 中。PassManager 将根据 pass 之间的依赖关系合理调度 pass 的运行。

3. 将 PassManager 在需要优化的 IR 上运行，完成 IR 的优化。

1.4 本文主要工作及贡献

二进制翻译作为软件迁移与生态扩充的重要手段，其效率的高低直接影响着新指令集的推广成功与否。然而现有的二进制翻译手段大多以动态翻译为主，效率不高，难以达到用户的需求。以 QEMU[15] 为例，其 X86 到 LoongArch 的定点翻译效率在 SPEC CPU 2000 上只有 16.5%，这意味着在主频为2.5GHz的龙芯 CPU 上执行 X86 程序只有大约 412MHz 的性能。鉴于动静结合翻译系统能离线实施复杂优化，其潜在翻译效率有较高的探索空间。因此，本文实现了一个基于 LLVM 优化制导的动静结合二进制翻译系统 COGBT，相关工作及贡献如下：

1. 设计并实现了一种基于 QEMU 和 LLVM 的动静结合二进制翻译系统。该翻译系统利用 LLVM 对翻译代码实施离线优化，相比纯 QEMU 取得了约 80.9% 的性能提升。

2. 提出了基于 LLVM 栈模式的翻译方案。该方案利用 LLVM IR 的栈来缓存客户 CPU 的寄存器状态，通过 LLVM 后续的栈提升优化 (mem2reg) 减少了访存指令的数量。

3. 提出了在 LLVM 上实现寄存器映射的方式。利用内联约束将客户 CPU 的部分寄存器绑定到宿主机 CPU 的寄存器可降低翻译代码客户寄存器加载与同步的访存开销。

4. 实现了在 LLVM 上的 LBT(LoongArch Binary Translation) 硬件辅助优化以及基本块链接优化。利用 LoongArch 特有的 EFLAGS 辅助计算指令减少了标志位计算的翻译指令，利用基本块链接技术降低了上下文切换开销。

此外，COGBT 具有较强的扩展性，可以为后续的优化提供平台基础。这主要体现在以下几个方面：

1. 基于本系统可以便利地开发优化 pass：LLVM 具有一套完善的 pass 管理机制，对 pass 的调度、运行由内部的 PassManager 完成。为翻译后的 LLVM IR 实行某种特定优化只需编写一个 pass 并添加到 PassManager 即可。这种翻译、优化的隔离模式大大降低了翻译器的开发难度，开发者只需将精力集中在设计优化 pass 上，而无需担忧添加优化对原有的翻译造成破坏或者调整。

2. 基于本系统可以直接复用 LLVM 通用优化 pass：LLVM 的优化模块提供

了大量可直接使用的优化 pass。如用于实施指令融合的instcombine、用于简化控制流图的simplifycfg、以及用于做内存提升的mem2reg等。相较于其他二进制翻译系统，开发者将不再需要重复性地实现早已成熟稳定的优化代码，因此可以显著降低开发工作量与调试难度。

3. 基于本系统可以重定向目标架构 (retargetable)：对于采用直接翻译的二进制翻译器来说，其翻译过程将 Guest 指令直接翻译成了具体的 Host 指令。当宿主机指令发生变更，则需要重新实现一遍翻译过程，不利于做架构的迁移。借助于 LLVM IR，本系统可以不需要关心具体的目标架构，仅需要实现 Guest 指令到 LLVM IR 的翻译即可，当目标架构发生变化，只需重新指定 LLVM 的后端。

4. 基于本系统可以实验 PGO(Profile Guided Optimization) 等编译优化方法：利用程序执行时的信息来进行优化的工具称之为 DBO(Dynamic Binary Optimizer)，这类工具通过采样的方式收集程序的热点路径，然后重新优化热点路径上的代码来达到加速的目的。如惠普公司开发的 Dynamo[16] 通过重组热点路径上的基本块、消除热路径上的直接跳转、内联热路径上的函数调用等方式提升了系统效率。由于本系统可采用离线翻译优化 (AOT) 的方式进行，无需过度考虑优化开销。因此，可以在本系统上实验 PGO 等反馈式编译优化方法。

1.5　论文组织结构

本文以 QEMU 为基础，设计和实现了一个基于 LLVM 编译优化制导的二进制翻译系统 COGBT。该系统通过对客户程序进行离线优化改善了翻译代码的质量，降低了翻译开销，提高了二进制翻译系统的效率。本文各章节的组织架构如下：

第一章是引言，介绍了当前产业界对高效、高可靠、易于维护的二进制翻译系统的需求。随后介绍了静态翻译、动态翻译以及动静结合翻译这几种翻译方式，并对其优劣做了简要描述。接着分析了逐指令翻译的不足，以及如何引入 IR 解决此问题。最后介绍了 LLVM 的翻译框架以及 LLVM IR 的优势。

第二章分别介绍了几款采用不同翻译方式的二进制翻译器，分析了这些翻译器的一些技术细节以及特征。随后介绍了几种常见的二进制翻译优化。最后介绍了几款基于 LLVM 优化的二进制翻译器，分析其架构设计及优缺点。

第三章介绍了 COGBT 的设计和实现细节。首先对该系统的框架做了简要概

述，介绍了系统的各部分组成及作用。之后深入翻译模块，探讨 COGBT 的翻译模型、典型指令的翻译方式。接着论述了 COGBT 中使用到的一些优化策略以及需要解决的一些问题。最后对 COGBT 的代码挖掘、AOT 的生成与解析做了简单介绍。

第四章首先介绍本系统的实验平台及环境。接着从代码覆盖度的角度评价 COGBT 静态代码挖掘的指令覆盖率。之后对 COGBT 的整体性能进行测试，通过运行 SPEC 2000 CINT 与 CoreMark 多种测试集测试 COGBT 相较 QEMU 的优化效果。最后评价 LLVM 优化、寄存器映射、栈提升优化等措施对降低翻译代码动态指令数与访存指令比例的效果。

第五章总结全文，分析当前 COGBT 的不足以及可改进的地方。并探讨后续基于本项目可以实施的一些优化手段。

第 2 章　相关工作

本章首先介绍几种经典的二进制翻译系统。将以过去实际开发并产生一定影响力的翻译器为例，分析静态翻译、动态翻译以及动静结合翻译几种不同翻译方式的特点。然后将介绍二进制翻译中采用的一些优化技术，如基本块链接优化、寄存器分配、冗余标志位消除等技术。最后，本章将介绍一些基于 LLVM 优化的二进制翻译器，分析其架构设计及优缺点。

2.1　典型二进制翻译系统架构

由于二进制翻译技术具有重要的工程价值，近几十年来已得到广泛研究和发展。不同二进制翻译器虽然实现方式多样，但是仍然可以根据翻译时间以及工作方式将其划分为动态翻译器、静态翻译器以及动静结合翻译器。动态翻译器的主要特点是运行时翻译，但是翻译与运行有可能同时进行，如使用多线程方案将翻译线程与执行线程并行执行。静态翻译器的典型特点是离线翻译，可以将一个客户执行文件离线编译成宿主机的执行文件。动静结合翻译器则试图综合前两者的特点，将部分客户代码离线翻译，并生成一部分预翻译文件。几类翻译器都有典型代表，下面将分别介绍。

2.1.1　静态二进制翻译器

Cifuentes 等人于 2000 年开发了一款静态二进制翻译器 UQBT[17]。UQBT 是一个支持可重定源与目标的静态二进制翻译框架。通过引入多种中间表示来实现架构的隔离与代码复用，可以方便地切换源指令架构与目标架构。图 2.1 展示了 UQBT 的基本架构。该系统首先将源客户程序的二进制反汇编成指令流，对指令流实行语义映射将其映射到源客户机相关的 RTL(Register Transfer Lists) 中间表示上，之后将该 RTL 再翻译到 HRTL 上 (High Level RTL)，在 HRTL 上即可实施特定的二进制翻译优化，优化后的 HRTL 将逐步下降到目标机器的 RTL 以及机器指令上。由于优化是离线进行的，翻译器可在多个阶段实施复杂的优化，如在 HRTL 与目标机器的 RTL 上均有对应的优化处理，因而翻译生成的代码更加高效。然而，UQBT 依然需要一个运行时的解释器来处理间接跳转的目标查询、静态翻译未覆盖到的指令以及自修改代码等问题。

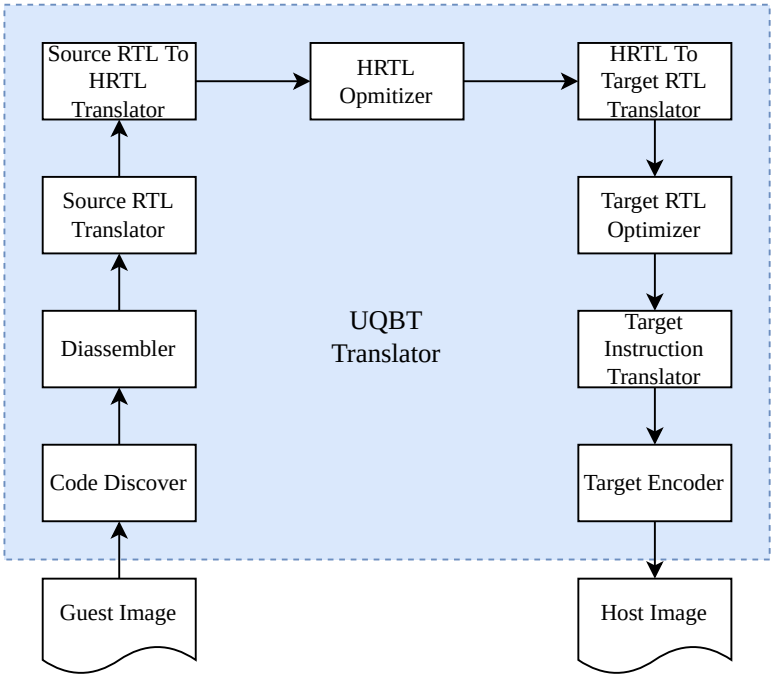


图 2.1 UQBT 基本架构

REV.NG[18] 是一个综合的二进制分析框架，可用于程序分析以及反编译。其核心组件是一个静态二进制翻译器 revamb。revamb 实行两段翻译，先将源二进制翻译为 QEMU 的 TCG，之后再将 TCG 翻译为 LLVM IR，最后使用 LLVM 对 IR 进行优化以及生成目标执行文件。图 2.2 展示了 revamb 的翻译流程。

1. 首先，revamb 对输入二进制进行加载，完成代码段、数据段的映射。

2. 之后翻译器开始实施目标代码挖掘。为保证翻译的正确性，需要识别出所有客户基本块。此阶段翻译器采用了多种策略，包括通过 call/return 指令进行函数入口提取以及 OSRA，SET[19] 算法进行跳转表的恢复。

3. 接下来，翻译器使用 QEMU 的指令翻译模块将客户指令翻译到 QEMU 的中间表示 TCG。借助 QEMU 的跨架构特性以及完备性，可以减轻开发的工作量以及出错概率。

4. 实施 TCG 到 LLVM IR 的翻译与优化。由于 QEMU 对翻译代码缺乏优化，revamb 将 TCG 转化成 LLVM IR 后，可利用 LLVM 丰富的分析优化库对翻译代码实施激进优化。

5. 最后，翻译器利用 LLVM 完成目标代码生成，完成整个翻译流程。

revamb 在将 TCG 翻译为 LLVM IR 过程中遵循如下翻译模型：

1. 使用 LLVM 全局变量映射客户 CPU 的寄存器，客户指令对寄存器的读写

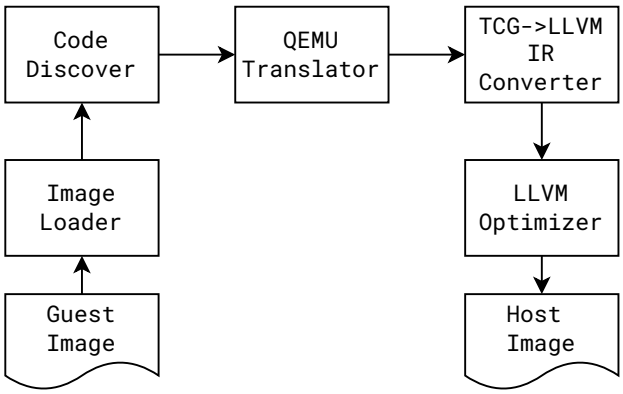


图 2.2 revamb 基本架构

映射为 LLVM 对全局变量的读写。由于全局变量无法像局部变量那样可以实施栈提升优化，此种映射方式需要有较多访存完成翻译。

2. 客户基本块由一个或多个 LLVM 基本块来模拟。大多数情形下单个客户基本块对应单个 LLVM 基本块，当客户指令较复杂时，可能需要生成多个翻译块。如带 rep 前缀 X86 的字符串操作指令，需要多个基本块才能完成其中的循环语义。

3. 客户基本块之间的直接跳转映射为 LLVM 翻译基本块之间的跳转。revamb 提取除了所有的客户基本块，单个客户基本块都对应唯一的翻译块入口，因此只需根据客户基本块之间的跳转即可生成翻译基本块之间的跳转。

4. 客户基本块直接的间接跳转由一段 LLVM 的 dispatcher 代码完成。虽然 revamb 已经挖掘出所有基本块，但是间接跳转在翻译时是未知的，需要注入一段查找代码方能完成。这段查找代码根据客户基本块的地址，得到对应翻译基本块入口，其实现方式如图 2.3 所示。

```
%0 = load i64, i64 *@pc                 // 取出目标地址
switch i64 %0, label %abort [           // 默认跳转的基本块
  i64 0x4000e80, label %block_4000e80   // 匹配block_4000e80基本块
  i64 0x400cd6a, label %block_400cd6a
  i64 0x400cd7c, label %block_400cd7c
]
```

图 2.3 LLVM dispatcher 代码

5. 复杂指令以及系统调用分别由 QEMU 的辅助函数以及系统调用处理模块完成。由于 TCG 是通过外部辅助函数来完成复杂指令的翻译，在实现 TCG 到 LLVM IR 的翻译时同样需要调用辅助函数。revamb 通过将所有 helper 辅助函数

以及系统调用处理模块静态链接到翻译代码来实现这一机制。

revamb 虽然能较快地识别大部分客户指令，在代码挖掘方面已经做到较高水平。但是作为静态翻译器仍无法保证所有代码都能正确执行，因此并不具有完备性。

2.1.2 动态二进制翻译器

QEMU[15] 是由 fabrice bellard 开发的一款可重定源和目标的动态二进制翻译系统。该系统既支持全系统模拟也支持用户态程序模拟。对于用户态模拟，其基本架构如图 2.4 所示：

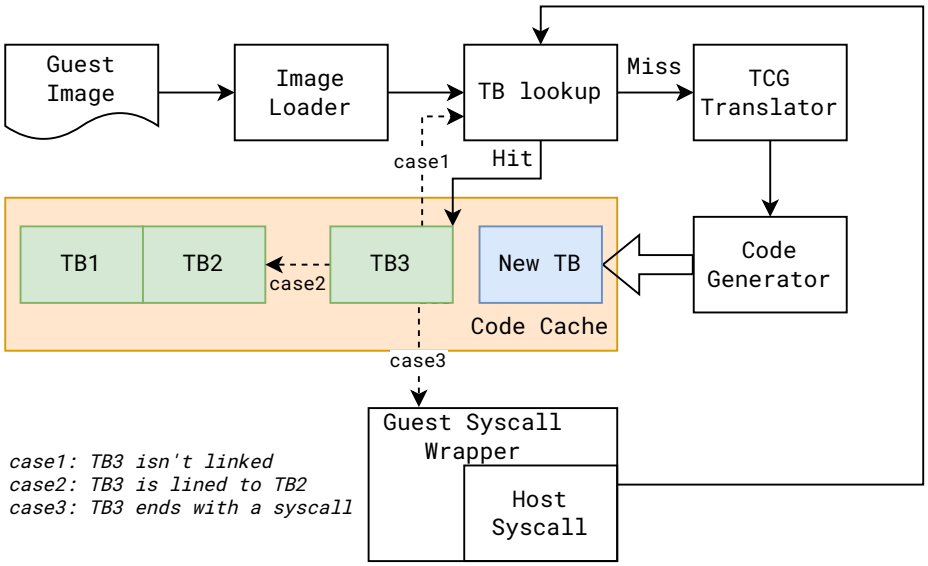


图 2.4 QEMU 基本架构

1. QEMU 首先加载客户程序镜像，将客户执行程序中代码段、数据段映射到 QEMU 进程的地址空间。同时，加载器会模拟内核 execv 系统调用创建进程的步骤，在地址空间中创建客户进程的辅助向量表，在栈底压入环境变量与用户命令行参数。

2. 之后 QEMU 将设置模拟 CPU 的入口地址设置为客户程序的代码入口地址。对于动态链接程序，该地址为 ld.so 的入口，静态链接程序则为客户程序镜像中的入口。设置完成后 QEMU 开始翻译执行。

3. 执行前，QEMU 首先查找目标基本块是否已被翻译。如果已翻译，则可直接跳入代码缓存 (Code Cache) 执行该翻译基本块，否则需要执行翻译逻辑。翻译时 QEMU 首先将客户程序的指令翻译到架构无关的 TCG IR 上，之后会在 TCG

这一层次做少量编译优化，如活性分析、死代码消除、常量传播和折叠等。优化后的 TCG 会再下降到目标机器的机器指令上，如此完成一次翻译过程。

4. 翻译基本块在执行时有三种情形。第一种情形是该基本块的后继基本块尚未翻译或者后继基本块未知 (间接跳转)，此时翻译基本块需要返回翻译器进行目标块的查找或翻译；第二种情形是目标基本块已翻译且与当前块链接在一起，该基本块执行完可以直接跳入目标块；第三种情形是当前基本块以系统调用结尾，需要回到翻译器，由 QEMU 的系统调用子模块完成客户系统调用的模拟。

5. 对于客户系统调用，大部分都可以通过直接调用宿主机系统调用完成。而其余与架构相关的部分则需要 QEMU 进行一些封装。典型的如 mmap，由于客户系统与宿主机系统的页大小可能不一致，需要 QEMU 对宿主机的页进行合并以及调整页权限等。

QEMU 发展多年，本身已经具有了非常好的完备性。由于 TCG 的引入，QEMU 具有可重定目标的特性，用户只需要添加新架构到 TCG 的前端翻译，以及 TCG 到该架构的后端翻译就可以与其他架构进行任意转换。这种良好的扩展性使得 QEMU 发展迅速，目前支持的指令集达二十几种之多。然而为了保证系统的可重定目标，QEMU 本身无法假定宿主机与客户机在寄存器数目上的关系，也因此无法做寄存器分配，如当宿主机具有较多通用寄存器时，可以用部分寄存器直接映射客户机的寄存器状态，而无需从内存中读写状态；此外，为了保证运行时效率，QEMU 无法实施过多运行时优化，这导致 QEMU 仍存在进一步优化的空间；再者，TCG 作为跨架构的中间表示，其表达能力依然较弱，如类型只支持 32 位与 64 位定点与对应的指针，不支持浮点，向量支持较差等。这导致 QEMU 在翻译较复杂的指令时需要调用辅助函数进行模拟，制约了翻译效率，同时也使翻译难度变高。

2.1.3　动静结合二进制翻译器

FX!32[20] 是由 Raymond 等人开发的一款经典动静结合二进制翻译系统。该系统用于将 Windows NT 上的 X86 应用程序翻译到同操作系统的 Alpha 平台上。其基本架构如图 2.5 所示：

FX!32 在完成客户镜像的加载后，首先由动态端的解释器对 X86 程序进行模拟运行。如果客户指令已经存在翻译镜像，则直接运行，否则需要解释执行该指令。动态端在在运行期间会收集程序的运行时信息形成 profile 文件，FX!32 主

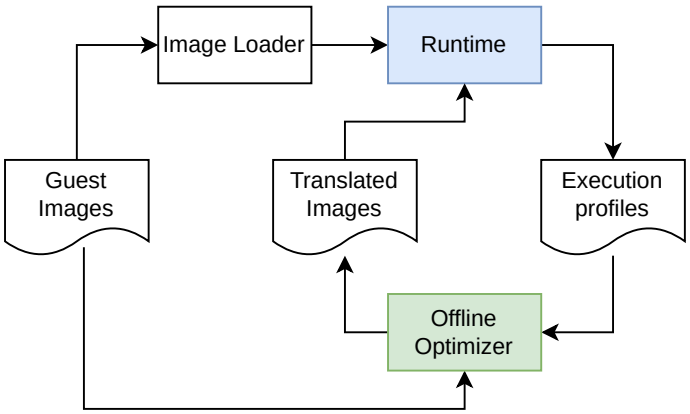


图 2.5 FX!32 基本架构

要收集了 call 指令的目标地址用于函数识别；间接跳转的源地址、目的地址用于优化间接跳转；以及非对齐内存访问的指令地址等。静态端的优化器利用收集到的 profile 信息执行指令的翻译与优化，生成本地的 Alpha 翻译镜像并储存于系统的数据库中。在随后的运行中，二进制翻译器将能直接使用翻译镜像而达到加速执行的目的。该系统在 500MHz 的 Alpha 21164 机器上能达到大约 200MHz 的 Pentium Pro 的性能。

Rosetta 2[9] 是苹果公司开发的一款 X86 到 ARM 的动静结合二进制翻译器。该翻译器在安装程序时会执行 AOT 预先翻译，将整个执行程序的代码段从 X86 翻译到 ARM 上以减轻 JIT 的负担。由于大部分代码都能由 AOT 预翻译到，因此效率较高。根据一些逆向分析 [10, 11]，Rosetta 2 能高效翻译 X86 大致有如下原因：

1. 大部分指令都能执行一对一的翻译。由于 X86 指令与 ARM 指令在寻址方式、子寄存器运算、标志位寄存器的硬件支持等方面具有较大的相似性，使用 ARM 指令翻译 X86 指令相对轻松。

2. 使用硬件返回地址预测。Rosetta 2 将 X86 的call与ret指令翻译成 ARM 对应的bl与ret指令可以利用硬件的返回地址栈做预测，提高硬件的预测精度。

3. 硬件实现 TSO 内存序。由于 X86 架构的内存系统是 TSO 强一致性模型，ARM 架构是一个弱一致性模型。当由 ARM 翻译 X86 指令时，如果不显示加入内存屏障指令，翻译出 ARM 访存序列可能由于处理器的乱序执行、投机调度等导致访存序列发生变化，无法保证与 X86 的访存行为一致。而由翻译器插入屏障指令将显著降低翻译性能。Rosetta 2 通过在硬件上实现 TSO 强一致性内存序，

可以以极小的开销保证翻译出的访存指令在 ARM 上也能符合 X86 上的约束。

4. 使用库直通技术避免程序库的翻译开销。由于苹果对其生态具有较强的的掌控力，许多程序库都能直接获取源代码，并且能定制其内核与工具链。通过对 Mach-O[1] 文件增加 AOT 格式支持、修改调整编译工具链与内核，苹果能将库文件编译成符合翻译器的调用约定。X86 程序在进行库函数调用的时候，Rosetta 2 可以将其翻译成对本地函数库的调用，如此可以避免库函数翻译导致的性能损失。

尽管 Rosetta 2 已经取得较大成功，但由于其代码和原理未公开，对其研究和学习很难进行，因此无法直接复制其成功性。此外，由于 ARM 与 X86 指令集在诸多方面具有相似性，这两种指令之间的翻译在理论上能达到较高的效率上限，但是在用其它架构如 LoongArch 来翻译 X86 时则很难做到一对一，需要有新的研究来继续探索高效的二进制翻译器。

2.2　二进制翻译中的常见优化技术

在二进制翻译的发展过程中，已有大量工作研究了如何提升翻译器的效率，并提出了各种优化方法。HP 公司开发的 Dynamo[16] 动态二进制优化系统利用运行时信息选择热点 trace，将热点 trace 上的基本块放到临近的 trace cache 上加强局部性，并对 trace 中的基本块实施函数内联、跳转指令消除等措施优化热点代码；IBM 公司开发的 Daisy [21] 二进制翻译系统与 Transmeta 开发的 Crusoe [22] 系统通过挖掘指令之间的并行性，利用指令调度的方案将多条无依赖的指令组成 VLIW 指令来达到翻译优化的目的；Zhang[23] 等人提出了 post-optimization 的优化方式来二次优化翻译出的宿主机代码，通过对翻译出的宿主机指令构建数据依赖图，分析图中的冗余性与低效节点模式后再实施二次优化来达到优化的目的；Wang[24] 等人提出了一种基于翻译规则学习的优化思路，将来自相同行源代码编译出的不同架构指令序列进行学习，通过形式化验证技术证明等价性之后，得到不同架构之间的翻译规则来改进翻译；Kim[25] 等人则提出了通过硬件辅助的方案来优化间接跳转。

这些优化方式对翻译系统具有较高的依赖性，较难在其他的系统中实施。如 Dynamo 适用于带有热点收集策略的系统、Crusoe 适用于 VLIW 架构、二次优化

[1] 苹果公司设计的二进制执行文件格式

适合初次翻译有较多冗余的系统。本节将重点介绍一些对翻译 X86 指令集具有普适性的优化。

2.2.1 基本块链接

翻译器一般以基本块作为翻译单元。在翻译器完成一个基本块的翻译并准备跳转到翻译代码中执行时，需要先保存当前翻译器的上下文并从虚拟 CPU 中加载各个客户寄存器的状态，否则从翻译代码跳出时将无法恢复翻译器的现场；同样地，当翻译代码执行完毕准备返回翻译器查找后继基本块时，需要将客户寄存器的状态同步回虚拟 CPU 并恢复翻译器现场。如此就会造成较多上下文切换开销。图 2.6 展示了这一过程。

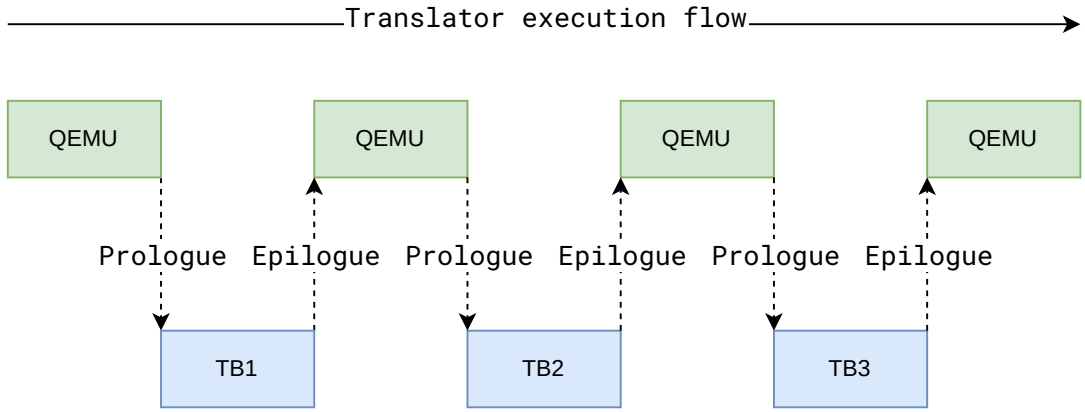


图 2.6 QEMU 上下文切换示意图

以 LoongArch 翻译 X86 为例。根据 LoongArch ABI 规定，函数调用时由被调者保存的寄存器 (Callee Saved Registers，简称 CSR) 有 S0-S8、FP、FS0-FS7 共 18 个。这些寄存器如果被翻译代码修改，则需要由翻译代码进行保存和恢复。此外，翻译代码在执行前需要先加载虚拟 CPU 中 X86 寄存器到指定 LoongArch 寄存器，也需要进行读内存的操作。假定平均一个基本块需要读取 5 个 X86 寄存器，且翻译代码都改写了 CSR，则一次上下文切换 (Prologue 或 Epilogue) 就需要有 23 次访存。以图 2.6 中执行三个基本块为例，仅上下文切换开销就需要访存 138 次，再加上切换回翻译器后，翻译器需要进行后继基本块的查找以及各种检查，所消耗的 CPU 时间同样不小。因此，逐基本块执行的性能损失是巨大的。

基本块链接的思路是一次尽可能多的执行翻译基本块 (TranslationBlock，简称 TB)。只有当目标基本块不确定或者目标基本块未翻译时，执行流才从翻译代码切换回翻译器。这种思路是可行的，因为大部分基本块之间的跳转都是直接跳

转，跳转目标都是固定的，而目标不确定的间接跳转在大部分程序中只是少数。这一点从编译的角度理解更为直观，间接跳转一般由高级语言中的 switch-case 跳转表、函数指针、虚函数表引入，相较普通的条件判断，出现频率较低。图 2.7 是 QEMU 中实现基本块链接的示意图。

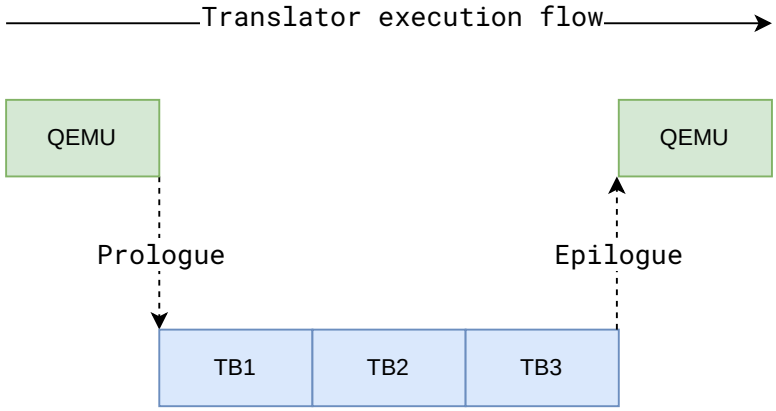


图 2.7 QEMU 基本块链接示意图

在上图中，如果三个基本块都是直接跳转，那么就可以让它们的翻译基本块直接链接起来。这样就将上下文切换的次数从 6 次降低到了 2 次，且减少了 2 次回到 QEMU 进行后继基本块查找的开销。

2.2.2　直接寄存器映射

二进制翻译代码是一段虚拟 CPU 的操作代码。翻译基本块按照客户基本块的指令顺序逐一读写其中的寄存器状态或者客户程序内存值。对于访存指令而言，翻译器使用本地访存进行模拟，翻译指令产生的开销与客户指令基本一致，不太会成为翻译器的性能瓶颈。而对于寄存器操作，翻译器的不同建模方式可能会引起性能的巨大差异。一般而言，翻译器内部都存在一个用内存模拟的客户 CPU，CPU 中寄存器的状态由翻译器内存记录。这意味着如果不加优化，任何 X86 的寄存器操作指令都将翻译为对虚拟 CPU 的访存指令，将与真实物理机的效率产生较大的差距。

以图 2.8 为例，在翻译目标 X86 指令时，如果源操作数是寄存器，翻译代码需要一条 ld 指令从内存中取出源寄存器状态；如果目的操作数是寄存器，翻译代码需要一条 st 指令将结果进行写入。假定普通的算数指令需要 1 个时钟周期，访存指令需要 10 个时钟周期。在实际的 X86 机器上执行该基本块 (不算最后一条跳转) 将需要 15 个时钟周期，而翻译基本块则需要 86 个时钟周期，翻译性能

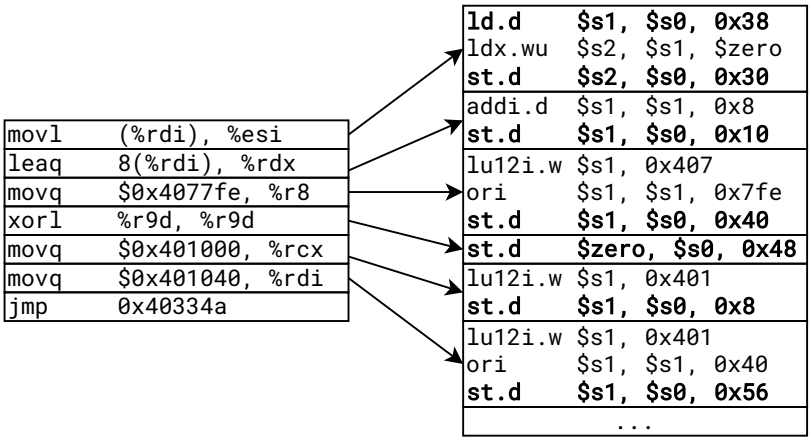


图 2.8 未实施寄存器映射的翻译方式

将只有本地执行的 17%。

直接寄存器映射的思路是想利用宿主机的一部分寄存器直接记录客户寄存器的状态。由于大多数 RISC 指令集都具有较多通用寄存器 (如 MIPS、LoongArch 具有 32 个，ARM 具有 16 个)，基本都不少于 X86 的 16 个寄存器。如果将 X86 的寄存器都映射到宿主机上，那么翻译代码对虚拟 CPU 的寄存器访问基本都能消除掉。

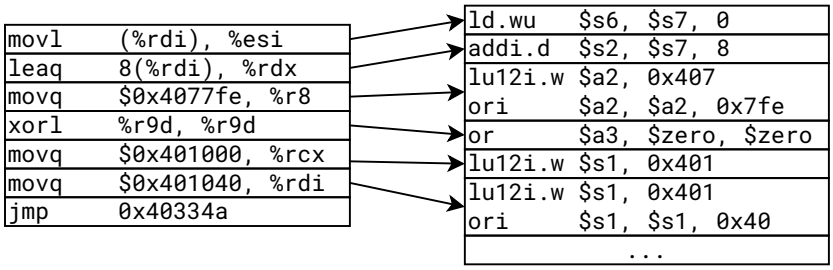


图 2.9 实施寄存器映射的翻译方式

如图 2.9 所示，假定翻译器使用 s6、s7、s1、s2，a2, a3 分别映射 rsi、rdi、rcx、rdx、r8、r9。那么可以完全消除对虚拟 CPU 的访问，按照之前的假设只需要 17 个时钟周期就能翻译同样的 X86 基本块，能达到较高的翻译效率。

2.2.3 冗余标志位消除

在 X86 架构中，存在一个特殊的标志位寄存器 (eflags 寄存器)。该寄存器内部由多个标志位组成，和用户态相关的主要是 CF、PF、AF、ZF、SF 与 OF 6 个。对于大部分算术逻辑指令，在完成运算的同时还会产生若干标志位来表征运算结果的某些属性。随后的条件跳转指令可以根据这些标志位的值来实现条件跳

转。

翻译器在翻译这些影响标志位的指令时，需要用较多指令来模拟硬件的置位操作。如果将所有影响的标志位都翻译出来，X86 指令的翻译膨胀将会非常高。以图 2.10 为例，假定翻译采用了rdx->t7，rdi->s7，eflag->fp 的直接寄存器映射，为了完成 ZF、SF、OF 标志位的计算，翻译器需要生成 16 条 LoongArch 指令。这种高膨胀的翻译方式将会显著降低翻译器的性能，影响用户体验。

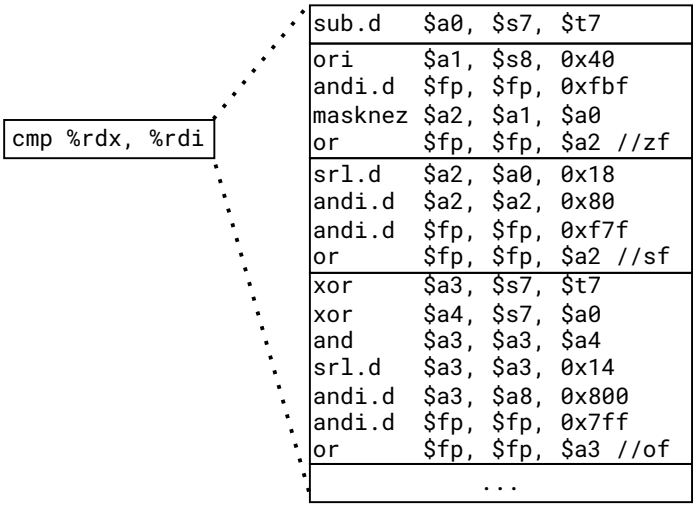


图 2.10 eflag 标志位计算

对标志位进行分析会发现，并不是所有标志位都需要进行翻译。许多标志位的定值并没有其他指令使用，因而对这种标志位进行翻译是冗余的。图 2.11 中第一条 add 指令的所有标志位都是冗余的，因为会被随后的 add 覆盖；同样第二条指令中只有 AF 标志位可能会被用到，其余标志位是冗余的。

```
addq    $1, %rax      // define OSZACP(All are dead)
addl    $1, (%rdx)    // define OSZACP(A is alive)
movzbl  (%rax), %ecx
testb   %cl, %cl      // define OSZCP(All are alive)
je      0x402abf
```

图 2.11 X86 指令中的冗余标志

大多数 X86 翻译器都对标志位的计算进行了优化。QEMU 使用了一种延迟计算的机制。对于标志位的定值指令，QEMU 会记录该指令的源操作数、目的操作数以及操作数类型。当后续的指令需要用到某个标志位时，QEMU 才依据之前保存的信息计算出该标志位的值。另外一些翻译器则通过对标志位进行活性分析，只有分析出标志位会被用到时才对其进行计算。不管何种方式，基本思

路都是避免对冗余标志位进行计算。

2.3 LLVM 优化二进制翻译的相关研究

传统二进制翻译器在架构与优化方式上存在较大差异，这使得将其中的优化策略移植到其他二进制翻译器变得困难。例如，在 UQBT 中，优化策略是针对自定义的 RTL 中间表示进行设计的，将其中的优化在其他翻译器复现需要投入大量工作。与此不同的是，LLVM 提供了层次化的开发框架，该框架基于统一的 IR 设计，可实现可复用的优化策略。通过这种框架，开发者可以节省翻译器的构建时间，且可以直接使用 LLVM 现有的优化库，具有明显优势。

近年针对二进制翻译相关的研究都开始转向 LLVM 进行。这些工作针对如何基于 LLVM 构建翻译系统、如何选择优化对象提出了多种翻译架构。本节主要从使用 LLVM 优化自动生成的函数、优化热点 trace、优化静态识别代码三个方面介绍几个典型的翻译系统。

2.3.1 CrossDBT

CrossDBT[26] 是一个多阶段的动态二进制翻译器。通过引入指令集格式化描述来自动生成翻译函数，而后由 LLVM 对生成的函数进行优化。系统的基本架构如图 2.12 所示。

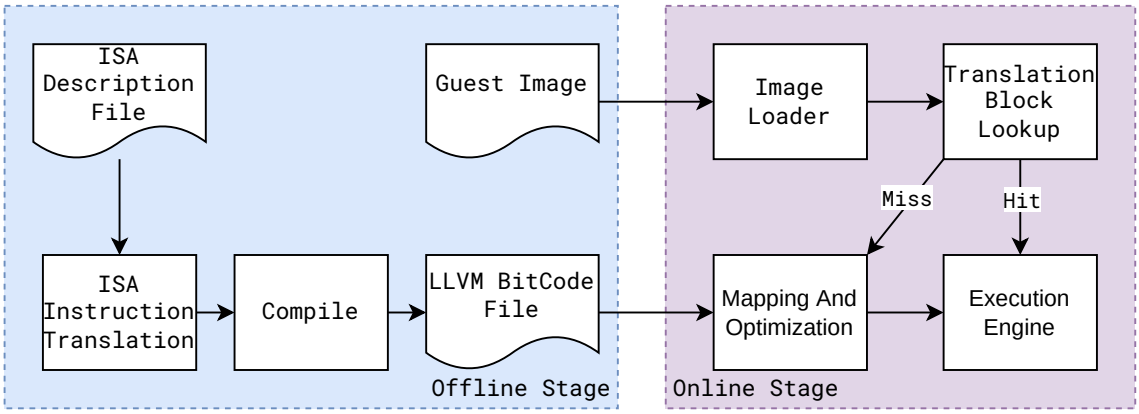


图 2.12 CrossDBT 基本框架

离线阶段，CrossDBT 根据客户指令集的描述，为客户指令自动生成翻译函数，而后由 clang 将翻译函数编译成 LLVM IR 的 bitcode 形式。bitcode 文件的作用相当于翻译函数库，当翻译器执行时只需要根据指令从库中提取翻译函数使用即可。具体步骤如下：

1. 为客户指令集生成 ISA 描述文件，用作翻译函数自动生成的模板。ISA 描述文件主要包括指令类型、指令编码、指令长度、寄存器类型以及可用寄存器数目等信息。通过将指令集信息抽象出来可以减少繁琐的指令翻译工作。

2. 依据描述文件为指令生成具体的翻译函数。由于同类型指令具有较高的相似性，利用描述信息可以批量生成同类型的翻译函数，减轻开发量。指令的翻译函数具有类似的模板，翻译函数接收虚拟 CPU 的地址以及指令的操作数作为参数，函数内部依据客户指令的语义操作虚拟 CPU 中的寄存器或者进行内存读写。从某种角度上说，这种翻译函数与解释器的解释函数或者 QEMU 翻译器的辅助函数类似。

3. 使用 clang 将指令翻译文件编译成 LLVM IR 的 bitcode 形式。这一步将指令翻译文件中的 C++ 函数离线编译成 LLVM IR 文件，形成所有客户指令集的翻译函数库。

在线运行阶段，CrossDBT 读取客户执行文件以及指令翻译的 LLVM IR 文件。随后开始进行动态翻译、执行。具体步骤如下：

1. 加载客户执行文件并将其映射到内存。与其他动态二进制翻译器类似，首先需要加载解析执行程序中的代码段、数据段并映射到内存，随后创建客户进程的执行环境。

2. 查找目标基本块。翻译器根据目标基本块的地址执行翻译基本块的查找，如果在代码缓存中命中，则直接跳入翻译代码执行。否则执行基本块的在线翻译。

3. 对未翻译过的基本块进行指令映射与优化。由于离线阶段已经生成客户指令的翻译函数库。此步骤只需创建一个 LLVM 函数来翻译客户基本块。在 LLVM 函数中调用翻译库中的指令翻译函数即可。图2.13 展示了该翻译方式。由于 LLVM 有大量优化 pass，这些翻译函数最终会被内联、优化成普通的 LLVM IR 指令，最终的代码生成将会产生较优的宿主机指令序列。

CrossDBT 利用 LLVM 的优化将翻译辅助函数内联成了普通的 LLVM IR，减轻了开发负担。开发者只需要按照解释器的开发方式生成指令的翻译函数，并不需要像 QEMU 等翻译器那样实施具体的指令翻译，因而开发难度得到降低。然而，在翻译模型的设计上，CrossDBT 与 QEMU 等翻译器并没有很大差异，翻译器均是通过访存指令来操作客户寄存器的状态，依然有较大的性能损失。此外，

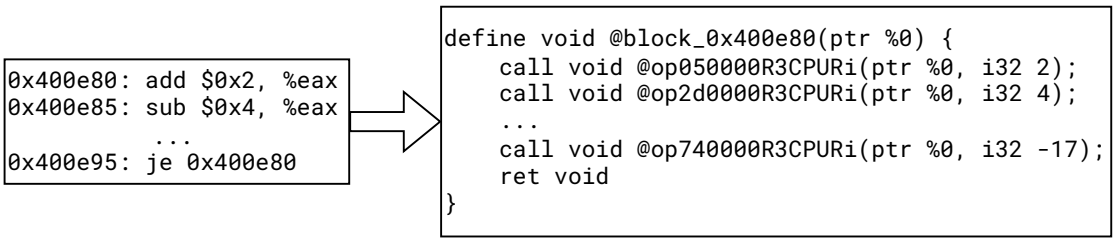


图 2.13 CrossDBT 指令在线翻译的方式

最终的优化依然是在运行时进行的，大量的激进优化会降低翻译器的响应速度，且 LLVM 的代码生成具有较大开销。

2.3.2 HQEMU

HQEMU[27] 是由 Hong 等人基于 QEMU 和 LLVM 开发的多线程动态二进制翻译器。该翻译器由前端的轻量级二进制翻译器 QEMU 与后端的重型翻译器 LLVM 组成。其基本架构如图 2.14 所示。

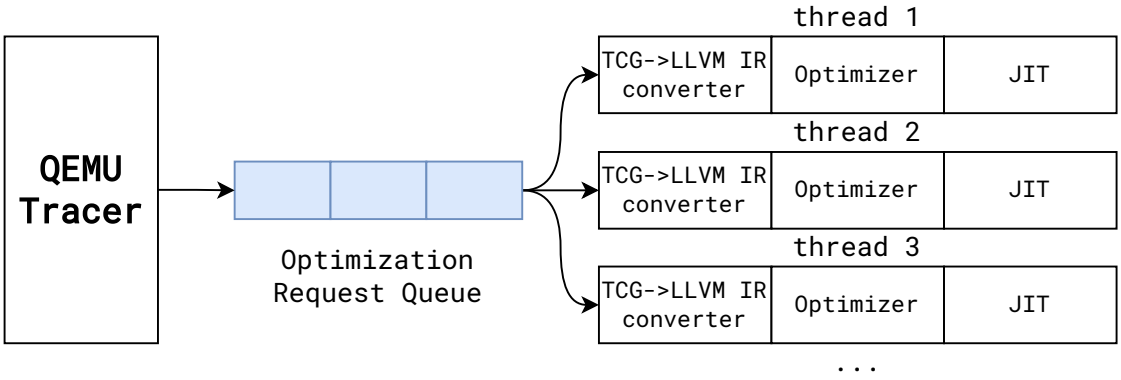


图 2.14 HQEMU 基本架构

HQEMU 前端使用 NET[28] 算法对热点代码进行收集，通过在 QEMU 的翻译块中插入一段计数逻辑来判断代码是否变热，当该基本块 (也称为 trace 头) 执行次数达到阈值时，开始进行热路径收集。该收集算法的执行流程如图 2.15 所示。说明如下：

1. 首先在前端 QEMU 的 TCG 中插入一条 hotpatch 特殊指令，该指令随后被展开成一段插桩代码，执行具体的计数逻辑。为了避免给所有基本块插桩引入过多开销，翻译器仅在间接跳转目标、函数跳转目标以及一条 trace 的退出目标基本块上执行插桩统计。

2. 当 trace 头的执行次数达到阈值，enable_predict 标记被使能，执行代码开

始对基本块进行收集。收集到的基本块被放在一个 trace 数组里面，当数组内出现重复基本块或数组长度过长时，开始封装成 trace 提交优化请求队列。一般而言，这些收集到的 trace 往往是热点循环，因而值得进一步优化。

收集到的 trace 通过一个线程间通讯的 FIFO 队列被通知给后端 LLVM 优化线程。LLVM 优化线程循环从队列中取出 trace，开始实施 TCG 到 LLVM IR 的翻译，之后再运行优化 pass 将 IR 进行优化，最后利用 LLVM JIT 完成本地代码的生成。

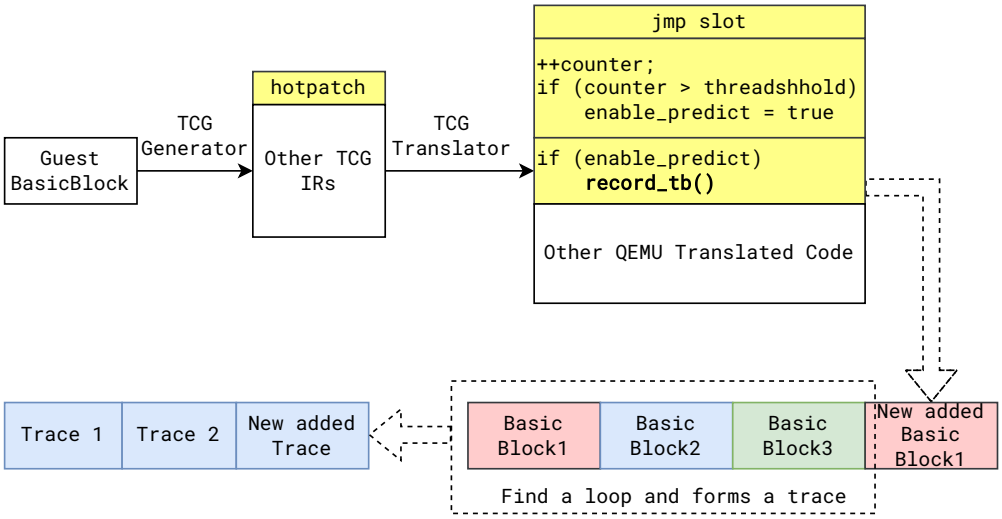


图 2.15 HQEMU 热点 trace 收集算法流程

HQEMU 通过多线程的方式将 LLVM 的优化开销隐藏起来，前端 QEMU 可以并行地继续翻译运行客户程序，一旦热点代码完成优化，翻译器将可直接执行优化后的代码。通过这种冷热隔离的机制，冷代码可以由前端轻量级翻译器快速翻译执行，而热点代码则可以通过后端 LLVM 线程执行耗时的激进优化，提升系统的综合性能。

尽管 HQEMU 在效率上相对 QEMU 有较大的提升，但是随着 LLVM 后端架构的愈发复杂，LLVM 的优化与编译开销得到了较大增长，这导致了一个问题：前端 QEMU 已经翻译执行完了热点 trace，后端优化器才完成优化编译，导致优化效果大打折扣。

2.3.3　HBT

HBT[29] 是一个基于 LLVM 开发的动静结合二进制翻译系统。该系统尝试让静态端尽可能多的识别与翻译客户指令，当遇到没有翻译的代码时才调用动

态端就行补充。在系统设计上，HBT 动态端与静态端都基于 LLVM IR 进行翻译：

静态端首先通过 LLVM 的 MC 反汇编器完成客户指令的识别与译码，随后静态翻译器将客户指令翻译为 LLVM IR 的中间表示，并将结果保存到 LLVM 的 bitcode 文件。之后翻译器通过使用 LLVM 的优化器对 bitcode 文件实施架构无关的优化，最后使用 LLVM 的后端将 IR 下降为宿主机的机器指令。

动态端与其他二进制翻译器类似，首先在翻译器中为客户执行程序分配一段地址空间供其映射，同时初始化客户进程的栈，压入命令行参数、环境变量等内容。随后将翻译器的执行入口设置为客户程序的入口地址。当目标基本块已被翻译，翻译器直接进入翻译代码执行，只有遇到未翻译的代码时，才会回到动态端将基本块翻译为 LLVM IR，再将其优化、下降到宿主机指令。

HBT 提出了多种策略来加速翻译效率，其中最显著的是优化间接跳转的查找表。在静态翻译器中，HBT 通过采用与 REV.NG 类似的思路使用 switch-case 语句生成目标地址与翻译地址的查找表。然而，静态翻译中识别到的基本块过多，直接翻译将生成较多的表项，影响性能。HBT 通过使用一个哈希函数将一个大型的 switch 语句分解为多个小型 switch 语句来提高查找效率。此外，HBT 也实现了基本块链接的优化。

HBT 的动静结合方案可以充分利用静态端离线优化的特性实施激进的优化手段。并且通过代码挖掘可以让翻译代码尽可能地在静态翻译代码中执行，能达到较好的效率。然而，HBT 使用 LLVM 作为动态端会有较高的运行时开销，目前的 LLVM 的后端复杂度与时间开销显著高于之前的版本，因此不太适合用于动态端。此外，HBT 依然使用全局变量来表示客户寄存器，因此会有大量访存操作。表2.1总结了部分现有工作的特点与不足。

表 2.1 现有二进制翻译工作的特点及不足

相关工作	翻译方式	特点	不足
QEMU	动态翻译	基于 TCG IR, 易于扩展, 完备	无法实施复杂优化
HQEMU	动态翻译	收集 Trace 二次优化, 易于扩展, 完备	仍有优化开销, 全局变量建模客户寄存器
REV.NG	静态翻译	基于 IR 易于扩展, 可离线实施复杂优化	无法处理代码挖掘与自修改问题
Rosetta 2	动静结合	逐指令翻译, 离线优化, 软硬协同	无 IR, 不易扩展
HBT	动静结合	基于 LLVM IR, 离线优化, 多级地址查找表	全局变量建模客户寄存器

2.4　本章小结

本章首先介绍了静态翻译、动态翻译以及动静结合翻译几种翻译方式的特点，并以几款典型二进制翻译器为例进行了分析与说明。其中动态翻译器可获取程序的运行时信息、处理自修改与代码挖掘问题，因此具有较高的完备性与可靠性。然而由于动态翻译器无法实施复杂优化，其效率容易受到限制；静态翻译器可以离线实施优化，能达到较高的效率上限，但是静态翻译器无法完全识别出客户指令，并且无法处理自修改问题，不具有通用性与完备性；而动静结合翻译器则可兼具两者的优点，使用静态端进行离线优化，使用动态端进行翻译补充，因此能保证既完备又高效。

随后，本章介绍了几种二进制翻译常见的优化方法，分析了基本块链接、寄存器映射与标志位消除的优化动机与原理。其中基本块链接能显著降低上下文切换的次数，减少客户机状态的加载与同步开销；寄存器映射能将访存操作优化为寄存器读写操作，提高翻译代码的质量；标志位消除则可将冗余的标志位进行消除，降低无用指令的数量。已有的工作大多是针对客户机器指令或者自定义 IR 来实施这些优化，较少在 LLVM IR 上执行，因此本工作对后续基于 LLVM IR 的二进制翻译器具有重要参考价值。

最后，本章介绍了几种基于 LLVM 的翻译系统，指出了它们的不足。这些翻译器大多采用全局变量对客户机的寄存器建模，导致访存相对较高，而本文实现的 COGBT 以 LLVM 栈变量进行建模，能有效降低访存比例，达到较高效率。

第 3 章　COGBT 的设计与实现

本章将详细探讨一个基于 LLVM 编译优化制导的二进制翻译系统 COGBT。首先从顶层视角介绍 COGBT 的基本框架，包括动态端与静态端的作用及交互方式；其次，深入 COGBT 的翻译模块，介绍指令的翻译模型，并以若干典型指令为例讨论其翻译方式；接着，介绍 COGBT 在 LLVM IR 上实施的一些优化手段，包括使用 LLVM 的一些通用优化、自定义优化、寄存器映射优化与基本块链接优化；最后，本章简要地介绍一下 X86 的代码挖掘方式与 AOT 的生成、解析。

3.1　COGBT 框架概述

3.1.1　总体架构

重新实现一个二进制翻译系统需要非常大的工程投入，特别是针对动静结合二进制系统而言，不仅需要一个功能完备的动态端还需要实现一个具有丰富优化功能的静态端。为了减少开发工作量，提升实验的可行性，COGBT 以 QEMU 作为动态端，以自行设计的 LLVM 优化框架作为静态端来探究动静结合二进制翻译系统的实现细节与优化效果。

选择 QEMU 作为动态端有几点优势。其一是 QEMU 的翻译优化简单，相较其他重型翻译器更加轻量。以 CoreMark 为例，其翻译与优化开销仅占运行时的 0.1%，不会带来过多的运行时负担。其二是 QEMU 足够完备，经过 QEMU 社区多年的发展，目前已经支持的指令架构达二十几种，几乎涵盖了目前所有的指令集，具有较强的完备性。基于 QEMU 开发动态端可以避免重复开发，能将有限精力集中在静态端的开发与优化上。

选择 LLVM 作为静态端同样有若干优势。其一是 LLVM 具有较完善的分析优化基础设施。LLVM 良好的模块化设计与丰富的 API 接口使其非常适合二次开发，目前已有大量基于 LLVM 开发的二进制翻译器如 REV.NG、HQEMU 等，且 LLVM 集成了大量通用优化 pass 可供直接使用。其二是基于 LLVM 可重定目标，LLVM IR 的强大表达能力与易用性非常适合作为翻译的中间语言，切换后端架构即可生成不同宿主机的机器码，易于扩展。

图 3.1 展示 COGBT 的基本架构。COGBT 由一个完备的轻量级动态端和一

个重型优化的静态端两部分组成。动态端读取执行文件与 AOT 预翻译文件模拟程序的执行，静态端翻译可执行文件并完成离线优化生成 AOT。两者相辅相成保证了系统的完备与高效。

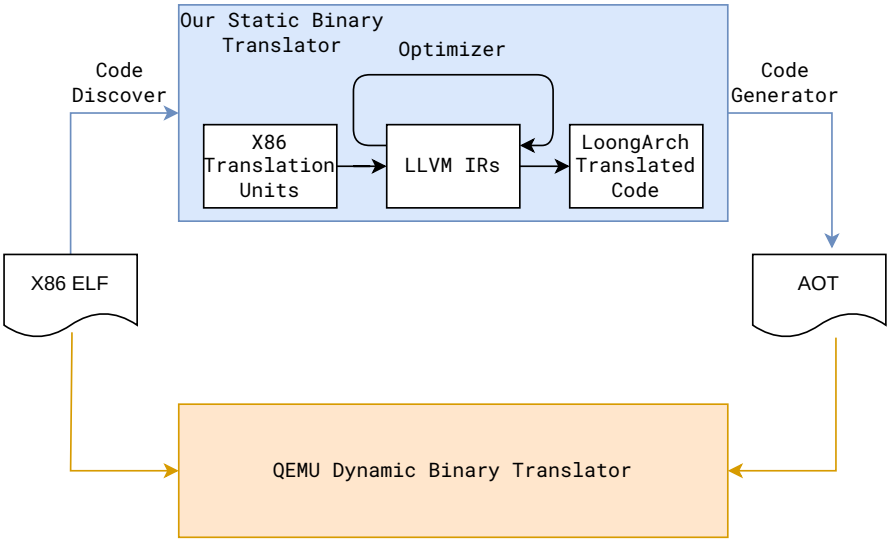


图 3.1 COGBT 动静结合二进制翻译系统架构

3.1.2 动态端概述

动态端是系统运行时的主体，承载着创建翻译系统运行时环境的任务。其基本架构与 QEMU 基本一致 (参见图 2.4)，主要作用如下：

1. 引导与加载客户可执行文件。与其他二进制翻译器类似，动态端需要将客户程序映射到翻译器的地址空间以便获取客户程序的代码与数据进行翻译。

2. 加载解析 AOT 预翻译文件。这一步是实现静态翻译加速翻译效率的关键。静态端通过离线翻译优化客户程序，将产生一个经过优化后的 AOT 预翻译文件。从翻译质量上看，AOT 文件中的翻译代码显著高于动态端翻译的代码。动态端按照 AOT 文件格式对翻译代码进行读取，将翻译基本块的代码与该基本块的元数据信息注册到翻译系统的全局哈希表中，供后续动态查找。与纯动态翻译器相比，AOT 文件可帮助翻译器减少运行时翻译的开销。

3. 设置客户程序的运行时环境。客户程序所依赖的环境变量、辅助向量表、命令行参数以及翻译系统运行期间所依赖的组件统称为运行时环境。动态端需要设置客户程序所依赖的各种参数，同时也需要管理翻译系统的翻译查找表、代码缓存等运行时组件。提供一个完备的运行时环境方能实现动态端的透明运行。

4. 作为运行时补充翻译静态端未识别的代码。静态端的代码识别虽然能达

到较高的覆盖率。但是由于代码混淆、代码压缩、以及代码自修改等问题的存在，静态端很难离线识别所有的客户代码。当翻译器运行时遇到未翻译的基本块时需要回到动态端进行翻译补充。

5. 提供系统调用与信号处理子模块。客户进程所需要的系统调用由动态端进行封装模拟。大部分系统调用可以直接由宿主机提供，涉及到架构相关的系统调用如 mmap、readlink 则需要动态端进行封装。此外，动态端将自身的一部分信号处理函数预留给客户进程使用。通过翻译客户进程的信号处理函数来模拟客户进程的处理。

3.1.3　静态端概述

静态端是系统实现优化加速的部分。翻译器借助 LLVM 编译框架将客户指令提升到 LLVM IR，然后对 IR 运行优化 pass 进行编译优化，最后将优化后的 IR 编译到本地，生成 AOT 预翻译文件。基本框架如图 3.2 所示。

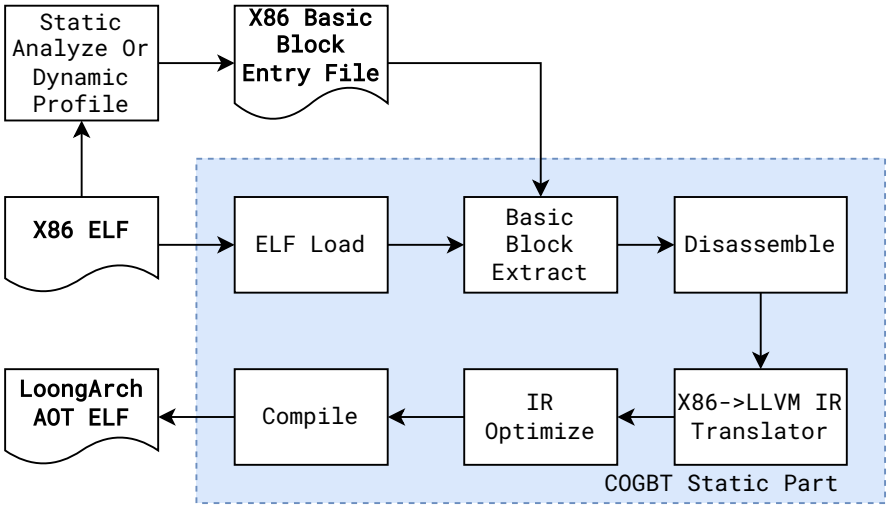


图 3.2 COGBT 静态端基本框架

静态端工作方式如下：

1. 通过静态分析或动态采样完成客户执行文件的代码挖掘。静态翻译中，离线优化所能覆盖的代码量直接影响着系统的性能。静态优化的代码越多，需要动态翻译的代码就越少。为了尽可能多地识别客户指令，COGBT 可结合多种代码挖掘策略获取基本块入口。如借助一些静态分析工具静态提取基本块入口，或者运行动态翻译器对客户基本块进行采样。代码挖掘完成后会生成一个包含基本块入口的 path 文件。

2. 加载客户可执行文件并提取客户基本块。得到基本块入口之后可以按照基本块的定义来提取基本块。COGBT 遍历所有入口，对每个入口进行顺序扫描，遇到分支指令时就构造一个客户基本块。

3. 反汇编客户基本块。COGBT 使用 capstone 将二进制指令反汇编成良好的结构化表示。翻译器可以直接获取指令的操作数以及类型信息，进而实施翻译。选用 capstone 的好处是其支持多种架构，且指令信息结构设计良好。

4. 翻译基本块中的指令到 LLVM IR 上。将客户指令提升到 LLVM IR 是 COGBT 工作量最大也是最容易出错的部分，COGBT 按照特定的翻译模型将虚拟 CPU 的状态映射到 LLVM 的操作对象中，然后根据指令语义选取 IR 来实施翻译。

5. 离线优化 IR。翻译后的 LLVM IR 尚有较多冗余，需要运行 LLVM 的一些优化 pass 优化来提高 IR 质量。另外，COGBT 使用了 LLVM 的栈变量缓存客户 CPU 的寄存器状态，存在大量对栈的访存指令，需要运行优化 pass 将栈提升到寄存器，降低访存操作等。

6. 将 IR 编译到本地。优化后的 IR 可以直接使用 LLVM 提供的代码生成库下降到本地指令。LLVM 提供两种代码生成的方式。一种是使用 MCJIT 将一个翻译模块即时地编译到本地，另一种是直接编译生成可重定位目标文件。COGBT 实现了两种方式，使用 MCJIT 进行动态翻译调试，使用生成可重定位目标文件 (也即 AOT 文件) 的方式进行翻译加速。

3.1.4 调试框架概述

随着翻译指令逐渐增多以及程序运行规模逐步扩大，翻译器的调试难度将大幅增加。首先是程序在动态运行时所执行的基本块数将非常大，要确定出错基本块的位置将无比困难；其次是由于 LLVM 存在包括指令调度、控制流图简化等在内的编译优化，翻译出的代码与原指令差异较大，较难还原出翻译代码的逻辑；再者，翻译代码在实际运行出错时也不一定是第一出错现场，要追踪最早出现问题的指令也比较麻烦。针对调试难的问题，COGBT 设计一套调试机制用于减轻调试负担。基本思路是与 QEMU 作差异性分析。由于 QEMU 本身已经经过了大量测试与实际使用，稳定性已经达到了很高的程度。我们可以与 QEMU 动态地逐基本块对比，并分析基本块执行结果不一致的原因。此外，也可以通过动态地用 COGBT 翻译基本块替换 QEMU 块来定位问题。这两种思路如图 3.3 所

示。

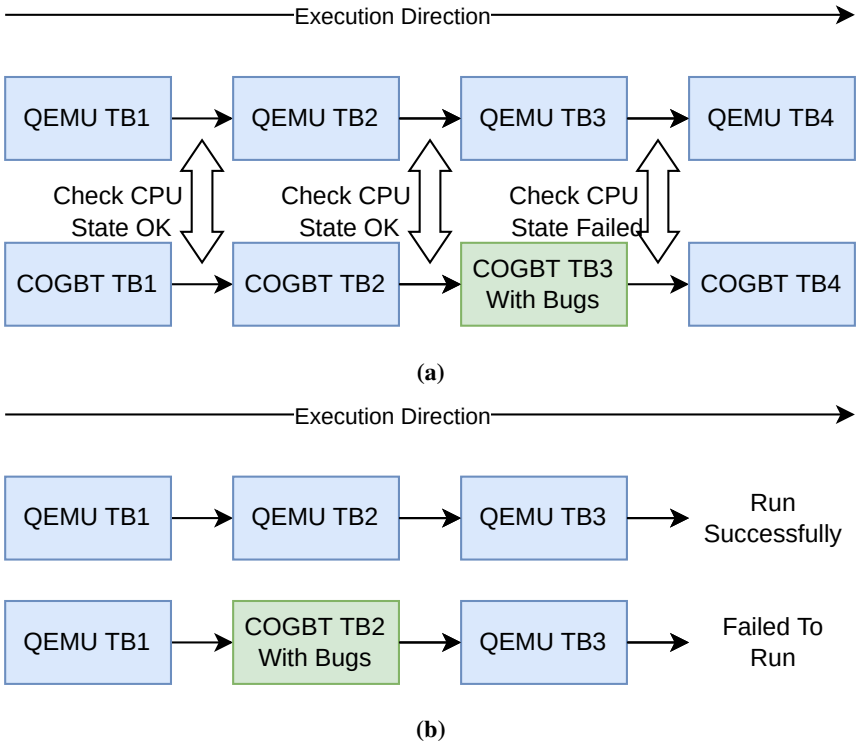


图 3.3 调试框架原理。(a) 逐基本块对比调试法，(b) 基本块替换调试法

逐基本块对比调试法在每个翻译基本块执行结束后对比 COGBT 与 QEMU 虚拟 CPU 的状态，如果寄存器状态完全一致，则认为该基本块正确，否则可以认为该 COGBT 基本块存在问题，需要进一步检查翻译的正确性。实际调试过程中发现即使是访存指令出现问题，一般这种错误也可以很快体现在寄存器的差异上，因此该调试方案对于快速定位出错现场是实际可行的；基本块替换调试法是将一部分 QEMU 翻译基本块替换为 COGBT 翻译基本块，如果替换前后程序运行结果一致，则可认为这部分 COGBT 翻译基本块是正确的，反之则认为这部分基本块存在问题，可以通过二分的方式进一步划分这些基本块，通过更加细粒度的测试定位翻译出错的现场。

实现逐基本块对比调试法最关键的一点是消除 QEMU 的随机因素以及两者之间不一致的地方，COGBT 主要做了如下几个方面的工作：

1. 消除环境变量、执行目录、命令行参数等差异。因为内核在使用 execv 系统调用创建进程时，会先在栈底压入环境变量与命令行参数，这些差异将影响模拟的 X86 栈指针。

2. 确保 QEMU 与 COGBT 基本块划分一致。当基本块划分不同时，逐基本

块打印的值将产生差异，不利于对比。典型的差异是 QEMU 将带有 rep 前缀的指令当作基本块的块尾指令。COGBT 通过屏蔽这类基本块的输出来保证打印日志的一致。

3. 取消基本块链接确保逐基本块打印。进行逐基本块打印时需要解除链接优化，包括直接跳转优化与间接跳转优化，否则会出现差异。

4. 消除随机因素。翻译器在进行可执行文件加载、指令翻译时可能存在一些随机因素。如 ELF 解析时会设置 Auxiliary Vector Table 的一些 random 变量，rdtsc 指令翻译时会读取时间值等。可以通过固定随机值来保证输出的一致。

两种调试方式各有优势，互为补充，通过将两种调试方式配合使用可以帮助 COGBT 快速定位翻译问题。

3.2 COGBT 指令翻译

指令翻译模块是 COGBT 的基础模块，是实施后续分析优化的前提和基础。COGBT 实现了从 X86 指令到 LLVM IR 的翻译，通过建立合理的翻译模型使 COGBT 具有良好的翻译效果。本节将首先介绍 COGBT 的翻译模型—栈翻译模型，论述如何将 X86 指令的操作映射到 LLVM IR 上，以及这种模型的优势。其次，将介绍 COGBT 中上下文切换的逻辑以及与 QEMU 翻译代码混合执行所需解决的问题。接着，将介绍几种典型 X86 指令的翻译方式，如普通算术逻辑指令、串操作指令、分支指令以及系统调用指令等。最后，将介绍 X86 eflags 标志位的翻译方式，对比纯软件翻译与 LBT 硬件辅助翻译的效果并讨论在 COGBT 中加入 LBT 的方式。

3.2.1 翻译模型

CPU 可以认为是一个状态机，其寄存器与内存值构成了 CPU 的当前状态。当 CPU 执行一条指令时，将根据指令的语义对寄存器或者内存进行操作，实现状态的变迁。二进制翻译本质上是模拟客户机 CPU 的状态行为，或者说实现客户机 CPU 的状态机。当模拟的状态机与物理 CPU 完全一致时，用户程序将感知不到底层硬件的差异，从而实现软件的透明运行。典型的翻译器的翻译过程如下：

1. 读取客户指令，分析客户指令的语义。

2. 根据指令的语义修改虚拟 CPU 的状态或者修改内存的状态。

3. 回到第 1 步，继续读取下一条客户指令进行翻译。

在该翻译过程中，核心的问题是如何模拟虚拟 CPU 的状态以及如何操作虚拟 CPU 的状态变迁。对虚拟 CPU 状态的建模方式以及操作方式统称为翻译器的翻译模型。不同翻译器采用的翻译模型略有差异、互有优劣。表 3.1 总结了 QEMU、HQEMU、REV.NG 与 Rosetta 2 四种翻译器的翻译模型：

表 3.1 典型翻译器的翻译模型

翻译器	CPU 寄存器建模	基本块翻译	直接跳转翻译	间接跳转翻译	复杂指令翻译
QEMU	全局变量	若干翻译基本块	直接跳转或上下文切换	查找地址映射表	调用 helper
HQEMU	全局变量	若干翻译基本块	直接跳转或上下文切换	IBTC 与地址映射表查询	调用 helper
REV.NG	全局变量	若干翻译基本块	直接跳转	生成 Dispatcher 查找代码	调用 helper
Rosetta2	宿主机寄存器	若干翻译基本块	直接跳转	查找地址映射表	指令翻译

四种翻译器中除了 Rosetta 2 使用宿主机寄存器映射虚拟 CPU 寄存器外，其余都使用翻译器的全局变量来记录 CPU 寄存器状态。如果选用宿主机寄存器来表示客户 CPU 的寄存器状态，那么客户指令对寄存器读写的操作都可被翻译为对宿主机寄存器的读写操作。如果使用翻译器全局变量进行客户 CPU 寄存器建模，那么客户指令的寄存器读写操作则会被翻译为读全局变量的访存操作。

QEMU 使用全局变量建模客户寄存器。图 3.4 展示了 QEMU 翻译一条 X86 指令 *add $0x8, %rax* 的过程。该指令对寄存器 rax 执行自增 8 的操作。在该指令的翻译过程中，QEMU 首先使用一条 load 指令将 rax 的值从虚拟 CPU 中读取出来，然后执行一条 addi.d 的运算指令计算自增的结果，最后再使用 store 指令将结果写回虚拟 CPU。在这种翻译模型下，所有对 X86 寄存器的读写都被映射为访存操作，因此翻译效率容易受到限制。

HQEMU 依然使用全局变量建模虚拟 CPU 的寄存器，但是做了一些改进来减少过多的访存。HQEMU 的基本思路是如果一个基本块多次使用到了同一个寄存器，那么翻译器没必要每次都将结果写回到全局变量中，只需要一次访存读取该寄存器值，并将结果缓存起来即可。后续对该寄存器的读写都可以直接操作缓存值。只有当翻译器遇到同步点 (如访存、跳转) 才将结果写回到虚拟 CPU 的全局变量中。图 3.5 是该过程的示意图。

尽管使用全局变量缓存的方式能降低一部分客户寄存器状态的加载与同步，

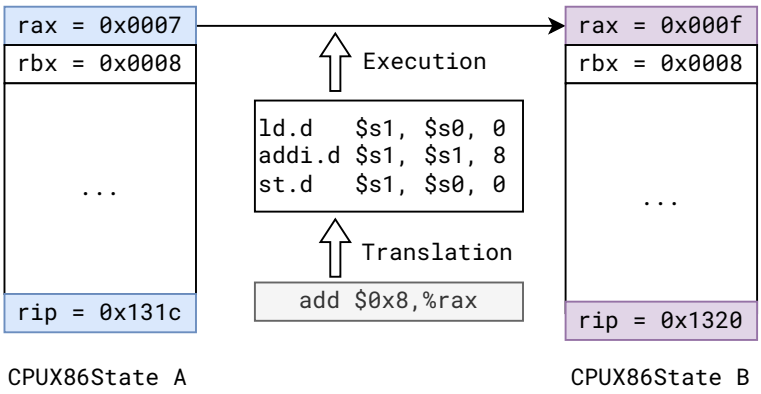


图 3.4 QEMU 指令翻译模型

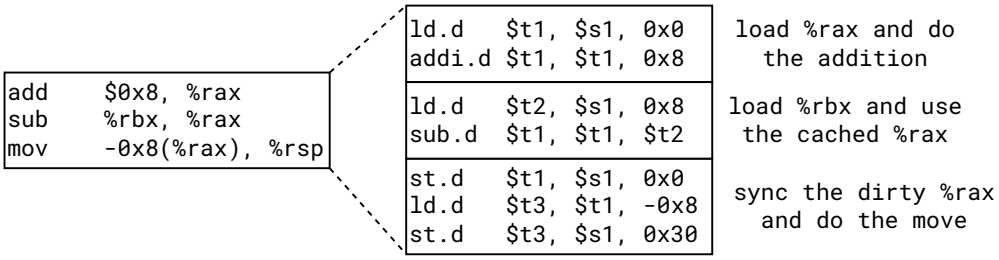


图 3.5 HQEMU 寄存器状态缓存示意图

但是依然需要在每个翻译单元 (基本块、trace 等) 的起始与结尾处加载、写回寄存器状态值。特别地，当该翻译单元是核心循环，这种加载与写回的开销也是不可忽视的。针对使用全局变量建模容易产生较多访存这一问题。COGBT 设计了 LLVM 的栈翻译模型来对客户 CPU 的寄存器进行建模。在 LLVM IR 中，能直接操作的对象只有全局变量、局部变量与虚拟寄存器。全局变量的访问具有副作用，编译器一般不会对其进行优化，因此会产生较多访存，不适合做映射。LLVM 虚拟寄存器是 SSA 形式的，每次对客户寄存器的更新都需要写入一个新的虚拟寄存器，翻译器很难追踪最新的客户寄存器状态, 因此也不适合映射客户寄存器。而 LLVM IR 中的局部变量却可以多次读写，并且这些栈变量随后会被编译器提升到虚拟寄存器层次，由寄存器分配器进行统一的物理寄存器分配，因此不会产生过多访存，适合映射 CPU 的寄存器。

图 3.6 展示了 COGBT 的栈式翻译模型。在该模型中，一个客户基本块被映射为 LLVM 的一个翻译函数。翻译函数由入口基本块 (Entry Block)、翻译基本块 (Translation Blocks) 以及退出基本块 (Exit Block) 组成。入口基本块分配栈变量，读取映射物理寄存器并将值存到相应的栈变量中 (初始化栈变量)。翻译基本块根据具体的 X86 指令将其翻译为对应的 LLVM IR；退出基本块读取栈变量的

值，将结果同步回固定映射寄存器。

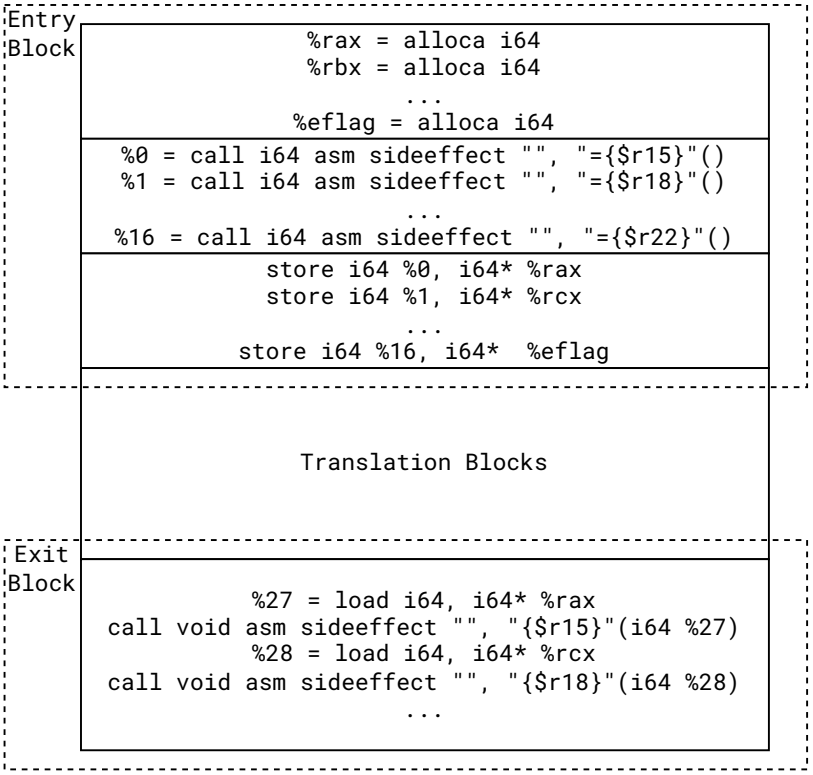


图 3.6 COGBT 栈式翻译模型

COGBT 的翻译基本块遵循如下翻译模型：

1. 以基本块做为翻译单元。同大部分二进制翻译器与一样，COGBT 每次将一个客户基本块翻译为若干个宿主机的基本块，而这些翻译基本块都位于一个 LLVM 翻译函数中，由优化模块对翻译函数进行优化。

2. 直接跳转被翻译为上下文切换或宿主机上的直接跳转。未实施基本块链接优化之前，直接跳转将被翻译为上下文切换，由翻译器实施目标块的查找；基本块链接后，客户基本块之间的直接跳转被翻译为翻译基本块之间的直接跳转。

3. 间接跳转由基本块查找函数完成。翻译间接跳转指令时，COGBT 调用动态端的基本块查找函数进行查询。如果目标基本块已翻译，则跳转到目标翻译块，否则进行上下文切换。

4. 复杂指令通过 helper 函数完成。一些复杂指令或者架构相关的指令由动态端的辅助函数完成，系统调用则通过动态端翻译器模拟完成。

综上，COGBT 的翻译模型如表 3.2 所示：

表 3.2 COGBT 的翻译模型

翻译器	CPU 寄存器建模	基本块翻译	直接跳转翻译	间接跳转翻译	复杂指令翻译
COGBT	LLVM 栈变量	LLVM 翻译函数	直接跳转或上下文切换	查找地址映射表	调用 helper

3.2.2 上下文切换

上下文切换是翻译器与翻译代码进行执行流切换时的入口代码。切换包含保存与加载两个动作。当翻译器完成基本块的翻译准备执行时，需要先保存当前翻译器的现场然后加载虚拟 CPU 的寄存器状态；对应地，在翻译器离开翻译代码准备返回时，需要保存最新的虚拟 CPU 寄存器值并恢复翻译器上下文。如下是 COGBT 在准备进入 LLVM 翻译代码时需要执行的 prologue 步骤：

1. 调整栈指针寄存器 $sp，分配一段栈空间供保存翻译器上下文。
2. 保存翻译器现场，包括 $s0-$s8, $fp(Callee Saved Registers, CSR) 以及 $ra 寄存器。
3. 从虚拟 CPU 中加载 X86 寄存器的状态并放入固定映射的宿主机寄存器。
4. 跳转到翻译代码入口执行。

同样地，COGBT 在退出 LLVM 翻译代码时需要执行如下 epilogue 切换代码：

1. 将 X86 寄存器状态从固定映射寄存器同步回虚拟 CPU。
2. 从栈中恢复 CSR 寄存器值并获取返回地址值。
3. 调整栈指针寄存器。
4. 返回翻译器执行流。

相比于 COGBT 进入 LLVM 翻译代码的上下文切换逻辑，COGBT 进入 QEMU 翻译基本块所执行的上下文切换略有不同。主要的差异在于 QEMU 并未执行固定寄存器映射，因此切换时无需将固定映射寄存器从虚拟 CPU 中进行加载或同步。切换时的 prologue 代码将不需要步骤 3，epilogue 代码将不需要步骤 1。而 COGBT 需要用到 QEMU 翻译基本块主要有如下几个原因：

1. 翻译器运行时遇到未离线翻译的基本块。此时需要 QEMU 进行翻译补充。
2. X86 程序发生了代码自修改。此时也需要 QEMU 重新翻译这些基本块。

由于 COGBT 可能同时存在 QEMU 与 LLVM 的翻译代码，翻译器需要有两套上下文切换的逻辑。图 3.7 展示了 COGBT 的上下文切换方式。当 COGBT 的目标查询模块找到位于代码缓存的翻译基本块时，首先会判断该基本块的类型

是 QEMU 翻译代码还是 LLVM 翻译代码。这可以通过目标基本块所在地址进行判断，也可以通过读取基本块的类型标志进行判断。根据基本块类型的不同，翻译器将跳转到不同的上下文切换入口。当翻译基本块执行完毕后，翻译块的退出代码会自动跳转到所属的上下文切换接口完成上下文切换。

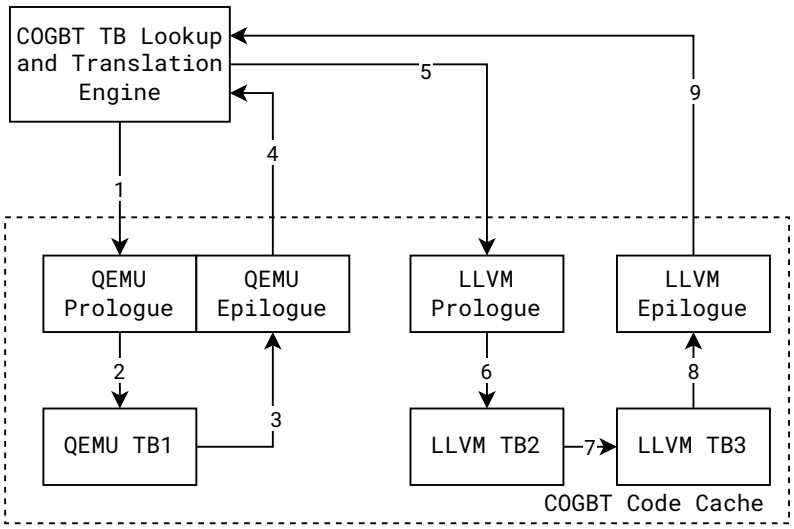


图 3.7 COGBT 上下文切换示意图

要使 QEMU 与 LLVM 的混合翻译系统正常工作，还需要解决两种翻译代码在互相跳转时的差异。这些差异主要由 QEMU TCG 与 LLVM IR 对 eflag 的计算不一致造成。QEMU 使用了标志位的延迟计算方式，所有标志位并不会立即计算，而是在定值标志位时使用虚拟 CPU 中的 *cc_dest*、*cc_src*、*cc_op* 来保存该指令的目的、源操作数与操作码。后续指令需要用到标志位时才使用保存的信息来计算所需标志位。在这种计算方式下，当一个 QEMU TB 执行完后，其虚拟 CPU 中的 eflag 并不是最新的结果，需要由 *cc_op* 等信息进行恢复；而 LLVM 的翻译由于使用了 LoongArch 的 LBT 扩展指令，其对标志位的模拟是精确的。当一个 LLVM 翻译块执行完并上下文切换后，虚拟 CPU 中的 eflag 寄存器就是最新的结果。这就导致如下问题：

- 当发生 LLVM TB 到 QEMU TB 的切换时，由于 LLVM TB 并不会保存 *cc_dest*，*cc_src* 之类的信息，如果后面的 QEMU TB 需要读取 *cc_dest* 进行标志位计算时，计算的结果将会出错。
- 当发生 QEMU TB 到 LLVM TB 的切换时，由于 QEMU TB 并不会直接更新 eflag，后面的 LLVM TB 直接使用虚拟 CPU 中的标志位也会发生错误。

COGBT 使用如下解决方案处理这些问题 (参见图 3.8)：

- 从 QEMU TB 切换到 LLVM TB：取消不同种类基本块之间的直接链接，QEMU TB 通过上下文切换回到翻译器后，由翻译器调用 *cpu_compute_eflags* 计算所有延迟计算的标志位，写入虚拟 CPU 的 eflag 寄存器。之后再切换到 LLVM TB。
- 从 LLVM TB 切换到 QEMU TB：设置虚拟 CPU 中的 *cc_op* 为 *CC_OP_EFLAGS*，将最新的 eflags 值写入 *cc_src*。后续 QEMU TB 读取 *cc_op* 时可知所有标志位都已计算并位于 *cc_src* 中，可以直接使用。

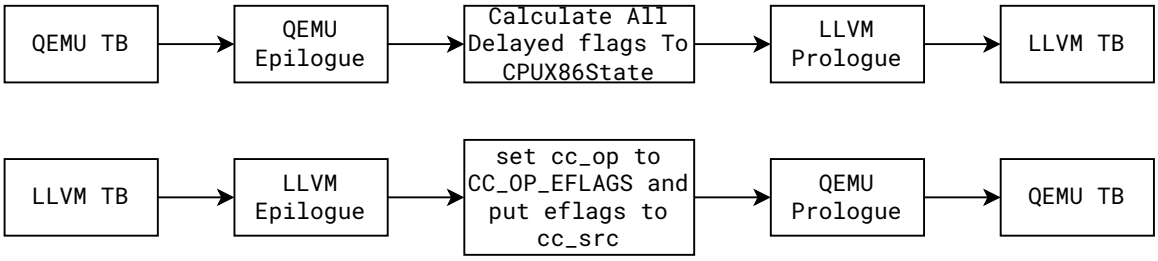


图 3.8 不同类型翻译基本块之间切换的方式

通过将不同类型的翻译基本块在虚拟 CPU 中的差异消除掉，QEMU 与 LLVM 翻译代码即可正确地互相切换。

3.2.3 典型指令翻译

COGBT 的翻译流程大致分为翻译单元提取、指令反汇编以及指令翻译三部分。COGBT 以基本块作为翻译单元，根据代码挖掘得到的基本块入口地址，向后线性扫描直到遇到终止指令。终止指令包括分支跳转、函数调用以及系统调用。从入口到终止指令即构成了 COGBT 的一个基本块。完成基本块的提取后下一步是进行反汇编，这一步是将基本块内的指令反汇编到易于分析的数据结构中。翻译的最后一步是遍历基本块内的指令，根据反汇编信息调用对应的 LLVM 翻译函数将其翻译为 LLVM IR。后续就是使用 COGBT 的优化与代码生成模块生成 AOT 预翻译文件。整个翻译流程如图 3.9 所示。

在翻译客户指令时，COGBT 将指令翻译函数的处理流程抽象成了如下三部分：

1. 读源操作数，获取源操作数的值。
2. 根据指令语义实施运算或者访存。
3. 将运算结果写目的操作数。

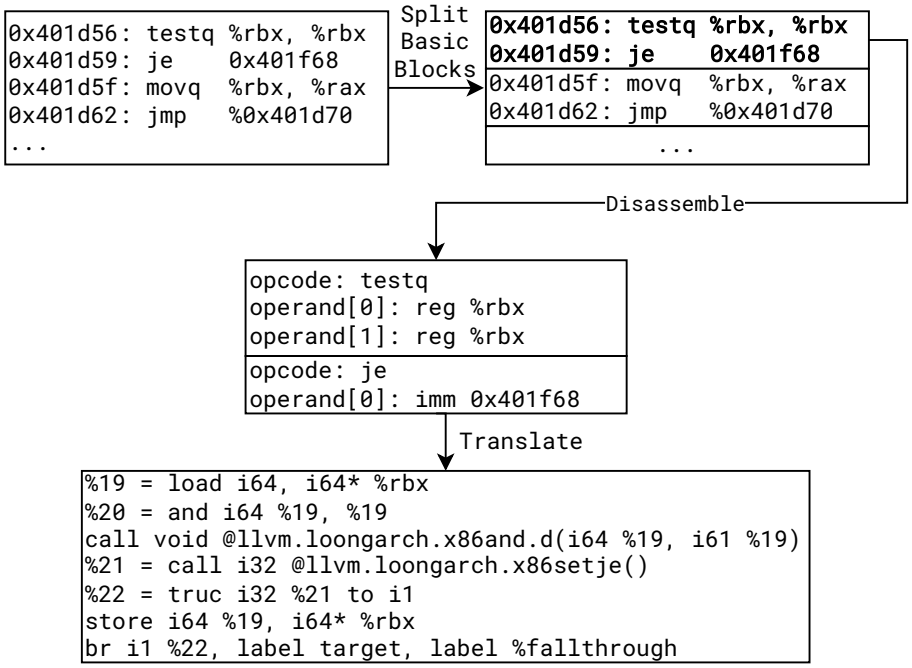


图 3.9 COGBT 翻译流程

COGBT 将读源操作数的操作封装成了 LoadOperand 函数。该函数根据操作数类型，采用不同的方式读取操作数的值并返回，具体过程如下：

1. 如果操作数是立即数，则直接构造一个 LLVM 的常量值并返回即可。

2. 如果操作数是寄存器，则需要从栈变量中或者缓存该寄存器状态的虚拟寄存器中获取操作数的值。为了降低对栈变量读写的次数，减少编译器后续栈提升优化的开销。COGBT 记录了每个映射寄存器的最新状态所在的虚拟寄存器。如果虚拟寄存器的缓存值有效的话，就可以直接返回该值，否则需要先从栈变量中获取该值并记录缓存的虚拟寄存器。图 3.10a 展示了该缓存机制。当翻译器需要读取操作数 rax 寄存器时，如果暂无该寄存器的缓存值，则从 rax 的栈变量中进行加载并记录。如果已有，直接返回即可。

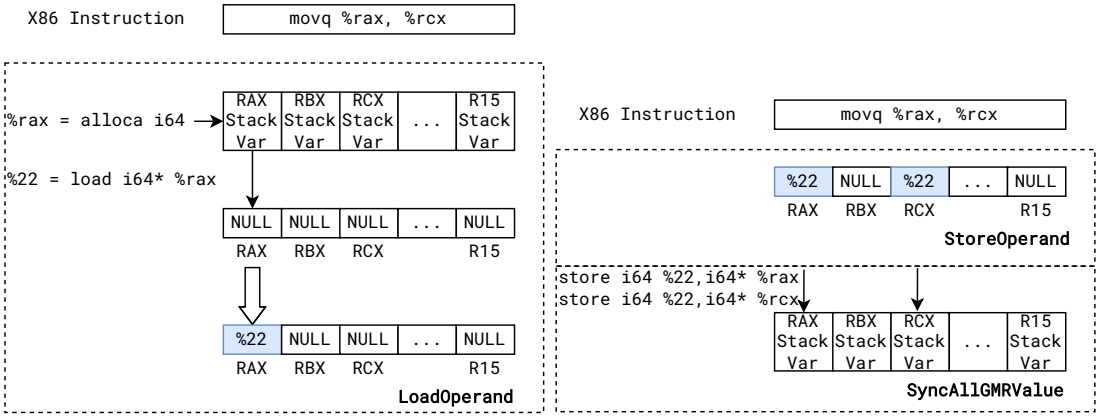
3. 如果操作数是内存变量，则需要先计算该变量的地址，再实施访存。X86 存在多种寻址模式，如基址寻址、变址寻址、基址加变址加偏移等。不管何种寻址模式，内存地址都可以用公式 segment+base+index*scale+displacement 来计算。segment、base、index 分别是段寄存器、基址寄存器与变址寄存器。scale 是比例因子，值可以取 1，2，4，8。displacement 是偏移量。LoadOpreand 使用 CalcMemAddr 来计算内存地址，该函数依次判断这些寄存器或者偏移值是否有效，有效则按照公式进行计算累加，最终返回该内存操作数的访存地址。之后就可以对该地址实

施访存了。

COGBT 将结果写回目的操作数的方式与读取源操作数方法基本一致。该过程由函数 StoreOperand 进行封装。X86 的目的操作数只有寄存器或者内存，对这两种操作数的处理方式如下：

1. 如果目的操作数是寄存器，且该寄存器未被映射，则直接写入虚拟 CPU 内存。如果寄存器被映射，则更新对应的缓存值。当基本块翻译结束或遇到同步点时再写回栈变量。图 3.10b 展示了指令 *mov %rax, %rcx* 的写入过程。假定此时 rax 映射寄存器的缓存值是%22，则调用 StoreOperand 后，rcx 的缓存值被更新为了%22。之后读取 rcx 操作数将直接返回缓存值%22。当基本块翻译结束，或者翻译块遇到分支时 SyncAllGMRValue 会将所有为 Dirty 的缓存值同步回栈变量。

2. 如果目的操作数是内存，则使用 CalcMemAddr 计算内存地址，并将值写入即可。



(a) COGBT 源寄存器操作数加载过程　　**(b) COGBT 目的寄存器操作数写回过程**

图 3.10 COGBT 寄存器操作数的读写

到目前为止，本小节已经介绍了翻译器实现操作数读写的方式，本节接下来将对几类典型的 X86 指令的翻译做简要介绍。如算术逻辑类指令、串操作指令、分支指令、系统调用指令等。

1. 算术逻辑类指令：对于普通的算术逻辑指令，LLVM 大部分都有与之对应的 IR。这些指令的翻译方式如下：

1) 使用 LoadOperand 读取源操作数。

2) 选择对应的 LLVM IR 计算运算结果。

3) 使用 StoreOperand 将结果写回目的操作数。

4) 调用标志位计算函数完成标志位的更新。

算法 1 给出了 add 指令的翻译过程。

算法 1 add 指令的翻译过程

```
1: function TRANSLATE_ADD(GuestInst)                    ▷ Translate an add instruction
2:     Src1 ← LoadOperand(GuestInst.getOpnd[0])
3:     Src2 ← LoadOperand(GuestInst.getOpnd[1])
4:     Dest ← CreateAdd(Src1, Src2)
5:     StoreOperand(Dest, GuestInst.getOpnd[1])    ▷ Assuming Opnd[1] is the dest operand
6:     CalculateEflags()
7: end function
```

2. 串操作指令：X86 有一系列针对字符串的操作指令，如比较指令 (cmps)、保存指令 (stos)、移动指令 (movs) 等。这类指令可以通过加上 REP/REPZ/REPNZ 前缀实现循环。在翻译这种带 REP 前缀的串操作指令时，COGBT 会在 LLVM 翻译函数中创建了几个翻译基本块来模拟循环的语义，基本块之间的跳转关系如图 3.11 所示。

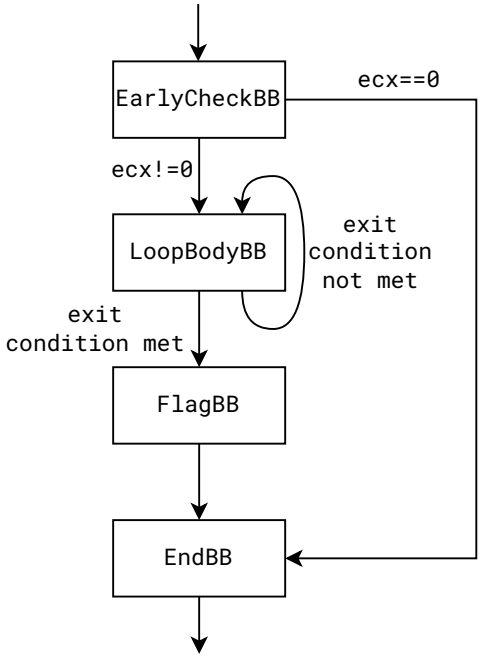


图 3.11 带 repz 前缀的串操作指令的翻译方式

EarlyCheck 基本块会先判断循环计数器 ecx 是否是 0，如果是 0 则可以直接结束进入 End 基本块翻译后续指令，否则进入 LoopBody 基本块。LoopBody 基

本块根据串操作的类型实施具体的串操作。如对于 cmps 比较指令，该基本块会从 rsi、rdi 寄存器指向的内存单元取出操作数进行比较，同时更新 rsi、rdi 的值。执行完操作后该基本块会检测退出条件是否已满足，如计数器是否为 0，两字符是否相同 (repnz cmps) 等，如果满足退出条件则退出进入 Flag 基本块，否则继续循环。Flag 基本块更新标志位寄存器后即跳转到 End 基本块。

3. *分支跳转指令*：分支指令是基本块的终止指令，分为直接跳转与间接跳转。直接跳转又以条件跳转 (jcc) 居多。X86 的 jcc 指令根据 eflags 中的标志位判断条件是否成立来确定是否跳转。COGBT 使用了 LoongArch 的 LBT 扩展来模拟 eflags 寄存器。翻译这类指令时，翻译器首先需要调用 SyncAllGMRValue 将 X86 寄存器的缓存值同步到栈变量中，然后使用 LBT 指令来得到条件判断的结果，根据结果跳到 Target 或者 Fallthrough 翻译块出口。其中，Target 出口对应跳转目标，Fallthrough 出口对应跳转失败的后继。每个出口都先生成一条 LLVM 的 cogbtexit 伪指令用以实现基本块链接，紧接着就是保存后继基本块地址并执行上下文切换。

4. *系统调用指令*：X86 提供 syscall 指令供用户程序使用内核的系统调用。COGBT 使用了 QEMU 的系统调用子模块来虚拟 X86 的系统调用。翻译该指令时，COGBT 首先将下一条指令的 pc 保存到虚拟 CPU 中，同时调用 FlushGMRValue 将所有虚拟寄存器的状态从栈中写回到虚拟 CPU。之后在翻译代码中调用 heper 辅助函数返回 QEMU，由 QEMU 进行系统调用的模拟。

5. *浮点向量类指令*：目前 COGBT 尚未对 X86 的浮点与向量指令进行直接翻译，而是采用与 QEMU 一样的机制，通过调用 QEMU 的 helper 函数来实现指令的功能。在浮点与向量的翻译函数中，COGBT 首先使用 LoadOperand 完成操作数的加载，之后根据指令类型与特征调用合适的 helper 完成计算，计算完成后再使用 StoreOperand 来完成操作数的写入。

3.2.4 标志位翻译

前一小节已经介绍了一些典型指令的翻译过程。其中提到在翻译算术逻辑类指令时需要对标志位进行处理。本节将对其进行更细致的讨论。首先将论述标志位翻译在 X86 指令翻译中的重要性，然后介绍常规的翻译方式以及这种翻译方式的劣势，最后将介绍 COGBT 使用的 LBT 硬件辅助翻译方式并提出在 LLVM IR 中添加特殊硬件指令支持的方法。

在 X86 的指令中，不少指令在完成其运算逻辑之后需要根据运算结果更新标志位。以 add 指令为例，在完成操作数的求和之后，该指令会根据运算结果设置 OF，SF，ZF，AF，CF 以及 PF。这些需要更新标志位的指令在 X86 程序的执行中占据了不小的比例，图 3.12 展示了标志位定值指令在一些测试程序运行期间所占动态指令数的比例。可以看到，在 CoreMark 等典型测试程序中，产生标志位的指令占据 43%。如果对其翻译不当，很容易造成性能瓶颈。

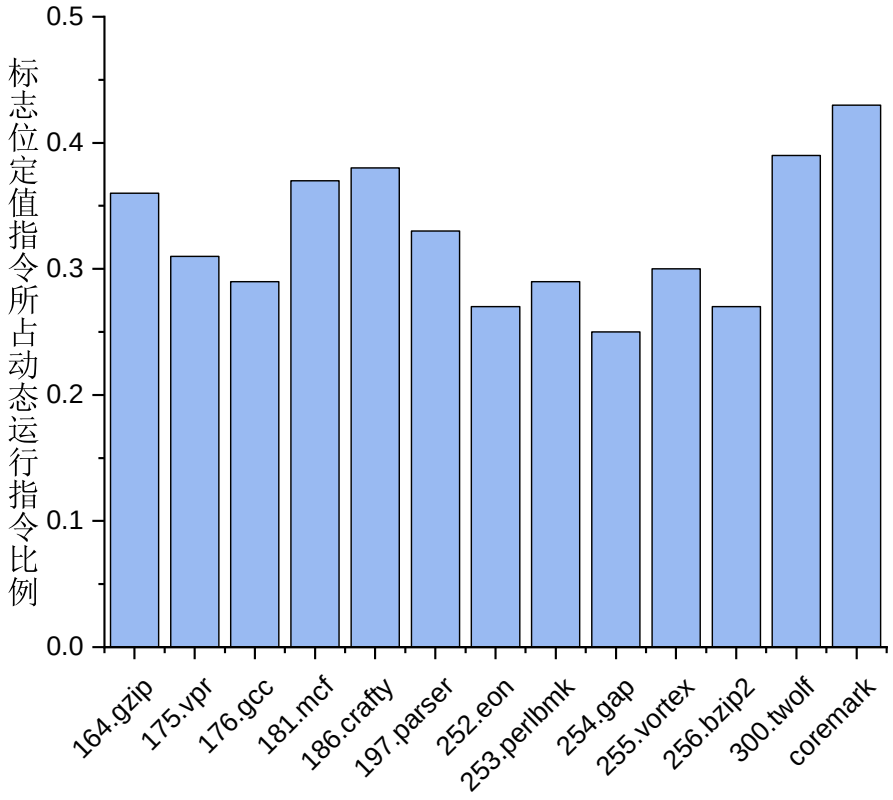


图 3.12 X86 标志位定值指令在动态运行指令中的占比

要使用纯软件进行翻译，首先需要选择一种对象来映射 X86 的标志位寄存器。由于标志位寄存器使用的频繁性，一般可以使用宿主机的寄存器来对其进行映射，而对于基于 LLVM 的翻译器来说，栈变量映射具有较大的优势。选定映射方式后，只需和其余通用寄存器的使用方式一样通过操作栈变量即可模拟 eflag 寄存器的状态。在翻译指令的标志位时，翻译函数首先判断该指令是否会定值标志位。如果需要更新标志位，则调用对应的标志位计算函数来实施翻译。以 sub 指令对 CF 标志位的计算为例，当两个操作数相减需要向前借位时该标志位会被定值。标志位计算函数可以将两个源操作数进行比较，如果被减数更小，则置 CF 标志。该标志位的计算所翻译出的 LLVM IR 如表 3.3 所示：

表 3.3 翻译 sub $0x18, %rsp 指令对 CF 标志位的定值

%32 = sub i64 %27, 24	
%35 = icmp ult i64 %27, 24	//比较两源操作数
%36 = select i1 %35, i64 1, i64 0	//确定 CF 的值
%37 = load i64, i64* %eflag	
%38 = and i64 %37, -2	//清除 eflag 的 CF 位
%39 = or i64 %38, %36	//或上 CF 的计算结果
store i64 %39, i64* %eflag	//eflag 写回

可以看到，使用纯软件翻译一个标志位就需要产生数条 IR，而实际指令产生的标志位大多为多个，翻译这些标志位所产生的 IR 将非常庞大。即使 LLVM 的优化 pass 可以消除一部分冗余，但最终翻译出的机器指令依然比较低效。此外，使用纯软件翻译标志位的方式需要较大的翻译工程量。X86 不同指令对相同标志位的定值方式大多不一致。仍以 CF 标志为例，移位指令对 CF 的定值为最后一个被移出的比特位，加法指令对其定值为求和进位，乘法指令对其定值为乘法结果是否产生高位等。翻译器需要为同一个标志的不同指令实施单独的翻译，提高了翻译的难度与出错概率。

由于纯软件翻译具有翻译低效与易错这些问题，不少处理器为了提高翻译效率，减少指令集之间的语义差距，在硬件中加入了一些二进制翻译辅助支持。这些支持总体上可分为用户无感知与用户感知两类。前者是在处理器微架构层面对二进制翻译作针对性的调整，对翻译器是透明的。典型的例子有 Rosetta 2 实现的 X86 TSO 强内存序；后者是提供特定的指令接口供用户操纵，需要用户显示调用从而完成二进制翻译的特定功能。典型的例子是 LoongArch 提供的 LBT 扩展指令集。

对于用户无感知的硬件翻译支持，无需翻译器进行干预，所有优化由底层硬件自动进行，因而具有较强的通用性。然而对于用户感知的硬件支持，需要由翻译器显示地调用对应的指令接口。这些指令接口在无 IR 的翻译器上有较直接的实现 (翻译器直接生成目标指令即可)。但是在基于 LLVM IR 的翻译器中，这些特殊指令往往没有与之相对的 IR，目前也尚未有通用的方法在 LLVM 之上使用这些硬件扩展。本文基于 LLVM IR 中的 Intrinsic 机制，提出了一种通用的方式在 LLVM 中加入特定硬件支持。通过在 LLVM 后端添加对应的 Intrinsic、指令选择

与机器指令编码，可以在 IR 层使用特殊的硬件辅助指令。下面将以 LoongArch 的 LBT 扩展指令为例介绍该方法的实施流程。

COGBT 借助了 LoongArch 指令集的 LBT 硬件辅助来实施标志位计算。为了弥补软件翻译的不足，LoongArch 在其内部实现了一个模拟的 eflags 寄存器，并定义了一系列操作该寄存器的指令。这些指令对标志位的操作基本与实际 X86 指令一致。例如 X86add.d 能完全模拟 add 指令对 eflags 的操作。通过使用 LBT 扩展指令，只需一条指令开销就能解决标志位的翻译，从翻译实现以及效率来说都是较优的。由于 LBT 扩展指令属于特殊硬件扩展，要在 LLVM IR 中使用该类指令需要在编译器后端添加支持。

如图 3.13 所示，LLVM 后端从 IR 到机器指令分别需要经过以下几个流程：首先 SelectionDAGBuilder 会根据 LLVM IR 建立一个供分析与指令选择的 SelectionDAG 有向无环图，图的形式更容易进行依赖性分析与指令匹配。之后会在图上做类型与数据的合法化 (Legalization)，如将 IR 中的数据类型拆分或者扩展成后端寄存器支持的数据类型。合法化完后开始进行指令选择，LLVM 的指令选择是从图的根部开始进行模式匹配，当满足匹配表中第一个模式时就将该节点替换为更底层的机器指令节点。指令选择完成后是一个机器指令构成的 DAG 图，需要通过指令调度将其变成一个线性的指令序列。指令调度的过程本质上是一个拓扑排序的过程，即根据数据、控制依赖性进行排序。合理的调度能充分利用硬件的并行性，隐藏指令延迟。经过调度后得到的 MachineInstr 经过寄存器分配后基本就能与后端的机器指令相对应了，此后代码发射就能根据要生成的文件格式得到汇编或者可执行文件。

要在 LLVM 后端中添加 LBT 的支持，首先需要在 IR 层面提供访问接口。COGBT 使用了 LLVM 中的 Intrinsic 机制来定制这些 IR 接口。Intrinsic 是一种特别的 IR，原用于实现编译器的内建函数。COGBT 将 Intrinsic 的使用进行了推广，将其用作衔接任意特殊硬件指令的通用手段。当硬件支持了某些特殊指令时，只需添加对应的 Intrinsic 即可供翻译器直接使用。定义 Intrinsic 需要在 LLVM 的 td 文件中声明该内建函数的特征，如返回类型，输入参数类型以及该函数的相关标记 (是否访存等)，LLVM 会根据声明生成相应的代码文件，之后翻译器可通过使用这些 Intrinsic 来使用硬件提供的特殊功能。图 3.14 是 x86add.d 所对应的 Intrinsic，表示无返回值，接收两个 64 位的参数，并且使用默认函数标记。

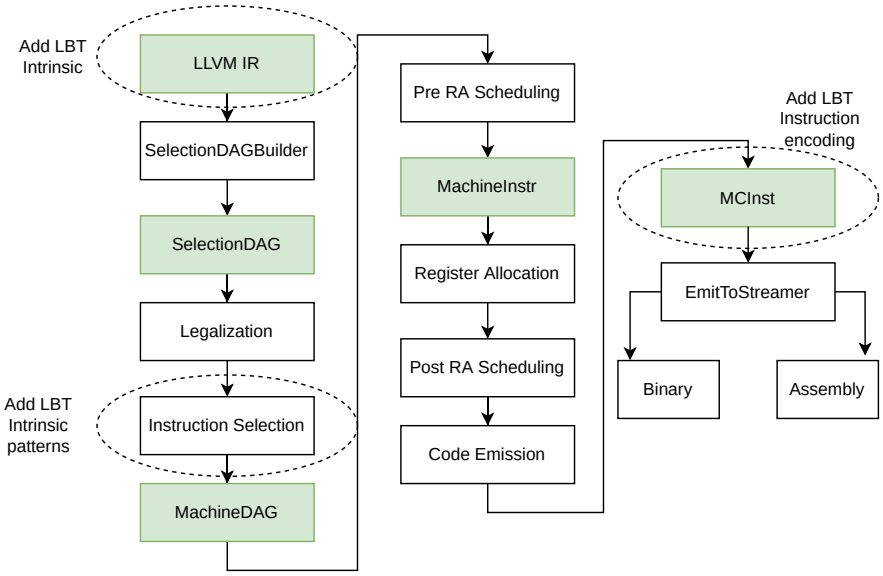


图 3.13 LLVM 后端中添加 LBT 指令支持的方式

```
def int_loongarch_x86add_d : Intrinsic<[],
    [llvm_i64_ty, llvm_i64_ty], [], "llvm.loongarch.x86add.d">
```

图 3.14 x86add.d intrinsic 声明

完成 Intrinsic 接口的定义之后，下一步是在指令选择阶段添加用于指令匹配的模式。LLVM 在指令选择时使用一个匹配表对当前节点进行匹配，如果参数符合匹配表中的约束，则贪心地选择该指令进行替换。要生成匹配表，需要在后端的 td 文件中加入匹配模式的描述。匹配模式首先描述待匹配的 DAG 节点，然后描述要替换的机器指令节点。以图 3.15 为例，这段代码定义了要匹配的 int_loongarch_x86add_d 节点，该节点需要满足操作码是 int_loongarch_x86add_d Intrinsic，且两操作数需要是 64 位通用寄存器，如果 DAG 节点满足条件，则会将其替换成 X86ADDD 机器指令节点。

```
def : LoongArchPat<(!cast<Intrinsic>("int_loongarch_x86add_d")
                                (i64 GPR64Opnd:$rj), (i64 GPR64Opnd:$rk)),
                   (!cast<Instruction>(X86ADDD) GPR64Opnd:$rj, GPR64Opnd:$rk)>
```

图 3.15 x86add.d 指令匹配模式

经过指令选择以及指令调度后，最后需要再描述 X86ADDD 机器指令的编码格式供机器指令生成。由于 LBT 指令之间存在对内部标志位寄存器的依赖关系，需要在定义指令的时候声明该 LBT 指令对标志位寄存器的影响，之后再描述机器指令的操作数信息与编码信息即可。由于 LBT 指令的显式操作数都是通

用寄存器，可以直接复用其他常规指令的寄存器操作数声明。编码信息则可根据指令手册进行描述。

以上描述了在 LLVM 中加入硬件辅助的一般流程，其余的硬件指令也可采用同样的方式进行支持。

3.3　COGBT 优化策略

影响二进制翻译器的效率的因素主要有两个。一个是翻译代码质量，翻译代码越精简，与客户指令语义越接近，性能损失越小。另一个是上下文切换的频次，上下文切换过于频繁将导致大量的上下文同步开销，影响性能。本节首先介绍几个比较重要的 LLVM 优化 pass，如通用优化 mem2reg、instcombine、自定义优化 LBTEliminate。这些优化能消除 IR 中的冗余，将代价高昂的访存操作提升为寄存器操作，提高翻译代码质量。其次会介绍 COGBT 的基本块链接优化。通过设计 LLVM 的 cogbtexit 伪指令将翻译基本块链接起来，可以减少翻译基本块上下文切换的次数，减少同步开销。最后会介绍如何在 LLVM 翻译块中加入寄存器映射优化功能，减少翻译基本块入口与出口的状态加载与写回。

3.3.1　LLVM 通用优化

COGBT 在基本块翻译完成后，会对 LLVM 翻译函数实施通用优化。基本做法是创建一个 PassManager 并添加-O2 级的优化 pass，之后将这些 pass 运行到该翻译函数上。LLVM 的优化 pass 非常丰富，如死代码消除、指令融合、窥孔优化、栈提升、循环不变量代码外提等。这些优化对于消除翻译冗余，提升 IR 质量非常重要。本小节将简要介绍栈提升优化 mem2reg 以及指令融合窥孔优化 instcombine。

mem2reg 的作用是将 IR 中的栈变量进行消除，将昂贵的访存操作替换成虚拟寄存器操作。这个优化本质上是构建一个 SSA 形式的 IR，实现的是一个 SSA 构建算法。LLVM 假定所有局部变量都是位于入口基本块 (entry) 的 alloca 指令 (不在 entry 的 alloca 也可以简单的移动到 entry)，算法目标是将这些 alloca 指令以及对其的 load/store 指令进行消除。首先，LLVM 筛选出所有潜在能消除的 alloca 指令，它们需要满足两个条件。其一是不带 volatile 标记，其二是直接读写的，即不会有取址操作。满足约束的 alloca 将会被后续优化。接着，LLVM 会先实施一些简单优化：

1. 没有用到的 alloca 指令会被移除。

2. 如果 alloca 的栈变量只有一个基本块对其定义 (store 某个值给该变量)，则所有被其支配[1]的 load 指令将被替换成定义值，并直接消除该变量及对应的 load/store。

3. 只在同一个基本块进行读写的 alloca 变量可以直接将每个 load 的值替换成最近 store 的值，而不需要遍历控制流图 (CFG) 以及插入 phi[2]指令。

最后，LLVM 开始对普通 alloca 实施提升。提升过程分为 phi 指令的插入与值的重命名两阶段。由于 SSA 形式的 IR 不允许对同一个变量进行定值，如果条件分支的两个后继基本块都对变量进行了定值，并且这两个后继的聚合基本块 (join block) 用到了该变量，那么当 IR 提升为 SSA 之后，两个定值的指令的名字将不同，比如分别命名到了%x, %y，聚合基本块就需要使用phi(%x, %y)来使用该值了。

LLVM 首先会构建支配树[3]，然后基于支配树计算定值基本块 (store 该 alloca 变量的基本块) 的支配边界[4]。这些支配边界就是需要进行 phi 指令插入的节点。插入 phi 指令后下一步就是消除该 alloca 变量以及对应的 load/store，这一步称为值的重命名。基本思路是将 store 直接删除，将 load 的值替换为最近的 store 值或者 phi 值。图 3.16 展示了使用了 mem2reg 优化前后 IR 的变化。

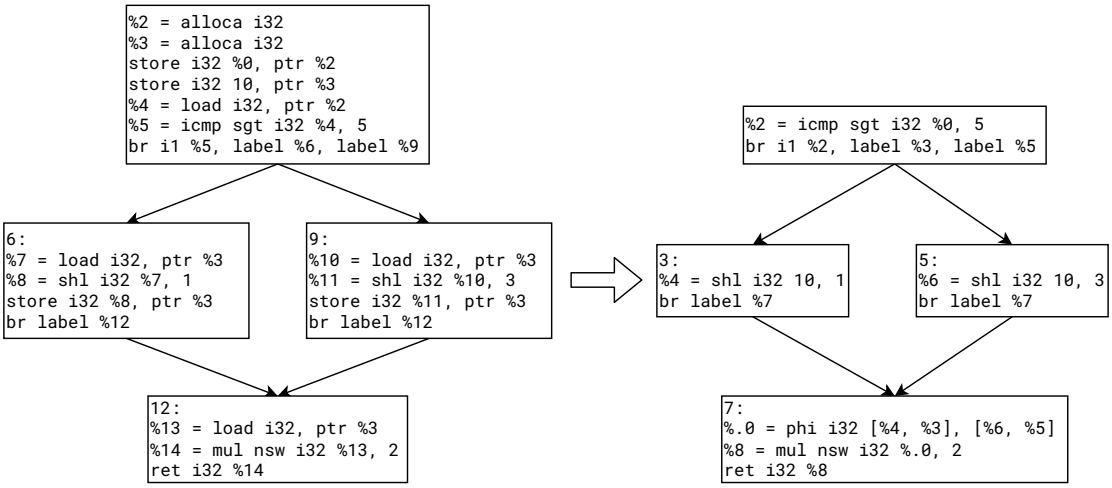


图 3.16 mem2reg 优化示意图

[1]从入口节点到 B 节点的所有路径都必然经过 A 节点，则说 A 支配 B

[2]SSA 中一种特殊指令，根据前驱基本块来产生不同的值

[3]基于节点支配关系生成的树

[4]B 是 A 的支配边界当且仅当 A 支配 B 的某个前驱并且 A 不支配 B

图中%2 号栈变量由于只在入口基本块被定值与使用，因此算法实施前的简单优化就会直接将其消除，并将用到该变量的%4 号 load 指令替换成其定义值%0。而%3 号栈变量则需要先计算所有定义基本块 (entry，block6，block9) 的支配边界，计算结果为 block12，后面就在该基本块中插入 phi 节点，并通过重命名消除各基本块对该变量的 load/store。

instcombine 是一个窥孔优化的 pass。其目标是通过模式匹配将复杂的指令模式转化成更简单的模式。如常量传播、折叠，强度削减，代数化简等。该 pass 分两阶段执行，第一阶段先对函数内的 IR 进行简单优化。将死代码指令消除，常量表达式折叠，循环内指令下沉等。简单优化后将指令装入工作链表 (worklist) 中。第二阶段开始执行工作链表的不动点算法，不断从中取出 IR 尝试窥孔优化，当指令优化成功后继续将优化后的指令放入工作链表中，直至最终链表为空，此时到达了优化的不动点。

对于每条 IR 指令，instcombine 均实现了大量窥孔的模式库。当尝试对一条 IR 进行优化时，instcombine 会根据 IR 操作码的不同，调用相应的访问函数，访问函数会对操作数的数据链进行优化模式匹配，如果匹配中则实施对应的优化。以 add 指令为例，优化函数对该指令实现了大量算术运算的代数化简模式，如将正负数相加直接优化成 0，将 A*B + A*C 优化成 A*(B+C) 等。图 3.17 显示了一个 instcombine 优化的例子，基本块 3 与基本块 5 中的运算指令均是常量运算，因此可以进行常量折叠与传播到基本块 7，并与%8 中的乘法合并，这样即可消除这些常量表达式，最后通过 simplifycfg 即可将无用基本块进行消除，得到最简的结果。

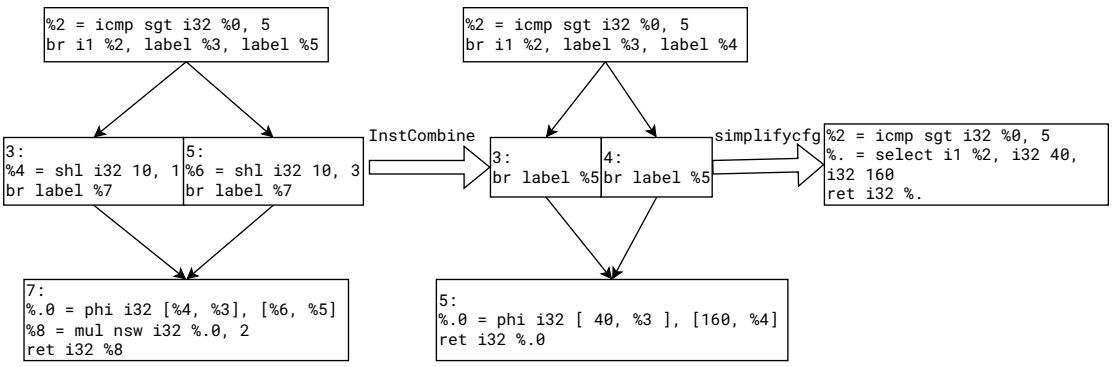


图 3.17 instcombine 优化示意图

COGBT 还使用了大量其他 LLVM 优化 pass，通过综合使用 LLVM 各种优

化库以及编写自定义优化 pass 可以较方便地实施各种跨指令级优化，得到非常精简的翻译 IR，进而提高翻译代码效率。

3.3.2 冗余 LBT 消除优化

冗余 LBT 消除是 COGBT 实现的一个消除冗余标志位计算的优化 pass。在 COGBT 中，标志位计算是通过 LoongArch 提供的 LBT 扩展指令实现的，每条会产生标志位的 X86 指令都会被翻译为运算指令与对应的 LBT 指令。由于 LBT 指令在 IR 中是以定制 Intrinsic 的形式存在，普通 LLVM 优化 pass 并未对其进行优化，因此在一个基本块内会存在较多冗余 LBT 指令。以图 3.18 为例，在该基本块中，产生标志位的指令有 shl 以及 test 两条指令。两条指令翻译时均生成了 LBT 指令用于计算标志位，其中 x86sll.w 与 x86and.w 均会修改所有 LBT 标志位，由于 x86sll.w 更新的标志位并没有被使用即被后续的 x86and.w 覆盖，因此该指令是一个死亡定值 (dead define)。

```
400289: mov   %edx, %r9d                          ...
40028c: shl   %cl,  %r9d  ──→  x86sll.w   $r7, $r6  //define all LBT flags
40028f: test  %esi, %r9d       x86and.w   $r4, $r5  //define all LBT flags
400292: jne   0x400848         setx86jne  $r4
                                                  ...
```

图 3.18 LBT 指令中的冗余定值

基于以上观察，可以对 LBT 定值的标志位变量进行活性分析，将不产生存活标志变量的 LBT 指令进行消除。要实现这一优化，COGBT 只需按照 LLVM 的 pass 接口实现一个冗余 LBT 的消除 pass，之后再使用 PassManager 将优化施加到 IR 即可。

优化算法的第一步是计算每个翻译基本块的 livein 与 liveout 集合。livein 集合是指进入该基本块时存活的标志位变量，liveout 则是指退出时依然存活的变量。通过计算基本块的 liveout 集合可以了解哪些标志位会被后继使用，进而可以分析出在基本块内任意一点的标志位变量的存活性。由于判断标志位定值是否存活依赖后继指令对该定值的引用，因此可以建立反向数据流方程并使用迭代算法来求解基本块的 livin 与 liveout 集合。具体而言，算法首先初始化退出基本块的 liveout 为标志位变量全集、初始化其余基本块的 livein 与 liveout 为空集。将退出基本块的 liveout 初始化为全集是因为无法知道后继基本块对标志位的使用情况，需要保守认为后继会使用所有标志位变量；之后使用如下两个数据流方

程来迭代更新基本块的 livein 与 liveout 集合：

$$livein[b] = use[b] \cup (liveout[b] - def[b]) \tag{3.1}$$

$$liveout[b] = \cup_{s \in succ[b]} livein[s] \tag{3.2}$$

方程 3.1 中的 use 集合表示在基本块中被使用但是并未被定值的标志位变量，def 集合表示在基本块中被定值的标志位变量。该方程表示在基本块入口存活的标志位变量要么在本基本块内被使用，要么在后继某个基本块被使用；方程 3.2 中的 succ 集合表示某个基本块的所有后继基本块，该方程的含义为在基本块结束时存活的变量必然流向某个后继。COGBT 使用这两个方程来循环更新 LLVM 翻译函数内的所有基本块，当所有基本块的 livein 与 liveout 都不再变化时 (到达了不动点)，即得到了集合的求解结果，相关算法如下：

算法 2 livein 与 liveout 迭代求解算法

```
1:  function CALCULATE_LIVEIN_LIVEOUT(LLVMFunc)
2:      for each basic block B in LLVMFunc do          ▷ Initialize livein and liveout
3:          livein[B] = ∅
4:          if B is an exit block of LLVMFunc then
5:              liveout[B] = {OF, ZF, SF, AF, PF, CF}
6:          else
7:              liveout[B] = ∅
8:          end if
9:      end for
10:     while any livein or liveout set changes do
11:         for each basic block B in LLVMFunc do
12:             liveout[B] = ∪_{s∈succ[B]} livein[s]
13:             livein[B] = use[B] ∪ (liveout[B] - def[B])
14:         end for
15:     end while
16: end function
```

完成 livein 与 liveout 求解后，第二步是计算翻译基本块内每条 LLVM IR 指令产生的存活标志位。基本流程是利用 liveout 反向遍历基本块内的所有指令，遍历过程中维护一个 liveflags 集合表示执行到该指令时存活的标志位变量。liveflags

初始化为基本块的 liveout，每遍历一条指令就判断该指令产生的标志位是否在 liveflags 集合中，在集合中即表示该标志位是存活的，反之就是一个死亡标志位。计算完存活标志位后将该指令定值的标志位从 liveflags 中移除，并将该指令使用的标志位并入集合来更新 liveflags，然后继续遍历下一条指令直到处理完所有指令。处理流程见算法3：

算法 3 基本块内 LBT 指令存活标志位求解算法

```
1: function CALCULATE_LIVEFLAGS(LLVMFunc,liveout)
2:     for each basic block B in LLVMFunc do
3:         liveflags = liveout[B]
4:         for each LLVM IR Inst in B according to reverse traversal do
5:             FlagsNeedCalculate[Inst] = Gen[Inst] ∩ liveflags
6:             liveflags = (liveflags - Gen[Inst]) ∪ Use[Inst]
7:         end for
8:     end for
9: end function
```

最后，算法只需要再遍历所有 LLVM IR，将只产生死亡标志位的 LBT 指令 (FlagsNeedCalculate 为空) 从 IR 中删除即可。通过运行以上优化算法，可以移除冗余 LBT 指令，减少动态运行指令数。

3.3.3 基本块链接优化

基本块链接技术是二进制翻译中普遍使用的一项技术，通过将互相跳转的翻译块链接起来降低上下文切换的频次。链接分为直接跳转链接以及间接跳转链接。直接跳转链接只需在翻译块末尾加上跳转到目标翻译块的分支指令即可，而间接跳转链接则需要通过插入地址转换代码获取目标翻译块的入口。本小节将首先介绍一种直接跳转的链接方式，讨论这种链接方式在 LLVM IR 中实现需要面临的问题以及 COGBT 的解决方式。随后将介绍 COGBT 实现间接跳转链接的做法。

在一些动态二进制翻译器中 (如 QEMU，LATX[5])，对分支指令的翻译方式大致如图 3.19 所示。翻译器为条件跳转指令设置了两个出口分别处理分支的 Target 与 Fallthrough，翻译代码根据分支条件跳转到其中一个出口。翻译器在每个分支

[5]一款翻译 X86 的 LoongArch 动态翻译器

出口中都预留了一条链接指令槽，如图中的 *b* 指令。链接指令的目标可被修改，初始值为后继指令。当目标基本块被翻译后，翻译器就可以改写该跳转指令的目标，使其重定向到后继翻译块的入口。在此图示例中，当翻译器首次执行该翻译基本块时，该指令如同 *nop*，直接跳到该指令的后继，保存目标基本块的地址，然后跳转到上下文切换入口。切换回翻译器后，翻译器将查找、翻译该块的后继，并将 *b* 指令重定向到后继基本块的翻译代码入口。之后，当该基本块再次执行时将直接跳转到目标块执行，无需再执行上下文切换。

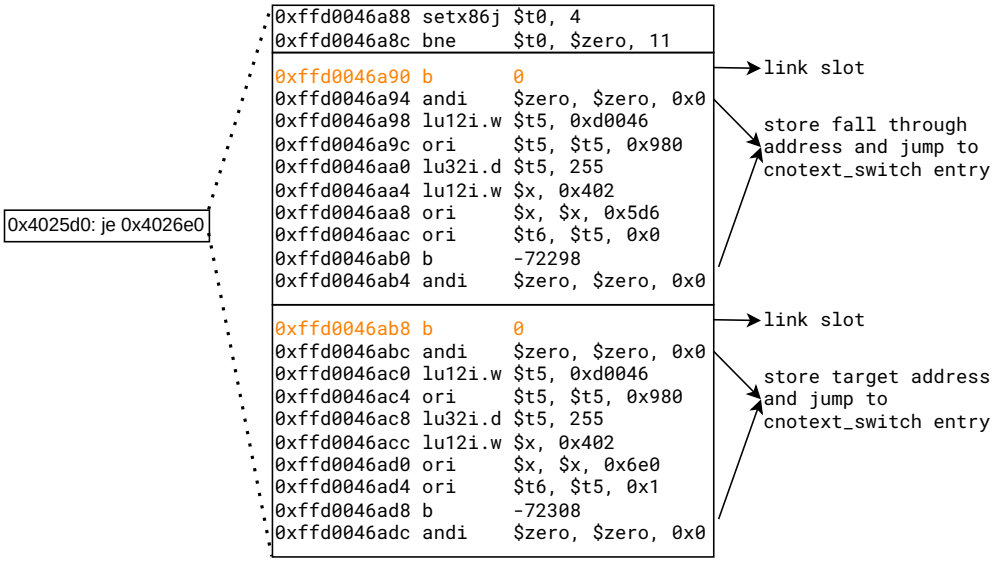


图 3.19 LATX 基本块链接实现逻辑

从这些翻译器的实现可以看出，要实施基本块链接需要在翻译指令中生成链接指令以及上下文切换指令，并且需要严格保证链接指令与上下文切换指令之间的相对关系，如此才可在首次执行时通过硬编码链接指令的偏移字段使其正确跳到上下文切换的入口。然而，在 LLVM IR 中要保证上述约束并不容易：(1) 在 LLVM IR 中跳转只能存在于翻译函数内的基本块之间，无法生成跳转指令既能在 LLVM 基本块之间跳转又能跳到函数外的某个入口；(2) 在 LLVM 函数内的直接跳转很可能会被直接优化掉而无法生成链接指令槽；(3)LLVM 的指令调度会根据指令依赖关系调整指令顺序，因而可能导致链接指令的偏移失效。

(1) 与 (2) 说明无法使用现有的 LLVM IR 来实现添加链接指令这一目标，(3) 说明链接指令与后续的上下文切换指令最好是原子的。那么一种比较合理的处理方式是设计一种新的“IR 指令”，该指令能原子地生成链接指令与切换指令。COGBT 在 LLVM 中设计了一条 cogbtexit 伪指令，该指令在 IR 优化、调度、寄

存器映射过程中始终不会被展开，而是会被认为是一条单独 IR 指令。只有在最终生成机器指令之前才会将其展开成如图 3.20 所示的指令序列，从而保证了这个序列的原子性。cogbtexit 在 IR 层的接口也是使用了 Intrinsic 机制，该指令接收后继基本块地址与上下文切换入口地址两个参数，在伪指令展开 pass 中会根据参数将该指令改写为符合要求的机器指令序列。

```
B       4                       // link slot
lu12i.w $ra, $ra, 0x424
ori     $ra, $ra, 0x979
st.d    $ra, $ra, $s2, 0x80 // store next pc to env
lu12i.w $ra, 0
ori     $ra, $ra, 0
lu32i.d $ra, 0
lu52i.d $ra, $ra, 0         // load epilogue address
jirl    $ra                 // jump to epilogue
```

图 3.20 COGBT 直接跳转链接指令序列

解决完链接指令的生成问题后还需要处理链接指令位置的记录问题。为了改写链接指令的跳转目标，翻译器需要知道每条链接指令的位置以及该指令的目标基本块。一种直观的思路是让 cogbtexit 携带一些信息并保存到 AOT 文件 (是一种 ELF 文件) 中，考虑到 gdb 等调试工具可以根据当前所执行的机器指令反向找到源代码，因此 ELF 中必然存在某种记录机器指令源代码位置的信息，这种思路是可行的。COGBT 使用了 ELF 中的 DWARF 调试信息保存机制。DWARF 是 ELF 中一种调试信息格式，可以追踪每条机器指令对应的源代码的行号、列号等信息。LLVM 在 IR 中设计了一套特殊的 metadata 来记录源代码信息，这类 metadata 通过前端 clang 在 IR 中生成，并在整个 LLVM 后端代码下降中会一直保存，因此只需要复用这些 metadata 的一些字段就可以将翻译器需要的信息保存到 AOT 中。具体而言，COGBT 在 cogbtexit 伪指令后附上了一条 DebugLoc 的 metadata 记录，该记录有两个字段，表示该 IR 对应的源代码的行与列，可以在行字段中加入 cogbt 属性表示由二进制翻译生成，在列字段中填充链接指令所在出口 (Target 或者 Fallthrough)。这样最终在 AOT 的调试段中将生成表项记录链接指令的位置以及目标基本块。

间接跳转的链接比较简单，可以直接在 IR 中调用目标基本块查找函数，根据返回值进行跳转。图 3.21 是 COGBT 在翻译 ret 指令时的处理过程，翻译器首

先按照 ret 指令的语义从 X86 栈中提取返回值，并将返回值写回到虚拟 CPU 中，写入后调用 *helper_cogbt_lookup_tb_ptr* 进行目标基本块的查找，如果该基本块已翻译则返回该基本块的翻译代码入口，否则返回上下文切换入口。之后翻译代码将根据返回值跳入目标基本块或者上下文切换。

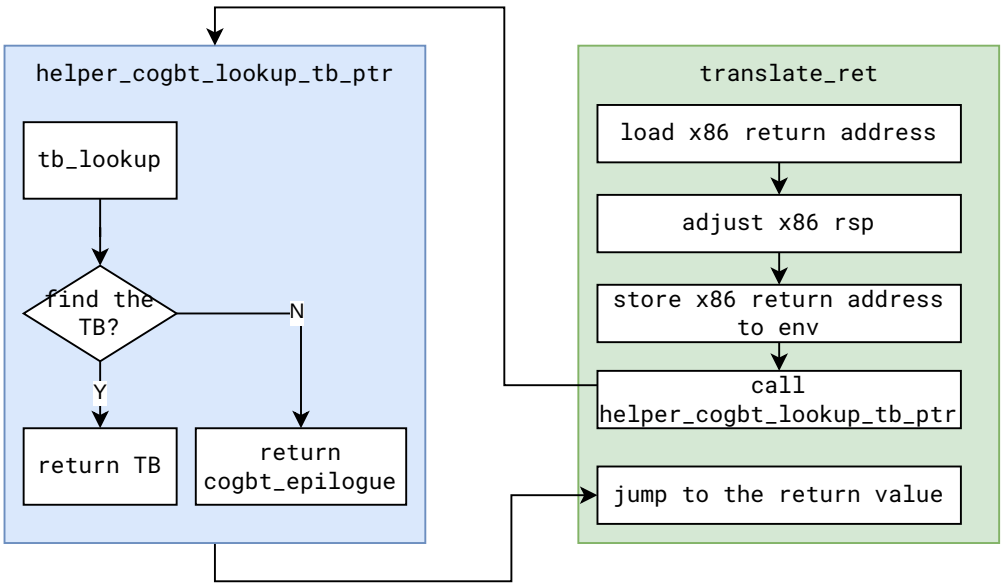


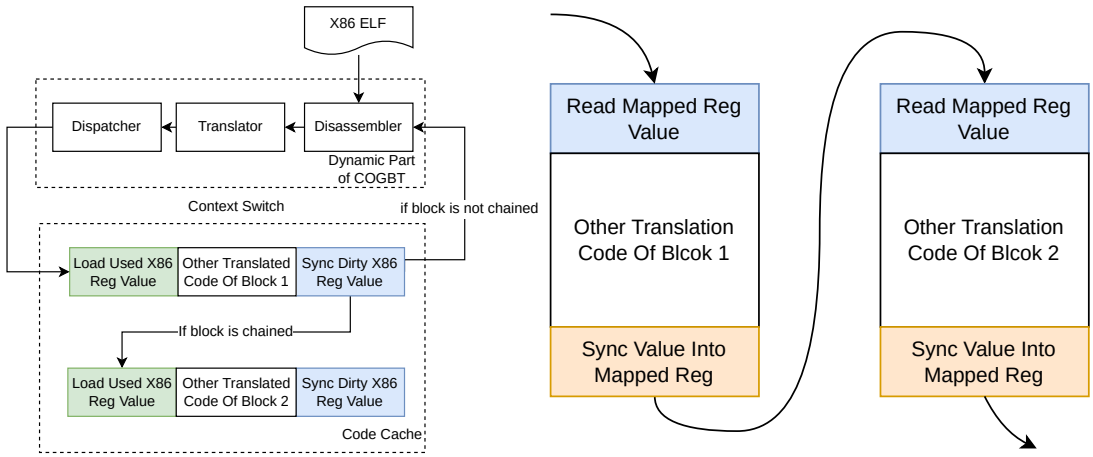
图 3.21 COGBT 间接跳转链接

3.3.4　寄存器映射优化

在 COGBT 中，一个很重要的优化是实施寄存器映射。由于 LoongArch 具有比 X86 更多的通用寄存器，可以预留一部分 LoongArch 寄存器来表示 X86 寄存器的状态。这样可以尽可能地避免从虚拟 CPU 中加载和写回寄存器值。本小节首先说明在 cogbt 中实施寄存器映射的必要性，然后介绍 cogbt 实现寄存器映射的方式以及映射方案，最后再讨论该方案需要面对的问题以及解决办法。

以图 3.22a 为例，COGBT 在未实施寄存器映射时其翻译基本块需要从虚拟 CPU 中读取 X86 寄存器状态并初始化栈变量，之后翻译代码才能操作栈变量并被提升到寄存器，当基本块退出时又需要将状态同步回虚拟 CPU。可以看到翻译代码在基本块的入口与出口均需访存，特别是当基本块被链接起来后，这些访存将对性能造成一定影响。要减少这些访存，可以使用寄存器映射来实现。假定宿主机约定 $s0 寄存器固定映射 X86 的 $rax 寄存器，并且保证在进入翻译代码之前该寄存器已经被放置最新的 $rax 值 (上下文切换时加载同步值)、退出翻译代码时最新的值也被同步到 $s0 寄存器。那么每个块的翻译代码在开始以及结束

就只需要读取或写回映射寄存器，开销较大的内存读写就可以被优化成开销较小的寄存器读写。



(a) COGBT 实行寄存器映射前的执行逻辑 **(b) COGBT 实行寄存器映射后翻译代码执行逻辑**

图 3.22 COGBT 寄存器映射优化原理

在 LLVM IR 中能直接操作对象只有变量以及虚拟寄存器，而要实施寄存器映射必须与物理寄存器进行交互，这就需要一定手段在 IR 这一层次约束物理寄存器。COGBT 通过使用内联汇编来实现对固定映射寄存器的读写。如图 3.23 所示，COGBT 利用内联汇编的输出约束将输出值%0 绑定为物理寄存器 $r27，实现了物理寄存器的读值操作。利用输入约束将输入值%10 绑定到了物理寄存器 $r27，实现了物理寄存器的写值操作。在翻译基本块时只需在入口从固定映射寄存器中读值并初始化栈变量，在出口将栈变量的值写回映射寄存器就可以实现优化访存这一目标。目前 COGBT 已经实现了对 X86 16 个通用寄存器的映射，相关映射关系见表 3.4。

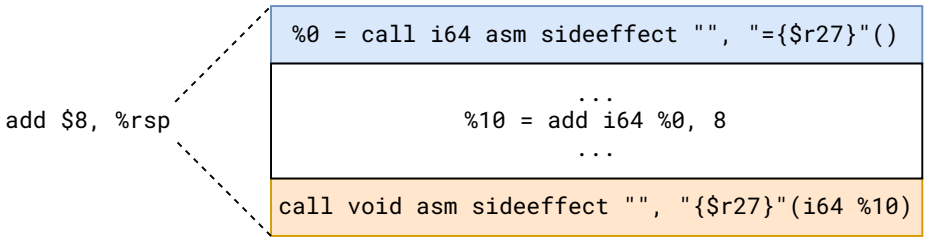


图 3.23 COGBT 寄存器绑定方式

COGBT 在实施了寄存器映射之后，要使其正常运行，需要保证读取寄存器的内联 IR 最先执行、同步寄存器的内联 IR 最后执行，如此才能确保翻译基本块在进入是读取到的映射寄存器是正确的，退出时写入映射寄存器的值是不会

表 3.4 COGBT 寄存器映射方案

X86 寄存器	映射寄存器	X86 寄存器	映射寄存器	X86 寄存器	映射寄存器	X86 寄存器	映射寄存器
RAX	$t3	RBX	$s3	RCX	$t6	RDX	$t7
RSI	$s6	RDI	$s7	RSP	$s4	RBP	$s5
R8	$s1	R9	$s8	R10	$a6	R11	$a7
R12	$t0	R13	$t1	R14	$t2	R15	$s0

被破坏的。但是由于 LLVM 在后端 IR 下降过程中会发生指令调度，因而翻译出的指令可能不能满足此约束。以图 3.24 为例。调度前，COGBT 通过内联约束读取映射寄存器 $r27 与 $r10 的值，然后参与计算。当发生调度后，后面的两条运算指令 *add i64 %4, 8* 与 *and i64 %21, 16* 由于不涉及数据依赖，因而可被调度到 $r10 的读取之前。如果调度后的寄存器分配将%22 映射到物理寄存器 $r10 上，那么就会导致 $r10 被破坏，利用内联汇编读取 $r10 的值将出错。

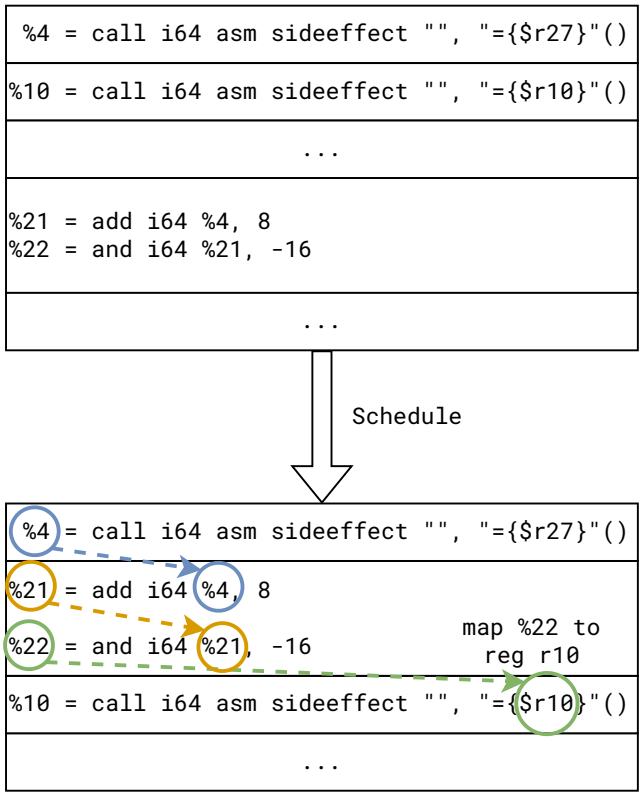


图 3.24 指令调度引发的问题

解决该问题的做法是修改 LLVM 的调度器，保证读取寄存器的内联 IR 具有最高优先级，同步物理寄存器的 IR 具有最低优先级。LLVM 在执行指令调度时会先建立调度单元，之后根据调度单元之间的依赖关系建立数据依赖图，每次调

度时会将没有依赖的调度单元放入优先队列中，然后根据处理器架构后端的功能部件数、发射宽度等参数将优先级最高的若干调度单元进行调度，同时将解除依赖后新的调度单元加入优先队列中。COGBT 通过改写内联指令的优先级保证普通指令不会调度到读值的内联指令之前，也不会调度到写值的内联指令之后，进而保证了正确性。

3.4 AOT 的生成与解析

COGBT 性能的关键在于能否把 X86 程序的代码尽可能多地识别出来并实施各种编译优化。因此需要先对 X86 程序实施代码挖掘。代码挖掘的关键在于覆盖到程序的热点基本块，对热点的优化才是实际有意义的，而对大量冷代码实行优化既不会得到明显收益而且也会增加大量编译优化开销。基于这些考量，COGBT 目前采用了基于 profile 的基本块识别方式。首先，COGBT 使用动态端运行一遍目标 X86 程序，运行时动态端将收集运行时信息，如基本块的入口地址、自修改基本块等，并将数据保存到 path 文件中。之后，静态端将读取 path 文件，识别文件中的基本块并形成翻译单元。

挖掘出的 X86 代码经过 LLVM 翻译器的翻译、优化后形成本地翻译代码，接着将通过 AOT 生成模块完成 AOT 的生成。在 COGBT 中，AOT 其实是 ELF 格式的可重定位目标文件。选用该文件格式的好处是可以与目前大量的 ELF 处理工具兼容，方便调试与查看。同时可直接复用一些 LLVM 的 ELF 生成与解析库，减少重复开发的工作量。在 AOT 生成过程中，COGBT 直接使用了 LLVM 后端目标文件生成的 pass。

完成 AOT 文件生成后，COGBT 可以直接加载 AOT 文件准备执行了。加载 AOT 的过程即是 AOT 解析的过程。解析工作包括代码段的读取、符号重定位以及基本块的注册。主要处理流程如下：

1. 使用 LLVM 的 MCJIT 模块对 AOT 文件进行解析。将其中的基本块翻译函数进行提取与收集。同时读取调试段中的 DWARF 信息将基本块链接指令的位置进行收集与记录。

2. 对翻译函数用到的外部符号进行重定位。翻译器将所有外部 helper 函数地址与 AOT 中对应的符号进行绑定，并由 MCJIT 实施符号重定位。

3. 将静态翻译基本块注册到代码缓存中。翻译器需要将已翻译好的基本块

信息注册到全局哈希表中，以方便后续的查找与执行。同时需要将翻译代码放入预设的代码缓存中。

4. 对所有翻译基本块进行链接。根据步骤 1 收集的信息改写链接指令的跳转目标。

3.5　本章小结

本章主要介绍了 COGBT 的设计和实现细节，从基本框架、指令翻译方式、优化策略等角度对系统进行了解析。

第一节从顶层视角介绍了 COGBT 的总体框架。该节首先介绍了 COGBT 动态端的基本框架，阐明了动态端如何创建运行时环境以及加载 AOT 等。随后介绍了静态端的框架，讲解了静态端代码挖掘、离线翻译优化的基本流程。最后介绍了 COGBT 的调试框架，通过采用基本块对比与基本块替换两种调试方式帮助翻译系统快速定位问题。

第二节介绍了 COGBT 的翻译模块。该节首先对比了几种翻译器的翻译模型，指出了这些模型的缺陷，并提出了 COGBT 的栈翻译模型。之后介绍了 COGBT 中上下文切换的方式，通过消除标志位的翻译差异使得 COGBT 能正确地在 LLVM 与 QEMU 翻译代码之间进行跳转。紧接着该节深入翻译模块的实现，以若干典型指令介绍不同指令翻译为 LLVM IR 的方式。最后该节详细讨论了 COGBT 使用 LBT 硬件辅助翻译标志位的方式

第三节则介绍了 COGBT 使用到的一些优化策略。首先介绍了 LLVM 的一些跨指令的通用优化如实现栈提升的 mem2reg、实现指令融合的窥孔优化 instcombine，这些优化都显著提高了翻译 IR 的质量。随后该节介绍了自定义的冗余 LBT 消除优化 LBTEliminate，通过对 LBT 定值的标志位变量进行活性分析可消除只包含死亡定值的 LBT 指令。接着该节介绍了基本块链接优化在 COGBT 中的实现方式，通过引入 cogbtexit 伪指令防止链接指令被调度与优化以及利用 DWARF 调试信息来记录链接指令的位置。最后该节介绍了使用内联约束实现寄存器映射的方式，并介绍了指令调度对寄存器映射的影响以及对应的解决办法。

第四节简要地介绍了 COGBT 使用动态端 profile 进行代码挖掘的方式，以及 AOT 的生成与解析流程。

第 4 章　COGBT 性能分析

本章对二进制翻译系统 COGBT 进行性能测试与评估。系统的测试环境为龙芯 3A5000 机器。该机器使用 LoongArch 指令集，主频为 2.3GHz，操作系统内核为 Linux 4.19。首先，本章将先测试 COGBT 静态端的代码识别率，检测基于 profile 的代码挖掘方案是否可行；其次，将对 COGBT 进行性能测试，对比在相同条件下 COGBT 与 QEMU 的性能差异；之后本章将对 COGBT 的优化进行分析。由于 COGBT 采用了与 QEMU 类似的基本块链接技术，所以两者的上下文切换都较小，不是性能差异的重点。而 LLVM 通用优化与 LBT 标志位辅助优化对翻译代码的质量会有明显提高。因此本章会先测试两者运行 SPEC 2000 CINT 的动态指令数，从指令数的角度分析 COGBT 是否能相对 QEMU 有所降低。其次，COGBT 使用到的栈翻译模型、LLVM 优化以及寄存器映射都能显著降低访存，本章也会测试在 SPEC 的动态运行中 QEMU 与 COGBT 翻译代码的访存指令占比。

4.1　代码挖掘覆盖率测试

首先对 COGBT 的代码挖掘进行测试。在 COGBT 的运行过程中，通过检测 QEMU 动态翻译的基本块个数可以知道代码挖掘的遗漏情况，之后统计静态翻译基本块在总的运行基本块的比例就能得到代码挖掘的覆盖率。从图 4.1 的结果来看，基于 profile 的基本块挖掘方案能覆盖到大部分执行代码。在 SPEC 2000 CINT 的测试结果中，该方案能达到平均 95% 的代码覆盖率。

基于 profile 的挖掘方案依赖动态采样所能覆盖到的代码，如果程序的运行比较集中，那么就能得到比较好的结果。图中 176.gcc 与 252.eon 的覆盖率分别为 84.1% 与 89.2%，相对其他程序较低。一个直接的原因是这两款测试程序的执行路径受参数与环境变量等的影响较大，对 SPEC test 集进行采样的结果与实际运行的 ref 集的执行路径有一定差异。在实际的应用场景下最好配合其他静态分析等代码挖掘方案来得到一个更加稳定与全面的代码覆盖情况。

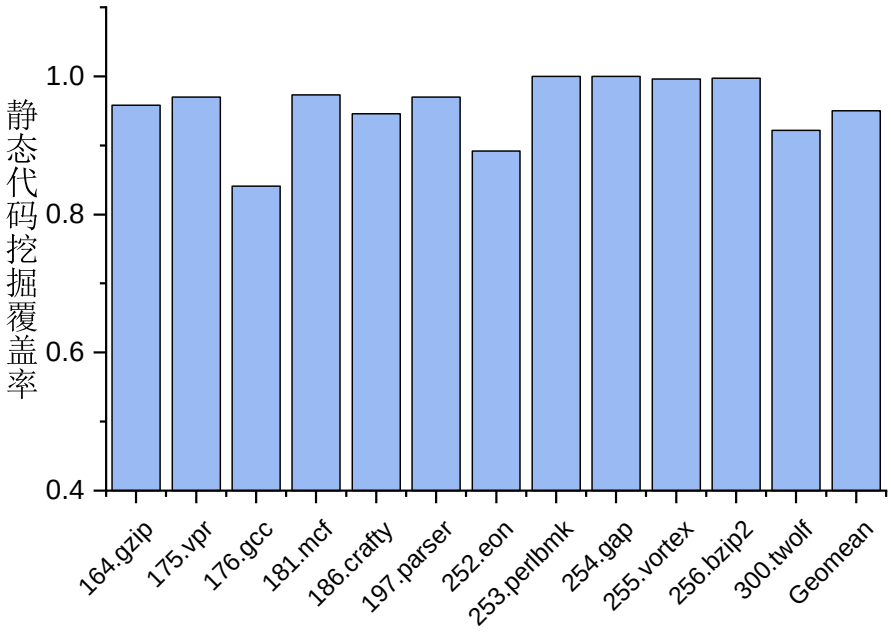


图 4.1 COGBT 静态代码挖掘覆盖率

4.2 翻译性能测试

4.2.1 测试集

性能测试主要使用 CoreMark 与 SPEC 2000 CINT 两种测试集。CoreMark 是一款面向嵌入式领域的标准测试集。包含列表处理、矩阵计算、状态机与 CRC 校验四种算法。该测试集计算单位时间内对以上算法的执行次数，单位时间执行次数越多，性能越高；SPEC 2000 是用于评估 CPU 性能的基准测试集。该测试集分为 CINT 与 CFP 两部分分别测试处理器的定点与浮点性能。CINT 与 CFP 选取了一些典型应用场景的测试程序，并对这些程序的得分进行几何平均来评估处理器的性能，较能准确地反应处理器的实际水平。COGBT 由于暂时并未直接翻译浮点与向量，而是采用 helper 函数的方式对其进行模拟，因此在浮点与向量方面暂时不好评估，本次测试采用 CINT 来评价其翻译水平。

4.2.2 编译选项

由于 X86 glibc 中含有大量向量与浮点指令用于优化库的性能，COGBT 对这浮点与向量并未实施优化，因而采用 musl 库来作为替代。另外为了方便调试与性能分析，测试集均使用静态链接的方式，并且为了避免编译器生成 AVX 向量指令，对测试集添加了-static -mno-avx -fno-tree-vectorize 选项。

4.2.3　测试方式

首先使用 COGBT 对所有测试集进行静态翻译与优化，生成对应的 AOT 预翻译文件。之后配置好实验环境下的 binfmt，使得内核能根据执行的文件类型自动调用 COGBT 翻译系统。测试时先将实验机器下无关进程关闭，保证实验机的 CPU 处于空闲负载下，之后先使用 QEMU 对以上测试集进行运行，收集分数，再使用同样的方式使用 COGBT 运行测试集并收集分数。

4.2.4　测试结果

对 CoreMark 的测试结果如表 4.1 。相较 QEMU 的得分，COGBT 能提升 92.8%，具有明显收益。SPEC 2000 CINT 的测试结果如表 4.2 。在该测试集中，COGBT 的性能提升比例最多可达 302.4%，最低也能达到 20.9%，平均能达 80.9% 的提升。

表 4.1 CoreMark 性能测试结果

测试程序	QEMU 性能得分	COGBT 性能得分	提升比例
CoreMark	3436	6624	92.8%

表 4.2 SPEC 2000 CINT 性能测试结果

测试程序	QEMU 性能得分	COGBT 性能得分	提升比例
164.gzip	418.3	700.61	67.5%
175.vpr	459.87	561.68	22.1%
176.gcc	384.77	578.01	50.2%
181.mcf	1277.09	2212.93	73.3%
186.crafty	382.18	858.95	124.8%
197.parser	207.48	834.82	302.4%
252.eon	179.79	217.32	20.9%
253.perbmk	316.54	596.57	88.5%
254.gap	432.12	783.17	81.2%
255.vortex	348.34	822.55	136.1%
256.bzip2	500.76	941.19	88.0%
300.twolf	785.33	1096.01	36.9%
geomean	413.86	748.84	80.9%

同时本次测试也对比了 QEMU 与 COGBT 相对本地执行的性能，结果如图 4.2 。COGBT 平均能得到本地 29.9% 的性能，相对 QEMU 的 16.5% 提高了 13.4% 的绝对性能。部分测试程序如 164.gzip，181.mcf 能达到本地 55% 以上，具有明显的优化收益。但是在一些测试样例如 252.eon 中，COGBT 的翻译效率仍然较

低，只有 6%。主要原因在于 252.eon 是一个 C++ 程序，含有较多的间接跳转指令 (虚函数等)，该指令的翻译需要通过 helper 函数查找目标基本块的入口，因此效率较低；另外该测试样例中含有较多的 SSE 向量指令，而这类指令目前尚未优化，故而优化提升不明显。

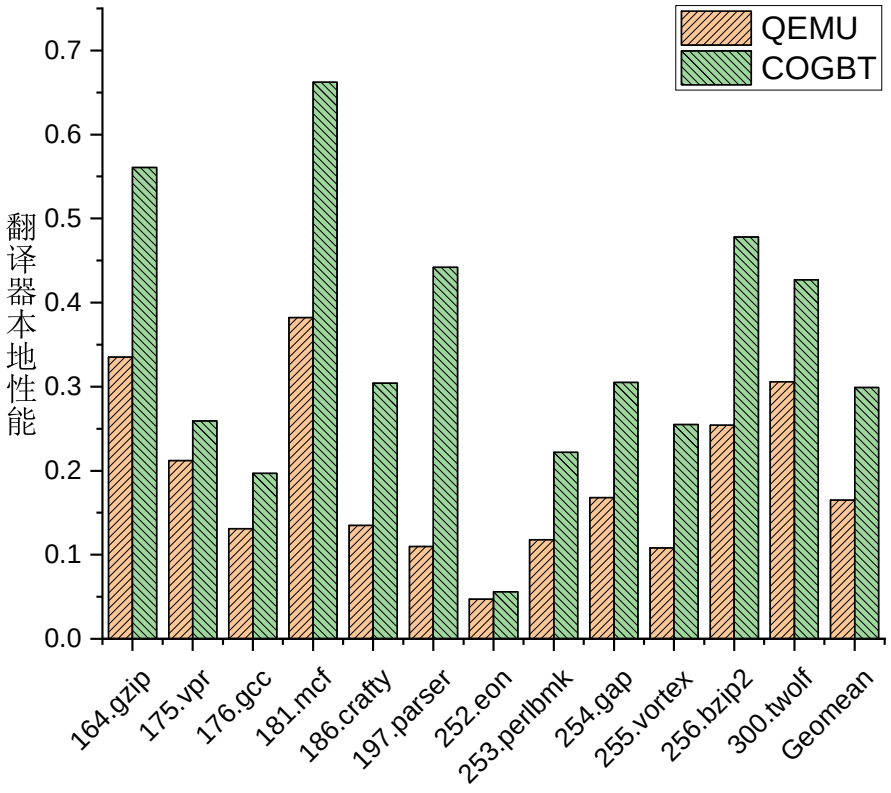


图 4.2 QEMU 与 COGBT 本地绝对性能

表 4.3 统计了 QEMU 与 COGBT 在运行 SPEC 2000 CINT 时 helper 调用指令所占动态指令比例。虽然两者动态指令数不一致，helper 调用比例也互有差异，但是仍然可以看出几点规律:

1. helper 调用比例与程序的翻译性能具有较明显的反相关性。在一些翻译性能提升较低的程序中，如 252.eon(提升 20.9%)、175.vpr(提升 22.1%)，这些程序的 helper 调用比例都相较其他程序更高，说明浮点或者向量有可能成为这些程序的性能瓶颈，因此对浮点向量等指令实施翻译可以作为下一步提升性能的方案。

2. COGBT 的 helper 调用比例总体比 QEMU 低。在大部分程序中，COGBT 的动态 helper 执行比例都比 QEMU 更低，只有少量程序如 175.vpr、176.gcc、

197.parser 与 300.twolf 略有升高[1]。由于调用一次 helper 的开销巨大，更少的 helper 调用的次数也许是 COGBT 性能更高的原因之一。

表 4.3 SPEC 2000 CINT 动态 helper 函数调用指令占比

测试程序	QEMU 比例	COGBT 比例	测试程序	QEMU 比例	COGBT 比例
164.gzip	0.24%	0.16%	252.eon	4.31%	3.87%
175.vpr	1.52%	1.75%	253.perbmk	1.94%	1.01%
176.gcc	0.84%	1.56%	254.gap	1.27%	1.27%
181.mcf	0.64%	0.03%	255.vortex	1.15%	0.77%
186.crafty	1.16%	0.73%	256.bzip2	0.51%	0.44%
197.parser	0.81%	0.93%	300.twolf	0.84%	1.02%

为验证本系统开发优化 pass 的便利性与有效性。本实验对比了冗余 LBT 消除优化使能前后翻译系统的效率，相关结果如表4.4所示：

表 4.4 冗余 LBT 消除优化对性能的影响

测试程序	优化开启前性能得分	优化开启后性能得分	提升比例
164.gzip	644.23	700.61	8.75%
175.vpr	558.71	561.68	0.53%
176.gcc	552.6	578.01	4.60%
181.mcf	2055.18	2212.93	7.68%
186.crafty	762.26	858.95	12.68%
197.parser	793.6	834.82	5.19%
252.eon	216.73	217.32	0.27%
253.perbmk	576.05	596.57	3.56%
254.gap	696.08	783.17	12.51%
255.vortex	761.99	822.55	7.95%
256.bzip2	812.84	941.19	15.79%
300.twolf	955.05	1096.01	14.76%
geomean	695.05	748.84	7.74%

可以看到相对优化开启前，冗余 LBT 消除优化能平均提高 7.74% 的翻译性能，效果显著。利用 LLVM 的优化框架，本优化的开发只需不到 400 行的代码量，并且可以与其他优化互相隔离，方便调试与扩展。这一结果充分表现了 COGBT 的高效性与可扩展性，可以为后续进一步优化提供平台支持。需要提及的是，由于部分优化与指令翻译紧密相连 (如寄存器映射、基本块链接等)，单独测试其优化效果较难实现，本部分暂未测试这类优化的单独贡献。

[1]其实绝对 helper 数量并没有更多，只是 COGBT 动态指令数更少，导致比例看起来更高

4.3 动态指令数测试

为分析 COGBT 取得相对 QEMU 80.9% 性能提升的原因。首先对两者的动态指令数做对比。以 QEMU 运行 SPEC 2000 的动态指令数为基准，COGBT 相对 QEMU 的动态指令比例如图 4.3 。平均来看，COGBT 能降低约 38% 的动态指令数。这主要得益于 LLVM 的通用优化 pass 以及自定义优化 pass 对冗余指令的消除。LLVM 的诸多优化 pass 如窥孔优化、公共子表达式消除优化、活性分析优化、死代码消除优化等都能实施跨指令的分析，对降低翻译程序的动态指令数都能起到非常好的效果；冗余 LBT 消除 pass 也能将多余的 LBT 指令删除，对部分测试程序具有明显效果。

但是 COGBT 在一些测试程序中并未明显降低动态指令数，甚至出现指令数增加的情形。如 176.gcc 相对 QEMU 增加了约 5% 的动态指令数，252.eon 相对 QEMU 的动态指令数基本无变化。但两者整体性能相对 QEMU 分别提高了 50.2% 与 20.9%。说明即使 COGBT 的翻译代码动态指令数较多，但是翻译质量应该显著高于 QEMU。翻译质量集中在两方面，一方面在于 COGBT 的访存指令数更少，总的执行效率更高。另一方面可能在于 COGBT 的指令调度更能利用硬件架构的并行性，取得更优的结果。为验证这一猜想，下一节将从翻译代码中访存指令占比这一角度来评价各种优化对翻译代码质量的影响。

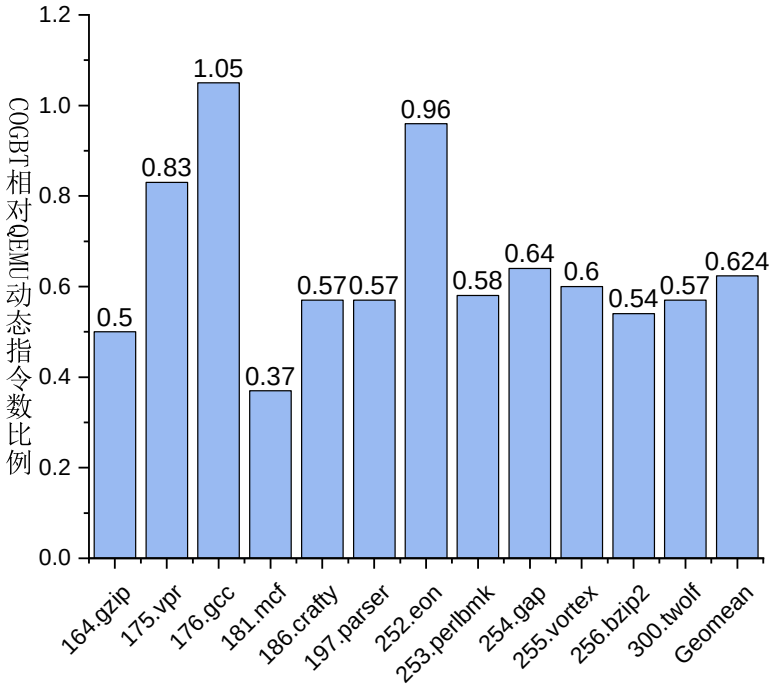


图 4.3 COGBT 相对 QEMU 的动态指令数

4.4　访存指令占比测试

COGBT 的寄存器映射优化、mem2reg 栈提升优化等都能降低翻译代码的访存指令数。这也是提升翻译代码质量的关键指标。为评估 COGBT 对访存的优化效果如何，本次测试统计了 COGBT 与 QEMU 动态运行 SPEC 2000 CINT 时翻译代码中的访存指令占比。得到的结果如图 4.4 。

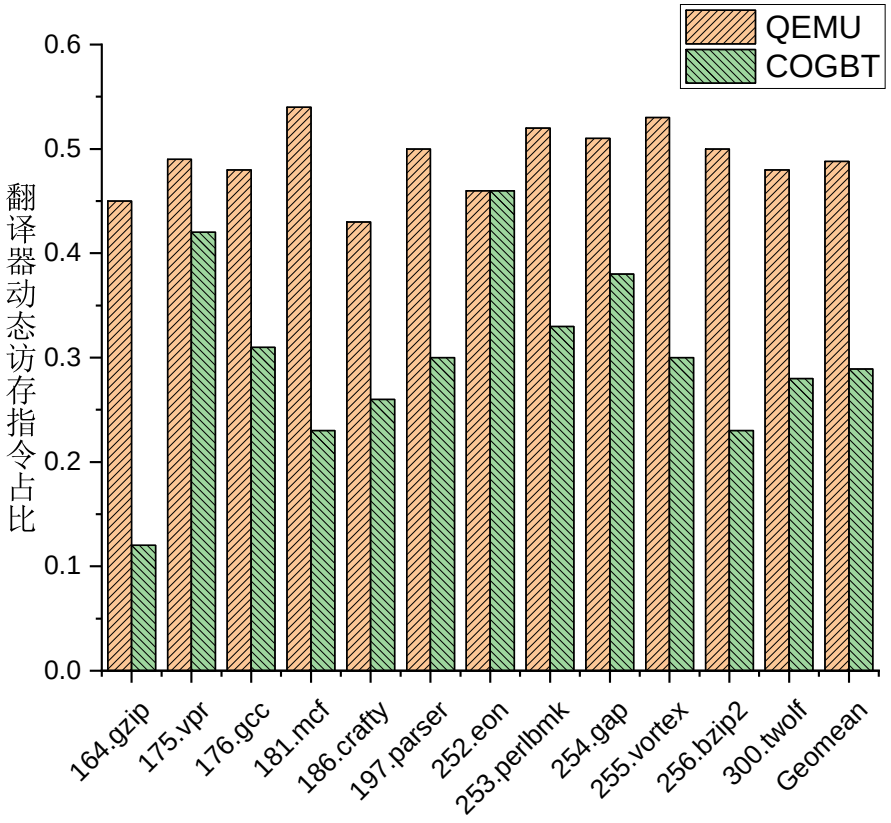


图 4.4 COGBT 与 QEMU 的动态访存指令占比

可以看到 COGBT 相对 QEMU 显著降低了访存指令的比例。平均来看，COGBT 能将 SPEC 2000 CINT 的访存指令比例从 49% 降低到 29%，显著提高了翻译代码的质量。这也解释了为什么 176.gcc 动态指令数虽然变多，但是性能却比 QEMU 要高这一反常现象。虽然 QEMU 的动态指令数更少，但是其包含更多的访存指令，而访存的开销比寻常算术逻辑指令要大，因此总体上 COGBT 效果能更好。另外需要提及的一点是 252.eon 在访存占比上与 QEMU 相差也不大，基本一致，而动态指令数与 QEMU 也基本接近。该测试程序能提高 20.9% 的一大原因可能来源于编译器的指令调度的优化。QEMU 采用逐指令翻译的方式进行，翻译代码的布局基本上与 X86 的指令布局接近，但是翻译代码却没有针对宿主机的硬

件特性做特定性的调度优化，因此无法充分利用宿主机的硬件参数来进行优化调整。而 COGBT 由于采用了 LLVM 的优化与代码生成框架，因此能将生成指令调度成适合宿主机硬件的序列，从而达到更好的优化效果。

4.5 指令翻译膨胀测试

在翻译效率方面，本系统相较 QEMU 已经取得了 80.9% 的提升，但是相较原生程序依然有不小的差距。为了进一步探索后续优化方向，评估当前翻译的不足，本实验测试了 COGBT 与 QEMU 的指令翻译膨胀[2]。通过分析 X86 指令翻译后的 LoongArch 指令数目，可以辅助分析翻译过程存在的冗余，供后续优化提供参考。相关结果如图 4.5 所示。

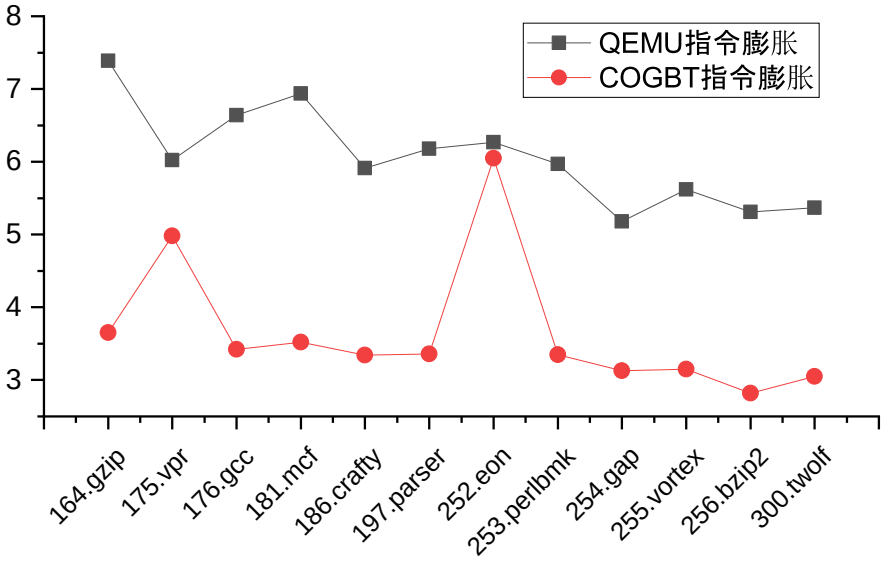


图 4.5 COGBT 与 QEMU 的指令翻译膨胀

可以看到 COGBT 的指令翻译膨胀集中在 3-4 附近，而 QEMU 的指令翻译膨胀则集中在 6 附近。说明 COGBT 能实际降低膨胀率，提高翻译性能。在测试结果中，几个翻译效率较低的程序如 252.eon 以及 175.vpr 指令膨胀较高，而这两个程序的浮点与向量指令较多，说明下一步应该重点处理浮点与向量类的翻译；同时，目前的膨胀率依然有优化空间，后续可以继续分析翻译过程中存在的冗余，编写优化 pass 将其消除。

[2]指单条客户指令翻译后的宿主机指令数

4.6　本章小结

本章对 COGBT 进行性能测试与分析。第一节测试了 COGBT 的静态代码挖掘方案，测试表明基于 profile 的代码挖掘机制能覆盖 SPEC 95% 以上的代码，具有一定的优越性。第二节对 COGBT 的性能进行了测试，测试结果显示 COGBT 在 CoreMark 与 SPEC 2000 CINT 上分别能取得 92.8% 与 80.9% 的性能提升，明显优于 QEMU。第三节测试了 COGBT 与 QEMU 的动态指令数，结果显示 COGBT 相比 QEMU 能降低约 38% 的动态指令，反应了 COGBT 跨指令分析优化的能力。第四节测试了 COGBT 与 QEMU 的动态访存比例，结果表明 COGBT 能将访存比例从 49% 降低到 29%，显著提高了翻译代码质量。第五节测试了 COGBT 与 QEMU 的指令翻译膨胀，结果表明 COGBT 能将翻译膨胀从 6 降低到 3 附近，具有显著的效果，同时也为下一步优化的开展提供了方向。

第 5 章　总结与展望

5.1　总结

本文分析了二进制翻译实现跨架构软件兼容面临的挑战：翻译效率、可靠性、可维护性，并提出了基于 LLVM 编译优化制导的动静结合方案—COGBT。已有的二进制翻译器在效率、可靠性与可维护性方面难以同时满足要求，使用受到一定限制。如 QEMU 虽然可靠，且维护难度适中，但是翻译效率较低，无法满足用户需求；REV.NG 虽然效率尚可，但是静态翻译难以覆盖所有情形，可靠性无法得到保证；Rosetta 虽然高效且可靠，但是没有引入 IR，使用场景仅限于 X86 到 ARM 的翻译，新增架构支持的维护难度较大。相比于以上翻译器的不足，COGBT 能利用动静结合的方式对翻译代码进行深度优化，能达到较高的效率。同时由于 COGBT 配合了动态端做补充，因此也能覆盖所有代码，具有较高的可靠性。再者，COGBT 使用 LLVM IR 作为中间表示，新增前后端架构都较为简单，具有较强的维护性。

本文以 X86 与 LoongArch 分别作为 COGBT 的前后端，实现了一个从 X86 到 LoongArch 的二进制翻译系统。该系统利用 LLVM IR 的强大表达能力实现了大部分定点的翻译，利用 LLVM 丰富的优化分析库，配合自定义的优化方案提升了翻译的效率。实验结果表明，COGBT 在 SPEC 2000 CINT 测试集上性能比 QEMU 高 80.9%，在 CoreMark 测试集上比 QEMU 高 92.8%。更重要的一点，COGBT 作为一个编译优化的平台性系统，可为后续开发其他优化 pass 提供基础支持。

5.2　展望

尽管 COGBT 已经取得了一定的效率，但是目前 COGBT 仍然存在以下几个方面的问题：

1. 离线编译时间过长。在 AOT 生成时，需要离线编译所有翻译的 IR，当基本块过多时，翻译生成的 LLVM 函数将非常多，编译有较大的时间开销。实测在 2.9GHz 的 i7-10700 下，编译 4500 个基本块需要花费 103 秒。而很多程序动辄上万个基本块，编译时间更长。后续可以考虑以函数为基本单位来执行翻译，

通过将多个基本块组成一个函数翻译单元减少 LLVM 生成的函数数，进而减少 pass 的执行次数。

2. 暂不支持浮点向量的翻译。目前 COGBT 借助 QEMU 的辅助函数来翻译浮点向量，会导致浮点向量密集的程序运行效率较低，如 SPEC 2000 CINT 中的 252.eon 目前只有 20.9% 的性能提升，本地性能更是只有 6%，值得进一步改进。

3. 代码挖掘可以更细致。目前 COGBT 的代码挖掘需要先运行一次动态端进行采样，挖掘出的代码很大程度上取决于采样的覆盖率，这在实际应用中将难以实施。可以考虑借鉴一下其他静态程序分析技术，通过静态分析结合一些启发式策略提高代码挖掘覆盖率。

4. 精确异常尚未支持。COGBT 以 LLVM 作为代码生成的后端，在代码生成过程中会进行指令调度与优化，导致生成的翻译指令无法与原 X86 指令一一对应，进而无法保证精确异常。后面可以考虑从硬件与软件两方面尝试解决。硬件上可以参考 Crusoe[22] 的机制设置影子寄存器 (shadow registers) 与写缓存 (store buffer)，通过显示的提交指令更新真实的机器状态，通过回滚指令撤销修改；软件上也可以在翻译代码中设置一些检测点 (checkpoint)，在检测点进行全局状态的更新，发生异常时则回滚至上一个检测点。

在性能优化方面 COGBT 依然有大量优化工作可以实施：

1. 优化 LLVM LoongArch 后端的指令匹配。LoongArch 指令集由于推出时间较短，在编译器后端适配方面依然有较大提升空间。在目前的观察中，一些简单 IR 匹配的指令尚有较大改进空间，如使用 bstrpick 匹配 LLVM 的 and 操作等。

2. 切换翻译单元为函数或 trace。当翻译单元更大时，编译器能够进行分析优化的范围就更广。能够进行更大粒度的活性分析、消除更多的冗余标志位、发现更多公共子表达式、进行更大范围的窥孔优化等。

3. 加入 PGO 等反馈式优化机制。COGBT 的静态端能离线进行优化，可以离线实施复杂优化。客户程序大多通过静态编译形成，缺乏运行时信息，通过对客户程序进行 profile 采样可以获取程序的运行时行为，对偏好路径做针对性优化。如调整热路径上翻译基本块的布局提升 cache 局部性、内联热路径上的函数、多级预测热路径上的间接跳转等。

4. 实施 X86 栈上访存优化。LoongArch 指令集相比 X86 有更多的寄存器，因此可以存放更多的临时变量。在高级语言中，临时变量先被编译器放到寄存器

中，寄存器不足时会溢出到栈上，对这类变量的访问需要通过访存进行。因此可以利用 LoongArch 的寄存器优势识别 X86 函数中的栈变量，将其建模为 LLVM IR 的栈变量，通过 LLVM 的栈提升优化实施更合适的 LoongArch 寄存器分配。

参考文献

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

[2] Venkat A, Tullsen D M. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor [J]. ACM SIGARCH Computer Architecture News, 2014, 42(3): 121-132.

[3] [高翔张戈]. 龙芯指令系统架构及其软件生态建设 [J]. 信息通信技术与政策, 2022, 48(4): 43-48.

[4] Altman E R, Kaeli D, Sheffer Y. Welcome to the opportunities of binary translation [J]. Computer, 2000, 33(3): 40-45.

[5] Sites R L, Chernoff A, Kirk M B, et al. Binary translation [J]. Communications of the ACM, 1993, 36(2): 69-81.

[6] Smith J E, Nair R. Chapter two - emulation: Interpretation and binary translation [M/OL]// Smith J E, Nair R. The Morgan Kaufmann Series in Computer Architecture and Design: Virtual Machines. Burlington: Morgan Kaufmann, 2005: 27-82. https://www.sciencedirect.com/science/article/pii/B9781558609105500033. DOI: https://doi.org/10.1016/B978-155860910-5/50003-3.

[7] Cifuentes C, Malhotra V M. Binary translation: Static, dynamic, retargetable? [C]//icsm: volume 96. 1996: 340-349.

[8] Probst M. Dynamic binary translation [C]//UKUUG Linux Developer's Conference: volume 2002. 2002.

[9] Apple. Rosetta 2 on a mac with apple silicon [EB/OL]. 2021[2023-02-18]. https://support.apple.com/en-hk/guide/security/secebb113be1/web.

[10] Dougallj. why is rosetta 2 fast [EB/OL]. 2022[2023-02-18]. https://dougallj.wordpress.com/2022/11/09/why-is-rosetta-2-fast.

[11] Nakagawa K M. Welcome to project champollion web page [EB/OL]. 2021[2023-02-18]. https://ffri.github.io/ProjectChampollion/.

[12] Bellard. Tiny code generator(tcg) [EB/OL]. 2019[2023-03-01]. https://wiki.qemu.org/Documentation/TCG.

[13] Lattner C, Adve V. Llvm: A compilation framework for lifelong program analysis & transformation [C]//International symposium on code generation and optimization, 2004. CGO 2004. IEEE, 2004: 75-86.

[14] Cytron R, Ferrante J, Rosen B K, et al. Efficiently computing static single assignment form

and the control dependence graph [J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1991, 13(4): 451-490.

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

[16] 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.

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

[18] 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.

[19] Di Federico A, Agosta G. A jump-target identification method for multi-architecture static binary translation [C]//Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. 2016: 1-10.

[20] Chernoff A, Herdeg M, Hookway R, et al. Fx! 32: A profile-directed binary translator [J]. IEEE Micro, 1998, 18(02): 56-64.

[21] 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.

[22] Klaiber A, et al. The technology behind crusoe processors [J]. Transmeta Technical Brief, 2000.

[23] Zhang X, Guo Q, Chen Y, et al. Hermes: A fast cross-isa binary translator with post-optimization [C]//2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2015: 246-256.

[24] Wang W, McCamant S, Zhai A, et al. Enhancing cross-isa dbt through automatically learned translation rules [J]. ACM SIGPLAN Notices, 2018, 53(2): 84-97.

[25] Kim J H, Chilimbi T, Deisher M. Dynamic binary translation using hardware support for transparent indirect branch prediction [C]//Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2004: 15-26.

[26] Li W, Luo X, Zhang Y, et al. Crossdbt: An llvm-based user-level dynamic binary translation emulator [C]//Euro-Par 2022: Parallel Processing: 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22–26, 2022, Proceedings. Springer, 2022: 3-18.

[27] 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.

[28] Duesterwald E, Bala V. Software profiling for hot path prediction: Less is more [J]. ACM SIGARCH Computer Architecture News, 2000, 28(5): 202-211.

[29] Shen B Y, You J Y, Yang W, et al. An llvm-based hybrid binary translation system [C]// 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12). IEEE, 2012: 229-236.