学 校 代 码 10459
学号或申请号 201712362013841
密 级

郑州大学

硕士学位论文

基于申威平台LLVM编译器的窥孔优化研究

作 者 姓 名：胡浩
导 师 姓 名：周清雷 教授
学 科 门 类：工学
专 业 名 称：计算机科学与技术
培 养 院 系：信息工程学院
完 成 时 间：2020 年 4 月

A thesis submitted to

Zhengzhou University

for the degree of Master

Research on Peephole Optimization of LLVM Compiler Based on Sunway Platform

By Hu Hao

Supervisor: Prof. Zhou Qinglei

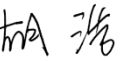
Computer Science and Technology

School of Information Engineering

April 2020

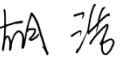
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究所取得的成果。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。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律责任由本人承担。

学位论文作者：  日期： 2020 年 06 月 09 日

学位论文使用授权声明

本人在导师的指导下完成的论文及相关的职务作品，知识产权归属郑州大学。根据郑州大学有关保留、使用学位论文的规定，同意学校保留或向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅；本人授权郑州大学可以将本学位论文的全部或部分编入有关数据库进行检索，可以采用影印、缩印或者其他复制手段保存论文和汇编学位论文。本人离校后发表、使用学位论文或与该学位论文直接相关的学术论文或成果时，第一署名单位仍然为郑州大学。保密论文在解密后应遵守此规定。

学位论文作者：  日期：2020 年 06 月 09 日

摘要

自主可控是信息安全乃至国家安全的重要保障。随着国家自主可控战略的不断推进，涌现了一大批国产化的软硬件技术，如申威处理器、龙芯处理器、深度操作系统、麒麟操作系统和 UOS 操作系统等。基于国产平台和操作系统的高效编译器的研发越来越受到人们的重视。

LLVM 是以 C++编写的架构编译器的框架系统，支持多后端和交叉编译，基于申威平台的 LLVM 编译器的研究具有重要意义。本文重点研究了申威平台 LLVM 编译器中的窥孔优化方法，对 LLVM 编译器中的窥孔优化存在过度优化和申威平台特有指令未能充分利用的问题，提出了节点融合优化方法。完成的主要工作如下：

1. 提出了节点融合优化方法。节点融合的基本思想为将多个节点优化为一个高效的融合节点，减少诸如指令、寄存器、时钟周期、访存等开销，达到减少程序运行时间、提升访存效率等目的。通过在申威平台上研究 LLVM 编译器中的窥孔优化技术，结合申威平台指令集的特点，提出了节点融合优化方法。

2. 基于申威平台进行了节点融合优化实验与分析。为了提升申威平台 LLVM 编译器的性能，在 LLVM 编译流程的中间表示阶段、DAG 合并阶段、指令选择阶段实现了节点融合优化。以申威 1621 处理器为实验平台，CLANG 和 FLANG 为编译器前端，LLVM 为编译器后端，基于 SPEC CPU2006 基准测试集进行了评估。实验结果表明，节点融合优化有利于提高编译器性能、减少程序运行时间，优化后最大加速比为 1.59，平均加速比为 1.13。且已在申威平台 LLVM 编译器中得到实际应用。

关键词：申威平台；LLVM；窥孔优化；节点融合

Abstract

Independent control is an important guarantee of information security and even national security. With the continuous promotion of national independent control strategy, a large number of domestic software and hardware technologies have emerged, such as Sunway processor, Loongson processor, Deepin operating system, Kylin operating system and UOS operating system. More and more attention has been paid to the development of efficient compiler based on domestic platform and operating system.

LLVM is a framework system of architecture compiler written by C++, which supports multi-backend and cross-compilation. It is of great significance to study the LLVM compiler based on Sunway platform. This paper focuses on the study of the peephole optimization method in the LLVM compiler of Sunway platform, and proposes the node fusion optimization method to solve the problem of over-optimization of the peephole optimization and underutilization of Sunway platform specific instructions in the LLVM compiler. The main tasks completed are as follows:

1. The node fusion optimization algorithm is proposed. The basic idea of node fusion is to optimize multiple nodes into an efficient fusion node, reducing overhead such as instructions, registers, clock cycles, memory access, etc., to reduce program runtime and improve memory access efficiency And other purposes. By studying the peephole optimization technology in LLVM compiler on Sunway platform and combining the characteristics of Sunway platform instruction set, a node fusion optimization method is proposed.

2. The node fusion optimization experiment and analysis are carried out based on Sunway platform. In order to improve the performance of the LLVM compiler, a node fusion optimization method is proposed in the intermediate representation stage, DAG merge stage, and instruction selection stage of the LLVM compilation process. Under the Sunway processor of 1621, CLANG and FLANG are used as the front-end of the compiler, LLVM is used as the compiler back-end, and the evaluation is based on the SPEC CPU2006 benchmark set. The experimental results show that node fusion

optimization is beneficial to improve the compiler performance and reduce the program running time, after optimization, the maximum acceleration ratio is 1.59, and the average acceleration ratio is 1.13. And has been practical application in LLVM compiler on Sunway platform.

Key words: Sunway platform; LLVM; Peephole optimization; Node fusion

目录

图清单

表清单

1 绪论

1.1 研究背景及意义

随着以互联网为基本架构的网络空间成为继陆、海、空、天四个疆域之后的第五个疆域，各国均重视网络空间的安全问题。保障网络空间安全、信息安全的前提是软、硬件的自主可控。自主可控简而言之就是指核心技术、关键硬件以及各类配套的软件全都国产化，自己开发、自己制造，不受制于人[1][2][3][4][5]。核心技术、关键硬件受制于人将为大国崛起带来重重阻碍，并且国外软、硬件的预装后门的事件时有发生，这将带来严重的网络信息安全隐患，只有把核心技术、关键硬件掌握在自己手中，才能从根本上保障国家经济安全、国家信息安全、国家网络安全、国防安全及其他安全[6]。众所周知，信息产业发展的核心领域是芯片，自从中兴事件[7]之后，国家充分认识到自主可控的重要性，开始大力支持发展我国自主可控的芯片研发工作。CPU(Central Processing Unit, 中央处理器)是整个芯片行业中最核心的技术，在 CPU 设计方面我国与发达国家相比还有显著差距。虽然经过多年自主创新的努力，技术差距已经越来越小，但是在民用、商业领域内仍然难以见到国产 CPU 的身影。目前民用、商业领域的 CPU 被 Intel 芯片和 Windows 操作系统所垄断，要想实现我国自主研发的替代 CPU 和操作系统，不是通过几年的自主创新就能实现的，需要不断的进行自主研发以及建立完善的软件生态系统，这样才能为国产 CPU 和操作系统注入生生不息的创新活力。

“神威•太湖之光”计算机系统是目前我国运算速度最快的超级计算机，全系统峰值运算速度为 12.54 亿亿次/秒，持续运算速度为 9.3 亿亿次/秒，性能功耗比每瓦 60.5 亿次，和其他相同量级的超级计算机相比节能 60%以上。“神威•太湖之光”是由国家并行计算机工程技术研究所在国家 863 计划支持下研制的新一代超级计算机系统，它是世界上首台峰值运算性能超过十亿亿次量级的超级计算机，也是我国第一台全部采用国产处理器构建的世界第一的超级计算机。目前开发的主要平台包括船舶设计平台，新药研发平台，动漫渲染平台，CFD（Computational Fluid Dynamics，计算流体动力学）模拟平台，数据智能综合分析平台和海洋灾害预测平台等[8]。编译器是一种软件程序，它能将开发人员以高级编程语言编写的高级源代码转换为机器能识别的低级目标代码，即二进制代

码。编译器对操作系统的重要性不言而喻，特别是国产平台的指令集、处理器以及操作系统都是我国自主设计研发，拥有自主知识产权的操作系统，对于保障国家信息安全具有十分重要的意义[9]，所以开发出针对于国产操作系统和国产操作平台的高效编译器变得极为迫切。

LLVM[10]是底层虚拟机（Low Level Virtual Machine）的简称，它是以伊利诺伊大学厄巴纳-香槟分校开源许可（University of Illinois/NCSA Open Source License）进行开源的框架编译器。虽然 LLVM 名为底层虚拟机，但是并不属于传统的虚拟机，甚至和传统虚拟机没有任何关系，它只是一个项目的全称。在 2000 年，伊利诺伊大学发起了开源项目 LLVM，其目标是提供一种现代的、基于 SSA[11]（Static Single Assign, 静态单赋值）的编译策略，能够支持任意编程语言的静态编译和动态编译。在 2005 年 Apple 公司雇佣了 Chris Lattner，LLVM 成为 Apple 公司的官方编译器，在苹果公司的鼎力支持下，LLVM 项目快速发展，扩展出了很多子项目，如 LLVM 核心库、CLANG[10]、LLDB、LIBC++和 LIBC++ ABI、compiler-rt 等。LLVM 核心库提供一个独立于高级语言和目标平台的优化器，以及支持多个目标平台的代码生成器。CLANG 是一个支持 C/C++、Objective-C 语言的轻量级编译器，可以作为 LLVM 的前端编译器[12]。LLDB 项目以 LLVM 和 CLANG 提供的库为基础，提供了一个本地调试器，在内存效率、符号加载等性能方面比 GCC 的 GDB 要高得多。LIBC++和 LIBC++ ABI 项目提供了 C++标准库的一致性和高性能实现，对 C++11，C++14 等标准都完全支持。Compiler-rt 项目不仅提供了对底层代码生成器进行高度调优的工具，而且还提供了动态检测工具的运行时库，如 AddressSanitizer、ThreadSanitizer、MemorySanitizer 等检测库。

LLVM 编译器可以优化任意程序语言编写的程序的编译时间、链接时间、运行时间和空闲时间[13]。通过目标无关的优化工具 OPT，LLVM 可以提供多种目标无关的优化，比如死代码删除[14]、循环向量化[15]、窥孔优化[16][17][18][19]、常量传播[20]和公共子表达式消除[21]等。窥孔优化是一种局部的优化方法，编译器在一个或多个基本块中，针对已经生成的代码，结合目标平台 CPU 的指令特点，进行冗余指令删除、控制流优化、强度削弱、特有指令替换等操作进行程序的优化。窥孔优化虽然是一种局部优化方法，但是可能将带来较大的性能提升，如对循环代码进行窥孔优化，一点点优化将导致优化效果成倍增长。由于国产平台使用的芯片都是自主设计，所以其指令集中存在较多的特有指令。为了充分利用国

产平台提供的特有指令、进一步挖掘国产指令集的潜力、提升 LLVM 编译器国产后端的编译和优化效率，需要对 LLVM 的窥孔优化方法进行进一步研究。开发和完善申威平台上的 LLVM 编译器，在增加我国自主可控能力、建设软件生态等方面都具有重要意义。

本文在申威平台下基于 LLVM 编译器对窥孔优化进行了研究，在中间表示阶段，DAG（Direct Acyclic Graph，有向无环图）合并阶段、指令选择阶段提出节点融合优化方法，使用 SPEC CPU2006[22]基准测试集进行了实验。实验结果表明，节点融合优化有利于提高编译器性能、减少程序的运行时间，优化后最大加速比为 1.59，平均加速比为 1.13。

1.2 国内外研究现状

窥孔优化方法采用一个类似于滑动窗口的技术对窗口内的代码进行分析、优化，容易实现，在多个软件中均有应用。

针对动态二进制翻译受动态执行限制无法进行深度优化导致效率低下，传统的静态二进制翻译难以处理间接分支，并且现有的优化方法集中在中间代码层，对目标码中存在的大量冗余指令关注较少这一现状，谭捷等[23]提出了一种静态二进制翻译框架 SQEMU，基于 SQEMU 框架提出了一种针对目标码冗余指令进行删除的优化算法。该算法利用目标码生成的指令特定数据依赖图和窥孔优化方法，对目标码中的冗余指令进行删除，利用该算法对目标码进行优化后，其执行效率最大可提升 42%。葛吴超等[24]针对嵌入式系统软件开发过程中 GCC 编译器循环程序时的窥孔优化欠缺导致编译出的代码在性能上不如 ARM 商业编译器的问题，提出了针对于 ARM9 处理器的循环计数值组合、循环处理数据合并和循环最优展开等 3 种窥孔优化方法对汇编代码进行优化，实验结果表明优化的代码在寄存器使用上平均节省了 50%，性能提升近 2 倍。刘颂超[25]针对传统的窥孔优化方法中的优化窗口大小固定、对符合优化条件但非连续排列的指令序列无法有效识别等问题，提出了一种构造汇编器的方法对传统的窥孔优化方法进行了改进。该方法利用正则表达式的匹配能力，能够准确地匹配符合要求但非连续存放的指令序列，替换为代码执行效率更高的、长度更短的指令序列。实现结果表明该优化方法能有效减少目标代码的大小和运行时间，提升程序执行效率。

Lee 等[26]发现用于验证 LLVM 的窥孔优化的工具 Alive[27]本身并没有经过验证，这将导致至少有一次声明了未通过正确性验证的优化。为了解决这个问题提出了针对 LLVM 的经过正式验证的窥孔优化验证器 AliveInLean。在假设正确的证明义务由可满足性理论求解器承担的条件下，AliveInLean 提供和 CompCert[28]，Peek[29]，Vellvm[30]等先进的正式框架同等级的正确性保证，同时还继承了 Alive 工具的易于使用和自动化等特点。Van Oirschot J[31]分析和研究了不同的树模式匹配方法并且进行了扩展，用以满足窥孔优化引入的约束条件。通过实现一个动态代码生成器，利用扩展的树模式匹配器来提高 LLVM 的窥孔优化器的编译速度。实验结果表明，在给定一组有限的窥孔优化方法的条件下，扩展的树模式匹配方法与 LLVM 默认的匹配策略相比具有更快的编译时间同时不会牺牲可执行码的大小和执行时间。

1.3 本文内容与结构安排

本文从自主可控引出国产平台、国产软件对保障国家信息安全的重要性，编译器又是国产软件中的重中之重。编译器是编译其他软件的软件，编译器的质量对保障软件的质量具有重要影响。本文研究和分析了 LLVM 编译器中的窥孔优化，结合申威平台的特点提出了节点融合优化方法，使用 SPEC CPU2006 基准测试集进行测试，实验结果表明节点融合优化方法具有可观的加速效果。本文分为六个章节，各章节的安排如下：

第一章：绪论。阐述了课题的研究背景及意义，介绍了核心技术自主可控的重要性和申威平台的性能及应用场景；介绍了 LLVM 编译器的相关技术及发展历史，从申威平台自主设计的角度讨论了如何进一步开发申威平台的潜力，从而提出研究窥孔优化技术的课题；介绍了窥孔优化的国内外研究现状和研究成果；列出了本文各章节内容和结构安排。

第二章：相关背景知识介绍。简要介绍了申威平台的高性能多线程处理器 26010、高性能单核处理器 111 和高性能单核处理器 1621；介绍了编译器的入门知识、LLVM 编译器框架、LLVM IR（Intermediate Representation，中间表示）和中间表示的降级过程；介绍了窥孔优化技术的基本概念和方法；介绍了论文测试所使用的基准测试集。

第三章：LLVM 中的窥孔优化研究。主要介绍了常见的窥孔优化方法：冗余

指令消除、控制流优化、代数简化和特有指令替换，给出了优化前后的 LLVM IR 代码。

第四章：节点融合优化方法。主要介绍了节点融合优化方法的原理和代价评估的计算过程；在中间表示阶段、DAG 合并阶段、指令选择阶段进行了节点融合的实现；对中间表示阶段、DAG 合并阶段指令选择阶段的节点融合优化进行了优缺点比较。

第五章：实验与分析。对实验环境、测试工具和测试方法进行了概述；对实验后的结果进行了分析，对加速效果差的程序进行了进一步的探讨和分析。

第六章：总结和展望。对本文的研究内容进行简要概述及总结，同时指出本文研究中的不足，提出了下一步的研究方向。

2 相关背景知识介绍

国产化的处理器及其相关的软硬件技术对于增强自主可控能力和保障国家信息安全具有十分重要的意义。编译器是将高级语言源代码翻译为二进制代码的软件，其重要性不言而喻，可以说没有编译器将只能使用汇编和二进制等低级语言来编写程序，极大影响了开发效率、软件质量和项目管理。本章将介绍国产平台的基本情况、LLVM 编译器的基础知识、中间表示的降级流程、窥孔优化的基本知识和实验中使用的基准测试集。

2.1 申威平台介绍

Alpha 体系最具代表性的 CPU 芯片非申威系列处理器莫属。申威平台对自主的 Alpha 架构不断升级优化，在双核 Alpha 基础上拓展了多核架构和 SIMD 等特有指令集。申威 CPU 主要面向高性能的计算服务领域，在 2016 年国际超算大会上，基于申威 26010 处理器的“神威•太湖之光”超级计算机系统以峰值计算能力 12.54 亿亿次/秒称为世界上首台峰值运算性能超过十亿亿次量级的超级计算机，也是我国第一台全部采用国产处理器构建的世界第一的超级计算机[32]。申威平台的处理器有三种主要的型号：一是高性能多线程处理器有申威 26010；二是高性能单核处理器申威 111；三是高性能多核处理器申威 1621。本文实验部分使用的处理器为申威 1621 处理器，以下将对这三种主要型号进行介绍。

2.1.1 高性能多线程处理器

申威处理器系列中的高性能多线程处理器有申威 26010。该处理器是由国家高性能集成电路设计中心采用自主核心技术成功研制的，主要特点有：（1）面向构建十亿亿次超级计算系统；（2）采用片上融合的异构众核架构;（3）拥有自主知识产权的申威指令集（SW-64）；（4）集成 4 个运算控制核心和 256 个运算核心；（5）核心根据需求扩展了 256 位向量指令集。我国目前最快的超级计算机系统“神威•太湖之光”使用的就是申威 26010 处理器 [33]。申威 26010 处理器的片内结构如图 2.1 所示。

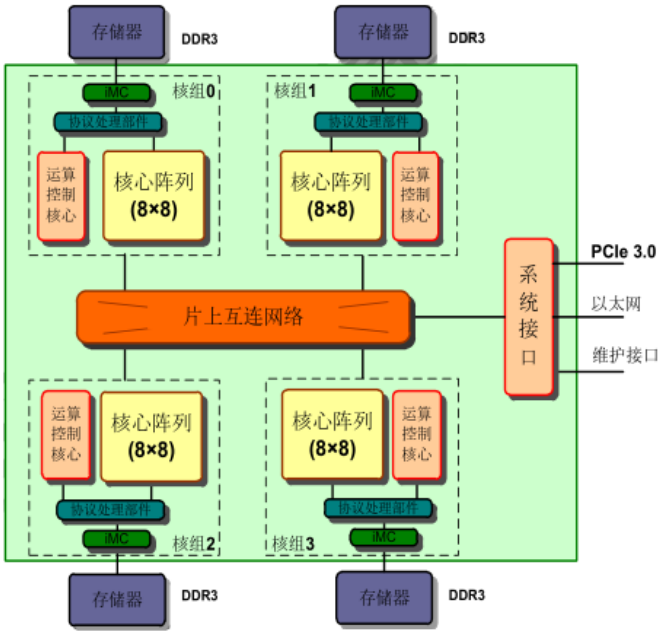


图 2.1 申威 26010 处理器片内结构

申威 26010 处理器采用片上融合的异构体系结构，由四个核组构成，编号分别为核组 0、核组 1、核组 2 和核组 3。每个核组都有一个运算控制核心（主核）和一个 8×8 核心阵列（64 个从核），核组之间通过片上互连网络进行连接，整个申威 26010 芯片有 260 个核心，核心频率为 1.45GHz。芯片还提供了系统接口 PCIe 3.0、以太网及维护接口，方便数据传输。

2.1.2 高性能单核处理器

申威高性能单核处理器有申威 111。申威 111 是一款国产嵌入式处理器，主要面向的应用领域为高密度计算型嵌入式。申威 111 处理器的特点为：（1）流水线采用 4 译码 7 发射结构，使单核性能有大幅提升；（2）为实现更低功耗，采用低功耗的流片工艺，在典型程序下芯片的整体运行功耗为保持在 3 瓦以内；（3）为适应多种不同的应用场景，集成了丰富的 I/O 外设接口；（4）产品的性能和质量已达到国军标 B 级标准，可以应用在军工、工控等领域[34]。申威 111 片内结构图如图 2.2 所示。

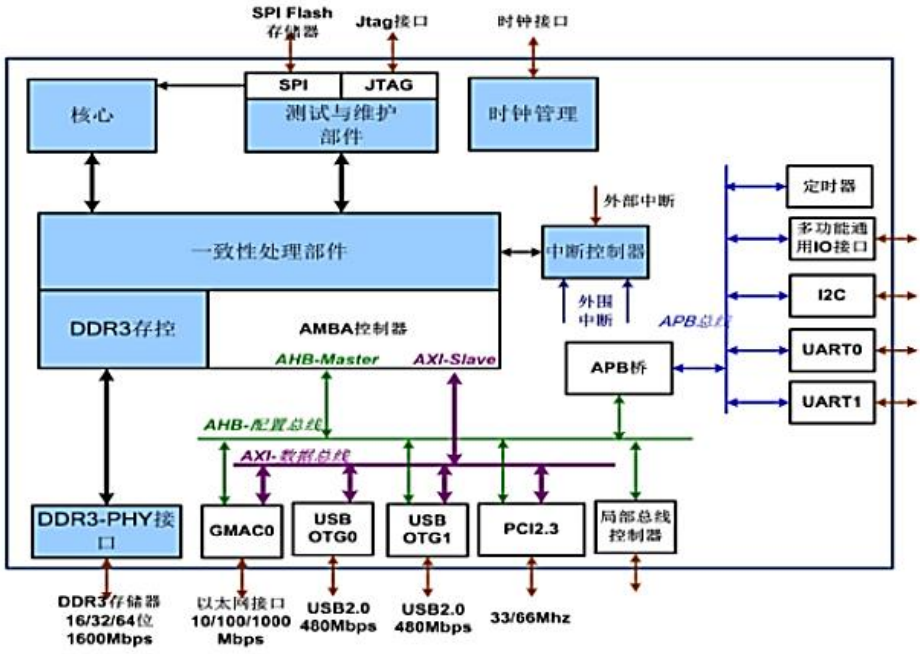


图 2.2 申威 111 片内结构

申威 111 处理器采用片上系统（System on a Chip，SoC）技术，集成了 1 个 64 位精简指令集（Reduced Instruction Set Computing，RISC）结构的申威处理器核心，工作频率在 800MHz 到 1.0GHz 之间。集成单路第三代内存规格（Double Data Rate Three，DDR3）存储控制器，还集成了 PCI、Ethernet（以太网）、USB、I2C、LBC、UART 等标准 I/O 接口，这些 I/O 接口可以根据系统应用需求配置成不同模式。

2.1.3 高性能多核处理器

申威的高性能多核处理器有申威 221、申威 411、申威 421、申威 421M 和申威 1621 等多个型号。限于篇幅，以下将仅对本文实验所使用的申威 1621 处理器进行介绍。

申威 1621 处理器是一款国产高性能 16 核处理器，基于第三代申威 64 位核心技术，主要应用在中高端服务器和高性能计算应用。目前，该处理器已经实现量产。申威 1621 处理器采用对称多核结构和 SoC 技术，集成了 16 个主频为 2GHz、64 位 RISC 结构的申威处理器核心，还集成了两路 PCI-E 3.0 标准 I/O 接口和八路 DDR3 存储控制器。申威 1621 处理器的主要特点有：（1）采用 28nm

代工工艺，设计工作频率 2GHz；（2）16 个核心以连贯缓冲非统一内存寻址(Cache-Coherent Non Uniform Memory Access ，CC-NUMA)方式共享三级共享 Cache，并且三级 Cache 容量为 32MB；（3）在主频 2GHz 工作时的双精度浮点性能可高达每秒 512G 次；（4）集成八路 64 位 DDR3 存储控制器，传输率在 1066 到 2133MBps 之间，支持错误检查和纠正（Error Correcting Code，ECC）校验，内存可扩展至 256GB；（5）集成两路 PCI-E3.0 接口，双向聚合有效带宽提高到 32Gbps[35]。申威 1621 处理器的片内结构如图 2.3 所示。

图 2.3 中有六个环网节点编号分别为#0、#1、#2、#3、#4 和#5，负责核心、Cache 和 PCIE I/O 接口之间的通信和数据交换。八个 MC（Memory Controller，内存控制器）对应八路 64 位 DDR3 控制器，两个 PCIE 对应两路 PCI-E3.0 接口。申威 1621 处理器的核心结构如图 2.4 所示。

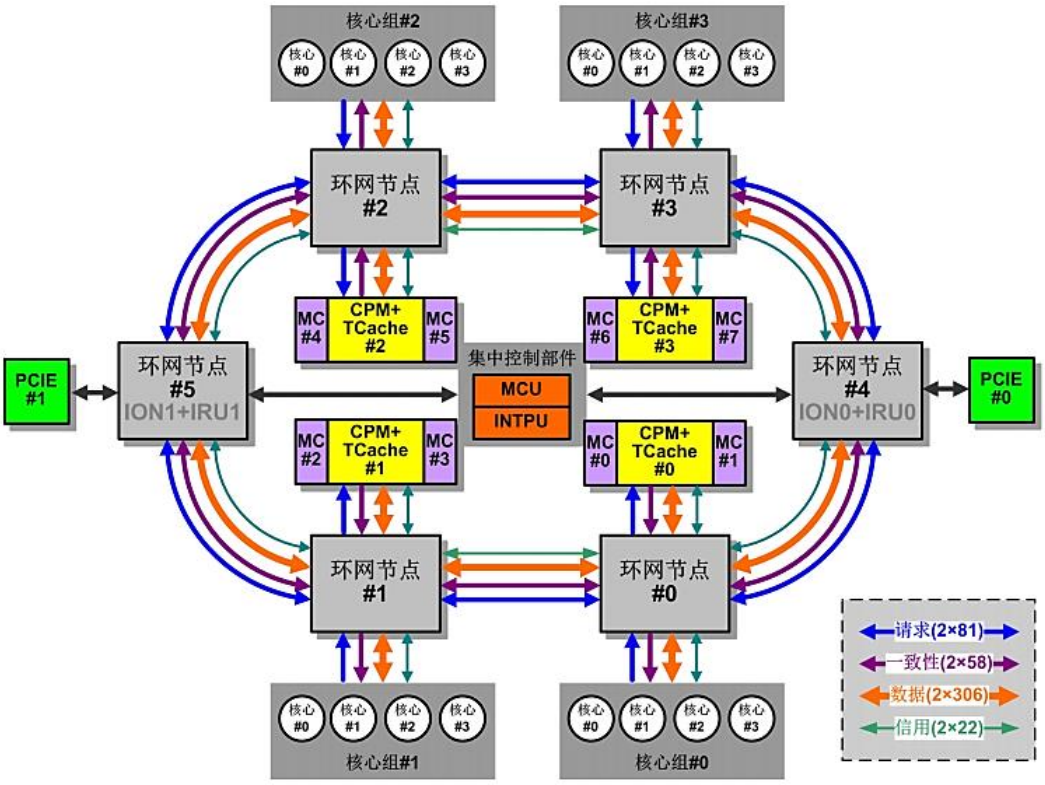


图 2.3 申威 1621 片内结构

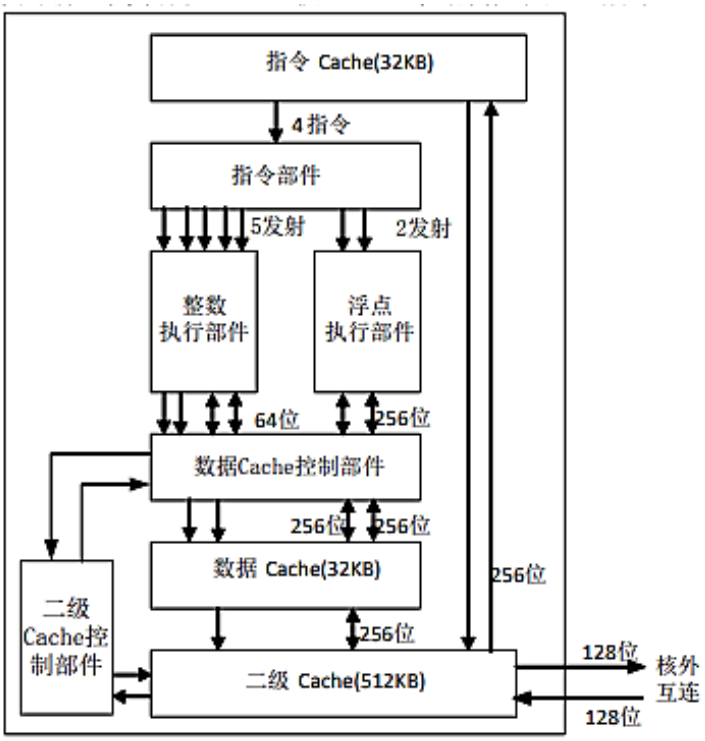


图 2.4 申威 1621 处理器核心结构

图 2.4 中，申威 1621 处理器核心由指令 Cache、指令部件、整数执行部件、浮点执行部件、数据 Cache 控制部件、二级 Cache 控制部件、数据 Cache 以及二级 Cache 组成。核心采用 4 译码 7 发射结构，支持并行发射、乱序发射、乱序执行和推测执行技术，支持浮点双路 256 位向量流水线，数据 Cache 为 32KB，指令 Cache 为 32KB，二级 Cache 为 512KB，硬件自动支持指令与数据的 Cache 一致性[36]。

2.2 LLVM 编译器介绍

2.2.1 编译器预备知识

编译器是一种软件程序，它能将开发人员以高级编程语言编写的高级源程序转换为机器能识别的低级目标程序，即二进制代码，中间可能还会输出一些错误或警告信息。将高级语言转换为机器语言的过程称为编译，将机器语言转换为高级自然语言的过程称为反编译[37]。编译器的编译过程如图 2.5 所示。

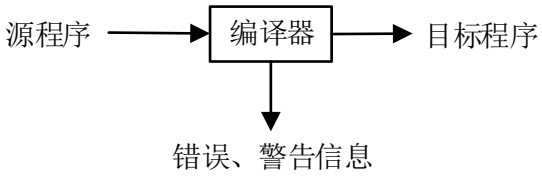


图 2.5 编译器编译过程

目前，世界上存在着几千种编程语言，诸如 Go，Kotlin 等新编程语言在不断的涌现。在这些编程语言中既有 Fortran 和 Pascal 这样的传统的程序设计语言，也有各种计算机应用领域中出现的专用编程语言。不同的编程的语言需要不同的编译器，从表面来看编译器的种类似乎千变万化，但是从实质上看任何编译器所要完成的基本任务都是相同的，如图 2.6 所示。

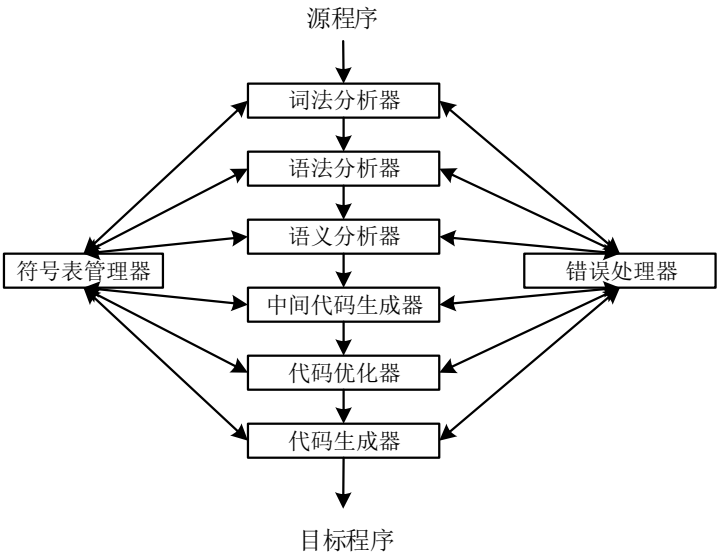


图 2.6 编译处理流程

图 2.6 主要分为以下几个步骤：

1) 词法分析器。词法分析器被成为线性分析器或者扫描器，它的输入为高级编程语言编写的程序即字符流，词法分析器对字符流进行扫描，将字符流变为符号流，通过符号表管理器进行记录，比如运算符、变量名等都是符号。

2) 语法分析器。语法分析器的输入是词法分析器的输出即符号流，语法分析的任务是在词法分析的基础上结合符号表管理器中的符号信息将符号流组合成各类语法短语，比如语句、表达式等。语法分析器能够检测出语

法错误，判断源程序在结构上是否正确，通过错误处理器进行处理。

3) 语义分析器。语义分析器能检测出源程序中的语义错误，收集代码生成阶段要用到的类型信息。语义分析器利用语法分析阶段确定的层次结构AST(Abstract Syntax Tree，抽象语法树)来识别表达式和语句中的操作符和操作数。

4) 中间代码生成器。编译器在完成语法分析和语义分析之后产生源程序的一个中间表示，该中间表示应该具有两个重要的性质：一是易于产生；二是易于翻译成目标程序。

5) 代码优化器。代码优化器试图对中间代码进行优化，以产生高效的机器代码。代码优化器所进行的变换是等价变换，它不改变程序的运行结果，但是将使程序运行效率更高（运行时间更短，占用空间更小）。

6) 代码生成器。代码生成器生成可重定位的机器代码或汇编代码，在这一阶段中，中间表示将被翻译成目标机器指令序列，即二进制代码。

编译器的多个阶段可以分为前端、中端和后端三个大阶段。前端依赖于源语言并且在很大程度上独立于目标机器。前端一般包括词法分析、语法分析、语义分析和中间代码生成。相当一部分代码的优化工作在前端完成，比如常量传播优化等。中端是指代码优化器，完成的工作主要是对前端生成的中间代码进行目标无关的优化。后端包括编译器中依赖于目标机器的阶段，指的是代码生成器。一般来说，后端完成的任务不依赖于源语言，只依赖于中间语言[38]。

2.2.2 LLVM 编译器框架

LLVM 是构架编译器的框架系统，以面向对象编程语言 C++编写而成，用于优化以若干程序语言编写的程序的编译时间、链接时间、运行时间以及空闲时间，对开发者保持开放，兼容已有版本[39]。LLVM 计划启动于 2000 年，最初由 University of Illinois at Urbana-Champaign 的 Chris Lattner 主持开展。2005 年 Chris Lattner 加盟 Apple 公司，致力于 LLVM 在 Apple 开发体系中的应用，Apple 是 LLVM 计划的主要资助者，LLVM 被苹果 IOS 开发工具、Xilinx Vivado、Facebook、Google 等各大公司采用。LLVM 编译器基于传统的三段式设计，通过翻译成通用的中间表示语言作为中端优化器的输入，用于支持不同的前端语言和架构[40]，其结构如图 2.7 所示。

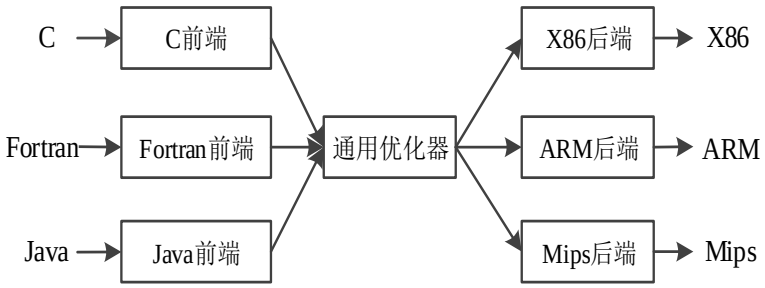


图 2.7 LLVM 结构

从图 2.7 中可以看出，LLVM 的编译流程可以分为三大部分：高级语言前端、中间代码优化器（通用优化器）、后端代码生成器。高级语言前端将使用高级语言 C、Fortran、Java 等编写的程序转换为 LLVM IR，相对高级语言前端和目标平台处理器后端均独立的中间代码优化器则对转换得到的 LLVM IR 进行优化，经过优化后的 IR 通过后端代码生成器生成针对目标平台处理器的机器代码。

LLVM 官方提供的高级语言前端为 CLANG 项目，CLANG 项目为 LLVM 提供了 C、C++、Objective C/C++、OpenCL、CUDA 和 RenderScript 语言的编译器前端和工具基础结构。提供了一个 GCC 兼容的编译器驱动程序 clang 和一个 MSVC（Microsoft Visual C++）兼容的编译器驱动程序 clang-cl.exe。CLANG 前端完成的工作如图 2.8 所示。

图 2.8 中词法分析将 C、C++、Ojbective-C 的语言结构拆分为一组单词和标记来处理输入的源代码文本，删除注释、空格、换行和制表符等与编译无关的字符。C/C++预处理器在进行语义分析之前进行预处理，负责扩展宏或通过以#开头的预处理器指令跳过部分代码。预处理器和词法分析器紧密相关并且互相影响。在词法分析对源代码进行标记后，语法分析将标记好的标识符(Token)进行进一步的处理以形成表达式、语句和函数体等。语法分析将检查 token 是否符合编程语言的语法规则，但对表达式的含义不进行进一步的分析。语法分析的过程被称之为解析，它接收由词法分析所产生的 token 流作为输入，然后输出生成的抽象语法树。语义分析以语法分析生成的抽象语法树作为输入，检查源程序有无语义错误，为代码生成阶段收集类型信息，检查完毕之后输出抽象语法树。语义分析的一个主要任务是进行类型的检查，检查每一个运算符是否具有语言规范所允许的操作数类型，当不符合语言规范时向用户报告错误。语义分析完成之后，编译器将生成的抽象语法树作为 LLVM IR 生成器的输入，遍历抽象语法树生成等价的 LLVM IR。

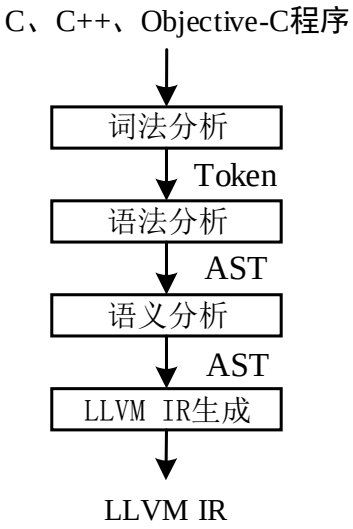


图 2.8 CLANG 工作流程

2.2.3 LLVM 的中间表示

LLVM IR 属于高级语言前端的输出，将作为中间代码优化器的输入。中间表示应该具有两个重要的性质：一是易于产生；二是易于翻译成目标平台代码[41]。易于产生可以保证不同的高级语言源程序易于转换为中间表示，也可以与其他中间表示相互转换。易于翻译成目标平台代码，表明中间表示应具有高度抽象的特性。中间代码优化器对中间表示进行优化时，其输入是 LLVM IR，其输出也是 LLVM IR，所以对中间表示的优化不影响其翻译成目标平台代码。LLVM IR 有三种表现形式：（1）编译过程中内存形式的中间表示；（2）磁盘中的比特码(bitcode)；（3）用户可读的汇编码。其中，编译过程中内存形式的中间表示存在于各个优化和分析 Pass 中，比特码和汇编码分别以.bc 格式和.ll 格式保存在磁盘中[42]。LLVM 编译器中 LLVM IR 存在的阶段如图 2.9 所示。

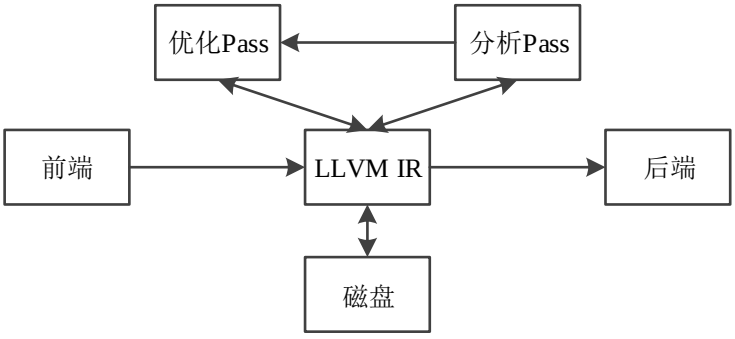


图 2.9 LLVM IR 存在阶段

从图 2.9 中不难看出，LLVM IR 是连接前端和后端的重要枢纽。前端将高级程序语言转化为平台无关的中间表示，中端对平台无关的中间表示进行分析、优化，后端将中间表示降级为目标平台的可执行文件或目标平台文件。LLVM IR 所处的位置决定了其应是平台无关的语言并且易于产生和翻译成目标平台代码。

LLVM IR 具有三个重要的特性：（1）静态单赋值。SSA 形式的中间表示要求每个变量只分配一次，并且每个变量在使用之前定义，这种形式的中间表示，具有简单的 use-def（使用点-定义点）链，对于代码优化具有很大优势（例如常量传播和公共子表达式消除）。SSA 形式的中间表示具有类型安全性、底层操作性、灵活性，并且能够表达绝大多数高级语言。SSA 的这些优良特性，使得 LLVM IR 成为一种足够底层足够通用的 IR，通过 LLVM IR 高级语言的诸多特性可以得到实现。（2）三地址指令。数据处理指令有两个源操作数，将结果放在目标操作数中。（3）无限数量的寄存器。寄存器的数量不限，不用考虑寄存器分配问题[43]。以一个简单的程序来说明 LLVM IR 的特点，程序代码如图 2.10 所示。

```
int a, b, c;
int main()
{
   c = a + b;
   return 0;
}
```

图 2.10 一个简单的 C 程序

图 2.10 的代码中声明了 int 类型的全局变量 a、b、c，然后在 main 函数中将变量 a 和变量 b 相加的结果赋值给变量 c，通过返回一个 0 结束 main 函数的调用。使用 LLVM 提供的前端工具 clang 对代码进行编译，指定-emit-llvm –S 选项生成 LLVM IR，如图 2.11 所示。

```
; ModuleID = 'a.c'
source_filename = "a.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
@a = common dso_local local_unnamed_addr global i32 0, align 4
@b = common dso_local local_unnamed_addr global i32 0, align 4
@c = common dso_local local_unnamed_addr global i32 0, align 4
; Function Attrs: norecurse nounwind uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
entry:
  %0 = load i32, i32* @a, align 4, !tbaa !2
  %1 = load i32, i32* @b, align 4, !tbaa !2
  %add = add nsw i32 %1, %0
  store i32 %add, i32* @c, align 4, !tbaa !2
  ret i32 0
}
```

图 2.11 图 2.10 程序对应的 LLVM IR

从图 2.11 的代码中可以看到，LLVM IR 文件中的内容都被称为 Module，Module 是 LLVM IR 的顶层数据结构。每个 Module 都包含一系列的 Function，每个 Function 都包含若干个 Basic block，每个 Basic block 中又包含一系列的 Instruction 语句。Module 包含全局变量、目标数据布局、外部函数原型声明和数据结构结构的声明。其组织结构如图 2.12 所示。

图 2.11 的代码中 target triple 结构包含了目标平台 cpu 架构操作系统信息，如 x86_64-unknown-linux-gnu 表示目标平台为 x86_64 的处理器，生产商未知，操作系统为 linux，运行环境为 GNU。Target datalayout 结构是目标平台的数据布局包含了目标平台的字节序和类型大小信息，如 e-m:e-i64:64-f80:128-n8:16:32:64-S128 表示目标平台字节序为小端，类型的信息格式为 type<size>:<abi>:<preferred>，比如 i64:64 表示 i64 类型的 ABI（Application Binary Interface，应用二进制接口）对齐为 64 位，n8:16:32:64 表示目标平台自然支持的整型数据类型宽度为 8、16、32、64 位，S128 表示栈对齐为 128 位。

图 2.11 的代码中变量可以是以%符号开头的任何名称，包括从零开始的数字，例如%0，%1 等且数量没有限制。以@为前缀的标识符为全局变量，如@a，@b，@c，以%为前缀的变量为局部变量或栈变量，如%1，%2，%3。Load、add

和 store 等为操作，load 表示从内存取值，add 表示加法操作，store 为将值存回内存。Align 4 表示待访问的内存地址需按 4 字节对齐。I32，i32*等表示变量的类型，如 i32 表示 int 类型，i32*表示 int 类型的指针。Add 标识符后面的 nsw 表示该 add 操作不会导致结果溢出，用于中间表示优化。%1 = load i32, i32* @a, align 4 表示从全局变量 a 的内存地址中按 4 字节对齐取出变量 a 的值存放到变量%1 中，%2 = load i32, i32* @b, align 4 表示从全局变量 b 的内存地址中取出变量 b 的值存放到%2 中，%3 = add nsw i32 %1, %2 表示将变量%1 和%2 的值相加并且将结果赋值给%3，然后 store i32 %3, i32* @c, align 4 将%3 的值按 4 字节对齐存放到全局变量 c 中。赋值号(=)左边的变量为定义点(def 点)，赋值号右边的变量为使用点(use 点)，所以通过 SSA 形式的 LLVM IR 可以极容易找出变量之间的 use-def 链，比如对于 IR 语句%3 = add nsw i32 %1, %2 来说，add 的两个操作数分别为%1，%2，当进行数据引用关系分析时，可以在 LLVM IR 中寻找%1，%2 的定义点，即 IR 语句%1 = load i32, i32* @a, align 4 和%2 = load i32, i32* @b, align 4，然后根据优化的需要可以选择继续分析下去，直到得到足够的优化信息。

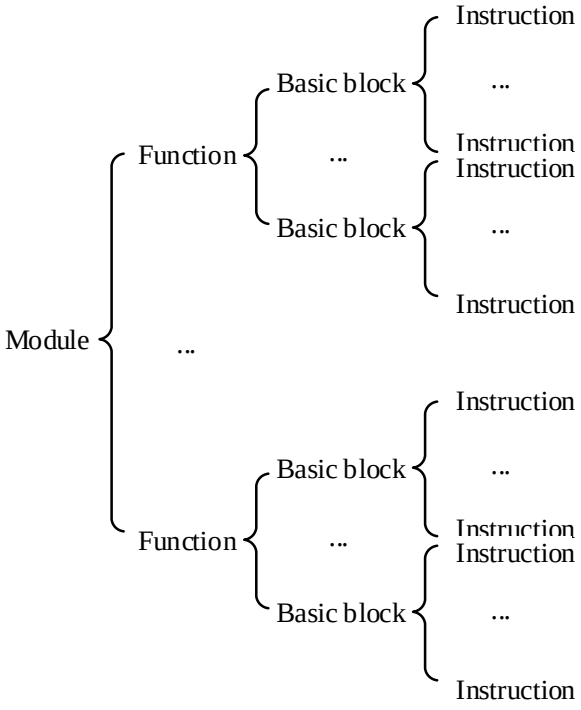


图 2.12 Module 结构

2.2.4 LLVM 的中间表示降级

DAG 是一种重要的数据结构，指的是一个无回路的有向图，在寻求最短路径、数据压缩等多种算法中均有使用。LLVM 在对中间表示进行降级处理的过程中，首先将 LLVM IR 转换成 DAG 的形式，然后对 DAG 图进行降级处理，其处理流程如图 2.13 所示。

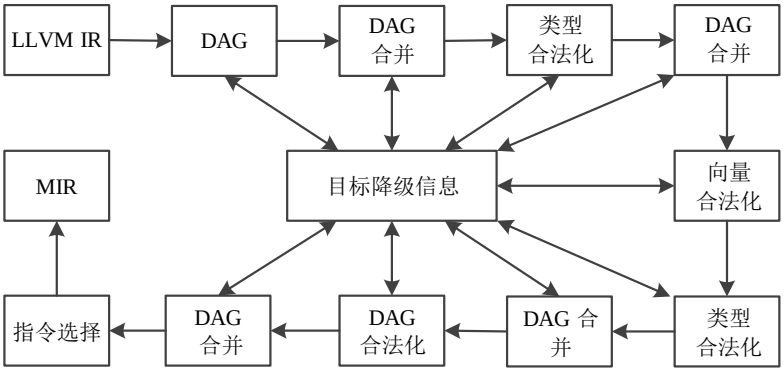


图 2.13 LLVM IR 降级处理

从图 2.13 中可以看到在初始阶段 LLVM IR 是平台无关的，LLVM IR 转化成 DAG 图时，需要 TLI（TargetLoweringInfo, 目标平台降级信息）的支持，然后再进行 DAG 合并、类型合法化和 DAG 合法化等处理，最后再进行指令选择处理生成 MIR(Machine IR, 机器中间表示)，完成 LLVM IR 的降级。在整个降级的过程中主要是 DAG 合并和合法化这两个过程在交替进行，逐步完成中间表示的降级操作。

DAG 是 LLVM IR 的一种等价形式，通过 DAG 可以更好的对中间进行降级处理。从 LLVM IR 降级到 MIR 的过程中是以 Basic Block 为基本单位进行处理的，即对每一个 Basic Block 进行图 2.13 所示的降级过程，从 Basic Block 生成 DAG 的流程如图 2.14 所示。

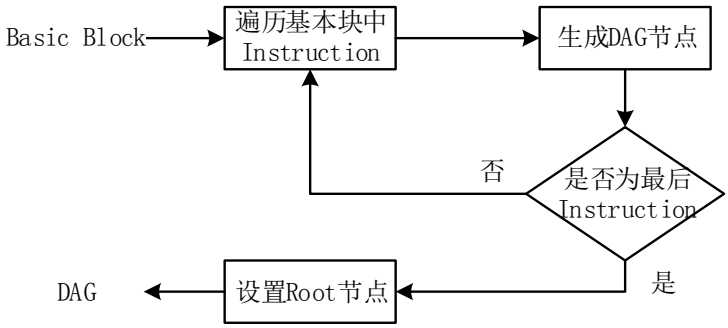


图 2.14 LLVM IR 生成 DAG 的流程

图 2.14 中的处理流程以 Basic Block 输入，然后对 Basic Block 中的 Instruction 语句进行遍历，根据操作关键字和操作数信息生成 DAG 节点，如果当前的 Instruction 语句为最后一条 Instruction 语句则设置 DAG 的 Root 节点然后返回生成的 DAG，否则继续处理下一条 Instruction 语句。2.2.3 节中的图 2.11 中的 LLVM IR 对应的 DAG 如图 2.15 所示。

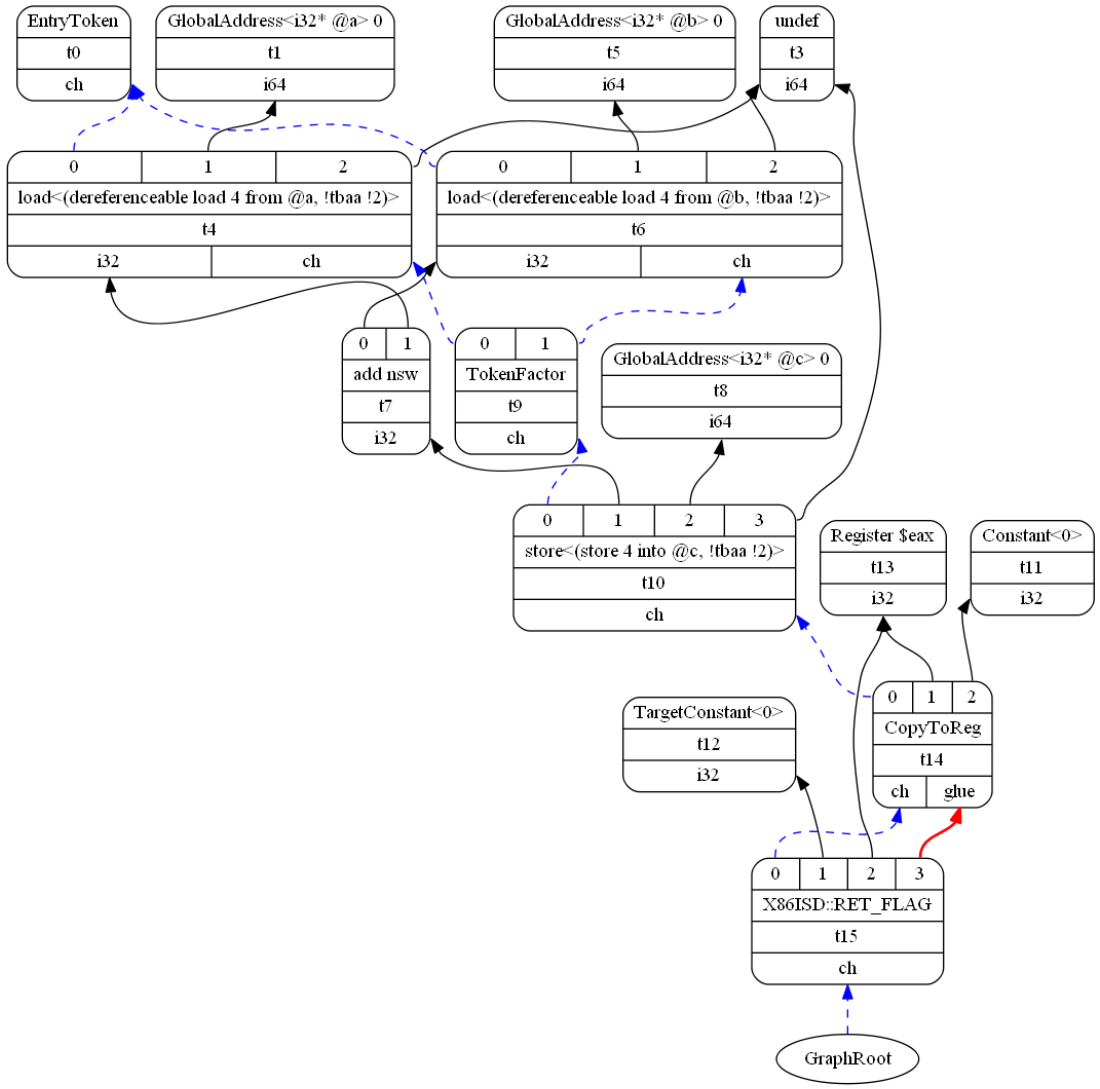


图 2.15 表达式 c = a + b 的 DAG

图 2.15 中，每个圆角方框都表示一个 DAG 节点，椭圆形的 GraphRoot 节点是为了处理方便加上的辅助节点。图中黑色的曲线箭头表示数据流，蓝色的虚线箭头表示控制流边，红色的实线表示黏合边(Glue 边)，由控制流连接的节点其在

生成指令序列时保持相对顺序不变，由黏合边连接的节点在生成指令序列时不会被拆分，即中间不插入任何指令。EntryToken 是 DAG 中控制流边最后一个节点，用来代表 Basic Block 的入口。GlobalAddress 节点的输出为一个内存地址，对应 LLVM IR 中的 i32* @a 或 i32* @b。Load 节点从输入的内存地址中取值，其第一个操作为控制流操作数，第二个操作数为内存地址，第三个内存操作为偏移值。Undef 节点用来表示一个未定义的节点，代表节点的值未知。Add 节点将两个输入操作数的值进行相加，然后输出结果值，对应 LLVM IR 中的%3 = add nsw i32 %1, %2。Store 节点将 add 节点的结果输出作为第 2 个输入操作数，将其值存取全局变量 c 中，对应的 LLVM IR 语句为 store i32 %3, i32* @c, align 4。Load 节点和 Store 节点中的 LD4 和 ST4 表示按 4 字节对齐访问内存，TokenFactor 节点是一个辅助节点用来维持节点间的控制流关系。CopyToReg 节点将输入的操作值拷贝到另一个输入的寄存器操作数，该寄存器可以是物理寄存器也可以是虚拟寄存器。Constant 和 TargetConstant 节点都是用来代表常数，Constant 节点是目标无关的节点，TargetConstant 是目标相关的节点，由于该节点已经目标相关了所以不需要重复处理，Constant 节点需要经过降级才能成为目标相关的节点。X86ISD::RET_FLAG 是 X86 平台下的目标相关的节点，该节点的作用是处理函数调用的的返回值。从 DAG 中可以看出，从中间表示到生成 DAG 的过程中已经使用了目标平台相关的信息，所以 DAG 不是完全目标无关的。

从图 2.15 中可以非常清晰的看出节点之间数据流和控制流关系，有利于进行平台相关和平台无关的多种优化。从 DAG 转换为 MIR 的过程，其实是 DAG 节点降级和合法化的过程，其中涉及指令的降级、节点返回值类型的合法化、节点之间的合并等操作。遍历 LLVM IR 中的 Instruction 语句创建的 DAG 节点未必被目标架构全部支持，因此还需要对 DAG 节点做出修改以适应目标平台，这一过程叫做合法化（legalization）。在 DAG 的初始节点阶段，一些不被支持的节点被认为是不合法的，在 DAG 节点的指令选择之前，这些不合法的节点都将做一些转换以支持目标平台。因此，合法化是指令选择之前最重要的阶段之一。

2.3 窥孔优化介绍

许多编译器软件可以通过精心构造的指令选择和寄存器分配算法来生成高效目标代码，但是这种方式缺乏通用性，通常只针对特定目标平台。还有一些编

译器先将高级语言源程序生成中间代码或目标代码，然后对中间代码或目标代码进行优化转换，以提升目标代码的质量。虽然这种优化转换无法保证在任何数学度量模型下都是最优的，但是一些简单的优化转换的确可以改善目标代码的运行效率和目标代码的大小。

窥孔优化技术就是这样的一种优化转换，用于改进局部代码的质量，这种技术虽然局部且简单，但是有效。它在对目标指令进行优化时，检查目标指令的上下文关系或者一个窗口内的目标指令（窥孔），一旦发现优化机会（可以生成效率更高和体积跟小的代码）就进行优化转换。窥孔优化是一种局部的优化方法，每一次优化都可能为下一次优化创造条件，所以窥孔可以多次进行以得到质量更高的目标代码。常用的窥孔优化技术有：冗余指令消除；控制流优化；代数简化；特有指令替换。这些技术将在第三章（LLVM 中的窥孔优化研究）中结合 LLVM 编译器的中间表示进行研究和分析。

2.4 实验测试集介绍

SPEC(Standard Performance Evaluation Corporation，标准性能评估公司)是一个非盈利性的组织，由计算机厂商、系统集成商、研究机构、大学等公司或机构组成。SPEC 的目标是建立和维护一套用于评估计算机系统性能的标准。SPEC 推出的测试集有三个，分别为 SPEC CPU2000 基准测试集、SPEC CPU2006 基准测试集和 SPEC CPU2017 测试集。

目前，SPEC CPU2006 是业内首选的 CPU 基准测试集。SPEC CPU2006 基准测试集是行业标准化的 CPU 测试基准套件，它是一个标准的计算密集型、高性能的跨硬件的 CPU 测试工具。套件包含定点基准测试程序和浮点基准测试程序，主要测试系统的处理器、内存子系统和编译器。SPEC CPU2006 基准测试一共有 31 个测试程序，其中有两道测试程序为 998.specrand（定点随机数测试）和 999.specrand（浮点随机数测试）测试，主要用来保证其他测试程序将用到的随机数生成功能的正确性。SPEC CPU2006 基准测试集的说明如表 2.1 所示。

表 2.1 SPEC CPU2006 基准测试程序说明

程序名称	编程语言	程序类型	应用领域
400.perlbench	C	定点	PERL 编程语言

401.bzip2	C	定点	压缩
403.gcc	C	定点	C 编译器
410.bwaves	F77	浮点	流体动力学
416.games	Fortran	浮点	量子化学
429.mcf	C	定点	组合优化
433.milc	C	浮点	量子色动力学
434.zeusmp	F77	浮点	物理/流体动力学
435.gromacs	C、Fortran	浮点	生物化学/分子动力学
436.cactusADM	F90、C	浮点	物理/广义相对论
437.leslie3d	F90	浮点	流体动力学
444.namd	C++	浮点	生物学/分子动力学
445.gobmk	C	定点	人工智能：围棋
447.dealII	C++	浮点	有限元分析
450.soplex	C++	浮点	线性规划
453.povray	C++	浮点	图像光线跟踪
454.calculix	F90、C	浮点	结构力学
456.hmmer	C	定点	搜索基因序列
458.sjeng	C	定点	人工智能：象棋
459.GemsFDTD	F90	浮点	计算电磁学
462.libquantum	C	定点	量子计算
464.h264ref	C	定点	视频压缩
465.tonto	F95	浮点	量子化学
470.lbm	C	浮点	流体动力学
471.omnetpp	C++	定点	离散事件模拟
473.astar	C++	定点	路径寻找算法
481.wrf	F90、C	浮点	天气预报
482.sphinx3	C	浮点	语音识别
483.xalancbmk	C++	定点	XML 处理

表 2.1 中包含 12 道定点程序和 17 道浮点程序，编程语言有 C、C++和 Fortran，程序的所涉及的领域有物理、化学、生物、多媒体和计算机软件等等。

2.5 本章小结

本章介绍了申威平台的高性能多线程处理器 26010、高性能单核处理器 111、高性能多核处理器 1621。接着介绍了 LLVM 编译器的相关知识，重点介绍了 LLVM 的中间表示及其降级处理流程。LLVM 的中间表示所处的位置决定了其应是平台无关的语言并且易于产生和翻译成目标平台代码，同时为进行各种优化提供了便利。然后，介绍了窥孔优化的基本概念，它主要对中间代码和目标代码进行优化，虽然是一种局部的优化转换，但是有时能够带来意想不到的效果。最后，对本文所使用的基准测试集 SPEC CPU2006 进行了介绍。

3 LLVM 中的窥孔优化研究

窥孔优化技术在 LLVM 编译器中有较多应用，例如在通用优化工具 OPT 中提供了窥孔优化器（peephole optimizer），在代码生成阶段提供了针对目标平台的窥孔优化器(<target>peephole optimizer)。LLVM 编译器中的窥孔优化使用 Pass（趟）的方式编写，方便与其他优化 Pass 和分析 Pass 组合。对于不同的目标平台有不同的窥孔优化规则，这是由目标平台的特性决定的，通常使用的技术有：冗余指令消除、控制流优化、代数简化和特有指令替换等，以下将使用 LLVM 的中间表示对上述窥孔优化技术进行研究分析。

3.1 冗余指令消除

中端代码生成器和目标代码生成器容易产生冗余指令，删除这些冗余指令并不对生成的可执行代码产生任何副作用。通过对中间代码或目标代码进行局部检查来识别和删除冗余指令，这种方法虽然比较简单和局部，但是可以减少中间代码和目标代码的大小、提升执行效率。

3.1.1 冗余的 load 和 store 指令消除

Load 和 store 指令是访存指令，load 用于从内存中读取数据到寄存器，store 指令用于将寄存器中的数据存储到内存。寄存器是 CPU 中一种数量相当较少的高速存储装置，和普通的存储器相比，有限的大小以及相对较快的存储速度使得寄存器成为绝大多数机器体系结构中的临界资源。内存是计算机中 CPU 和外存进行沟通的桥梁，计算机中所有程序的运行都要加载到内存中进行，因此内存是计算机中的重要部件。计算机存储装置的速度和容量如图 3.1 所示。

图 3.1 中 TLB 是地址转换后援缓冲器(Translation Lookaside Buffer)的简称，也称快表。从图 3.1 中的存储器金字塔可以发现不同层级的存储器之间其时钟周期相差是指数级别的，容量也是呈指数级变化。图中的箭头是计算机中存储器之间数据的流动方向，可以看出 Cache(高速缓存)是计算机为了缓和寄存器与内存之间巨大的速度差异，利用程序的局部性原理引入的一个缓冲装置，用以充分利用 CPU 资源提升计算机的计算性能。寄存器和内存之间巨大的速度差异导致

load 和 store 指令的开销不菲，所以消除冗余的 load 和 store 指令不仅有利用提升程序性能，而且还能减小程序的体积。含有冗余 load 和 store 指令的 C 程序如图 3.2 所示。

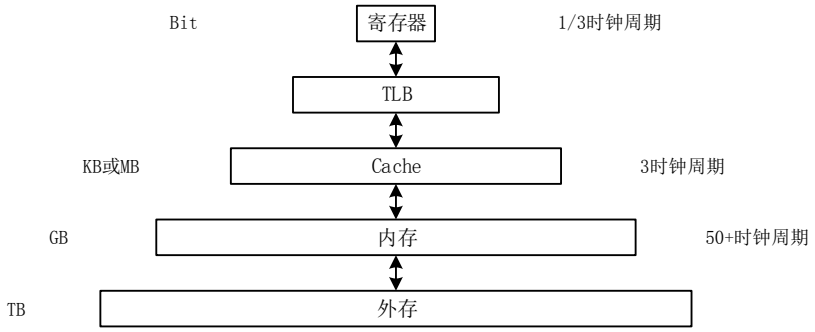


图 3.1 存储器金字塔

```
int a=1, b;
void main()
{
        a = b;
        b = a;
}
```

图 3.2 含有冗余 load 和 store 指令的 C 程序

图 3.2 中，语句 b = a 是冗余的。不使用 LLVM 优化，即使用-O0 选项将图 3.2 中的代码使用 CLANG 编译工具生成的 LLVM IR 及优化后的 LLVM IR 如图 3.3 所示。

```
@b = common dso_local global i32 0, align 4
@a = dso_local global i32 1, align 4
define dso_local void @main() #0 {
  %1 = load i32, i32* @b, align 4
  store i32 %1, i32* @a, align 4
  %2 = load i32, i32* @a, align 4
  store i32 %2, i32* @b, align 4
  ret void
}
```

优化

```
@b = common dso_local global i32 0, align 4
@a = dso_local global i32 1, align 4
define dso_local void @main() #0 {
  %1 = load i32, i32* @b, align 4
  store i32 %1, i32* @a, align 4
  ret void
}
```

图 3.3 图 3.2 程序对应的 LLVM IR 及优化后的 LLVM IR

图 3.3 的左图表示未进行冗余的 load 和 store 指令消除的 LLVM IR，图 3.3 的右图表示进行优化后的 LLVM IR。从图 3.3 的左图可以看到存在对同一内存的存取操作，即 store i32 %1, i32* @a, align 4 和%2 = load i32, i32* @a, align 4，store 指令将变量%1 的值存入变量 a 的地址中，然后又用 load 指令将变量 a 的值取到变量%2 中，此时%1 和%2 是相等的，那么 load 指令之后的所有用到%2 的地方都可使用%1 替代，所以该条 load 指令是冗余指令可以删除。%1 的值是从变量 b 的地址中取出的，store i32 %2, i32* @b, align 4 语句将变量%2 存回变量 b 中，因为变量%2 的值与变量%1 相等，所以这条 store 指令可以删除。需要删除的指令为图 3.3 左图中加粗部分的语句。进行冗余的 load 和 store 指令消除优化后，LLVM IR 代码中少了一条 load 指令和一条 store 指令，即减少了两次访存操作，提升了程序的执行效率。

3.1.2 不可达指令消除

如果当一段程序中包含一些在任何情况下都无法执行的指令时，那么这些无法执行的指令就被称之为不可达指令。不可达指令一般是某个条件的分支，当条件恒为假时，该分支的指令将永远得不到执行，从而成为不可达指令。举例如图 3.4 所示。

```
int a = 1;
int b;
void main()
{
   a = 0;
   if(a)
      b = a;
}
```

图 3.4 含有不可达指令的 C 程序

从图 3.4 中的程序可以发现 if 条件恒为假，从而语句 b = a 永远得不到执行，使用 LLVM 的前端工具命令 clang 在-O0 选项(不进行优化)下生成的 LLVM IR 及优化后的 LLVM IR 如图 3.5 所示。

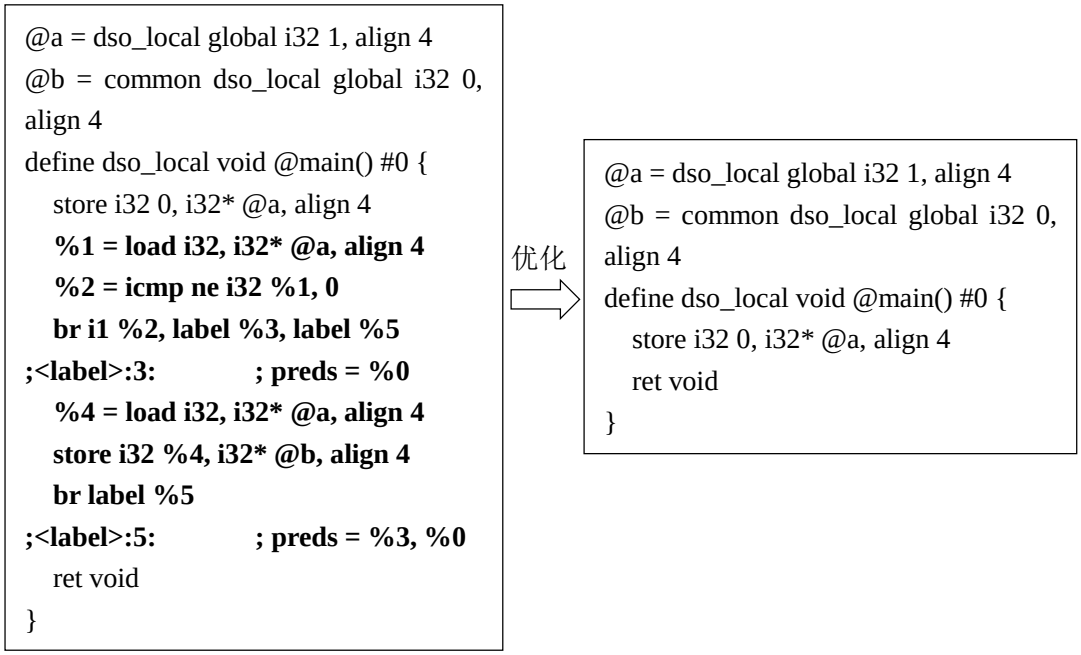


图 3.5 图 3.4 进行不可达指令消除前后的 LLVM IR

图 3.5 左图为优化前的 LLVM IR，右图为优化后的 LLVM IR。图 3.5 左图中 br 指令为分支跳转指令，将根据第一个操作数的真假值来选择跳转到哪个分支。Icmp 指令是整数比较指令，ne 是 icmp 指令的比较条件，即不等于(not equal)。Br 指令根据变量%2 的值来跳转，若%2 为真则跳转到<label>:3 对应的指令出，否则跳转到<label>:5 对应的指令处。变量%2 的定义点(def 点)来自%2 = icmp ne i32 %1, 0，icmp 指令比较变量%1 的值是否不等于 0。变量%1 的定义点(def 点)为%1 = load i32, i32* @a, align 4，该 load 指令从全局变量 a 中取值。继续往上查找对全局变量 a 修改的语句，找到 store i32 0, i32* @a, align 4，即将 0 存入全局变量 a 中，从而可知变量%1 的值为 0。因此，%1 不等于 0 的真值为假，br 指令将调转到<label>:5 对应的指令出，从而标签<label>:3 对应的指令将永远不被执行，是不可达指令可以被删除。图 3.5 左图中加粗的 IR 语句为冗余指令，进行删除后生成的 LLVM IR 如图 3.5 右图所示。优化后 main 函数内只剩下 a = 0;语句了，不仅删除了不可达指令，而且删除冗余的 icmp、load 和 br 指令。不可达指令的删除可以减小程序的体积，冗余的 icmp、load 和 br 指令的删除可以提高程序的运行效率，避免执行冗余指令。

3.1.3 无用测试和比较指令消除

在程序中存在测试和比较指令，若测试条件或比较条件的真值恒为真或恒为假，那么执行测试或比较的指令将显得冗余。为了提升程序的执行效率，减小程序的体积，对于无用测试和比较指令应当予以删除。程序举例如图 3.6 所示。

```
int a = 1;
int b;
void main()
{
   a = 0;
   b = a == 0;
}
```

图 3.6 无用测试和比较指令消除例子 C 程序

图 3.6 的程序中将 a 和 0 的比较结果存入全局变量 b 中，从图 3.6 中的代码中可以清楚的看到表达式 a==0 的结果应该是恒为真的，因为 a = 0，所以存在冗余的比较操作。使用 LLVM 的前端编译命令 clang 在 O0 选项下生成 LLVM IR 及优化后的 LLVM IR 如图 3.7 所示。

```
@a = dso_local global i32 1, align 4
@b = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @main() #0 {
   store i32 0, i32* @a, align 4
   %1 = load i32, i32* @a, align 4
   %2 = icmp eq i32 %1, 0
   %3 = zext i1 %2 to i32
   store i32 %3, i32* @b, align 4
   ret void
}
```

优化 ⇨

```
@a = dso_local global i32 1, align 4
@b = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @main() #0 {
   store i32 0, i32* @a, align 4
   store i32 1, i32* @b, align 4
   ret void
}
```

图 3.7 图 3.6 程序进行无用测试和比较指令消除前后的 LLVM IR

图 3.7 左图为为优化的 LLVM IR，右图为优化后的 LLVM IR。从图 3.7 左图中可以看到 icmp 指令的第一个操作数为变量%1，第二个操作数为常量 0，如

果可以分析清楚变量%1 得取值范围，那么就可以删除该条比较指令。变量%1 的定义点（def 点）为%1 = load i32, i32* @a, align 4，对于变量 a 最近的一次访问在语句 store i32 0, i32* @a, align 4，该 store 指令将常量 0 存入了变量 a 中，然后又通过 load 将值取出，显然该条 load 指令是冗余的。由此可知，%1 的值为 0， 从而 icmp 指令的结果恒为真，所以 icmp 属于冗余的测试和比较指令，应当得到删除，然后将所有使用到变量%2 的地方替换为 1。%3 = zext i1 %2 to i32 将%2 的值进行零扩展，然后存储到变量 b 中，因为%2 是一个常量值，所以可以直接将常量值存入变量 b 中。需要删除的指令如图 3.7 左图中加粗的 IR 语句，优化后为右图中的 LLVM IR，其中用下划线标记的语句为有更改的语句。

图 3.7 右图中可以发现，除了删除了冗余的 icmp 指令之外，还删除了冗余的 load 指令，然后将所有使用变量%1 值地方全部替换为 0。通过进行无用的测试和比较指令的消除，减少了访存指令 load 和比较指令 icmp，加快了程序的运行速度。

3.1.4 归零序列消除

程序中存在一些无用的算术或逻辑指令序列，这些无用的算术或逻辑指令序列被称为归零序列（Null sequence），删除这些指令序列不仅不影响程序的正确执行，而且还可以提升程序的执行效率。比如表达式 b = a + 0;中，这个 a+0 的操作完全可以省去，直接执行 b = a;即可。举例如图 3.8：

```
int a = 1;
int b;
void main()
{
   b = a + 0;
}
```

图 3.8 归零序列消除例子 c 程序

使用 clang 命令的-O0 选项将图 3.8 中的程序生成 LLVM IR，未优化和优化后的 LLVM IR 如图 3.9 所示。

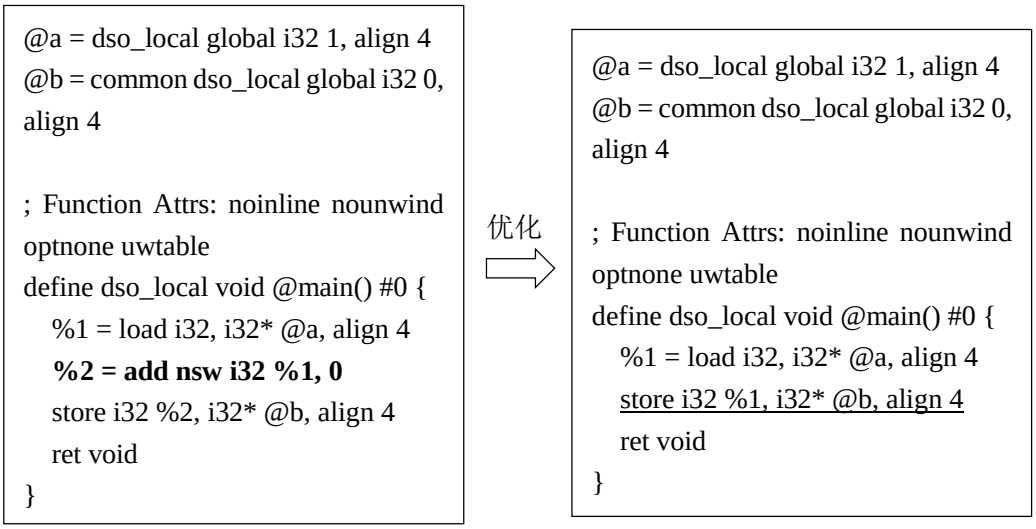


图 3.9 图 3.8 进行归零序列消除前后的 LLVM IR

图 3.9 左图为未进行优化的 LLVM IR，右图为进行优化后的 LLVM IR。图 3.9 左图中，从 add 指令可以看到第二个操作数为常量 0，第一个操作数%1 的定义点为(def 点)为%1 = load i32, i32* @a, align 4，通过将 LLVM IR 中所有%2 的使用点（use 点）全部替换为%2，可以删除调归零序列指令 add。待删除的 IR 语句如图 3.9 左图加粗部分所示，优化后生成了图 3.9 的右图，其中用下划线标记的语句为有更改的语句。优化后 LLVM IR 与未优化的 LLVM IR 相比，少了一条 add 指令，提升了程序的执行效率。

3.2 控制流优化

中间代码生成器或代码生成器生成的无条件跳转指令或条件跳转指令可以以另一条跳转指令为目标，适当的改变跳转指令的目标可以改善程序的控制流，在某些情况下甚至可以删除第二个跳转指令。如果一个条件跳转指令之前有一个比较指令，那么这个条件跳转指令可以和比较指令合并为一个新的条件跳转指令。通过对跳转指令进行优化，从而达到优化控制流的目的，提升程序的执行效率。

如果无条件或条件跳转指令以另一条无条件跳转指令为目标，那么改变前一条跳转指令的跳转目标为第二条无条件跳转指令的目标不仅不影响程序的正确性，反而能节省一条指令的执行的时间，有利于提升程序的性能。举例如图 3.10 所示。

```
int a;
int b = 1;
void main()
{
   goto tb;
   a = 100;
   ta:
   b = 0;
   tb:
   goto tc;
   a = 10000;
   tc:
   a = 100000;
   if(b) goto ta;
}
```

图 3.10 控制流优化例子 C 程序

图 3.10 的程序中有 ta、tb、tc 三个标签，然后通过 goto 语句进行跳转。Goto 属于无条件跳转指令，若和 if 等条件结合起来可以生出条件跳转指令。在程序开始时先通过 goto tb;语句跳转到标签 tb 处，然后在通过 goto tc;语句跳转到标签 tc 处，这两条跳转指令都为无条件跳转指令，那么第一条跳转指令无需通过第二条跳转指令跳转至标签 tc 处，可以直接跳转到 tc 处。使用 clang 命令在-O0 选项下将图 3.10 中的程序生成 LLVM IR，为优化的 LLVM IR 和优化后的 LLVM IR 如图 3.11 所示。

图 3.11 中左图为优化前的 LLVM IR，右图为优化后的 LLVM IR。从图 3.11 的左图中可以看到 br label %2 语句中通过 br 指令跳转到标签 2（<label>:2:）处，然后标签 2 处的第一条指令是一条 br 指令 br label %3，该条指令没有条件判断所以属于无条件跳转指令，通过该条跳转指令跳转到标签 3（<label>:3:）处。两条跳转指令之间没有其他的指令，所以可以修改第一条跳转指令的目标标签为第二条跳转指令的目标标签，这样一来就可以节省一次跳转指令的时间，从而提升程序的性能。待修改的指令如图 3.11 左图加粗部分所示，优化后的 LLVM IR 如图 3.11 右图所示，其中用下划线标记的语句为有修改的语句。优化后的 LLVM IR 与未优化的 LLVM IR 相比少了一条 br 指令，即节省了一条跳转指令。

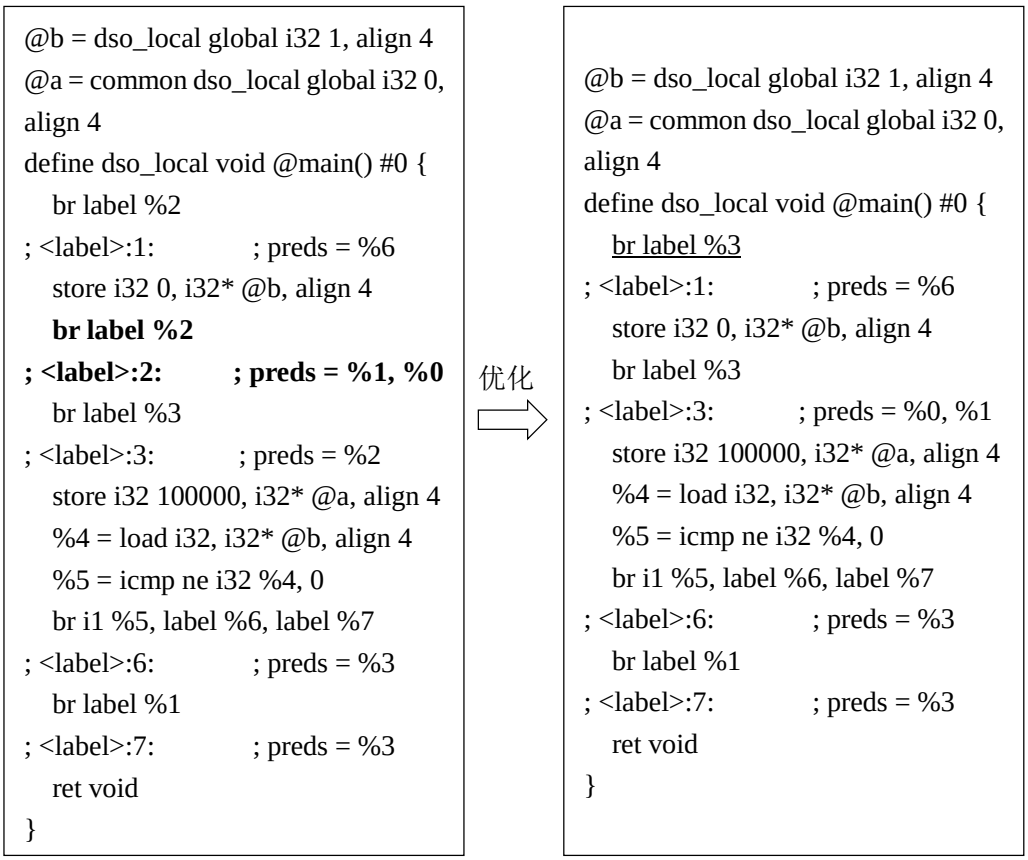


图 3.11 图 3.10 进行控制流优化前后的 LLVM IR

3.3 代数简化

如果程序使用两个或多个常量组成表达式，那么可以在程序编译的过程中对表达式求值，如表达式 2 * 3，可以求得其值为 6。程序中的某些算术指令可以被替换为更简单的指令，比如 c = a * 2，可以使用逻辑左移指令替换为 c = a << 1。在代数简化中，一般使用的方法有常数折叠、强度削弱等方法。

3.3.1 常数折叠

对于程序中的常量表达式其值是可直接在程序的编译过程中计算出来的，该过程称为常数折叠。常数折叠优化一般和常量传播算法一起使用，常数折叠有利于常量传播，常量传播反过来为常数折叠提供更多优化机会。举例如图 3.12 所示。

```
int a;
int main()
{
   a = 365 * 24 * 60 * 60;
}
```

图 3.12 常量折叠例子 C 程序

图 3.12 中的代码用来计算一年有多少秒(不考虑闰年)，显然一年有多少秒，这是一个确定的常量值。使用 clang 命令在-O0 选项下将图 3.12 中的程序生成 LLVM IR，优化前及优化后的 LLVM IR 如图 3.13 所示。

```
@a = common dso_local global i32 0, align 4
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
   %1 = mul i32 365, i32 24
   %2 = mul i32 %1, i32 60
   %3 = mul i32 %2, i32 60
   store i32 %3, i32* @a, align 4
   ret i32 0
}
```

优化

```
@a = common dso_local global i32 0, align 4
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
   store i32 31536000, i32* @a, align 4
   ret i32 0
}
```

图 3.13 图 3.12 中程序进行常量折叠前后的 LLVM IR

图 3.13 左图为未优化的 LLVM IR，右图为优化后的 LLVM IR。图 3.13 左图中的 LLVM IR 通过三次乘法操作进行计算，然后将计算结果存入变量 a 中，然而这三次乘法操作所涉及的操作数都为常量，所以可以进行常量折叠优化。待优化的语句如图 3.13 左图加粗的 IR 语句所示，优化后的 LLVM IR 如图 3.13 右图所示，其中用下划线标记的语句为有修改的语句，该语句直接将常量 31536000 存入变量 a 中，省去了三次乘法操作。

3.3.2 强度削弱

强度削弱是指将程序中执行时间较长的运算转换为时间较短的运算，比如表达式 a * 2 可以转换为 a << 1，左移运算无需使用乘法器，所以执行速度要快于乘法运算。表达 a * 2 可以转换为 a + a，使用加法器而不是乘法器可加快程序

的执行，举例如图 3.14 所示。

```
int a = 100, b;
void main()
{
    b = a * 2;
}
```

图 3.14 强度削弱例子 C 程序代码

图 3.14 程序中将变量 a 的值乘以 2 然后存入 b 中，使用 clang 命令在-O0 选项下将图 3.14 中的程序生成 LLVM IR，优化前后的 LLVM IR 如图 3.15 所示。

```
@a = dso_local global i32 100, align 4
@b = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @main() #0 {
  %1 = load i32, i32* @a, align 4
  %2 = mul nsw i32 %1, 2
  store i32 %2, i32* @b, align 4
  ret void
}
```

优化

```
@a = dso_local global i32 100, align 4
@b = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @main() #0 {
  %1 = load i32, i32* @a, align 4
  %2 = shl i32 %1, 1
  store i32 %2, i32* @b, align 4
  ret void
}
```

图 3.15 图 3.14 中程序进行强度削弱前后的 LLVM IR

图 3.15 左图为未优化的 LLVM IR，右图为优化后的 LLVM IR。从图 3.15 左图中的 LLVM IR 可以发现生成了 mul 指令（乘法指令），该指令可以被优化为 shl 指令（逻辑左移指令），优化后的 LLVM IR 如图 3.15 右图所示。优化后的代 LLVM IR 与优化前的 LLVM IR 相比，乘法指令 mul 被替换成逻辑左移指令 shl（如图 3.15 左图中加粗部分和右图中下划线部分），在 CPU 中逻辑操作所需的资源比乘法指令所需的资源要少，所以替换为逻辑指令后，提升了程序的性能。

3.4 特有指令替换

不同的目标平台以及目标平台指令集都提供了目标相关的特有指令，比如

数据预取、寻址方式、浮点运算加速、数据传送等指令。为了提升程序的运行效率，充分发挥目标平台指令集的潜力，需要在程序编译的过程中进行特有指令替换，比如一条相邻的乘法和加法指令可以替换为一条效率更高的乘加指令[44]。表达式 d = a ×b + c 中，变量 d,a,b,c 都是浮点类型的变量，该表达式将变量 a 和 b 的乘积加上变量 c 的结果值存入变量 d 中。如果没有乘加特有指令，那么完成 a × b + c 的运算分别需要一条浮点乘法指令和一条浮点加法指令。如果使用乘加特有指令，完成 a ×b + c 只需使用一条乘加指令，从而提升了程序的执行。表达式 d = a ×b + c 生成的 LLVM IR 及优化后的代码如图 3.16 所示。

```
@a = common dso_local global double 0.000000e+00, align 8
@c = common dso_local global double 0.000000e+00, align 8
@b = common dso_local global double 0.000000e+00, align 8
@d = common dso_local global double 0.000000e+00, align 8
define dso_local void @main() #0 {
  %1 = load double, double* @a, align 8
  %2 = load double, double* @c, align 8
  %3 = fmul double %1, %2
  %4 = load double, double* @b, align 8
  %5 = fadd double %3, %4
  store double %5, double* @d, align 8
  ret void
}
```

⇩ 优化

```
@a = common dso_local global double 0.000000e+00, align 8
@c = common dso_local global double 0.000000e+00, align 8
@b = common dso_local global double 0.000000e+00, align 8
@d = common dso_local global double 0.000000e+00, align 8
define dso_local void @main() #0 {
  %1 = load double, double* @a, align 8
  %2 = load double, double* @c, align 8
  %4 = load double, double* @b, align 8
  %5 = call double @llvm.fmuladd.f64(double %1, double %4, double %2)
  store double %5, double* @d, align 8
  ret void
}
```

图 3.16 表达式 d = a × b + c 进行特有指令替换前后的 LLVM IR

图 3.16 上图为未进行优化的 LLVM IR，下图为优化后的 LLVM IR。对图 3.16 左图中的 LLVM IR 进行特有指令优化后，浮点乘法指令 fmul 和浮点加法指令 fadd 将融合成内建函数乘加函数的调用，该调用在 LLVM IR 降级阶段逐步降级为目标平台相关的特有的乘加指令，从而完成特有指令的替换，优化后的代码如下图 3.16 右图所示。优化后的 LLVM IR 和未优化的 LLVM IR 相比，原来的浮点乘法指令 fmul 和浮点加法指令 fadd 被 call 指令所替换，该 call 指令将在 LLVM IR 的降级过程转换成目标平台特有的浮点乘加指令 fma，从而达到优化的目的。

3.5 激进的窥孔优化

窥孔优化的本质是一种优化转换，它在对目标指令进行优化时，检查目标指令的上下文关系或者一个窗口内的目标指令（窥孔），一旦发现优化机会（可以生成效率更高和体积跟小的代码）就进行优化转换。这种贪心的优化方式可能导致过度（激进的）优化，反而引起目标代码的运行效率下降，不利于程序的优化。LLVM 编译器中存在激进的窥孔优化，举例如图 3.17 所示。

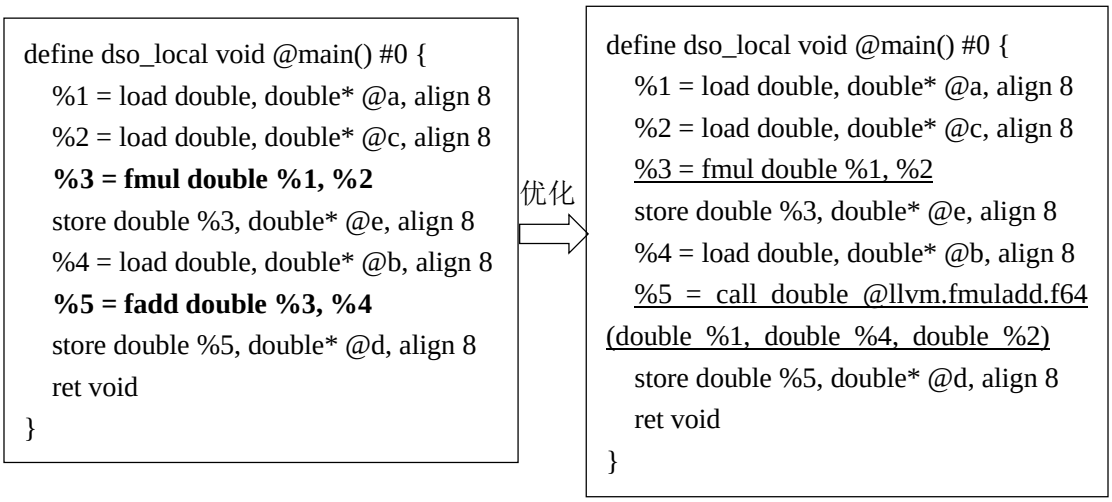


图 3.17 激进的窥孔优化

图 3.17 左图是未优化前 LLVM IR，右图是优化后的 LLVM IR。从图 3.17 左图的 LLVM IR 可以发现，该 LLVM IR 与图 3.16 左图的 LLVM IR 十分相似，但少了 store double %3, double* @e, align 8 语句。变量%3 除了被语句 store double %3,

double* @e, align 8 使用外，还被语句%5 = fadd double %3, %4 使用。由于变量%3 有多个使用点(use 点)，所以进行优化转后不能被删除，优化后的 LLVM IR 如图 3.17 右图所示。与图 3.16 右图的 LLVM IR 相比，虽然同样生成了浮点乘加指令的内建函数，但是语句%3 = fmul double %1, %2 被保留了下来。一条浮点乘法指令和一条浮点加法指令所消耗的资源不比一条浮点加法指令和浮点乘加指令多，所以，进行优化后其性能反而下降，不宜进行优化转换。为了解决激进的窥孔优化，本文在第 4 章提出了节点融合优化方法。

3.6 本章小结

冗余指令消除、控制流优化、代数简化、特有指令替换等都是常用的窥孔优化方法。这些优化方法虽然简单和局部，但是其优化效果有时也很可观，比如在一个循环中通过进行窥孔优化就可能带来极大的性能提升。本章对冗余指令消除、控制流优化、代数简化和特有指令替换优化结合 LLVM 的中间表示进行了举例说明，对优化过程进行了分析。

4 节点融合优化方法

节点融合的基本思想为将多个节点优化为一个高效的融合节点，减少诸如指令、寄存器、时钟周期、访存等开销，达到减少程序运行时间、提升访存效率等目的。通过在申威平台上研究 LLVM 编译器中的窥孔优化技术，结合申威平台指令集的特点，提出了节点融合优化方法。节点融合优化提供了一种代价评估模型以避免过度优化，评估节点融合前后的代价，若融合前的代价大于融合后的代价，则进行节点融合优化，否则保持原状。本章在 LLVM 编译器的中间表示阶段、DAG 合并阶段和指令选择阶段实现节点融合优化方法并且进行了优缺点分析。

4.1 节点融合优化简介

节点融合是指将多个节点融合为一个高效节点，节点融合优化方法在许多编译器中均有应用。比如，加速线性代数编译器(Accelerated Linear Algebra, XLA)利用节点融合优化将计算图中的多个算子融合为一个高效算子，以提高执行速度、改善内存使用、减少对自定义操作的依赖。节点融合优化属于窥孔优化技术，它利用 DAG 图匹配或 Pattern 匹配的方式来寻找可进行融合的节点，通过对融合前后的节点进行代价评价，根据代价决定是否进行节点融合优化。节点融合过程如图 4.1 所示。

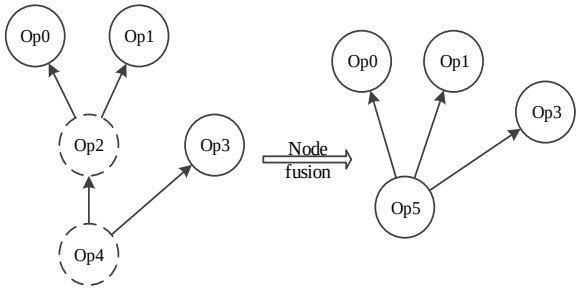


图 4.1 节点融合

图 4.1 中的圆圈为操作节点，圆圈中的标识符为节点名称，虚线圆圈为待融合的操作节点，箭头为节点间的数据依赖关系。图 4.1 左图中，操作节点 Op2 依赖操作节点 Op0 和操作节点 Op1 的运算结果，操作节点 Op4 依赖操作节点 Op2

和操作节点 Op3 的运算结果。图 4.1 右图中，操作节点 Op5 依赖操作节点 Op0，Op1 和 Op3 的运算结果。图 4.1 左图经过节点融合优化，将操作节点 Op2 和 Op4 融合成右图中的操作节点 Op5，同时维持了 Op2 和 Op4 的数据依赖关系。未进行节点融合前完成图 4.1 左图所有的操作需要 5 个操作，进行节点融合后需要进行的操作只需要 4 个，数据依赖链的长度由 2 变为 1。

中间表示中的操作或节点在降级处理过程中降级为目标平台指令集中的指令，所以可使用指令在目标平台上运行时所花费的时钟周期数作为代价，对节点融合前后的代价进行评估。使用 cost(OpN)来表示 OpN 操作降级为目标平台指令集中的指令后执行指令所花费时钟周期数，则图 4.1 左图的总代价 costBefore(融合前的总代价)为 cost(Op0) + cost(Op1) + cost(Op2) + cost(Op3) + cost(Op4)，图 4.1 右图的总代价 costAfter(融合后的总代价)为 cost(Op0) + cost(Op1) + cost(Op3) + cost(Op5)，其中 Op5 是由节点 Op2 和节点 Op4 融合而来，节点 Op5 的代价满足不等式 max(cost(Op2), cost(Op4)) <= cost(Op5) < cost(Op2) + cost(Op4)，即融合后节点所需的时钟周期数小于节点融合前所需的总时钟周期数，同时融合后节点所需的时钟周期数不小于融合前节点所需的时钟周期数中的最大值。costBefore - costAfter = cost(Op4) + cost(Op2) - cost(Op5)，从而可知 costBefore > costAfter，即进行节点融合后其所花费的代价小于节点融合前的代价，所以可以进行节点融合优化。图 4.1 描述的是较理想情况下的节点融合优化，需要指出的是：并非在任何情况下，节点融合后的代价都小于节点融合前的代价，如图 4.2 所示。

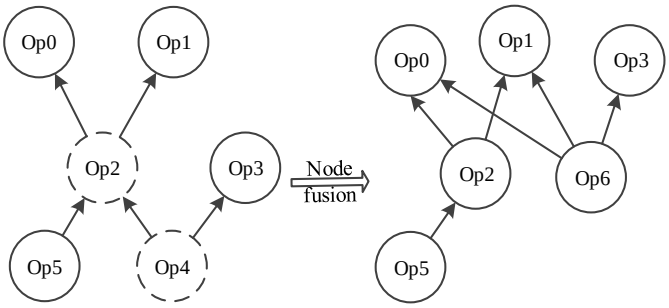


图 4.2 激进的节点融合

图 4.2 的左图中 Op2 和 Op4 是需要进行节点融合的两个节点，进行节点融合后转化为图 4.2 右图中的 Op6 节点。图 4.2 与图 4.1 不同的是节点 Op2 多了一个使用者节点 Op5，该节点的操作依赖 Op2 操作的结果，所以进行节点融合后，

Op2 仍将保留。图 4.2 左图的代价 costBefore(融合前的总代价)为 cost(Op0) + cost(Op1) + cost(Op2) + cost(Op3) + cost(Op4) + cost(Op5)，图 4.2 右图的代价 costAfter(融合后的总代价)为 cost(Op0) + cost(Op1) + cost(Op3) + cost(Op2) + cost(Op6) + cost(Op5)，其中 max(cost(Op2), cost(Op4)) <= cost(Op6) < cost(Op2) + cost(Op4)。 costBefore – costAfter= cost(Op4) - cost(Op6)，从而 costBefore - costAfter 小于或等于 0，即进行节点融合后其代价不小于节点融合前的代价，进行节点融合后不能带来性能的提升，所以不宜进行节点融合优化。

从图 4.1 和图 4.2 中的代价计算过程可以发现，代价的计算只与融合的节点相关，例如图 4.1 中融合前的 Op2 节点、Op4 节点及融合后的 Op5 节点，图 4.2 中融合前的 Op2 节点、Op4 节点及融合后的 Op6 节点。在图 4.1 中 costBefore - costAfter = cost(Op4) + cost(Op2) - cost(Op5)，在图 4.2 中 costBefore – costAfter= cost(Op4) - cost(Op6)。两者的区别在于图 4.2 中的 Op2 节点存在除节点 Op4 外的其他依赖节点，即节点 Op5，所以进行节点融合后节点 Op2 将不被删除，costAfter 中仍需加上 Op2 的代价。若进行节点融合前与节点融合相关的节点有 n 个，记为 Op1，Op2，…，Opn，进行节点融合后生成了 m 个节点，记为 FOp1，Fop2，…，FOpm 代价评估的模型如下等式所示。

$$costBefore = cost(Op1) + cost(Op2) + \cdots + cost(Opn) \tag{4.1}$$

$$costAfter = cost(FOp1) + cost(FOp2) + \cdots + cost(FOpm) \tag{4.2}$$

$$costBefore - costAfter = \begin{cases} > 0, & \text{进行节点融合} \\ \leq 0, & \text{不进行节点融合} \end{cases} \tag{4.3}$$

通过等式（4.1）、（4.2）、（4.3）对融合前后的子图进行代价评估，若节点融合前的代价大于融合后的代价，则进行节点融合优化，否则不进行节点融合优化。

4.2 中间表示阶段节点融合优化方法

中间表示阶段的节点融合是指在生成 LLVM IR 或者对 LLVM IR 进行优化的过程中进行的节点融合操作，其输入为 LLVM IR，输出也为 LLVM IR。SSA 形式的 LLVM IR 要求每个变量只被分配一次，并且每个变量在使用之前需被定义。在 SSA 形式的 IR 语句中，其输入操作数为使用点(use 点)，其结果值为定义点(def 点)。SSA 形式的这种特点，使得其变量的 use-def 链是显式的，所以通过 IR 语句的操作数可以找到其定义点（def 点）。以表达式 a×b＋c 的中间表示

代码为例说明 SSA 形式中间表示的特点，如图 4.3 所示。

```
%0 = load double, double* @a, align 8
%1 = load double, double* @b, align 8
%mul = fmul double %0, %1
%2 = load double, double* @c, align 8
%add = fadd double %mul, %2
```

图 4.3 表达式 a×b＋c 的 IR 代码

图 4.3 中，以%开头的变量都是虚拟寄存器属于局部变量，以@开头的变量为全局变量，赋值运算符右边的 load，fmul，fadd 分别为装载操作、浮点乘和浮点加[45]操作。比如 IR 语句%mul = fmul double %0, %1 中%mul 为 fmul 操作的结果值，其结果值为 double 类型，由于%mul 位于赋值运算符的左边，所以该处为定义点（def 点）。Fmul 操作的两个输入操作数分别为%0，%1，其值为 double 类型，%0，%1 位于赋值运算符的右边，所以该处为%0，%1 的 use 点。根据 SSA 形式 IR 的特点，只需找到输出操作数为%0，%1 的 IR 语句就可以知道%0，%1 的定义点分别为%0 = load double, double* @a, align 8 和%1 = load double, double* @b, align 8，其中 align 8 表示该 load 操作需要按 8 字节对齐访存。

中间表示阶段的节点融合优化，由于不需要目标平台后端的信息，所以属于平台无关的优化。在中间表示阶段有两处可以进行节点融合优化：一个是在前端将抽象语法树转化为 LLVM IR 时进行节点融合优化；另一个使用 LLVM 提供的平台无关的优化工具 OPT 进行节点融合优化。OPT 是一个通用优化器，其中包含了多个优化 Pass，用来对 LLVM IR 进行平台无关的优化，比如公共子表达式消除（Early CSE），内联优化（Inline）等。中间表示阶段的 LLVM IR 进行浮点乘加节点融合的前后变化如图 4.4 所示。

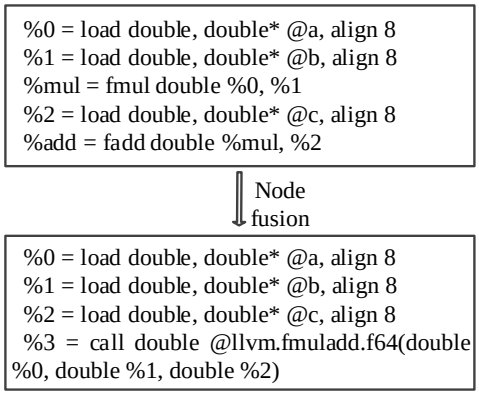


图 4.4 中间表示层节点融合

在图 4.4 上图是节点融合前的 LLVM IR，下图是节点融合后的 LLVM IR。从图 4.4 上图可以看到语句%mul=fmul double %0, %1 和%add = fadd double %mul, %2 融合为图 4.4 下图语句%3 = call double @llvm.fmuladd.f64(double %0, double %1, double %2)形式的内建函数调用，该内建函数在后续的中间表示降级过程中逐步转化为相应的浮点乘加指令。中间表示阶段的节点融合算法如表 4.1 所示。

表 4.1 算法 1 中间表示阶段节点融合

输入：(*Op*)/*待融合节点*/

输出：(*fusionOp*)/*融合后节点*/

1. *costBefore* ← *cost*(*Op*)
2. *costAfter* ← 0
3. *fusionOp* ← *Op*
4. **case** *Op.getOpcode*() **in**
5. Opcode1)
6. *Op0* ← *Op.getOperand*(0)
7. **if** *Op0.getOpcode*() is Opcode2 **then**
8. *fNode* ← *emitBuiltin1*(*Op*, *Op0*)
9. **if** *hasOneUse*(*Op0*) is true **then**
10. *costBefore* ← *costBefore* + *cost*(*Op0*)
11. **end if**
12. *costAfter* ← *cost*(*fNode*)
13. **if** *costBefore* > *costAfter* **then**
14. *fusionOp* ← *fNode*
15. **end if**
16. **end if**
17. **break**
18. OpcodeN)/*其他节点融合处理*/
19. **end case**

表 4.1 中的算法 1 是中间表示阶段进行节点融合的算法伪代码，该算法将每一条语句视为一个节点，对节点试图进行节点融合优化，hasOneUse 方法用来判

断节点是否只有一个使用者（依赖该节点的运行结果的节点）。通过节点的getOpcode 方法获取其操作码，将节点的操作码作为 case 的匹配值，从而达到对不同节点进行不同的节点融合处理。当节点的数据依赖关系满足匹配 case 的融合条件且融合后节点的代价 costAfter 小于节点融合前的代价 costBefore 时，通过发射内建函数调用的方式，完成节点融合优化，否则不进行节点融合优化。生成的内建函数调用 IR 将在 LLVM IR 的降级过程中被降级为目标平台相关的指令。

中间表示层的节点融合利用中间表示具有显式 use-def 链的特点，通过发射内建调用或操作节点的方式进行节点融合优化。但是由于中间表示应具有平台无关的属性，所以在该层进行平台相关的节点融合优化所生成的中间表示可能最终不按照预期生成相应的机器指令。如果生成的融合节点不被目标平台后端所支持，则进行节点融合后的中间表示将影响后端的兼容性，而且破坏了中间表示应具有的平台无关的特性。

4.3 DAG 合并阶段节点融合优化方法

从 LLVM IR 到 MIR 的过程，是从平台无关的中间表示向平台相关的中间表示进行降级过程。在这一降级的过程中，LLVM IR 转换成 DAG，利用图的匹配算法完成 LLVM IR 的降级处理。

DAG 降级阶段的节点融合优化是在 DAG 降级的过程中进行的节点融合处理，如在图 2.14 中有 DAG 合并，DAG 合并 1， DAG 合并 2 等，这些 DAG 合并进行了一系列的平台无关和平台相关的节点融合优化。完成这些节点融合优化，不仅可以简化 DAG，而且还可以提高目标平台代码的执行效率，减少运行时间，从而达到充分挖掘目标平台指令集潜力的目的。

由于 LLVM 良好的封装和高度抽象的特性，LLVM 提供了实现目标平台相关的节点融合优化的虚函数接口 PerformDAGCombine，LLVM 的目标平台后端只需继承 TargetLowering 类，实现 PerformDAGCombine 方法，就可以从通用的 DAG 合并流程，借助 TLI 信息进行自定义的节点融合优化，从而实现平台无关以及平台相关的节点融合。实现 DAG 阶段的节点融合优化的算法伪代码如表 4.2 所示。

表 4.2 算法 2 DAG 阶段的节点融合

输入：(*N*, *DCI*) /*待融合节点，DAG 信息*/

输出：(*fusionNode*)/*融合节点*/

```
1. DAG ← DCI.DAG
2. costBefore ← cost(N)
3. costAfter ← 0
4. fusionNode ← N
5. case N.getOpcode() in
6. Opcode1 )
7.     Op0 ← N.getOperand(0)
8.     if Op0.getOpcode() is Opcode2 then
9.         fNode ← DAG.getNode(Opcode3, N, Op0)
10.        if hasOneUse(Op0) is true then
11.            costBefore ← costBefore + cost(Op0)
12.        end if
13.        costAfter ← cost(fNode)
14.        if costBefore > costAfter then
15.            fusionNode ← fNode
16.        end if
17.    end if
18. break
19. OpcodeN ) /*其他 Opcode 处理*/
20. end case
```

表 4.2 中算法 2 中 DAG 阶段的节点融合优化算法与算法 1 中的中间表示阶段的节点融合优化算法有些类似，但算法 1 进行的平台无关的节点融合优化，算法 2 既可以进行平台相关的节点融合优化(发射平台相关的节点)，又可以进行平台无关的节点融合优化(发射平台无关的节点)，这种差异是由其所处不同的编译阶段所引起的。算法 2 通过传入的待融合节点 N 的 Opcode 来区分不同的节点融合处理流程，若节点存在相应的 case 处理，则继续判断节点的数据依赖关系是否满足要求，然后计算融合前后的的代价 costBefore 和 costAfter，若融合后的代价 costAfter 要小于融合前的代价 costBefore 则通过返回融合节点 fNode 完成节

点融合优化，否则返回原节点 N 不进行节点融合优化。

DAG 降级阶段的节点融合优化是在 DAG 降级的过程中进行的节点融合处理，它利用 DAG 有向无环的特点，根据拓扑排序算法对每一个节点进行处理。在这一降级过程中不仅提供平台相关的优化，而且提供平台无关的优化。本阶段的节点融合优化是通过实现 LLVM 提供的虚函数接口，根据待融合节点的数据依赖关系来完成节点融合优化，该优化的代码只在目标平台后端进行修改，既不影响 LLVM 的版本升级，也不影响中间表示的兼容性，而且还可以进行多次节点融合优化。

4.4 指令选择阶段节点融合优化方法

除了 4.4 小节提到的在 DAG 合并阶段进行节点融合优化之外，还可以通过 Pattern 匹配的方式进行节点的融合优化。虽然该过程属于 DAG 降级阶段，但是又不同于 DAG 合并阶段中的节点融合优化。Pattern 是 td 文件中的一个类，td 文件是 LLVM 用来定义后端指令集、寄存器、调用约定等的一套抽象描述。对应的 td 文件及其功能如表 4.3 所示。

表 4.3 后端架构相关的 td 文件

文件名	描述
<target>.td	定义机器的特征
<target>InstrFormats.td	定义指令格式
<target>RegisterInfo.td	定义寄存器和寄存器类
<target>CallingConv.td	定义寄存器的调用约定
<target>InstrInfo.td	定义指令、模板和格式
<target>Schedule.td	定义指令流水

表 4.3 中的 td 文件描述了目标平台后端的特征信息，比如指令集，寄存器描述等。 Pattern 类的定义如图 4.5 所示。从图 4.5 代码中可以看到指令 Pattern 的两个输入 patternToMatch 和 resultInstrs 都是 dag 类型的变量，其中 PatternToMatch 是要匹配的图（即模板输入），ResultInstrs 定义了从 PatternToMatch 转换成的结果图（即模板输出）。通过指令 Pattern 可以完成节点的融合优化，该过程是在指令选择阶段完成的。以浮点乘加指令的 Pattern 举例说明如图 4.6 所示。

```
class Pattern<dag patternToMatch, list<dag> resultInstrs> {
    dag         PatternToMatch  = patternToMatch;
    list<dag>   ResultInstrs    = resultInstrs;
    list<Predicate> Predicates  = [];
    int         AddedComplexity = 0;
    }
class Pat<dag pattern, dag result> : Pattern<pattern, [result]>;
```

图 4.5 Pattern 类定义

```
def : Pat<(fadd (fmul F8RC:$A, F8RC:$B), F8RC:$C),
          (FMA F8RC:$A, F8RC:$B, F8RC:$C)>;
def : Pat<(fadd F8RC:$C , (fmul F8RC:$A, F8RC:$B)),
          (FMA F8RC:$A, F8RC:$B, F8RC:$C)>;
def : Pat<(fneg (fsub (fneg F8RC:$C), (fmul F8RC:$A, F8RC:$B))),
          (FMA F8RC:$A, F8RC:$B, F8RC:$C)>;
def : Pat<(fneg (fsub (fneg (fmul F8RC:$A, F8RC:$B)), F8RC:$C)),
          (FMA F8RC:$A, F8RC:$B, F8RC:$C)>;
```

图 4.6 乘加 pattern 定义

图 4.6 中其中 fadd 是浮点加节点，fmul 是浮点乘节点，FMA 是浮点乘加节点，fneg 是浮点取负节点，fsub 是浮点减节点。$A，$B，$C 分别表示变量 A，B，C。以上四个 Pattern 分别对应的表达式为 A×B＋C， C＋A×B ，－(－C－A×B) ，－(－A×B－C)，其中 A，B，C 都为浮点类型的变量，F8RC 表示双精度浮点类型寄存器。这四个表达式都可以化简为 A×B＋C 的形式，所以都可以通过浮点乘加指令来完成。Pattern 匹配通过在<Target>InstrInfo.td 中编写相应的 Pattern 模板来完成节点融合，Pattern 的匹配过程如图 4.7 所示。

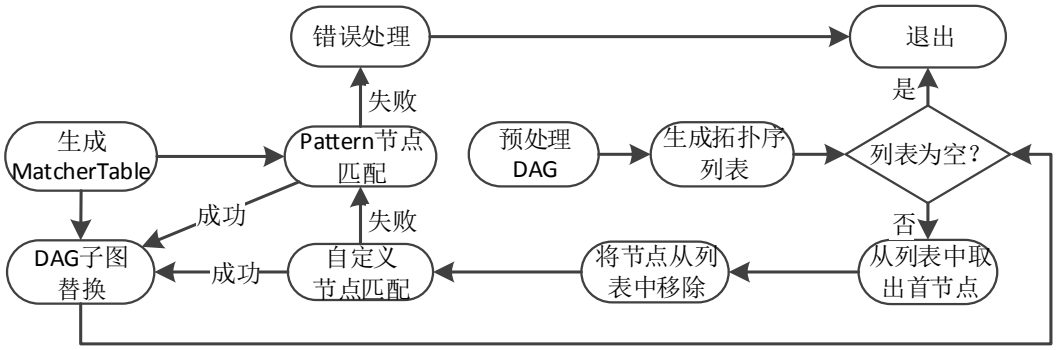


图 4.7 Pattern 匹配流程

图 4.7 中 Pattern 的匹配流程为先取得待匹配节点，判断节点偏移表是否为

空，如果为空则根据 Pattern 匹配表（MatcherTable）信息构建偏移表，否则从偏移表中取出节点的偏移值；如果偏移值为零则进行错误处理，将显示无法选择节点错误，然后退出编译程序；否则检查节点类型及约束，检查以该节点为根的 Pattern 子图能否和 DAG 相匹配，若成功则进行 DAG 子图替换将相匹配的子图（PatternToMatch）替换为目标子图（ResultInstrs），否则从 MatcherTable 中获取以该节点为根节点的下一 Pattern 子图的偏移，然后判断偏移是否为零，进行循环处理。

Pattern 匹配方式的节点融合优化，是利用 Tablegen 工具将 td 格式的 Pattern 代码生成 MatcherTable，然后利用通用的 Pattern 匹配流程完成节点融合优化。该种方式只需添加 Pattern 代码到目标平台相关的指令 td 文件中，不影响 LLVM 的版本升级和其他后端，但是该种方式由于无法有效获取子图节点的数据流信息，所以不能进行节点融合前后的代价评估，只能尽最大努力进行节点融合优化，因此可能生成质量较差的目标平台代码。

4.5 各阶段节点融合优化方法优缺点比较

节点融合优化在 LLVM 编译流程的各个阶段都可进行，本文选取中间表示阶段，DAG 合并阶段以及指令选择阶段进行了研究和实现。这三个阶段都可对节点进行融合优化处理，不同阶段处理的效果和方式各有差异，以下将进行各阶段节点融合优化方法的优缺点比较。

中间表示阶段。静态单赋值形式的中间表示具有显式的 use-def 链，通过 use-def 链可以得到变量之间的数据流关系，利用数据流关系可以进行节点融合优化。中间表示阶段的节点融合优化是将匹配上的多条中间表示语句通过内建函数的方式发射到 LLVM IR 中，替换原先的中间表示语句。LLVM IR 应具有的一个属性是平台无关，所以在中间表示阶段进行的节点融合属于平台无关优化。当然进行平台相关的优化是可行的，但这势必将破坏中间表示应具有的平台无关的属性，导致生成的中间表示仅适合于特定的平台后端，同时影响其他平台无关优化，比如常量传播和公共子表达式消除等。此外在中间表示层进行的平台相关的节点融合优化生成的融合节点如果不被平台后端所支持，则该融合节点将在 DAG 降级阶段被拆分为多个节点，导致此节点融合优化失效，反而增加了编译时间，不利于编译器性能提升。

DAG 合并阶段。DAG 合并阶段由于可以获取到 TLI 信息，所以既可以进行平台相关的节点融合优化，又可以进行平台无关的节点融合优化。该阶段在保留了前端丰富的中间表示信息的基础上，结合后端平台指令集的信息，进行针对性的节点融合优化。在此阶段，由于拥有目标平台后端的指令集信息和架构相关的信息，所以可以保证融合后的节点转化为相应的指令。不仅如此，在该阶段添加的融合优化代码只是针对目标平台后端的修改，不影响其他后端，也不影响 LLVM 的版本升级。此外，在中间表示降级过程中将多次进行 DAG 合并处理，所以在 DAG 合并阶段的节点融合优化亦将多次进行。

指令选择阶段。指令选择阶段完成的功能是将所有的平台无关的节点降级为平台相关的节点，经过指令选择阶段后 DAG 只能包含平台相关的节点，否则将导致编译器报错。指令选择阶段的节点融合优化是根据后端指令集描述文件中定义的 Pattern 进行模板匹配来实现的。通过 Pattern 匹配的方式进行节点的融合优化，利用了通用的 Pattern 匹配流程，所以实现起来极其简单，只需在后端指令集描述文件中编写节点融合 Pattern 即可。但是 Pattern 匹配的方式无法获取各个节点之间完整的数据流关系，导致无法进行节点融合前后的代价评估，只能尽最大努力进行节点融合，所以可能导致优化后的代码质量下降。

因为中间表示阶段、DAG 合并阶段、指令选择阶段的节点融合优化实现于目标平台无关的 LLVM IR 降级为目标平台相关的 MIR 的处理流程的不同阶段，所以在 LLVM 编译器中应用这三种优化不仅不会导致冲突，而且能够结合三种优化方法优点起到最大的优化效果。

4.6 本章小结

本章对 LLVM 中的节点融合优化进行介绍，在中间表示阶段、DAG 合并阶段和指令选择阶段进行了实现。中间表示阶段适合进行目标平台无关的节点融合优化，DAG 合并阶段和指令选择阶段适合目标平台无关和目标平台相关的节点融合优化，指令选择阶段的节点融合优化无法对融合前后的代价进行有效的评估只能进行尽最大努力的优化。

5 实验与分析

5.1 实验概述

本文采用国产平台申威 1621 处理器为实验平台，以 CLANG 和 FLANG[46] 为前端编译器，LLVM 为后端编译器，编译器的版本为 7.0，所使用的测试集为 SPEC CPU2006。在上述配置下，进行两个实验：不进行节点融合优化和进行节点融合优化的实验。通过这两个实验的结果对比，评估节点融合的优化效果。SPEC CPU2006 基准测试集包含 13 道定点程序，18 道浮点程序，由 C，C++以及 Fortran 编程语言编写。本实验对其中 29 道程序进行测试，编译优化选择为-O3 -static，规模为 train 规模，单进程运行，每道程序测试三次，然后取平均运行时间。

5.2 实验结果及分析

在上述配置及基准测试集下，测试结果如表 5.1 所示。

表 5.1 节点融合实验结果

程序名称	编程语言	结果校验	节点融合优化前(s)	节点融合优化后(s)	加速比
400.perlbench	C	√	114.17	103.09	1.11
401.bzip2	C	√	164.99	142.76	1.16
403.gcc	C	√	3.98	3.67	1.09
410.bwaves	F77	√	129.04	114.83	1.12
416.games	Fortran	√	426.97	340.07	1.26
429.mcf	C	√	83.84	83.41	1.01
433.milc	C	√	65.09	64.74	1.01
434.zeusmp	F77	√	118.55	105.90	1.12
435.gromacs	C、Fortran	√	522.21	327.91	1.59
436.cactusADM	F90、C	√	44.85	32.07	1.40
437.leslie3d	F90	√	268.94	238.38	1.13
444.namd	C++	√	38.3	31.76	1.21

445.gobmk	C	√	417.18	371.47	1.12
447.dealII	C++	√	156.50	143.77	1.09
450.soplex	C++	√	23.77	20.30	1.17
453.povray	C++	√	29.46	26.57	1.11
454.calculix	F90、C	√	3.75	3.24	1.16
456.hmmer	C	√	170.90	168.91	1.01
458.sjeng	C	√	535.37	489.92	1.09
459.GemsFDTD	F90	√	195.26	173.28	1.13
462.libquantum	C	√	6.67	5.79	1.51
464.h264ref	C	√	274.19	232.26	1.18
465.tonto	F95	√	1678.97	1567.78	1.07
470.lbm	C	√	191.30	144.46	1.32
471.omnetpp	C++	√	278.21	267.13	1.04
473.astar	C++	√	315.63	283.98	1.11
481.wrf	F90、C	√	432.90	547.45	0.79
482.sphinx3	C	√	31.73	31.23	1.02
483.xalancbmk	C++	√	570.09	548.46	1.04
平均加速比					1.13

从表 5.1 可知所有程序均运行正确且通过结果校验。从表 5.1 可以看出节点融合优化对大部分程序具有明显的加速效果，加速比最高的程序有 1.59 倍的加速，平均加速比为 1.13。通过对加速效果最好的程序 435.gromacs 进行性能分析发现程序执行过程的时间消耗有 70%集中在函数 inl1130 上，进一步分析函数 inl1130 的代码发现其中包含多个开方函数和浮点乘加表达式。通过节点融合优化可以对开方函数和浮点乘加表达式进行节点融合优化，使其转换为针对申威平台的特有指令，从而减少程序的执行时间，提高程序运行效率。程序能够加速的主要原因是通过节点融合减少了冗余指令和用特有指令来替代原有的指令提升了程序的执行效率，使系统的资源得到了更加充分的利用。

虽然多数程序都具有正加速的效果，但是其中也有个别程序的加速比小于 1.00，比如程序 481.wrf 的加速比为 0.79。程序未进行节点融合优化前其运行(进程从开始执行直到结束的时间)时间为 432.90 秒，进行节点融合优化后其运行时间增加到 547.45 秒，节点融合优化显著地增加了程序的运行时间。通过对程序

进一步分析发现，程序执行的用户时间(进程花费在用户模式中的 CPU 时间)在进行节点融合前后差异不大，分别为 354.23 秒和 331.82 秒。从用户时间的减少可以看出节点融合优化具有加速效果。程序 481.wrf 性能下降的主要原因是进行节点融合优化后，该程序的系统时间(进程花费在内核模式中的 CPU 时间，代表在内核中执行系统调用所花费的时间)显著增加，由未进行节点融合前的 78.21 秒增加到 215.63 秒，是未进行节点融合优化所花系统时间的 3 倍。系统时间的大幅度增加导致了 481.wrf 程序进行节点融合优化后性能下降，说明其中存在不合理的节点融合优化。

5.3 本章小结

在申威平台 1621 处理器上，使用 SPEC CPU2006 基准测试集对节点融合优化进行测试，对测试的 29 道基准程序的平均加速比为 1.13，说明节点融合优化具有一定的加速效果，且达到预期。通过对倒加速的程序进行分析发现，所实现的节点融合优化方法仍存在缺陷，需要进行深入分析。

6 总结与展望

6.1 总结

本文对申威平台 LLVM 编译器中的窥孔优化进行了研究和分析，介绍了经典的窥孔优化技术：冗余指令消除、控制流优化、代数简化和特有指令替换。针对 LLVM 编译器中的窥孔优化存在过度优化和申威平台特有指令未能充分利用的问题，提出了节点融合优化方法，在申威平台 1621 处理器上进行了节点融合优化的实现，通过使用基准测试集 SPEC CPU2006 进行测试，实验结果表明节点融合优化达到了预期的效果。主要工作如下；

（1）研究和分析了 LLVM 中的窥孔优化技术，主要包括：冗余指令删除、控制流优化、代数简化和特有指令替换。针对窥孔优化存在激进优化的情况提出了节点融合优化方法，通过对融合前后节点的代价进行评估，根据代价的大小决定是否进行节点融合优化。

（2）结合申威平台指令集特点，在中间表示节点、DAG 合并阶段和指令选择阶段实现了节点融合优化，对三个阶段的节点融合优化进行了优缺点分析。在中间表示阶段进行融合，虽然有比较丰富的中间表示信息，但是无法获知后端是否支持该种优化，所以存在融合节点被拆分为多个节点的风险；DAG 合并阶段的节点融合，该阶段既有丰富的中间表示信息，又有后端的指令集信息，既可以进行平台无关的优化，又可以进行平台相关的融合优化，还能进行多次节点融合优化；指令选择阶段的 Pattern 匹配节点融合，该阶段通过通用的 Pattern 匹配流程进行 DAG 子图的融合优化，由于无法获取 DAG 子图节点之间完整的数据依赖关系，导致无法进行融合前后的代价评估，只能尽最大努力进行节点融合，所以容易进行过度优化，反而不利于性能的提升。

（3）在申威平台 1621 处理器上，使用 SPEC CPU2006 为基准测试集进行了实验，实验结果表明节点融合优化有利于提高编译器性能、减少程序运行时间，优化后最大加速比为 1.59，平均加速比为 1.13。且在申威平台 LLVM 编译器中已得到实际应用。

6.2 未来展望

由于时间和实验环境的限制，本文的研究工作仍然存在不足，一些问题需要进一步的深入研究。由于本文的工作是在申威平台上进行的，申威平台上关于LLVM 编译器的移植与开发仍然处于初步阶段，所以该平台上的 LLVM 编译器功能并不十分完善，比如缺少向量化功能的支持。对程序进行自动向量化是加速程序运行，提升程序执行效率的方法。由于缺少向量化功能的支持，所以本文实验中的节点融合优化只是针对标量类型以及标量指令。因此，本文中的节点融合优化方法对于向量类型及向量指令是否仍具有加速效果及有多大程度的性能提升，需要进一步研究。

实验中的节点融合优化方法虽然对多数基准测试程序具有加速效果，但是仍然存在倒加速的程序。对于产生倒加速的原因需要进一步的研究与分析，同时由于时间限制，在申威平台上的 LLVM 编译器中进行的节点融合优化仍然不完善，所以下一步的工作主要是分析 SPEC CPU2006 部分程序倒加速的原因，完善申威平台 LLVM 编译器的节点融合优化。

参考文献

[1] 丁禹,逯志安,焦建伟.2014 年网络空间安全问题综述与展望[J].信息安全与通信保密,2015(02):16-21.
[2] 阿不都克里木·玉素甫,王亮亮.基于自主可控平台的 FFMPEG 在线视频转换系统[J].计算机与现代化,2020(01):81-84+116.
[3] 邓中翰. 如何实现自主创新和自主可控？[N]. 中国科学报,2019-12-19(006).
[4] 夏斌. 让“中国制造”自主可控[N]. 解放日报,2019-11-22(009).
[5] 倪光南.加快构建本质安全、自主可控的工业互联网[J].网信军民融合,2019(10):22-23.
[6] 朱金凤.国产化与自主可控[J].电气时代,2019(07):3.
[7] 本刊编辑部.微议华为中兴事件[J].中国信息安全,2012(11):54.
[8] 杨广文,赵文来,丁楠,段芳.“神威•太湖之光”及其应用系统[J].科学,2017,69(03):12-16+2.
[9] 唐琳.国产平台下基于 SSH 框架的 Web 系统开发研究[A]. 中国高科技产业化研究会智能信息处理产业化分会.第九届全国信号和智能信息处理与应用学术会议专刊[C].中国高科技产业化研究会智能信息处理产业化分会:中国高科技产业化研究会,2015:6.
[10] Lattner C. LLVM and Clang: Next generation compiler technology[C] //The BSD conference. 2008, 5.
[11] 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.
[12] 龚丹,苏小红,王甜甜.Clang 编译平台优势分析[J].智能计算机与应用,2017,7(03):188-190+193.
[13] Guobin Y E. Getting to know the LLVM compiler[D]. Master’s thesis, The University of Edinburgh, 2011.
[14] Knoop, Jens, Rüthing, Oliver, Steffen, Bernhard. Partial dead code elimination[C]// Acm Sigplan Conference on Programming Language Design & Implementation. :147-158.
[15] 魏帅, 赵荣彩, 姚远. 面向 SLP 的多重循环向量化[J]. 软件学报, 2012(07):87-98.
[16] McKeeman W M. Peephole optimization[J]. Communications of the ACM, 1965, 8(7): 443-444.
[17] Jack W. Davidson,Christopher W. Fraser. The Design and Application of a Retargetable Peephole Optimizer[J]. ACM Transactions on Programming Languages and Systems (TOPLAS),1980,2(2).
[18] Wulf, William, etal. The Design of an optimizing compiler[M]. 1975.
[19] Tanenbaum A S , Staveren H V , Stevenson J W . Using Peephole Optimization on Intermediate Code[J]. ACM Transactions on Programming Languages and Systems, 1982, 4(1):21-36.
[20] Callahan D, Cooper K D, Kennedy K, et al. Interprocedural constant propagation[C]//ACM SIGPLAN Notices. ACM, 1986, 21(7): 152-161.

[21] Cocke J. Global common subexpression elimination[J]. ACM Sigplan Notices, 1970, 5(7): 20-24.
[22] Spradling C D. SPEC CPU2006 benchmark tools[J]. ACM SIGARCH Computer Architecture News, 2007, 35(1): 130-134.
[23] 谭捷,庞建民,单征,岳峰,卢帅兵,戴涛.二进制翻译中冗余指令优化算法[J].计算机研究与发展,2017,54(09):1931-1944.
[24] 葛吴超,周亦敏.基于 ARM9 体系架构的编译优化研究[J].电子科技,2016,29(09):106-110.
[25] 刘颂超. 嵌入式交叉汇编器的设计与优化[D].湖南大学,2012.
[26] Lee, Juneyoung, Chung-Kil Hur, and Nuno P. Lopes. "AliveInLean: A Verified LLVM Peephole Optimization Verifier." International Conference on Computer Aided Verification. Springer, Cham, 2019.
[27] Lopes N P, Menendez D, Nagarakatte S, et al. Provably correct peephole optimizations with alive[J]. ACM SIGPLAN Notices, 2015, 50(6): 22-32.
[28] Leroy X. Formal verification of a realistic compiler[J]. Communications of the ACM, 2009, 52(7): 107-115.
[29] Mullen E, Zuniga D, Tatlock Z, et al. Verified peephole optimizations for CompCert[C]//ACM SIGPLAN Notices. ACM, 2016, 51(6): 448-461.
[30] Zhao J, Nagarakatte S, Martin M M K, et al. Formalizing the LLVM intermediate representation for verified program transformations[C]//Acm sigplan notices. ACM, 2012, 47(1): 427-440.
[31] van Oirschot J. Extending tree pattern matching for application to peephole optimizations[J].2019 master thesis.
[32] 施光源.基于国产申威处理器的自主可控产品实践之路[J].网络空间安全,2018,9(07):78-81.
[33] Sunway, http://www.swcpu.cn/show-130-261-1.html
[34] Sunway, http://www.swcpu.cn/show-176-262-1.html
[35] Sunway, http://www.swcpu.cn/show-190-254-1.html
[36] Sunway, http://www.swcpu.cn/uploadfile/2018/0709/20180709033115724.pdf
[37] 刘坚. 编译原理基础[M]. 2002.
[38] Aho A, RaviSethi, Ullman J. Compilers:Principles, Techniques, and Tools[M]// Compilers, principles, techniques, and tools /. 2002.
[39] Guobin Y E. Getting to know the LLVM compiler[D]. Master's thesis, The University of Edinburgh, 2011.
[40] Lattner C. Introduction to the llvm compiler infrastructure[C] //Itanium conference and expo. 2006.
[41] Pandey M, Sarda S. LLVM cookbook[M]. Packt Publishing Ltd, 2015.
[42] Lopes B C. Getting Started with LLVM Core Libraries[M]. 2014.
[43] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin. Formal verification of SSA-based optimizations for LLVM[C]// Acm Sigplan Conference on Programming Language Design &

Implementation. ACM, 2013.

[44] Hokenek E, Montoye R K, Cook P W. Second-generation RISC floating point with multiply-add fused[J]. IEEE Journal of Solid-State Circuits, 1990, 25(5): 1207-1213.

[45] Lattner C, Adve V. The LLVM instruction set and compilation strategy[J]. CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS, 2002.

[46] Osmialowski P. How The Flang Frontend Works: Introduction to the interior of the Open-Source Fortran frontend for LLVM[C]//Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC. ACM, 2017: 1.

个人简历、在学期间发表的学术论文与研究成果

一、个人简历

胡浩，1994 年 11 月 03 日生，湖北咸宁人。

2017 年 7 月毕业于武汉工程大学计算机科学与技术学院，软件工程专业，工学学士。

2020 年 7 月毕业于郑州大学信息工程学院，计算机科学与技术专业，工学硕士。

二、在学期间发表的学术论文

[1] 胡浩, 沈莉, 周清雷, 巩令钦. 基于 LLVM 编译器的节点融合优化方法[J]. 计算机科学（已录用）

三、研究成果

[1] 国家重点研发计划“网络空间拟态防御技术机制研究”，项目编号 2016YFB0800100。

[2] 国家自然科学基金面上项目：“计算树逻辑模型检测的 DNA 计算方法研究”，项目编号 61572444

[3] 专利：基于异构平台的自适应节点融合编译优化方法，申请号：201910885756.1（申请中）。

[4] 专利：基于异构平台的常量数据访问优化方法，申请号：201910886036.7（申请中）。

[5] 软件著作权：计算树逻辑模型检测的 DNA 计算仿真平台 V1.0，登记号：2018SR668512。

致谢

时光荏苒，转眼间三年时间已经飞逝，经历的分分秒秒仍历历在目，留下的有收获也有失落，有开心也有悲伤。这些酸甜苦辣的经历是人成长的不竭动力，要学会感恩，不仅要感谢那些让你快乐、开心的人，也要感谢那些给你“制造”困难的人，正是这些人使你不断成长、不断强大。回首往事，唯有感激，感谢你们让我经历人生种种，感谢你们让我不断成长，唯有感激，才会倍加珍惜。

首先，我要感谢我的导师周清雷教授。三年来，导师渊博的专业知识、诲人不倦的高尚师德、精益求精的治学态度和工作作风、平易近人的人格魅力对我影响深远。学业上，给我以精心指导、循循善诱；生活上，给予无微不至的关怀；思想上，要求我积极上进。虽然只有短短三年时间，但是给予我终生受益无穷的收获。

其次，真诚的感谢无锡江南计算所的沈莉老师。在无锡江南计算所实习的一年时间，感谢沈老师在工程、学术研究上的引导与提点。在我学术论文的撰写过程中提供了许多帮助，提出了许多建设性的修改意见，使我顺利完成了学术论文的撰写。

感谢实验室的各位同门，感谢你们的帮助、支持和理解，正是有你们作为我坚强的后盾，我才能克服一个个困难和疑惑，顺利完成本文的撰写，顺利完成学业。

感谢我的家人，是你们在我求学近二十年的路上给予我积极的肯定，你们的帮助、支持和无私的奉献，让我感受到家人的温暖与幸福，这也是我一直以来不断努力向前的动力。

最后再次感谢我生命中遇到的所有人，是你们让我不断成长，让我体验到生活的千姿百态。衷心的祝福你们，愿你们开心快乐。人生路长，我将继续前行。