分类号：_____ 密级：_____

UDC：_____ 编号：_____

工学硕士学位论文

基于 LLVM 的迭代间数据重用优化研究

硕士研究生：刘　刚

指 导 教 师：吴艳霞　副教授

学科、专业：计算机系统结构

论文主审人：武俊鹏　教授

哈尔滨工程大学

2014 年 3 月

分类号：_____ 密级：_____

U D C ：_____ 编号：_____

工学硕士学位论文

基于 LLVM 的迭代间数据重用优化研究

硕 士 研 究 生 ： 刘刚

指 导 教 师 ： 吴艳霞　副教授

学 位 级 别 ： 工学硕士

学 科 、 专 业 ： 计算机系统结构

所 在 单 位 ： 计算机科学与技术学院

论文提交日期 ： 2014 年 3 月

论文答辩日期 ： 2014 年 3 月

学位授予单位 ： 哈尔滨工程大学

Classified Index:

U.D.C:

A Dissertation for the Degree of M. Eng

Research on LLVM Inter-iteration Data Reuse Optimization

Candidate: Liu Gang

Supervisor: Ass.Prof. Wu Yanxia

Academic Degree Applied for: Master of Engineering

Specialty: Computer Architecture

Date of Submission: Mar. , 2014

Date of Oral Examination: Mar. , 2014

University: Harbin Engineering University

哈尔滨工程大学
学位论文原创性声明

本人郑重声明：本论文的所有工作，是在导师的指导下，由作者本人独立完成的。有关观点、方法、数据和文献的引用已在文中指出，并与参考文献相对应。除文中已注明引用的内容外，本论文不包含任何其他个人或集体已经公开发表的作品成果。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。本人完全意识到本声明的法律结果由本人承担。

作者（签字）：

日期：　　年　月　日

哈尔滨工程大学
学位论文授权使用声明

本人完全了解学校保护知识产权的有关规定，即研究生在校攻读学位期间论文工作的知识产权属于哈尔滨工程大学。哈尔滨工程大学有权保留并向国家有关部门或机构送交论文的复印件。本人允许哈尔滨工程大学将论文的部分或全部内容编入有关数据库进行检索，可采用影印、缩印或扫描等复制手段保存和汇编本学位论文，可以公布论文的全部内容。同时本人保证毕业后结合学位论文研究课题再撰写的论文一律注明作者第一署名单位为哈尔滨工程大学。涉密学位论文待解密后适用本声明。

本论文（□在授予学位后即可　□在授予学位 12 个月后　□解密后）由哈尔滨工程大学送交有关部门进行保存、汇编等。

作者（签字）：　　导师（签字）：

日期：　　年　月　日　　年　月　日

摘　　要

随着程序设计语言和计算机系统结构的发展，如何对新的语言特性和系统结构进行优化成为现代编译设计的核心。由于“存储墙问题”，循环中数组访问的时间占程序总执行时间的比重通常都比较大。因此，如何更好地优化循环中的数组访问是提升编译性能的关键问题之一。

LLVM 是目前比较流行的编译器架构，由于在代码优化上的良好表现，越来越多的编译优化研究转向于此框架。但是在处理循环中数组访问时，LLVM 所采取的归纳变量优化造成了数组引用计算过于复杂的问题。为了解决此问题，本文提出了一种迭代间数据重用优化算法，以弥补 LLVM 在循环优化中对数组引用计算的不足。

迭代间数据重用优化算法基于 LLVM 的循环规范化和循环优化处理，用于简化循环中数组访问的地址运算。在该算法的实现过程中，本文结合了归纳变量优化和数组引用标量替换两种优化技术。即在一次循环迭代中，首先对数组引用进行标量替换，将地址变量转化为可识别的归纳变量，使得寄存器分配对数组引用有效；然后对寻址操作进行强度削弱，针对不同体系结构设定不同的归纳变量自增步长，将地址计算转化为简单的标量运算；最后将地址计算结果保存至新的寄存器，以便用于下一次迭代，从而增加数据在循环迭代之间的重用。该优化算法可以有效降低循环体内地址计算的复杂度，再加上 LLVM 自身的优势，使整个程序的性能得到提升。

实验结果表明，在 LLVM 中加入本文提出的迭代间数据重用优化算法后，与 GCC 以及添加优化之前的 LLVM 的对比中，从汇编代码和程序执行时间两个方面验证了该优化算法对于提升程序性能的作用。

关键字：LLVM；循环迭代；数据重用；归纳变量；标量替换

Abstract

With the development of programming languages and computer architectures, optimization for new language features and new architectures is becoming the heart of modern compiler design. Since “Memory Wall” problem, time spent on array access in loops accounted for the proportion of total execution time is usually large. Therefore optimization on array access in loops has been one of the key problems for improving compilers’ performance.

LLVM is a popular optimizing compiler framework currently. More and more researches on compiling optimization turn to this framework for its good performance on code optimization. When dealing with array access in loops, traditional induction variable optimization is used in LLVM, which results in the complication of array reference. To solve this problem, an inter-iteration data reuse optimization algorithm is proposed in this thesis, which makes up for the deficiencies of LLVM in array access.

The inter-iteration data reuse optimization algorithm is used to simplify the calculation of array reference in loops based on LLVM loop canonicalization and optimization. This algorithm combines induction variable optimization and scalar replacement of array reference. At each iteration, array reference is substituted with scalar, which converts address variables to recognizable induction variables, and makes register allocation effective for array reference. Then strength reduction is implemented on induction variables by setting stride for different architecture, which converts address calculations to simple scalar ones. At last, the result is stored in a register for the next iteration, so that data can be reused between iterations. This algorithm can effectively reduce the complexity of address computation in loops, and improve the whole program’s performance, coupled with the advantage of LLVM itself.

The experiment results show that both assembly code and execution time are better when adding the inter-iteration data reuse optimization presented in the thesis, compared with the result generated by GCC or LLVM without this optimization.

Key words: LLVM, Data Reuse, Induction Variable, Scalar Replacement

目　　录

第 1 章 绪论

1.1 研究目的及意义

从 20 世纪 50 年代中期以来，编译器设计一直是计算机研究和开发中的活跃主题[1]。在编译器设计中，“优化”是指编译器为了生成更加高效的代码而做的工作[2]。不同的编译器在代码优化阶段所做的工作相差很大，那些在优化阶段花费大量时间，代码优化做得好的编译器，就是所谓的“优化编译器”。随着程序设计语言和计算机系统结构的发展，编译器所做的优化变得更加重要，而且更加复杂。之所以变得更加复杂，是因为处理器体系结构变得更加复杂，也有了更多改进代码执行方式的机会；之所以变得更加重要，是因为并行计算的发展要求更多实质性的优化，否则它们的性能将会呈数量级地下降。随着多核技术的日益流行，所有的编译器都将面临充分利用多核处理器的优势的问题[2]。

程序设计领域的一个共识是“程序 90%的时间花费在执行 10%的代码上”，并且绝大多数的执行时间是与循环相关的。因此对编译器而言，最重要的优化也是与循环有关的优化[1]。如果循环每一次迭代中都存在一些不必要的计算，那么整个循环执行下来，会造成程序巨大的性能损失。由局部性原理[3]知道，当程序具有良好的局部性（locality）的时候，就可以在迭代中将数据进行重用(reuse)。如果一次迭代中可以重用上一次迭代的部分数据，就会缩短本次迭代的执行时间。因此在循环，特别是内层循环中的一点优化，都会对循环乃至整个程序带来性能上的巨大提升。由此可见，如何对迭代间的数据进行重用是十分重要且有意义的。

LLVM（Low Level Virtual Machine）是由伊利诺伊大学厄巴纳-香槟分校（University of Illinois at Urbana-Champaign）的 Vikram Adve 与 Chris Lattner 与 2000 年研究发展而来的优秀开源编译框架，具有模块化、可复用、全时优化等特点[4]。从研究之初起，LLVM 就被设计作为一套模块化、可重用的库，并定义了明确的接口[5]，因此，具备了良好的适应性，可移植性。经过了十年的发展，LLVM 从一个学术研究项目发展成为了包含 C、C++编译器的通用编译框架。它成功的关键在于其良好的性能和适应能力，这得益于 LLVM 独特的设计和实现。。

目前，对于 X86，ARM，MIPS，SPARC 等主流平台 LLVM 已经能够提供很好的支持，但是作为一个仍在不断完善的编译架构，LLVM 尚存在一些可用于提高目标代码质量的体系结构相关或无关的优化策略未实现。本文就是在 LLVM 编译架构中，完善其对

循环迭代间数据重用优化的支持。

编译器作为一种高级语言到机器语言的映射工具，对它的优化不仅可以提高某一类输入程序的性能，使优化后的程序具有良好的可移植性，而且可以减轻程序员的压力。LLVM 结构清晰，高度的模块化和可重用的特点，不仅有利于分析中间代码的转化策略，也便于在此基础上编写可重用的优化代码。通过 LLVM 提供的工具链，可以方便地完成中间代码的优化过程，进而获得高性能、代码空间小的机器指令。

1.2 国内外研究现状

编译器设计和优化已经拥有很长的历史，并且是相对成熟的计算技术。但是随着语言设计、目标机体系结构的不断变化和程序复杂性的提高，编译器内部也在不对地进行改进，关于它的研究从来没有停止过。在编译理论的发展过程中，文献[2]被誉为编译领域的“龙书”，是编译理论的集大成之作。文献[1]是有关编译技术的另一部经典著作，被誉为“鲸书”，它侧重于阐述编译器的后端以及编译优化的理论，是介绍编译优化技术最完备的一部著作。这两部书不仅为许多编译领域的研究提供了理论基础，也是本文中绝大多数理论依据的主要来源。

1.2.1 数据重用理论研究现状

现如今，“存储墙问题”（Memory Wall）[6]日益突出，并成为提升计算速度的第一难题。在现有的计算机体系结构下，如何利用编译优化技术充分挖掘数据的重用性、并行性是缓解“存储墙问题”的一个重要手段。

1991 年，Wolf 和 Lam[3]在编程语言设计与实现大会上提出了经典的数据重用和局部性分析理论，它利用距离向量的概念来描述迭代空间中的数据依赖关系。该理论适用于分析循环中具有仿射数组下标的数据重用问题，对于编译优化技术[7][8]以及其他多个研究领域产生了重要影响。在该理论的影响下，如何通过优化来提高数据局部性成为编译领域研究的热点。其中，存储层次优化就是利用局部性原理应对存储墙问题，减小存储层次间速度差距负面影响的一项重要应用。编译器可以通过依赖分析，获取有用的存储访问信息，并依此对程序进行优化变换，提高数据的局部性和重用性。这种方法对于处理嵌套循环中的数组访问最为有效，因为通常这一部分的执行时间占程序总执行时间的比重非常大，而且由于数组在内存中的连续存储，提高其空间和时间上的局部性，能有效减少访存，大量缩短程序的执行时间。

Li W[9]提出了一种提高并行性和数据局部性的优化方法。该方法首先求出循环迭代

中访问同一块内存单元的的重用向量、重用距离，最终求得一种重用矩阵（data reuse matrix），然后通过循环变换矩阵来对重用矩阵进行高度削弱（height reduction），并通过循环铺砌（loop tiling）来对重用矩阵进行宽度削弱（width reduction），从而达到优化 cache 局部性的目的。

Chen Ding[10]提出一种通过 profiling 预测程序局部性的方法，该方法主要利用了重用距离近似分析、模式识别和基于距离的采样三种技术对全局的数据重用性进行研究，暴露出一些短距离重用（short-distance reuse）或者局部控制流（local control flow）所没有体现的全局模式。当基准测试满足所给出的有效性和可预测性的分析条件下，取得了良好的性能提升效果。

关于数据重用性和局部性卓有成效的研究还有很多，国内这方面的研究成果主要来源于以杨学军院士为带头人的国防科技大学编译课题组。他们长期从事并行处理技术以及编译优化技术的研究，文献[11] [12] [13]就是他们众多研究成果中的三篇。他们将经典串行数据重用理论应用到并行领域，在存储层次上，他们的优化主要针对的是多核处理器的 cache。

上述这些研究成果大多是针对存储层次中的 cache 这一层，通过提高数据局部性、增加 cache 命中率来提高计算性能。但是 cache 是由硬件维护的，编译器无法控制，不是本文所研究的重点。现在考虑最坏的情况，即每次访存都是和内存进行数据交换，而访问内存和访问寄存器的延迟差别是数百倍，这里的优化空间就非常明显了。程序开发者都希望尽可能地重用每一次迭代中计算过的数据——“计算任何事情都不要超过一次”。为实现迭代间的数据重用，本研究所采用的优化方法主要包括：归纳变量优化（Induction Variable Optimization: IVO）和数组引用标量替换等技术。这两种优化也都是研究比较早，相对比较成熟的技术。其中数组引用标量替换是存储层次优化技术的一种，可以看作是局部性理论的一类应用。

归纳变量（Induction Variable: IV）是指那些在循环的每次迭代中都增加或减少固定数值，或者与其他归纳变量成一定线性关系的变量。归纳变量优化是最重要的一类循环优化，主要包括归纳变量的识别、强度削弱、归纳变量删除、线性函数测试替换等技术。大多数循环优化是以归纳变量的识别为基础的，经典的算法是由 Aho[2]提出来的，该算法以循环的达定义（reaching definition）信息和循环不变量信息为输入，输出一个由归纳变量的（变量名，步长，初始值）组成的三元域。其他重要的关于循环归纳变量的识别方法包括 Cytron[14]和 Wolfe[15]的基于 SSA 和 FUD（Factored use-def）链的线性归纳变量识别，Haghighat[16]的基于符号差分的方法，Gerlek[17]使用递归系统（recurrence

system），vanEngelen[18]使用递归链（chains of recurrence）的方法对循环中的归纳变量最重要的一项优化就是强度削弱（Strength Reduction），早在 1981 年，Allen[19]等人就在论文中对强度削弱优化进行了描述，他们的研究被认为是传统的强度削弱，但是由于他们提出的算法过于复杂，很难在编译器中实现。Cooper[20]给出的强度削弱优化算法（OSR）被认为是传统算法的良好替代，该算法是基于静态单赋值（SSA）的，更容易理解和实现，并且时间复杂度在最差情况下也和传统算法有相同的数量级。该算法也已经被应用到某些编译器中。上述两项研究被认为是强度削弱优化最具代表性的成果。

尽管已经有很长的研究历史，但关于归纳变量优化的研究从未停止。由于它对于循环优化有如此重要的作用，因此被添加到几乎所有主流的编译架构中。另外，围绕归纳变量，衍生出其他一些研究分支，如国内的有：周雷[21]提出了一种在处理循环时，基于符号运算的归纳变量识别与约化技术。刘杰[22]基于归纳变量优化，提出了一种解决符号执行方法在处理循环时路径爆炸问题的方法。

目前关于归纳变量优化的研究仍在继续，如何让现代编译器为自身架构更好地提供对这项优化的支持是各个芯片制造商关注的焦点。ARM 中国[23]在 2013 年 HelloGCC 技术研讨会[24]上展示了他们在 ARM 架构下关于 GCC 归纳变量优化的研究，该研究为他们的芯片性能带来了不小的提升。由此可见，现有的编译优化技术并非能够在某款编译器上完全实现，或者对某种平台架构完全适用，针对特定的编译器和平台架构，可以通过改进编译优化算法或者定制平台相关的优化，达到提升程序性能的效果。

1.2.2 LLVM 优化架构研究现状

得益于编译优化理论的发展和成熟，使得 LLVM 可以在很好的理论成果基础上建立起来，并成为一款优秀的工业级编译器。由于在系统结构设计上的优越性，LLVM 于 2012 年被 ACM（美国计算机协会）授予系统软件大奖[25]。越来越多的公司或团队投入到对 LLVM 的使用和开发中。这其中包括 LLVM 最主要的开发和使用者 Apple Inc.[26]，Sun 微系统实验室的 Parfait[27]，Adobe 公司的 AVM2[28]，Intel 的 OpenCL 的研究[29], NVIDIA 的 OpenCL 运行时编译器[30]。除了上述这些公司使用和开发 LLVM 外，还涌现出许多基于 LLVM 的开源项目，如 Objective Modula-2 Project[31]用 LLVM 对 w/ObjC 编译器提供运行时支持；PyPy Project[32]实现 Python 解释器；Faust 信号处理语言使用 LLVM JIT 作为运行时的代码生成器[33]。在国内的研究中，不少高校有关于 LLVM 的研究课题，包括哈尔滨工业大学实现 ARCA3 处理器的移植[34]；上海交通大学实现了 ARM 后端的移植[35]；中南大学实现了 Nios II 处理器后端移植及优化[36]；哈尔滨工程大学有基于

LLVM 的可重构编译技术的相关研究[37]。可见，LLVM 在国内也掀起了一股研究热潮。

上述这些研究成果，大多是利用 LLVM 作为工具，来实现自身工具链的搭建或者一部分功能模块。下面这些研究成果主要是针对 LLVM 编译器的优化。Ghassan Shobaki[38]等人提出了一种指令调度算法，该算法很好地平衡了最大化指令并行和最小化寄存器压力的矛盾，对于大基本块（包含很多指令的基本块）十分有效。他们将这种算法整合进 LLVM，并且在 SPEC CPU2006 上取得了非常不错的优化效果。Jablin[39]等人为了解放 CPU-GPU 异构加速的性能，着眼于 CPU 和 GPU 的通信优化，提出了一种 CPU-GPU 通信管理系统，该系统包含一个运行时库和一系列编译器变换共同优化 CPU-GPU 通信，该方法不依赖于编译时静态分析的强度和程序员的注释。Jianzhou Zhao[40]等人提出了 Vellvm （verified LLVM），该框架给出了 LLVM 中间表示以及基于中间表示变换的形式化证明。Vellvm 提供了 LLVM IR 语义的形式化表述，包括它的类型系统、它的静态单赋值性质。他们从 Coq（一个形式化证明系统）形式语义中提取出一个解释器，来执行 LLVM 测试套件（test suite），从而可以和 LLVM 的参考实现相比。

但是这些优化中并没有考虑编译器对迭代间数据重用的支持。本文主要研究的迭代间数据重用技术目前在 LLVM 编译器中还没有实现。为了能直观地看出研究结果，论文的实验结果将主要以 LLVM 中间表示和汇编的形式描述，实验平台采用的是 ARM。

1.3 论文的主要工作

本论文研究的是在 LLVM 编译框架中，实现对循环迭代间的数据重用优化技术的支持。论文的主要工作如下：

（1）研究现有的编译优化理论，以及它们在编译器中的应用情况。

（2）深入探讨 LLVM 编译框架，包括 LLVM 的中间表示，LLVM 的 Pass 方法，对循环优化算法的支持等方面。

（3）分析 LLVM 中循环优化技术的具体实现，特别是归纳变量优化和数组引用标量替换优化，找出其实现策略中的不足。

（4）设计实现了一种改进的数组引用访问策略，完善了 LLVM 对迭代间数据重用技术的支持。

（5）通过 LLVM 提供的 API，在此编译架构中，进行了优化遍的设计与实现。

1.4 论文的组织结构

本文在研究循环优化的基础上，分析了 LLVM 循环迭代优化算法的实现，包括归纳

变量优化、标量替换优化。针对其中的不足，提出了迭代间数据重用优化算法，该优化基于 LLVM 的中间表示，针对某一类数值应用，这类应用使用数组作为数据结构，并按照规则的模式访问这些数据结构。从而实现了一种迭代间数据重用的优化技术，并做了相关的实验验证。全文共分五个章节，组织结构如下：

第 1 章：介绍了选题的背景、目的和意义；分析了循环编译优化技术，以及 LLVM 代码优化的研究现状；给出了本文的主要工作。

第 2 章:介绍了与课题相关的背景知识。包括与循环相关的编译器优化技术和 LLVM 优化编译框架的相关知识。为分析 LLVM 的循环优化，以及提出的算法提供了理论依据。

第 3 章：详细分析了 LLVM 的循环优化算法，特别是归纳变量优化和标量优化两种技术。结合实际例程，分析得出 LLVM 的实现过程中的不足，以便提出改进的方案。

第 4 章：针对 LLVM 循环优化的不足，通过数据依赖分析，提出了迭代间数据重用优化算法，并详细描述了算法的设计与实现过程。

第 5 章：对测试用例进行实验验证，并对测试结果进行分析和对比。

最后是对全文的结论，总结了通过归纳变量优化和数组引用标量替换实现迭代间数据重用的研究经验，并对接下来的工作前景进行展望。

第 2 章 相关背景知识介绍

2.1 编译器优化

编译器是一个将高级语言编写的程序转换成能在一台计算机上执行的等价目标代码或机器语言程序的软件系统[1]。

对于大型软件项目，一个重要的设计原则就是用较小的、功能不同的模块构成大的程序，每个模块尽可能简单，以方便设计实现和维护。现代编译器设计就趋向于这种模块化、清晰化的结构，这对于编译器的开发和优化都具有非常重要的意义。一个典型的现代编译器工作模式可以看作是一种如图2.1所示的流水线（pipeline）。

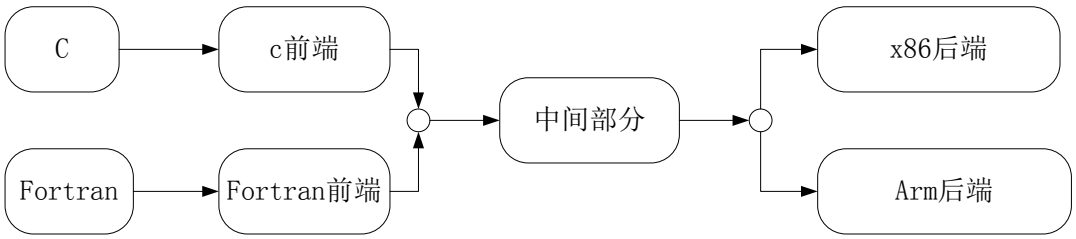

图2.1 典型的编译器结构

从图2.1中可以看出，编译器可以分为前端、中间组件和后端三部分，它们构成了一个对源程序处理的流水线。编译器前端部分负责将高级语言转化为中间表示（Intermediate Representation）；中间组件负责对中间表示进行分析和优化；编译器的后端将中间表示转化为目标代码或机器语言程序。

这个模型看起来很简单，似乎设计一个编译器并不是特别困难的事情，但是为了能使编译器产生高质量的目标代码需要做的工作就非常大了。在从高级语言到机器语言的转化过程中，常常需要对源程序、中间表示，以及后端代码进行等价变换；或者制定一些策略，使得所产生的目标代码能够更合理地利用目标计算机的资源，这就是编译器代码优化需要做的事情。

优化是高级编译器设计的核心，经过几十年的发展，现代编译器的一些优化技术已经比较成熟。下面定义一种“激进型”优化编译器[1]，“激进”的意思是在不破坏程序正确性的前提下尽可能地优化程序性能，图2.2几乎涵盖了现在已知的大部分重要的编译优化技术。

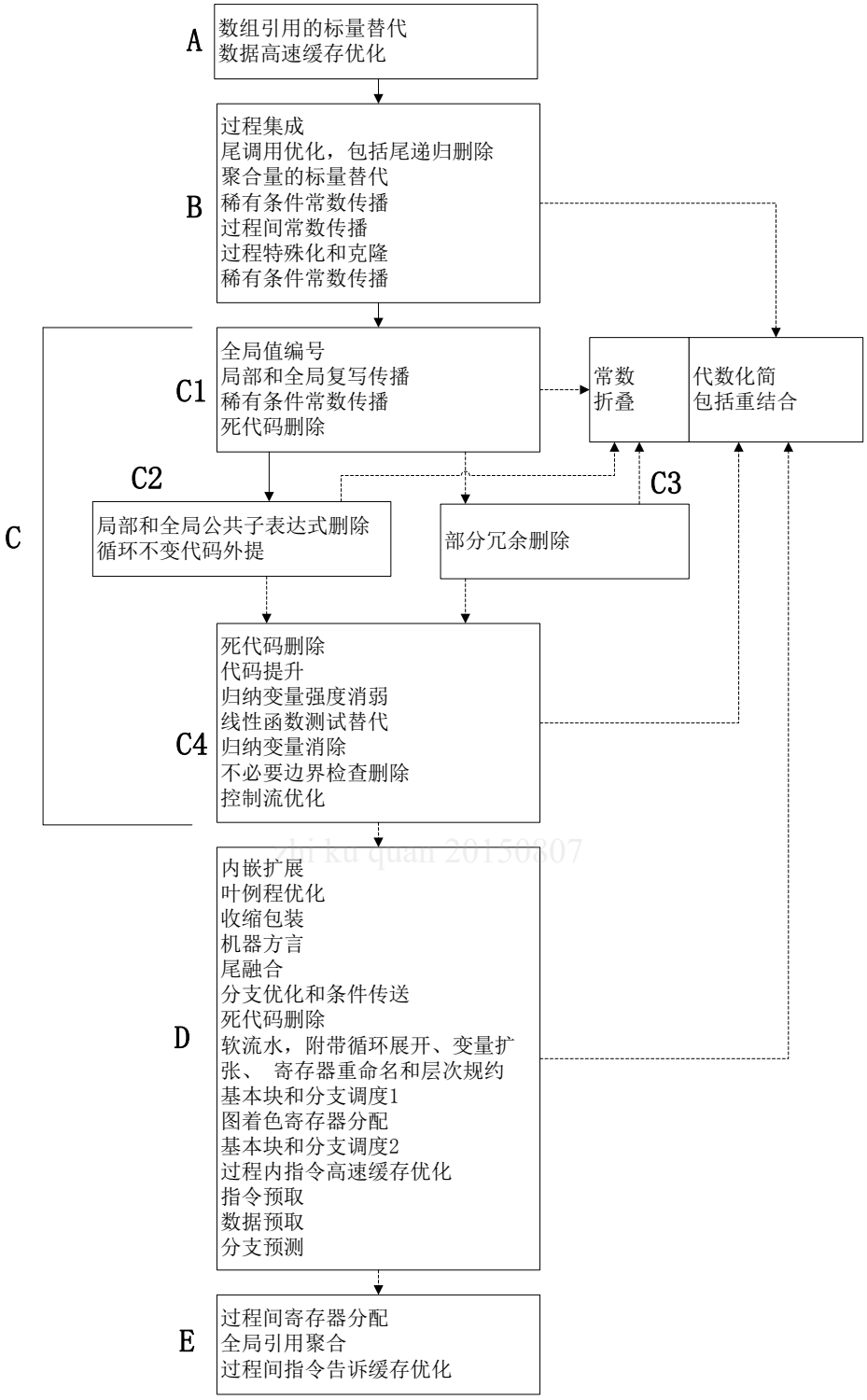

图2.2 “激进”优化的位置

按照各种优化的作用阶段大致可以将它们按照图 2.2 中给出的顺序进行排列和分组。其中各个字母表示在编译器中的作用位置，如表 2.1 所示。

表 2.1 各种优化在编译器中的作用位置

A 中的优化最好在编译前期施加于高级中间表示，两种优化都需要依赖关系分析提供的信息。
B 中的优化通常是在早期阶段针对中级中间表示进行的。
C 中的这些优化最好在中级或低级中间表示上进行，并且要在 B 之后，按照 C1 框中的顺序更适合对 SSA 形式的代码执行复写传播。C2 和 C3 中的优化可以减少之后循环优化需要处理的代码量。C4 框中在进行死代码删除后，进行了归纳变量优化，不可达代码删除等重要优化。
D 中的优化主要作用于低级中间代码，而且大多是平台相关的。
E 中的优化主要在链接时进行，所以它们施加于可重定位的目标代码。

图 2.2 中展示了如此多的优化技术，虽然编译器希望能充分发挥他们的作用，改善程序的性能，但也完全可能对于某些特定的输入，优化没有改善性能，甚至出现性能降低的情况。事实上，一个具体的优化是否能改善程序性能，在多数情况下是不可判定的。在进行优化时，通常希望尽可能地改善代码，但这种改善必须是安全的/保守的，也就是说必须以正确性为前提。

一般地，对一个过程应该实施图 2.2 中的那些优化最重要的判别标准通常是程序的执行速度，有些时候优化后的程序空间也非常重要，但多数情况下是速度尽可能快比空间尽可能小要更重要，这也是判断优化效果的主要参考依据。在这些优化中，其中有一些是提高程序性能的关键技术，如循环优化、寄存器分配和指令调度。当然，哪些优化最重要是因特定的程序结构和目标平台架构而异的。有一些特定的优化是对某些体系结构更重要，例如，全局寄存器分配对 RISC 架构的机器而言非常重要，因为它们提供了大量的寄存器，但这种优化对于只有少量寄存器的 CISC 机器重要性就不那么大。

那种相对于获得的性能而言，代价太大的，或者只对程序中几乎不执行的部分有作用的优化，一般不值得去做。大部分程序的执行时间都花在循环上，因此循环通常是最值得优化的对象。一般在优化程序之前都需要先运行它，并对它做运行时的剖面分析（Profiling），以找出程序的热点（程序执行时间最多的部分），然后利用得到的信息来指导优化。对于具体应该采用这么多不同优化中的哪些种，很难给出一种统一的方法，但大部分优化包括下面的三个过程：

首先，要寻找一种变换匹配模式。也就是说，要给出优化是针对哪种模式的，没有哪种优化是普适的。例如循环合并（loop fusion），将两个迭代空间相同的循环体合并为一个循环，就要求两个循环必须有相同的循环边界，并且在已合并的循环中没有因为第一个循环中的指令依赖于第二个循环中的指令引起的流依赖、反依赖和输出依赖，满足这些匹配模式，才能进行相应的变换。

其次，要确保这种变换对于匹配实例是安全的。这种检查很有必要，因为编译器必须是语义保守的，满足匹配模式所进行的变换必须是正确的，但是这一类证明算法通常比较复杂。

最后做代码变换，更新程序代码。在选好的匹配模式上，实施所提出的变换，完成优化。

这三个过程是实现一种优化的基本步骤，在下一节中，将具体来探讨 LLVM 的架构，以及在这款编译器上是如何实现某一种优化的。

2.2 LLVM 优化编译框架

LLVM 是一款如 2.1 节中所介绍的高度模块化、结构清晰的工业级编译框架。它对源程序的处理过程可以用如图 2.3 所示的流水线描述，整个处理过程由许多遍（Pass）组成。遍是编译器功能的基本单位，每个遍可以看成是流水线的一个阶段，由 Pass Manager 统一组织，并按照依赖关系进行调度，实现特有的功能。

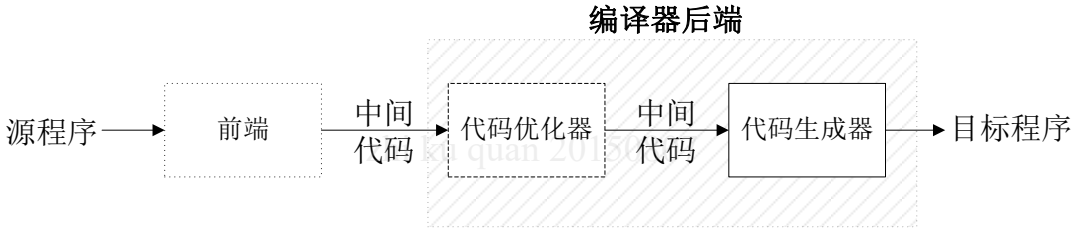

图2.3 LLVM编译器结构

LLVM 的设计目标是：分析和优化需要尽可能提早，因为编译时优化可以整体上减少代码变换-重建-执行的周期，也可以减少链接时的工作；所有的过程间分析和优化（IPA/IPO）要是开放的，也就是说它们应该作用于编译时，并且是基于库的；分析和优化用同一种 IR（Intermediate Representation 的缩写），即在编译时和链接时均可使用。IR 的设计是实现以上三点目标的关键。

LLVM 的 IR 有 3 种形式：汇编形式（assembly）、二进制形式（bitcode），以及内存中的二进制形式（in-memory）。这三种表示是等价的。为了方便描述，本文中所讨论的 IR 均是可读的汇编形式，约定以.ll 作为文件名后缀。

IR 是语言无关（Language-independent）和平台无关（Target-Independent）的，即不依赖于前端的高级语言以及后端平台，可以说是一种低级平台无关语言。它的特性包括：类 RISC 三地址指令；使用无限虚拟寄存器的 SSA 形式；低层次控制流结构；Load/Store 指令使用类型指针。IR 语言中的变量可以是下面的类型：

（1）全局变量：以@开头，形如@var = common global i32 0, align 4。

（2）局部变量：以%开头，形如%2 = load i32* %1, align 4。

（3）函数参数：形如 define i32 @fact(i32 %n)。

LLVM 提供了 opt 工具用于修改 IR。Opt 可以读取 IR 输入文件（包括汇编形式和二进制形式），然后按照用户指定的顺序对其执行特定 LLVM 遍。命令行语法是

opt < input.ir [arguments] [pass-name] > output.ir

这个工具非常强大，可以以任意顺序执行 LLVM 的任意遍，同时可以利用-stats 参数选项打印遍的统计信息；用-time-passes 统计并输出每个遍的执行时间。例如可以使用下面的命令输出对 bar.ll 这个 IR 模块添加 O3 优化选项时经过了哪些遍，这些遍的执行时间是多少：

opt < bar.ll -O3 -stats -time-passes > bar-O3.ll

通过此命令可以得到表 2.2 的信息：

表 2.2 LLVM O3 优化遍执行信息

Statistics Collected

……

1 indvars - Number of loop exit tests replaced

6 loop-simplify - Number of pre-header or exit blocks inserted

6 loop-unswitch - Total number of instructions analyzed

1 memdep - Number of block queries that were completely cached

……

Pass execution timing report

Total Execution Time: 0.0010 seconds (0.0136 wall clock)

---User Time---	--User+System--	---Wall Time---	--- Name ---
0.0000 (0.0%)	0.0000 (0.0%)	0.0049 (36.3%)	Induction Variable Simplification
0.0010 (100.0%)	0.0010 (100.0%)	0.0048 (35.3%)	Print module to stderr
0.0000 (0.0%)	0.0000 (0.0%)	0.0013 (9.5%)	Early CSE

……

LLVM IR Parsing

Total Execution Time: 0.0000 seconds (0.0077 wall clock)

---Wall Time---	--- Name ---
0.0077 (100.0%)	Parse IR
0.0077 (100.0%)	Total

表 2.2 中包括了三类信息：程序所经历的遍的统计信息、遍执行时间报告、IR 分析的时间统计。例如，从遍的统计信息这栏里可以看出，在 loop-simplify 优化遍被执行的时候，插入了 6 个 pre-header 结点和 exit block，关于它们的介绍详见 3.2 节。

LLVM 通过遍的组合实现编译器算法，通常可以把遍分为以下三类：分析遍（analysis pass）、优化遍（optimization pass）、规范化/平凡化遍（normalization/canonicalize pass）。

优化遍和规范化遍都需要对代码结构进行修改，因此可统称为变换（Transform pass）。这三种遍各自有不同的职能，以循环计算外提（Loop hoisting）优化为例。把循环不变运算外提是优化遍的工作，但是它需要事先知道“循环”和“循环不变表达式”在哪里，是什么，这便是分析遍的工作。分析遍会检测到程序中的循环，以及循环不变表达式，也就是说上面这个过程自动建立起一条分析-优化的流水线。但是循环分为很多种，如 do-while 循环，while 循环，for 循环。如果分析、优化一种循环，其他类型的循环该如何处理？相比较对每一类循环都做特别的处理而言，用规范遍将他们转化为事先定义的规范形式要容易得多，这就是规范化遍所做的事情。所以在执行循环不变量外提的时候必须告诉 pass manager 先执行循环规范化，这样可以识别更多的循环，优化效果也更好。综合这种思想，编译器的算法设计可以总结为下面的过程：

（1）提出并分析待解决的问题。

（2）给出该问题具体实例。

（3）归纳出问题的一般情形。

（4）确定输入的格式，也就是做好匹配模式。

（5）指定要做的分析遍。

（6）设计优化遍。

（7）证明变换的语义保守性。

（8）在一般情形下提高算法性能。

（9）通过添加规范化遍来提升算法的有效性。

LLVM 的程序结构由大到小依次包括模块（Module）、函数（Function）、基本块（Basic Block）、指令（Instruction）。模块包含函数和全局变量，是编译、分析、优化的单位；函数包括基本块和参数，基本和 C 语言中函数的概念相对应；基本块是一些指令序列，并且每个基本块是以控制流指令结束；指令是由操作码和操作数向量构成，指令结果以及所有的操作数都需要被指定类型。其中，基本块是一种对中间代码的划分方法，它满足下列约束条件的最大指令序列：

（1）控制流只能从基本块的第一条指令进入该块。也就是说，不存在能够直接跳转到基本块中间的转移指令。

（2）除了基本块的最后一条指令，控制流在离开基本块之前不会停机或跳转。

根据对程序处理的粒度（模块、函数、基本块、指令）的不同，可以把 LLVM 的遍划分为图 2.4 所示的类型。每一种遍都用来处理程序中的一类特定元素。

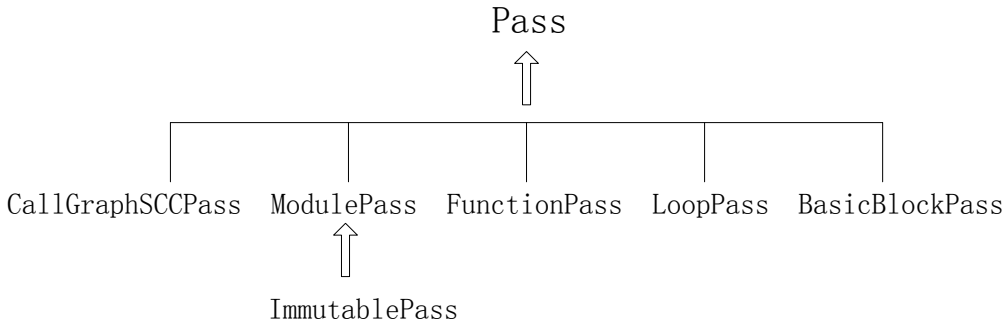

图2.4 LLVM中遍的类型

在图 2.4 的各类遍中，ImmutablePass 用于编译器配置；CallGraphSCCPass 可以后续访问 CallGraph SCC（顺序控制图）；ModulePass 用于访问某一个模块；FunctionPass 用作访问某一个函数；LoopPass 专门用于访问循环嵌套；BasicBlockPass 用于访问某一个基本块。针对不同的功能需求，应该选择实现最简便、效果最佳的遍。这些遍协同合作，实现编译器功能。

2.3 归纳变量优化

循环迭代间数据重用模型主要针对的是循环中具有仿射（线性）下标的数组访问。如果程序具备了良好的可重用性，那么被访问的数据及其相邻的数据很可能在短时间内会被再次访问，这使得即将访问的数据很有可能位于离处理器较近、且较快的存储器中（比如寄存器），所以处理器很有可能从较快的存储器中读取数据，从而能极大地减少存储访问延迟，提高程序的执行性能[13]。在大多数 RISC 平台上，访存指令（Load/Sore）的代价要明显大于其他指令，相比而言，寄存器的存取速度则要快得多，因此，如何高效实用寄存器可能是优化一个程序时要处理的最重要的问题。

为实现循环中的数据在每次迭代之间可以进行重用，本位所采用的编译优化技术主要是归纳变量优化以及数组引用的标量替换。

归纳变量优化（IVO）是循环优化中一种十分重要的方法，关于这种优化方法的理论也已经比较完善，但是对于不同的编译器，对这种优化技术的支持程度，以及采用的方法不尽相同。本节将重点介绍这种技术所包含的一些过程和方法。

2.3.1 识别归纳变量

执行归纳变量优化的第一步就是要识别归纳变量。为了便于识别，通常将归纳变量可分为基本归纳变量（biv）与依赖归纳变量（div）。基本归纳变量是指在每次迭代中显

式地增加或减少一个固定常量的变量。依赖归纳变量是指对基本归纳变量呈现一定依赖关系的变量，这种关系通常是线性的。例如图 2.5 中所示的归纳变量是：*i*，&*a*[*i*]，2**i*，2**i*+5。其中，变量 *i* 是基本归纳变量，&*a*[*i*]，2**i*，2**i*+5 是依赖归纳变量。

```
int a[100], i;
for (i =0; i < 100; i++)
  a[i] = 2*i + 5 ;
```

图2.5 归纳变量例程

识别归纳变量的一般方法是遍历循环体中的所有指令。将循环中的所有变量作为候选，并找到每一个归纳变量 *j*，确定一个形如 $j = b*biv + c$ 的线性方程，这个方程将 *j* 和基本归纳变量 biv 联系起来，其中的 *b*，*c* 是常量。在其线性方程中具有相同基本归纳变量的(基本/依赖)归纳变量组成一个类(class)，这个基本归纳变量就是他们的基(basis)。具体来说，可以按照下面的过程寻找归纳变量：

（1）首先寻找这种变量：它们在循环中是形如 $i \leftarrow i + d$ 的赋值，其中 *d* 是循环常量，*i* 是一个归纳变量类的基。

（2）重复考察循环体内的每一条指令，循环满足如下条件的 *j*：*j* 出现在赋值的左部，并且这个赋值有表 2.3 所示的形式之一：

表 2.3 归纳变量的赋值形式

j ← i*e
j ← e*i
j ← i+e
j ← e+i
j ← i-e
j ← e-i
j ← -i

其中，*i* 是一个归纳变量，*e* 是一个循环常量。若 *i* 是基本归纳变量，则 *j* 属于 *i* 类，且它的线性方程可直接从定义它的赋值形式得出。

一旦识别出归纳变量，接下来便可以对它们施加三种重要的变换：强度削弱、归纳变量删除和线性函数测试替换。

2.3.2 强度削弱

强度削弱是把一个高代价的运算（比如乘法）替换为一个代价较低的运算（比如加法）的转换，是对归纳变量最有效的一种优化。它不仅局限于用加法替代乘法和用增量操作替代加操作，Allen 和 Cock[41]讨论了它的一系列应用，如用乘法运算替代指数运算。但是简单的强度削弱是使用最频繁获益通常也最大的。为了对已识别出来的循环归纳变

量执行强度削弱，可以通过下面的算法依次处理每一类归纳变量：

第一步，令 i 是基本归纳变量，j 是和 i 具有线性关系：$j=b*i+c$ 的依赖归纳变量；

第二步，分配一个新的临时变量 t_j，并用 $j \leftarrow t_j$ 替代循环中对 j 的单一赋值；

第三步，在循环中，每一个对 i 的赋值 $i \leftarrow i+d$ 之后，插入赋值 $t_j \leftarrow t_j + d*b$；

第四步，将赋值 $t_j \leftarrow b*i$，$t_j \leftarrow t_j + c$，放置在前置块的末尾，以保证 t_j 被正确初始化；

第五步，用 t_j 替代循环中的每一个 j；

第六步，以线性方程 $t_j = b*i+c$ 将 t_j 加入到 i 的归纳变量类中。

2.3.3 归纳变量删除和线性函数测试替换

在执行这两种转换之前，通常需要进行活跃变量分析。一个变量在程序的某个特定点上是活跃的（live），如果存在一条通向出口的路径，此路径上其值的使用先于对它的重新定义。如果不存在这种路径，则称该变量是死去的（dead）[1]。

除了进行强度削弱外，常常还可以直接删除某些归纳变量，因为这些变量是死去的。产生这种死去的归纳变量可能是由于以下几种原因：

（1）这种变量从一开始就没参与计算。

（2）由于强度削弱或者其他变换，导致这种变量变成无用

（3）在一次强度削弱过程中创建，但在另一次强度削弱之后就变成无用。

（4）只用在循环结束测试中，并且可以被上下文中另外的归纳变量所替代，这种情形称为线性函数测试替换。

例如图 2.6(a)中的归纳变量 *j* 是 *i* 类的一个归纳变量，并且它每次使用的值实际上都是 *i+1*，因此可以用 *j* 代替 *i*+1，从而直接删除归纳变量 *i*，得到图 2.6(b)中的情形。

```
j = 2
do i = 1, 10
    a[i] = i + 1
    j = j + 1
enddo
```

(a)

```
j = 2
do i = 1, 10
    a[i] = j
    j = j + 1
enddo
```

(b)

图2.6 归纳变量删除

还有一种情形是线性函数测试替换，如 *i* 在循环之前被初始化，并且在循环内作为循环结束控制变量被测试和递增。除了在循环之后可能需要它的终值外，在循环中没有任何其他用途。这样就可以用 *i* 类中的某些归纳变量来替代由 *i* 控制的终止测试，并删除原来使用的 *i*。

2.4 数组引用标量替换

标量替换是存储层次优化中的一种手段，可改善数组元素寄存器分配的一种方法。循环中可能存在两种变量，即标量变量和数组变量。标量变量是不用访存的变量，一般编译器中的数据流分析就可以得到其数据依赖关系。而对于数组变量，情况相对比较复杂，有时比较难以确定两个数组变量是否访问同一个空间地址，因为这依赖于数组变量的基地址和下标的值。数据依赖关系分析主要是分析两个下标含循环控制变量的数组访问在给定的条件下是否会访问同一个存储单元。在实际程序中，大多是数组元素的下标均是循环索引变量的线性表达式，即数组元素与下标是有仿射关系的。但是在为循环生成较好的目标代码时，很少编译器尝试将那种带下标的变量分配到寄存器中，尽管寄存器分配通常对标量非常有效。

用标量替换带下标的变量，从而使寄存器分配对他们有效的方法叫做标量替换（scalar replacement），也叫做寄存器流水（register pipelining）。实质上，这种方法是寻找重用数组元素的机会并用标量临时变量的引用来替代这些在重用。如图 2.7 中矩阵乘法的例子。

```
do i = 1, N
    do j = 1, N
        do k = 1, N
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
        enddo
    enddo
enddo
```

图2.7 矩阵乘法

在图 2.7 的例子中，$C(i, j)$用于每次迭代中存放结果的内存变量，并且具有读后写相关，即每一次迭代都需要访问两次它。如果用变量 c 代替 $C(i, j)$，并将 c 分配到寄存器中，会得到图 2.8 所示的代码形式，这样可以将存储访问次数减少 $2*(N^3 - N^2)$ 次。

```
do i = 1, N
    do j = 1, N
        c = C(i,j)
        do k = 1, N
            c = c + A(i,k) * B(k,j)
        enddo
        C(i,j) = c
    enddo
enddo
```

图2.8 矩阵乘法标量替换后

在优化的过程中，适当地使用循环交换和循环合并等转换，可以使得标量替换对于给定的循环嵌套起作用，或者使它们更有效果：循环交换可以使得循环携带的依赖是最内层循环携带的依赖，从而可以增加标量替换的机会；循环合并可以将对一个或多个数组元素的多个使用集中到一个循环内，从而为标量替换创造机会。另外，对嵌套循环施加标量替换可以得到显著的效果。如图 2.9(a)所示的嵌套循环，首先对最内层循环中的*x*[*i*]执行标量替换，就能得到 2.9(b)的代码，接下来可以继续进行循环展开和标量替换得到更优秀的代码。

```
for i ← 1 to n do
   x[i] ← 0
   for j ← 1 to n do
   x[i] ← x[i] + a[i,j]*y[j]
   endfor
endfor
```

(a)

```
for i ← 1 to n do
   t0 ← 0
   for j ← 1 to n do
      t0 ← t0 + a[i,j]*y[j]
   endfor
   x[i] ← t0
endfor
```

(b)

图2.9 嵌套循环标量替换

数组引用的标量替换利用对数组元素的连续多次访问，使得用标量替换了的数组元素可以成为寄存器分配的候选，并可适应于其他典型作用于标量的所有优。与数据有关的优化中最重要通常是与高速缓存有关的优化，除了本节介绍的数组引用的标量替换，其他重要的方法是循环铺砌（loop tiling），以及数据预取。

2.5 本章小结

本章首先介绍了编译器优化的基础理论。在此基础上，对 LLVM 这一流行的现代编译优化框架做了初步的探讨，包括 LLVM 中间表示语言 IR 和 LLVM Pass 框架。本论文采用的优化理论基础是循环迭代之间的数据重用性理论，为了实现重用，利用的编译优化技术主要是归纳变量优化和数组引用标量替换。它们都是比较传统的循环优化技术，对它们的介绍，有利于接下来对 LLVM 实现方式的研究。

第 3 章 LLVM 循环优化算法研究

在上一章对编译器以及循环优化方法介绍的基础上，本章将进一步分析 LLVM 对这些循环优化的具体实现，并通过实例，分析其算法的优越性和不足。

3.1 LLVM 的归纳变量优化

LLVM 归纳变量优化的流程以及涉及的分析、规范化、优化遍如图 3.1 所示。

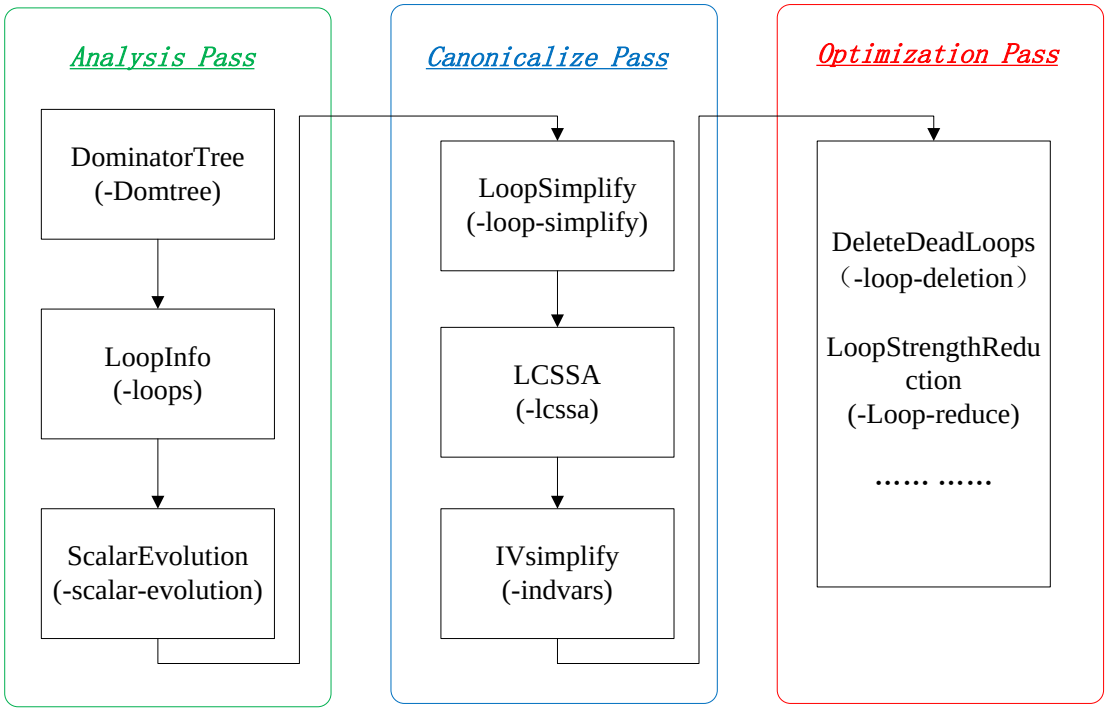

图3.1 LLVM归纳变量优化流程

图 3.1 描述了一段程序经过的 LLVM 归纳变量优化遍序列：编译器首先对程序建立支配树（Dominator Tree），依此来进行数据流分析；通过 loops 遍实现对循环的分析，找出循环的特征信息；ScalarEvolution（标量演变）是这个过程中最重要的分析遍，它会以 SCEV 格式分析程序的主要特征信息；LoopSimplify 用来对循环作初步变换，最终转换为 LCSSA 形式；通过 IVSimplify 对标准形式中的归纳变量进行合并、冗余删除等基本的优化；最后对归纳变量进行强度削弱等优化。这个顺序并不是严格的，中间有一些分析或者规范化的 pass 可能被重复的使用，比如 Scalar Evolution 会在各个阶段被多次使用，来分析和规范化程序。

3.2 归纳变量的识别

LLVM 在识别归纳变量阶段，需要多个分析和规范化遍共同合作。这个阶段的特点是：

（1）基于 SSA（IR）。

（2）通过循环分析（LoopInfo）、标量演变（ScalarEvolution）、循环化简（LoopSimplify）等手段，分析、规范化循环程序，识别出归纳变量。

LLVM 的规范化遍比较严格，这样有利于让优化变得更简单，通用性更好，是后续优化的基础。

3.2.1 循环规范化

归纳变量优化主要针对循环的，本节将从一般循环开始，看如何通过 LLVM 的规范化遍使之变成编译器容易操作的形式。图 3.2(a)描述的是一个一般的嵌套循环（Natural Loop），其中的每个数字代表一个结点，在 IR 层上，每个结点代表一个基本块。由控制流可以看出，结点 1 和结点 2 构成一个内层循环，结点 1，2，3 构成了外层循环。经过 Loop-simplify 规范化后，图 3.2(a)的循环结构变为图 3.2(b)的形式。

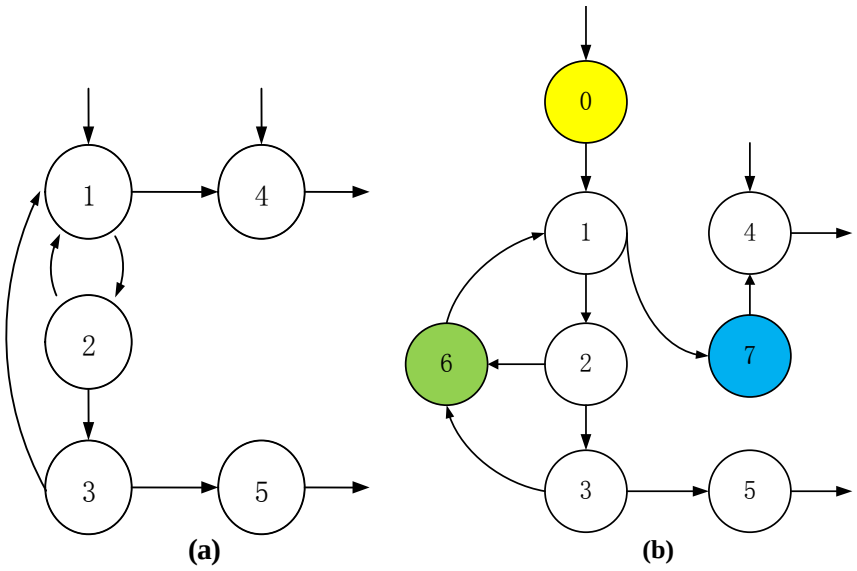

图3.2 循环规范化

根据图 3.2 中所描述的循环，给出关于循环中结点和边的一些概念：

（1）Header：循环开始处，控制进入循环的结点，如图 3.2(a)中结点 1。

（2）Back-edge：返回 header 结点的边，如图 3.2(a)中{(3, 1) (2, 1)}。

（3）Exiting：循环内的结点，且有后继结点在循环体之外，如图 3.2(a)中{1, 3}。

（4）Exit：循环外的结点，且有前驱结点在循环内，如图 3.2(a)中{4, 5}。

（5）Preheader：Header 的唯一前驱结点，如图 3.2(b)中结点 0。

（6）Latch：循环中只有唯一的回边，如图 3.2(b)中的(6, 1)，Latch 结点是这条边的起始点，即图 3.2(b)中结点 6。

（7）Exit-block：由头结点支配的 exit 结点，如图 3.2(b)中结点 7。

经过规范化遍的处理后图 3.2(a)中的循环要变成图 3.2(b)中的形式，即必须满足以下几个条件：

（1）Preheader 必须在每次进入循环之前执行。

（2）Latch 必须在每次新的迭代开始之前执行。

（3）Exit-block 必须在每次跳出循环之后执行。

这就是经过 Loop-simplify 遍之后一般循环所具备的形式。规范化的循环在 LLVM IR 中有相应的表示，如图 3.3 中给出了 header 结点和 latch 结点的 IR 例程。

```
header:
    %i = phi i64 [ 0, %preheader ], [ %next, %backedge ]
    ...
    %p = getelementptr @A, 0, %i
    %a = load float* %p
    ...
latch:
    %next = add i64 %i, 1
    %cmp = icmp slt %next, %N
    br i1 %cmp, label %header, label %exit
```

图3.3 header和latch结点对应的IR

LLVM IR 中使用 phi 指令来表示控制流信息，phi 后面可以有若干个域，每个域中的元素分别表示数据从哪个前驱基本块的哪个变量流入。%i = phi i64 [0, %preheader], [%next, %backedge]这条指令就表明变量%i 是 i64 类型的，它的值可能是来自%preheader 基本块的 0 或者是来自%backedge 块的%next 变量。getelementptr 指令用于计算变量在内存中的地址。表达式%p = getelementptr @A, 0, %i 表示变量%p 得到的是内存中全局变量 A 偏移%i 的地址。load 指令表示从内存加载如寄存器。icmp 是比较指令，根据 slt 标志（小于等于），若指令的第一个操作数小于等于它的第二个操作数，则指令结果为 1，否则结果为 0。根据 icmp 结果进行分支的选择，即最后的 br 指令。

3.2.2 LCSSA 形式

对于已经规范化的循环结构，在进行优化前还要进一步通过 LCSSA（Loop-closed

SSA）的规范化。LCSSA 会在循环边界处为那些在循环体内定义、循环体之外使用的变量插入 phi 指令。这样可以保证循环内的优化与循环外是完全隔绝的，phi 结点也阻断了循环外代码改变所造成的传播，更容易维持 IR 的 SSA 形式。如图 3.4 中的例子。

```
unsigned search( float *x, unsigned n, float y) {
    unsigned i, j = 0;
    for (i = 0; i != n; ++i)
        if (x[i] = = y)
            j = i;
    return j;
}
```

图3.4 LCSSA规范化的C例程

图 3.4 中的程序在循环中有一个条件判断，从而形成两个不同的分支。在 LCSSA 规范化之前的 IR 形式在图 3.5 中给出。

```
for.cond:
    %i.0 = phi i32 [ 0, %entry ], [ %inc , %for.inc ]
    %j.0 = phi i32 [ 0, %entry ], [ %j.1, %for.inc ]
    %cmp = icmp ne i32 %i.0, %n
    br i1 %cmp , label %for.body , label %for.end
...
if.end:
    %j.1 = phi i32 [ %i.0, %if.then ], [ %j.0, %for.body ]
    br label %for.inc
for.inc:
    %inc = add i32 %i.0, 1
    br label %for.cond
for.end:
    ret i32 %j.0
```

图3.5 LCSSA规范化之前的IR

for.cond 基本块是用于测试循环控制条件的，由 phi 指令控制程序的分支，由 icmp 指令的结果决定进行继续循环迭代（for.body）还是跳出循环（for.end）。由此可见，该基本块是典型的 Latch 结点。for.end 基本块是跳出循环后的执行的，因此它是 Exit-block。if.end 是表示 if 判断的结束的基本块。for.inc 是负责循环归纳变量自增的基本块。

经过 LCSSA 规范化后，for.end 基本块变为图 3.6 中的形式。也就是在 Exit-block 中插入了 phi 结点，它使得返回值可以直接来自 for.cond（Latch 结点），这样循环就变为封闭的（closed），使之与循环外能很好地隔绝。

```
for.end:
    %j.0. lcssa = phi i32 [ %j.0, %for.cond ]
    ret i32 %j.0. lcssa
```

图3.6 LCSSA规范化之后的for.end块

由这个例子可以看出，LCSSA 形式确定了每个循环退出时的活跃值，当然这可能引入很多多余的 phi 结点，但是它们最终会被 InstCombine Pass 删除。归结起来，LLVM 中 LCSSA 规范化用于处理函数中的循环，它的实现算法是：

（1)从分析遍中获得程序信息：getAnalysis->DominatorTree，getAnalysis->LoopInfo，getAnalysis->ScalarEvolution 详见图 3.1。

（2）把循环中所有的基本块放入 LoopBlocks 这个 vector 中。遍历每个基本块，根据 DominatorTree 信息，找出控制 Exit 块的所有基本块，如果没有，说明没有在循环内定义而在循环外使用的值，跳出处理过程，否则转入（3)。

（3）遍历每个 Exiting block 中的指令，排除没有被使用的指令（如 store 指令）和只有被在当前基本块使用的指令。

（4）对于 Exiting block 中未被排除的指令（记作 Instrs)，遍历并保存它们的使用。

（5）在 exit 块中插入的开始出插入 LCSSA 的 phi 结点，phi 的输入是来自 Exiting block（循环内）的 Instrs。

（6）将 Instrs 在循环外的使用进行改写，用新插入的 phi 指令进行替换。

（7）删除没被使用的 phi 结点。

循环化为 LCSSA 形式后，LLVM 可以更方便地对循环执行其他的优化，例如循环判断外提，循环化简等。

3.2.3 SCEV 分析格式

SCEV 是 LLVM 内部进行循环分析的一种表示格式，它来自 SCalar EVolution 两个词首字母的缩写，被设计用来支持一般的归纳变量[42]。

```
void foo () {
    int bar [10][20];
    for ( int i = 0; i < 10; ++i)
        for ( int j = 0; j < 20; ++j)
            bar[i][j] = 0;
}
```

图3.7 SCEV分析格式的C例程

图 3.7 给出的是对一个二维数组初始化 C 程序片段。截取它的一段 IR 形式，如图 3.8 所示。

```
for.cond:
    %i.0 = phi [ 0, %entry ], [ %i.inc , %for.inc ]
    %cond = icmp ne %i.0, 10
    br %cond , label %for.body , label %for.end
for.inc:
    %i.inc = add nsw %i.0, 1
    br label %for.cond
for.end:
    .....
```

图3.8 SCEV分析格式的IR片段

其中，for.cond 是循环的 latch 块，%i.0 是一个归纳变量。它的初始值是来自 entry 块的 0，当它小于等于 10 时，会控制循环继续迭代，否则跳出循环。该归纳变量的 SCEV 表示需要包含下面几个方面：初始值是 0；单调性是递增的；每次迭代的步长是 1；终值是 10。更一般地，SCEV 可以表示成 {A，B，C}<%D>的形式。其中，A 代表初始值，B 代表操作码，C 代表操作数，D 代表定义的基本块。

LLVM 中每个归纳变量都有其对应的 SCEV 分析格式。图 3.7 中的 *i*，*j* 是典型的归纳变量，图 3.9 给出了它们 IR 语句的 SCEV 表示。%i.0 指令的 SCEV 表示是：{0,+,1}< nuw ><nsw ><%for.cond > Exits: 10。说明归纳变量%i.0 的初始值是 0，终值是 10，以步长 1 递增，所在的基本块是%for.cond。%j.0 的初始值是 0，终值是 20，以步长 1 递增，所在的基本块是 for.cond1。

```
%i.0 = phi i32 [ 0, %entry ], [ %inc6 , %for.inc5 ]
;;SCEV: {0,+,1}< nuw ><nsw ><%for.cond > Exits: 10
%j.0 = phi i32 [ 0, %for.body ], [ %inc , %for.inc ]
;;SCEV: {0,+,1}< nuw ><nsw ><%for.cond1 > Exits: 20
```

图3.9 SCEV表示

事实上，SCEV 分析格式是一种基于递归链[43]（chains of recurrence CR）的表示形式，用递归链识别归纳变量的方法目前主流编译器（GCC 和 LLVM 中称之为 scalar evolution）中有着广泛的使用。它的基本形式是：{初始值（init），操作码（⊕），步长（stride）}。每一次迭代就在上一次迭代结果上 ⊕ 一个步长，例如，第一次迭代实际执行的是：*init* ⊕ *stride*。另外，也可以把非常量的 CR 步长用 CR 形式本身表示，这样就形成了一条递归链，如对于例子 $f(i) = i^2 = \{0, +, s(i\text{-}1)\}$，其中 $s(i) = \{1, +, 2\}$，表 3.1 给出了前四次的迭代结果。

表 3.1 递归链例子

迭代	{*init*, ⊕, *s*(*i*-1)}	*s*(*i*) = {1, +, 2}	*f*(*i*) = {0, +, *s*(*i*-1)}
i = 0	*init*	1	0
i = 1	*init* ⊕ *s*(0)	3	1
i = 2	*init* ⊕ *s*(0) ⊕ *s*(1)	5	4
i = 3	*init* ⊕ *s*(0) ⊕ *s*(1) ⊕ *s*(2)	7	9

强度削弱作用于 CR 形式的方法是：将每个 CR 和其中的 CR 步长作为循环嵌套的归纳变量。例如考虑多项式函数 $f(i) = a + b*i + c*i^2 = \{a, +, \{b+c, +, 2c\}\}$，设$\{b+c, +, 2c\} = s(i)$，从中可以找出两个归纳变量 x 和 y：

$$f(i) = x = \{x_0, +, y\}, \quad x_0 = a$$

$$s(i) = y = \{y_0, +, 2c\}, \quad y_0 = b+c$$

把 x 和 y 作为循环归纳变量，以单位步长 $i = 0, \cdots, n$ 递增，这样就得到图 3.10 的程序。

```
x = a
y = b+c
for i=0 to n
  f[i] = x
  x = x+y
  y = y+2*c
endfor
```

图3.10 将*x*和*y*作为归纳变量后的例程

再通过一种 CR 的代数方法可以得到强度削弱的 CR 形式。计算符号函数 $f(i)$的算法是：首先在用 f 的符号表示形式{0,+,1}代替 i，再利用表 3.2 中的 CR 代数重写规则计算 CR 格式。

表 3.2 代数重写后的 CR 格式

$\{x, +, y\} + c$	$\Rightarrow \{x+c, +, y\}$
$c\{x, +, y\}$	$\Rightarrow \{c \bullet x, +, c \bullet y\}$
$\{x, +, y\} + \{u, +, v\}$	$\Rightarrow \{x+u, +, y+v\}$
$\{x, +, y\} * \{u, +, v\}$	$\Rightarrow \{x \bullet u, +, y\{u, +, v\}+v\{x, +, y\}+y \bullet v\}$

识别归纳变量的编译器算法可以描述为下面的过程：

第一步找出递归关系。这一步的输入是循环中的活跃变量信息，输出是归纳变量递归关系的集合 S。S 的初始值是这样的一个集合 $S = \{ \langle v, v\rangle \mid v$ 是循环 header 结点的活跃

变量}。从下往上搜索循环，对于每次赋值 $v = x$ 就更新 S 中的二元组，最终输出结果集合。如表 3.3 中的例子。

表 3.3 找递归关系的例程

循环例程	S = {<c, c><i, i>}
do	
a = 1	S_4 = {<c, 1+m+n>, <i, i+1>}
b = m+n	S_3 = {<c, a+m+n>, <i, i+1>}
c = a + b	S_2 = {<c, a+b>, <i, i+1>}
i = i + 1	S_1 = {<c, c>, <i, i+1>}
enddo	

第二步将有递归关系的 S 集合为输入，输出 S 中的 SCEV 格式。对每个 S 中关系 $\langle v, x\rangle$，其中如果 x 有形如 v 的形式，那么 $v = v_0$ （v 是循环不变量）；如果 x 有形如 $v + y$ 的形式，那么 $v = \{v_0, +, y\}$；如果 x 有 $v * y$ 的形式，那么 $v = \{v_0, *, y\}$；如果 x 不包含 v，那么 $v = \{v_0, \#, y\}$ （v 是环绕的）；利用 SCEV 的代数重写规则，化简 SCEV 形式。

第三步是处理。其输入是归纳变量的 CR 形式，输出是归纳变量的封闭解。对 v 的每个 CR 形式实行 CR 代数变换（假设循环已经被规范化为 $i = 0, \cdots, n$），某些由非多项式和非指数的 CR 形式可能不具有封闭性，因此应该注意把化简的 CR 形式化为封闭的。如表 3.4 中的例子。

表 3.4 处理过程的例子

Loop L	Step	$S = \{\langle x, x\rangle, \langle z, z\rangle\}$	CR form	Closed form
do			$x = \{x_0, +, z\}$	$x(i) = x_0 + z_0 i + i^2 - i$
x = x+z	3	$S_3 = \{\langle x, \underline{x+z}\rangle, \langle z, z+2\rangle\}$	$z = \{z_0, +, 2\}$	$z(i) = z_0 + 2i$
y = z+1	2	$S_2 = \{\langle x, x\rangle, \langle z, \underline{z+2}\rangle\}$		
z = y+1	1	$S_1 = \{\langle x, x\rangle, \langle z, \underline{y+1}\rangle\}$		
while (…)				

LLVM 中的 CR 就是 SCEV 形式，通过 SCEV 格式的分析，以及其他一些分析和规范化遍之后，循环被变为 LCSSA 形式，然后 LLVM 利用归纳变量化简遍（-indvars）得到规范化的归纳变量，即循环初始值为 0，以步长 1 递增的变量。另外，indvar 还会通过比较归纳变量的值和循环结束的值将循环结束条件的值进行规范化，如把 for (i = 7; i*i < 1000; ++i)转变成 for (i = 0; i != 25; ++i)。那些在循环外使用的关于归纳变量的表达式，被变换为在循环外计算其值，并删除对归纳变量循环出口值的依赖。如果循环的唯一目的是计算一些表达式的结束值，这个变换就会让这个循环变为死的（dead）。

至此，对于归纳变量的识别就基本完成了，在接下来的优化遍中会利用到这些识别的归纳变量进行替换、删除、强度削弱等优化。

3.3 LLVM 的强度削弱

在 LLVM 中，强度削弱（LSR）依然是归纳变量优化中最重要的环节之一。它主要用在对循环内的数组引用：利用标量索引地址的方式重新生成用归纳变量索引数组的表达式。强度削弱以及线性函数替换、归纳变量删除在 2.2 中描述中优化的位置如图 3.11。

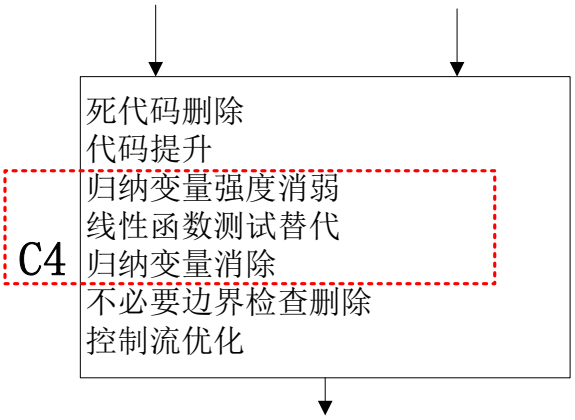

图3.11 归纳变量优化的位置

Indvars 优化遍识别出规范化的归纳变量后，强度削弱会替换掉循环中代价高的运算。LLVM 是通过-loop-reduce 遍实现循环强度削弱的：它会在第一次迭代中，用一个变量存放数组访问的初始值，然后在循环中产生一条新的 getelementptr 指令以适当的步长进行自增。如图 3.12 中的例子。

```
int a[100], i;
for (i =0; i < 100; i++)
    a[i] = 2*i + 5 ;
```

图3.12 强度削弱的C例程

图 3.12 中程序片段的 SSA 形式可以表示为图 3.13。

```
      t1←5      //t1存放数组元素
      i←0
L1: t2←i>=100
      if t2 goto L2
      t3←addr a      //t3 存放数组首地址
      t4←4 * i
      t5←t3 + t4      //首地址+偏移量
      *t5←t1          //将数组元素存回
      i←i + 1
      t1←t1 + 2      //每次迭代，a[i]自增 2
      goto L1

L2:
```

图3.13 强度削弱前的SSA形式

其中，*t*3 存的是数组 *a* 的首地址，偏移地址存在 *t*4 中，数组随着归纳变量 *i* 的每次迭代递增后，*t*5 用于存放计算的下一数组元素的地址。对 *t*4 进行强度削弱后，可得到图 3.14 的形式

```
        t1←5
        i←0
        t6←4
L1: t2←i>=100
        if t2 goto L2
        t3←addr a
        t4←t6
        t5←t3 + t4
        *t5←t1
        i←i + 1
        t1←t1 + 2
        t6←t6 + 4
        goto L1
L2:
```

图3.14 强度削弱后的SSA形式

从图 3.14 中容易看出，LLVM 的实现是将对归纳变量 *i* 的乘法计算削弱为了加法，用 *t*6 作为数组地址的自增步长变量。这样每次迭代通过 *t*6 自增一个数组元素所占空间的步长，然后加上事先保存好的数组首地址，二者相加得到下一次迭代的数组元素地址。

从这里可以看出一个问题，此程序每次迭代自增的步长是 4，这就涉及 32 位架构和 64 位架构中一点不同的地方。32 位处理器中常见的整型大小（int）和地址的大小是相同的，都是 4 个字节。但是对于 64 位系统，最常见的整型大小是 4 个字节，但是地址的大小是 8 个字节（64-bits）。关键问题是 32 位整型归纳变量可能溢出，如果 LLVM 从一个 32 位归纳变量产生一个 64 位地址的归纳变量，32 位整型算法中引发的溢出在 64 位地址算法中不会引发。

图 3.15 中的例子来解释这个溢出问题：

```
int i, sum, a[256];
char *j;
j = 0;
sum = 0;
for (i=0; i<1000; i++) {
   j += 3;
   sum += a[j];
}
```

图3.15 整型归纳变量溢出

在循环内，*j* 总是有 0-256 之间的 8 位宽值，这样所有通过 *a*[j]对 *a* 的访问也都会在数组界限内。但是在考虑强度削弱的时候必须要保留 *j* 的溢出行为，如果直接在数组 *a* 中给指针加 3，那么不会在超过 8 位宽的时候发生溢出，会访问数组边界外的内容，产生错误。不仅如此，强度削弱的目的是计算 *a*[j]的地址，会隐含地进行乘 4 的操作（如果 int 型变量占 4 字节），算法会要求用一个基于 *j* 的新归纳变量且每次迭代自增 4 的操作替换乘法运算，加 4 后会导致超过 8 位宽边界的速度会更快。为了保留 *j* 的溢出特征，乘法逻辑上需要 *offset* = (*j**4) % (256*4)，削弱算法产生的归纳变量应该是 *offset*2 = (*offset*1+4) % (256*4)。

解决这个问题就是限制强度削弱只能处理那些相同的位宽。这样，LLVM 为了和地址大小匹配，只削弱 64 位整型。这种方法会降低优化的几率，因为 C 程序可能有更多的 32 位整型。为此，需要强制将 32 位整型变量使用 64 位整型。

3.4 LLVM 的标量替换

数组引用标量替换是对重复访问的数组引用用临时标量对其进行替换。由于数组元素在内存中是连续存储的，每次取值都具有自增 sizeof（元素类型）的步长。因此，数组引用是十分典型的归纳变量。数组引用标量替换是改善数组元素寄存器分配的一种方法，在优化中的位置如图 3.16 所示。

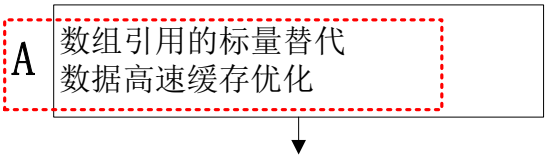

图3.16 数组引用标量替换的位置

由图中可以看出，这种优化需要包含循环表示和下标，以及依据关系所提供的信息，因此需要作用于高级中间表示，在 A 框中。在早期就对数组引用执行标量替代，将数组引用转换成标量变量，从而减少高速缓存优化需要处理的数组引用的数量。实际上，LLVM 源码中并没有单独将数组引用标量替换作为一个优化遍来实现。LLVM 实现的是聚合量的标量替换，此优化的算法名称是 SRoA（Scalar Replacement of Aggregates）。它能够使其他优化作用于聚合对象（结构体和数组）的分量，判断聚合对象的哪些分量具有标量值，然后将这种分量赋给临时变量。结果使得这种分量成了寄存器分配，以及其他作用于标量优化的候选对象。而数组引用标量替换实质上也是对数组引用赋给临时变量，使寄存器分配对之有效。由此可见，SRoA 实质上是包含了数组引用标量替换的思想。

LLVM 的标量替换主要是针对 IR 中的“alloca”指令。alloca 指令会为当前执行的函数在栈上分配内存，当函数返回时会自动释放。通常用于表示那些必须有地址空间的自动变量。由此可知，该指令是访存指令（load/store）得以执行的必要条件，它的大量使用会限制寄存器使用，造成大量的性能损失。对于这个指令的处理还要结合 Mem2Reg 这个 pass。Mem2Reg 算法的作用是将存储器变量提升为寄存器变量（Promote Memory To Register），采用的是 Sreedhar[44]提出的一种放置 PHI 结点的算法。当 alloca 指令结果只被 load/store 使用时，alloca 指令就会被 Mem2Reg 提升。SRoA 与 Mem2Reg 几乎总是同时存在的，共同发挥优化作用的。

图 3.17 的程序片段 caller 对 callee 的调用：

```
int callee(const int *X) {
        return *X+1;   // load
}
int caller() {
        int T;    // on stack
        T = 4;    // store
        return callee(&T);
}
```

图3.17 Mem2Reg算法的C例程

把它所有的 load/store 都明确地表示出来，程序变为图中所示的样子。声明变量%T 会在栈上为其分配 int 类型大小空间，对%T 赋值后，它需要存入内存。调用 Callee 时，它的参数也需要从内存加载。

```
internal int %callee(int* %X) {
        %tmp.1 = load int* %X
        %tmp.2 = add int %tmp.1, 1
        ret int %tmp.2
}
int %caller() {
        %T = alloca int
        store int 4, int* %T
        %tmp.3 = call int %callee(int* %T)
        ret int %tmp.3
}
```

图3.18 Mem2Reg处理前的IR

LLVM 的变换，首先会将形参%X 替换为实参，从而在主调函数 caller 中加载变量%T，这样为%T 分配的 int 类型大小的空间只被用在 store 和 load 指令中，这样的 alloca 指令会被 Mem2Reg Pass 清理，alloca，store，load 指令一起被删除，实参%T 被常量 4 代替，这样一来，程序变为图 3.19 中清晰易读的目标指令序列。

```
int %caller() {
        %tmp.3 = call int %callee(int 4)
        ret int %tmp.3
}
```

图3.19 Mem2Reg处理后的IR

这就是 Mem2Reg 做的事情，此算法会为数据流分析构造稀疏评价图（SEG），在迭代必经边界 IDF（iterated dominance frontier）处插入 PHI 结点。它是基于 SSA 的，SSA 被证明是非常有效的数据流分析方法，对于分析程序中定义到使用（D-U）的路径非常有效。关于 SSA 中的算法非常多，也比较复杂，并不是本文的讨论范围，Mem2Reg 基于 SSA 的具体实现算法可参考文献[38]。

SRoA 的实现思路是：把聚合数据类型的 alloca 指令变换为其中每个元素的 alloca 指令，然后将每个单独的 alloca 指令尽可能地转变成标量 SSA 形式。这需要结合 Mem2Reg 算法，通过反复执行 SRoA 和 Mem2Reg，直到得到可以提升效果好的形式。SRoA 算法做的事情包括：找出聚合类型的 alloc 指令中是否有可以提升的元素，并将其提升至寄存器；将 alloca 的元素转换成合适的向量或者位域格式的整型标量。算法会一直在函数体内进行迭代，直至找出所以可被提升的变量。由于这种变换只针对 alloca，因此那些很少和外部存储器进行数据交换的代码区域基本不会被改变，也不会被标量化。

3.5 循环优化的不足

由于数组中每个元素的类型都是相同的。因此，如果在循环中依次对数组元素进行访问，那么数组引用（每个数组元素的地址）就以线性关系变化，是很典型的归纳变量。如果归纳变量优化做得好，那么就能节省寻址运算的开销，从而减小循环内指令的代价。

```
void bubbleSort(int *arr, int n) {
  int i, j;
  int sz = n - 1;
  for (i = 0; i < sz; i++)
    for (j = 0; j < sz - i; j++)
      if(*(arr+j) > *(arr+j+1)) {
        int t = *(arr+j);
        *(arr+j) = *(arr+j+1);
        *(arr+j+1) = t;
      }

}
```

图3.20 冒泡排序函数

以图 3.20 的冒泡排序函数为例，在该函数的嵌套循环中，LLVM 对于其仿射下标的数组元素的访问，可以描述为下面的过程：

（1）识别并规范化循环以及循环归纳变量。

（2）将数组首地址存入寄存器 *r*1，在访问数组的过程中，这个寄存器值保持不变，以便能计算数组引用。

（3）从 *r*1 处加载数组首元素至寄存器 *r*2，保持数据局部性，在其被使用之前不会被替换。

（4）通过地址 *r1* 自增或者左移操作依次实现地址的自增。当用到下一个数组元素时，先将 *r*1 左移两位得到元素的引用，再从 *r*1<<2 处加载数组元素至寄存器 *r*3。

（5）将 *r*3 元素保存至安全的寄存器内（局部时间内不会被冲掉），以便重用。

在这个过程中，数组引用是非常典型的依赖归纳变量。循环控制变量，即数组下标 *i* 是基本归纳变量，每当 *i* 自增步长，地址就增加 sizeof(int)*步长。第（4）步中通过强度削弱，将乘法运算可转化为移位计算/加法计算。从算法中可以看出，LLVM 每次迭代中，计算地址的方式非常繁琐：都需要利用数组首地址加上计算得到的偏移地址来得到新的元素地址。该问题可以在图 3.21 中的 RISC 汇编中反映。

```
.BB0_4:
    mov r13,r6
    lsli   r13, 2          #虚线框里的指令，只是计算出 a[i+1]
    mov r14,r2             的地址
    addur14,r13
    ldw r13, (r14, 4)      #将 a[i+1]加载入寄存器 r13
    addi r6,1
    cmplt      r13,r7
    jbf   .BB0_5
# BB#6:                                 #    in Loop: Header=BB0_4 Depth=2
    stw r13, (r14)
    stw r7, (r14, 4)       } #将 a[i]和 a[i+1]存入内存相应位置
    jbr   .BB0_7
.BB0_5:                                 #    in Loop: Header=BB0_4 Depth=2
    mov r7,r13
.BB0_7:                                 #    in Loop: Header=BB0_4 Depth=2
    cmpne     r5,r6
    jbt   .BB0_4
```

图 3.21 冒泡排序汇编

LLVM 在外层循环中将数组一个元素 *a*[i]加载进 *r*14，在内层循环中，首先计算出

数组下一个元素 *a*[*i*+1]的地址（虚线框里的部分），然后从此地址将 *a*[*i*+1]加载入寄存器参加比较运算。由此可以看出 LLVM 在计算 *a*[*i*+1]地址的时候，使用了四条指令，非常繁琐。针对此不足，可以利用归纳变量和标量替换对其优化进行。

3.6 本章小结

本章是对 LLVM 实现迭代间优化算法的基础研究部分。归纳变量优化和标量替换是循环优化以及存储层次优化的主要的两种技术。本章首先分析了归纳变量优化算法，其实现算法庞大，借助第二章的预备知识可以将其概括为三种规范化 Pass，以及后续的优化 Pass；第二部分对标量替换算法做了详细的分析，这部分涉及了 SSA 的基本算法，比较复杂。通过对这两种编译优化基本算法的分析，可以为后面找出其实现的不足，提出优化方案做准备。

第 4 章 迭代间数据重用算法的设计与实现

本章将针对 LLVM 的归纳变量优化和标量替换优化在具体实现中的不足，提出改进方案。基于中间表示，通过在函数运行时加入 LLVM 的 Pass，设计并实现迭代间数据重用优化算法。

4.1 控制流分析

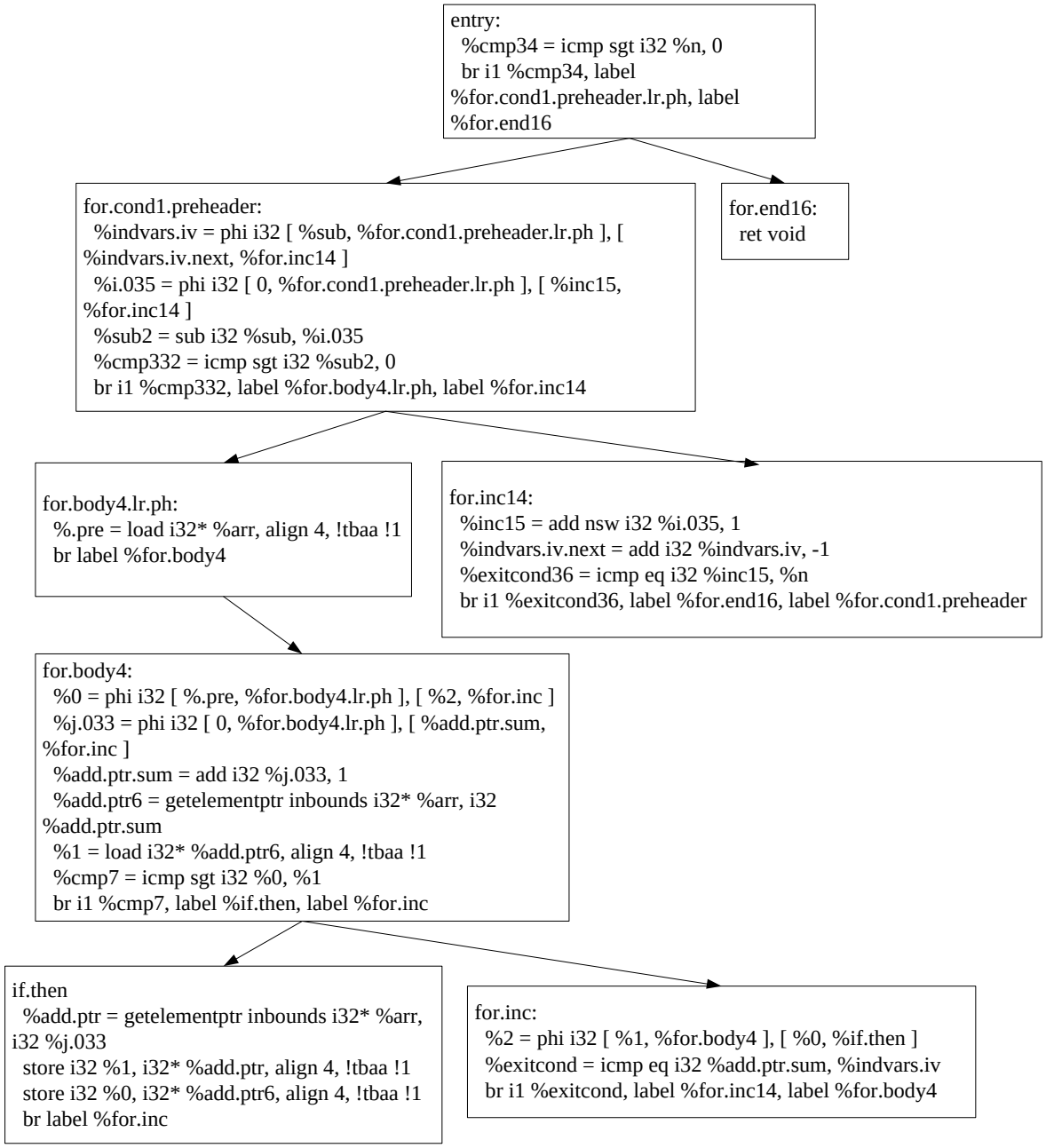

图4.1 冒泡排序IR支配树

首先对 3.5 节图 3.20 的冒泡排序在 IR 层进行程序结构分析，图 4.1 给出了该程序 IR 层的支配树（Dominant Tree），包含了函数体所有的 IR 代码，由此可以看出每个基本块的之间的控制关系。将上图化简为基本块之间的依赖图，如图 4.2。

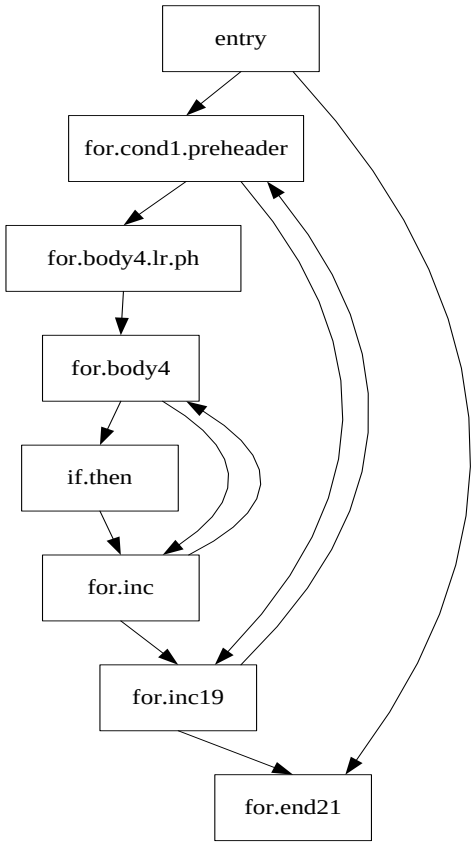

图4.2 基本块依赖图

从图 4.2 中可以更清楚地看出基本块之间的控制关系，其中 entry 块是函数的入口；for.cond1.preheader 是外层循环 header 结点，它控制了内层循环的 preheader 结点 for.body4.lr.ph；for.body4，if.then，for.inc 三个基本块是内层，其中 for.body4 是内层循环的 header 结点，for.inc 是内层循环的 latch 结点。LLVM 将 for.body4 块（header）中的一条加载指令移到 for.body4.lr.ph（preheader）中，由于一次迭代中比较的是 $a[i]$与 $a[i+1]$，在下一次迭代中用到的是 $a[i+1]$和 $a[i+2]$，因此，迭代间的 $a[i+1]$是可重用数据。为它分配一个临时变量后，寄存器分配对它有效，那么，它就会 $a[i+1]$就会在本次迭代之后保存到寄存器中，下一次迭代直接操作保存它的寄存器。

在上面对各个循环结点信息分析的基础上，下一节将给出如何利用这些循环结点实现迭代间数据重用优化算法。

4.2 迭代间数据重用优化算法设计

根据前面的分析，下面针对图 4.3 所示的一般情况下循环中的数组访问，给出迭代间数据重用优化算法。

```
for (i=i0; i<N; i=i+s)
    function(a[ki+b]);
```

图4.3 循环中数组访问的一般情形

其中，循环控制变量 *i* 是基础归纳变量，其初始值为 i_0，自增步长为 *s*，终值为 *N*，循环体内对数组 *a* 的访问方式为 *ki* + *b*，若用 *base* 表示数组 *a* 的首地址，则元素 *a*[*ki*+*b*] 的地址可表示为 *base* + (*ki* + *b*) * *m*，*m* 是数组元素在内存中所占的字节数。在循环体内这样计算地址通常需要四条汇编指令，通过下面给出的化简，可以重用上一次迭代结果进行本次迭代地址的计算。

$$
\begin{aligned}
&\&a[ki+b] \\
&= base + (ki+b)\times m \\
&= base + [k(i-s)+ks+b]\times m \\
&= base + [k(i-s)+b]\times m + ksm \\
&= \&a[k(i-s)+b] + ksm
\end{aligned}
$$

经过计算，可以得到数组地址是典型的归纳变量，其自增步长为 *ksm*，这三个数分别为：依赖归纳变量的系数、基础归纳变量的自增步长、数组元素在内存中的大小，三个数均为常量。LLVM IR 层地址计算使用的是 getelementptr 指令，其操作数第一个域为基址，第二个域为偏移地址，由此总结出图 4.3 中所示的迭代间数据重用优化算法。

输入：规范化处理后的循环体 IR
输出：添加迭代间数据重用优化的 IR

1. 识别循环控制变量 *i*，确定其自增步长 *s*。
2. 遍历循环体中的 getelementptr 指令，判断其地址索引域 *j* 是否为 *i* 的依赖归纳变量，是则转入 3，否则转入第 7 步。
3. 确定 *j* 和 *i* 的依赖关系： *j* = (*ki* + *b*)**m*。
4. 在循环中 preheader 结点，将数组首地址转化为整型变量 *base*。
5. 在 getelementptr 指令所在的基本块增加整型归纳变量 *cast*，其可能值分别为 *base* 和 *cast.next*。将整型 *cast* 变为指针类型，作为 getelementptr 的寻址操作数，替代当前的指令。
6. 在循环的回边中，增加变量 *cast.next*，使 *cast.next* = *cast* + *ksm*。
7. 如果循环体内还有未处理的 getelementptr，则访问下一条 getelementptr 指令，转至 2，否则转至 8。
8. 结束。

图4.4 迭代间数据重用优化算法

该算法的第 4 步中需要将数组首地址转化为整型，因为它需要作为新定义归纳变量

的传入值，所以需要做类型转换，可以使用 LLVM 提供的 ptrtoint 方法完成此操作。第 5 步中，仍需要使用 getelementptr 计算地址，且 getelementptr 的第一个操作数必须是指针，所以利用用 inttoptr 指令将 *cast* 变量由整型转换为指针类型。另外，由于 IR 是 SSA 形式的，因此不会因为重复定义新的变量而导致冲突，即每次处理 getelementptr 指令所定义的 *cast*，*cast.next* 都会被命名为不同的变量。

特别地，对于冒泡排序，图 4.4 中的算法可以描述为下述过程：

（1）识别并规范化循环以及循环归纳变量。

（2）将数组首地址&*a* 存入寄存器 *r*1，在访问数组的过程中，这个寄存器可以被重新赋值，不需要一直保存数组首地址的值。

（3）从 *r*1 处加载数组首元素 *a*[0]至寄存器 *r*2，保持数据局部性，作为归纳变量，其被使用之前不会被替换。

（4）通地址 *r*1 增加固定的步长，这个步长是数组元素类型所占的空间大小（*sizeof*(*typeof*(*a*[*i*]))）。得到 *a*[1]的地址并保存至 *r*3，这个地址是下一次计算数组地址的基址，在被使用前不会消亡。

（5）依此类推，在每一次数组引用的时候，都利用上一次迭代所保存的数组地址，增加固定的步长。从而将数组地址替换为标量，并进行强度削弱。

可以用图 4.5 中 SSA 形式的例程说明此优化算法。

```
      t1←5
      i←0
      t6←4
L1: t2←i>=100
      if t2 goto L2
      t3←addr a
      t4←t6
      t5←t3 + t4
      *t5←t1
      i←i + 1
      t1←t1 + 2
      t6←t6 + 4
      goto L1
L2:
```

图4.5 SSA例程

这个例程在 3.3 节中出现过，它利用强度削弱得到了比较好的输出代码，但经过本节的优化算法后，可以得到更加优秀的目标代码，如图 4.6。

```
        t1←5
        i←0
        t3←addr a
        t4←t3
L1: t2←i>=100
        if t2 goto L2
        *t4←t1
        i←i + 1
        t1←t1 + 2
        t4←t4 + 4
        goto L1
L2:
```

图4.6 优化后的SSA

在图 4.6 中 SSA 形式的代码中，首先将数组地址&*a* 在循环外进行加载，存入 *t*3，并把它赋给变量 *t*4，*t*4 在循环中是自增变量，步长是数组元素在内存中所占的空间大小（此处为 4 个字节），这样，*t*4 每次自增就得到下一次迭代的数组引用，从而实现数组引用的迭代间数据重用。每次迭代中 *t*4 计算数组元素的地址，*t*1 计算数组元素的值，最终将 *t*1 的值存入 *t*4 的地址中，完成一次迭代求解。

4.3 算法实现

由于 LLVM 良好的模块化，因此直接写一个优化遍来实现优化算法的方法是可行的，也是相对容易的。编写 pass 的流程是：

（1）挑选测试用例 foo.c 作为 LLVM 编译器的输入。

（2）利用 clang 前端生成 LLVM 中间表示 foo.ll，通过 LLVM 后端的 CodeGen 生成 Target 代码（Target 是目标平台）。命令是 clang -emit-llvm foo.c –S –o foo.ll，需要参考的文档可能包括 LLVM Command Guide[45]。

（3）生成目标平台的汇编代码，命令是 llc foo.ll –march=Target –o foo.s，参考文档 Writing an LLVM Backend[46]，The LLVM Target-Independent Code Generator[47]。

（4）使用汇编器和链接器，将 foo.s 编译成平台可执行 exe 文件。执行测试程序的执行时间。

（5）用 oprofile 等性能分析工具对程序做 profiling，找出程序的热点，也就是程序的性能瓶颈，看汇编中哪段代码耗时比较多，有可提升的空间。

（6）在分析 foo.s 后，找出程序的缺陷，分析一般形式，提出改进后的目标代码 foo_opt.s。

（7）找出与热点代码相对应的 IR，在对 IR 实现理解的基础上，结合改进的目标代

码，写出改进后的 IR。这是最关键的一步，因为 IR 到目标代码之间还要进行很多的优化、转化，必须对程序以及 IR 进行足够的分析，才能知道什么样的 IR 可以生成期望的汇编代码。这需要参考一些 LLVM 的文档，包括 LLVM Language Reference Manual[48]，LLVM's Analysis and Transform Passes[49]。

（8）编写 LLVM 转化 Pass，参考文档 LLVM Programmer's Manual[50]，LLVM Coding Standards[51]，Doxygen generated documentation[52]，Writing an LLVM Pass[53]。

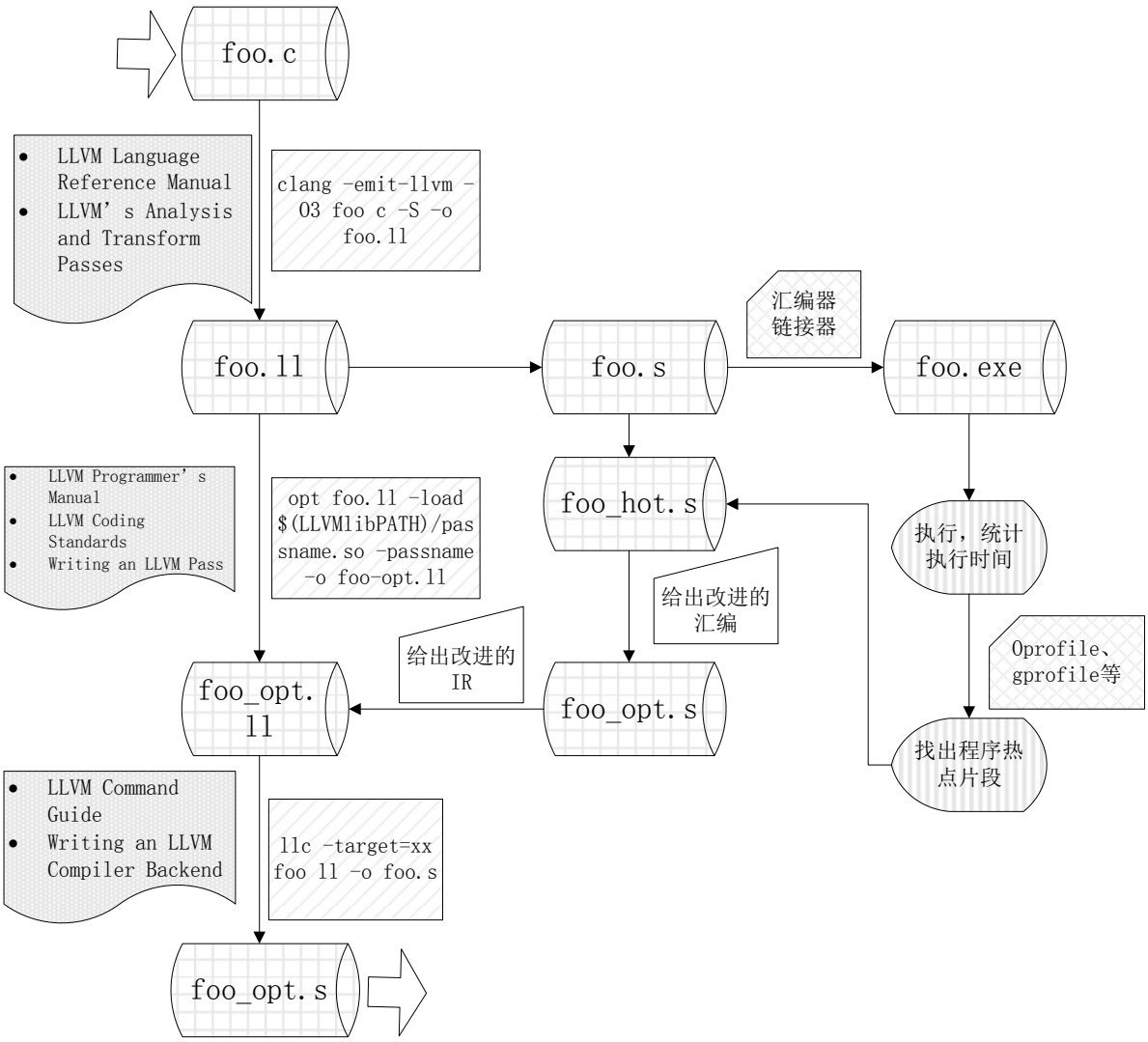

图4.7 编写pass流程

该过程在图 4.7 中给出。通过上面的步骤就可以实现一个优化遍。该优化算法最重要解决的问题就是如何使数组地址能够实现自增，以及在何处插入 PHI 结点。

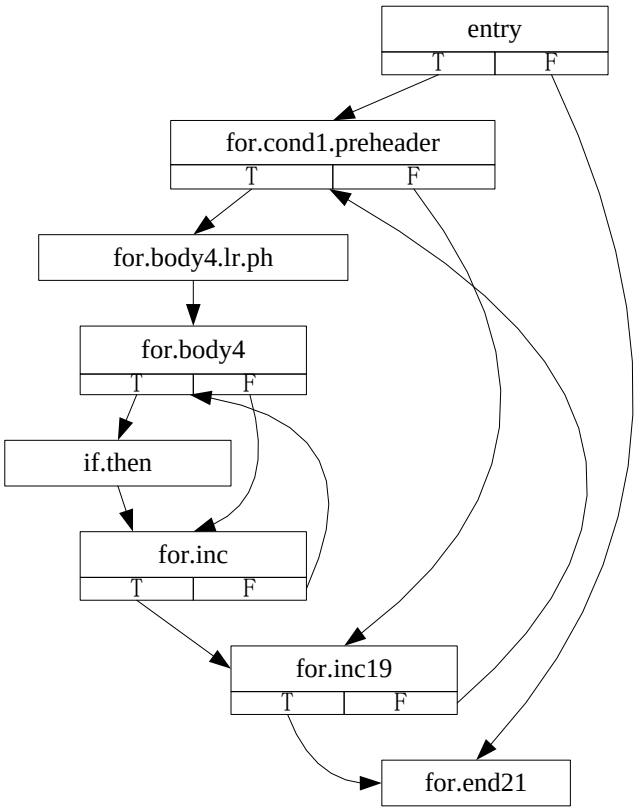

图4.8 控制流图（CFG）

根据图 4.8 的控制流图，它所在的基本块是内层循环，getelementptr 位于 for.body4 中，可以直接来自外层循环到内层循环的入口 for.body4.lr.ph，也可以来自内层循环的结束 for.inc，这样入口信息是由 PHI 结点决定的。

在 IR 中对数组元素地址操作的指令是：

%.sum = add i32 %j.01, 1

%7 = getelementptr inbounds i32* %arr, i32 %.sum

在数组地址增加的基本块 for.inc，计算地址首先需要对归纳变量%j.01 自增 1，而计算下一个数组元素地址的 getelementptr 指令需要给数组元素加上这个和，因此，要寻找的 getelementptr 指令是：第一个域是数组元素首地址，第二个域是要自增的操作，非这类的 getelementptr 指令可以从 vector 中 pop 出去。即首先必须定位到第二个域是 BinaryOp（sum）的 getelementptr 指令。

另一个问题是需要确定这个 BinaryOp 指令的操作数，由于数组自增的操作数必定是一个常数类型，也就是说 BinaryOp 的增量操作数需要是常量。而它的基础操作数也必定是有两个可能值：一个来从本层循环外的 Preheader 结点传入的初始值，另一个来自本层循环 Back-edge 的 Latch 结点，因此 BinaryOp 的基础操作数需要是一个 PHI 结点，

PHI 的值分别来自 Preheader 和 Latch 结点。总之，所要寻找的 BinaryOp 的第一个操作数是一个 PHI 结点，第二个操作数是一个常数。

有了上述两个基于 IR 的约束，说明需要变换的 getelementptr 满足约束条件，可以利用之前所给出的算法进行变换。

为完成这一操作需要借助基址加偏移量的形式。首先定义一个 BasePointer，来保存数组元素的首地址，并将指针运算转化为整型运算，这样可以更方便地对地址进行算数运算，获得目的地址的值。在约束 BinaryOp 的时候已经分析出，它的第一个操作数是 PHI 结点，PHI 的第一个前驱值是 Preheader，第二个前驱值是 Latch，显然之前设置的 BasePointer 用来保存首地址的，应该放到 Preheader 结点中。偏移量 NewIdx（下一个元素地址，表示自增）应该放入 Latch 中。这样，BasePointer 只在进入循环前记录一次，而 NewIdx 在每次迭代过程中都可以进行自增操作。NewIdx 是一条替代之前增量计算的 BinaryOp，他每次递增的步长是 sizeof（数组元素的类型大小）。为了实现迭代间的数据重用，NewIdx 使用的操作数，必须是上一次已有的结果。为了满足这样的重用，在 getelementptr 的所在的基本块，插入一个 PHI 结点，使它分别可以来自 BasePointer 和 NewIdx，这就是新的 getelementptr 指令所使用的地址，不要忘了在使用之前要把它从整型变回指针型。

4.4 本章小结

本章首先找出 LLVM 在寻址模式上的不足，通过对算法以及实例的控制流分析，提出了结合归纳变量以及标量替换的优化算法，该算法能有效利用迭代之间的可重用数据，简化了数组引用在计算上的复杂度。在分析程序中间表示的基础上，设计出了改进后的中间表示。最后利用 LLVM 提供的 API，实现了中间表示的转化，使得 LLVM 更好地发挥了其在设计上的优越性，提高了程序的性能。

第 5 章 实验结果与分析

5.1 实验环境

本文所实现的优化是一种基于中间表示的通用优化，可以作用于不同的后端平台，但对于 RISC 架构最为有效。本文采用 ARM 架构作为实验平台，实验过程中所使用的编译器包括 GCC 和 LLVM。具体的平台信息和编译器版本如表 5.1 所示。

表 5.1 ARM 开发板参数

软硬件配置	配置参数/版本
开发板	Xilinx Zynq-7000 ZC706
ARM 处理器	dual-core Cortex™-A9
主频	667HZ
内存	1G
存储卡	8G
系统	Linaro 11.12(Ubuntu 内核)
GCC 编译器	arm-xilinx-linux-gnuabi-gcc-4.6.1
LLVM 编译器	3.4 ARM Target

编译开发板的交叉编译工具链（arm-xilinx-linux-gnuabi-gcc），以及 LLVM 编译框架的搭建过程可分别参考他们的官方文档[54][55]。

假设 LLVM 的源码目录是 LLVM_SRC，编译目录是 LLVM_BUILD，安装目录是 LLVM_LOCAL，则在 LLVM 中添加平台无关优化遍的具体方法是：首先在 $(LLVM_SRC)/lib/Transforms/目录下加入优化遍的文件夹 getelementptr，文件夹中包括程序实现的 c++文件，LLVMBuild 以及 Makefile，修改当前文件夹下的 Makefile 和 LLVMbuild 文件；然后在$(LLVM_BUILD)中重新 configure 生成新的 makefile，并在 $(LLVM_BUILD)/lib/Transforms 下编译 getelementptr，执行安装会将生成的动态库 LLVMmyPass.so 拷贝至$(LLVM_LOCAL)/lib/目录下。在利用 opt 工具进行 IR 的转化时需要指定该动态库路径，即命令行为：

opt –load $(LLVM_LOCAL)/lib/LLVMmyPass.so –myPass InputIR –o OutputIR

5.2 测试用例

本节选取了 4-tap FIR 滤波程序和冒泡排序作为测试用例，它们具有典型的迭代间

数据重用特点，程序简单，能够达到较好的测试分析效果。

5.2.1 冒泡排序

在第三章和第四章已经利用冒泡排序进行分析，现在图 5.1 给出完整的冒泡排序例程。它是一个典型的嵌套循环，具备迭代间数据重用算法所要求的特点。在该程序中，对 10 万个整数进行冒泡排序，由 bubbleSort 函数实现，其驱动程序中用 clock()函数统计排序的起始和结束时间，并通过差值计算冒泡排序的时间。

```
#include<stdio.h>
#include<stdlib.h>
#include <time.h>
#define LEN 100000
void bubbleSort(int *arr, int n)
{
    int i, j, t;
    for (i = 0; i < n; i++)
        for (j = 0; j < n-1-i; j++)
            if (*(arr+j) > *(arr+j+1)){
                t = *(arr+j);
                *(arr+j) = *(arr+j+1);
                *(arr+j+1) = t;
            }
}
int main()
{
    long i;
    int arr[LEN];
    clock_t start, finish;
    double duration;
    for (i = 0; i < LEN; i++)
      arr[i]=rand()%LEN;
    start = clock();   /*开始计时*/
    bubbleSort(arr,LEN);
    finish = clock();   /*计时结束*/
    /*函数执行时间*/
    duration = (double)(finish-start)/CLOCKS_PER_SEC;
    printf("%f seconds\n", duration);
}
```

图5.1 冒泡排序测试例程

对于图 5.1 中的冒泡排序函数，使用命令：

clang –emit-llvm –bubbleSort.c –O3 –S –o bubbleSort.ll

生成汇编形式的 IR，其 IR 程序如图 5.2 所示。

```
define void @bubbleSort(i32* nocapture %arr, i32 %n) #0 {
  %1 = add i32 %n, -1
  %2 = icmp sgt i32 %1, 0
  br i1 %2, label %.preheader, label %._crit_edge3
.preheader:                                       ; preds = %0, %._crit_edge
  %indvars.iv = phi i32 [ %indvars.iv.next, %._crit_edge ], [ %1, %0 ]
  %i.02 = phi i32 [ %14, %._crit_edge ], [ 0, %0 ]
  %3 = sub i32 %1, %i.02
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %.lr.ph, label %._crit_edge
.lr.ph:                                           ; preds = %.preheader
  %.pre = load i32* %arr, align 4, !tbaa !1
  br label %5
; <label>:5                                       ; preds = %.backedge, %.lr.ph
  %6 = phi i32 [ %.pre, %.lr.ph ], [ %11, %.backedge ]
  %j.01 = phi i32 [ 0, %.lr.ph ], [ %7, %.backedge ]
  %7 = add nsw i32 %j.01, 1
  %8 = getelementptr inbounds i32* %arr, i32 %7
  %9 = load i32* %8, align 4, !tbaa !1
  %10 = icmp sgt i32 %9, %6
  br i1 %10, label %12, label %.backedge
.backedge:                                        ; preds = %5, %12
  %11 = phi i32 [ %9, %5 ], [ %6, %12 ]
  %exitcond = icmp eq i32 %7, %indvars.iv
  br i1 %exitcond, label %._crit_edge, label %5
; <label>:12                                      ; preds = %5
  %13 = getelementptr inbounds i32* %arr, i32 %j.01
  store i32 %6, i32* %8, align 4, !tbaa !1
  store i32 %9, i32* %13, align 4, !tbaa !1
  br label %.backedge
._crit_edge:                                      ; preds = %.backedge, %.preheader
  %14 = add nsw i32 %i.02, 1
  %indvars.iv.next = add i32 %indvars.iv, -1
  %exitcond4 = icmp eq i32 %14, %1
  br i1 %exitcond4, label %._crit_edge3, label %.preheader
._crit_edge3:                                     ; preds = %._crit_edge, %0
  ret void
}
```

图5.2 冒泡函数的IR代码

其中，虚线框内的即是需要进行优化的内层循环的部分。经过优化遍后，该部分 IR 程序变为图 5.3 中的形式。

```
.lr.ph:                                           ; preds = %.preheader
  %.pre = load i32* %arr, align 4, !tbaa !1
  %base = ptrtoint i32* %arr to i32
  br label %4
; <label>:4                                        ; preds = %.backedge, %.lr.ph
  %5 = phi i32 [ %.pre, %.lr.ph ], [ %10, %.backedge ]
  %j.01 = phi i32 [ 0, %.lr.ph ], [ %6, %.backedge ]
  %cast = phi i32 [ %base, %.lr.ph ], [ %cast.next, %.backedge ]
  %6 = add nsw i32 %j.01, 1
  %ptr = inttoptr i32 %cast to i32*
  %7 = getelementptr inbounds i32* %ptr, i32 1
  %8 = load i32* %7, align 4, !tbaa !1
  %9 = icmp sgt i32 %8, %5
  br i1 %9, label %11, label %.backedge
.backedge:                                        ; preds = %11, %4
  %10 = phi i32 [ %8, %4 ], [ %5, %11 ]
  %cast.next = add i32 %cast, 4
  %exitcond = icmp eq i32 %6, %indvars.iv
  br i1 %exitcond, label %._crit_edge, label %4
```

图5.3 经过优化后的IR代码

图 5.2 中虚线框中的 IR 程序经过优化，转化为了图 5.3 中的形式。再使用命令：

llc bubbleSort-opt.ll –S –march=arm –o bubbleSort-opt.s

clang bubbleSort-opt.s –O3 –o bubbleSort-opt.exe

即可以生成优化后的 ARM 汇编以及可执行程序。

5.2.2 FIR 滤波

FIR 滤波程序是典型的可进行迭代间数据重用的单层循环的例子。图 5.4 给出了一个简单的 4-tap FIR 滤波程序。它利用 Linux 提供的系统函数 gettimeofday 统计 $b[i] = 3*a[i] + 5*a[i+1] + 2*a[i+2] + 4*a[i+3]$循环体的执行时间。

同 5.2.1 节中的过程一样可以生成 ARM 的汇编和可执行程序。

```
#include <stdio.h>
#include<sys/time.h>
#include <stdlib.h>
void fir(int a[], int N)
{
  int i,sum;
  int b[N-3];
  struct timeval start;
  struct timeval end;
  unsigned long diff1;
  gettimeofday(&start,NULL);   /*获得循环开始处的系统时间*/
  for (i = 0; i < N-3; i++)
    b[i] = 3*a[i] + 5*a[i+1] + 2*a[i+2] + 4*a[i+3];
  gettimeofday(&end,NULL);    /*获得循环结束时的系统时间*/
   diff1=(end.tv_sec-start.tv_sec)*1000000+(end.tv_usec-start.tv_usec);
  printf("time1   :   \t   %lu   \n",diff1);;
  /*为防止循环被优化掉所添加的计算和输出*/
  for (i = 0; i < N-3; i++)
    sum=sum+b[i];
  printf("%d\n", sum);
}

int main()
{
  int i, m[1000000];
  for (i = 0; i<1000000; i++)
    m[i] = random()%100;
  fir(m, 1000000);
  return 0;
}
```

图5.4 4-tap FIR测试例程

5.3 实验结果及分析

5.3.1 冒泡排序结果

表 5.2 中列出了 GCC O3，LLVM O3，以及优化后的 LLVM 生成的汇编程序。其中，最左边一列是开发板原生的 GCC 工具链生成的汇编代码，可以看出它使用了如 itt 等平台特有指令，所以代码量比较小，但是真正对程序性能影响最大的是内层循环.L5 这个程序块。.L5 块中有两条 ldr 指令，即 ldr r1, [r3, #0] 和 ldr r4, [r3, #4]! ，由于循环次数比较多，这两条指令造成了巨大的访存延迟。中间一列是 LLVM O3 优化产生的 ARM

汇编。其内层循环是.LBB0_4 块，虽然该块内只有一条 ldr 指令，但是多出两条 SIMD 的 add 指令，以及 mov 指令。经过优化后的程序在表的第三列给出，可以看出内层循环.LBB0_4 中的计算指令被化简了。

表 5.2 优化前后的汇编对比

GCC –O3	LLVM –O3	LLVM Opt

bubbleSort:
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 0, uses_anonymous_args = 0
@ link register save eliminated.
cmp r1, #0
push {r4, r5}
ble .L1
subs r5, r1, #1
.L3:
movs r2, #0
cmp r5, #0
mov r3, r0
ble .L6
.L5:
ldr r1, [r3, #0]
adds r2, r2, #1
ldr r4, [r3, #4]!
cmp r1, r4
itt gt
strgt r4, [r3, #-4]
strgt r1, [r3, #0]
cmp r2, r5
bne .L5
.L6:
adds r5, r5, #-1
bcs .L3
.L1:
pop {r4, r5}
bx lr
	bubbleSort:
	@ BB#0:
	push {r4, r5, r6, r7, lr}
	cmp r1, #1
	blt .LBB0_6
	@ BB#1:
	sub r12, r1, #1
	mov lr, #0
	mov r2, r12
	.LBB0_2:
	sub r3, r12, lr
	cmp r3, #1
	blt .LBB0_5
	@ BB#3:
	ldr r3, [r0]
	mov r4, #0
	.LBB0_4:
	add r6, r0, r4, lsl #2
	add r5, r4, #1
	ldr r7, [r6, #4]
	cmp r3, r7
	strgt r7, [r0, r4, lsl #2]
	strgt r3, [r6, #4]
	movle r3, r7
	mov r4, r5
	cmp r2, r5
	bne .LBB0_4
	.LBB0_5:
	add lr, lr, #1
	sub r2, r2, #1
	cmp lr, r1
	bne .LBB0_2
	.LBB0_6:
	pop {r4, r5, r6, r7, lr}
	mov pc, lr
		bubbleSort:
		@ BB#0:
		push {r4, r5, r6, lr}
		cmp r1, #1
		blt .LBB0_6
		@ BB#1:
		sub r12, r1, #1
		mov lr, #0
		mov r2, r12
		.LBB0_2:
		sub r3, r12, lr
		cmp r3, #1
		blt .LBB0_5
		@ BB#3:
		ldr r4, [r0]
		mov r3, #0
		mov r5, r0
		.LBB0_4:
		ldr r6, [r5, #4]
		add r3, r3, #1
		cmp r4, r6
		strgt r6, [r5]
		strgt r4, [r5, #4]
		movle r4, r6
		add r5, r5, #4
		cmp r2, r3
		bne .LBB0_4
		.LBB0_5:
		add lr, lr, #1
		sub r2, r2, #1
		cmp lr, r1
		bne .LBB0_2
		.LBB0_6:
		pop {r4, r5, r6, lr}
		mov pc, lr

对 10 万个整型数据进行冒泡排序，分别在这些数据排列有序、倒序和随机的情况

下测试执行时间。实验结果在表 5.3 中给出。

表 5.3 冒泡排序测试结果

输入数据	GCC 编译后程序的执行时间(s)	LLVM 编译后程序的执行时间(s)	添加优化的 LLVM 编译后程序的执行时间(s)
有序	74.02	81.98	66.87
倒序	77.02	76.97	67.73
随机	76.53	82.30	68.49

从表 5.3 可以看出，LLVM 添加优化前的效果要比 GCC 差，但是在加入本文的优化后，程序性能要明显好于 GCC，图 5.5 更加直观地反映出这一结果。

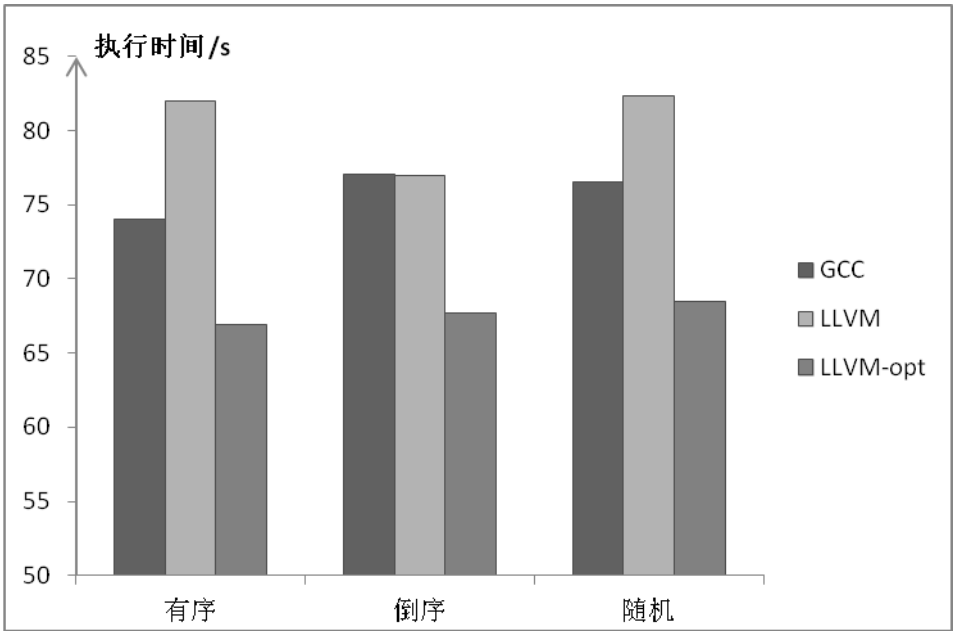

图5.5 GCC和LLVM冒泡排序性能对比

5.3.2 4-tap FIR 滤波程序结果

表 5.4 给出了 4-tap FIR 滤波程序优化前后的汇编结果。从结果中明显可以看出，在优化前，循环体内每次迭代均加载了本次迭代需要的数组元素：*a*[*i*]，*a*[*i*+1]，*a*[*i*+2]，*a*[*i*+3]，而前三个数据于上一次迭代中已经从内存中取出，无需再一次加载；经过优化后的汇编，在 preheader 结点中，加载了最初的三个数组元素，每次迭代只需要新加载 *a*[*i*+3]，大大减少了循环体内的访存数量，提高了程序的性能。

表 5.4 4-tap FIR 滤波优化前后汇编对比

LLVM –O3	LLVM Opt

.LBB0_2: @ %for.body
@ =>This Inner Loop Header: Depth=1
sub r0, r6, #8
sub r1, r6, #4
ldr r0, [r0]
ldr r1, [r1]
add r0, r0, r0, lsl #1
add r1, r1, r1, lsl #2
add r0, r1, r0
ldr r1, [r6]
add r0, r0, r1, lsl #1
ldr r1, [r6, #4]!
add r2, r0, r1, lsl #2
sub r1, r7, #3
add r7, r7, #1
cmp r4, r7
bne .LBB0_2
	@ BB#2: @ %for.body.i.preheader
	add r9, r5, #12
	ldm sp, {r0, r5, r7}
	ldr r8, .LCPI1_1
	mov r4, #0
	.LBB1_3:
	mov r6, r5
	mov r5, r7
	ldr r7, [r9, r4, lsl #2]
	add r0, r0, r0, lsl #1
	add r1, r6, r6, lsl #2
	add r0, r1, r0
	mov r1, r4
	add r0, r0, r5, lsl #1
	add r2, r0, r7, lsl #2
	add r4, r4, #1
	mov r0, r6
	cmp r4, #97
	bne .LBB1_3

对 GCC、LLVM 优化前后的每种编译后程序结果在 ARM 开发板上均进行三次测试，取其平均值，结果如表 5.5 所示。可以看出，添加优化后，程序的执行性能较添加优化前有了很大提升，优化后的结果也要好于 GCC。

表 5.5 4-tap FIR 滤波测试结果

输入数据	GCC 编译后程序的执行时间(us)	LLVM 编译后程序的执行时间(us)	添加优化的 LLVM 编译后程序的执行时间(us)
第一次	28183	39739	25465
第二次	28024	39679	25538
第三次	28093	39725	25541
平均	28100	39714	25514

测试使用的是 100 万个输入，即计算 10^6-3 次 $b[i] = 3*a[i] + 5*a[i+1] + 2*a[i+2] + 4*a[i+3]$，统计各测试结果的平均值，如图 5.6 所示。

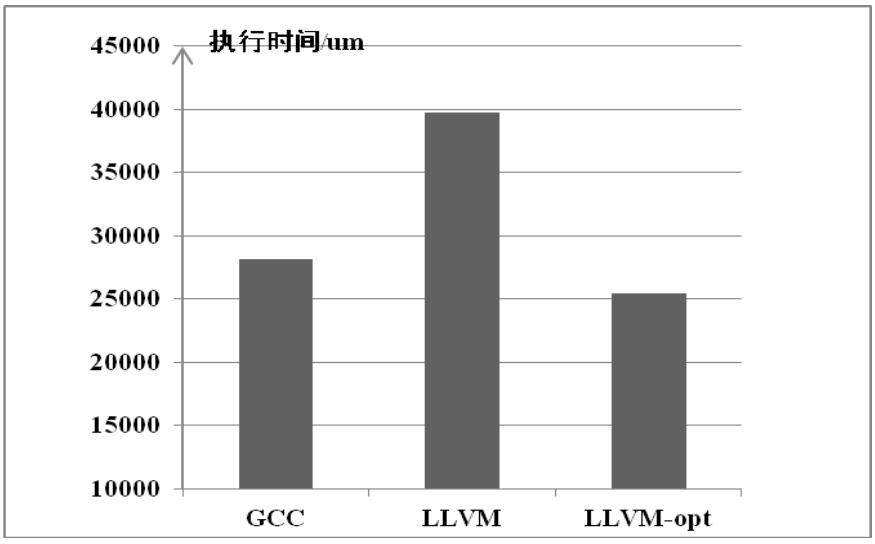

图5.6 4-tap FIR程序测试结果

5.4 本章小结

本章首先介绍了实验所使用的测试平台，分别搭建两种编译器环境，对具体的测试用例（冒泡排序和 4-tap FIR）进行了测试，并分析对比了两种编译器的汇编结果。最后，通过在开发板上实际地执行程序，统计执行时间，验证了优化算法的正确性和有效性。实验结果基本符合预期，达到了实验设计的目的。

结　论

在编译优化技术中，面向循环的优化是最重要的一类优化。本文基于 LLVM 提出了一种面向循环的迭代间数据重用优化，用以简化循环中数组元素引用的计算。围绕此算法，本文主要完成工作如下：

（1）研究现有的循环优化理论，以及这些理论在现有编译器中的应用。

（2）深入 LLVM 编译框架，包括 LLVM 的中间表示、优化方法，以及对循环优化的支持情况。

（3）分析 LLVM 中循环优化的具体实现，主要是归纳变量优化，并同 GCC 的实现作对比，从中找出其实现策略的优势和不足。

（4）设计了一种迭代间数据重用优化算法，结合归纳变量和数组引用标量替换两种优化，改进了 LLVM 处理循环中数组访问的方式。

（5）通过 LLVM 提供的 API，在其编译框架中，进行了优化遍的设计与实现。

迭代间数据重用优化算法主要用于处理循环中数组引用的计算，可减少内层循环的指令数，从而降低整个程序的执行时间。本优化算法改进了 LLVM 循环中访问数组的方式，并通过实验验证了它的可行性和正确性，与 GCC 以及优化前相比，能够达到比较好的效果。本文所研究的结果对编译循环优化有一定的应用价值。

本文的还有后续工作需要完成，主要包括：

（1）将优化模块整合进 LLVM 的归纳变量的优化遍中，在 LLVM 执行默认归纳变量优化时能够更好地利用迭代间的可重用数据，提高循环体的执行效率。

（2）利用 LLVM 规范化提供的框架，需要进一步对规范迭代间数据重用优化算法的适用范围进行约束，以扩展优化算法的适用性。

参考文献

[1] Steven S. Muchnick. Advanced Compiler Design Implementation[M]. Morgan Kaufmann, 1997: 1, 7.

[2] Aho A V. Compilers: principles, techniques, and tools[M]. Pearson Education India, 2007: 9.

[3] Wolf M E, Lam M S. A data locality optimizing algorithm[C]. In Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. New York, NY, USA, 1991: 30-44.

[4] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation[C]. International Symposium on Code Generation and Optimization, Palo Alto, 2004:75-86.

[5] The Architecture of Open Source Applications: LLVM[EB/OL]. [2014-03-04]. http://www.aosabook.org/en/llvm.html.

[6] Wulf W A, McKee S A. Hitting the memory wall: implications of the obvious[J]. ACM SIGARCH computer architecture news, 1995, 23(1): 20-24.

[7] Mahmut Kandemir, Prithviraj Banerjee, Alock Choudhary, J. Ramanujam, Eduard Ayguade. Static And Dynamic Locality Optimizations Using Integer Linear Programming[J]. IEEE Trans Parallel Distrib Syst. 2001, 12(19): 922-941.

[8] Uday Bondhugula, Albert Hatono, J. Ramanujam, P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer[J]. SIGPLAN Not. 2008, 43(6): 101-113.

[9] Li W. Compiling for NUMA parallel machines[R]. Cornell University, 1994.

[10] Ding C, Zhong Y. Predicting whole-program locality through reuse distance analysis[C]. ACM SIGPLAN Notices. 2003, 38(5): 245-257.

[11] 夏军. 数据局部性及其编译优化技术研究[D]. 国防科技大学博士论文. 2004.

[12] 吴俊杰. 层次存储的访问分析与优化方法研究[D]. 国防科技大学博士论文. 2009.

[13] 唐滔, 杨学军, 林一松. 基于迭代序的流程序局部性分析和优化[J]. 计算机研究与发展. 2012.49(6): 1363-1375.

[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] Wolfe M. Beyond induction variables[C]. ACM SIGPLAN Notices. 1992, 27(7): 162-174.

[16] Haghighat M R, Polychronopoulos C D. Symbolic analysis for parallelizing compilers[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1996, 18(4): 477-518.

[17] Gerlek M P, Stoltz E, Wolfe M. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1995, 17(1): 85-122.

[18] Van Engelen R A. Efficient symbolic analysis for optimizing compilers[C]//Compiler Construction. Springer Berlin Heidelberg, 2001: 118-132.

[19] Allen F E, Cocke J, Kennedy K. Reduction of operator strength[J]. Program Flow Analysis, 1981: 79-101.

[20] Cooper K D, Simpson L T, Vick C A. Operator strength reduction[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2001, 23(5): 603-625.

[21] 周雷, 陈克非. 基于符号运算的归纳变量识别与约化[J]. 计算机工程. 2010, 36(24): 70-71.

[22] 刘杰, 曹琰, 魏强. 彭建山符号执行中的循环依赖分析方法[J]. 计算机工程. 2012, 38(22): 24-27.

[23] Locations Around the World - ARM[EB/OL]. [2014-03-04]. http://www.arm.com/zh/careers/ life-at-arm/locations-around-the-world.php.

[24] Open Source Toolchain Blog | 开源工具链相关技术[EB/OL]. [2014-03-04]. http://www.hellogcc.org/.

[25] Software System Award - Award Winners: List By Year[EB/OL]. [2014-03-04]. http://awards.acm.org/software_system/year.cfm.

[26] Apple Developer[EB/OL]. [2014-03-04]. https://developer.apple.com/.

[27] Oracle Labs | Project Details[EB/OL]. [2014-03-04]. http://research.sun.com/pls/apex/f?p=labs:49:::::P49_PROJECT_ID:13.

[28] Alchemy - Adobe Labs[EB/OL]. [2014-03-04]. http://labs.adobe.com/technologies/alchemy/.

[29] Nadav Rotem.Intel® OpenCL Implicit Vectorization Module[C]. LLVM Developers'

Meeting. 2011.11.

[30]The LLVM Compiler Infrastructure Project[EB/OL]. [2014-03-04]. http://llvm.org/Users.html.

[31]Modula Tips[EB/OL]. [2014-03-04]. http://modula2.net/.

[32]PyPy - Welcome to PyPy[EB/OL]. [2014-03-04]. http://pypy.org/.

[33]What is Faust?[EB/OL]. [2014-03-04]. http://faust.grame.fr/index.php/documentation/what-faust.

[34]韩永杰. LLVM编译系统结构分析及ARCA3后端移植[D]. 哈尔滨工业大学硕士论文. 2010.

[35]董峰. LLVM 编译系统结构分析与后端移植[D]. 上海交通大学硕士论文. 2007.

[36]卢念. 基于 LLVM 的 Nios II 处理器后端快速移植及优化[D]. 中南大学. 2011.

[37]杨敏, 吴艳霞, 顾国昌等. 面向可重构编译技术的 RAM 访问优化算法[J]. 计算机工程. 2011, 37(2).

[38]Shobaki G, Shawabkeh M, Rmaileh N E A. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2013, 10(3): 14.

[39]Jablin T B, Prabhu P, Jablin J A, et al. Automatic CPU-GPU communication management and optimization[J]. ACM SIGPLAN Notices, 2011, 46(6): 142-151.

[40]Zhao J, Nagarakatte S, Martin M M K, et al. Formalizing the LLVM intermediate representation for verified program transformations[C]. ACM SIGPLAN Notices. 2012, 47(1): 427-440.

[41]Allen F E, Cocke J, Kennedy K. Reduction of operator strength[J]. Program Flow Analysis, 1981: 79-101.

[42]Bachmann O, Wang P S, Zima E V. Chains of recurrences — a method to expedite the evaluation of closed-form functions[C]. Proceedings of the international symposium on Symbolic and algebraic computation. ACM, 1994: 242-249.

[43]Bachmann O. Chains of recurrences[D]. Kent State University, 1996.

[44]Sreedhar V C, Gao G R. A linear time algorithm for placing phi-nodes[C]. Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1995: 62-73.

[45]LLVM Command Guide — LLVM 3.4 documentation[EB/OL]. [2014-03-04].

http://llvm.org/docs/CommandGuide/index.html.

[46]Writing an LLVM Backend — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/WritingAnLLVMBackend.html.

[47]The LLVM Target-Independent Code Generator — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/CodeGenerator.html.

[48]LLVM Language Reference Manual — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/LangRef.html.

[49]LLVM's Analysis and Transform Passes — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/Passes.html.

[50]LLVM Programmer's Manual — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/ProgrammersManual.html.

[51]LLVM Coding Standards — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/CodingStandards.html.

[52]LLVM: LLVM[EB/OL]. [2014-03-04]. http://llvm.org/doxygen/.

[53]Writing an LLVM Pass — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/WritingAnLLVMPass.html.

[54]Xilinx Wiki - Install Xilinx Tools[EB/OL]. [2014-03-04]. http://www.wiki.xilinx.com/Install+Xilinx+Tools.

[55]Getting Started with the LLVM System — LLVM 3.4 documentation[EB/OL]. [2014-03-04]. http://llvm.org/docs/GettingStarted.html.

攻读硕士学位期间发表的论文和取得的科研成果

致　　谢

两年多的研究生学习生活即将结束，值此论文完成之际，谨向在我攻读硕士学位期间所有关心和指导过我的老师、同学致以最诚挚的感谢！

衷心感谢我的导师吴艳霞老师。本课题从选题、设计到论文撰写的整个过程，老师都给予了孜孜不倦的指导。正是吴老师在这两年时光中的耐心指导和帮助，才使我顺利完成了硕士学业和本次毕业设计。在此，向我的老师表示衷心的感谢，祝老师工作顺利，祝实验室越来越好。

感谢孙延腾、张祖羽、张博为、郭振华师哥，牛晓霞师姐，以及王泽宇、安龙飞、隋天祥、王彦璋、孟凡印同学在课题完成过程中给予我的帮助。感谢 325、336、337 实验室的所有同学，在朝夕相处的日子里，他们带给了我丰硕的知识和无尽的欢乐，祝实验室的大家前程似锦。

感谢我的父母和亲人，他们无私的关怀和默默的支持是我前进的动力。祝他们身体健康，天天快乐。

谨祝哈尔滨工程大学计算机科学与技术学院的明天更加美好！