软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(7):3022−3042 [doi: 10.13328/j.cnki.jos.006858]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 LLVM Pass 的复杂嵌套循环自动并行化框架*

马春燕[1], 吕炳旭[1], 叶许姣[1], 张 雨[2]

[1](西北工业大学 软件学院, 陕西 西安 710129)

[2](海南大学 计算机科学与技术学院, 海南 海口 570228)

通信作者: 张雨, E-mail: yuzhang2015@hainanu.edu.cn

摘 要: 随着多核处理器的普及应用, 针对嵌入式遗留系统中串行代码的自动并行化方法是研究热点. 其中, 针对具有非完美嵌套结构、非仿射依赖关系特征的复杂嵌套循环的自动并行化方法存在技术挑战. 提出了一种基于 LLVM Pass 的复杂嵌套循环的自动并行化框架(CNLPF). 首先, 提出了一种复杂嵌套循环的表示模型, 即循环结构树, 并将嵌套循环的正则区域自动转换为循环结构树表示; 然后, 对循环结构树进行数据依赖分析, 构建循环内和循环间的依赖关系; 最后, 基于OpenMP共享内存的编程模型生成并行的循环程序. 针对SPEC2006数据集中包含近 500 个复杂嵌套循环的 6 个程序案例, 分别对其进行复杂嵌套循环占比统计和并行性能加速测试. 结果表明, 提出的自动并行化框架可以处理 LLVMPolly 无法优化的复杂嵌套循环, 增强了 LLVM 的并行编译优化能力, 且该方法结合 Polly 的组合优化, 比单独采用 Polly 优化的加速效果提升了 9%–43%.

关键词: 复杂嵌套循环; 自动并行化; LLVM Pass; 依赖分析

中图法分类号: TP311

中文引用格式: 马春燕, 吕炳旭, 叶许姣, 张雨. 基于 LLVM Pass 的复杂嵌套循环自动并行化框架. 软件学报, 2023, 34(7): 3022−3042. http://www.jos.org.cn/1000-9825/6858.htm

英文引用格式: Ma CY, Lü BX, Ye XJ, Zhang Y. Automatic Parallelization Framework for Complex Nested Loops Based on LLVM Pass. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3022−3042 (in Chinese). http://www.jos.org.cn/1000-9825/6858.htm

Automatic Parallelization Framework for Complex Nested Loops Based on LLVM Pass

MA Chun-Yan[1], LÜ Bing-Xu[1], YE Xu-Jiao[1], ZHANG Yu[2]

[1](School of Software, Northwestern Polytechnical University, Xi'an 710129, China)

[2](School of Computer Science and Technology, Hainan University, Haikou 570228, China)

Abstract: With the popularization of multi-core processors, automatic parallelization of serial codes in embedded legacy systems is a research hotspot, while there are technical challenges in the automatic parallelization method for complex nested loops with imperfect nested structure and non-affine dependency characteristics. This study proposes an automatic parallelization framework (CNLPF) for complex nested loops based on LLVM Pass. Firstly, a representation model of complex nested loops, namely loop structure tree, is proposed, and the regular region of nested loops is automatically converted into a loop structure tree representation. Then, the data dependency analysis is carried out on the loop structure tree to construct intra-loop and inter-loop dependency relationship. Finally, the parallel loop program is generated based on the OpenMP shared memory programming model. For the 6 program cases in the SPEC2006 data set containing nearly 500 complex nested loops, the statistics of the proportion of complex nested loops and the parallel performance acceleration test were carried out respectively. The results show that the automatic parallelization framework proposed in this study can deal with complex nested loops that cannot be optimized by LLVM Polly, which enhances the parallel compilation and optimization capabilities of LLVM, and the method combined with Polly optimization improves the acceleration effect of Polly optimization alone by 9%–43%.

Key words: complex nested loop; automatic parallelization; LLVM Pass; dependency analysis

* 基金项目: 国家自然科学基金(62192733, 62062030); 航空基金(20185853038, 201907053004)
本文由"形式化方法与应用"专题特约编辑董云卫教授、刘关俊教授、毛晓光教授推荐.
收稿时间: 2022-09-04; 修改时间: 2022-10-08; 采用时间: 2022-12-05; jos 在线出版时间: 2022-12-30

多核处理器的发展, 为现代嵌入式系统应用的并行化带来了机遇.

- 一方面, 嵌入式的高实时性要求让并行优化迎来机遇. 随着我国航天事业的快速发展, 嵌入式软件在航天器中的作用和地位越来越突出, 航天器上很多关键、复杂的功能都逐渐由嵌入式软件来实现[1]. 对于这些嵌入式应用程序来说, 实时性能力是一个不可或缺的基本要求, 其要求计算要在系统给定的时间内做出响应, 否则可能发生致命错误, 例如自动驾驶中的刹车制动系统[2]等. 利用好多核处理器提供的并行能力, 可以有效减少嵌入式系统响应时间, 提高系统实时性.
- 另一方面, 嵌入式系统应用的可并行化空间巨大, 许多嵌入式遗留系统仍然是以顺序方式编写的串行代码. 如何有效地发掘串行代码中的并行性, 让串行代码并行化执行, 是一个值得研究的问题.

为使得嵌入式系统从多核中受益, 串行执行的任务需要被划分为并行执行的任务. 手动划分并行分区是一项容易出错且耗时的工作, 因此, 串行程序自动并行化[3]的研究一直在持续推进. 随着多核处理器的出现和发展, 学术界涌现出许多不同并行层次上的自动并行化方法, 例如指令级并行性、超字级并行性[4]、循环级并行性、函数级并行性等. 在嵌入式应用程序中, 循环是程序执行时间占比较大的部分, 因此, 解决好应用程序中循环层次的自动并行化问题, 可以为应用程序带来极大的性能提升.

循环级自动并行化是一个综合了程序语言、静态分析、并行编译、高性能计算等多个研究领域的复杂问题. 自循环自动并行化领域发展以来, 虽然在完美嵌套循环[5]、仿射嵌套循环[6]方面取得了较大进展, 但是复杂嵌套循环的并行化依然是一个难点. 本文对嵌入式软件中复杂嵌套循环特征总结为下述 3 个方面.

(1) 循环嵌套结构的复杂性. 完美嵌套循环仅允许每层至多嵌套一个 for 循环结构, 且仅允许最内层循环存在非循环迭代语句. 相比完美嵌套循环, 复杂嵌套循环允许 for、while 循环的任意嵌套结构, 且任意层循环都允许存在非循环迭代语句.

(2) 循环中语句之间依赖的复杂性. 复杂嵌套循环拥有复杂的数据依赖和控制依赖. 复杂数据依赖既有单个循环内语句之间的数据依赖, 也包括不同层级循环之间的数据依赖. 复杂控制依赖则主要来自循环层次结构和条件分支结构.

(3) 循环中语句类型的复杂性. 仿射嵌套循环仅允许下标仿射的数组运算和数组赋值语句, 这大多适用于数据科学应用, 但并不适用于嵌入式应用. 复杂嵌套循环对循环语句类型基本没有限制, 允许存在非仿射数组访问、函数调用、指针引用等难以分析的语句.

为了解决复杂嵌套循环的自动并行化问题, 本文提出了一种基于 LLVM Pass[7]的复杂嵌套循环自动并行化框架. 本文首先提出了一种复杂嵌套循环的表示模型——循环结构树及其构建方法, 然后提出了基于循环结构树的依赖分析方法, 最后提出了并行循环程序自动生成方法. 该并行化框架基于 LLVM (low level virtual machine)编译器的 Pass 扩展机制, 针对低级别的 LLVM IR (intermediate representation)代码中的复杂嵌套循环进行检测识别、依赖分析, 并将串行循环程序自动转换为并行循环程序.

本文的主要贡献包括:

(1) 提出了一种针对复杂嵌套循环的自动并行化框架. 本框架可以处理 LLVMPolly 无法优化的复杂嵌套循环, 且可与 LLVM Polly 组合优化, 使加速比提升 9%–43%, 增强 LLVM 的并行编译优化能力.

(2) 提出了一种新的复杂嵌套循环的表示模型, 即循环结构树, 并给出循环结构树的自动构建方法. 此模型可以表示 LLVMPolly[8]中多面体模型[9]无法表示的复杂嵌套循环, 且模型易于理解和构建.

(3) 提出了一种基于循环结构树模型的复杂嵌套循环的依赖分析方法. 以往的循环依赖分析局限于单个循环内迭代间的依赖分析, 但是忽略了循环块间的依赖分析. 此方法不仅考虑了单个循环内的依赖性, 也考虑了不同循环块之间的依赖性, 发掘了循环内和循环间的并行可能性.

本文第 1 节介绍相关工作. 第 2 节介绍 LLVM 的中间表示 IR. 第 3 节给出自动并行化框架的概览. 第 4 节详细阐释并行化框架中各个部分的模型定义与方法, 包括第 4.1 节的正则区域检测方法、第 4.2 节的循环结构树的定义与构建方法、第 4.3 节的依赖分析方法和第 4.4 节的并行代码生成方法. 第 5 节通过在 SPEC CPU 2006 测试集上进行实验验证, 说明本文方法的有效性. 第 6 节对全文进行总结和展望.

1 相关工作

循环级自动并行化领域中, 经典的研究工作是基于多面体模型的循环优化. 具有代表性的工作是 Bondhugula 等人于 2008 年在文献[10]中提出的一个自动多面体源到源转换框架, 此框架可以同时优化规则程序的并行性和局部性. 这项工作具有划时代意义, 著名的 Pluto 编译器就是出自这篇文章. 后来, 很多主流编译器中的多面体优化功能都参考了这项工作, 包括但不限于 GCC Graphite[11]、LLVM Polly[8]和 MLIR Polygeist[12]. Bondhugula 于 2013 年在文献[6]提出了一种在分布式内存并行体系结构中编译具有仿射依赖关系的任意嵌套循环的新技术. Bondhugula 使用多面体模型生成以消息传递接口(MPI)表示的并行代码. 同年, Nadezhkin 等人[13]首次提出了将具有动态循环边界的仿射嵌套循环程序自动转换为等效多面体过程网络的方法, 揭示了此类应用程序中可能存在的并行性. 基于多面体模型的循环并行优化方法已趋于成熟, 目前, 很多主流编译器都已经内置了多面体优化功能.

然而, 随着嵌入式系统的快速发展, 应用程序的规模越来越大、复杂性越来越高, 多面体循环优化技术的局限性慢慢暴露. 多面体模型对循环本身的要求过于严苛, 这些要求包括但不限于循环头部必须符合一定格式、必须是 For 循环、只有数组相关的赋值语句才是目标对象语句等. 这些限制使得多面体优化技术适用于处理部分数值计算型循环, 但嵌入式系统应用还存在大量非数值计算型的复杂嵌套循环无法得到有效的并行化. 为了提升循环级自动并行化的能力, 近 10 年来, Geuns 等人[14]在 2011 年介绍了一种包含未知迭代上界的 while 循环的嵌套循环程序的自动并行化方法; Palkowski 等人[15]在 2015 年提出了一种源到源的 TRACO 编译器, 这种编译器利用依赖图的传递闭包, 为选定的数值应用程序中的嵌套循环自动生成并行平铺的代码, 进而提高程序局部性和并行性; Zarei 等人[5]在 2017 年研究了异构分布式系统中 3 级完美嵌套循环的三维数据划分问题, 提出了一种启发式分区方法来求解三维数据分区, 并通过两个数据并行应用对方法进行实验, 实现了近似线性的加速; 同年, 丁丽丽等人[16]提出了一种能够处理分支嵌套循环的依赖测试方法, 在一定程度上扩展了循环级自动并行化的能力; Azadeh 等人[17]在 2019 年提出了一种为顺序嵌套循环自动生成模块化并行循环的方法, 通过抽象内部循环, 产生一个更简单的并行循环嵌套版本, 然后对这个新版本的嵌套循环进行并行化, 此方法可以有效地并行高度非平凡的循环嵌套; Abdi 等人[18]在 2020 年的一篇文献中提出了一种新的循环平铺算法(*k*-StepIntraTiling 算法)来解决具有非一致依赖关系的两级完美嵌套循环的并行化问题.

目前, 针对复杂嵌套循环并行化的相关研究, 存在以下挑战: (1) 相比 Bondhugula 提出了多面体源到源的代表性转换框架, 缺乏针对复杂嵌套循环的自动并行框架的设计; (2) 相比多面体模型可以表示仿射嵌套循环, 复杂嵌套循环缺乏表示模型; (3) 相比仿射嵌套循环的基于多面体模型的依赖分析, 缺乏对复杂嵌套循环依赖的有效分析方法. 这也是本文工作的动机. 针对挑战(1), 本文提出了基于 LLVM Pass 的复杂嵌套循环的自动并行化框架; 针对挑战(2), 本文提出了循环结构树对复杂嵌套循环结构进行建模; 针对挑战(3), 本文提出了基于循环结构树的循环内和循环间的依赖分析方法.

2 LLVM 中间表示 IR

LLVM 是模块化、可重用的编译器和工具链技术的集合[8]. 本文关注于 LLVM 的中端优化部分, 基于 LLVM 中端的 Pass 扩展机制来构建复杂嵌套循环的自动并行化框架.

LLVM IR 是一个类 RISC 指令集的低层次的编程语言, 可以用来表示高级语言信息, 且具有语言无关性, 适用于高效的代码优化. IR 的布局自顶向下大致分为模块、函数、基本块和指令, 其将一个源代码文件视为一个模块, 源代码中的全局变量和函数在 IR 中依然表示为全局变量和函数, 函数内的语句由多个基本块顺序组成, 每个基本块由多条指令序列组成.

本文要讨论的循环属于 IR 的基本块级别的代码块, 由多个特定模式的基本块组成, 这些基本块分别是 Preheader 块、Header 块、Latch 块、Exiting 块以及 Exit 块. 控制循环开始和结束的基本块称为 Header 块, 其支配[19]了循环中所有其他节点, 即从循环外到循环内任何基本块的执行路径都必须经过 Header 块. Preheader 块是一个循环外的基本块, 具有唯一指向 Header 块的边, 其主要作用是使循环成为单入口的区域. Preheader

块往往对循环中的共享变量进行初始化, 没有变量初始化时仅保留一条跳转指令. Latch 块是一个循环内的基本块, 其具有到 Header 块的边, Latch 块到 Header 块的边被称为 Back edge. Exiting edge 是从循环内部到循环外部的边. Exiting edge 的入口基本块称为 Exiting 块, 边的出口基本块称为 Exit 块. Exiting 块属于循环内的基本块, 而 Exit 块属于循环外的基本块. 因为循环正常结束时一定是从循环头部退出, 所以 Header 块是一个天然的 Exiting 块. 一个循环中可以存在多个 Exiting 块, 所有 Exiting 块都指向同一个 Exit 块. 从 Header 块到 Exiting 或 Latch 块之间是循环的 Body 部分, 是一系列合法基本块. 图 1 展示了循环在 IR 中的通用表示形式. 对于 for、while-do 循环来说, 图 1 中的通用表示形式都适用, 但是对于 do-while 循环, Preheader 块之后并非 Header 块, 此时需要将 do-while 循环修改为 while-do 结构, 即 do-while 循环的首次迭代分离到循环之外, 将循环内基本块复制并展开到 Preheader 块之前, 然后将 Header 块提前到 Preheader 块之后, 即可将 do-while 循环转换为图 1 的统一形式. 鉴于 for 循环或 while 循环的 IR 表示形式无差别, 后文将以 for 循环为例介绍相关方法.

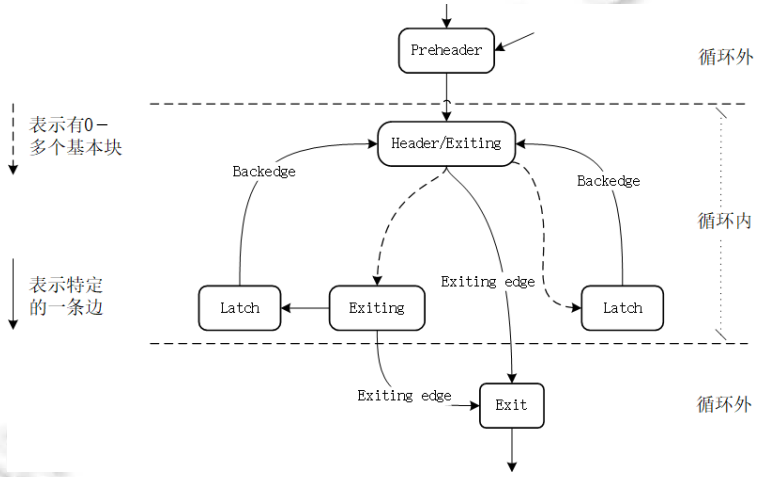


图 1　循环在 IR 中的通用表示形式

条件分支也属于基本块级别, 由多个特定模式的基本块组成, 这些基本块是 Header 块、Exiting 块和 Exit 块. 图 2 展示了条件分支在 IR 中通用表示形式. 各个基本块的概念和图 1 中的循环类似, 其中, 虚线表示 0 到多个合法基本块的串联, 当虚线路径中没有基本块时, Header 块和 Exiting 块同属于一个基本块, Header 块的左边或右边直接指向 Exit 块.

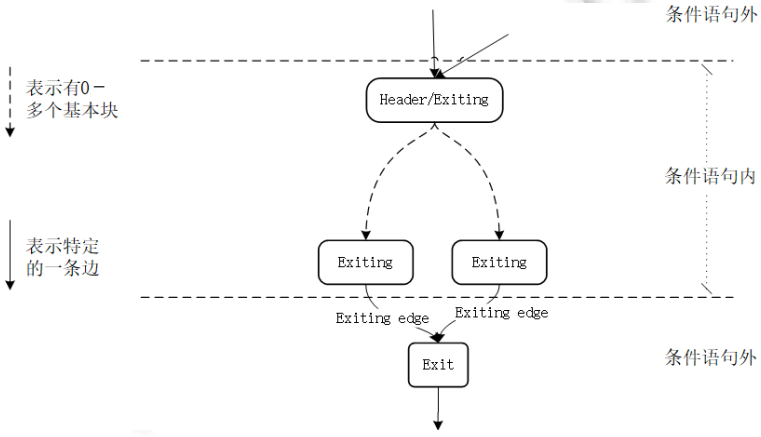


图 2　条件分支在 IR 中通用表示形式

3 框架概览

为了描述本文的复杂嵌套循环并行化框架, 首先给出本文复杂嵌套循环相关的定义 3.1–3.4.

定义 3.1(循环嵌套层数). 对于一个多层嵌套循环, 循环嵌套层数是最外层循环到所有最内层循环的嵌套深度的最大值.

定义 3.2(完美嵌套循环). 如果一个 N 层嵌套循环满足以下递归定义, 则该嵌套循环是完美嵌套循环.

(1) 如果 N=1, 该循环是一个单层 for 循环, 循环体至少有一条语句, 且循环体内无过程间调用、跳转语句或指针引用语句.

(2) 如果 N>1, 该循环是一个多层嵌套循环, 其最外层循环是一个 for 循环, 且最外层循环的循环体是一个 N–1 层的完美嵌套循环.

定义 3.3(仿射嵌套循环). 如果一个 N 层嵌套循环满足以下性质, 则该嵌套循环是仿射嵌套循环.

(1) 任意层循环都是 for 循环, 且循环上下界是循环迭代变量的仿射函数.

(2) 循环迭代中数组的索引是循环迭代变量的仿射函数.

(3) 条件分支的条件表达式是循环迭代变量的仿射不等式.

(4) 循环中不存在过程间调用、流程跳转语句以及非仿射的循环迭代变量的赋值语句.

完美嵌套循环侧重从嵌套结构上定义, 而仿射嵌套循环侧重从数据访问的模式上定义, 因此, 它们的定义具有重叠, 即可能存在一个 N 层嵌套循环既是完美嵌套循环, 又是仿射嵌套循环.

定义 3.4(复杂嵌套循环). 如果一个 N 层嵌套循环既不是完美嵌套循环, 也不是仿射嵌套循环, 且满足如下约束条件, 则本文称其为复杂嵌套循环.

(1) 针对嵌套结构的复杂性, 允许 for 循环和 while 循环的任意嵌套结构, 但循环需要满足上下界静态可知, 且循环迭代变量是归纳变量, 即迭代步长固定.

(2) 针对循环依赖的复杂性, 允许任意合法的数据依赖, 允许由条件分支语句和循环语句构成的控制依赖, 但是不允许条件分支嵌套循环, 也不允许出现跳转语句构成的控制依赖, 因为其动态特性会使得循环上下界静态不可知, 违背约束条件(1).

(3) 针对语句类型的复杂性, 允许值传递的过程调用, 但不允许有引用传递的过程调用语句.

针对本文定义的复杂嵌套循环, 基于 LLVMPass 扩展机制的自动并行化框架由循环检测、依赖分析、并行代码生成 3 个部分组成, 如图 3 所示.

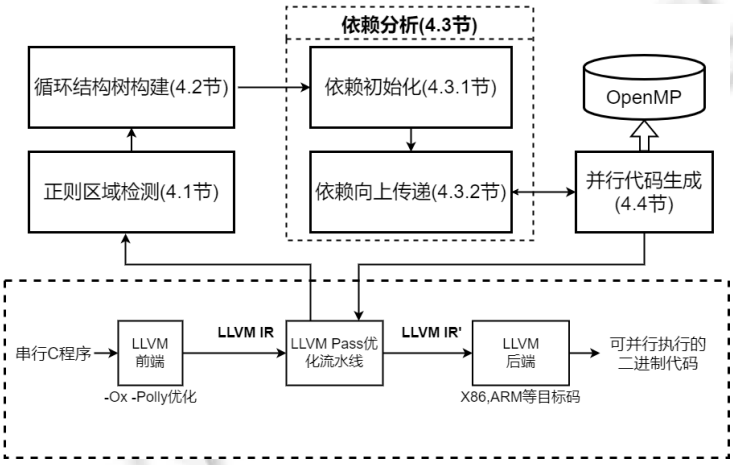


图 3 基于 LLVMPass 的复杂嵌套循环自动并行化框架

循环检测包括正则区域检测和循环结构树构建. 正则区域检测将在 IR 中搜索满足上述约束(1)–(3)的最大正则区域, 第 4.1 节将详细阐述对正则区域的规范定义和检测识别方法. 循环结构树构建将检测出的正则区

域构建为本文提出的一种复杂嵌套循环的表示模型——循环结构树, 第 4.2 节将给出循环结构树的定义和将正则区域构建为循环结构树的方法.

依赖分析包括初始化依赖关系和向上传递依赖关系两个阶段: 在初始化依赖阶段, 循环结构树中块代码节点依赖由内存指令依赖初始化获得, 第 4.3.1 节将给出循环结构树节点依赖的定义和依赖初始化的方法; 在依赖向上传递阶段, 初始化的依赖关系向循环结构树的循环头节点传递, 第 4.3.2 节将详细阐述向上传递的具体方法.

并行代码生成以共享内存模型的 OpenMP 为支撑, 遍历循环头节点, 依据依赖分析的结果, 使用不同的并行指令对循环程序进行并行标注, 然后交由 LLVM 基础设施进行并行化的二进制代码的生成和执行. 本文将在第 4.4 节介绍这部分的内容.

4 基于 LLVM Pass 的复杂嵌套循环并行化框架

4.1 嵌套循环的正则区域检测方法

正则区域的检测过程是对 IR 所有函数中的顶层嵌套循环进行约束检测的过程. 为了搜索最大正则区域, 检测过程从最内层向最外层检测, 直到首次检测到非正则区域停止, 此时, 内层正则区域即为最大正则区域.

定义 4.1(正则区域、最大正则区域). 对于一个 N 层嵌套循环, 若存在一个 k 层子嵌套循环($k \leqslant N$)满足定义 3.4 的约束条件(1)–(3), 则称其为正则区域. 若此 k 层子嵌套循环的 k+1 层父嵌套循环不存在或不满足正则区域的定义, 则此 k 层子嵌套循环为最大正则区域.

针对复杂嵌套循环的约束条件(1), 正则区域检测方法需要识别循环上下界是否是静态可知以及循环迭代变量是否是基本归纳变量.

- 第 1 步, 找到循环 Header 块中判断条件所涉及的所有变量.
- 第 2 步, 在循环内寻找循环迭代变量的写入位置. 因为循环上下界需要静态可知, 所以除了循环迭代变量之外, 其他变量在循环内只读不写. 从循环 Header 开始, 在循环 Body 中遍历基本块, 并将与内存写入指令所操作的变量比对, 如果仅有循环迭代变量被写入且写入位置唯一, 则当前循环满足约束(1).
- 第 3 步, 分析循环迭代变量写入位置所在基本块, 判断写入模式是否符合基本归纳变量的定义(如算法 1 的第 32–35 行).

针对复杂嵌套循环的约束条件(2)中不允许出现跳转语句构成的控制依赖, 正则区域检测方法主要对条件分支、循环的结构特性进行分析. 因为本文中正则区域的循环不允许跳转语句, 所以有且仅有一个 Exiting 块, 即, 循环除了 Header 之外没有其他出口. 图 4 展示了正则区域的循环在 IR 中的表示形式以及一个例子. 通过检测循环内是否有指向 Exit 块的跳转指令来判断循环是否满足约束(如算法 1 的第 26–28 行).

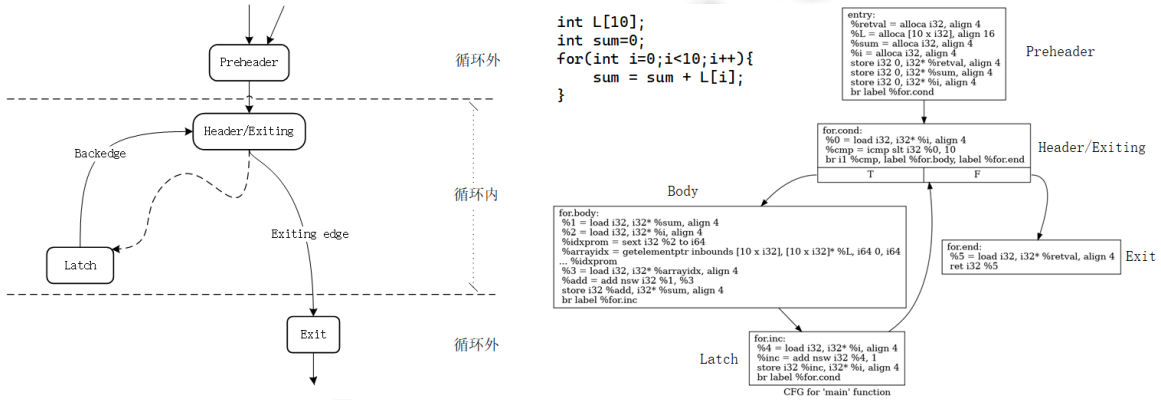
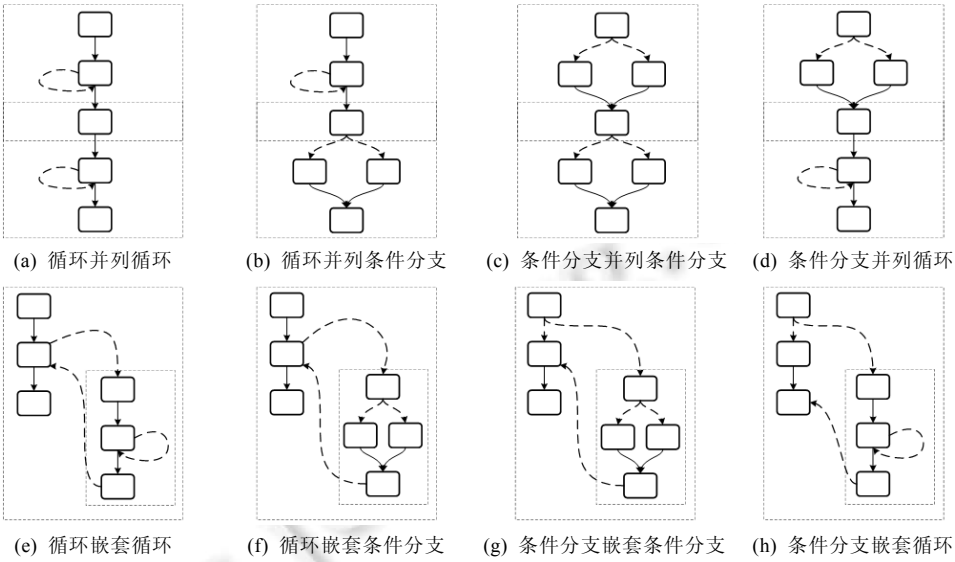


图 4 正则区域的循环在 IR 中的表示形式以及一个例子

针对复杂嵌套循环的约束条件(2)中不允许条件分支嵌套循环, 正则区域检测方法主要对条件分支、循环的排布方式进行分析. 如图 5 所示, 除了图 5(h)的条件分支嵌套循环的排布方式不被允许之外, 其他排布方式(图 5(a)–(g))都满足约束. 为了检测条件分支中是否存在循环, 需要在向内遍历到循环结构时做标记(如算法 1 的第 13 行), 并在向外返回遇到条件分支结构时判断是否具有标记, 如果有, 则不满足约束(如算法 1 的第 16–23 行).



(a) 循环并列循环 (b) 循环并列条件分支 (c) 条件分支并列条件分支 (d) 条件分支并列循环

(e) 循环嵌套循环 (f) 循环嵌套条件分支 (g) 条件分支嵌套条件分支 (h) 条件分支嵌套循环

图 5 条件分支与循环的 8 种排布方式

针对复杂嵌套循环的约束条件(3), 正则区域检测方法主要对循环是否包含引用传递的过程调用进行分析. 在遍历基本块过程中, 对 call 指令的参数进行分析即可(如算法 1 的第 29–31 行).

算法 1 描述了对一个顶层嵌套循环进行正则区域检测的过程. 算法的输入是串行程序的某个顶层循环的 Header 块, 输出是此顶层循环中的所有最大正则区域的 Header 块的集合 *Collection*. 整个算法自底向上地检测当前循环是否为正则区域, 且检测过程中仅记录最大正则区域(第 12、38 行).

算法 1. 正则区域检测(*CheckRegularArea*).

输入: 顶层循环的 *Header*.

输出: 所有最大正则区域的集合 *Collection*.

```
1.  Function CheckRegularArea(Header)
2.  Collection←∅
3.      __CheckRegularArea(the first block of Header's Body, Header,
4.          get variables from icmp instruction of Header, the exit block of Header)
5.      Return Collection
6.  Function __CheckRegularArea(start,end,vals,exit)
7.  check←true, hasloop←false, matched←false, T←∅
8.      While start≠end
9.      If start is LoopHeaderBlock then
10. res←CheckRegularArea(the first block of start's Body,
11.     start, get variables from icmp instruction of start, the exit block of start)
12.     If res.check=true then add start to T
```

13. *check*←*check* and *res.check*, *hasloop*←true
14. *start*←the exit block point of *start*
15. continue
16. **If** *start* is *CondHeaderBlock* **then**
17. *resl*←*CheckRegularArea*(the first block of *start*'s left Cond,
18. the exit block of *start*, *vals*, *exit*)
19. *resr*←*CheckRegularArea*(the first block of *start*'s right Cond,
20. the exit block of *start*, *vals*, *exit*)
21. *check*←*check* and *resl.check* and *resr.check*
22. *hasloop*←*hasloop* or *resl.hasloop* or *resr.hasloop*
23. **If** *hasloop*=true **then** *check*←false //约束(2): 不允许条件嵌套循环
24. *start*←the exit block of *start*
25. continue
26. **If** the next block of *start*=*exit* **then** //约束(2): 不允许跳转语句
27. *check*←false
28. break
29. **If** *start* include call instruction **then** //约束(3)
30. analyze *parameters* of call instruction
31. **If** *parameters* exists reference type **then** *check*←false
32. **If** *start* include store instruction **then** //约束(1)
33. *M*←match *vals* with all store variables
34. **If** *matched*=true and *M*≠∅ **then** *check*←false
35. **If** (*size*(*M*)=1) and (*M*[0] is inductive) and (*matched*=false) **then** *matched*←true
36. *start*←the next block of *start*
37. *check*←*check* and *matched*
38. **If** *check*=false **then** merge *T* into *Collection*
39. Return {*check*,*hasloop*}

算法 1 的核心是__CheckRegularArea 函数, 其中,

- *start* 参数表示循环内或条件分支内的起始基本块. 对于循环来说, *start* 是循环体内第 1 个基本块; 对于条件分支来说, *start* 是左右分支的第 1 个基本块.
- *end* 参数表示循环内或条件分支内的结束基本块. 对于循环来说, *end* 是循环的 Header 块; 对于条件分支来说, *end* 是条件分支的 Exit 块.
- *vals* 参数表示循环 Header 块中 icmp 指令相关的变量集合. *vals* 是为约束(1)设计的参数, 当递归进入子循环时, 需要传入子循环的 *vals*; 当递归进入条件分支时, 需要传入当前循环的 *vals*.
- *exit* 参数表示循环的 Exit 块. 当在条件分支内时, *exit* 表示的是外层循环的 Exit 块, 而非条件分支的 Exit 块. *exit* 是为约束(2)中不允许跳转语句限制而设计的参数.
- 返回值 *check* 表示从 *start* 到 *end* 的区域是否是正则区域, 若当前区域的子区域有任意一个不是正则区域, 则当前区域也不是正则区域(第 13、21 行).
- 返回值 *hasloop* 是为约束(2)中不允许条件分支嵌套循环限制而设计的返回值(第 23 行).

算法 1 本质上是对某个顶层循环中所有嵌套层级循环的深度优先遍历. 假设某个顶层循环是由 *M* 个基本块组成的 *N* 层嵌套循环, 则因为需要遍历所有基本块, 所以算法 1 的时间复杂度为 O(*M*); 同时, 深度优先的递归栈长度是循环嵌套的最大深度, 因此, 算法 1 的空间复杂度为 O(*N*).

4.2 循环结构树模型定义与构建方法

4.2.1 复杂嵌套循环的循环结构树模型定义

本文将复杂嵌套循环抽象为两类元素的层次结构, 如定义 4.2 所示: 一类是循环头, 其包含了当前循环的迭代变量、迭代域、迭代步长等信息; 另一类是循环块代码, 其是由多个连续基本块组成的语句序列, 被同一层级的多个循环头分割为不同的块代码. 这两类元素可以以任意的层次结构进行排布组合, 本文用结构化的树来表示这种层次结构, 称其为循环结构树, 见定义 4.3. 图 6 直观地展示了一个由 6 个循环头和 7 个循环块代码排布组成的嵌套循环及其循环结构树表示.

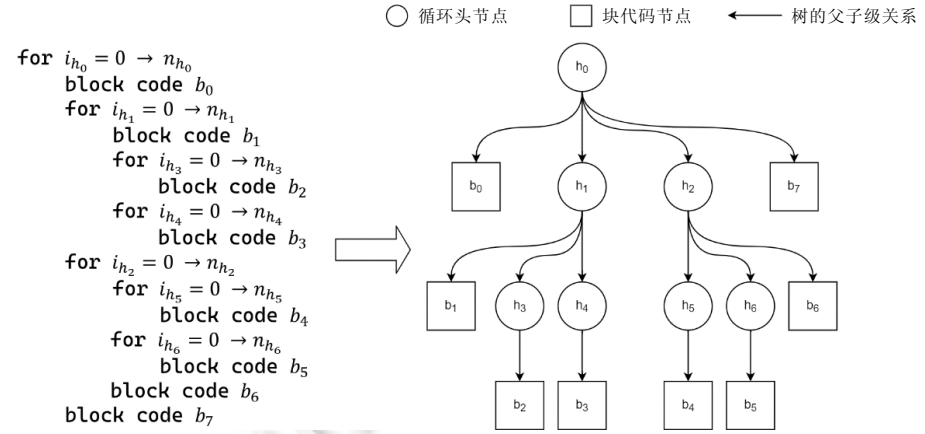


图 6 复杂嵌套循环的循环结构树表示

定义 4.2(循环头、循环块代码). 一个满足上述约束条件的 N 层复杂嵌套循环, 由循环头和循环块代码两种基本元素构成. 循环头是控制循环开始和结束的语句, 用三元组 $h=\{I,D,S\}$表示. I 表示迭代变量, D 表示迭代域, S 表示迭代步长. 循环块代码是循环体内被嵌套循环所分割的连续基本块, 用序列 $b=[s_0,s_1,\ldots,s_k]$, $k\geqslant 0$ 表示. s_k 表示连续基本块中第 k 条内存操作指令.

定义 4.3(循环结构树). 循环结构树是一个三元组 $LST=\{H,B,R\}$. H 是包含 n 个循环头节点的有限集合($n\geqslant 0$); B 是包含 m 个循环块代码节点的有限集合($m\geqslant 0$), 且 $H\cap B=\varnothing$; R 表示循环头节点、循环块代码节点之间的层次关系的有限集合. 当 $n=0$ 时为空树, 关系 R 为空集; 否则, 关系 R 满足以下条件.

(1) 有且仅有一个节点 $h_0\in H$, 它对于关系 R 来说没有前驱节点, 节点 h_0 称为循环结构树的根节点.

(2) $\forall d\in(H\cup B)-\{h_0\}$, d 节点有且仅有一个前驱节点.

(3) $\forall h\in H$, h 节点至少有一个后继节点.

(4) $\forall b\in B$, b 节点没有后继.

为了方便下文描述, 约定如下符号用来表示循环结构树中$\forall d\in H\cup B$ 节点的属性.

(1) 循环结构树上的$\forall d\in H\cup B$ 节点的层级是指从根节点 h_0 到 d 所经过的路径的长度, 记作 $level_d$. 由此可知, $level_{h_0}=0$.

(2) 循环结构树中$\forall d\in H\cup B$ 节点的孩子节点集合是指 d 节点所有直接后继节点从左到右的有序列表, 记作 $childnodes_d$.

(3) 循环结构树中$\forall d\in H\cup B$ 节点的后代节点集合是指 d 节点所有子树上的所有节点的集合, 记作 $descnodes_d$. 循环结构树中$\forall d\in H\cup B$ 节点的父亲节点是指 d 节点的直接前驱节点, 记作 $parentnode_d$.

(4) 循环结构树中$\forall d\in H\cup B$ 节点的祖先节点集合是指从 d 节点到根节点所经过的路径上的除 d 节点之外所有节点的有序列表, 记作 $ancenodes_d$.

4.2.2 复杂嵌套循环的循环结构树模型自动构建方法

算法 2 描述了从正则区域的 Header 块进行循环结构树模型的自动构建的过程. 算法整体上是嵌套的正则

区域的递归遍历过程, 核心方法是函数__ConstructLoopStructureTree. 其中, *header* 参数表示当前正则区域的 Header 块, *parent* 参数表示当前正则区域的父区域的 Header 块. __ConstructLoopStructureTree 函数旨在收集循环结构树模型的三元{*H*,*B*,*R*}为算法递归地遍历正则区域的 Header 块, 因此, *H* 中元素在递归的开始进行收集(第 6、7 行).

算法 2. 循环结构树模型的构建(*ConstructLoopStructureTree*).

输入: 正则区域的 *Header*′.

输出: 循环结构树 *T*={*H*,*B*,*R*}.

1. Function *ConstructLoopStructureTree*(*Header*′)
2. *H*←∅, *B*←∅, *R*←∅
3. __*ConstructLoopStructureTree*(*Header*′,*null*)
4. Return {*H*,*B*,*R*}
5. Function __*ConstructLoopStructureTree*(*header*,*parent*)
6. analyze *header* block to get {*I*,*D*,*S*} //定义 4.2
7. add {*I*,*D*,*S*} to *H*
8. **If** *parent*≠*null* **then** add (*parent*,*header*) to *R*
9. *BS*←∅, *start*←the first block of *header*'s Body, *end*←*header*
10. **While** *start*≠*end*
11. **If** *start* is *LoopHeaderBlock* **then**
12. **If** *BS*≠∅ **then** add (*parent*,*BS*) to *R*, add *BS* to *B*
13. *BS*←∅
14. __*ConstructLoopStructureTree*(*start*,*header*)
15. *start*←the exit block of *start*
16. **Elseif** *start* is *CondHeaderBlock* **then**
17. __*traverseCond*(the first block of *start*'s left Cond, the exit block of *start*, *BS*)
18. __*traverseCond*(the first block of *start*'s right Cond, the exit block of *start*, *BS*)
19. *start*←the exit block of *start*
20. **Else**
21. *I*←search alloca/load/store instruction of *start*
22. add *I* to *BS*
23. *start*←the next block of *start*
24. **If** *BS*≠∅ **then** add (*parent*,*BS*) to *R*, add *BS* to *B*
25. Function __*traverseCond*(*start*,*end*,*BS*)
26. **While** *start*≠*end*
27. **If** *start* is *CondHeaderBlock* **then**
28. *traverseCond*(the first block of *start*'s left Cond, the exit block of *start*, *BS*)
29. *traverseCond*(the first block of *start*'s right Cond, the exit block of *start*, *BS*)
30. *start*←the exit block of *start*
31. *I*←search alloca/load/store instruction of *start*
32. add *I* to *BS*
33. *start*←the next block of *start*

根据定义 4.2, 循环块代码是循环体内被嵌套循环所分割的连续基本块. 第 9 行定义 *BS* 集合来收集连续的基本块. 收集过程中当遇到循环头时, 就将当前 *BS* 集合中的所有基本块作为一个块代码节点加入 *B* 中, 同

时将(*parent*,*BS*)关系对加入 *R* 中, 并将 *BS* 重新置空以便下一次收集(第 12 行、第 13 行). 条件分支中基本块的收集与普通基本块类似, 算法 2 中__traverseCond 函数递归地对条件分支的左右分支中的基本块进行收集. 最后, 当循环内基本块遍历结束后, *BS* 集合需要再次检查是否为空, 若不为空, 则需要将当前 *BS* 集合作为一个块代码节点加入 *B* 中, 同时, 将(*parent*,*BS*)关系对加入 *R* 中(第 24 行).

header 参数和 *parent* 参数是构建循环头节点之间层次关系的关键所在, 递归起始时, *header* 是最外层正则区域的 Header 块, *parent* 为空. 此时, 循环结构树也为空, *header* 就是循环结构树的根节点. 当 *parent* 不为空时, 将(*parent*,*BS*)关系对加入 *R* 中(第 8 行). 当在正则区域检测到循环时, 递归地进行循环头节点之间层次关系的构建(第 14 行).

与算法 1 类似, 算法 2 本质上是对一个正则区域中所有嵌套的正则区域的深度优先遍历. 假设某个正则区域是由 M'个基本块组成的 N'层嵌套的正则区域, 则算法 2 的时间复杂度为 O(M'), 空间复杂度为 O(N').

4.3 基于循环结构树的依赖分析方法

4.3.1 依赖关系初始化

循环结构树中节点之间的依赖是由内存操作指令之间的依赖关系引起的. 依据循环结构树中节点类型的不同, 节点之间依赖关系分为块代码节点之间的依赖、循环头节点之间的依赖以及循环头节点与块代码节点之间的依赖. 基于循环结构树的依赖分析分为依赖初始化和依赖向上传递两个阶段. 所有块代码节点都处于循环结构树的叶子节点, 包含了循环体内部实际的指令序列, 因此, 块代码节点依赖由指令内存依赖初始化, 初始化规则如定义 4.8 所示. 循环头节点依赖由块代码节点依赖向上传递获得, 这部分将在第 4.3.2 节详细介绍. 循环头与块代码之间的依赖客观存在, 但对于并行性分析没有实际意义, 不做分析. 本节介绍块代码节点依赖关系的初始化方法.

定义 4.4(循环携带依赖、循环独立依赖). 循环携带依赖是指依赖存在于循环迭代之间, 即如果删除循环头, 则依赖关系不再存在. 循环独立依赖是指依赖存在于一个循环迭代中, 即如果删除循环头, 则依赖关系仍然存在.

定义 4.5(指令内存依赖). 指令内存依赖是指对于$\forall b_0,b_1\in B$, $\forall s_0\in b_0$, $\forall s_1\in b_1$, 若指令 s_0 的内存读写依赖于指令 s_1 的内存读写, 则指令 s_0 依赖于指令 s_1, 记作 $s_0\rightarrow s_1$. 若指令 s_0 的内存读写不依赖于指令 s_1 的内存读写, 则指令 s_0 不依赖于指令 s_1, 记作$\neg s_0\rightarrow s_1$. 内存操作指令之间的依赖的性质如下.

(1) 内存操作指令之间的依赖具有传递性, $\forall b_0,b_1,b_2\in B$ 且$\forall s_0\in b_0$, $\forall s_1\in b_1$, $\forall s_2\in b_2$, 若 $s_0\rightarrow s_1$ 且 $s_1\rightarrow s_2$, 则有 $s_0\rightarrow s_2$.

(2) 两个相互依赖的内存操作指令之间具有依赖环, $\forall b_0,b_1\in B$, $\forall s_0\in b_0$, $\forall s_1\in b_1$, 若 $s_0\rightarrow s_1$ 且 $s_1\rightarrow s_0$, 则指令有 $s_0\rightarrow s_0$ 和 $s_1\rightarrow s_1$.

内存依赖按照读写先后顺序分为 4 种: 流依赖(写后读)、反依赖(读后写)、输出依赖(写后写)以及输入依赖(读后读). 除了输入依赖不影响并行执行结果之外, 其他 3 种读写依赖都可能影响并行执行的结果. 因此, 本文中所述指令内存依赖是指流依赖、反依赖和输出依赖这 3 种.

定义 4.6(指令执行顺序). 对于循环中任意两个指令 s_0, s_1, 如果 s_0 的首次执行比 s_1 的首次执行早, 则称 s_0 先于 s_1; 否则, 称为 s_0 后于 s_1.

定理 4.1. 对于$\forall b_0,b_1\in B$, $\forall s_0\in b_0$, $\forall s_1\in b_1$, 若 $s_0\rightarrow s_1$ 且 s_0 先于 s_1, 则一定有 $s_1\rightarrow s_0$. 即前执行的指令依赖于后执行的指令时, 后执行的指令也必然依赖于前执行的指令, 并构成依赖环.

证明: 如果删除 b_0, b_1 共同所在的循环的循环头, 则循环体内的指令都按顺序执行. 若 s_0 先于 s_1, 则只能是 $s_1\rightarrow s_0$, 与条件相互矛盾. 因此, 依赖 $s_0\rightarrow s_1$ 一定是跨迭代间的共享变量引起的依赖, 则 s_0, s_1 的执行顺序可以是 s_0, s_1, s_0 或 s_1, s_0, s_1. 由此可知, 一定存在 $s_1\rightarrow s_0$. 再根据定义 4.5 可知, $s_0\rightarrow s_1$ 且 $s_1\rightarrow s_0$ 构成依赖环.

定理 4.2. 对于$\forall b_0,b_1\in B$, $\forall s_0\in b_0$, $\forall s_1\in b_1$, 若 s_0 和 s_1 具有依赖环, 则 s_0, s_1 共同所在的循环具有循环携带依赖.

证明: 如果删除 b_0, b_1 共同所在的循环的循环头, 则循环体内的指令都顺序执行一次. 假设指令 s_0 先于指

令 s_1, 则只可能发生 $s_1 \to s_0$, 不可能发生 $s_0 \to s_1$. 由定义 4.4 的循环携带依赖定义可知, 删除循环头后 $s_0 \to s_1$ 关系不复存在. 因此, s_0, s_1 共同所在的循环具有循环携带依赖, 且此依赖就是 $s_0 \to s_1$.

定理 4.3. 对于$\forall b_0, b_1 \in B$, $\forall s_0 \in b_0$, $\forall s_1 \in b_1$, 若 $s_1 \to s_0$ 且$\neg s_0 \to s_1$, 则 s_0 先于 s_1. 即若循环内两个指令之间是单向依赖, 则一定是后执行的指令依赖于前执行的指令(定理 4.1 的推论).

证明: 假设逆命题"若 $s_1 \to s_0$ 且$\neg s_0 \to s_1$, 则 s_0 后于 s_1"成立. 根据定理 4.1 知, s_0 后于 s_1 且 $s_1 \to s_0$, 则一定有 $s_0 \to s_1$. 与逆命题的条件矛盾, 故逆命题为假, 原命题为真.

定理 4.4. 循环内单向的内存依赖关系一定是循环独立依赖(定理 4.3 的推论).

证明: 由定理 4.2 可知, 循环内如果两个指令是单向依赖, 则一定是后执行依赖于先执行. 删除循环头, 此单向依赖依然存在. 因此, 其一定是循环独立依赖.

定义 4.7(循环结构树的节点依赖). 循环结构树的节点依赖定义为一个五元组 $DEP=(s_x, s_y, d_i, d_j, flag)$, 其中, $d_i, d_j \in H \cup B$, $s_x \in d_i$, $s_y \in d_j$, $flag \in \{0,1\}$, 表示 d_i, d_j 之间的依赖关系由内存指令 s_x, s_y 之间的依赖引起. *flag* 表示 s_x, s_y 引入的依赖是否具有依赖环. 当 *flag*=0, 表示 d_i, d_j 之间的单向依赖由 $s_x \to s_y$ 引起; *flag*=1, 表示 d_i, d_j 之间的依赖环由 $s_x \to s_y$, $s_y \to s_x$ 引起.

定义 4.8(块代码节点依赖). 块代码节点依赖由内存操作指令内存依赖初始化而来, 规则如下.

(1) 对于$\forall b \in B$, $\forall s_x, s_y \in b$, 若 $s_0 \to s_1$ 且 $s_1 \to s_0$, 则 b 具有指向自身的依赖, 记作$(s_x, s_y, b, b, 1)$. 即如果同一块代码内指令具有内存读写依赖环, 则相应的块代码节点具有指向自身的依赖.

(2) 对于$\forall b_i, b_j \in B$, $\forall s_x \in b_i$, $\forall s_y \in b_j$, 若 $s_x \to s_y$ 且$\neg s_y \to s_x$, 则 b_i 单向依赖于 b_j, 记作$(s_x, s_y, b_i, b_j, 0)$. 即如果不同块代码指令之间具有单向内存依赖关系, 则块代码之间也具有单向的依赖关系.

(3) 对于$\forall b_i, b_j \in B$, $\forall s_x \in b_i$, $\forall s_y \in b_j$, 若 $s_x \to s_y$ 且 $s_y \to s_x$, 则 b_i, b_j 具有依赖环, 记作$(s_x, s_y, b_i, b_j, 1)$. 即如果不同块代码之间具有内存依赖环, 则块代码之间也具有依赖环.

(4) 对于$\forall b_i \in B$, $s_x \in b_i$, $\exists b_j \notin B \cup H$, $s_y \in b_j$, 若 $s_x \to s_y$, 则 b_i 依赖于 h_0, 记作$(s_x, s_y, b_i, h_0, 0)$. 这是一种特殊情况, 当块代码中的指令依赖于正则区域之外的指令时, 令块代码依赖于循环结构树的根节点.

定理 4.5. 对于$\forall b_i, b_j \in B$, 若 b_i, b_j 具有依赖环, 则 b_i, b_j 共同所在的循环具有循环携带依赖(定理 4.2 的推论).

证明: 假设 b_i 中的指令先于 b_j 中的指令. 若 b_i, b_j 具有依赖环, 则一定$\exists s_x \in b_i$, $\forall s_y \in b_j$, 使得 $s_x \to s_y$. 由定理 4.1 知, s_x 和 s_y 构成依赖环. 进而, 由定理 4.2 知, s_x, s_y 共同所在的循环具有循环携带依赖. 因此, b_i, b_j 共同所在的循环具有循环携带依赖. 若假设 b_j 中的指令先于 b_i 中的指令, 同理可得 b_i, b_j 共同所在的循环具有循环携带依赖.

定理 4.6. 对于$\forall b_i, b_j \in B$, 若 b_i, b_j 具有单向依赖, 则一定是后执行的块代码依赖于前执行的块代码(定理 4.3 的推论, 其证明过程与定理 4.5 同理, 不再赘述).

依据定理 4.1 的逆否命题可知: 循环只有当循环内任意两个块代码内的指令之间不存在依赖环时, 循环才能依赖独立, 才具有循环内迭代间的并行性. 定义 4.8 中, 从内存依赖分析中初始化块代码依赖关系时, 单个块代码内不构成依赖环的内存依赖关系不需要初始化. 因为循环内单个块代码中内单向内存依赖关系是循环独立依赖, 无须关注. 但是不同块代码之间的单向内存依赖关系需要初始化, 因为其蕴含了不同循环之间的并行的可能性.

如图 7 所示, 循环结构树中块代码指向自身的边表示由定义 4.8 的规则(1)所初始化的循环携带依赖, 而块代码之间的边表示由定义 4.8 的规则(2)所初始化的循环之间依赖. 块代码的依赖关系从内存读写依赖分析中推导而来. 内存读写依赖分析是一项复杂的技术, LLVM 编译器已经提供了一个可靠且稳定的内存读写依赖分析. LLVM 中的内存读写依赖分析主要通过 *MemoryDependencePass* 提供, 其以别名分析为基础, 可以确定内存操作所依赖的前导内存操作或声明. 内存读写依赖分析的结果以 *Key-Values* 集合的方式提供给用户查询, *Key* 是当前内存操作指令, *Values* 是包含了依赖的前导指令以及其位置的集合. 图 8 展示了一个内存依赖分析结果集的例子, 图 8 右边是源程序的 IR 表示, 左边是 IR 经过内存依赖分析后的结果. 本文利用 *MemoryDependencePass* 的结果集推导出块代码的依赖关系, 并初始化到循环结构树的块代码节点中.

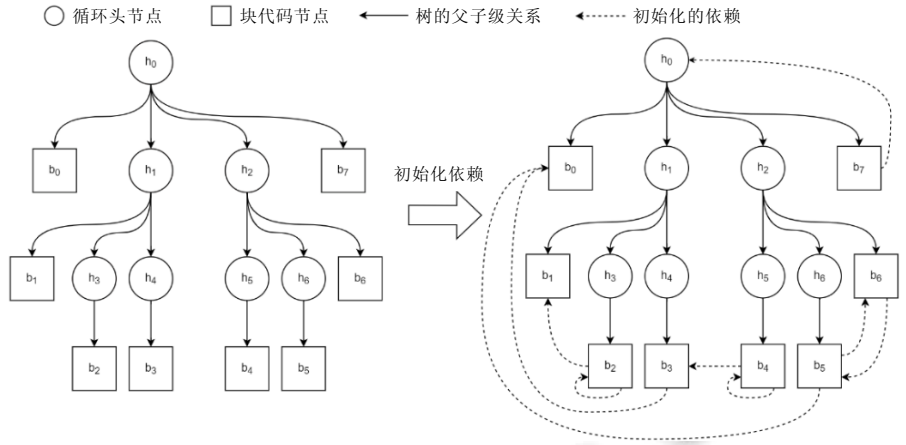


图 7　从内存依赖分析中初始化循环结构树

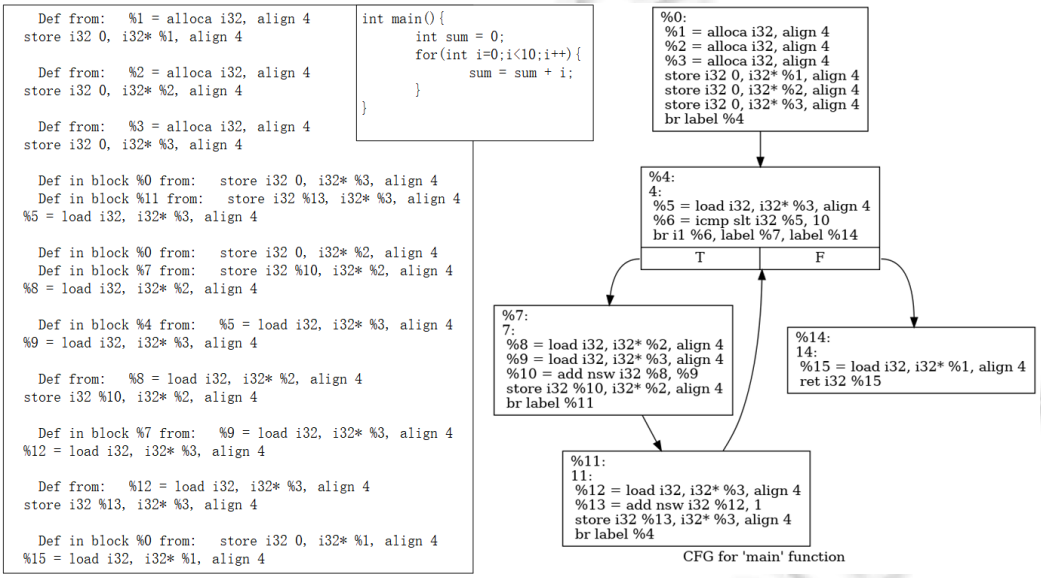


图 8　内存依赖分析结果集的一个例子

算法 3 描述了如何从内存依赖分析的结果集中初始化循环结构树中块代码的依赖关系. 因为初始化依赖只发生在块代码节点, 所以整个算法只关注块代码集合. 遍历块代码节点时, 首先搜索块代码中所有 load/store 内存操作相关的指令, 然后对这些指令进行遍历(第 3–5 行). 依据内存依赖分析的结果集, 对源指令所依赖的目标指令进行查询(第 6 行). 算法第 12–16 行依据内存依赖分析的结果集, 对目标指令反向查询, 以判断源指令与目标指令是否有依赖环. 如果源指令和目标指令之间是单向依赖且在同一个块代码中, 则根据定义 4.8 无须关注此依赖(第 17 行). 如果目标指令在正则区域外, 则创建一条源指令所在块代码到循环根节点的依赖关系边, 表示源指令依赖于更高层级作用域的指令(第 9–11 行), 例如图 7 中 b_2 到 h_0 的依赖边. 否则, 算法第 18–20 行将依据是否有依赖环以及源指令与目标指令是否同属一个块代码来创建一条源指令所在块代码到目标指令所在块代码的依赖关系边. 需要注意的是, 此时, 若源指令与目标指令所在块代码相同, 则将创建指向块代码本身的依赖边. 如图 7 所示, 当算法 3 遍历到块代码节点 b_2 时, 将会创建至少一条 b_2 自身的边和至少一条 b_2 到 b_1 的边. 为了方便简洁起见, 本文中的图例均以一条依赖边表示两个不同节点间的所有指令之间相同依赖方向上的依赖关系.

算法 3. 循环块代码依赖的初始化(*InitializeDependence*).

输入: 内存依赖分析的结果集 *MemRes*.

输出: 块代码依赖边的集合 *D*.

1. Function *InitializeDependence*(*MemRes*)
2. $D\leftarrow\varnothing$
3. **For each** b_i in B
4. $S\leftarrow$search all load/store instruction in b_i
5. **For each** s_x in S
6. $SV\leftarrow MemRes[s_x]$
7. **For each** s_y in SV
8. $b_j\leftarrow s_y$'s blockcode
9. **If** $b_j\notin B$ **then**
10. add $(s_x,s_y,b_i,h_0,0)$ to D
11. continue
12. $TV\leftarrow MemRes[s_y]$, $hascycle\leftarrow$false
13. **For each** s_k in TV
14. **If** $s_k=s_x$ **then**
15. $hascycle\leftarrow$true
16. break
17. **If** $hascycle\neq$true and s_y in S **then** continue
18. **If** $hascycle$=true and s_y in S **then** add $(s_x,s_y,b_i,b_i,1)$ to D
19. **If** $hascycle$=true and s_y not in S **then** add $(s_x,s_y,b_i,b_j,1)$ to D
20. add $(s_x,s_y,b_i,b_j,0)$ to D
21. Return D

算法 3 虽然有 4 层循环的遍历, 但是由于主要逻辑是在第 8–20 行内, 且第 13–16 行的最内层循环是对某条内存指令依赖结果集的遍历, 遍历数据量不大, 若进行展开, 可视为常数阶的操作, 因此, 算法 3 可视为一个 3 层的循环遍历操作. 假设一个循环结构树中有 K 个块代码节点, 每个块代码节点中平均有 I 条内存操作指令, 且每条内存操作指令平均有 P 个内存依赖结果, 则算法 3 的时间复杂度为 $O(K\times I\times P)$. 因为 *MemRes* 以字典形式存储着块代码中内存操作指令的依赖结果集, 所以算法 3 的空间复杂度也为 $O(K\times I\times P)$. 最终, 初始化的块代码依赖边集的大小 $e\approx K\times I\times P$.

4.3.2 依赖关系的向上传递

循环头节点处于循环结构树的非叶子节点, 而块代码节点处于循环结构树的叶子节点. 为了更直观地分析复杂嵌套循环的并行性, 依赖关系需要从块代码节点向上传递至循环头节点. 本节详细介绍块代码节点依赖向上传递的规则和算法.

定义 4.9(最近公共祖先节点). 任意两个块代码 $d_0,d_1\in B$, 令 $P_0=ancenodes_{d_0}$, $P_1=ancenodes_{d_1}$, 则 $P=P_0\cap P_1$ 就是 d_0, d_1 的公共祖先节点. 若$\exists p_h\in P$, 使得$\forall p\in P$, 都有 $level_{p_h}\geqslant level_p$, 则 p_h 称为 d_0, d_1 的最近公共祖先节点.

块代码节点之间依赖初始化完成后, 为了揭示循环携带依赖以及不同循环之间的依赖关系, 需要将块代码之间的依赖、块代码自身的依赖传递到所有循环头节点上去.

定义 4.10(循环头节点依赖). 循环头节点依赖由循环块代码节点依赖传递而来, 传递的规则如下.

(1) 对于块代码间依赖$(s_x,s_y,b_i,b_j,0)$, 令 b_i, b_j 的最近公共祖先节点为 h, 则在循环结构树中从 h 到 b_i 的路径上的所有循环头节点的集合 X 与从 h 到 b_j 的路径上的所有循环头节点的集合 Y 之间存在一个一对一的映射 $f:X\rightarrow Y$, 当$\forall h_k\in X$, $h_t\in Y$ 且 $level_{h_k}=level_{h_t}$ 时, 有$(s_x,s_y,h_k,h_t,0)$. 如图 9 中, b_4, b_3 的最近公共

祖先节点为 h_0, 则向上传递依赖关系后, 产生的依赖关系有 h_5 依赖于 h_4,h_2 依赖于 h_1.

(2) 对于块代码间依赖$(s_x,s_y,b,b,1)$, 令 b_k 为 s_x 所依赖的指令所在块代码中层级最高的块代码, $Y=ancenodes_b$, 则有$\forall h_t\in Y$, 当 $level_{b_k}\leqslant level_{h_t}$ 时, 有$(s_x,s_y,h_t,h_t,1)$. 如图 9 中的 b_2 的循环携带依赖可以传递给 h_3, b_4 的循环携带依赖可以传递给 h_5,h_2.

(3) 对于块代码间依赖$(s_x,s_y,b_i,b_j,1)$, 令 b_k 为 s_x 所依赖的指令所在块代码中层级最高的块代码, b_i, b_j 的最近公共祖先节点为 h, $Y\leftarrow ancenodes_h\cup h$, 则有$\forall h_t\in Y$, 当 $level_{b_k}\leqslant level_{h_t}$ 时, 有$(s_x,s_y,h_t,h_t,1)$. 如图 9 中的 b_5, b_6 具有依赖环, 经过向上传递后, h_2 具有循环携带依赖.

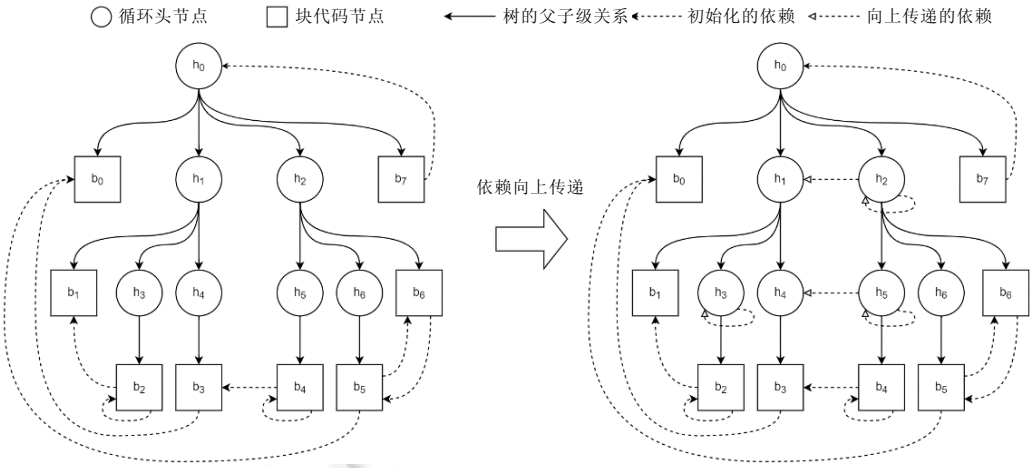


图 9 向上传递依赖性

由上述定义可知, 基于循环结构树的依赖分析可发掘的并行性体现在两个方面.

(1) 同一循环不同迭代间的并行性. 反映到循环结构树的依赖上就是循环头节点是否具有指向自身的依赖: 若一个循环头没有自依赖, 则其控制的循环具有循环迭代间的并行性; 否则, 无循环迭代间的并行性.

(2) 不同循环之间的并行性. 反映到循环结构树的依赖上就是不同循环头之间是否具有依赖关系: 若不同循环头节点间无依赖关系, 则其具有循环之间的并行性; 否则, 无循环之间的并行性.

算法 4 描述了如何将循环结构树中块代码间的依赖关系传递到循环头节点. 算法整体上是对块代码间依赖边集的遍历, 并在遍历过程中, 依据定义 4.10 中的规则构建循环头依赖边. 算法第 2–5 行描述了从块代码间依赖边集构建 s_x 列索引的过程. 算法遍历边集, 索引为空时构建索引, 索引不为空时将依赖边加入索引指向的序列中. 此算法服务于后文算法 6, 用于在搜索最高层块代码之前建立索引, 以提高搜索的效率. 算法第 7 行表示块代码间依赖$(s_x,s_y,b_i,h_0,0)$不向上传递. 算法第 8–14 行按定义 4.10 的规则(1)进行向上传递. 规则(2)可以看作规则(3)的特殊情况, 因为源块代码和目标块代码是同一个, 所以最近公共祖先就是其父节点. 因此, 算法第 15–20 行将定义 4.10 的规则(2)和规则(3)合并进行向上传递. *searchNearestComAnce* 表示搜索任意两个块代码的最近公共祖先, 详见算法 5. *searchHighestLevelBlock* 表示搜索当前指令所依赖的指令所在块代码中层级最高的块代码, 详见后文算法 6.

算法 4. 依赖关系的向上传递(*PassUpDependence*).

输入: 块代码依赖边的集合 D.

输出: 循环头依赖边的集合 D'.

1. Function *PassUpDependence*(D)
2. $DSK\leftarrow\{\cdot\}$ //集合 D 的 s_x 列索引字典
3. **For each** $(s_x,s_y,b_i,b_j,flag)$ in D

4. **If** $DSX[s_x]=\varnothing$ **then** create Index $DSX[s_x]$
5. add $(s_x,s_y,b_i,b_j,flag)$ to $DSX[s_x]$
6. **For each** $(s_x,s_y,b_i,b_j,flag)$ in D
7. **If** $b_j=h_0$ **then** continue
8. **If** $flag=0$ **then**
9. $k\leftarrow 0$, $t\leftarrow 0$, $X\leftarrow ancenodes_{b_i}$, $Y\leftarrow ancenodes_{b_j}$, $h_k\leftarrow X[k]$, $h_t\leftarrow Y[t]$
10. **While** $h_k\neq h_t$
11. **If** $level_{h_k}=level_{h_t}$ **then** add $(s_x,s_y,h_k,h_t,0)$ to D'
12. **Elseif** $level_{h_k}<level_{h_t}$ **then** $t\leftarrow t+1$, $h_t\leftarrow X[t]$
13. **Else** $k\leftarrow k+1$, $h_k\leftarrow Y[k]$
14. **If** $flag=1$ **then**
15. $b_k\leftarrow SearchHighestLevelBlock(s_x,s_y,b_i,b_j,flag,DSX)$
16. $h\leftarrow SearchNearestComAnce(b_i,b_j)$
17. $Y\leftarrow ancenodes_h\cup h$
18. **For each** h_t in Y
19. **If** $level_{b_k}\leqslant level_{h_t}$ **then** add $(s_x,s_y,h_t,h_t,1)$ to D'
20. Return D'

算法 5 描述了如何搜索任意两个块代码节点 b_i, b_j 的最近公共祖先节点. 根据定义 4.9, b_i, b_j 的最近公共祖先是其所有公共祖先节点中层级最低(*level* 属性值最大)的祖先节点. 算法首先初始化 k, t 两个下标标识为 0, 分别指向当前 $ancenodes_{b_i}$ 和 $ancenodes_{b_j}$ 的起始位置, 并以 h_k 和 h_t 分别表示在 k 和 t 位置的节点元素(第 2 行、第 3 行); 然后, 算法对 $ancenodes_{b_i}$ 和 $ancenodes_{b_j}$ 同时进行遍历, h_k 和 h_t 中层级较低者优先遍历; 最后, 经过 h_k 和 h_t 的相互追逐, 算法会达到停止条件 $h_k=h_t$. 此时, 第 1 次使得 $h_k=h_t$ 的位置处的节点就是 b_i, b_j 的最近公共祖先节点. 最好情况下, 当 b_i, b_j 是同一个块代码节点时, 算法无须遍历 $ancenodes_{b_i}$ 和 $ancenodes_{b_j}$, 此时, 算法的时间复杂度为 O(1). 最坏情况下, 当 b_i, b_j 任意一个处于 N 层循环结构树的最低层级, 算法需要遍历长度为 N 的 $ancenodes_{b_i}$ 或 $ancenodes_{b_j}$, 此时, 算法的时间复杂度为 O(N).

算法 5. 搜索最近公共祖先(*SearchNearestComAnce*).

输入: 任意两个不同块代码 b_i, b_j.

输出: b_i, b_j 的最近公共祖先 h.

1. Function $SearchNearestComAnce(b_i,b_j)$
2. $k\leftarrow 0$, $t\leftarrow 0$, $X\leftarrow ancenodes_{b_i}$, $Y\leftarrow ancenodes_{b_j}$
3. $h_k\leftarrow X[k]$, $h_t\leftarrow Y[t]$
4. **While** $h_k\neq h_t$
5. **If** $level_{h_k}<level_{h_t}$ **then** $t\leftarrow t+1$, $h_t\leftarrow X[t]$
6. **Else** $k\leftarrow k+1$, $h_k\leftarrow Y[k]$
7. Return h_k

算法无论最好还是最坏, 都需要记录 $ancenodes_{b_i}$ 和 $ancenodes_{b_j}$, 因此, 算法的空间复杂度为 O(N).

算法 6 描述了如何根据构建的 s_x 列索引 *DSX* 对 s_x 所依赖的指令所在块代码进行搜索, 并找到最高层级块代码. 算法根据依赖边进行广度优先搜索, 直到搜索完所有依赖的指令为止. 搜索过程中, b_k 记录当前所依赖的最高层级的块代码. 当算法停止时, b_k 就是最高层级的块代码. 因为依赖边可能存在环, 所以搜索过程中用 R 记录已经搜索过的依赖指令, 当遇到已经搜索过的指令, 则直接跳过, 避免死循环. 所有块代码的依赖边集合构成了多个互不连通的子图, 因此, 算法 6 的时间复杂度取决于所要查询的依赖边处于哪个子图内. 最好情

况下, 要搜索的子图仅有一条依赖边, 此时, 算法时间复杂度为 O(1). 最坏情况下, 所有依赖边都在一张图中, 此时, 算法时间复杂度为 O(e), 其中, e 为块代码节点依赖边集的大小. 因为 Q 和 R 随着子图边集的大小而变化, 所以最坏情况下, 算法空间复杂度为 O(1); 最好情况下, 算法空间复杂度为 O(e).

算法 6. 搜索最高层级块代码(*SearchHighestLevelBlock*).

输入: 任意块代码依赖 s_x, s_y, b_i, b_j, *flag*, *DSX*.

输出: b_i 所依赖的块代码中层级最高的块代码 b_k.

1. Function *SearchHighestLevelBlock*(s_x,s_y,b_i,b_j,*flag*,*DSX*)
2. $Q\leftarrow[s_y]$ //队列
3. $R\leftarrow\{s_x\}$ //集合
4. $b_k\leftarrow b_i$
5. **While** $Q\neq\varnothing$
6. $s\leftarrow$the top of Q
7. dequeue s from Q
8. $deps\leftarrow DSX[s]$
9. **For each** $(s'_x,s'_y,b'_i,b'_j,flag')$ in *deps*
10. **If** s'_y not in R **then**
11. **If** $level_{b'_i}<level_{b_k}$ **then** $b_k\leftarrow b'_i$
12. enqueue s'_y to Q
13. add s'_y to R
14. Return b_k

算法 4 的第 3–5 行是对所有块代码依赖边的遍历, 时间复杂度为 O(e). 算法 4 的第 6–20 行的主要逻辑可分为第 8–14 行和第 15–20 行两个部分. 其中, 第 8–14 行的逻辑和 *SearchNearestComAnce* 函数一样, 加上外层对所有依赖边的遍历, 时间复杂度最好情况下为 O(e), 最坏情况下为 O(eN); 第 15–20 行的时间复杂度主要取决于 *SearchHighestLevelBlock* 和 *SearchNearestComAnce* 函数, 时间复杂度最好情况下为 O(e), 最坏情况下为 O(e^2+eN). 由此可知, 第 6–20 行的时间复杂度最好情况下为 O(e), 最坏情况下为 O(e^2+eN).

综合来看, 算法 4 的时间复杂度最好情况下为 O(e), 最坏情况下为 O(e^2+eN).

4.4 复杂嵌套循环的并行代码生成方法

本节介绍如何根据循环结构树中蕴含的并行性, 为串行的复杂嵌套循环生成并行执行的代码. 得益于 Tian 等人[20]和 Jingu 等人[21]的工作, 本文可以在 LLVMIR 中通过加入 OpenMP 的并行指令来达到指导循环并行执行的目的. 对于循环头节点中不同的并行性, 本文使用不同的 OpenMP 指令进行标记.

(1) 对于循环不同迭代间的并行性, 使用#pragma omp parallel for 对可并行化的循环进行标注. 如图 10 所述, h_0, h_1, h_4, h_6 无循环携带依赖, 具有循环内并行性, 为其标注#pragma omp parallel for. h_2, h_3, h_5 虽然具有循环携带依赖, 但是有些循环携带依赖可以利用有效的依赖消除技术进行处理, 例如归纳变量的依赖消除、环路倾斜、循环分裂、构建数据副本[22]. 在某些情况下, 也可以用 OpenMP 的子句标注需要同步的依赖变量, 以保证并行执行的正确性. 例如, 用于标注归约变量的 reduction 子句, 或者对于输出依赖引起的循环携带依赖, 可以使用 lastprivate 子句进行标注等.

(2) 对于循环间并行性, 使用#pragmaomp sections 对可并行化的同级循环进行标注. 如图 10 所示, h_3, h_4 之间、h_5, h_6 之间具有循环间并行性, 可以将 h_3, h_4 看作一个 sections, 让 h_3, h_4 两个循环任务在不同的 section 中并行执行. 同理, 可以将 h_5, h_6 看作一个 sections, 让 h_5, h_6 两个循环任务并行执行.

图 10 展示了基于循环头节点依赖分析生成的并行指令标注的程序示例. 经过 OpenMP 并行指令标注, 图 4 中的串行嵌套循环代码被转换为图 10 中的并行嵌套循环代码. 其中, *omp_set_max_active_levels*(4)表示开启

OpenMP 嵌套并行且设置最大的可嵌套数量为 4. 并行区域嵌套层数是#pragma omp parallel for 指令标注和#pragmaomp sections 指令标注相互嵌套的层数, 所以图 10 中 *omp_set_max_active_levels* 的参数只要大于或等于 4 都是可以正常并行执行程序的.

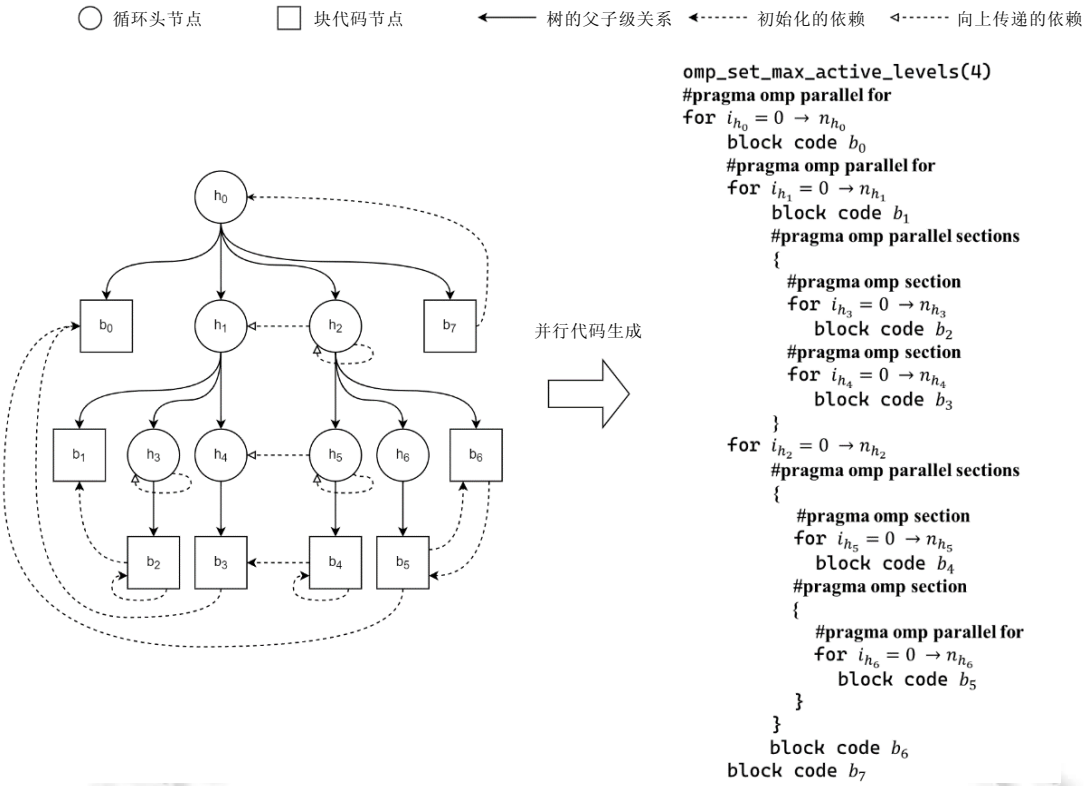


图 10 基于循环头节点依赖生成并行代码

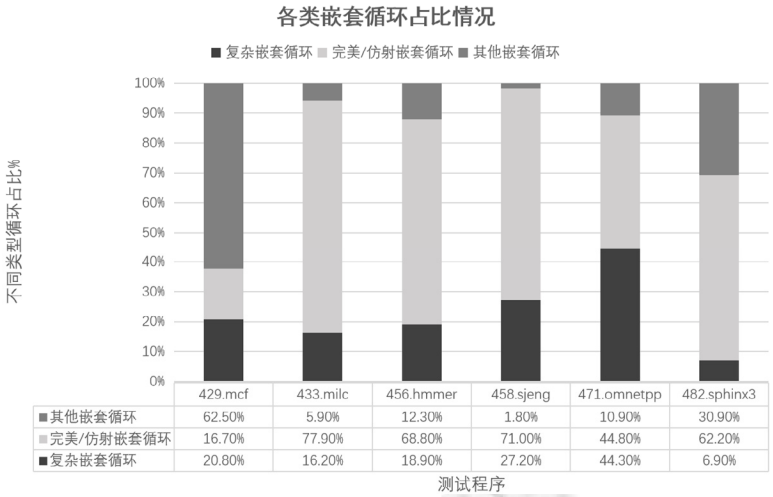
5 实验验证

实验测试部分包括各类嵌套循环占比统计和并行性能加速测试. 首先对待测程序中的循环进行静态分类统计, 通过各类嵌套循环的占比情况, 说明本文方法所能处理的循环的适用范围; 然后, 对待测程序采用两种策略进行并行优化测试, 通过对比两种优化策略的加速比, 说明本文并行化框架增强了 LLVM 的并行编译优化能力.

如表 1 所示, 实验测试从 SPECCPU 2006[23]测试集中选择了包含近 500 个复杂嵌套循环的 6 个程序案例进行测试分析, 包括 429.mcf, 433.milc, 456.hmmer, 458.sjeng, 471.omnetpp 和 482.sphinx3. 程序案例存在不同类型的嵌套循环, 包括完美/仿射嵌套循环、复杂嵌套循环和其他嵌套循环. 图 11 展示了不同程序案例中不同类型嵌套循环的占比情况, 其中, 复杂嵌套循环占比较多的程序案例是 429.mcf, 458.sjeng, 471.omnetpp, 它们的占比都超过了 20%, 最大占比达到 44.3%. 统计结果表明, 程序案例中存在较多有优化价值的复杂嵌套循环, 平均每个程序案例中有大约 22%的复杂嵌套循环可以使用本文框架尝试并行化.

表 1 复杂嵌套循环数统计情况

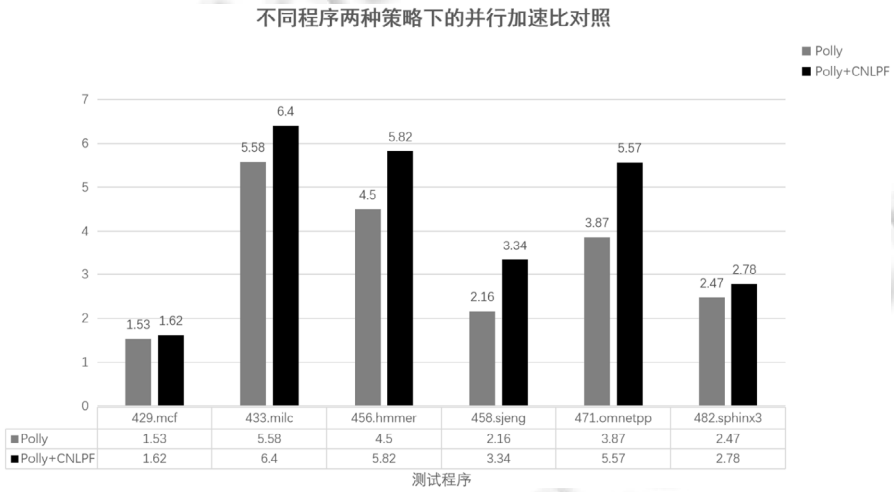
	429.mcf	433.milc	456.hmmer	458.sjeng	471.omnetpp	482.sphinx3
循环总数	48	353	780	217	393	646
复杂嵌套循环数	10	57	148	59	174	45



	429.mcf	433.milc	456.hmmer	458.sjeng	471.omnetpp	482.sphinx3
其他嵌套循环	62.50%	5.90%	12.30%	1.80%	10.90%	30.90%
完美/仿射嵌套循环	16.70%	77.90%	68.80%	71.00%	44.80%	62.20%
复杂嵌套循环	20.80%	16.20%	18.90%	27.20%	44.30%	6.90%

图 11 各类嵌套循环在程序案例中的比例

基于 SPEC 测试运行套件, 将默认 Config 文件的编译方式修改为 LLVM 编译. 对上述程序案例用两种策略分别进行优化测试, 一种是单独使用 LLVM Polly 框架进行优化, 另一种是组合使用 LLVM Polly 和 CNLPF. 图 12 展示了不同程序案例在这两种优化策略下的并行加速比对照.



	429.mcf	433.milc	456.hmmer	458.sjeng	471.omnetpp	482.sphinx3
Polly	1.53	5.58	4.5	2.16	3.87	2.47
Polly+CNLPF	1.62	6.4	5.82	3.34	5.57	2.78

图 12 不同程序在不同优化策略下的并行加速比

结果显示, 相对于单独 Polly 的策略, 具有显著加速比提升效果的是 456.hmmer、458.sjeng 和 471.omnetp 程序, 它们的加速比平均提升了 42.6%; 433.milc 的加速比提升效果不显著, 加速比提升了 14.7%; 429.mcf 和 482.sphinx3 的加速比提升效果较逊色, 平均提升了 9.2%. 实验测试说明, 本文的并行化框架可以处理 LLVMPolly 无法优化的复杂嵌套循环, 其与 LLVMPolly 框架互补, 在一定程度上增强了 LLVM 编译器自动并行化的能力, 使得源程序可以在 Polly 加速的基础上再获得 9%–43%的加速效果.

6 总结和展望

多核处理器的发展给嵌入式软件并行化带来了机遇和挑战. 嵌入式应用遗留代码中往往包含大量复杂嵌套循环, 手工优化易出错且耗时, 但是现有编译器中, 循环级自动并行化方法在处理复杂嵌套循环时能力不

足, 导致未能充分发挥多核并行的资源. 本文的研究旨在通过新的复杂循环自动并行化技术来拓展 LLVM 并行编译的能力, 减轻嵌入式开发人员对遗留系统的并行优化负担. 本文基于 LLVM Pass 的扩展机制, 提出了一种针对复杂嵌套循环的自动并行化方法, 通过基于循环结构树的依赖分析和并行代码生成, 来发掘复杂嵌套循环中蕴含的并行性. 实验结果表明, 本文方法具有一定的适用性, 能够对部分复杂嵌套循环实现并行加速. 但该框架目前仍有很多不足, 比如只能处理满足一定约束的复杂嵌套循环, 具有一定局限性. 后续研究工作将继续改进现在的并行化框架, 使得框架适用范围更广泛, 加速效果更显著.

References:

[1] Yang MF, Gu B, Duan ZH, *et al*. Intelligent program synthesis framework and key scientific problems for embedded software. Chinese Space Science and Technology, 2022, 42(4): 1−7 (in Chinese with English abstract). [doi: 10.16708/j.cnki.1000-758X.2022.0046]

[2] Osinski L, Langer T, Mader R, *et al*. Challenges and opportunities with multi-core embedded platform-a spotlight on real-time and dependability concepts. In: Proc. of the 9th European Congress Embedded Real Time Software and Systems. 2018. Article No.55.

[3] Midkiff SP. Automatic parallelization: An overview of fundamental compiler techniques. In: Synthesis Lectures on Computer Architecture. 2012.

[4] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets. ACM SIGPLAN Notices, 2000, 35(5): 145−156.

[5] Zarei ZE, Lotfi S, Mohammad KL, *et al*. 3-D data partitioning for 3-level perfectly nested loops on heterogeneous distributed systems. Concurrency and Computation: Practice and Experience, 2017, 29(5): Article No.e3976.

[6] Bondhugula U. Compiling affine loop nests for distributed-memory parallel architectures. In: Proc. of the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis. New York: IEEE, 2013. 1−12.

[7] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the Int'l Symp. on Code Generation and Optimization. New York: IEEE, 2004. 75−86.

[8] Grosser T, Zheng H, Aloor R, *et al*. Polly-polyhedral optimization in LLVM. In: Proc. of the 1st Int'l Workshop on Polyhedral Compilation Techniques. New York: IEEE, 2011. 1-6.

[9] Shivam A. Polygonal Iteration Space Partitioning Using the Polyhedral Model. Ann Arbor: ProQuest LLC, 2016.

[10] Bondhugula U, Hartono A, Ramanujam J, *et al*. A practical automatic polyhedral parallelizer and locality optimizer. In: Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: Association for Computing Machinery, 2008. 101−113.

[11] Pop S, Cohen A, Bastoul C, *et al*. GRAPHITE: Polyhedral analyses and optimizations for GCC. In: Proc. of the GCC Developers Summit. Ontario: Charles University in Prague, 2006. 179−198.

[12] Moses WS, Chelini L, Zhao R, *et al*. Polygeist: raising C to polyhedral MLIR. In: Proc. of the 30th Int'l Conf. on Parallel Architectures and Compilation Techniques. New York: IEEE, 2021. 45−59.

[13] Nadezhkin D, Nikolov H, Stefanov T. Automated generation of polyhedral process networks from affine nested-loop programs with dynamic loop bounds. ACM Trans. on Embedded Computing Systems, 2013, 13(1s): 1−24.

[14] Geuns SJ, Bekooij MJG, Bijlsma T, *et al*. Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. In: Proc. of the Design, Automation & Test in Europe. New York: IEEE, 2011. 1−6.

[15] Palkowski M, Klimek T, Bielecki W. TRACO: An automatic loop nest parallelizer for numerical applications. In: Proc. of the Federated Conf. on Computer Science and Information Systems. New York: IEEE, 2015. 681−686.

[16] Ding LL, Li YB, Zhang SP, *et al*. Auto-parallelization research based on branch nested loops. Computer Science, 2017, 44(5): 14−19, 52 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2017.05.003]

[17] Azadeh F, Victor N. Modular divide-and-conquer parallelization of nested loops. In: Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: Association for Computing Machinery, 2019. 610−624.

[18] Abdi RZ, Lotfi S, Isazadeh A, *et al*. Intra-tile parallelization for two-level perfectly nested loops with non-uniform dependences. The Computer Journal, 2021, 64(9): 1358−1383.

[19] Johnson R, Pearson D, Pingali K. The program structure tree: Computing control regions in linear time. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: Association for Computing Machinery, 1994. 171−185.

[20] Tian X, Saito H, Su E, *et al*. LLVM compiler implementation for explicit parallelization and SIMD vectorization. In: Proc. of the 4th Workshop on the LLVM Compiler Infrastructure in HPC. New York: Association for Computing Machinery, 2017. 1−11.

[21] Jingu K, Shigenobu K, Ootsu K, *et al*. Directive-based parallelization of for-loops at LLVM IR level. In: Proc. of the 20th IEEE/ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. New York: IEEE, 2019. 421−426.

[22] Barlas G. Multicore and GPU Programming: An Integrated Approach. Waltham: Elsevier, 2014.

[23] Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1−17.

附中文参考文献:

[1] 杨孟飞, 顾斌, 段振华, 等. 嵌入式软件智能合成框架及关键科学问题. 中国空间科学技术, 2022, 42(4): 1−7. [doi: 10.16708/j.cnki.1000-758X.2022.0046]

[16] 丁丽丽, 李雁冰, 张素平, 等. 分支嵌套循环的自动并行化研究. 计算机科学, 2017, 44(5): 14−19, 52. [doi: 10.11896/j.issn.1002-137X.2017.05.003]

马春燕(1978－), 女, 博士, 副教授, 博士生导师, 主要研究领域为嵌入式软件系统建模与验证, 软件自动化测试与故障定位.

叶许姣(1998－), 女, 硕士生, 主要研究领域为自动并行化, 程序分析.

吕炳旭(1998－), 男, 博士生, 主要研究领域为并行编译, 程序优化.

张雨(1983－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为智能嵌入式系统协同设计与验证, 智能信息处理.