基于条件语句半不变性分析的循环分裂优化

韩林, 邵晶晶, 聂凯, 李浩然, 刘浩浩, 陈梦尧

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.250300153

基于条件语句半不变性分析的循环分裂优化

韩 林[1] 邵晶晶[2] 聂 凯[1] 李浩然[1] 刘浩浩[1] 陈梦尧[2]
1 郑州大学国家超级计算郑州中心 郑州 450001
2 郑州大学计算机与人工智能学院 郑州 450001
(hanlin@zzu.edu.cn)

摘 要 循环分裂是一种重要的编译优化技术,可有效减少循环控制开销、提升指令流水线效率,并为后续优化创造机会。针对GCC编译器现有循环分裂策略适用范围受限的问题,提出了一种改进的循环分裂优化算法,该算法基于静态单赋值形式,根据循环内条件变量所在PHI节点的位置,将其区分为循环头PHI节点条件变量和合并PHI节点条件变量,针对这两种条件变量,算法分别进行半不变性分析,并以此选择分裂点,从而实现更通用的循环分裂优化。在申威GCC编译器中实现了该算法。在申威新一代处理器平台上的实验结果显示,相比原有的循环分裂算法,该算法使SPEC CPU 2006测试集中的470.lbm测试程序性能提升8.8%,SPEC CPU 2017测试集中的620.omnetpp_s测试程序性能提升4.3%。所提方法扩展了可优化循环结构的范围,提升了申威GCC编译器的优化效率,可助力国产申威平台的基础软件生态建设。

关键词:GCC编译器;循环优化;循环分裂;静态单赋值;控制依赖
中图分类号 TP311

Loop Splitting Based on Conditional Statement Invariance Analysis

HAN Lin[1],SHAO Jingjing[2],NIE Kai[1],LI Haoran[1],LIU Haohao[1] and CHEN Mengyao[2]
1 National Supercomputing Center in Zhengzhou,Zhengzhou University,Zhengzhou 450001,China
2 College of Computer and Artificial Intelligence,Zhengzhou University,Zhengzhou 450001,China

Abstract Loop splitting is a key compiler optimization technique that can reduce control overhead,improve instruction pipeline efficiency,and create opportunities for subsequent optimization. To address the limited applicability of the existing loop splitting strategy in GCC compiler,an improved algorithm is proposed based on static single assignment form. It categorizes condition variables by their positions in PHI nodes into loop-header PHI node condition variables and merge PHI node condition variables. For these two types of condition variables,the algorithm performs semi-invariance analysis separately and selects partitioning points accordingly,thereby achieving more general loop splitting optimization. The algorithm is implemented in the Sunway GCC compiler. Experimental results on the new-generation Sunway processor platform show that,compared to the original algorithm,the proposed algorithm improves the performance of the 470.lbm benchmark in SPEC CPU 2006 test suite by 8.8%,and the 620.omnetpp_s benchmark in SPEC CPU 2017 test suite by 4.3%. This method expands the scope of optimizable loop structures,improves the optimization efficiency of the Sunway GCC compiler,and can provide assistance for the basic software ecosystem construction of the domestic Sunway platform.

Keywords GCC compiler,Loop optimization,Loop splitting,Static single assignment,Control dependence

1 引言

循环在科学计算程序中广泛存在,其执行效率直接影响程序的整体性能,尤其是在处理大规模数据重复计算中更为显著。循环的执行效率受多个因素影响,主要包括循环体内语句的计算复杂度、循环迭代次数以及数据访问模式。通过优化循环结构,可以有效缩短计算时间并改善内存访问局部性,从而提升程序性能和系统资源利用率。在高级语言程序设计中,循环优化不仅能提升串行代码的执行效率,更能充分发挥硬件并行计算潜力。

在编译器优化过程(PASS)中,循环优化是核心研究领域之一。编译器通过静态分析技术,采用循环展开[1]、循环合

到稿日期:2025-03-28 返修日期:2025-06-24
基金项目:河南省重大科技专项(241100210100,221100210600);河南省科技攻关项目(242102211094);国家重点研发计划高性能计算专项(2023YFB3002505)
This work was supported by the Major Science and Technology Special Projects in Henan Province(241100210100,221100210600),Key Projects of Science and Technology of Henan Province(242102211094) and National Key Research and Development Program of China(2023YFB3002505).
通信作者:聂凯(ieknie@zzu.edu.cn)

并[2]、循环分块[3]和循环分裂[4]等方法重构循环结构。其中，循环分裂作为提升代码性能和并行化执行的关键技术，引起了众多计算机学者的广泛关注。GCC(GNU Compiler Collection)作为主流开源编译工具链，其循环优化能力直接影响软件的性能表现。特别是在多核处理器[5]和异构计算编译环境中，高效的循环分裂算法可显著缩短程序执行时间和降低系统能耗。

循环中控制流语句引入的控制依赖会破坏循环内数据的局部性，增加数据流分析的复杂度[6-8]。控制依赖是指语句执行取决于前驱语句结果的条件约束[8]。循环分裂技术通过将含条件分支的循环结构拆分为多个子循环，将不同条件下的迭代分配到独立循环中，从而减少分支预测开销。这种方法不仅能提高指令级并行性[9]，优化数据局部性，还能改善循环结构，为后续的循环优化提供机会。

尽管现有编译器(如 GCC，LLVM[10])已广泛支持循环分裂，但其主要针对简单循环结构，对复杂条件分支的处理能力有限，限制了循环分裂策略的优化范围和效果。现有方法[11-13]多聚焦于特定硬件架构或特定类型的循环结构，缺乏对通用编译器中循环分裂策略的普适性研究。

基于 GCC 编译器深入分析了 Tree-SSA[14]框架下的循环分裂优化过程，提出了改进的循环分裂算法，并在申威平台上实现了该算法。实验结果证实了算法的有效性，分析了测试结果并探讨了其局限性，为进一步优化循环分裂提供了思路，对提高编译器循环优化效率和编译器开发具有重要意义[15]。

2 背景及相关工作

2.1 GCC 的逻辑结构

GCC 的逻辑结构如图 1 所示。

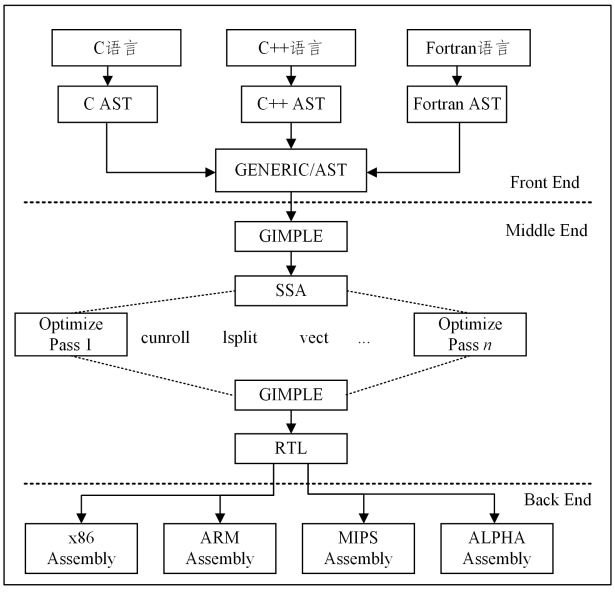


图 1 GCC 的逻辑结构图

Fig. 1 Logical structure diagram of GCC

从高级程序源代码转换为目标机器的汇编代码过程中，GCC 编译器主要使用了 4 种中间表示形式：抽象语法树(Abstract Ayntax Tree，AST[16])、GIMPLE、静态单赋值[17](Static Single Assignment，SSA)和寄存器传输语言(Register Transfer Language，RTL)。在编译的前端，输入的高级语言程序经历词法分析、语法分析和语义分析，首先被转换为 GENERIC 表示，GENERIC 是一种规范的抽象语法树。随后 GENERIC 进一步转换为 GIMPLE 表示，GIMPLE 是一种与高级语言无关的三地址码中间表示形式[18]。GIMPLE 表示转换为 SSA 形式后，经过多个 SSA 优化阶段，进一步转换为与后端机器相关的寄存器传输语言 RTL，以供后端生成目标平台汇编代码。

在 GCC 编译器中，若干逻辑阶段被组织为一种称为“PASS”(处理过程)的链表结构，每个“PASS”完成一种特定的优化处理。常见的循环优化 PASS 包括循环展开、循环分布[19]和循环向量化等[20]。本文主要研究的是“循环分裂 PASS”，由编译选项“-fsplit-loops”控制。

2.2 基本块与自然循环结构

在编译器优化过程中，基本块(Basic Block，BB)[21]作为程序控制流图(Control Flow Graph，CFG)的基本单元，具有单入口单出口的特性。一个基本块由线性顺序执行的指令序列组成，其内部不包含任何分支或跳转指令(如条件跳转、函数调用等)。基本块的划分为数据流分析、依赖分析以及其他优化技术提供了基础。

自然循环是控制流图中具有特定结构的子图，通常由一组基本块构成。根据 Cytron 等[22]提出的经典定义，自然循环是通过单一回边(Latch)构成的控制流子图。其结构特征表现为：1)包含唯一的头节点(Loop Header)作为循环入口；2)存在从循环体内节点返回头节点的回边，形成闭环控制流。

图 2 展示了一个典型的 while 循环的 C 代码及其对应的中间表示。其中，bb 3 作为循环头，是循环的支配节点和控制入口；bb 4 构成循环体，包含循环的核心执行逻辑；从 bb 4 指向 bb 3 的回边形成迭代闭环。

```
int main() {
  int sum = 0;
  int i = 0;
  while (i < 10) {
    sum += i;
    i++;
  }
  return sum;
}
```

```
<bb 2> :
sum = 0;
i = 0;

<bb 3> :
if (i > 9)
  goto <bb 5>;
else
  goto <bb 4>;

<bb 4> :
sum = sum + i;
i = i + 1;
goto <bb 3>;

<bb 5> :
D.2337 = sum;

<bb 6> :
return D.2337;
```

(a)C 语言代码 (b)中间表示

图 2 自然循环结构示例

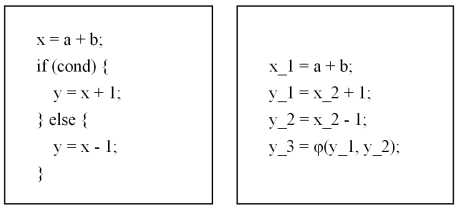
Fig. 2 Example of a natural loop structure

GCC 编译器通过 struct loop 数据结构详细记录了循环的拓扑特征，这些结构化信息为循环分析和优化提供了基础支持。

2.3 SSA 静态单赋值

静态单赋值是 GCC 编译器中端采用的一种中间表示形式，广泛应用于程序分析和优化。SSA 的核心特性是每个变量在其生命周期内仅被赋值一次，且每次赋值都对应唯一的变量名标识。这一特性消除了传统编程模型中变量重名的问题，使得程序中的每个变量在分析时都具有确定性，从而简化了依赖分析[23]和数据流分析等任务。

在 SSA 形式中，通过引入 Φ 函数（PHI 函数）来处理控制流合并点的值选择问题。如图 3(a)所示的代码段，其 SSA 转换结果如图 3(b)所示。变量 x 和 y 分别被重命名为 x_1，x_2，y_1，y_2 和 y_3，并在控制流分支的合并点通过 Φ 函数选择正确的值。



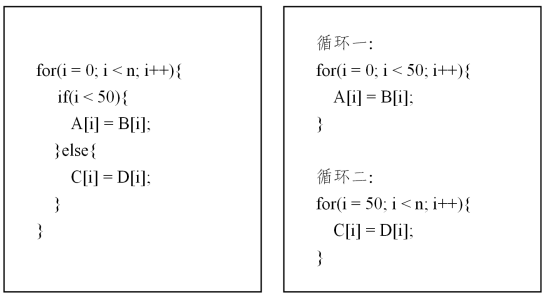
(a)原始代码　(b)SSA 形式

图 3　代码段的 SSA 形式示例

Fig. 3　Example of SSA format for a code snippets

2.4 现有循环分裂算法的局限性

通过对主流编译器循环分裂优化的系统分析，发现现有循环分裂算法仅能支持特定模式的循环结构分裂，这类结构通常包含“$IV<comp$”形式的简单条件语句（IV 为归纳变量，$comp$ 为比较值），且要求条件变量在迭代空间中呈现严格单调特性。如图 4 所示，此类典型循环结构（见图 4(a)）可通过计算分裂点将迭代空间划分为两个独立循环（见图 4(b)）。



(a)原始循环　(b)循环分裂后

图 4　原有循环分裂方法适用的典型循环结构

Fig. 4　Typical loop structures applicable to the original loop splitting method

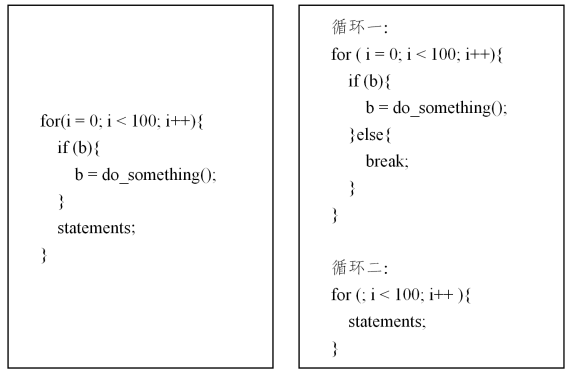
然而，这种严格限制导致编译器无法处理实际应用中常见的复杂场景，包括含有多条件分支的复合控制流、具有非单调变化特征的变量，以及包含非线性递推等复杂归纳表达式的循环结构。正是由于这些限制，现有循环分裂技术的适用范围显著受限，严重影响了其在真实代码中的优化效果和实用价值。

3 改进的循环分裂优化算法

3.1 含有半不变条件语句的循环结构

基于对循环结构的深入分析与研究，提出了一种不同于现有循环分裂算法的循环结构，将其称为含有半不变条件语句的循环结构，如图 5 所示。

图 5(a)中循环内的 if 条件语句中 b 是半不变的，即真分支支配的语句改变 b 值，而假分支及其支配的语句不对 b 值产生任何改变，同时循环内后续代码的执行也不影响 b 值。当循环自某次迭代开始不再进入真分支，则将循环的迭代空间进行分裂，将上述循环转换成图 5(b)所示的两个循环。



(a)原始循环　(b)循环分裂后

图 5　含半不变条件语句的循环结构示例

Fig. 5　Example of a loop structure with semi-invariant condition statements

分裂后的循环二不再对 b 进行条件判断。此外，如果“statements”不包含任何内容，则第二个循环可以删除；如果“statements”是直线指令，那么在后面的优化过程中将有机会对第二个循环进行变换或向量化，达到提升程序性能的效果。

如前文所述，可以在一个具有“半不变”性质的条件语句处进行循环分裂。因此，分析条件语句的半不变性是实施循环分裂的关键步骤。由于编译器使用 SSA 形式的控制流图作为程序的中间表示，分析过程也使用此种类型的示例叙述。

3.2 循环头 PHI 节点条件变量分析

根据静态单赋值形式下条件语句中的条件变量所在 PHI 节点的位置将其区分为两种：循环头 PHI 节点条件变量和合并 PHI 节点条件变量。

首先从一个简单的示例开始分析，接着扩展到一种更加复杂的形式。

图 6 展示了 test1 的代码及其控制流图。判断条件语句的半不变性，即判断条件变量 x_1 的半不变性。定义条件变量 x_1 的 PHI 节点位于循环头，其参数的来源有两个：一个是来自循环外部的初始值 x_0；另一个是通过循环的 latch 边回流得到的值 x_3，称为 latch value。循环外部的定义在进入循环后总是保持不变，因此循环头中 PHI 节点的定义取决于 latch value 的值。latch value 来自于控制流交汇处的 PHI 节点。当条件语句执行时，latch value 可能被不同来源的参数所定义。如果一个分支的执行将改变这些定义中的任意一个，而另一个分支不做出任何改变，那么 latch value 是半不变的，故由 latch value 定义的条件变量也是半不变的。在示例 test1 中，条件语句的真分支修改 x_2 的值，而假分支不对任何参数做出改变，这说明 x_3 和 x_1 是半不变的。

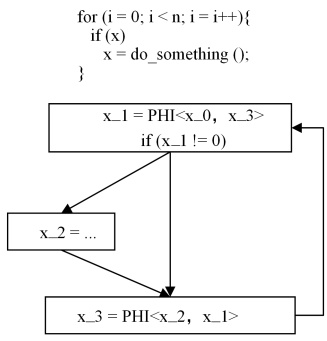


图 6 test1 的代码及控制流图

Fig. 6 Code and control flow graph of test1

3.3 合并 PHI 节点条件变量分析

图 7 为 test2 的代码及控制流图，示例循环中含有两个条件语句。对于第一个条件语句，变量的定义语句为循环头 PHI 节点，依照前文的分析方法判断循环头条件变量的半不变性，结果为非半不变性。对于第二个条件语句，变量 v_8 的定义语句为控制流合并处的 PHI 节点。分支的执行并不会直接修改条件变量的值，而是通过改变循环头中的条件变量 ga_28，间接地影响合并 PHI 节点中的参数。因此，在分析控制流合并处的 PHI 节点时，除了要考虑分支对参数定义位置的影响外，还需要考虑分支是否会改变参数的流入。当分支既不影响参数定义，也不影响参数的流入，那么分支对合并 PHI 节点的定义不会做出任何修改。

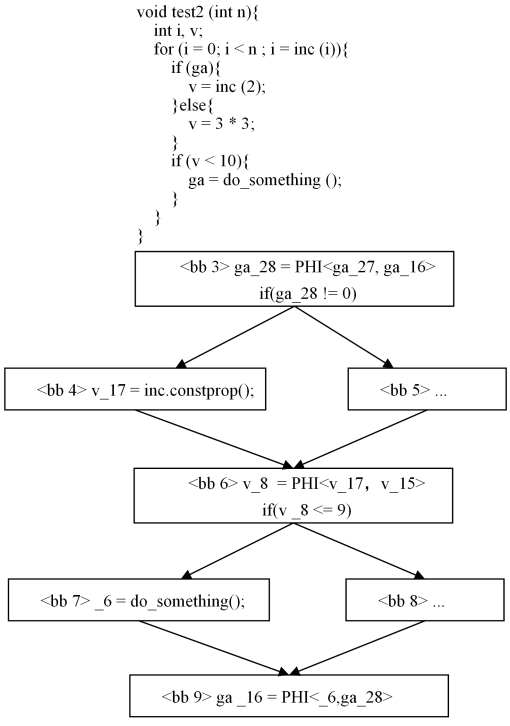


图 7 test2 的代码及控制流图

Fig. 7 Code and control flow graph of test2

分析参数定义的具体方法是：对于合并 PHI 节点中的每个参数，定义在循环外部的变量是不变的，定义在循环内部的变量追踪其定义语句。变量的定义语句类型分为 PHI 节点语句和非 PHI 节点语句两种。对于非 PHI 节点语句，遍历语句中的 SSA 操作数(即算法 1 第 9 行)，获取操作数的定义语句和定义语句所在基本块，定义在循环外部的变量一定是不变的；否则递归调用语句半不变性算法，对定义语句进行追踪分析。对于 PHI 节点类型的语句，进一步区分为位于循环头 PHI 节点和合并 PHI 节点。由于该算法需要多次递归调用算法自身分析语句的半不变性，因此使用哈希表存储语句的半不变性状态，已经完成分析的语句直接读取状态值，未存储对应状态值的语句进入后续分析。

分支对参数流入的影响可以通过检测其控制不变性获得。令 src 是流出 PHI 参数的基本块，且控制依赖于 $\{pb1, pb2, pb3, \cdots, pbm\}$，如果对于任意 pbi 末尾的条件语句 cond，分支的执行不改变条件变量的值，则分支不影响 PHI 参数的流入。

判定语句半不变性的具体过程如算法 1 所示。

算法 1 判断语句半不变性

输入：loop，stmt，bb，skip，stmt_stat

输出：stmt 半不变返回 true，否则 false

1. stmt_stat←∅
2. existed，invar←stmt_stat. get(stmt)
3. if(existed) return invar
4. if(stmt 是 PHI 语句)
5. if(stmt 位于循环头)
6. 判断循环头 PHI 节点半不变性
7. else 判断合并 PHI 节点的半不变性
8. else
9. foreach name：use oprands of stmt
10. if(name 不是半不变的) return false
11. 令 stmt 位于基本块 bb
12. if(!基本块 bb 控制半不变于 skip) return false
13. stmt_stat. put(stmt，true)；
14. return true.

算法 1 中，输入是待分析循环、待分析语句、语句所在基本块、基本块的直接后继基本块以及存储语句不变性状态的变量。如果待分析语句相对于直接后继基本块是半不变的，则返回 true，否则返回 false。

对于 test2 的第二个条件语句，变量定义节点为合并 PHI 节点，第一个参数 v_17 定义在循环内部，其定义语句为非 PHI 节点语句，并且不含 SSA 操作数，直接判断控制流半不变性，其定义语句所在基本块 bb 4 中参数的流入受条件语句后继基本块 bb 7 的影响，并且控制不变于条件语句的后继基本块 bb 8，该参数是半不变的；第二个参数定义在循环外部为不变的，因此第二个条件语句是半不变的。

3.4 算法复杂度分析

算法 1 的时间复杂度主要取决于语句操作数的遍历和状态查询操作。首先，算法通过哈希表查询语句状态，时间复杂度为 O(1)。对于 PHI 语句，算法分别处理循环头和合并 PHI 节点的半不变性判断，时间复杂度为 O(1)。对于非 PHI 语句，算法遍历其操作数，时间复杂度为 $O(N)$，其中 N 为操作数的数量。此外，算法还需判断基本块控制流的半不变性，时间复杂度为 O(1)。综合来看，算法 1 的时间复杂度为 $O(N)$，其中 N 为语句操作数的数量。由于 N 通常较小，因此算法运行时的复杂度较低，具有较好的运行时性能。

4　面向半不变条件语句的循环分裂

基于前文条件语句半不变性的分析方法,实现了面向半不变条件语句的循环分裂优化。本章介绍该优化的设计与实现,整体流程如图 8 所示。

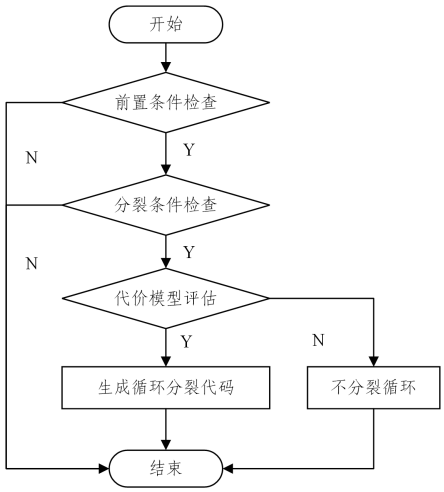


图 8　面向半不变条件语句的循环分裂流程

Fig. 8　Loop splitting process for semi-invariant conditional statements

4.1　前置条件检查

在循环分裂的初始阶段,编译器首先对目标循环进行前置条件检查,以排除不满足基本条件的循环,避免无效的优化操作。具体检查内容包括 4 个方面:1)验证循环是否具有单一出口,并检查其 latch 块是否为空;2)要求循环退出条件为简单表达式,且不包含不等于操作符;3)循环分裂需要复制循环体,因此循环体内不得包含不可复制的结构(如复杂控制流或外部依赖);4)在嵌套循环中,条件变量的作用域分析困难,并且嵌套循环分裂会导致代码体积膨胀,因此改进的循环优化方法与编译器现有循环分裂策略保持一致,即仅适用于最内层循环。

4.2　分裂条件检查

在确认目标循环满足前置条件后,编译器进一步检查循环内是否存在符合分裂要求的基本块。一个循环内可能有一个或多个条件语句,条件语句位于基本块的末尾,通过遍历循环内的基本块,并判断每个基本块的末尾语句是否是条件语句类型来定位条件语句,条件语句所在基本块的两个后继基本块分别对应条件语句的真分支和假分支,使用数组分别标记条件语句相对于两个分支的不变性状态,利用语句半不变性判定算法判断其半不变性。如果条件语句相对于两个后继基本块的不变性状态相异,则该条件语句为半不变的,可以进行循环分裂,分裂位置为条件语句所在基本块流向不变分支的边。

4.3　代价评估模型

循环分裂的代价模型主要约束分支执行概率和指令增加数量。

分支执行概率,即控制从源基本块传递到目标基本块的概率。GCC 中以分支执行概率估算程序块的执行频率,编译器在程序执行时以反馈优化的形式收集分支信息并生成分支执行概率,分支执行概率不能超过编译器中给定的分支执行概率下限 0.3。

指令增加数量,即因分裂增加的指令数。编译器通过解析语句的类型,如赋值、条件、调用等,基于特定的权重估算给定类型的语句在展开后的指令数量,以指令数目估算代码量。式(1)给出了将不变分支分裂至第二个循环后增加指令数的计算方法:

$$num=\sum_{bb\in loop}\begin{cases}0, & \text{if } bb \text{ 被 } opposite\_bb \text{ 支配}\\ estimate\_num\_insns(bb), & \text{否则}\end{cases} \tag{1}$$

其中,$opposite\_bb$ 是不变分支的相反分支的首个基本块(如果不变分支为条件语句的假分支,则 $oppsite\_bb$ 为真分支的第一个基本块);函数 $estimate\_num\_insns$ 估算语句展开后的指令数目。$param\_max\_peeled\_insns$ 代表编译器预定义的最大允许值,默认值是 100。当增加的指令数量超过预定义的最大允许值时,不进行分裂操作。

4.4　代码生成

在对条件语句进行半不变性判定并通过收益评估后,对符合要求的循环结构中的条件分支进行分裂。首先,将包含条件语句的原始循环复制为两个循环。原始循环由前置头部基本块、循环头基本块、条件语句、循环不变分支、循环体内的其他语句块和循环体的回边基本块组成,其中前置头部基本块通常包含初始化代码,回边基本块用于控制循环的迭代。

由于循环不变分支在后续的每次迭代中保持不变,因此在确定分裂位置后,将不变分支移至第二个循环中。具体步骤为:首先在原始循环的回边基本块中插入一个新的条件语句,该语句是由半不变条件语句复制而来的;新的条件语句的一个分支返回到第一个循环的循环头作为回边,而另一个分支则进入第二个循环的前置头部,作为入口边。在第二个循环中,舍弃条件语句中的变化分支,并将条件语句替换为恒定布尔条件,确保在循环迭代次数达到预定条件时,退出循环。在代码转换过程结束后,调用函数 $update\_ssa()$ 对循环的静态单赋值形式进行更新,同时重新计算基本块之间的支配关系。

经过变换后,初始循环被分裂为两个循环:第一个循环处理原始的变化部分,第二个循环处理后续迭代中的不变部分。由于第二个循环的条件分支被简化为恒定值,因此其有助于提高分支预测的准确性。分裂后的循环在后续迭代中减少了不必要的条件判断,使代码执行更高效。此外,分裂后的循环可以更好地利用 CPU 缓存,增强代码执行的局部性。

5　实验测试与分析

5.1　实验配置

所提出的方法在申威 GCC-10.3.0 版本编译器中实现,硬件平台采用申威(Sunway)3231 处理器。实验对比了优化前后的编译器性能,以评估所提出方法的有效性。实验平台信息如表 1 所列,实验平台配备 Sunway 3231 型号的 CPU,采用 SW-64 指令集架构,CPU 主频为 2.4 GHz,内存为 128 GB,L1 指令缓存容量为 32 kB,操作系统为 Deepin Linux。

表 1 实验平台信息

Table 1 Information of experimental platforms

名称	配置环境
CPU 型号	Sunway 3231
架构	SW-64
操作系统	Deepin Linux
CPU 内核数	32
内存/GB	128
主频/GHz	2.4
编译器版本	SWGCC-10.3.0

测试集采用 SPEC CPU2006 和 SPEC CPU2017 标准基准套件。SPEC 基准测试涵盖了多种应用场景和编程语言(如 C,C++和 Fortran),主要用于评估单线程 CPU 的性能,关注处理器、内存子系统和编译器的表现。SPEC2006 包括两组测试:SPECint2006(12 个整型基准)和 SPECfp2006(17 个浮点基准)。SPEC2017 在此基础上更新,包含 SPECint2017(10 个整型基准)和 SPECfp2017(13 个浮点基准)。两套测试均支持 test,train 和 ref 3 种测试规模,分别用于验证结果的正确性、优化反馈及标准性能评估。

5.2 实验方法

为了验证所提方法的有效性,从 SPEC CPU2006 和 SPEC CPU2017 测试集中选取具有相应循环分裂特征的课题进行测试,测试规模均采用 ref。针对图 5 所示的循环结构,原有的分裂算法无法处理,而本文提出的改进算法可以对该循环结构进行分裂。在编译以及运行条件相同的情况下,通过对比同一程序使用优化算法前后执行时间的方式,判断优化算法的效果。具体的测试方法为:

1)运行并统计原有循环分裂算法下的运行时间 T_1。

2)运行并统计优化后的循环分裂算法下的运行时间 T_2。

3)以 T_1 为基准,检查优化后的循环分裂算法是否对测试集的运行结果产生负面影响,若无负面影响,则表明该优化方法正确。

4)以 T_1 为基准,计算优化后程序的加速比提升,具体计算方法为 $\left(\frac{T_1}{T_2}-1\right)\times 100\%$,并据此评估优化方法对程序性能的提升效果。

为降低测试环境波动的干扰,每个基准程序至少运行 3 次,并取平均执行时间作为最终实验数据。

5.3 实验结果与分析

表 2 列出了采用不同版本的循环分裂算法对 SPEC 部分基准测试进行加速比对比的结果。

表 2 循环分裂算法修改前后课题运行时间统计

Table 2 Runtime statistics of benchmarks before and after loop splitting algorithm modifications

程序名称	原循环分裂版本运行时间/s	改进的循环分裂版本运行时间/s	加速/%
447.dealII	932	916	1.75
450.soplex	965	934	3.32
465.tonto	802	761	5.39
470.lbm	853	784	8.78
483.xalancbmk	741	728	1.79
600.perlbench_s	1376	1349	2.00
620.omnetpp_s	1001	960	4.27

实验结果表明,改进的循环分裂版本相较于原循环分裂算法版本,在多个 SPEC 基准测试上均表现出明显的加速效果。在 SPEC2006 中的 470.1bm 基准测试上,优化后的版本实现了 8.78%的加速;在 SPEC2017 中的 620.omnetpp_s 基准测试上,优化后的版本实现了 4.27%的加速。此外,多个基准测试的加速效果均超过 1.75%,且未发现对基准测试产生负加速的情况。

通过开启-O2 -fsplit-loops -fdump-tree-lsplit-details 选项分析编译过程中循环分裂优化后的中间信息,选取了如图 9 所示的循环结构示例。可以看出,使用优化前的循环分裂算法运行时,该循环未实现循环分裂,而优化后的循环分裂算法版本成功对该循环结构实现了循环分裂。

```
for (GateIterator i(mod); !i.end(); i++){
  const cGate *gate = i();
  if (gate && gate->getChannel() &&
      gate->getChannel()->getType() == ChannelType::FAST)
    channels.push_back(gate->getChannel());
}
```

图 9 优化算法生效的循环分裂示例

Fig. 9 Example of loop splitting enabled by the optimizing algorithm

该循环内的条件语句由多个逻辑与(AND)运算构成,具体包括对 gate 指针的非空检查、gate 所关联对象的存在性检查以及关联对象的类型是否为 ChannelType::FAST 的验证。在编译器的中间表示中,这些条件表现为多层嵌套的 if 语句结构。在循环分裂优化过程中,首先对最外层条件语句(即 gate 指针的非空检查)进行半不变性分析,确认其在循环迭代过程中是否保持稳定。基于分析结果,优化算法在该条件的假分支处对循环进行分裂,使得分裂后的循环体在后续迭代过程中无需重复判断内层条件,从而减少分支控制开销。

为了验证所提出优化算法对目标循环的有效性,在源代码中引入 clock_gettime 函数对目标循环代码段的运行时间进行测量,取 3 次测量结果的平均值来降低系统抖动带来的误差。测试结果如表 3 所列。由表 3 可知,使用改进的循环分裂优化算法后,目标循环的平均运行效率提升 25.8%。

表 3 循环分裂算法的性能对比

Table 3 Performance comparison of loop splitting versions

循环运行时间/ms		性能提升/%
原循环分裂版本	改进的循环分裂版本	
18.736	13.902	25.8

结束语 本文围绕 GCC 编译器中循环分裂优化策略适用范围受限的问题展开研究,提出了一种基于条件语句半不变性分析的改进算法,并在 SPEC CPU2006 和 SPEC CPU2017 测试集中相关课题上进行了实验评估,实验结果证明了该算法能够拓展循环分裂适用范围,提升应用程序的执行效率。

循环分裂通常处于编译流程中 GIMPLE 阶段的前期,随着中间优化过程的持续进行,编译器将在 GIMPLE 阶段继续应用条件语句转换(if-conversion)、控制流简化等技术,对循环结构进行进一步优化。考虑到接下来的优化过程可能会显

著改变循环结构,下一步工作将继续探索在完成 GIMPLE 阶段优化后,将循环分裂策略延伸至 RTL 阶段,进一步分析循环结构,提升该技术在编译流程中的优化能力。

参 考 文 献

[1] SENGUPTA A,BHADAURIA S. Intellectual property core protection of control data flow graphs using robust watermarking during behavioural synthesis based on user resource constraint and loop unrolling factor[J]. Electronics Letters,2016,52(6):439-441.

[2] ZIRAKSIMA M,LOTFI S,LZADKH H. Using an evolutionary approach based on shortest common supersequence problem for loop fusion[J]. Soft Computing,2019,24(10):1-22.

[3] BIELECKI W,PALKOWSKI M. Space-time loop tiling for dynamic programming codes[J]. Electronics,2021,10(18):2233.

[4] GAO W,XU J L,SUN H H,et al. Research on loop optimization techniques for SIMD vectorization[J]. Journal of Information Engineering University,2016,17(4):496-503.

[5] TSIRAMUA S,MELADZE H,DAVITASHVILI T,et al. Structural Analysis of Multi-Core Processor and Reliability Evaluation Model[J]. Mathematics,2025,13(3):515.

[6] YANG X C,JIANG J,MA X D,et al. Program Slicing Technique Based on GCC Key Variable Data Flow Analysis Algorithm[J]. Computer Engineering and Applications,2017,53(24):40-54.

[7] WANG S D,YIN W J,DONG Y K,et al. Data Flow Analysis for Sequential Storage Structures[J]. Journal of Software,2020,31(5):1276-1293.

[8] GAO W,LI Y Y,SUN H H,et al. An Improved Control Flow SIMD Vectorization Method [J]. Journal of Software,2017,28(8):2046-2063.

[9] SWEET Y,CHAUDHARY P. An efficient branch predictor for improved accuracy of instruction level parallelism[J]. The Journal of Supercomputing,2021,77(10):1-23.

[10] ROGERS S,SLYCORD J,RAHEJA R,et al. Scalable LLVM-Based Accelerator Modeling in gem5[J]. Computer Architecture Letters,2019,18(1):18-21.

[11] JEONG S J. A loop splitting method for single loops with non-uniform dependences[J]. Journal in computer virology:An independant journal dedicated to computer viral and antiviral technologies,2014,10(2):137-143.

[12] LIU J,WICKERSON J,CONSTANTINIDES G. Loop splitting for efficient pipelining in high-level synthesis[C]// IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines(FCCM). IEEE,2016:72-79.

[13] XU R F,ZHAO Z Y,SHENG W G,et al. An Imperfect Loop Mapping Method for Coarse - Grained Reconfigurable Architectures[J]. Microelectronics & Computer,2018,35(7):50-57.

[14] YANG X. Advanced Loop Optimization Base d on Tree-SSA Framework[J]. Computer Knowledge and Technology,2009,5(24):7035-7037.

[15] DONG Y S,LI C J,XU Y. Implementation and Evaluation of Loop Array Prefetching Optimization in GCC Compiler[J]. Computer Engineering and Applications,2016,52(6):19-25.

[16] BERG D V J,HAGA H. Matching Source Code Using Abstract Syntax Trees in Version Control Systems[J]. Journal of Software Engineering and Applications,2018,11(6):318-340.

[17] COLOMBET Q,BRANDNER F,DARTE A. Studying Optimal Spilling in the Light of SSA[J]. ACM Transactions on Architecture and Code Optimization(TACO),2015,11(4):1-26.

[18] CHEN M Y. Research on Loop Distribution Techniques for Sunway GCC Compilation System[D]. Zhengzhou:Zhengzhou University,2021.

[19] HAN L,XU J L,LI Y Y,et al. Loop Distribution and Fusion Optimization for Partial Vectorization[J]. Computer Science,2017,44(2):70-81.

[20] GAO W,HAN L,ZHAO R C,et al. Vector Parallelism-Guided Loop SIMD Vectorization Method[J]. Journal of Software,2017,28(4):925-939.

[21] LIM I H. An approach to comparing control clow graphs based on basic block matching[J]. Indian Journal of Computer Science and Engineering,2020,11(3):289-296.

[22] 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,1991,13(4):451-490.

[23] XIE X F,CHEN B H,ZOU L,et al. Automatic Loop Summarization via Path Dependency Analysis[J]. IEEE Transactions on Software Engineering,2019,45(6):537-557.

HAN Lin, born in 1978, Ph.D, professor, Ph.D supervisor, is a senior member of CCF(No. 16416M). His main research interests include high-performance computing, advanced compilation techniques, program optimization and indigenous innovation of key technologies.

NIE Kai, born in 1997, Ph.D, lecturer, master's supervisor. His main research interests include advanced compilation technologies and high-performance computing.

(责任编辑:何杨)