计算机科学
COMPUTER SCIENCE

基于循环代价分析的循环不变量外提算法

姜军, 翟彦河, 曾志恒, 顾轶超, 黄亮明

引用本文

姜军, 翟彦河, 曾志恒, 顾轶超, 黄亮明. 基于循环代价分析的循环不变量外提算法[J]. 计算机科学, 2025, 52(6): 44-51.

JIANG Jun, ZHAI Yanhe, ZENG Zhiheng, GU Yichao, HUANG Liangming. Loop-invariant Code Motion Algorithm Based on Loop Cost Analysis [J]. Computer Science, 2025, 52(6): 44-51.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于申威平台寄存器溢出策略的预选先验优化

Pre-selection Optimization for Spill Heuristic on Shenwei Platform

计算机科学, 2025, 52(6): 82-87. https://doi.org/10.11896/jsjkx.240800128

面向申威平台的SIMD编程接口设计与研究

Design and Research of SIMD Programming Interface for Sunway

计算机科学, 2025, 52(6): 66-73. https://doi.org/10.11896/jsjkx.240700009

高阶密码算子在FPGA的编译优化与实现

Compilation Optimization and Implementation of High-order Cryptographic Operators on FPGA

计算机科学, 2024, 51(11A): 231200184-11. https://doi.org/10.11896/jsjkx.231200184

基于国产c86处理器的CP2K软件移植与优化

CP2K Software Porting and Optimization Based on Domestic c86 Processor

计算机科学, 2023, 50(6): 58-65. https://doi.org/10.11896/jsjkx.230200213

基于GCC编译器的流式存储优化方法

Optimization Method of Streaming Storage Based on GCC Compiler

计算机科学, 2022, 49(11): 76-82. https://doi.org/10.11896/jsjkx.211200252

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240300166

基于循环代价分析的循环不变量外提算法

姜 军 翟彦河 曾志恒 顾轶超 黄亮明
无锡先进技术研究院 江苏 无锡 214122
(goodsun_jj@163.com)

摘 要 循环不变量外提算法是一种针对程序中循环结构的常用编译优化算法,其通过将循环体中的不变计算移动到循环外部来减少重复计算的开销,从而提高程序运行的速度。但在LLVM编译器中,传统的循环不变量外提算法会将全部循环不变量外提到循环体外部,当循环不变量达到一定数量时会导致寄存器溢出,在循环内引入额外的访存代价,对循环产生负优化效果。针对上述问题,在传统LLVM循环不变量外提算法的基础上,引入了一种循环代价分析算法,通过计算循环不变量在循环体中的运行代价和外提操作可能带来的溢出代价,评估其外提可能带来的收益,只对产生正收益的循环不变量进行外提,在有效减少循环体内重复计算的同时,规避引入额外开销的风险。在国产申威831处理器平台,使用典型用例进行优化效果测评,在千万级循环下,相较于传统循环不变量优化算法,提出的新优化算法具有17%以上的性能提升;使用SPEC CPU2017基准测试集(SPECspeed 2017 Interger套件)、Perl解释器DKbench基准测试集、Python解释器pyperformance基准测试集进行综合优化效果测评,结果表明,相较于传统循环不变量优化算法,提出的新优化算法分别具有0.4%,0.63%和1%的性能提升。

关键词:LLVM编译器;编译优化;循环不变量外提;寄存器溢出;循环代价分析
中图分类号 TP314

Loop-invariant Code Motion Algorithm Based on Loop Cost Analysis

JIANG Jun, ZHAI Yanhe, ZENG Zhiheng, GU Yichao and HUANG Liangming
Wuxi Institute of Advanced Technology, Wuxi, Jiangsu 214122, China

Abstract Loop-invariant code motion(LICM) is a commonly used compilation optimization algorithm for loop structures in programs. By moving the invariant calculations in the loop body to outside the loop, the algorithm reduces the overhead of duplicate calculations, thus improving program execution speed. However, in LLVM compiler, the traditional LICM algorithm hoists all loop-invariants outside the loop body, which will lead to register overflow when the number of loop-invariant reaches a certain level. It will introduce additional memory access cost in the loop, resulting in a negative optimization effect on the loop. To address this issue, a loop cost analysis algorithm is introduced based on the traditional LLVM LICM algorithm. This algorithm evaluates the running cost of loop-invariant code inside the loop and the overflow cost that may be caused by moving the code outside the loop, and assesses the benefits of moving the code outside the loop. Only the loop-invariant code that produces positive benefits is moved outside the loop, effectively reducing the overhead of duplicate calculations in the loop while avoiding the risk of introducing additional costs. The proposed new optimization algorithm achieves more than 17% performance improvement compared to the traditional LICM algorithm in typical use cases for the domestic SW831 processor platform under millions of loops. Comprehensive optimization effect evaluations are conducted using the SPEC CPU 2017 benchmark test suite(SPECspeed2017 Integer Suite), Perl interpreter DKbench benchmark test suite, and Python interpreter pyperformance benchmark test suite. The results show that compared with the traditional LICM algorithm, the proposed algorithm has 0.4%, 0.63% and 1% performance improvement respectively.

Keywords LLVM compiler, Compilation optimization, Loop-invariant code motion, Register overflow, Loop cost analysis

1 引言

LLVM(Low-Level Virtual Machine)[1]是一个开源的编译器基础设施项目,旨在提供一个通用的编译器框架,用于各种编程语言和目标架构。其具有轻量级、模块化设计以及高度可扩展性等特点,被广泛应用于云计算、人工智能、科学计算、游戏设计等领域[2-5]。随着这些领域的快速发展,应用程序代码的复杂度越来越高,数据计算量越来越大。为了使程序能够高效地运行,LLVM的编译优化引起了学术界和工业界的广泛关注[6-8]。

在大数据、云计算等场景中经常出现复杂的数学计算,LLVM编译器通常将其编译生成多层嵌套循环结构,每层循环可能存在与循环变量无关的复杂表达式计算,如果每次循环都进行相关计算,会浪费计算机资源,影响性能。为消除循环内的冗余计算,LLVM利用循环不变量外提(Loop-Invariant Code Motion,LICM)优化算法[9]识别循环中的不变量,并

到稿日期:2024-03-26 返修日期:2024-07-15
基金项目:科技部重点支持项目(GG20210701)
This work was supported by the Key Project of the Ministry of Science and Technology(GG20210701).
通信作者:黄亮明(liangming_huang@126.com)

将其提到循环体外进行计算，有效避免了大量重复的计算，提高了代码生成质量和程序运行性能。

但在 Perl 解释器场景中，其热点函数中存在一种由 while+switch/goto 组成的特殊循环结构，且每个 case 中包含一定量用于访问结构体成员的地址计算操作。在 LLVM 中，这类操作被称为 GEP 节点（Get Element Ptr）[10]。代码示例如下：

```
while() {
Label:
    switch() {
        ……
        case A:
            GEP;
            goto Label;
        case B:
            GEP;
            goto Label;
        ……
    }
}
```

针对此类场景，传统 LICM 算法将作为循环不变量的 GEP 全部外提，使用大量虚拟寄存器存放 GEP 结果，在寄存器分配阶段产生溢出，GEP 结果被存储在栈上，导致循环中原始的 GEP 从低延迟的简单运算操作变成高延迟的访存操作，降低了循环的运行效率。该现象表明，传统的 LICM 算法存在一定缺陷，无法适用于部分特殊场景。

针对上述问题，本文提出了一种基于循环代价分析的 LICM 算法，对循环不变量外提操作进行代价分析，在确定其外提不会加重循环负担的情况下再执行外提操作，有效避免了优化使得循环运行效率下降的情况。同时，为了验证该算法的有效性，在 LLVM 13.0.0 编译器上进行实现，并在国产申威 831 处理器平台上针对典型样例与基准测试集进行了优化效果验证。

2　研究背景

2.1　LLVM 编译器系统结构

LLVM 是以 C++语言为基础编写的开源编译器项目，它提供了一套强大的工具和框架，用于构建编译器、即时编译器、静态分析工具和其他与编译器相关的软件。LLVM 同时支持多种前端语言，如 C，C++，Rust 等，以及多种目标架构，如 X86，AArch64，RISCV 等。LLVM 的整体架构采用传统的三段式结构设计，如图 1 所示，将编程语言转换到目标机器代码的过程分为前端、中端、后端 3 个部分。

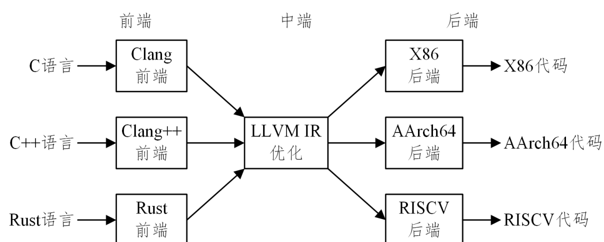

图 1　LLVM 整体结构

Fig. 1　LLVM architecture

LLVM 前端通过词法分析、语法分析、语义分析等过程将编程语言转换为一种高度抽象的中间表示 LLVM IR（Intermediate Representation），这是一种静态单赋值（Static Single Assignment，SSA）[11] 形式的表示，有助于编译器适用于多种语言和架构。

LLVM 中端的任务是对 IR 进行一系列的优化操作[12]，形成更高效的 IR。LLVM 目前针对 IR 已经实现了多种优化方法，它们可以分为以下几类。

1）基础块（Basic Block）级优化：对基本块内部的指令进行优化调整，如指令合并[13]、常量折叠、无用代码删除[14]等。

2）函数级优化：对整个函数的结构进行优化，如函数内联[15]等。

3）循环优化：对循环结构进行优化，包括循环展开[16]、循环合并、循环不变量外提、循环剥离[17]等。

4）控制流优化：优化分支和跳转的控制流，如控制流简化[18]等。

5）数据流分析：通过数据流分析来进行更高级别的优化，包括过程内和过程间数据流分析[19]。

此外，LLVM 中端优化模块可以按照一定的顺序应用不同的优化。通常，LLVM 使用一系列优化遍（Optimization Passes），每个遍都会执行一组相关的优化。优化遍的顺序可以由用户配置，以满足特定的性能目标或需求。

LLVM 后端的主要任务是将经过优化后的 IR 转化为目标机器代码[20]。这个过程涵盖了多个阶段，如图 2 所示。其中，右侧的优化遍是可选项，能够结合中端优化进一步提高生成代码的质量。左侧的框表示后端的各个必要阶段，其中指令选择阶段会根据 IR 中的指令匹配相应的目标机器指令[21]；指令调度阶段用于调整生成的机器指令的顺序，以最大程度地利用目标机器的流水线和其他硬件特征[22]；寄存器分配阶段为虚拟寄存器分配目标机器的物理寄存器[23]；代码发射阶段将最终的目标机器指令生成到汇编代码或目标机器代码。

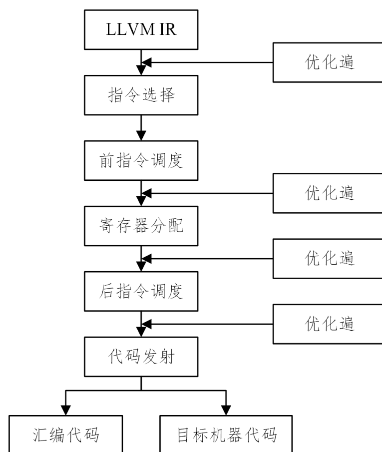

图 2　LLVM 后端代码生成流程图

Fig. 2　LLVM backend codegen flow chart

2.2　LLVM 循环不变量外提

如图 3 所示，LLVM 的循环结构主要由 preheader 基本块、循环体和 exit 基本块组成（虚线表示路径上可存在多个基本块）。其中，循环体内由多个循环内部基本块相互连接形成

环路，完成循环计算；preheader 和 exit 基本块分别负责循环前、循环后的工作。

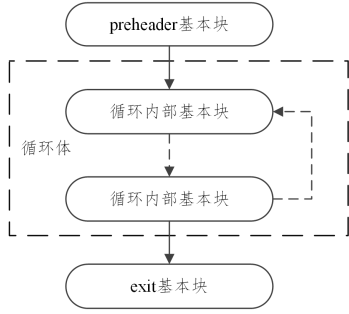

图 3 LLVM 循环结构

Fig. 3 LLVM loop structure

循环不变量外提的基本思想是识别循环内部基本块中取值不随循环发生变化的表达式，在不影响程序正确性的前提下，尽可能多地将其外提至 preheader 基本块或 exit 基本块[24]。此类表达式包括常量、在循环外赋值的变量、循环中保持相对固定的运算表达式、内存地址或偏移量等。

LLVM 在中端优化模块和后端代码生成模块中分别设置了 LICM 优化遍，前者在 IR 层面进行初步 LICM 优化，后者结合寄存器压力和指令延迟进一步进行 LICM 优化。

3 传统 LICM 算法分析

本章在理论上分析了对循环结构进行循环不变量外提优化会产生的收益，并设计了典型样例，结合汇编代码阐述了传统的 LICM 算法在一些特殊场景中的缺陷。

3.1 循环不变量外提代价分析

如 2.2 节所述，一个任意层数的循环可以分为 3 个部分，即 preheader 基本块、循环体、exit 基本块，其中循环体内又包含多个内部基本块。使用 C_{P}，C_{B}，C_{E} 表示 3 个部分的运行代价，用 C_{loop} 表示整个循环的运行代价，则 C_{loop} 可以表示为：

$$C_{\mathrm{loop}}=C_{\mathrm{P}}+C_{\mathrm{B}}+C_{\mathrm{E}} \tag{1}$$

而循环体的运行代价与循环体的执行次数以及循环体内的每个基本块的运行代价和执行频率相关，可以表示为：

$$C_{\mathrm{B}}=L*\sum_{i=1}^{n}C_{bi}*f_{bi} \tag{2}$$

其中，L 表示循环执行次数，n 表示循环体中基本块的总数，C_{bi} 表示循环体中每个基本块的执行代价，f_{bi} 表示循环体内基本块的执行频率。将式(2)代入式(1)，可得循环的运行代价为：

$$C_{\mathrm{loop}}=C_{\mathrm{P}}+L*\sum_{i=1}^{n}C_{bi}*f_{bi}+C_{\mathrm{E}} \tag{3}$$

从式(3)可以看出，随着循环次数的增加，循环的主要运行代价会由循环体的运行代价决定。当循环体内存在与循环变量不相关的复杂表达式计算时，大量重复计算会严重影响整个循环的运行效率。LICM 优化算法的核心思想是通过将循环体中的循环不变量外提，来降低循环体的运行代价，从而降低整个循环的运行代价。

用 C_{B}' 表示循环不变量外提后循环体的运行代价：

$$C_{\mathrm{B}}'=L*\sum_{i=1}^{n}(C_{bi}-C_{vi})*f_{bi} \tag{4}$$

其中，C_{vi} 是每个基本块中循环不变量执行的代价。

但在将循环不变量外提后，如果其在循环体内被引用会被提升到 preheader 基本块，反之会下沉到 exit 基本块。分别用 C_{pvi} 和 C_{evi} 表示每个基本块中循环不变量外提操作带来的提升到 preheader 基本块和下沉到 exit 基本块的运行代价，则执行 LICM 算法后，preheader 基本块和 exit 基本块的运行代价 C_{P}' 和 C_{E}' 可以表示为：

$$C_{\mathrm{P}}'=C_{\mathrm{P}}+\sum_{i=1}^{n}C_{\mathrm{pvi}} \tag{5}$$

$$C_{\mathrm{E'}}=C_{\mathrm{E}}+\sum_{i=1}^{n}C_{\mathrm{evi}} \tag{6}$$

根据式(1)，在将循环不变量外提后，循环的整体运行代价 C'_{loop} 为：

$$\begin{aligned}C'_{\mathrm{loop}}&=C_{\mathrm{P}}'+C_{\mathrm{B}}'+C_{\mathrm{E}}'\\&=C_{\mathrm{P}}+\sum_{i=1}^{n}C_{\mathrm{pvi}}+L*\sum_{i=1}^{n}(C_{bi}-C_{vi})*f_{bi}+C_{\mathrm{E}}+\sum_{i=1}^{n}C_{\mathrm{evi}}\end{aligned} \tag{7}$$

利用式(3)减去式(7)，可得循环不变量外提后整体循环运行减少的代价 C_{reduce}：

$$C_{\mathrm{reduce}}=L*\sum_{i=1}^{n}C_{vi}*f_{bi}-\sum_{i=1}^{n}C_{\mathrm{hvi}}+C_{\mathrm{evi}} \tag{8}$$

而在实际的 LICM 算法执行中，由于运行机器寄存器数量有限，可能会导致部分不变量被存放在内存中，在 preheader 基本块和循环体中会存在额外的访存操作。分别用 C_{psti} 和 C_{brei} 表示这两部分的运行代价，则实际执行 LICM 算法后，循环的运行代价可以表示为：

$$\begin{aligned}C'_{\mathrm{loop}}=&C_{\mathrm{P}}+\sum_{i=1}^{n}C_{\mathrm{pvi}}+C_{\mathrm{psti}}+L*\sum_{i=1}^{n}(C_{bi}-C_{vi}+C_{\mathrm{brei}})*\\&f_{bi}+C_{\mathrm{E}}+\sum_{i=1}^{n}C_{\mathrm{evi}}\end{aligned} \tag{9}$$

利用式(3)减去式(9)可得，在实际进行循环不变量外提后，循环运行代价降低了。

$$C_{\mathrm{reduce}}=L*\sum_{i=1}^{n}(C_{vi}-C_{\mathrm{brei}})*f_{i}-\sum_{i=1}^{n}C_{\mathrm{pvi}}+C_{\mathrm{psti}}+C_{\mathrm{evi}} \tag{10}$$

从式(10)可以看出，LICM 对循环的优化依赖于循环体内循环不变量外提后产生的代价降低，即 $C_{vi}-C_{\mathrm{brei}}$ 的值。显然，当 $C_{vi}<C_{\mathrm{brei}}$，即循环不变量在循环体中原本的运行代价小于因为外提操作而产生的额外的访存操作的运行代价时，LICM 不仅不会对循环进行优化，反而会增加循环运行的代价。

使用 CPU 时钟周期作为参考标准，对循环代价计算的准确性进行论证。现代处理器的超标量和乱序执行设计，导致精确地确定不同程序单个指令的 CPU 时钟周期(即 C_{vi})较为困难，因此设计一个典型样例来间接验证公式的准确性。源代码如下：

```
int func(int x,int y,int z) {
  for(int i=0;i<1 000000000;i+=16) {
    a[i]=21 * y+y * z+x;
    x+=y * z;
  }
}
```

在样例循环中有两个不变量，分别是 21 * y 和 y * z，在 O2 优化选项下生成如下汇编代码：

```
func:
  mull $18,$17,$0
  s4addl $17,$17,$1
```

```
  sll $17,4,$21
  addl $21,$1,$1
  ldl $2,a($29)
  ldi $3,-16($31)
.LBB0_1:
  addl $16,0,$16
  addl $16,$1,$4
  stw $4,0($2)
  ldi $3,16($3)
  ldih $4,1526($31)
  ldi $4,-7952($4)
  cmpult $3,$4,$4
  ldi $2,64($2)
  bne $4,.LBB0_1
.LBB0_2:
  bis $31,$31,$0
  ret
```

在上述汇编中,计算不变量 21*y 和 y*z 的指令分别是由 s4addl,sll 以及 addl 组成的指令序列和 mull 指令。通过移动计算循环不变量汇编位置的方法,分别测量 4 个实验场景所耗费的 CPU 时钟周期。

实验 1　仅包含循环控制

实验 2　仅包含循环不变量

实验 3　循环不变量外提

实验 4　循环不变量不外提

实验 1 循环体为空,只测量循环本身所消耗的 CPU 时钟周期;实验 2 测量循环中包含 21*y 和 y*z 两个循环不变量所消耗的 CPU 时钟周期;实验 3 测量在循环不变量外提的情况下,完整循环所消耗的 CPU 时钟周期;实验 4 测量在循环不变量不外提的情况下,完整循环所消耗的 CPU 时钟周期。

测量结果如表 1 所列。

表 1　循环代价测量实验

Table 1　Loopcost measurement experiments

实验编号	实验描述	CPU 时钟周期
实验 1	仅循环控制	12550505
实验 2	仅循环不变量	27141695
实验 3	循环不变量外提	384547297
实验 4	循环不变量不外提	400172340

根据式(8)的推导过程,C_{reduce} 由式(3)减去式(7)得出,即实验 4 的结果减去实验 3 的结果,为 15625043;根据C_{vi}定义,C_{reduce}可由实验 2 的结果减去实验 1 结果得出,为 14591190。由公式推导得出的计算结果与由定义得出的计算结果的比值为 1.07,证明公式具备一定的准确度。

3.2　传统 LICM 缺陷分析

为了更好地阐述传统 LICM 算法在优化部分循环场景时的不足,设计了一个典型用例。源代码如下所示:

```
extern void init_func();
long func(long x,long y,long z,int control){
  for(long i=0;i < 30000000;i +=16){
    switch(control){
    case 0:
      a[i]=x+y;
      a[i+1]=x-y;
      a[i+2]=x*y+z;
      a[i+3]=x*y-z;
      a[i+4]=x∧y-z;
      a[i+5]=x*y-z;
      break;
    case 1:
      a[i+6]=x*y-z;
      a[i+7]=x-y*z;
      a[i+8]=x+y+z;
      a[i+9]=x|y-z;
      a[i+10]=x*y|z;
      a[i+11]=x*y|z;
      break;
    case 2:
      a[i+12]=x∧y|z;
      a[i+13]=x|y∧z;
      a[i+14]=x-y*z;
      a[i+15]=~x+y|z;
      break;
    }
    init_func();
  }
  return 0;
}
```

在上述用例中,for/switch 构成了程序的主循环,在每个 case 节点下都具有一定的计算操作,由于变量 x,y,z 的取值不随循环发生变化,因此基于它们的算术表达式的结果都是循环不变量。

在没有开启传统 LICM 算法优化的情况下,所生成的部分汇编代码如下所示:

```
#preheader
……
#case
.LBB0_7:
……
subl $12,$11,$0
stl $0,0($1)
bis $2,16,$1
addl $14,$10,$0
stl $0,0($1)
br $31,.LBB0_2
```

从汇编代码可以看出,循环体中的循环不变量的运行代价主要是 addl/ subl 指令(长字加法/减法运算)带来的代价,即式(10)中的C_{vi}。

在上述场景中,进行中端优化时,LLVM 传统的 LICM 算法会将符合外提条件的循环不变量全部外提。在后端代码生成中,由于寄存器数量的限制,外提的不变量会有一部分溢出到栈中,导致生成了如下所示的部分汇编代码。

```
#preheader
……
addl $17,$16,$5
stl $5,104($30)
```

```
subl $16,$17,$1
stl $1,72($30)
……
#case
.LBB0_7:
……
ldl $3,104($30)
stl $3,0($2)
addl $11,$3,$5
ldl $6,72($30)
stl $6,0($5)
br $31,.LBB0_2
```

从上述汇编代码可以看出，case 内部的 addl/subl 操作被提取到了循环体外，放到了 preheader 中；并且这些操作的结果被存在栈中，导致循环体内用到该结果时需要从栈中取出，循环体内因为 addl/subl 操作外提而产生的访存代价 C_{brei} 显然大于 addl/subl 操作在原本循环体中的执行代价，导致 LICM 算法不仅没有进行循环优化，反而增加了循环运行的代价，降低了程序的运行效率。

4 基于循环代价分析的 LICM 算法

本章将详细介绍本文提出的基于循环代价分析的 LICM 算法的流程和实现，从中端和后端两个阶段阐述了相对于传统 LICM 算法的改进，并重点介绍了所提循环代价分析算法。

4.1 总体算法设计

传统 LICM 优化遍涉及中端和后端两个部分。中端主要在 IR 层面识别循环不变量，并将识别到的循环不变量不加选择地进行外提。后端结合寄存器压力和指令延迟，对不变量进行选择性的外提。

为了能够解决传统 LICM 算法无法对一些包含大量循环不变量的循环结构进行优化，且会增加循环运行代价的问题，基于传统 LICM 算法，在 LLVM 中端和后端两个阶段进行改进，有选择地对循环不变量进行外提。总体算法框架示意图如图 4 所示。

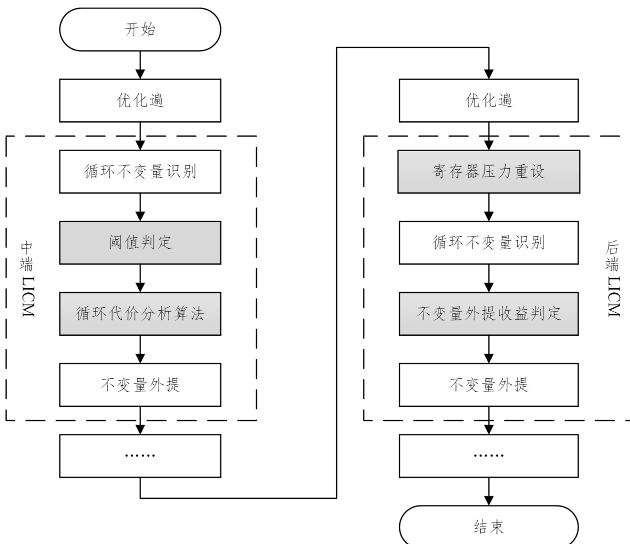

图 4　总体 LICM 算法框架示意图

Fig. 4　Overall framework diagram of LICM algorithm

从图 4 可以看出，在中端 LICM 中，对循环不变量进行识别后，引入了用于判定寄存器是否溢出的阈值；基于该阈值，设计了对循环不变量外提代价进行评估的循环代价分析算法，有选择地对循环不变量进行外提。在后端 LICM 中，在循环不变量识别前，对寄存器压力进行了重新设置；在识别后，利用后端对指令延迟的精确计算，对不变量外提收益进行更精确的判定。

4.2 中端 LICM 算法

在中端优化模块中，传统的 LICM 算法将识别到的循环不变量全部外提，并不考虑外提后产生的溢出代价可能会大于循环体内循环不变量本身的执行代价。为了解决该问题，本文在 LLVM 中端 LICM 优化模块中引入了循环代价分析算法，有选择地对中端识别到的循环不变量进行外提。整个 LLVM 中端 LICM 优化模块阶段的算法流程如图 5 所示。

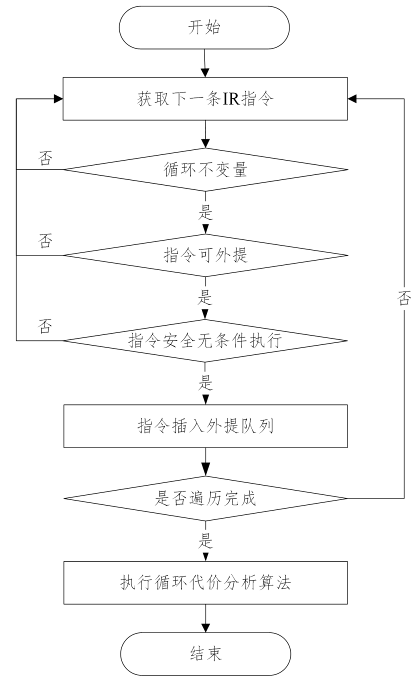

图 5　LLVM 中端优化阶段的 LICM 算法流程

Fig. 5　LICM algorithm flow in LLVM intermediate optimization stage

从图 5 可以看出，所提出的 LICM 算法在中端对 IR 的优化具体包含以下步骤。

步骤 1　识别循环中的不变量。遍历循环对应的 IR 指令，检查指令含有的操作数，如果该指令所有操作数均不是来自其他指令，则该指令是循环不变量。此外，如果指令的所有操作数的定义都不在本循环中，那么该指令是本循环的循环不变量。

步骤 2　对于步骤 1 得到的循环不变量指令，判断其是否可以被外提。对于算法中预设的指令，如访存、调用、比较和 GEP 等指令，编译器可以通过内存分析、操作数来源分析等手段判断该指令是否可以外提。对于不在预设范围内的指令，算法默认其不可以外提。

步骤 3　判断指令是否可以被安全、无条件地执行。如果指令可能会产生自陷，或者指令有副作用，则不能将其外提。

步骤 4　将经过步骤 3 得到的可外提的指令插入到等待

外提的优先队列。

步骤 5　执行循环代价分析算法，对具备外提条件的循环不变量 IR 指令进行评估，对符合评估条件的指令进行外提。

4.3　循环代价分析算法

为了评估当前循环中的循环不变量的运行代价与对其进行外提可能产生的溢出代价，并根据式(10)挑选其中最值得外提的指令进行不变量外提，设计了一种新的循环代价分析算法。在该算法中设计了两个阈值 T_{call} 和 T_{normal}，其分别表示循环中存在函数调用的情况与普通循环的情况下能够被寄存器保持的最大变量数。T_{call} 主要依据被调用者保护寄存器的数量进行设定，T_{normal} 主要依据可用的临时寄存器(包括被调用者保护寄存器)的数量进行设定，两者的值可以根据程序特征进行调整，设计相应的编译选项，允许用户手动设置经验值。

算法的具体步骤如下：

步骤 1　遍历循环。收集循环内的指令类型信息，如果循环内存在函数调用指令，只有部分寄存器被子函数保护，循环内能够用于保持变量的寄存器数较少，则设置判定阈值为 T_{call}，否则为 T_{normal}。

步骤 2　统计待外提队列中的指令数量，如果数量小于或等于阈值，则循环不变量指令外提不会产生其他影响，将待外提队列的指令全部进行外提操作。

步骤 3　对于外提指令数量大于阈值的情况，计算并维护待外提指令优先级队列。通过 LLVM 预留的接口，调用后端架构的目标平台转换信息计算该条指令的预计运行代价 C_{vi}，并根据代价数值将指令插入优先级队列，保证队列头的指令在循环体内的运行代价最大。

步骤 4　遍历优先级队列，依次从队列中取出等待外提的循环不变量指令，得到指令在循环体内的运行代价 C_{vi}，根据指令数据类型可以获得指令进行外提可能产生的溢出代价 C_{brei}。根据式(10)，当 $C_{vi} \leqslant C_{\text{brei}}$ 时，该循环不变量外提会使得优化后循环降低的代价小于零，产生负收益，故不进行外提操作。仅当 $C_{vi} > C_{\text{brei}}$ 时，进行外提操作。当取出的指令运行代价数值为 0 时，不再需要对其进行外提，停止从队列中获取指令。

4.4　后端 LICM 算法改进

在完成中端 LICM 优化后，在测试分析的过程中，发现以下缺陷需要后端协同解决。

1)在中端 LICM 算法设计中的阈值尽管可以使用编译选项调整，但有时并不能很好地反映不同架构寄存器数量的区别。

2)在中端进行预期代价的计算时，没有考虑因 CPU 型号支持的指令不同而实际执行代价不同的情况。

针对这两类缺陷，在 LLVM 后端对传统 LICM 优化遍进行了改进。首先，对函数调用阈值进行动态调整，针对循环内存在函数调用的情况，查看寄存器表，根据不同平台的保留寄存器重新计算 LICM 的阈值。其次，在指令调度阶段针对指令的延迟进行精确计算，当指令延迟大于典型的访存延迟时，该指令外提不会劣化循环性能，带来的潜在性能收益高于不外提，从而将该指令进行外提。通过这两方面的改进，来弥补所提出的算法在中端 LICM 模块中存在的缺陷。

5　实验与结果分析

5.1　实验环境

本文在开源编译器 LLVM 13.0.0 上实现了基于循环代价分析的 LICM 算法，并在国产申威 831 处理器平台上针对传统的 LICM 算法和本文提出的新 LICM(以下简称本文 LICM)算法进行了对比测评。实验软硬件环境参数如表 2 所列。

表 2　软硬件环境参数

Table 2　Software and hardware environment parameters

参数	参数值
指令集架构	SW64
CPU 主频	2.5GHz
内存大小	16GB
内核版本	4.19.0
操作系统	UOS Desktop 20 Pro

5.2　典型用例性能提升效果

针对 3.2 节设计的典型用例，本文 LICM 对循环体内 x，y，z 组成的表达式进行代价分析，有选择地进行外提，以有效规避潜在的寄存器溢出风险。在千万级的循环规模下，本文 LICM 较传统 LICM 有 17%以上的性能提升。

5.3　基准测试集

为了能够更全面地评估本文 LICM 算法的有效性，采用 CPU 标准性能评估测试集 SPEC CPU 2017、典型应用 Perl 解释器的基准测试集 DKbench 和典型应用 Python 解释器的基准测试集 pyperformance 这 3 个公开基准测试集来评测算法的综合表现。其中，Perl 和 Python 解释器的核心代码均主要由循环结构组成，涉及大量循环不变量的处理，适用于本文算法的效果验证。

5.3.1　SPEC CPU 2017

SPEC CPU 2017 包含 SPECspeed 2017 Interger，SPECspeed 2017 Floating Point，SPECrate 2017 Interger，SPECrate 2017 Floating Point 共 4 个套件，分别用于测评处理器整数单任务、浮点单任务、整数多任务、浮点多任务的处理能力[25]，本文采用 SPECspeed 2017 Interger 套件(包含 10 个测试程序)，使用 SPEC CPU 2017 规定的几何平均计算综合结果。

5.3.2　DKbench

本文基于 Perl 5.28.1 对 DKbench 2.4 进行实验。DKbench 2.4[1] 包含 Moose(Perl 对象框架)、DBI、SQL(Perl 数据库接口)、Regex(Perl 正则表达式)等 18 个 Perl 典型应用，支持 Single 和 Multi 共 2 个测试套件，分别用于测评处理器单任务和多任务的处理能力。本文采用 Single 套件，使用 DKbench 规定的算术平均计算综合结果。

5.3.3　pyperformance

本文基于 Python 3.9.9 对 pyperformance 1.10.0 进行实验。pyperformance 1.10.0[2] 包含 91 个 Python 典型用例，根据测试重点的不同分为高层应用、异步 IO、数学运算、正则表

[1] https://github.com/dkechag/Benchmark-DKbench

[2] https://pyperformance.readthedocs.io

达式、序列化、启动时间、模板库共 7 个子类。本文采用 pyperformance 默认的测试和结果对比方法。

5.4 基准测试集实验数据与分析

在国产申威 831 处理器平台上,分别在 SPEC CPU 2017,DKbench 和 pyperformance 相应的套件上进行测试,均采用-O2 和-static 编译选项。

5.4.1 SPEC CPU 2017 测试结果

在测评 SPEC CPU 2017 时,LICM 算法阈值设置为 $T_{call}=6$,$T_{normal}=24$ 的效果较好,测评结果如表 3 所列。从表中可以看出,本文 LICM 算法相比传统 LICM 算法在 600. perlbench_s,625. x264_s,648. exchange2_s 等程序上具有一定幅度的性能提升,在几何平均上具有 0.4%的性能优势,且针对单个程序无负效果。

表 3 SPEC CPU 2017 测试结果

Table 3 SPEC CPU 2017 test results

(%)

程序	传统 LICM 的性能提升效果	本文 LICM 的性能提升效果
600. perlbench_s	−2.68	0
602. gcc_s	1.27	0
605. mcf_s	0	0
620. omnetpp_s	0	0
623. xalancbmk_s	3.12	3.01
625. x264_s	0.58	1.16
631. deepsjeng_s	0	0
641. leela_s	3.88	3.88
648. exchange2_s	4.09	5.85
657. xz_s	−0.23	0.11
几何平均	0.98	1.38

5.4.2 DKbench 测试结果

在测评 DKbench 时,LICM 算法阈值设置为 $T_{call}=6$,$T_{normal}=24$ 的效果较好,测评结果如表 4 所列。从表中可以看出,本文 LICM 算法相比传统 LICM 算法在 Math∷DCT,Regex/Subst,Regex/Subst utf8 等程序上具有一定幅度的性能提升,在算术平均上以 0.63%的性能优势规避了传统 LICM 引入的负效果。

表 4 DKbench 测试结果

Table 4 DKbench test results

(%)

程序	传统 LICM 的性能提升效果	本文 LICM 的性能提升效果
Astro	−0.72	−0.72
CSS∷Inliner	0	−1.77
Crypt∷JWT	0	0
DBI/SQL	0	0
DateTime	0.77	0
Digest	0	0.33
Encode	1.61	1.20
HTML∷FormatText	0.92	1.83
Imager	0	0
JSON∷XS	0.52	1.56
Math∷DCT	−1.95	0.65
Math∷MatrixReal	1.56	0.78
Moose	0.96	0.96
Moose prove	−1.87	−0.93
Primes	1.32	−1.97
Regex/Subst	−4.82	1.81
Regex/Subst utf8	−4.78	−1.37
Text∷Levenshtein	−0.29	−0.29
算术平均	−0.57	0.06

5.4.3 pyperformance 测试结果

在测评 pyperformance 时,LICM 算法阈值设置为 $T_{call}=0$,$T_{normal}=24$ 的效果较好,测评结果如表 5 所列。从表中可以看出,本文 LICM 算法相比传统 LICM 算法,在 7 个测试子类上普遍存在性能提升,在整体上具有 1%的性能优势。

表 5 pyperformance 测试结果

Table 5 pyperformance test results

程序	本文 LICM 较传统 LICM 的性能提升
apps	1.01× faster
asyncio	1.02× faster
math	1.00× faster
rege×	1.01× faster
serialize	1.01× faster
startup	1.02× faster
template	1.01× faster
all	1.01× faster

5.4.4 实验结论

综上,相比传统 LICM 算法,本文 LICM 算法在 CPU 标准基准测试集和典型应用上具有较好的综合性能表现,实验结果大多为正向提升,具有较好的普适性;部分程序存在性能下降的问题,如 SPEC CPU 2017 的 602. gcc_s 程序、DKbench 的 CSS∷Inliner 和 Primes 程序,原因为设定的 T_{call} 和 T_{normal} 阈值不适用于这些程序的行为特征。

结束语 针对传统的 LLVM LICM 算法无法对部分含有大量循环不变量的循环结构进行有效优化的问题,本文提出了一种新的基于循环代价分析的 LICM 算法。通过对循环中的不变量进行代价评估,有选择地进行循环不变量外提操作,避免出现寄存器溢出、循环体内因为大量不变量外提而带来的额外访存开销等问题,有效地对循环进行了优化。通过在国产申威 831 处理器平台对典型样例和基准测试集 SPEC CPU2017,Perl DKbench,Python pyperformance 进行性能测试,验证了本文提出的基于循环代价分析的 LICM 算法相比传统的 LICM 算法的优化效果更好。本文中端 LICM 算法中用于判定寄存器是否溢出的阈值主要依赖于针对应用特征和架构特点进行的手动设定,无法针对复杂代码场景进行自适应;后端 LICM 算法的改进空间有限,尤其是无法还原中端已外提的循环不变量。下一步将深入研究该阈值的自动计算技术,以进一步提高算法优化效果。

参 考 文 献

[1] LATTNER C, ADVE V. LLVM: A compilation framework for lifelong program analysis & transformation[C]// International Symposium on Code Generation and Optimization. IEEE, 2004: 75-86.

[2] CHO J, CHO D, KIM Y. Study on LLVM application in Parallel Computing System[J]. The Journal of the Convergence on Culture Technology, 2019, 5(1): 395-399.

[3] WANG L, GAO K, ZHAO Y Q, et al. Deep Learning Compiler Load Balancing Optimization Method for Model Training[J]. Journal of Frontiers of Computer Science and Technology, 2024, 18(1): 111-126.

[4] ZHU X L. The implementation and optimization of deep learning

compiler based on GPDSP[D]. Changsha: National University of Defense Technology, 2021.

[5] XIE H C. An Auto Performance Predict Research for Scientific Program Based on LLVM[D]. Harbin: Harbin Institute of Technology, 2016.

[6] PEELER H, LI S S, SLOSS A N, et al. Optimizing LLVM pass sequences with Shackleton: a linear genetic programming framework[C]// Proceedings of the Genetic and Evolutionary Computation Conference Companion. 2022: 578-581.

[7] CHAI Y, CHEN M, LI J, et al. Implementation and optimization of data prefetching algorithm based on LLVM compilation system[C]// 6th International Conference on Electronic Technology and Information Science. 2021.

[8] HUANG Z F, SHANG J D. Optimization based on LLVM global instruction selection[C]// 2021 International Conference on Computer Network Security and Software Engineering. 2021. 2021.

[9] AHO A V, SETHI R, ULLMAN J D. Compilers: principles, techniques, and tools[M]. Reading: Addison-Wesley, 1986.

[10] ZAKOWSKI Y, BECK C, YOON I, et al. Modular, compositional, and executable formal semantics for LLVM IR[J]. Proceedings of the ACM on Programming Languages, 2021, 5 (ICFP): 1-30.

[11] ANDREW W A. SSA is Functional Programming[J]. Acm Sigplan Notices, 1998, 33(4): 17-20.

[12] YAN Y, PAN Z, YU L, et al. Research on the influencing factors of LLVM IR optimization effect[C]// 2023 IEEE 3rd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA). IEEE, 2023, 3: 756-763.

[13] SHANG J, XU K, HAN L, et al. Optimization of Access Address Calculation for LLVM[C]// 2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS). IEEE, 2023: 458-464.

[14] RODRIGUEZ-CANCIO M, COMBEMALE B, BAUDRY B. Automatic microbenchmark generation to prevent dead code elimination and constant folding[C]// Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016: 132-143.

[15] GUO Z H, WU Y X, AN L F, et al. Study on function inlining optimization technologies based on LLVM[J]. Computer Engineering and Applications, 2017, 53(3): 41-46.

[16] WANG C X, HAN L, LIU H H. Optimization of loop unrolling based on instruction Cache and register pressure[J]. Computer Engineering & Science, 2022, 44(12): 2111-2119.

[17] HU Y X, ZHEGN Q L. Research on Deep Learning Based Loop Auto-schedule[J]. Journal of Chinese Computer Systems, 2024, 45(7): 1770-1777.

[18] MELLOR-CRUMMEY J, ADVE V. Simplifying control flow in compiler-generated parallel code[J]. International Journal of Parallel Programming, 1998, 26(5): 613-638.

[19] LI H X, LIU J. Methods of Data Flow Analysis [J]. Computer Engineering and Applications, 2003, 39(13): 142-144.

[20] GONG L Q, SHEN L, ZHOU Q L, et al. Research and Implementation of Compile Lock Mechanism Based on LLVM[J]. Computer Application and Software, 2021, 38(11): 11-17.

[21] EBNER D, BRANDNER F, SCHOLZ B, et al. Generalized instruction selection using SSA-graphs[C]// Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. 2008: 31-40.

[22] LOZANO R C, CARLSSON M, DREJHAMMAR F, et al. Constraint-based register allocation and instruction scheduling[C]// International Conference on Principles and Practice of Constraint Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: 750-766.

[23] DE SOUZA XAVIER T C, OLIVEIRA G S, DE LIMA E D, et al. A detailed analysis of the llvm's register allocators[C]// 2012 31st International Conference of the Chilean Computer Science Society. IEEE, 2012: 190-198.

[24] LIANG J L, HUA B J, LV Y S, et al. Loop Invariant Code Motion Algorithm for Deep Learning Operators[J]. Journal of Frontiers of Computer Science and Technology, 2023, 17(1): 127-139.

[25] SHI H K, WANG Z S, ZHANG S Z, et al. Performance Evaluation Benchmark of General-Purpose CPU: A Survey[J]. Acta Electronica Sinica, 2023, 51(1): 246-256.

JIANG Jun, born in 1980, master, senior engineer. His main research interests include compiler optimization, architecture-oriented performance analysis and optimization, etc.

HUANG Liangming, born in 1988, Ph.D, engineer. His main research interests include compiler optimization, architecture-oriented performance analysis and optimization, etc.

（责任编辑：何杨）