9 过程内优化

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 掌握常量分析优化方法
- 掌握冗余代码优化方法
- 掌握循环优化方法

9.1 概述

代码优化是一个复杂的过程，由面向特定优化模式的若干个编译流程（pass）组成。这些流程有的工作在 IR 层面，有的则是工作在汇编代码层面。其中工作在 IR 层面的优化方法是与具体指令集无关的通用优化方法，具有更好的普适性。LLVM 编译器中提供了很多 IR 层面的优化流程 [1]。本章内容探讨其中常见的一些针对单个函数的代码优化模式，暂不考虑跨函数的情况。

9.2 基于常量分析的优化

9.2.1 常量分析

常量分析的目的是找出在某一程序节点，某一变量或寄存器是否为固定不变的特定值。该分析任务可以通过上一章学习的循环迭代分析方法完成，即前向遍历控制流图中的每条语句，并维护已识别的常量标识符信息；如果遇到合并节点取交集即可。

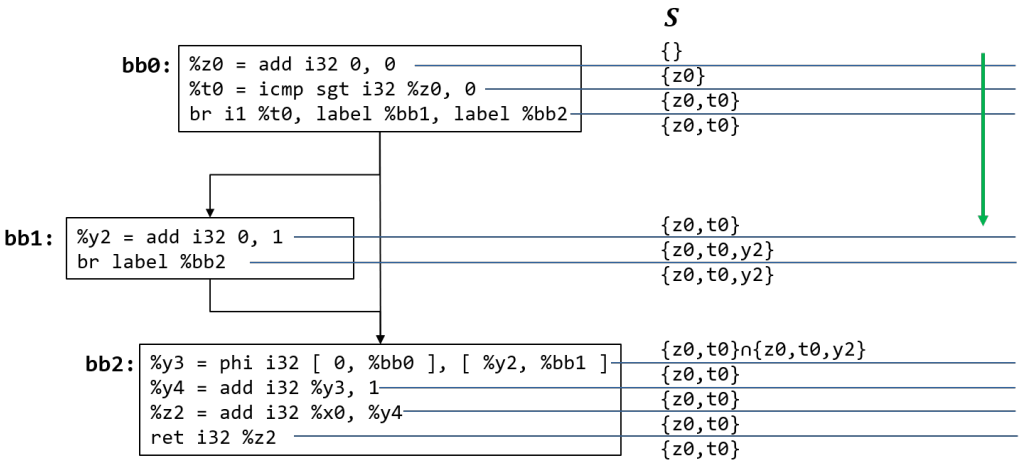


图 9.1: 面向 LLVM IR 的常量分析

以图 9.1中 SSA 形式的 LLVM IR 为例，我们可以采用表 9.1中定义的 Transfer 函数分析每一个程序节点的常量标识符集合。SSA 形式的常量分析比较简单，一个虚拟寄存器一旦被识别为常量，则不会发生变化成为变量。非 SSA 形式的代码还需要考虑该虚拟寄存器或变量是否会被重新赋值的情况。

表 9.1: Transfer 函数定义：常量分析

IR 指令	举例	Transfer 函数
add/sub/mul/sdiv	`%r = add i32, op1, op2`	$S = S \cup \{r\},\ s.t.\ op1 \in S \cup Num$ and $op2 \in S \cup Num$
xor/and/or	`%r = xor i32, op1, op2`	$S = S \cup \{r\},\ s.t.\ op1 \in S \cup Num$ and $op2 \in S \cup Num$
icmp	`%r = icmp sgt i32, op1, op2`	$S = S \cup \{r\},\ s.t.\ op1 \in S \cup Num$ and $op2 \in S \cup Num$
zext/trunc	`%r = zext i8 op1 to i32`	$S = S \cup \{r\},\ s.t.\ op1 \in S \cup Num$

9.2.2 常量分析应用

常量分析的直接应用是使用具体数值替换虚拟寄存器（常量传播），从而在编译时完成一些计算（常量折叠），避免运行时开销。常量传播算法可在常量分析算法的基础上略微改进实现，即在维护常量标识符集合的同时记录每个标识符对应的具体数值或计算方式。

当二元运算中仅有一个操作数为常量时，虽然不能进行常量折叠，但可以考虑是否存在指令合并的可能性。常见的指令合并情况是：指令 I_1 的一个操作数为常量，另一个为变量；指令 I_2 的一个操作数为常量，另一个为指令 I_1 的运算结果。此时，可以对指令 I_2 的操作数进行优化。代码 9.1展示了一个指令合并案例。

```
%x1 = add i32 %x0, 1 ; 如果%x1没有被使用，则可以删除该指令
%x2 = add i32 %x1, 2 ; 优化结果：%x2 = add i32 %x0, 3
```

代码 9.1: 指令合并示例

9.3 冗余代码优化

9.3.1 无效代码优化

程序 IR 中可能包含一些指令或虚拟寄存器，其运行结果不会被后续指令使用，则这些指令都是冗余的，带来无谓的运行时开销，应当将其删除。以代码 9.1为例，对其进行指令合并优化后，计算`%x1` 的指令很可能成为多余的。这类无效代码优化可以通过活跃性分析实现，即后向分析控制流图，如果遇到 IR 指令将某虚拟寄存器作为其操作数，则将该虚拟寄存器标记为活跃，直至其被定义为止。如果一个虚拟寄存器在不活跃状态下被定义，则定义该寄存器的 IR 指令很可能是冗余的。例外情况是该虚拟寄存器作为函数调用返回值时，由于函数调用会有副作用，不能将其删除。

9.3.2 死代码优化

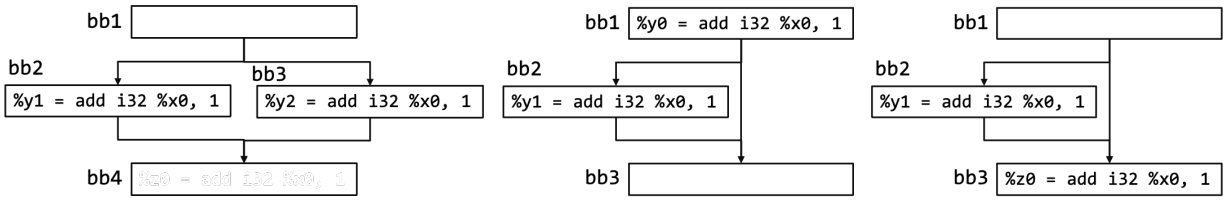
比较典型的死代码是不可达代码块，即条件恒为真或假的条件跳转语句。以图 9.1为例，对其进行常量传播优化后，发现 bb1 是不可达的，应当删除。

9.3.3 全局值编号

如果 IR 中操作数数值相同的同名指令出现多次，只需保留一个副本或计算一次即可。分析同名指令操作数数值是否相同的过程称为全局值编号（GVN：global value numbering）。GVN 的分析一般在常量传播优化之后进行，其主要思路是为操作数数值相同的同名指令维护一个集合。由于 SSA 形式的 IR 无须考虑操作数是否被重新赋值的情况，只需通过操作数标识符是否相同便可识别出很大一部分操作数数值相同的指令，大大简化了 GVN 的分析和优化过程。如果两条同名指令的某个操作数标识符不相同，但在一个集合中，也应将其认定为同一编号，即表格 9.2定义了 GVN 分析对应的 Transfer 函数。因此，我们可以沿用循环迭代的分析方法直至 GVN 集合不再更新为止。

表 9.2: Transfer 函数定义：全局值编号

IR 指令	举例	Transfer 函数
add/sub/mul/sdiv	`%r = add i32, op1, op2`	$S = S \cup r,\ s.t.\ S = Find(ADD(S_{op1}, S_{op2}))$
xor/and/or	`%r = xor i32, op1, op2`	$S = S \cup r,\ s.t.\ S = Find(XOR(S_{op1}, S_{op2}))$
icmp	`%r = icmp sgt i32, op1, op2`	$S = S \cup r,\ s.t.\ S = Find(ICMPsgt(S_{op1}, S_{op2}))$
zext/trunc	`%r = zext i8 op1 to i32`	$S = S \cup r,\ s.t.\ S = Find(ZEXTi32(S_{op1}))$



(a) 繁忙表达式　(b) 公共表达式　(c) 弱公共表达式

图 9.2: 基于 GVN 的优化

根据重复计算指令出现位置的不同，GVN 具体可以优化的情况又分为繁忙表达式、公共表达式、和弱公共表达式。

- 繁忙表达式：操作数相同的指令出现在不同的分支。例如图 9.2a中的 bb2 和 bb3 都包含 `add i32 %x0, 1`，可将其提前到条件跳转指令之前，从而优化代码体积。
- 公共表达式：操作数相同的指令具有支配关系。例如图 9.2b中的 bb1 和 bb2 都包含 `add i32 %x0, 1`，且 bb1 支配 bb2。在这种情况下，bb2 中的对`%y1` 的求值运算是多余的，可以将其删除，使用`%y0` 代替`%y1`。
- 弱公共表达式：两条操作数相同的指令不存在支配关系也存在可以优化的情况。例如图 9.2c中 bb2 和 bb3 中都包含 `add i32 %x0, 1`，将该表达式提前至 bb1 则可以避免走左侧分支时的重复计算。

9.4 循环优化

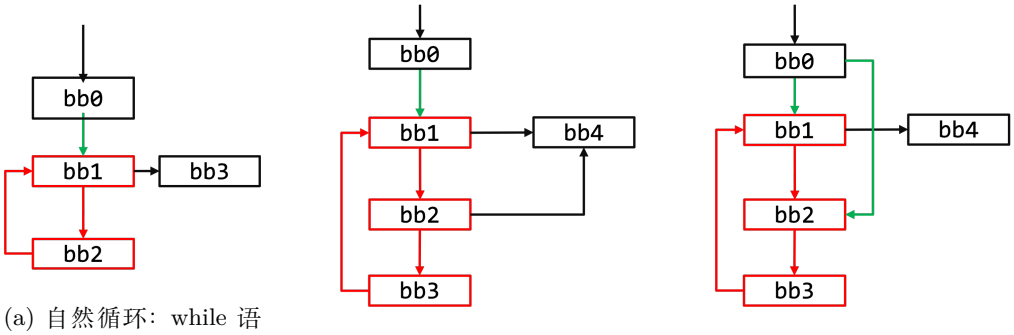
循环优化是对于提升代码运行效率效果最为显著的优化手段之一，其核心思想是将循环内重复执行的代码提前至循环外的支配节点，避免代码被重复执行。下面先介绍面向代码控制流图中循环路经的检测方法，再介绍具体的循环优化技巧。

9.4.1 循环检测算法

由于 TeaPL 中只有 `if-else` 和 `while` 语句会引入控制流，使得其对应 IR 中的每个循环都只会有一个入口节点，该节点支配循环中的所有其它节点，这种循环称为自然循环。因此，TeaPL 对应的控制流图中的循环都是自然循环，这种都是自然循环的图是可规约图。

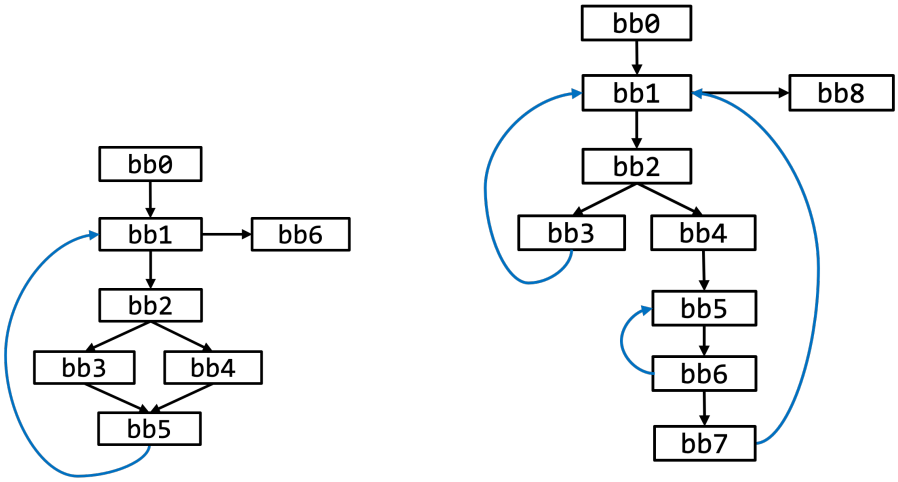
图 9.3展示了几个控制流图的例子。其中，图 9.3a是 `while` 循环的控制流图，很明显 bbb1->bb2->bb1 是一个自然循环；图 9.3b中的 bbb1->bb2->bb3->bb1 也是自然循环，通过在循环体内部加入一个条件跳转分支和 `break` 语句可以达到此效果；图 9.3c中的循环存在两个入口，因此是非自然循环。

自然循环中的入口节点是唯一的，因此可以采用入口节点标识一个自然循环；但不同的循环可能出现入口节点相同的情况，所以我们采用更细粒度的返回边唯一标识一个循环。以图 9.4a为例，其中包含一个自然循环：bb5->bb1。图 9.4b则包含三个自然循环：bb3->bb1、bb6->bb5、和 bb7->bb1。其中，循环 bb7->bb1 包含循环 bb6->bb5 的所有节点，这两个循环为嵌套关系；循环 bb7->bb1 和循环 bb3->bb1



(a) 自然循环：while 语句　(b) 自然循环：while+break　(c) 非自然循环

图 9.3: 自然循环



(a) 控制流图示例：1 个循环　(b) 控制流图示例：3 个循环

图 9.4: 通过返回边标识自然循环

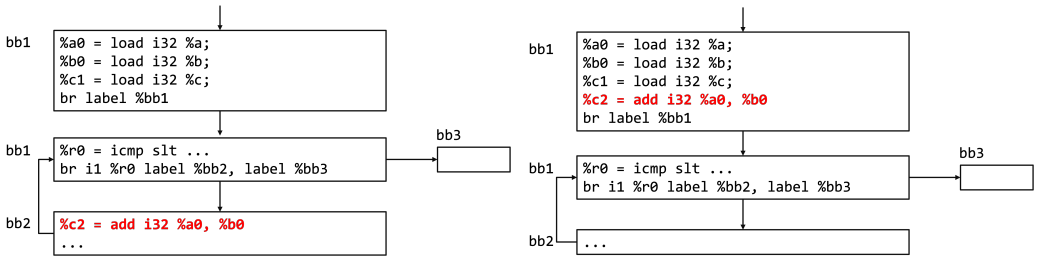
有两个公共节点 bb1 和 bb2，以及一条公共边 bb1->bb2，这两个循环为相切的关系。存在嵌套或相切关系的两个循环不需要经过其它节点可以组成一个大的强联通分量。自然循环之间不会出现相交，即存在多个入口节点的情况。

基于上述分析，我们可以设计出算法 1，用于检测 IR 中的自然循环。

算法 1 自然循环搜索算法

```
1: s ← ∅; // 栈，用于记录访问过的节点
2: Loop ← ∅; // 记录识别出的循环，使用返回边作为唯一标识
3: procedure FindLoops(v) // 从控制流图入口开始搜索其中的循环
4:     s.push(v);
5:     for each w in v.next() do
6:         if s.contains(w) then // 已经访问过该节点，说明找到循环
7:             AddLoop(w, v);
8:         else
9:             FindLoops(w); // 深度优先递归搜索
10:        end if
11:    end for
12:    s.pop(v);
13: end procedure
14: procedure AddLoop({v, w}) // 将识别到的循环添加到结果中
15:    if !Loop.exists(v, w) then
16:        l ← CreateLoop(top n items of s until w);
17:        Loop.add((v, w), l);
18:    else// 循环已经出现过：以图 9.4a为例，由于循环内部的条件分支导致其再次被检测到
19:        l ← CreateLoop(top n items of s until w);
20:        Loop.merge((v, w), l);
21:    end if
22: end procedure
```

9.4.2 循环优化应用

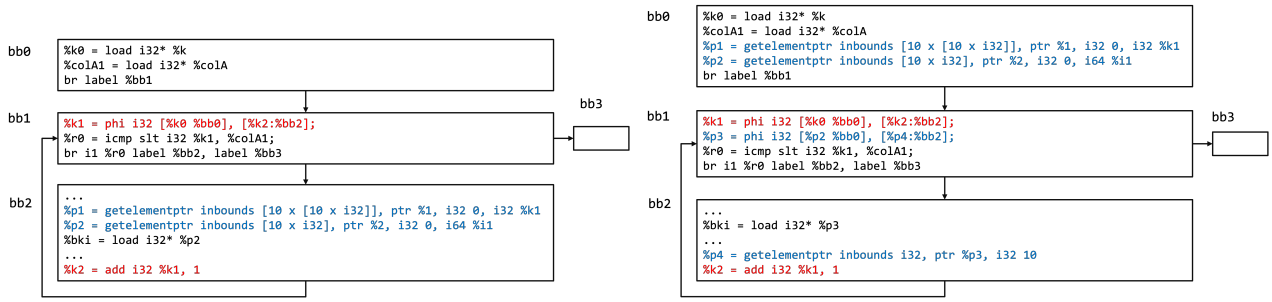


(a) 原始 IR　　(b) 优化后的 IR

图 9.5: 循环不变代码优化

```
while (i<rowA) {
    while (j<colB) {
        while (k<colA) {
            R[i][j] = R[i][j] + A[i][k]*B[k][j];
            //优化：改为t = t + A[i][k]*B[k][j];
            k = k+1;
        }
        j = j+1;
    }
    i = i+1;
}
```

代码 9.2: 标量替换优化示例：TeaPL 实现矩阵乘法代码片段



(a) 原始 IR　　(b) 优化后的 IR

图 9.6: 归纳变量代码优化

本节介绍三种典型的循环优化应用：

- 循环不变代码：如果一条指令在循环体内被多次执行，但其操作数未发生变化，则应将这条指令前移至循环外部，避免重复执行。图 9.5展示了一个示例，bb3 中指令`%c2 = add i32 %a0, %b0` 的操作数`%a0` 和`%b0` 均定义自循环外部，因此可将这条指令前移至 bb0，从而避免重复计算。

- 标量替换：如果由于循环导致需要多次仿存同一内存地址上的标量数据，应当使用寄存器替换该仿存操作。代码 9.2展示了一个典型的矩阵乘法案例，将其中的 R[i][j] 替换为临时变量 t 则可以避免在循环体内对 R[i][j] 的重复仿存操作。需要注意的是，该替换是有风险的。例如，当数组 R 和数组 A 或 B 存在别名关系时，其部分元素会共用同一块内存地址，更新 R[i][j] 的同时也更新了 A 或 B 的某个元素值，使用标量替换后则无法保持一致的计算结果。

- 归纳变量优化：这种优化一般与循环的条件变量相关。以代码 9.2中最内层的循环为例，我们可以将其控制流图表示为图 9.6a的形式。其中，`%k1` 是循环的条件变量，每一轮循环增加 1，直至等于`%colA1` 时退出循环。我们可以将 bb2 中的以`%k1` 作为操作数的相关指令进行优化，如将数组 `B[k+1][j]` 的寻址方式转化 `&B[k][j]+%colB1` 或 `&B[k][j]+10`（数组的大小为 10x10）的形式，从而优化寻址过程。由于 LLVM IR 的寻址指令以类型而非字节为基本单位，这种归纳变量优化方法在翻译汇编代码后可以再次发挥作用。

练习

1. 代码 9.3是否可以被优化？如何在编译器中实现相应的优化流程？

```
define void @collatz(i32 %0) {
bb1:
  br label %bb2
bb2:
  %t0 = phi i32 [ %0, %1 ], [ %t1, %12 ]
  %b0 = icmp ne i32 %t0, 1
  br i1 %b0, label %bb3, label %bb7
bb3:
  %t2 = srem i32 %t0, 2 ; srem指令：取余数
  %b1 = icmp eq i32 %t2, 0
  br i1 %b1, label %bb4, label %bb5
bb4:
  %t3 = sdiv i32 %t0, 2
  br label %bb6
bb5:
```

```
  %t4 = mul i32 3, %t0
  %t5 = add i32 %t4, 1
  br label %bb6
bb6:
  %t1 = phi i32 [ %t3, %bb4 ], [ %t5, %bb5 ]
  br label %bb2
bb7:
  ret void
}
```

代码 9.3: IR 代码：Collatz 函数

Bibliography

[1] LLVM's Analysis and Transform Passes, https://llvm.org/docs/Passes.html