

# 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

过程内优化指的是以函数为对象进行的优化, 与之相对的是过程间优化。

## 9.1 基于常量分析的优化

### 9.1.1 常量分析

常量分析指的是分析在某一程序点, 某一变量或寄存器的值是否为固定不变的数值。该分析可以通过前向数据流分析完成, 如果遇到合并节点取交集即可。

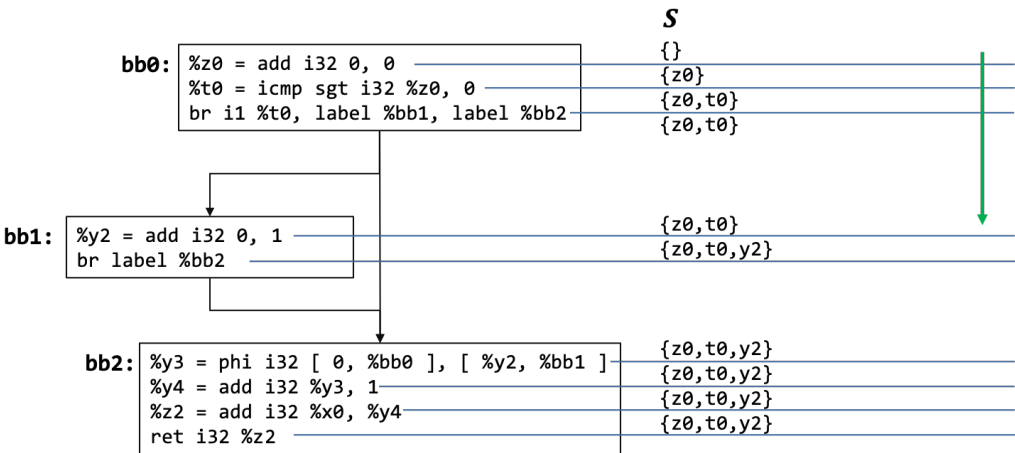


图 9.1: 面向 LLVM IR 的常量分析

以图 ?? 中 SSA 形式的 LLVM IR 为例, 我们可以采用表 ?? 中的定义的 Transfer 函数分析每一个程序节点的常量集合。SSA 形式的常量分析比较简单, 一个临时寄存器 %t0 一旦被识别为常量, 则不会变为变量; 而非 SSA 形式的代码还需要考虑该寄存器或变量是否会被重新赋值的情况。

表 9.1: 常量分析 Transfer 函数定义

IR	举例	Transfer 函数
binOp 指令	%t2 = add i32, %t0, %t1	$S = S \cup \{t2\}, \text{if } t0 \in S \text{ and } t1 \in S$

## 9.1.2 优化应用

基于常量分析的结果使用具体数值替换临时寄存器，从而在编译时完成一些计算，避免运行时开销，该过程称为常量传播。对于二元运算中仅有一个运算数为常量的情况，可以考虑是否存在指令合并的可能性。指令合并最简单的情况是：指令 1 的一个运算数为常量，另一个为变量；指令 2 的一个运算数为常量，另一个为指令 1 的运算结果，则可以对指令 2 的运算数进行优化。代码 ??展示了这种情况和优化方式。

```
%y0 = add i32 %x0, 1
%y1 = add i32 %y0, 2 // 优化: %y1 = add i32 %x0, 3
```

代码 9.1: 指令合并

## 9.2 冗余代码优化

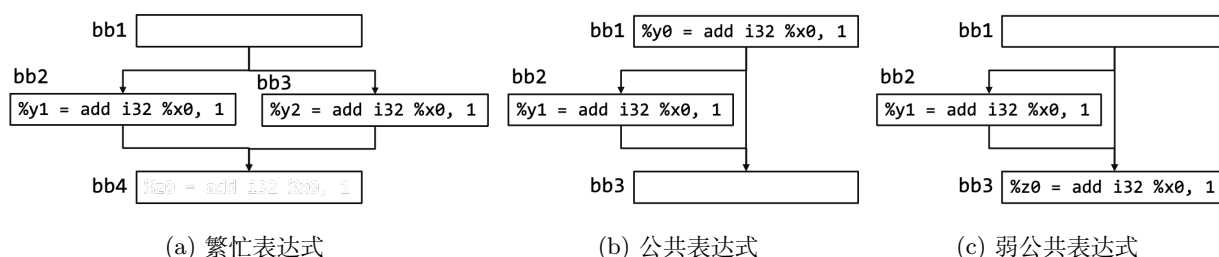


图 9.2: 基于 GVN 的优化

冗余代码一般分为以下几类：

- 死代码：即不会被执行的代码，如不可达的冗余代码块。以图 ??为例，对其进行常量优化以后，发现 bb1 是不可达的，应当删除。
- 重复出现的代码：同一段代码在函数中出现多次，增大了代码体积。如对于相同的代码块可以合并，通过修改控制流实现等价的代码语义；另外一种情况是繁忙表达式，即同一表达式在不同的分支出现，并且其操作数值相同。图 ??展示了一个繁忙表达式的例子。
- 重复计算的代码：典型的情况是在操作数值未发生变化的情况下，对同一表达式其进行多次求值。以图 ??为例，%y0 和 %y1 的计算方式相同，由于 bb1 支配 bb2，因此 %y1 的求值运算是多余的，可以直接采用 %y0 的结果，这种情况称为公共表达式。如果这两个表达式不存在支配关系也存在优化的可能性，即图 ??所示的弱公共表达式情况，可将表达式提前 (hoist) 至 bb1。

繁忙表达式和公共表达式本质上是对代码中的表达式进行 GVN (global value numbering) 分析，从而优化表达式的位置和计算次数。

## 9.3 循环优化

循环优化是对于提升代码运行效率效果最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。下面先介绍代码控制流图中的循环检测方法，再介绍具体的循环优化技巧。

### 9.3.1 循环检测算法

由于 TeaPL 中只有 `if-else`、`while` 和逻辑运算语句会引入控制流，使得其对应 IR 中的每个循环都只会一个入口节点，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，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。

#### 算法 1 自然循环搜索算法

```

1:  $s \leftarrow \emptyset$ ; // s is an empty stack
2:  $Loop \leftarrow \emptyset$ ; // an empty set of loops
3: procedure FINDLOOPS( $v$ )
4:    $s.push(v)$ ;
5:   for each  $w$  in  $v.next()$  do
6:     if  $s.contains(w)$  then
7:       AddLoop( $w, v$ );
8:     else
9:       FindLoops( $w$ );
10:    end if
11:  end for
12:   $s.pop(v)$ ;
13: end procedure
14: procedure ADDLOOP( $\{v, w\}$ )
15:  if  $!Loop.exists(v, w)$  then
16:     $l \leftarrow \text{CreateLoop}(\text{top } n \text{ items of } s \text{ untill } w)$ ;
17:     $Loop.add(l, v, w)$ ;
18:  else
19:     $l \leftarrow \text{CreateLoop}(\text{top } n \text{ items of } s \text{ untill } w)$ ;
20:     $Loop.merge(l, v, w)$ ;
21:  end if
22: end procedure

```

### 9.3.2 优化应用

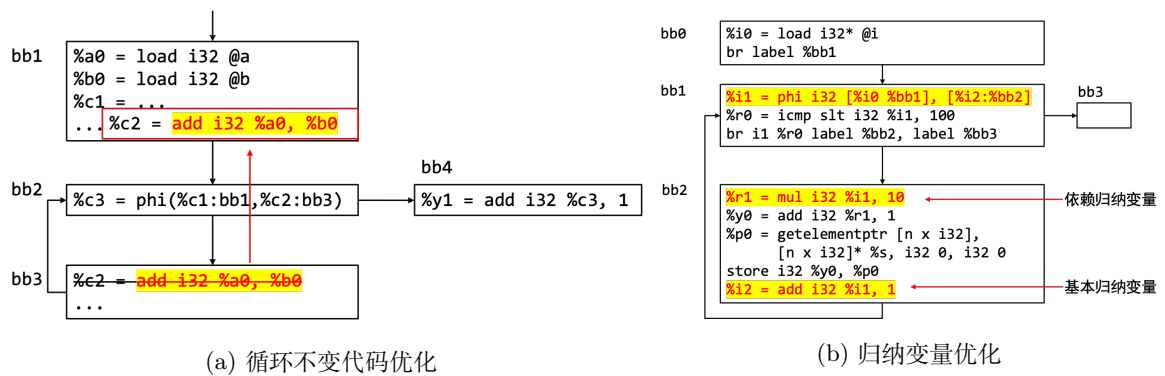


图 9.3: 循环优化

本节介绍三种典型的循环优化应用：

- 循环不变代码：与公共表达式类似，该方法检测在循环体内被多次重复执行的表达式（运算数未发生变化）。图 ??，展示了一个示例，将 bb3 中的语句 `%c2 = add i32 %a0, %b0` 前移至 bb0 可以避免重复计算。

- 归纳变量优化：这种优化一般与循环的条件变量相关。图 ??展示了一个例子，其中%i1 是循环体的条件变量，bb2 中%r1 = mul i32 %i1, 10 和该变量有关，并且%i1 每轮循环增加 1。因此可以将 bb2 中%r1 和%y0 的计算合并为%y0 = add i32 %t1, 10 的形式。这种思想在对循环内的数组寻址（如 a[i] 的地址是 a + i \* sizeof(a[0])）优化时使用机会比较多。
- 标量替换：如果由于循环导致需要多次存取某个内存地址上的标量数据，应当使用寄存器替换该内存存取操作。代码 ??展示了一个典型矩阵乘法标量替换优化案例。

```
for i in 0..rowA {
  for j in 0..colB {
    ; // 优化：增加 t = R[i][j]
    for k in 0..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];
    }
    ; // 优化：增加 R[i][j] = t
  }
}
```

代码 9.2: 标量替换示例