编译原理实验报告
PA5

姓名：谢兴宇
学号：2017011326

2020 年 1 月

在 PA4 的基础上，我继续在完整框架上开发 PA5。

1 算法基本流程

1.1 建立干涉图

干涉图的节点包括函数所有的参数和函数体中使用的所有临时变量。
干涉图按下述规则建边（需要合并重边）：

- 函数的所有参数之间两两连边。
- 如果一条指令有赋值对象且其赋值对象是一个 TAC 临时变量，则将该赋值对象与该语句的 LiveOut 集合中所有的临时变量连边。
- 每条指令的 LiveIn 集合中所有的临时变量需要两两连边。

1.2 干涉图染色

我实现了文档中给出的启发式算法，取 $k = 19$ 着色：

1. 假设图 G 中某个结点 v 的度数小于 k，从 G 中删除 v 及其邻边得到图 G'，将 G 的 k-着色问题转化为先对 G' 进行 k-着色，然后给结点 v 分配一个其相邻结点在 G' 的 k-着色中没有使用过的颜色。
2. 重复 1 的过程从图中删除度数小于 k 的结点。如果可以到达一个空图，说明对原图可以成功实现 k-着色。

1.3 生成 MIPS 指令

遍历 CFG 中每一条指令，根据寄存器分配的结果来生成，注意在调用函数时需要存储和读取一些 caller-save 的寄存器即可。

2 问题回答

2.1 如何确定干涉图的节点？

函数所有的参数和函数体中使用的所有临时变量。

2.2 连边的条件是什么？

- 函数的所有参数之间两两连边。
- 如果一条指令有赋值对象且其赋值对象是一个 TAC 临时变量，则将该赋值对象与该语句的 LiveOut 集合中所有的临时变量连边。
- 每条指令的 LiveIn 集合中所有的临时变量需要两两连边。

3 算法比较

对于 decaf 代码

```
static int gcd(int a, int b) {
    while (a != 0) {
        int c;
        c = b % a;
        b = a;
        a = c;
    }
    return b;
}
```

之前的暴力算法的分配结果为：

```
_L_Main_gcd:  # function FUNCTION<Main.gcd>
    # start of prologue
    addiu   $sp, $sp, -36  # push stack frame
    sw      $ra, 32($sp)  # save the return address
    # end of prologue

    # start of body
    sw      $a0, 0($sp)  # save arg 0
    sw      $a1, 4($sp)  # save arg 1
_L2:
    li      $v1, 0
    lw      $t0, 0($sp)
    sne     $t1, $t0, $v1
    sw      $t0, 0($sp)
    beqz    $t1, _L1
    li      $v1, 0
    lw      $t0, 0($sp)
    seq     $t1, $t0, $v1
    sw      $t0, 0($sp)
    beqz    $t1, _L3
    la      $v1, _S2
    move    $a0, $v1
    li      $v0, 4
    syscall
    li      $v0, 10
    syscall
_L3:
    lw      $v1, 4($sp)
    lw      $t0, 0($sp)
    rem     $t1, $v1, $t0
    move    $v1, $t1
    move    $t1, $t0
    move    $t0, $v1
```

```
    sw      $t0, 0($sp)
    sw      $t1, 4($sp)
    j       _L2
_L1:
    lw      $v1, 4($sp)
    move    $v0, $v1
    j       _L_Main_gcd_exit
    # end of body

_L_Main_gcd_exit:
    # start of epilogue
    lw      $ra, 32($sp)  # restore the return address
    addiu   $sp, $sp, 36  # pop stack frame
    # end of epilogue

    jr      $ra  # return
```

我的算法的分配结果为:

```
_L_Main_gcd:  # function FUNCTION<Main.gcd>
    # start of prologue
    addiu   $sp, $sp, -36  # push stack frame
    sw      $ra, 32($sp)  # save the return address
    sw      $s4, 16($sp)  # save value of $s4
    sw      $s5, 20($sp)  # save value of $s5
    sw      $s7, 28($sp)  # save value of $s7
    # end of prologue

    # start of body
    sw      $a0, 0($sp)  # save arg 0
    sw      $a1, 4($sp)  # save arg 1
    lw      $s5, 0($sp)
    lw      $s4, 4($sp)
_L2:
```

```
    li      $s7, 0
    sne     $s7, $s5, $s7
    beqz    $s7, _L1
    li      $s7, 0
    seq     $s7, $s5, $s7
    beqz    $s7, _L3
    la      $s7, _S2
    move    $a0, $s7
    li      $v0, 4
    syscall
    li      $v0, 10
    syscall
_L3:
    rem     $s7, $s4, $s5
    move    $s7, $s7
    move    $s4, $s5
    move    $s5, $s7
    j       _L2
_L1:
    move    $v0, $s4
    j       _L_Main_gcd_exit
    # end of body
```

可见，在我实现的算法中，寄存器的分配被大大优化了，主体的计算部分几乎完全使用一个寄存器来完成。