寄存器分配

Welcome to the lower end

问题

- 寄存器是 CPU 能直接访问的资源，速度快于访问内存；
- 中间变量的数量是无限的（或者非常多的）；
- CPU 的寄存器数量是极其有限的；
- 同一时间只有一部分中间变量正在被使用。

我们希望让程序尽可能少的访问内存，使程序速度更快。

“**寄存器分配**”

寄存器分配

- 分配中间变量的存储位置（寄存器、内存、两者都有）
- 目标：尽可能少访问内存。
- 限制：寄存器数量有限（如 x86_64 有 16 个，RISC-V 有 32 个）
  不一定有足够寄存器分给所有中间变量。
- 可以在不同层级上做：基本块级别、函数级别、跨函数。
  其中函数级别的最为常用。
- 名词：Spill -- 寄存器里塞不下了，把对应的变量存到内存里

常用算法

- 不分配
- 贪心
- 线性扫描
- 图着色
- 上述算法的混合

不分配行不行?

在所有运算中都使用同一套输入和输出寄存器，每次运算之前从内存取出数据，运算完成之后立即存入对应的内存位置。

`c = a + b` =>

```
load r1, <memory of a>
load r2, <memory of b>
add r3, r1, r2          # r3 ← r1 + r2
store r3, <memory of c>
```

性能会差到爆 :)

一个朴素的贪心算法

- 遇到新变量
  - 把不再使用的变量从寄存器分配里面删掉
  - 有空闲寄存器 $\rightarrow$ 分配寄存器给它
  - 没有 $\rightarrow$ spill 掉一个现有的变量，换成它
- 需要读变量
  - 在寄存器里 $\rightarrow$ 直接读取寄存器
  - 不在寄存器里 $\rightarrow$ 给它分配一个寄存器，然后读寄存器

贪心算法的实现

后面还用不用这个变量？用 `fv(expr)` 计算后续表达式的自由变量就好。

```
fn alloc_var(live_regs, var, following: Expr) -> Allocation {
  let live = fv(following)                  // 计算活跃变量，也可以换成数据流分析
  for reg, var in live_regs {
    if !live.contains(var) { live_regs.remove(reg) } // 删掉已经死了的变量
  }
  for reg in all_regs {
    if !live_regs.contains(reg) { return Alloc(reg) }  // 分配空闲的寄存器
  }

  // 到这里已经没有空闲的寄存器了，spill 掉一个幸运变量（注意不要把马上要用到的变量删了）
  guard let Some((spilled, reg)) = live_regs.first()
    else { abort("No registers available?!") } // 如果不成功肯定是你写错了
  return Spill(spilled, reg)  // Spill 掉这个变量，把寄存器挪给我们自己用
}
```

贪心算法的实现

首先要给表达式分配寄存器，再分配表达式计算的结果的寄存器

```
fn alloc(live_regs, expr) -> Expr {
  match expr {
    Expr(mut expr) => while true {
      match alloc_expr(expr) {  // 实现就不写了，就是把变量替换成对应的寄存器
        Ok(expr) => return expr
        // 如果有变量被 spill 掉了没读回来，就加一个给他的 restore 然后再试一次
        Err(var) => expr = Let(var, Restore(var), expr)
      }
    }
    Let(...) => ...
  }
}
```

贪心算法的实现

```
fn alloc(live_regs, expr) -> Expr {
  match expr {
    Expr(...) => ...
    Let(var, expr, cont) => {
      let expr = alloc(live_regs, expr)  // 给表达式本身分配寄存器
      match alloc_var(var) {
        Alloc(reg) => {
          live_regs.add(reg, var)
          Let(reg, expr, alloc(live_regs, cont))
        }
        Spill(spilled, reg) => {
          live_regs.set(reg, var)
          Let(_, Spill(spilled, reg)    // 从寄存器把对应的变量存起来
            Let(reg, expr, alloc(live_regs, cont)))
        }
      }
    }
  }
}
```

实现的注意事项

- 没有结果的指令不需要分配寄存器（`let _: Unit = <expr>`），可以放个占位符
- Spill 的时候注意不要把马上要用的寄存器 spill 掉（可以用 FIFO 队列）
- 朴素的算法会有很多优化空间

函数调用约定（以 RISC-V 为例）

- 调用者保存
  - 传递参数 -- `a0--a7`
  - 传递返回值 -- `a0--a1`
  - 临时寄存器 -- `t0--t6`
- 被调用者保存 -- `s0--s11`
- 特殊寄存器
  - `ra` (return address) 是函数返回地址
  - `sp` (stack pointer) 指向栈顶，`fp` (`s0`; frame pointer) 指向栈帧底部

函数出入口示例

```
foo:
  sd ra, -8(sp)      # 存储返回地址
  sd s0, -16(sp)     # 存储栈帧指针
  sd s2, -24(sp)     # 存储被调用者保存的寄存器
  mv s0, sp          # 更新栈帧指针
  addi sp, sp, -48   # 更新栈顶指针
# ... 函数体
  mv sp, s0          # 恢复栈顶指针
  ld s2, -24(sp)     # 恢复被调用者保存的寄存器
  ld ra, -8(sp)      # 恢复返回地址
  ld s0, -16(sp)     # 恢复栈帧指针
  ret                # 函数返回
```

函数调用示例

```
mv r0, t1           # 加载参数
mv r1, r5
mv r2, t3
mv r4, s1
sd s2, -32(sp)      # 存储调用者保存的寄存器

call my_function    # 调用函数

mv t0, r0           # 加载函数返回值
ld s2, -32(sp)      # 恢复调用者保存的寄存器
```

更加正经的分配算法

- 图着色算法：慢，效果较好
- 线性扫描算法：快，效果一般，贪心算法的变体

两者都需要活跃变量分析（一种数据流分析）支持。

活跃变量分析 (Liveness analysis)

- 每条指令执行的时候哪些变量还在用?

```
1: c = a + b   // a, b,
2: d = a + c   // a, b, c,
3: e = c + b   //    b, c
4: f = c + e   //       c,    e
5: f           //                f
```

- 反过来：每个变量在哪些区间活跃?

```
a: [1, 3)
b: [1, 4)
c: [2, 5)
d: [2, 2)
e: [4, 5)
f: [5, 6)
```

进行活跃变量分析

> 算法不简单，推荐看教材

- 每个基本块内倒序分析
  - 变量在被使用时被标记为活跃
  - 变量在被赋值时被标记为死亡
- 每个基本块入口的活跃变量是上个基本块出口的活跃变量

图着色算法 [Chaitin 1981]

- 构建冲突图：每个变量是一个节点
  - 如果有变量 a 和 b 在同一时间活跃，就在他们之间加一条边
- 对冲突图的节点进行着色（NP-hard）
  - 每个寄存器一种颜色
  - 相邻的节点颜色不能相同（同时使用的变量不能使用同一个寄存器）
  - 不能着色的变量会被 Spill 掉
- 着色的结果就是分配结果

线性扫描算法 [Poletto 1999]

类似之前的贪心算法

- 将控制流拍平，将变量按照活跃区间的开始时间排序
- 对于每个变量：
  - 将已经不活跃的变量从寄存器分配中删除
  - 如果没有寄存器可供分配，将活跃结束时间最晚的变量 spill 掉
- 优化
  - Second Chance Binpacking [Poletto 1999]
  - SSA、带洞的活跃区间 [Wimmer & Franz 2010] [Rogers 2020]
  - 启发式算法调优

其他优化方法

- Rematerialization -- 比起从内存里读，还是重新算一遍更快 [Chaitin 1981]
- Coalescing -- 预判这个值需要在哪个寄存器里用到，减少 `mov` [Chaitin 1982]
- 混合两种算法，根据条件决定启用哪个