编译原理 PA5 实验报告

2017011620 计 73 李家昊

2020 年 1 月 7 日

1 工作内容

本次实验中实现了寄存器分配的图染色算法，为简单起见，并未实现 spill。

首先实现一个寄存器干涉图 `InterferenceGraph`，在里面实现图染色算法，由于图染色是 NP 问题，因此采取近似算法以加快编译速度，具体流程如下：先对 MIPS 物理寄存器进行预染色，然后依次遍历图中的每个节点，选择一种与它的所有邻居都不同的颜色，作为当前节点的颜色。实现完成后，对图染色算法进行充分的单元测试，保证算法的正确性。

由于我在 PA4 实现了复写传播、常量传播和死代码消除，对 PA5 有一定帮助，因此这里将 PA4 的优化代码应用到 PA5。然后新建一个图染色算法类 `GraphColorRegAlloc`，替换原来的贪心算法 `BruteRegAlloc`，重写其中的 `accept` 方法，这里需要特别注意，如果函数有参数，则首先需要在第一个基本块的头部增加几条代码，将参数加载到虚拟寄存器中，保证其干涉图节点能够正常创建。然后使用 `LivenessAnalyzer` 重新求解数据流，得到每一行 MIPS 代码出口处的活跃变量 `LiveOut`。然后构建干涉图，依次遍历每一个基本块的每一行代码，将当前代码 `LiveOut` 中的每个寄存器与当前代码的目标寄存器在干涉图上连一条边，如果节点不存在，则先创建节点。接下来对干涉图进行图染色，得到每个虚拟寄存器的颜色，即为其所对应的可分配寄存器的下标。最后再次遍历每一个基本块的每一行代码，按照干涉图的颜色，将虚拟寄存器映射到对应的物理寄存器即可。

2 PA5 相关问题

Q1 描述你实现的算法的基本流程。

请参见 Section 1。

Q2 如何确定干涉图的节点？连边的条件是什么？

干涉图的节点包括 `emitter` 处理后得到的全部物理寄存器，以及所有 TAC 阶段产生的虚拟寄存器。在具体实现中，维护了一个从寄存器下标到干涉图节点的哈希表，将寄存器映射到干涉图节点。

干涉图的两个节点之间需要连一条边，当且仅当其中一个节点为某一条 MIPS 代码的目标寄存器，且另一个节点是这条代码出口处的活跃变量。

Q3 结合实际的程序 (decaf 或 TAC 程序)，比较你实现的算法与原来的贪心算法的寄存器分配结果。只从这个例子来看，两种算法哪个效果更好？

选取测例`basic-math.decaf` 中的代码片段如下

```
class Maths {
    static int pow(int a, int b) {
        int result = 1;
        for (int i = 0; i < b; i = i + 1) {
            result = result * a;
        }
        return result;
    }
}
```

原来的贪心算法的编译结果如下

```
_L_Maths_pow:  # function FUNCTION<Maths.pow>
    # start of prologue
    addiu   $sp, $sp, -44  # push stack frame
    # end of prologue

    # start of body
    sw      $a0, 0($sp)  # save arg 0
    sw      $a1, 4($sp)  # save arg 1
    li      $v1, 1
    li      $t0, 0
    sw      $v1, 36($sp)
    sw      $t0, 40($sp)
_L4:
    lw      $v1, 40($sp)
    lw      $t0, 4($sp)
    slt     $t1, $v1, $t0
    sw      $t0, 4($sp)
    sw      $v1, 40($sp)
    beqz    $t1, _L3
    lw      $v1, 36($sp)
    lw      $t0, 0($sp)
    mul     $t1, $v1, $t0
    move    $v1, $t1
```

```
    li      $t1, 1
    lw      $t2, 40($sp)
    add     $t3, $t2, $t1
    move    $t2, $t3
    sw      $t0, 0($sp)
    sw      $v1, 36($sp)
    sw      $t2, 40($sp)
    j       _L4
_L3:
    lw      $v1, 36($sp)
    move    $v0, $v1
    j       _L_Maths_pow_exit
    # end of body

_L_Maths_pow_exit:
    # start of epilogue
    addiu   $sp, $sp, 44  # pop stack frame
    # end of epilogue

    jr      $ra  # return
```

我实现的算法编译结果如下

```
_L_Maths_pow:  # function FUNCTION<Maths.pow>
    # start of prologue
    addiu   $sp, $sp, -36  # push stack frame
    # end of prologue

    # start of body
    sw      $a0, 0($sp)  # save arg 0
    sw      $a1, 4($sp)  # save arg 1
    move    $v1, $a0
    move    $t0, $a1
    li      $t1, 1
    li      $t2, 0
_L4:
    slt     $t3, $t2, $t0
    beqz    $t3, _L3
    mul     $t1, $t1, $v1
    move    $t1, $t1
    li      $t3, 1
    add     $t2, $t2, $t3
    move    $t2, $t2
    j       _L4
_L3:
    move    $v0, $t1
```

```
    j       _L_Maths_pow_exit
    # end of body

_L_Maths_pow_exit:
    # start of epilogue
    addiu   $sp, $sp, 36  # pop stack frame
    # end of epilogue

    jr      $ra  # return
```

从生成的代码行数来看，原来的贪心算法需要 30 行汇编代码，我实现的算法仅需 19 行；从寄存器分配情况来看，原来的贪心算法每次写入寄存器后都将寄存器保存到栈上，每次读取寄存器前都从栈上读取出来，效率较低，而我实现的算法中，除了框架本身需要保存参数、维护栈顶外，其他生成的代码并未对栈进行任何操作，因此效率较高。综上，从这个例子中可以看出，我实现的算法效果更好。

3 注意事项

本次 PA 的评测代码存在较多问题，建议助教在评测过程中注意以下几点。

首先，实验说明中规定 PA5 的测例与 PA4 相同，但评测代码对于 PA5 用的是 PA3 的测例，建议在`testAll.py` 做如下修改。

```
TARGETS = {
    ...
    'PA5': (['S4'], [], MipsTester),
}
```

然后，在 Linux 下，如果直接采用`sudo apt install spim` 的方式安装 spim 模拟器，则在运行时会输出版权信息，如下所示。

```
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
```

为了解决这个问题，需要在`testAll.py` 做如下修改。

```
with open(actual, 'r') as f:
    actual_lines = [line.rstrip()
                    for line in f.readlines() if line.rstrip() != '']
    if len(actual_lines) > 5 and actual_lines[4].startswith('Loaded:'
        ): # ignore spim header lines
        actual_lines = actual_lines[5:]
```

最后，如果存在超时的测例，则瓶颈不是在编译器，而是在 spim 模拟器，建议适当调整时间限制，将`testAll.py` 的`TIMEOUT_SECONDS` 调整为 10（秒）。