report-PA5

2017012289 李岱轩 计72

工作简述

本次实验中，我主要在基础框架的基础上，实现了基于图着色的寄存器分配，最终生成了目标代码。主要工作是增加了GraphColorRegAlloc类，在这个类中实现了基于图着色的寄存器分配。下面按照顺序，介绍算法流程和回答相关思考题。

图着色算法流程

外部通过调用accept 函数来进行内存分配。accept函数体现了图着色算法的整体流程：

```
public void accept(CFG<PseudoInstr> graph, SubroutineInfo info) {
        var subEmitter = emitter.emitSubroutine(info);
        //System.out.println("in new func!");
        //构建干涉图
        edges.clear();
        adjList.clear();
        degree.clear();
        for (var bb : graph) {
            buildGraph(bb);
        }
        //分配寄存器
        AllocByColor();
        //填写到tac里面
        //首先确定需要填写的所有初始化变量
        inits.clear();
        for (var reg : emitter.allocatableRegs) {
            reg.used = false;
            reg.occupied = false;
        }
        for (var bb : graph) {
            findInits(bb);
        }
        //System.out.println("inits:"+inits+"  bindings:"+bindings);
        for(var temp: inits) {
            subEmitter.emitLoadFromStack(bindings.get(temp), temp);
            bindings.get(temp).occupied = true;
            bindings.get(temp).temp = temp;
        }
        for (var bb : graph) {
            bb.label.ifPresent(subEmitter::emitLabel);
            localAlloc(bb, subEmitter);
        }
```

```
        subEmitter.emitEnd();
    }
```

在GraphColorRegAllc类中建立三个表，edges表存储所有的边，adjList表存储对于某个点的邻接点，degree表存储度数。整个算法的流程是首先建立干涉图，在干涉图上根据启发式算法求解着色问题（分配寄存器），最终把分配的结果重新在Tac代码块中生成对应的代码（使用emit函数）即可。

建立干涉图的过程，是对每一个语句(HoleInstruction除外)处的liveOut（加上第一个语句的liveIn）中的所有伪寄存器互相连接，表示彼此之间不能着同一色（同时使用同一个寄存器），最终建立成图的过程。主要代码是添加边代码，如下：

```
private void addEdge(Temp a, Temp b) {
        //孤立点情况
        if (a.compareTo(b) == 0) {
            //System.out.println("in single "+a+":"+b);
            if(!(a instanceof Reg) && !(adjList.containsKey(a))) {
                adjList.put(a,new TreeSet<>());
            }
        }
        //非孤立点情况
        if (!edges.contains(new ImmutablePair<>(a, b)) && a.compareTo(b) != 0)
{
            //System.out.println("in pair "+a+":"+b);
            edges.add(new ImmutablePair<>(a, b));
            edges.add(new ImmutablePair<>(b, a));
            if (!(a instanceof Reg)) {
                if (adjList.containsKey(a)) {
                    adjList.get(a).add(b);
                } else {
                    adjList.put(a, new TreeSet<>());
                    adjList.get(a).add(b);
                }
                if (degree.containsKey(a)) {
                    degree.put(a, degree.get(a) + 1);
                } else {
                    degree.put(a, 1);
                }
            }
            if (!(b instanceof Reg)) {
                if (adjList.containsKey(b)) {
                    adjList.get(b).add(a);
                } else {
                    adjList.put(b, new TreeSet<>());
                    adjList.get(b).add(a);
                }
                if (degree.containsKey(b)) {
                    degree.put(b, degree.get(b) + 1);
                } else {
                    degree.put(b, 1);
```

```
                }
            }
            //System.out.println(adjList);
        }
    }
```

分配寄存器使用启发式算法实现。因为题目中给定了不会发生spill，因此k直接取可以使用的寄存器最大值即可。

在这种情况下，每次循环找出度数小于k的一个点，把这个点和它的当前邻接表放在栈中，删除所有和它相邻的点的邻接表中的这个点的信息。循环往复直到图为空。然后将点从栈中一个个拿出来，每次拿出来都可以立刻着色，这是因为拿出的节点最多与k-1个已经着色的点有连线。最终形成完整的着色图。代码如下：

```
private void AllocByColor() {
        bindings.clear();
        for (var reg : emitter.allocatableRegs) {
            reg.used = false;
            reg.occupied = false;
        }
        Stack<Temp> checkPointStack = new Stack<>();
        Stack<Set<Temp>> checkEdgeStack = new Stack<>();
        int num = emitter.allocatableRegs.length;
        //System.out.println("keys:"+adjList.keySet()+"\nlength:"+num +
"\nadjs:"+adjList);
        while(adjList.keySet().size() != 0) {
            boolean check = true;
            for(var temp : adjList.keySet()) {
                if(adjList.get(temp).size() < num) {
                    checkPointStack.push(temp);
                    checkEdgeStack.push(adjList.get(temp));
                    for(var temp1 : adjList.get(temp)) {
                        if(adjList.containsKey(temp1)) {
                            adjList.get(temp1).remove(temp);
                        }
                    }
                    adjList.remove(temp);
                    check = false;
                    break;
                }
            }
            if(check) {
                //System.out.println("cannot be colored.");
                exit(1);
            }
        }
        while(checkPointStack.size() != 0) {
            var temp = checkPointStack.pop();
            var adjSet = checkEdgeStack.pop();
```

```
            Set<Reg> notEqual = new TreeSet<>();
            for(var node : adjSet) {
                if(node instanceof Reg) {
                    notEqual.add((Reg) node);
                }else {
                    notEqual.add(bindings.get(node));
                }
            }
            //System.out.println("refill:"+temp+","+notEqual+":"+adjSet);
            allocRegByColor(temp,notEqual);
        }
    }
```

最终将寄存器配合tac产生汇编。这里最重要的坑在于，要让处在调用栈上的参数在每一个函数开始的地方赋值到对应的寄存器处。如果仿照以前的框架，在每一条需要用到的地方进行取值，如果这条运算在delay-slot中，就会发生这条运算不能被执行的错误。这部分代码位于最开始的accept函数中，其中findInit函数是仿照分配寄存器函数进行编写的。

思考题的回答

Q1： 如何确定干涉图的节点？连边的条件是什么？

如前所述，干涉图的节点就是所有的伪寄存器（不考虑寄存器数量限制，每一个变量分配一个寄存器），和已经预先分配好的寄存器。连边的条件是，连边的两个伪寄存器同时出现在某一条语句的liveOut（或者第一条语句的liveIn）之中。

Q2：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?

图着色算法整体来讲更好。相对于贪心算法，图着色主要的优化有两点，一点是减少了寄存器的使用数量，第二点是减少了访存的数量。以实际的例子（basic-queue的main部分）为例：

```
main:  # function main
    # start of prologue
    addiu   $sp, $sp, -76  # push stack frame
    sw      $ra, 44($sp)  # save the return address
    # end of prologue

    # start of body
    jal     _L_Queue_new
    move    $v1, $v0
    move    $t0, $v1
    lw      $v1, 0($t0)
    lw      $v1, 16($v1)
    move    $a0, $t0
    sw      $v1, 48($sp)
    sw      $t0, 52($sp)
    jalr    $v1
    lw      $v1, 48($sp)
    lw      $t0, 52($sp)
```

```
    li      $v1, 0
    move    $t1, $v1
_L4:
    li      $v1, 10
    slt     $v1, $t1, $v1
    beqz    $v1, _L3
    lw      $v1, 0($t0)
    lw      $v1, 12($v1)
    move    $a0, $t0
    move    $a1, $t1
    sw      $v1, 56($sp)
    sw      $t0, 52($sp)
    sw      $t1, 60($sp)
    jalr    $v1
    lw      $v1, 56($sp)
    lw      $t0, 52($sp)
    lw      $t1, 60($sp)
    li      $v1, 1
    add     $v1, $t1, $v1
    move    $t1, $v1
    j       _L4
```

这是图着色的版本。

```
main:  # function main
    # start of prologue
    addiu   $sp, $sp, -76  # push stack frame
    sw      $ra, 44($sp)  # save the return address
    # end of prologue

    # start of body
    jal     _L_Queue_new
    move    $v1, $v0
    move    $t0, $v1
    lw      $v1, 0($t0)
    lw      $t1, 16($v1)
    move    $a0, $t0
    sw      $t0, 48($sp)
    sw      $t1, 52($sp)
    jalr    $t1
    lw      $t0, 48($sp)
    lw      $t1, 52($sp)
    li      $v1, 0
    move    $t1, $v1
    sw      $t0, 48($sp) #
    sw      $t1, 56($sp) #
_L4:
    li      $v1, 10
    lw      $t0, 56($sp) #
```

```
    slt     $t1, $t0, $v1
    sw      $t0, 56($sp) #
    beqz    $t1, _L3
    lw      $v1, 48($sp) #
    lw      $t0, 0($v1)
    lw      $t1, 12($t0)
    move    $a0, $v1
    lw      $t0, 56($sp)  #
    move    $a1, $t0
    sw      $v1, 48($sp)
    sw      $t0, 56($sp)
    sw      $t1, 60($sp)
    jalr    $t1
    lw      $v1, 48($sp)
    lw      $t0, 56($sp)
    lw      $t1, 60($sp)
    li      $t1, 1
    add     $t2, $t0, $t1
    move    $t0, $t2
    sw      $v1, 48($sp) #
    sw      $t0, 56($sp) #
    j       _L4
```

这是贪心的版本。

可以看到贪心版本多了store指令（所有标定# 的，这是因为每次出一个块都要在栈上保存所有的变量，后来再恢复），因此访存变多。贪心版本多使用了一个t2寄存器。

困难与感想

本次的主要困难在于发现从栈上取参数要在函数最开始进行，否则会出现delay-slot处无法执行运算报错。

这是最后一次PA了。回顾整个PA历程，虽然艰苦，也仅仅是体会到了系统工作者艰辛的一小部分吧。希望PA越来越好。