优化文档_21373061_方沐阳

我选择做的优化有：

1.删去死函数

在第一次生成中间代码的时候，我统计了每个函数被调用的数量。从而在优化的时候删除被调用次数为0的函数。

```
public void startOptimaze() {
        this.optimaze = true;
        if (!name.equals("main")) {
            if (this.calledNum == 0) {
                //判断为死函数
                //如果不优化是不会判断任何函数为死的
                output = false;
            }
        }
    }
```

2.基本快合并

在中间代码层面，遍历每一个BasicBlock，如果有只含一条无条件跳转到下一个BasicBlock的BrInst，将这个BasicBlock删除，并且把所有跳转到它的指令改为跳转到它的下一个BasicBlock。

```
public void optimazeBlockJump() {
        for (BasicBlock block : this.basicBlocks) {
            if (block.instructions.size() == 1 &&
                    block.instructions.get(0) instanceof BrInst) {
                //只有一条Br指令
                BrInst brInst = (BrInst) block.instructions.get(0);
                int curBlockNum = block.registerNum;
                if (brInst.type == 1 &&
                        ((BasicBlock) brInst.dest).registerNum == curBlockNum +
1) {
                    //无条件跳转到下一个BasicBlock的情况
                    block.delete = true;//优化
                    //以下:标记需要更改的跳转语句
                    Iterator<Map.Entry<BasicBlock, BasicBlock>> iterator =
replaceJumpInstructions.entrySet().iterator();
                    while (iterator.hasNext()) {
                        Map.Entry<BasicBlock, BasicBlock> entry =
iterator.next();
                        //System.out.println("Key = " + entry.getKey() + ",
Value = " + entry.getValue());
                        BasicBlock key = entry.getKey();
                        BasicBlock value = entry.getValue();
                        if (value.equals(block)) {
                            replaceJumpInstructions.put(key, ((BasicBlock)
brInst.dest));
                        }
                    }
                    replaceJumpInstructions.put(block, ((BasicBlock)
brInst.dest));
                    //done
                }
            }
        }
        for (BasicBlock block : this.basicBlocks) {
            if (block.terInst instanceof BrInst) {
                BrInst brInst = (BrInst) block.terInst;
                if (brInst.type == 1) {
                    BasicBlock dest = (BasicBlock) brInst.dest;
                    if (replaceJumpInstructions.containsKey(dest)) {
                        brInst.dest = replaceJumpInstructions.get(dest);
                    }
                } else {
                    BasicBlock ifTrue = (BasicBlock) brInst.ifTure;
                    if (replaceJumpInstructions.containsKey(ifTrue)) {
                        brInst.ifTure = replaceJumpInstructions.get(ifTrue);
                    }
                    BasicBlock ifFalse = (BasicBlock) brInst.ifFalse;
                    if (replaceJumpInstructions.containsKey(ifFalse)) {
                        brInst.ifFalse = replaceJumpInstructions.get(ifFalse);
                    }
                }
            }
        }
        replaceJumpInstructions = new HashMap<>();
    }
```

3.常量优化

我进行常量优化的基本思想是，如果一条指令的的操作数都是常数，即这条指令的结果可以被计算出来，替换为某个常数，则把以该指令结果为操作数指令中的该指令结果替换为常数。由于替换之后可能回出现新的满足果**一条指令的的操作数都是常数**的指令，因此需要迭代多次，直至没有变化为止。这个优化是跨块的。在`Function`类中实现。

```
 public void optimizeCalculation() {
        int change;
        do {
            change = replaceConstantValue();
            System.out.println("change is " + change);
        } while (change != 0);
    }

    private int replaceConstantValue() {
        int change = 0;
        HashMap<Value, ConstValue> toReplace = new HashMap<>();
        for (BasicBlock block : this.basicBlocks) {
            ArrayList<Instruction> toRemove = new ArrayList<>();
            for (Instruction inst : block.instructions) {
                if (inst instanceof BinaryInst) {
                    BinaryInst binaryInst = (BinaryInst) inst;
                    if (binaryInst.op1 instanceof ConstValue && binaryInst.op2
instanceof ConstValue) {
                        //标记这条Instruction
                        //block.instructions.remove(inst);//?直接remove
                        toRemove.add(inst);
                        int ans = Operator.cal(
                                ((ConstValue) (binaryInst.op1)).getNum(),
                                ((ConstValue) (binaryInst.op2)).getNum(),
                                binaryInst.operator
                        );
                        ConstValue newValue = new
ConstValue(String.valueOf(ans));
                        Value oldResult = binaryInst.result;
                        //要把所有的这条binaryInst删掉
                        //并把其余Instruction里所有的oldResult全部替换为newValue
                        toReplace.put(oldResult, newValue);
                        change++;
                    }
                }
            }
            for(Instruction inst:toRemove){
                block.instructions.remove(inst);
            }
        }
        for (BasicBlock block : this.basicBlocks) {
            for (Instruction inst : block.instructions) {
                for (Value key : toReplace.keySet()) {
                    inst.replaceValueWithConst(key,toReplace.get(key));
                }
            }
        }
        return change;
    }
```

4.算数优化，主要包含乘法优化

算数优化主要在后端实现，其主要目的是简化计算指令。其中最重要的乘法优化的主要思想为：

- 如果 $d = 0$，那就直接不用算，直接得到结果 $p = 0$
- 如果 $d = \pm 1$，那也不用算，直接令 $p = a$
- 如果 $d = 2^k$，那就直接计算 $p = a << k$
- 如果 $d = 2^k + 1$，那就直接计算 $p = (a << k) + a$
- 如果 $d = 2^k - 1$，那就直接计算 $p = (a << k) - a$

实现如下：

```
if (optimize) {
            if (v1 instanceof ConstValue &&
                    v2 instanceof ConstValue) {
                //都为常量
                String reg = "$t0";
                int a = Integer.parseInt(((ConstValue) v1).num);
                int b = Integer.parseInt(((ConstValue) v2).num);
                int ans = Operator.cal(a, b, op);
                mipsFactory.genLi(String.valueOf(ans), reg);
                this.minusSp();
                mipsFactory.genSw("$t0");
                this.spTable.put(result.getMipsName(), new ValuePlace(curSp));
                return;
            } else if ((v1 instanceof ConstValue || v2 instanceof ConstValue)) {
                //有一是常数
```

```
        //不一定能成功优化，只有成功优化才返回
        boolean success = false;
        int num;
        //变为t1与num做运算保存到t0
        if (v1 instanceof ConstValue) {
            num = Integer.parseInt(((ConstValue) v1).num);
            this.visitValueToReg("$t1", v2);
        } else {
            num = Integer.parseInt(((ConstValue) v2).num);
            this.visitValueToReg("$t1", v1);
        }
        //优化上述运算
        String reg = "$t0";
        if (op == Operator.add) {
            //加
            this.texts.add("addiu " + reg + ",$t1," + num + "\n");//可以少一条li
            success = true;
        } else if (op == Operator.mul) {
            if (num == 0) {
                //*0
                mipsFactory.genLi("0", reg);
                success = true;
            } else if (num == 1) {
                //*1
                reg = "$t1";//直接把t1存进去
                success = true;
            } else if (num == -1) {
                //* -1
                this.texts.add("subu " + reg + ",$zero,$t1\n");
                success = true;
            } else if (isPowerOfTwo(num)) {
                //2的n次方
                int k = countPowerOfTwo(num);
                this.texts.add("sll " + reg + "," + "$t1" + "," + k + "\n");
                success = true;
            } else if (isPowerOfTwo(num + 1)) {
                //2的n次方-1
                int k = countPowerOfTwo(num + 1);
                this.texts.add("sll " + reg + "," + "$t1" + "," + k + "\n");//t0 = t1 << k
                this.texts.add("subu " + reg + "," + reg + "," + "$t1" + "\n");//t0 = t0 - t1;
                success = true;
            } else if (isPowerOfTwo(num - 1)) {
                //2的n次方+1
                int k = countPowerOfTwo(num - 1);
                this.texts.add("sll " + reg + "," + "$t1" + "," + k + "\n");//t0 = t1 << k
                this.texts.add("addu " + reg + "," + reg + "," + "$t1" + "\n");//t0 = t0 + t1;
                success = true;
            }
        } else if (op == Operator.sdiv) {
            //todo 除法优化
        }
        if (success) {
```

```
                    this.minusSp();
                    mipsFactory.genSw(reg);
                    this.spTable.put(result.getMipsName(), new
ValuePlace(curSp));
                    return;
                }
            }
        }
```