优化文章

优化主要有三个部分,分别是mem2reg、removePhi、regAlloca。

但是为了支撑三者的顺利进行，必须要提前完成删除死代码、支配关系的分析以及活跃变量的分析。

死代码删除

这里其实并不是一个完整的死代码删除，更像是删除不会进入的BasicBlock，因为不这么做mem2reg时会出bug，有点分析有向图的连通分量的味道。具体实现代码如下：

```
public static void deleteDeadCode(Module module) {
    for (Function function : module.getFunctions()) {
        // 删除每一个bb里的死代码,这个我们通过新建bb解决了
        // 删除死的bb,也就是entry到不了的bb，这里我们只看指令，不看具体结果
        HashSet<BasicBlock> liveBbs = new HashSet<>();
        dfs2findLive(function.getFirstBb(), liveBbs);
        // 接下来开始删了
        ArrayList<Integer> indexs = new ArrayList<>();
        LinkedList<BasicBlock> bbs = function.getBbs();
        for (int i = 0; i < bbs.size(); i++) {
            if (!liveBbs.contains(bbs.get(i))) {
                indexs.add(i);
            }
        }
        // 需要倒着来
        for (int i = indexs.size() - 1; i >= 0; i--) {
            function.deleteBb(indexs.get(i));
        }
    }
}

private static void dfs2findLive(BasicBlock bb, HashSet<BasicBlock> liveBbs) {
    if (!liveBbs.contains(bb)) {
        // 只有不包含时才有意义做下去
        liveBbs.add(bb);
        Instruction end = bb.getInstrAtEnd();
        if (end instanceof Branch branch) {
            BasicBlock trueBb = (BasicBlock) branch.getOperand(1);
            BasicBlock falseBb = (BasicBlock) branch.getOperand(2);
            dfs2findLive(trueBb, liveBbs);
            dfs2findLive(falseBb, liveBbs);
        } else if (end instanceof Jump jump) {
            BasicBlock toBb = (BasicBlock) jump.getOperand(0);
            dfs2findLive(toBb, liveBbs);
        }
    }
}
```

控制流和支配关系分析

这部分可以直接按照教程中给出的算法，依次完成如下四个子任务：

- 构建**控制流程图**：

```
public void calcCFG() {
    for (BasicBlock bb : bbs) {
        bb.isVisited = false;
        bb.frontBbs = new HashSet<>();
        bb.backBbs = new HashSet<>();
    }
    makeFrontAndBack(getFirstBb());
}

private void makeFrontAndBack(BasicBlock curBb) {
    if (curBb.isVisited) {
        return;
    }
    curBb.isVisited = true;
    Instruction instruction = curBb.getInstrAtEnd();
    if (instruction instanceof Jump jump) {
        BasicBlock targetBb = (BasicBlock) jump.getOperand(0);
        curBb.backBbs.add(targetBb);
        targetBb.frontBbs.add(curBb);
        makeFrontAndBack(targetBb);
    } else if (instruction instanceof Branch branch) {
        BasicBlock bb1 = (BasicBlock) branch.getOperand(1);
        BasicBlock bb2 = (BasicBlock)  branch.getOperand(2);

        curBb.backBbs.add(bb1);
        bb1.frontBbs.add(curBb);
        makeFrontAndBack(bb1);

        curBb.backBbs.add(bb2);
        bb2.frontBbs.add(curBb);
        makeFrontAndBack(bb2);
    }
}
```

- 获得每个BasicBlock的支配者集合：

```
public void calcDomByRel() {
    initDomBys();
    boolean calcing = true;
    while(calcing) {
        calcing = false;
        // 跳过entry
        for (int i = 1; i < bbs.size(); i++) {
            BasicBlock bb = bbs.get(i);
            HashSet<BasicBlock> ansBbs = new HashSet<>(){{
                addAll(bbs);
            }};
            // 遍历每一个先驱
            for (BasicBlock frontBB : bb.frontBbs) {
                // 求二者domBys交集
                HashSet<BasicBlock> tempBbs = new HashSet<>();
                for (BasicBlock tmp : ansBbs) {
                    if (frontBB.domBys.contains(tmp)) {
```

```
                        tempBbs.add(tmp);
                    }
                }
                ansBbs = tempBbs;
            }
            // 还需要加上自身
            ansBbs.add(bb);
            // 判断ansBbs是否和bb的domBys相同
            if (ansBbs.size() != bb.domBys.size()) {
                calcing = true;
            } else {
                boolean flag = false;
                for (BasicBlock tmp : bb.domBys) {
                    if (!ansBbs.contains(tmp)) {
                        flag = true;
                        break;
                    }
                }
                if (flag) {
                    calcing = true;
                }
            }
            bb.domBys = ansBbs;
        }
    }
}
```

- 获得每个BasicBlock被谁直接支配和直接支配的集合：

```
// 计算直接支配者
public void calcImmeDom() {
    for (BasicBlock curBb : bbs) {
        // 越过entry，因为entry没有imme
        if (curBb == bbs.get(0)) {
            continue;
        }
        // 遍历所有的支配者
        for (BasicBlock domBy : curBb.domBys) {
            // 因为自己也是自己的支配者，应该略过这种情况
            if (domBy == curBb) {
                continue;
            }
            boolean flag = true; // 说明是直接支配者
            for (BasicBlock otherDomBy : curBb.domBys) {
                if (otherDomBy == curBb) {
                    continue;
                }
                if (otherDomBy == domBy) {
                    continue;
                }
                if (otherDomBy.domBys.contains(domBy)) {
                    flag = false;
                    break;
                }
            }
```

```
            if (flag) {
                domBy.immeDomTos.add(curBb);
                curBb.immeDomBy = domBy;
                break;
            }
        }
    }
}
```

- 计算支配边界

```
// 计算支配边界
public void calcDomFro() {
    // 目前先不需要清空原有的df
    for (BasicBlock curBb : bbs) {
        for (BasicBlock backBb : curBb.backBbs) {
            BasicBlock tmp = curBb;
            while(tmp == backBb || !backBb.domBys.contains(tmp)) {
                tmp.domFro.add(backBb);
                tmp = tmp.immeDomBy;
            }
        }
    }
```

mem2reg

在做完之前的预备工作后，我们终于可以大张旗鼓地开始进行mem2reg了。

实际上其实就是只有两部分工作：在合适的位置插入Phi指令+变量重命名。

插入Phi指令

由于在生成中间代码的时候，我们已经将所有alloca指令移到了entryBlock中，所以这里我们遍历entryBlock中的指令，判断其是否是alloca指令以及是否不是数组，来确定要不要尝试加入Phi指令。

```
private static void addPhi(BasicBlock entryBb, Alloca alloca) {
    // 准备工作
    writeStack.clear();
    readBbList.clear();
    writeBbList.clear();
    readInstrList.clear();
    writeInstrList.clear();
    // 更新一波list
    for (Use use : alloca.getUses()) {
        User user = use.getUser();
        if (!((BasicBlock) user.getHost()).getIsLive() ) {
            continue;
        }
        if (user instanceof Store store) {
            if (!writeBbList.contains((BasicBlock) store.getHost())) {
                writeBbList.add((BasicBlock) store.getHost());
            }
            writeInstrList.add(store);
        } else if (user instanceof Load load) {
            if (!readBbList.contains((BasicBlock) load.getHost())) {
```

```
                readBbList.add((BasicBlock) load.getHost());
            }
            readInstrList.add(load);
        }
    }

    HashSet<BasicBlock> vis = new HashSet<>();
    LinkedList<BasicBlock> work = new LinkedList<>(writeBbList);
    // 开始插入phi
    while (work.size() != 0) {
        BasicBlock curBb = work.get(0);
        work.remove(0);
        for (BasicBlock dfBb : curBb.domFro) {
            if (!vis.contains(dfBb)) {
                // 如果vis中没有dfBb， 将其加入进去
                vis.add(dfBb);
                // 如果dfBb不在writeBbList中，dfBb的支配边界可能未被探索
                if (!writeBbList.contains(dfBb)) {
                    work.add(dfBb);
                }
                Phi phi = IrFactory.makePhi(((PointerIrTy)
alloca.getType()).deRefIrTy, dfBb);
                dfBb.add2head(phi);
                readInstrList.add(phi);
                writeInstrList.add(phi);
            }
        }
    }
    // 进行重新命名的工作
    renew(alloca, entryBb);
}
```

变量重命名

在引入Phi指令的时候，我们改变了许多指令的Use关系，这里我们需要进行维护，具体而言就是进行一个DFS：

```
private static void renew(Alloca curAlloca, BasicBlock curBb) {
    int counter = 0;
    Iterator<Instruction> iter = curBb.getAllInstr().iterator();
    while(iter.hasNext()) {
        Instruction instruction = iter.next();
        if (instruction instanceof Phi && writeInstrList.contains(instruction))
{
            counter += 1;
            writeStack.push(instruction);
        } else if (instruction == curAlloca) {
            iter.remove();
        } else if (instruction instanceof Store &&
writeInstrList.contains(instruction)) {
            counter += 1;
            // 将store所使用的值压栈
            writeStack.push(instruction.getOperand(0));
            instruction.clearAllOperands();
            iter.remove();;
```

```
        } else if (instruction instanceof Load &&
readInstrList.contains(instruction)) {
            Value top = writeStack.empty() ? new ConstData(DataIrTy.I32, 0,
true) : writeStack.peek();
            instruction.giveUserNewUsed(top); // 感觉没必要加一个undefined
            instruction.clearAllOperands();
            iter.remove();
        }
    }
    // 遍历curBb的后继，将最新的write 或者说 define 写进phi指令中
    for (BasicBlock back : curBb.backBbs) {
        Instruction instruction = back.getAllInstr().get(0);
        if (readInstrList.contains(instruction) && instruction instanceof Phi
phi) {
            Value top = writeStack.empty() ? new ConstData(DataIrTy.I32, 0,
true) : writeStack.peek();
            phi.fill(top, curBb);
        }
    }
    for (BasicBlock immeDomTo : curBb.immeDomTos) {
        renew(curAlloca, immeDomTo);
    }
    while (counter > 0) {
        counter--;
        writeStack.pop();
    }
}
```

removePhi

phi指令过于抽象了，mips中没有任何一种指令可以简单的对应过来。必须将其转换为更具体的操作指令。为了实现 Phi 指令的去除，需要将其逻辑显式化为具体的控制流与变量赋值指令。

- phi 转化为 copy 指令
  - 对于单一来源的基本块，直接加入 copy 指令即可。
  - 对于多来源的基本块，则需要创建一个中间块，将 copy 指令插入其中，并修改相关分支控制语句。
- 通过插入新的 copy 指令解决并行赋值的问题和寄存器共享的问题。

具体代码如下：

```
private static void tackleOneBb(BasicBlock curBb, HashMap<Value, Reg> value2reg)
{
    // 获取curBb的每一个前序块的copy指令,并更新...
    HashMap<BasicBlock, ArrayList<Copy>> front2copy = getFront2copy(curBb);

    for (BasicBlock frontBb : curBb.frontBbs) {
        ArrayList<Copy> copyList = front2copy.get(frontBb);
        // 如果frontBb中根本就没有copy，那直接跳过即可
        if (copyList.isEmpty()) {
            continue;
        }
        // 构造可以并行的copy指令序列
        ArrayList<Copy> fakePCopyList = getParallelCopyList(copyList, curBb);
```

```
        // 寄存器分配之后，可能会存在寄存器共用的问题
        ArrayList<Copy> pCopyList = tackleRegError(fakePCopyList, curBb,
value2reg);
        // 开始将copy指令插入进去，对于有多个后继的块，需要构造一个中间块
        if (frontBb.backBbs.size() < 2) {
            for (Copy copy : pCopyList) {
                LinkedList<Instruction> allInstr = frontBb.getAllInstr();
                int index = allInstr.size() - 1;
                allInstr.add(index, copy);
            }
        } else {
            // 有多个后继的情况
            // 生成一个bb并放到合适位置
            Function function = (Function) curBb.getHost();
            BasicBlock middleBb = new BasicBlock(function,
IrFactory.nameCounter++);
            function.addBbBeforeOne(curBb, middleBb);
            // 向bb里面插入指令，包括copy和跳转指令
            for (Copy copy : pCopyList) {
                middleBb.add2end(copy);
            }
            Jump jump = new Jump(middleBb, curBb);
            middleBb.add2end(jump);
            // 修改frontBb的最后一条branch指令
            Branch branch = (Branch) frontBb.getInstrAtEnd();
            branch.replaceOperand(curBb, middleBb); // Use关系应该就不准了，但是无所
谓，反正以后也不用了
        }
    }
}

private static ArrayList<Copy> getParallelCopyList(ArrayList<Copy> inCopyList,
BasicBlock curBb) {
    LinkedList<Copy> outCopyList =  new LinkedList<>();
    for (int i = 0; i < inCopyList.size(); i+=1) {
        Copy iCopy = inCopyList.get(i);
        for (int j = i + 1; j < inCopyList.size(); j+=1) {
            // i 目前在 j 前面
            Copy jCopy = inCopyList.get(j);
            if (iCopy.getTarget() != jCopy.getFrom()) {
                // i 写入的东西不是 j 使用的东西，那当然没有问题
                // 话说如果能用Verilog写就好了，，，
                continue;
            }
            // 下面说明有问题，也就是i的target是j的from
            // 可以构造一个新的copy，将i写入一个新的middle value，然后将middle value写入
j及之后需要的copy
            Value middleValue =
IrFactory.makeMiddleValue(iCopy.getTarget().getType());
            Copy middleCopy = new Copy(curBb, middleValue, iCopy.getTarget());
            outCopyList.addFirst(middleCopy);
            for (int k = j; k < inCopyList.size(); k+=1) {
                Copy kCopy = inCopyList.get(k);
                if (kCopy.getFrom() == iCopy.getTarget()) {
                    kCopy.setFrom(middleValue);
```

```
                }
            }
        }
        outCopyList.addLast(iCopy);
    }
    return new ArrayList<>(outCopyList);
}

private static ArrayList<Copy> tackleRegError(ArrayList<Copy> fakePCopyList,
BasicBlock curBb, HashMap<Value, Reg> value2reg) {
    ArrayList<Copy> pCopyList = new ArrayList<>();
    for (int i = 0; i < fakePCopyList.size(); i+=1) {
        Copy iCopy = fakePCopyList.get(i);
        for (int j = i + 1; j < fakePCopyList.size(); j+=1) {
            Copy jCopy = fakePCopyList.get(j);
            if (value2reg.containsKey(iCopy.getTarget()) &&
value2reg.containsKey(jCopy.getFrom()) &&
                value2reg.get(iCopy.getTarget())  ==
value2reg.get(jCopy.getFrom())) {

                Value middleValue =
IrFactory.makeMiddleValue(iCopy.getTarget().getType());
                Copy middleCopy = new Copy(curBb, middleValue,
iCopy.getTarget());
                pCopyList.add(0, middleCopy);
                for (int k = j; k < fakePCopyList.size(); k+=1) {
                    Copy kCopy = fakePCopyList.get(k);
                    if (value2reg.containsKey(kCopy.getFrom()) &&
value2reg.get(iCopy.getTarget())  == value2reg.get(kCopy.getFrom())) {
                        kCopy.setFrom(middleValue);
                    }
                }
            }
        }
        pCopyList.add(iCopy);
    }
    return pCopyList;
}


private static HashMap<BasicBlock, ArrayList<Copy>> getFront2copy(BasicBlock
curBb) {
    // 先初始化一波
    HashMap<BasicBlock, ArrayList<Copy>> front2copy = new HashMap<>();
    Iterator<Instruction> iterator = curBb.getAllInstr().iterator();
    for (BasicBlock frontBb : curBb.frontBbs) {
        front2copy.put(frontBb, new ArrayList<>());
    }
    // 开始干活
    while(iterator.hasNext()) {
        Instruction instruction = iterator.next();
        if (instruction instanceof Phi phi) {
            // 只有instruction是Phi指令的时候才需要干活
            // 首先先删除不可能到达的<value, frontBbs>对
            LinkedList<BasicBlock> fronts = phi.getBbs();
```

```
            ArrayList<Value> values = phi.getValues();
            ArrayList<Integer> toDeletes = new ArrayList<>();
            for (int i =  0; i < fronts.size(); i++) {
                if (curBb.frontBbs.contains(fronts.get(i))) {
                    continue;
                }
                toDeletes.add(i);
            }
            for (int i = toDeletes.size() - 1; i >= 0; i--) {
                fronts.remove(i);
                values.remove(i);
            }
            // 之后更新front2copy
            for (int i = 0; i < values.size(); i+=1) {
                Value value = values.get(i);
                if (value.isJustPlaceholder()) {
                    continue;
                }
                BasicBlock front = fronts.get(i);
                Copy copy = new Copy(front, phi, value);
                front2copy.get(front).add(copy);
            }
            iterator.remove();
        }
    }
    return front2copy;
}
}
```

寄存器分配

这里我首先做了活跃变量的分析，代码如下：

```
private void analyze(Module module) {
    for (Function function : module.getFunctions()) {
        HashMap<BasicBlock, HashSet<Value>> bb2In = new HashMap<>();
        HashMap<BasicBlock, HashSet<Value>> bb2Out = new HashMap<>();
        for (BasicBlock curBb : function.getBbs()) {
            curBb.makeDefUse();
            bb2Out.put(curBb, new HashSet<>());
            bb2In.put(curBb, new HashSet<>());
        }
        boolean needBreak = false;
        while(needBreak == false) {
            needBreak = true;
            LinkedList<BasicBlock> bbs = function.getBbs();
            for (int i = bbs.size() - 1; i >= 0; i-=1) {
                BasicBlock curBb = bbs.get(i);
                HashSet<Value> outSet = new HashSet<>();
                HashSet<Value> inSet = new HashSet<>();
                for (BasicBlock back : curBb.backBbs) {
                    HashSet<Value> backInSet = bb2In.get(back);
                    for (Value v : backInSet) {
                        outSet.add(v);
                    }
```

```
                }
                for (Value v : outSet) {
                    inSet.add(v);
                }
                for (Value v : curBb.defSet) {
                    if (inSet.contains(v)) {
                        inSet.remove(v);
                    }
                }
                for (Value v : curBb.useSet) {
                    if (!inSet.contains(v)) {
                        inSet.add(v);
                    }
                }
                if (!outSet.equals(bb2Out.get(curBb)) ||
!inSet.equals(bb2In.get(curBb))) {
                    needBreak = false;
                }
                bb2Out.put(curBb, outSet);
                bb2In.put(curBb, inSet);
            }
            for (BasicBlock curBb : function.getBbs()) {
                curBb.outSet = bb2Out.get(curBb);
                curBb.inSet = bb2In.get(curBb);
            }
        }

    }
}
```

之后，基于活跃变量集，我做了引用计数寄存器分配法：

- 确定每个Value的引用值：

```
for (int i = 0; i < size; i++) {
    Instruction instruction = instructions.get(i);
    if (null != instruction.getName()) {
        if (value2citeNum.containsKey(instruction)) {
            double tmp = value2citeNum.get(instruction);
            // tmp = tmp  + (1.0 * size - 1.0 * i) * i + 1;
            tmp = tmp + 1;
            value2citeNum.put(instruction, tmp);
        } else {
            double tmp = 1; // (1.0 * size - 1.0 * i) * i + 1;
            value2citeNum.put(instruction, tmp);
        }
    }
    for (Value v : instruction.getAllOperands()) {
        if (value2citeNum.containsKey(instruction)) {
            double tmp = value2citeNum.get(instruction);
            // tmp = tmp  + (1.0 * size - 1.0 * i) * i + 1.2;
            tmp = tmp + 1;
            value2citeNum.put(instruction, tmp);
        } else {
            double tmp = 1; // (1.0 * size - 1.0 * i) * i + 1.2;
```

```
                value2citeNum.put(instruction, tmp);
            }
        }
    }
```

- 尝试分配寄存器：

  这里说了是“尝试”，是因为有时候寄存器可能没有空闲的，此时将会比较当前占有寄存器的Value的引用值是否小于当前value来决定是否要将两者替换。

```
private void allocaBlock(BasicBlock curBb) {
    HashSet<Value> noUsedSet = new HashSet<>();
    HashMap<Value, Instruction> value2finalUse = new HashMap<>();
    HashSet<Value> hasDefSet = new HashSet<>();
    LinkedList<Instruction> instructions = curBb.getAllInstr();

    for (Instruction instruction : instructions) {
        for (Value operand : instruction.getAllOperands()) {
            value2finalUse.put(operand, instruction);
        }
    }

    for (Instruction instruction : instructions) {
        if (!(instruction instanceof Phi)) {
            for (Value value : instruction.getAllOperands()) {
                if (value2finalUse.get(value) == instruction) {
                    if (value2reg.containsKey(value) &&
curBb.outSet.contains(value) == false) {
                        Reg reg = value2reg.get(value);
                        noUsedSet.add(value);
                        reg2value.remove(reg);
                    }
                }
            }
        }

        if (instruction.getName() == null) {
            continue;
        }
        if (instruction instanceof Alloca alloca && ((PointerIrTy)
alloca.getType()).deRefIrTy.isArray()) {
            continue;
        }
        hasDefSet.add(instruction);
        // 下面尝试分配寄存器
        Reg res = null;
        for (Reg reg : freeRegs) {
            if (reg2value.containsKey(reg)) {
                continue;
            }
            res = reg;
            break;
        }
        if (res != null) {
            reg2value.put(res, instruction);
```

```
            value2reg.put(instruction, res);
        } else {
            Double min = Double.MAX_VALUE;
            for (Reg tmpReg : freeRegs) {
                Value tmpValue = reg2value.get(tmpReg);
                Double tmpDouble = value2citeNum.get(tmpValue);
                if (tmpDouble < min) {
                    res = tmpReg;
                    min = tmpDouble;
                }
            }
            if (value2citeNum.get(instruction) > min) {
                if (reg2value.containsKey(res)) {
                    Value resValue = reg2value.get(res);
                    value2reg.remove(resValue);
                }
                reg2value.put(res, instruction);
                value2reg.put(instruction, res);
            }
        }
    }

    for (BasicBlock bb : curBb.immeDomTos) {
        HashMap<Reg, Value> afterNoUse = new HashMap<>();
        for (Reg reg : reg2value.keySet()) {
            Value v = reg2value.get(reg);
            if (bb.inSet.contains(v)) {
                continue;
            }
            afterNoUse.put(reg, v);
        }
        for (Reg reg : afterNoUse.keySet()) {
            reg2value.remove(reg);
        }
        // dfs
        allocaBlock(bb);
        for (Reg reg : afterNoUse.keySet()) {
            Value v = afterNoUse.get(reg);
            reg2value.put(reg, v);
        }
    }

    for (Value v : hasDefSet) {
        if (value2reg.containsKey(v)) {
            Reg r = value2reg.get(v);
            reg2value.remove(r);
        }
    }

    for (Value v : noUsedSet) {
        if (hasDefSet.contains(v)) {
            continue;
        }
        if (!value2reg.containsKey(v)) {
            continue;
```

```
        }
        Reg r = value2reg.get(v);
        reg2value.put(r, v);
    }
}
```