优化设计

> 因为能力有限而且时间不太够，所以简单做了两个优化，没有做中间代码层次的优化。一个是运算方面对于乘除法，模运算进行了形式转化并优化，第二个小优化是在生成中间代码的时候结合符号表，进行常量直取，包括遇到常量、常量数组直接取出数字，如果表达式只包括常量之间运算直接算出结果等优化，将运算和数值提取直接放到代码生成阶段，不必放到mips运行阶段。

1 运算优化

1.1 乘法优化

- 对2的次方的数的乘积直接替换为sll指令减少乘法指令
- 如果有一乘数为0直接li为0，减少mult指令

```
                if (isPowerOfTwo(rightnum)) {
                    //2的次方
                    int mi = (int) (Math.log(rightnum) / Math.log(2));
                    genaddSll(destreg, leftreg, mi);
                } else if (rightnum == 0) {
                    genli(destreg, 0);
                } else {
                    genli(RegEnum.fp, rightnum);
                    genadd("mult", leftreg, RegEnum.fp);
                    genadd("mflo", destreg);
                }
```

1.2 除法优化

- 当两个操作数中一个除数为常数时，进行优化，用位移与加减运算代替除法
- 相关实现参考了一篇论文《Division by Invariant Integers using Multiplication》中的计算公式，论文来源于一篇学长的优化文档，对其进行了参考实现：

```
**procedure** CHOOSE_MULTIPLIER(**uword** $d$, **int** $prec$);
**Cmt**. $d$ – Constant divisor to invert. $1 \le d < 2^N$.
**Cmt**. $prec$ – Number of bits of precision needed, $1 \le prec \le N$.
**Cmt**. Finds $m$, $sh_{\text{post}}$, $\ell$ such that:
**Cmt**.        $2^{\ell-1} < d \le 2^{\ell}$.
**Cmt**.        $0 \le sh_{\text{post}} \le \ell$. If $sh_{\text{post}} > 0$, then $N + sh_{\text{post}} \le \ell + prec$.
**Cmt**.        $2^{N+sh_{\text{post}}} < m * d \le 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})$.
**Cmt**. Corollary. If $d \le 2^{prec}$, then $m < 2^{N+sh_{\text{post}}} * (1 + 2^{1-\ell})/d \le 2^{N+sh_{\text{post}}-\ell+1}$.
**Cmt**.                  Hence $m$ fits in $\max(prec,\ N - \ell) + 1$ bits (unsigned).
**Cmt**.
**int** $\ell = \lceil \log_2 d \rceil$,    $sh_{\text{post}} = \ell$;
**udword** $m_{\text{low}} = \lfloor 2^{N+\ell}/d \rfloor$,    $m_{\text{high}} = \lfloor (2^{N+\ell} + 2^{N+\ell-prec})/d \rfloor$;
**Cmt**. To avoid numerator overflow, compute $m_{\text{low}}$ as $2^N + (m_{\text{low}} - 2^N)$.
**Cmt**. Likewise for $m_{\text{high}}$. Compare $m'$ in Figure 4.1.
**Invariant**. $m_{\text{low}} = \lfloor 2^{N+sh_{\text{post}}}/d \rfloor < m_{\text{high}} = \lfloor 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})/d \rfloor$.
**while** $\lfloor m_{\text{low}}/2 \rfloor < \lfloor m_{\text{high}}/2 \rfloor$ **and** $sh_{\text{post}} > 0$ **do**
    $m_{\text{low}} = \lfloor m_{\text{low}}/2 \rfloor$;    $m_{\text{high}} = \lfloor m_{\text{high}}/2 \rfloor$;    $sh_{\text{post}} = sh_{\text{post}} - 1$;
**end while**;                          /* Reduce to lowest terms. */
**return** $(m_{\text{high}},\ sh_{\text{post}},\ \ell)$;       /* Three outputs.           */
**end** CHOOSE_MULTIPLIER;
```

Figure 6.2: Selection of multiplier and shift count

```
Inputs: **sword** $d$ and $n$, with $d$ constant and $d \neq 0$.
**udword** $m$;
**int** $\ell$, $sh_{\text{post}}$;
$(m,\ sh_{\text{post}},\ \ell) = \text{CHOOSE\_MULTIPLIER}(|d|,\ N-1)$;
**if** $|d| = 1$ **then**
   Issue $q = d$;
**else if** $|d| = 2^{\ell}$ **then**
   Issue $q = \text{SRA}(n + \text{SRL}(\text{SRA}(n,\ \ell-1),\ N-\ell),\ \ell)$;
**else if** $m < 2^{N-1}$ **then**
   Issue $q = \text{SRA}(\text{MULSH}(m, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
**else**
   Issue $q = \text{SRA}(n + \text{MULSH}(m - 2^{N},\ n),\ sh_{\text{post}})$
             $- \text{XSIGN}(n)$;
   **Cmt.** Caution — $m - 2^{N}$ is negative.
**end if**

**if** $d < 0$ **then**
   Issue $q = -q$;
**end if**
```

Figure 5.2: Optimized code generation of signed $q = \text{TRUNC}(n/d)$ for constant $d \neq 0$

- 除法优化代码摘录

```
    public void DivOptimize(RegEnum dest, RegEnum left, int rightnum) {
        Divopt ans = ConstValue.ChooseMultiplier(Math.abs(rightnum));
        //System.out.println("here"+ans.get(0) + " "+ans.get(1)+"
"+ans.get(2));
        long m = ans.m_high;
        int sh_post = ans.sh_post;
        //MyError.errorat(m + sh_post+" haha",1174);
        if (Math.abs(rightnum) == 1) {
            genadd("move", dest, left);

        } else if (isPowerOfTwo(Math.abs(rightnum))) {
            int mi = (int) (Math.log(rightnum) / Math.log(2));
            genaddSraSrl("sra", dest, left, (mi - 1));
            genaddSraSrl("srl", dest, dest, (32 - mi));
            genadd("add", dest, dest, left);
            genaddSraSrl("sra", dest, dest, mi);


        } else if (m < Math.pow(2, 31)) {  // q = SRA(MULSH(m, n), shpost)
- XSIGN(n)

            genli(RegEnum.a3, m);
            genadd("mult", left, RegEnum.a3);
            genadd("mfhi", dest);
            genaddSraSrl("sra", dest, dest, sh_post);

            genaddSraSrl("slti", RegEnum.a3, left, 0);
            genadd("add", dest, dest, RegEnum.a3);


        } else {
            genli(RegEnum.a3, (int) (m - Math.pow(2, 32)));
            genadd("mult", left, RegEnum.a3);
            genadd("mfhi", dest);
            genadd("add", dest, dest, left);
            genaddSraSrl("sra", dest, dest, sh_post);

            genaddSraSrl("slti", RegEnum.a3, left, 0);
            genadd("add", dest, dest, RegEnum.a3);
        }

        if (rightnum < 0) {
            genadd("sub", dest, RegEnum.zero, dest);
        }
    }
```

1.3 模运算优化

- 利用上述除法优化的基础，进行取余数优化： 将a % b 翻译为 a - a / b * b。实现代码如下
- 总体思路便是利用了除法优化基础，进一步进行乘积减法运算，额外的如果模数为1的话直接li为0，减少模运算。

```
if (rightnum == 1) {
    genli(destreg, 0);
} else {
    DivOptimize(destreg, leftreg, rightnum);
    genli(RegEnum.fp, rightnum);
    genadd("mult", destreg, RegEnum.fp);
    genadd("mflo", RegEnum.fp);
    genadd("sub", destreg, leftreg, RegEnum.fp);
}
```

2 常量直取

常量直取在生成中间代码时初步实现，在建立符号表时遇到const常量无论是int还是数组都直接将其初值存储在符号表以及中间代码类之中，以便之后生成目标代码时直接取用。

- 代码实现如下，在遇到const int和const array时直接存储其值，以便后续生成

```
private Var parseIdent(ASTNode n) {
    //MyError.sout("In parseIdent " + n.getName() + " " + n.getKind());
    NodeKind kind = n.getKind();

    if (kind.equals(NodeKind.FUNC)) {   //函数调用！！！！！！！
        String funcname = n.getName();
        ASTNode rparams = n.getLeft();


        Symbol func =
MidCodeGenerate.lookGlobalTableFindsamename(funcname);


        if (rparams != null) {       //函数有参数则push
            for (int i = 0; i < rparams.getLeafs().size(); i++) {
                ASTNode para = rparams.getLeafs().get(i);

                Symbol fparami = func.getParams().get(i); //函数的第i个参数类
型

                int arraydimen = fparami.getDimension();

                if (fparami.isArray()) {       //如果是array类型的函数参数,传参
数的时候用
                    Var paraexp = parseParaArray(para, arraydimen);
//需返回array类型
                    createMidCode(MidType.Push, paraexp);

                } else {
                    Var paraexp = parseExp(para);       //正常的var类型exp
                    createMidCode(MidType.Push, paraexp);
                }
            }
        }

        //call
        MidCode ir = new MidCode(MidType.call, funcname);
        ir.setSymbol(func);
        ir4init(ir);

        //ret
        if (func.getFuncKind() != null &&
!func.getFuncKind().equals(FuncKind.VOID)) {
            Var tmpvar = getTmpVar();
            createMidCode(MidType.assign_ret, tmpvar);
            return tmpvar;
        }
        return null;


    } else if (kind.equals(NodeKind.INT) || kind.equals(NodeKind.CONSTINT))
{
        if (kind.equals(NodeKind.CONSTINT)) {     //优化 constint直接存储
            String name = n.getName();
            Symbol symbol = MidCodeGenerate.lookallTableFindsamename(name);
            assert symbol != null;
            int num = symbol.getNum();
            return new Var("num", num);

        }

        Var intvar = new Var("var", n.getName());
        intvar.setiskindofsymbolTrue();//设置成是符号表里面的东西
        return intvar;


    } else if (kind.equals(NodeKind.CONSTARRAY)) {
        //return parseArrayVisit(n);
        Var arrnnum = parseArrayVisit(n);
        Var tmp = getTmpVar();
        createMidCode(MidType.assign2, tmp, arrnnum);
        return tmp;
    } else if (kind.equals(NodeKind.ARRAY)) {
        //这里必须分两类
        //我们要把临时数组的访问最终聚合为临时变量

        Var arrnnum = parseArrayVisit(n);
        Var tmp = getTmpVar();
        createMidCode(MidType.assign2, tmp, arrnnum);
        return tmp;
    }
    MyError.errorat("Midgen", 801, "不知道ident是什么类型");
    return null;
}
```

- 在建立语法树的时候如果遇到数字之间的加减法，不生成中间变量而是直接进行运算得到最终数字,以下是代码实现

```
public int calcuValue() {
    if (OperDiction.hasOperator(opstring)) {
        switch (opstring) {
            case "+":
                if (right != null) {
                    return left.calcuValue() + right.calcuValue();
                }
                return left.calcuValue();
            case "-":
                if (right != null) {
                    return left.calcuValue() - right.calcuValue();
                }
                return -left.calcuValue();
            case "*":
                return left.calcuValue() * right.calcuValue();
            case "/":
                return left.calcuValue() / right.calcuValue();
            case "%":
                return left.calcuValue() % right.calcuValue();
            default:
                MyError.errorat("ASTnode",151);
                return 0;
        }
    }else if(type.equals(NodeType.Number)){
        return num;
    }else if(type.equals(NodeType.Ident)){
        if(kind.equals(NodeKind.FUNC)){
            //func 不再这里算，这里只能算一些简单的加减乘除
            MyError.errorat("ASTNODE",158,"这里不应该有这种情况，不能在编译时计算
函数");
        } else if
(kind.equals(NodeKind.INT)||kind.equals(NodeKind.CONSTINT)) {
            Symbol symbol = MidCodeGenerate.lookallTableFindsamename(name);
            return symbol.getNum();
        }else
if(kind.equals(NodeKind.ARRAY)||kind.equals(NodeKind.CONSTARRAY)){
            Symbol symbol = MidCodeGenerate.lookallTableFindsamename(name);
            if(symbol != null){
                if(right != null){ //二维
                    int dimen2 = symbol.getDimen2();
                    int index1 = left.calcuValue();
                    int index2 = right.calcuValue();
                    int index = index1*dimen2 + index2;
                    return symbol.getArrayValue().get(index);
                }else {//一维数组
                    int index = left.calcuValue();
                    return symbol.getArrayValue().get(index);
                }
            }else {
                MyError.errorat("ASTNODE",179,"符号表没找到，但理应之前定义");
            }
        }else {
            MyError.errorat("ASTNODE",178);
        }

    }else {
        MyError.errorat("ASTNODE",182);
    }
    return -92929299;
}
```

在最终生成目标mips码时，将表达式中所有const常量直接用常量的值替换，并且我们将表达式中数字和数字之间的直接运算已经在建立语法树时运算完毕，也直接用数值替换，这样可以减少很多中间临时变量的使用。以上时所有的常量直取的相关优化。