代码优化文档

一、中端优化

1.死代码删除

1.1无用函数删除

有些函数返回值为`void`类型，并且在执行过程中没有打印、修改全局变量、修改参数数组和调用其他函数的情况，对于这些函数我们将其调用部分直接删除。

具体实现为：

```
public void optimizeFunc() {
        initTable();
        TableItem funcItem = null;
        boolean isDeadFunc = true;
        boolean isInFunc = false;
        for (int i = 0; i < optimizedIRList.size(); i++) {
            IRCode irCode = optimizedIRList.get(i);
            IROperator operator = irCode.getOperator();
            if (operator == IROperator.DEF) {
                curTable.addItem(irCode.getDefItem());
            } else if (operator == IROperator.func_begin) {
                isDeadFunc = true;
                funcItem = irCode.getFuncItem();
                curTable.addItem(funcItem);
                if (funcItem.getType().equals("int")) {
                    isDeadFunc = false;
                }
                isInFunc = true;
                addLevel();
            } else if (operator == IROperator.block_begin) {
                addLevel();
            } else if (operator == IROperator.block_end) {
                deleteLevel();
            } else if (operator == IROperator.PRINT_STR || operator ==
IROperator.PRINT_INT) {
                if (isInFunc) {
                    isDeadFunc = false;
                }
            } else if (operator == IROperator.ASSIGN || operator ==
IROperator.GETINT) {
                String desName = irCode.getResultIdent();
                TableItem desItem = findVarParam(desName);
                if (isInFunc && desItem != null &&
                        (desItem.isGlobal() ||
(desItem.getKind().equals("param") && desItem.isArray()))) {//全局变量或者数组参数
                    isDeadFunc = false;
                }
            } else if (operator == IROperator.CALL) {
                TableItem callFuncItem = irCode.getFuncItem();
```

```
                if (funcItem != null &&
!callFuncItem.getName().equals(funcItem.getName())) {
                    isDeadFunc = false;
                }
            } else if (operator == IROperator.func_end) {
                if (isDeadFunc) {
                    int originSize = optimizedIRList.size();
                    //删除所有调用该函数的语句
                    deleteDeadFunc(funcItem);
                    deleteLevel();
                    //重新运行该方法
                    if (originSize != optimizedIRList.size()) {
                        optimizeFunc();
                        return;
                    }
                }
                isInFunc = false;
                funcItem = null;
            }
        }
    }
```

1.2无用定义点删除

无用定义点及相关语句的删除主要分为两部分：

- 块内无用定义点和相关语句删除
- 非活跃变量无用定义点删除

1.2.1块内无用定义点删除

这一步的删除操作基于2中对基本块的划分以及到达定义数据流的分析。

主要思想是在块内如果对于某一个**局部非数组变量**进行连续多次定义(>=2)，且存在后一次定义之前完全未使用前一次定义的情况，则前一次定义为死代码，可以直接删除。

例如对于以下代码：

```
    int i = 2,j = 5;
    const int a1 = 1, a2 = 2;
    i = getint();
    j = getint();
```

其中i和j的第一次定义完全没有用到，所以直接优化掉即可。

第一次块内无用定义点删除可以在常量传播优化完成之后进行。

具体实现如下：

```
public boolean deleteDeadDefInBlock(HashMap<String, DefPoint> defPoints,
HashMap<String, BasicBlock> blockMap, int start) {
        boolean unFinished = false;
        for (int i = start; i < optimizedIRList.size(); i++) {
            IRCode irCode = optimizedIRList.get(i);
            IROperator operator = irCode.getOperator();
```

```
            if (operator == IROperator.func_end || operator ==
IROperator.main_end) {
                break;
            }
            boolean isDefPoint = isInPointMap(defPoints, irCode);
            if (isDefPoint) {
                DefPoint defPoint = getDefPointByIRCode(defPoints, irCode);
                TableItem defItem = defPoint.getDefItem();
                BasicBlock curBlock = getBlockByIRNum(blockMap, i);
                int blockEnd = curBlock.getEnd();
                boolean isDeadDef = true;
                int nextDefPointNum = -1;
                for (int j = i + 1; j <= blockEnd; j++) {
                    //寻找块内该变量的下一个定义点
                    if (getSameItemDefPoint(optimizedIRList.get(j), defPoints,
defItem) != null) {
                        nextDefPointNum = j;
                        break;
                    }
                }
                if (nextDefPointNum == -1) {
                    //块内该变量没有下一个定义点，保留该定义点
                    isDeadDef = false;
                } else {
                    //块内该变量有下一个定义点，判断该定义点的值是否被使用
                    for (int j = i + 1; j <= nextDefPointNum; j++) {
                        if (optimizedIRList.get(j).getOpIdent1() != null &&
optimizedIRList.get(j).getOpIdent1().equals(defItem.getName())) {
                            isDeadDef = false;
                            break;
                        }
                        if (optimizedIRList.get(j).getOpIdent2() != null &&
optimizedIRList.get(j).getOpIdent2().equals(defItem.getName())) {
                            isDeadDef = false;
                            break;
                        }
                    }
                }
                if (isDeadDef) {
                    irCode.setDead();
                }
            }
        }
        unFinished = killDeadIrCode();
        return unFinished;
    }
```

1.2.2块间无用定义点删除

该优化主要基于**活跃变量分析**进行，如果某个**局部非数组变量**的定义在整个数据流中都不再活跃，那么直接将对应的定义点以及相关中间代码删除。

最终实现是在块内无用定义点删除的基础上增量开发，具体实现如下：

```
public boolean deleteDeadDef(HashMap<String, DefPoint> defPoints,
HashMap<String, BasicBlock> blockMap, int start) {
        boolean unFinished = false;
        for (int i = start; i < optimizedIRList.size(); i++) {
            IRCode irCode = optimizedIRList.get(i);
            IROperator operator = irCode.getOperator();
            if (operator == IROperator.func_end || operator ==
IROperator.main_end) {
                break;
            }
            boolean isDefPoint = isInPointMap(defPoints, irCode);
            if (isDefPoint) {
                DefPoint defPoint = getDefPointByIRCode(defPoints, irCode);
                TableItem defItem = defPoint.getDefItem();
                BasicBlock curBlock = getBlockByIRNum(blockMap, i);
                int blockEnd = curBlock.getEnd();
                boolean isDeadDef = true;
                int nextDefPointNum = -1;
                for (int j = i + 1; j <= blockEnd; j++) {
                    //寻找块内该变量的下一个定义点
                    if (getSameItemDefPoint(optimizedIRList.get(j), defPoints,
defItem) != null) {
                        nextDefPointNum = j;
                        break;
                    }
                }
                if (nextDefPointNum == -1) {
                    //块内该变量没有下一个定义点，判断该定义点的值是否在该块内使用
                    isDeadDef = isDeadDef(i, defItem, blockEnd, isDeadDef);
                    //判断是否在活跃变量的out集合中
                    if (curBlock.getOutActiveVarList().contains(defItem)) {
                        isDeadDef = false;
                    }
                } else {
                    //块内该变量有下一个定义点，判断该定义点的值是否被使用
                    isDeadDef = isDeadDef(i, defItem, nextDefPointNum,
isDeadDef);
                }
                if (isDeadDef) {
                    irCode.setDead();
                }
            }
        }
        unFinished = killDeadIrCode();
        return unFinished;
    }
```

活跃变量分析的具体实现：

```
public void setInOutActiveVarList(HashMap<String, BasicBlock> blockMap) {
        boolean unFinished = true;
        while (unFinished) {
            unFinished = false;
            for (BasicBlock curBlock : blockMap.values()) {
                ArrayList<BasicBlock> nextBlocks = curBlock.getNextBlocks();
```

```
                ArrayList<TableItem> inActiveVarList =
curBlock.getInActiveVarList();
                ArrayList<TableItem> outActiveVarList =
curBlock.getOutActiveVarList();
                //求解活跃变量的out集合
                for (BasicBlock nextBlock : nextBlocks) {
                    for (TableItem outActiveVar :
nextBlock.getInActiveVarList()) {
                        if (!outActiveVarList.contains(outActiveVar)) {
                            outActiveVarList.add(outActiveVar);
                            unFinished = true;
                        }
                    }
                }
                //求解out - def
                ArrayList<TableItem> outSubDefVarList = new ArrayList<>
(outActiveVarList);
                outSubDefVarList.removeAll(curBlock.getDefVarList());
                //求解in：use 并 (out - def)
                ArrayList<TableItem> useVarList = curBlock.getUseVarList();
                for (TableItem useVar : useVarList) {
                    if (!inActiveVarList.contains(useVar)) {
                        inActiveVarList.add(useVar);
                        unFinished = true;
                    }
                }
                for (TableItem outSubDefVar : outSubDefVarList) {
                    if (!inActiveVarList.contains(outSubDefVar)) {
                        inActiveVarList.add(outSubDefVar);
                        unFinished = true;
                    }
                }
            }
        }
    }
```

1.3无用DEF语句删除

由于某些变量或者常量的值在前面的优化中已经全部赋值到了相应的操作指令中，导致这些变量或常量的定义语句和对应的赋值语句失效了，对于这些语句我们可以直接删除（否则可能导致多次的无效的**内存分配和内存操作**，因此在进行完上述优化之后我又进行了一次死定义删除的优化，结果对于测试点1效果显著。

优化的主要思路是对于某个DEF定义点，如果后面的所有语句都没有用到该变量或常量的值，那么可以直接删除该DEF语句，同时删除所有对该变量进行赋值的语句。

具体的优化方法如下：

```
public boolean deleteDeadDef(HashMap<String, DefPoint> defPoints,
HashMap<String, BasicBlock> blockMap, int start) {
        boolean unFinished = false;
        for (int i = start; i < optimizedIRList.size(); i++) {
            IRCode irCode = optimizedIRList.get(i);
            IROperator operator = irCode.getOperator();
```

```
            if (operator == IROperator.func_end || operator ==
IROperator.main_end) {
                break;
            }
            boolean isDefPoint = isInPointMap(defPoints, irCode);
            if (isDefPoint) {
                DefPoint defPoint = getDefPointByIRCode(defPoints, irCode);
                TableItem defItem = defPoint.getDefItem();
                BasicBlock curBlock = getBlockByIRNum(blockMap, i);
                int blockEnd = curBlock.getEnd();
                boolean isDeadDef = true;
                int nextDefPointNum = -1;
                if (irCode.getOperator() == IROperator.DEF &&
irCode.getDefItem().getKind().equals("param")) {
                    //参数豁免
                    isDeadDef = false;
                }
                for (int j = i + 1; j <= blockEnd; j++) {
                    //寻找块内该变量的下一个定义点
                    if (getSameItemDefPoint(optimizedIRList.get(j), defPoints,
defItem) != null) {
                        nextDefPointNum = j;
                        break;
                    }
                }
                if (nextDefPointNum == -1) {
                    //块内该变量没有下一个定义点，判断该定义点的值是否在该块内使用
                    isDeadDef = isDeadDef(i, defItem, blockEnd, isDeadDef);
                    //判断是否在活跃变量的out集合中
                    if (curBlock.getOutActiveVarList().contains(defItem)) {
                        isDeadDef = false;
                    }
                } else {
                    //块内该变量有下一个定义点，判断该定义点的值是否被使用
                    isDeadDef = isDeadDef(i, defItem, nextDefPointNum,
isDeadDef);
                }
                if (isDeadDef) {
                    irCode.setDead();
                }
            }
        }
        unFinished = killDeadIrCode();
        return unFinished;
    }
```

优化过程中出现的问题:

由于开始对无用变量的定义是后面没有被使用过，但是对于函数的参数而言，如果该参数是数组参数的话，那么对它的赋值也会产生实际影响，除此之外，函数参数的定义还会影响到参数压栈和出栈的指令，因此对于函数参数的定义我们应该选择豁免（即在无用DEF删除的过程中，应该对**函数的参数不予考虑**）。

2.常量传播

利用到达定义数据流分析，当某个计算中出现的变量的**所有有效定义点的值均为同一个常数**时，可以直接将该变量用常数替换，以减少运算量。为了实现这一目标，首先需要将研究对象（一般一个函数体）切分为基本块，然后对这些基本块进行到达定义数据流分析（由于全局变量和局部数组变量存在被其他函数修改的风险，因而该分析**仅针对局部非数组变量进行**）

数据流方程：$out = gen \cup (in - kill)$

为了实现该优化，我设计了表示基本块的类`BasicBlock`和表示定义点的类`DefPoint`，具体的数据类型如下：

BasicBlock：

```
private String blockName;
    private int start;
    private int end;
    private ArrayList<IRCode> blockIRCodes = new ArrayList<>();
    private ArrayList<BasicBlock> nextBlocks = new ArrayList<>();
    private ArrayList<BasicBlock> preBlocks = new ArrayList<>();
    private ArrayList<DefPoint> genPoints = new ArrayList<>();
    private ArrayList<DefPoint> killPoints = new ArrayList<>();
    private ArrayList<DefPoint> inDefPoints = new ArrayList<>();
    private ArrayList<DefPoint> outDefPoints = new ArrayList<>();

    public BasicBlock(String blockName) {
        this.blockName = blockName;
    }
```

DefPoint：

```
private String pointName;
    private IRCode irCode;
    private TableItem defItem;
    private BasicBlock block;

    public DefPoint(String pointName, IRCode irCode, TableItem defItem,
BasicBlock block) {
        this.pointName = pointName;
        this.irCode = irCode;
        this.defItem = defItem;
        this.block = block;
    }
```

常量传播优化主要分为以下几个阶段：

- 基本块划分
- gen、kill集合生成
- 迭代计算in、out集合
- 根据每个基本块的in集合和基本块内的定义点，判断某处使用的变量是否可以替换为常量

实现逻辑如下：

```
public void constSpread() {
        boolean unFinished = true;
```

```
        while (unFinished) {
            initTable();
            TableItem funcItem = null;
            for (int i = 0; i < optimizedIRList.size(); i++) {
                IRCode irCode = optimizedIRList.get(i);
                IROperator operator = irCode.getOperator();
                if (operator == IROperator.DEF) {
                    curTable.addItem(irCode.getDefItem());
                } else if (operator == IROperator.func_begin || operator ==
IROperator.main_begin) {
                    HashMap<String, BasicBlock> blockMap = new HashMap<>();//初始
化基本块
                    HashMap<String, DefPoint> defPoints = new HashMap<>();//初始
化定义点列表
                    funcItem = irCode.getFuncItem();
                    curTable.addItem(funcItem);
                    addLevel();
                    splitBasicBlocks(blockMap, i);//划分基本块

                    backupTable();//备份符号表
                    setDefPoints(defPoints, blockMap, i);//设置定义点，即gen集合
                    rollBackTable();//恢复符号表

                    setKillPoints(defPoints, blockMap);//设置kill集合

                    setInOutDefPoints(defPoints, blockMap);//迭代求解in、out集合

                    backupTable();//备份符号表
                    unFinished = constSpreadInBlock(defPoints, blockMap, i);//常
量传播优化
                    rollBackTable();//恢复符号表

                    if (unFinished) {
                        break;
                    }
                } else if (operator == IROperator.block_begin) {
                    addLevel();
                } else if (operator == IROperator.block_end) {
                    deleteLevel();
                } else if (operator == IROperator.func_end || operator ==
IROperator.main_end) {
                    deleteLevel();
                }
            }
        }
    }
```

在实现过程中，我遇到的问题主要有：

- 划分基本块并且生成数据流关系的时候，没有考虑跳转到后面块的情况，解决方案是先划分好基本块，然后再完善一次数据流的关系。
- 之前设计的mips生成器没有完全适配操作数1和操作数2均为常数的情况，具体表现在函数参数和打印参数等地方，对此进行了一定的适配。
- 基本块划分的方法考虑不够细致，开始将return作为一个划分标准，但是没有考虑void函数可以没有return语句，导致该类函数确实最后一个基本块。

3.计算优化

在各种优化过后，可能会出现很多计算中一个操作符为常数的情况，对于这些情况，我们可以对**常数是0和1的情况**针对性处理。

例如，对于`MUL, #t0, 1, #t1`的情况，可以直接将后续的所有`#t1`替换为`#t0`进行计算，这样可以处理掉很多不必要的多余运算。

具体实现如下：

```
public boolean optimizeCalculate() {
        boolean unFinished = true;
        for (IRCode irCode : optimizedIRList) {
            IROperator operator = irCode.getOperator();
            boolean canDelete = false;
            if (irCode.getResultIdent() != null &&
irCode.getResultIdent().startsWith("#")) {
                if (irCode.op1IsNum() && irCode.getOpIdent2() != null &&
irCode.getOpIdent2().startsWith("#")) {
                    int opNum = irCode.getOpNum1();
                    String opIdent = irCode.getOpIdent2();
                    String resultIdent = irCode.getResultIdent();
                    if (opNum == 1) {
                        canDelete = calculateSpread1(operator, opIdent,
resultIdent, true);
                    } else if (opNum == 0) {
                        canDelete = calculateSpread0(operator, opIdent,
resultIdent, true);
                    }
                } else if (irCode.op2IsNum() && irCode.getOpIdent1() != null &&
irCode.getOpIdent1().startsWith("#")) {
                    int opNum = irCode.getOpNum2();
                    String opIdent = irCode.getOpIdent1();
                    String resultIdent = irCode.getResultIdent();
                    if (opNum == 1) {
                        canDelete = calculateSpread1(operator, opIdent,
resultIdent, false);
                    } else if (opNum == 0) {
                        canDelete = calculateSpread0(operator, opIdent,
resultIdent, false);
                    }
                }
            }
            if (canDelete) {
                irCode.setDead();
            }
        }
        unFinished = killDeadIrCode();
        return unFinished;
    }
```

其中，对于常数是0和1的情况处理逻辑如下：

常数是0：

```
public boolean calculateSpread0(IROperator operator, String opIdent, String
resultIdent, boolean isOp1Num) {
        boolean canSubstitute = false;
        String substituteIdent = null;
        int substituteNum = 0;
        boolean substituteIsNum = false;
        boolean finish = false;
        switch (operator) {
            case ADD:
                substituteIdent = opIdent;
                canSubstitute = true;
                break;
            case SUB:
                if (!isOp1Num) {
                    //只有op2是数字才能替换
                    substituteIdent = opIdent;
                    canSubstitute = true;
                }
                break;
            case MUL, AND:
                canSubstitute = true;
                substituteIsNum = true;
                break;
            case DIV, MOD:
                if (isOp1Num) {
                    //只有被除数是0可以替换
                    substituteIsNum = true;
                    canSubstitute = true;
                }
                break;
            default:
                break;
        }
        return substituteCalculate(resultIdent, canSubstitute, substituteIdent,
substituteNum, substituteIsNum);
    }
```

常数是1:

```
public boolean calculateSpread1(IROperator operator, String opIdent, String
resultIdent, boolean isOp1Num) {
        boolean canSubstitute = false;
        String substituteIdent = null;
        int substituteNum = 0;
        boolean substituteIsNum = false;
        boolean finish = false;
        switch (operator) {
            case MUL:
                substituteIdent = opIdent;
                canSubstitute = true;
                break;
            case DIV:
                if (!isOp1Num) {
                    //只有op2是1才能替换
                    substituteIdent = opIdent;
```

```
                    canSubstitute = true;
                }
                break;
            case MOD:
                if (!isOp1Num) {
                    //只有op2是1才能替换
                    substituteNum = 0;
                    substituteIsNum = true;
                    canSubstitute = true;
                }
                break;
            case OR:
                substituteNum = 1;
                substituteIsNum = true;
                canSubstitute = true;
                break;
            default:
                break;
        }
        return substituteCalculate(resultIdent, canSubstitute, substituteIdent,
substituteNum, substituteIsNum);
    }
```

4.循环优化

在之前的循环实现逻辑中，每次在循环结束更新完循环变量之后，都会跳转到`for_checkin`的部分进行判断，这样无形之中增加了很多跳转的工作。实际上，我们可以采用`do-while`循环的思想，在第一次进入循环之前判断是否能进入循环，之后每次循环结束后继续判断是否需要回到循环开始的位置，而不是直接跳回循环快之前再进行判断。这样以来可以节省掉每次循环结束后的`JMP`指令，使得中间代码更加简洁。

例如：

```
bb1:
addi $1, $1, 1
beq $1, $2, bb3
j bb2
....
bb2:
...    #code1
j bb1
bb3:
....
```

可以优化为：

```
bb1:
...     #code1
addi $1, $1, 1
bne $1, $2, bb1
bb3:
...
```

优化后的循环逻辑为：

1.执行循环变量赋值语句 `ForStmt1`

2.解析条件表达式 `Cond`

3.生成标签 `for_in`

4.生成 `Stmt` 的四元式

5.生成标签 `for_update`

6.执行循环变量更新语句 `ForStmt2`

7.生成标签 `for_checkin`

8.解析条件表达式 `Cond`

9.生成标签 `for_out`

5.公共子表达式删除（DAG）

在一些基本块中，有可能会出现不同变量的值相同的情况，例如：

```
c <- a + b
d <- c - b
e <- a + b
f <- e - b
```

其中c和e的取值显然相同，可以直接优化为：

```
c <- a + b
d <- c - b
f <- c - b
```

这时又可以发现d和f的取值相同，可以继续优化为：

```
c <- a + b
d <- c - b
```

当然，在进行公共子表达式消除的过程中，由于我们改变后的结果只能保证程序在该基本块内部的正确性，无法保证跨基本快或者跨函数等情况下程序是否正确，因此我们进行公共子表达式消除的对象依然只能是**局部非数组变量。**

至于DAG图的实现，我们采用 `DAGMap` 类来保存每个基本块的DAG图，将**局部变量或数字**作为叶节点，运算符作为中间节点，同时运算结果也保存在中间节点当中，后续如果出现某个局部变量A的运算结果落在已有的某个中间节点（包含局部变量B）中，那么考虑两种情况。

第一种，该**局部非数组变量跨基本块不活跃。**如果该基本块中**后续没有对局部变量B重新赋值的地方**，那么直接删除对A赋值的语句，将该基本块中后续对A的读取都转换成对B的读取。如果**后续有对B重新赋值的地方**，那么本次对A的赋值变成**将B的值赋给A**，同时将下一次给B赋值之前对A的读取都转换成对B的读取（目的是暴露更多可优化的机会）。

第二种，该**局部非数组变量跨基本块活跃。**此时应该采取和上一种情况中第二种解决方案。

具体逻辑如下：

```
public boolean dagOptimize() {
```

```
        boolean unFinished = false;
        initTable();
        for (int i = 0; i < optimizedIRList.size(); i++) {
            IRCode irCode = optimizedIRList.get(i);
            IROperator operator = irCode.getOperator();
            if (operator == IROperator.DEF) {
                curTable.addItem(irCode.getDefItem());
            } else if (operator == IROperator.func_begin || operator ==
IROperator.main_begin) {
                addLevel();
                HashMap<String, BasicBlock> blockMap = new HashMap<>();//初始化基
本块
                splitBasicBlocks(blockMap, i);//划分基本块

                backupTable();//备份符号表
                setDefUseVarList(blockMap, i);//设置活跃变量Use Def集合
                rollBackTable();//恢复符号表
                setInOutActiveVarList(blockMap);//迭代求解活跃变量in、out集合

                backupTable();//备份符号表
                HashMap<String, DefPoint> defPoints = new HashMap<>();
                setDefPoints(defPoints, blockMap, i);//设置定义点，即gen集合
                rollBackTable();//恢复符号表

                ArrayList<TableItem> allGlobalVarList = new ArrayList<>();//统计函
数体内所有的跨基本块活跃变量
                for (BasicBlock block : blockMap.values()) {
                    ArrayList<TableItem> inActiveVarList =
block.getInActiveVarList();
                    for (TableItem item : inActiveVarList) {
                        if (!allGlobalVarList.contains(item)) {
                            allGlobalVarList.add(item);
                        }
                    }
                }

                //分析每个基本块内部的公共子表达式
                backupTable();
                unFinished = dagAnalyse(blockMap, defPoints, allGlobalVarList,
i);
                rollBackTable();
                if (unFinished) {
                    break;
                }
            } else if (operator == IROperator.block_begin) {
                addLevel();
            } else if (operator == IROperator.block_end) {
                deleteLevel();
            } else if (operator == IROperator.func_end || operator ==
IROperator.main_end) {
                deleteLevel();
            }
        }
        return unFinished;
    }
```

二、后端优化

1、指令选择

前言：为了更方便的实现更多功能，以及更加方便翻译高级语言，Mars给我们提供了充分的伪指令使用，例如 `subi $t1, $t2, 100`、`move $1, $2`、`ble $t1, $t2, label` 等，能够缩短指令的条数，增加代码的可读性。但是由于Mars的局限性，以及体系结构的特性，很多时候伪指令虽然表面上降低了指令的条数，但是实际上反而会使FinalCycle增加，在非循环语句当中，这样的问题当然可以忽略，而在循环次数很多的循环体当中，哪怕每个语句多翻译了一条代码都会严重影响性能，而subi等语句其实是十分常见的计算语句，在循环当中也可能高频出现。另一方面，在进行图着色寄存器分配时，伪指令可能会隐蔽地更改寄存器的值，导致数据流分析错误，可能会产生一些很难发现的bug。所以不使用Mars低效的伪指令，转而自己封装一套翻译机制是提升性能的有效方法。

1.1分支指令

beq

beq在进行寄存器和立即数的比较时，会先把立即数加 `$0` 赋值给新的寄存器，而我们程序中大多数跳转又都是和0进行比较，因此可以考虑把立即数0都换成 `$0`，即使用 `beqz` 指令。

bne,bgt,bge,blt,ble指令也都可以按照上述方法进行优化。

subi

subi指令在翻译时会被翻译成一条和 `$0` 相加的addi指令和一条sub指令，但其实我们只需要翻译为一条addi指令就可以满足条件。

例如：

```
subi $t1, $t0, -1
```

可以翻译为：

```
addi $t1, $t0, 1
```

lw、sw

lw、sw指令在操作一个标识符时，会首先计算标识符和立即数的和，再加上$0的值，因此如果立即数为0时，我们可以直接省略该立即数，减少目标代码量。

2.乘法优化

乘法优化主要分为两部分：

- 2的整数幂优化
- 2的整数幂周围的数

优化方法：

```
public void optimizeMulRegImm(String opReg, int opNum, String desReg) {
        if (!optimize) {
            String immReg = getReg(String.valueOf(opNum));
```

```
            addMIPSCode(new MIPSCode(MIPSOperator.li, opNum, null, immReg));//常
数opNum赋值给寄存器
            addMIPSCode(new MIPSCode(MIPSOperator.mult, immReg, opReg, null));
            addMIPSCode(new MIPSCode(MIPSOperator.mflo, null, null, desReg));
            freeReg(immReg);//释放常数寄存器
        } else {
            boolean isNegate = opNum < 0;
            if (isNegate) {
                opNum = -opNum;//后面都考虑非负整数
            }
            if (opNum == 0) {
                addMIPSCode(new MIPSCode(MIPSOperator.li, 0, null, desReg));
            } else if (opNum == 1) {
                addMIPSCode(new MIPSCode(MIPSOperator.move, opReg, null,
desReg));
                if (isNegate) {
                    addMIPSCode(new MIPSCode(MIPSOperator.sub, "$0", desReg,
desReg));
                }
            } else if (isPowerOfTwo(opNum)) {
                int shift = (int) (Math.log(opNum) / Math.log(2));
                addMIPSCode(new MIPSCode(MIPSOperator.sll, opReg, shift,
desReg));
                if (isNegate) {
                    addMIPSCode(new MIPSCode(MIPSOperator.sub, "$0", desReg,
desReg));
                }
            } else if (nearPowerOfTwo(opNum) != 0) {
                int off = nearPowerOfTwo(opNum);
                int shift = (int) (Math.log(opNum + off) / Math.log(2));
                addMIPSCode(new MIPSCode(MIPSOperator.sll, opReg, shift,
desReg));
                for (int i = 0; i < Math.abs(off); i++) {
                    addMIPSCode(new MIPSCode(off > 0 ? MIPSOperator.sub :
MIPSOperator.addu, desReg, opReg, desReg));
                }
                if (isNegate) {
                    addMIPSCode(new MIPSCode(MIPSOperator.sub, "$0", desReg,
desReg));
                }
            } else {
                if (isNegate) {
                    opNum = -opNum;//恢复操作数
                }
                String immReg = getReg(String.valueOf(opNum));
                addMIPSCode(new MIPSCode(MIPSOperator.li, opNum, null,
immReg));//常数opNum赋值给寄存器
                addMIPSCode(new MIPSCode(MIPSOperator.mult, immReg, opReg,
null));
                addMIPSCode(new MIPSCode(MIPSOperator.mflo, null, null,
desReg));
                freeReg(immReg);//释放常数寄存器
            }
        }
    }
```

3.除法优化

除法优化的思路为将除法转换为乘法和移位操作。

优化方法：

```
public void optimizeDiv(String opReg, int divisor, String desReg) {
        if (!optimize) {
            String immReg = getReg(String.valueOf(divisor));
            addMIPSCode(new MIPSCode(MIPSOperator.li, divisor, null, immReg));//
常数opNum赋值给寄存器
            addMIPSCode(new MIPSCode(MIPSOperator.div, opReg, immReg, null));
            addMIPSCode(new MIPSCode(MIPSOperator.mflo, null, null, desReg));
            freeReg(immReg);//释放常数寄存器
        } else {
            boolean isNegate = divisor < 0;
            if (isNegate) {
                divisor = -divisor;//后面都考虑非负整数
            }
            if (divisor == 1) {
                addMIPSCode(new MIPSCode(MIPSOperator.move, opReg, null,
desReg));
                if (isNegate) {
                    addMIPSCode(new MIPSCode(MIPSOperator.sub, "$0", desReg,
desReg));
                }
            } else if (isPowerOfTwo(divisor)) {
                int shift = (int) (Math.log(divisor) / Math.log(2));
                String tempReg = getReg("temp");
                addMIPSCode(new MIPSCode(MIPSOperator.sll, opReg, 32 - shift,
desReg));
                addMIPSCode(new MIPSCode(MIPSOperator.sltu, "$0", desReg,
desReg));
                addMIPSCode(new MIPSCode(MIPSOperator.slt, opReg, "$0",
tempReg));//被除数为负数置1
                addMIPSCode(new MIPSCode(MIPSOperator.and, tempReg, desReg,
tempReg));
                addMIPSCode(new MIPSCode(MIPSOperator.sra, opReg, shift,
desReg));
                addMIPSCode(new MIPSCode(MIPSOperator.add, desReg, tempReg,
desReg));
                freeReg(tempReg);
                if (isNegate) {
                    addMIPSCode(new MIPSCode(MIPSOperator.sub, "$0", desReg,
desReg));
                }
            } else {
                long multiplier = chooseMultiplier(divisor, 32);
                if (multiplier != -1 && multiplier < Math.pow(2, 32) +
Math.pow(2, 31)) {//能找到multiplier
                    if (multiplier < Math.pow(2, 31)) {//multiplier不会溢出，且在
int范围内
                        int l = (int) Math.floor(Math.log(multiplier * divisor)
/ Math.log(2)) - 32;
```

```
                        String multiplierReg =
getReg(String.valueOf(multiplier));
                        addMIPSCode(new MIPSCode(MIPSOperator.li, (int)
multiplier, null, multiplierReg));
                        addMIPSCode(new MIPSCode(MIPSOperator.mult, opReg,
multiplierReg, null));
                        addMIPSCode(new MIPSCode(MIPSOperator.mfhi, null, null,
desReg));
                        addMIPSCode(new MIPSCode(MIPSOperator.sra, desReg, l,
desReg));

                        addMIPSCode(new MIPSCode(MIPSOperator.slt, opReg, "$0",
multiplierReg));//被除数为负数置1
                        addMIPSCode(new MIPSCode(MIPSOperator.add, desReg,
multiplierReg, desReg));
                        freeReg(multiplierReg);//释放常数寄存器
                    } else {
                        int l = (int) Math.floor(Math.log(multiplier * divisor)
/ Math.log(2)) - 32;
                        String multiplierReg =
getReg(String.valueOf(multiplier));
                        addMIPSCode(new MIPSCode(MIPSOperator.li, (int)
(multiplier - Math.pow(2, 32)), null, multiplierReg));
                        addMIPSCode(new MIPSCode(MIPSOperator.mult, opReg,
multiplierReg, null));
                        addMIPSCode(new MIPSCode(MIPSOperator.mfhi, null, null,
desReg));
                        addMIPSCode(new MIPSCode(MIPSOperator.add, desReg,
opReg, desReg));
                        addMIPSCode(new MIPSCode(MIPSOperator.sra, desReg, l,
desReg));

                        addMIPSCode(new MIPSCode(MIPSOperator.slt, opReg, "$0",
multiplierReg));//被除数为负数置1
                        addMIPSCode(new MIPSCode(MIPSOperator.add, desReg,
multiplierReg, desReg));
                        freeReg(multiplierReg);//释放常数寄存器
                    }
                } else {//无法进行优化
                    String immReg = getReg(String.valueOf(divisor));
                    addMIPSCode(new MIPSCode(MIPSOperator.li, divisor, null,
immReg));//常数opNum赋值给寄存器
                    addMIPSCode(new MIPSCode(MIPSOperator.div, opReg, immReg,
null));
                    addMIPSCode(new MIPSCode(MIPSOperator.mflo, null, null,
desReg));
                    freeReg(immReg);//释放常数寄存器
                }
                if (isNegate) {
                    addMIPSCode(new MIPSCode(MIPSOperator.sub, "$0", desReg,
desReg));
                }
            }
        }
    }
```

4.取模优化

将取模运算优化成 a - a / b * b

```
public void optimizeMod(String opReg, int opNum, String desReg) {//取模运算优化为a
- a / b * b，从而运用前面除法的优化方法
        if (!optimize) {
            String immReg = getReg(String.valueOf(opNum));
            addMIPSCode(new MIPSCode(MIPSOperator.li, curIRCode.getOpNum2(),
null, immReg));//常数赋值给寄存器
            addMIPSCode(new MIPSCode(MIPSOperator.div, opReg, immReg, null));
            addMIPSCode(new MIPSCode(MIPSOperator.mfhi, null, null, desReg));
            freeReg(immReg);//释放常数寄存器
        } else {
            boolean isNegate = opNum < 0;
            if (isNegate) {
                opNum = -opNum;//后面都考虑非负整数,mod符号与除数无关
            }
            if (opNum == 1) {
                addMIPSCode(new MIPSCode(MIPSOperator.li, 0, null, desReg));
            } else {
                optimizeDiv(opReg, opNum, desReg);
                String immReg = getReg(String.valueOf(opNum));
                addMIPSCode(new MIPSCode(MIPSOperator.li, opNum, null, immReg));
                addMIPSCode(new MIPSCode(MIPSOperator.mult, desReg, immReg,
null));
                addMIPSCode(new MIPSCode(MIPSOperator.mflo, null, null,
desReg));
                addMIPSCode(new MIPSCode(MIPSOperator.sub, opReg, desReg,
desReg));
                freeReg(immReg);//释放常数寄存器
            }

        }
    }
```

5.全局寄存器分配

由于在之前生成目标代码的过程中，我们对所有非临时变量的操作都通过访存进行，没有考虑全局寄存器的分配问题，导致了大量多余的访存操作。现在，我计划使用图着色的方法为所有**局部非数组变量**进行全局寄存器`$s0-$s7`的分配，分配完成后，在该函数体内，**该变量与该寄存器捆绑**，所有对于该变量的赋值操作（ASSIGN，GETINT）和读取该变量的操作（ASSIGN）都将转换为对该寄存器进行相应操作。**变量开始与全局寄存器捆绑的时机是进入函数体时，释放全局寄存器的时机是退出函数体时。**

为了实现该优化，我们需要在中间代码优化的阶段维护一个**全局寄存器池**，并且在变量的`TableItem`中添加全局寄存器属性。之后，根据活跃变量的数据流分析生成冲突图，然后再按照图着色算法从全局寄存器中给全局变量分配寄存器，分配完成后，将分配结果存入对应的`TableItem`中。

在后端生成MIPS代码时，如果对某一**局部非数组变量**进行赋值或者读取，那么可以将对应的操作施加在它对应的全局寄存器上。

需要注意的是，使用全局寄存器后，在函数调用时需要额外对全局寄存器的值进行维护，维护的方法与临时寄存器不同，而是将寄存器中的值存入到相应变量在栈中的位置，调用完成以后再把它们读取回相应全局寄存器当中。

对后端的修改主要有：

- ASSIGN给临时变量赋值时，如果赋值操作符拥有全局寄存器，则使用该寄存器进行赋值
- ASSIGN给变量赋值时，如果被赋值操作符拥有全局寄存器，则将值赋给该寄存器
- GETINT给变量赋值时，如果被赋值操作符拥有全局寄存器，则将值赋给该寄存器
- 被调用函数给参数赋值时，如果被赋值参数拥有全局寄存器，则将值赋给该寄存器
- 调用函数时，对全局寄存器进行保护，将寄存器的值写入栈中变量所在位置；与此对应，函数调用结束后，将寄存器的值从栈中取出。
- 一个函数体结束时，释放全局寄存器堆

中端分配全局寄存器的实现：

```
public void optimizeGlobalReg() {
        initTable();
        for (int i = 0; i < optimizedIRList.size(); i++) {
            IRCode irCode = optimizedIRList.get(i);
            IROperator operator = irCode.getOperator();
            if (operator == IROperator.DEF) {
                curTable.addItem(irCode.getDefItem());
            } else if (operator == IROperator.func_begin || operator ==
IROperator.main_begin) {
                addLevel();
                HashMap<String, BasicBlock> blockMap = new HashMap<>();//初始化基
本块
                splitBasicBlocks(blockMap, i);//划分基本块

                backupTable();//备份符号表
                setDefUseVarList(blockMap, i);//设置活跃变量Use Def集合
                rollBackTable();//恢复符号表
                setInOutActiveVarList(blockMap);//迭代求解活跃变量in、out集合

                //构造冲突图
                ConflictGraph conflictGraph = new ConflictGraph();
                for (BasicBlock block : blockMap.values()) {
                    ArrayList<TableItem> inActiveVarList =
block.getInActiveVarList();
                    conflictGraph.addConflict(inActiveVarList);
                }
                //为变量分配寄存器
                conflictGraph.initAllocateStack(8);
                TableItem node = conflictGraph.getOneNode();
                while (node != null) {
                    boolean success = allocateGlobalReg(node);
                    if (!success) {
                        break;
                    } /*else {
                        System.out.println("分配全局寄存器成功：" + irCode.print() +
" " + node.getName() + " " + node.getGlobalRegName());
                    }*/
                    node = conflictGraph.getOneNode();
                }
            } else if (operator == IROperator.block_begin) {
                addLevel();
            } else if (operator == IROperator.block_end) {
                deleteLevel();
```

```
            } else if (operator == IROperator.func_end || operator ==
IROperator.main_end) {
                deleteLevel();
                freeGlobalReg();
            }
        }
    }
```

后端进行寄存器池维护的具体实现：

```
public void allocateGlobalReg(String name, MIPSTableItem item) {
        for (Register register : globalRegisters) {
            if (register.isAvailable() && register.getName().equals(name)) {
                register.setGlobalBusy(item);
                return;
            }
        }
    }

    public void preventGlobalReg() {
        for (Register register : globalRegisters) {
            if (!register.isAvailable()) {
                //全局寄存器被分配，进行保护
                addMIPSCode(new MIPSCode(MIPSOperator.sw, register.getName(),
null, -register.getGlobalItem().getOffset() - 4 + "($fp)"));
            }
        }
    }

    public void recoverGlobalReg() {
        for (Register register : globalRegisters) {
            if (!register.isAvailable()) {
                //全局寄存器被分配，进行恢复
                addMIPSCode(new MIPSCode(MIPSOperator.lw, "$fp", -
register.getGlobalItem().getOffset() - 4, register.getName()));
            }
        }
    }

    public void freeGlobalReg() {
        for (Register register : globalRegisters) {
            register.setAvailable();
        }
    }
```

对于传参时全局寄存器的保护和恢复问题的针对性优化：

一般情况下，程序的大部分函数调用都发生在main函数当中，而main函数使用的全局寄存器一般而言也比较多，因此在多次调用其他函数的过程中会因为全局寄存器的保护和恢复产生很大的开销（也许是竞速中一个点在进行全局寄存器分配后性能反而下降的原因）。为了解决这一问题，我们可以定义一个**函数全局寄存器的闭包**，即该函数即其调用函数所有用到的全局寄存器的集合，这时当我们在一个函数中调用其他函数时，需要保护和恢复的全局寄存器的值仅仅是**当前活跃的全局寄存器集合与被调函数全局寄存器闭包集合的交集**。除此之外，为了尽量减少不同函数用到的全局寄存器交集中寄存器的数目，

我们在分配全局寄存器时也进行一定的优化，**每次分配的寄存器为目前使用次数最少的全局寄存器**，这样能够让不同函数用到的全局寄存器尽量错开，减少了维护的代价。

6.局部窥孔

在完成寄存器分配之后，容易发现生成的MIPS代码中出现了大量冗余的操作，例如在将一个全局寄存器的值`move`到一个临时寄存器当中后，马上又使用该临时寄存器进行计算，这时我们完全可以将两条指令合并，**将后面对该临时寄存器再次赋值之前所有用到该临时寄存器的地方都替换为对应的全局寄存器。**

比如对于下面这段代码：

```
# printf
    # $s0 is flag
    move $t0, $s0
    addi $sp, $sp, -4
    sw $t0, ($sp)
```

可以直接优化为：

```
# printf
    # $s0 is flag
    addi $sp, $sp, -4
    sw $s0, ($sp)
```

除此之外，一些比较赋值语句和跳转语句也可以进行优化合并，例如：

```
sle $t1, $t2, $t3
bnez $t1, label
```

可以优化为：

```
ble $t2, $t3,label
```

再例如，有些寄存器被赋值之后马上又赋值给了其他寄存器，那么我们可以直接将两条语句进行合并，省去第二次赋值的操作。如：

```
lw $t0, 0($sp)
move $a0, $t0
```

可以优化为：

```
lw $a0, 0($sp)
```

还有一些因为之前的优化产生的寄存器将自己的值赋给自己的情况，对于这种语句显然也需要删除。

如：

```
move $a0, $a0
```

这一句直接可以删除。

再比如连续多次对同一内存地址进行读写，并且中间没有读取该内存地址的值，那么仅有最后一次的写操作是有效的，前面的写操作都可以被优化掉。

例如：

```
sw $t0, -4($sp)
...#没有修改$sp的值，也没有读取该地址中的值
sw $t1, -4($sp)
```

可以优化为：

```
...#没有修改$sp的值，也没有读取该地址中的值
sw $t1, -4($sp)
```

7.临时寄存器分配

在之前的寄存器分配方案中，我们只考虑了对跨越多个基本块的全局变量的寄存器分配方案，但是实际情况中也有许多变量**仅在一个基本块中起作用**，对于这部分变量，我们可以使用和全局寄存器分配类似的逻辑为它们分配目前仍处于空闲状态的`$t5~$t9`寄存器，并且在后端生成代码的过程中将它们和响应的寄存器捆绑发生作用。

中端进行寄存器和**局部非数组变量**的捆绑：考虑的空间为基本块内部，临时寄存器池释放的时机为**退出基本块**时。分配方案采用最简单的**先来先服务**原则（此处用图着色有点杀鸡用牛刀的感觉）。

后端需要进行的适配：和适配全局寄存器分配时的操作基本一致，不同点在于现在临时寄存器池释放的时机位于基本块的结尾，而在后端原先没有引入基本块的概念，为此我们需要**将ircode作为桥梁**，基本块开始和结束的ircode加上相应的标识，方便后端进行判断和使用。

中端确定基本块和IR的对应关系逻辑如下：

```
public void bindBlockAndIr() {
        HashMap<String, BasicBlock> blockMap = new HashMap<>();//初始化基本块
        for (int i = 0; i < optimizedIRList.size(); i++) {
            IRCode irCode = optimizedIRList.get(i);
            IROperator operator = irCode.getOperator();
            if (operator == IROperator.func_begin || operator ==
IROperator.main_begin) {
                addLevel();
                splitBasicBlocks(blockMap, i);//划分基本块
                //确定基本块和IR的对应关系
                for (BasicBlock block : blockMap.values()) {
                    for (int j = block.getStart(); j <= block.getEnd(); j++) {
                        optimizedIRList.get(j).setBasicBlock(block);
                        if (j == block.getStart()) {
                            optimizedIRList.get(j).setBasicBlockBegin();
                        }
                        if (j == block.getEnd()) {
                            optimizedIRList.get(j).setBasicBlockEnd();
                        }
                    }
                }
            } else if (operator == IROperator.block_begin) {
                addLevel();
            } else if (operator == IROperator.block_end) {
```

```
                deleteLevel();
            } else if (operator == IROperator.func_end || operator ==
IROperator.main_end) {
                deleteLevel();
            }
        }
    }
```

对临时寄存器的保护和恢复机制和全局寄存器的基本一致，仍然采用求各个函数临时寄存器闭包的策略。

但是发现进行临时寄存器优化后出现了负优化的现象，应该是和局部窥孔的优化发生了冲突，因此放弃该优化，选择回滚。

8.指令顺序调整

优化之前在进行参数和临时寄存器入栈等操作的时候，都是入栈一个数据就调整栈顶指针`$sp`，这样就产生了很多对`$sp`指针的计算操作，而这些操作其实很多都可以合并到一起。比如对于临时寄存器，可以等它们全部入栈之后一次性调整栈顶指针。同时，这样的操作还能带来新的**窥孔优化**机会，比如对于如下代码：

```
    sw $v0, -4($sp)
    addi $sp, $sp, -4
    addi $sp, $sp, -4
    sw $a0, ($sp)
```

容易看到在临时寄存器入栈完成，参数入栈开始的位置有两条连续对`$sp`寄存器做减法的指令，而这两条指令实际可以进行合并，并且这样的结构并不在少数。