21375049-彭欣阳-优化文档

在编译实验中，可以分为两个优化模块：**中间代码优化**，**目标代码优化**。实际上两者是相辅相成的，只做一个效果并不好。

由于本人在代码生成阶段开始时间较晚，实际上优化做的不是很多，主要参考实验给出的课件和教程。下面是我实现优化的过程

一、中间代码优化

1、SSA——静态单赋值

静态单赋值的作用主要是方便优化，因为每个变量只被定义一次，这样更好确定每个变量在什么时候被定义，在什么时候使用了。这样才能更好地确定寄存器如何分配。

但是存在一种情况：从一个基本块走到一个分叉点，分叉点都涉及到了这个变量的运算

为了保证SSA，于是定义了ϕ函数

$$y_3 \leftarrow \phi(A:y_1, B:y_2)$$

从基本块A跳转而来时，y3 取y1 的值；从基本块B跳转而来时，y3 取y2 的值。

这样就可以保证每个变量只赋值一次。

2、控制流图类CfgGraph

我在整个实验的中间代码采用了LLVM IR的结构。这种结构已经有较为成熟的定义，并且在处理跳转语句的时候，已经成功建立起了基本块，**并且在跳转语句中，已经记录了如何跳转**，这就相当于已经知道了流图的边，只需要将它们联系起来。

算法14.1　划分基本块

输入：四元式序列

输出：基本块列表。每个四元式仅出现在一个基本块中！

方法：

1、首先确定入口语句（每个基本块的第一条语句）的集合。

- 规则1：整个语句序列的第一条语句属于入口语句；
- 规则2：任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句；
- 规则3：紧跟在跳转语句之后的第一条语句属于入口语句。

2、每个入口语句直到下一个入口语句，或者程序结束，它们之间的所有语句都属于同一个基本块。

流图

- 流图是一种有向图
- 流图的结点是基本块
- 如果在某个执行序列中，B2的执行紧跟在B1之后，则从B1到B2有一条有向边
- 我们称B1为B2的前驱，B2为B1的后继
  - 从B1的最后一条语句有条件或者无条件转移到B2的第一条语句；或者：
  - 按照程序的执行次序，B2紧跟在B1之后，并且B1没有无条件转移到其他基本块。

在LLVM中，每个函数会记录自己有几个基本块，而且在课堂上，流图也只是基于基本块分析，并不会跨越函数。所以这里以函数为单位，构建流图。

下面是CfgGraph的声明

```
/**
* 控制流图，为了方便处理，以函数为基准
*/
class BasicBlock;
class Function;
class CfgGraph {
private:
    //流入的基本块，这里面存的就是每个基本块的前序基本块
    unordered_map<BasicBlock*,vector<BasicBlock*>> inBlocks;
    //流出的基本块，这里面存的就是每个基本块的后序基本块
    unordered_map<BasicBlock*,vector<BasicBlock*>> outBlocks;
    Function* function = nullptr;
public:
    CfgGraph() = default;
    explicit CfgGraph(Function* function);
    unordered_map<BasicBlock*,vector<BasicBlock*>>& getInBlocks();
    unordered_map<BasicBlock*,vector<BasicBlock*>>& getOutBlocks();
    /**
    * 构造这个函数的控制流图
    */
    void buildCFG();
    /**
    * 在两个基本块之间添加边，分别加入前驱map和后驱map
    * @param from_block 从哪个block来
    * @param to_block 到哪个block去
    */
    void addEdge(BasicBlock* from_block, BasicBlock* to_block);

    /**
     * 向两个基本块中间插入一个基本块
     * 用于消除phi指令
     * @param from_block 前序基本块
```

```
     * @param to_block 后序基本块
     * @param mid_block 中间基本块
     */
    void insertBlock(BasicBlock* from_block, BasicBlock* to_block, BasicBlock*
mid_block);

    /**
     * 将两个基本块进行合并，修改对应的in和out
     * @param from_block
     * @param to_block
     */
    static void mergeBlock(BasicBlock* from_block, BasicBlock* to_block);

    /**
    * 输出控制流图，debug用
    */
    string printCFG();
private:
    /**
    * 向流图添加前序基本块
    * @param from_block 从哪个block来
    * @param to_block 到哪个block去
    * @param index 将基本块插入到哪个位置？一般用于在两个基本块之间插
    */
    void addInBlock(BasicBlock* to_block,BasicBlock* from_block, int index = -1);
    /**
    * 向流图添加后序基本块
    * @param from_block 从哪个block来
    * @param to_block 到哪个block去
    * @param index 将基本块插入到哪个位置？一般用于在两个基本块之间插
    */
    void addOutBlock(BasicBlock* from_block,BasicBlock* to_block, int index =
-1);
};
```

这个类以函数为单位，构建了两个map：一个表示基本块的出边，另一个表示基本块的入边。由于这个是图，所以对于每个节点（基本块），可能有多条边，所以采用vector进行存储。

调用buildCFG()方法，便可以将整个流图生成出来。根据跳转指令判断

3、Mem2Reg——内存引用转换为寄存器引用

在中间代码LLVM实现的过程中，绝大多数指令都是通过alloca->load/store指令实现的。但是事实上，这样做会产生大量的冗余代码，并且还是内存冗余的。比如一个加法，常规只需要让虚拟寄存器相加即可，而在中间代码生成的，却是先load再相加，然后再store目标虚拟寄存器。

Mem2Reg就是为了解决这个问题

总体上，分为两个步骤：**插入ϕ函数**，**变量重命名**

3.1、支配关系和支配集

在课堂上，只介绍了支配关系dom，而实验课件介绍了支配关系的求法。

程序流图中，如果从首节点出发，任何到达节点B的路径上都要经过A， 那么A就是B的必经节点(Dominator)，记为A dom B。

如果A dom B, 且A 不等于 B， 则称A严格支配B

课件上介绍的支配集是“被谁支配”，实际上我们需要的也只是支配树，而支配树的构建也可以采用“谁支配了谁”。对于支配集的求解，我直接采用了dfs的暴力求法，那么从流图入口处开始遍历，我直接顺着路线考虑，所以支配集实际上求的是“这个点支配了哪些节点”。

在遍历过程，对于所有基本块都搜索，当遍历到当前基本块A便停止。那么经过的基本块代表着“**不通过A也能到达**”。反过来，没有经过的基本块就是“**必须通过基本块A才能到达**”，而这个便是**支配的定义**。记录没有经过的基本块，就是A的“支配集”。

3.2、支配树和支配边界

课件上给出了关于支配树的定义：如果x严格支配y，则让x成为y的祖先。通过这种方式能让支配关系唯一地确定一个树状结构，这即是支配树。支配树的根结点是CFG图的入口结点。

直接支配者：若x严格支配y，并且不存在被x严格支配同时严格支配y的结点， 则称x是y的直接支配者。

实际上，x就是支配树上y的父节点。

生成支配树时，需要判断functionBlock支配的节点中，有没有支配dominate_Block的节点

如果有，那么说明functionBlock不是直接支配dominate_block

如果是直接支配，则加入到对应的map中即可建立边的映射。

支配边界：

若结点n不严格支配结点x，但严格支配x的一个前驱结点，则x在n的支配边界DF(n)中。

在LLVM中，由于没有纯定义的基本块，所以基本上有如下公式：

$$DF^{+}(S) = J(S \cup entry) = J(S)$$

实际上按照实验教程给的伪代码计算即可

- **完成支配关系计算后可以按照以下方式计算支配边界：**

Algorithm 3.2, The SSA Book

for $\langle a, b\rangle \in$ CFG 图的边集 **do**
  $x \leftarrow a$
  while x 不严格支配 b **do**
    $DF(x) \leftarrow DF(x) \cup b$
    $x \leftarrow x$ 的直接支配者

- **在上述伪代码中：**
  - 先枚举 DF 中的点 b 以及其被严格支配的前驱 a 之间的边 $\langle a, b\rangle$。
  - 对于严格支配 a 的点 x，$DF(x)$ 包含点 b，故枚举 x 并将 b 加入 $DF(x)$ 中。

同样，每个函数拥有一个支配关系集和支配树

```
/**
     * 生成该函数的支配集
     * 在for循环中找到所有不被 basicBlock支配的基本块，放入集合中
     * 这样所有不在集合中的基本块，都是被basicBlock支配的基本块
     * 这样可以将这些基本块加入到支配集里
     * 注意，这里生成的是每个节点支配了谁，而不是被支配；
```

```
    */
    void generateDominateBlocks();

    /**
     * 利用dfs进行深搜，找到所有不被block支配的基本块，放入集合中
     * 原理：
     * 当我们的入口基本块和所求的目标基本块相等时，函数停止
     * 这意味着集合并不会加入目标基本块block的后继
     * 如果block的某个后继基本块，可以由其它基本块经过，
     * 那么当entry等于这个“其它基本块”时，也会将这个后继加入集合
     * 所以最终没有加入集合的基本块，就是basic支配的基本块，必须通过basic才能到达
     * @param entryBlock dfs的入口，调用的时候应当是第一个基本块
     * @param block
     * @param noDominateSet 不被block支配的基本块集合
     */
    void dfsDominateBlocks(BasicBlock* entryBlock, BasicBlock* block,
unordered_set<BasicBlock*>& noDominateSet);

    /**
     * 生成支配树，其实就是求直接支配关系
     */
    void generateDominantTree();

    /**
     * 构建支配边界
     * 按照教程给的伪代码遍历即可
     * 首先，遍历函数的所有基本块和后记，记为a和b
     * 令x等于a
     * 然后，当x不严格支配b，将b纳入DF(x)中
     * x更新为x的直接支配者，也就是支配树的父节点
     */
    void generateDominateEdge();
```

3.3、插入phi函数

教程也给出了伪代码：

- 以下是插入 ϕ 函数的算法：

Algorithm 3.1, The SSA Book

```
for v：原程序中的变量 do
    F ← {}                                          ▷ 已插入 φ 函数的基本块集合
    W ← {}                          ▷ v 的定义（包括新插入的 φ 函数）所在的基本块集合
    for d ∈ v 的定义 Defs(v) do
        W ← W ∪ {d 所在的基本块}
    while W ≠ {} do
        从 W 中选择并删除一个基本块 X
        for Y：DF(X) 中的基本块 do
            if Y ∉ F then
                在基本块 Y 的入口处插入 φ 函数 v ← φ(...)
                F ← F ∪ {Y}
                if Y ∉ Defs(v) then
                    W ← W ∪ {Y}
```

  - 对于变量 v，W 初始化为 D_v，并不断与 $DF(W)$ 合并。最终 $DF^+(D_v)$ 中的结点都会被 Y 遍历。

具体代码实现如下：

```
void MemToReg::insertPhiInstruction() {
    vector<BasicBlock*> F;
    vector<BasicBlock*> W;
    W.reserve(defineBlocks.size());
    for(auto d: defineBlocks) {
        W.push_back(d);
    }
    while(!W.empty()) {
        BasicBlock* x =W.back();
        W.pop_back();
        vector<BasicBlock*>& DFX = x->getDominateEdge();
        for(auto y: DFX) {
            if(count(F.begin(),F.end(),y) == 0) {
                //首先在F加入
                F.push_back(y);
                //在基本块Y的入口处插入空的φ函数
                auto phiInstruction = new PhiInstruction(
                    new IRInt(),
                    IRName::getLocalVariableName(y->getFunctionParent()),
                    y->getInBlocks());
                y->addInstruction(phiInstruction,0);
                loadInstructions.push_back(phiInstruction);
                storeInstructions.push_back(phiInstruction);
                if(count(defineBlocks.begin(),defineBlocks.end(),y) == 0) {
                    //y不在定义块中，要把y添加到W中
                    W.push_back(y);
                }
            }
        }
    }
}
```

3.4、变量重命名

这里是最重要的一步

在Mem2Reg中，本质上就是消除不必要的alloca，store和load指令。但是原本的计算，调用指令等，可能使用了这些虚拟寄存器，那么在这里要做的便是把它们使用的虚拟寄存器，更换对应的名字。

实验课件也给出了伪代码：

■ **接下来给出变量重命名的伪代码：**

Algorithm 3.3, The SSA Book

```
for v：原程序中的变量 do
    v.reachingDef ← null                    ▷ 变量的到达定义（支配当前位置的定义）
for BB：前序遍历支配树 do
    for i：基本块 BB 中的指令 do
        for v：非 φ 函数 i 引用的变量 do
            updateReachingDef(v, i)
            用 v.reachingDef 替换 v 的这处引用
        for v：i 定义的变量 do
            updateReachingDef(v, i)
            创建新变量 v'
            用 v' 替换 v 的这处引用
            v'.reachingDef ← v.reachingDef
            v.reachingDef ← v'
    for φ：BB 后继结点中的 φ 函数 do
        for v：φ 中引用的变量 do
            updateReachingDef(v, φ)
            用 v.reachingDef 替换 v 的这处引用
```

对于定义——使用链，我是在User类中和Value进行构建，实际上也是在维护这个链

4、不运行的死代码删除

通过以上的方法，已经能够初步将LLVM代码提升为虚拟寄存器

但是，在进行公共测试库测试时，发现了PHINode错误和死循环的现象

经过debug，发现PHINode的错误是因为一个前驱块结尾有多个跳转语句。它当然会在第一个跳转指令跳转，但是LLVM 的phi函数似乎是通过最后一条语句进行确定。所以这里需要把不运行的代码都删除：**主要是return后面的代码和跳转语句之后的代码**

对于死循环的错误，经过探查发现是有一些基本块明明不能到达，但是由于跳转语句，将其纳入了流图，导致建立支配树后可能出现回路。所以从第一个基本块开始dfs遍历，将所有能走到的基本块标记，删除没有标记的基本块，就可以解决这个问题。

当然，这么做对于时间性能没有改进，但是不会报错和死循环，并且减少了代码体积。

由于我的开始时间晚，所以并没有做“未使用但运行的死代码删除”

```
/**
* 删除死代码
* 主要原因是有一些return，跳转指令后的语句，这些没有必要保留
*/
class DCE {
public:
    /**
    * 删除基本块内的分支后代码。
    * 对于一个基本块，跳转/return代码之后的所有代码都不会到达，直接删除即可
```

```
    * 而且如果不删除，那么在插入phi函数会出现异常
    * @param basic_block
    */
    static void deleteBranchInstruction(BasicBlock* basic_block);

    /**
    * 删除函数中不可到达的基本块
    * 主要原因是，有些块不可到达，但是后面流图可能会有很多后继，导致构建支配集的时候出现问题
    * 进而影响支配树的生成
    * 采用dfs暴力搜索
    * @param function
    */
    static void deleteBlock(Function* function);
private:
    static void dfsDeleteBlock(BasicBlock* basic_block);
};
```

5、常数折叠

这一部分主要是优化常量计算

如果只有一个常数，直接传播

对于一个计算指令，看操作数有几个常数。

- 两个常数：直接计算
- 一个常数：优化类似于a+0,a-0,a/1,a*1，0/a等指令
- 没有常数：判断两个数会不会相等，优化除法

计算后，将返回的Value替换到所有使用原来寄存器的指令，就可以实现常量传播

```
/**
* 用于处理常量折叠，对于一个基本块内进行处理
*/
class ConstFold {
public:
    /**
    * 计算指令无外乎有0个立即数，1个立即数，2个立即数，所以分三种情况处理即可
    */
    static void handleCalculateInstruction(BasicBlock* basic_block);
private:
    /**
    * 0个立即数，如果为null，则表示没有折叠
    * 简单处理，如果两个操作数的指针都相同，那么优化减法和除法
    */
    static Value* handle0(CalculateInstruction* calculate_instruction);
    /**
    * 1个立即数，如果为null，则表示没有折叠
    * 主要是用于优化a+0，m*0或1，0/c，0%d，c/1，d%1
    */
    static Value* handle1(CalculateInstruction* calculate_instruction);
    /**
    * 2个立即数，如果为null，则表示没有折叠
    * 这种直接计算，然后移出指令，换成常数
```

```
    */
    static Value* handle2(CalculateInstruction* calculate_instruction);
};
```

二、目标代码优化

1、活跃变量分析

后端采用图着色分配寄存器，所以首先进行活跃变量分析

就是课堂课件的做法

算法14.5 基本块的活跃变量数据流分析

- **输入：** 程序流图，且基本块的use集合和def集合已经计算完毕
- **输出：** 每个基本块入口和出口处的in和out集合，即in[B]和out[B]
- **方法：**
  1. 将包括代表流图出口基本块B_{exit}在内的所有基本块的in集合，初始化为空集。
  2. 根据方程$out[B] = \cup_{B的后继基本块P} in[P]$，$in[B] = use[B] \cup (out[B] - def[B])$，为每个基本块B依次计算集合out[B]和in[B]。如果计算得到某个基本块的in[B]与此前计算得出的该基本块in[B]不同，则循环执行步骤2，直到所有基本块的in[B]集合不再产生变化为止。

```
class ActiveVarAnalysis {
private:
    Function*function = nullptr;
    unordered_map<BasicBlock*, unordered_set<Value*>> use;
    unordered_map<BasicBlock*, unordered_set<Value*>> def;
    unordered_map<BasicBlock*, unordered_set<Value*>> in;
    unordered_map<BasicBlock*, unordered_set<Value*>> out;
public:
    ActiveVarAnalysis() = default;
    explicit ActiveVarAnalysis(Function* function);
    unordered_set<Value*>& getUseVar(BasicBlock* block);
    unordered_set<Value*>& getDefVar(BasicBlock* block);
    unordered_set<Value*>& getInVar(BasicBlock* block);
    unordered_set<Value*>& getOutVar(BasicBlock* block);

    /**
    * 遍历所有的基本块，生成use和def集合
    */
    void generateUseDef();

    /**
```

```
    * 活跃变量分析
    * 首先，最初的out均为空
    * in = use + (out-def)，集合运算
    * out是后继的in的并集
    * 当in不再发生变化时，退出循环
    */
    void generateInOut();

    /**
    * debug使用
    * @return
    */
    string printActiveAnalysis() const;
};
```

2、寄存器分配

使用两个map来存储寄存器和Value的映射。

首先，初始化寄存器，由于之前我采用t8和t9作为栈式寄存器，所以这里只用t0-s7这些寄存器

遍历所有的指令，记录每个变量在该块中最后一次被使用的位置，然后，第二次遍历，释放寄存器，遍历所有直接支配的节点，调用buildRegisterMap方法（这个方法也递归调用了分配寄存器的方法），然后，将该基本块定义的变量对应的寄存器释放，将使用的变量重新恢复。

成功构建寄存器和Value的map后，将其传递给Function，这样在生成mips时根据这个map就能够确定哪个变量该对应哪个寄存器。

3、消除phi指令

首先转换成并行的PC指令

再转换为move指令

注意：对于循环赋值的情况，需要引入中间变量寄存器进行赋值

例如，直接串行化成为move会出现如下情况：

```
move $t2, $t1
move $t3, $t2
```

要转换成

```
move $temp_t2, $t2
move $t2, $t1
move $t3, $temp_t2
```

至于$temp_t2，这个可以使用t8,t9等不参与分配的寄存器

- 以下是分割关键边，并将 ϕ 函数替换为 pc 指令的算法：

Algorithm 3.5, The SSA Book

```
for B : CFG 中的基本块 do
    for E_i = ⟨B_i, B⟩ : 基本块 B 的入边 do
        PC_i ← 空的 pc 指令
        if B_i 有多条出边 then
            创建新的基本块 B'_i
            将边 E_i = ⟨B_i, B⟩ 替换为边 ⟨B_i, B'_i⟩ 与边 ⟨B'_i, B⟩
            将 PC_i 加入 B'_i 中
        else
            将 PC_i 加入 B_i 末尾
    for a_0 = φ(B_1 : a_1, ..., B_n : a_n) : 基本块 B 中的 φ 函数 do
        for a_i : φ 函数中对应各个基本块 B_i 的参数 do
            将 PC_i 指令置为 a'_i ← a_i
        删除该 φ 函数 a_0 = φ(B_1 : a_1, ..., B_n : a_n)
```

  - 在枚举入边的循环中分割关键边。
  - 枚举 ϕ 函数的循环将 ϕ 函数替换为各个前驱基本块末尾的 pc 指令。

- 以下是将 pc 指令转化为串行 move 指令的算法：

Algorithm 3.6, The SSA Book

```
pcopy ← 需要串行化的 pc 指令
seq ← ()                                  ▷ pc 指令串行化后的 move 指令序列
while pcopy 中存在 a ≠ b 的指令 b ← a do
    if 存在指令 (b ← a) ∈ pcopy 使得不存在 (c ← b) ∈ pcopy then
        将 b ← a 加入到 seq 末尾，并从 pcopy 中删除 b ← a
    else
        从 pcopy 中选择 a ≠ b 的指令 b ← a
        新建寄存器 a'
        将 a' ← a 加入到 seq 末尾
        将 pcopy 中的 b ← a 替换为 b ← a'
```

  - 每次选择目标寄存器未被依赖的 pc 指令改写为 move 指令，对应 if 分支。
  - 否则依赖图中存在环，选择某个结点拆点破环，对应 else 分支。

```
/**
* 主要用于phi消除，消除后才能转换为mips指令
* 主要是封装了pc转move的情况
*/
class deletePhi {
public:
    /**
    * 消除phi的启动类
    */
    static void DeletePhi(Module* module);
    /**
    * 生成临时变量的temp move
    * 由于可能存在循环和冲突，所以需要引入临时寄存器，保证串行执行后和并行一致
    * @param function
    * @param phi_move_instructions
    */
    static vector<PhiMoveInstruction*> generateMoveTemp(Function* function,
vector<PhiMoveInstruction*>& phi_move_instructions);
```

```
private:
    static Module* module;
    /**
    * 生成循环的move指令集合
    * 比如： move a, b;  move c, a;，这种串行化后会改变PC的语义，所以需要引入临时寄存器
    * 首先，检查每个move指令之后的指令，如果这个指令的dest是某个move的src，那么存在循环赋值的问题
    * 需要增加中间变量，即将所有source使用dest的move指令全部替换为临时寄存器，
    * 最后在moveList开头添加临时寄存器的指令
    * @param function
    * @param phi_move_instructions
    * @return
    */
    static vector<PhiMoveInstruction*> generateMoveCycle(Function* function, vector<PhiMoveInstruction*>& phi_move_instructions);

    /**
     * 用于生成共享寄存器的Move指令集合
     * 1.检查该指令之前的所有指令，
     * 如果source对应的reg同时是某一个move的dst的reg，那么存在寄存器冲突的问题
     * 2. 如果出现了寄存器冲突的情况，我们需要增加中间变量，
     * 将所有使用作为dest的move指令，改为临时今存其，并在开头添加寄存器
     * @param function
     * @param phi_move_instructions
     * @return
     */
    static vector<PhiMoveInstruction*> generateMoveSameRegister(Function* function, vector<PhiMoveInstruction*>& phi_move_instructions);
};
```