基于LLVM的SysYF进阶优化

- 基于LLVM的SysYF进阶优化
  - 实验选题
  - 实验设计
  - 组员分工
  - 贡献比
  - 具体实现
    - 合并实验框架
    - 中间代码优化
      - 支配树算法
      - Mem2Reg
      - 常量折叠与传播
      - 函数性质分析
      - 局部公共子表达式消除
      - 死代码删除
      - 循环表达式外提
      - 活跃变量分析
    - 后端代码生成与优化
      - 寄存器分配
      - phi指令数据搬移
      - 栈分配
  - 评测结果
    - 中间代码优化
      - 正确性
      - 运行时间
      - 对不同优化Pass效果的测试
    - 后端代码生成与优化

实验选题

基于LLVM在若干方向对SysYF语言生成的LLVM中间代码进行优化，并进一步完成汇编代码的生成和优化。

实验设计

基于可行性和工作量的角度考虑，本实验在PW6中间代码生成和去年PW8的实验框架基础上进行，主要分为以下三个步骤：

- i. 将去年的代码合并入PW6实验框架
- ii. 添加Pass类的派生类，实现对LLVM IR的各种优化
- iii. 实现Arm汇编代码的生成和优化

注：本实验使用的中间代码生成是第六次实验中李牧龙和徐翊然完成的版本，在其基础上做了一些修改（用PW8框架中的代码替换了一部分）。感谢徐翊然同学的付出。

组员分工

李牧龙：将去年框架的中间代码优化部分合并入PW6中，完成LLVM IR的优化。

徐航宇：将去年框架的后端部分合并入PW6中，完成Arm汇编代码的生成和优化。

贡献比

李牧龙：50%
徐航宇：50%

具体实现

合并实验框架

这一部分的工作主要是将代码文件加入文件目录，修改CMakeLists，并将去年代码中的裸指针改为智能指针，总体并不复杂。值得一提的是在此过程中发现了PW6代码框架的一点小问题：

- i. 创建 `PhiInst` 时出现段错误：主要是 `PhiInst` 的 `init` 函数中未正确设置 `parent_` 成员变量所致。
- ii. `Value` 类的 `replace_all_use_with` 方法未正确维护use链。

现在这两个问题均已修复。

中间代码优化

中间代码优化采用了Mem2Reg、常量折叠与传播、函数性质分析、局部公共子表达式消除、死代码删除、循环表达式外提这几种优化方式。其中Mem2Reg及其依赖的支配树算法是PW8框架中已有的优化，其余优化均在本次实验中完成。另外，为进行后端代码的寄存器分配，还完成了活跃变量分析。

支配树算法

支配树计算是框架中自带的，采用了https://www.cs.rice.edu/~keith/EMBED/dom.pdf中的算法。

该算法是一种数据流分析。对于每个基本块，遍历它的前驱。如果存在一个直接支配节点已经确定的前驱块，就先将当前基本块的直接支配节点设为该前驱；如果在此之后又找到了这样的前驱块，就取这个新前驱块与当前直接支配节点的最近共同支配节点作为支配节点（因为直接支配节点一定支配这一基本块的所有前驱，所有前驱的公共支配节点一定支配这一基本块）。

算法开始时将entry块的直接支配节点设为自己，其余基本块设为空。按逆后序依次计算每个基本块的直接支配节点，直到所有基本块的直接支配节点不再发生改变。

这里按逆后序计算而不用先序的原因是，按逆后序计算可以保证每个节点在计算时，所有前驱已经被计算过。而先序无法保证。

该算法只算出了直接支配节点，但知道直接支配节点后即可沿直接支配节点的链找到所有支配节点。

除支配节点外，后面的Mem2Reg还需用到支配边界。支配边界可以按如下算法计算：

```
//计算 CFG 中每个 node 的支配边界
For (a, b) in CFG edges do:
    x <- a;
    while x dose not strictly dominate b do:
        DF(x) = (DF(x) ∪ b);
        x = immediate dominator(x);
```

这一部分的完整代码见DominateTree.h和DominateTree.cpp。

Mem2Reg

Mem2Reg是一种将内存变量转换为寄存器变量的优化。由于LLVM IR是无限寄存器的静态单赋值形式，因此局部的非数组变量都可以被转换为寄存器变量。

这一优化过程不是我们做的（框架带的），源代码中也没有注释，因此下面的解释是我们自己阅读代码的理解。

要进行这一变换，主要需要处理 `alloca`，`load` 和 `store` 指令。对于一个变量，如果在一个基本块内有对它的定值（`store`），那该基本块内所有的引用（`load`）都可以被最近的一个定值替换。被替换的 `load` 可以全部删除，而值被使用过的 `store` 也可以全部删除。这种情况很容易处理。

然而，对于 `load` 语句对应的定值来自基本块外的情况就比较复杂。这种情况需要对支配树进行分析。

首先讨论 phi 指令的插入。考虑两个基本块A和B，其中A直接支配B。如果B中 load 了某个变量，而A中有对它的定值，且从A到B的路径上没有再次定值，那么就可以直接用该定值替换B中的 load 的使用。否则，这个 load 对应的定值来自多个基本块，需要添加 phi 指令进行汇合。而对于后一种情况，我们只需要处理每个定值所在基本块的迭代支配边界（这是因为这个定值不会“导致”所在基本块支配的基本块上出现 phi 指令，除此之外它到达的基本块要么在迭代支配边界中，要么被迭代支配边界中的基本块支配（这里注意基本块自身支配自身）），在迭代支配边界的每个基本块开头处为对应变量插入 phi 指令。

然后是对 phi 指令操作数的填充和对引用基本块外定值的 load 语句的替换。从entry块开始dfs，用一个map记录每个变量对应的最新的定值（ phi 或者 store ）。每遍历到一个基本块，就依次遍历其中的指令。对于定值指令（ phi 或 store ），将其加入map中；对于 load ，在map中查找对应的定值，用最新的定值替换，将其删除。然后查找后继中所有的 phi 指令，用对应变量的最新定值和当前基本块作为操作数插入 phi 指令。最后在退出时，弹出map中当前基本块的所有定值。这一步中遇到的 store 也会被删除。

最后移除所有的 alloca 即可。

这一部分的完整代码见Mem2Reg.h和Mem2Reg.cpp。

常量折叠与传播

常量折叠与传播，即将部分操作数全为常量的表达式在编译期间进行计算，并用计算结果直接替代该表达式的使用。尽管在IRBuilder中生成中间代码时，我们已经尽可能地将所有常量表达式直接计算为常量，但Mem2Reg优化后还是会产生大量的常量表达式。这为常量折叠与传播提供了可能。

这一部分的实现思路较为简单。对于操作数全为常量的计算表达式（如四则运算和取模，以及强制类型转换），直接计算出结果，创建相应的常数，并调用 replace_all_use_with 方法替换所有引用即可。

值得一提是对条件br语句的常量折叠。如果条件br语句引用的比较语句为常量表达式，那么可以直接确定br语句的跳转方向，可以将条件br换为无条件的跳转。但在这一替换后，另一个分支的基本块少了一个前驱基本块，可能需要对它的phi指令进行维护，相关代码如下：

```
for(auto &inst: del_bb->get_instructions()) {// 删除phi指令中对应前驱块的操作数
    if(!inst->is_phi()) {
        break;
    }
    auto phi = dynamic_pointer_cast<PhiInst>(inst);
    auto ops = phi->get_operands();
    for(int i = 0; i < ops.size(); i+=2) {
        auto op = ops[i];
        auto op_block = dynamic_pointer_cast<BasicBlock>(ops[i+1]);
        if(op_block != block) {
            continue;
        }
        if(phi->get_num_operand() == 4) {// 如果原本就只有两对操作数，那么直接用另一对中的Value替换即可
            Ptr<Value> new_op;
            if(i == 0) {
                new_op = phi->get_operand(2);
            }
            else {
                new_op = phi->get_operand(0);
            }
            phi->replace_all_use_with(new_op);
        }
        else {
            phi->remove_operands(i, i+1);
        }
        break;
    }
}
```

这一部分的完整代码在ConstCalc.h和ConstCalc.cpp中。

函数性质分析

这一部分主要为分析函数是否为纯函数。

在这一轮优化中，我们的要求比较严格，将纯函数定义为：

- i. 没有对全局变量的引用；
- ii. 没有传入数组参数；

- iii. 如果有函数调用，则调用的函数必须是纯函数。

理论上，即使传入了数组参数，如果没有对数组进行赋值，那么函数仍然是纯函数（不考虑并行的情况下）。但是这样的函数的运行结果依赖于调用时数组的状态，而数组状态是否改变并不好确定。因此我们将这样的函数视为非纯函数。

满足上面这些性质的函数，执行结果仅由传入的参数决定。这样的函数可以为优化带来很多方便。

分析函数的这种性质也非常简单。

- 首先，对于函数体为空的函数，我们认为是外部函数，稳妥起见视为非纯函数。
- 然后扫描函数的所有参数，如果有数组参数则记录下来。
- 最后扫描函数体中所有的指令。如果有引用全局变量的指令、调用非纯函数的函数调用指令或者对上面记录的数组参数进行赋值的语句，则该函数不是纯函数。
- 没有被上述过程记录为非纯函数的函数即为纯函数。

这一部分的完整代码见FindPureFunc.h和FindPureFunc.cpp。

局部公共子表达式消除

在同一基本块内，如果有两个表达式完全相同，那么就可以去掉后面的表达式，将它的结果替换为前面的表达式的结果。这一优化只能对除 `alloca` ， `br` ， `ret` ， `store` ， `load` 以外的语句进行（因为这些表达式的效果不完全在于返回值，或者返回值不能完全由参数决定。

LLVM IR的静态单赋值特性为这一优化的实现带来了方便。因为每个变量只会被赋值一次，因此在一个基本块内，被引用的值不可能在两个引用它的指令之间被再次定值。因此我们可以只比较操作数是否是同一个对象。

比较表达式是否相同也很简单，只需要依次比较类型、操作数数量，每个操作数是否是同一对象即可。这里有两个例外情况：

其中一个是常数，因为在中间代码生成传入常数时，每一个常数都是新建的不同对象，尽管它们的值可能相同。因此比较操作数时，对常数需要特别处理，比较它们的值是否相同。比较操作数的核心代码如下：

```
bool compare_op(Ptr<Value> op1, Ptr<Value> op2) {
    if(op1->get_type()->get_type_id() != op2->get_type()->get_type_id()) {
        return false;
    }
    auto const_int1 = dynamic_pointer_cast<ConstantInt>(op1);
    auto const_int2 = dynamic_pointer_cast<ConstantInt>(op2);
    auto const_float1 = dynamic_pointer_cast<ConstantFloat>(op1);
    auto const_float2 = dynamic_pointer_cast<ConstantFloat>(op2);
    if(const_int1 && const_int2) {
        return const_int1->get_value() == const_int2->get_value();
    }
    else if(const_float1 && const_float2) {
        return const_float1->get_value() == const_float2->get_value();
    }
    else {
        return op1 == op2;
    }
}
```

另一个即为函数调用。如果两个表达式调用了相同的纯函数，且传入的参数相同，那么这两个表达式也是公共子表达式。

这一部分的完整代码见LocalCSE.h和LocalCSE.cpp。

死代码删除

死代码删除即删除不会被执行/执行结果不会被使用的代码。我们将这一部分分为死指令删除和死基本块删除。

死指令删除：

死指令删除仅涉及除 `br` ， `ret` 和 `store` 外的指令。如果一条指令的use链为空，那么这条指令被视为死指令，可以被删除。对于函数调用指令，则额外要求它调用的必须是纯函数。

这一部分的完整代码见DCE.h和DCE.cpp。

死基本块删除：

可以被删除的死基本块分为两类：一种是从entry开始dfs，没有被访问到的基本块；另一种是只有一个前驱和一个后继，且前驱也只有这一个后继，并且内部没有除 `br` 外的指令的基本块。

对于第一种情况，只需一遍dfs即可找出需要被删除的基本块，非常简单。关键在于像常量折叠中对 `br` 的处理一样，这一基本块的后继少了一个前驱，其中的 `phi` 指令可能需要维护。这一部分的代码与常量折叠中的处理类似，不再赘述。

第二种情况，只需对每个基本块检查它的前驱和后继即可。然而由于后继的前驱发生了变化，因此同样可能需要维护其中的 `phi` 指令。需要将涉及到的 `phi` 指令中被删除的基本块替换为它的前驱，这一部分的代码如下：

```
for(auto &inst: succ->get_instructions()) {// 后继的前驱发生了改变，需要更新phi
    if(!(inst->is_phi())) {
        break;
    }
    auto phi = std::dynamic_pointer_cast<PhiInst>(inst);
    auto ops = phi->get_operands();
    for(int i = 0; i < phi->get_operands().size(); i+=2) {
        auto op = ops[i];
        auto op_block = std::dynamic_pointer_cast<BasicBlock>(ops[i+1]);
        if(op_block != block) {
            continue;
        }
        phi->set_operand(i+1, prev);// 将block对应的操作数修改为它的前驱即可
        block->remove_use(phi);
    }
}
```

还需注意的是，在每一次删除基本块后，都需正确维护其它基本块的前驱和后继，否则可能毒害后续的基本块删除。

死基本块删除的完整代码见DBE.h和DBE.cpp。

循环表达式外提

循环表达式外提，即将循环中不依赖于循环变量的表达式提到循环外。这一优化可以减少循环中的计算量，提高效率。要实现这一优化，首先需要找到循环，然后才能判断循环表达式是否依赖于循环变量，并进行外提。

循环识别：

循环识别分为以下几步：

- 采用课内讲过的基于回边的算法寻找自然循环。对流图从entry开始dfs，然后对于每一条回边n->d，从n开始沿流图逆向dfs，将经过的基本块全部加入该回边对应的循环中，直到回到d。这样就可以找到所有的自然循环。
- 找出每个自然循环的入口块。这一步只需遍历循环中所有的块，找到支配其他所有节点的块即可。
- 计算循环间的嵌套关系。首先对所有循环按基本块数量升序排序，从最小的循环开始往后遍历，如果当前循环的基本块集合是排在它后面的某个循环的子集，那么它就是这个循环的内层循环。
- 合并循环。有些循环内可能存在多条回边，因此会存在于多个自然循环中，但不应该被当作多个不同循环处理。因此，需要遍历所有的循环，如果存在两个循环的入口块相同，又没有嵌套关系，那么就将这两个循环合并为一个循环。

这一部分的完整代码见FindLoop.h和FindLoop.cpp。

循环表达式外提：

循环表达式外提的第一步是扫描循环不变表达式。这一部分的思路与前面一些优化Pass类似。首先，循环不变式只考虑除 `alloca` ， `br` ， `load` ， `store` ， `phi` 之外的指令。然后，对于每一条指令，如果它的操作数全部在循环外被定值，且不是全局变量，那么它就是循环不变式。同样，如果是函数调用指令，则额外要求调用的函数是纯函数。

第二步是移动语句。这里分为两种情况：

- 如果循环的入口块的前驱中只有一个基本块在循环外（即外界进入循环只有一条路径），那么直接把循环不变式移动到这一个循环外的基本块内即可。
- 但如果循环入口的前驱中有多个基本块不在循环内，那就需要创建一个新的基本块，将这些不在循环内的基本块连接到这个新的基本块上（同时移除与循环入口块的连接），再将这个新的基本块连接到循环入口块，然后将循环不变式移动到这个新的基本块上。由于这种情况下基本块的结构发生了改变，因此还需要维护其中的 `phi` 指令，将 `phi` 指令的操作数分为两部分（拆成两条指令），一部分留在循环入口块，一部分留在新的基本块上，这一部分的核心代码如下：

```
PtrVec<Instruction> insts(loop_entry->get_instructions().begin(), loop_entry->get_instructions().end());
for(auto inst : insts) {
    if(!inst->is_phi()) {
        break;
    }
    auto phi = std::dynamic_pointer_cast<PhiInst>(inst);
    std::set<std::pair<Ptr<Value>, Ptr<BasicBlock>>> in_entry, in_pre;
    for(int i = 0; i < phi->get_num_operand(); i+=2) {
        auto op = phi->get_operand(i);
        auto op_block = std::dynamic_pointer_cast<BasicBlock>(phi->get_operand(i+1));
        if(loop->get_blocks().find(op_block) != loop->get_blocks().end()) {// 这一对操作数包含循环内的基本块，留在循环入口
            in_entry.insert({op, op_block});
        }
        else {// 这一对操作数用到了循环外的基本块，应该被移到循环前的新基本块里
            in_pre.insert({op, op_block});
        }
        if(in_entry.empty()) {// 没有用到循环内的基本块，直接移到pre中
            loop_entry->delete_instr(phi);
            loop_pre_bb->add_instr_begin(phi);
        }
        else if(in_pre.size()) {// 既有循环内的基本块，又有循环外的基本块
            if(in_pre.size() == 1) {// 这种情况需要在entry里创建新的phi
                auto new_phi = PhiInst::create_phi(phi->get_type(), loop_entry);
                for(auto pair : in_entry)
                    new_phi->add_phi_pair_operand(pair.first, pair.second);
                new_phi->add_phi_pair_operand(in_pre.begin()->first, in_pre.begin()->second);
                phi->replace_all_use_with(new_phi);
                loop_entry->delete_instr(phi);
                loop_entry->add_instr_begin(new_phi);
            }
            else {// 这种情况需要把原来的phi拆成2部分，一部分在entry里，一部分在pre里
                auto pre_phi = PhiInst::create_phi(phi->get_type(), loop_pre_bb);
                for(auto pair : in_pre) {
                    pre_phi->add_phi_pair_operand(pair.first, pair.second);
                }
                loop_pre_bb->add_instr_begin(pre_phi);
                auto entry_phi = PhiInst::create_phi(phi->get_type(), loop_entry);
                for(auto pair : in_entry) {
                    entry_phi->add_phi_pair_operand(pair.first, pair.second);
                }
                entry_phi->add_phi_pair_operand(pre_phi, loop_pre_bb);
                phi->replace_all_use_with(entry_phi);
                loop_entry->delete_instr(phi);
                loop_entry->add_instr_begin(entry_phi);
            }
        }
    }
}
```

这一部分的完整代码见LoopCodeMotion.h和LoopCodeMotion.cpp。

活跃变量分析

活跃变量分析大体上按照课内所讲的数据流分析过程进行，但由于涉及到phi指令，需要对数据流方程做一点小修改。

对于phi指令中的操作数对[op, block]，op只在block的出口处是活跃变量，在这条phi指令所在基本块的其他前驱块出口处不是活跃变量。

因此，在扫描所有基本块中的所有指令时，需要使用一个map记录每个基本块中指令用到的操作数在哪些基本块出口可以是活跃变量。对于除了phi指令之外的指令，则每一个操作数在所有前驱基本块出口都可以是活跃变量。

对于数据流方程，计算IN[block]的过程不变，但计算OUT[block]时，对于后继块IN集合中的活跃变量，需要在map中查询它在block出口处是否是活跃变量，如果是，则将它加入OUT[block]。

这一部分的完整代码见ActiveVar.h和ActiveVar.cpp。

后端代码生成与优化

寄存器分配

寄存器分配采用的是线性扫描的寄存器分配方法：

```
void RegAlloc::walk_intervals() {

    /*you need to finish this function*/
    used_reg_interval.clear();
    for(auto current_it=interval_list.begin();current_it!=interval_list.end();current_it++){
        current = *current_it;
        //更新最新区间上端点位置
        cur_loc=current->range_list.front()->from;
        //尝试直接分配寄存器
        if(try_alloc_free_reg()){
            continue;
        }
        //尝试分配超出范围的寄存器
        else if(try_alloc_outofuse_reg()){
            continue;
        }
        //尝试争夺其他变量的寄存器
        else if(try_alloc_leastimportant_reg()){
            continue;
        }
        else{
            //分配失败
            current->reg_num = -1;
        }
    }
}
```

具体地，我们按照变量的活跃区间的左端点进行升序排列，对于按序到来的区间，我们依次尝试直接分配寄存器、分配已经退出活跃区间的寄存器、尝试争夺其他变量的寄存器，如果以上三类操作均失败，则令当前变量的寄存器编号为-1，表示将其分配到栈上。

值得一提的是，每次为变量分配寄存器时，我们都会用该变量的活跃区间的左端点更新行进的位置cur_loc。当没有可以直接分配的寄存器后，我们会用cur_loc判断是否存在已经退出活跃区间的寄存器，如果存在，则将其分配给当前变量。

此外，我们维护了used_reg_interval优先队列，它按照区间结束的先后顺序排序。当不存在空闲寄存器时，我们会使用used_reg_interval的队首，即最后退出活跃区间的寄存器，来分配给当前变量。（这是因为将寄存器分配给当前变量可以为其他变量留出更大的分配到寄存器的可能）

phi指令数据搬移

按照Phi指令的语义，我们需要在 `data_move` 函数中并行地将 `src` 搬移到 `dst` 上。

我们将 `src` , `dst` 中所有变量（包括寄存器、常数、栈地址），视作图中的结点，并为每对 `src``dst` 连边，使得 `src` 指向 `dst` ，从而构建出一张依赖图。可以注意到，图中没有出边的顶点是安全的，即可以被赋值；有出边的结点则要等待到所有出边的赋值均完成后才能被赋值。

我们观察到，一次phi指令的数据搬移不存在一个变量被赋值两次。即所有节点的出度至多为一，通过数学推导可以证明，该依赖图在依次删除所有出度为0的结点后，剩下的图为若干环的并集。

```
std::string CodeGen::data_move(PtrVec<IR2asm::Location> &src,
                               PtrVec<IR2asm::Location> &dst,
                               std::string cmpop){
    std::string code;

    /* TODO: put your code here */
    //寻找空闲寄存器（该方法中需要最多一个永久空闲的临时寄存器）
    auto reg_tmp=Ptr<IR2asm::Reg>(new IR2asm::Reg(12));
    //建立有向依赖图
    depend_graph.clear();
    depend_graph=create_dep_graph(src,dst);
    //由于限制，所有顶点都至多有一条出边，所以，按逆拓扑先处理无出度顶点后，剩余图为若干圈的并集
    //为没有出边的顶点赋值
    std::set<Ptr<IR2asm::Location>,cmp_out_num> dec_out_vec;
    for(auto it=depend_graph.begin();it!=depend_graph.end();it++){
        dec_out_vec.insert(it->first);
    }
    auto cur_dst=*(dec_out_vec.begin());
    while(cur_dst!=nullptr&&depend_graph[cur_dst].size()==1){
        //找到了无出度的顶点
        auto cur_src=depend_graph[cur_dst][0];
        if(cur_src!=nullptr){
            //有入度
            code+=single_data_move(cur_src,cur_dst,reg_tmp,cmpop);
            PtrVec<IR2asm::Location>& src_out_list=depend_graph[cur_src];
            for(auto it=src_out_list.begin();it!=src_out_list.end();it++){
                if(*it==cur_dst){
                    src_out_list.erase(it);
                    cur_src->out_deg--;
                    break;
                }
            }
        }else{
            //无入度
        }
        depend_graph.erase(cur_dst);
        dec_out_vec.erase(dec_out_vec.begin());
        if(cur_src!=nullptr){
            for(auto it=dec_out_vec.begin();it!=dec_out_vec.end();it++){
                if(*it==cur_src){
                    dec_out_vec.erase(it);
                    break;
                }
            }
            dec_out_vec.insert(cur_src);
        }
        if(dec_out_vec.empty())
            cur_dst=nullptr;
        else
            cur_dst=*(dec_out_vec.begin());
    }
    //获取空闲寄存器
    auto reg_tmp_2=Ptr<IR2asm::Location>(new IR2asm::RegLoc(10));//To Be Opmized(可以入栈)
    //找环，直到所有顶点都被赋值
    while(!depend_graph.empty()){
        //找到一个环(1<-2<-3<-4<-...)
        std::set<Ptr<IR2asm::Location>> cur_circle;
        auto cur_dst=depend_graph.begin()->first;
        auto cur_src=depend_graph[cur_dst][0];
        cur_circle.insert(cur_dst);
        auto start=cur_dst;
        while(cur_src!=start){
            cur_circle.insert(cur_src);
            cur_dst=cur_src;
            cur_src=depend_graph[cur_dst][0];
```

```
        }
        //处理环
        auto it=cur_circle.begin();
        auto last=it++;
        code+=single_data_move(*last,reg_tmp_2,reg_tmp,cmpop);
        for(;it!=cur_circle.end();it++){
            code+=single_data_move(*it,*last,reg_tmp,cmpop);
            last=it;
        }
        code+=single_data_move(reg_tmp_2,*last,reg_tmp,cmpop);
        //删除环
        for(auto it=cur_circle.begin();it!=cur_circle.end();it++){
            depend_graph.erase(*it);
        }
    }
    return code;
}
```

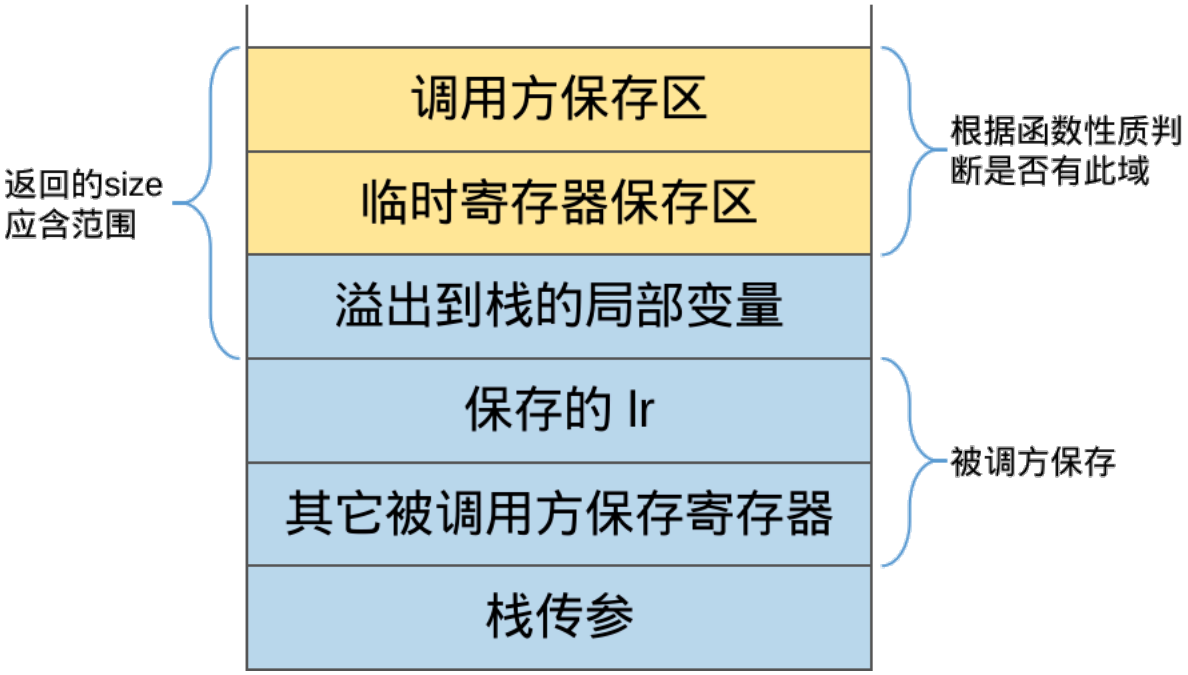
我们预先分配了11,12号寄存器作为临时寄存器。我们使用拓扑排序，将节点按出度大小升序排列，得到队列 dec_out_vec ，每次取出队首，若其出度为0，则可以直接赋值；否则，说明出度为0的结点已经全部赋值完毕。

找环过程中，我们首先从任意点出发，通过DFS找到一个环，然后将环中的点依次赋值，这样就完成了一次phi指令的数据搬移。

栈分配

```
int CodeGen::stack_space_allocation(Ptr<Function>fun)
{
    int size = 0;

    // std::map<Ptr<Value>, Interval *> CodeGen::reg_map
    auto _reg_map = &reg_map;   // Hint: use this to get register for values

    // std::map<Ptr<Value>, Ptr<IR2asm::Regbase>> CodeGen::stack_map
    stack_map.clear();          // You need to fill in this container to finish allocation

    // std::vector<Ptr<IR2asm::Regbase>> CodeGen::arg_on_stack (未使用, 不必要维护)
    arg_on_stack.clear();       // You need to maintain this information, the order is the same as parameter

    /* TODO: put your code here */
    int offset=0;

    //临时变量分配
    if(have_temp_reg){
        offset+=temp_reg_store_num*reg_size;
    }
    //函数调用
    if(have_func_call){
        offset+=caller_saved_reg_num*reg_size;
    }

    //局部数组分配（遍历alloca语句）

    for(auto block:fun->get_basic_blocks()){
        for(auto inst:block->get_instructions()){
            if(inst->is_alloca()){
                stack_map[inst] = Ptr<IR2asm::Regbase>(new IR2asm::Regbase(IR2asm::Reg(IR2asm::sp), offset));
                auto alloca_inst=dynamic_pointer_cast<AllocaInst>(inst);
                offset += alloca_inst->get_alloca_type()->get_size();
            }
        }
    }
    //溢出的局部变量分配(_reg_map中)
    for(auto reg: *_reg_map){
        if(reg.second->reg_num==-1){
            stack_map[reg.first] = Ptr<IR2asm::Regbase>(new IR2asm::Regbase(IR2asm::Reg(IR2asm::sp), offset));
            offset += reg.first->get_type()->get_size();
        }
    }

    //参数分配
    int stack_arg_offset=(used_reg.second.size()+1)*reg_size+offset;
    for(auto arg:fun->get_args()){
        if(arg->get_arg_no()<4){
            stack_map[arg]=Ptr<IR2asm::Regbase>(new IR2asm::Regbase(IR2asm::Reg(IR2asm::sp),offset));
            offset+=reg_size;
        }
        else if(arg->get_arg_no()>=4){
            stack_map[arg]=Ptr<IR2asm::Regbase>(new IR2asm::Regbase(IR2asm::sp,stack_arg_offset));
            stack_arg_offset+=arg->get_type()->get_size();
        }
    }

    size=offset;
    //返回分配的总空间的大小（字节数）
    return size;
}
```

在函数调用后，按照ARM标准组织栈空间。具体如下图：



评测结果

中间代码优化

对中间代码优化的所有测试均使用clang的后端，优化等级为O0。

正确性

仅开启Mem2Reg优化的测试结果：

```
lml@VServer:~/SysYF_optimize/test$ python3 correctness.py
IR Optimization Level: -O
==========TEST START==========
now in ./Test_H/Easy_H/
        Success in dir ./Test_H/Easy_H/
===========TEST END===========
==========TEST START==========
now in ./Test_H/Medium_H/
        Success in dir ./Test_H/Medium_H/
===========TEST END===========
==========TEST START==========
now in ./Test_H//Hard_H/
        Success in dir ./Test_H//Hard_H/
===========TEST END===========
==========TEST START==========
now in ./Test/Easy/
        Success in dir ./Test/Easy/
===========TEST END===========
==========TEST START==========
now in ./Test/Medium/
        Success in dir ./Test/Medium/
===========TEST END===========
==========TEST START==========
now in ./Test/Hard/
        Success in dir ./Test/Hard/
===========TEST END===========
        All Tests Passed
```

开启全部优化的测试结果：

```
lml@VServer:~/SysYF_optimize/test$ python3 correctness.py
IR Optimization Level: -O2
==========TEST START==========
now in ./Test_H/Easy_H/
        Success in dir ./Test_H/Easy_H/
============TEST END============
==========TEST START==========
now in ./Test_H/Medium_H/
        Success in dir ./Test_H/Medium_H/
============TEST END============
==========TEST START==========
now in ./Test_H//Hard_H/
        Success in dir ./Test_H//Hard_H/
============TEST END============
==========TEST START==========
now in ./Test/Easy/
        Success in dir ./Test/Easy/
============TEST END============
==========TEST START==========
now in ./Test/Medium/
        Success in dir ./Test/Medium/
============TEST END============
==========TEST START==========
now in ./Test/Hard/
        Success in dir ./Test/Hard/
============TEST END============
        All Tests Passed
```

可以看到，采取的优化手段没有对正确性产生影响。

运行时间

为体现优化效果，选择了 `test/Test_H/Hard_H` 目录下的所有样例进行测试。这一部分样例大多需要较长的运行时间。

测试在CPU为i9 14900k，内存为64GB的机器上进行。操作系统为Ubuntu 22.04.3 LTS。

对 `test/Test_H/Hard_H` 目录下的所有样例，不开启优化的运行时间：

```
lml@VServer:~/SysYF_optimize/test$ python3 test.py
IR Optimization Level: -O0
===========TEST START===========
now in ./Test_H//Hard_H/
Case bitset:    Pass    Time:   ir:0.0074s      exegen:0.0386s  exe:0.9884s
Case conv:      Pass    Time:   ir:0.0032s      exegen:0.0366s  exe:8.2297s
Case expr_eval: Pass    Time:   ir:0.0069s      exegen:0.0406s  exe:0.0014s
Case fft:       Pass    Time:   ir:0.0029s      exegen:0.0388s  exe:5.1890s
Case kmp:       Pass    Time:   ir:0.0022s      exegen:0.0359s  exe:0.0008s
Case long_code: Pass    Time:   ir:0.0052s      exegen:0.0364s  exe:0.0010s
Case nested_calls:      Pass    Time:   ir:0.0025s      exegen:0.0355s  exe:0.0008s
Case percolation:       Pass    Time:   ir:0.0024s      exegen:0.0338s  exe:0.0006s
Case scope2:    Pass    Time:   ir:0.0013s      exegen:0.0325s  exe:0.0010s
Case side_effect:       Pass    Time:   ir:0.0016s      exegen:0.0343s  exe:0.0009s
        Success in dir ./Test_H//Hard_H/
============TEST END============
        All Tests Passed
Total IRBuild time cost: 0.0356s
Total ExeGen time cost: 0.3630s
Total Exe time cost: 14.4135s
```

对 `test/Test_H/Hard_H` 目录下的所有样例，仅开启Mem2Reg优化的运行时间：

```
lml@VServer:~/SysYF_optimize/test$ python3 test.py
IR Optimization Level: -O
===========TEST START===========
now in ./Test_H//Hard_H/
Case bitset:    Pass    Time:   ir:0.0078s      exegen:0.0351s  exe:0.8559s
Case conv:      Pass    Time:   ir:0.0042s      exegen:0.0350s  exe:7.5827s
Case expr_eval: Pass    Time:   ir:0.0091s      exegen:0.0360s  exe:0.0012s
Case fft:       Pass    Time:   ir:0.0035s      exegen:0.0340s  exe:6.0951s
Case kmp:       Pass    Time:   ir:0.0027s      exegen:0.0346s  exe:0.0007s
Case long_code: Pass    Time:   ir:0.0077s      exegen:0.0342s  exe:0.0008s
Case nested_calls:      Pass    Time:   ir:0.0029s      exegen:0.0340s  exe:0.0009s
Case percolation:       Pass    Time:   ir:0.0031s      exegen:0.0357s  exe:0.0007s
Case scope2:    Pass    Time:   ir:0.0015s      exegen:0.0333s  exe:0.0007s
Case side_effect:       Pass    Time:   ir:0.0017s      exegen:0.0351s  exe:0.0007s
        Success in dir ./Test_H//Hard_H/
============TEST END============
        All Tests Passed
Total IRBuild time cost: 0.0442s
Total ExeGen time cost: 0.3469s
Total Exe time cost: 14.5395s
```

对 `test/Test_H/Hard_H` 目录下的所有样例，开启全部优化后的运行时间：

```
lml@VServer:~/SysYF_optimize/test$ python3 test.py
IR Optimization Level: -O2
==========TEST START==========
now in ./Test_H//Hard_H/
Case bitset:    Pass    Time:   ir:0.0100s      exegen:0.0361s  exe:0.6947s
Case conv:      Pass    Time:   ir:0.0143s      exegen:0.0376s  exe:7.4084s
Case expr_eval: Pass    Time:   ir:0.1102s      exegen:0.0361s  exe:0.0010s
Case fft:       Pass    Time:   ir:0.0060s      exegen:0.0353s  exe:4.6495s
Case kmp:       Pass    Time:   ir:0.0043s      exegen:0.0367s  exe:0.0007s
Case long_code: Pass    Time:   ir:0.0167s      exegen:0.0359s  exe:0.0006s
Case nested_calls:      Pass    Time:   ir:0.0041s      exegen:0.0337s  exe:0.0006s
Case percolation:       Pass    Time:   ir:0.0147s      exegen:0.0363s  exe:0.0007s
Case scope2:    Pass    Time:   ir:0.0018s      exegen:0.0334s  exe:0.0007s
Case side_effect:       Pass    Time:   ir:0.0022s      exegen:0.0342s  exe:0.0008s
        Success in dir ./Test_H//Hard_H/
==========TEST END==========
        All Tests Passed
Total IRBuild time cost: 0.1842s
Total ExeGen time cost: 0.3553s
Total Exe time cost: 12.7577s
```

如果仅开启Mem2Reg优化，则总的运行时间并无太大变化，甚至略微增加。观察每个样例的运行时间后发现主要是fft样例运行时间增加导致。对于Mem2Reg导致运行时间增加的原因，我们分析生成的中间代码后并未发现明显异常。推测是因为fft样例的大部分操作代价在数组上，因此单纯的Mem2Reg优化并未起到明显的优化效果（Mem2Reg不会影响数组的访问），反而可能因为引入phi指令影响最终的运行效率。但同时Mem2Reg为后续的优化提供了可能（比如产生了大量的新的常量表达式），且后端代码生成也依赖于此，因此我们仍然将其作为优化的基础。

如果开启全部优化，则总的运行时间与前两者相比均有较为明显的减少。这证明对中间代码的优化是有效的。

对不同优化Pass效果的测试

下面的所有测试都默认开启了Mem2Reg优化。

仅开启函数性质分析与死代码删除：

```
lml@VServer:~/SysYF_optimize/test$ python3 test.py
IR Optimization Level: -O2
==========TEST START==========
now in ./Test_H//Hard_H/
Case bitset:    Pass    Time:   ir:0.0095s      exegen:0.0347s  exe:0.8521s
Case conv:      Pass    Time:   ir:0.0127s      exegen:0.0379s  exe:7.5741s
Case expr_eval: Pass    Time:   ir:0.0173s      exegen:0.0396s  exe:0.0012s
Case fft:       Pass    Time:   ir:0.0058s      exegen:0.0338s  exe:6.0905s
Case kmp:       Pass    Time:   ir:0.0041s      exegen:0.0364s  exe:0.0006s
Case long_code: Pass    Time:   ir:0.0144s      exegen:0.0339s  exe:0.0007s
Case nested_calls:      Pass    Time:   ir:0.0040s      exegen:0.0374s  exe:0.0008s
Case percolation:       Pass    Time:   ir:0.0111s      exegen:0.0353s  exe:0.0008s
Case scope2:    Pass    Time:   ir:0.0017s      exegen:0.0361s  exe:0.0007s
Case side_effect:       Pass    Time:   ir:0.0021s      exegen:0.0349s  exe:0.0007s
        Success in dir ./Test_H//Hard_H/
==========TEST END==========
        All Tests Passed
Total IRBuild time cost: 0.0827s
Total ExeGen time cost: 0.3601s
Total Exe time cost: 14.5220s
```

仅开启常量折叠与传播、函数性质分析和死代码删除：

```
lml@VServer:~/SysYF_optimize/test$ python3 test.py
IR Optimization Level: -O2
==========TEST START==========
now in ./Test_H//Hard_H/
Case bitset:    Pass    Time:   ir:0.0094s      exegen:0.0373s  exe:0.6838s
Case conv:      Pass    Time:   ir:0.0117s      exegen:0.0366s  exe:7.5628s
Case expr_eval: Pass    Time:   ir:0.0170s      exegen:0.0355s  exe:0.0010s
Case fft:       Pass    Time:   ir:0.0057s      exegen:0.0367s  exe:6.0888s
Case kmp:       Pass    Time:   ir:0.0040s      exegen:0.0367s  exe:0.0006s
Case long_code: Pass    Time:   ir:0.0143s      exegen:0.0352s  exe:0.0006s
Case nested_calls:      Pass    Time:   ir:0.0038s      exegen:0.0360s  exe:0.0007s
Case percolation:       Pass    Time:   ir:0.0108s      exegen:0.0352s  exe:0.0008s
Case scope2:    Pass    Time:   ir:0.0017s      exegen:0.0378s  exe:0.0007s
Case side_effect:       Pass    Time:   ir:0.0022s      exegen:0.0344s  exe:0.0007s
        Success in dir ./Test_H//Hard_H/
============TEST END============
        All Tests Passed
Total IRBuild time cost: 0.0805s
Total ExeGen time cost: 0.3615s
Total Exe time cost: 14.3404s
```

开启常量折叠与传播、函数性质分析、局部公共子表达式消除和死代码删除

```
lml@VServer:~/SysYF_optimize/test$ python3 test.py
IR Optimization Level: -O2
==========TEST START==========
now in ./Test_H//Hard_H/
Case bitset:    Pass    Time:   ir:0.0101s      exegen:0.0345s  exe:0.6847s
Case conv:      Pass    Time:   ir:0.0118s      exegen:0.0362s  exe:7.6111s
Case expr_eval: Pass    Time:   ir:0.1034s      exegen:0.0377s  exe:0.0014s
Case fft:       Pass    Time:   ir:0.0058s      exegen:0.0344s  exe:4.6788s
Case kmp:       Pass    Time:   ir:0.0040s      exegen:0.0365s  exe:0.0009s
Case long_code: Pass    Time:   ir:0.0153s      exegen:0.0354s  exe:0.0008s
Case nested_calls:      Pass    Time:   ir:0.0042s      exegen:0.0354s  exe:0.0008s
Case percolation:       Pass    Time:   ir:0.0111s      exegen:0.0339s  exe:0.0007s
Case scope2:    Pass    Time:   ir:0.0015s      exegen:0.0331s  exe:0.0008s
Case side_effect:       Pass    Time:   ir:0.0021s      exegen:0.0342s  exe:0.0008s
        Success in dir ./Test_H//Hard_H/
============TEST END============
        All Tests Passed
Total IRBuild time cost: 0.1692s
Total ExeGen time cost: 0.3512s
Total Exe time cost: 12.9808s
```

开启全部优化（在上面的基础上加上循环表达式外提）：

```
lml@VServer:~/SysYF_optimize/test$ python3 test.py
IR Optimization Level: -O2
==========TEST START==========
now in ./Test_H//Hard_H/
Case bitset:    Pass    Time:   ir:0.0100s      exegen:0.0361s  exe:0.6947s
Case conv:      Pass    Time:   ir:0.0143s      exegen:0.0376s  exe:7.4084s
Case expr_eval: Pass    Time:   ir:0.1102s      exegen:0.0361s  exe:0.0010s
Case fft:       Pass    Time:   ir:0.0060s      exegen:0.0353s  exe:4.6495s
Case kmp:       Pass    Time:   ir:0.0043s      exegen:0.0367s  exe:0.0007s
Case long_code: Pass    Time:   ir:0.0167s      exegen:0.0359s  exe:0.0006s
Case nested_calls:      Pass    Time:   ir:0.0041s      exegen:0.0337s  exe:0.0006s
Case percolation:       Pass    Time:   ir:0.0147s      exegen:0.0363s  exe:0.0007s
Case scope2:    Pass    Time:   ir:0.0018s      exegen:0.0334s  exe:0.0007s
Case side_effect:       Pass    Time:   ir:0.0022s      exegen:0.0342s  exe:0.0008s
        Success in dir ./Test_H//Hard_H/
============TEST END============
        All Tests Passed
Total IRBuild time cost: 0.1842s
Total ExeGen time cost: 0.3553s
Total Exe time cost: 12.7577s
```

可以看到，每一个优化Pass都有一定效果，且对不同样例而言，优化效果最显著的Pass也不尽相同。比如局部公共子表达式消除在fft样例上的效果最为显著，循环表达式外提在conv样例上最为明显，而常量折叠在bitset样例上起到了良好的效果。死代码删除单独使用时没有产生明显的优化，这是因为正常编写代码时并无太多死代码可供删除，但死代码删除可以使其他优化发挥出效果（其他优化会产生死代码，但不会立刻删除，而是在死代码删除时一并删去）。与此同时，在这些除Mem2Reg外的优化中，并没有观察到进行优化之后运行时间明显下降的现象。

总而言之，每一个优化Pass都是较为有效的，在合适的场景下能发挥出较为明显的效果。

开启各种优化的运行时间总结如下：

	样例运行时间(s)									
开启的优化Pass	bitset	conv	expr_eval	fft	kmp	long_code	nested_calls	percolation	scope2	side_effect
无优化	0.9884	8.2297	0.0014	5.1890	0.0008	0.0010	0.0008	0.0006	0.0010	0.000
仅开启Mem2Reg	0.8559	7.5827	0.0012	6.0951	0.0007	0.0008	0.0009	0.0007	0.0007	0.000
Mem2Reg+函数性质分析+死代码删除	0.8521	7.5741	0.0012	6.0905	0.0006	0.0007	0.0008	0.0008	0.0007	0.000
上面几项 +常量折叠与传播	0.6838	7.5628	0.0010	6.0888	0.0006	0.0006	0.0007	0.0008	0.0007	0.000
上面几项 +局部公共子表达式消除	0.6847	7.6111	0.0014	4.6788	0.0009	0.0008	0.0008	0.0007	0.0008	0.000
开启全部优化 （再加循环表达式外提）	0.6947	7.4084	0.0010	4.6495	0.0007	0.0006	0.0006	0.0007	0.0007	0.000

后端代码生成与优化

总共140个测试样例，均能成功生成汇编代码：

```
Case reverse_output:    IRBuild Success
Case side_effect:       IRBuild Success
Case skip_spaces:       IRBuild Success
Case sort_test1:        IRBuild Success
Case sort_test2:        IRBuild Success
Case sort_test3:        IRBuild Success
Case sort_test4:        IRBuild Success
Case sort_test5:        IRBuild Success
Case sort_test6:        IRBuild Success
Case time_prior_plus:   IRBuild Success
Case while_break_test:  IRBuild Success
Case while_if_test:     IRBuild Success
Case while_test:        IRBuild Success
        Success in dir ./int_only_test/Test/Medium/
============BUILD END============
===========TEST START===========
now in ./int_only_test/Test/Hard/
Case backpack:  IRBuild Success
Case brainfk:   IRBuild Success
Case expr_eval: IRBuild Success
Case kmp:       IRBuild Success
Case long_array:        IRBuild Success
Case n_queens:  IRBuild Success
Case nested_calls:      IRBuild Success
Case nested_calls2:     IRBuild Success
Case percolation:       IRBuild Success
Case scope:     IRBuild Success
        Success in dir ./int_only_test/Test/Hard/
============BUILD END============
        All Tests Passed
        Pass Rate: 140/140
```

鉴于时间原因，截至2024年1月22日22:00前，我们共通过了133/140个测试。

```
 Case side_effect:       Pass
 Case skip_spaces:       Pass
 Case sort_test1:        Pass
 Case sort_test2:        Pass
 Case sort_test3:        Pass
 Case sort_test4:        Pass
 Case sort_test5:        Pass
 Case sort_test6:        Pass
 Case time_prior_plus:   Pass
 Case while_break_test:  Pass
 Case while_if_test:     Pass
 Case while_test:        Pass
         Fail in dir ./test/int_only_test/Test/Medium/
 ============TEST END============
 ===========TEST START===========
 now in ./test/int_only_test/Test/Hard/
 Case backpack:  Wrong Answer
 Case brainfk:   Pass
 Case expr_eval: Pass
 Case kmp:       Pass
 Case long_array:        Pass
 Case n_queens:  Pass
 Case nested_calls:      Pass
 Case nested_calls2:     Pass
 Case percolation:       Pass
 Case scope:     Pass
         Fail in dir ./test/int_only_test/Test/Hard/
 ============TEST END============
         Test Fail in dirs ./test/int_only_test/Test_H/Medium_H/ ./test/int_only_test/Test/Hard/ ./test/int_only_test/Test_H/Ha
 rd_H/   ./test/int_only_test/Test/Medium/
         Pass Rate: 133/140
```

2024年1月22日5:25，我们通过了全部的测试。

```
Case reverse_output:    Pass
Case side_effect:       Pass
Case skip_spaces:       Pass
Case sort_test1:        Pass
Case sort_test2:        Pass
Case sort_test3:        Pass
Case sort_test4:        Pass
Case sort_test5:        Pass
Case sort_test6:        Pass
Case time_prior_plus:   Pass
Case while_break_test:  Pass
Case while_if_test:     Pass
Case while_test:        Pass
        Success in dir ./test/int_only_test/Test/Medium/
============TEST END============
===========TEST START===========
now in ./test/int_only_test/Test/Hard/
Case backpack:  Pass
Case brainfk:   Pass
Case expr_eval: Pass
Case kmp:       Pass
Case long_array:        Pass
Case n_queens:  Pass
Case nested_calls:      Pass
Case nested_calls2:     Pass
Case percolation:       Pass
Case scope:     Pass
        Success in dir ./test/int_only_test/Test/Hard/
============TEST END============
        All Tests Passed
        Pass Rate: 140/140
```

至此，我们的编译大作业圆满结束，**轻舟已过万重山**！