

中山大学计算机院本科生实验报告

(2023 学年春季学期)

课程名称：编译原理实验

批改人：

实验	Task4 – LLVM IR 优化	专业（方向）	计算机科学与技术
学号	21307049	姓名	周育林
Email	zhouyulin23@mail2.sysu.edu.cn	完成日期	2024/6/22

1. 实验过程和核心代码

1) 常量传播

原理：直接使用常量代替变量，减少程序的存储和访存开销。

实现方式：

- 标记被写入过的全局变量，识别程序中可以被当作常量的全局变量（未被写入过的全局变量）
- 替换未被写入的全局变量，实现常量传播
- 删除被折叠为常量的指令

```
// 1. 标记被写入的全局变量
for (auto& func : mod) {
    // 遍历每个函数的基本块
    for (auto& bb : func) {
        // 遍历每个基本块的指令
        for (auto& inst : bb) {
            if (StoreInst *storeInst = dyn_cast<StoreInst>(&inst)) {
                // Value *StoreInst::getPointerOperand() { return getOperand(1); }

                Value *ptrOperand = storeInst->getPointerOperand();
                if (GlobalVariable *GV = dyn_cast<GlobalVariable>(ptrOperand)) {
                    isWrote[GV] = true;
                }
            }
        }
    }
}
```

```
std::vector<Instruction*> instToErase;
// 2. 替换未被写入的全局变量
for (GlobalVariable &GV : mod.globals())
{
```

```

    if (GV.isConstant() || isWrote[&GV]) continue;
    else if (GV.getValueType()->isArrayTy()) {
        // TODO 处理全局数组
        continue;
    }
    else {
        if (GV.hasInitializer()) // 如果全局变量有初始值
        {
            Constant* init = GV.getInitializer();
            for (Use &use : GV.uses()) {
                User *user = use.getUser();
                if (LoadInst *loadInst = dyn_cast<LoadInst>(user)) { // 如果使用者是
Load 指令
                    loadInst->replaceAllUsesWith(init);
                    instToErase.push_back(loadInst);
                    ++constPropagateTimes;
                } else if (Instruction *inst = dyn_cast<Instruction>(user)) { // 如
果使用者是其他指令
                    inst->replaceUsesOfWith(&GV, init);
                    ++constPropagateTimes;
                }
            }
        }
    }
}

// 统一删除被折叠为常量的指令
for (auto& i : instToErase)
    i->eraseFromParent();

```

2) 常量折叠

原理：通过检测和计算编译时能确定的常量表达式，将计算结果直接替换到程序中，避免运行时重复计算，提高程序执行效率。

实现方式：

- i. 判断当前指令是否为二元运算指令
- ii. 获取该指令的操作数并尝试转换为常整数
- iii. 若二元运算的指令的左右操作数都是常整数，则根据运算符号进行常量折叠和替换

```
for (auto& func : mod) {
    for (auto& bb : func) {
        std::vector<Instruction*> instToErase;
        // 遍历每个基本块的指令
        for (auto& inst : bb) {
            // 判断当前指令是否是二元运算指令
            if (auto binOp = dyn_cast<BinaryOperator>(&inst)) {
                // 获取二元运算指令的左右操作数，并尝试转换为常整数
                Value* lhs = binOp->getOperand(0);
                Value* rhs = binOp->getOperand(1);
                auto constLhs = dyn_cast<ConstantInt>(lhs);
                auto constRhs = dyn_cast<ConstantInt>(rhs);
                switch (binOp->getOpcode()) {
                    case Instruction::Add: {
                        // 若左右操作数均为整数常量，则进行常量折叠与 use 替换
                        if (constLhs && constRhs) {
                            binOp->replaceAllUsesWith(ConstantInt::getSigned(
                                binOp->getType(),
                                constLhs->getSExtValue() + constRhs->getSExtValue()));
                            instToErase.push_back(binOp);
                            ++constFoldTimes;
                        }
                        break;
                    }
                    .....

                    default:
                        break;
                }
            }
        }
        // 统一删除被折叠为常量的指令
        for (auto& i : instToErase)
            i->eraseFromParent();
    }
}
```

3) 强度削减

原理：

将一条高计算复杂度的指令，转化为一条或多条低复杂度的指令，从而提高程序的运行效率。

实现方式：

- i. 判断当前指令是否为二元运算指令
- ii. 获取二元运算指令的左右操作数
- iii. 根据指令类型和左右操作数的情况分别进行处理，实现强度削减

```
// 遍历所有函数
for (auto& func : mod) {
    // 遍历每个函数的基本块
    for (auto& bb : func) {
        std::vector<Instruction*> instToErase;
        // 遍历每个基本块的指令

        for (auto instIter = bb.begin(); instIter != bb.end(); ++instIter) {
            auto& inst = *instIter;

            if (auto binOp = dyn_cast<BinaryOperator>(&inst)) { // 如果指令是二元运
算
                Value* lhs = binOp->getOperand(0);
                Value* rhs = binOp->getOperand(1);
                auto constLhs = dyn_cast<ConstantInt>(lhs);
                auto constRhs = dyn_cast<ConstantInt>(rhs);

                if (binOp->getOpcode() == Instruction::Mul) {
                    // 处理乘法指令：如果左操作数是 0，则将所有使用替换为 0 常量
                    if (constLhs && constLhs->getSExtValue()==0) {
                        binOp->replaceAllUsesWith(ConstantInt::get(lhs->getType(), 0));
                        instToErase.push_back(binOp);
                        ++strengthReductionTimes;

                        // 如果左操作数是 1，则将所有使用替换为右操作数
                    } else if (constLhs && constLhs->getSExtValue()==1) {
                        binOp->replaceAllUsesWith(rhs);
                        instToErase.push_back(binOp);
                        ++strengthReductionTimes;

                        .....

                        // 如果左操作数是一个大于 0 且是 2 的幂的常数
                    } else if (constLhs && rhs->getType()->isIntegerTy()) {
                        auto intVal = constLhs->getSExtValue();
                        if (intVal>0 && (intVal & (intVal - 1))==0) {
```

```

        auto shamt = ConstantInt::get(lhs->getType(),
static_cast<uint64_t>(std::log2((double)intVal)));
        llvm::IRBuilder<> TheBuilder(&inst);
        auto newInst = TheBuilder.CreateShl(rhs, shamt,
"strengthReduction");
        binOp->replaceAllUsesWith(newInst);
        instToErase.push_back(binOp);
        ++strengthReductionTimes;
    }

    .....

    } else if (binOp->getOpcode() == Instruction::UDiv || binOp->getOpcode()
== Instruction::SDiv) {
        // 处理除法指令：如果左操作数是 0，则将所有使用替换为 0 常量
        if (constLhs && constLhs->getSExtValue()==0) {
            binOp->replaceAllUsesWith(ConstantInt::get(lhs->getType(), 0));
            instToErase.push_back(binOp);
            ++strengthReductionTimes;

            // 如果左操作数和右操作数都是整数常量，并且右操作数是大于 0 且是 2 的幂的常
数
            else if (constLhs && lhs->getType()->isIntegerTy() && constRhs &&
rhs->getType()->isIntegerTy()) {
                auto intValLhs = constLhs->getSExtValue();
                auto intVal = constRhs->getSExtValue();
                if (intValLhs>=0 && intVal>0 && (intVal & (intVal - 1))==0) {
                    auto shamt = ConstantInt::get(rhs->getType(),
static_cast<uint64_t>(std::log2((double)intVal)));
                    llvm::IRBuilder<> TheBuilder(&inst);
                    auto newInst = TheBuilder.CreateAShr(lhs, shamt,
"strengthReduction");
                    binOp->replaceAllUsesWith(newInst);
                    instToErase.push_back(binOp);
                    ++strengthReductionTimes;
                }
            }
        }

        .....

    }
}

// 统一删除标记的指令
for (auto& i : instToErase)
    i->eraseFromParent();
}
}

```

4) 公共子表达式消除 (CSE)

原理：检测在代码中多次出现且结果相同的表达式，将其计算结果保存起来，并在后续使用相同子表达式的地方直接使用已保存的结果，而不是重新计算。

实现方式：

- i. 对每个二元运算指令，在其所在基本块及其后继块中查找相同的二元运算指令。
- ii. 如果发现相同的子表达式，将后续出现的子表达式替换为第一次出现的计算结果，并将其标记为删除。
- iii. 遍历标记为删除的指令，并将它们从基本块中删除。

```
for (auto& func : mod) {
    for (auto& bb : func) {
        std::vector<Instruction*> instToErase;
        std::vector<BasicBlock*> succBlocks {&bb};

        for (auto instAIter = bb.begin(); instAIter != bb.end(); ++instAIter)
        {
            auto& instA = *instAIter;

            // 判断当前指令是否是二元运算指令
            if (auto temp = dyn_cast<BinaryOperator>(&instA)) {}
            else continue;

            // 如果指令已经在删除列表中，跳过
            if (std::find(instToErase.begin(), instToErase.end(), &instA) !=
instToErase.end()) continue;

            // 将基本块的后继块加入 succBlocks 向量中
            for (auto succBBIter = succ_begin(&bb); succBBIter !=
succ_end(&bb); ++succBBIter) {
                succBlocks.emplace_back(*succBBIter);
            }

            // 遍历当前基本块及其后继块
            for (auto succBB:succBlocks) {

                BasicBlock::iterator instBIterBegin;
                BasicBlock::iterator instBIterEnd;

                // 如果当前块与后继块相同，设置起始和结束迭代器
                if (&bb == succBB) {
                    instBIterBegin = std::next(instAIter);
                    instBIterEnd = bb.end();
                }
            }
        }
    }
}
```

```

    } else {
        instBIterBegin = succBB->begin();
        instBIterEnd = succBB->end();
    }

    auto instBIter = instBIterBegin;
    int i = 0;        // 限制遍历深度，防止无限循环
    while(true) {
        if (instBIter == instBIterEnd || i>=100) break;
        auto& instB = *instBIter;
        if (std::find(instToErase.begin(), instToErase.end(),
&instB) != instToErase.end()) continue;

        auto OpA = instA.getOpcode();
        auto OpB = instB.getOpcode();

        // 检查两个指令的操作码是否相同
        if (OpA == OpB) {
            auto binOpA = dyn_cast<BinaryOperator>(&instA);
            auto binOpB = dyn_cast<BinaryOperator>(&instB);
            if (binOpA && binOpB) {
                // 检查操作数是否相同
                if (binOpA->getOperand(0)==binOpB->getOperand(0)
&& binOpA->getOperand(1)==binOpB->getOperand(1)) {
                    binOpB->replaceAllUsesWith(binOpA);        //
用 binOpA 替换 binOpB 的所有使用
                    instToErase.push_back(binOpB);            //
标记 binOpB 为删除
                    CSECounts++;
                }
            }
        }

        ++instBIter;
        ++i;
    }
}

// 统一删除标记的指令
for (auto& i : instToErase)
    i->eraseFromParent();
}
}

```

5) 指令合并

原理：识别并合并可以简化的计算指令，将多条指令合并成一条指令，从而减少不必要的操作。

实现方式：

- i. 遍历每个基本块中的指令，如果当前指令已经在删除列表中，跳过。
- ii. 判断当前指令是否是二元运算指令。
- iii. 如果当前指令只有一个用户，并且该用户指令也是二元运算指令且为加法操作，同时其第一个操作数是当前指令，检查它们的第二个操作数是否都是常量。
- iv. 如果上述条件满足，创建一个新的加法指令，将用户指令的所有使用替换为新的加法指令，并将旧指令标记为删除。

```
for (auto& func : mod) {
    for (auto& bb : func) {
        std::vector<Instruction*> instToErase;
        for (auto instIter = bb.begin(); instIter != bb.end(); ++instIter) {
            auto& inst = *instIter;
            // 如果指令已经在删除列表中，跳过
            if (std::find(instToErase.begin(), instToErase.end(), &inst) !=
instToErase.end()) continue;
            // 确保当前指令不是基本块中的最后一条指令
            if (instIter != bb.end()) {
                // 判断当前指令是否是二元运算指令
                if (auto binOp = dyn_cast<BinaryOperator>(&inst)) {
                    if (binOp->hasOneUse()) { // 如果当前指令只有一个 user
                        if (auto user =
dyn_cast<BinaryOperator>(*binOp->user_begin())) { // 获取用户指令
                            // 如果 user 指令是加法指令并且它的第一个操作数是当前
指令
                                if (user->getOpcode() == Instruction::Add &&
user->getOperand(0) == binOp) {
                                    auto constVal =
dyn_cast<ConstantInt>(binOp->getOperand(1));
                                    auto userConstVal =
dyn_cast<ConstantInt>(user->getOperand(1));
                                    // 如果二元运算指令和用户指令的第二个 operand 都
是常量
                                    if (constVal && userConstVal) {
                                        // 计算新的常量值
                                        auto combinedConst =
ConstantInt::get(constVal->getType(),
constVal->getSExtValue()+userConstVal->getSExtValue());
                                        // 创建新的加法指令
                                        auto newBinInst =
BinaryOperator::Create(Instruction::Add, binOp->getOperand(0), combinedConst);
```



```
// 将 user 的所有 use 替换为新的加法指令
user->replaceAllUsesWith(newBinInst);
newBinInst->insertAfter(user);
instToErase.push_back(&inst);
instToErase.push_back(user);
}
}
}
}
}
}
}
for (auto& i : instToErase) {
    i->eraseFromParent();
}
}
```

2. 实验结果

提交 ID	提交时间	得分
1822	2024-06-22 16:52:39	1914 / 3600

排行榜分数

总分 52.69

提交排行

评测机结果