lab4：中间代码优化

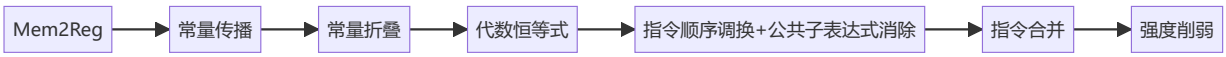
21307140 李明俊

一. 实验过程简述

本次实验的实验内容是在lab3的基础上实现一个LLVM IR优化器，对中间代码生成的结果进行优化。实验的输入与输出均为LLVM IR，要求在保证代码正确性的基础上面向给定测试样例进行代码优化。在本次实验中，通过实现多个Transform Pass对LLVM IR进行优化。

```
mpm.addPass(StaticCallCounterPrinter(llvm::errs()));
mpm.addPass(Mem2Reg());
mpm.addPass(ConstantPropagation(llvm::errs()));
mpm.addPass(ConstantFolding(llvm::errs()));
mpm.addPass(AlgebraicIdentities(llvm::errs()));
mpm.addPass(CommonSubexpression(llvm::errs()));
mpm.addPass(InstructionCombining(llvm::errs()));
mpm.addPass(StrengthReduction(llvm::errs()));
```

本次实验实现的Transform Pass及运行顺序如图所示



以下是在本次代码中实现的优化的简单思路，具体原理实验文档中有详细描述，这里不多赘述

常量传播：除了`const int`类型的变量外，某些`int`类型的变量也可以作为常量在代码中的传播。以全局变量为例：先遍历所有指令，如果是存储指令，则相关的变量不可以被视为常量，计入黑名单中，再遍历全局变量，将没有在黑名单中的变量记录在intlist，表明该变量可以视为常量，最后遍历所有指令，消除地址在intlist中的load指令，从而实现常量传播。

常量折叠：如果二元计算表达式（加减乘除等）的左右两边均为常量，则可以在编译阶段直接计算表达式的值，得到一个常量。实现常量的折叠。

代数恒等式：删除一些无用的或直接可以知道结果的的二元运算指令，本次实验代码实现了对含有：`+0` `0+` `-0` `%1` `0%` `/1` `0/` `*1` `*0` `1*` `0*`代数式的简化

指令顺序调换+公共子表达式消除：公共子表达式消除的原理很简单，对于每一条可能为公共子表达式的指令，遍历其后面一定数量`k`的指令，判断当前指令与后续指令是否计算相同的内容。如果计算内容相同，则直接替换删除重复指令。但是对于样例`hoist`的加法顺序无法直接使用公共子表达式消除，因此需要先调换一下指令的顺序，先将`i1`到`i15`相加，再将加法结果加到sum中（一个交换顺序的简单示例如下）。实现指令顺序调换后，就可以利用公共子表达式消除优化代码了

```
%add1 = add i32 %i1, %i2                %add1 = add i32 %i1, %i2     //公共子表达式1
%add2 = add i32 %add1, %i3              %add2 = add i32 %add1, %i3   //公共子表达式2
%add3 = add i32 %add2, %i1      ====>   %sum1 = add i32 %add2, 0
%add4 = add i32 %add3, %i2              %add3 = add i32 %i1, %i2     //可消除
%add5 = add i32 %add4, %i3              %add4 = add i32 %add3, %i3   //可消除
                                        %sum2 = add i32 %add4, %sum1
     没有公共子表达式                          有公共子表达式
```

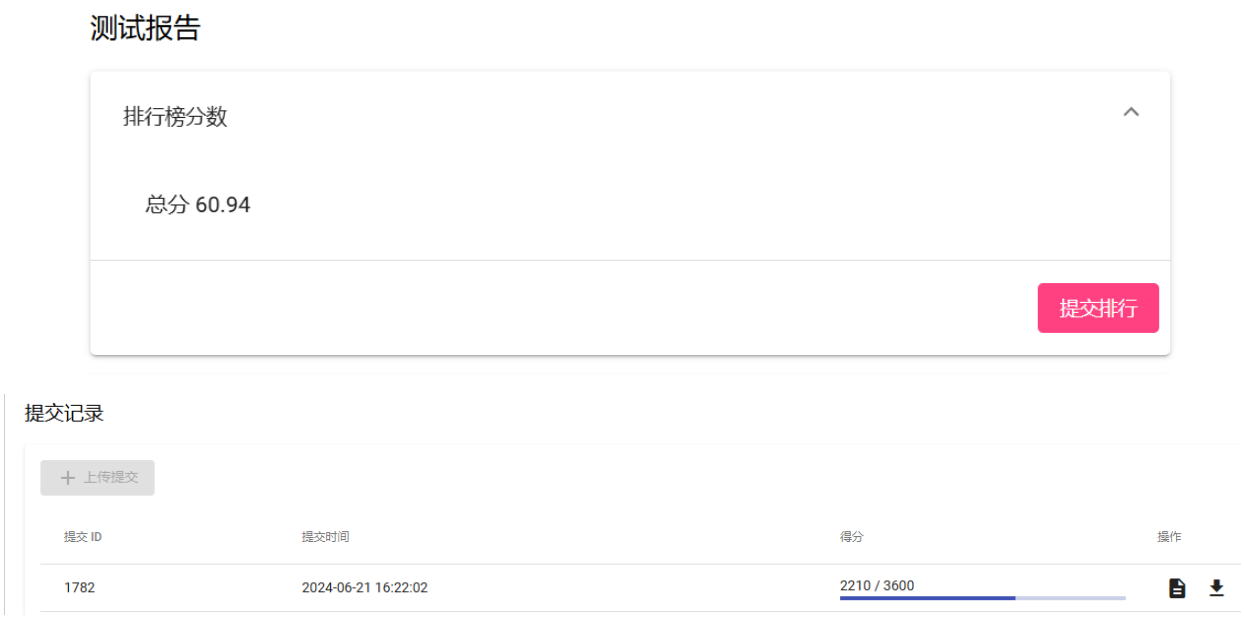
可以想象，在`hoist`样例中，在实现这种优化后，结果的最后会有很多连续的加法，每次加的数都等于`i1到i15的合`，这显然也是冗余的，这就可以通过接下来的指令合并来实现。

指令合并：指令合并可以将一连串的运算合并到一起运算，示例如下,代码中是通过判断使用关系，并用一个列表存储每次加的数字实现的

```
%add1 = add i32 %input, %value
%add2 = add i32 %add1, %value                %add1 = mul i32 %value, 5
%add3 = add i32 %add2, %value     ====>      %add2 = add i32 %input, %add1
%add4 = add i32 %add3, %value
%add5 = add i32 %add4, %value
```

强度削弱：代码中实现了将乘法换为左移和`/2`转换为右移一位的强度削弱

二. 实验运行结果



三. 感想与建议

本次实验上手体验了下对中间代码的简单的优化，让我对中间代码优化的过程和原理有了初步的了解，也学习了一系列的有关算法，整个实验做下来感觉收获很大。

有关实验的改进意见的话，我觉得lab4 performance测试样例有很多优化的空间，以下仅仅列出部分我在实验过程中想到的可以改进的地方：

1. 添加只针对公共子表达式删除的样例：我个人感觉`hoist`样例比较不错，需要同时做到指令顺序调换 + 公共子表达式删除 + 指令合并才可以拿到较高的接近满分的成绩，但是好像并没有直接可以大量使用公共子表达式的样例，因此导致这条优化路线门槛会高一点点，如果只实现公共子表达式分数变化不大
2. 适当添加一些确保优化完备性的样例：实验中对一些算法的测试样例有点简陋，因此会导致有很多tricky的空间，比如死代码消除的样例（由于分数到了60加上期末时间不大够，因此我没有实现这个优化，但是我在自己看这个样例的时候想到了一些tricky的优化），每一个死代码似乎只被另一个父亲死代码使用到（也就是不存在一个运算结果，既被非死代码用到，又被死代码用到），因此实现的时候可能通过从根节点处直接遍历就可以完成对死代码的消除，而不需要通过拓扑排序就可以拿到满分成绩。

这里只是举一个简单例子并不是针对这个样例，因此我觉得可以在原样例的基础上添加一些较为复杂的样例，以确保一些tricky的优化只能拿部分优化分而不是全部优化分。