Midpoint Report

Chenwei Xie , Jiaixiang Chen , Yue Wang

1 Collecting Static Instruction Counts

(1) Algorithm

This section mainly consists of 2 procedures: first is to gather information about all instructions found, and this should be done during the procedure of traveling all instructions. And second is to print out all instructions and their counts after traveling all instructions.For storing all information, we used a map to keep track of names of all instructions and how many found.

Since we need to analyze a program , we need to travel all modules , so we implemented our pass under ModulePass class, and override the function runOnModule. In the function runOnModule, we iterated all functions , all basic blocks and all instructions. getOpcodeName() is used to get instruction names of every instructions. So for every instructions , we first find if it is in our map, then to decide whether to add it to map or increase the number stored.

(2) Challenge

As this is the first problem in this project , it isn't so hard as other problems, and the major difficulty comes from the use of LLVM, since it is the first llvm pass we write , there are some concepts like module, function and basicblock that we are not familiar with.

2 Collecting Dynamic Instruction Counts

(1) Algorithm:

A basic block is a single-entry, single-exit section of code. So whether the program is running or not, the static analysis for a basic block is the same with the dynamic analysis for this block. The idea is to calculate the statistic for the basic block statically when the LLVM Pass optimize the target program and in the end of each basic block, the LLVM Pass we implement will insert a function call to a global function *merge*. When the program is running, every time the program comes across the inserted point, the program will call *merge* and the function will combine the statistic of this basic block to a global map. So probably a basic block can run several time, which means the statistic of this basic block will be merged into global map several time. Also in the termination of the program, we need to output the result. So when the LLVM Pass optimize the program, it also find the *main* function and then find the *return* statement and just before the *return* statement, LLVM Pass inserts a function call to *print* which can output the statistic result of this program. The two function *print* and ?merge? are put in a file called merge.cpp.

At high level, one program and merge.cpp are passed to clang, and clang generates their LLVM IR. And then, the LLVM Pass will optimize the program?s

LLVM IR. And then the compiler will link the modified program LLVM IR and merge LLVM IR together and then compile them into the executable.

(2) Challenge:

We use FunctionType::get to construct the type of function we will insert. And Module::getOrInsertFunction to get the function from the function library of the target program and IRBuilder:: CreateCall to insert a function call. The big challenge we encountered is to pass a parameter when insert a function call. It?s extremely hard to pass a string, so we pass the a int parameter to our *merge* function and in our *merge* function.

3 Profiling Branch Bias

(1) Algorithm: We have four pieces of inserting functions here, their function name and usage are listed as below: Counting all functions in this module. Counting all branch instruction in this module. Counting all taken instruction in this module. Print the result.

Our algorithm to insert the code is shown as below: First insert code in each basic block to count all functions, then for each instruction in basic, we judge whether it is a ?ret? instruction or it is a conditional ?br? instruction, if it is a ?ret? instruction, then we record it and insert the print result function latter, if it is a ?br? instruction, then we insert the code to count all branch instruction, at the same time, we should store all basic block label that the ?br? instruction might jump to. Finally, we do another loop to judge if a running basic block is from a ?br? instruction, if it is, then we insert code to count the taken instruction.

API used: isConditional(), getOperand(1), getName()

(2) Challenge:

How to judge if a "br" instruction execute or not. We try three different ways to solve the problems. Get the value of ?br? instruction directly In llvm::BranchInst, we find there is a member function called getCondition(). At first, we think that it will return a value to tell us whether the condition is true or false. Since the return type is Value*, we try to cast it to a integer and the get the value, however, after we try this and search the internet, we find this function actually return the expression of ?br? instruction, so we give up this way and try our second solution. 2. Evaluate the value of expression in ?br? Since we now have the expression of ?br? instruction by using getCondition(), we want to evaluate this expression to know if the instruction will be taken or not. We find there is a class called llvm::MCExpr and it has a function called evaluateAsAbsolute() that could get the value. So, we try to cast out expression to it and want to get the value. We failed that llvm tells us that this two class can not be cast. 3. Store the label of ?br? and count it later We think that we could first store all possible basicblock that ?br? instruction will go to, and when run the program, we could search if one running basicblock is from a ?br?

instruction. So, we use two loop here, one to store and one to count the actually taken basic blocks.

Another challenge comes from how to passing string values to functions to be inserted. At first we want to use getName() method the of function iterator to get the function name and then pass it to the function counting branches. However , string is not a valid type to be passed using llvm::Value* , since there is no such a type called stringTy in llvm::Value. Then we tried to pass a pointer standing for a C type string as argument, but the idea didn't work either, since the string in the pass file is not same as the string in the file to be combined with the target code file, so passing such a pointer will only lead to segment fault at last. Finally we found the solution , to use global variables to store strings. And to get the instructions needed to insert global variables, we write a simple cpp file containing only a definition of global string and then compile it to llvm instruction and then reconstruct it, to find how llvm store global variables.

4 pseudo code

Algorithm 1 Collecting Static Instruction Counts

```
for each function in the module do
  for each basicblock in the function do
    for each instruction in the basicblock do
      increase the number of corresponding instruction
    end for
  end for
end for
```

Algorithm 2 Collecting Dynamic Instruction Counts

```
for each function in the module do
  for each basicblock in the function do
    for each instruction in the basicblock do
      if the instruction is return and the function is main then
        insert a function call to print the statistic of the program
      end if
      increase the number of corresponding instruction
    end for
    insert a function call to merge the statistic into global map in the end
    of each basic block
  end for
end for
```

Algorithm 3 Profiling Branch Bias

```
for function in Module do
  for basicblock in function do
    for instruction in basicblock do
      if instruction is ret then
        insert print_result function
      end if
      if instruction is conditional br then
        getcondition value
        insert count_branch function
      end if
    end for
  end for
end for
```