EECS 583 – Homework 1

Winter 2024
Assigned: Wed, January 17, 2024
Due: Mon, January 29, 2024 (11:59pm Eastern USA)

Late Submission Policy

Submissions will be accepted a maximum of **two days** after the specified deadline. For each day late, a **10% penalty** will be deducted from your score.

Statistics Computation Pass

The goal of this homework assignment is to learn to write your first real LLVM pass. As part of this, you will learn to use the profiler which provides dynamic execution frequencies for the compiler to make use of.

Write a statistics computation pass in LLVM that computes several dynamic operation counts for each function. First, the total number of dynamic operations should be computed along with the percentages in the following categories: integer ALU, floating-point ALU, memory, biased-branch, unbiased branch, and all other operations. Use the following rules when categorizing the operations:

- Branch: br, switch, indirectbr
- Integer ALU: add, sub, mul, udiv, sdiv, urem, shl, lshr, ashr, and, or, xor, icmp, srem
- Floating-point ALU: fadd, fsub, fmul, fdiv, frem, fcmp
- Memory: alloca, load, store, getelementptr, fence, atomiccmpxchg, atomicrmw
- Others: everything else

Every operation should be placed in one of the above categories. Print this information to terminal in the following format (comma separated, one function in each line):

FuncName, DynOpCount, %IALU, %FALU, %MEM, %Biased-Br, %Unbiased-Br, %Others

FuncName is the name of the function, DynOpCount refers to the **dynamic** operation count, i.e. the total number of all instructions that were executed as part of that function, when we run the program. Note that this differs from **static** operation count, which is the number of instructions in the static IR of the source code. The remaining fields are all fractional values computed to the **third** decimal place. All fractional values are computed as the fraction of the targeted instruction type's dynamic instruction count that makes up the total dynamic operation count (e.g. %Unbiased-Br = unbiased_branch_instr_dyn_count/total_instr_dyn_count).

For example, if 50% of a function's operations are integer ALU operations, then its %IALU should be 0.500. You can use the following to print these values: #include "llvm/Support/Format.h" and errs() << format("%.3f", val). Print all zeros if a function has never been executed, i.e.

func_name, 0, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000

The bias of a branch represents the fraction of times that the branch is taken. For branches with only a single target, the branch bias is 100%. For a simple branch with no extra entries or exits, the branch bias is the ratio of the execution frequencies of the successor basic block, with the parent basic block. However, for more complicated CFGs, this simple formula may break down. Consider using LLVMs tools to check branch biases directly, instead of computing them yourself. For this assignment, branches are considered biased if the bias **> 80%** and unbiased otherwise.

To get started, download the compressed file from the course website. This file contains 3 benchmarks, skeleton code for your pass and a run script for running LLVM. To unpack the file, run tar -zxvf <file_name>.tgz

Benchmarks to Run

There are 3 benchmarks that you should profile and collect statistics for: *simple (benchmark1)*, *anagram (benchmark2)*, and *compress (benchmark3)*. Each is progressively larger and more complex. Each benchmark contains directories for the source code (src) and the input file (input).

The *simple* benchmark is just a C program that contains a single main function with loops and array accesses. The *anagram* benchmark takes a few input words and a dictionary of words and computes all possible anagrams of the input words. Note that this benchmark takes time to run (30–60 sec). The *compress* benchmark runs a compression algorithm on the given input file.

We recommend that you use the run script (run.sh) that we have provided to execute your pass on the benchmarks. Instructions for using the script are given in comments at the beginning of the script.

If running it manually remember to copy the input file to the same directory as your binary executable to ensure that your profile counts match ours. You can compile each benchmark with gcc to make sure they work before running LLVM. The compress benchmark may check if an output with the same name already exists (compress.in.Z). This will make a difference in program execution, and lead to different statistics. Therefore, please make sure you delete outputs from the previous run before collecting your statistics.

Server Submission

To submit your homework, put a single .tgz (gzipped tar file) into the directory **/hw1_submissions/** on **eecs583a.eecs.umich.edu** via scp (later versions will automatically overwrite the earlier versions if you submit multiple times):

```
$ tar cvzf ${uniquename}_hw1.tgz ${uniquename}_hw1
$ scp ${uniquename}_hw1.tgz ${uniquename}@eecs583a.eecs.umich.edu:/hw1_submissions/
```

Please name your tar file ${uniquename}_hw1.tgz, and organize the directory as follows:

```
${uniquename}_hw1/
    README
    src/
        hw1pass.cpp
    results/
        simple.opcstats
        anagram.opcstats
        compress.opcstats
```

Since you will be using cp/scp to submit, there is no validation on submitting. Use *ls* on the submission folder to verify that your submission is present.

Autograder Submission

When submitting to the autograder, only submit the hw1pass.cpp on Gradescope. Once you submit your code, it will be run on the three benchmarks discussed above. You will see whether the output produced by your code is what we expect. We will not reveal the correct solution, just whether the output of your code matches or not.

1. Ensure your pass prints out the statistics in the correct format on the **error stream: errs()**.
2. Ensure your pass does not print out any additional statements to that stream. Ensure that any print statements you add for the purpose of debugging have been removed.
3. The format for output needs to be followed exactly, otherwise the test case will fail.
4. Verify that your code works locally or on the server before uploading to Gradescope. Because of the way the testing is happening under the hood, you may not see all the errors that occur if your code, say, fails to compile, which can make debugging only using autograder output impossible.
5. The score the autograder gives you may not be the final score, we may change it based on the severity of the mistakes made (which means that even if you get 0 on the autograder, you can get a high score, 90+, if your errors are judged by us to be minor). You should definitely submit your code via the server method as well in that case.
6. Since we are still experimenting with autograder for this course, the Server method of submission mentioned above is also a valid way of submitting.

Please contact the GSIs in case of any issues.