# EECS 583 – Homework 1

Fall 2025
Assigned: Wed, September 3, 2025
Due: Mon, September 15, 2025 (11:59pm Eastern USA)

## Late Submission Policy

Submissions are 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 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. Additionally, calculate the percentage of static instructions and dynamic instructions that are contained within innermost loops (more on this below). Print this information out in a text file named **benchmark.opcstats** in the following format (comma separated, one function in each line):

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

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 ran the program. Note that this differs from the Static Operation count, which will be the number of instructions present in the static IR of the program. The next six fields are percentage values computed to the third decimal place. All percentage values are computed as the targeted instruction type over the DynOpCount (e.g. %Biased-Br = biased_branch_inst_count / total_dyn_inst_count). The penultimate field (%StaticLoop) is the total number of instructions (static) that are within an innermost loop (that is, within a loop which does not have loops within it) divided by the static instruction count of the function. Similarly, %DynamicLoop is the number of dynamic instructions within innermost loops divided by the dynamic instruction count of the function. A function can have more than one innermost loop - note that in this case the numerators for %StaticLoop and %DynamicLoop are the sums of static/dynamic instruction counts within all inner loops in the function.

For example, if 50% of the 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 for the dynamic ratios if a function has never been executed, i.e. (func_name, 0, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, %StaticLoop, 0.000)

The bias of a branch is: (frequency_of_most_likely_target / total_execution_frequency). For branches with only a single target, the branch bias is considered to be 100%. 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 code 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.

## 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 using 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 difficult.
5. The score the autograder gives you may not be the final score and may be changed 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 if your errors are judged to be minor).