6.823
Computer System Architecture
Lab 0

Assigned Feb. 7, 2020 ***Due Feb. 14, 2020***

http://csg.csail.mit.edu/6.823/

This term in 6.823 we will be using Pin, a binary instrumentation tool provided by Intel. The purpose of this lab is to familiarize you with compiling and running Pintools, as well as to make sure that you are able to use the class infrastructure and computers. To test the infrastructure, you will build and run two small Pintools.

In order to develop architectures that run programs efficiently, architects must thoroughly understand the properties of representative programs. Architects experiment with different programs to help them make design decisions about features that will be included in processors. There are a few approaches that can be used to evaluate the behavior a program. One is simulation. In this case, a software model of a processor is built, and the program executed on the model. Simulators have the advantage of being arbitrarily detailed – in theory one could build a SPICE processor simulator. Typically, architectural simulators give cycle-accurate timing estimates. The penalty for this level of detail is simulation speed: the more detailed a software simulator is, the slower it executes programs; advanced software simulators can simulate at rates of tens of KIPS (kilo instructions per second). A second option is program instrumentation. Instrumentation inserts code into a program to collects information about program characteristics. Code instrumentation may be less detailed than simulation, but is often faster to implement and enjoys faster program execution. Thus, code instrumentation can be very useful in guiding architectural decisions early in the development process, before detailed simulators are available. You've almost certainly used a simple form of manual instrumentation, by inserting print statements into a program to generate a log of program activity. However, manual instrumentation is time-consuming if you want to record the characteristics of complex programs, both in terms of writing code and collecting execution results. A better choice is to use a meta language to describe the code instrumentation and develop a tool that will efficiently instrument the target program at compile or runtime.

Pin is an industrial-grade binary instrumentation tool produced by Intel and used widely in industry and academia. Pin accepts as inputs a compiled Pintool and a generic binary executable. A Pintool consists of C++ code that tells Pin where to insert code (instrumentation) and what code to insert (analysis). At runtime, executable itself is just-in-time recompiled by Pin with the analysis inserted. The code is then executed on the host machine. A major advantage of Pin is that it can instrument programs without requiring recompilation of the executable from its original source code. Thus, even legacy binaries can be analyzed. Since Pin executes large portions of the target program natively, it can be very fast, however, this constrains the programs analyzed to the host architecture, namely x86. Yet, Pin is surprisingly versatile: new instructions can be emulated by hijacking unused x86 opcodes. Pin can be downloaded from http://www.pintool.org/ and run on any Linux or Windows platform.

As always, this lab is to be completed individually. You are encouraged to discuss lab concepts with fellow classmates.

Setting up:

We will use Git for lab submission. If you are not familiar with Git source control, `git help` is a good command to remember. In general, we will provide skeleton code that you will edit and check in before the submission deadline. As specified in the syllabus, no late hand-ins will be accepted, so submit early and often.

Although you can develop Pintools on your laptop or any Athena computer, we will have thirteen dedicated machines we have configured for class use. These machines are:

```
vlsifarm-01.mit.edu
vlsifarm-02.mit.edu
          …
          …
vlsifarm-08.mit.edu
```

and,

```
eecs-ath-45.mit.edu
eecs-ath-46.mit.edu
          …
eecs-ath-49.mit.edu
```

These class machines use Athena passwords, but only class members may log into them. If you have trouble using a class machine, report the issue to the TAs, and try logging into another machine. Additionally, the lab starter code and your individual git repositories are stored on Athena AFS. If you are unable to access the lab files, please contact your TAs to obtain the AFS permissions.

On every login, you must set up your environment for the labs. On the lab machines, you can do this by sourcing the file `/mit/6.823/Spring20/setup.sh`. For example:

```
% ssh <athena username>@vlsifarm-01.mit.edu
% add 6.823
% source /mit/6.823/Spring20/setup.sh
```

To obtain the materials for lab 0, use the following commands:

```
% mkdir ~/6.823 && cd ~/6.823
% git clone $GITROOT
% cd $USER
% cp -r $LAB0FILES ./
% git add lab0handout
% git commit -m "Lab 0 Initial Check-in"
% git push origin master
```

The `git clone $GITROOT` will clone your repository with your user name. The `cp -r $LAB0FILES ./` will create a copy of the starter code in a directory named lab0handout. In the lab0handout directory that you just created, you should find two sub-directories part1 and part2.

Part 1: Instruction Count

In the first part of this lab, you will develop a simple Pintool to count the number of instructions executed by an application binary. In sub-directory part1, you should find a makefile, some sample source code, and a test script. Type the following at the command prompt:

```
% cd lab0handout
% cd part1
% make
```

You will notice that make fails to compile due to a parse error. While your TAs are experts in Python, his C is not so good. Fix his silly parse error run `make` to compile the Pintool, which will be named "inscount0.so". Pin can be invoked from the command line in the following manner:

```
% pin -t toolname -- command_to_run
```

Experiment with running various commands with the "inscount0.so" Pintool. This will generate a file named "results.out" where you can see the number of instructions that executed.

We have provided a test perl script `part1test.pl`. The perl script will invoke Pin using the inscount0 Pintool on multiple SPEC binaries. Run the perl script by typing:

```
% ./part1test.pl
```

Executing the script will produce a results directory which contains a set of files each of which is named in the following manner:

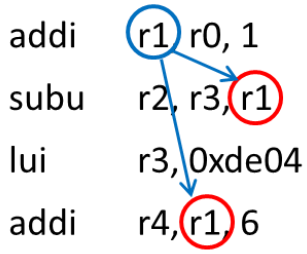
```
ExecName_Arg1_Arg2_..._.out
```

As you might imagine, the naming scheme is based on the executable and arguments that are instrumented and analyzed by Pintool. Since the `inscount0` Pintool counts the number of instructions in the instrumented program, these output files will contain the number of instructions executed. You should examine `inscount0.cpp` to see the workings of this Pintool, which are discussed in the first recitation.

Part 2: Instruction Dependency Distance

For the second part, you will develop a Pintool that generates a histogram of the distances between instruction dependencies in a set of benchmarks. We define instruction dependency distance to be the number of instructions between when a register is written

by instruction and when the register is read by a subsequent instruction. In the following toy example written in MIPS-like assembly (with the modified register in the leftmost position), r1 has dependency distances of one and three.



Dependency distance distributions provide useful insights to computer architects that inform design choices. For example, it allows evaluating the effects of stalling vs data-forwarding for handling pipeline hazards, which we will cover in lecture shortly.

To get started, go to the part2 sub-directory. You should find a makefile, sample source code and a test script. The code that we have provided does most of the heavy-lifting. It instruments every instruction, and captures the registers read and written. You will have to edit the updateDependencyDistanceInfo() function suitably. The code also provides a dependency histogram array, dependencySpacing and a Fini function which dumps the dependency histogram data into a file. The dependency histogram should be updated each time a dependency is detected. The index into the histogram array represents the dependency distance minus one, which is equivalent to the number of other instructions "skipped over" by the dependency. For r1 above, dependencySpacing[0] and dependencySpacing[2] would be updated, although these would not be the only updates to the histogram in this code. For initialization purposes, assume that all machine registers have been modified at time 0. The Fini function will be called at the end of program execution and will produce an output file for easy importation into a spreadsheet program. **Do not modify the Fini function, or you will not pass our test benches.**

To build your Pintool, do:

```
% make
```

Make will build the regDeps Pintool. The regDeps Pintool can be invoked from the command line in the following manner:

```
% pin -t regDeps [-o outputFile -s maxSpace] --
<target_executable>
```

We have provided a test perl script part2test.pl. The perl script will invoke Pin using the regDeps Pintool on multiple SPEC binaries. To invoke the perl script, type:

```
% ./part2test.pl
```

Although we will release a script that will test the lab Pintools, these scripts will not verify your Pintools, and we will not release the expected results of the test cases. In general, we reserve the right to run test cases not included in the released test script. You are encouraged to compare your lab results with your classmates.

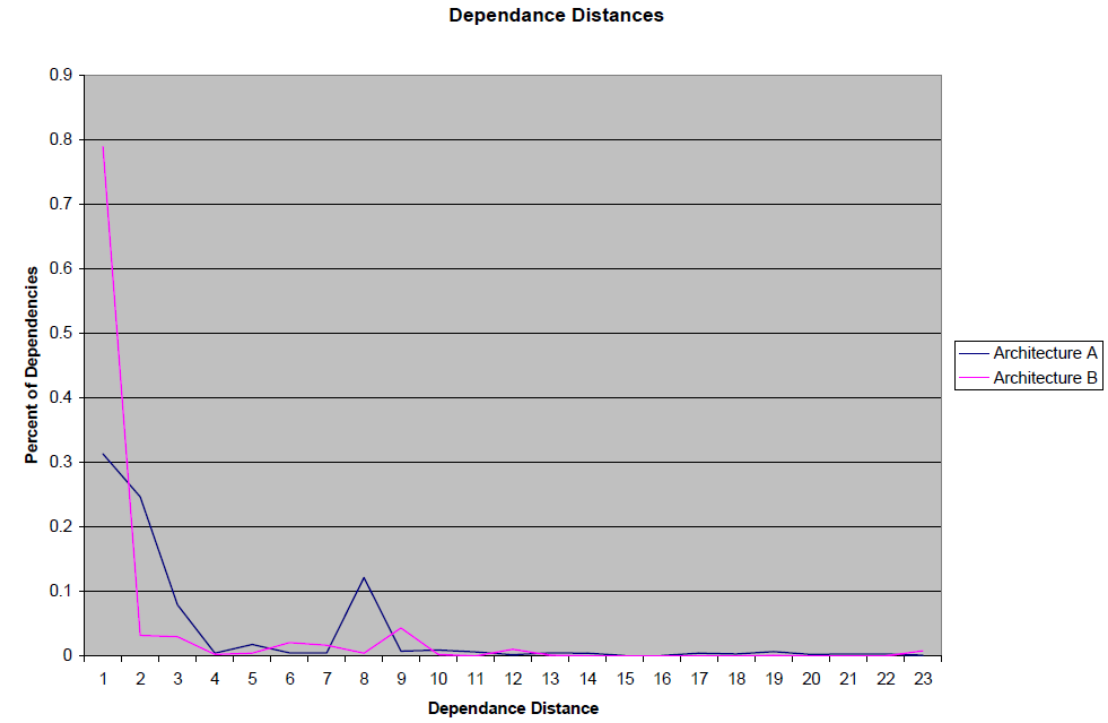
Lab Questions:

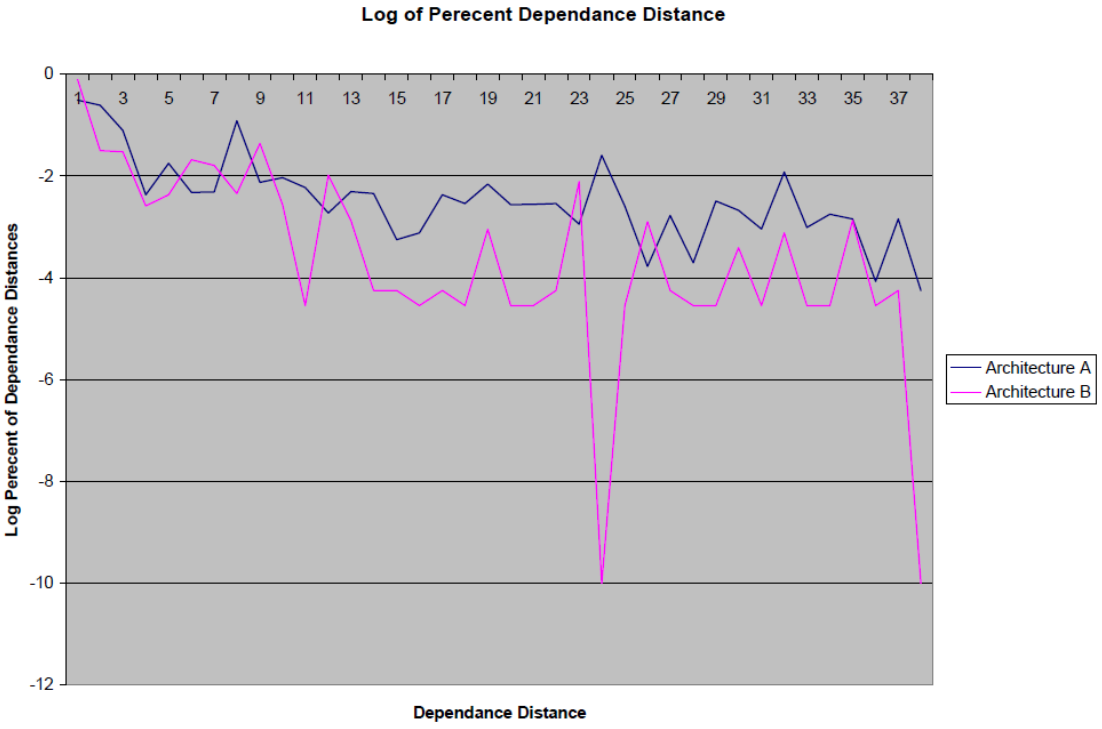
Your response to the lab questions should be typed and save as lab0questions.pdf (or lab0questions.docx) in the lab0handout directory. The course material necessary to answer some questions in lab assignments may be covered after the lab assignment is first distributed, so if some of the material looks foreign, it probably just hasn't been covered yet. In general, lab questions may require coding, and as such should not be put off until the last minute.

1. Plot the dependency distance histogram (Number of dependences vs dependency distance) for the applications.

2. You should notice that the dependency histograms have fairly long tails. Which registers account for most of the weight in the tail? How can this behavior be explained? Suggest some architectural changes that we might make to take advantage of the difference in lifetime lengths of the architectural registers.

3. The following register dependency histograms were obtained by running a benchmark on two machines. The same compiler version (GCC 3.4) was used to compile the benchmark on both machines, but the machines have different ISAs. Based on the instruction dependency graphs below, can you guess which machine has more general purpose registers? Explain your choice.


Dependance Distances
Percent of Dependencies
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Dependance Distance
Architecture A
Architecture B


Log of Perecent Dependance Distance
Log Perecent of Dependance Distances
0
-2
-4
-6
-8
-10
-12
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37
Dependance Distance
Architecture A
Architecture B

Submission:

When you have completed the lab to your satisfaction, use `git add` to stage all the changes you have made to the provided code as well as your answers to the questions, then push your changes (not results) to the git repository by running

```
% git commit -m "Lab 0 Final Check-in"
% git push origin master
```

The deadline for submission is 23:59:59 EST 14 February 2020.

Reporting Issues:

If you notice any 'interesting' or 'unexpected' behavior it could be a problem in the code or infrastructure. Report these bugs immediately to the TAs, on Piazza or in an email with header *6.823 Bug Report*. This will help to ensure prompt fixing of any issues that may arise.

Guides for the perplexed:

http://www.pintool.org/ - Pin home page
https://help.github.com/articles/git-and-github-learning-resources - Git learning resources
https://git-scm.com/book/en/v2 – ProGit ebook