Lab 0: Bootstrapping

The goal of this lab is to set up and learn about the tools and environments needed for the rest of the course.

The tools you will be learning about in this lab are Pin, Destiny, and the SPEC2017 benchmarks. You will be making extensive use of each of these throughout the semester.

Setting up and checking your environment

Log in to the ECE Cluster (ece<XXX>.local.ece.cmu.edu) Access the /afs/ece.cmu.edu/class/ece344/ folder, ensure that you have the following access permissions (e.g. using ls -l):

- Read access to the opt directory and the README-18344.txt file.
- Write access to the opt/spec2017/config directory

Working through the 18-344 on-boarding readme

Follow the steps outlined in the /afs/ece.cmu.edu/class/ece344/opt/README-18344.txt file. This should enable you to run Pin, Destiny and SPEC2017 without fully specifying their paths.

Get acquainted with Pin

Pin is a "binary instrumentation" framework that you will be using to build computer architecture simulators. This lab will help establish you as a user of pin, without going into a lot of detail on how pin works. Later in the semester, we will learn more about the principles behind binary instrumentation tools such as pin (and Valgrind, which you may have used in 213).

The term binary instrumentation refers to a "binary" -- a compiled executable program -- and a special type of analysis called "instrumentation".

What is instrumentation? Instrumentation is an analysis that selectively inserts, removes, or modifies a program's code. This idea is pretty wild, so you should pause to think about what this process entails.

Pin modifies your binary!

These modifications have access to information in and about your executing program.

As an example that we will make use of throughout the semester, and that you will make use of in this lab, it is possible to ask Pin to insert an arbitrary function call to a function that you wrote every time your program accesses memory. Furthermore, Pin can pass to this function through its arguments the instruction address, whether the access is load, store, or both, and the target memory location of the memory access.

As another example that you will not use in this lab (but you will in lab 1), you can ask Pin to insert an arbitrary function call at each branch instruction, with access to its instruction address and the branch's outcome (taken or not taken).

In 18-344, we will provide as starter code all that you need to ask Pin to instrument binaries. You mostly will not have to change anything in the Pin-specific part of the code to implement your lab assignments. Instead, you will be mainly responsible for implementing functions that Pin inserts calls to.

The collection of the Pin-specific code that selectively modifies a binary, and the instrumented analysis code that you write (i.e., the call inserted at each memory access) is called a "pintool".

In this lab, you will write the "hello world" of pintools: a tool that counts the number of loads and the number of stores that happen during a program's execution. We provide starter code that asks Pin to call a function called Load() on each load operation and a function called Store() on each store operation. Your tasks is to add code to implement these functions and keep a running total of the number of loads and the number of stores.

Later in the course, we will use Pin's ability to instrument programs to generate a trace of memory access instructions, branches, and other operations. In the instrumented code, you will implement a computer architecture simulator (e.g., in Lab 1, a branch predictor simulator) that consumes this trace of operations. If that sounds mysterious, that's OK for now.

Developing your first pintool:

As discussed above, we provide you with starter code -- mem-count.cpp. This file parses all instructions, and calls the functions Load and Store for memory reads and writes respectively. You are responsible for writing the body of this function, such that you count and report the number of loads and stores for a binary.

The Fini function executes *after* the execution of the binary, so that is where you will put code for outputting your results. Specifically, you should write the number of loads and stores measured for the binary into an output file (whose name can be specified from the command line with the `-o` flag).

Once you have inserted the logic for counting loads and stores, as well as the Fini function, you can test the operation of the pintool by first compiling the pintool, and then running it.

To compile, simply type `make` in the bootstrapping directory.

If your program compiles successfully, without errors, you should have a folder named `obj-intel64`, with a `*.so` pintool file inside it. You are now ready to run the pintool!

We have provided detailed instructions on running pin in the on-boarding readme: /afs/ece.cmu.edu/class/ece344/opt/README-18344.txt. To quickly test the pintool, type:

```
pin -t <path-to-bootstrapping>/obj-intel64/<pintool-name>.so -o <output-file-path-and-name> --
```

Upon successful execution, this should generate the load/store count for the curl command, stored in the file specified after -o.

Performing a benchmark sweep with your pintool:

Computer architecture artifacts are typically evaluated by running the artifacts on multiple standardized workloads, so that their results can be directly compared with other existing systems also evaluated on the

same workloads. Throughout this class, we will evaluate all our labs on a workload benchmark suite that is commonly used in computer architecture research -- SPEC2017.

To run your pintool on the SPEC2017 benchmarks, follow the instructions provided in the on-boarding readme (/afs/ece.cmu.edu/class/ece344/opt/README-18344.txt) closely. Once you have set up the config file in the SPEC2017 folder to call the $HOME/run.sh or $HOME/run.py scripts, you will add your pintool command to your run script as explained in the 18-344 readme.

Specifically, you can manually run (or your run script should include) the following command:

```
pin -t <path-to-bootstrapping>/obj-intel64/<pintool-name>.so -o <output-file-path-and-name> --
```

where $command is the argument passed to your script by SPEC2017.

SPEC2017 provides a wide range of integer and floating-point benchmarks. For this class, we will focus on one benchmark subset called `intspeed`. This subset consists of different integer benchmarks that run for a single iteration, and report the execution speed. While this lab does not require you to report the execution speed of your pin-instrumented binary, we will be using these statistics in Lab 2.

To finally run your pintool on the `intspeed` benchmark suite, you will run the following command:

```
runcpu -c <path-to-spec2017-config>/18344-f21-<andrewid>.cfg --action=onlyrun --noreportable --
```

What this command does is, for each individual benchmark in the `intspeed` suite, it executes the script command you specify in the `submit` field of the config file, with a benchmark-specific $command. Your script file receives this $command as a command-line argument. Your script should read this argument, and execute the pin command, with this argument after `--`, as specified above. Note that you need to provide a benchmark-specific output file name to pin, otherwise the results of each benchmark will get overwritten by the next one.

Finally, you must tabulate the results, reporting the total loads and stores for each SPEC2017 benchmark.

Simulating SRAM caches with Destiny:

In class, you will model memories and caches. Understanding the performance and power/energy-efficiency of SRAM caches at different configurations will be key to designing systems that can operate with high performance and/or efficiency.

We will use the Destiny memory/cache modeling tool, generating the access latencies and energies for caches of different sizes, with different wordwidths and different associativity. While Destiny provides support for different memory technologies, such as SRAM, eDRAM, PCM, MRAM and such, we will focus on SRAM-based caches.

To get you started, we have provided a template configuration file -- SRAM_template.cfg. You can specify several parameters for the cache through this configuration file. For this lab, you will edit the Capacity, WordWidth and Associativity parameters.

To run the Destiny tool, use the following command:

```
destiny SRAM_template.cfg
```

This should report the access latencies and energies for the specified cache configuration, which you could optionally pipe to a text file for easy parsing.

Your task will be to generate the following values for different cache configurations: Total Area, Access Latencies and Energies (hit,miss,write) and Total Leakage Power. You will generate different configurations by specifying the following values in the configuration file (one at a time): Capacity: {16kB, 32kB, 64kB}, WordWidth: 64 bytes, and Associativity: {1, 2, 4} -- total 9 configurations.

Tabulate the generated results for all configurations into a separate file.

Evaluation & Submission Artifacts:

*You will submit a document (pdf) with tabulated results obtained from running your pintool on the SPEC2017 benchmarks, and the different cache configurations modeled using Destiny.