Computer Science 146/246
Homework #1

Due 11:59 P.M. Friday, February 10, 2023

Welcome to CS146/246. In this class, we will mainly be using the Pin framework and performance monitoring tools for homeworks and projects. The goal of this homework is to help you familiarize yourself with these tools as well as the class infrastructure such as submitting to canvas. It will also include some review questions on CPI/ISA and pipelining.

If you need any help, feel free to contact the teaching staff over slack.

1 Account Setup

This course will be hosted on the FAS Compute Cluster using the **FAS-OnDemand Dashboard**. After logging into Canvas and selecting the COMPSCI 146 course, a tab on the menu options titled *FAS OnDemand* should exist. This gives students access to virtual desktops, tools such as Jupyter notebooks and RStudio, and many other resources.

Clicking on the *FAS OnDemand* tab will bring you to your **FAS Dashboard**, where all the applications for this course are installed. We will be using the *Remote Desktop* app. After clicking on the *Remote Desktop* icon, it will ask you questions about how much memory you need, the number of compute cores, MPI tasks, and expected run-time. The default values will be fine for the purposes of this assignment.

Once your virtual desktop is ready, you will have access to several applications including a terminal, VSCode and the File System explorer. If you are not familiar with linux or need a refersher on basic linux commands, check out this cheat sheet https://commons.wikimedia.org/wiki/File:Unix_command_cheatsheet.pdf. In your home directory you will see two folders; `shared_data` and `scratch_folder`. Read-only course material, such as homework assignments or installed tools, can be copied or run from the `shared_data` folder. The `scratch_folder` is useful for generating / moving large temporary files.

It is recommended you create a third folder in your home directory called `workspace`, where you will keep all your personal files for homework assignments and the course project. Any file generated in your home directory (not in the `scratch_folder`) is persistent data, meaning it will last in-between interactive sessions. Each user should have $\sim$ 20GB or storage for this course.

Another method for accessing your file system is through the **FAS Dashboard** by clicking on "Tools/Home Directory." This interface is also great for uploading / downloading files between your personal computer and the FAS On-Demand file system. When you need to submit an assignment, download the folder to your laptop first, then include all other files and compress as a `.zip` file (I do not want a .tar file in the submission).

More info on how to use FAS OnDemand: https://fasrc.github.io/fas-ondemand-user-guide/

This course will require several modules to run tools such as `Pin`. Open the `.bashrc` file in your home directory (`$HOME/.bashrc`) and add the following to the end of the file. After editing, either close all terminals or source the updated `.bashrc` file.

```
# load modules for make and gcc
module load intel-mkl cmake intel/21.2.0-fasrc01 gcc/9.2.0-fasrc01

# add color to terminal
export TERM=xterm-256color

# export path to Pin tool
export PIN_ROOT=$HOME/shared_data/pin
```

2 Pin Introduction

Pin is a framework for the instrumentation of programs. It supports binary instrumentation of executables on all Intel platforms. Pin allows you to inject arbitrary code (written in C or C++) at arbitrary places in the executable. Pin adds code dynamically while the executable is running, not statically.

Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies, allowing context information such as register contents to be easily passed to the injected code as parameters. This allows programmers to focus on program introspection instead of worrying about how to inject code into the program safely so that it does not break.

Please review more information available about Pin here: https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html. You will want to familiarize yourself with the webpage, as it has a lot of information on Pin, including examples and API documentation. The Examples section in the User's Manual is also a good starting point. If you want to learn a little more about how Pin works internally, there is a widely-cited paper that you can read at your leisure [1].

2.1 Example: Instruction Count

We include an example Pintool in the HW1. Copy HW1 from the `shared_data` to your own directory and start working from there.

(1) Check to make sure the `$PIN_ROOT` environment variable is set (`echo $PIN_ROOT`). If it is not, review the instructions described in account setup.

(2) Build the Pintool `hw1.cpp` by navigating into your copy of HW1 and running `make`. This will generate a Pintool executable within the subdirectory obj-intel64 called "hw1.so".

(3) Run the compiled Pintool using Pin.

```
$PIN_ROOT/pin -t obj-intel64/hw1.so -- /bin/ls
```

The above command will count the number of instructions that /bin/ls executes and write the results into a new file called `tool.out`. Please review "hw1.cpp", as it explains step-by-step how the Pintool is able to count the number of instructions that `ls` executes.

2.2 Create Your First Pintool [40 pts]

Edit the "hw1.cpp" file to do instruction mix profiling. Currently, the Pintool does not count instructions based on instruction types. Instead, we want to use Pin API calls to determine whether an instruction is a Load, Store, or Branch and present an instruction mix breakdown at the end of a run. The following API calls will allow you to determine what the instruction type of an instruction; `INS_IsMemoryRead(ins)` for load instructions, `INS_IsMemoryWrite(ins)` for store instructions, `INS_IsBranch(ins)` for branch instructions.

You must extend the instrumentation function (`Instruction`) to identify the instruction's type and increment a corresponding counter. You will need to understand the difference between instrumentation and analysis functions in Pin for this. Currently, the tool has one counter called `icount` to count the total number of all instructions executed. You will instead need to create more counters based on the instruction types, and increment them in appropriate analysis functions that count loads, stores or branches.

In order to find out more information about an INS object you have to use the Pin API documentation: https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INS__REF.html.

3 CPU Performance Counters

3.1 perf

Pin does not know when micro-architectural events happen since they are subject to concrete design choices of the hardware. `Perf` is a tool included in the Linux kernel used for instrumenting CPU performance counters, trace-points, and probes which can read the hardware counters while running an application. There is no binary instrumentation occurring when running an application with `perf`.

The advantage of `perf` is that it is easy to use directly on an application and measure micro-architectural events such as the cache misses or miss-predicted branches. The disadvantage with `perf` is that high privileges are needed on the machine where it is run. For example, `perf` can only be run as `sudo` and thus is unable to directly run on most shared server systems due to the high-privileges necessary, leading to security risks.

3.2 likwid

Instead, we will be using `likwid` (Like I Know What I'm Doing). This is an open source tool which provides common users access to the hardware performance registers using access daemons. This tool was written with security in mind, restricting accesses to hardware performance related registers, so users cannot read or write system related registers.

`likwid` has many awesome tools which are useful for understanding and controlling the system you are using such as `likwid-perfctr`, `likwid-topology`, `likwid-pin` and `likwid-powermeter`. Please play around with them if you have time. `Likwid` also provides a python API which can be installed using pip for simple benchmarking. Further documentation on `likwid` can be found here: https://github.com/RRZE-HPC/likwid/wiki.

We will be using `likwid-perfctr`. Each core has a definite number of registers to read performance counters. To simplify the user experience, `likwid` has predefined groups of performance counters. For example, there are groups for branch predictions, single and double precision FLOPS, L2 and L3 cache, TLB, Memory Accesses, etc... You can also create your own custom groups. Examples of groups (depending on your CPU) can be found here: https://github.com/RRZE-HPC/likwid/tree/master/groups. You can find the CPU "Code Name" by running `lscpu` in the terminal and copying the "Model name" into a search engine.

3.3 Running likwid [10 pts]

The `likwid` tool suite **cannot** be accessed from the VDI. Instead we must go through a separate shell access which avoids using the virtual machine (because the base image for the VDI does not have `likwid` installed). To access the shell, go to the **FAS-OnDemand Dashboard** and open up the Tool drop down menu. Select the last menu item - *FAS-OnDemand Shell Access*. This will bring you to a login node. Next, we must move to a compute node by running the following command:

```
salloc -p academic -c 32
```

This requests a compute node from the academic cluster with 32 cores. As of Spring 2023, the academic compute cluster only appears to have 32 core Broadwell machines; thus, requesting 32 cores should give you a full node to yourself (if newer nodes are added to the compute cluster then try requesting a system with more cores). Requesting a full node is important for accurate performance characterization.

In this shell you can access all the `likwid` tools as well as your **FAS-OnDemand** file system. To bring up help for the tool you will be using run: `likwid-perfctr -h`. To list all available performance counter groups for the system you are using run `likwid-perfctr -a`. The line below runs `likwid-perfctr` using physical CPU 10 and measures performance counters for the L2 group while `ls` is running. Verify that the number of "Instructions Retired" is similar to the number of instructions recorded by pin for `ls`.

```
likwid-perfctr -C 10 -g L2 -- ls
```

Submit a screen-shoot of the `likwid-perfctr` output along with your submission.

4 Review of Performance / ISA

Question 1: CPI and Clock Rate [10 pts]

Consider three different processors P1, P2, and P3 executing the same instruction set. P1 has a 3.0 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.

a. Which processor has the highest performance expressed in instructions per second? [5 pts]

b. If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions. [5 pts]

Question 2: Amdahl's Law [15 pts]

When making changes to optimize part of a processor, it is often the case that speeding up one type of instruction comes at the cost of slowing down something else. For example, if we put in a complicated fast floating-point unit, that takes space, and something might have to be moved farther away from the middle to accommodate it, adding an extra cycle in delay to reach that unit. the basic Amdahl's Law equation does not take into account this trade-off.

a. If the new fast floating-point unit speeds up floating-point operations by, on average, 2x, and floating-point operations take 20% of the original program's execution time, what is the overall speedup (ignoring the penalty to any other instructions)? [5 pts]

b. Now assume that speeding up the floating-point unit slowed down data cache accesses, resulting in a 1.5x slowdown (or 2/3 speedup). Data cache accesses consume 10% of the execution time. What is the overall speedup now? [5 pts]

c. After implementing the new FP operations, what percentage of execution time is spent on FP operations? What percentage is spent on data cache accesses? [5 pts]

5 Review of Basic Pipeline (5-stage)

Question 3: Pipelining [25 pts]

Consider the following code fragment:

```
LOOP:   LD     R1, A(R0)    ;load R1 from address A+R0
        LD     R2, B(R0)    ;load R2 from address B+R0
        DADDI  R3, R1, R2   ;R3 = R1 + R2
        SD     C(R0), R3    ;store R3 at address C+R0
        DADDI  R0, R0, 4    ;R0 = R0 + 4
        DSUB   R4, R5, R0   ;R4 = R5 - R0
        BNEQZ  R4, LOOP     ;branch to LOOP if R4 != 0
```

Assume that the initial value of R5 is 400+R0. Throughout this exercise use the classic RISC five-stage integer pipeline from the lecture and assume all memory accesses take 1 clock cycle. Assume the last cycle of the loop is when the branch is resolved. State all of your assumptions.

a. Identify all WAR, WAW, and RAW dependencies in the instruction stream. [5 pts]

b. Show the timing of this instruction sequence for the RISC pipeline *without* any forwarding or bypassing hardware but assuming a register read and write in the same clock cycle "forwards" through the register file (WB and ID share cycle). Use a pipeline timing chart to demonstrate the sequence. Assume that the branch is handled by flushing the pipeline and the branch is resolved after completing the MEM stage. How many cycles does this loop take to execute? [10 pts]

c. Show the timing of this instruction sequence for the RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart to demonstrate the sequence. Show all forwarding with an arrow. Assume that the branch is handled by predicting it as NOT taken. Also assume that the hardware supports fast branch resolution during the ID stage and full forwarding. How many cycles does this loop take to execute? [10 pts]

6 Submission Instruction

Please upload the following information in a .zip file to canvas under the HW1 Assignment. Title the submission using the following format `cs148_248_hw1_[first]_[last].zip`

- ☐ Modified `hw1.cpp` code for determining instruction type. [30 pts]
- ☐ Pin profiling results, which include the instruction breakdown for `ls`. [10 pts]
- ☐ A screen-shot of `likwid-perf` results for `ls` showing number of retired instructions and other performance counters from L2 group. [10 pts]
- ☐ Please include a copy of your solutions to questions 1-3 as a single PDF. [50 pts]

References

[1] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation (PLDI)*, 2005.

Updated February 2, 2023, Matthew Adiletta