Lab Assignment 2: Dynamic Branch Prediction

1 Objective

The objective of this assignment is to investigate the performance of different dynamic branch prediction schemes. All work on this assignment is to be done in groups of two.

This assignment involves a bonus competition aspect; you are encouraged to start early!

2 Problem Statement

Use the Championship Branch Prediction (CBP-4 [1]) simulator to determine the number of mispredictions per thousand instructions (MPKI) of the following predictors:

- a 2-bit saturating counter predictor (see Figure 3.18 from your textbook [2])
- a two-level predictor that uses 2-bit saturating counters (see PAp predictor in Figure 3.20 [2])
- an open-ended predictor (with additional bonus marks for the top-performers)

Simulate all three predictors on the following benchmarks: `astar`, `bwaves`, `bzip2`, `gcc`, `gromacs`, `hmmer`, `mcf`, and `soplex`. Instructions on how to build the simulator and run these benchmarks are given in Section 4.

Create one short microbenchmark of 10-20 lines of C code to verify that the correct statistics are being collected for the two-level predictor. Though you will only be submitting one for the two-level predictor, you are still advised to write microbenchmarks to verify the correctness of the other predictors.

In addition, provide area, delay, and power measurements for your two-level and open-ended branch predictors using CACTI [3]. Section 3 provides a detailed walk-through example.

2.1 Dynamic Branch Predictors

The dynamic predictors (excluding the open-ended predictor) were discussed in class and are also described in Section 3.4.3 of your textbook.

For the 2-bit saturating counter predictor, assume that your prediction tables have 8192 bits.

For the two-level predictor, use the PAp predictor from Figure 3.20 [2]. The first-level is a 512-entries table of local history registers (BHT), with 6 history bits per entry. The second-level has eight pattern history tables (PHTs), with a 2-bit saturating counter per entry. The BHT and PHT table indices should be computed from the branch PC as follows:

Use `weak not-taken` as the initial state of your saturating counters.

2.2 Open-Ended Branch Predictor

The open-ended predictor design can be your own creation or inspired by previous research. You can use up to **128 Kbits** of storage for your open-ended predictor. Note: 128Kbits of storage refers to the hardware storage that would be required to build your predictor in a processor, not the

amount of memory your software simulator requires to create the open-ended predictor. Searching for "branch prediction" on `scholar.google.com` will provide you with quite a few references.

The open-ended predictor must have less than 7.5 average mispredictions per thousand instructions (MPKI) across the eight provided benchmarks, in order to receive marks on this portion of the assignment. Marks will be based on performance and implementation creativity. Simply making a larger version of one of the prescribed predictors will not merit any marks for creativity. In addition, your open-ended predictor implementation will compete against those of your classmates. Up to 2 bonus points will be awarded to the top ∼10 open-ended predictors.

3 Area, Delay and Power of different Branch Predictors

Branch prediction is critical to processor performance. A common rule of thumb is that one out of five instructions is a branch. A successful branch prediction scheme should (1) be accurate most of the time (i.e., have few mispredictions), (2) provide fast predictions (i.e., needing ten clock cycles to predict a branch is not acceptable), and (3) be within a reasonable hardware budget.

Computer architects always work within a given set of constraints. All design decisions have their own trade-offs. For example, it might be that requiring 1MB of storage for a branch predictor scheme yields a miniscule misprediction rate. However, it is highly unlikely anyone would dedicate that much chip real estate.

Even if storage were not the issue, larger structures are also slower to access. "Fitting" such a longer access time within a pipeline stage, could force you to lower your processor's frequency, thus making your entire system slower.

Finally, large structures, and especially fully-associative structures, are known to be very power-hungry. One of the challenges of moving towards smaller process sizes (e.g., Intel's Haswell uses 22nm technology) is that the static (i.e., leakage) power becomes considerable.

To better enhance your understanding of these trade-offs, we will use Cacti 6.5 to estimate the area, power, and delay of your branch prediction structures. This is especially critical for your open-ended branch predictor, as it will allow you to grasp how unrealistic some designs can be.

3.1 Using CACTI 6.5

CACTI is an integrated tool which models "cache access time, cycle time, area, leakage and dynamic power" [3]. Although CACTI was developed for analyzing caches, it can be used to approximate any storage structures you might have within your processor. You need two files to run CACTI: the CACTI executable and a configuration file. You can obtain these as follows:

```
cd ~/ece552
mkdir CACTI
cd CACTI

cp /cad2/ece552f/CACTI/cacti .
cp /cad2/ece552f/CACTI/cache.cfg .
cp /cad2/ece552f/CACTI/pureRAM.cfg .
```

We have provided you with two configuration files. The first file, `cache.cfg`, models a cache-like data structure, with a tag and a data array. The second file, `pureRAM.cfg`, models an untagged memory structure. Depending on your configuration you might prefer to use one over the other.

You can run CACTI by passing the configuration file as a parameter.

```
cacti -infile cache.cfg
```

This command prints out the area, delay and power info for a predetermined cache configuration. Feel free to redirect the output to a file for you convenience. Each configuration file provides a variety of options. The ones you are allowed to change are annotated with a **#ECE552: Modify if needed** comment.

The following two options are useful for debugging purposes:

- "Print input parameters": prints all your configuration settings when set to true.
- "Print level": prints more detailed information when set to "DETAILED"; the default value is "CONCISE".

3.1.1 Pure RAM Configuration File

The pure RAM configuration file models an untagged structure. You can control the following parameters:

- size (in bytes): total size of your structure
- block size (in bytes): this is the size of an entry. Caution! Unfortunately, the tool requires the block size to be in bytes. If your entry size is not divisible by eight, then round it up to the next byte. Make sure you use this *rounded* block size when you specify the total size of your structure (previous parameter).
  As a reminder: *cache size = #sets * associativity * block size*. The number of sets should match the number of rows in your predictor.
- bus width (in bits): this should be at least as big as the block size.
- associativity: note that a fully-associative, untagged structure is not meaningful here.
- Number of read, write or read/write ports: A single read/write port should be sufficient for your purpose, but feel free to modify this.

The generated statistics of interest are listed below:

```
Access time (ns): 0.279886
Cycle time (ns):  0.277706
Total dynamic read energy per access (nJ): 0.0022076
Total leakage power of a bank (mW): 2.87418
Cache height x width (mm): 0.133416 x 0.0953244
```

You can compute the total area of the structure from the provided cache height and width. Access time (in ns) reflects the access time for your structure, whereas cycle time is the minimum time between the start times of two consecutive accesses. Usually cycle time is larger than access time, by a small percentage, but it can also be slightly smaller in pipelined circuits. For this assignment, you can focus on access time values. Finally, you can find the dynamic read energy per access and the leakage power of your structure.

Absolute numbers are useful, but it can be equally interesting to compare how these values change for different organizations.

3.1.2 Cache Configuration File

The cache configuration file models a tagged structure, similar to a cache. We will use the modeled tag array to estimate area, power, and energy of the used predictor structures.

The benefit of this approach is that we can control the width of the tag array. Therefore, if we have an entry with 4 history bits, we will not need to round this up to a byte. However, tag arrays involve (unnecessary for our purpose) tag comparators, which the pure RAM implementation lacked. Yet another design trade-off to consider.

To use the cache configuration file do the following:

1. Specify the size (in bytes) of the data array. With a block size of one, this will control the number of entries in your structure. We assume an associativity of one. In general, you need to specify the cache size, block size and associativity so that the data array (and thus the tag array that we care about) has the same number of sets (rows) as your predictor.

2. Specify the tag size. Cacti gives us the option to specify this in bits, so it can be arbitrarily small. For now we set it to eight bits, to be comparable with the pure RAM example.

Note that the cache access type is set to sequential, i.e., we first access the tag and then the data array. We have also set the "Print Level" option to Detailed, as we are only interested in the tag array component. The following statistics are of interest:

```
Tag side (with Output driver) (ns): 0.279886

Tag array:  Total dynamic read energy/access (nJ): 0.00203752
       Total leakage read/write power of a bank (mW): 2.87883

Tag array: Area (mm2): 0.0125494
       Height (mm): 0.131624
       Width (mm): 0.095343
```

As you can see, the generated numbers are comparable. Use the configuration file that is the most applicable to your design. Remember to list all configuration parameters you modified in your report.

4 Methodology

The Championship Branch Prediction (CBP-4 [1]) competition provides a simple framework for implementing and comparing different branch predictors. Our simulator and benchmarks are derived from the the CBP-4 framework. Modify the simulator to measure the MPKI for all three dynamic branch prediction schemes. Then use the collected statistics to prepare a brief report as described in Section 6. In addition, use CACTI to provide area, delay, and power comparisons for the two-level and the open-ended branch predictors.

4.1 Compiling the simulator

You can obtain the CBP-4 simulator source code and set it up in your ug account using the following Unix commands:

```
cd ~/ece552

cp /cad2/ece552f/cbp4-assign2.tgz .

tar -zxf cbp4-assign2.tgz
```

This sequence of commands extracts the simulator files into your working directory (`~/ece552/cbp4-assign2`). These instructions assume that you already created the `ece552` directory for the Pre-Assignment. Do not leave the Unix permissions to your code open.

You can then build the simulator using the provided `Makefile` by typing:

```
cd ~/ece552/cbp4-assign2

make
```

4.2 Running the simulator

You can run the CBP-4 benchmarks as follows:

```
cd ~/ece552/cbp4-assign2

predictor /cad2/ece552f/cbp4_benchmarks/astar.cbp4.gz

predictor /cad2/ece552f/cbp4_benchmarks/bwaves.cbp4.gz

predictor /cad2/ece552f/cbp4_benchmarks/bzip2.cbp4.gz

predictor /cad2/ece552f/cbp4_benchmarks/gcc.cbp4.gz

predictor /cad2/ece552f/cbp4_benchmarks/gromacs.cbp4.gz

predictor /cad2/ece552f/cbp4_benchmarks/hmmer.cbp4.gz

predictor /cad2/ece552f/cbp4_benchmarks/mcf.cbp4.gz

predictor /cad2/ece552f/cbp4_benchmarks/soplex.cbp4.gz
```

On each run, the simulator executes all three dynamic branch predictors: `2bitsat`, `2level` and `openend`. It then prints out the number of mispredicted branches (`NUM_MISPREDICTIONS`) and the MPKI (`MISPRED_PER_1K_INST`) for each of the predictors.

4.3 Modifying the simulator

You will only be making changes to `predictor.cc`. Do not modify or add any other files; they will not be taken into account.

In `predictor.cc`, you are provided with the following skeleton functions for you to implement each of the three branch prediction schemes (`<predictor>` refers to `2bitsat`, `2level` or `openend`):

- `InitPredictor_<predictor>`: Called only once, before the benchmark begins. Use this function to set up your data structures.

- `GetPrediction_<predictor>`: Called on each conditional branch instruction, **before** the branch direction and target address are resolved. Use this function to generate your prediction: return `TAKEN` or `NOT_TAKEN`. Based on your return values, the simulator counts the number of mispredicted branches and computes the MPKI for you. Note that by default, this function implements a static always-taken predictor, since it does nothing but return the value `TAKEN`. This function's only parameter is `PC`, which is the branch instruction address (32-bit byte-aligned).
- `UpdatePredictor_<predictor>`: Called on each conditional branch instruction, **after** the branch prediction and target address are resolved. Use this function to update your data structures with the branch outcome. This function takes in four parameters:
  1. `PC`: the branch instruction address.
  2. `resolveDir`: the resolved branch direction: either `TAKEN` or not `NOT_TAKEN`.
  3. `predDir`: your predicted branch direction (result of `GetPrediction_<predictor>`).
  4. `branchTarget`: the branch target address.

Do not modify the function names and parameters; they need to match the function declarations in `predictor.h`.

4.4 Microbenchmarking

This section describes how to run a microbenchmark program on the CBP-4 simulator. Write your microbenchmark in a file `mb.c`. Please refer to the Assignment 1 handout for good practices on writing microbenchmarks. Unlike in previous assignments, use the standard `gcc` compiler instead of the SimpleScalar version:

```
gcc mb.c -o mb
```

To run your microbenchmark on the simulator, you first need to generate a branch trace file. To do this, you are provided with a BranchTrace tool implemented with Pin, a binary instrumentation framework [4]. Run the tool with your microbenchmark via the following commands:

```
cd ~/ece552/cbp4-assign2

/cad2/ece552f/pin/run_branchtrace /<absolute_path>/mb
```

Where `<absolute_path>` is the absolute path to your microbenchmark executable; Pin does not accept relative paths. Note that you can see the absolute path of your current working directory by using the Linux command `pwd`.

The BranchTrace tool generates the trace file `branchtrace.gz` in your working directory. This file stores the dynamic stream of all branch instructions executed by your microbenchmark. You can now run the CBP-4 simulator with the following commands:

```
cd ~/ece552/cbp4-assign2

predictor branchtrace.gz
```

Note that the simulator assumes 32-bit PC values, but your microbenchmark will be compiled on the 64-bit ug machines. This means that the higher-order 32 bits of the PC will be omitted. We do not expect this to be a problem though, since your microbenchmark should be small enough such that the higher-order bits do not affect the indexing into your predictor tables.

5 Prelab

The prelab is worth 1/6 of the overall lab mark. Please complete the following steps before coming to the lab:

- Read all necessary background on branch prediction (textbook, lecture slides).
- Answer the following questions:
  1. Why do we use predictors with 2-bit saturating counters (i.e., bimodal)? Contrast this with the 1-bit approach.
  2. You are given the following code snippet, with two conditional branches clearly marked:

```
int a;
for (int i = 0; i < 100000; i++) { // conditional branch B1
    if ((i % 4) == 0) { // conditional branch B2
       a = 10;
    }
    a = 15;
 }
```

     (a) Write down the taken/not-taken sequence for the second conditional branch B2. List any assumptions you make about how this if-statement translates in assembly.
     (b) Assume you are keeping track of prior branch history for conditional branch B2. What is the smallest number of history bits (h) you need to correctly predict all B2's branch outcomes?
     (c) Assume you have a PAg predictor (Figure 3.20) with an eight-entries BHT and h history bits per entry (h is the previously computed number). Why would it be a bad idea to index the BHT with the 3 most significant PC bits?
- Write most of your code, along with some microbenchmarks, before coming to the lab.
- Finally, be prepared to answer any high-level questions about the simulator, your code, and the lab material (e.g., compare different branch prediction schemes).

6 Lab Deliverables

At the end of this assignment you should submit the following files using the `submitece552f` command:

1. **predictor.cc**: the modified CBP-4 simulator file.
2. **mb.c**: your microbenchmark code for the two-level predictor. Your microbenchmark must include comments and snippets of generated assembly code that explain how it verifies the collected statistics. Reread the good microbenchmark practices from Assignment 1. Failure to include appropriate explanation will result in a grade of 0 for the microbenchmark portion of the assignment.
3. **report.pdf**: a brief two-pages pdf report (single-spaced with 12-point font size). Make sure your report can be viewed on the ug machines through `xpdf` or `acroread`.
   - Your report should include both you and your lab partners names.

- Briefly explain how your microbenchmark collected statistics to validate the correctness of your code for the two-level predictor. Refer to specific branches in your code and their branch history. Feel free to refer to comments within the mb.c file, as needed. Specify which compilation flags you used.
- Provide a table with the number of mispredicted branches and the MPKI for all three predictors and benchmarks. Be wary of the two-page limit, since this table may be large.
- Describe your open-ended branch predictor implementation. This section must include how you calculated the storage requirements (in bytes) for your open-ended predictor.
- Report area, access latency, and leakage power for the two-level and your open-ended branch predictors, using CACTI. Specify the configuration parameters you modified.
- Include a brief statement of work completed by each partner.

4. **open-ended-bpred.cfg**: the CACTI configuration file you used for your open-ended predictor. It should be based either on cache.cfg or pureRAM.cfg. If you use multiple predictor tables, you can submit multiple configuration files with one for each table. For naming, please add the suffix `-1`, `-2`, etc. (e.g. `open-ended-bpred-1.cfg`). Be sure to clarify in your report what each `.cfg` file models.

5. **2level-bpred.cfg**: the CACTI configuration file for your two-level predictor.

The submit command is the following:

```
submitece552f 2 predictor.cc report.pdf mb.c open-ended-bpred.cfg 2level-bpred.cfg
```

You can view the files that you have submitted via the command

```
submitece552f -l 2
```

Only one lab member from each group should submit code. Submissions from both partners will result in added confusion during marking.

Do not leave the Unix permissions open to your code or micro-benchmarks. You are strongly encouraged to submit your code early to determine how your open-ended predictor ranks compared to your classmates.

7 Due Date and Marking Scheme

This assignment is due on **Friday October 22, 2021 at 6:00pm EDT**. It is worth **6%** of the total course mark. The pre-lab will constitute 1/6 of the overall lab mark.

In order to receive any points on the open-ended predictor, your predictor must have less than 7.5 average MPKI across the eight benchmarks provided. Marks will be awarded based on the performance of your predictor and how adequately you describe your predictor in the report. Additional bonus points will be awarded to the top $\sim$10 performing open-ended predictors. During the week prior to the due date, you can submit your code each day; each night the TA will run the open-ended predictor and post a ranking of the results on Quercus. You can continue to update and resubmit your code until the deadline. Please plan ahead and start early!

An automarker will be used to compile and run your implementation and verify the statistics generated. Therefore, your implementation is required to follow some strict rules. To begin with, use the simulator package specifically provided for this assignment as described in Section 4.

Things to watch for:

- run the benchmarks as specified in the handout
- initialize properly all the data structures
- the only file to modify is `predictor.cc`
- do not add additional files, they will not be taken into account; your simulator should compile with the provided `Makefile` by typing `make`
- an automatic comparison of your submissions will be done to check for copied code; changing variable names and formatting will not defeat the checker. Using code found on the internet is also prohibited.
- specify the absolute path to your microbenchmark executable when using the BranchTrace tool.

8 Questions

Please post clarification questions on Piazza. Do not post solution code or microbenchmark code. This constitutes cheating and will not be tolerated.

References

[1] Championship Branch Prediction (CBP-4). `http://www.jilp.org/cbp2014/`

[2] Dubois, Michel, Murali Annavaram, and Per Stenström. Parallel Computer Organization and Design. 1st ed. Cambridge: Cambridge University Press, 2012.

[3] CACTI. HP Labs. `http://www.hpl.hp.com/research/cacti/`

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Proc. Conf. Programming Language Design and Implementation, 2005.