Assignment 2: Branch Prediction

Quantitatively compare the bi-mode and the perceptron branch predictor

Outline

- **Due date:** 31 October 2022, 23:59
- **Interviews:** By Invitation Only
- **Mark weighting:** 20%
- **Submission:** Submit your assignment through GitLab
- **Policies:** For late policies, plagiarism policies, etc., see the policies page

Time left to deadline
13:11:45:41
Days Hrs Mins Secs

Introduction

This assignment aims to enhance your conceptual knowledge of various dynamic branch prediction schemes with practical experience implementing them in an architectural simulator. You will test your intuition about the branch behavior of different benchmarks that represent real-world behavior. You will implement new branch predictors in the Sniper multicore simulator and measure the accuracy of your new predictors against a set of baselines. In the process, you will learn to design experiments and generate meaningful conclusions from the experimental data. This experience will enable you to understand how architects propose and evaluate new ideas for future processors.

Dynamic branch prediction is a microarchitectural optimization to reduce delays in the pipeline due to branch hazards. Branch predictors sit in the fetch stage of a modern high-performance processor. The processor needs to know which instruction in memory to fetch next before decoding the current instruction. The branch predictor tells the processor whether the branch will be taken (and consequently fetch instructions from the target) or not taken (fetch the fall-through code). As the branch instruction is not resolved for several cycles after it has been fetched; the processor relies on the branch outcome. Therefore, high prediction accuracy is critical for delivering good performance.

Evaluating New Computer Architecture Ideas

Fabricating a new processor in real life is a massive investment of time and money. It is paramount to analyze the performance of newly proposed ISA and microarchitecture changes rigorously and quantitatively before the actual tape-out. The approach computer architects take to explore new ideas resembles the established methodology in scientific research.

- First, architects take a hypothesis about the environment based on intuition, insight, or experience. One hypothesis may be that using more history bits in local history two-level predictors may yield high prediction accuracy than increasing the number of entries in the branch history table. Another hypothesis can be that using local history benefits programs in one class but hurts others.
- Next, architects typically test such ideas in a restrictive setting, such as an architectural simulator. Simulation precludes interference that is unavoidable in a live experimental setup. It also empowers the architect to model hardware yet not fabricated. (In Sniper, we implement new ideas in the C++ backend.)
- The next step for the architect is to design meaningful experiments to validate the hypothesis. This step involves picking: (1) realistic workloads, (2) fair baselines for comparison, and (3) meaningful metrics, and (4) configuration parameters. This step is also known as the experimental methodology. Note that limiting the configuration space also requires intuition and experience. Simulations take a long time to finish. A brute force approach to sweeping through the design space inhibits an architect's progress.
- Next, the architect analyzes the experimental data. They must draw meaningful insights and fair analyses in favor of or against the hypothesis. Interpretation of the experimental data is a critical step in the validation of a new architectural idea.
- Finally, if the results are surprising, the architect revisits the hypothesis and proposes a new one.

Your Task

From the view of hardware, implementing dynamic branch prediction requires two key elements: (1) a set of hardware structures to store the predictor's state, and (2) logic that informs the processor whether the branch is likely to be taken or not taken. The logic includes a way to generate a prediction and a way to update the predictor's state. Your task is to implement the data structures and the logic to predict branch outcomes and state updates for three new branch predictors in Sniper's C++ backend.

For the predictors, we ask you to compare their accuracy for a fixed hardware budget of **1024** Bytes. Note that each predictor uses a different context and may thus require distinct hardware structures. We leave it to you to configure the various parameters of your newly implemented branch predictors based on your intuition and experimental analysis. You will compare the accuracy of your new branch predictors against a set of baselines. These baselines are already implemented in Sniper. You will also write a brief report quantitatively analyzing your observations for a set of benchmarks.

The Predictors

The predictors you will need to implement include:

1. `gshare` predictor
2. bi-mode
3. perceptron

Note that the two direction predictors in `bi-mode` are similar to the `gshare` predictor.

Whenever required, use two-bit saturating counters. You can initialize the two-bit saturating counters to any state based on your intuition and analysis.

The Bi-Mode Predictor

The bi-mode predictor is fully described in a MICRO, 1997 publication. In addition, there is a post on the course forum that clarifies its working.

The bi-mode predictor is an interference-reducing predictor consisting of two direction predictors and a choice predictor. Both direction predictors simultaneously predict the branch outcome. A meta-predictor (choice predictor) chooses the final prediction.

The motivation for the bi-mode predictor is to split branches into two groups (strongly taken and strongly not-taken). Bi-mode then uses two PHTs (direction predictors) and index with the same address-history hash. Since most branches are biased in one direction or the other, the choice predictor (indexed using branch address only) isolates them to a separate PHT, mitigating interference. This way, if two branches map to the same entry in the PHT, they are unlikely to harm each other.

For the direction predictors, we ask you to implement a global history two-level predictor (`GAp`) predictor, namely `gshare`. The `gshare` predictor combines global branch history (stored in the branch history register or `BHR`) with the branch address using an exclusive-or (XOR) function.

The choice predictor is indexed by the branch address. For the choice predictor, you should implement the prediction using the two-bit saturating Smith counters.

The Perceptron Predictor

The perceptron predictor is fully described in an HPCA, 2001 publication.

> We will make a tutorial video on the perceptron predictor available next week.

Maintaining larger branch history provides more opportunities for correlating the branch predictions. However, there are two drawbacks with this approach. First, the size of the PHT is exponential in the width of the BHR. Second, many of the history bits may not actually be relevant, and act as training "noise." Two-level predictors with large BHR widths typically take longer to train. The perceptron predictor tackle these drawbacks. Each branch address (not address-history pair) is mapped to a single entry in a perceptron table. Each entry in the table consists of the state of a single perceptron. A perceptron is the simplest form of a neural network. A perceptron can be trained to learn certain boolean functions.

The perceptron predictor can adjust the weights corresponding to each bit of the history, since the algorithm can effectively ignore any history bits that exhibit low correlation. Because of this ability to selectively filter the branches, the perceptron predictor can be trained much faster than conventional PHT-based approaches.

The hardware organization of the perceptron predictor is shown in the lecture slides. The lower-order bits of the branch address are used to index into the table of perceptrons in a per-address fashion. The weights of the selected perceptron and the BHR are forwarded to a block of combinatorial logic that computes y. The prediction is made based on the complement of the sign bit (most-significant bit) of y. The value of y is also forwarded to an additional block of logic and combined with the actual branch outcome to compute the updated values of the weights of the perceptron.

The design space for the perceptron branch predictor is larger than that of the `gshare` and `bi-Mode` predictors. The predictor has several parameters, such as, the number of perceptrons, the number of bits of history to use, the width of the weights, and the learning threshold.

Simulation Environment

You will implement new branch predictors in the Sniper multicore simulator. We have made an instructional video to help you getting started with implementing a new branch predictor in Sniper. We advise you to see the video after reading the assignment.

Resources

We provide two tutorial videos to help you with implementing new branch predictors in Sniper.

- Adding a new branch predictor in Sniper
- How does the perceptorn predictor work?

Benchmarks

We have already provided the sift traces of eleven benchmarks as part of the laboratory environment. In this assignment, we ask you to use only the following eight benchmarks: `gems`, `astar`, `hmmer`, `mcf`, `namd`, `omnetpp`, `povray`, and `leslie`.

Baseline Predictors

We ask you to compare and report the prediction accuracy of your new branch predictors against two Sniper baselines: (1) `one_bit` and (2) `pentium_m`. The best practice is to create two new configuration files, one for the `one_bit` predictor and the other for the `pentium_m` predictor. You can then use `-c one_bit` to select the specific branch predictor for an experiment. We provide the following configuration parameters for the two baseline predictors.

```
[perf_model/branch_predictor]
type = one_bit
mispredict_penalty = 14
size = 1024
```

```
[perf_model/branch_predictor]
type = pentium_m
mispredict_penalty = 14
size = 1024
```

Assuming `one_bit.cfg` and `pentium_m.cfg` have been created in the `sniper/config` directory and populated with the above lines, you should use the following command for running each of your experiments.

> ./run-sniper -c gainestown -c rob -c one_bit -d result_directory –traces=trace-name

When you create a new branch predictor for this assignment in the C++ front-end, you should only change the `type` field in the branch predictor's configuration. You can include any new parameters for your branch predictor if you want. For example, to avoid recompiling the Sniper codebase each time you change the size of the `PHT`, you can add a new configuration parameter, namely `pht_size`, in Sniper. You should keep all other settings unchanged.

Deliverables

In your assignment submission, you need to deliver:

1. The C++ code for each of your newly implemented predictors
2. A report detailing your quantitative analysis of experimental results

We advise you to check the course webpage for instructions to submit the assignment. Also, we provide instructions to submit your C++ code files on the course website. Below, we provide instructions for writing your report. You should submit your report as a single PDF document.

To make it easy to distribute marks and submit the quantitative analysis, we break down the submission of the bi-mode predictor into two parts:

- direction predictor (`gshare`)
- two direction predictors + choice predictor (bi-mode)

In the following sections, we thus treat the bi-mode predictor as two separate predictors.

Quantitative Analysis

In this section, we provide you specific guidelines on writing you report crystallizing your analysis.

Benchmark Characterization

In this section of your report, use the default settings for the `pentium_m` branch predictor as your baseline for gathering the branch-related statistics. You do not need to use the `one_bit` predictor for the analysis in this section.

1. Provide a plot with benchmarks on the horizontal axis and the branch component of the CPI stack (as a percentage of the total time) on the vertical axis. (Note: The metric on the Y-axis is a percentage.) Sort the benchmarks on the horizontal axis according to the branch component. The benchmark with the highest branch component (as a percentage of the total time) appears on the extreme right.
2. Provide a similar plot as above but use the branch misprediction rate (percentage) on the vertical axis. Sort the benchmarks on the horizontal axis according to their misprediction rates.
3. Provide a similar plot but use the branch misses (or mispredictions) per kilo instructions (`mpki`) as the metric on the vertical axis. Use `mpki` to sort the benchmarks on the horizontal axis.
4. (A) Clearly and concisely explain the differences (if any) across the three plots. Provide a list of bullets, one for each observation. (B) Suppose we had a perfect branch predictor with no branch mispredictions instead of the `pentium_m` predictor. Compared to perfect branch prediction, which benchmark suffers the most severe degradation in performance due to the `pentium_m` predictor? Which real-world application does this benchmark implement (use your favorite search engine)? Why is there high irregularity and branch intensity in the specific application/algorithm?
5. Explain which of the metrics: (1) branch component of the CPI stack, (2) branch misprediction rate, (3) branch `mpki`, and (4) some combination of the metrics, is the most appropriate to evaluate new branch predictors for a set of benchmarks. Are there specific scenarios where one metric (or combination) is more valuable than the others? Can you propose a new plot that delivers more insight than any of the three plots in the above questions? (It is not mandatory to have the new plot in your report, but you should explain it clearly. Think about arranging the benchmarks on the horizontal axis and the metric to use for the vertical axis. You can consider a secondary vertical axis as well.)

In the remaining sections of your report, use both the one-bit and the pentium_m (default settings) branch predictors as your baselines for gathering the branch-related statistics. You will compare each new predictor you implement against at least these two baselines. You will use the branch misprediction rate (%) and the branch `mpki` as the evaluation metrics. We leave it to you to configure the different parameters of your newly implemented branch predictor based on intuition or experimental analysis.

`gshare` Predictor

In addition to submitting the C++ code files for the global history two-level predictor (`gshare`), we ask for the following plots and analysis in your report.

1. A plot comparing the branch misprediction rates obtained with the `gshare` predictor for all benchmarks against the two baseline predictors: `one_bit` and `Pentium_m`.
2. A plot comparing the branch `mpki` obtained with the `gshare` predictor for all benchmarks against the two baseline predictors: `one_bit` and `Pentium_m`.
3. Explain the rationale behind your chosen configuration parameters. For example, what is the size of the `PHT` table (and why)? How many branch history and address bits do you use to form the hash index of the PHT table?
4. Briefly explain your observations from the two plots. Provide a list of at least three bullets (one observation per bullet) and as many as you want.

Bi-Mode Predictor

In addition to submitting the C++ code files for the bi-mode predictor, we ask for the following plots and analysis in your report. Note that the `BHR` storage is not counted in the total budget of **1024** Bytes.

1. A plot comparing the branch misprediction rates obtained with the bi-mode predictor for all benchmarks against three predictors: the baselines, `one_bit` and `Pentium_m`, and `gshare`.
2. A plot comparing the `mpki` obtained with the bi-mode predictor for all benchmarks against three predictors: the baselines, `one_bit` and `Pentium_m`, and `gshare`.
3. Briefly explain your observations from the two plots. Provide a list of at least three bullets (one observation per bullet) and as many as you want.

Bi-Mode vs. Perceptron

In addition to submitting the C++ code files for the perceptron predictor, we ask for the following plots and analysis in your report.

1. A plot comparing the branch misprediction rates obtained with the perceptron predictor for all benchmarks against the bi-mode predictor.
2. A plot comparing the branch `mpki` obtained with the perceptron predictor for all benchmarks against the bi-mode predictor.
3. Explain the rationale behind your chosen configuration parameters for the perceptron predictor.
4. Briefly explain your observations from the two plots. Provide a list of at least three bullets (one observation per bullet) and as many as you want.
5. Do you observe similar trends to the ones in the perceptron paper?

Mark Allocation

We will grade your submission on:

- Implementation and C++ code delivery
- Written report

Below we provide the breakdown of marks across the two categories.

C++ Implementation

1. C++ implementation of the direction predictor (`gshare`) (10)
2. C++ implementation of the bi-mode predictor (15)
3. C++ implementation of the perceptron predictor (25)

Quantitative Analysis

1. Benchmark characterization (10)
2. `gshare` predictor (comparison against two baselines) (10)
3. Bi-mode predictor (two baselines and `gshare`) (10)
4. Perceptron predictor (two baselines and `gshare` and `bi-mode`) (20)

Code Marking and Expectations

We will grade the C++ code you submit across three dimensions: (1) correctness, (2) clarity and documentation, and (3) flexibility. By flexibility, we mean that the main parameters of your branch predictors, such as the size of `PHT` and the number of branch history bits, should be easy to modify. We advise you to add meaningful comments to explain the main parts of your code. You do not need to explain each line of your code, but you should point us to the primary data structures, such as the `BHT` and the `PHT`.

Finally, you should indicate and specify with comments the code you borrow from external sources.