CS 4290 - Summer 2021
Project 2 : Branch Prediction

Patrick Lavin
Due: July 6^{th} 2021 @ 11:59 PM
Automatic Extension to July $7th$
Version : 1.2

I Updates

Version 1.1:

- Added a section on stats clarifications in **Section III.vii.1**.

Version 1.2:

- Added instructions for the new verification script in **Section VI**
- Changed the files needed for submission in **Section VIII**

II Rules

- **This is an individual assignment. ABSOLUTELY NO COLLABORATION IS PERMITTED.** All cases of honor code violations will be reported to the Dean of Students. See Appendix A for more details.
- Late assignments will not be accepted.
- Please use office hours and Piazza for getting your questions answered.
- This is a tough assignment that requires a good understanding of concepts before you can start writing code. **Make sure to start early**.
- Read the entire document before starting. Critical pieces of information and hints might be provided along the way.
- Unfortunately, experience has shown that there is a high chance that errors in the project description will be found and corrected after its release. **It is your responsibility to check for updates on Canvas, and download updates if any.**
- Make sure that all your code is written according to **C++11** standards, using only the standard libraries.

III Introduction

Stalls due to control flow hazards and the presence of instructions such as, function calls, direct and in-direct jump, returns, etc. that modify the control flow of a program, are detrimental to a pipelined processor's performance. To mitigate these stalls, various branch prediction techniques are used. In fact, branch prediction is so vital that it continues to be an ongoing area of

research and many new ideas are proposed even today. To better understand branch prediction and various existing techniques, in this project, you will first implement various branch predictors, a two level Global History branch predictor (GShare), a two level Local History predictor (Yeh-Patt), and finally a perceptron branch predictor. You will then run experiments to find a single optimal predictor for a set of workloads under a certain bit-budget constraint.

Recall from lectures that branch prediction has 3 W's - Whether to branch, Where to branch (if the branch is taken) and When to branch. In this project we will concern ourselves with only the first W, i.e. Whether a branch is taken or not.

You have been provided with a framework that reads in branch traces generated from the execution of benchmarks from SPEC 2017 and drives your predictors. The simulator framework support configurable predictor sizes and parameters. Your task will be to fill functions called by the driver that initialize the predictor, perform a branch direction prediction, update the predictor state, and finally update the statistics for each branch in the trace. Section IV provides the specifications of each type of branch predictor and section V provides useful information concerning the implementation.

IV Simulator Specifications

The simulator supports three different branch predictors. You will **pick two** to implement.

i Basic Branch Predictor Architecture

Your simulator should model a Pattern table (Smith Counters) or perceptron table, and a global history register or history table depending on the predictor being simulated. Figures 1, 2, and 3 show the architecture diagrams of the Gshare, Yeh-Patt and Perceptron based branch predictors respectively. The driver for the simulation chooses the predictor based on the command line inputs, and calls the appropriate functions for prediction and update. More details are provided in section V.

ii The GShare Predictor

- The Global History Register is a G bits wide shift register. The most recent branch is stored in the *least-significant-bit* of the GHR, and a value of 1 denotes a taken branch. The initial value of the GHR is 0.
- The Pattern Table (PT) contains 2^G smith counters. The Predictor XORs (Exclusive OR) the GHR with bits $[(2 + G - 1 : 2]$ of the branch PC to index into the PHT.
- Each Smith Counter in the PHT is 2-bits wide and initialized to $0b01$, the Weakly-NOT-TAKEN state.
- Although in a real implementation, the PT does not contain Tags, in this project to track interference, each PT entry has an associated Tag for the most recent branch to access the counter. Bits $[63 : G + 2]$ of the branch address make up the tag.

iii The Yeh-Patt Predictor

A two level adaptive branch predictor [1].

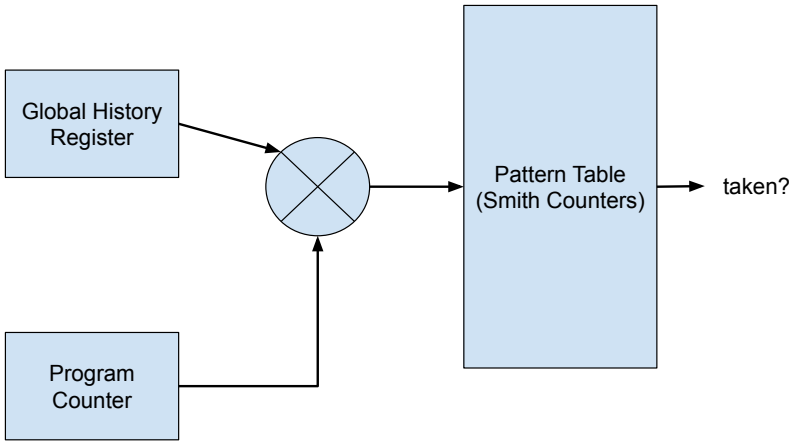

Figure 1: The architecture of the Gshare branch predictor

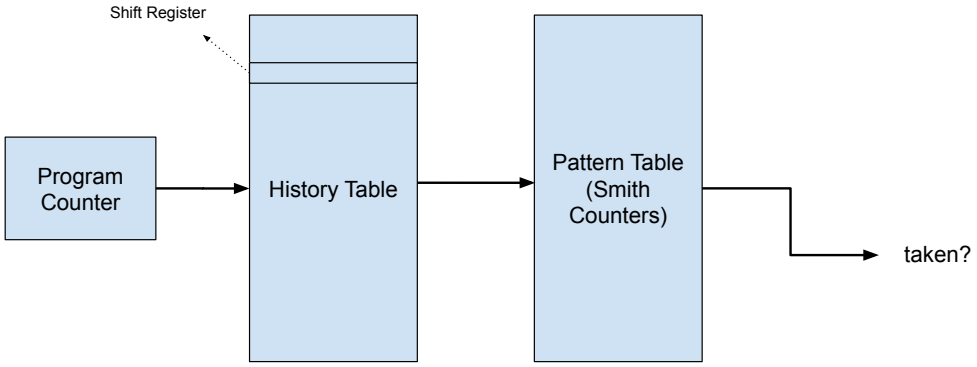

Figure 2: The architecture of the Yeh-Patt branch Predictor

- The History Table (HT) contains 2^G, P-bit wide shift registers. The most recent branch is stored in the *least-significant-bit* of these history register, and a value of 1 denotes a taken branch.
- The predictor uses bits $[(2+G-1:2]$ of the branch address (PC) to index into the History Table. All entries of the history table are set to 0 at the start of the simulation.
- The Pattern Table (PT) contains 2^P smith counters. The value read from the History Table is used to index into the pattern table.
- Each Smith Counter in the PT is 2-bits wide and initialized to $0b01$, the Weakly-NOT-TAKEN state.
- The History Table uses bits $[63:G+2]$ of the branch address as Tag bits per location of the HT. Again, this is unlike a real implementation, but required in order to track interference statistics.

iv Perceptron Predictor

The perceptron based branch predictor was one of the first forays of neural networks (in their simplest form) in the world of branch prediction [2]. Unlike Smith counter based predictors, perceptron branch predictors are able to make use of long branch histories and despite being more complex than alternatives can be implemented within moderate bit-budgets.

iv.1 Perceptron Prediction Architecture

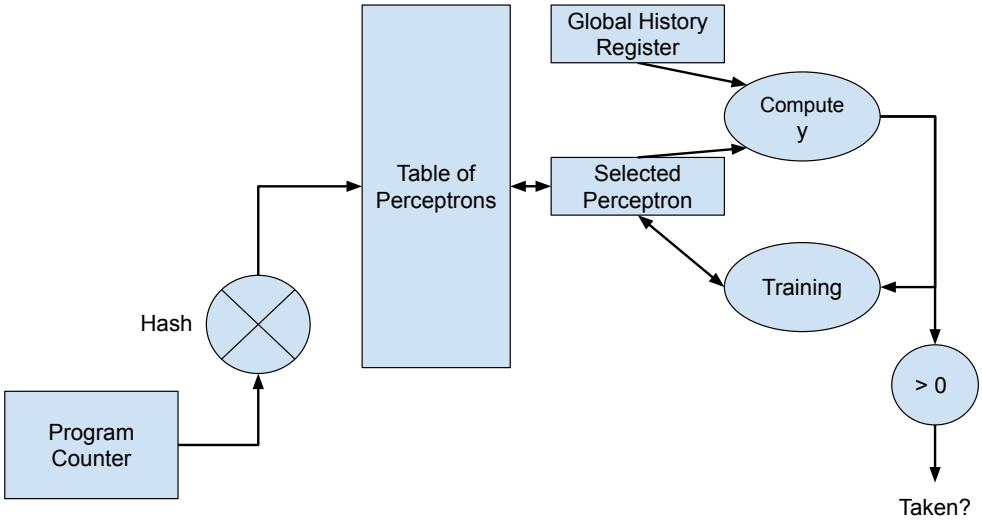

Figure 3: The architecture of the Perceptron based branch Predictor

- There is a global history register which is G bits wide.
- The perceptron table can hold 2^P perceptrons, that are indexed using bits $[(2+P-1):2]$ of the branch address. Each perceptron has $G+1$ weights, which can have a value in the range $[-(\theta+1), \theta]$, where $\theta = 1.93 \cdot G + 14$ (Found experimentally).
- Due to the unique nature of perceptrons for branch prediction, all weights, inputs, etc. are *signed integers*. Unlike other machine learning applications, using floating point numbers provides no observable advantages.
- Again to track interference statistics, each entry of the perceptron table has an associated Tag comprised of bits $[63 : P+2]$ of the branch address.

iv.2 Perceptron Prediction and Training

The perceptron predictors operations are slightly different from the gshare and Yeh-Patt predictors. The following steps are followed:

- The branch address (PC) is used to index into the perceptron table. The weights of the perceptron are called $w_0, w_1, ..., w_G$

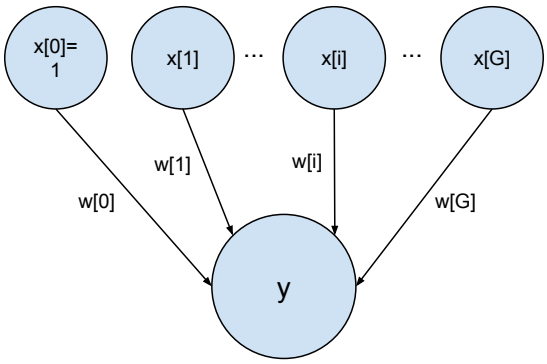

Figure 4: Perceptron Prediction Summary

- The inputs, i.e. a bi-polar representation of the global history register bits (say $x_i \forall i \in [1, G]$) and a constant 1 (x_0), are element wise multiplied with the weights and the partial results added together to form the output, y ($y = \sum_{i=0}^{G} w_i \cdot x_i$).
- The bi-polar representation here implies that a taken branch in the GHR is represented as a 1, while a not-taken branch is represented as -1. This implies that $x_i \forall i \in [1, G]$ are either 1 or -1, while $x_0 = 1$.
- Figure 4 shows a graphical representation of the process described above.
- **Prediction Output:** If the value of the perceptron output, y is negative, the branch is predicted as *not-taken*, and if the output is non-negative, the branch is predicted as *taken*
- **Training:** Once the branch outcome is determined, the perceptron used for the prediction is trained under the following criterion:
  - Recall from above that $\theta = 1.93 \cdot G + 14$. It is deemed to be the threshold value for when enough training has been done.
  - Let t be the actual behavior of the branch, i.e. $t = 1$ if the branch was actually taken and $t = -1$ if the branch was not-taken. The below algorithm is then run on the perceptron that needs to be trained:

```
if (sign(y) != t) or abs(y) <= theta:
    for i from 0 to G:
        w[i] = w[i] + t * x[i];
    end for
end if
// Note - x[0] is always 1
// sign(y) = -1 if y < 0 else 1
// abs(y) = absolute value of y
```

v Simulator Parameters

The following command line parameters can be passed to the simulator:

- `O` - The predictor option {1 - Gshare, 2 - Yeh-Patt, 3 - Perceptron}
- `G` - The number of bits in either History Table registers or the Global History Register
- `P` - Log_2 of the number of entries in the Pattern Table / Perceptron Table

- `I` - The input trace file
- `N` - The number of pipeline stages (used for calculating stalls per miss predict - see section vii)
- `H` - Print the usage information

vi Simulator Operation

The simulator operates in a trace driven manner and follows the below steps:

- The appropriate branch predictor initialization function is called where you setup the predictor data structures, etc.
- Branch instructions are read from the trace one at a time and the following operations ensue for each line of the trace:
  - The branch address is used to index into the predictor, and a direction prediction is made. The branch interference statistics are also updated at this time. The prediction is returned to the driver (true - TAKEN, false - NOT-TAKEN). Note:
    - That the steps to index into the pattern/perceptron table may be different for each predictor
    - A prediction is made regardless of the branch tag matching the tag being stored in the auxiliary first level table.
  - The driver calls the update prediction stats function. Here the actual behavior of the branch is compared against the prediction. A branch is considered correctly predicted if the direction (TAKEN/NOT-TAKEN) matches the actual behavior of the branch. Stats for tracking the accuracy of the branch predictor are updated here.
  - The branch predictor is updated with the actual behavior of the branch.
- A function to cleanup your predictor data structures (i.e. The destructor for the predictor) and a function to perform stat calculations is called by the driver.

Listing 1: Simulator Operation Overview Pseudo-code

```
predictor.init();   // Allocate predictor data structures, etc. here

while (NOT TRACE == EOF) {
    branch = read from trace;
    prediction = get_prediction(branch);
    update_prediction_stats_and_predictor(prediction);
    update_predictor(actual behavior);
}

branchsim_cleanup();
~predictor(); // Predictor destructor to release memory, etc.
```

vii Branch Prediction Statistics (The Output)

The branch prediction statistics are tracked in the below data structure. Apart from calculating the branch predictors accuracy, this simulator models the performance of a N-stage pipelined processor using a black box approach to CPI. The pipeline is assumed to be perfect (i.e. CPI = 1) except for stalls due to branch miss predictions. The average CPI accounting for branch miss predictions can be computed using:

$$Stall_{BranchMissPredict} = \begin{cases} 2 & \text{if } N \leq 7 \\ (N/2) - 1 & \text{if } N > 7 \end{cases}$$

$$CPI_{avg} = 1 + Stall_{AveragePerInstruction}$$

```
struct branch_sim_stats_t {
    // Look at branchsim.h for detailed comments
    uint64_t total_instructions;
    uint64_t num_branch_instructions;
    uint64_t num_branches_correctly_predicted;
    uint64_t num_branches_miss_predicted;
    uint64_t misses_per_kilo_instructions;
    uint64_t num_tag_conflicts;
    double fraction_branch_instructions;
    double prediction_accuracy;
    uint64_t N;
    uint64_t stalls_per_miss_predicted_branch;
    double average_CPI;
};
```

vii.1 Stats Explanations

Most stats are explained in the `branchsim.h` file. Clarifications will be added here.

- `num_tag_conflicts` - A tag conflict is when the tag of the current branch instruction does not match the tag of the last instruction to access that same entry in the first level of the predictor. In GShare, tags will be associated with entries in the Pattern Table, in Yeh-Patt, tags are associated with entries in the History Table, and in the Perceptron, tags are associated with entries in the Perceptron table, so that you can track this stat. *Additionally, the first access to each entry should count as a conflict to match the solution file.*

V Implementation Details

You have been provided with the following files:

- `src/`
  - `branchsim.cpp` - Your branch predictor implementations will go here

    The below header files contain class definitions for the 3 predictors. You can add class variables and data structures such as arrays, etc. for each predictor in its corresponding file:
  - `gshare.h` - The `gshare` predictor class definition
  - `yeh_patt.h` - The `yeh_patt` predictor class definition
  - `perceptron.h` - The `perceptron_predictor` class definition

    Don't modify/add any code in the below files!
  - `branchsim_driver.cpp` - The driver for the trace driven branch prediction simulator including the `main` function
  - `branchsim.h` - A header containing useful definitions and function declarations
- `CMakeLists.txt` : A cmake file that will generate a makefile for you.

- `traces/` : A folder containing branch traces from real SPEC 2017 programs. Each trace contains 10 Million branch instructions. A trace looks like:

```
7f7c619bee41    0       23
7f7c619bee72    1       32
7f7c619beea3    0       34
7f7c619beeb4    1       39
```

  The first column is the branch address (Program Counter / Instruction Pointer), the second column is the branch's actual behavior (TAKEN/NOT-TAKEN). The third column meanwhile indicates the instructions executed until (and including) the branch instruction.

  Note that you will have to decompress the `traces.tar.gz` file in the download in order to get the `traces/` directory.

i Provided Framework

I have provided you with a comprehensive framework where you will add data structure declarations and write the following functions in the provided `C++ classes` per branch predictor:

- ```
  void init_predictor(branchsim_conf *sim_conf)
  ```

  Initialize class variables, allocate memory, etc for the predictor in this function

- ```
  bool predict(branch *branch, branchsim_stats *stats)
  ```

  Predict a branch instruction and return a `bool` for the predicted direction {true (TAKEN) or false (NOT-TAKEN)}. The parameter `branch` is a structure defined as:

```
typedef struct branch_t {
    uint64_t ip;            // Branch address (PC)
    uint64_t inst_num;      // Instruction count of the branch
    bool is_taken;          // Actual direction behavior

    branch_t() { }
} branch;
```

- ```
  void update_predictor(branch *branch)
  ```

  Function to update the predictor internals such as the history register and smith counters, etc. based on the actual behavior of the branch

- ```
  branch_predictor::~branch_predictor()
  ```

  Destructor for the branch predictor. De-allocate any heap allocated memory or other data structures here

Apart from the per predictor functions, you will need to implement some general functions for book-keeping and final statistics calculations:

- ```
  void branchsim_update_stats(bool prediction, branch *branch,
     branchsim_stats *stats);
  ```

  Function to check the correctness of the prediction and update statistics.

- ```
  void branchsim_cleanup(branchsim_stats *stats);
  ```

  Function to perform final calculations such as miss-prediction rate, Average CPI, etc

ii Building the Simulator

I have provided you with a `CMakeLists.txt` file that can be used to build the simulator. `cmake` generates a makefile depending on the system configuration you are trying to build the simualator on. Follow these steps on a UNIX like machine:

```
Unix: $ cd <Project Directory>
Unix: $ mkdir build && cd build
Unix: $ cmake ..
Unix: $ make
```

This will generate an executable `branchsim` in the `<Project Directory>/build` folder. This is all that you need to know about using `cmake`. If you are interested in learning more about this build system, you can start by reading this link.

The generated executable `branchsim` can be run with the following command:

```
Unix: $ ./branchsim -i <Trace file> [Optional Parameters]
```

For example, if the executable is in the build folder as described above and the other directory structures have been preserved, you can run a simulation for the default configuration on the gcc trace by:

```
Unix: $ ./branchsim  -i ../traces/gcc.br.trace
```

Note: You will need to have cmake installed on the machine you are using. You should be able to install it using the package manager on your choice of Linux distribution or from this link if you are using a machine running MacOS.

iii Things to Watch Out For and Hints

This section describes the design choices the validation solution has made. You will need to follow these guidelines to perfectly match the validation log.

- Use type `uint64_t` for the history register in the perceptron predictor.
- All history registers and weights (where applicable) are set to zero at the start of the simulation. For the perceptron predictor this means all bits of the GHR are initially interpreted as -1 since it uses a bi-polar representation. θ is an integer, not a floating point number.
- When calculating the size of the perceptron predictor in bits for the experiments section, you can assume that each weight takes $floor(log_2(\theta)) + 1$ bits.

VI Validation Requirements

You must run your simulator and debug it until the statistics from your solution **perfectly (100 percent)** match the statistics in the validation log for all test configurations. This requirement must be completed before you can proceed to the next section.

A `Python` run script is provided which will run the simulator for various configurations and traces, and generate a log file. This verification script will run configurations for the predictor you specify as input. For example, you can run `python3 verify2.py 1` to run only GShare tests, which will generate a file called `gshare.out`. You can then compare this generated log file against the `gshare.solution` file. The script assumes that the simulator executable will be in the `build` folder.

Check the Canvas assignment for the updated verification files, `verify2.tar.gz`.

```
$ python3 verify2.py 1
$ diff -w gshare.out gshare.solution
  - or -
$ python3 verify2.py 2
$ diff -w yehpatt.out yehpatt.solution
 - or -
$ python3 verify2.py 3
$ diff -w perceptron.out perceptron.solution
```

.

VII Report

There is no report component for this project.

VIII What to Submit to Canvas

Compress the contents of your `src/` folder with the following command:

- `tar cvzf <lastname>_project2.tar.gz src`

Submit the following files to Canvas:

- `<lastname>_project2.tar.gz`
- `PREDICTOR.txt` - indicate in this file which predictors you implemented

IX Grading

You will be evaluated on the following criterion:

+0 :	You don't turn in anything (significant) by the deadline
+60 :	You turn in well commented significant code that compiles and runs and produces mostly correct output (+/- 10%)
+30 :	Your simulator **completely matches** the validation output
+10 :	Your code is well formatted, commented and does not have any memory leaks!

Appendix A - Plagiarism

Any and all cases of plagiarism are reported to the Dean of Students. You may not do the following in addition to the Georgia Tech Honor Code:

- Copy/share code from/with your fellow classmates or from people who might have taken this course in prior semesters.
- Look up solutions online. Trust me, I will know if you copy from online sources.
- Debug other people's code. You can ask for help with using debugging tools (Example: Hey XYZ, could you please show me how GDB works), but you may not ask or give help for debugging your simulator.
- You may not reuse any code from earlier courses even if you wrote it yourself. This means that you cannot reuse code that you might have written for this class if you had taken it in a prior semester. You must write all the code yourself and during this semester.

Appendix B - Helpful Tools

You might the following tools helpful:

- **GDB:** The GNU debugger will be really helpful when you eventually run into that seg fault. The cmake file provided to you enables the debug flag which generates the required symbol table for GDB by default.
- **Valgrind:** Valgrind is really useful for detecting memory leaks. Use the below command to track all leaks and errors.

```
Unix: $ valgrind --leak-check=full
   --show-leak-kinds=all --track-fds=yes
   --track-origins=yes -v ./branchsim <arguments as
   you would usually provide them>
```

References

[1] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24. New York, NY, USA: Association for Computing Machinery, 1991, p. 51–61. [Online]. Available: https://doi.org/10.1145/123465.123475

[2] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206.