CS {4/6}290 & ECE {4/6}100 - Spring 2023
Project 2 : Branch Prediction

Dr. Thomas Conte

Due: March 17th 2023 @ 11:55 PM

Version: 1.0.1

Change Log

- Version 1.0.1 (2023-03-03): The formula for the perceptron table height used the wrong Parameter ~~P~~ N. Updated the debug outputs to correctly display history length for Yeh-Patt Predictor, added logging for the perceptron weights, and clarified logging for tournament updates. Changes highlighted.

1 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.
- Please see the syllabus for the late policy for this course.
- Please use office hours for getting your questions answered. If you are unable to make it to office hours, please email the TAs.
- 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 **C99 or C++11** standards, using only the standard libraries.

2 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 Local History branch predictor (Yeh-Patt), a Perceptron branch predictor, and then a Tournament predictor between the two. 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 Ws: Whether to branch, Where to branch (if the branch is taken) and Which instruction is a branch. In this project we will concern ourselves with only the first W, i.e., Whether a branch is taken or not.

We have provided you with a framework that reads in branch traces generated from the execution of benchmarks from SPEC 2017 and drives your predictors. The simulator framework supports configurable predictor 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 3 provides the specifications of each type of branch predictor, and Section 4 provides useful information concerning the implementation.

Note: This project MUST be done individually. Please follow all the rules from Section 1 and review Appendix A.

3 Simulator Specifications

3.1 Basic Branch Predictor Architecture

Your simulator should model a Yeh-Patt predictor, Perceptron predictor, and a pc-indexed table of counters for a tournament. Figures 1, 2, 4 show the architecture diagrams of the Yeh-Patt, Perceptron, and Tournament 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 4.

3.2 Simulator Parameters

The following command line parameters can be passed to the simulator (details on the predictors are explained in subsequent sections):

- `-M`: The predictor option {1 - Yeh-Patt, 2 - Perceptron, 3 - Tournament}
- `-P`: Log_2 of the number of entries in the Pattern Table
  - **Restriction:** $P \geq 9$, small pattern tables are not very performant
- `-L`: Log_2 of the number of entries in the History Table
  - **Restriction:** $L \geq 9$, small history tables are not very performant
- `-N`: Log_2 of the number of entries in the Perceptron Table
  - **Restriction:** $N \geq 8$, Because perceptrons are slow to learn, aliasing can result in substantial performance degredation

- `-G`: Width of the Global History Register for the Perceptron Predictor
  - **Restriction:** $G \geq 32$, $G \leq 64$, Perceptrons do not perform well with short histories and do not scale with very large histories
- `-C`: Initial state of Tournament Counters {0 - Strongly Not Taken, 1 - Weakly Not Taken, 2 - Weakly Taken, 3 - Strongly Taken}
- `-I`: The input trace file
- `-H`: Print the usage information

Note that the upper bounds for some parameters are not defined. You will need to constrain these yourself when running the experiments.

3.3 The Yeh-Patt Predictor

A two level adaptive branch predictor [1].

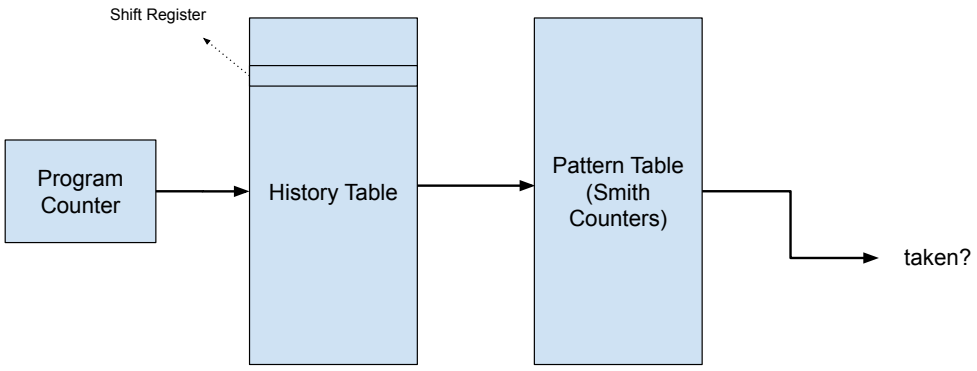

Figure 1: The architecture of the Yeh-Patt branch Predictor

- The History Table (HT) contains 2^L, P-bit wide shift registers. The most recent branch is stored in the *least-significant-bit* of these history registers (based on the branch pc), and a value of 1 denotes a taken branch.
  - When you update a history, shift it left by 1 and set the lowest bit based on the branch outcome.
- The predictor uses bits $[2 + L - 1 : 2]$ 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.
- This makes the total predictor size for the Yeh-Patt Predictor $2^L * P + 2 * 2^P$ bits.

3.4 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.

3.4.1 Perceptron Prediction Architecture

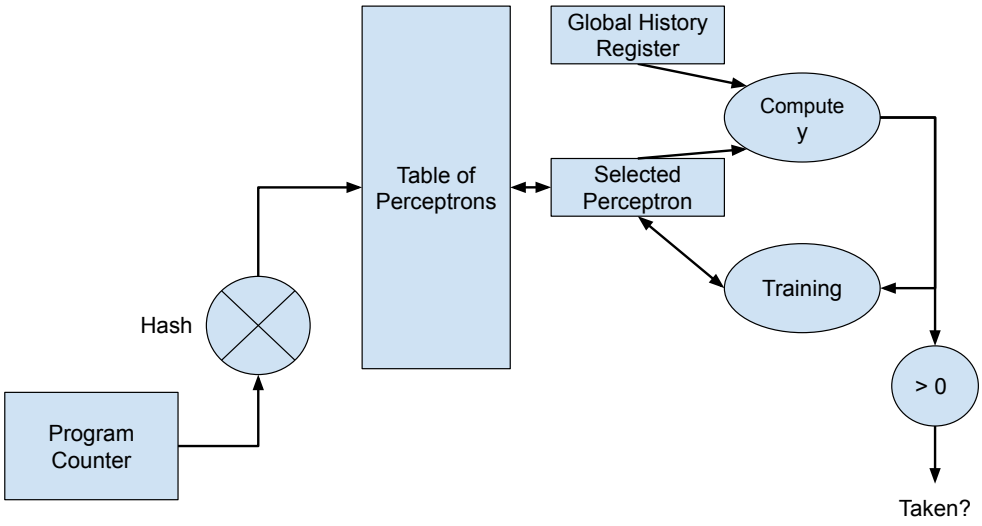

Figure 2: The architecture of the Perceptron based branch Predictor

- There is a global history register which is G bits wide.
- The perceptron table can hold ~~2^P~~ 2^N perceptrons, that are indexed using bits ~~$[(2+P-1):2]$~~ $[(2+N-1):2]$ of the branch address. Each perceptron has $G+1$ weights, which can have a value in the range $[-\theta, \theta]$, where $\theta = \lfloor 1.93 \cdot G + 14 \rfloor$ (Found experimentally). This means that the bitwidth of each weight is $floor(log2(\theta)) + 1$ since the weights are stored using a sign bit. Your implementation does not need to use the "sign-bit + integer" representation

  Note: While this bit width may allow for values outside the range, please adhere to the range as described: $[-\theta, \theta]$.
- 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 and require substantially more expensive hardware.
- The total storage budget for a perceptron predictor can be represented as: $(floor(log_2(1.93 * G + 14)) + 1) * (G + 1) * 2^N$ bits

3.4.2 Perceptron Prediction and Training

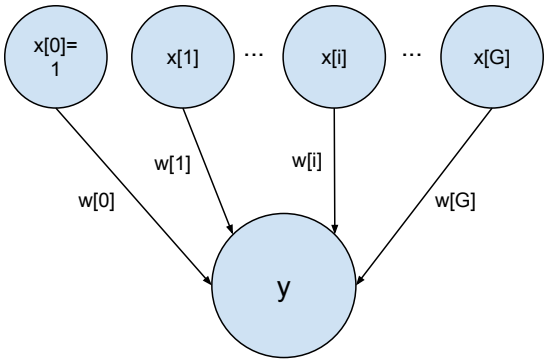

Figure 3: Perceptron Prediction Summary

The perceptron predictors operations are quite different from the counter based 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 $w_0, w_1, ..., w_G$. These weights are initialized to 0.
- 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 3 shows a graphical representation of the process described above.
- **Prediction Output:** If the value of the perceptron output, y is negative or zero, the branch is predicted as *not-taken*, and if the output is positive, 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 $\theta = 1.93 \cdot G + 14$. θ 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
// x[1] - x[G] are the global history
// sign(y) = -1 if y <= 0 else 1
// abs(y) = absolute value of y
```

3.5 Tournament Prediction

The tournament predictor uses a small table of (2^{12}) 4-bit saturating counters [3].

3.5.1 Tournament Predictor Architecture

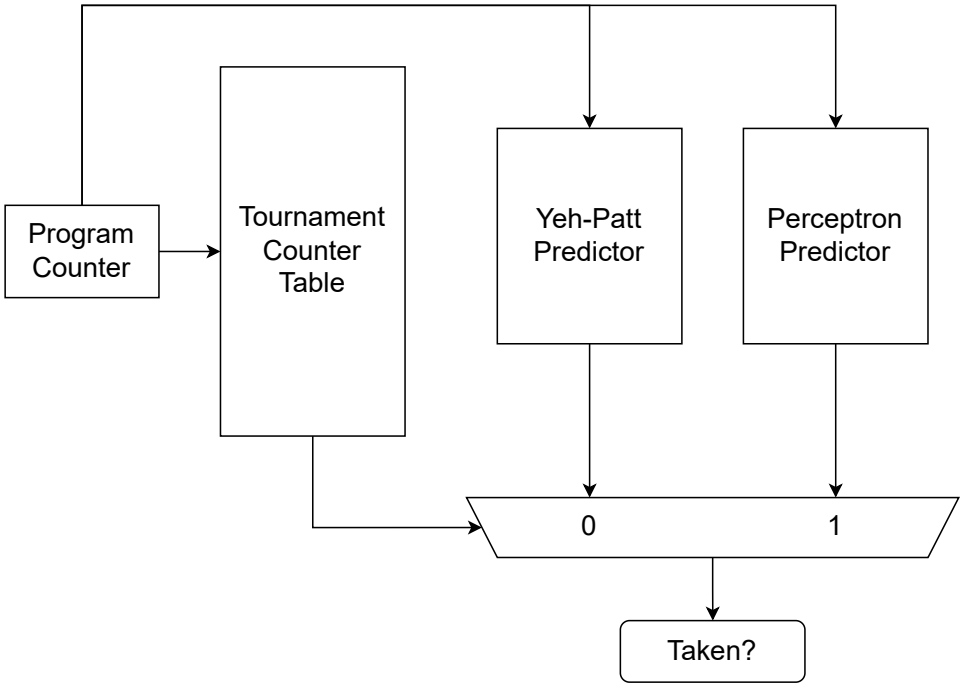

Figure 4: The architecture of the Tournament based branch Predictor between the Yeh-Patt predictor and the Perceptron predictor

- The tournament consists of 2^{12} entries.
- Each entry is a 4 bit saturating counter that is initialized to either:

  0: $4b0000$

  1: $4b0111$

  2: $4b1000$

  3: $4b1111$

3.5.2 Using the Tournament Predictor

- We choose a counter from the table by addressing it from bits $[13:2]$
- **Prediction Output:** If the upper bit of the counter is a 0, we select the prediction from the Yeh-Patt Predictor, if the upper bit is a 1, we select the prediction from the Perceptron Predictor.
- **Training:** We need to update the Yeh-Patt predictor, the Perceptron Predictor, and the tournament counters themselves.
  - Update the Yeh-Patt Predictor in the same way
  - Update the Perceptron Predictor in the way defined above
  - If the predictions from the Yeh-Patt predictor and the Perceptron predictor are not the same, we update the counter in the tournament counter table. We want the counter to have a higher value when the perceptron is correct, and lower when the yeh-patt is correct.

```
if perceptron_output != yeh_patt_output:
    if perceptron_output == actual_behavior:
        counter.update(true)
    else
        counter.update(false)
    end if
end if
```

3.6 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 predictor table(s) may be different for each predictor
  - 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_predictor(); // Allocate predictor data structures etc.

while (trace not at EOF) {
    branch = read from trace;
    prediction = get_prediction(branch);
    update_prediction_stats(branch, prediction, actual behavior);
    update_predictor(branch, actual behavior);
}

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

3.7 Branch Prediction Statistics (The Output)

The branch prediction statistics are tracked in the below data structure.

```
struct branch_sim_stats_t {
    // Look at branchsim.hpp for detailed comments
    uint64_t total_instructions;
    uint64_t num_branch_instructions;
    uint64_t num_branches_correctly_predicted;
    uint64_t num_branches_mispredicted;
    uint64_t misses_per_kilo_instructions;
    double fraction_branch_instructions;
    double prediction_accuracy;
};
```

4 Implementation Details

You have been provided with the following files:

- `src/`
  - `branchsim.cpp` - Your branch predictor implementations will go here
  - `Counter.hpp` - definition of the Counter class.
  - `Counter.cpp` - Implementation of the Counter class. You will need to implement the counter based on the comments in the header file (`Counter.hpp`)

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

    You are strongly encouraged to leave the files below alone
  - `branchsim_driver.cpp` - The driver for the trace driven branch prediction simulator including the `main` function
  - `branchsim.hpp` - A header containing useful definitions and function declarations
- `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.

4.1 Tournament Implementation Suggestions

For the tournament predictor, you are encouraged to create instances of the yeh-patt and perceptron predictor you already created. You can use the methods you implement for `init_predictor()`, `predict()`, `update_predictor()` from within the tournament methods to avoid re-implementing the predictors.

4.2 Provided Framework

We 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)`

  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 branch outcome
} 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 update statistics based on the correctness of the prediction (you can use the `is_taken` field of the `branch` to check if the branch was actually taken or not).

- `void branchsim_finish_stats(branchsim_stats *stats);`

  Function to perform final calculations such as misprediction rate, etc

4.3 Docker Image

We have provided an Ubuntu 22.04 LTS Docker image for verifying that your code compiles and runs in the environment we expect — it is also useful if you do not have a Linux machine handy. To use it, install Docker (`https://docs.docker.com/get-docker/`) and extract the provided project tarball and run the `6290docker*` script in the project tarball corresponding to your OS. That should open an Ubuntu 22.04 `bash` shell where the project folder is mounted as a volume at the current directory in the container, allowing you to run whatever commands you want, such as `make`, `gdb`, `valgrind`, `./branchsim`, etc.

> **Note**: Using this Docker image is not required if you have a Linux system (or environment) available and just want to make sure your code compiles on 22.04. We will set up a Gradescope autograder that automatically verifies we can successfully compile your submission.

5 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.

You can run `make validate` to compare your output with the reference outputs. If you want to test only one configuration for all four benchmarks, you can use the `validate.sh` script directly and pass it a configuration name it understands, like `./validate.sh tage_1k_p3`.

We do not have a hard efficiency requirement in this assignment, but please make sure your simulator finishes a simulation (particularly for TAGE) for one of the provided traces in a few minutes. Otherwise, it will be difficult for the TAs to verify your code produces matching outputs.

5.1 Debug Outputs

We have provided debug outputs for you in the `debug_outs` directory. These use the 100k branch trace for `gcc` and should cover most code paths in your simulator. We have also provided `debug_printfs.txt` which contains all of the print statements in the simulator that were used to generate the debug outputs. To enable debug output generation in your own code, use `make DEBUG=1` (you should `make clean` first!) which will allow you to use preprocessor directives to control whether or not your code generates print statements, like this:

```
#ifdef DEBUG
    printf("Yeh-Patt: Creating a history table of %" PRIu64
        " entries of length %" PRIu64 "\n", myvar1, myvar2);
#endif
```

5.2 Debugging

To debug, please use GDB and Valgrind as indicated in Appendix B. We recommend running the 100,000-branch trace `gcc.100k.br.trace`, i.e., passing `-I traces/gcc.100k.br.trace` to `branchsim`, since the 10 million–branch traces will likely take a while to run under GDB or Valgrind. We also encourage comparing with the debug outputs (the tool `diff` is useful) before coming to office hours for help debugging your code.

6 Experiments

Once you have debugged and validated your branch predictors, you will find a single optimum (optimum = highest accuracy) Tournament branch predictor under the following constraints:

- A predictor can be at most 26KiB (1 KiB = 1024 Bytes = 8192 bits) in terms of total size. This does not include the storage for history registers, but does include the storage used for smith counter tables, history tables, perceptron tables, etc (where applicable). (You should be varying `-P`, `-L`, `-N`, `-G`, and `-C`)
- 2KiB are always consumed by the Tournament Predictor's counter table
- The Yeh-Patt and Perceptron Predictors must each be at least 6KiB in size
- The total storage size for the Yeh-Patt and Perceptron Predictors cannot exceed 24KiB
- You will select a singular predictor that performs the best across the board (all traces).

Write a short report on your findings, describing the optimum branch predictor for the pipeline and explain any surprising and unexpected results. Ensure that the report is in a file named `<last name>_report.pdf`.

7 What to Submit to Gradescope

Please run `make submit` and submit the resulting tarball (`tar.gz`) to Gradescope. The `Makefile` will include PDFs in the project directory in the tarball, but please make sure it worked properly and your experiments report PDF is present in your submission tarball. We have created a simple Gradescope autograder that will verify that your code compiles and matches the reference traces for the 10 million–branch `gcc` trace. This autograder is a smoke test to check for any incompatibilities or big issues; it is not comprehensive.
Make sure you untar and check your submission tarball to ensure that all the required files are present in the tar before submitting to Gradescope!

8 Grading

You will be evaluated on the following criteria:

+0 :	You don't turn in anything (significant) by the deadline
+50 :	You turn in well commented significant code that compiles and runs but does **not** match the validation
+10 :	Your simulator **completely matches** the validation outputs for Yeh-Patt
+15 :	Your simulator **completely matches** the validation outputs for Perceptron
+10 :	Your simulator **completely matches** the validation outputs for Tournament
+10 :	You ran experiments and found the optimum configuration for the 'experiments' workload and presented sufficient evidence and reasoning
+5 :	Your code is well formatted, commented and does not have any memory leaks! We may also run some special cases for these points. Check out Appendix B for some useful tools

Points for the experiments and/or the memory leak check cannot be awarded without first matching all validation traces. This is non-negotiable.

Appendix A - Plagiarism

Preamble: The goal of all assignments in this course is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. As a Georgia Tech student, you have read and agreed to the Georgia Tech Honor Code. The Honor Code defines Academic Misconduct as "any act that does or could improperly distort Student grades or other Student academic records."

2. You must submit an assignment or project as your own work. Absolutely No Collaboration on Answers Is Permitted. Absolutely no code or answers may be copied from others — such copying is Academic Misconduct. NOTE: Debugging someone else's code is (inappropriate) collaboration.

3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person*").

4. Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.

5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.

6. Students suspected of Academic Misconduct are informed at the end of the semester. Suspects receive an Incomplete final grade until the issue is resolved.

7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.

8. If you are not sure about any aspect of this policy, please ask Dr. Conte.

Appendix B - Helpful Tools

You might the following tools helpful:

- gdb: The GNU debugger will prove invaluable when you eventually run into that segfault. The Makefile provided to you enables the debug flag which generates the required symbol table for gdb by default.
  - You can invoke gdb with `gdb ./branchsim` and then run `run -I traces/gcc.100k.br.trace <more branchsim args>` at the gdb prompt
- Valgrind: Valgrind is really useful for detecting memory leaks. Use the following command to track all leaks and errors:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
    ./branchsim -I traces/mcf.10m.br.trace <more branchsim args>
```

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.

[3] S. McFarling, "Combining Branch Predictors," 1993.