CS {4/6}290 & ECE {4/6}100 - Spring 2022
Project 2 : Branch Prediction

Dr. Thomas Conte

Due: March 11th 2022 @ 11:55 PM

Version: 1.3

Change Log

1. Version 1.1 (2022-02-22): The formula for geometric history incorrectly indicated the recurrence relation in history lengths. Changes highlighted.
2. Version 1.2 (2022-02-25): Formula for Hash 2 of GShare incorrectly selected the bits for the index. Also clarified meaning of `-S`. Changes highlighted.
3. Version 1.3 (2022-03-01): Typo in Example 4 resolved. Added some notes on debugging and the Gradescope smoke test autograder. Student noticed a bug in the debug traces (size was printed incorrectly, remainder of trace was correct), this has been corrected. Updated Experiments section. 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[1].

[1] If you choose to write your project from scratch in Java (highly discouraged), please use Java 11 and only the standard library for Java 11.

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 Global History branch predictor (GShare) and a simplified version of the state-of-the art, TAGE (TAgged GEometric history). 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 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 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 GShare (Table of Smith Counters) or set of TAGE tables, and a global history register. Figures 1, 2 show the architecture diagrams of the GShare and TAGE 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 4.

3.2 Simulator Parameters

The following command line parameters can be passed to the simulator (details on GShare and TAGE-S are explained in subsequent sections):

- `-O`: The predictor option {1 - GShare, 2 - TAGE-S}
- `-S`: The total size of the predictor in $\log_2(\text{total\_bits})$ This must be in the range $[13, 19]$ (1KiB-64KiB). This does not include GHR registers etc.
- `-P`: The "parameter," this will be used to determine the hash function when using GShare, and the number of tagged tables for TAGE-S. When using GShare, the range for P is $[0, 2]$; for TAGE, the range is $[3, 10]$
- `-I`: The input trace file
- `-N`: The number of pipeline stages used in final stats calculation after the simulation. See Section 3.6
- `-H`: Print the usage information

3.3 The GShare Predictor

GShare is a cheap branch predictor discussed in class which hashes the PC and GHR to index into a table of Smith counters [1].

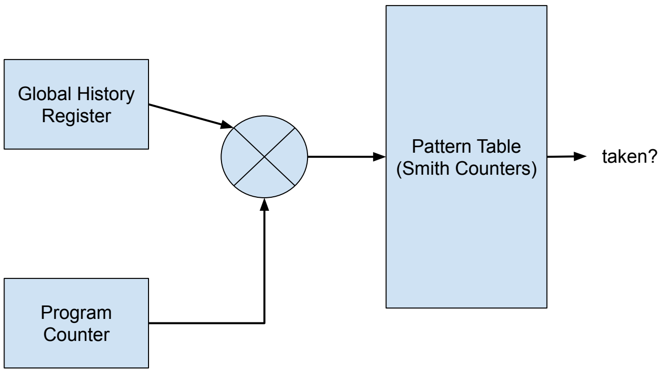

Figure 1: The architecture of the GShare branch predictor

- 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 Table contains 2^G smith counters. The Predictor hashes the GHR with the branch PC to index into the table.
- The simulator parameter P will decide the hash function:
    0. Hash(GHR, PC) = GHR $\bigoplus$ PC$[2 + G - 1 : 2]$
    1. Hash(GHR, PC) = GHR $\bigoplus$ PC$[G - 1 : 0]$
    2. Hash(GHR, PC) = (PC$[2 + G - 1 : 2]$ + GHR)$[G - 1 : 0]$ (Add the shifted PC to the GHR and truncate to G bits.)
- Each Smith Counter in the Table is 2-bits wide and initialized to 2'b10, the Weakly-Taken state.
- Although in a real implementation, GShare entries do not contain tags, in this project (to track interferences) each table entry has an associated tag for the most recent branch to access the counter. Since the hash function can cause aliasing, the tag for each entry is the entire branch address. (Since they are only present in the simulator to collect statistics, do not factor in the bits required to store this tag when determining the size of the table)
- Based on S (total size of predictor in $\log_2$(bits)), you will need to calculate G. Remember that Smith counters are saturating 2-bit counters.

3.4 The TAGE Predictor

NOTE: The version of TAGE we implement for this project is substantially simplified from the original TAGE predictor proposed in 2006.

The TAGE predictor [2] is a combination of 2 predictors, the GPPM (Global History Prediction by Partial Matching) predictor [3] and the O-GEHL (Optimized GEometric History Length) predictor [4]. While we do not require that you read/skim the papers relating to these predictors (as they will not be discussed in class nor will you be tested on their details), they are provided below for you to read/reference if you wish. Please do not try to implement the papers in this project as they are significantly more complex and will give drastically different performance results.

- TAGE: https://jilp.org/vol8/v8paper1.pdf
- O-GEHL: https://www.irisa.fr/caps/people/seznec/ISCA05.pdf
- GPPM: https://jilp.org/vol7/v7paper10.pdf

While the TAGE predictor is relatively simple to implement in hardware, simulation of its actions can be quite difficult and as such, much of the complexity has been stripped out or black-boxed for this project. The version of TAGE you will implement is TAGE-S (TAGE-Simplified).

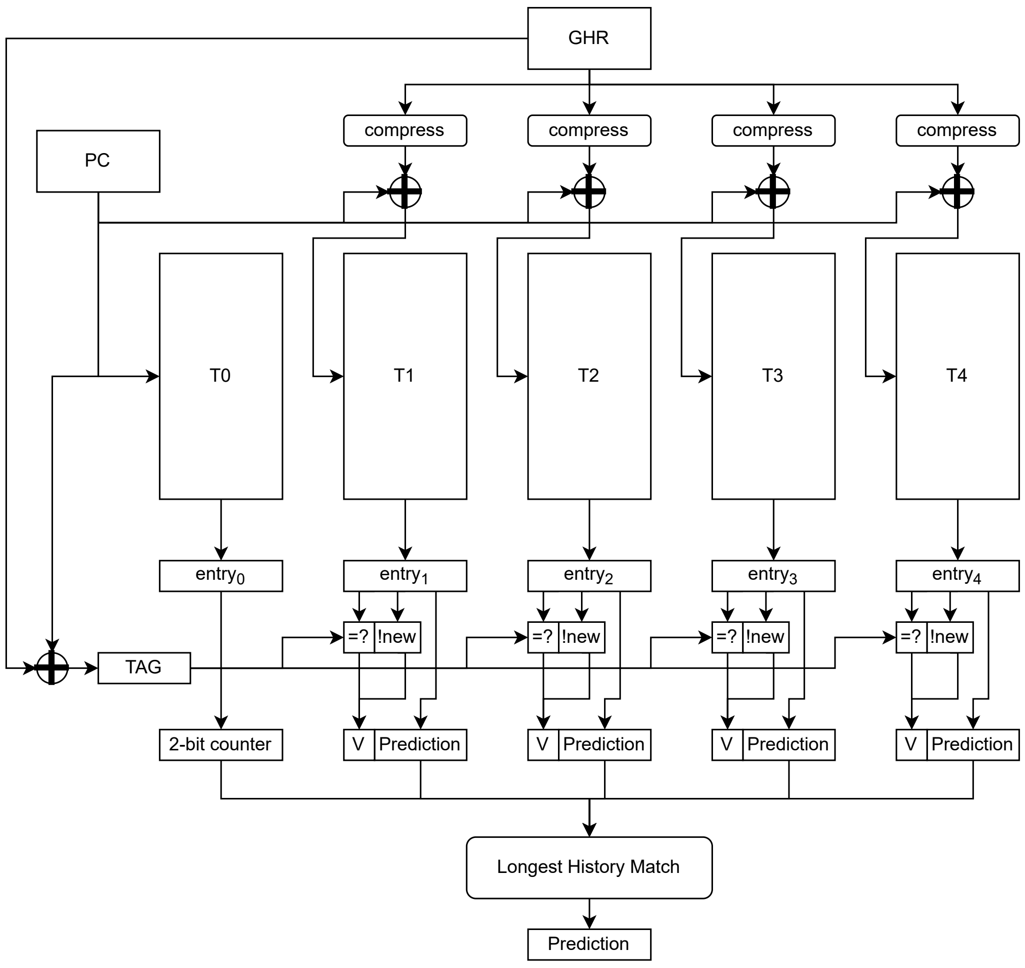

Figure 2: The architecture of the Simplified TAGE (TAGE-S) predictor with P=4

The characteristics of the TAGE-S predictor you will implement are as follows:

1. The TAGE predictor uses a very long global history (hundred bits range) to determine far reaching branch relations. (We have provided a TAGE_GHR class that you will use to maintain and access this history).
2. The TAGE predictor consists of P+1 tables, T_0 and $T_1 - T_P$
   (a) T_0 is a small Bimodal Predictor table with 2^H entries (2-bit Smith counters). This is very similar to GShare except that only the PC$[2 + H - 1 : 2]$ bits are used to index into the table. For the sake of the project $H = S - 4$.
   (b) $T_1 - T_P$ are "Tagged Tables" comprised of many entries consisting of a **T-bit partial-tag, a 3-bit prediction counter, and a 2-bit useful counter**. For the sake of the project, $T = P + 4$. In the initial state, the GHR will be 0, partial tags will be 0, prediction counters will be weakly taken for T_0 and 0 for tagged table entries, and useful counters will be 0.
3. The tagged tables will all have the same number of entries (you will need to calculate this based on the information provided). These are indexed using an XOR of $PC[2+E-1 : 2]$ and the "compressed history."
   (a) We use XOR to compress the global history into a smaller number of bits while retaining some information. For example, with an 80 bit history and a tagged table with 10 index bits, we break the history into 8 chunks of 10 bits and XOR all of them together. This compression is performed for you in the `TAGE_GHR` class (provided code).
4. Each tagged table will have the same number of entries (2^E). Since an entry has a partial tag, 3-bit prediction counter and 2-bit useful counter, size of the entry is known. You also know how much of the budget will be used by T_0. Thus, you should be able to calculate the number of entries you can fit in 2^S bits. Remember that the number of entries must be a power of 2 (E is an integer). (This means there is a chance you will have extra space in your budget you will not be using for anything)
5. The partial tag is calculated by using the lower bits of the uncompressed GHR and using XOR to combine the information with the lower bits of the PC that have not been used for another function yet. TAG = GHR$[T - 1 : 0] \bigoplus$ PC$[2 + E + T - 1 : 2 + E]$.
6. The length of history that is used for each table is a geometric series given below. We have simplified it for this assignment. Remember to use floating point arithmetic for this step and then floor (truncate) the value to the nearest integer.
   - $L(x)$ corresponds to the length of the history for table T_x
   - $L(1) = \lfloor E/2.0 + 0.5 \rfloor$
   - $\alpha = 1.75$
   - $L(i) = \lfloor \alpha * L(i - 1) + 0.5 \rfloor$
   - Remember that each value L(i) is an integer.

3.4.1 Getting a Prediction from TAGE

Each of the tables $T_0 - T_P$ is indexed using its associated index function and compressed history (if relevant). We then search the P+1 tables for all entries that match the associated partial tag (note T_0 has no partial tags, so it always has a matching entry: the entry at the index calculated by the index function). We search the matching entries from each table (if present) to find the one which has the longest history *AND* is not "new" (see below). The entry found is used to predict.

A “new” entry is any entry that has its prediction counter in the Weakly Taken or Weakly Not Taken states (3’b100 and 3’b011 respectively) *AND* has its useful counter set to 0. Notice that T_0 will always have a matching entry as it has no partial tag to match, but if T_0 is selected, this means that there is no entry in the tagged tables for this combination of branch address and global history that is not “new.” We use the “num_tag_conflicts” statistic to track the number of times that the T_0 is the table we use for prediction. Figure 2 shows how a prediction is retrieved from the TAGE structure.

Example 1: If we have an 8 table TAGE (P = 7), and we have a partial tag match on T_2, T_4, and T_6 and the implicit T_0 hit. Lets say $entry_6$ has a prediction counter in the weakly taken state and a useful counter that is 0. This means that despite T_6 considering the longest history and having a partial tag match, we use T_4 ($entry_4$) to get the prediction for this event.

Example 2: If we have a 5 table TAGE (P = 4) and a partial tag match on T_1 but the entry has a useful counter of 0 and a weak prediction counter, we would then use the prediction from T_0 ($entry_0$) and increment “num_tag_conflicts”.

3.4.2 Updating TAGE

Updating the Longest Matching Entry in TAGE: The first step of updating TAGE is to determine which table has the longest match as above, except this time "new" entries are allowed. One of the following cases can occur:

1. We only hit on T_0 (that is, we don't hit on any tagged tables): we update the associated Smith counter
2. We hit on a tagged table: We update the longest matching entry. The useful counter is incremented if the prediction of the entry is correct and decremented otherwise. The prediction counter of the entry is updated according to the branch outcome.

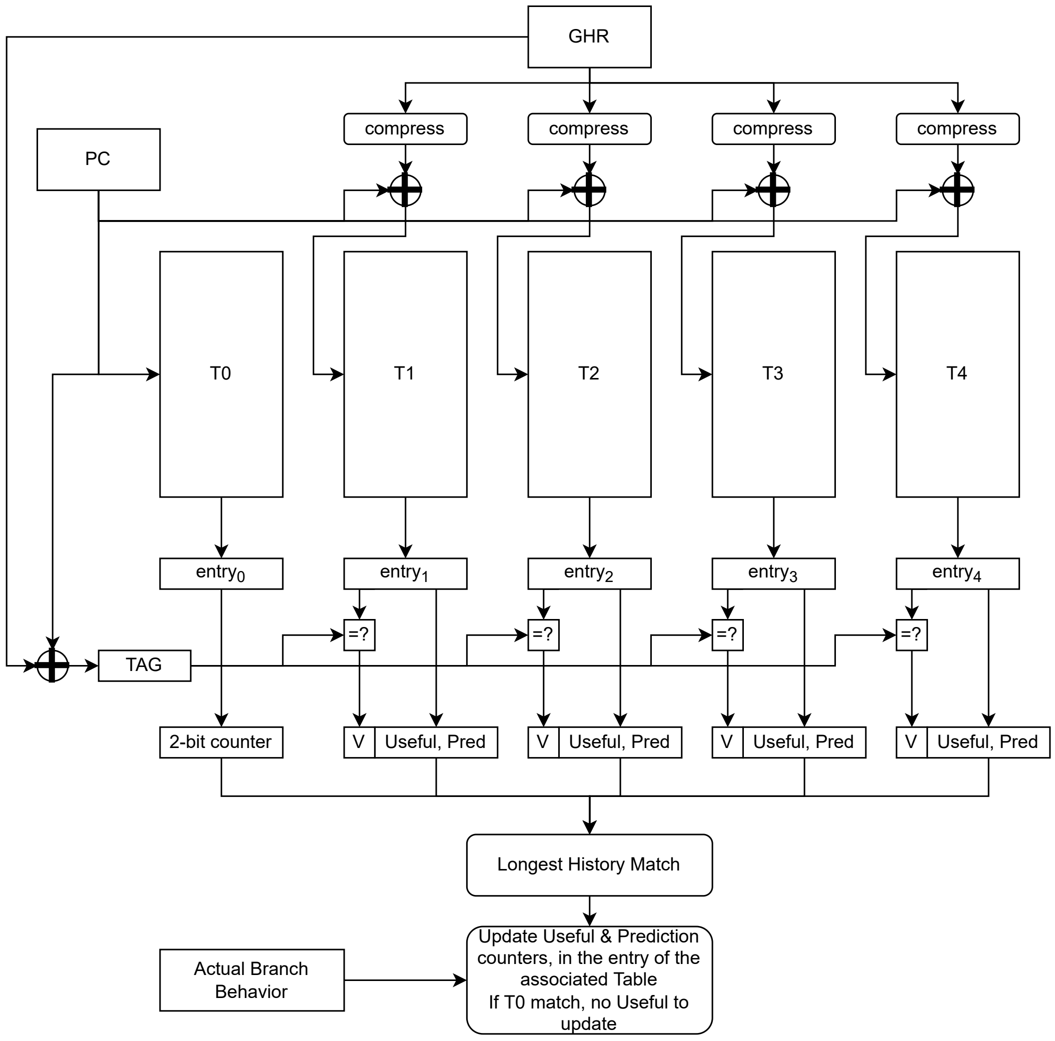

Figure 3: Architecture of the update process for TAGE-S with P=4. Note that "new" entries can be used here

NOTE: the longest matching entry may not be the same entry used to make the prediction since we may update a "new" entry that was ignored for prediction.

Allocating New Entries in TAGE: After we update the longest matching entry, we will handle the process of allocating new entries ("allocating" $\neq$ `malloc()`; we mean modifying a table). You will notice that the above cases do not handle the allocation of entries in the tagged tables. If the prediction of the counter of the longest matching entry found above does not match the branch behavior, we will attempt to allocate a new entry in a table for longer global history. Notice that the longest matching table's prediction may not be the same prediction you made for the branch because we are filtering out "new" entries for the prediction step.

We again look at all of the tagged tables at their respective index functions. We look for entries in the tables that consider longer global histories than the longest match. We will select the first table with a useful counter value of 0. One of two cases can occur:

1. There is no entry with useful counter 0: we decrement the useful counter of the entries that consider a longer history than the longest matching entry.
2. We find an entry that considers a longer history than the longest match AND has a useful counter == 0: We update the entry to contain the partial tag for this combination of GHR and branch address. We set the useful counter of the entry to 0. We set the prediction counter to be weak in the direction of the branch (if branch is actually taken, counter is set to weakly taken, and if branch is actually not taken we set the counter to not taken.)

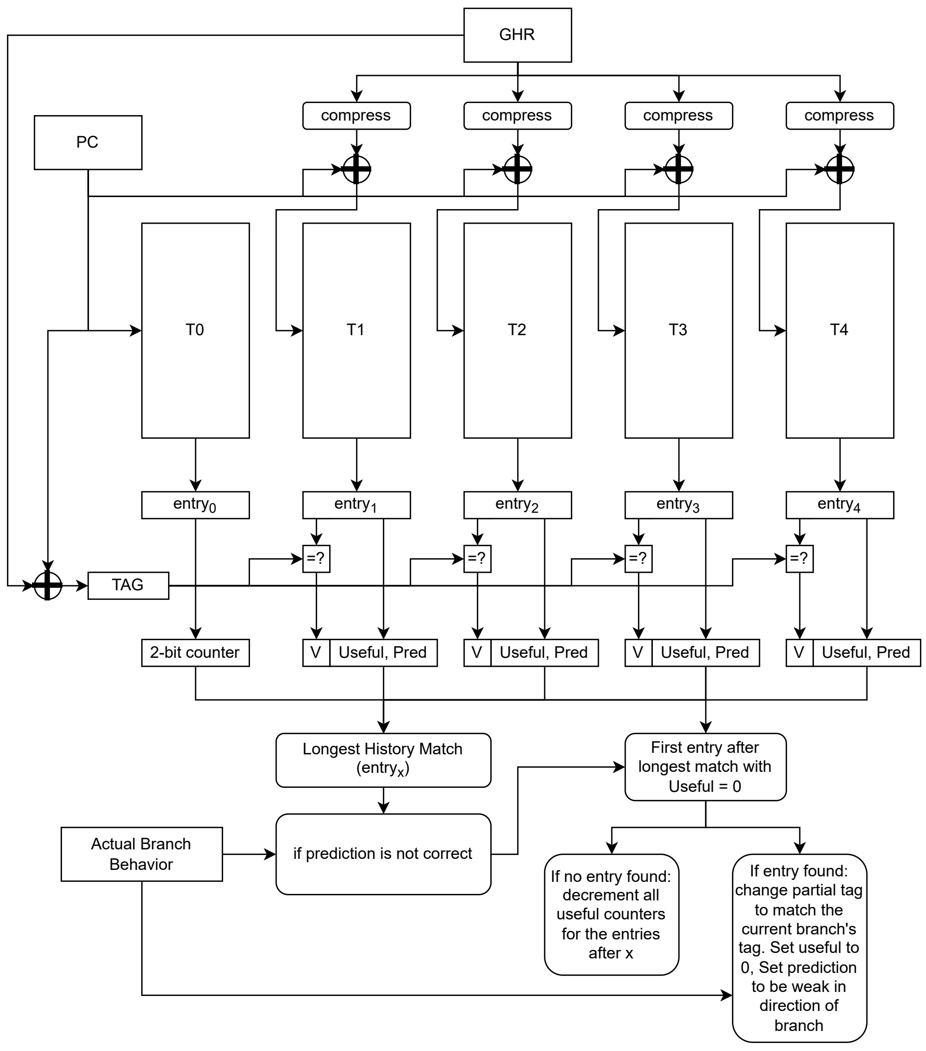
GHR
compress
compress
compress
compress
PC
T0
T1
T2
T3
T4
entry$_0$
entry$_1$
entry$_2$
entry$_3$
entry$_4$
=?
TAG
2-bit counter
V
Useful, Pred
Longest History Match (entry$_x$)
First entry after longest match with Useful = 0
Actual Branch Behavior
if prediction is not correct
If no entry found: decrement all useful counters for the entries after x
If entry found: change partial tag to match the current branch's tag. Set useful to 0, Set prediction to be weak in direction of branch

Figure 4: Architecture of the allocation process for TAGE-S with P=4

Finally, we update the GHR. That is, we add the actual outcome to the GHR only after all the steps above.

Example 3: Consider TAGE-S with $P = 7$, where we have no matches in any of the tagged tables (T_1 through T_7). There is always a match in T_0, so we use the Smith counter at the T_0 entry to make the prediction. In this example, suppose the Smith counter in the bimodal table entry is 2'b01 (weakly not-taken); thus, the TAGE prediction was not-taken. Now suppose the actual outcome of the branch is taken. To update TAGE (per Section 3.4.2), we update the entry in T_0, since there are no matches in tagged tables. We transition the Smith counter for the T_0 entry from 2'b01 (weakly not-taken) to 2'b10 (weakly taken). Now the entry in T_0 has a prediction of taken (since it is 2'b10), and this matches the actual branch outcome, so we do not proceed with allocating new entries. The GHR is then updated.

Example 4: Consider a TAGE-S with $P = 4$, where we have a tag match in T_2 ($entry_2$: useful = 3, prediction = strongly taken (3'b111)) and T_3 ($entry_3$: useful = 0, prediction = weak not-taken (3'b011)). Based on Figure 2, we would use the $entry_2$ for prediction (taken). The true outcome of the branch is taken. Based on Figure 3 we would update $entry_3$, decrement its useful counter (still 0) and update the prediction counter to weakly taken (3'b100). Now, when we look at allocation (Figure 4), we find that the prediction is correct since we updated the T_3 prediction counter. No new entry is allocated. The GHR is then updated.

Example 5: Consider a similar situation with TAGE-S and $P = 4$, where we have a tag match in T_1 ($entry_1$: useful = 3, prediction = strongly taken (3'b111)) and T_2 ($entry_2$: useful = 1, prediction = medium not-taken (3'b010)). Based on Figure 2, we would use $entry_2$ for prediction (not-taken). The true outcome of the branch is taken. Based on Figure 3 we would update $entry_2$, decrement its useful counter (0) and update the prediction counter to weakly not-taken (3'b011). Now when we look to allocate (Figure 4) we see that we need to allocate a new entry since the updated $entry_2$ prediction does not match the true behavior. We look at the entries in the tables after T_2. This means we look at the T_3 and T_4 entries. A few cases may occur:

1. Both entries have useful counters = 0: we allocate in $entry_3$ (the first entry in a table after T_2). We set the tag to the current tag, set the useful counter to 0, and set the prediction counter to be weak in the direction of true branch behavior (weak taken = 3'b100)
2. Neither entry has a useful counter = 0: we decrement the useful counters for $entry_3$ and $entry_4$.
3. Only $entry_3$ has useful counter = 0: we allocate in $entry_3$. We set the tag to the current tag, set the useful counter to 0, and set the prediction counter to be weak in the direction of true branch behavior (weak taken = 3'b100)
4. Only $entry_4$ has useful counter = 0: we allocate in $entry_4$. We set the tag to the current tag, set the useful counter to 0, and set the prediction counter to be weak in the direction of true branch behavior (weak taken = 3'b100)

We commit the updated entries back to their associated tables. The GHR is then updated.

3.5 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.
```

3.6 Branch Prediction Statistics (The Output)

The branch prediction statistics are tracked in the below data structure. Apart from calculating the branch predictor's 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:

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

$$CPI_{avg} = 1 + \text{Stalls}_{\text{AveragePerInstruction}}$$

```
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;
    uint64_t num_tag_conflicts;
    double fraction_branch_instructions;
    double prediction_accuracy;
    uint64_t N;
    uint64_t stalls_per_mispredicted_branch;
    double average_CPI;
};
```

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:
  - `gshare.hpp` - The `gshare` predictor class definition
  - `tage.hpp` - The `tage` predictor class definition

    It is strongly discouraged to modify/add any code in the below files!
  - `TAGE_GHR.hpp` - definition of the TAGE_GHR class
  - `TAGE_GHR.cpp` - implementation of the TAGE_GHR class. Do not modify this class.
  - `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 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, 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 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, Average CPI, etc

4.2 Implementation in Java (Strongly Discouraged)

If you make the highly discouraged choice to write your project from scratch in Java 11, you need to set up the `Makefile` with a default target that will compile your code with `javac`. Please also include a `clean` target that will remove `.class` files. You will also need to set up `./run.sh` to run your Java simulator, which must accept all the same configuration flags as `branchsim_driver.cpp` (`-I`, `-O`, `-S`, etc.). Your code must also parse the provided trace files, or any other trace files of the same format. Your code must depend only on any libraries or functions included in the standard library for Java 11.

4.3 Docker Image

We have provided an Ubuntu 18.04 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 18.04 `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 18.04. 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. The lower bits of GHR in the TAGE debug outputs are retrieved using `getHistory()`. 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("Using GShare, size: %d KiB, G: %d, using hash function: %d\n",
           my_var1, my_var2, my_var3);
#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 at the lowest bit budget) branch predictor for a **50-Stage** pipeline (for the provided traces) under the following constraints:

- A predictor can be in the range [1 KB - 64 KB] (1 KB = 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 and tagged tables (where applicable). (You should be varying `-O` and `-P`)

- ~~We don't suggest trying to brute force the search space, but instead running a few distinct configurations to understand trends which you can then explain in the report.~~

- It is feasible to do a brute-force search of the space of parameters, if you choose not to do this be very careful as you don't want to miss a great configuration!

- You will select a singular predictor for the 50 stage pipeline 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 GShare
+20 :	Your simulator **completely matches** the validation outputs for TAGE
+15 :	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! 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] S. McFarling, “Combining Branch Predictors,” 1993.

[2] A. Seznec and P. Michaud, “A case for (partially) tagged geometric history length branch prediction,” *The Journal of Instruction-Level Parallelism*, vol. 8, 2006.

[3] P. Michaud, “A PPM-like, tag-based branch predictor,” *The Journal of Instruction-Level Parallelism*, vol. 7, p. 10, Apr. 2005.

[4] A. Seznec, “Analysis of the O-GEometric history length branch predictor,” in *32nd International Symposium on Computer Architecture (ISCA’05)*, Jun. 2005, pp. 394–405, iSSN: 1063-6897.