NC STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 463/521: FALL 2016: PROJECT 2: BRANCH PREDICTOR

Due: Tuesday Nov 8th, 11:59 PM

1. Ground rules

- This is an *individual* project.
- Collaboration by sharing code is strictly prohibited and is considered cheating.
- A TA will scan code from current *and past* semesters through automated tools available to us that can detect cheating; code that is flagged by these tools will be dealt with severely and will receive appropriate action in accordance with university policy.
- Use the tag Project2 in the Discussion Forum for questions (or write to TA/instructor).
- Don't submit any *code* in the Discussion Forum (check with TA or instructor if in doubt)
- You *must* use C/C++ programming languages. Exceptions must be approved by TA or instructor.
- Compatibility with EOS Linux environment is *required*.

2. Project Summary

In this project, you will implement a branch predictor simulator and use it to design and study various branch predictors. Different applications have different control flow behavior and your analysis will bring out the predictability of such behavior across a set of workload traces from the SPECint95 benchmark suite. The traces will be provided to you. Your simulator must estimate the branch prediction accuracies and other related metrics for the different workloads when run against different branch predictor designs and geometries. The rest of this document describes precisely what you will implement (Specification), how you will test your simulator (Validation runs), how will you communicate your findings (Report) and what all you will submit (Report, source code etc.).

3. Specification

3.1. Branch Predictors

3.1.1. Bimodal branch predictor

Model a bimodal branch predictor with parameter *i*. The bimodal branch predictor contains a single prediction table as shown in Figure 1. Each entry in the prediction table contains a 2-bit counter. The table is indexed by using the branch's PC's bits in the range *i*+1:2 (both inclusive), where *i* is the number of bits needed to index the predictor table. The lowest 2 bits are ignored because they are always 0s (since the assumed instruction is 4B in size the instruction address is always aligned to a 4B address). A single PC, therefore, has (at most) a single entry in the predictor table (subject to interference from other PCs that also map to the same entry). All entries in the prediction table should be initialized to **2** ("weakly taken") when the simulation begins.

As mentioned above, different branches may index the same entry in the prediction table. This is called *interference*. Interference is not explicitly detected or avoided; it just happens. There is no tag associated with each entry and therefore no tag checking and no miss signal for the prediction table!

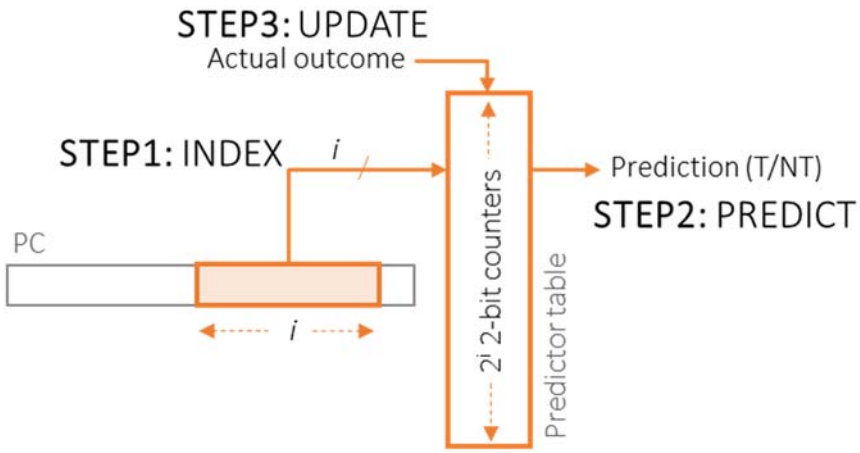

Figure 1: Bimodal Predictor

When you get a branch from the trace file, there are three steps to complete:

1. INDEX: Determine the branch's *index* into the prediction table.
2. PREDICT: Make a prediction. Predict *taken* if counter >= 2, else predict *not taken.*
3. UPDATE: Update the branch predictor based on the branch's *actual* outcome. Increment the counter if the branch was actually taken, else, decrement. The counter *saturates* at the extremes (0 and 3).

3.1.2. Gshare branch predictor

Model a *gshare* branch predictor with parameters {i, h} where *i* is the number of bits needed to index the prediction table and *h* is the number of bits in the global branch history register (GHR). Assume $h \leq i$. If *h* is 0 the gshare branch predictor reduces to the *bimodal* branch predictor.

When h > 0, there is an *h*-bit GHR in the simulated design. In this case, the index used to access the predictor table is based on both the branch's PC as well as the GHR, as shown in Figure 2. That is, unlike the *bimodal* predictor, a single PC may now have multiple entries in the predictor table depending on the T/NT history of the past *h* branches. In the extreme case, a single PC may use up 2^h entries in the predictor table, one per unique path history leading to the said PC.

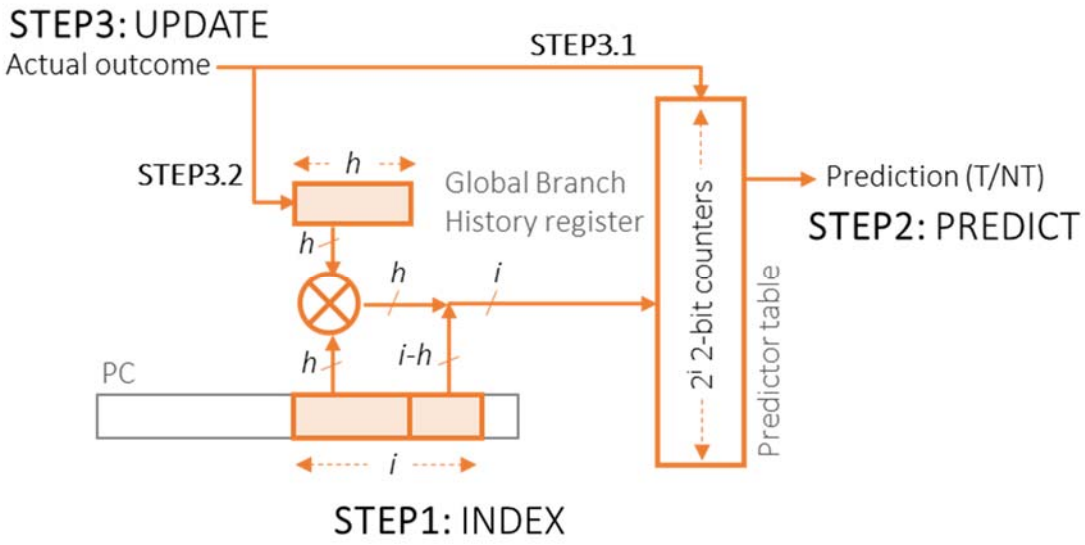

Figure 2: gshare predictor

At the beginning of simulation, the GHR is initialized to *all zeroes* and all entries in the prediction table should be initialized to **2** ("weakly taken").
When you get a branch from the trace file, there are three steps to complete:

1. INDEX: Determine the branch's *index* into the prediction table.
   Figure 2 shows how to generate the index. The current h-bit GHR is XORed with the uppermost h bits of the i bits being supplied from the PC for indexing.
2. PREDICT: Make a prediction. Predict *taken* if counter >= 2, else predict *not taken*.
3. UPDATE: Update the branch predictor and the GHR based on the branch's *actual* outcome.
   - Increment the predictor table's counter if the branch was actually taken, else decrement. The counter *saturates* at the extremes (0 and 3).
   - Update the GHR. Shift the register right by 1 bit position and place the branch's actual outcome (1 for actually taken, 0 for actually not-taken) into the most-significant (i.e. leftmost) bit position.

3.1.3. Hybrid branch predictor (ECE 521 only or ECE463 with extra credits)

Model a *hybrid predictor* that contains both the *bimodal* and *gshare* predictors, and uses a third table called the *chooser table* of 2^k 2-bit counters, in order to select which predictor's prediction to use. All counters in the chooser table are initialized to 1 at the beginning of simulation.
When you get a branch from the trace file, there are six top-level steps to be completed.

1. Obtain *two* predictions, one from the *bimodal* predictor (follow steps 1 and 2 in Section 3.1.1) and one from the *gshare* predictor (follow steps 1 and 2 in Section 3.1.2).
2. Use bits k+1 through 2 of the branch PC (discarding, as always, the lowest two bits of the PC) as the index into the *chooser* table.
3. Make an overall prediction. Get the indexed counter from the *chooser* table. If the counter is >= 2 use the *gshare* prediction, else use the *bimodal* prediction.
4. Update only the *selected* branch predictor based on the branch's actual outcome. If *bimodal* was selected follow step 3 in Section 3.1.1, else follow step 3 in Section 3.1.2.
5. Update *gshare's* global branch history register (step 4 in Section 3.1.2) *regardless* of the predictor chosen.
6. Update the branch's chooser counter. If the predictions from both the gshare and bimodal predictors match the actual outcome (both correct) or are both opposite of the actual outcome (both incorrect) leave the chooser table unchanged. The rationale is that there is no reason to prefer one predictor over the other. If gshare was correct and bimodal incorrect, then increment the chooser counter (saturate at 3). If gshare was incorrect and bimodal correct, then decrement the chooser counter (saturate at 0).

3.1.4. Yeh/Patt branch predictor (ECE 521/ECE463 with extra credits)

Model a *Yeh/Pat* branch predictor with parameters {h, p} where h is the number of bits needed to index the history table and *p* is the number of bits to index the pattern table. As shown in Figure 3, the history table is a set of p-bit local history (shifted) registers and the pattern table is a set of 2-bit saturating counters for each history entry.
At the beginning of simulation, all shifted registers in the history table are initialized to *all zeroes* and all entries in the prediction table should be initialized to **2** ("weakly taken").
When you get a branch from the trace file, there are three steps to complete:

1. INDEX: Determine the branch's *index* into the history table.
   Figure 3 shows how to generate the index – use h bits in the range of [h+1:2]from the PC for indexing.
2. INDEX: Determine the branch's *index* into the pattern table.

Figure 3 shows how to generate the index – use the p bits from the selected shifted register for indexing.

3. PREDICT: Make a prediction. Predict *taken* if counter >= 2, else predict *not taken*.
4. UPDATE: Update the branch predictor and the GHR based on the branch's *actual* outcome.
   - Update the local history register. Shift the register right by 1 bit position and place the branch's actual outcome (1 for actually taken, 0 for actually not-taken) into the most-significant (i.e. leftmost) bit position.
   - Increment the predictor table's counter if the branch was actually taken, else decrement. The counter *saturates* at the extremes (0 and 3).

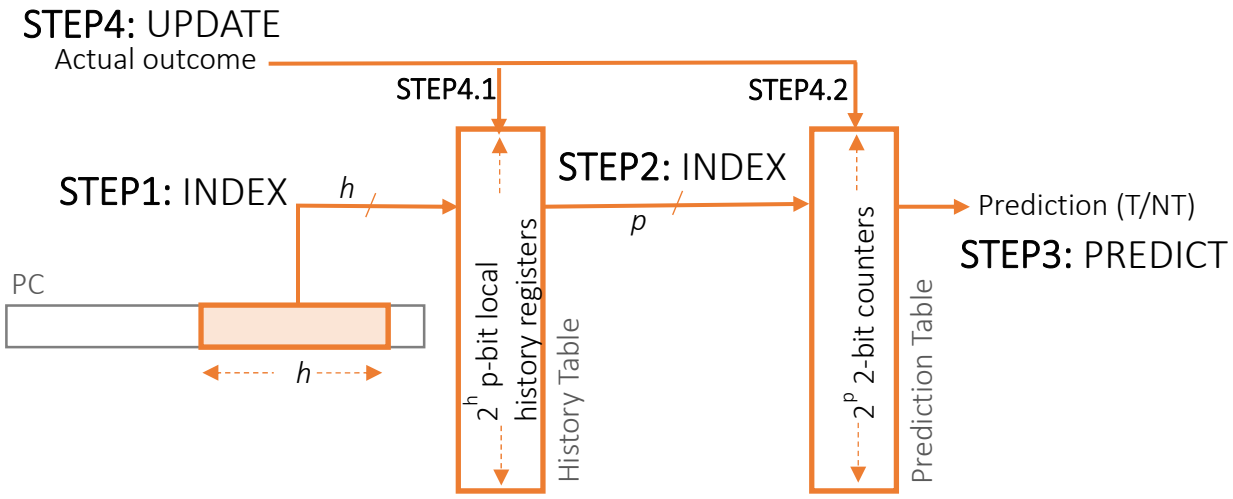

Figure 3: Yeh/Patt predictor

3.2. Branch Target Buffer (BTB)

Model a set-associative branch target buffer (BTB) as shown in Figure 4.
The BTB stores information about previously seen branches (their taken destination, whether the branch is or is not conditional etc.). In the simulator, we will use the BTB only to identify whether a request should be predicted using the branch predictor or not.
The rationale for using the BTB to filter access to the predictors is as follows. Recall that before you can access the branch predictor you need to be sure that you are dealing with a branch instruction. Recall also that the instructions fetched from the I Cache are, typically, not yet decoded at the time the branch predictor is accessed. Therefore, without a BTB, the machine is unsure about whether the instruction it fetched just now is a branch and therefore requires a prediction.
Further, note that you only need to model the tag array and not the data array since we are interested only in identifying whether or not we hit in the BTB and not in any target prediction.
When you get a branch from the trace file, the BTB is first checked for a hit. Again, ignore the lowest 2 bits of the PC since those are always 0s for an ISA with a 4-byte instruction size. Use the next higher i bits as index, and the rest of the higher order bits as tag bits (for simulation purposes it is fine to use the entire PC as the tag). The indexing results in 1 of 2 outcomes.

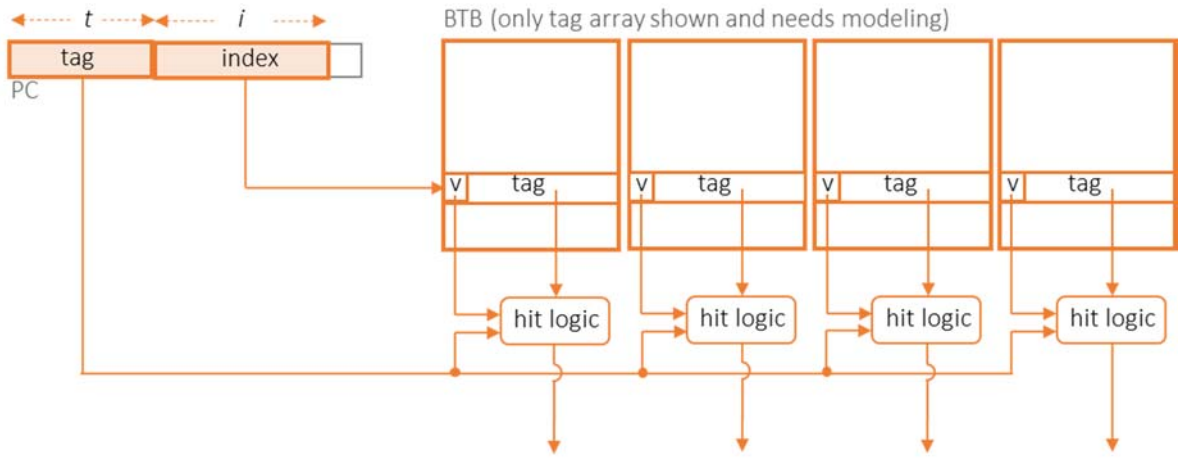

Figure 4: Branch Target Buffer

BTB miss: If there is no tag match in the indexed BTB set, the branch is implicitly predicted as being *not taken*. The rationale is that this is most likely not a branch at all therefore just continues to the next instruction. (In our traces all instructions are branches, but we will continue to use the same heuristic as above). Complete the following two steps.

1. Make space for the new PC. If there is at least one invalid block in the set, use that. Else use the LRU replacement policy to find a victim block V. No writeback is needed upon an eviction.
2. Add the PC's tag bits (or, simply the entire PC) into the BTB's selected tag array entry.

The implicit prediction was *not taken*; therefore, if the branch turns out to actually be taken, then the BTB is considered to have mispredicted.

BTB hit: In this case, nothing happens in the BTB. Predictions will be made from the branch predictor. Hint: You can, if you wish, reuse code from Project 1's cache model to model the BTB.

4. Simulator Input

The simulator reads a trace file in the following format:

```
<branch PC in hex> t|n
<branch PC in hex> t|n
...
```

The first field is the address of the branch instruction in memory. This field is used to index into predictors. The second field indicates whether the branch is *actually* taken or not. Here is a sample trace.

```
00a3b5fc t
00a3b604 t
00a3b60c n
```

5. Simulator Output

The simulator outputs the command line, the state of the relevant tables (predictor, chooser, GHR, BTB etc.) and finally the following metrics at the end. See validation runs for the exact format for each.

a	Number of branches
b	Number of predictions from the branch predictor
c	Number of mispredictions from the branch predictor
d	Number of mispredictions from the BTB
e	Branch misprediction rate ((item d + item c) / item a)

6. Compiling, running and validating the simulator

6.1. Compiling the simulator source

Part of what you need to hand in is source code. The TA will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see Section 10).

1. You must be able to compile and run your simulator on Eos Linux machines. This is required so that the TA can compile and run your simulator. If you are logging into an Eos machine remotely and do not know whether or not it is Linux (as opposed to SunOS), use the *uname* command to determine the operating system.
2. Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named *sim_bp*. The TA should be able to type only *make* and the simulator should compile successfully. The TA should be able to type only *make clean* to automatically remove object (.o) files and the simulator executable. An example Makefile will be posted on the web page; you can copy and modify it for your needs.

6.2. Running the simulator binary

Your simulator must accept command-line arguments as follows:

Bimodal predictor:

```
sim_bp bimodal <iB> <iBTB> <assocBTB> <tracefile>
```

<i_B> is the number of PC bits used to index the bimodal table, the rest are obvious.

<i_{BTB}> is the number of PC bits used to index the BTB.

<$assoc_{BTB}$> is the BTB associativity.

Setting either <i_{BTB}> or <$assoc_{BTB}$> to 0 implies that there is no BTB.

Gshare predictor:

```
sim_bp gshare <iG> <h> <iBTB> <assocBTB> <tracefile>
```

<i_G> and <h> are the number of PC and GHR bits used to index the gshare table, respectively. The other parameters are identical to the bimodal case.

Hydbrid predictor:

```
sim_bp hybrid <iC> <iG> <h> <iB> <iBTB> <assocBTB> <tracefile>
```

<i_C> is the number of PC bits used to index the chooser table. The other parameters are same as in the bimodal and gshare cases.

YehPatt predictor:

```
sim_bp yehpatt <h> <p> <iBTB> <assocBTB> <tracefile>
```

<h> and <p> are the number of PC and local history register bits used to index the history table and pattern table, respectively. The other parameters are identical to the bimodal case.

6.3. Validating the simulator output

Sample simulation outputs will be provided on the website. Each validation output provided includes:

- The simulator command line to use
- The final contents of the appropriate branch predictor tables
- All measurements described in Section 5

You must run your simulator and debug it until it matches these provided *validation outputs*. Your simulator must print outputs to the console (i.e., to the screen). Your output must match both numerically and in terms of formatting because the TA will literally "diff" your output with the correct output. Therefore, redirect the console output of your simulator to a temporary file. This can be achieved by placing

```
> <your_output_file>
```

after the simulator command. You can test whether or not your outputs match the expected output, by running this unix command:

```
diff –iw <your_output_file> <posted_output_file>
```

The –iw flag tells diff to ignore case (uppercase vs. lowercase) and whitespace. Therefore, just make sure there is some whitespace (such as a tab) where you see whitespace in the validation output. If the above command returns without printing anything to the screen, your validation was successful.

7. Experiments and Report

7.1. Exploring the Bimodal Predictor

Simulate Bimodal Predictor configurations for $7 \le i_B \le 12$. Use the traces in the trace directory. Make sure all validation runs pass. Then work on the following steps for your report.

Plot 1 Produce one plot per benchmark (Plot 1a, Plot 1b etc.) with
- plot title: <*benchmark*>, bimodal
- y-axis: branch misprediction rate
- x-axis: i_B

Each plot has one curve with 6 datapoints (one per i_B). Connect datapoints with a line.

Analysis Draw conclusions and discuss trends. Discuss similarities/differences across benchmarks.

Design For each trace, choose a bimodal predictor design that minimizes misprediction rate AND minimizes predictor cost in bits. Assume a maximum budget of **16 kilobytes** of storage if you wish to explore beyond the stipulated size of $i_B \le 12$. When "minimizing" misprediction rate, look for diminishing returns, *i.e.*, the point where misprediction rate starts leveling off and additional hardware provides only minor improvement. Bottom line: use good sense to provide high performance and reasonable cost. Justify your designs with supporting data and explanations.

7.2. Exploring the Gshare Predictor

Simulate Gshare predictor configurations for $7 \le i_G \le 12$, $2 \le h \le i_G$, h is even. Use the traces in the trace directory. Make sure all validation runs pass. Then work on the following steps for your report.

Plot 2 Produce one plot per benchmark with
- plot title: <*benchmark*>, gshare
- y-axis: branch misprediction rate
- x-axis: i_G

Each plot has 6 curves (one per h) and 27 datapoints in all. Connect datapoints with the same value of h with a line. The 6 curves will have 6, 6, 6, 5, 3 and 1 points respectively.

Analysis Draw conclusions and discuss trends. Discuss similarities/differences across benchmarks.

Design For each trace, choose a gshare predictor design that minimizes misprediction rate AND minimizes predictor cost in bits. Assume a maximum budget of **16 kilobytes** of storage if you wish to explore beyond the stipulated size of $i_G \le 12$. When "minimizing" misprediction rate, look for diminishing returns, *i.e.*, the point where misprediction rate starts leveling off and additional hardware provides only minor improvement. Bottom line: use good sense to provide high performance and reasonable cost. Justify your designs with supporting data and explanations.

7.3. Exploring the Branch Target Buffer

Simulate the BTB and ensure that the validation runs pass

7.4. Exploring the Hybrid Predictor (This is only applicable to ECE521 students or ECE463 students who want to gain extra credits)

Simulate the hybrid predictor and ensure that the validation runs pass

8. What to Submit via Wolfware

You must hand in a single zip file called **proj2.zip** that is no more than 1MB in size. Please respect the size limit on behalf of fellow students as the Wolfware submission space is limited. Notify the TA beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient. Below is an example showing how to create proj2.zip from an Eos Linux machine. Suppose you have a bunch of **source code files** (*.cc, *.h), the **Makefile**, and your project report (**report.pdf**). Run "zip proj2 *.cc *.h Makefile report.pdf"

proj2.zip must only contain the following (any deviation may result in point deductions):

- Project report: This must be a single PDF document named report.pdf (or a single MS Word document called report.doc). The report must include a cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A sample cover page will be posted on the project webpage. See Section 7 for the content required in the report.
- Source code: You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.
- Makefile: A sample Makefile will be posted on the project webpage. See Section 6.1 for what is expected of the makefile.

Note: Zip (no tar) only the *files* listed above, not the directory containing these files.

9. Grading

The following is the high level break-up of the points awarded.

- 20 points for substantial programming effort.
- 50 or 70 (ECE463) or 56 (ECE521) points for correct validation.
- 30 (ECE463) or 24 (ECE521) points for experiments and report.

Validation will be scored as follows:

Item		Points for ECE463	Points for ECE521
bimodal	val_bimodal_1.txt	5	4
	val_bimodal_2.txt	5	4
	val_bimodal_3.txt	5	4
	val_bimodal_4.txt	5	4
gshare	val_gshare_1.txt	5	4
	val_gshare_2.txt	5	4
	val_gshare_3.txt	5	4
	val_gshare_4.txt	5	4
BTB	val_BTB_1.txt	5	4
	val_BTB_2.txt	5	4
hybrid	val_hybrid_1.txt	Not applicable or 5	4
	val_hybrid_2.txt	Not applicable or 5	4
	Mystery A	Not applicable or 5	4
	Mystery B	Not applicable or 5	4
Yeh/Patt	val_yehpatt_1.txt	5	5
	val_yehpatt_2.txt	5	5
	Mystery C	5	5
	Mystery D	5	5

Experiments and reports will be scored as follows:

Item		Points for ECE463	Points for ECE521
Experiments and Report	Bimodal Plots	7	4
	Bimodal Analysis	4	4
	Bimodal Design	4	4
	gshare Graphs	7	4
	gshare Analysis	4	4
	ghsare Design	4	4

10. Deductions

- -10 point for each hour late, according to Wolfware timestamp. TIP: Submit whatever you have completed by the deadline just to be safe. If, after the deadline, you want to continue working on the project to get more features working and/or finish experiments, you may submit a second version of your project late. The TA will grade both the on-time and late versions (with late penalty applied to the late version), and use the one scoring more as the official submission. That said, please try and complete everything by the deadline.
- Up to -20 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties.
- Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and referral to the Office of Student Conduct.

11. Other Recommendations

11.1. Keep your simulator parametric

Always assume that parameters are changeable. As an example, you should not assume a specific BTB size or associativity, or GHR size etc. Such parameters should be parsed from the input of the simulator and then used to instantiate an instance of the generic class.

11.2. Keep backups

It is good practice to frequently make backups of all your project files, including source code, your report, etc. You can backup files to another hard drive (Eos account vs. home PC vs. laptop vs. USB memory stick etc.). Just keep consistent copies in multiple places.

11.3. Make your simulator fast

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important for a couple of reasons. First, the TA needs to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. If your simulator takes longer than 2 minutes to finish a single run, please see the TA. Second, you will be running many experiments: many branch predictor configurations and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast. One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag. Note that when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and compile with –g. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. When you are done with debugging, recompile with –O3 and without –g.

11.4. Use the VCL

In addition to using the grendel cluster, and other Eos linux machines, NCSU's Virtual Computing Lab. (VCL) allows you to reserve virtual linux machines. Go to http://vcl.ncsu.edu/

1. select "Reservation > New Reservation",
2. you'll see a popup asking for environment and duration,
3. for the environment, select "Linux Lab Machine (Realm RHEnterprise 6)"
4. you may choose to reserve a shell for up to 4 hours beginning now or later (with the possibility to extend reservation before it expires),
5. click "Create Reservation" button,
6. wait until the screen updates with a reservation and click "Connect!",
7. you will be provided an internet address that you can then ssh into using putty or other ssh clients (i.e. the same way that you remotely and securely login to other linux machines).