Project 4:
Value Prediction

Version 1.0

ECE 721: Advanced Microarchitecture
Spring 2024, Prof. Rotenberg

Due: Friday, April 19, 2024, 11:59pm
This is a hard deadline. All project deliverables are due at this time.

- READ THIS ENTIRE DOCUMENT.
- Academic integrity:
    - **Source code:** Each team must design and write their source code alone. They must do this (design and write their source code) without the assistance of any other person in ECE 721 or not in ECE 721. They must do this (design and write their source code) without searching the web for past semester's projects, computer architecture simulators or components of them, and the like, which is strictly forbidden. They must do this (design and write their source code) without looking at anyone else's source code, without obtaining electronic or printed copies of anyone else's source code, *etc*. **Exception: For Section 2.2 of this project (competition phase), and ONLY Section 2.2 (NOT Section 2.1), you may leverage value predictor code from the First Championship Value Prediction.**
    - **Explicit debugging:** With respect to "explicit debugging" as part of the coding process (*e.g.*, using a debugger, inspecting code for bugs, inserting prints in the code, iteratively applying fixes, *etc*.), each team must explicitly debug their code without the assistance of any other person in ECE 721 or not in ECE 721.
    - **Sanctions:** The sanctions for violating the academic integrity policy are (1) a score of 0 on the project and (2) academic integrity probation for a first-time offense or suspension for a subsequent offense (the latter sanctions are administered by the Office of Student Conduct). Note that, when it comes to academic integrity violations, both the giver and receiver of aid are responsible and both are sanctioned. Please see the following RAIV form which has links to various policies and procedures and gives a sense of how academic integrity violations are confronted, documented, and sanctioned: RAIV form.
- Reasonable assistance: If a team has any doubts or questions, or if a team is stumped by a bug, the team is encouraged to seek assistance using both of the following channels.
    - Teams may receive assistance from the TA(s) and instructor.
    - Teams are encouraged to post their doubts, questions, and obstacles, on the Moodle message board for this project. The instructor and TA(s) will moderate the message board to ensure that reasonable assistance is provided to the team. Other students are encouraged to contribute answers so long as no source code is posted.
      * An example of reasonable assistance via the message board: Student A: "*I'm encountering the following assertion/problem, has anyone else encountered something like this?*" Student B: "*Yes, I encountered something similar and you might want to take a look at how you are doing XYZ in your renamer, because the problem has to do with such-and-such.*"

* Another example of a reasonable exchange: Student A: "*I'm unsure how to size my Free List based on the other parameters.*" Instructor/TA/Student B: "*You can reference the lecture notes on this topic but I'll also answer here. The key is that the PRF has a specified number of physical registers and, at any given time, a fixed number of these are committed registers. The number of committed registers is the number of logical registers. The committed registers cannot be free, ever. From that, you should be able to infer an upper bound required for the size of the Free List. For example, if the PRF size is 160 and the # logical registers is 32, then at most there can be 128 free registers. Again, also refer back to the lecture notes.*"

- **The intent of the academic integrity policy is to ensure teams code and explicitly debug their code by themselves. It is NOT our intent to stifle robust, interesting, and insightful discussion and Q&A that is helpful for students (and the instructor and TA(s)) to learn together. We also would like to help teams get past bugs by offering advice on where they may be doing things incorrectly, or where they are making incorrect assumptions, *etc*., from an academic and conceptual standpoint.**

1. Project overview

1. Students will work in teams of two. Some students may already know their partners. For students who have not partnered with anyone yet, a google sheet will be shared with the class to facilitate finding partners. If there is an odd number of students in the class: for the one student who does not have a partner, the project scope will be adjusted in consultation with the instructor, unless the student wishes to keep the project scope the same. Exceptions to two-student teams will require approval of the instructor, a strong justification, and an appropriate plan for the project scope.
2. In this project, you will first implement a Stride Value Predictor (SVP) and support for value prediction in 721sim and validate it against the TAs' simulator. Then, you will improvise on the value predictor, its parameters, and value prediction support, to compete against other teams for highest harmonic mean IPC across all benchmarks available in ECE 721.

2. Tasks and grading rubric

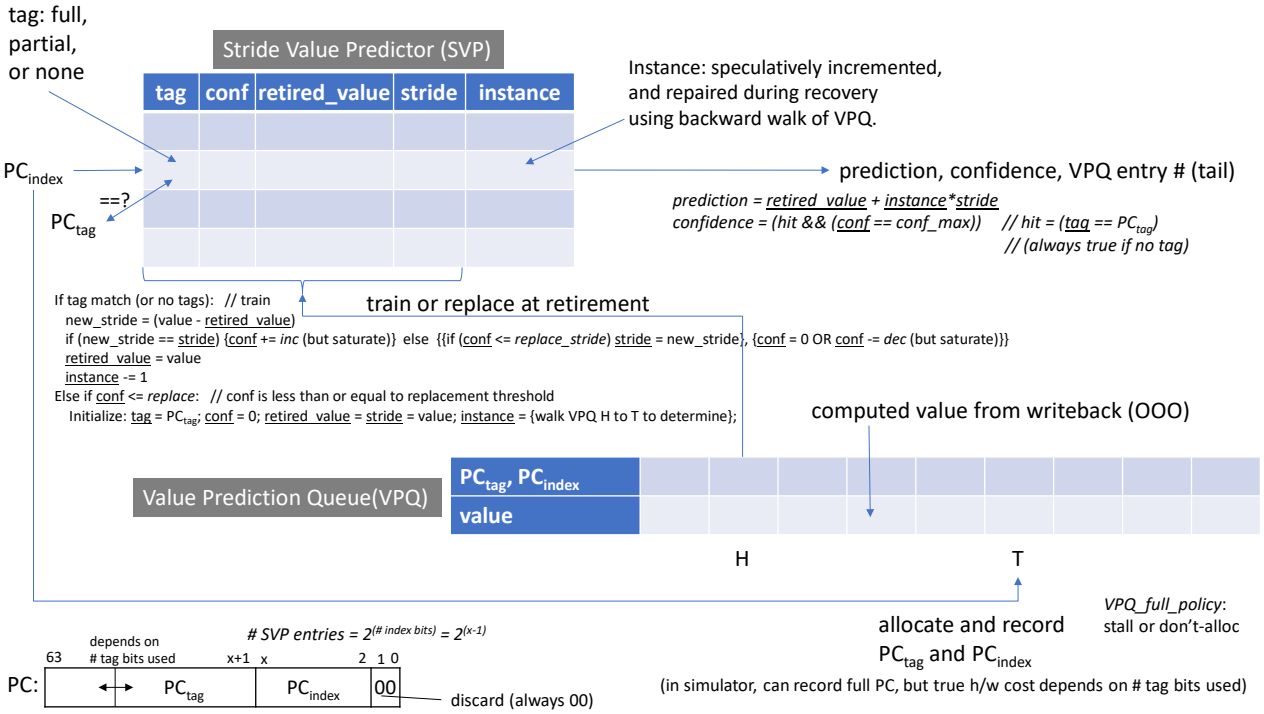
2.1. [80 points] Stride Value Predictor (SVP) with Value Prediction Queue (VPQ)

In the first phase of this project, you will implement the following:

1. Implement support for injecting confident value predictions in the Dispatch stage to break data dependencies between producers and consumers, following from the lecture notes on this aspect.
2. Implement a perfect value prediction mode, leveraging existing 721sim functionality that links each correct-path instruction in the pipeline simulator to its counterpart in the checker simulator (links are created in the Fetch stage, so they can be referenced in the Dispatch stage where value predictions are injected as per task 1 above). The counterpart's value is available in advance and can be used as a perfect value prediction. Perfect value prediction mode is useful for evaluating the speedup potential of value prediction, and for confirming the breaking of data dependencies implemented in step 1 above without needing support for value misprediction detection and recovery.

TIP: You can implement tasks 1 and 2 above to test breaking of data dependencies, without the need to implement value misprediction detection and recovery. After testing this aspect, you can move onto real value prediction, value misprediction detection, and recovery.

3. Implement a Stride Value Predictor (SVP) with Value Prediction Queue (VPQ), following from the lecture notes on this topic and depicted in the figure below.



4. Implement an oracle confidence mode. Like perfect value prediction mode, you will leverage the link between each correct-path instruction in the pipeline to its counterpart in the checker simulator. But instead of using the counterpart's value as a perfect value prediction, you will use it to check correctness of the real value prediction from your SVP, in advance. In oracle confidence mode, the SVP's confidence estimation is discarded and the real value prediction is only deemed confident if it matches the counterpart's value.

TIP: You can implement tasks 3 and 4 above to test breaking of data dependencies with a real value predictor (SVP), without the need to implement value misprediction detection and recovery (in oracle confidence mode, all incorrect value predictions are discarded and all correct value predictions are used). After testing that the SVP appears to be delivering some correct value predictions in oracle confidence mode, you can move on to the next step and run with real confidence. Ultimately, once everything is working, the utility of oracle confidence mode is in gauging the performance impact of real confidence.

5. Implement support for checking value predictions in the Writeback stage and posting value mispredictions in the Active List as they are detected (only if the instruction was actually value predicted), following from the lecture notes on this aspect. 721sim *already* has BR-1-like recovery support for value mispredictions: wait until posted value misprediction reaches the head of Active List, retire the value-mispredicted producer instruction, and full-squash all instructions in the pipeline.
6. Implement new command-line arguments to configure value prediction.
   `--vp-enable=x`: "x" should be 0 (disable value prediction) or 1 (enable value prediction). All other value prediction related parameters are treated as don't-cares when value prediction is disabled.
   If value prediction is enabled (**`--vp-enable=1`**), specify *one and only one* of N possible value predictors: perfect value predictor, SVP, and other value predictors you explore in the

competition:

```
// Choose the perfect value predictor.
--vp-perf=1

// Choose and configure the Stride Value Predictor (SVP).
--vp-svp=<VPQsize>,      // number of VPQ entries
         <oracleconf>,   // 0: real confidence, 1: oracle confidence
         <# index bits>, // # SVP entries is 2**(# index bits)
         <# tag bits>,   // 0: no tag, 62-(# index bits): full tag, or
                         // somewhere between for partial tag
         <confmax>,      // confidence threshold to be confident
         <confinc>,      // amount by which to increment confidence
                         // (but saturate at confmax)
         <confdec>,      // 0: reset confidence, >0: amount by which to
                         // decrement confidence (but saturate at 0)
         <replace_stride>,// If stride changes, only update the stride if
                         // conf <= replace_stride.  This can be used for
                         // stride inertia.
         <replace>,      // Replacement confidence threshold.  Only used if
                         // SVP has tags and when the tag mismatches while
                         // training: replace if conf <= replace.
         <predINTALU>,   // 1: predict integer ALU instructions with dest.
                         // reg., 0: don't predict this instruction class
         <predFPALU>,    // 1: predict flt.pt. ALU instructions with dest.
                         // reg., 0: don't predict this instruction class
         <predLOAD>,     // 1: predict load instructions with dest. reg.,
                         // 0: don't predict this instruction class
         <VPQ_full_policy>// If VPQ full and need entries for VP-eligible
                          // instructions in bundle: 0: stall bundle,
                          // 1: don't allocate VPQ entries (gaps in
                          // training will affect confidence).
```

NOTE: Do NOT value-predict branches with destination registers (call, call indirect). The current value misprediction recovery logic is hardwired to rollback the fetch unit to the next sequential instruction. This recovery PC is incorrect for call, call indirect. It's fixable, but let's not bother for now. In any case, the value written by call, call indirect is the return target, which is the purview of branch prediction (RAS).

```
// Choose and configure other predictors during competition phase.
--vp-???=?,?,…
--vp-???=?,?,…
```

7. Your simulator must accurately tally and output the number of bytes for the value predictor, for whichever value predictor is selected for a given run, based on its configuration. Moreover, the total bytes must be broken down by various structures: value prediction table(s), structures like value prediction queue(s) for VP training and rolling back speculatively-updated VP context, value prediction context, *etc*. For example, for the value predictor implemented in the first phase of this project, the simulator must output total bytes of storage and also break it down into costs for the SVP and VPQ. The validation runs will show the format of this output for the latter value predictor. When you get to the competition phase, accurate cost accounting is essential because you must stay within a given storage budget.

8. Your simulator must breakdown retired instructions into five categories for insight: (1) "ineligible for value prediction" (any instruction without a destination register, all types of branches, and any otherwise eligible instruction class not configured for value prediction), (2) "correct and confident" (possible IPC benefit), (3) "correct and unconfident" (lost opportunity), (4) "incorrect and confident" (value mispredictions), and (5) "incorrect and unconfident". Correct/incorrect is an indication of whether the value prediction was correct or incorrect, regardless of whether or not it was used (*i.e.*, regardless of confidence). Confident/unconfident is an indication of confidence, hence, whether or not the value prediction was used. The format of this output will be evident in the validation runs.

PROF.+TA TODO #1: Validation runs will be posted under the Project 4 topic in Moodle. You will self-grade your simulator using Gradescope, which will automatically compile and run the simulator, check for completion of grading plateaus as was done in prior projects, check IPC as was done in prior projects, and check the new outputs (storage cost accounting and retired instruction breakdown into the five categories). All team partners should submit the team's simulator to Gradescope so that everyone receives a score.

Here is how Gradescope will score your simulator:

BASE:
If you pass at least one validation run with SVP and real confidence, ***BASE*** = 30. Otherwise, ***BASE*** = 0 - 30 based on manual inspection of the code (partial credit for key aspects implemented).

N: total number of validation runs.

validation_run_score$_i$:
Score for each validation run:

- 8: Simulator runs for > 1,000 instructions but < 10,000 instructions (faults, asserts, or deadlocks before 10,000 instructions).
- 15: Simulator runs for > 10,000 instructions but < 100,000 instructions (faults, asserts, or deadlocks before 100,000 instructions).
- 22: Simulator runs for > 100,000 instructions but < 1,000,000 instructions (faults, asserts, or deadlocks before 1,000,000 instructions).
- 29: Simulator runs for > 1,000,000 instructions but does not complete (faults, asserts, or deadlocks before completion).
- 36: Simulator runs to completion, but IPC differs from instructor's version by > 1%.
- 43: Simulator runs to completion and IPC matches within 1%, but one or more value prediction breakdown measurements differs by > 1% and/or storage cost accounting doesn't match exactly.
- 50: Simulator runs to completion, IPC matches within 1%, all value prediction breakdown measurements match within 1%, and storage cost accounting matches exactly.

The score *for this first phase* is calculated as follows (maximum of 80 points):

$$project\ score = BASE + \frac{\sum_{i=1}^{N} validation\_run\_score_i}{N}$$

2.2. [20 points] Competition

You must complete the first phase of the project (Section 2.1) before you are eligible for the competition phase. If your team enters the competition, you will use your value prediction simulator to see how far you can push performance within a fixed storage budget for value prediction.

2.2.1. Example strategies

You are free to explore the following strategies and possibly others. **These are just suggestions and ultimately you decide how to budget your time in the competition.**

- Perhaps explore the design space of your SVP+VPQ within the fixed storage budget. For example, you can tradeoff the number of bits per SVP entry (*e.g.*, full, partial, or no tags) with more or fewer SVP entries; you can experiment with different policies and their parameters (replacement threshold, confidence counter parameters, stride inertia, *etc*.); you can experiment with which instruction classes to predict (some instructions may be more critical to performance than others, and the more eligible instructions the more competition for SVP entries and the larger the VPQ may need to be); *etc*.
- Perhaps improvise on your SVP, for example: add some global branch history context to specialize strides and attempt to address the problem of skipped instances caused by control flow; I believe there are prior works that maintain multiple strides per PC and select among them; *etc*. It may be wise to characterize value patterns before carrying out time-consuming improvisations. It may be wise to first try CVP-1 predictors or other predictors from the literature since their authors carried out studies (see below).
- **Perhaps try out one or more value predictors from the First Championship Value Prediction (CVP-1).** Contestants of CVP-1 provided their source code. Keep in mind that these predictors were integrated into a specific trace-driven simulator and will need to be ported to 721sim. In particular, CVP-1's value predictor API is peculiar in that contestants enjoyed the illusion of speculatively updating value predictor context as needed, without the burden of repairing context after rollbacks (owing to the trace-driven nature of the CVP-1 simulator). In other words, I doubt they model machinery akin to the VPQ. Thus, some 721sim integration effort is required. Moreover, there is no guarantee the CVP-1 predictors will perform similarly in the context of 721sim and our SimPoints. Once deployed, experiment with relevant parameters and policies within the fixed storage budget.
- Perhaps implement CVP-1 value predictors from scratch rather than use their source code as a starting point, for more control and a cleaner 721sim-compatible implementation. Once deployed, experiment with relevant parameters and policies within the fixed storage budget.
- Perhaps research references in the CVP-1 bibliography and other value predictor papers, and explore one or more of these value predictors. If and when you deploy these value predictors, experiment with relevant parameters and policies within the fixed storage budget.
- Perhaps explore your own ideas if you uncover insights from ECE 721 SimPoints.
- **Perhaps implement immediate value misprediction recovery, using BR-5 instead of BR-1.** There is interplay between recovery performance and aggressiveness of value prediction. If the misprediction penalty is reduced, it might be possible to decrease the confidence threshold → more aggressive value prediction → increase value prediction coverage ("confident and correct" up) at the price of more mispredictions ("confident and incorrect" up). For fair play in the competition, existing BR-5 resources must remain fixed. That is, GBM bits / branch IDs / checkpoints must be shared among branches and value-predicted

instructions. Improvising further, you can explore using dual confidence levels: high and medium confidence. High-confident value-predicted instructions don't consume BR-5 resources and use BR-1 recovery if they are mispredicted, whereas medium-confident value-predicted instructions consume BR-5 resources. You can also play with different policies when BR-5 resources are unavailable for value-predicted instructions in the rename bundle: stall rename vs. class it for BR-1 recovery vs. don't value predict it.

2.2.2. Key competition rules

- There will be one baseline superscalar processor configuration. PROF.+TA TODO #2: This configuration will be announced when it has been chosen by the instructor and TAs. The choice of baseline superscalar processor is important because of the interplay between window size, superscalar width, and value prediction.
- There will be a single fixed storage budget for the value predictor including all its subcomponents. PROF.+TA TODO #3: The fixed storage budget is to-be-determined and will be announced later. You must painstakingly account for and document all bytes of storage (prediction tables, any substantial indexing context, value prediction queue(s), *etc*.). Your simulator must print out storage cost accounting (total, and broken down among key components, as you did for SVP+VPQ).
- A given team will submit a "final entry" for evaluation by the TAs. The final entry is comprised of (1) the team's simulator, (2) flags that select the final value predictor and its final configuration, (3) a spreadsheet of final SimPoint IPCs[1], final benchmark IPCs, and final overall IPC (calculations of the latter two are explained below), for corroboration with the TAs' runs with the final entry, and (4) final storage cost (total, and broken down among key components), for corroboration with the TAs' examination of storage cost and to check that the final entry is within budget. It is my expectation that you have one configurable simulator that has the SVP+VPQ from Section 2.1 and at least the final value predictor from Section 2.2 (if not all value predictors explored in Section 2.2).
- You are prohibited from *a priori* configuring your simulator based on the benchmark or benchmark SimPoint that is being run. That is, the modeled superscalar processor (and its value predictor, value prediction support, *etc*.) does not have *a priori* knowledge of the benchmark being run nor *a priori* knowledge of that benchmark's preferred value predictor parameters. If you want to dynamically adapt the value predictor's parameters to program behavior, you need to invent hardware to achieve such adaptivity and include its storage cost in the fixed storage budget. Moreover, such adaptivity must be limited to policies, thresholds, *etc*.; on the other hand, it is unfair to assume storage can be shifted among different subcomponents (assumption of reconfigurable storage is prohibited in this competition). Hybrid value predictors are acceptable because (1) they are intrinsically adaptive, (2) all subcomponents are accounted for in the fixed storage budget, and (3) storage is not shifted among subcomponents (each subcomponent is fixed).
- Recovery is limited to squash-based recovery, either BR-1 (721sim already implements BR-1 for value misprediction recovery), BR-5, or hybrids. Complete or selective re-execution is prohibited for two reasons: (1) it takes a long time to implement re-execution based recovery,

[1] If your simulator fails to complete a given SimPoint (faults, asserts, or deadlocks before completion), then substitute its baseline IPC for your IPC (*i.e.*, no speedup on the failed SimPoint).

plus it is challenging to get right (particularly selective re-execution), so I advise against it, (2) some implementations require more storage throughout the pipeline (dependence vectors, replay buffer, *etc*.), and (3) aside from the BR-1 vs. BR-5 aspect, the focus of the competition is the value predictor.

- PROF.+TA TODO #4: *Other rules as needed based on how the competition unfolds.*

2.2.3. Competition scoring

The competition points are divided into two parts:

- IPC rank [0-10 points]: A given team will be awarded 0-10 points based on their final entry's performance. Performance is calculated in two steps: (1) a given benchmark's IPC is calculated as the weighted harmonic mean of its SimPoint IPCs; (2) overall IPC is calculated as the harmonic mean of IPCs among all benchmarks. The mapping of percentile rank (percentage of contestants below you) and/or overall IPC, to points, is yet to be determined, and may be linear or non-linear, continuous or discrete, and so forth.
- Recorded presentation [0-10 points]: Every competing team must create a powerpoint presentation that describes additional value predictors that were attempted, their total and per-subcomponent storage costs, their design space aspects that were explored, *etc.*, and the results and insights of these efforts. The team must record a 10-minute presentation (*e.g.*, you can use Zoom to record). If you can't fit everything you explored, prioritize your final entry and other competitive design points. The instructor and TAs will evaluate the recorded presentation and assign 0-10 points based on the quality of the team's research. Recordings that exceed the 10-minute time limit will not be considered (0 points).

2.2.4. HPC

PROF.+TA TODO #5: The TAs and instructor are working on setting up a project in HPC, NCSU's batch computing facility, to facilitate the huge number of jobs that teams will need to submit during the competition. Status: An HPC project already exists from a past semester. Old users need to be removed and new users from the current class added. Benchmark checkpoints need to be installed in storage accessible by the HPC project. Finally, the class will be provided with scripts for submitting all jobs with one command.