Assignment — Evaluating Branch Prediction Schemes with an Out-of-Order Core in gem5

Goal: implement/compare several branch prediction schemes in gem5, run each with an out-of-order core on representative workloads, and analyze the effect of predictor design on IPC, misprediction rates, and overall processor performance.

---

1) Setup & baseline

1. Build gem5 with the out-of-order CPU model (DerivO3CPU / O3CPU / O3-like CPU).
    - Example configure/build:
        - scons build/X86/gem5.opt -j<nprocs> (adapt to ISA)
2. Choose workloads:
    - PARSEC / MiBench / simple compiled programs (e.g., gcc -O2 compile of small kernels, perl, fft, matrix multiply, bitmap, branch-heavy microbenchmarks)
        - Parsec Tutorial
        - MiBench Tutorial
    - Include at least one **branch-heavy** and one **compute-heavy** workload.
3. Baseline run: run gem5 with the default branch predictor and OOO core to capture baseline stats.

---

2) Branch predictors to evaluate

Implement or enable and test the following predictors (at minimum):

- **Bimodal (one-level) predictor** (simple table, 2-bit saturating counters).
- **Gshare** (global history XOR with PC indexing).
- **Local (per-PC) predictor** (local history table + local counters).
- **Tournament / hybrid** predictor (combines global and local with selector).
- **Perceptron predictor** (if available/feasible) — optional but high value.

If gem5 already provides some predictors (e.g., BiModeBP, GShareBP, TournamentBP, LocalBP, PerceptronBP), configure them. Otherwise implement missing ones in src/cpu/pred/ (or equivalent in the version you use).

Implementation notes:

- Predictor API: implement predict(), update() with branch PC, outcome, target.
- Keep parameters tunable: table size, counter bits, history length, perceptron weights length.
- Ensure predictor state can be instantiated from a command-line flag in gem5 (e.g., --bp-type=GShareBP).

---

3) OOO core configuration

- Use gem5's out-of-order pipeline (DerivO3CPU / O3CPU) with realistic ROB, IQ sizes. Example settings to vary:
    - ROB size: 128, 256
    - IQ entries: 64, 128
    - Fetch width / issue width: 4
    - Branch recovery penalty: use gem5 defaults, but record branchPredicted, branchMispredicted, and pipeline stalls.

Run all predictors on the **same** OOO configuration to isolate predictor effects.

---

4) Experiment methodology

- Fast-forward warmup: run with --cmd/checkpoints or run short input then enable stats collection for a "region of interest" (ROI).
- For each workload & predictor:
    1. Fast-forward to ROI (or use checkpoint).
    2. Run for a fixed number of committed instructions (e.g., 100M instructions) or simulate for a fixed simulated time.
    3. Collect gem5 stats (stats.txt).
- Repeat each configuration 3 times to check variability (optional but recommended).
- Keep all other microarchitectural parameters fixed across predictor runs.

---

5) Stats & metrics to collect

From gem5 stats.txt (or m5.stats), extract:

Primary metrics

- sim_seconds (sim time)
- system.cpu.<cpu>.committed_instructions (or instrs_committed)
- **IPC** = committed_instructions / sim_seconds or use gem5's IPC stat
- branchPredicted and branchMispredicted (or branchPredicted.??? depending on gem5 version)
- **Branch misprediction rate** = branchMispredicted / branchPredicted
- pipeline stalls (if available), fetch bubbles, squash count

Additional useful metrics

- l1d_accesses, l1i_accesses (see secondary effects)
- ROB occupancy, IQ utilization (if accessible)
- branchPred.MispredRecoveryCycles or stats for misprediction penalty (if present)
- Runtime breakdown (execute vs stall cycles)

---

6) Analysis tasks

1. For each workload and predictor, produce a table summarizing:
    - Predictor name & configuration (history length, table entries, counter bits)
    - IPC
    - Branch prediction rate (accuracy and misprediction rate)
    - Average branch recovery penalty (or observed average stall cycles per misprediction)
2. Plot:
    - Bar chart of IPC for each predictor (per workload)
    - Line chart of misprediction rate vs predictor complexity (e.g., history length)
    - Scatter plot: misprediction rate vs IPC
3. Explain:
    - Which predictor performed best overall and why.
    - Where perceptron (if used) helps relative to tournament/gshare.
    - How predictor improvements translate (or not) into IPC improvements (discuss pipeline bottlenecks).
4. Statistical significance: comment on variability across runs.

---

7) Suggested command-line runs (example)

(adapt path/flags for your gem5 version and ISA; these are template-style)

```
# Example: run DerivO3CPU with GShare predictor
build/X86/gem5.opt configs/example/se.py \
  --cpu-type=DerivO3CPU \
  --caches --l2cache \
  --bp-type=GShareBP \
  --gshare-history=12 \
  --gshare-table-size=8192 \
  --cmd=/path/to/workload \
  --options="input args" \
  --maxinsts=100000000 \
  > out_gshare.log 2>&1
```

For Bimodal:

```
--bp-type=BiModeBP --bimodal-table-size=4096 --bimodal-counter-bits=2
```

For Tournament:

```
--bp-type=TournamentBP --global-history=12 --local-history=10 --chooser-size=4096
```

Perceptron (if available):

--bp-type=PerceptronBP --perceptron-h=32 --perceptron-table=4096

(If your gem5 doesn't accept these flags, extend the config script to accept them, or edit configs/ to create presets.)

---

8) Deliverables

1. **Code**: predictor implementations and config scripts (clean, commented).
2. **Run scripts**: shell/python scripts that reproduce the experiments.
3. **Data**: raw stats.txt for each run and a CSV summary.
4. **Report (6–10 pages)**:
   - Background & hypothesis
   - Experiment methodology (including warmup and ROI)
   - Results (tables + figures)
   - Analysis and discussion (why results look the way they do)
   - Conclusion and limitations
5. **Short presentation** (<=7 slides) summarizing key findings. We will use this for the viva.

6. This can be done in groups of two --- the report must clearly identify the exact contributions of the partners.
7. We will check for variance in the data that is generated by different groups. Please indicate the machine configuration of the machine on which the data is generated --- either from /proc or equivalent windows/mac places.

---

9) Grading rubric

- Correctness of implementations (25%) — predictors implemented or configured correctly.
- Experiment design & reproducibility (20%) — reproducible scripts, consistent methodology.
- Quality of analysis (25%) — clarity of plots, correct interpretation of results, insight into why predictors perform as they do.
- Report & code quality (15%) — readability, documentation, adherence to deliverable requirements.
- Viva (15)