Project #5

(Report due is 11:59 PM, 12/8/2022)

Project description:
In project 2, you have already explored different branch predictor (BP) designs. In this project, you will try to implement various BPs in Gem5 by yourselves. Specifically, you will firstly implement a simple BP in Gem5 following the tutorial. Then, you are required to read a paper which firstly described the perceptron BP. You need to summarize the paper and implement the perceptron BP in Gem5. Next, you are required to compare your BP's performance with other BPs that Gem5 has provided. Whenever you face the trouble or problem on this project, please refer to Gem5 document here or contact TA Yuda An. If you are facing trouble with the provided server, please contact TA Shushu Yi. All the following information have been verified on Ubuntu 20.04.1 with Linux Kernel 5.15.0.

Submission:
- Please submit (in a report form, both PDF or Word are acceptable) the deliverables for each part in order and clearly defined. Please give a brief description of the results in your report and DO NOT SUBMIT ANY SCRIPTS.
- Please send your report as an attachment at ca2022fall@163.com. The titles of your email AND attachment should both be Student ID_Name_Proj5 (e.g., 123456789_XX_Proj5).

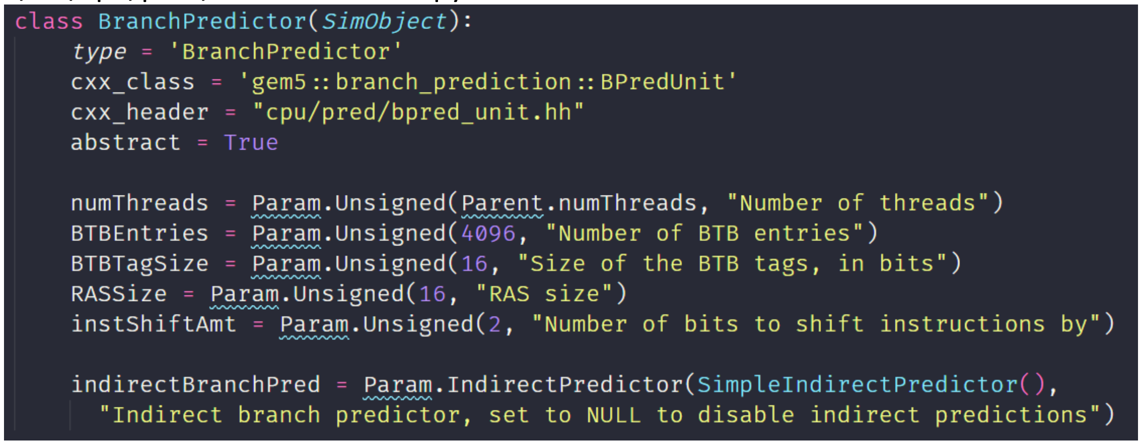
Late policy:
- You will be given 3 slip days (shared by all projects), which can be used to extend project deadlines, e.g., 1 project extended by 3 days or 2 projects each extended by 1 days.
- Projects are due at 23:59:59, no exceptions; 20% off per day late, 1 second late = 1 hour late = 1 day late.

Part 1: Implement a simple BP in Gem5

A prediction result of a BP can be 'taken', which means the branch instruction should take the branch and jump to the branch target, or 'not taken', which means the branch instruction should not branch. In this part, we will build a simple branch predictor, which always predicts 'taken'. We will leverage the cache hierarchy described in Project 3 Part 1 (with a 32kB L1D cache, a 32kB L1I cache and a 256kB unified L2 cache).
To start the project, make a folder for this project and copy the configuration scripts from Project 3:

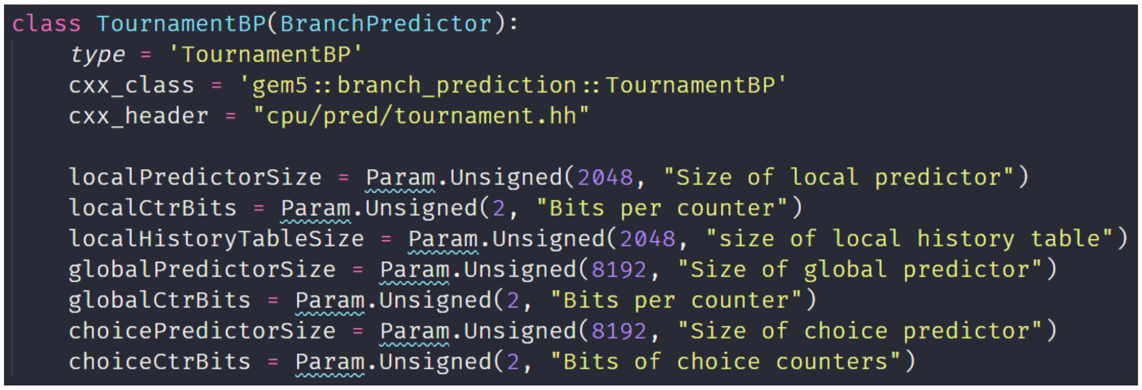
```
mkdir configs/proj5

cp configs/proj3/two-level.py configs/proj5/two-level.py

cp configs/proj3/new_cache.py configs/proj5/new_cache.py
```

Now we are going to implement the simple BP. Gem5 has provided many types of BP. The source codes can be find under gem5/src/cpu/pred/. A Python abstract class *BranchPredictor* is provided to describe basic BP structure. It can be found in gem5/src/cpu/pred/BranchPredictor.py:



```
class BranchPredictor(SimObject):
    type = 'BranchPredictor'
    cxx_class = 'gem5::branch_prediction::BPredUnit'
    cxx_header = "cpu/pred/bpred_unit.hh"
    abstract = True

    numThreads = Param.Unsigned(Parent.numThreads, "Number of threads")
    BTBEntries = Param.Unsigned(4096, "Number of BTB entries")
    BTBTagSize = Param.Unsigned(16, "Size of the BTB tags, in bits")
    RASSize = Param.Unsigned(16, "RAS size")
    instShiftAmt = Param.Unsigned(2, "Number of bits to shift instructions by")

    indirectBranchPred = Param.IndirectPredictor(SimpleIndirectPredictor(),
      "Indirect branch predictor, set to NULL to disable indirect predictions")
```

It provides some basic parameters, and we can see that all BPs are integrated with a BTB, a RAS and an indirect branch predictor.
In BranchPredictor.py, Gem5 provides multiple designs of BP, including the tournament BP which is used as the default BP. We can check the defination of it:

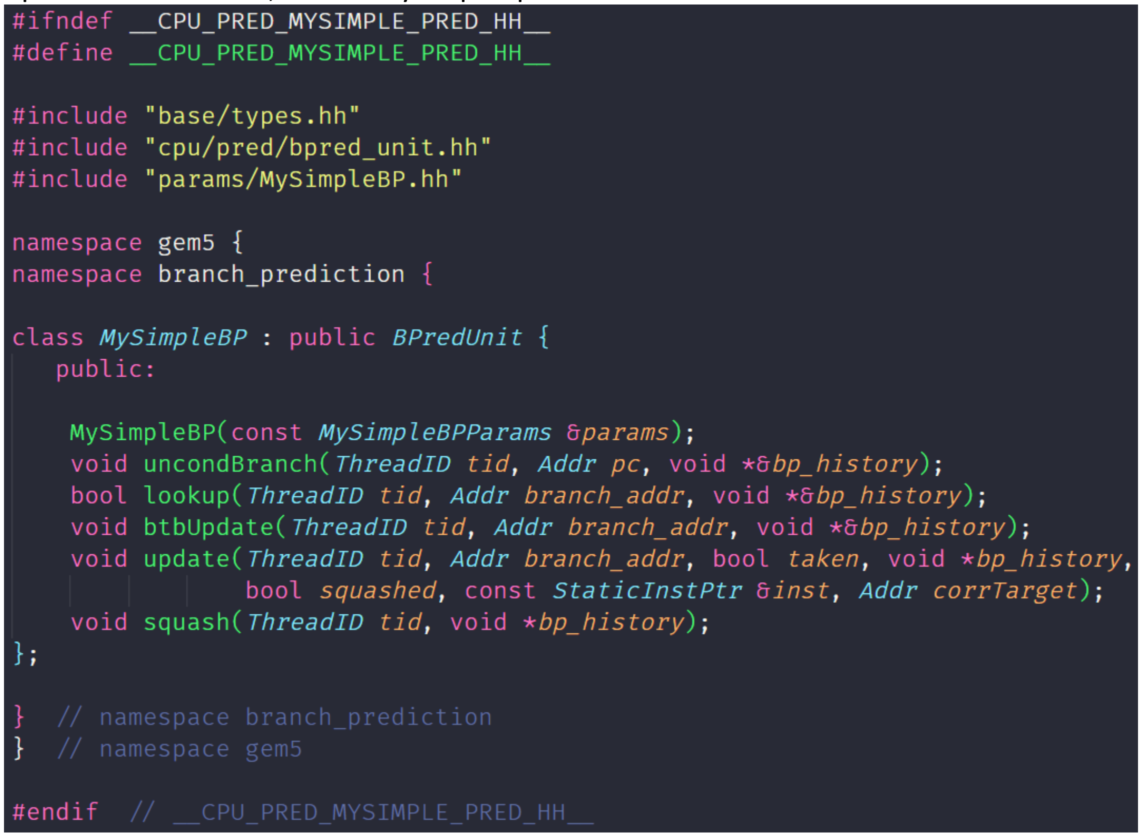


```
class TournamentBP(BranchPredictor):
    type = 'TournamentBP'
    cxx_class = 'gem5::branch_prediction::TournamentBP'
    cxx_header = "cpu/pred/tournament.hh"

    localPredictorSize = Param.Unsigned(2048, "Size of local predictor")
    localCtrBits = Param.Unsigned(2, "Bits per counter")
    localHistoryTableSize = Param.Unsigned(2048, "size of local history table")
    globalPredictorSize = Param.Unsigned(8192, "Size of global predictor")
    globalCtrBits = Param.Unsigned(2, "Bits per counter")
    choicePredictorSize = Param.Unsigned(8192, "Size of choice predictor")
    choiceCtrBits = Param.Unsigned(2, "Bits of choice counters")
```

We will work under src/cpu/pred/ when implementing the simple BP. Let’s start by adding a new Python class to BranchPredictor.py, which inherits the parameters from *BranchPredictor*:

```
class MySimpleBP(BranchPredictor):

    type = 'MySimpleBP'

    cxx_class = 'gem5::branch_prediction::MySimpleBP'

    cxx_header = "cpu/pred/mysimplebp.hh"
```

Next, we need to create mysimplebp.hh and mysimplebp.cc which contain the codes of simple BP implementation. First, create mysimplebp.hh:

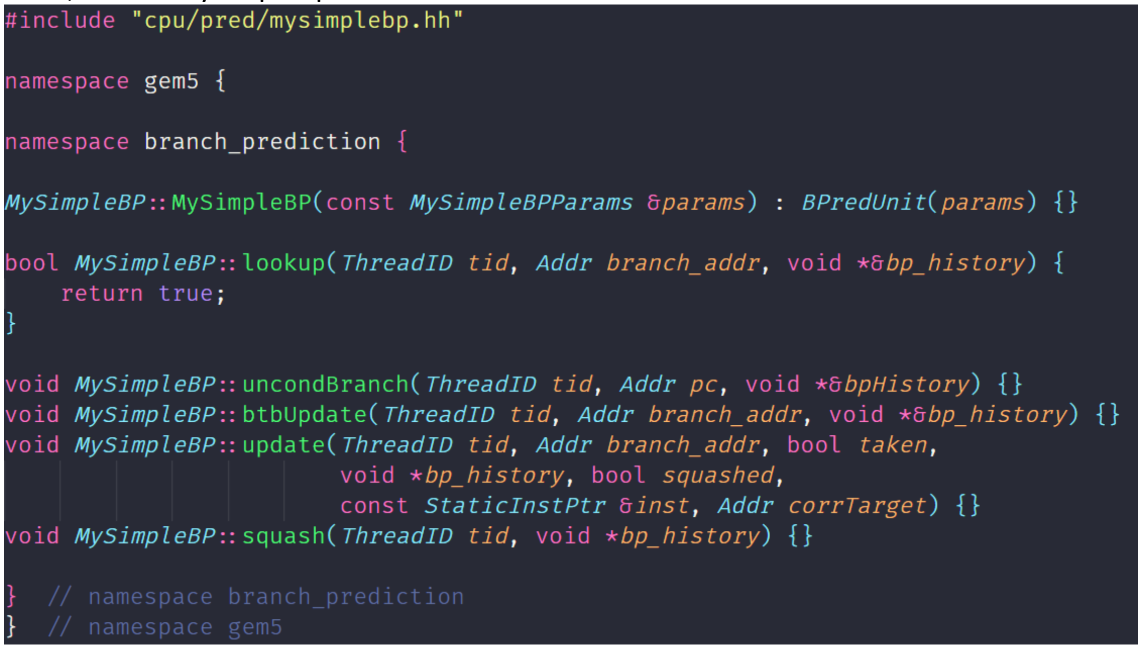


```
#ifndef __CPU_PRED_MYSIMPLE_PRED_HH__
#define __CPU_PRED_MYSIMPLE_PRED_HH__

#include "base/types.hh"
#include "cpu/pred/bpred_unit.hh"
#include "params/MySimpleBP.hh"

namespace gem5 {
namespace branch_prediction {

class MySimpleBP : public BPredUnit {
   public:

    MySimpleBP(const MySimpleBPParams &params);
    void uncondBranch(ThreadID tid, Addr pc, void *&bp_history);
    bool lookup(ThreadID tid, Addr branch_addr, void *&bp_history);
    void btbUpdate(ThreadID tid, Addr branch_addr, void *&bp_history);
    void update(ThreadID tid, Addr branch_addr, bool taken, void *bp_history,
                bool squashed, const StaticInstPtr &inst, Addr corrTarget);
    void squash(ThreadID tid, void *bp_history);
};

}  // namespace branch_prediction
}  // namespace gem5

#endif  // __CPU_PRED_MYSIMPLE_PRED_HH__
```

MySimpleBP inherits the basic parameters, BTB, RAS and indirect branch predictor from *BPredUnit*. The methods it should implement are:

- `uncondBranch()` helps the BP when it needs to record or update infomation on an execution of unconditional branch instruction;
- `lookup()` is used to predict if a branch instruction should taken or not taken by giving its PC, and store the prediction history in *bp_history* (if it is needed);
- `update()` is used to update the BP when the actual branch outcome of a branch instruction ('taken' or 'not taken') is known, and the *bp_history* may also be used;
- `btbUpdate()` is used to update the BP (if needed) when a BTB entry is found invalid or missing;
- `squash()` is used to squash an outstanding update to the BP.

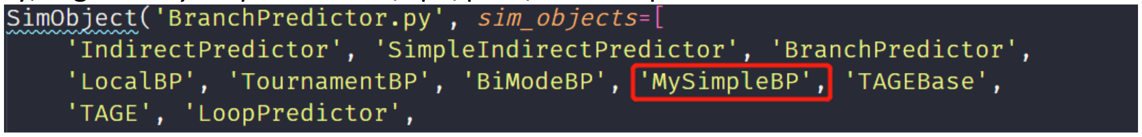
After this, create mysimplebp.cc:



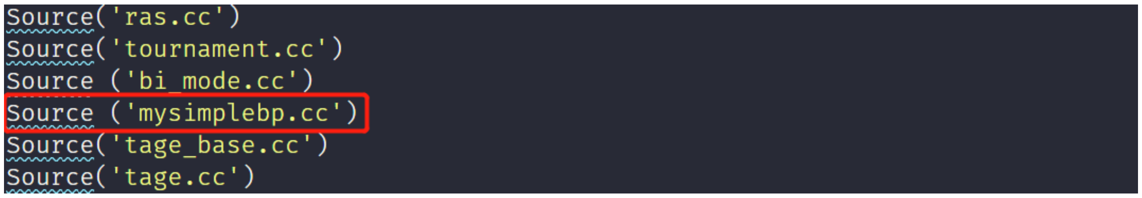
```
#include "cpu/pred/mysimplebp.hh"

namespace gem5 {

namespace branch_prediction {

MySimpleBP::MySimpleBP(const MySimpleBPParams &params) : BPredUnit(params) {}

bool MySimpleBP::lookup(ThreadID tid, Addr branch_addr, void *&bp_history) {
    return true;
}

void MySimpleBP::uncondBranch(ThreadID tid, Addr pc, void *&bpHistory) {}
void MySimpleBP::btbUpdate(ThreadID tid, Addr branch_addr, void *&bp_history) {}
void MySimpleBP::update(ThreadID tid, Addr branch_addr, bool taken,
                        void *bp_history, bool squashed,
                        const StaticInstPtr &inst, Addr corrTarget) {}
void MySimpleBP::squash(ThreadID tid, void *bp_history) {}

}  // namespace branch_prediction
}  // namespace gem5
```

Since simple BP always predicts ‘taken’, it needs no record or parameter, and needs not to update itself. Thus, `uncondBranch()`, `update()`, `btbUpdate()` and `squash()` do nothing. The `lookup()` will always return `true` (means ‘taken’) without recording the prediction history, and the contructor will simply call the constructor of *BPredUnit*.

Finally, register *MySimpleBP* to src/cpu/pred/SConscript:



```
SimObject('BranchPredictor.py', sim_objects=[
    'IndirectPredictor', 'SimpleIndirectPredictor', 'BranchPredictor',
    'LocalBP', 'TournamentBP', 'BiModeBP', 'MySimpleBP', 'TAGEBase',
    'TAGE', 'LoopPredictor',
```



```
Source('ras.cc')
Source('tournament.cc')
Source ('bi_mode.cc')
Source ('mysimplebp.cc')
Source('tage_base.cc')
Source('tage.cc')
```

To enable the simple BP, modify the configuration script configs/proj5/two-level.py, let CPU use the simple BP:

```
root.system.cpu.branchPred = MySimpleBP()
```

Then, recompile Gem5 and run the simulator with following commands:

```
scons build/ARM/gem5.opt -j8

build/ARM/gem5.opt configs/proj5/two-level.py
```

If everything goes correctly, we can see the BP has changed in config.ini.

Part 1 deliverables:

1. Implement *MySimpleBP*, provide a screenshot that proves the BP has been correctly modified to *MySimpleBP*;
2. run the 2MM and BFS workloads with Gem5. Use the simple BP and at least two other BPs provided by Gem5 (like tournament BP or local BP). Compare the performance of these BPs using the statistics in stats.txt. Statistics like branch predictor hit rates could be helpful to show the performance of BPs themselves, and statistics like IPC is useful when you are interested in the performance of the whole system. Please refer to Appendix B for the execution commands. You are not required to modify the other parameters like BTB size or RAS size.

Part 2: Read paper, implement and test perceptron BP

A brief introduction to perceptron BP

To introduce perceptron BP, we first describe a 2-bit BP as a base example. A simple 2-bit BP contains a saturating counter table. When a branch instruction is executed, the BP indexs into the counter table by the instruction's PC. If the counter is 2 (10) or 3 (11), BP predicts 'taken',otherwise it predicts 'not taken'. When the actual branch outcome is known, the counter will be updated. The counter is increased by 1 if the actual outcome is 'taken', and vice versa.
To be aware of branch history parttern, a global history register (GHR) can be added to the BP. The GHR records the history of each branch instruction and updates itself when an actual branch outcome is known. Further, when indexing into the counter table, the PC is firstly hashed (XOR) with GHR, thus the history information is taken in consideration. The value of GHR will be recorded in prediction history. When the actual branch outcome is known, this history is used to indexing into the counter table. It is also possible to replace GHR with a table of local history registers (LHR), which will be indexed by PC.
A simple perceptron BP, which is firstly described in the paper

Jiménez, Daniel A. and Calvin Lin. "Dynamic branch prediction with perceptrons." *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture* (2001): 197-206.

https://www.cs.utexas.edu/~lin/papers/hpca01.pdf

replaces the 2-bit counters with perceptrons. A perceptron is a vector of signed integers, which is its weights. When doing prediction, the BP computes the dot product of the weight vector and GHR (regard the bits as integers, and '0' as '-1'), instead of checking the saturating counter. If the dot product is non-negative, the BP predicts 'taken', and vice versa.
The BP updates the perceptron with a 'training' method when an actual branch outcome is known. The weights will be increased or decreased by 1 according to the GHR history, the

prediction correctness and the actual outcome. The BP also has a threshold for the weights. If the absolute value of the dot product reaches the threshold, the BP will not further train the perceptron if its prediction is correct.
In this part, you are required to read the paper listed above, summarize the method described in the paper, and implement the perceptron BP in Gem5. For your convenience, Appendix A provides a brief description of the method.

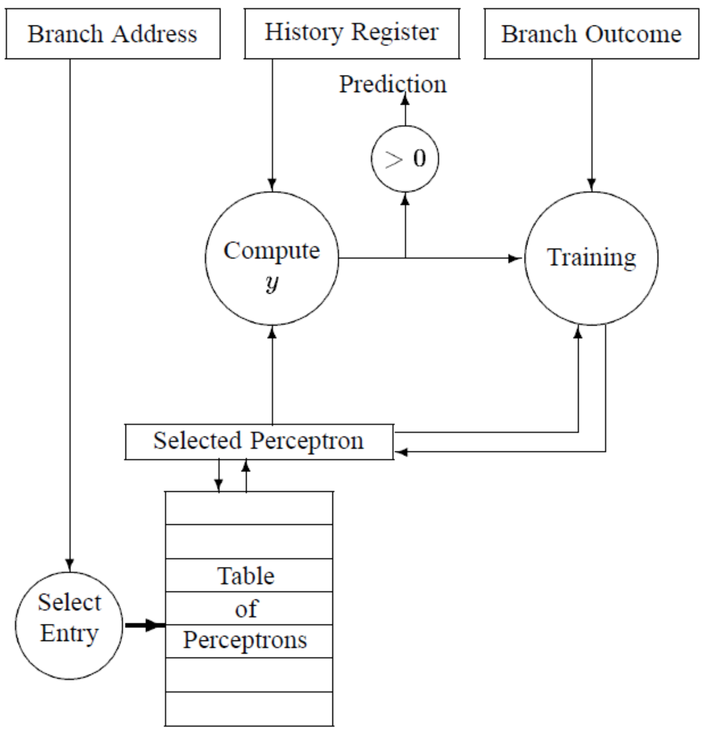
Part 2 deliverables:

1. Write a summary of the paper, which should include motivations, challenges, designs and evaluations.
2. Implement the perceptron BP described in the paper. To show the performance of your implementation, compare the statistics with other BPs on the 2MM and BFS workloads. You may have already noticed that Gem5 has provided a more complex perceptron BP, defined as *MultiperspectivePerceptron* (note this class itself is an abstract class). Compare your implementation with this BP. Also remember to include simple BP as a baseline. It is fine to use the statistics you have gained in Part 1.
3. Try to modify the parameters of your perceptron BP, like the threshold, the length of GHR and the size of perceptron table. Show the impact of these parameters on the performance. For simplicity, 3~5 values for each parameter is enough.

Appendix A

Perceptron BP method brief description

The overall structure of a perceptron BP is shown below:



The Select Entry step is to hash the PC. The hash can be XORing PC with global history register, or other methods. Each perceptron is a vector of weights $[w_i]_n$, and each weight is a signed integer. When computing the output y, the BP computes:

$$y = w_0 + \sum_{i=1}^{n} w_i x_i$$

where the x_i is the i^{th} bit of global history register (the '0' is regarded as '-1'). If $y > 0$, the prediction will be 'taken', and vice versa. When the actual branch outcome is known, the perceptron which is used to predict this branch is trained with following method:

```
if $sign(y) \neq t$ or $|y| < \theta$ then
    for $i$ in $0..n$ do
        $w_i += tx_i$
    end for
end if
```

where t is the actual outcome, set $t = 1$ for 'taken' and $t = -1$ for 'not taken'. θ is the threshold of the weights. About the strategy of choosing threshold, please refer to the paper.

Appendix B

```
process.cmd = ['test_bench/2MM/2mm_base']

process.cmd = ['test_bench/BFS/bfs','-f','test_bench/BFS/USA-road-d.NY.gr']

process.cmd = ['test_bench/bzip2/bzip2_base.amd64-m64-gcc42-nn','test_bench/bzip2/inp
ut.source','280']

process.cmd = ['test_bench/mcf/mcf_base.amd64-m64-gcc42-nn','test_bench/mcf/inp.in']
```