Advanced Computer Architecture: Lab 2 Report

Due on Octobor 30th at 11:59am

Name: **Zhenghong Yu**
Mailbox: **yuzhh1@shanghaitech.edu.cn**
Student ID: 2020533156

Contents

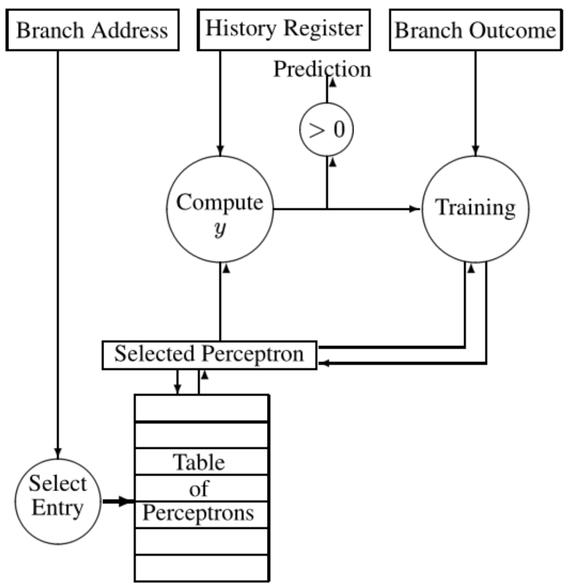
0.1 Preknowledge

The Lab2 is implemented in the branch **Lab2-naive-perceptron** and branch **Lab2-proved**.
Branch **Lab2-naive-perceptron** realize original implementation of dynamic branch prediction perceptron, while branch **Lab2-proved** realize some personal improvement based on naive implementation.
Here is some macro definition in **BranchPredictor.h**,

```
#define HARDWARE_BUDGET 8
#define THRESHOLD 79
#define HISTORY_LENGTH 34
```

These are default parameters acquired in the Lab2.pdf on piazza, if you want to modify, just change these macro definitions. The paper point out that a $-1, 1$ label perceptron will use signed type of weights, so I do not offer macro definition that can change the type of perceptron weights.

1 Requirement of Branch Prediction with Perceptron



In general, we need:

- A **Perceptrons Table** to store all groups of weights respect to different hash value of branch addresses
- A **History Register** to store all informations (take as 1/not take as -1) of different previous dynamic flow branches.

Each time we get a branch instruction at decode stage:

- First we hash the PC address, take the hash value as the index of **Perceptrons Table** and select out a group of weights.
- Second, we calculate the output y of perceptron by using the weights w_i and history branches value x_i in **History Register**. $y = w_0 + \sum_{i=1}^{n} x_i w_i$. The w_0 is a bias.
- If y is negative, return a result of branch prediction untaken. If y is non-negative, return a result of branch prediction taken.

Each time we get the result of a branch at excute stage:

- We calculate the y_{out} which is used for training, if output y is larger than thresold, y_{out} is 1, if y is less than negative threshold, y_{out} is -1, otherwise, y_{out} is 0.
- Compare y_{out} with the result t (1 as taken, -1 as not taken) of the corresponding branch, if they are same, the previous prediction is correct, while others are incorrect.
- If the previous prediction is correct
  - Do nothing
- If the previous prediction is not correct
  - Update the corresponding(PC hash value index) perceptron weights in **Perceptron Table**, $w = w + t^T x$
- We update **History Register**
  - We first find LRU index i of **History Registor**
  - We evit the ith register, replace it by the current branch and its result.

2 Code Implementation

BranchPredictor.h

```
class BranchPredictor
{
public:
  enum Strategy
  {
    ...
    PERCEPTRON /* Branch Perceptron */
  } strategy;
  ...
private:
  std::vector<std::vector<int8_t>> tableOfPerceptron;
  std::vector<std::vector<int8_t>> historyTable;
  uint32_t lastRefence = 0;    /* To maintian LRU of history branch */
  int32_t yOut = 0;
}
```

BranchPredictor.c, BranchPredictor::predict()

```
bool BranchPredictor::predict(uint32_t pc, uint32_t insttype,
    int64_t op1, int64_t op2, int64_t offset)
  {
    switch (this->strategy)
    {
    ...
    case PERCEPTRON:
    {
      uint32_t hashPC = (pc % (HARDWARE_BUDGET * 1024)); /* PC hash value */
```

```
        int32_t output = 0; /* initialize */
        for (int i = 0; i < HISTORY_LENGTH; i++)
        {
          output += this->tableOfPerceptron[hashPC][i]*this->historyTable[i][0];
        }
        /* add bias */
        output += this->tableOfPerceptron[hashPC][HISTORY_LENGTH];
        /* calculate y-out for training */
        if (output > THRESHOLD)
        {
          this->yOut = 1;
        }
        else if (output < -THRESHOLD)
        {
          this->yOut = -1;
        }
        else
        {
          this->yOut = 0;
        }
        /* result of prediction */
        if (output < 0)
        {
          return false;
        }
        else
        {
          return true;
        }
      }
      break;
      default:
        dbgprintf("Unknown Branch Perdiction Strategy!\n");
        break;
    }
    return false;
  }
```

BranchPredictor.c, BranchPredictor::update()

```
void BranchPredictor::update(uint32_t pc, bool branch)
{
  ...
  if (branch)
  {
    this->lastRefence++;  /* Maintain LRU refence */
    if (this->strategy == PERCEPTRON)
    {
      if (this->yOut != 1)
      { /* If yout != result, update weights */
```

```
        uint32_t hashPC = (pc % (HARDWARE_BUDGET * 1024));
        for (int i = 0; i < HISTORY_LENGTH; i++)
        {
          this->tableOfPerceptron[hashPC][i] += this->historyTable[i][0];
        }
        this->tableOfPerceptron[hashPC][HISTORY_LENGTH]++;
      }
    }
    else
    { /* Other predict mode keep same */
      if (state == STRONG_NOT_TAKEN)
      {
        this->predbuf[id] = WEAK_NOT_TAKEN;
      }
      else if (state == WEAK_NOT_TAKEN)
      {
        this->predbuf[id] = WEAK_TAKEN;
      }
      else if (state == WEAK_TAKEN)
      {
        this->predbuf[id] = STRONG_TAKEN;
      } // do nothing if STRONG_TAKEN
    }

    int8_t index = findReplaceIndexOfHistoryTable();  /* Find the LRU place */
    this->historyTable[index][0] = 1;                /* Update result of branch */
    this->historyTable[index][1] = this->lastRefence; /* Update LRU refence */
  }
  else
  {
    this->lastRefence++;
    if (this->strategy == PERCEPTRON)
    {
      if (this->yOut != -1)
      { /* If yout != result, update weights */
        uint32_t hashPC = (pc % (HARDWARE_BUDGET * 1024));
        for (int i = 0; i < HISTORY_LENGTH; i++)
        {
          this->tableOfPerceptron[hashPC][i] -= this->historyTable[i][0];
        }
        this->tableOfPerceptron[hashPC][HISTORY_LENGTH]--;
      }
    }
    else
    {
      /* Other predict mode keep same */
      if (state == STRONG_TAKEN)
      {
        this->predbuf[id] = WEAK_TAKEN;
```

```
      }
      else if (state == WEAK_TAKEN)
      {
        this->predbuf[id] = WEAK_NOT_TAKEN;
      }
      else if (state == WEAK_NOT_TAKEN)
      {
        this->predbuf[id] = STRONG_NOT_TAKEN;
      } // do noting if STRONG_NOT_TAKEN
    }

    int8_t index = findReplaceIndexOfHistoryTable();  /* Find the LRU place */
    this->historyTable[index][0] = -1;               /* Update result of branch */
    this->historyTable[index][1] = this->lastRefence; /* Update LRU refence */
  }
}
```

BranchPredictor.c, BranchPredictor::findReplaceIndexOfHistoryTable()

```
/* Function to find replace branch in history table */
int8_t BranchPredictor::findReplaceIndexOfHistoryTable(void)
{
  int32_t minTmp = this->historyTable[0][1];
  int8_t index = 0;
  for (int i = 0; i < HISTORY_LENGTH; i++)
  {
    if (this->historyTable[i][1] < minTmp)
    {
      minTmp = this->historyTable[i][1];
      index = i;
    }
  }
  return index;
}
```

While some other basic implementation I'll not list here, such as, modify functions to parse **pc** in class **BranchPredictor**, initialize vectors stated in **BranchPredictor.h**, etc.. They are too simple and naive to list here.

3 Basic Implementation Correctness and Performance

3.1 Correctness

I simply run three test, **quicksort.c, ackermann.c, matrixmulti.c**, here is three result

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/matrixmulti.riscv -b PERCEPTRON
The content of A is:
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9
The content of B is:
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
The content of C=A*B is:
0 0 0 0 0 0 0 0 0 0
0 10 20 30 40 50 60 70 80 90
0 20 40 60 80 100 120 140 160 180
0 30 60 90 120 150 180 210 240 270
0 40 80 120 160 200 240 280 320 360
0 50 100 150 200 250 300 350 400 450
0 60 120 180 240 300 360 420 480 540
0 70 140 210 280 350 420 490 560 630
0 80 160 240 320 400 480 560 640 720
0 90 180 270 360 450 540 630 720 810
Program exit from an exit() system call
```

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/ackermann.riscv -b PERCEPTRON
Ackermann(0,0) = 1
Ackermann(0,1) = 2
Ackermann(0,2) = 3
Ackermann(0,3) = 4
Ackermann(0,4) = 5
Ackermann(1,0) = 2
Ackermann(1,1) = 3
Ackermann(1,2) = 4
Ackermann(1,3) = 5
Ackermann(1,4) = 6
Ackermann(2,0) = 3
Ackermann(2,1) = 5
Ackermann(2,2) = 7
Ackermann(2,3) = 9
Ackermann(2,4) = 11
Ackermann(3,0) = 5
Ackermann(3,1) = 13
Ackermann(3,2) = 29
Ackermann(3,3) = 61
Ackermann(3,4) = 125
Program exit from an exit() system call
```

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/quicksort.riscv -b PERCEPTRON
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28
 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 7
6 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
```

We can clear see that three programs all excuted correctly.

3.2 Performance

I still use these three programs, **quicksort.c, ackermann.c, matrixmulti.c**, I apply all **Always Taken, Always Not Taken, Back Taken Forward Not Taken, Branch Prediction Buffer, Dynamic Branch Prediction with Perceptron** prediction policy and thus get a table. In this table, horizontal axis is the policies, the vertical axis is three test programs, and the remaining is the correct prediction percent.

	AT	NT	BTFNT	BPB	DP
quicksort	50.75	49.26	95.06	95.87	96.23
ackermann	49.55	50.45	50.53	95.93	92.48
matrixmulti	62.35	37.65	63.25	62.75	60.98

The highlighed block is the highest correct prediciton percent of each program among five policies.
We find that, **Dynamic Branch Prediction with Perceptron** perform best in **quicksort.c**, but perform worse in **ackermann.c** with 3.45% distance, in **matrixmulti.c** with 2.27% distance.

4 Analysis and Improvement

4.1 Why all perform bad in matrixmulti.c

As we look into the codes, we can find that **matrixmulti.c** has many large nested(three maximum) loop, the deepest loop's code only excute 10 times each enueration, which means many dynamic flow branch do not have a proper pattern to predict or learn. This problem cannot solved by only change the parameter of perceptron, multilayer perceptron might learn this kind of pattern well, but it will be a large overhead on hardware.

4.2 How to prove naive single layor perceptron

As I noticed, if the prediction is wrong, **Dynamic Branch Prediction with Perceptron** policy will training the weights by adding an inverse gradient step. Thus, this policy often fails when there come a new branch in dynamic logical flow. Also, it always fails in the next few prediction because each time the weights only change by only 1 step. My thinking is that, if I detect that the previous branch is same with what I current excute, and my prediction is same with the previous branch result, I will change the weights by another gradient step.
Also, I noticed that each time we evit a LRU dynamic flow branch out of **History Register**, the corresponding weights in all entries of **Perceptrons Table** should be changed, too. Now, these invalid weights just keep the same, but it will cause problem if the new branch result is different with the LRU branch result, which means the weights should not be inherited. So I add more steps, if two branches noted before are same in result, I just keep the weights unchanged, if they are different, I will change the weights all to 1 is the new branch is result in taken, -1 otherwise.
Here is the code.
BranchPredictor.h

```
class BranchPredictor
{
  ...
private:
  ...
  uint32_t lastPc = 0; /* To store previous branch PC */
  bool lastRes = 0;    /* To store previous branch result */
  ...
}
```

BranchPredictor.c, BranchPredictor::update()

```
void BranchPredictor::update(uint32_t pc, bool branch)
{
```

```
  ...
  if (branch)
  {
    ...
    if (this->strategy == PERCEPTRON)
    {
      ...
      if (pc == this->lastPc && branch == this->lastRes)
      { /* If all same, give additional training */
        this->tableOfPerceptron[hashPC][HISTORY_LENGTH] += 1;
      }
    }
    ...
    this->lastPc = pc;
    this->lastRes = branch;
  }
  else
  {
    ...
    if (this->strategy == PERCEPTRON)
    {
      ...
      if (pc == this->lastPc && branch == this->lastRes)
      { /* If all same, give additional training */
        this->tableOfPerceptron[hashPC][HISTORY_LENGTH] -= 1;
      }
    }
    ...
    this->lastPc = pc;
    this->lastRes = branch;
  }
}
```

Here is the result, which highlighed in purpose.

	AT	NT	BTFNT	BPB	DP	DP-improve
quicksort	50.75	49.26	95.06	95.87	96.23	96.1
ackermann	49.55	50.45	50.53	95.93	92.48	94.99
matrixmulti	62.35	37.65	63.25	62.75	60.98	58.33

We can see that, this implementation sacrificed only 0.13% percent of **quicksort.c** branch prediction correct rate to gain 2.41% percent of **ackermann.c** branch prediction correct rate. But the **matrixmulti.c** branch prediction correct rate is still a big problem.