IS2202 Lab 2: Micro-architectural exploration

Renzhi Xing
931011-2025
renzhi@kth.se

May 22, 2017

1 Direct Portion

1.1 Collecting IPC statistics using the MAI

- What is the recorded IPC for each benchmark? Which benchmark had the best IPC, and which had the worst?

benchmarks	vortex	equake	parser
cycle 0M	0	0	0
cycle 1M	848471	2005209	1158364
cycle 2M	1751824	4027208	2294951
cycle 3M	2624048	6064758	3441473
cycle 4M	3502003	8007200	4574463
cycle 5M	4429172	10013393	5685590
cycle 6M	5317086	11949640	6832926
cycle 7M	6240305	13941297	7983926
cycle 8M	7154859	15967986	9156310
cycle 9M	8034798	18001645	10315086
recorded IPC	0.89	2.00	1.15

Since one step refers to one instruction, the IPC in the table above is calculated by number of total steps divided by 9M cycles. From the results, it's obvious that *vortex* has the lowest IPC and *equake* has the highest.

1.2 Collecting data about the effect of superscalar pipeline width on IPC

- Are there diminishing returns on increasing pipeline width? How does reorder buffer size affect this performance? What factors might limit the effectiveness of increasing pipeline width?

The benchmark I chose to test was *equake*. Since it has a best IPC among three benchmarks

Reorder Buffer Size 32					
pipeline width	1	2	4	8	16
cycle 0M	0	0	0	0	0
cycle 1M	845116	1414416	2005209	2057374	1852034
cycle 2M	1698947	2818676	4027208	4012891	3683466
cycle 3M	2526914	4270435	6064758	5994671	5519974
cycle 4M	3346955	5747528	8007200	7952847	7373998
cycle 5M	4212029	7159255	10013393	9975928	9232008
cycle 6M	5038722	8593175	11949640	11964315	11179858
cycle 7M	5865841	10003655	13941297	14071432	13011583
cycle 8M	6724994	11379761	15967986	16093273	14861028
cycle 9M	7555690	12791356	18001645	18141815	16832626
recorded IPC	0.84	1.42	2.00	2.02	1.87

Reorder Buffer Size 64					
pipeline width	1	2	4	8	16
cycle 0M	0	0	0	0	0
cycle 1M	852958	1439981	2158615	2376536	2208289
cycle 2M	1708791	2928207	4323376	4610286	4420617
cycle 3M	2537924	4362362	6492570	6842553	6635836
cycle 4M	3355140	5841372	8617442	9330933	8846805
cycle 5M	4183923	7340385	10732903	11827124	11060893
cycle 6M	5010609	8747280	12884435	14352195	13230306
cycle 7M	5827762	10185231	15077882	16872205	15426611
cycle 8M	6657094	11628097	17246461	19400552	17633061
cycle 9M	7506117	13052288	19374166	21943327	19867678
recorded IPC	0.83	1.45	2.15	2.44	2.21

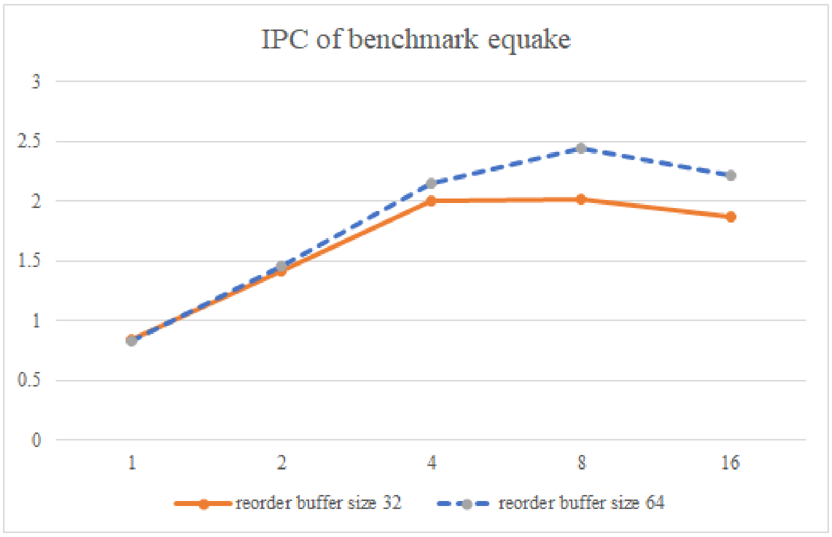

Figure 1: IPC of benchmark *equake*

Based on the tables and figure 1 above, the changing of IPC along with pipeline width can be analysed.

First, in general, IPC of benchmark increases with the increasing size of reorder buffer. This means a large reorder buffer helps improving the performance. This is because the larger reorder buffer can store more results, which makes the performance better.
Second, we can see from the figure that when with a very small pipeline width, for example 1, even the increase of buffer size can't improve the performance anymore. This is because small pipeline restricts instruction running speed to a low level, making reorder buffer work with a low efficiency.
Third, an interesting thing is IPC is not always increasing with the expending of pipeline width. When width increased to 16, the performance decreased. The reason why is when the pipeline is wide enough, there were more instructions running in parallel, which may cause more data hazard at one time, and longer penalty time.

1.3 Collecting data about the effect of memory latency on OoO efficiency

- What impact does increased memory latency have on performance? To what degree does out-of-order execution mask the increased memory hierarchy delays?

(cache access delay,.., memory access delay)	(1, 1, 10)	(2, 2, 10)	(5, 5, 10)	(1, 1, 20)	(1, 1, 50)
cycle 0M	0	0	0	0	0
cycle 1M	2005209	1641334	1145893	1807896	1417491
cycle 2M	4027208	3295873	2310219	3563124	2740645
cycle 3M	6064758	4968069	3457906	5512039	4075091
cycle 4M	8007200	6681246	4641228	7358603	5533027
cycle 5M	10013393	8337589	5776469	9126270	6886112
cycle 6M	11949640	9999773	6934744	10900646	8124199
cycle 7M	13941297	11668511	8064638	12648933	9644919
cycle 8M	15967986	13359232	9230144	14350822	11213231
cycle 9M	18001645	15055890	10356865	16043176	12702340
recorded IPC	2.00	1.67	1.15	1.78	1.41

When access delay increases, either cache access or memory access, IPC decreased. We can learn that increased memory latency will make the performance worse.
When analyzing the execution of OoO, we can check the relation between the penalty changing and the IPC changing. Comparing (1, 1, 10), (2, 2, 10) and (5, 5, 10), we can see that the decreasing of IPC is not linear to the increasing of cache access delay, when delay turns from 2 to 5, the decreasing of IPC becomes slower, which is because the mask of OoO. What's more, when comparing(1, 1, 10), (1, 1, 20) and (1, 1, 50), things went in the same way. We can learn that OoO plays an important role in covering the increase of memory hierarchy delays, however, this can't prevent the performance become worse.

2 Open-ended Portion

2.1 Branch predictor study

- Submit well-commented source code for your predictor, an overall description of its functionality, and a summary of its performance on 3-5 of the benchmarks provided with the framework. Report which benchmarks you tested your predictor out on.

The function of a predictor is actually to classify whether the branch is taken or not. Thus, an artificial neural network (ANN) algorithm called *perceptron*, which has a strength of dealing with classification problem, can be introduced in this task.

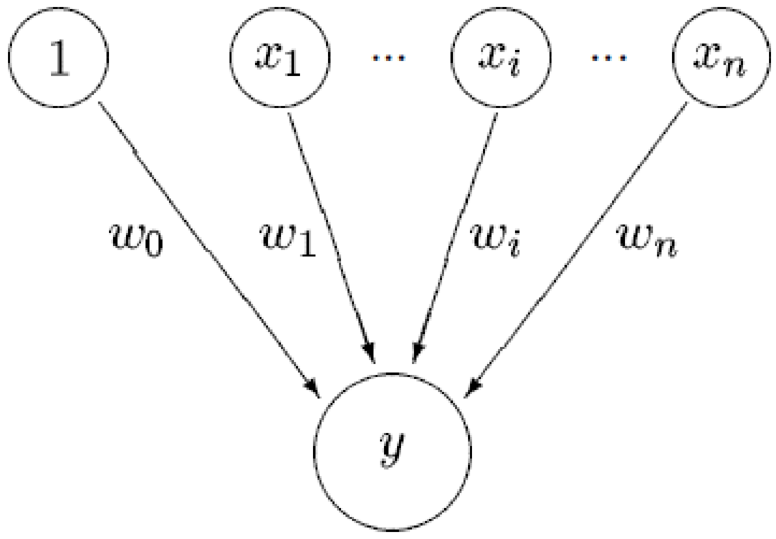

Figure 2: Perceptron Neural Network Model[1]

Input and output of a perceptron neural network are all in binary format. The network is working as in figure 2. Every input will have its own weight, which is based on how much the input affect on the output. The sum of results of inputs multiplied by weights will then go through a threshold, which makes it turn into a binary form output. This definitely fits the branch prediction problem: The binary format can be translated into taking a branch or not, thus, perceptron can be trained by the historical branch information, and give an output of current branch prediction. The great thing about perceptron is that it can achieve online learning, which means training is done at the same time as giving output. The algorithm of perceptron can be described as:

1. Hash branch address from the Program Counter.
2. According to branch address, get the certain perceptron weight array.

3. Calculate the value of output y according to the equation

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

   in which x_i is fetched from the branch taken history register.

4. if $y \geq 0$, then branch is taken; otherwise not.

5. Once a branch instruction is over, update the predictor weight and history register.

6. Write the updated weight back to perceptron weight table.

The code achievement is shown as following.

```
//predictor.h

#ifndef _PREDICTOR_H_
#define _PREDICTOR_H_

#include "utils.h"
#include "tracer.h"
#include "math.h"

class PREDICTOR{

 private:
  int   *ghr;            // global history register
  UINT32  inputLength;
  UINT32  outputLength;
  UINT32  theta;
  int **W;

 public:

  // The interface to the four functions below CAN NOT
      be changed

  PREDICTOR(void);
  ~PREDICTOR(void);
  bool     GetPrediction(UINT32 PC);
  void     UpdatePredictor(UINT32 PC, bool resolveDir,
     bool predDir, UINT32 branchTarget);
  void     TrackOtherInst(UINT32 PC, OpType opType,
     UINT32 branchTarget);

  // Contestants can define their own functions below

};

#endif
```

```
//predictor.cc

#include "predictor.h"
// the predictor is based on preceptron learning
   neural network, and paper "Neural Methods for
   Dynamic Branch Prediction"

//Num of input nodes
//which is the length of history, best performing
   between 4 and 66 according to the paper
#define IN_LEN  35

//Num of output nodes
//which is also the size the weight table
//depends on the budget
//in this case with 32kB budget, size <= 32 * 1024 *
   8/ ((IN_LEN + 1) * 8)
#define OUT_LEN 250

//Weight Size
//According to the paper, 8-bit is a good choice, the
   defines are for the checking of data overflow
#define WEIGHT_SIZE_MAX 128
#define WEIGHT_SIZE_MIN -127

PREDICTOR::PREDICTOR(void){

  inputLength = IN_LEN;
  outputLength = OUT_LEN;
  theta = floor(1.93 * (float)IN_LEN + 14);

  // Defination of Weight Matrix
  W = new int*[outputLength];
  for (int i = 0; i < outputLength; i++)
          W[i] = new int[inputLength + 1];
  // Arrays storing the history
  ghr = new int[inputLength + 1];

  //initialize W and ghr
  ghr[0] = 1; //this is for bias, will always be 1
  for (int j = 1; j <= inputLength; j++){
          ghr[j] = -1;
  }
  for (int i = 0; i < outputLength; i++){
          for (int j = 0; j <= inputLength; j++){
                  W[i][j] = 0;
          }
  }
  //end initialization
```

```
}

PREDICTOR::~PREDICTOR(void){
        for (int i = 0; i < outputLength; i++)
                delete[] W[i];
        delete[] W;
        delete[] ghr;
}

bool  PREDICTOR::GetPrediction(UINT32 PC){
//***** 1. The branch address is hashed to produce an
   index 0~IN_LEN-1 into the table of perceptrons.
  UINT32 index = PC % outputLength;
  int y = 0;
//***** 2. fetch the perceptron
//***** 3. calculate value of y
  for(int i = 0; i < IN_LEN ; i++){
          y += W[index][i] * ghr[i];
  }
//***** 4. if y is negative, branch is not taken,
   otherwise taken
  return ((y < 0)?NOT_TAKEN:TAKEN);
}

void  PREDICTOR::UpdatePredictor(UINT32 PC, bool
   resolveDir, bool predDir, UINT32 branchTarget){
  UINT32 index = PC % (outputLength);
// Update of Weight Matrix
// 5. updating vector register
// 6. write back to the vector register
  if (predDir != resolveDir){
          for(int i = 0; i <= inputLength; i++){
                  if(((ghr[i]+1)/2) == resolveDir){
                         if(W[index][i] < 127) //
                            overflow checking
                                 W[index][i]++;
                  }
                  else{
                         if(W[index][i] > -128)
                                 W[index][i]--;
                  }
          }
  }

// update the GHR
  for(int i = 1; i < inputLength; i++){
          ghr[i] = ghr[i+1];
  }
  if(resolveDir == TAKEN){
          ghr[inputLength] = 1;
  }else{
          ghr[inputLength] = -1;
  }
}

void    PREDICTOR::TrackOtherInst(UINT32 PC, OpType
   opType, UINT32 branchTarget){
  return;
}
```

The results of original code GSHARE and the perceptron neural network branch predictor is collected.

MISPRED PER 1K INST		
Benchmarks	original code(GSHARE.32KB)	perceptron branch predictor
LONG-SPEC2K6-00	3.974	4.509
LONG-SPEC2K6-02	5.176	6.949
SHORT-MM-1	9.172	11.002
SHORT-MM-3	4.267	3.090
SHORT-FP-1	3.479	2.549
SHORT-FP-2	1.061	1.505
SHORT-SERV-1	3.646	7.586
SHORT-SERV-3	5.870	9.893

From the results, we can get that the performance of perceptron predictor is working a bit worse than the original code. There may be various reasons, what I can think of are:

1. Due to time limit, the predictor is not well designed, and some the parameters, including the number of input nodes and size of weight table, are not fully tested and adjusted. In further research this will be the top task on the list.

2. According to the property of perceptron learning, it can only deal with separable data, which is not guaranteed in our benchmarks. Actually, this can be a main reason of the different behaviour of the two predictors on same benchmark. For example, with benchmark *SHORT-MM-3*, the perceptron branch predictor works much better than the original predictor, while with *SHORT-SERV-3*, original predictor's behaviour didn't change much, but the miss predictions of perceptron predictor increased by over 3 times. This might be the result of *SHORT-SERV-3* works with non-linear separable branch taken history. *Compare to original predictor, perceptron needs more calculations in every prediction and update, with many multiplications and other complicate calculations, the efficiency of code must be increased.*

References

[1] Jiménez, Daniel A and Lin, Calvin, *Neural methods for dynamic branch prediction.* ACM Transactions on Computer Systems (TOCS), 2002, 20(4): 369-397.