18-640: Foundations of Computer Architecture

**Project 1: Branch Prediction**

Group Id: 7 Member: luting Wang/ Wei Feng

**Part 1: Analysis of Tournament and Bi-mode Predictor:**

1. ***Bi-mode Predictor:***
2. Simulate the stringsearch benchmark

We used the simulator in syscall emulation (SE) mode for simulation, and the command are as follows:

*build/X86/gem5.opt configs/example/se.py -c binary [-o options] --cpu-type=detailed --caches --pred-type=bimode –global-pred-size=2048 --choice-pred-size=4096*

The configurations we used for global predictor size and branch prediction size are shown in Table1.

1. Performance

Table1 configuration table of Bi-mode and its performance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Config 1** | **Config 2** | **Config 3** | **Config 4** |
| Global Predictor Size | 2048 | 4096 | 8192 | 8192 |
| Choice Predictor Size | 4096 | 8192 | 4096 | 8192 |
| Branch  Prediction rate | 92.8667% | 92.8889% | 92.9087% | 92.9085% |
| IPC | 0.9799 | 0.9814 | 0.9817 | 0.9816 |

1. Analysis

As you can see, the branch prediction rate of Bi- mode predictor varies for different configurations. The prediction rate reaches its maximum value with global predictor size and choice predictor size of 8192, and the maximum IPC is 0.9816. Basically, as the size of predictor increases, the prediction rate increase, as well as the IPC. The reason is obvious. When the predictor size increases, it would decrease the aliasing, so there would be more branches predicted corrected. And when the prediction rate increases, it would reduce the instruction stalls, and IPC will therefore increases.

1. ***Alpha 21264 Tournament Predictor:***
2. Simulate the stringsearch benchmark

The command are as follows:

*build/X86/gem5.opt configs/example/se.py -c binary [-o options] --cpu-type=detailed --caches --pred-type=tournament --local-pred-size=2048 –global-pred-size=8192 --choice-pred-size=8192*

The configurations we used for local predictor size, global predictor size and branch prediction size are shown in Table2.

1. Performance

Table2 configuration table of tournament and its performance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Config 1** | **Config 2** | **Config 3** | **Config 4** |
| Local Predictor Size | 2048 | 4096 | 4096 | 4096 |
| Global Predictor Size | 8192 | 4096 | 8192 | 8192 |
| Choice Predictor Size | 8192 | 8192 | 4096 | 8192 |
| Branch  Prediction rate | 92.9536% | 92.9689% | 92.9775% | 92.9898% |
| IPC | 0.9969 | 0.9969 | 0.9972 | 0.9988 |

1. Analysis

The performance of Tournament Predictor are better than Bi-mode predictor. Branch prediction rate increases by 0.08% with the best configuration, and IPC gets up to 0.9988, which is very close to 1. It is because Tournament hybrid with gshare and Pap, and a meta-predictor M indexed by the low-order bits of the branch address, which makes prediction of which predictor to be used, gshare or Pap. It combines both the Pap and gshare’s performance and has better flexibility.

**Part 2: Implement gshare and YAGS Predictor**

1. ***Gshare Predictor:***
2. Implementation design

Gshare is a global-history two-level predictor which makes better use of the index bits. The algorithm is quite simple. It hashes the BHR and the PC address together to select an entry from the PHT. For example, in Gap, even though the overall PC and history bits are different, using 2 bits from BHR and PC will lead to the same entry. And it would leave out some information. But through hashing we can avoid it. We use an XOR operator to yield different PHT indices, and it works very well.

The main “look up” is implemented as follows: the PHT index is replaced with an XOR operator, which operates on bits global history and the upper bits of the branch address bits. And the final prediction is made by the PHT’s most-significant bit, which is a two-bit saturating counter.

*bool*

*GshareBP::lookup(Addr branchAddr, void \* &bpHistory)*

*{*

*unsigned localPredictorIdx = (((branchAddr >> instShiftAmt)//XOR*

*^ globalHistoryReg)*

*& localPredictorMask);*

*assert(localPredictorIdx < localPredictorSize );*

*bool finalPrediction = localCtrs[localPredictorIdx].read()*

*> localThreshold;*

*BPHistory \*history = new BPHistory;*

*history->globalHistoryReg = globalHistoryReg;*

*history->finalPred = finalPrediction;*

*bpHistory = static\_cast<void\*>(history);*

*updateGlobalHistReg(finalPrediction);*

*return finalPrediction;*

*}*

When we get the final prediction outcome, we update both the global history register and PHT. You can refer to it in the “update” function.

*if (taken)*

*{*

*localCtrs[localPredictorIdx].increment();*

*} else {*

*localCtrs[localPredictorIdx].decrement();*

*}*

*if (squashed) {*

*if (taken) {*

*globalHistoryReg = (history->globalHistoryReg << 1) | 1;*

*} else {*

*globalHistoryReg = (history->globalHistoryReg << 1);*

*}*

*globalHistoryReg &= historyRegisterMask;*

*} else {*

*delete history;*

*}*

1. Performance

Table3 configuration table of Gshare and its performance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Config 1** | **Config 2** | **Config 3** | **Config 4** |
| Local Predictor Size | 2048 | 4096 | 8192 | 16384 |
| Branch  Prediction rate | 92.4982% | 92.6361% | 92.8569% | 92.8700% |
| IPC | 0.9384 | 0.9541 | 0.9732 | 0.9750 |

1. Analysis

The relationship between predictor performance and predictor size of Gshare is similar as we discussed above for Bi-mode and Tournament predictor. Predictor’s performance is improved as we increase the local predictor size. But since Gshare only use one PHT, its prediction rate is lower than Bi-Mode, which use two PHTs to reduce the effects of aliasing. And you can see, the prediction rate growth drops every time we double the predictor size. Maybe it is because of the branch aliasing, which doesn’t be avoid using Hashing.

1. ***YAGS Predictor***
2. Implementation design

YAGS predictor is similar to Bi-mode predictor, except for that the two PHTs save only the instances that do not agree with the direction bias. So the PHTs can be replaced with two caches: T-cache and an NT-cache. And for each cache entry, it has two structures: a 2-bit counter and a tag. The tag is usually 6-8 bits, and in our implementation, we use 8 bits. The tag is used to record the branch instance that do not agree with its overall bias.

If a branch does not have an entry in its corresponding cache, then the choice counter is used to make the prediction.

*if(notTakenTags[tagsIdx]!=leastSigAddr)*

*finalPrediction = choicePrediction;*

And if it does have an entry, for example, if the choice counter predicts taken, and we consult the NT-cache, and we find there is a hit, we use the cache’s counter to predict. The cache is indexed the same way as Gshare, and the tag is compared the least-significant bits of branch address, in our implementation, it is 8 bits. The implementation of “look up” function is shown here.

*if (choicePrediction) {*

*if(notTakenTags[tagsIdx]!=leastSigAddr){*

*finalPrediction = choicePrediction;*

*}else {*

*finalPrediction = notTakenGHBPrediction;*

*cacheUsed = true;*

*}*

*}*

*else {*

*if(takenTags[tagsIdx] !=leastSigAddr){*

*finalPrediction = choicePrediction;*

*}else {*

*finalPrediction = takenGHBPrediction;*

*cacheUsed = true;*

*}*

*}*

*history->finalPred = finalPrediction;*

*history->cacheUsed = cacheUsed;*

*bpHistory = static\_cast<void\*>(history);*

*updateGlobalHistReg(finalPrediction);*

*return finalPrediction;*

*}*

And for update, it is a bit complicated. For the choice PHT, it is updated the same way as Bi-mode (always updated with the branch outcome, except when the choice PHT's direction is the opposite of the branch outcome, but the overall prediction of the selected PHT bank was correct). And for cache, it is updated when the choice PHT's direction is the opposite of the branch outcome.

*if(history->cacheUsed){*

*if (history->takenUsed) {*

*if (taken) {*

*notTakenCounters[globalHistoryIdx].increment();*

*} else {*

*notTakenCounters[globalHistoryIdx].decrement();*

*}*

*} else {*

*if (taken) {*

*takenCounters[globalHistoryIdx].increment();*

*} else {*

*takenCounters[globalHistoryIdx].decrement();*

*}*

*}*

*}*

*else{*

*if(history->finalPred!=taken){*

*if(taken){*

*takenTags[globalHistoryIdx]=leastSigAddr;*

*}*

*else{*

*notTakenTags[globalHistoryIdx]=leastSigAddr;*

*}*

*}*

*}*

*if(history->cacheUsed){*

*if (history->finalPred == taken) {*

*if (taken == history->takenUsed) {*

*if (taken) {*

*choiceCounters[choiceHistoryIdx].increment();*

*} else {*

*choiceCounters[choiceHistoryIdx].decrement();*

*}*

*}*

*} else {*

*if (taken) {*

*choiceCounters[choiceHistoryIdx].increment();*

*} else {*

*choiceCounters[choiceHistoryIdx].decrement();*

*}*

*}*

*}else{*

*if(taken){*

*choiceCounters[choiceHistoryIdx].increment();*

*}else {*

*choiceCounters[choiceHistoryIdx].decrement();*

*}*

*}*

*if (squashed) {*

*if (taken) {*

*globalHistoryReg = (history->globalHistoryReg << 1) | 1;*

*} else {*

*globalHistoryReg = (history->globalHistoryReg << 1);*

*}*

*globalHistoryReg &= historyRegisterMask;*

*} else {*

*delete history;*

*}*

*}*

1. Performance

Table4 configuration table of yagst and its performance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Config 1** | **Config 2** | **Config 3** | **Config 4** |
| Global Predictor Size | 2048 | 4096 | 8192 | 8192 |
| Choice Predictor Size | 4096 | 8192 | 4096 | 8192 |
| Branch  Prediction rate | 92.9060% | 92.9239% | 92.9517% | 92.9489% |
| IPC | 0.9849 | 0.9866 | 0.9888 | 0.9887 |

1. Analysis

The performance has been improved, compared to Gshare predictor as well as Bi-mode predictor, but a little inferior than Tournament predictor. Compared with Bi-mode predictor, it saves space as well as keeps better performance.

**Part 3: Challenge**

We use set-associate and search strategy to improve the performance of Gshare and YAGS. However, we get the same result compared with the original Gshare and YAGS algorithm. Then we change the global BHR size to 16384, and get a better result, but it doesn’t match the size in question.

1. Gshare

We add “tags” and “time” in the PHT structure. Tag is similar to the YAGS, which record the result of branch address XOR global BHR. And times record how many times the branch instruction used.

|  |  |  |
| --- | --- | --- |
| Sat-counter | Tag | times |

*GshareBP::search(Addr addr){*

*unsigned pos = 0;*

*unsigned min = times[0];*

*for(int i=0;i<localPredictorSize;i++){*

*if(time[i]==0){*

*tags[i] = addr;*

*flag[i] = true;*

*return i;*

*} else if ( addr == tags[i]){*

*times[i]++;*

*return i;*

*}*

*if(times[i]<min){*

*min = times[i];*

*pos = i;*

*}*

*}*

*tags[pos] = addr;*

*localCtrs[pos].setBits(localCtrBits);*

*times[pos]=0;*

*return pos;*

*}*

1. YAGS

Its improvement implementation is in the same logic with gshare, except for we just add the “times” structure. Unfortunately, we fail to improve the performance as we expected.

**Part 4: Questions**

Answer the following questions.

**1. On the basis of your analysis, which predictor has the highest prediction rate? And, why?**

A：Tournament has the highest prediction rate. Because it uses two predictors, one is based on global history, another is based on local history, and a meta-predictor M indexed by the low-order bits of the branch address, which makes prediction of which predictor to be used. The combination of two predictors makes performance better with better flexibility.

**2. When is gshare more accurate than bimodal and why? Also, when is bimodal more accurate than gshare and why?**

A：When the branch address has more aliasing, gshare is more accurate because of hashing PC and BHSR to reduce aliasing, and if not, bimodal is more accurate.

**3. How does the length of history bits in gshare predictor affect the accuracy? Does increasing the length of history bits improve accuracy?**

A: If we get longer history bits, we can get more history information, so that it can improve the predictor’s accuracy. However, if the history bits become too long, the accuracy would not be improved significantly because of the unused space in PHT .

**4. How does Yags predictor reduce destructive aliasing?**

A: Because some branches are predominantly biased towards one direction. Yags utilizes a choice predictor to first indicate the direction of the branch, and then to consult one of the two PHTs. The branches that have a strong taken bias are placed in one PHT, and the branches that have a strong not-taken bias are separate into the other PHT, thus reducing destructive aliasing.

**5. In Yags predictor, the size of the direction caches is tuned according to the overall size of the predictor. Small predictor sizes need small direction caches and large predictor sizes need large direction caches. Is the statement true or false? Also, explain the reason behind your answer.**

A: It is false. For Yags, we make most of branch prediction by choice predictor, and only use caches to predict when there is an exception. So the size of caches will not affect the performance significantly.

**Part 5: Feedback**

1. We run our codes in virtual machine as well as in linux system, and it seems that predictor’s performance is better when we use virtual machine.
2. For Gshare predictor, its performance would be improved about 0.3% if we use 8-bit tag instead of 6-bit tag.
3. We try to improve the performance of Gshare and YAGS by using set-associate strategy, and implement the search () function in our code. But the results are not satisfying as we wished.