Project 2 Report

Introduction

This projector is to implement a different kind of branch predictor and simulate it in Gem5. 3 kinds of predictor, alloyed predictor, 2-level adaptive predictor, G-share predictors are provided.

Predictor Chosen for Implementation

In this project, 2-level adaptive branch predictor is chosen. This kind of predictor was proposed in 1991, by Yeh & Patt, in the paper named “Two-Level Adaptive Training Branch Prediction”. The 2 levels are history register table, pattern table, respectively [In Chapter 2]. It effectively increases branch predicting correction rate. In the paper introducing this predictor, the history register table can be implemented as hash table or set-associative cache [In Section 3.1]. And in this project, it is implemented as set-associative cache, which is described to provide better performance in section 3.1.

Workloads from SPECCPU 2006

There are 5 different configurations in the project, and each of them will run with 2 kinds of benchmarks, perlbench and mcf.

The 2 benchmarks I choose are perlbench and mcf.

Perlbench is a benchmark testing cut-down version of Perl language. Since it is an essential part for most Linux systems, and also because this benchmark tells the result very soon, this is the first benchmark for each configuration.

Mcf is a benchmark derived from a program for public mass transportation single-depot vehicle scheduling. This test mainly concern with integer arithmetic operations. It reflects the CPU performance about integer calculation for network flow problem.

Experimental Setup

2-level Adaptive Branch Predictor

Describing the CPU Configuration

This predictor has 5 parameters needs to configure.

For the history register table, its number of entries is determined by globalPredictorSize, usually it can be set to 256 or 512. Another parameter to configure it is globalCtrBits, which can be set to value between 6 and 12.

As for the pattern table, number of entries, which is power of 2 and number of bits in history register, should be defined. Also, number of bits in a saturate counter is defined by localCtrBits, which is usually set to 2.

Since it is implemented as a set-associative cache, associativity should be denoted. It can be indicated by number of bits in one history register set dividing number of bits in one history register.

Configuration 1

First configuration, which is described in the paper first, is using a 512-entry, 4 way-set-associative 12-bit history register table. And using a pattern table contains 4096 entries with 2-bit saturate counter in each entry.

Configuration 2

AHRT has 512 entries, which are 6 way-set-associative 12-bit history registers. The pattern table contains 4096 entries with 2-bit saturate counter in each.

Configuration 3

AHRT has 512 entries, 8 way-set-associative 12-bit history registers. Pattern table has 4096 entries with 2-bit counters inside.

Configuration 4

AHRT has 256 entries, 4 way-set-associative 12-bit registers. Pattern table has 4096 entries with 2-bit counters inside.

Configuration 5

AHRT has 512 entries, 4 way-set-associative 10-bit registers. Pattern table has 1024 entries with 2-bit counters inside.

Configuration 6

AHRT has 512 entries, 4 way-set-associative 8-bit registers. Pattern table has 256 entries with 2-bit counters inside.

Size of Predictor Storage

The predictor is consisted by 2 parts, history register table and pattern table. Size of history register

table equals to the product of history register bits and number of history registers. Number of history registers is the number of entries timing set-associativity. Size of pattern table can be calculated by entries multiplying counter bits. Since the history register table is implemented as set- associative cache, size of tags should also be considered. The total size is their sum.

In this project, we assume we are using a 32-bit address machine with fixed-size instruction of 4 bytes each, for calculation of tag.

In configuration 1, the size of 2 tables is 512×4×12+4096×2=32768 bits. Tags take 512×4×20= 40960 bits. Considering tags, total size is 32768+40960=73728 bits.

In configuration 2, the size of 2 tables is 512×6×12+4096×2=45056 bits. Tags take 512×6×20= 61440 bits. Considering tags, total size is 45056+61440=106496 bits.

In configuration 3, the size of 2 tables is 512×8×12+4096×2=57344 bits. Tags take 512×8×20= 81920 bits. Considering tags, total size is 57344+81920=139264bits.

In configuration 4, the size of 2 tables is 256×4×12+4096×2=20480 bits. Tags take 256×4×20= 20480 bits. Considering tags, total size is 20480+20480=40960 bits.

In configuration 5, the size of 2 tables is 512×4×10+1024×2=22528 bits. Tags take 512×4×22= 45056 bits. Considering tags, total size is 22528+45056=67584 bits.

In configuration 6, the size of 2 tables is 512×4×8+1024×2=18432 bits. Tags take 512×4×24= 49152 bits. Considering tags, total size is 67584 bits.

How the Predictor Work

This predictor mainly has 2 parts, history register table and pattern table. Then prediction is divided into 2 parts: history table operation and pattern table operation.

In the first, predictor using branch address, the value in program counter, to access the history table. Since the table is implemented as a set-associative cache in this project, the procedure to look up the table is the same. In this case it is not necessary to calculate offset value. Looking up the table uses lower bits of address as table index, while higher bits are used as tag for comparison. In one line of the table, LRU replacement is implemented.

Value in one register indicates the latest several branch taking history, 1 as taken (T) while 0 as untaken (N), and number of bits can be designated, e.g. in a 6-bit history register, a branch has a history of NTTNNT is stored as 0b011001. In this project, when there is a new branch needs recording, branch predictor will always return taken (T).

The second part is to look up the pattern table. In this table, each pattern has its own 2-bit saturate counter, providing prediction for each corresponding pattern. Taking the example mentioned in the last paragraph, pattern result is 0b011001, or 25. This pattern result is used as index to access pattern table. Then the counter value in table entry 25, is used as prediction. If the counter has a value of 2 or 3, it is taken, otherwise it is not taken.

For updating 2 tables, a newly placed branch has a history of 1, since a new branch is always

predicted taken. A branch exists in the table is updated as usual, adding most recent result in the least significant bit and shift other bits to the left by 1. The pattern table will be updated by index of old history register value. While for a new branch, the latest result is an exception for table updating, since history of a new branch will always be 1, and the branches truly have a pattern of 1 will be disturbed.

Following chart is an example of few lines of a 512-entry 2-way set-associative 6-bit AHRT.

AHRT Entries	Way 0: Tag	Way 0: History	Way 1: Tag	Way 1: History
000000000	01101010	110011	01101111	001100
…				
011001100	01001111	010101	00100111	101010
…				
111111111	11000110	100100	11100110	011011

This is merely an example above. For reading this table, if there is a branch address has an index of 204, or 011001100 in binary, and the branch has a tag of 01001111, history value of 21, 010101 in binary, is returned to be used as index for pattern table. This history value indicates a history of NTNTNT, since 0 stands for N and 1 stands for T.

To update this table, for the same branch, taking an example of untaken, shift values to left by 1 and add recent history to least bit, new value is 101010, or 42.

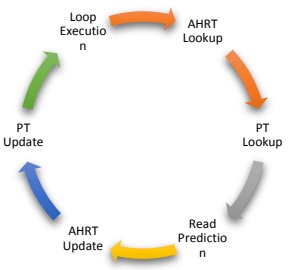
Following chart is an example of corresponding pattern table, which has 64 entries.

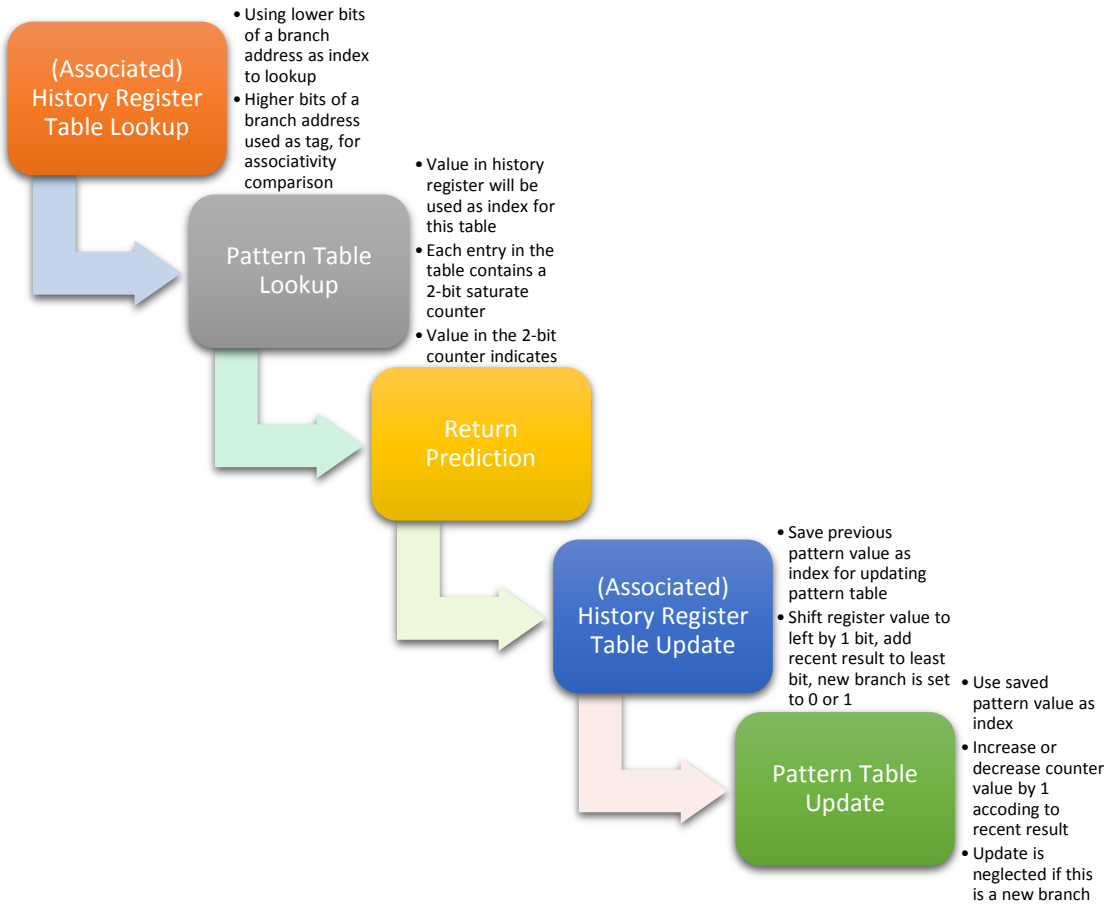
PT Entry	2-bit Counter Value
000000	00
…	
010101	00
…	
111111	11

In this table, I try to show a reasonable prediction. For the entry of 0, we can infer that corresponding history is NNNNNN, so prediction is likely to be 0. In our example, entry of 21 should have a pattern of 010101, so at this point predictor should be 0. And prediction of untaken is returned.

To update the counter, an old value in corresponding AHRT should be read as entry number of this table, then the table value should be increased or decreased by 1.

Graphs shows how it works as following.





How the Predictor Implemented in Gem5

The method of lookup is implemented as following:

```
106 unsigned int
107 ECEC621BP::AHRTread(unsigned int _index, unsigned int _tag)
108 {
109     if (_index >= AHRTs.size()) {
110         fatal("Out of AHRT bound!\n");
111     }
112     for (unsigned int i = 0; i < set_associativity; i++) {
113         if (AHRTs[_index].HRs[i].tag == _tag) {
114             return AHRTs[_index].HRs[i].HistoryPattern;
115         }
116     }
117     return std::numeric_limits<unsigned int>::max();
118 }
119
120 bool
121 ECEC621BP::PTread(unsigned int _pattern)
122 {
123     if (_pattern == std::numeric_limits<unsigned int>::max()) {
124         return true;
125     }
126     if (_pattern >= PT.size()) {
127         fatal("Out of PT bound!\n");
128     }
129     return (PT[_pattern].read()>>(localCtrBits-1));
130 }
131
132 bool
133 ECEC621BP::lookup(Addr branch_addr, void * &bp_history)
134 {
135     // implement the branch prediction lookup
136     unsigned int AHRTIndex;
137     unsigned int AHRTtag;
138     unsigned int pattern;
139     bool prediction;
140
141     branch_addr = branch_addr >> instShiftAmt;
142     // Last few bits is used as table index.
143     AHRTIndex = branch_addr & AHRTlookupMask;
144     // Creating tag.
145     AHRTtag = branch_addr >> AHRTindexLength;
146     // Read history pattern from AHRT.
147     pattern = AHRTread(AHRTIndex, AHRTtag);
148     // Read predictor value from PT.
149     prediction = PTread(pattern);
150
151     return prediction;
152 }
```

This method invokes 2 other methods called AHRTread and PTread. Codes are also shown in the screenshots.

In this method, index and tag for AHRT lookup are detached from branch address, between line 141 and 145. Then method of AHRTread is invoked to get history pattern value. In AHRTread method, since a line of AHRT is set-associative, tags needs to be compared. If there are no tag matched, which means the branch is new to the table, it returns a value that is impossible to be a history pattern. After pattern is received, PTread is called. This method looks counter value in pattern table and then return the prediction of the counter. If the history pattern value is the impossible value, which indicates this is a new branch, the method will always return true.

Update method is shown below.

```
154 unsigned int
155 ECEC621BP::AHRTupdate(unsigned int _index, unsigned int _tag, bool taken)
156 {
157     if (_index >= AHRTs.size()) {
158         fatal("Out of AHRT bound while updating!\n");
159     }
160     unsigned int lastpattern;
161     for (std::list<unsigned int>::iterator it = AHRTs[_index].LRUstack.begin(); it != AHRTs[_index].LRUstack.end(); ++it) {
162         if (AHRTs[_index].HRs[*it].tag == _tag) {
163             lastpattern = AHRTs[_index].HRs[*it].HistoryPattern;
164             if (taken) {
165                 AHRTs[_index].HRs[*it].HistoryPattern = ((AHRTs[_index].HRs[*it].HistoryPattern<<1)|1) & hp_mask;
166             } else {
167                 AHRTs[_index].HRs[*it].HistoryPattern = (AHRTs[_index].HRs[*it].HistoryPattern<<1) & hp_mask;
168             }
169             AHRTs[_index].LRUstack.push_front(*it);
170             it = AHRTs[_index].LRUstack.erase(it);
171             return lastpattern;
172         }
173     }
174     AHRTs[_index].HRs[AHRTs[_index].LRUstack.back()].tag = _tag;
175     lastpattern = std::numeric_limits<unsigned int>::max();
176     if (taken) {
177         AHRTs[_index].HRs[AHRTs[_index].LRUstack.back()].HistoryPattern = 1;
178     } else {
179         AHRTs[_index].HRs[AHRTs[_index].LRUstack.back()].HistoryPattern = 0;
180     }
181     AHRTs[_index].LRUstack.push_front(AHRTs[_index].LRUstack.back());
182     AHRTs[_index].LRUstack.pop_back();
183     return lastpattern;
184 }
185
186 void
187 ECEC621BP::update(Addr branch_addr, bool taken, void *bp_history, bool squashed)
188 {
189     // implement the branch prediction update
190     unsigned int AHRTIndex;
191     unsigned int AHRTtag;
192     unsigned int pattern;
193
194     branch_addr = branch_addr >> instShiftAmt;
195     // Last few bits is used as table index.
196     AHRTIndex = branch_addr & AHRTlookupMask;
197     // Creating tag.
198     AHRTtag = branch_addr >> AHRTindexLength;
199     // Read history pattern from AHRT, as table entry for PT, and update AHRT.
200     pattern = AHRTupdate(AHRTIndex, AHRTtag, taken);
201     if (pattern != std::numeric_limits<unsigned int>::max()) {
202         if (taken) {
203             DPRINTF(Fetch, "Branch updated as taken.\n");
204             if (pattern >= PT.size()) {
205                 fatal("Out of PT bound, updating as taken!\n");
206             }
207             PT[pattern].increment();
208         } else {
209             DPRINTF(Fetch, "Branch updated as not taken.\n");
210             if (pattern >= PT.size()) {
211                 fatal("Out of PT bound, updating as not taken!\n");
212             }
213             PT[pattern].decrement();
214         }
215     }
216 }
```

This method invokes another method called AHRTupdate. Index and tags are detached the same as lookup method. AHRT table is updated firstly. As for AHRT updating, history pattern is shifted left and latest history is added. If this is a new branch, set history register as 0 or 1. LRU information is also updated at the last, line 169 and 170 for existing branches, line 181 and 182 for new branches.

The if statement in line 201 checks if this is a new branch, and if this is a new branch, update for pattern table is ignored. If this is not a new branch, corresponding counter in pattern table is increased or decreased by 1 according to last correct branch result.

Results and Analysis

Configuration 1

Perlbench

IPC (system.cpu.ipc): 0.567185

Execution cycles (system.cpu.numCycles): 3713621

Branch rate (system.cpu.fetch.branchRate): 0.212651

Branch mis-prediction rate (system.cpu.branchPred.condPredicted ÷ system.cpu.branchPred.condIncorrect): 9.35%

Fetch-squashed (system.cpu.fetch.SquachCycles ÷ system.cpu.fetch.Cycles): 29.84%

MCF

IPC (system.cpu.ipc): 1.339624

Execution cycles (system.cpu.numCycles): 186619487

Branch rate (system.cpu.fetch.branchRate): 0.377629

Branch mis-prediction rate (system.cpu.branchPred.condPredicted÷system.cpu.branchPred.condIncorrect): 2.06%

Fetch-squashed (system.cpu.fetch.SquachCycles÷system.cpu.fetch.Cycles): 8.31%

Configuration 2

Perlbench

IPC (system.cpu.ipc): 0.569510

Execution cycles (system.cpu.numCycles): 3698459

Branch rate (system.cpu.fetch.branchRate): 0.212667

Branch mis-prediction rate (system.cpu.branchPred.condPredicted ÷ system.cpu.branchPred.condIncorrect): 9.22%

Fetch-squashed (system.cpu.fetch.SquachCycles ÷ system.cpu.fetch.Cycles): 29.64%

MCF

IPC (system.cpu.ipc): 1.339624

Execution cycles (system.cpu.numCycles): 186619487

Branch rate (system.cpu.fetch.branchRate): 0.377629

Branch mis-prediction rate (system.cpu.branchPred.condPredicted÷system.cpu.branchPred.cond Incorrect): 2.06%

Fetch-squashed (system.cpu.fetch.SquachCycles÷system.cpu.fetch.Cycles): 8.31%

Configuration 3

Perlbench

IPC (system.cpu.ipc): 0.571690

Execution cycles (system.cpu.numCycles): 3684360

Branch rate (system.cpu.fetch.branchRate): 0.212788

Branch mis-prediction rate (system.cpu.branchPred.condPredicted ÷ system.cpu.branchPred.cond Incorrect): 9.13%

Fetch-squashed (system.cpu.fetch.SquachCycles ÷ system.cpu.fetch.Cycles): 29.46%

MCF

IPC (system.cpu.ipc): 1.339624

Execution cycles (system.cpu.numCycles): 186619487

Branch rate (system.cpu.fetch.branchRate): 0.377629

Branch mis-prediction rate (system.cpu.branchPred.condPredicted÷system.cpu.branchPred.cond Incorrect): 2.06%

Fetch-squashed (system.cpu.fetch.SquachCycles÷system.cpu.fetch.Cycles): 8.31%

Configuration 4

Perlbench

IPC (system.cpu.ipc): 0.561147

Execution cycles (system.cpu.numCycles): 3753577

Branch rate (system.cpu.fetch.branchRate): 0.212010

Branch mis-prediction rate (system.cpu.branchPred.condPredicted ÷ system.cpu.branchPred.cond Incorrect): 9.66%

Fetch-squashed (system.cpu.fetch.SquachCycles ÷ system.cpu.fetch.Cycles): 30.34%

MCF

IPC (system.cpu.ipc): 1.339298

Execution cycles (system.cpu.numCycles): 186664887

Branch rate (system.cpu.fetch.branchRate): 0.377777

Branch mis-prediction rate (system.cpu.branchPred.condPredicted÷system.cpu.branchPred.cond Incorrect): 2.06%

Fetch-squashed (system.cpu.fetch.SquachCycles÷system.cpu.fetch.Cycles): 8.33%

Configuration 5

Perlbench

IPC (system.cpu.ipc): 0.564851

Execution cycles (system.cpu.numCycles): 3728963

Branch rate (system.cpu.fetch.branchRate): 0.212229

Branch mis-prediction rate (system.cpu.branchPred.condPredicted ÷ system.cpu.branchPred.cond Incorrect): 9.43%

Fetch-squashed (system.cpu.fetch.SquachCycles ÷ system.cpu.fetch.Cycles): 29.97%

MCF

IPC (system.cpu.ipc): 1.323240

Execution cycles (system.cpu.numCycles): 188930152

Branch rate (system.cpu.fetch.branchRate): 0.375635

Branch mis-prediction rate (system.cpu.branchPred.condPredicted÷system.cpu.branchPred.cond Incorrect): 2.26%

Fetch-squashed (system.cpu.fetch.SquachCycles÷system.cpu.fetch.Cycles): 8.93%

Configuration 6

Perlbench

IPC (system.cpu.ipc): 0.561694

Execution cycles (system.cpu.numCycles): 3749923

Branch rate (system.cpu.fetch.branchRate): 0.212833

Branch mis-prediction rate (system.cpu.branchPred.condPredicted ÷ system.cpu.branchPred.cond Incorrect): 9.54%

Fetch-squashed (system.cpu.fetch.SquachCycles ÷ system.cpu.fetch.Cycles): 30.33%

MCF

IPC (system.cpu.ipc): 1.313273

Execution cycles (system.cpu.numCycles): 190364067

Branch rate (system.cpu.fetch.branchRate): 0.374328

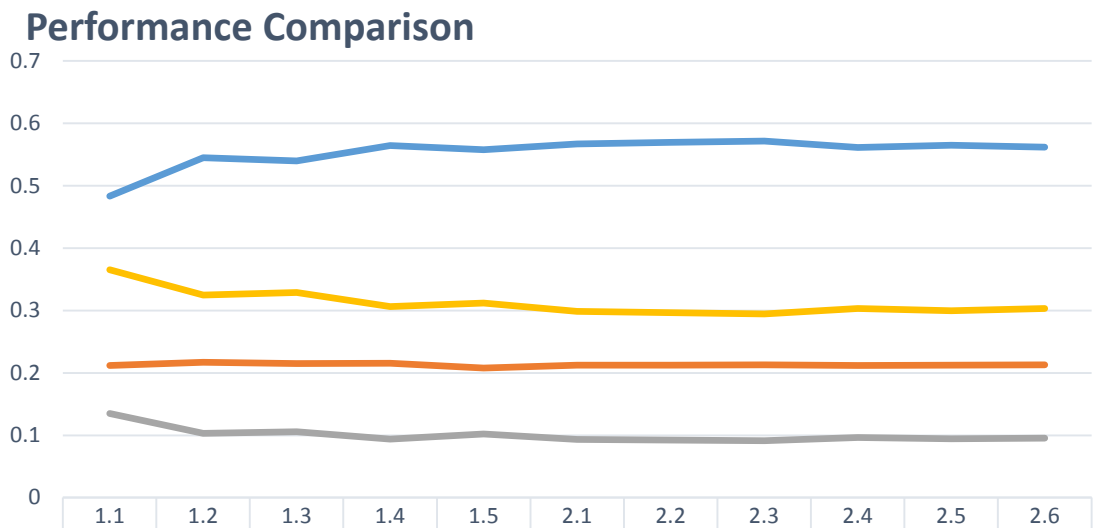
Branch mis-prediction rate (system.cpu.branchPred.condPredicted ÷ system.cpu.branchPred.cond Incorrect): 2.34%

Fetch-squashed (system.cpu.fetch.SquachCycles ÷ system.cpu.fetch.Cycles): 9.30%

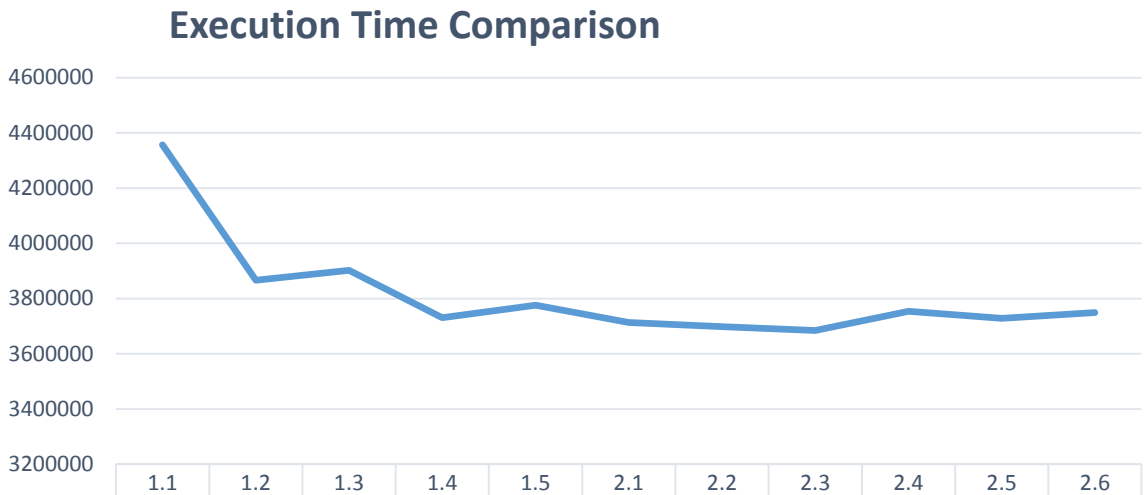
Performance Comparison

Perlbench Result Table

Results in Project 1 are denoted as 1.x, while results of configuration 1 to 6 are denoted as 2.x.



	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	2.5	2.6
IPC (Higher is Better)	0.48351	0.54481	0.53979	0.56462	0.55785	0.56719	0.56951	0.57169	0.56115	0.56485	0.56169
Branch Rate (Higher is Better)	0.21201	0.21682	0.21519	0.2154	0.20779	0.21265	0.21267	0.21279	0.21201	0.21223	0.21283
Branch Mis-Perdiction Rate (Lower is Better)	0.135	0.103	0.1055	0.0941	0.1023	0.0935	0.0922	0.0913	0.0966	0.0943	0.0954
Fetch-Squashed Rate (Lower is Better)	0.3656	0.325	0.3289	0.3061	0.3118	0.2984	0.2964	0.2946	0.3034	0.2997	0.3033



	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	2.5	2.6
Execution Cycles (Lower is Better)	4356253	3866138	3902115	3730466	3775761	3713621	3698459	3684360	3753577	3728963	3749923

Perl Result Analysis

As we can see learn from the graphs, for perlbench, 2-level adaptive has better performance than others. It can be assumed that, in the perspective of executing script language, 2-level adaptive predictor is good at handling the situation where there are a bunch of branches.

In configuration 4, 1, 2, and 3 , the capacity of AHRT is increasing: 256×4, 512×4, 512×6, 512×8, respectively. As shown from the graphs, when capacity of AHRT grows larger, performance gets better, e.g. in term of execution time, configuration 3 is the quickest, 0.38% quicker than 2, 0.41% quicker than 1, and 1.84% quicker than 4.

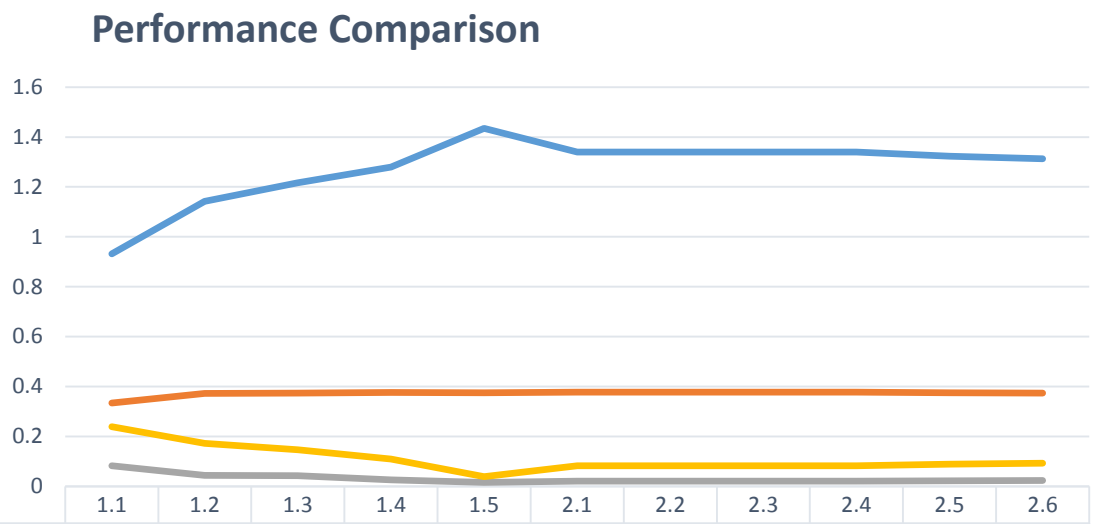
It can also be inferred from configuration 4 and 1 that an increase in AHRT entries improves performance of predictor, so does increasing set-associativity of AHRT that revealed from configuration 1, 2, and 3.

In configuration 6, 5, and 1, number of bits in a history register, or number of entries in a PT, is growing from 8 to 12. Comparing data from the 3 configurations, it can be concluded that more bits in a register leads to a higher performance. E.g. in term of branch mis-prediction rate, configuration 1 has the lowest miss rate of 9.35%, while 5 has a rate 0.08% higher and 6 has a rate 0.19% higher.

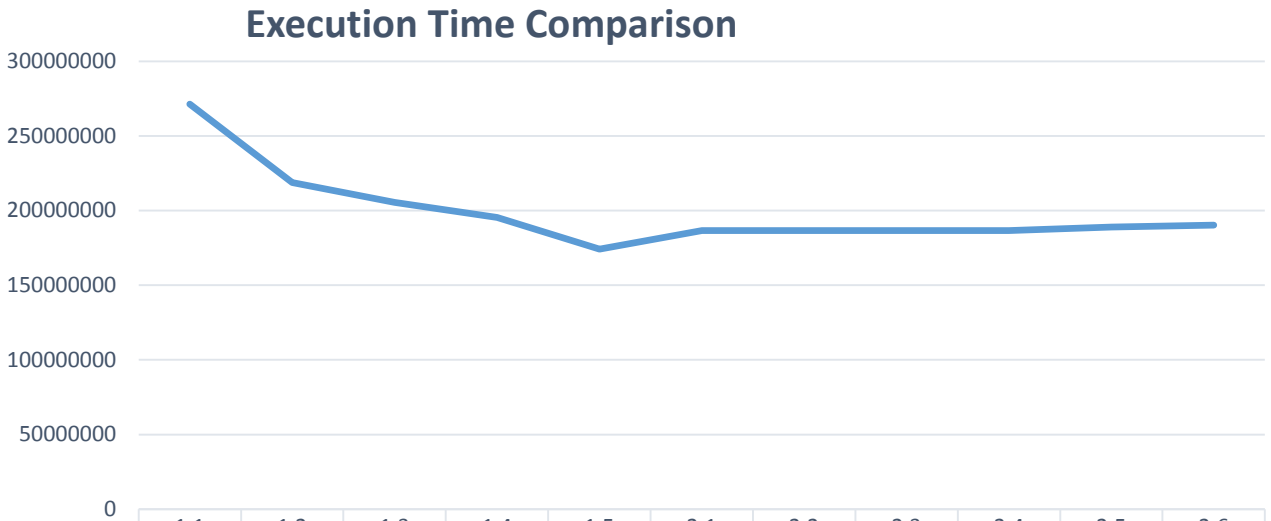
2-level adaptive predictor is good at executing script language, for a predictor which has 512-entry 4-way-associative AHRT with 4096 PT, it has higher IPC and branch rate, and lower branch mis-prediction rate, fetch-squash rate and execution time. All of these results mentioned above show a higher performance. This configuration is superior to any simulation results in project 1.

In term of predictor size, configuration 1, if tags are ignored, is of 32768 bits. Tags can take 40960 bits, and tags may not always be necessary to take that many bits. Configuration 5, just change history registers to 10-bit ones, takes less bits and acceptable performance. In general, this is a better choice for script language execution.

MCF Result Table



	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	2.5	2.6
IPC (Higher is Better)	0.9315	1.14271	1.21666	1.27979	1.43501	1.33962	1.33962	1.33962	1.3393	1.32324	1.31327
Branch Rate (Higher is Better)	0.33436	0.37301	0.37426	0.37581	0.37572	0.37763	0.37763	0.37763	0.37778	0.37564	0.37433
Branch Mis-Perdiction Rate (Lower is Better)	0.0822	0.0448	0.0425	0.0267	0.016	0.0206	0.0206	0.0206	0.0206	0.0226	0.0234
Fetch-Squashed Rate (Lower is Better)	0.2388	0.1728	0.147	0.1096	0.0385	0.0831	0.0831	0.0831	0.0833	0.0893	0.093



	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	2.5	2.6
Execution Cycles (Lower is Better)	2.71E+08	2.19E+08	2.05E+08	1.95E+08	1.74E+08	1.87E+08	1.87E+08	1.87E+08	1.87E+08	1.89E+08	1.9E+08

MCF Result Analysis

In term of MCF, which is an integer operation intensive benchmark, 2-level adaptive predictor is better than static and local predictors in project 1, among which the highest IPC is 1.27979, while it still cannot beat the performance of tournament predictor in project 1, whose IPC is 1.43501.

The trend of project 2 predictor results for MCF is similar to that of perlbench. Increasing capacity of AHRT, either adding new entries or increasing set-associativity, can improve performance. While as capacity grows larger than 512×4, there is no performance improve.

While configuration 4 has a slightly higher branch rate, however we can see that other results of it is worse the configuration 1, 2, and 3, so we do not regards this is a sign that 256×4-capacity has a better performance than 512×4 or more ones.

It is possible that 512×4 branches is the most of MCF need to use. In an integer operation intensive task, there may not be so many different branches in this program, so in this benchmark simply increasing AHRT capacity may not be effective. It is also possible that in arithmetic calculations, there may not be so many different history patterns. While tournament predictor has both local and global prediction and choice predictor, the recognition of local and global history pattern may be more important for this benchmark.

As for comparison between configuration 6, 5, and 1, performance is improving when we add more bits into history registers, with enlarging pattern table size.

In summary, this predictor may not be perfect for MCF, however it is still a better choice than local predictors. Generally, for a script intensive user, this predictor is a good choice, and for integer operation intensive tasks, it is still not a bad choice.

Conclusion

In conclusion, this predictor is better for task which has huge number of different branches such like script execution, language compiling, loading webpages, etc. It may not fit perfectly for integer operation intensive tasks since these tasks may either not have that many branches or not have that many history patterns. In average, this is a good predictor worth implementing.

This predictor can provide mis-prediction rate less than 4% in MCF and less than 10.5% in perlbench, not considering tags, predictor size is remarkable. There is also a way to avoid tag storage, using Hashing History Register Table can save from storing tags [In Section 3.1]. It is also mentioned in that section that, using HHRT may affect performance, making it lower than those with AHRT. In this project, AHRT is used to achieve higher performance.