

2-分支预测

原理

3.2 预测分析

有一些文法使用一种称为递归下降（recursive descent）的简单算法就很容易进行分析。这种算法的实质是将每一个文法产生式转变成递归函数中的一个子句。为了举例说明这种算法，我们来为文法 3-5 写一个递归下降语法分析器。

文法 3-5

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$	$L \rightarrow \text{end}$
$S \rightarrow \text{begin } S L$	$L \rightarrow ; S L$
$S \rightarrow \text{print } E$	$E \rightarrow \text{num} = \text{num}$

这个语言的递归下降语法分析器对每个非终结符有一个函数，非终结符的每个产生式对应一个子句。

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
    case IF:      eat(IF); E(); eat(THEN); S();
                  eat(ELSE); S(); break;
    case BEGIN:  eat(BEGIN); S(); L(); break;
    case PRINT:  eat(PRINT); E(); break;
    default:     error();
}}

void L(void) {switch(tok) {
    case END:    eat(END); break;
    case SEMI:   eat(SEMI); S(); L(); break;
    default:     error();
}}

void E(void) { eat(NUM); eat(EQ); eat(NUM); }
```

若恰当地定义了 error 和 getToken，这个程序就能很好地对文法 3-5 进行分析。
这种简单方法的成功给了我们一种鼓励，让我们再用它来尝试文法 3-4：

其实就是设计算法提前猜测if后的条件是true 还是 false，如果提前猜对了，那就提前执行了需要执行的内容，就加速

算法

Taken predictor 全部认为true

Predict all addresses are taken.

Not Taken predictor 全部认为false

Predict all addresses are not taken.

One-bit predictor 记录最后的结果作为下次的预测

It only record the last bit of result and do the same prediction as last do.

Backward taken forward taken 记录最后的记过作为下次的反预测

It do the taken if it was taken and do the not taken if it was not taken.

Bimodal predictor[1] 4状态变换的状态机预测

A bimodal predictor is a state machine with four states: Strongly not taken, Weakly not taken, Weakly taken, Strongly taken.

When a branch command is evaluated, the corresponding state machine is modified. If the branch is not adopted, the state value is decreased in the direction of "strong no selection"; if the branch is adopted, the state value is increased in the direction of "strong selection". The advantage of this method is that the conditional branch instruction must select a certain branch twice in succession to flip from the strong state, thereby changing the predicted branch.

Delayed predictor 延迟几个位子的历史记录结果的预测

Similar to taken/not taken predictor, but prediction is determined by result several bit before.

History-based predictor 记录上次该地址的结果作为下次相同地址的预测

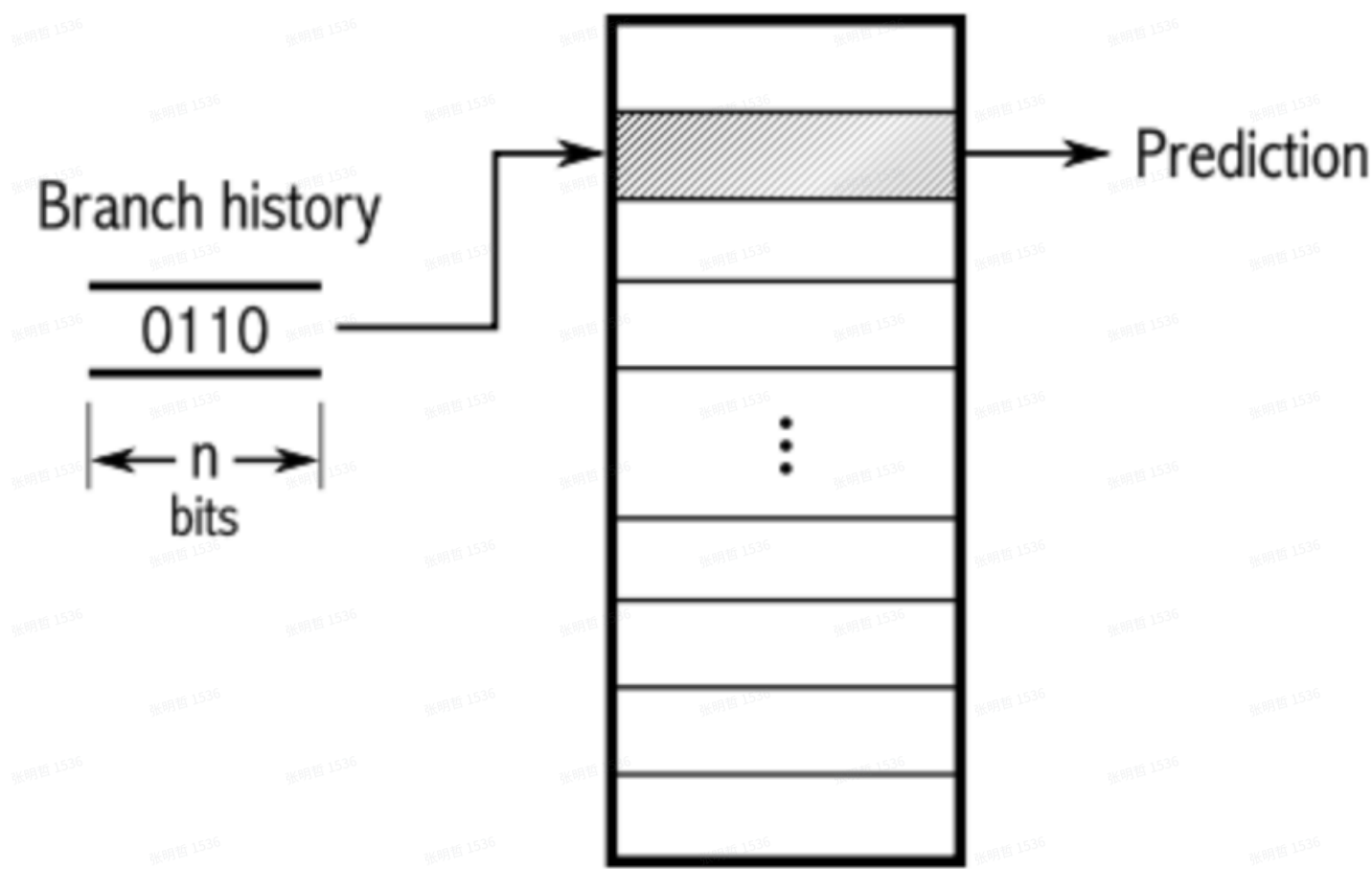
If this address has been done with taken, predict it taken again or predict it not taken.

Two level adaptive predictor[2] 根据前面多次结果的历史作为下次的预测

During the 1990s Two-level Adaptive Branch Predictors were developed to meet the requirement for accurate branch prediction in high-performance superscalar processors.

If an if statement is executed three times, the decision made on the third execution might depend upon whether the previous two were taken or not. In such scenarios, a two-level adaptive predictor works more efficiently than a saturation counter. Conditional jumps that are taken every second time or have some other regularly recurring pattern are not predicted well by the saturating counter. A two-level adaptive predictor remembers the history of the last n occurrences of the branch and uses one saturating counter for each of the possible 2^n history patterns.

Pattern history table



练习题

Write a simple simulator that models the 2 predictors and processes the instruction trace provided. Submit both the

code for your branch predictor simulator and report on the number of buffer misses (first time taken

branches), and the number of correct and incorrect predictions for this address trace for each predictor. Add a discussion explaining why one predictor does better than the other?

尝试自己写一种分支预测器

答案

这里是之前我做的一个模拟预测器，多个预测器同时跑，采用当前正确率最高的，具体说明可以看

https://github.com/mingzheTerapines/modern_comipler_learning/tree/main/%E8%AF%AD%E6%B3%95%E5%88%86%E6%9E%90/%E5%88%86%E6%94%AF%E9%A2%84%E6%B5%8B%E5%99%A8%E5%AE%9E%E7%8E%B0

```

1 #include <iostream>           // std::cout
2 #include <thread>             // std::thread
3 #include <mutex>              // std::mutex
4 #include <vector>             // std::vector
5 #include <string>             // std::string
6 #include <set>                // std::set
7 #include <fstream>           // std::fstream
8 #include <algorithm>         // std::algorithm
9 #include <unistd.h>          // unistd.h
10 //to compile: use command g++ swither_predictor.cpp -o swither_predictor.out
11 //to run: ./swither_predictor.out
12 using namespace std;
13 volatile int counter(0); // non-atomic counter
14 std::mutex mtx;          // locks access to counter
15 vector<bool> jump(0);    //the list of jump history
16 vector<long long> address(0);
17 void readfile(){
18     ifstream fin("itrace.out");
19     string str="";
20     string::size_type sz;
21     sz = 0;
22     long long tmp;
23     while(getline(fin,str)){
24         if(str!="#eof"){
25             tmp=stoll(str,&sz,0);
26             address.push_back(tmp);
27         }
28     }
29
30     tmp=address[0];
31     long long diff;
32     for(int i=1;i<address.size();i++){
33         diff=address[i]-tmp;
34         if(diff<0||diff>15){
35             jump.push_back(true);
36         }
37         else{
38             jump.push_back(false);
39         }
40         tmp=address[i];
41     }
42 }
43 vector<float> taken_corr(0); //the correction of taken predictor
44 void taken_predictor(){//predict taken
45     long taken_cnt(0);
46     for(int i=0;i<jump.size();i++) {
47         if (jump[i]) {

```

```

48         taken_cnt++;
49     }
50     taken_corr.push_back(((float) taken_cnt / (float) i)*100);
51 }
52 while(true){
53     if (mtx.try_lock()) {    // counter finished
54         ++counter;
55         mtx.unlock();
56         break;
57     }
58 }
59 }
60 vector<float> not_taken_corr(0);    //the correction of taken predictor
61 void not_taken_predictor(){
62     long not_taken_cnt(0);
63     for(int i=0;i<jump.size();i++){
64         if(!jump[i]){
65             not_taken_cnt++;
66         }
67         not_taken_corr.push_back((float)not_taken_cnt/(float)i*100);
68     }
69     while(true){
70         if (mtx.try_lock()) {    // counter finished
71             ++counter;
72             mtx.unlock();
73             break;
74         }
75     }
76 }
77 vector<float> backward_taken_forward_not_taken_corr(0);
78 vector<bool> backward_taken_forward_not_taken_predict(0);
79 void backward_taken_forward_not_taken_predictor(){
80     long cnt=0;
81     bool predict(false);
82     for(int i=0;i<jump.size();i++){
83         backward_taken_forward_not_taken_predict.push_back(predict);
84         if(jump[i]==predict)
85             cnt++;
86         predict=!jump[i];
87     }
88     backward_taken_forward_not_taken_corr.push_back((float)cnt/(float)i*100);
89 }
90 while(true){
91     if (mtx.try_lock()) {    // counter finished
92         ++counter;
93         mtx.unlock();
94         break;

```

```

94     }
95 }
96 }
97 vector<float> one_bit_corr(0);
98 vector<bool> one_bit_predict(0);
99 void one_bit_predictor(){
100     long cnt=0;
101     bool predict(false);
102     for(int i=0;i<jump.size();i++){
103         one_bit_predict.push_back(predict);
104         if(predict==jump[i])
105             cnt++;
106         predict=jump[i];
107         one_bit_corr.push_back((float)cnt/(float)i*100);
108     }
109     while(true){
110         if (mtx.try_lock()) {    // counter finished
111             ++counter;
112             mtx.unlock();
113             break;
114         }
115     }
116 }
117 enum class binarystatus{
118     stronglynot,//strongly not taken
119     weaklynot,//weakly not taken
120     weaklytaken,//weakly taken
121     stronglytaken,//strongly taken
122 };
123 class bipredictor{
124 public:
125     binarystatus bst;
126     bipredictor(){
127         bst=binarystatus::stronglynot;
128     }
129     void iftaken(bool tk){
130         if(tk){
131             switch(bst){
132                 case binarystatus::stronglynot:
133                     bst=binarystatus::weaklynot;
134                     break;
135                 case binarystatus::weaklynot:
136                     bst=binarystatus::weaklytaken;
137                     break;
138                 case binarystatus::weaklytaken:
139                     bst=binarystatus::stronglytaken;
140                     break;

```

```

141         case binarystatus::stronglytaken:
142             break;
143     }
144 }else{
145     switch(bst){
146         case binarystatus::stronglynot:
147             break;
148         case binarystatus::weaklynot:
149             bst=binarystatus::stronglynot;
150             break;
151         case binarystatus::weaklytaken:
152             bst=binarystatus::weaklynot;
153             break;
154         case binarystatus::stronglytaken:
155             bst=binarystatus::weaklytaken;
156             break;
157     }
158 }
159 }
160 };
161 vector<float> bimodal_corr(0);
162 vector<bool> bimodal_predict(0);
163 void bimodal_predictor(){
164     auto *bpd=new bipredictor;
165     long cnt=0;
166     bool predict(false);
167     for(int i=0;i<jump.size();i++){
168         bimodal_predict.push_back(predict);
169         if(predict==jump[i])
170             cnt++;
171         if(bpd->bst==binarystatus::stronglynot||bpd-
>bst==binarystatus::weaklynot){
172             predict=false;
173         }else
174             predict=true;
175         bpd->iftaken(jump[i]);
176         bimodal_corr.push_back((float)cnt/(float )i*100);
177     }
178     while(true){
179         if (mtx.try_lock()) {    // counter finished
180             ++counter;
181             mtx.unlock();
182             break;
183         }
184     }
185 }
186 vector<float> two_level_adaptive_corr(0);

```



```

187 vector<bool> two_level_adaptive_predict(0);
188 void handle(bipredictor *bpd,bool &predict,long &cnt,int i){
189     if(bpd->bst==binarystatus::stronglynot||bpd->bst==binarystatus::weaklynot){
190         predict=false;
191     }else
192         predict=true;
193     if(predict==jump[i])
194         cnt++;
195     bpd->iftaken(jump[i]);
196 }
197 void two_level_adaptive_predictor(){
198     auto *bpd00=new bipredictor;
199     auto *bpd01=new bipredictor;
200     auto *bpd10=new bipredictor;
201     auto *bpd11=new bipredictor;
202     bool predict(false);
203     bool first(jump[0]);
204     bool second(jump[1]);
205     long cnt=2;
206     two_level_adaptive_predict.push_back(first);
207     two_level_adaptive_predict.push_back(second);
208     two_level_adaptive_corr.push_back(100);
209     two_level_adaptive_corr.push_back(100);
210     for(int i=2;i<jump.size();i++){
211         if(!first&&!second){
212             handle(bpd00,predict,cnt,i);
213         }else if(!first && second){
214             handle(bpd01,predict,cnt,i);
215         }else if(first && !second){
216             handle(bpd10,predict,cnt,i);
217         }else if(first&&second){
218             handle(bpd11,predict,cnt,i);
219         }
220         first=jump[i-1];
221         second=jump[i];
222         two_level_adaptive_predict.push_back(predict);
223         two_level_adaptive_corr.push_back((float)cnt/(float)i*100);
224     }
225     while(true){
226         if (mtx.try_lock()) { // counter finished
227             ++counter;
228             mtx.unlock();
229             break;
230         }
231     }
232 }
233 vector<float> delayed_corr(0);

```

```

234 vector<bool> delayed_predict(0);
235 void delayed_predictor(){
236     long cnt=2;
237     delayed_corr.push_back(100);
238     delayed_corr.push_back(100);
239     delayed_predict.push_back(false);
240     delayed_predict.push_back(false);
241     for(int i=2;i<jump.size();i++){
242         if(jump[i]==delayed_predict[i-2])
243             cnt++;
244         delayed_predict.push_back(jump[i]);
245         delayed_corr.push_back((float)cnt/(float)i*100);
246     }
247     while(true){
248         if (mtx.try_lock()) { // counter finished
249             ++counter;
250             mtx.unlock();
251             break;
252         }
253     }
254 }
255 vector<float> history_based_corr(0);
256 vector<bool> history_based_predict(0);
257 set<long long> jumped;
258 void history_based_predictor(){
259     float cnt(1);
260     bool prediction(false);
261     for(int i=1;i<address.size();i++){
262         history_based_corr.push_back(prediction);
263         if(prediction==jump[i-1])
264             cnt++;
265         if(jumped.count(address[i])){
266             prediction=true;
267         }else
268             prediction=false;
269         if(jump[i-1])
270             jumped.insert(address[i-1]);
271         history_based_corr.push_back((float)cnt/(float)i);
272     }
273     while(true){
274         if (mtx.try_lock()) { // counter finished
275             ++counter;
276             mtx.unlock();
277             break;
278         }
279     }
280 }

```

```

281 void do_prediction(bool &predict,int i){
282     float maxvalue=max(taken_corr[i],not_taken_corr[i]);
283     maxvalue=max(maxvalue,backward_taken_forward_not_taken_corr[i]);
284     maxvalue=max(maxvalue,one_bit_corr[i]);
285     maxvalue=max(maxvalue,bimodal_corr[i]);
286     maxvalue=max(maxvalue,two_level_adaptive_corr[i]);
287     maxvalue=max(maxvalue,delayed_corr[i]);
288     maxvalue=max(maxvalue,history_based_corr[i]);
289     if(maxvalue==taken_corr[i]) {
290         predict = true;
291         return;
292     }
293     if(maxvalue==not_taken_corr[i]){
294         predict=false;
295         return;
296     }
297     if(maxvalue==backward_taken_forward_not_taken_corr[i]){
298         predict=backward_taken_forward_not_taken_predict[i];
299         return ;
300     }
301     if(maxvalue==one_bit_corr[i]){
302         predict=one_bit_predict[i];
303         return ;
304     }
305     if(maxvalue==bimodal_corr[i]){
306         predict=bimodal_predict[i];
307         return ;
308     }
309     if(maxvalue==two_level_adaptive_corr[i]){
310         predict=two_level_adaptive_predict[i];
311         return ;
312     }
313     if(maxvalue==delayed_corr[i]){
314         predict=delayed_predict[i];
315         return ;
316     }
317     if(maxvalue==history_based_corr[i]){
318         predict=history_based_predict[i];
319         return ;
320     }
321 }
322 void do_reduction(){
323     bool predict(false);
324     long cnt(0);
325     for(int i=0;i<jump.size();i++){
326         if(predict==jump[i])
327             cnt++;

```

```

328         do_prediction(predict,i);
329     }
330     cout<<"prediction correction: "<<(float)cnt/(float)jump.size()*100<<endl;
331 }
332 void reduction(){
333     while(true){
334         if (mtx.try_lock()) { //read counter
335             if (counter == 8) { //counter finished
336                 do_reduction();
337                 mtx.unlock();
338                 break;
339             } else {
340                 mtx.unlock();
341                 sleep(5); //wait 5s for counter finished
342             }
343         } else
344             sleep(5);
345     }
346 }
347 int main () {
348     readfile();
349     std::thread threads[9];
350     threads[0]=thread(reduction);
351     threads[1]=thread(taken_predictor);
352     threads[2]=thread(not_taken_predictor);
353     threads[3]=thread(backward_taken_forward_not_taken_predictor);
354     threads[4]=thread(one_bit_predictor);
355     threads[5]=thread(bimodal_predictor);
356     threads[6]=thread(two_level_adaptive_predictor);
357     threads[7]=thread(delayed_predictor);
358     threads[8]=thread(history_based_predictor);
359     for (auto& th : threads) th.join();
360     return 0;
361 }
362

```

refer

[1]"Dynamic Branch Prediction",

(http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/). *web.engr.oregonstate.edu*.

Retrieved 2017-11-01

[2]C. Egan, G. B. Steven, Won Shim and L. Vintan, "Applying caching to two-level adaptive branch prediction," Proceedings Euromicro Symposium on Digital Systems Design, Warsaw, 2001, pp.

