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

$S \to$ if E then S else S	$L \to$ end
$S \to$ begin S L	$L \to$; S L
$S \to$ print E	
	$E \to$ num = 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.

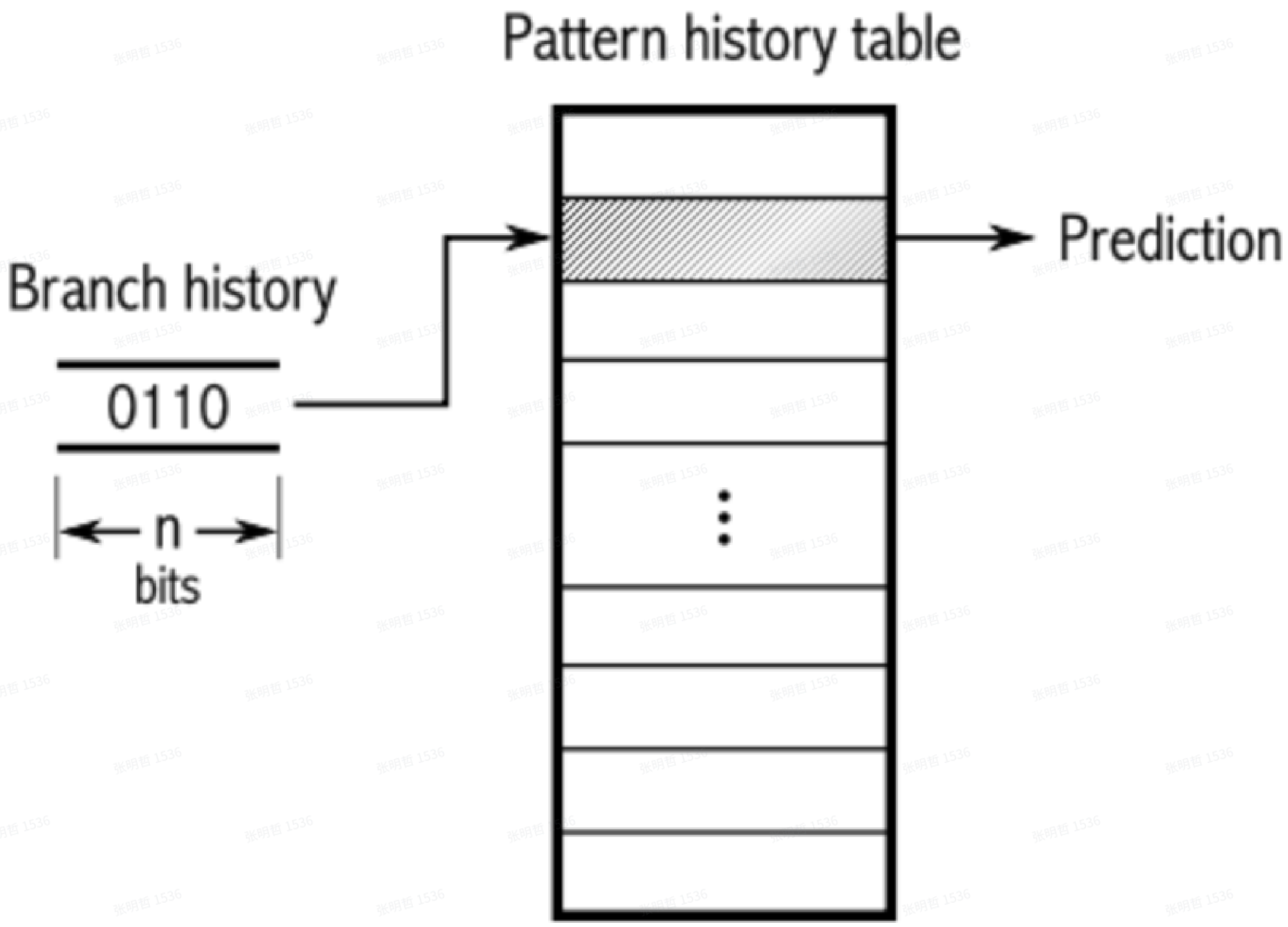
History-based predictor 记录上次该地址的结果作为下次相同地址的预测

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.



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

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex
#include <vector>         // std::vector
#include <string>         // std::string
#include <set>            // std::set
#include <fstream>        // std::fstream
#include <algorithm>      // std::algorithm
#include <unistd.h>       // unistd.h
//to compile: use command g++ swither_predictor.cpp -o swither_predictor.out
//to run: ./swither_predictor.out
using namespace std;
volatile int counter(0); // non-atomic counter
std::mutex mtx;           // locks access to counter
vector<bool> jump(0);      //the list of jump history
vector<long long> address(0);
void readfile(){
    ifstream fin("itrace.out");
    string str="";
    string::size_type sz;
    sz = 0;
    long long tmp;
    while(getline(fin,str)){
        if(str!="#eof"){
            tmp=stoll(str,&sz,0);
            address.push_back(tmp);
        }
    }

    tmp=address[0];
    long long diff;
    for(int i=1;i<address.size();i++){
        diff=address[i]-tmp;
        if(diff<0||diff>15){
            jump.push_back(true);
        }
        else{
            jump.push_back(false);
        }
        tmp=address[i];
    }
}
vector<float> taken_corr(0);         //the correction of taken predictor
void taken_predictor(){//predict taken
    long taken_cnt(0);
    for(int i=0;i<jump.size();i++) {
        if (jump[i]) {
```

```
            taken_cnt++;
        }
        taken_corr.push_back(((float) taken_cnt / (float) i)*100);
    }
        while(true){
            if (mtx.try_lock()) {   // counter finished
                ++counter;
                mtx.unlock();
                break;
            }
        }
}
vector<float> not_taken_corr(0);     //the correction of taken predictor
void not_taken_predictor(){
    long not_taken_cnt(0);
    for(int i=0;i<jump.size();i++){
        if(!jump[i]){
            not_taken_cnt++;
        }
        not_taken_corr.push_back((float)not_taken_cnt/(float)i*100);
    }
    while(true){
        if (mtx.try_lock()) {   // counter finished
            ++counter;
            mtx.unlock();
            break;
        }
    }
}
vector<float> backward_taken_forward_not_taken_corr(0);
vector<bool> backward_taken_forward_not_taken_predict(0);
void backward_taken_forward_not_taken_predictor(){
    long cnt=0;
    bool predict(false);
    for(int i=0;i<jump.size();i++){
        backward_taken_forward_not_taken_predict.push_back(predict);
        if(jump[i]==predict)
            cnt++;
        predict=!jump[i];
        backward_taken_forward_not_taken_corr.push_back((float)cnt/(float)i*100);
    }
    while(true){
        if (mtx.try_lock()) {   // counter finished
            ++counter;
            mtx.unlock();
            break;
```

```
        }
    }
}
vector<float> one_bit_corr(0);
vector<bool> one_bit_predict(0);
void one_bit_predictor(){
    long cnt=0;
    bool predict(false);
    for(int i=0;i<jump.size();i++){
        one_bit_predict.push_back(predict);
        if(predict==jump[i])
            cnt++;
        predict=jump[i];
        one_bit_corr.push_back((float)cnt/(float)i*100);
    }
    while(true){
        if (mtx.try_lock()) {   // counter finished
            ++counter;
            mtx.unlock();
            break;
        }
    }
}
enum class binarystatus{
    stronglynot,//strongly not taken
    weaklynot,//weakly not taken
    weaklytaken,//weakly taken
    stronglytaken,//strongly taken
};
class bipredictor{
public:
    binarystatus bst;
    bipredictor(){
        bst=binarystatus::stronglynot;
    }
    void iftaken(bool tk){
        if(tk){
            switch(bst){
                case binarystatus::stronglynot:
                    bst=binarystatus::weaklynot;
                    break;
                case binarystatus::weaklynot:
                    bst=binarystatus::weaklytaken;
                    break;
                case binarystatus::weaklytaken:
                    bst=binarystatus::stronglytaken;
                    break;
```

```
                case binarystatus::stronglytaken:
                    break;
            }
        }else{
            switch(bst){
                case binarystatus::stronglynot:
                    break;
                case binarystatus::weaklynot:
                    bst=binarystatus::stronglynot;
                    break;
                case binarystatus::weaklytaken:
                    bst=binarystatus::weaklynot;
                    break;
                case binarystatus::stronglytaken:
                    bst=binarystatus::weaklytaken;
                    break;
            }
        }
    }
};
vector<float> bimodal_corr(0);
vector<bool> bimodal_predict(0);
void bimodal_predictor(){
    auto *bpd=new bipredictor;
    long cnt=0;
    bool predict(false);
    for(int i=0;i<jump.size();i++){
        bimodal_predict.push_back(predict);
        if(predict==jump[i])
            cnt++;
        if(bpd->bst==binarystatus::stronglynot||bpd->bst==binarystatus::weaklynot){
            predict=false;
        }else
            predict=true;
        bpd->iftaken(jump[i]);
        bimodal_corr.push_back((float)cnt/(float )i*100);
    }
    while(true){
        if (mtx.try_lock()) {   // counter finished
            ++counter;
            mtx.unlock();
            break;
        }
    }
}
vector<float> two_level_adaptive_corr(0);
```

```
vector<bool>  two_level_adaptive_predict(0);
void handle(bipredictor *bpd,bool &predict,long &cnt,int i){
    if(bpd->bst==binarystatus::stronglynot||bpd->bst==binarystatus::weaklynot){
        predict=false;
    }else
        predict=true;
    if(predict==jump[i])
        cnt++;
    bpd->iftaken(jump[i]);
}
void two_level_adaptive_predictor(){
    auto *bpd00=new bipredictor;
    auto *bpd01=new bipredictor;
    auto *bpd10=new bipredictor;
    auto *bpd11=new bipredictor;
    bool predict(false);
    bool first(jump[0]);
    bool second(jump[1]);
    long cnt=2;
    two_level_adaptive_predict.push_back(first);
    two_level_adaptive_predict.push_back(second);
    two_level_adaptive_corr.push_back(100);
    two_level_adaptive_corr.push_back(100);
    for(int i=2;i<jump.size();i++){
        if(!first&&!second){
            handle(bpd00,predict,cnt,i);
        }else if(!first && second){
            handle(bpd01,predict,cnt,i);
        }else if(first && !second){
            handle(bpd10,predict,cnt,i);
        }else if(first&&second){
            handle(bpd11,predict,cnt,i);
        }
        first=jump[i-1];
        second=jump[i];
        two_level_adaptive_predict.push_back(predict);
        two_level_adaptive_corr.push_back((float)cnt/(float)i*100);
    }
    while(true){
        if (mtx.try_lock()) {   // counter finished
            ++counter;
            mtx.unlock();
            break;
        }
    }
}
vector<float> delayed_corr(0);
```

```
vector<bool> delayed_predict(0);
void delayed_predictor(){
    long cnt=2;
    delayed_corr.push_back(100);
    delayed_corr.push_back(100);
    delayed_predict.push_back(false);
    delayed_predict.push_back(false);
    for(int i=2;i<jump.size();i++){
        if(jump[i]==delayed_predict[i-2])
            cnt++;
        delayed_predict.push_back(jump[i]);
        delayed_corr.push_back((float)cnt/(float)i*100);
    }
    while(true){
        if (mtx.try_lock()) {   // counter finished
            ++counter;
            mtx.unlock();
            break;
        }
    }
}
vector<float> history_based_corr(0);
vector<bool> history_based_predict(0);
set<long long> jumped;
void history_based_predictor(){
    float cnt(1);
    bool prediction(false);
    for(int i=1;i<address.size();i++){
        history_based_corr.push_back(prediction);
        if(prediction==jump[i-1])
            cnt++;
        if(jumped.count(address[i])){
            prediction=true;
        }else
            prediction=false;
        if(jump[i-1])
            jumped.insert(address[i-1]);
        history_based_corr.push_back((float)cnt/(float )i);
    }
    while(true){
        if (mtx.try_lock()) {   // counter finished
            ++counter;
            mtx.unlock();
            break;
        }
    }
}
```

```
void do_prediction(bool &predict,int i){
    float maxvalue=max(taken_corr[i],not_taken_corr[i]);
    maxvalue=max(maxvalue,backward_taken_forward_not_taken_corr[i]);
    maxvalue=max(maxvalue,one_bit_corr[i]);
    maxvalue=max(maxvalue,bimodal_corr[i]);
    maxvalue=max(maxvalue,two_level_adaptive_corr[i]);
    maxvalue=max(maxvalue,delayed_corr[i]);
    maxvalue=max(maxvalue,history_based_corr[i]);
    if(maxvalue==taken_corr[i]) {
        predict = true;
        return;
    }
    if(maxvalue==not_taken_corr[i]){
       predict=false;
       return;
    }
    if(maxvalue==backward_taken_forward_not_taken_corr[i]){
        predict=backward_taken_forward_not_taken_predict[i];
        return ;
    }
    if(maxvalue==one_bit_corr[i]){
        predict=one_bit_predict[i];
        return ;
    }
    if(maxvalue==bimodal_corr[i]){
        predict=bimodal_predict[i];
        return ;
    }
    if(maxvalue==two_level_adaptive_corr[i]){
        predict=two_level_adaptive_predict[i];
        return ;
    }
    if(maxvalue==delayed_corr[i]){
        predict=delayed_predict[i];
        return ;
    }
    if(maxvalue==history_based_corr[i]){
        predict=history_based_predict[i];
        return ;
    }
}
void do_reduction(){
    bool predict(false);
    long cnt(0);
    for(int i=0;i<jump.size();i++){
        if(predict==jump[i])
            cnt++;
```

```
            do_prediction(predict,i);
    }
    cout<<"prediction correction: "<<(float)cnt/(float)jump.size()*100<<endl;
}
void reduction(){
    while(true){
        if (mtx.try_lock()) {   //read counter
            if (counter == 8) { //counter finished
                do_reduction();
                mtx.unlock();
                break;
            } else {
                mtx.unlock();
                sleep(5);//wait 5s for counter finished
            }
        }else
            sleep(5);
    }
}
int main () {
    readfile();
    std::thread threads[9];
    threads[0]=thread(reduction);
    threads[1]=thread(taken_predictor);
    threads[2]=thread(not_taken_predictor);
    threads[3]=thread(backward_taken_forward_not_taken_predictor);
    threads[4]=thread(one_bit_predictor);
    threads[5]=thread(bimodal_predictor);
    threads[6]=thread(two_level_adaptive_predictor);
    threads[7]=thread(delayed_predictor);
    threads[8]=thread(history_based_predictor);
    for (auto& th : threads) th.join();
    return 0;
}
```

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.

186-193, doi: 10.1109/DSD.2001.952280.