

（深圳）

实验报告

开课学期： 2021春季

课程名称： 计算机体系结构(实验)

实验名称： 分支预测器设计

实验性质： 综合设计型

实验时间： 6 地点： T2210

学生班级：18计算机科学与技术7班

学生学号： 180110711

学生姓名： 郑漫莎

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2021年4月

# 1. 实验内容

*简述实验内容*

本实验将基于Pin插桩分析的工作机理，使用C/C++实现分支预测器的软件模型，从而在进一步熟悉插桩工具使用的同时，加深对分支预测原理的理解。

实验包含基础实验和附加实验。基础实验实现三种动态分支预测方法，分别为基于BHT的分支预测，基于全局历史的分支预测，基于局部历史的分支预测。基础实验还包括两种锦标赛分支预测，分别为基于全局历史的选择方法和基于局部历史的选择方法。

附加实验实现Bi-mode动态分支预测和分支目标的检测。

文件brchPredict.cpp中为各预测器的实现代码，brchPredict\_addr.cpp为实现分支目标地址检测的代码。

# 2. 设计与实现

## 2.1 题目分析

*描述个人对实验题目、实验内容和实验要求的分析和理解，并得出相应的结论，或根据分析得出设计方案。*

和静态分支预测相比较，动态分支预测能根据分支指令过去的表现来预测将来的行为。在分支行为发生变化，那么分支预测的结果也相应的发生改变。因此，动态分支预测具有更高的预测准确率和不同条件下的适应性。分支预测分为两部分，预测和更新。以下为实现的分支预测器分析：

1. 基于BHT的分支预测是根据分支历史表来记录分支的历史行为，通过与指令地址相关的Tag进行索引，得到的分支历史进行分支预测。
2. 基于全局历史的分支预测是在BHT的基础上增加了GHR，将所有分支指令关联起来，现在的分支历史不仅和指令地址相关，还和最近的部分分支指令相关。
3. 基于局部历史的分支预测用LHT代替全局历史预测中的GHR，因为不是所有分支指令都具有关联性。近似于每条分支指令地址都维护一个局部转移历史。
4. 基于全局选择历史的锦标赛分支预测含两个子预测器和一个全局选择历史寄存器GSHR。GSHR记录历史预测器的选择，指导下一次预测器的选择。子预测器独立进行更新。
5. 基于局部选择历史的锦标赛分支预测含两个子预测表和一个局部选择历史表LSHT。不同的地址维护不同的选择历史。
6. 基于Bi-mode的分支预测在全局历史的分支预测的基础上，一定程度上减少了分支别名干扰，即分支预测过程中PHT表项的冲突问题，不同的地址分支访问同一个表项，这是由于哈希的冲突。Bi-mode将PHT表分为两层，分别为方向PHT和选择PHT。方向PHT利用指令地址和GHR的异或进行索引 进行跳转或不跳转判断，选择PHT利用指令地址进行索引，进行方向表的选择。

分支目标地址的预测，将基础的BHT结构进行改进，增加了有效位和分支目标地址字段，独立进行更新，能够预测分支目标地址。

## 2.2 实验过程

*图文并茂地展示实验分析、优化和求解的过程。对于关键实现，可用图表叙述（必要时可贴关键代码，但禁止大段粘贴代码）。*

**基本结构**：

1. BHT表项：

每个表项维护一个分支历史val，一般选择为2bit，因为2bit有一次犯错机会，但不会过多。

变量val：分支历史

* 1. Increase：更倾向于跳转预测。分支历史加一，但不超过最大值，2位为11。
  2. decrease：更倾向于不跳转预测。分支历史减一，但不小于0。
  3. Reset：重置，最开始预测不跳转，2位为01
  4. GetVal：获得分支历史的值
  5. IsTaken：根据val的最高位判断是否跳转，跳转返回true，不跳转返回false

1. 历史寄存器：

每个历史寄存器维护一个分支跳转历史，初始为0

变量val：最近k条指令历史跳转

* 1. ShiftIn(b)：更新分支跳转历史val=(val<<1)|b。

1. val取低bits位：一般用于Tag的计算

#define truncate(val, bits) ((val) & ((1 << (bits)) - 1))

**基本过程**：

预测：

BOOL predict(ADDRINT addr)

更新：

void update(BOOL takenActually, BOOL takenPredicted, ADDRINT addr)

### 2.2.1 基于BHT的分支预测

**分析：**

基于BHT的分支预测是根据分支历史表来记录分支指令的历史行为。如图2-1所示，Tag字段是指令地址的一部分，分支历史由2bit饱和计数器组成。在预测过程中，在一定程度上，Tag字段可以近似为一个指令地址，随着分支的跳转，每个Tag也即每条指令地址维护一个分支历史。在分支历史的基础上，进行跳转的判断。分支历史采用2bit饱和计数器，如图2-2所示，00为强不跳转，01为弱不跳转，10为弱跳转，11为强跳转。对于一直跳转的指令，即11，当突然不跳转时，转为10，给一次机会，预测下一次还是为跳转，如果再次为跳转，则认为下次不跳转，转为01，否则转为11，反之亦然。



图 2‑1 基础BHT的结构

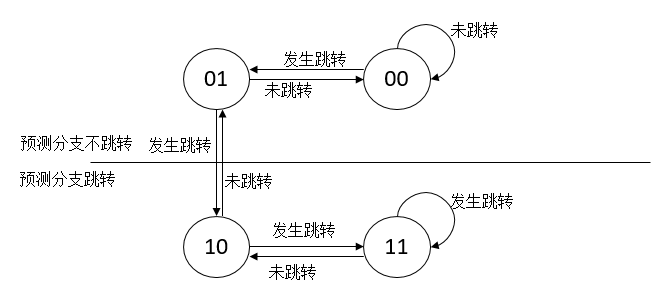


图 2‑2 基于2bit饱和计数器的分支预测更新策略1

**变量：**

SaturatingCnt<2> counter[1 << L]

：维护Tag为L位的BHT表，每行含2bit的分支历史。

**预测：**根据addr低L位为索引的counter对应的分支历史，返回最高位，即0不跳转，1跳转

return counter[truncate(addr, L)].isTaken();

**更新**：根据addr低L位索引，counter对应的分支历史进行更新，如果结果跳转，则分支历史increase操作，反之decrease操作。

            UINT64 Tag = truncate(addr, L);

            // 10->01

            if(takenActually)

                counter[Tag].increase();

            else

                counter[Tag].decrease();

**优化**：

1. 指导书中更新策略如图2-3，采用更新策略如图2-2。相对而言，图2-3的更新策略跳转较大。在10时如果未跳转则变为00，在至少在未来的两次跳转预测中，都预测为不跳转。在进行测试后，发现图2-2的更新策略准确率更高。

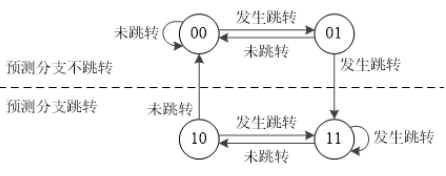


图 2-3基于2bit饱和计数器的分支预测更新策略2

1. L取20。Tag为分支指令地址的低L位进行索引。如果L过小，则分支指令地址很有可能冲突，即不同的分支指令对应同一个counter表项。 由于在该策略中，一个分支指令对应一个counter表项是最好的，不会产生冲突，所以L尽可能往大取。在本次实验中，表项最多4096条，即L最大20，故L取20。

### 2.2.2 基于全局历史的分支预测

**分析：**

基于全局历史的分支预测是在BHT的基础上增加了GHR，将所有分支指令关联起来。如图2-4，k比特的GHR记录最近k条分支指令的历史跳转方向，PHT的Tag字段由分支指令地址和GHR异或得到。在一定程度上，PHT的每一行可以对应为在过去k条分支指令的跳转历史为GHR的情况下，分支指令对应的分支历史，很大程度上，减少了同一指令由于跳转历史不同而造成的的结果冲突。例如，A->C，B->C为A调用C和B调用C，对于C而言的分支指令可能由不同的跳转结果，但对应的分支指令地址相同。GHR通过左移位的方式进行更新，1为跳转，0为不跳转。

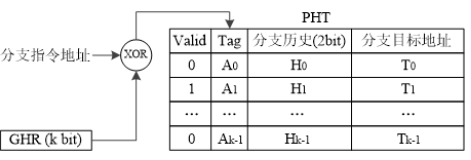


图 2-4 基于全局历史的分支预测

**变量：**

    SaturatingCnt<BITS> bhist[1 << L];  // PHT中的分支历史字段

    ShiftReg<H> GHR;    // 全局历史

**预测：**取addr与全局历史GHR亦或结果的低L位作为索引，得到bhist对应的分支历史，返回最高位，即0不跳转，1跳转

    UINT64 Tag = truncate(addr ^ GHR.getVal(), L);

    return bhist[Tag].isTaken();

**更新**：更新全局历史GHR，即GHR = (GHR << 1)|takenActually。取addr与全局历史GHR亦或结果的低L位作为索引，同BHT更新bhist对应的分支历史

    UINT64 Tag = truncate(addr ^ GHR.getVal(), L);

    GHR.shiftIn(takenActually);

    if(takenActually)

        bhist[Tag].increase();

    else

        bhist[Tag].decrease();

**优化**：

1. 分支指令地址和GHR等类似结构进行异或操作，将地址和历史信息进行结合。相对于与、或等其他操作，异或更加的中性化，不会偏向于0或者1，导致Tag分布不均匀。与操作容易使得Tag中的0偏多，相对而言1多的行就较少占用，容易加重冲突。或操作亦然。
2. 同BHT的分支预测，L取20。
3. H取20。H对应为GHR的长度，即过去H个分支指令的跳转情况。当H过小时，所包含的历史跳转情况过少，准确率较低。当H过大时，所包含的历史跳转情况过多，将不相关的跳转情况也包含了，相对而言，分支历史较为分散，准确度降低。当准确率差不多时，尽可能取小，相对而言时空开销小。经过一定的比较和调试，选择H取20。

### 2.2.3 基于局部历史的分支预测

**分析：**

基于局部历史的分支预测用LHT代替全局历史预测中的GHR，因为不是所有分支指令都具有关联性。如图2-5所示，LHT每条记录包含Tag和局部转移历史，分支指令地址和对应的局部转移历史进行异或，得到的结果作为PHT的Tag。即近似为每条分支指令地址都维护一个局部转移历史，每条分支指令在不同的局部转移历史下维护一行分支历史。

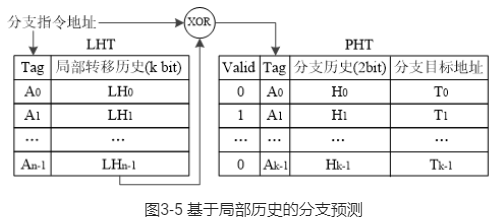


图 2-5 基于局部历史的分支预测

**变量：**

    SaturatingCnt<BITS> bhist[1 << L];  // PHT中的分支历史字段

    ShiftReg<H> LHT[1 << HL];   // 局部转移历史表

**预测：**取addr低HL位作为索引获取局部转移历史LHT[i]。取addr与局部转移历史LHT[i]亦或结果的低L位作为索引，得到bhist对应的分支历史，返回最高位，即0不跳转，1跳转

    UINT64 LH = LHT[truncate(addr, HL)].getVal();

    UINT64 Tag = truncate(addr ^ LH, L);

        return bhist[Tag].isTaken();

**更新**：更新局部历史LHT[i]，即LHT[i] = (LHT[i] << 1)|takenActually。取addr与LHT[i]亦或结果的低L位作为索引，同BHT更新bhist对应的分支历史

        UINT64 LHTTag = truncate(addr, HL);

        UINT64 Tag = truncate(addr ^ LHT[LHTTag].getVal(), L);

        // LHT update

        LHT[LHTTag].shiftIn(takenActually);

        // PHT update

        if(takenActually)

            bhist[Tag].increase();

        else

            bhist[Tag].decrease();

**优化**：

1. 同全局历史选择器，L取20，H取20

### 2.2.4 基于全局历史的锦标赛预测

**分析：**

基于全局选择历史的锦标赛分支预测含两个子预测器和一个全局选择历史寄存器GSHR。GSHR的更新规则如图2-6，只有一个预测期结果正确时进行更新操作。子预测器按照跳转结果和预测结果进行预测。

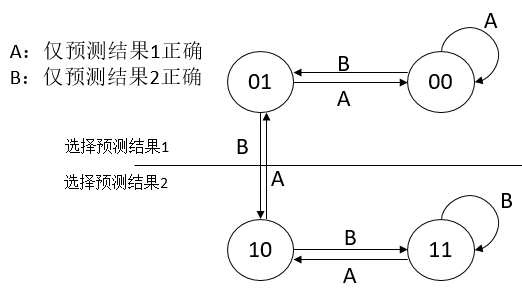


图 2-6 GSHR的更新规则

**变量：**

    SaturatingCnt<BITS> GSHR;   // 分支历史

    BranchPredictor\* BPs[2];    // 子预测器

**预测：**根据GSHR的最高位选择子预测期，最高位0取子预测器1的结果，1取子预测器2的结果

    if(!GSHR.isTaken())

        return BPs[0]->predict(addr);

    else

        return BPs[1]->predict(addr);

**更新**：独立更新子预测器。按照图2-6更新GSHR，仅子预测器1正确时，GSHR做decrease操作，倾向于选择子预测器1。仅子预测器2正确时，GSHR做increase操作，倾向于选择子预测器2。

    BOOL result[2];

    result[0] = BPs[0]->predict(addr);

    result[1] = BPs[1]->predict(addr);

    // BPs update

    BPs[0]->update(takenActually, result[0], addr);

    BPs[1]->update(takenActually, result[1], addr);

    // GSHR update

    if(result[0] == takenActually && result[1] != takenActually)

        GSHR.decrease();

    else if(result[0] != takenActually && result[1] == takenActually)

        GSHR.increase();

**优化**：

1. 子预测器的选择，子预测器选择基于全局历史和基于局部历史两种预测器。相对而言，循环操作的情况下，局部历史预测器可能较为准确，而其余情况可能全局历史较为准确，锦标赛预测器能够在不同的时间倾向于不同的预测器。而基于BHT的选择预测器和基于全局历史的预测器结合的锦标赛预测器，预测结果和基于全局历史的预测器相近，且略低于及与全局历史的预测器。这可能由于全局历史预测器是将基于BHT的预测器按照历史进行了一定划分，整体而言，全局历史预测器包含基于BHT预测器，锦标赛预测器更倾向于全局历史预测器。

### 2.2.5 基于局部历史的锦标赛预测

**分析：**

基于局部选择历史的锦标赛预测分支预测含两个子预测表和一个局部选择历史表LSHT，LSHT的结构如图2-1，LSHT每个相应的选择历史LSHT[i]的更新策略如图2-6。预测器按照跳转结果和预测结果进行预测。

**变量：**

    SaturatingCnt<BITS> LSHT[1 << L];   // 分支历史表

    BranchPredictor\* BPs[2];            // 子预测器

**预测：**根据分支指令地址的低L位选择LSHT对应的分支历史LSHT[i]，根据LSHT[i]的最高位选择子预测期，最高位0取子预测器1的结果，1取子预测器2的结果

    UINT64 Tag = truncate(addr, L);

    if(!LSHT[Tag].isTaken())

        return BPs[0]->predict(addr);

    else

        return BPs[1]->predict(addr);

**更新**：独立更新子预测器。按照图2-6更新LSHT[i]，仅子预测器1正确时，LSHT[i]做decrease操作，倾向于选择子预测器1。仅子预测器2正确时，LSHT[i]做increase操作，倾向于选择子预测器2。

    UINT64 Tag = truncate(addr, L);

    BOOL result[2];

    result[0] = BPs[0]->predict(addr);

    result[1] = BPs[1]->predict(addr);

    // BPs update

    BPs[0]->update(takenActually, result[0], addr);

    BPs[1]->update(takenActually, result[1], addr);

    // LSHT update

    if(result[0] == takenActually && result[1] != takenActually)

        LSHT[Tag].decrease();

    else if(result[0] != takenActually && result[1] == takenActually)

        LSHT[Tag].increase();

**优化**：

1. 子预测器的选择同基于全局历史的锦标赛预测器

*\*如果有做附加题，需按照指导书要求，在报告中添加相应的分析、设计过程和运行结果截图和对比等内容。*

### 2.2.6 基于Bi-mode的分支预测

**分析：**

基于Bi-mode的分支预测在全局历史的分支预测的基础上，一定程度上减少了分支别名干扰，即分支预测过程中PHT表项的冲突问题，不同的地址分支访问同一个表项，这是由于哈希的冲突。Bi-Mode如图2-6所示，将PHT表分为两层，分别为方向PHT和选择PHT。方向PHT分为跳转表PHTT和不跳转表PHTNT。和全局历史分支预测类似，利用PC和GHR的异或进行方向PHT的索引，用PC直接进行选择PHT的索引。根据选择PHT进行选择使用哪个方向表的结果。对于方向表而言，仅被选择的方向PHT会进行更新，更新策略与图2-2类似。对于选择表而言，当结果跳转时，选择表更倾向于跳转表，否则倾向于不跳转表。但是，当选择PHT选择方向和结果不一致，但选择的方向表的结果和跳转结果一致，即预测正确，则不更新选择表。

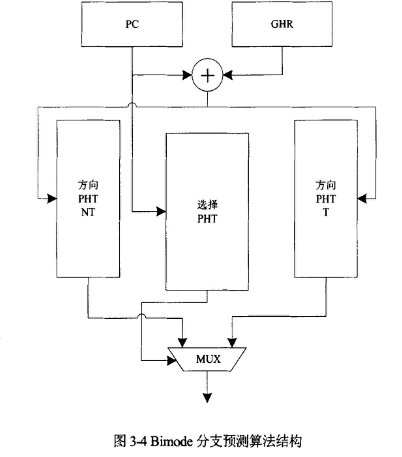


图 2-6 Bi-mode的分支预测算法结构

**变量：**

    SaturatingCnt<BITS> directionTPHT[1 << L];  // 跳转方向表

    SaturatingCnt<BITS> directionNTPHT[1 << L]; // 不跳转方向表

    SaturatingCnt<BITS> choosePHT[1 << L];      // 选择表

    ShiftReg<H> GHR;                            // 全局历史

**预测：**根据分支指令地址的低L位，获得对应选择的方向choosePHT[i]，choosePHT[i]为1，倾向于跳转方向表，否则倾向于不跳转方向表。根据分支指令地址和GHR的亦或结果，索引对应方向表的结果，0预测不跳转，1预测跳转。

    UINT64 directionTag = truncate(addr ^ GHR.getVal(), L);

    if(choosePHT[truncate(addr, L)].isTaken())

        return directionTPHT[directionTag].isTaken();

    else

        return directionNTPHT[directionTag].isTaken();

**更新**：全局历史更新。仅被选择的方向PHT根据实际跳转结果更新，更新方式如图2-1。选择PHT根据实际跳转情况更新，跳转则increase操作，不跳转则decrease操作。但当选择的方向表和结果不一致，但选择的方向表的结果和跳转结果一致，则不更新选择表。

    UINT64 directionTag = truncate(addr ^ GHR.getVal(), L);

    UINT64 chooseTag = truncate(addr, L);

    BOOL chooseTaken = choosePHT[chooseTag].isTaken();

    BOOL chooseUpdate = !(takenActually != chooseTaken && takenActually == takenPredicted);

    GHR.shiftIn(takenActually);

    // 仅被选择的方向PHT更新，若选择PHT和结果不一致，而方向PHT和结果相同，则不更新选择PHT

    if(takenActually)

    {

        if(chooseTaken)

            directionTPHT[directionTag].increase();

        else

            directionNTPHT[directionTag].increase();

        if(chooseUpdate)

            choosePHT[chooseTag].increase();

    }

    else

    {

        if(chooseTaken)

            directionTPHT[directionTag].decrease();

        else

            directionNTPHT[directionTag].decrease();

        if(chooseUpdate)

            choosePHT[chooseTag].decrease();

    }

**优化**：

1. 同基于全局历史的分支预测，L取20，H取20

### 2.2.7 分支目标地址预测

**分析：**

以基于BHT的分支预测为例，其余分支预测类似。为预测分支目标地址，将BHT改进为图2-7，增加了有效位和分支目标地址字段。分支目标地址按照下述方案进行独立更新。当前行无效时，且分支指令不跳转，则不进行分支目标地址预测。当前行无效，但分支指令跳转，则将当前行标记为有效，复位饱和计数器，同时更新饱和计数器，并将分支指令的目标地址更新到BHT中。如果当前行有效，则用BHT的分支目标地址进行预测，预测失败时，当前行标记为无效。

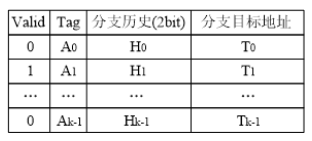


图 2-7 改进的BHT结构

**更改的基本结构**：

1. 结构体PredictResult：预测结果.
   1. Taken：是否跳转
   2. Valid：分支目标地址有效位
   3. targetAddr：分支目标地址

typedef struct predict\_result

{

    BOOL taken; // 是否跳转

    BOOL valid; // 分支目标地址是否有效

    ADDRINT targetAddr; // 分支目标地址预测

} PredictResult;

1. BHT表项：

每个表项维护一个分支历史val，一个分支目标地址有效位valid，分支目标地址targetAddr.分支历史一般选择为2bit，因为2bit有一次犯错机会，但不会过多。

变量：

* 1. val：分支历史
  2. valid：分支目标地址有效位
  3. targetAddr：分支目标地址

操作：

* 1. Increase：更倾向于跳转预测。分支历史加一，但不超过最大值，2位为11。
  2. decrease：更倾向于不跳转预测。分支历史减一，但不小于0。
  3. Reset：重置，最开始预测不跳转，2位为01
  4. GetVal：获得分支历史的值
  5. IsTaken：根据val的最高位判断是否跳转，跳转返回true，不跳转返回false
  6. GetValid：获得是否有效
  7. SetValid：设为有效
  8. ResetValid：设为无效
  9. getAddr：获取目标地址
  10. setAddr(addr)：将目标地址更新为addr

**更改的基本过程**：

预测：

    PredictResult predict(ADDRINT addr)

    {

        PredictResult re;

        re.taken = false;

        re.valid = false;

        re.targetAddr = 0;

        return re;

    };

更新：

void update(BOOL takenActually, ADDRINT addrActually, PredictResult predicted, ADDRINT addr);

**以基于BHT的分支预测为例：**

**变量：**

    SaturatingCnt<2> counter[1 << L];

**预测：**根据addr低L位为索引的counter对应的分支历史和有效位，和分支目标地址

    PredictResult ret;

    UINT64 Tag = truncate(addr, L);

    ret.taken = counter[Tag].isTaken();

    ret.valid = counter[Tag].getValid();

    ret.targetAddr = counter[Tag].getAddr();

    return ret;

**更新**：根据addr低L位索引，counter对应的行进行更新。分支历史的更新同基于BHT的分支预测，跳转则increase，不跳转decrease操作。如果分支目标地址预测有效，当预测失败时，将有效位复位为无效。如果分支目标地址预测无效，当跳转时，则将有效位置位为有效，更新分支目标地址。

    UINT64 Tag = truncate(addr, L);

    BOOL valid = counter[Tag].getValid();

    ADDRINT addrPredict = counter[Tag].getAddr();

    // taken

    if(takenActually)

        counter[Tag].increase();

    else

        counter[Tag].decrease();

    // 有效

    if(valid)

    {

        // 预测失败

        if(addrActually != addrPredict)

            counter[Tag].resetValid();

    }

    // 无效

    else

    {

        // 变为有效

        if(takenActually)

        {

             counter[Tag].setValid();

             counter[Tag].setAddr(addrActually);

        }

    }

## 2.3 实验结果及分析

*描述测试方法，列出实验结果，并进行相应的分析。*

*如果进行了优化，需给出优化前后的结果对比截图及文字分析（如果优化没有效果，也需分析原因）。*

### 基本实验结果

**测试方法**：实验使用SPEC 2006标准测试程序集中的bzip2（压缩算法）、sjeng（国际象棋算法）、wrf（天气预报算法）、sphinx3（语音识别算法）对分支预测模型进行测试。其中，bzip2和sjeng是定点测试程序，wrf和sphinx3是浮点测试程序。

**基本测试结果**：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| BHT(20) | 94.91 | 94.81 | 94.92 | 95.02 |
| GHP(20,20) | 98.05 | 98.01 | 98.06 | 98.19 |
| LHP(20,20) | 93.76 | 93.56 | 93.83 | 94.69 |
| TP\_GSH(GHP,LHP) | 98.21 | 98.18 | 98.24 | 98.40 |
| TP\_LSH(GHP,LHP) | 98.46 | 98.43 | 98.49 | 98.68 |
| Bi-mode(20,20) | 98.18 | 98.14 | 98.19 | 98.32 |

**分析**：

整体而言，基于局部历史的分支预测准确率最低，可能是因为LHT根据分支指令地址进行索引，转移历史过于分散。基于BHT的分支预测次之。基于全局历史的分支预测解决了基于BHT分支预测护士分支指令之间关联性，从而提高准确率。

基于Bi-mode的分支预测在基于全局历史的分支预测的基础上，减少了分支别名的干扰，从而提高准确率，在单个预测器而言，是这几个中准确率最高的。

锦标赛分支预测较单个分支预测而言，准确率有所提高。基于局部选择历史的锦标赛分支预测的准确率最高，和基于全局选择历史的锦标赛分支预测相比，他将不同的分支指令对应到不同的分支历史中。

### 2.3.1 基于BHT的分支预测

**1. 更新策略优化**

**结果对比：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| BHT(20) | 94.91 | 94.81 | 94.92 | 95.02 |
| BHT(20)指导书策略 | 94.78 | 94.68 | 94.79 | 94.89 |

**2. L优化**

**结果对比：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| BHT(2) | 64.15 | 63.59 | 63.80 | 63.75 |
| BHT(16) | 94.86 | 94.73 | 94.87 | 94.97 |
| BHT(20) | 94.91 | 94.81 | 94.92 | 95.02 |

**分析：**相对而言L越大，分支指令冲突越少，准确率越高。

### 2.3.2 基于全局历史的分支预测

**1. 异或优化**

**结果对比：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| GHP(20,20)xor | 98.05 | 98.01 | 98.06 | 98.19 |
| GHP(20,20)and | 96.66 | 96.57 | 96.67 | 96.68 |
| GHP(20,20)or | 96.54 | 96.54 | 96.63 | 96.66 |

**2. H优化**

**结果对比：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| GHP(20,2) | 95.47 | 95.39 | 95.49 | 95.57 |
| GHP(20,20) | 98.05 | 98.01 | 98.06 | 98.19 |
| GHP(20.25) | 98.05 | 98.00 | 98.06 | 98.21 |
| GHP(20,30) | 98.03 | 98.01 | 98.06 | 98.18 |

**分析：**H对应为GHR的长度，即过去H个分支指令的跳转情况。当H过小时，所包含的历史跳转情况过少，准确率较低。当H过大时，所包含的历史跳转情况过多，将不相关的跳转情况也包含了，相对而言，分支历史较为分散，准确度降低。当准确率差不多时，尽可能取小，相对而言时空开销小。结果取20。

### 2.3.3 基于局部历史的分支预测

无

### 2.3.4 基于全局历史的锦标赛预测

**1. 子预测器选择优化**

**结果对比：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| TP\_GSH(GHP,LHP) | 98.21 | 98.18 | 98.24 | 98.40 |
| TP\_GSH(BHT,GHP) | 98.04 | 97.99 | 98.05 | 98.18 |

**分析**：基于BHT的分支预测和基于全局历史的分支预测结果比不上基于全局历史的分支预测和基于局部历史的分支预测。基于BHT的选择预测器和基于全局历史的预测器结合的锦标赛预测器，预测结果和基于全局历史的预测器相近，且略低于及与全局历史的预测器。其余搭配方式没有进行测试。

### 2.3.5 基于局部历史的锦标赛预测

无

### 2.3.6 基于Bi-mode的分支预测

无

### 2.3.7 分支目标地址预测

**1. 实验结果：**

**无分支目标地址实验结果：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| BHT(20) | 94.91 | 94.81 | 94.92 | 95.02 |
| GHP(20,20) | 98.05 | 98.01 | 98.06 | 98.19 |
| LHP(20,20) | 93.76 | 93.56 | 93.83 | 94.69 |
| TP\_GSH(GHP,LHP) | 98.21 | 98.18 | 98.24 | 98.40 |
| TP\_LSH(GHP,LHP) | 98.46 | 98.43 | 98.49 | 98.68 |
| Bi-mode(20,20) | 98.18 | 98.14 | 98.19 | 98.32 |

**分支预测和分支目标地址预测结果**

**跳转预测准确率：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| BHT(20) | 94.90 | 94.82 | 94.92 | 95.02 |
| GHP(20,20) | 98.05 | 98.01 | 98.07 | 98.21 |
| LHP(20,20) | 93.74 | 93.55 | 93.81 | 94.70 |
| TP\_GSH(GHP,LHP) | 98.21 | 98.18 | 98.23 | 98.40 |
| TP\_LSH(GHP,LHP) | 98.46 | 98.43 | 98.49 | 98.68 |
| Bi-mode(20,20) | 98.18 | 98.14 | 98.19 | 98.32 |

**分支目标地址预测准确率：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Bzip2 | Sjeng | Wrf | Sphinx3 |
| BHT(20) | 93.67 | 93.51 | 93.69 | 93.83 |
| GHP(20,20) | 97.25 | 97.17 | 97.26 | 97.43 |
| LHP(20,20) | 86.90 | 86.25 | 86.92 | 88.79 |
| TP\_GSH(GHP,LHP) | 97.13 | 97.04 | 97.13 | 97.38 |
| TP\_LSH(GHP,LHP) | 97.47 | 97.38 | 97.50 | 97.80 |
| Bi-mode(20,20) | 97.32 | 97.25 | 97.32 | 97.49 |

**分析**:

能够检测分支目标地址的准确率与仅检测跳转方向的结果一致。

分支目标地址的预测准确率略低于跳转预测准确率，与跳转预测准确率趋势相当。略低于可能是由于最开始进行的跳转没有分支目标，分支目标地址是经过跳转后记录的。

# 3. 总结和感想

*请填写实验过程中遇到的问题及解决方案、学习本课程的收获和反思，并对课程提出合理的建议。*

问题1：对于template<>不熟悉，锦标赛预测器不知道如何初始化。

解决方案：百度

问题2：程序跑的慢

解决方案：四个测序集同时跑

问题3：在基于局部历史的分支预测器中，局部历史转移表应该使用分支指令地址的低HL位，即6位进行索引，因为根据指导书，设置为64行。但是，因为错误，使用了低H位，即20位进行索引，程序不报错，且结果更好。

解决方案：不知道。我知道他是错的，但是他给了更好的结果，就很神奇。

收获：对于分支预测有了很好的了解，同时知道如何进行更多分支预测的查找和了解。