

实验报告

开课学期： 2022秋季

课程名称： 计算机体系结构(实验)

实验名称： 实验2：分支预测器设计

学生班级： 6

学生学号： 200110618

学生姓名： 邓皓元

实验与创新实践教育中心印制

2022年11月

# 1. 题目分析

*回顾课上讲解的几个动态分支预测器，用自己的话描述其原理及工作过程，并尝试分析其设计有哪些合理之处和不合理之处。*

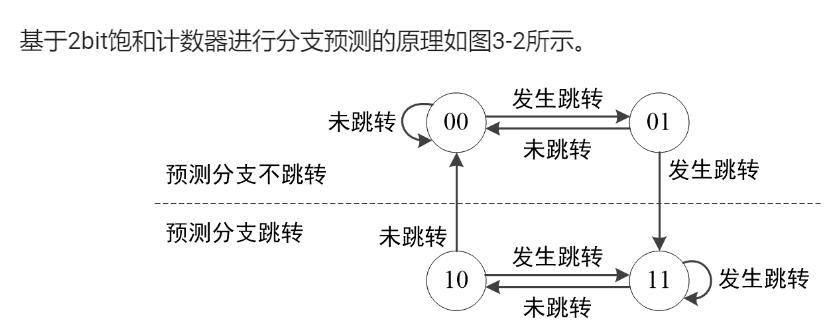
1.基于BHT的分支预测

基于BHT的分支预测方法使用分支历史表来记录分支指令的历史行为。

即是设置一个BHT表记录K个tag的分支历史，对于每个跳转指令查找BHT表中对应tag的分支历史(饱和计数器)并利用分支历史信息预测分支跳转，之后根据预测分支跳转和实际分支跳转更新对应tag的分支历史。

对分支历史(饱和计数器)的分支预测和更新

分支预测：若饱和计数器的最高位为1，预测分支跳转，否则预测分支不跳转。



更新：这是2bit饱和计数器，若是基于nbit则上述图中有2n个状态，预测分支跳转和预测分支不跳转的数量相同，发生跳转则状态向右移动++(若状态为2n-1-1，则状态进入2n-1、若状态为2n-1，则状态不改变)，未跳转则状态向左移动--(若状态为2n-1，则状态进入0、若状态为0，则状态不改变)

合理之处：BHT的分支预测方法利用了先前的跳转指令来预测分支跳转，作为最简单的动态分支预测器。

不合理之处：BHT的分支预测方法只利用了有限的先前跳转指令，对指令之间的关联性考虑较少

2.基于全局历史的分支预测

基于全局历史的分支预测方法在BHT的基础上增加了GHR来将所有分支指令关联起来，把BHT表改成了PHT表。

将分支指令的地址和GHR当前值进行hash运算，用得到的hash值来查PHT；最后根据PHT当前行的分支历史和分支目标地址，对该分支指令的分支跳转方向和分支目标地址进行预测

除了更新PHT中每个tag的分支历史(饱和计数器)，还需要更新GHR，当分支指令的实际跳转行为被确定时，GHR通过移位的方式进行更新——若指令跳转，则GHR=(GHR<<1)|1GHR=(GHR<<1)|1，否则GHR=(GHR<<1)|0GHR=(GHR<<1)|0。

合理之处：基于全局历史的分支预测方法相对于BHT更多地考虑了全局指令之间的关联性

不合理之处：基于全局历史的分支预测将所有分支指令都关联到一起，却不是所有指令都有关联性

1. 基于局部历史的分支预测

基于局部历史的分支预测方法使用LHT来代替全局历史预测中的GHR。

与GHT不同，LHT利用LHT替代GHR，首先用分支指令的地址查LHT，得到分支指令的局部转移历史LHT[i]；然后将分支指令的地址和LHT[i]进行hash运算，再用得到的hash值来查PHT；最后根据PHT当前行的分支历史和分支目标地址，对该分支指令的分支跳转方向和分支目标地址进行预测

除了更新PHT中每个tag的分支历史(饱和计数器)，还需要更新LHT，当分支指令的实际跳转行为被确定时，LHT[i]和GHR一样，也通过移位的方式进行更新——若指令跳转，则LHT[i]=(LHT[i]<<1)|1LHT[i]=(LHT[i]<<1)|1，否则LHT[i]=(LHT[i]<<1)|0LHT[i]=(LHT[i]<<1)|0。

合理之处：基于局部历史的分支预测方法相对于全局历史更多地考虑了局部指令之间的关联性

不合理之处：基于局部历史的分支预测方法对于全局指令之间的关联性考虑较少

4.锦标赛分支预测

锦标赛分支预测（又称混合分支预测或组合分支预测）是一种博采众长的分支预测方法，其基本原理是将两个或以上的分支预测方法进行结合，充分发挥各预测方法的优势，以进一步提高分支预测的准确度。

利用全局历史或者局部历史判断子预测器的预测能力，根据历史预测结果最好的子预测器来进行分支预测。其中全局历史和局部历史的选择方法先前有提到过。

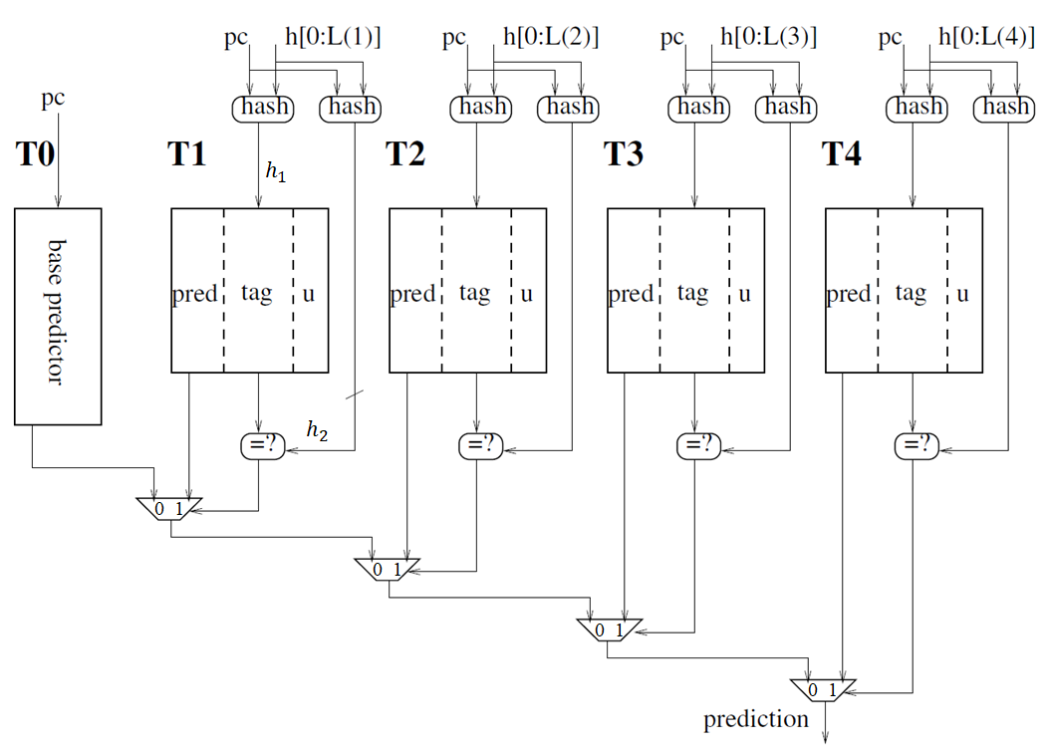
每次分支预测都要更新历史选择器和每个子预测器。

合理之处：选取子预测器中预测能力最强的子预测器来进行分支预测，进一步加强了分支预测的准确性。

不合理之处：没有结合各个子预测器的结果，只选择了最优的结果。

5.TAGE分支预测

TAGE是将若干个GHR位宽呈几何倍数关系增长的全局历史预测器结合起来做预测的复合预测器，其基本结构形如图所示。



TAGE是一个子预测器数量、各子预测器的GHR位宽、PHT表大小等参数均可配置的分支预测器

预测机制： 预测时，将指令地址同时送往子预测器T0~T4。T0提供基础预测结果；T1~T4则各自将指令地址与GHR1 ~ GHR4进行两种类型的hash映射，产生两个hash值h1和h2。h1用于查PHT得到pred、tag和usefulness，h2用于选择子预测器 —— 若h1查表得到的tag等于h2，则称相应的子预测器发生了 tag匹配。 若多个子预测器均发生了tag匹配，则在这些子预测器里，选择GHR位宽最大的作为provider，选择GHR位宽次最大的作为altpred；若T1~T4当中只有一个子预测器发生匹配，则选择发生匹配者为provider，选择T0作为altpred；若T1 ~ T4均无匹配，则选择T0作为provider和altpred。provider的预测结果将作为TAGE预测器的最终预测结果。

更新策略： 当分支指令的实际跳转行为被确定时，首先需要更新provider中发生tag匹配的entry。只有在provider的预测结果与altpred不相同时，才更新entry中的usefulness字段 —— 若provider预测正确，则令usefulness增加1，否则令其减小1。至于entry中的pred字段，其更新规则与一般的饱和计数器相同。 此外，为了防止某些entry被一直标记为useful，需定期对所有子预测器的全部usefulness的高、低位分别进行清零。除了更新provider和各子预测器的usefulness字段，当TAGE预测失败时，还要视情况进行entry的分配。分配时，首先寻找满足以下条件的子预测器：

a) GHR位宽比provider大

b) 对应entry的usefulness字段为零

在所有同时满足条件a)、b)的子预测器中，选出GHR位宽最小的子预测器进行更新。更新时，将对应entry的pred字段设置为"weaktaken"，并将usefulness字段清零。若所有满足条件a)的子预测器都不满足条件b)，则令这些子预测器的对应entry的usefulness字段减小1。

合理之处：充分考虑了不同程序的分支局部性，能够获得比较稳定的分支预测效果

# 2. 设计与实现

*\*注：若做了附加题，除了提交相应代码，还需按照指导书要求，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，否则不加分。*

## 2.1 方案设计

*详细阐述各个分支预测器模型的关键实现，建议结合算法流程图等图形化工具来辅助描述。*

1.基于BHT的分支预测

预测方法：

首先将地址addr截取适合长度的部分(m\_entries\_log位长)，再根据截取得到的值找到对应BHT的饱和计数器，返回饱和计数器当前预测分支跳转的结果

return (m\_scnt + truncate((addr >> 2), m\_entries\_log))->isTaken();

更新方法：

只利用分支实际是否跳转作为对应tag是否进行更新的判断依据，分支实际跳转则increase，否则decrease，其中increase(),decrease()函数做了一点修改，便于使用

if(takenActually)

(m\_scnt + truncate((addr >> 2), m\_entries\_log))->increase();

else if(!takenActually)

(m\_scnt + truncate((addr >> 2), m\_entries\_log))->decrease();

increase()：

void increase() {

if (m\_val == m\_init\_val - 1) m\_val = (1 << m\_wid) - 1;

else if (m\_val < (1 << m\_wid) - 1) m\_val++;

}

decrease()：

void decrease() {

if (m\_val == m\_init\_val) m\_val = 0;

else if (m\_val > 0) m\_val--;

}

2.基于全局历史的分支预测

初始化方法：

m\_entries\_log = entry\_num\_log;

m\_scnt = m\_saturatingCnt\_alloc.allocate(1 << entry\_num\_log);

m\_ghr = m\_shiftReg\_alloc.allocate(1);

for (int i = 0; i < (1 << entry\_num\_log); i++)

m\_saturatingCnt\_alloc.construct(m\_scnt + i, scnt\_width); m\_shiftReg\_alloc.construct(m\_ghr, ghr\_width);

预测方法：

首先将地址addr与当前GHR的值进行hash运算，将得到的hash值截取适合长度的部分(m\_entries\_log位长)，再根据截取得到的值找到对应BHT的饱和计数器，返回饱和计数器当前预测分支跳转的结果isTaken()

return (m\_scnt + truncate(hash(m\_ghr->getVal(), addr), m\_entries\_log))

->isTaken();

更新方法：

只利用分支实际是否跳转作为对应tag是否进行更新的判断依据，PHT更新方法与BHT一样，GHR也需要进行更新，直接利用移位寄存器的shiftIn方法传入分支实际是否跳转进行更新

if(takenActually)

(m\_scnt + truncate(hash(m\_ghr->getVal(), addr), m\_entries\_log))->increase();

else if(!takenActually)

(m\_scnt + truncate(hash(m\_ghr->getVal(), addr), m\_entries\_log))->decrease();

m\_ghr->shiftIn(takenActually);

3.锦标赛分支预测

初始化方法：

将参数的两个子预测器填入寄存器指针数组中，创建GSHR创建器，为GSHR分配空间

m\_BPs[0] = BP0;

m\_BPs[1] = BP1;

m\_gshr = m\_alloc.allocate(1);

m\_alloc.construct(m\_gshr, gshr\_width)

删除方法：

删除GSHR创建器，收回为GSHR分配的空间

m\_alloc.destroy(m\_gshr);

m\_alloc.deallocate(m\_gshr, 1);

预测方法：

根据全局历史预测器选择其中一个子预测器的预测结果作为返回

if (!m\_gshr->isTaken())

return m\_BPs[0]->predict(addr);

else

return m\_BPs[1]->predict(addr);

更新方法：

首先根据每个预测器的预测结果、实际分支跳转和指令地址传入每个预测器分别进行更新，当其中一个子预测器预测正确而另一个子预测器预测错误时进行GHR的更新

m\_BPs[0]->update(takenActually, m\_BPs[0]->predict(addr), addr);

m\_BPs[1]->update(takenActually, m\_BPs[1]->predict(addr), addr);

if(m\_BPs[0]->predict(addr) == takenActually && m\_BPs[1]->predict(addr) != takenActually)

m\_gshr->decrease();

else if(m\_BPs[0]->predict(addr) != takenActually && m\_BPs[1]->predict(addr) == takenActually)

m\_gshr->increase();

4.TAGE分支预测

对GHR预测器的修改：

UINT128 get\_tag(ADDRINT addr)

{

return truncate(hash(m\_ghr->getVal(), addr), m\_entries\_log);

}

UINT128 get\_ghr()

{

return (UINT128)m\_ghr;

}

void reset\_ctr(ADDRINT addr)

{

(m\_scnt + truncate(hash(m\_ghr->getVal(), addr), m\_entries\_log))->reset();

}

预测方法：

预测时，将指令地址同时送往子预测器T0~T4。T0提供基础预测结果；T1~T4则各自将指令地址与GHR1 ~ GHR4进行两种类型的hash映射，产生两个hash值h1和h2。h1用于查PHT得到pred、tag和usefulness，h2用于选择子预测器 —— 若h1查表得到的tag等于h2，则称相应的子预测器发生了 tag匹配。 若多个子预测器均发生了tag匹配，则在这些子预测器里，选择GHR位宽最大的作为provider，选择GHR位宽次最大的作为altpred；若T1~T4当中只有一个子预测器发生匹配，则选择发生匹配者为provider，选择T0作为altpred；若T1 ~ T4均无匹配，则选择T0作为provider和altpred。provider的预测结果将作为TAGE预测器的最终预测结果。

int tagNum = 0;

provider\_indx = 0;

altpred\_indx = 0;

for (size\_t i = 0; i < m\_tnum; i++)

{

m\_T\_pred[i] = m\_T[i]->predict(addr);

}

for (size\_t i = 1; i < m\_tnum; i++)

{

UINT128 hash =

truncate(hash2(m\_T[i]->get\_ghr(),addr),m\_entries\_log);

if(m\_T[i]->get\_tag(addr) == hash)

{

if(tagNum != 0)

{

altpred\_indx = provider\_indx;

}

provider\_indx = i;

tagNum++;

}

}

return m\_T\_pred[provider\_indx];

更新方法：

当分支指令的实际跳转行为被确定时，首先需要更新provider中发生tag匹配的entry。只有在provider的预测结果与altpred不相同时，才更新entry中的usefulness字段 —— 若provider预测正确，则令usefulness增加1，否则令其减小1。至于entry中的pred字段，其更新规则与一般的饱和计数器相同。 此外，为了防止某些entry被一直标记为useful，需定期对所有子预测器的全部usefulness的高、低位分别进行清零。除了更新provider和各子预测器的usefulness字段，当TAGE预测失败时，还要视情况进行entry的分配。分配时，首先寻找满足以下条件的子预测器：

a) GHR位宽比provider大

b) 对应entry的usefulness字段为零

在所有同时满足条件a)、b)的子预测器中，选出GHR位宽最小的子预测器进行更新。更新时，将对应entry的pred字段设置为"weaktaken"，并将usefulness字段清零。若所有满足条件a)的子预测器都不满足条件b)，则令这些子预测器的对应entry的usefulness字段减小1。

// TODO: Update provider itself对provider预测器的更新

m\_T[provider\_indx]->update(takenActually, takenPredicted, addr);

// TODO: Update usefulness对provider的usefulness的更新

if(m\_T\_pred[provider\_indx] != m\_T\_pred[altpred\_indx])

{

if(takenPredicted == takenActually && m\_useful[provider\_indx][truncate(hash2(m\_T[provider\_indx]->get\_ghr(),addr),m\_entries\_log)] < (1 << 8) - 1)

m\_useful[provider\_indx][truncate(hash2(m\_T[provider\_indx]->get\_ghr(),addr),m\_entries\_log)]++;

else if(takenPredicted != takenActually && m\_useful[provider\_indx][truncate(hash2(m\_T[provider\_indx]->get\_ghr(),addr),m\_entries\_log)] > 0)

m\_useful[provider\_indx][truncate(hash2(m\_T[provider\_indx]->get\_ghr(),addr),m\_entries\_log)]--;

}

// TODO: Reset usefulness periodically对usefulness的周期性刷新

m\_rst\_cnt++;

if(m\_rst\_cnt == m\_rst\_period)

{

m\_rst\_cnt = 0;

for (size\_t i = 1; i < m\_tnum; i++)

{

for(size\_t j = 0; j < (size\_t)(1 << m\_entries\_log); j++)

{

m\_useful[i][j] = m\_useful[i][j] >> 2;

m\_useful[i][j] = m\_useful[i][j] << 1;

}

}

}

// TODO: Entry replacement对entry的分配机制

if(takenPredicted != takenActually)

{

BOOL find = false;

for (size\_t i = (size\_t)(provider\_indx + 1); i < m\_tnum; i++)

{

if(m\_useful[i][truncate(hash2(m\_T[i]->get\_ghr(),addr),m\_entries\_log)] == 0)

{

m\_T[i]->reset\_ctr(addr);

find = true;

break;

}

}

if(!find)

{

for (size\_t i = (size\_t)(provider\_indx + 1); i < m\_tnum; i++)

{

if(m\_useful[i][truncate(hash2(m\_T[i]->get\_ghr(),addr),m\_entries\_log)] > 0)

{

m\_useful[i][truncate(hash2(m\_T[i]->get\_ghr(),addr),m\_entries\_log)]--;

}

}

}

}

**2.2 实验结果及分析**

*描述实验过程及各预测器的关键参数，阐述参数选择过程。计算各预测器实际大小。自行绘制并填写测试表，并对实验结果进行分析。*

*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）。*

关键参数

BP = new BHTPredictor(16);

BP = new GlobalHistoryPredictor<f\_xor>(64, 16);

BP = new TournamentPredictor(new BHTPredictor(15),new GlobalHistoryPredictor<f\_xor>(64, 15));

BP = new TAGEPredictor<f\_xor,f\_xnor>(8, 12, 1, 2, 12);

基于BHT的分支预测：

在空间分配不超过33KB>218位的基础上进行空间分配，默认分支历史为2位长，则分配16位数量的BHT表格数

基于全局历史的分支预测：

在空间分配不超过33KB>218位的基础上进行空间分配，默认分支历史为2位长，则分配16位数量的PHT表格数

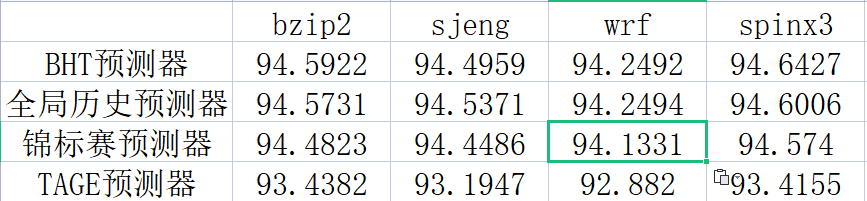
基于锦标赛的分支预测：

在空间分配不超过33KB>218位的基础上进行空间分配，为两个子预测器分配等额的空间(217)，默认分支历史为2位长，则分配15位数量的BHT表格数和15位数量的PHT表格数。

基于TAGE的分支预测：

在空间分配不超过33KB>218位的基础上进行空间分配，为8个子预测器分配等额的空间(215)，默认分支历史为3位长，则分配12位数量的BHT表格数和12位数量的PHT表格数。

预测器实际大小约为32KB：为各种预测器分配相同大小的空间



得到的结果中前三个预测器预测效果相近，而TAGE预测器预测效果稍微差一点，原因可能是预测器分配空间的限制问题，让TAGE预测器在保证了子预测器的等额空间分配之后，留给每个子预测器的实际空间并不多，所以导致TAGE的预测效果反而降低了