分支预测器的设计与实现

实验报告

学　　号：2016211392

姓　　名：张绍磊

班　　级：2016211310

课程名称：计算机体系结构

指导老师：黄智濒

目录

一、分支预测背景

当包含流水线技术的处理器处理分支指令时就会遇到一个问题，根据判定条件的真/假的不同，有可能会产生跳转，而这会打断流水线中指令的处理，因为处理器无法确定该指令的下一条指令，直到分支执行完毕。流水线越长，处理器等待的时间便越长，因为它必须等待分支指令处理完毕，才能确定下一条进入流水线的指令。

分支预测技术便是为解决这一问题而出现的。分支预测技术包含编译时进行的静态分支预测和硬件在执行时进行的动态分支预测。

静态分支预测：

最简单的静态分支预测方法就是任选一条分支。这样平均命中率为 50%。更精确的办法是根据原先运行的结果进行统计从而尝试预测分支是否会跳转。

任何一种分支预测策略的效果都取决于该策略本身的精确度和条件分支的频率。

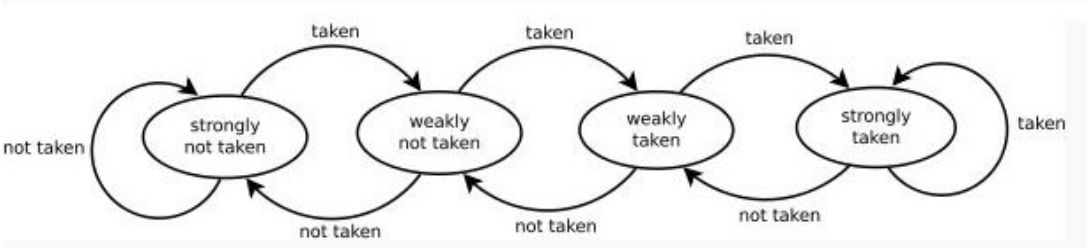
动态分支预测：

动态分支预测是近来的处理器已经尝试采用的的技术。最简单的动态分支预测策略是分支预测缓冲区（Branch Prediction Buff)或分支历史表(branch history table)。

二、现有分支预测器

2.1 饱和计数器

饱和计数器（saturating counter）或者称双模态预测器（bimodal predictor）是一种有 4 个状态的状态机:



2 位饱和计数器:

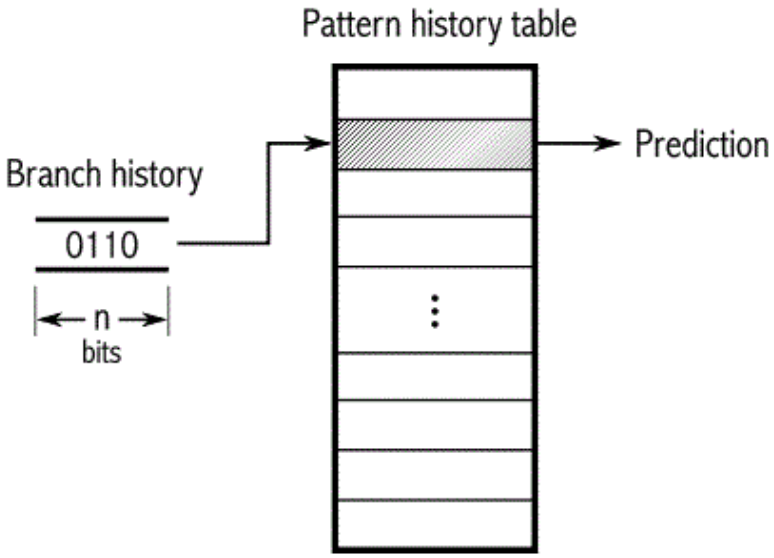
- 强不选择 Strongly not taken
- 弱不选择 Weakly not taken
- 弱选择 Weakly taken
- 强选择 Strongly taken

当一个分支命令被求值，对应的状态机被修改。分支不采纳，则向“强不选择”方向降低状态值；如果分支被采纳，则向“强选择”方向提高状态值。这种方法的优点是，该条件分支指令必须连续选择某条分支两次，才能从强状态翻转，从而改变了预测的分支。

最初的不具有 MMX 的 Intel Pentium 处理器使用了这种饱和计数器。虽然实现不够完美。在 SPEC'89 benchmark 测评中, 饱和预测达到了 93.5%正确率，如果每条条件分支指令都映射了自己的计数器。预测器表使用条件分支指令的地址

作为索引。因此处理器可以在分支指令解码前就给它分配一个预测器。

2.2 两级自适应预测器



两级自适应预测器。pattern history table 中的每个条目是上文的 2 位饱和计数器。

对于一条分支指令，如果每 2 次执行发生一次条件跳转，或者其它的规则发生模式，那么用上文提到的饱和计数器就很难预测了。如图所示，一种二级自适应预测器可以记住过去 n 次执行该指令时的分支情况的历史，可能的 2n 种历史模式的每一种都有 1 个专用的饱和计数器，用来表示如果刚刚过去的 n 次执行历史是此种情况，那么根据这个饱和计数器应该预测为跳转还是不跳转。

例如，n = 2。这意味着过去的 2 次分支情况被保存在一个 2 位的移位寄存器中。因此可能有 4 种不能的分支历史情况：00, 01, 10, 11。其中 0 表示未发生跳转，1 表示发生了分支跳转。现在，设计一个模式历史表 (pattern history table)，

有 4 个条目，对应于 2n= 4 种可能的分支历史情况。4 中历史情况的每一种都在模式历史表对应于一个 2 位饱和计数器。分支历史寄存器用于选择哪个饱和计数器供现在使用。如果分支历史寄存器是 00，那么选择第一个饱和计数器；如果分支历史寄存器是 11，那么选择第 4 个饱和计数器。

假定，例如条件跳转每隔 2 次执行就发生一次，即分支情况的历史串行是 001001001…。在这种情况下，00 对应的饱和计数器将是状态“强选择”（strongly taken），表明在两个 0 之后必然是出现一个 1。01 对应的饱和计数器将是状态“强不选择”（strongly not taken），表示在 01 之后必然是出现一个 0。这也同样适用于 10 状态。而 11 状态从未使用，因为不可能出现连续两个 1。

2 级自适应预测器的一般规则是 n 位分支历史寄存器，可以预测在所有 n 周期以内出现的所有的重复串行的模式。2 级自适应预测器的优点是能快速学会预测任意的重复模式。此方法 1991 年被提出。已经变得非常流行。以此为基础很多变种方法被用于现代微处理。

2.3 本地分支预测

本地分支预测（local branch predictor）对于每个条件跳转指令都有专用的分支历史情况缓冲区；模式历史表可以是专用的，也可以是所有条件分支指令共用。

Intel Pentium MMX, Pentium II, Pentium III 使用本地分支预测器，记录 4 位的历史情况，每条条件跳转指令使用专用的本地模式历史表，当然是包含 24= 16 个条目。对 SPEC'89 benchmark 测评，非常大的本地预测器的正确率达到 97.1%。

2.4 全局分支预测

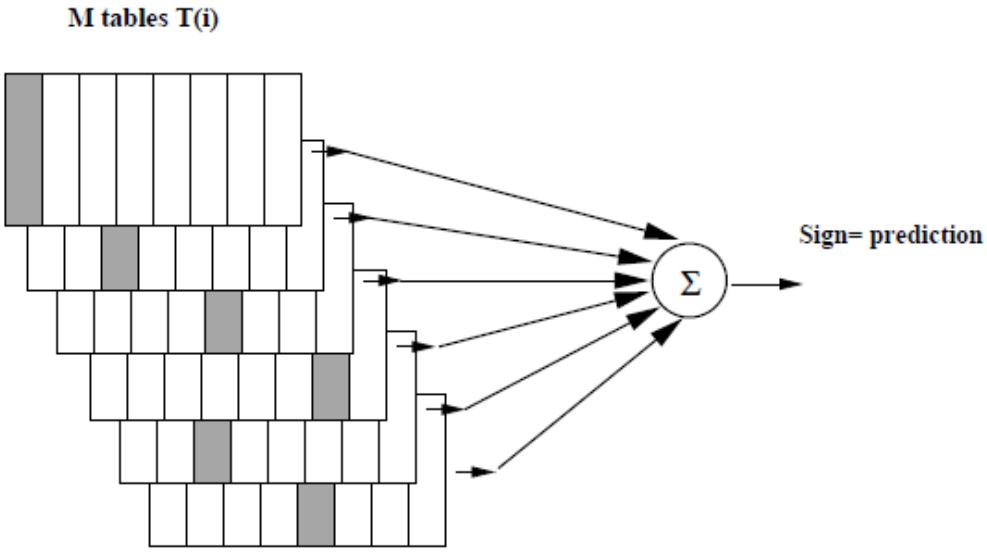


Figure 1: The GEHL predictor

全局分支预测器（global branch predictor）并不为每条条件跳转指令保持专用的历史记录。相反，它保持一份所有条件跳转指令的共用的历史记录。优点是能识别出不同的跳转指令之间的相关性。缺点是历史记录被不相关的不同的条件跳转指令的执行情况稀释了（diluted）。

这种方法只有在历史缓冲区足够长，才能发挥出性能。但是模式历史表的条目数是历史缓冲区位数的指数量级。因此只能是在所有的条件跳转指令间共享这个大的模式历史表。

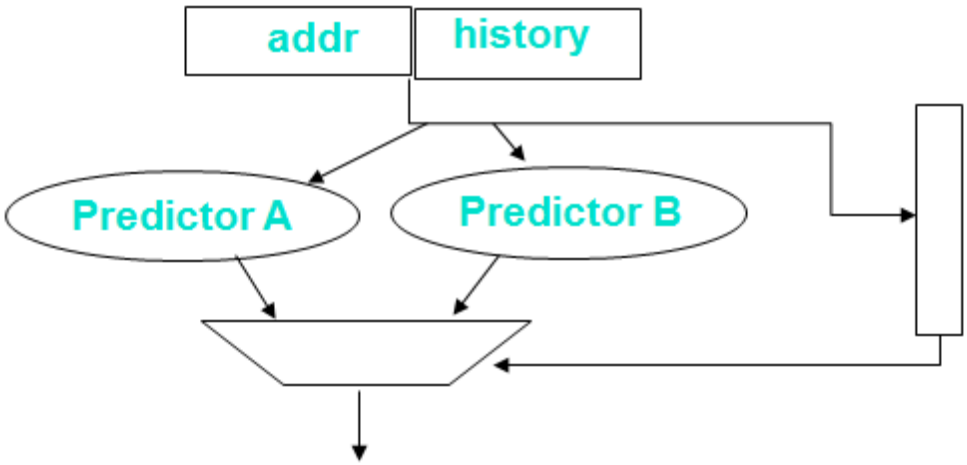
AMD 的 CPU，以及 Intel 的 Pentium M, Core，Core 2 使用了此种方法。SPEC'89 benchmark 评测，非常大的 gshare 预测器达到了 96.6%正确率，略低于本地分支预测。

2.5 融合分支预测

融合分支预测器（alloyed branch predictor）组合了本地与全局预测原理，把本地与全局的分支历史情况连接（concatenating）起来。VIA Nano 处理器可能采用此方法。

Tournament 预测器：整体局部自适应预测器

Tournament 预测器通过使用多个预测器-----其中一个基于全局信息，另一个基于局部信息，通过一个选择器将二者结合，而将这种方法又向前推进了一步。



Tournament 预测器的先进性在于，它可以为特定的转移选择正确的预测器，而这一点对于定点基准测试程序来说十分关键。Alpha 的 Tournament 预测器由局部转移地址索引的 4K 个 2bit 计数器，在全局预测器和局部预测器间进行选择。

全局预测器共有 4K 个入口，由最近执行的 12 个转移进行索引：其中每个入口为一个标准 2bit 预测器。局部预测器由两层组成。上层是由 1024 个 10bit 入口组成的局部历史表；每一个 10bit 入口对应这个转移最近 10 次的执行情况。

局部历史表中被选中的入口用来对一个表进行索引，这个表由 1K 个入口组成，每个入口为 3bit 饱和计数器，该计数器用来进行局部预测。这种组合共使用

了 4K*2 +4K*2+1K*10 + 1K*3=29K bit。错误率在 1.6%~4%之间。

2.6 神经分支预测器

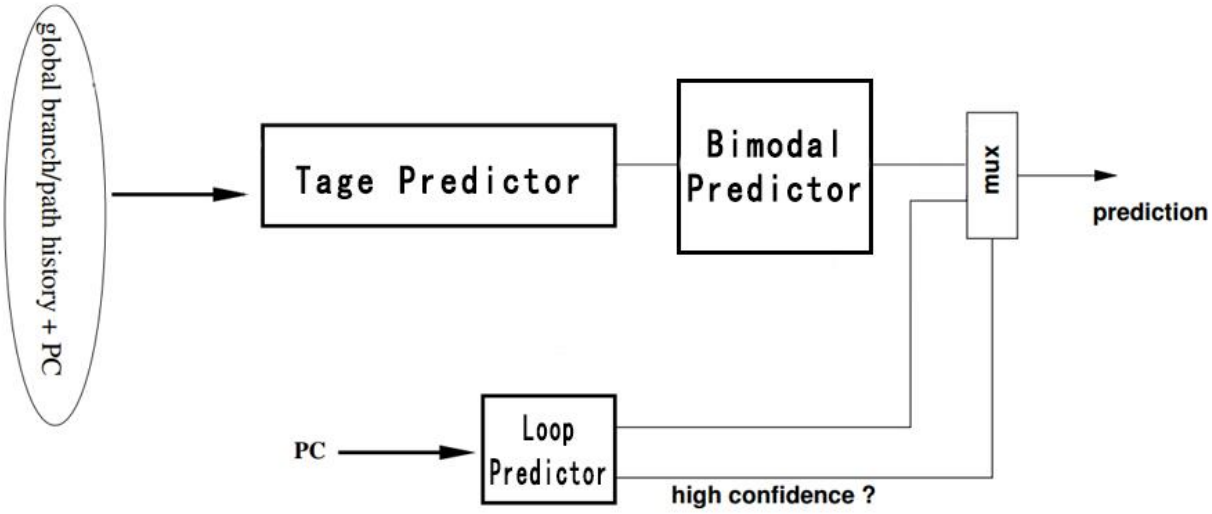
1999 年提出的神经分支预测器（neural branch predictor）。突出优点是能够利用很长的历史记录，仅导致了资源的线性增长。而传统预测器的资源需要量与历史长度是指数增长关系。这种方法主要缺点是高延迟。

神经分支预测器的准确度非常突出。（参见 Intel's "Championship Branch Prediction Competition" ）。Intel 在 IA-64 的模拟器 (2003)中实现了这一方法。

三、改进的算法

本文提供一种算法，将 Tage 算法、双模态预测器和 Loop 预测器相结合。Tage 预测器为主，Tage 预测器的结果输出控制双模态预测器在强不选择、弱不选择、弱选择、强选择四种状态中跳转。同时，Loop 预测器程序专门来预测循环的控制部分所对应的条件跳转指令。另外，我还加入了一个 clock 计时器，每 2 ^ clock_max 个分支重置所有表的 usefulness。用于限制相距很远的分支之间的冲突。最后，对整个算法相应的参数进行优化。

算法流程图如下所示：



接下来，3.1 节到 3.4 节将分别介绍 Tage 预测器、Bimodal 预测器、Loop 预测器、Clock 定时器的原理及实现方式。每部分具体参数将在第 4 节中给出。

3.1 Tage 预测器

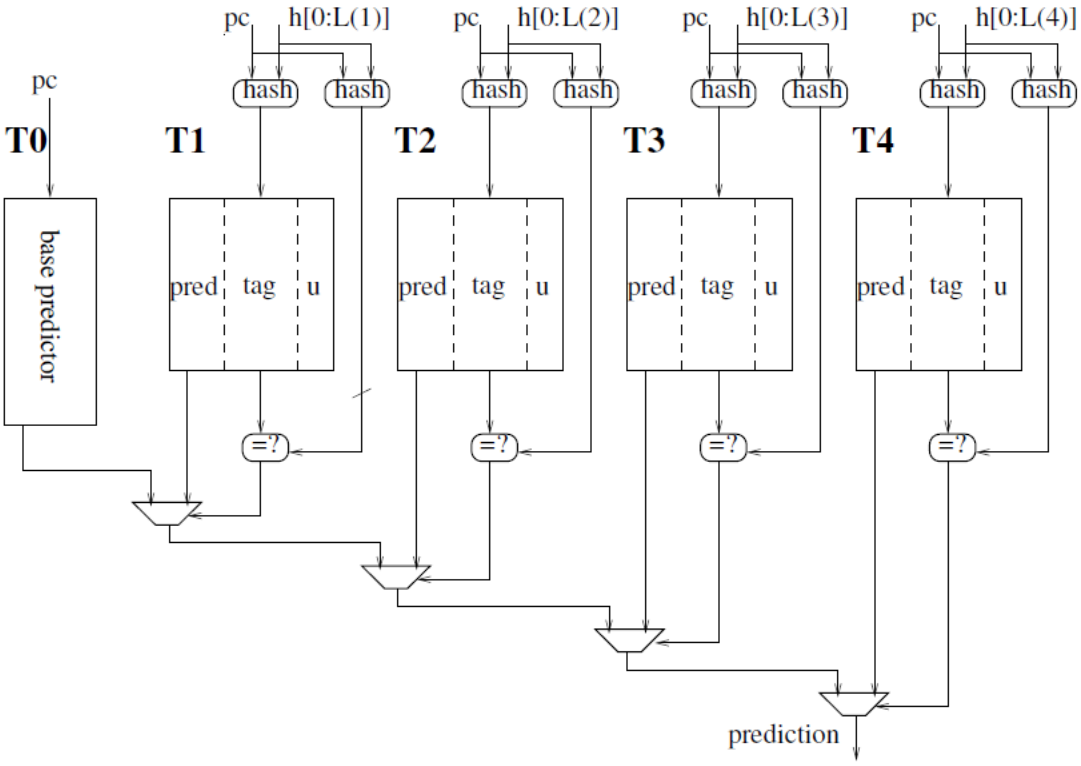


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

TAGE 预测器由一个 base 预测和 M 个 tagged 预测表 Ti（1<i<M）构成，base 是 PC 索引的 2 位计数器双峰预测。

索引 Ti 的历史长度满足几何级数序列 L(i) = (int)(alpha^(i-1)*L(1)+0.5)。Ti 中的一个 entry 包括 3 个域： ctr（提供预测值、带符号，3bit）、tag、u（有用位计数器、无符号，2bit）。

Provider：tage hit 中，由最长历史索引的部件。

Alternate：provider 以外的 tag hit 的部件。Tage 表中没有，就是 base 表。

【预测计算】给 pc，计算 taken/not taken

Base 和 taged 同时计算。使用 taged 表命中里面，历史信息最长的；如果没有，就用 base。存在 alternate 可能比 provider 更准确的情况。用 4 bit 计数器（USE ALT ON NA）来捕获这种全局现象。

```
1. Find the longest matching component and the alternate component

2. if (the confidence counter is not weak or USE ALT ON NA is negative)
       then the provider component provides the prediction
   else
       the prediction is provided by the alternate component
```

【更新 TAGE】

- （1）预测正确时

  更新 ctr，当预测的置信度低的时候，alt 也需要更新。

- （2）预测错误时

  1.更新 provider 的 ctr

  2.分配新 entry

- （3）更新 u

  预测正确自增，周期性重置。

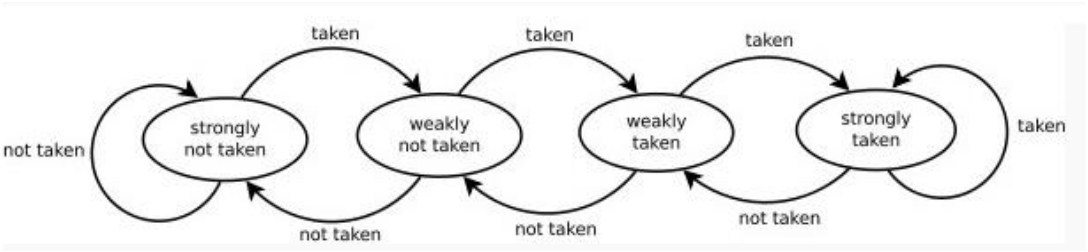
```
--更新 useful counter：

    prime 和 alt 预测值不一致才更新 useful bit，prime 和实际结果 resolve 一致，则增加，不一致就自减。每隔 256k 个 branch 需要重置 usefulbit。
        先 MSB，后 LSB，即 u=u&1，然后 u=u&2

--更新 counter value（ctr|pred）
    if resolve//结果 taken
      prime 的 ctr 自增，否则，自减。
     if 由 base 提供的预测值，同样，taken，base 对应的 ctr 自增，否则自减。

--检测当前给出预测值得 entry 是否非新分配。
```

```
    if prime.u==0&&(prime.ctr==3||prime.ctr==4)//ctr==100||011 并且 u=0，
则为新分配的 entry
        new_entry=true
        if primepred!=altpred
            if altpred==resolve//备选预测正确
                altbetterCount++
            else //备选预测失败
                alterbetterCount--

--处理新分配的 entry

    if !new_entry||(new_entry&&(primepred!=resolve))//非新 entry 或者预测失
败的新 entry
        if predDir! =resolve & 不是 base 提供的预测
            如果 primebank 的 useful bit 为 0，strong_old_present=true.
            if strong_old_present==false
                prime bank 以前的 usefulbit 自减。
            else
                生成随机数 randNo=rand（）%100；
                统计 primebank 以前 usefulbit 为 0 的 bank 数；
                bank_store 数组{-1，-1，-1}存储第 i 个 bank 是否 usefulbit 为 0；
                matchbank,匹配 bank，将要分配新 entry 的 bank；
                if count==1 matchbank=0；
                else if count>1
                    if randNo 在（33，99]之间，matchbank=bank_store[count-1];
                    else matchbank=bank_store[count-2];
                对 matchbank 之前的 bank
                    if bank[i]对应的 usefulbit==0
                        if taken .ctr 置为 4；else .ctr 置为 3；
                        置 tag 和 usefull bit
```

3.2 Bimodal 预测器

bimodal 预测器的输入为 Tage 预测器的输出结果。bimodal 预测器使用 2 比 bit 计数器。

4 个状态的状态机：



2 位饱和计数器：

- 强不选择 Strongly not taken
- 弱不选择 Weakly not taken
- 弱选择 Weakly taken
- 强选择 Strongly taken

根据 Tage 预测器的输出结果，如果分支不采纳，则向“强不选择”方向降低状态值；如果分支被采纳，则向“强选择”方向提高状态值。这种方法的优点是，该条件分支指令必须连续选择某条分支两次，才能从强状态翻转，从而改变了预测的分支。

3.3 Loop 预测器

程序循环的控制部分所对应的条件跳转指令最好用专门的循环控制器来预测分支。将要重复 N 次的循环的底部的条件跳转指令，前 N-1 次选择跳转，只有一次不跳转而是顺序执行。条件跳转指令有很多次选择了一条分支，只有一次选择另一分支，这种行为将被作为循环行为被检测。这种条件跳转指令可以用简单的计数器很容易地检测出来。循环预测器是一种混合预测器，其中元预测器判断该条件跳转指令是否具有循环行为。

具体算法如下：

```
log("Check loop");
//check loop counter
UINT32 loopTag = (PC) % (1<<LOOP_TAG_SIZE);
if(loopTable[loopIndex].tag == loopTag &&
        loopTable[loopIndex].currentIter <
loopTable[loopIndex].loopCount)
{ //if the loop is executing
        loopTable[loopIndex].pred = TAKEN;
}
else if(loopTable[loopIndex].tag == loopTag &&
                loopTable[loopIndex].currentIter ==
loopTable[loopIndex].loopCount)
{ //if loop is over
        loopTable[loopIndex].pred = NOT_TAKEN;
}
if(loopTable[loopIndex].tag == loopTag &&
        loopTable[loopIndex].conf == LOOP_CONF_MAX)
{ //if loop predictor is confident
        loopTable[loopIndex].used = true;       //use and return
        return loopTable[loopIndex].pred;
}
//if prediction hasn't been made, used = false
loopTable[loopIndex].used = false;
```

3.4 Clock 定时器

由于分支之间的关联程度随着分支间的距离变远会逐渐变小。我们发现在相距很远的分支之间，表中记录的 usefulness 反而对预测结果具有反作用。

所以，我加入了一个 clock 定时器。时钟每 2 ^ clock_max 个分支重置所有表的 usefulness。 这有助于限制相距很远的分支之间的冲突。

四、具体参数

经过不断调试、对结果分析，不断优化各预测器参数。

4.1 Tage & Bimodal Predictor

- 12 TAGE 表, 1 bimodal 表
- 每张表包含 2^9 到 2^11 entries
- 每个 entry 包含 7 到 15 tag bits, 3 pred bits
- 历史长度:

HIST_1	640	HIST_7	40
HIST_2	403	HIST_8	25
HIST_3	240	HIST_9	16
HIST_4	160	HIST_10	10
HIST_5	101	HIST_11	6
HIST_6	64	HIST_12	4

- 整体大小：237.5KB

4.2 Loop Predictor

- 共 512 entries
- 14 bit tag, 28 其他 bits

4.3 Clock

- 时钟每 2 ^25 个分支重置所有表的 usefulness

五、性能测试

5.1 测试方法

下载模拟环境及数据集：

1.下载 simulation infrastructure：

http：//hpca23.cse.tamu.edu/cbp2016/cbp2016.final.tar.gz

2.下载 training traces：

http：//hpca23.cse.tamu.edu/cbp2016/trainingTraces.Final.tar

3.下载 evaluation traces：

http：//hpca23.cse.tamu.edu/cbp2016/evaluationTraces.Final.tar

设置模拟基础架构：

1.打开套件包装。

tar -xzvf cbp2016.tar.gz

cd cbp2016

2.应该有五个目录：sim, scripts, traces, bin, and results。

3.需要安装 BOOST 库并将库链接添加到 sim 目录中的 Makefile。

sudo apt-get update

sudo apt-get install libboost-all-dev

sudo apt-get install g++

9. sim 目录包含模拟器。

cd sim

make clean

make

10. scripts 目录包含的脚本可以运行所有 223 条跟踪的预测器。查看 scripts 目录中的 doit.sh 文件。

cd ../scripts

./doit.sh

11.我们将 AMEAN 用于所有痕迹作为品质因数。使用 getdata.pl 脚本计算它。用法如下。

./getdata.pl -d ../results/new_traces*

5.2 测试集

测试集使用从 http://hpca23.cse.tamu.edu/cbp2016/trainingTraces.Final.tar 下载的 training traces。

共 LONG_MOBILE-1、LONG_MOBILE-4、SHORT_MOBILE-25、SHORT_MOBILE-2、SHORT_MOBILE-4、LONG_MOBILE-2、SHORT_MOBILE-1、SHORT_MOBILE-27、SHORT_MOBILE-30、LONG_MOBILE-3、SHORT_MOBILE-24、SHORT_MOBILE-28、SHORT_MOBILE-3 13 个测试集。

测试集共 5.32G，约 1 亿条指令。

5.3 测试结果

分别测试了原始 Tage predictor、Tage predictor + bimodal predictor、Tage predictor + bimodal predictor + clock、Tage predictor + bimodal predictor + Loop predictor、Tage predictor + bimodal predictor + Loop predictor + clock 五种情况下的性能，以对照不同组件对于性能优化的贡献程度。单位为 misp/KI，表示每千条指令错误的次数，越小表示性能越优。AMEAN 表示平均性能。

各性能如下所示：

原始 Tage predictor：

ResultDirs ==>	lts/new_traces
LONG_MOBILE-1	0.227
LONG_MOBILE-4	0.006
SHORT_MOBILE-25	0.002
SHORT_MOBILE-2	5.749
SHORT_MOBILE-4	5.380
LONG_MOBILE-2	1.422
SHORT_MOBILE-1	1.186
SHORT_MOBILE-27	0.032
SHORT_MOBILE-30	0.722
LONG_MOBILE-3	8.376
SHORT_MOBILE-24	0.001
SHORT_MOBILE-28	0.007
SHORT_MOBILE-3	3.124
AMEAN	2.018

Tage predictor + bimodal predictor：

ResultDirs ==>	lts/new_traces
LONG_MOBILE-1	0.180
LONG_MOBILE-4	0.005
SHORT_MOBILE-25	0.002
SHORT_MOBILE-2	3.779
SHORT_MOBILE-4	5.061
LONG_MOBILE-2	0.716
SHORT_MOBILE-1	0.909
SHORT_MOBILE-27	0.054
SHORT_MOBILE-30	2.425
LONG_MOBILE-3	7.374
SHORT_MOBILE-24	0.001
SHORT_MOBILE-28	0.008
SHORT_MOBILE-3	1.606
AMEAN	1.701

Tage predictor + bimodal predictor + clock：

ResultDirs ==>	lts/new_traces
LONG_MOBILE-1	0.176
LONG_MOBILE-4	0.005
SHORT_MOBILE-25	0.002
SHORT_MOBILE-2	3.618
SHORT_MOBILE-4	5.160
LONG_MOBILE-2	0.819
SHORT_MOBILE-1	0.893
SHORT_MOBILE-27	0.063
SHORT_MOBILE-30	2.390
LONG_MOBILE-3	7.270
SHORT_MOBILE-24	0.001
SHORT_MOBILE-28	0.008
SHORT_MOBILE-3	1.526
AMEAN	1.687

Tage predictor + bimodal predictor + Loop predictor：

ResultDirs ==>	lts/new_traces
LONG_MOBILE-1	0.051
LONG_MOBILE-4	0.005
SHORT_MOBILE-25	0.002
SHORT_MOBILE-2	0.665
SHORT_MOBILE-4	4.706
LONG_MOBILE-2	0.263
SHORT_MOBILE-1	0.860
SHORT_MOBILE-27	0.086
SHORT_MOBILE-30	0.347
LONG_MOBILE-3	7.009
SHORT_MOBILE-24	0.001
SHORT_MOBILE-28	0.009
SHORT_MOBILE-3	0.630
AMEAN	1.126

Tage predictor + bimodal predictor + Loop predictor + clock：

ResultDirs ==>	lts/new_traces
LONG_MOBILE-1	0.050
LONG_MOBILE-4	0.005
SHORT_MOBILE-25	0.002
SHORT_MOBILE-2	0.718
SHORT_MOBILE-4	4.689
LONG_MOBILE-2	0.255
SHORT_MOBILE-1	0.845
SHORT_MOBILE-27	0.063
SHORT_MOBILE-30	0.320
LONG_MOBILE-3	6.876
SHORT_MOBILE-24	0.001
SHORT_MOBILE-28	0.008
SHORT_MOBILE-3	0.647
AMEAN	1.114

六、结果分析

测试结果汇总如下表所示。

通过对比原始 Tage 和 Tage+bimodal，我们发现在 Tage predictor 后面加入 Bimodal predictor 在强不选择、弱不选择、弱选择、强选择四种状态中跳转，总体性能提升较高。但在 SHORT_MOBILE-27 、 SHORT_MOBILE-30 、 SHORT_MOBILE-28 中，性能反而下降。

通过对比 Tage+bimodal、Tage+bimodal +Loop，我们发现有些测试集提升较高，有些几乎没有提升，这可能是部分测试集中循环跳转较少，故 Loop predictor 对其性能提升较少。有些特殊的测试集则会因为加入 Loop predictor，产生错误判断。但整体上来看，错误率从 1.701 提高为 1.126，加入 Loop predictor 对性能还是有较大提升。

通过对比 Tage+bimodal 和 Tage+bimodal +clock、Tage+bimodal +Loop 和 Tage+bimodal +Loop+clock，我们发现对于原本正确率较高的测试集上，加入 Clock 定时器对性能提升不明显，甚至有反作用。但是对于原本正确率较低的测试集上，加入 Clock 定时器对性能提升显著，这是由于原本正确率较低的测试集上相距很远的分支之间的冲突较多，而 Clock 恰恰避免了这种冲突。

整体来看，Tage predictor+bimodal predictor +Loop predictor +clock 相比原始算法，性能提升显著。并取得了良好的性能测试结果，正确率达到 99.8886%

可见我们的算法已经达到了较优的性能。

	LONG_MOBILE-1	LONG_MOBILE-4	SHORT_MOBILE-25	SHORT_MOBILE-2	SHORT_MOBILE-4	LONG_MOBILE-2	SHORT_MOBILE-1
原始 Tage	0.227	0.006	0.002	5.749	5.380	1.422	1.186
Tage+bimodal	0.180	0.005	0.002	3.779	5.061	0.716	0.909
Tage+bimodal +clock	0.176	0.005	0.002	3.618	5.160	0.819	0.893
Tage+bimodal +Loop	0.051	0.005	0.002	**0.665**	4.706	0.263	0.860
Tage+bimodal +Loop+clock	**0.050**	**0.005**	**0.002**	0.718	**4.689**	**0.255**	**0.845**
	SHORT_MOBILE-27	SHORT_MOBILE-30	LONG_MOBILE-3	SHORT_MOBILE-24	SHORT_MOBILE-28	SHORT_MOBILE-3	AMEAN
原始 Tage	**0.032**	0.722	8.376	0.001	**0.007**	3.124	2.018
Tage+bimodal	0.054	2.425	7.374	0.001	0.008	1.606	1.701
Tage+bimodal +clock	0.063	2.390	7.270	0.001	0.008	1.526	1.687
Tage+bimodal +Loop	0.086	0.347	7.009	0.001	0.009	**0.630**	1.126
Tage+bimodal +Loop+clock	0.063	**0.320**	**6.876**	**0.001**	0.008	0.647	1.114

***加粗字体为每个测试集下，性能最优的算法的错误率。AMEAN 为在各测试集上的平均错误率。**

七、总结

本次实验中，花费了大量的时间进行分支预测相关论文研读，改进了得到了一种算法，设计各种实验不断优化，并且最终达到了不错的性能。

虽然实验中花费了大量时间，但是收获了许多关于计算机体系结构和分支预测的知识，是非常值得的。

分支预测（Branch Prediction）是现代处理器用来提高 CPU 执行速度的一种手段，其对程序的分支流程进行预测，然后预先读取其中一个分支的指令并解码来减少等待译码器的时间。分支预测器对现代处理器至关重要。

通过本次实验，我对分支预测有个更深一步的理解，并且对计算机体系机构更加清晰。同时，还收获了阅读英文文献的能力。同时，新自动手做实验，让自己收获颇丰。

八、参考文献

[1]《处理器分支预测研究的历史和现状》——冯子军 肖俊华 章隆兵

[2] Storage free confidence estimation for the TAGE branch predictor——Seznec, André

[3] A new case for the TAGE branch predictor——Seznec, André

[4] Bias-Free Branch Predictor——Dibakar Gope，Mikko H. Lipasti

九、源代码

见附件