

分支指令

表1. x86中的跳转指令

投机执行的动机

分支方向预测器

静态预测器

饱和计数器

图1. 预测表示意图。

一比特饱和计数器

图2. 一比特饱和计数器的状态转移图

两比特饱和计数器

图3. 两比特饱和计数器的状态转移图

多比特饱和计数器

图4. 三比特饱和计数器的状态转移图

两级预测器

全局两级预测器

图5. 全局两级预测器

局部两级预测器

图6. 局部两级预测器

基于偏置的预测器

Agree预测器

图7. Agree预测器

YAGS预测器

图8. Yags预测器

组合的分支预测器

Hybrid预测器

图9. Hybrid预测器

Bimod预测器

图10. Bimod预测器

Gskew预测器

图11. Gskew预测器

TAGE预测器

图12. TAGE预测器进行预测的过程

基于感知的分支预测器

小结

循环预测器

分支目标地址预测器

BTB

IBTB

RAS

补充

分支历史信息的编码方式

预测表单元的更新

伪LRU算法

表2. 伪LRU算法的访问和更新过程示例

设计实例

Godson2

Intel

P1

PMMX

PPro、P2和P3

P4

P4E

PM和Core2

Nehalem

Sandy Bridge和Ivy Bridge

图13. CNX003中的取指前预测表

图14. CNX003中的译码后预测过程

BTB

IBTB

TAGE

TAGE0

TAGE1-10

循环预测器

FBHT

RAS

FRAS

历史信息编码

分支指令

分支指令（Branch）是能够改变程序指令流的指令，例如 `JMP`, `Jcc`, `CALL`, `RET` 等。当处理器遇到分支指令的时候，可能需要改变取指指令的位置，而不是分支指令的下一条指令。

分支指令的执行结果包含两方面，分支指令是否改变指令流以及取指的地址。如果分支指令改变了指令流，称为**分支发生**（Taken）；如果分支指令没有改变指令流，称为**分支不发生**（Not-taken）。在后文中，分支发生和不发生以英文 Taken (T) 和 Not-taken (NT) 表示。取指的地址称为**目标地址**（target address），但是更多的时候，特指分支发生时的取指地址。分支不发生时的取指地址称为**通过地址**（fall-through address）。

- **非条件分支**（Unconditional Branch）：当处理器遇到非条件分支指令的时候，一定会改变指令流，分支一定发生（Taken）。
- **条件分支**（Conditional Branch）：当处理器遇到条件分支指令的时候，是否改变指令流取决于一些特定的条件，比如上一条指令计算的结果。如果分支指令的条件得到满足，则改变指令流，分支发生（Taken）；如果条件不满足，则不改变指令流，分支不发生（Not-taken）。

继续，如果按照分支指令的目标地址来源，分支指令可以分为直接跳转分支和间接跳转分支：

- **直接跳转分支**（Direct Branch）：目标地址来源于指令本身。指令机器码中直接编码了目标地址相对于当前取指位置的偏移。当前取指位置加上偏移，可以得到目标地址。
- **间接跳转分支**（Indirect Branch）：目标地址来源于寄存器或内存。指令的机器码中只编码了目标地址的寄存器或内存指针。

这两种划分方法是完全正交的，也就是说，指令可以进一步分为四种组合。一种指令集可以完全支持这四种组合，也可以支持部分。比如x86指令集就只支持了三种，不支持间接跳转的条件分支。因为x86指令集的条件分支（`Jcc`）只支持偏移寻址，不支持寄存器和内存寻址。MIPS指令集与x86指令集类似。ARM指令集只支持直接跳转分支。如果需要间接跳转，可以直接修改指令计数器PC。

x86指令集中的各条分支指令的行为都不太一样，每一条指令都有自己的名字。

表1. x86中的跳转指令

指令	名称	发生与否	目标地址	边际影响
JMP	跳转	无条件分支	直接或间接跳转	
JL	远跳转	无条件分支	间接跳转	
JCC	条件跳转	条件分支	直接跳转	
CALL	调用	无条件分支	直接或间接跳转	返回地址压栈。
RET	返回	无条件分支	间接跳转	栈顶作为目标地址，出战。

在后文中，指令全部用其英文缩写表示。

投机执行的动机

现代处理器都以流水线的方式完成指令的取指、译码和执行。完成当前周期的取指之后，会直接给取指地址累加得到下一个周期的取指地址。比如在x86中，取指一个取指单元（16B）之后，取指地址直接加16。如果处理器遇到分支指令，由于分支指令是否Taken以及目标位置还未知，所以无法在当前周期就确定新的取指地址，必须要等待这条指令译码或执行完成，才能够确定下一条指令的取指地址。随着流水线深度的增加，分支指令会引入越来越长的流水线气泡，对性能的影响也越来越大。

为了降低跳转指令的代价，设计师试图投机执行一些指令，也就是在跳转指令没有执行完成的这段时间，流水线继续取指、译码和执行那些可能会被执行的指令。这种行为称为投机（Speculation）。如果分支指令确定的取指地址与投机执行的地址是一致，称为投机正确，那么就可以保留这些投机的指令，这些指令就成为了有效的执行；反之，称为投机错误，处理器丢弃这些投机指令，从正确的取指地址重新开始，称为流水线刷新（flush）。

投机执行是不会引入额外的性能代价的。如果投机指令正好是需要执行的指令，投机执行可以显著降低投机指令的性能损失，提高指令吞吐率。如果投机指令不是要执行的指令，那么也不会再引入新的延迟。投机执行会引入额外的功耗。额外的功耗是由错误的投机执行指令引起的。所以，如果投机正确的比例越高，系统性能提升越高，引入的功耗代价越小。

为了能够获得更高的投机正确比例，设计师在处理器中加入了投机预测器。这种预测器会保留分支指令的记录，包括分支指令的位置、目标地址、T或者NT。这个记录称为**分支历史信息**。当遇到分支指令时，预测器会根据分支历史信息对最可能的分支方向（Taken或Not-taken）和目标地址进行预测。如果预测的分支方向和目标地址与真实情况相同，称为**预测正确**。预测正确的分支占所有执行的分支的比例称为预测正确率。显然，投机预测器的预测正确率越高，对于系统性能越好。目前，我们实测得到的预测正确率可以达到90%以上。

不过分支预测器会引入一些功耗和代价，尤其是记录分支历史的分支预测表。所以，投机预测器的设计是投机正确率和投机预测表大小的平衡。投机正确率越高，可以提高系统性能、降低错误投机的功耗，但是常常需要更多的预测表，引入更多的面积和由于访问预测表引起的功耗。不过，目前总趋势还是追逐更高的预测正确率。

分支预测期的发展完全遵循了由简单到复杂的预测，一方面是预测表的复杂度增加；另一方面是预测算法复杂度的增加。而现在CPU中使用的预测器，则是由简单预测器通过组合得到的。由于分支指令需要确定分支方向和目标地址，所以分支预测器的发展脉络也可以归纳为两条线：方向预测器和地址预测器。两条线有交叉，但是基本独立。当前分支预测器的特点可以归纳为：按照不同分支类型采用不同分支预测器；利用尽可能多的分支历史信息来逼近各种各样的场景。

分支方向预测器

静态预测器

静态预测器认为所有的分支都是not-taken的，因此这种预测器也被称为**always-not-taken预测器**。从处理器的角度来说，静态预测器可以认为没有预测。不论是否遇到分支指令，处理器都按照线性取指。如果分支指令的执行结果是Not-taken，则不影响处理器运行；反之，则刷新流水线，从正确的目标地址开始。

显然，静态预测器的缺陷很明显：无条件跳转（JMP, CALL, RET）全部预测错误；只有一部分非条件跳转（Jcc）可以预测正确。这样的结果显然不如人意，因为我们知道无条件跳转是一定Taken的。

因此，将静态预测器的方向设置为Taken显然是一个更好的选择，这就是**always-taken预测器**。

always-taken预测器认为所有的指令都Taken。always-taken可以正确预测所有的无条件跳转，并且可以较好地预测在循环中使用的条件跳转（这种条件跳转一般都是Taken的）。不过always-taken对于条件分支仍然不是一个很好的结果。

较之与always-not-taken和always-taken的死板，有一种静态预测器要稍微灵活一下。这种预测器的思路是，如果是构成循环的跳转，则认为总是taken的；反之则认为是not-taken的。由于构成循环的跳转，大概率都是跳回到某个执行过的地方，也就是说偏移为负。所以，这种预测器根据偏移的正负选择不同的跳转方向。

- 如果分支指令的偏移是正的，一般来说是跳过一些指令再执行，预测器认为分支指令not-taken；
- 如果分支指令的偏移是负的，一般来说是跳回到执行过的地方，预测器认为分支指令taken。

所以，这种预测器可以称为Backwards-taken-forwards-not-taken（BTFTNT）。不过这种分支预测器要求偏移信息才能判断，所以只能在译码之后才能使用。

饱和计数器

为了提高条件跳转（Jcc）的预测正确率，设计师研究了程序的分支序列，发现对于同一个分支，分支的Taken和Not-taken呈现出一些稳定的模式。因此，设计师们试图通过各种方式将这些分支历史模式（history pattern）提取出来，从而提高预测正确率。

饱和计数器就是第一种通过提取分支历史模式来提高预测率的方法。其核心想法是按照前一次或几次的方向进行预测。计数器的数值分为高一半和低一半。高一半表示Taken；低一半表示Not-taken。如果分支是Taken，计数器加1；反之计数器减1。如果计数器达到了最大值或最小值，那不能再增加和减少，也就说不能发生溢出，所以被称为**饱和计数器**。

预测器为每一个分支提供一个包含几个比特的计数器。曾经有过整个处理器只有一个计数器的设计，但是这显然过于简单，无法呈现效果。这些饱和计数器构成一个**预测表**。这是我们首次提到预测表，如图1所示。预测表与Cache类似，只为最近经常使用的分支提供预测器，例如饱和计数器。预测表采用分支指令地址的低位比特作为索引，高位比特作为标签。一般来说，表示一个取指单元或指令内偏移的低位比特会被忽略。比如对于x86指令来说，低4位是被忽略的。

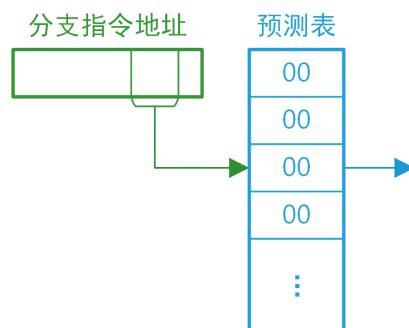


图1. 预测表示意图。

预测表访问过程分为**预测**和**更新**两个过程。

- 预测过程：如果分支在预测表中，根据预测表分配给这个分支的饱和计数器进行预测；反之，提供某个设定的默认值，一般是always-not-taken。
- 更新过程：如果分支在预测表中，根据分支的真实跳转方向更新预测表分配给这个分支的饱和计数器；反之，则在预测表中为其分配一个单元（可以占用新的单元或者取代一个已有的分支），并更新这个饱和计数器。

一比特饱和计数器

顾名思义，饱和计数器只有两个状态0和1。如果计数器为0，预测为Not-taken；如果计数器为1，预测为Taken。如果分支的真实跳转方向为Taken，计数器由0变1，或者保持1；反之，计数器由1变0，或者保持0。图2展示饱和计数器的状态转移图。

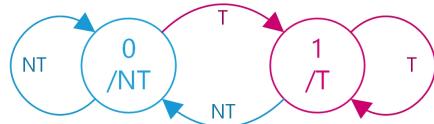


图2. 一比特饱和计数器的状态转移图

简单来说，一比特饱和计数器就是按照上一次的跳转方式来预测这一次的跳转方向。上一次分支发生Taken，这一次预测为Taken；上一次分支不发生Not-taken，这一次预测为Not-taken。

两比特饱和计数器

一比特饱和计数器已经可以提高预测正确率了。但是，单比特饱和计数器的高干扰能力比较差。比如，对于循环使用的分支，绝大多数情况下都是Taken，只有在跳出循环的时候才是Not-taken。如果用单比特饱和计数器，则会发生两次分支预测错误：第一次是在跳出循环的时候，分支被预测为Taken因为之前的分支都是Taken，而实际上分支是Not-taken；第二次是再次进入循环的时候，分支被预测为Not-taken，而实际上分支是Taken。

为了获得比较稳定的预测结果，设计师试图增加饱和计数器的宽度，比如两比特饱和计数器，如图3所示。两比特饱和计数器由4个状态，状态2和3预测为Taken，状态0和1预测为Not-taken。如果分支的真实跳转方向为Taken，计数器累加直到状态3，并保持；反之，计数器累减直到状态0，并保持。

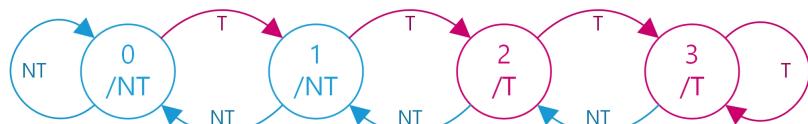


图3. 两比特饱和计数器的状态转移图

两比特饱和计数器是目前最为常用的预测器。后面介绍的复杂的预测器中，都使用饱和计数器作为最终的预测器。

多比特饱和计数器

理论上，饱和计数器的宽度可以继续增加，比如图4中的三比特饱和计数器。不过，过多的状态可以使预测方向过于稳定，而无法反映出分支的模式。如果要使得两比特饱和计数器的预测结果从Taken变为Not-taken，最多需要两个连续的Not-taken（从状态3到状态2再到状态1）。如果要使得三比特饱和计数器的预测结果从Taken变成Not-taken，则最多需要四个连续的Not-taken（从状态7到状态6再到状态5到状态4再到状态3）。

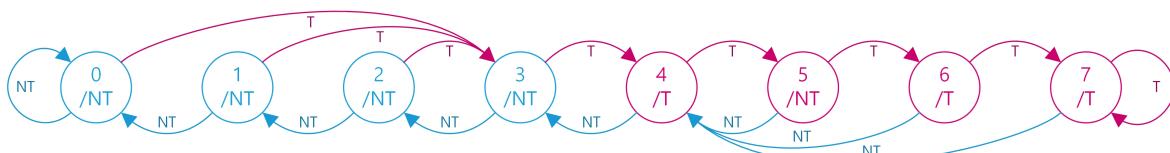


图4. 三比特饱和计数器的状态转移图

为了提高预测的灵活性，对于多比特饱和计数器，会引入快速改变的技术。如图3所示，如果在状态5、状态6和状态7遇到Not-taken，则会直接跳到状态4，而不需要经过状态7到状态6到状态5再到状态4。图4只是一个例子。快速改变的路径设置也是一个很tricky的话题。

不过，即便再复杂的饱和计数器，对于提取分支历史模式的能力还是有限的。N个比特的计数器，只能最多反应N个历史信息（N次分支方向）。所以，预测正确率的提升也是有限的。

两级预测器

饱和计数器的缺陷在于，少量的计数器位数无法反映较长的分支历史信息。另一方面，即便增加计数器位数，也无法进一步反映出分支历史信息的内在规律，比如定长循环的次数以及嵌套循环等等。对此，设计师的解决方案是直接在预测表的索引中引入分支历史。

对于饱和计数器时代的预测表来说，分支指令地址直接产生访问预测表的索引。到了两级计数器时代，分支指令与分支历史信息的组合才会用作访问预测表的索引。也就是，同一个分支指令，可能由于执行时的分支历史信息不同，占用不同的预测表单元。比如一条构成循环的跳转指令，每一次执行这条分支都会占用一个不同的单元。第二次执行这个循环的时候，会与之前占用的单元一一吻合，从而对每一条执行都进行准确预测。

分支指令地址和分支历史信息的组合方式可以是组合或者异或。

- 组合：访问预测表的索引将分支指令地址中的一些比特和分支历史信息中的一些比特组合而成。
- 异或：分支指令地址的一些比特与分支历史信息的一些比特进行异或。这种方式可以在索引中纠缠更多的地址信息和分支历史信息，所以现在大部分都是使用这种方法。

对于预测表中的每一个单元，仍然使用饱和计数器进行预测。也就是对于每一个地址和分支历史信息的组合，预测表都提供了饱和计数器进行预测和更新。

相对于饱和计数器，两级预测器由于延长了历史信息的长度，同时将同一条分支指令的不同次执行区分为不同的预测表单元，所以可以分析更加复杂的分支历史模式。两级预测器的能力除了取决于预测表大小，还取决于分支历史信息的长度。

按照分支历史信息的组织方式，两级预测器可以分为全局两级预测器和局部两级预测器。

全局两级预测器

全局两级预测器只提供一个分支历史信息，其中包含了处理器执行的所有分支指令的历史信息，如图5所示。访问预测表的索引是分支指令地址与全局历史信息的异或（或组合）。

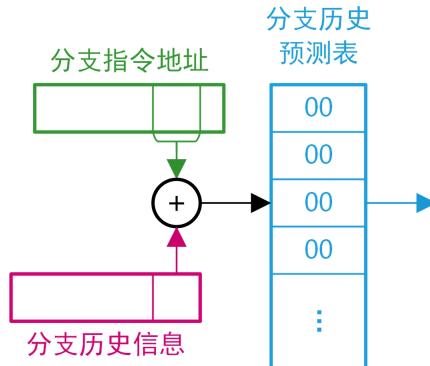


图5. 全局两级预测器

全局两级预测器，可以简称为全局预测器。分支指令地址和分支历史信息采用异或方式进行组合的方法，又称为**Gshare**。

局部两级预测器

局部两级预测器为不同的指令分别提供历史信息。除了预测表，局部两级预测器还要维护一个分支历史信息表，如图6所示。分支历史信息表以分支指令地址为索引，保存这个地址上的分支指令引入的历史信息。访问预测表的索引是分支指令地址与这个地址的历史信息的组合，从而屏蔽了不同分支指令的历史信息的相互干扰。当然，付出的代价就是更大的面积。

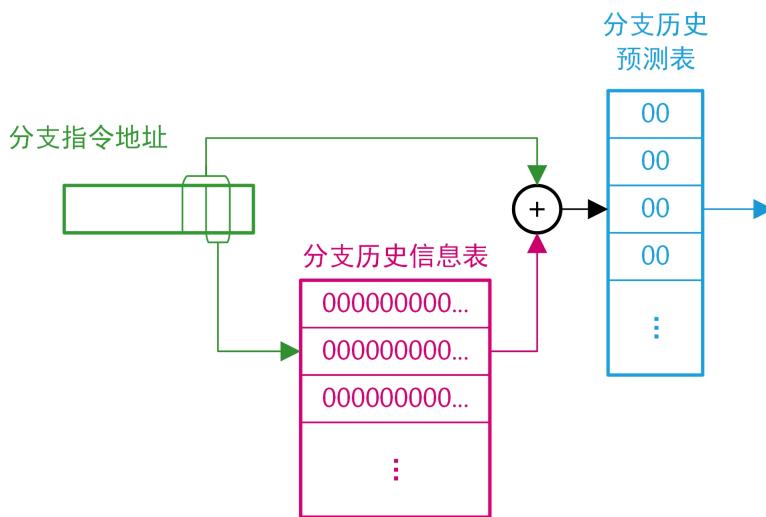


图6. 局部两级预测器

局部两级预测器可以简称为局部预测器。

参考文献：

- Tse-Yu Yeh and Yale N. Patt, Two-level adaptive training branch prediction, International Symposium on Microarchitecture, 1991.
- Tse-Yu Yeh and Yale N. Patt, Alternative Implementation of Two-Level Adaptive Branch Prediction, the 19th Annual International Symposium on Computer Architecture, 1992.
- Tse-Yu Yeh and Yale N. Patt, A comparison of Dynamic Branch Predictors that use Two Levels of Branch History, the 20th Annual International Symposium on Computer Architecture, 1993.

基于偏置的预测器

通过对分支历史信息的研究，设计者发现分支指令的跳转方向呈现非常明显的偏置（bias），就是大概率呈现Taken或Not-taken，只有在一小部分情况出现相反的方向。因此，可以用预测分支指令跳转方向是否与偏置一样，代替直接替换分支方向，从而避免别名。

Agree预测器

Agree预测器用bias表记录了分支的偏好（图7中紫色部分），分支指令地址直接索引，因为这是指令本身的属性，与分分支历史信息无关。分支历史预测表需要预测的不是分支的方向，而是是否同意预测表中记录的分支方向。如果分支历史预测表预测为Agree（状态2或状态3），则预测结果就是记录的bias；反之，取bias的反方向。

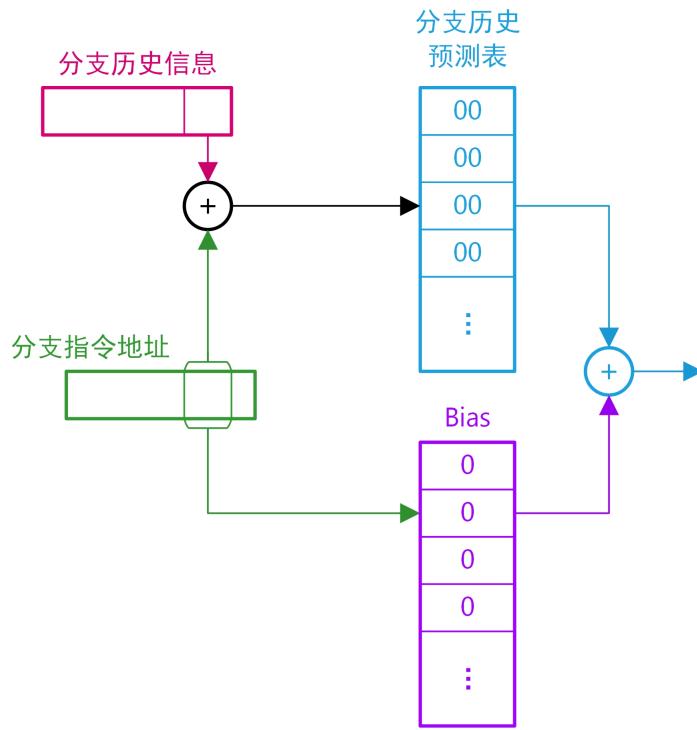


图7. Agree预测器

如果两个分支在bias中重名，那么可以利用分支历史预测表区分；如果在分支历史预测表中重名，可以利用bias区分。从而降低了预测表重名的概率。

Bias作为一个定值，设置为第一次见到这个分支的时候，分支的跳转方向。因为这个这个方向的正确率很高。分支历史预测器根据预测正确或者错误进行更新。

参考文献：Eric Sprangle, Robert S. Chappell, Mitch Alsup, Yale N. Patt, The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference, the 24th Annual International Symposium on Computer Architecture, 1997.

YAGS预测器

YAGS是Yet Another Global Scheme的缩写。其核心思路类似于Agree预测器，都是对一个bias提供同意和不同意的判断。Agree预测器通过一个预测表来进行，每一个分支地址和分支历史信息的组合都需要提供一个单元。而YAGS则只对不同意bias的分支地址和分支历史信息进行保存。通俗一点来说，就是查特例。

YAGS由一个选择预测表和两个cache组成，如图8所示。选择预测表由分支地址索引，提供分支的bias方向。如果bias方向为taken，则利用分支地址和分支历史的hash结果索引NT cache；反之，则利用相同的结果索引T cache。如果NT或者T cache命中，则表示当前分支历史是当前分支的特例，不同意bias，以bias的反方向位最终预测结果；反之，则以bias为最终预测结果。

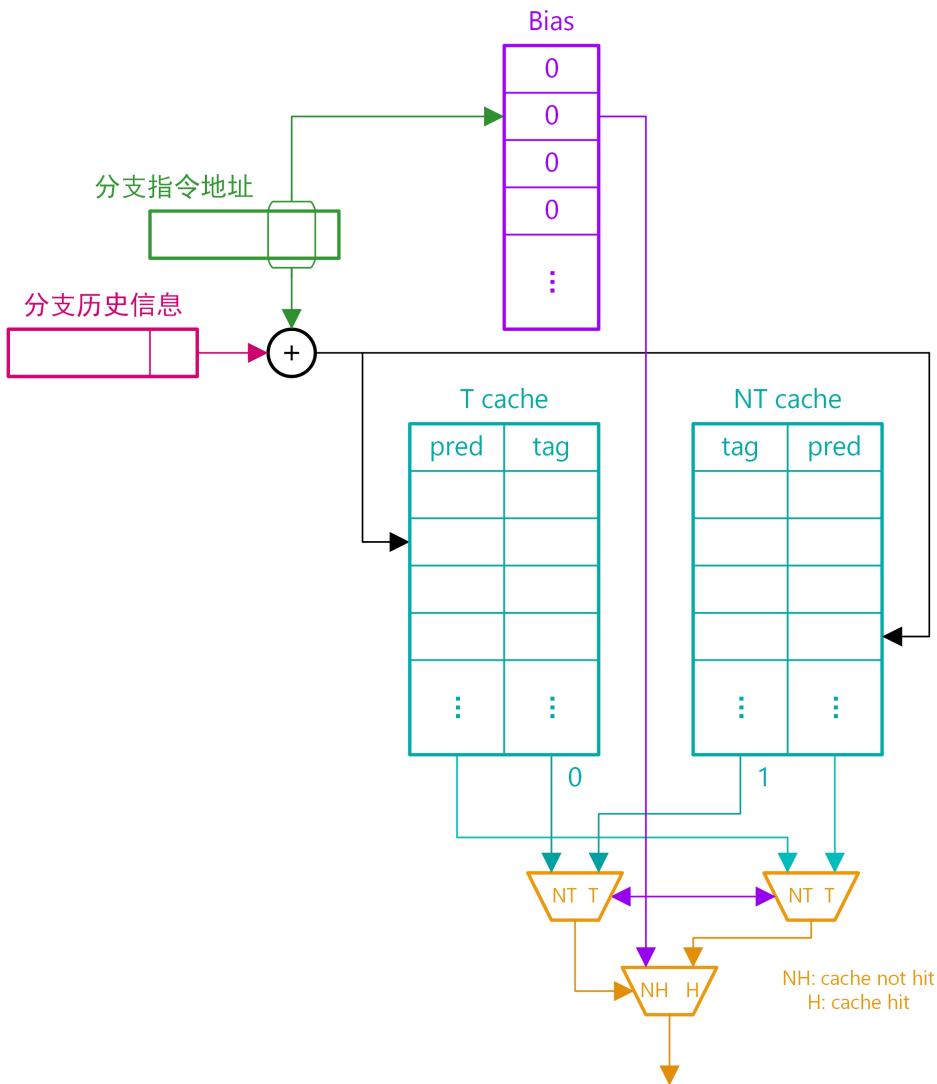


图8. Yags预测器

YAGS可以减少存储结构的要求，降低资源代价。

参考文献：A. E. Eden and T. Mudge, The YAGS Branch Prediction Scheme, the 31st annual ACM/IEEE international symposium on Microarchitecture (MICRO 31), 1998.

组合的分支预测器

组合式的分支预测器是值得有饱和计数器、两级预测器等组合而成的计数器。其主要设计思想是避免预测表单元的别名（alias）。由于预测表单元是有限的，那么一定会出现不同的分支指向同一个预测表单元的情况，会同时损害所有分支的预测正确率。

Hybrid预测器

Hybrid预测器的想法是，同时提供全局二级预测器和局部二级预测器。通过一个元预测器（meta）来选择使用哪一个预测器进行预测。图9是Hybrid预测器的示意图。

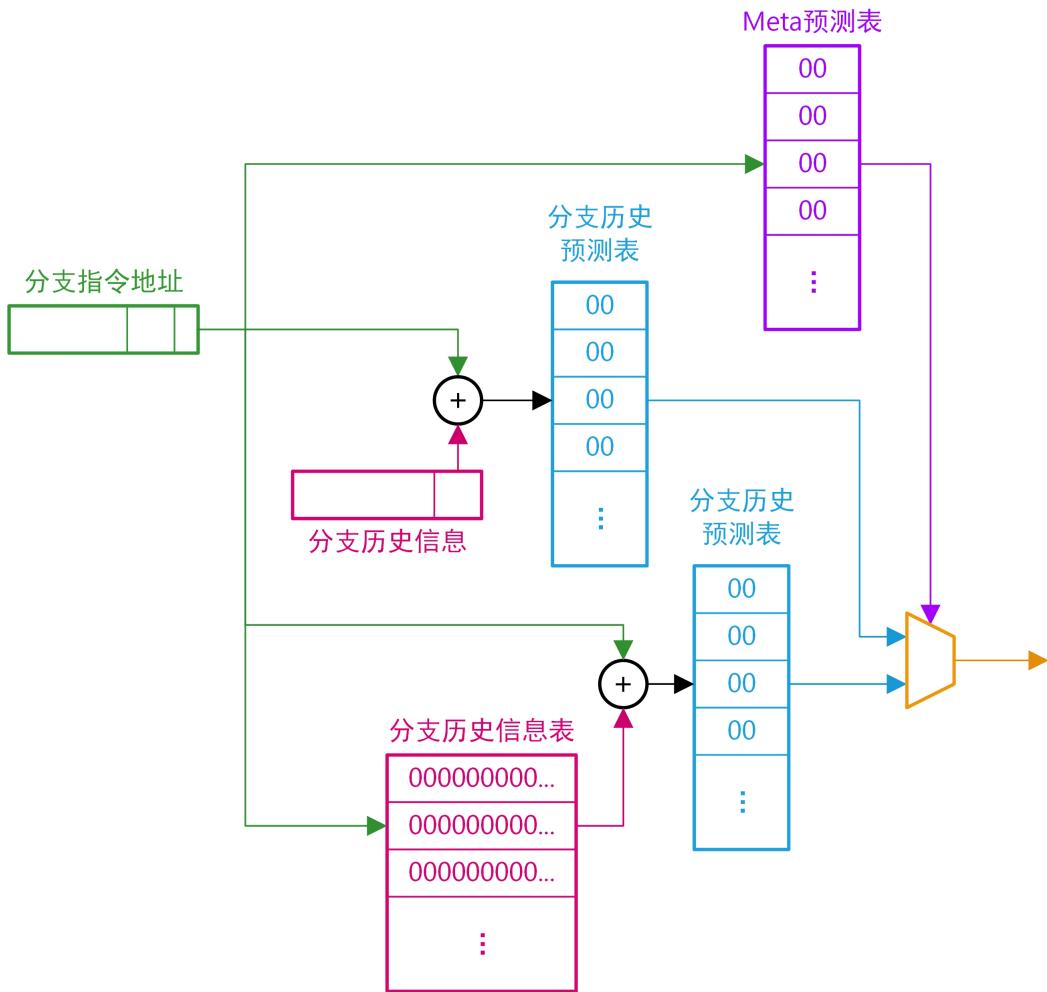


图9. Hybrid预测器

当发生预测错误的时候，更新分为两个步骤。首先，更新预测使用的全局或局部二级预测器。然后，更新Meta预测器。

- 如果Meta选择的预测器预测错误，而没有选择的Meta预测器预测正确，下一次采用另一个预测器预测。
- 如果两个预测器都预测错误，那么不调整Meta预测器。

Bimod预测器

Bimod预测器与Hybrid类似。不同在于，Bimod预测器的左右两个部分都是由全局两级预测器构成（如图10所示），而不是一个全局两级预测器和一个局部两级预测器。全局两级预测器由分支指令地址和分支历史信息的Hash结果索引，两个预测器的索引相同。选择预测器由分支指令地址索引，从两个全局预测器中做出选择。

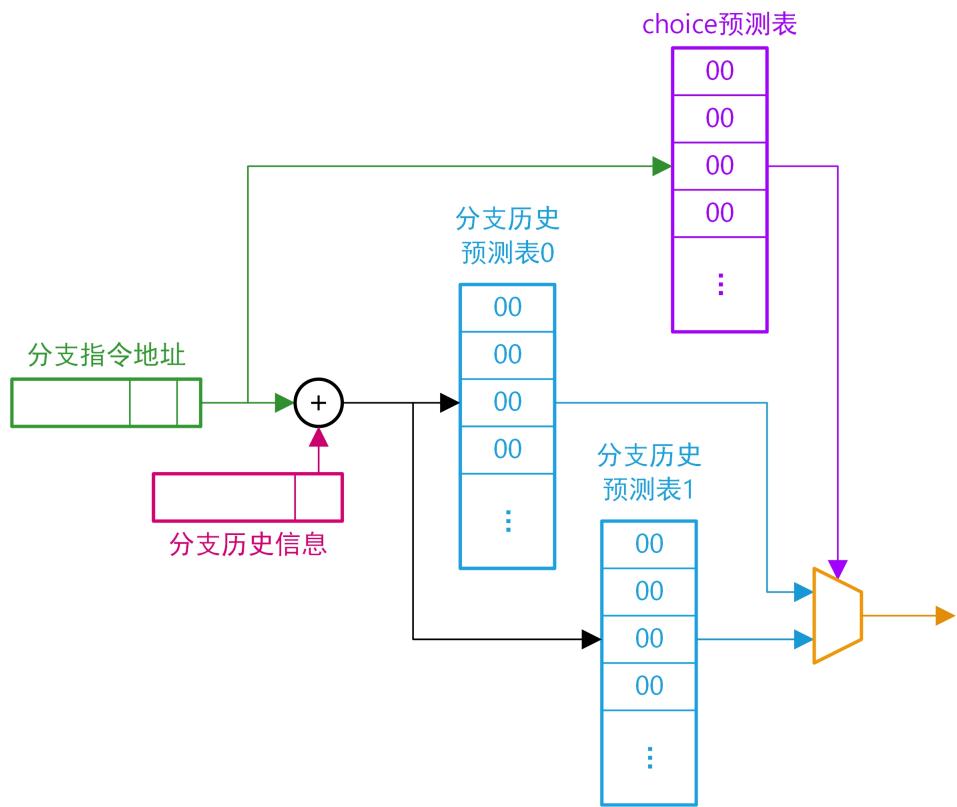


图10. BiMod预测器

当发生预测错误的时候，更新分为两个步骤。首先，更新预测使用的全局预测器。选择预测器则总是更新的。如果预测错误，下一次就倾向于采用另一个全局预测器。

参考文献：Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N.Mudge. The Bi-Mode Branch Predictor. 13th IEEE/ACM International Symposium on Microarchitecture, 1997.

Gskew预测器

Gskew预测器的思路就是三模冗余或者多模冗余。如图11所示，Gskew同时提供多个全局预测器，每个预测器的索引Hash函数不同。全局预测器的结果通过多数表决决定最后的预测结果。对于某一个分支，三个全部预测器同时遇到别名的概率就很小。只要有两个预测器没有遇到别名，那么预测结果就一定是针对这个分支的（不一定正确，但是一定是这个分支的历史记录训练出来的）；如果只有一个预测器没有遇到别名，那么也有三分之一正确的可能。

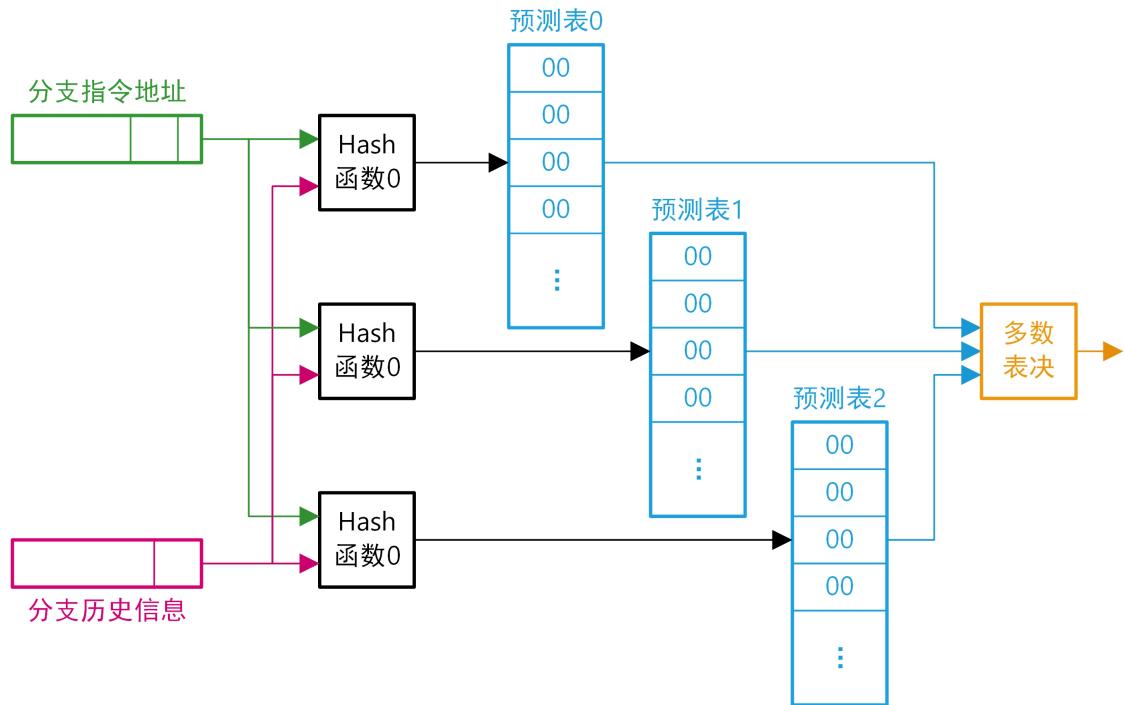


图11. Gskew预测器

Gskew的更新策略分为两种。全局更新策略就是对于每一次分支访问都更新所有的分支预测表。局部更新策略的想法是，如果最终更新的结果是正确，就不更新预测错误的全局预测器，避免干扰这个全局预测器对其他分支的预测。预测错误的全局预测器很可能发生了别名。既然预测器中已经有两个单元被这个分支独占，可以给出正确的预测结果，那么就把另一个全局预测器让给那个别名的分支吧。

参考文献：Pierre Michaud, Andre Seznec, Richard Uhlig, Trading Conflict and Capacity Aliasing in Conditional Branch Predictors, 24th Annual International Symposium on Computer Architecture, 1997.

TAGE预测器

TAGE是TAgged GEometric history length predictor预测器的缩写。TAGE类似于Bimod和Hybrid，但是比Bimod和Hybrid激进很多。除了一个有分支指令地址索引的预测表，TAGE还会提供很多个由不同长度的分支历史信息和分支地址的Hash函数索引的预测表。

TAGE可以分为基础表和扩展表（我自己起名字）。

- 基础表是必须有的，由分支指令地址直接索引，而且基础表保证有结果输出。所以基础表不需要提供标签，只提供一比特的预测结果。
- 扩展表的数量不定，可以有很多级。每一级都是用分支指令地址和不同长度的分支历史信息的Hash结果进行索引。级数越大，使用的历史信息越长。为了能够将尽可能长的分支历史信息体现在索引上，分支历史信息经过了折叠。扩展表也提供1比特的预测结果。扩展表的目的是区分不同分支指令。所以，扩展表提供标签以匹配不同的分支指令。

TAGE的预测过程如图12所示。首先，利用分支指令地址访问基础表（TAGE0），得到一个预测结果（蓝色信号）。接下来，将分支地址与分支历史信息进行hash得到扩展表的索引。每一级扩展表使用的分支历史信息都不一样。利用hash结果逐一访问各级扩展表（TAGE1-TAGE4），得到每一级扩展表的预测结果以及是否命中（绿色信号）。从所有命中的结果中，选择使用的历史信息最长的那一个作为预测结果。如果所有的扩展表都没有命中，使用基础表的结果。如图12所示，TAGE3和TAGE4命中，所以最终的预测结果是TAGE4给出的预测结果。

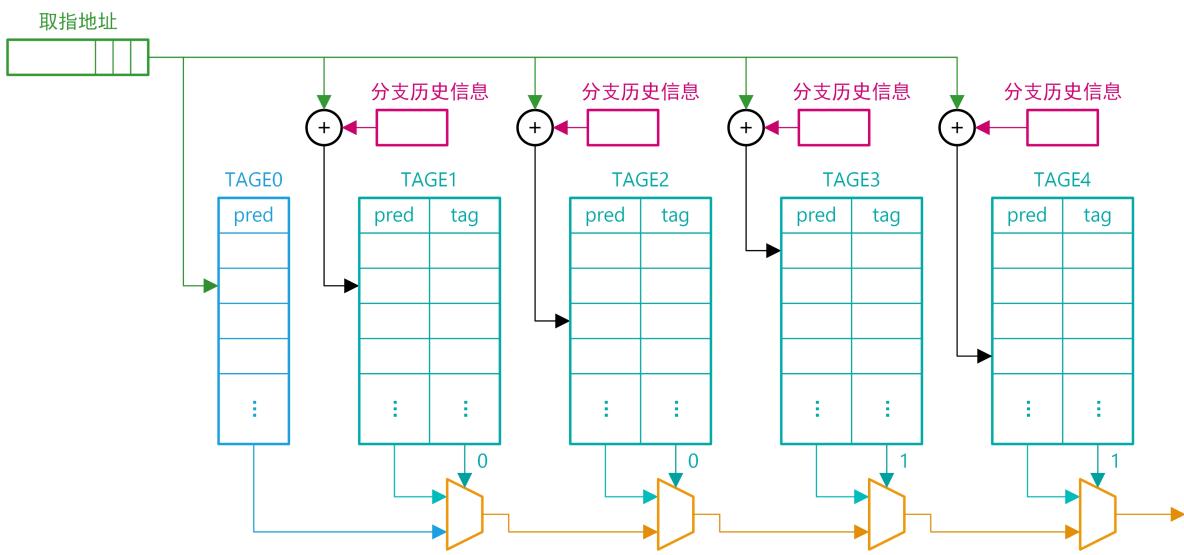


图12. TAGE预测器进行预测的过程

一方面，TAGE预测器可以根据分支历史信息分散同一条分支指令以及不同分支指令的单元，避免预测表别名。另一方面，TAGE还可以实现根据分支模式选择分支历史信息长度。重复模式短的分支，使用编号较低的扩展表，提供较短的历史信息；而重复模式长的分支，可以使用编号较高的扩展表，提供较长的历史信息。

TAGE预测器的更新过程就比较繁琐了。除了使用图中的表之外，还需要使用到HYST和USFL信息。基础表只提供hyst信息；扩展表提供hyst和usfl信息。

- Hyst信息表示置信度，也就是表示预测表中的预测结果的可信度。当分支预测正确的时候，可以调高置信度；当分支预测错误的时候，可以调低置信度。只有当置信度为0的时候，才能修改预测结果。
- Usfl信息表示单元被访问的频度。每一次访问都可以提高访问频度。只有Usfl为0的单元可以分配。访问频度的降低发生在无法分配单元的时候。如果处理器发现无法为一个分支指令分配单元，那么会将所有的Usfl都减1。

TAGE预测器需要一个较长的热身过程。分支指令占用的预测表会逐渐从延伸向高编号扩展表延伸。首先在TAGE1中分配一个单元。如果TAGE1中分配的单元再次命中，但是预测错误，那么会在TAGE2中分配一个单元。依次类推。如果TAGE级数比较多，这样的热身过程很长。为了提高热身速度，可以每一次预测错误，在高编号的两个预测表中分配单元。

基于感知的分支预测器

这个分支预测器很小众，只有在AMD的处理器中有使用。通过一个小型的神经网络替代了所有的预测表，由神经网络给出最终预测方向。

神经网络的输入可以但不限于包括：分支指令地址、全局历史信息、局部历史信息。用神经网络的优势在于，不需要花费大量的精力来分析分支历史模式，从而设计出更加有效的预测算法和更新算法。神经网络自己会调整其参数代表的特征，从而最大程度的匹配目前正在发生的分支的模式。不过，分支预测器的稳定性和训练时间是比较大的问题。

而且基于感知的分支预测器并不是在这两年提出的，而是在大约20年前就提出了，属于非常简单的神经网络。

小结

分支方向预测器目前已经能够取得很高的预测率。实验预测率可以达到95%以上，我们在真实应用中测得90%以上的预测率。分支方向预测器的主攻方向是条件分支。

分支预测器的设计参数包括：

- 分支预测表的数量、组数、路数。
- 分支预测表的索引构成的Hash算法：选择的分支历史信息片段和指令地址片段的长度，以及异或比特对应关系。
- 分支历史信息的总长度和编码方式。

循环预测器

循环预测器的目的是更加准确的控制构成循环的跳转，保证其不会被分支历史信息干扰。同时，也可以预测出跳出循环的那一次。之所以将循环预测器单列于小结之后，是因为循环预测器的设计千差万别。

循环预测器包含三个过程：识别、训练和预测。除了LOOP指令，其他分支指令并不能从指令本身获得其是否构成了循环。只有在执行过程中，发现指令指针跳回到了某个执行过的指令才能够识别出来。训练的目的是获得循环的次数。只有知道循环的次数才能预测出跳出循环的那一次。所以，循环预测器并不是识别出来就可以用于循环的。而是要多次执行以确定循环次数。循环预测器训练算法可以激进也可以保守，区别在于，训练过程会过滤掉多少复杂的循环模式。

训练出的循环会保存在LTT中。LTT是Loop Termination Table的缩写，保存构成循环的分支的地址，以及循环的次数。当再次遇到这条指令的时候，处理器认定开始执行循环，并且对循环次数进行计数。如果循环计数小于循环的次数，那么预测为Taken，继续循环；如果循环计数大于等于循环次数，那么预测为Not-taken，跳出循环。

分支目标地址预测器

分支目标地址预测器主要有三种：分支目标地址缓存（BTB）、间接分支目标地址缓存（IBTB）和返回地址堆栈（RAS）。IBTB为了间接跳转分支专门设置，但是不包含返回地址指令；RAS是为了返回指令专门设置的。而BTB则是整个分支预测的中心，他不仅包含了直接跳转指令的目标地址，还包含了分支指令类型的信息，这对于针对性的使用预测器是很重要的。

BTB

BTB是Branch Target Buffer的缩写。顾名思义，BTB产生的本意就是保存分支的目标地址。与任何一个cache类似，BTB也是组相连结构，每一组包含一些路，用来提高cache的利用率。BTB中的一个单元对应于一条分支指令，保存了这条分支指令的目标地址。根据具体处理器结构，BTB保存的目标地址会进行压缩。因为分支指令跳转的空间是有限的，并不是整个地址空间。

BTB的索引一般只是分支指令的地址，而不进行Hash，因为一条分支指令只需要占用一个单元。分支指令地址的低位比特作为索引，高位比特作为标签。一般来说，表示一个取指单元或指令内偏移的低位比特会被忽略。比如对于x86指令来说，低4位是被忽略的。

实际上，为了能够根据分支类型采用不同的预测方法，BTB中会保存分支的类型信息编码。这种编码可以使处理器在没有取指、译码的时候也能根据历史信息了解到指令的类型。因此，BTB也成为了分支预测的中心。只有了解了分支的类型（BTB命中，或者已经进行了译码），才能给出预测结果。

IBTB

IBTB是针对间接跳转对BTB进行的补充。间接跳转指令的目标地址来自于寄存器或内存，所以每一次执行同一条间接跳转指令，目标地址都可能不一样。这种指令很适合于构建高层语言中的Case语句和各种符号表、调用表。但是这种指令没有办法用BTB预测目标地址。在BTB中每条指令只对应于同一个单元，只保留上一次跳转的目标地址。对于间接跳转，BTB的预测结果概率是错误的。

借鉴于分支方向预测器的思路，设计师在访问预测表的索引中也引入了分支历史信息。用分支历史信息将同一条分支区分为不同的预测表单元。IBTB一般会复用分支方向预测器的分支历史信息。IBTB中的每个单元都保存一个目标地址，表示这条分支在遇到某一种分支历史模式的时候，应该跳转到指定的目标地址。与BTB类似，IBTB保存的目标地址也会进行压缩。

IBTB的设计参数包括：

- IBTB表的组数、路数。
- IBTB表的索引构成的Hash算法：选择的分支历史信息片段和指令地址片段的长度，以及异或比特对应关系。
- IBTB表的分支历史信息的总长度和编码方式，一般复用分支方向预测器。

RAS

返回地址堆栈是为返回指令专门设计的目标地址预测器。以为RET指令的目标地址一定是与其配对的CALL指令的下一条指令的地址，所以只需要能够在执行CALL指令的时候，记录下一条指令的地址，就可以在遇到RET指令的时候预测目标地址。

可以说，RAS是处理器中堆栈的缩小版和投机版。缩小版是因为处理器中的堆栈还会记录参数传递等信息，而RAS只需要记录返回地址即可。投机版是因为RAS中可能会记录投机路径上取指的CALL指令引起的压栈。在投机结束进行流水线刷新的时候，还需要将这些投机路径上的压栈消除。而处理器中的堆栈不存在这样的问题，因为没有commit的指令不会将压栈真正写入内存。

维护RAS包含三个步骤：

- 如果遇到CALL指令，则将返回地址压栈；
- 如果遇到RET指令，则将栈顶单元出栈，作为分支指令的目标地址；
- 如果遇到流水线刷新的情况，需要对RAS也进行刷新，将投机路径引起的压栈忽略。所以，每条分支指令都需要记录其对应的RAS栈顶位置。

RAS的流水线控制与分支方向预测器和BTB、IBTB都不同，RAS是唯一一种在预测阶段就更新缓存的预测器。当CALL和RET指令被commit的时候，不需要在更新RAS。

RAS的设计参数只有RAS的深度。如果没有RET的CALL超过了RAS的深度限制，设计师一般会保留最近执行的CALL，忽略早期执行的CALL。从硬件上的表现，就是RAS索引直接溢出。

补充

分支历史信息的编码方式

分支历史信息会以比特流的形式保存在分支历史信息的寄存器或者查找表中。如何编码分支历史信息是一个很有趣的话题，因为不同的编码方式可以构造不同的分支历史模式。

最简单的方法是，用一比特代表一条分支。0表示分支Not-taken；1表示分支Taken。

稍微复杂一点的方式是，用多个代表一条分支。在编码中，体现出不同的分支类型。这种应该比较常用。

再复杂一点的方式是，在分支编码中引入目标地址信息，以区分间接跳转。从测试程序分析，Intel应该采用了这一点。

预测表单元的更新

预测表与Cache是类似的，保存的都是最近使用的分支的信息。预测表也必然存在需要进行单元替换的情况。那么如何选择需要替换的单元呢？常用的算法有如下几种。

- 随机算法。利用一个简单的随机数生成器（累加器，因为访问时间随机，所以等效随机数），确定需要替换掉的单元所在的Way。
- 先入先出。被替换的单元是最先分配的单元。记录每一个单元分配的先后顺序，最先分配的单元第一个被替换。
- LRU。LRU表示Least-Recently-Used。按照单元被访问的顺序进行排队。任何一个单元被访问了，这个单元就被排到队列的最前面，其他单元的顺序不变。每次被替换的都是队尾的单元。例如：四个单元都被占用后的排序为3-2-1-0，队尾的单元是单元0。接下来，单元2和单元0被先后访问，此时排序为0-2-3-1。如果此时需要替换，那么被替换的是单元1。
- 伪LRU（伪树）。与LRU非常类似，但是实现比LRU简单很多，因为LRU要维护队列排序。伪LRU并不保证替换的一定是LRU，只保证一定不是上一次使用那个。伪LRU按照树的方式保存上一次访问单元的树的路径，可以通过简单的比特逻辑得到需要替换的单元，并且更新保存的伪LRU数据。

伪LRU算法

伪LRU算法构建一个二叉树，树的叶子节点数量与组中的单元数相同，也就是way。树的层数（不包含叶子）为 $\log_2(\text{way})$ ，树的结点（不包含叶子）为 $\text{way}-1$ 。每一个节点都有一个数值，0表示访问这个节点的左边；1表示访问这个节点的右边。所以，如果cache有way路，那么只需要way-1个比特来表示这个数值。

以一个4路的cache为例。4路的cache，需要3比特来表示。其中比特0表示根节点（节点0）；比特1表示根左边的节点（节点1），连接路0和路1；比特2表示根右边的节点（节点2），连接路2和路3。初始化，LRU值为000。

表2. 伪LRU算法的访问和更新过程示例

	LRU旧值	访问路径	分配单元	LRU更新	LRUnewValue
新单元	000	节点0的值为0，访问节点1； 节点1的值为0，访问左叶子	way0	节点0和节点1设置为1	011
新单元	011	节点0的值为1，访问节点2； 节点2的值为0，访问左叶子	way2	节点0设置为0； 节点2设置为1	110
新单元	110	节点0的值为0，访问节点1； 节点1的值为1，访问右叶子	way1	节点0设置为1； 节点1设置为0	101
新单元	101	节点0的值为1，访问节点2； 节点2的值为1，访问右叶子	way3	节点0设置为0； 节点2设置为0	000
访问way0	000	节点0的值为0，访问节点1； 节点1的值为0，访问左叶子		节点0和节点1设置为1	011
访问way3	011	节点0的值为1，访问节点2； 节点2的值为1，访问右叶子		节点0和节点2设置为0	010
新单元	010	节点0的值为0，访问节点1； 节点1的值为1，访问右叶子	way1	节点0设置为1； 节点1设置为0	001
新单元	001	节点0的值为1，访问节点2； 节点2的值为0，访问左叶子	way2	节点0设置为0； 节点2设置为1	100

当需要根据伪LRU获取需要分配的单元时，按照伪LRU值保存的二叉树的状态从根开始移动，直到叶子。找到的叶子就是需要替换的单元。比如表中第1行，伪LRU值为000。节点0的值（比特0）为0，访问根的左孩子，也就是节点1；节点1的值（比特1）也是0，访问根的右孩子，也就是way0。类似的，如表中第4行，伪LRU值为101。节点0的值（比特0）为1，访问根的右孩子，也就是节点2；节点2的值（比特2）也是1，访问根的右孩子，也就是way3。

任何对cache的访问（命中或分配新单元）都需要更新伪LRU。首先，根据分配的单元或者命中的单元，可以确定一条访问路径。更新伪LRU时，访问路径上设计的结点对应的比特都会被设置为路径上数值的取反；没有在路径上的结点保持不变。比如表中第2行，访问路径是，节点0的值为1；节点2的值为0。所以，需要将节点0设置为0；将节点2设置为1；节点1保持不变。再比如表中第6行，访问路径为结点0的值为1，节点2的值为1。所以，需要将节点0和节点2都设置为0，而节点1保持不变。

从表中可以看出伪LRU和LRU的区别。在第7行，伪LRU替换了way1；但是LRU会替换way2。way2在第2行被分配，再没有被访问过，是真正的LRU；way1在第3行分配，晚于way2。

设计实例

Godson2

龙芯2的流水线与x86有很大不同。龙芯流水线的第一个周期进行取指，同时进行预译码，获得指令信息（分支指令以及类型）。也就是说，龙芯是在知道指令信息的情况下，进行分支预测。

- 对于无条件跳转，跳转方向一定是Taken。目标地址由BTB预测器提供；返回指令的目标地址由RAS预测器提供。
- 对于条件跳转，跳转方向由全局两级预测器提供。目标地址来自与指令本身。

各个预测器的组织如下：

- 全局两级预测器包含9比特历史信息，和4096个单元的预测器表。Hash函数为取指地址右移5比特再与历史信息异或。取异或结果的低9位作为访问预测表地址的高9位，访问预测表地址的低3位由取指地址的4-2比特提供。预测表的每一个单元都是一个2比特饱和计数器。

```
index[11:0] = {((fetch_PC >> 5) ^ global_history)[8:0], fetch_PC[4:2]}
```

- BTB预测器提供128组，1路组相连，共保存128个分支。索引为取指地址的11-5比特，标签为取指地址18-12比特。保留目标地址的低24比特。
- RAS预测器放置在译码完成之后，根据确定的调用和返回指令进行压栈和出栈操作。RAS层为4层。

Intel

Intel的分支预测器与龙芯的预测器有很大的不同，主要是因为Intel处理器进行分支预测的时候，没有取指指令的信息。需要的信息都只能来自于BTB。各代Intel CPU架构使用的分支预测结构如下。

P1

- BTB提供64组，4路组相连，共256个单元。BTB索引为分支指令地址的0-5比特，标签为指令地址的6-31。
- 两比特饱和计数器。与图3不同，如果在状态0遇到Taken，则会直接跳到状态3。

PMMX

- BTB提供16组，16路组相连，共256个单元。BTB索引为分支指令地址的2-5比特，标签为指令地址的6-31。
- 全局两级预测器，提供4比特全局历史信息。

PPro、P2和P3

- BTB提供32组，16路组相连，共512个单元。BTB索引为分支指令地址的4-8比特，标签为指令地址的4-31。
- 局部两级预测器，预测分支跳转方向。

P4

- BTB提供512组，8路组相连，共4096个单元。BTB由Trace cache地址索引。
- Agree预测器（全局两级预测器），16-比特全局历史信息，4096个单元。

P4E

- BTB提供512组，8路组相连，共4096个单元。BTB由Trace cache地址索引。
- 16-比特全局历史信息表。

PM和Core2

- BTB提供512组，4路组相连，共2048个单元。BTB索引为分支指令地址的4-12比特，标签为指令地址的13-21。
- Hybrid预测器和两级预测器组合预测条件分支，引入循环预测器。
- 循环预测器，64组、2路组相连，共128个单元。索引为分支指令的4-9比特。
- IBTB，2048个单元，4路组相连，共8192个单元。IBTB索引为分支指令地址的0-10比特，其他比特作为标签。
- RAS具有16个单元。

Nehalem

- BTB有两级，类似两级cache系统。
- Hybrid预测器结合一级或两级预测器预测条件分支，引入循环预测器。
- RAS具有16个单元。

Sandy Bridge和Ivy Bridge

- 两级预测器，32比特全局历史信息。
- IBTB。

Haswell

- 开始使用TAGE预测器，预测分支跳转方向。

CHX003预测器设计

我司的分支预测器与Intel类似，也是以BTB为核心，因为BTB会保存分支指令的地址、长度、类型以及目标地址。不同类型的分支指令会采用不同的预测器得到预测方向和目标地址，也就是经过不同的路径。

- 无条件分支指令：如果BTB命中，则根据指令类型选择Taken或Not-taken（有Hint的情况下）。如果BTB不命中，则在译码之后，使用backward-taken-forward-not-taken预测器（BTFTN）。
 - 直接跳转指令的目标地址来自于BTB；
 - 间接跳转指令的目标地址来自于IBTB。
 - RET指令的目标地址来自于RAS。
- 条件分支指令：如果BTB命中，则由TAGE预测器确定跳转方向。目标地址来自于BTB。如果BTB不命中，则在译码后，使用BTFTN和FBHT（使用率很低）的组合进行预测。
 - 提供循环预测器，但是效果有待检验。

CHX003的分支预测分为两部分。第一部分在取指译码之前，根据BTB中缓存的指令信息选择合适的路径进行预测，如图13所示。

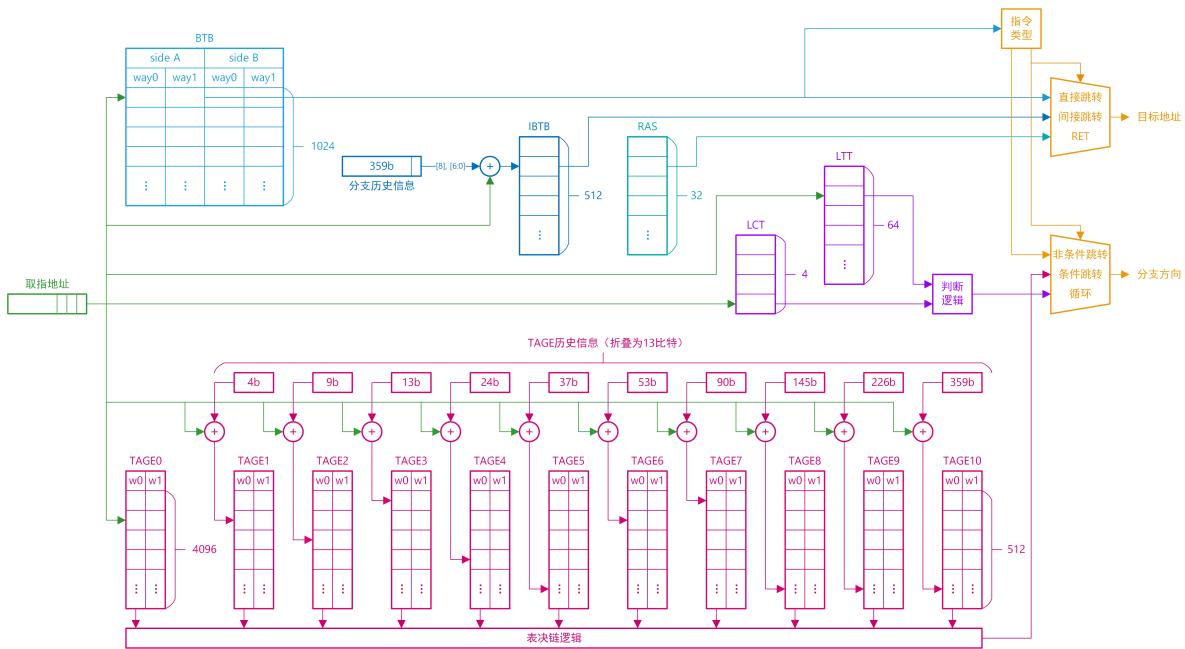


图13. CNX003中的取指前预测表

第二部分在译码之后，针对BTB没有命中的指令进行预测，并且修正预测错误的指令，如图14所示。

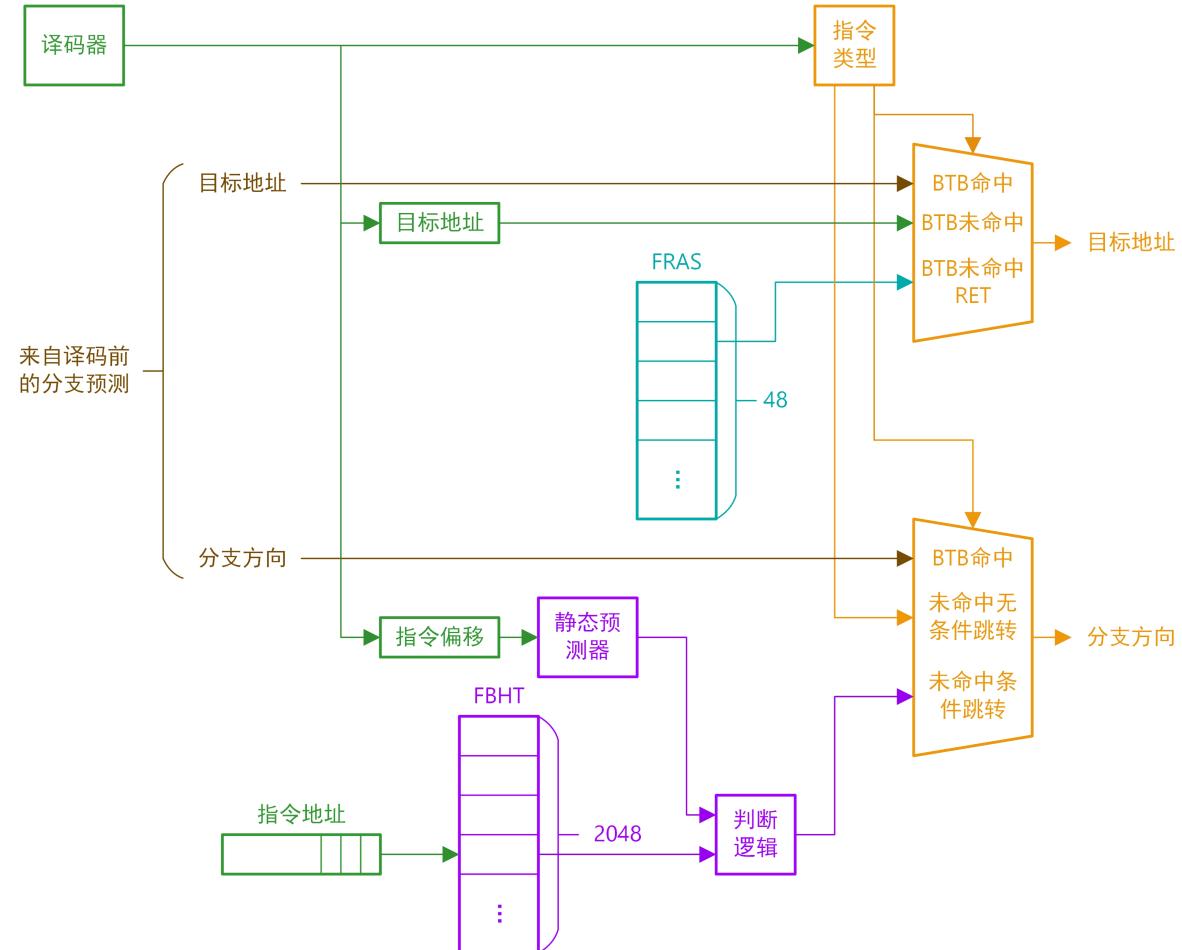


图14. CNX003中的译码后预测过程

BTB

BTB包含**1024组**，每组包含**2路又2-side**（一共4096个单元）。每一路包含最多2条分支的信息，分别标记为side A和side B。BTB索引是取指地址的比特 [13:4]。每个单元包含**19比特标签**，**46比特数据**，**2比特有效标志**。

19比特标签包括2比特预测模式信息和16比特分支指令地址（[47, 31, 27:14]）。46比特数据包括4比特偏移（分支指令在16B取指单元中的偏移），8比特分支类型和指令长度的编码，1比特跨边界标志（分支指令跨越了16B取指单元的边界）和33比特目标地址（低33比特）。

BTB中的每一组又对应于一个伪LRU值。LRU也包含**1024个单元**。索引是取指单元的比特[13:4]。每个单元包含**3比特数据**。比特2对应于根节点，1表示上一次写此单元时写入了side A。比特1对应于根节点的左节点，1表示上一次写此单元的sideA时，写入了way1。比特0对应于根节点的右节点，1表示上一次写此单元的sideB时，写入了way1。

对于同一个取指单元，BTB最多记录两条分支，分别为了两个side中，但是可以占用不同的way。BTB的一组中，最多可以保存4条分支指令，来自于2、3或4个取指单元（16B对齐）。

- 取指单元A的分支占用了sideA和sideB的way0；取指单元B的分支占用了sideA和sideB的way1。
- 取指单元A有分支占用了sideA和sideB的way0；取指单元B的分支占用了sideA的way1；取指单元C的分支占用了sideB的way1。
- 取指单元A-D分别有一条分支占用了组中的一个单元。

IBTB

IBTB包含**512个单元**。索引是取指地址和分支历史信息的哈希函数。哈希函数如下：

```
index[8:0] = addr[12:4] ^
    {bht[8], bht[6:0], 1'b0};
```

addr 表示取指地址；bht 表示分支历史信息。所以 index[0] 并没有参与哈希运算。

每个单元包含**64比特数据**，包括1比特有效标志，15比特标签和48比特的目标地址。

TAGE

TAGE预测期包含11级，总共有32个缓存。基础预测缓存（TAGE0）包含PRED和HYST两个缓存；TAGE1和TAGE10都包含PRED、HYST和USFL三个缓存。

TAGE0

TAGE0的索引是取指地址的比特[15:4]，不需要Hash，也不需要标签比较。所以，任何取指地址一定会有1个预测结果。TAGE0的PRED和HYST缓存都包含**4096组**，每组包含**2-side**，与BTB对应。

TAGE0的PRED缓存的每个单元包含**1比特数据**，表示预测结构。如果是1，表示分支发生Taken。TAGE0的HYST缓存的每个单元包含**1比特数据**，表示置信度标志。为1表示TAGE0的预测结果可信。

TAGE1-10

TAGE1-10的索引和标签是取指地址和分支历史信息的哈希函数。哈希函数如下：

```
index[9:0] = addr[12:4] ^
    {addr[21-stage:13-stage], 1'b0} ^
    {idxcomp[8] ^ idxcomp[0], idxcomp[7:1], 1'b0};
tag[12:0] = addr[16:4] ^ {tagcomp[12:1], 1'b0};
```

addr 表示取指地址；bht 表示分支历史信息。所以 index[0] 和 tag[0] 并没有参与哈希运算。

说明：在最新的RTL中，TAGE使用的tag信息变为12比特，地址计算方式变为

```
tag[12:0] = addr[16:4] ^ {tagcomp[11:0], 1'b0};
```

每级TAGE1-10所用的 `addr[21-stage:13-stage]` 选择 `addr` 的不同部分，如下：

TAGE1	TAGE2	TAGE3	TAGE4	TAGE5	TAGE6	TAGE7	TAGE8	TAGE9	TAGE10
21:13	20:12	19:11	18:10	17:9	16:8	15:7	14:6	13:5	14:6

TAGE1-10的PRED、HYST和USFL缓存都包含**512组**，每组包含**2-side**，与BTB对应。

TAGE1-10的PRED缓存的每个单元包含**14比特数据**，包括13比特标签和1比特预测结果。如果是1，表示分支发生Taken。TAGE1-10的HYST缓存的每个单元包含**2比特数据**，表示置信度标志。0表示预测结果不可信；3表示最高可信度。TAGE1-10的USFL缓存的每个单元包含**2比特数据**，表示使用频率。0表示没有使用；3表示高频率使用。

TAGE1-10的USFL缓存每个都包含**512组**，每组包含**2-side**。索引和标签是分支指令地址和分支历史信息的哈希函数。与TAGE1-10的PRED相同。每个单元包含**2比特数据**，表示使用频率。0表示没有使用；3表示高频率使用。

循环预测器

LTT包含**64个单元**。索引是取指地址的比特 `[9:4]`。每个单元包含**23比特数据**，包括1比特有效标志，15比特标签（取指地址的比特 `[24:10]`）和7比特循环次数。

循环预测还提供了4个预测计数器和4个训练计数器。也就是说，循环预测器同时可以训练4个循环分支指令；同时可以预测4个循环分支指令。

FBHT

FBHT包含**2048个单元**。索引是分支指令地址的比特 `[14:4]`。每个单元包含**1比特数据**。如果为1，表示不认可静态预测器的结果，需要对静态预测结果取反。这个预测器的实际作用比较小。

RAS

RAS堆栈包含**32个单元**。读写索引是堆栈的读写指针。每个单元包含**54比特数据**，包括48比特目标地址和6比特读指针。其中读指针的含义是，弹出这个单元后的读指针值，从而形成与前一个单元的链接。

因为CNX003的RAS的设定是，弹出单元后不会调回写指针，所以栈顶下的一个单元不一定是前一级，而可能是空闲的。这样做的目的是为了避免投机执行的RET指令会覆盖掉有效的单元。

FRAS

FRAS堆栈包含**48个单元**。读写索引是堆栈的读写指针。每个单元包含**55比特数据**，包括48比特目标地址和7比特读指针。其中读指针的含义是，弹出这个单元后的读指针值，从而形成与前一个单元的链接。

FRAS位于流水线的FGX级，处于译码之后。而RAS处于译码之前。

历史信息编码

CHX003使用的历史信息最长为359比特，编码反映了反映了分支指令的方向、类型和地址。

- 取指单元只有条件分支，但是都是NT，用 `1'b0` 表示。
- 取指单元中遇到的第一个分支是条件分支，而且第一个条件分支Taken，用 `1'b1` 表示。
- 取值单元只有条件分支，而且第一个条件分支Not-taken而且第二个条件分支Taken，用 `2'b01` 表示。只要是先NT后T就可以，NT和T之间还可以有其他NT分支。
- 取值单元只有非条件分支或者第一个分支就是非条件分支，用4比特表示。比特3为1，表示 Taken；比特2-0为非条件分支的地址。

- 取值单元有非条件分支而且非条件分支前有NT的条件分支，用5比特表示。比特4-3为`2'b01`，表示NT-T；比特2-0为非条件分支的地址。

CHX003一共使用了11个全局历史信息。其中1个完整的359比特历史信息用于生成IBTB的索引。IBTB每次使用低9比特用来生成索引地址、低14比特用来生成标签。另外20个历史信息都是折叠历史信息，其中10个用于生成TAGE1-TAGE10的索引、10个用来生成TAGE1-TAGE10的标签。

折叠的含义是将历史信息切成等长的数段，然后把每一段都异或起来，得到一个结果。这个结果虽然比原有的历史信息短，但是还是保存的一定量的信息。比如将37比特的历史信息折叠成9比特的过程为：

```
bht[8:0]^bht[17:9]^bht[26:18]^bht[35:27]
```

如果历史信息的长度不是折叠后长度的倍数，不足的比特补0。

CHX003中的TAGE1-TAGE10依次使用了4比特、9比特、13比特、24比特、37比特、53比特、90比特、145比特、226比特和359比特历史信息。为了生成访问TAGE的索引，历史信息需要被折叠成9比特；为了生成访问TAGE的标签，历史信息需要被折叠成13比特。