分支预测算法 TAGE

摘要

在高性能的乱序执行处理器当中, 为了尽可能的提高处理器的吞吐率, 一个准确的分支预测器可以起到至关重要的作用: 这是因为, 在深流水线以及乱序处理的微处理器中, 一条分支指令的预测错误会带来接近 10 个时钟周期甚至更多的冲刷惩罚, 换句话说, 从这条分支指令进行预测到 BRU（Branch Unit, 指专门处理分支指令的运算单元）发现错误这之间, 所有比该条分支指令更年轻的指令都将会是无用功, 而这可能已经设计到了几十条甚至上百条指令的取消, 这个代价是非常大的. TAGE 是现今最经典的分支预测算法,TAGE 及其后续的变体都是当今高性能微处理器的分支预测算法基础, 本文正是基于 TAGE 算法, 实现了一个分支预测器.

关键词： 分支预测; TAGE

1 引言

现代处理器为了充分利用指令级并行性，大多采用流水线技术，将一条指令的执行过程分解为多个阶段，使多条指令能在不同阶段同时执行，从而提高整体效率。然而，程序中广泛存在的分支指令却给流水线的高效运行带来了巨大挑战。分支指令根据条件判断决定程序的执行流向，如常见的 if - else 语句、循环结构等。在流水线执行时，由于分支指令的后续执行路径在条件判断完成前是不确定的，处理器可能会错误地预取分支后的指令，当条件判断结果与预取方向不符时，就需要清空流水线，重新取指、译码等，这会造成大量的时钟周期浪费，严重影响处理器性能。据统计，在一些通用程序中，分支指令带来的性能损失可达 10% - 30%，甚至更高，这使得分支预测成为提升处理器性能的关键环节。同时，随着人工智能、大数据处理、科学计算等领域的蓬勃发展，对处理器性能的需求持续增长。这些复杂应用包含大量的条件判断和循环结构，使得分支指令更为密集，分支预测的重要性愈发凸显。[4]

有效的分支预测能够显著减少流水线因分支指令产生的停顿，使处理器可以持续不断地取指、译码、执行指令，提高指令级并行度，进而大幅提升处理器的整体性能。对于数据中心的服务器处理器，高效的分支预测可加速海量数据的处理，缩短任务完成时间；对于移动端处理器，能提升设备响应速度，优化用户体验。除此之外，有效的分支预测算法科研减少因错误分支预测导致的流水线刷新操作，不仅能节省时间，还能降低处理器的功耗。在移动设备、嵌入式系统等对功耗敏感的场景下，精准的分支预测有助于延长电池续航时间，增强产品竞争力。分支预测技术还能推动体系结构发展, 分支预测技术的研究涉及到对程序行为的深入理解、预测算法的创新以及硬件实现的优化，这一系列探索为计算机体系结构的进一步发展提供了理论和实践基础，促使新的处理器架构设计理念不断涌现。

综上所述，分支预测课题无论是从当下的技术困境、未来的发展需求，还是理论与实践的可行性角度考量，都极具研究价值，对推动计算机领域发展意义重大。

2 相关工作

2.1 早期简单预测方法

静态预测

这是最早期的分支预测方法之一。它在程序编译阶段就确定分支的走向，而不考虑程序运行时的实际情况。例如，对于无条件分支指令（如跳转指令），可以很容易地预测其执行方向。对于有条件分支，一种常见的静态预测策略是“总是预测不跳转”（BTFNT，Backward Taken, Forward Not Taken）。这种策略基于观察到的一个现象：在循环结构中，向后的分支（通常是循环的结尾跳回开头）往往是被执行（Taken）的，而向前的分支（如 if 语句中的分支跳出条件判断部分）往往是不被执行（Not Taken）的。然而，这种静态预测方法的准确性有限，因为它无法适应程序运行时的动态变化。例如，在一个复杂的程序中，根据不同的输入数据，同一个有条件分支可能会有不同的执行方向，但静态预测不能根据这些变化做出调整。

基于历史的简单动态预测

随着对处理器性能要求的提高，动态预测方法开始出现。其中一种简单的动态预测方法是使用分支历史寄存器（BHR，Branch History Register）来记录分支指令的历史执行情况。例如，对于一个特定的分支指令，BHR 可以记录它最近几次执行时是跳转（Taken）还是不跳转（Not Taken）。基于这些历史信息，预测器可以采用简单的算法来预测下一次分支的执行方向。比如，最简单的“一位饱和计数器”（1 - bit Saturating Counter）方法。这个计数器初始值为 0（表示预测不跳转），当分支实际执行是跳转时，计数器加 1；当分支实际执行是不跳转时，计数器减 1。计数器的值大于等于 1 时，预测分支为跳转；计数器的值小于 1 时，预测分支为不跳转。这种方法比静态预测更灵活，但对于复杂的程序行为，其预测准确性仍有待提高。

2.2 基于硬件的复杂预测器

两级自适应预测器

为了提高预测准确性，两级自适应预测器被提出。它由两个部分组成：一个分支历史寄存器（BHR）和一个模式历史表（PHT，Pattern History Table）。BHR 记录分支指令的最近几次执行历史，而 PHT 则存储了基于不同历史模式的预测状态。例如，对于一个具有 4 位 BHR 的两级自适应预测器，BHR 可以记录分支最近 4 次的执行情况，这样就有种不同的历史模式。PHT 中有 16 个条目，每个条目对应一种历史模式，并且每个条目包含一个两位饱和计数器。当一个分支指令执行时，根据 BHR 中的历史模式在 PHT 中查找对应的条目，然后根据条目中的计数器来预测分支的执行方向。这种预测器能够根据分支的历史行为自适应地调整预测策略，比简单的动态预测器准确性更高。

全局历史分支预测器

全局历史分支预测器考虑了所有分支指令的历史，而不仅仅是单个分支指令的历史。它使用一个全局历史寄存器（GHR，Global History Register）来记录所有分支指令的执行历史。例如，在一个程序中有多个分支指令，GHR 会将这些分支指令的执行情况按照顺序记录下来。预测时，根据 GHR 中的全局历史信息和当前分支指令的部分信息（如指令地址的低位部分）来查找预测表，从而确定分支的执行方向。这种方法可以捕捉到不同分支指令之间的相关性，对于具有复杂分支结构的程序（如嵌套的 if - else 语句和多层循环）能够提供更准确的预测。

2.3 基于硬件的复杂预测器

编译器指令调度优化

编译器可以通过指令调度来辅助分支预测。例如，将最有可能执行的指令放在分支指令之后的顺序位置，这样即使分支预测错误，处理器也能更快地执行正确的指令路径。编译器还可以对循环进行展开或合并等操作，改变程序的分支结构，使其更有利于分支预测。例如，对于一个简单的循环，编译器可以通过循环展开将循环体中的指令复制多次，减少循环分支的频率。这样可以降低分支预测错误的概率，同时也可能提高指令级并行性。

基于程序剖析的静态预测改进

程序剖析（Profiling）是一种在程序运行前或运行过程中收集程序行为信息的技术。通过程序剖析，可以了解分支指令的执行频率、执行方向等信息。然后，编译器可以利用这些信息来改进静态预测。例如，对于一个在程序运行过程中经常被执行且执行方向固定的分支指令，编译器可以将这种信息反馈到静态预测策略中，改变原来简单的静态预测规则，使其更符合程序的实际运行情况。

3 本文方法

3.1 本文方法概述

本文基于 TAGE 算法, 完成了由 TAGE Predictor、Loop Predictor、Corrector Filter 等部件组成的分支预测器 [1]. 整个算法部件由 TAGE、Loop Predictor、Corrector Filter 三个主要部件组成. 叙述顺序也是 TAGE、Loop Predictor、Corrector Filter 依次叙述.

3.2 TAGE

TAGE 是一种 ppm-like 的分支预测器。其主要思路是通过不同长度的 history 和 PC 进行哈希来获得预测结果 [2]，对于与不同长度的 history 有强关联的分支都能取得比较好的效果。

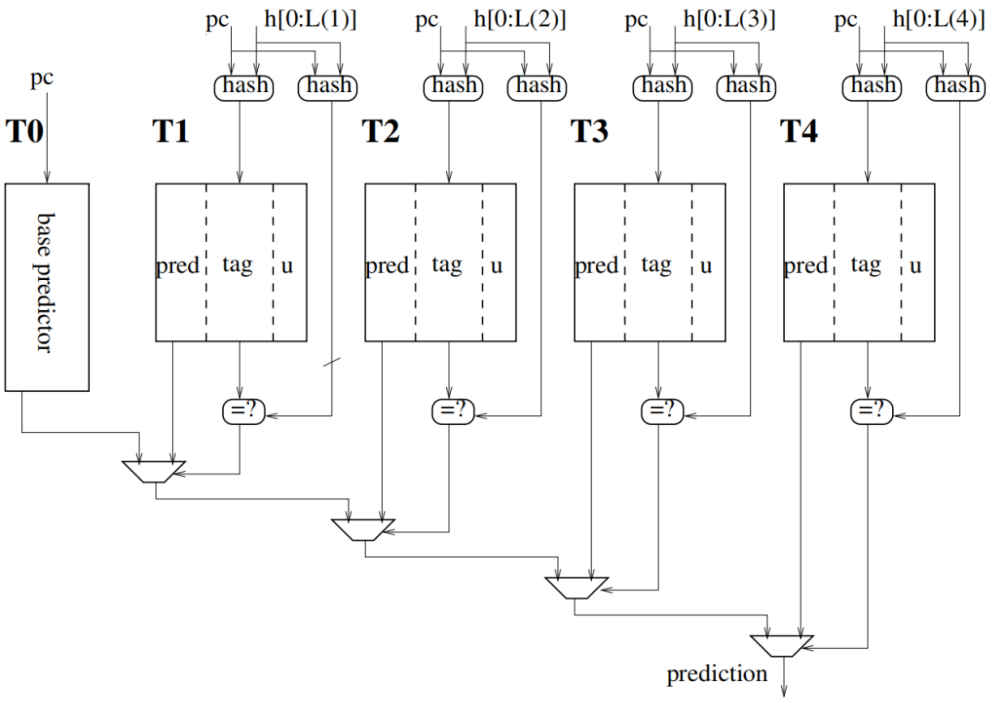

图 1. TAGE 算法结构图

如图 1所示,BaseTable 是一个简单的根据 PC 索引的二位饱和计数器，每个表项对应一个 index 的饱和计数器,Tagged Table（图中 T1-T4) 则是用 PC 和不同长度的 GHR（Global history register）生成索引和标签以获取表项。

预测过程

- BaseTable 直接根据 pc 获得对应表项，通过计数器判断结果
- Tagged Table 将 pc 与不同 len 的 GHR 做 hash1，获得表项的 index，取出 entry。然后判断 entry 的 tag 和 hash2 (PC, GHR[0:L]) 是否相等，相等则命中。如果命中，entry.ctr 就可以给出当前表的预测结果。注意，这里的 hash1 和 hash2 不能是同一个 hash 函数。
- 选择匹配长度最长的预测结果作为最终结果，其对应的 Tagged Table 为 provider component。匹配长度第二长的作为备选结果，其对应的 Table 为 altpred。

更新过程

- 根据实际结果，更新 provider component 的计数器
- 如果预测结果正确，不需要分配新的 entry。
- 如果预测结果错误，可能要分配新的 entry。
- 更新 useful 位

对新分配节点的优化

如果按照 PPM-LIKE 的逻辑，一个刚分配的 entry 很可能会被直接投入使用。但是在特定的测例下，新分配的 entry 有很大概率会出错，需要对这种情况做一些优化。

- 总体思路：用一个 4bit 的全局计数器，如果 provider-component 是新分配的 entry，且其预测结果出错，但是 altpred 预测正确，该计数器 +1；反过来计数器-1。然后如果这个计数器超过了阈值，在遇到 new-entry 的时候，就使用 altpred 作为结果。
- 判断是不是新 entry 只需要：useful counter == 0 且 weak(taken or not taken)

3.3 Loop Predictor

loop predictor 是 LTage 的组成部分，该部件主要是为了处理一些以常数次数循环的循环。[3] 其能弥补 TAGE 的 GHR 长度有限，对于一些非常长的循环并不能有效预测；或者 TAGE 对于循环的反应可能不够灵敏。

预测过程

- 如果 past-iter-count<now-iter-count，预测为 Taken；如果 past-iter-count=now-iter-count，预测为 Not Taken。
- 如果 confidence-count 为 MAX，就认为这个预测结果可以采用；否则预测结果不可以采用。

若 loop-tag 和 entry.tag 不同，这说明当前的表项不匹配，就减少 entry.age，如果 entry.age 已经是 0，就需要分配新的表项。（这样做的理由是：age 实际上标记了一个 entry 希望被替换的次数，如果一直希望被替换，证明原来的 loop 可能已经失效了）

若 loop-tag 和 entry.tag 相同:now-iter-count + 1

3.4 Corrector Filter

corrector filter 主要用于纠正一些 Tage 不能正确预测的场景。比如，一些分支与历史无关，可能就是统计意义上的偏向某个方向，[5] 这时候 Tage 可能不如单纯的基于 PC 的预测器。Corrector Filter 就可以部分纠正这种问题.

预测过程：首先考虑 Tage 的结果的可信度。如果 Tage 的可信度是高的（通过 ctr 来判断，ctr 偏离 weak 足够远就证明可信度高），那么就不需要再使用 Corrector filter。如果 Tage 的可信度低，就使用 Corrector filter 的结果。直接通过 pc 和 tage 预测结果索引得到表项，然后判断 tag 是否匹配，而后根据 ctr 获得结果即可。

更新过程

- 如果 tage 的预测结果是正确的，且 tag 不匹配，则不需要做任何操作（tage 已经能解决问题）
- 如果 tage 的预测结果是错的，且 tag 匹配，则根据分支结果更新饱和计数器（可能需要使用当前的表项）
- 如果 tage 的预测结果是错的，且 tag 不匹配，则根据不同情况，分配表项或者更新饱和计数器。

4 复现细节

4.1 与已有开源代码对比

此处参考了 PREDICTOR::UpdatePredictor 的实现，自行添加了一些格外的字段，用来计算正确率，除此之外进行了新分配节点的优化，优化逻辑是，用一个 4bit 的全局计数器，如果 provider-component 是新分配的 entry，且其预测结果出错，但是 altpred 预测正确，该计数器 +1；反过来计数器-1。然后如果这个计数器超过了阈值，在遇到 new-entry 的时候，就使用 altpred 作为结果。

其余部分没有借鉴其他的代码，根据算法思路进行代码编写.

```
void  PREDICTOR::UpdatePredictor(UINT32 PC, bool resolveDir, bool predDir, UINT32 branch

  UINT32 base_index   = PC % numBaseTableEntries;
  uint8_t base_counter = base_table[base_index];

  // update counter of provider component
  if(provider_component == -1){
      if(resolveDir == TAKEN){
        base_table[base_index] = SatIncrement(base_counter, BASE_CTR_MAX);
      }else{
        base_table[base_index] = SatDecrement(base_counter);
      }
  }
  else{
    uint8_t pred_ctr = tag_table[provider_component][tag_table_idx[provider_component]].
    if(resolveDir == TAKEN){
      tag_table[provider_component][tag_table_idx[provider_component]].ctr = SatIncremen
    }
    else{
      tag_table[provider_component][tag_table_idx[provider_component]].ctr = SatDecremen
    }
  }

  // if prediction is incorrect, allocate entry
  if(resolveDir != predDir && provider_component != TAGE_TABLE_NUM - 1){
    int unalloc_idx[TAGE_TABLE_NUM] = {-1, -1, -1, -1};
    int count = 0;
    for(int i = provider_component + 1; i < TAGE_TABLE_NUM; i++){
      if(tag_table[i][tag_table_idx[i]].u == 0){
        unalloc_idx[count] = i;
        count ++;
```

```
    }
  }
  // if uk > 0 for k in (i, M), then uk = uk-1 for all uk
  if(count == 0){
    for(int i = provider_component + 1; i < TAGE_TABLE_NUM; i++){
      tag_table[i][tag_table_idx[i]].u = SatDecrement(tag_table[i][tag_table_idx[i]].u
    }
  }
  else{
    srand(3407);
    // allocate one entry each time
    // if more than one T_i need allocate, for i < j, the probility of allocate entry
    // example:count = 3, rand = {0} for unalloc[2], rand = {1, 2} for unalloc[1], ran
    int total_pro = (1 << count) - 1;
    int r = rand() % total_pro;
    int choose_idx = -1;
    for (int i = 0; i < count; i++){
      if( r >= (1<<i) - 1 && r < ( 1 << (i + 1) ) - 1 ){
        choose_idx = unalloc_idx[count - 1 - i];
      }
    }
    UINT32 idx_in_tag_table_choose = tag_table_idx[choose_idx];
    tag_table[choose_idx][idx_in_tag_table_choose].tag = tag[choose_idx];
    tag_table[choose_idx][idx_in_tag_table_choose].u = 0;
    if(resolveDir)
      tag_table[choose_idx][idx_in_tag_table_choose].ctr = TAGGED_WEAK_CORRECT;
    else
      tag_table[choose_idx][idx_in_tag_table_choose].ctr = TAGGED_WEAK_CORRECT - 1;
  }
}

// update u
if(altpred != pred && provider_component != -1){
  uint8_t u = tag_table[provider_component][tag_table_idx[provider_component]].u;
  if(predDir == resolveDir){
    tag_table[provider_component][tag_table_idx[provider_component]].u = SatIncrement(
  }
  else{
    tag_table[provider_component][tag_table_idx[provider_component]].u = SatDecrement(
  }
```

```
  }

  // After 256k branch, reset u
  clock ++;
  uint8_t mask = 0;
  if(clock == 1 << 18){
    mask = 1;
  }
  else if(clock == 1 << 19){
    mask = 2;
    clock = 0;
  }
  if(mask){
    for(int i = 0; i < TAGE_TABLE_NUM; i++){
      for (UINT32 j = 0; j < numTageTableEntries; j++){
        tag_table[i][j].u = tag_table[i][j].u & mask;
      }
    }
  }

  // update ghr
  ghr = ghr << 1;
  if(resolveDir){
    ghr++;
  }

}


UINT32 PREDICTOR::get_tagged_idx(UINT32 PC, int bank_no){
  __uint128_t temp_ghr = ghr;
  int history_width = tage_table_len[bank_no];
  UINT32 temp_pc = PC & ( (1 << TAGGED_ENTRY_LEN) - 1 );

  // folder
  while(history_width > 0){
    int block_width = min(history_width, TAGGED_ENTRY_LEN);
    temp_pc ^= temp_ghr & ( (1 << block_width) - 1 );
    temp_ghr = temp_ghr >> block_width;
    history_width -= block_width;
```

```
  }

  return temp_pc & ( (1 << TAGGED_ENTRY_LEN) - 1 ) ;
}

uint16_t PREDICTOR::get_tag(UINT32 PC, int bank_no){
  __uint128_t temp_ghr = ghr & ((1 << tage_table_len[bank_no])-1);
  // TODO
  return (temp_ghr + PC * 1000000007) & ((1<<TAG_WIDTH) - 1);
}
```

4.2 实验环境搭建

复现环境为：Ubuntu22-04 LTS 环境，以及 C/C++ 代码编译工具链

4.3 创新点

创新点是，新分配节点的优化，因为如果按照 PPM-LIKE 的逻辑，一个刚分配的 entry 很可能会被直接投入使用。但是在特定的测例下，新分配的 entry 有很大概率会出错，需要对这种情况做一些优化。

优化效果：大约有 0.2MPKI 的优化 (3.27->3.09)

对 Loop Predictor 也进行了优化，其能弥补 TAGE 的 GHR 长度有限，对于一些非常长的循环并不能有效预测；或者 TAGE 对于循环的反应可能不够灵敏的缺点

优化效果：大约有 0.04MPKI 的优化 (3.275->3.237)

5 实验结果分析

图 2. 实验结果

最终测试结果如图所示，MPKI 为 3.097，意味着 1000 条指令，预测错 3 条。

6 总结与展望

本文实现了 TAGE 算法，并且 MPKI 的值为 3.097，这个效果较为不错，TAGE 算法虽然目前已经展现出优秀的分支预测能力，但只在 x86 平台是进行了测试，没有在其他的平台，

如 arm，RISC v 进行测试。此外，进一步研究如何让 TAGE 算法的各个参数（如不同表的历史长度、表项分配概率等）能够更加自适应地根据程序的实时特性进行动态调整，而不是依赖于预先设定的固定规则。例如，开发一种实时监测程序分支行为变化的机制，然后自动调整 TAGE 算法中不同 tage 表的相对重要性，使其能够更快地适应程序从一种运行模式切换到另一种运行模式（如从普通计算阶段切换到频繁循环迭代阶段）的情况。

又或者结合人工智能技术，比如利用深度学习中的神经网络来学习分支指令的深层次行为模式，辅助 TAGE 算法对不同历史长度信息进行更智能的权重分配或者对预测结果进行进一步修正，有望进一步提高预测准确性，同时也可能简化一些复杂的人工设定的更新规则等机制。

参考文献

[1] SEZNEC Andre. A 256 kbits l-tage branch predictor. *The 2nd JILP Championship Branch Prediction Competition (CBP-2), 2006*, 2006.

[2] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.

[3] Pierre Michaud. A ppm-like, tag-based branch predictor. *The Journal of Instruction-Level Parallelism*, 7:10, 2005.

[4] André Seznec. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*, 2014.

[5] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 8:23, 2006.