体系结构项目报告

1 处理器预测算法综述

1.1 分支预测器的背景

处理器分支预测算法是为了提高流水线面对条件分支指令时候的性能。早期的流水线设计策略是，如果当前指令为条件分支指令，那下一条指令会被**阻塞**在流水线上，直到分支指令进行到**执行**阶段。分支预测器的目的是，在不知道分支指令的目标时，投机地立即执行一个分支。如果预测错误，则放弃所有中间结果并重新执行正确的分支。

1.2 分支预测与分支目标预测

考虑五级流水线系统，确定一条指令是分支指令只有在两个指令周期以后，即译码阶段才能实现。只有当确定一条指令是分支指令时，才能使用分支预测器，效率很低。分支目标预测(Branch Target Prediction)可以解决这个问题。

分支目标预测以当前指令地址作为输入，通过BTB，即分支目标缓存(Branch Target Buffer)来获取跳转地址。BTB中只保存跳转指令的地址，若输入PC在BTB中能找到，则是一个跳转指令；如果找不到，则不是。

整个分支预测系统的工作流程如下：

1. 在**取址阶段**，把PC值传入BTB进行寻址，如果命中，则返回跳转地址；
2. 之后利用分支预测器进行**二次验证**，假如预测结果是跳转(taken)，则执行BTB的返回指令；如果是不跳转(not taken)，则放弃BTB的返回值，执行PC+4的指令。

1.3 分支预测算法的历史

分支预测算法的发展趋势是从简单到复杂，从**静态**到**动态**，以在预测准确率和预测代价之间做到折中和平衡(tradeoff)。

1.3.1 静态预测算法

静态预测算法的代价很小，但是较低的准确率会造成流水线的性能下降。

- **静态**的含义是，预测时不依赖以往的运行结果，每一次执行相同的指令总会返回一个相同的预测结果
- CPU在**译码阶段**进行预测，也即能确定一条指令是条件分支的时候，作出预测判断，比如：阻塞流水线，永远返回token，永远返回not taken。
- 因为CPU在**译码阶段**才会预测，因此插入一条永远会执行的分支延迟间隙(branch delay slot)

1.3.2 动态预测算法

如上所述，**动态预测**算法会参考当前的分支指令之前的**指令的执行历史**。动态预测算法有很多，以下先对动态预测算法的基础进行叙述，之后在Gem5的背景下具体描述算法流程

- 动态预测器主要有两个基本的逻辑，一个是之前描述过的、用于缓存跳转地址的BTB，另一个是**分支历史表**(Branch History Table，简称BHT)，用于保存当前指令的执行历史。
- BHT利用PC进行索引，对应的值是跳转历史。跳转历史的保存形式主要是位。比如1bit预测算法即用1表述跳转，0表述不跳转。
- 动态预测算法的发展主要有两个方向：
  1. 改进跳转历史的保存形式：比如使用2bit或更多位来保存历史
  2. 使用多级预测，即针对不同的情况选择不同的历史进行预测

2 Gem5预测算法分析与实验

下面以Gem5模拟器为样例，分析其中CPU预测算法和具体实现。

2.1 Gem5 预测操作模型

2.1.1 模块分析

Gem5作为一个完整的硬件模拟器尽管很复杂，但是它处理CPU的预测操作的模型却遵循一个很清晰的逻辑。为了处理预测操作，Gem5需要如下几个模块相互协作：

1. ISA模型：CPU获取一条指令后需要通过ISA的接口，比如**操作码**，**寄存器号**等等接口对指令进行分析
2. PC：PC保存了当前运行指令的地址，可以用于寻址，以及作为预测算法的关键输入参数。

3. 预测器的相关存储单元：即BTB和BHT，前者存储指令的跳转地址，后者存储指令的跳转历史。

下面只考虑第三部分——BTB和BHT，以及运行其上的算法——的实现。

2.1.2 具体实现

1 -- BTB的实现

BTB简单来说就是对指令地址做的哈希表，因此其实现也包括了一个**顺序数组**和一些与Hash相关的接口：（`src/cpu/pred/btb.*`）

- 基本单元：**BTBEntry**

  ```
  struct BTBEntry {
      Addr tag; // 被索引的标签
      TheISA::PCState target; // 跳转目标
      bool valid; // 当前项是否有效
  }
  ```

  BTB中主要维护的就是一个BTBEntry的数组，通过在这个数组上的操作模拟硬件

- 基本接口：
  1. `lookup`：被预测器访问，返回参数指令所跳转的地址；如果BTB没有对应的有效存储，则返回0

     ```
      // 3个条件：对应索引有效，tag相同，线程号相同(单线程不考虑)
     if (btb[btb_idx].valid
         && inst_tag == btb[btb_idx].tag
         && btb[btb_idx].tid == tid) {

         return btb[btb_idx].target;
     } else {
         return 0;
     }
     ```

  2. `update`：这一步出现在被预测的分支指令获取最终跳转地址时，用最终“是否跳转”的结果来更新BTB

     ```
     void
     ```

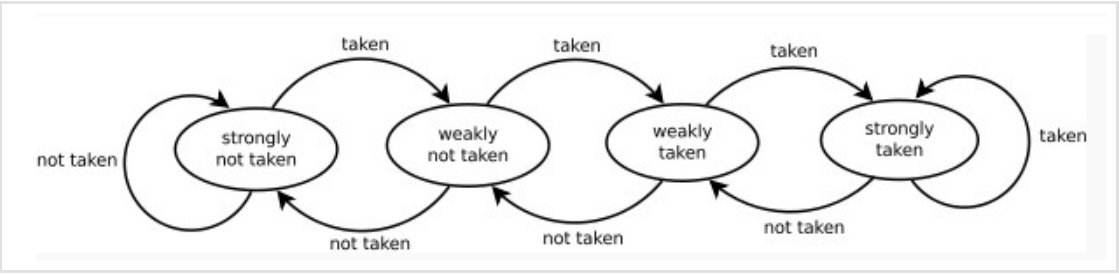
```
DefaultBTB::update(Addr instPC, const TheISA::PCSta
// instPC是被预测指令，target是预测目标
{
    unsigned btb_idx = getIndex(instPC);

    assert(btb_idx < numEntries);

    btb[btb_idx].tid = tid;
    btb[btb_idx].valid = true;
    btb[btb_idx].target = target;
    btb[btb_idx].tag = getTag(instPC);
}
```

2 -- BHT的实现

对于不同的Gem5的不同预测算法，这一部分的实现都不尽相同。主要原因是BHT与BTB不同，实现的并**不是简单的Hash机制**，不同算法的查询策略和更新策略都大不相同。因此这一部分在之后的算法实现中再加以描述。

但是大部分的BHT的设计都依赖一个存储单元，**饱和计数器**(Saturate Counter)。该计数器统计分支指令跳转的次数，若跳转则加1，否则减1。之所以称之为饱和计数器，是因为其计数值有上下限，比如一个2bit的饱和计数器，计数值不能超过3，也不能低于0。

这个机制的目的是什么？一个2bit的饱和计数器实际上对应了一个**有限状态自动机**的模型：



- taken与not taken的状态分别有两个：`strongly taken`和`weakly taken` 与 `strongly not taken` 和 `weakly not taken`。
- 四个状态的自动机可以更好地利用历史信息(相对于2个状态的1bit计数器而言)

在Gem5的实现：

```
void increment() { if (counter < maxval) ++counter; }
void decrement() { if (counter > 0) --counter; }
```

接下来进入对具体算法的分析

2.2 Gem5预测算法

之前描述了Gem5对BTB的实现，BTB作为对分支目标的缓存，只有当确定是否跳转时才会起作用。因此更关键的是预测是否跳转的算法。

正如前述，预测算法主要有两种，一种是不收集预测历史的静态预测算法，这种算法在Gem5中并未得到应用（但是我们会实现(to gcc:会吧？)）；另一种则是Gem5所用的动态预测算法，即通过与“历史缓存”BHT(Branch History Table，分支预测历史表)的交互获得历史信息，并把这些信息用在预测过程中。

不同的动态预测算法的差别主要在两个方面：

1. **单个BHT的优化**：主要是存储的历史作为什么形式存在，比如前述的1bit算法，和之后要提到的2bit算法；
2. **多个BHT的设计**：随着预测算法的发展，单个BHT的信息不足以提供准确的预测结果，因此利用多个BHT的算法开始得到关注，比如关联预测器，以及Gem5运用的tournament预测器

Gem5在`src/cpu/pred/`下提供了两个预测算法：`2bit_local`和`tournament`算法。前者是使用2个bit位作为BHT的存储单元，后者则是利用多个BHT来进行多级预测的算法。

2.2.1 2bit_local算法

基本原理：

其基本思想是维护`localCtrs`的数组，使得具有相同前缀的局部条件跳转指令为一个条目（元素），使用相同的预测结果，这就是local的含义。此处的预测结果，是通过预测位来确定的，在每个条目中的预测位，是之前这个条目中的跳转指令是否跳转的记录，由于存在局部性，若之前的指令跳转较多，则此次也预测跳转，否则预测不跳转。 具体预测方法将在下文阐述。

localCtrs设定

1. 指令映射规则 每个指令的PC设有设有n位，`instShiftAmt`为需要右

移的位数，右移后余下的前缀就是该指令映射到的数组元素下标。 `indexMask` 是前缀掩码。下列语句返回了下标：

```
return (branch_addr >> instShiftAmt) & indexMask;
```

2. 数组大小 程序中如下计算：

```
localPredictorSets = localPredictorSize / localCtrBits;
```

其中 `localPredictorSize` 是localPredictor的总大小， `localCtrBits` 是预测位个数，因为数组中只存放预测位，所以也就是每个元素的大小。 同时该数组的大小也要满足另一个条件： 因为前缀有 m-instShiftAmt 为，所以数组大小为2的m-instShiftAmt幂。

1. 初始化 初始化预测位，就是将每个预测位都恢复到 `initialVal` ，这个值可以通过构造函数传递，表示初始情况下预测位的值。

```
void LocalBP::reset(){ for (unsigned i = 0; i < localPredictorSets; ++i) { localCtrs[i].reset(); } }
void reset() { counter = initialVal; }
//localCtrs[i].reset()执行内容
```

预测位：

在Gem5中，预测位可以1位到多位，默认设位2位（可以在 `src/cpu/pred/BranchPredictor.py` 中修改）。每个预测位默认初始值设为0，可通过构造函数设定。 若某次该指令真实结果是跳转，则预测位加1(为最大值时不变），否则减1(为0时不变）。

```
if (taken) {
    DPRINTF(Fetch, "Branch updated as taken.\n");
    localCtrs[local_predictor_idx].increment();
}
else {
    DPRINTF(Fetch, "Branch updated as not taken.\n");
    localCtrs[local_predictor_idx].decrement();
}

void increment(){
    if (counter < maxVal) {++counter;}
}
```

```
void decrement(){
    if (counter > 0) {--counter;}
}
```

设预测位有n位，若counter < 2^(n-1)，表示之前不跳转的结果较多，这次预测跳转；反之counter > 2^(n-1)-1，则预测不用跳转。

2.2.2 tournament算法

基本原理：

Tournament算法之所以得名，主要是因为其维护了两个预测器：一个是基于全局信息的Global Predictor，一个是基于局部信息的Local Predictor，并在这两个预测器之间实现了竞争(tournament)的机制。下面分别介绍这三个模块的原理，以及具体的实现。

全局预测器

1. 基本原理
    - 分支预测算法中的**全局历史**指的是程序运行中所有的分支预测的结果，即整体运行中，跳转(taken)和不跳转(not taken)的“在饱和计数器意义下”的个数。
    - 在实现之前应该要考虑如下几个问题：
        1. 需要用到多少个历史分支指令？如果存储的分支指令过多，则会造成硬件开销过大。如果存储的过少，那么少量的特殊结果可能造成整体预测不精确
        2. 如何利用全局历史分支指令？最简单的做法是对历史结果维护一个饱和计数器，当前分支选择参考该计数器即可。同时也可以将全局结果和局部结果结合起来分析。
    - Gem5对全局预测器的实现思路：
        1. 维护一个饱和计数器的数组，和指向当前计数器的索引值
        2. 选择**当前计数器**的原则是：
            - 如果预测当前指令跳转，那么 `下一个计数器的索引值 = 当前索引值 * 2 + 1`
            - 如果预测当前指令不跳转，那么 `下一个计数器的索引值 = 当前索引值 * 2`

2. 具体实现

- 全局历史的数据容器：

```
std::vector<SatCounter> globalCtrs;
unsigned globalPredictorSize;
unsigned globalCtrBits;
```

显而易见，全局预测器维护的时一个饱和计数器的数组，PredictorSize与CtrBits分别指定数组大小和计数器位数

- 全局历史选择器：

```
unsigned globalHistory;
unsigned globalHistoryBits;
unsigned globalHistoryMask;
```

globalHistory是保存当前指定Entry的索引的寄存器值，Bits和Mask用于从该寄存器值中取数。比如获得当前全局历史在globalCtrs中的下标，需要：

```
unsigned global_predictor_idx =
    globalHistory & globalHistoryMask;
```

- 预测过程：

如果使用的是全局预测器，首先在全局预测器的历史表中获得预测结果：

```
global_prediction =
    globalCtrs[globalHistory & globalHistoryMask].r
```

之后更新globalHistory的值，如果预测跳转：

```
globalHistory = (globalHistory << 1) | 1;
```

如果预测不跳转：

```
globalHistory = (globalHistory << 1);
```

- 更新过程：

  当知道最终是跳转还是不跳转的时候，直接更新对应于当前历史结果的饱和计数器即可

3 预测算法实践