

# 高等计算机体系结构

## 第八讲: 分支预测

栾钟治  
北京航空航天大学 计算机学院 中德联合软件研究所  
2021-04-16

1

## 提醒: 作业

- 作业 3
  - 4月9日发布, 4月23日截止
  - 流水线1
- 作业 4
  - 4月23日发布, 5月7日截止
  - 流水线2

2

2

## 提醒: 实验

- 实验1, 今天截止
  - 用Logisim设计1个7指令单周期MIPS CPU
- 实验2-5, 预计5月7日发布, 7月11日截止

3

3

## 阅读材料

- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface* (计算机组成与设计: 软硬件接口)
  - 第四章 (4.9-4.11)
- 选读
  - Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
    - 更高级的流水线
    - 中断和异常处理
    - 乱序和超标量执行的概念
- 推荐阅读
  - McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993.

4

4

## 回顾：控制相关

- 问题: 下一个周期从PC里取出来的是什么?
- 答案: 下一条指令的地址
  - 所有的指令都和它们之前的指令存在控制相关。为什么?
- 如果取到的指令不是一个控制指令:
  - 下一次取的PC是下一条顺序执行的指令
  - 只要我们知道取到的指令尺寸就行了
- 如果取到的指令是控制指令:
  - 我们如何决定下一个要取的PC?
- 实际上, 我们怎么知道取的指令是不是一个控制指令?

5

5

## 回顾：分支的类型

类型	取指阶段能判断的分支方向	下一个可能地址的数量?	何时能够解析出下一个取指的地址?
条件分支	不知道	2	执行 (寄存器相关)
无条件分支	总是发生转跳	1	译码 (PC + offset)
调用	总是发生转跳	1	译码 (PC + offset)
返回	总是发生转跳	多	执行 (寄存器相关)
间接分支	总是发生转跳	多	执行 (寄存器相关)

不同类型的分支处理方式不同

6

6

## 回顾：如何处理控制相关

- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有:
  - 停顿流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址 (分支预测)
  - 采用延迟分支 (分支延迟槽/时隙)
  - 其它 (细粒度多线程)
  - 消除控制指令 (推断执行)
  - 从所有可能的方向取指 (如果知道的话) (多路径执行)

7

7

## 回顾：如何比停止取指更好...

- 与其等待关于PC的真相关被解决再行动, 不如猜测下一个PC = PC+4, 保持每个周期都取指  
这是好的猜测吗?  
如果猜错了会损失什么?
- ~20% 的指令组合是控制流
  - ~50% 的“向前”控制流 (比如 if-then-else) 被执行
  - ~90% 的“向后”控制流 (比如 loop) 被执行总的来说, 一般 ~70% 被执行, ~30% 不会执行  
[Lee and Smith, 1984]
- “下一个PC = PC+4”的期望在~86%的时间里是对的, 但是剩下那14%呢?

8

8

## 回顾：猜测下一个PC = PC + 4

- 如何能让这种方式更有效率?
- 思路：使下一条按顺序的指令就是下一条要执行的指令的可能性最大
  - 软件：制定控制流图，使得“可能的下一条指令”出现在不发生分支的路径上
  - 硬件：??? (如何能在硬件中做到这一点...)
- 还能怎样使这种方式更高效?
- 思路：去掉控制流指令（或者尽量减少它的发生）
  1. 去掉不必要的控制流指令 → 组合推断(把条件推断组合起来)
  2. 将控制相关转化为数据相关 → 推断执行

9

9

## 回顾：如何处理控制相关

- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有：
  - 停顿 流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址（分支预测）
  - 采用延迟分支（分支延迟槽/时隙）
  - 其它(细粒度多线程)
  - 消除控制指令（推断执行）
  - 从所有可能的方向取指（如果知道的话）（多路径执行）

10

10

## 回顾：推断执行

- 思路：将控制相关转化为数据相关
  - 消除分支 → 使代码“直线”推进（形成更大的基本代码块）
- 好处
  - 使预测“不发生分支”的效果更好（没有分支）
  - 编译器有更大的自由度来优化代码（没有分支）
    - 使控制流不妨碍指令的重排序优化
    - 只有数据相关会阻碍代码的优化
- 坏处
  - 无用功：一些指令被取指/执行，但最终被丢弃（尤其是在容易预测的分支中更糟糕）
  - 需要额外的 ISA 支持
- 可以像这样消除所有的分支吗？

11

11

## 回顾：如何处理控制相关

- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有：
  - 停顿 流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址（分支预测）
  - 采用延迟分支（分支延迟槽/时隙）
  - 其它(细粒度多线程)
  - 消除控制指令（推断执行）
  - 从所有可能的方向取指（如果知道的话）（多路径执行）

12

12

## 回顾：多路径执行

- 思路: 在条件分支之后两条路径都执行
- 好处:
  - + 当预测错的开销>无用功的开销时, 可以提高性能
  - + 不需要ISA做什么改变
- 坏处:
  - 当遇到下一个难预测的分支该怎么办? 再次执行两条路径?
  - 路径数成指数增加
  - 每条接下来的路径需要自己的寄存器, PC, GHR
  - 如果路径最终合并会导致无用功 (降低性能)

13

13

## 回顾：如何处理控制相关

- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有:
  - 停顿流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址 (分支预测)
  - 采用延迟分支 (分支延迟槽/时隙)
  - 其它 (细粒度多线程)
  - 消除控制指令 (推断执行)
  - 从所有可能的方向取指 (如果知道的话) (多路径执行)

14

14

## 回顾：分支延迟

- 改变分支指令的语义
  - N条指令后分支
  - N个周期后分支
- 思路: 延迟分支的执行, 无论分支的方向如何, 总是执行分支指令后紧跟的N条指令 (延迟槽)
- 问题: 如何找到指令填充延迟槽?
  - 分支必须与延迟槽指令相互独立
- 无条件分支: 更容易找到填充延迟槽的指令
- 条件分支: 条件计算不应依赖于延迟槽中的指令 → 填充延迟槽有难度

15

15

## 回顾：分支延迟

- 好处:
  - + 使流水线在一种简单的假设下保持充满有用的指令
  - 1. 延迟槽的数量 == 分支解决之前保持流水线充满的指令条数
  - 2. 所有的延迟槽可以用有用的指令填充
- 坏处:
  - 填充延迟槽不那么容易 (即使是2阶段流水线)
  - 1. 随着流水线深度或者超标量执行的宽度的增加, 延迟槽的数量也会增加
  - 2. 延迟槽的数量会随着操作延迟的变化而变化。为什么?
- ISA 语义与硬件实现的关系
  - SPARC, MIPS, HP-PA: 1 个延迟槽
  - 如果下一个设计中流水线的实现发生了变化会怎么样?

16

16

## 回顾：如何处理控制相关

- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有：
  - 停顿流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址（分支预测）
  - 采用延迟分支（分支延迟槽/时隙）
  - 其它(细粒度多线程)
  - 消除控制指令（推断执行）
  - 从所有可能的方向取指（如果知道的话）（多路径执行）

17

17

## 回顾：细粒度多线程

- 思路：硬件具有多线程的上下文。每个周期进行线程切换，指令获取机制从不同的线程获取指令，保证不会有同一个线程的两条指令并发的出现在流水线
  - 通过与来自其它线程有用的操作重叠延迟来容忍控制和数据相关带来的延迟，利用多线程的好处来改善流水线的利用率
- + 不需要做指令间的相关性检查（一个线程只有一条指令在流水线中）
- + 不需要分支预测逻辑
- + bubble周期被用来执行不同线程有用的指令
- + 改善系统的吞吐量，延迟容忍和利用率
- 额外的硬件复杂性：多硬件上下文，线程选择逻辑
- 单线程性能下降（每隔N个周期取一条指令）
- 线程之间对cache和memory的资源争用
- 仍然需要一些线程之间的相关性检查逻辑（load/store）

18

18

## 回顾：如何处理控制相关

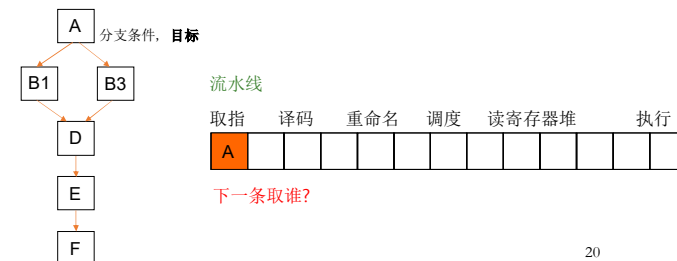
- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有：
  - 停顿流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址（分支预测）
  - 采用延迟分支（分支延迟槽/时隙）
  - 其它(细粒度多线程)
  - 消除控制指令（推断执行）
  - 从所有可能的方向取指（如果知道的话）（多路径执行）

19

19

## 回顾：分支预测

- 如何在存在分支时保持流水线满负荷？
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标
- 总是预测PC+4
  - 预测错误会受到清空流水线的惩罚



20

20

## 回顾：性能分析

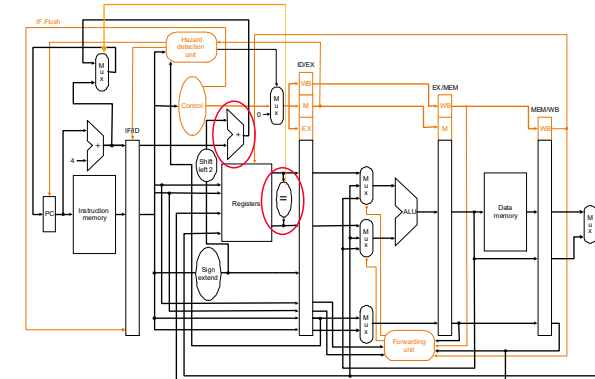
- 猜测正确  $\Rightarrow$  没有惩罚  $\sim 86\%$  的时间
  - 猜测不正确  $\Rightarrow$  2个气泡
  - 假设
    - 没有数据相关
    - 20% 的控制流指令
    - 70% 的控制流指令发生转跳
    - $CPI = [1 + (0.2 * 0.7) * 2]$   
 $= [1 + 0.14 * 2] = 1.28$
- 发生错误猜测的可能性      错误猜测的惩罚
- 我们有可能减小这两者中的任何一个吗?

21

21

## 回顾：减小分支预测错误的代价

- 提前处理分支条件和获得目标地址（分支判断提前）



[Based on original figure from P&H CO&O, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

$$CPI = [1 + (0.2 * 0.7) * 1] = 1.14$$

22

22

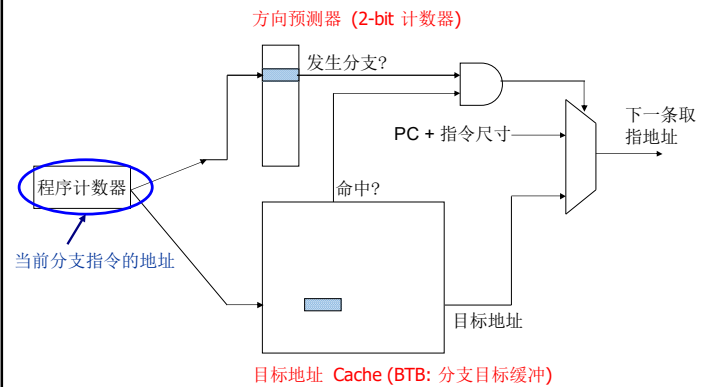
## 分支预测(增强版)

- 思路：预测下一个取指地址（下一个周期会用到）
- 需要在取指阶段预测三件事：
  - 取到的指令是不是一个分支指令
  - (条件) 分支的方向
  - 分支的目标地址 (如果分支发生)
- 观察：不同动态实例的条件分支目标地址可能是相同的
  - 思路：存储以前实例的目标地址，由PC访问它
  - 被称作分支目标缓冲 (BTB) 或者分支目标地址 Cache

23

23

## 有BTB的取指和方向预测

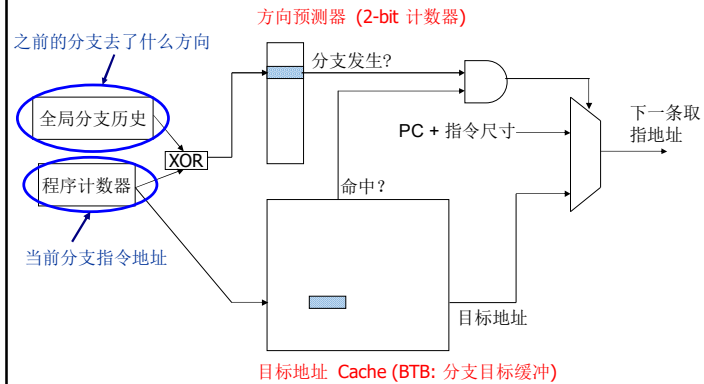


$$\text{总是发生的 } CPI = [1 + (0.2 * 0.3) * 2] = 1.12 \quad (70\% \text{ 的分支会发生})$$

24

24

## 更复杂的分支方向预测



25

25

## 简单的分支方向预测方案

- 编译时 (静态)
  - 总是不发生
  - 总是发生
  - BTFN (反向发生, 正向不发生)
  - 基于分析 (可能的方向)
- 运行时 (动态)
  - Last time 预测 (1-bit)

26

26

## 更复杂的方向预测

- 编译时 (静态)
  - 总是不发生
  - 总是发生
  - BTFN (反向发生, 正向不发生)
  - 基于剖析 (可能的方向)
  - 基于程序分析 (可能的方向)
- 运行时 (动态)
  - Last time 预测 (1-bit)
  - 基于2-bit计数器的预测
  - 两层预测 (全局vs. 局部)
  - 混合

27

27

## 静态分支预测(I)

- 总是不发生
  - 实现简单: 不需要BTB, 不需要方向预测
  - 准确率低: ~30-40%
  - 编译器可以重新布局代码, 这样能够使可能的路径就是“不发生分支”的路径
- 总是发生
  - 无方向预测
  - 更好的准确率: ~60-70%
    - 反向分支 (loop分支) 通常会发生
    - 反向分支: 目标地址比分支指令PC值小
- 反向发生, 正向不发生 (BTFN)
  - 预测反向(loop)分支总是发生, 其他的不发生

28

28

## 静态分支预测(II)

- 基于剖析 (profiling)
  - 思路: 编译器通过运行分析代码为每个分支决定可能的方向, 分支指令格式编码增加一个提示位表示分支方向
- + 逐个分支预测 (比前面讲到的方式更准确) → 如果分析代码有代表性就有准确率!
- 需要在分支指令格式中加提示位
- 准确性依赖于分支的动态行为:
  - TTTTTTTTTTNNNNNNNNN → 50% 准确率
  - TNTNTNTNTNTNTNTNTN → 50% 准确率
- 准确与否依赖于分析代码的输入数据集的典型性

29

29

## 静态分支预测(III)

- 基于程序 (或者基于程序分析)
  - 思路: 使用基于程序分析的启发式方法来确定静态预测的分支方向
  - 操作码启发式: 预测 BLEZ 不发生分支 (很多程序中用负整数代表错误值)
  - 循环启发式: 预测一个分支控制的循环操作会执行分支 (执行循环)
  - 指针的比较和浮点数的比较: 预测不相等
- + 不需要剖析
- 启发式方法可能不具有代表性或者不够好
- 需要编译器分析和ISA支持
- Ball and Larus, "Branch prediction for free," PLDI 1993.
  - 20% 的预测错误率

30

30

## 静态分支预测(III)

- 基于程序员
  - 思路: 程序员提供静态预测的方向
  - 通过编程语言中的Pragma使分支成为可能发生或可能不发生的分支
- + 不需要剖析或程序分析
- + 相比那些分析技术来说, 程序员可能对程序或分支更了解
- 需要编程语言、编译器和ISA支持
- 增加程序员的负担?

31

31

## Pragma

- 思路: 使程序员可以向更低层次的转换转达一些提示的关键词
- if (likely(x)) { ... }
- if (unlikely(error)) { ... }
- 很多提示和优化可以通过pragma实现
  - 例如, 一个循环是否可以并行化
  - #pragma omp parallel
  - 描述
    - 一个OpenMP并行指令, 显式地指示编译器对选定的代码段进行并行化

32

32



## 静态分支预测

- 所有前面讲到的技术都可以组合
  - 基于剖析 (profile)
  - 基于程序 (program)
  - 基于程序员 (programmer)
- 如何组合?
- 这三种技术有什么共同的缺陷?
  - 不能适应分支行为的动态变化
    - 可以利用动态编译器来解决这个问题, 但是不是细粒度的 (同时动态编译器也有它自己的开销...)

33

33

## 动态分支预测

- 思路: 基于动态信息预测分支 (运行时采集信息)
- 好处
  - + 基于分支执行的历史预测
  - + 可以适应分支行为的动态变化
  - + 无需剖析: 输入集的典型性问题不复存在
- 坏处
  - 更加复杂 (需要额外的硬件)

34

34

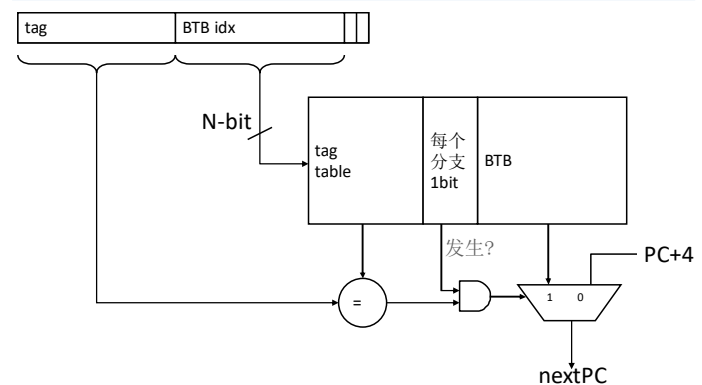
## Last Time 预测器

- Last time 预测器
    - 每个分支1bit (存在BTB中)
    - 显示上一次分支执行时的方向
- TTTTTTTTTNNNNNNNNN → 90% 准确率
- 对于循环分支总是预测错第一次和最后一次迭代
    - 对于N次迭代循环的准确率 =  $(N-2)/N$
- + 有大量迭代的循环分支
- 只有少量迭代的循环分支
- TNTNTNTNTNTNTNTN → 0% 准确率
- Last-time 预测器 CPI =  $[1 + (0.20 * 0.15) * 2] = 1.06$  (假设 85% 准确率)

35

35

## 实现Last-Time 预测器

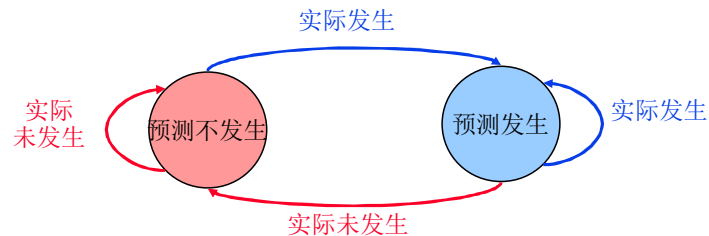


1-bit BHT (分支历史表) 表项在每次执行完一个分支后更新正确的结果

36

36

## Last-Time 预测的状态机



37

37

## 改进的Last Time 预测器

- 问题：last-time 预测器改变预测太快（ $T \rightarrow NT$ 或者 $NT \rightarrow T$ ）
  - 即使分支可能大部分发生或者大部分不发生
- 解决思路：为预测器增加滞后效果，让预测不要因为出现1次不同的结果就改变
  - 使用2bits而不是1bit跟踪分支预测的历史
  - T或者NT可以分别有2个状态
- Smith, “A Study of Branch Prediction Strategies,” ISCA 1981.

38

38

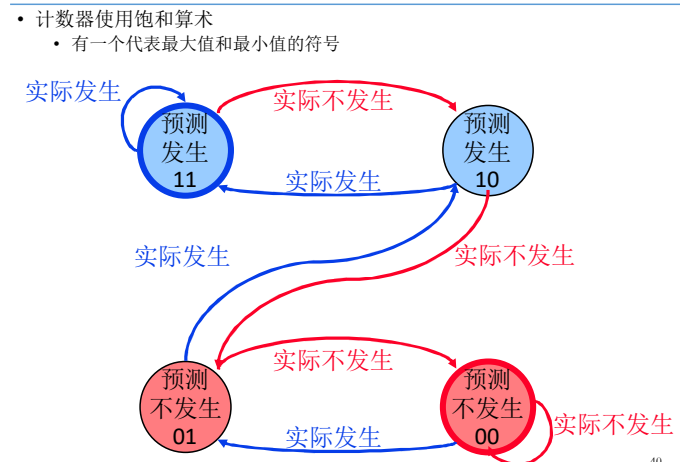
## 基于2-Bit计数器的预测

- 每个分支关联一个2-bit计数器
  - 增加的1bit提供一个“滞后”
  - 强预测不会因为一次不同结果而改变
- $N$ 次迭代循环的准确率 =  $(N-1)/N$   
 $TNTNTNTNTNTNTNTNTN \rightarrow 50\%$  准确率  
 (假设初始时弱发生)
- + 更好的预测准确率  
 $2BC \text{ 预测器 } CPI = [1 + (0.20 \times 0.10) \times 2] = 1.04 \text{ (90\% 准确率)}$   
 — 更多的硬件开销 (但是计数器可以是BTB表项的一部分)

39

39

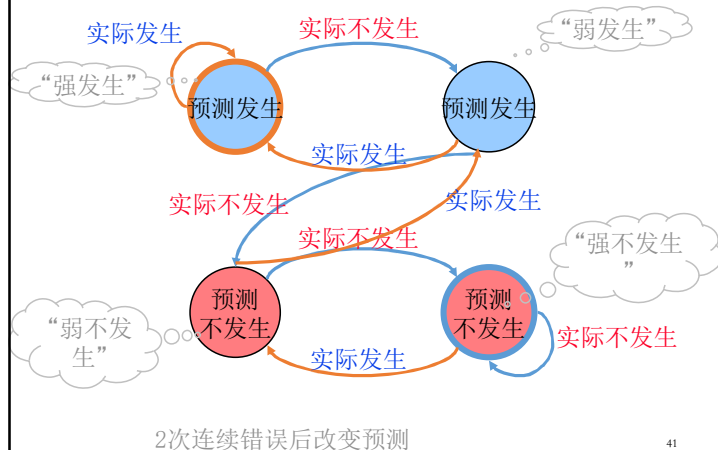
## 2-bit 饱和计数器的状态机



40

40

## 使用2-bit计数器产生的滞后效应



41

## 够好了吗?

- 很多基于2-bit预测的程序有~85-90% 的准确率 (也叫做双模态预测)
- 这样足够好了吗?
- 分支的问题有多大?

42

## 重新思考分支问题

- 控制流指令(分支) 很常见
  - 占有指令的15-25%
- 问题: 控制流指令之后的下一个取指地址在流水线处理器中会在N个周期后仍难以确定
  - N个周期: (最小) 分支解决延迟
  - 分支时停顿浪费指令处理带宽 (降低IPC)
    - $N \times IW$  个指令槽被浪费 (IW: 发射宽度)
- 如何在分支之后保持流水线充满?
- 问题: 需要在分支指令被取出时决定下一个取指地址 (避免流水线气泡)

43

## 分支问题的重要性

- 假设一个5发射宽度的超标量流水线有20个周期的分支解决时延
- 取500条指令要花费多长时间?
  - 假设连续取指, 并且5条指令中有1条是分支
  - 100% 准确率
    - 100 个周期 (获取的所有指令都在正确的路径)
    - 没有做无用功
  - 99% 准确率
    - $100 \text{ (正确路径)} + 20 \text{ (错误路径)} = 120 \text{ 个周期}$
    - 20% 额外的取指
  - 98% 准确率
    - $100 \text{ (正确路径)} + 20 * 2 \text{ (错误路径)} = 140 \text{ 个周期}$
    - 40% 额外的取指
  - 95% 准确率
    - $100 \text{ (正确路径)} + 20 * 5 \text{ (错误路径)} = 200 \text{ 个周期}$
    - 100% 额外的取指

44

## 能不能做的更好?

- Last-time和2BC预测器利用 “last-time” 可预测性
- 认识1: 一个分支的结果可能和其它分支的结果相关
  - 全局分支相关
- 认识2: 一个分支的结果可能和同一个分支过去的结果相关 (不仅仅是上一次分支执行的结果)
  - 本地分支相关

45

45

## 全局分支相关(I)

- 最近一个执行分支的结果与下一个分支结果相关

```
if (cond1)
...
if (cond1 AND cond2)
```

- 如果第一个分支不发生, 第二个也不会发生

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

- 如果第一个发生了, 第二个肯定不会发生

46

46

## 全局分支相关(II)

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)
```

- 如果 Y 和 Z 都发生, X 也发生
- 如果 Y 或 Z 不发生, X 也不发生

47

47

## 全局分支相关(III)

- Eqntott, SPEC 1992

```
if (aa==2) ;; B1
    aa=0;
if (bb==2) ;; B2
    bb=0;
if (aa!=bb) { ;; B3
    ....
}
```

如果 B1 不发生 (aa=0@B3) 并且 B2 不发生 (bb=0@B3), 则 B3 肯定不发生

48

48

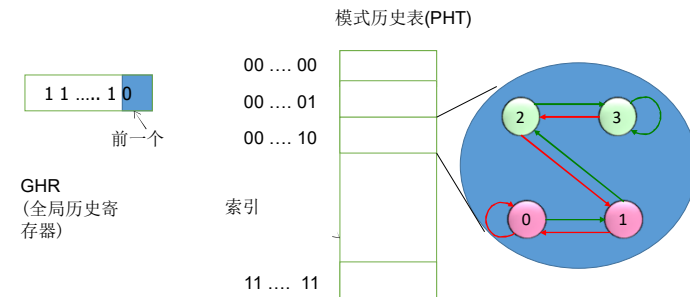
## 捕获全局分支相关

- 思路：将分支结果与所有分支的“全局T/NT历史”关联
- 根据上一次相同全局分支历史的分支结果作出预测
- 实现：
  - 用一个寄存器跟踪所有分支的“全局T/NT历史”→全局历史寄存器 (GHR)
  - 使用GHR索引到一张表，表中记录了最近的过去与GHR中值相应的分支的结果 → 模式历史表 (2-bit计数器表)
- 全局历史/分支预测器
- 使用两个层次的历史 (GHR+GHR的历史)

49

## 两层全局分支预测

- 第一层：全局分支历史寄存器 (N bits)
  - 前N次分支的方向
- 第二层：每个历史表项的饱和计数器表
  - 上一次相同的历史情况下的分支方向

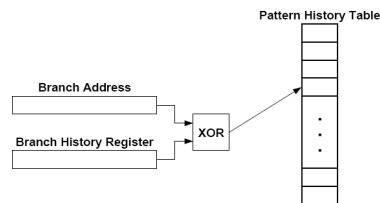


Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

50

## 改进全局分支预测的准确性

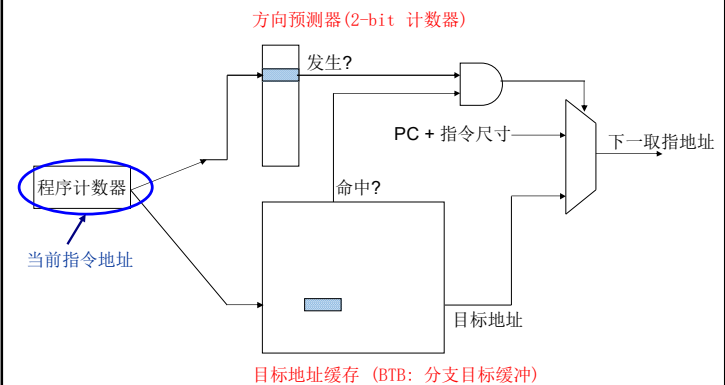
- 思路：为全局预测器增加更多的上下文信息来决定预测哪一个分支
  - Gshare预测器：GHR按分支地址哈希
    - 更多的上下文信息
    - 更好地利用模式历史表
    - 增加访问延迟



- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

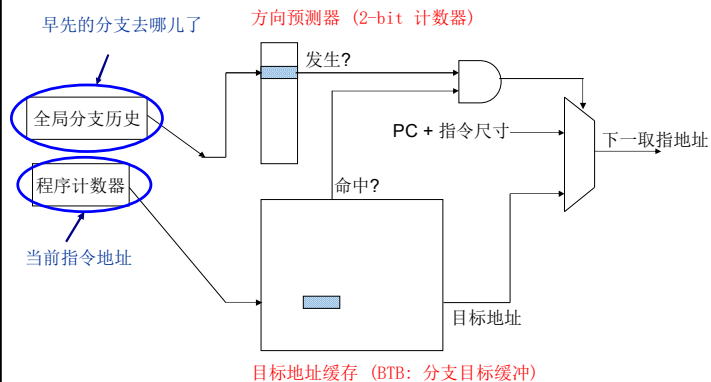
51

## 一层分支预测器



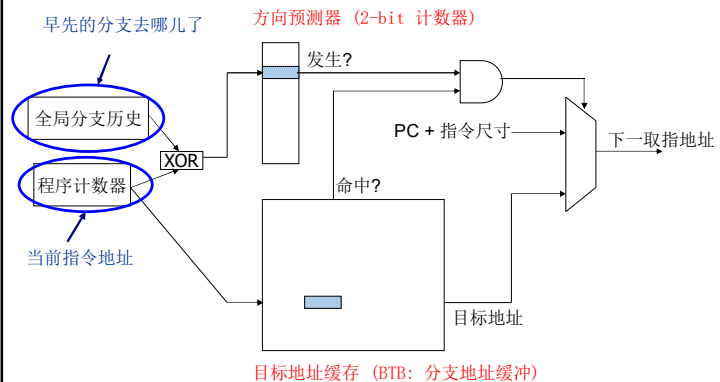
52

## 两层全局历史预测器



53

## 两层Gshare预测器



54

## 还能更好吗?

- Last-time和2BC预测器利用 “last-time” 可预测性
- 认识1: 一个分支的结果可能和其它分支的结果相关
  - 全局分支相关
- 认识2: 一个分支的结果可能和同一个分支过去的结果相关 (不仅仅是跟 “last-time” 分支执行的结果)
  - 本地分支相关

55

55

## 本地分支关联

```
for (i=1; i<=4; i++) { }
```

If the loop test is done at the end of the body, the corresponding branch will execute the pattern (1110)<sup>n</sup>, where 1 and 0 represent taken and not taken respectively, and <sup>n</sup> is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

- McFarling, “Combining Branch Predictors,” DEC WRL TR 1993.

56

56

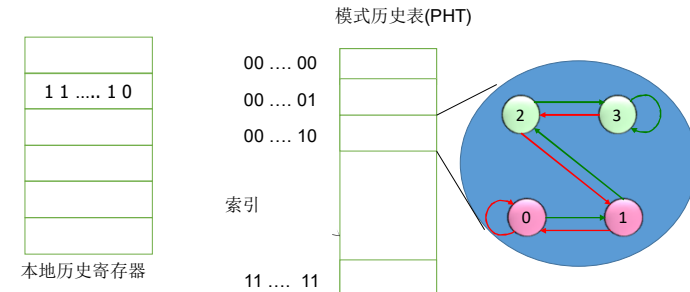
## 捕获本地分支的关联性

- 思路：每个分支都有一个历史寄存器
  - 将分支预测结果与该分支在“历史上发生/不发生”关联
- 根据上一次相同本地分支历史的分支结果作出预测
- 称为本地历史/分支预测器
- 使用两个层次的历史（每个分支历史寄存器 + 取那一个历史寄存器值的历史）

57

## 两层本地历史预测器

- 第一层：一组本地历史寄存器（每个N bits）
  - 选择基于分支指令地址的历史寄存器
- 第二层：每一个历史条目的饱和计数器表
  - 上一次相同的历史情况下的分支方向

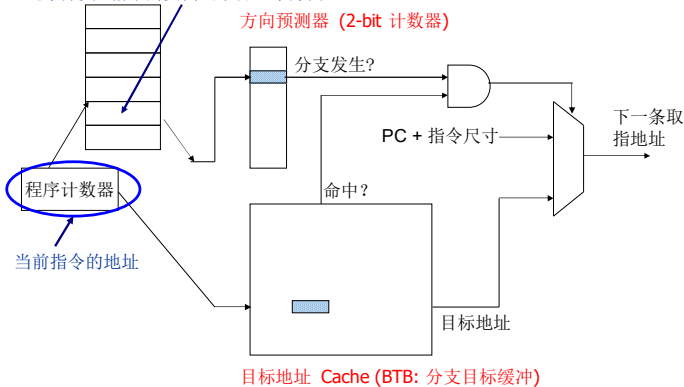


Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991. 58

58

## 两层本地历史预测器

\*这个分支\* 稍早的实例去了哪一个方向



59

59

## 混合分支预测器

- 思路：使用不止一种类型的预测器（采用多种算法），选择“最佳”的预测
  - 比如，2-bit 计数器和全局预测器的混合

- 好处：
  - + 更好的准确率：不同的预测器更适用不同的分支
  - + 减少“热身”时间（先使用“进入状态”快的预测器，直到“慢热”的预测器热身完毕）

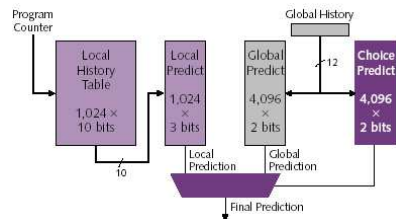
- 坏处：
  - 需要“元预测器”或“选择器”
  - 更长的访问时延

McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

60

60

## Alpha 21264 锦标赛预测器（混合预测）



- 最小的分支惩罚: 7 cycles
- 典型的分支惩罚: 11+ cycles
- 48K bits 的目标地址保存在 I-cache
- 预测器表在上下文切换时重置
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

61

61

## 分支预测准确率(示例)

- 双模态: 由分支地址索引的2bc表

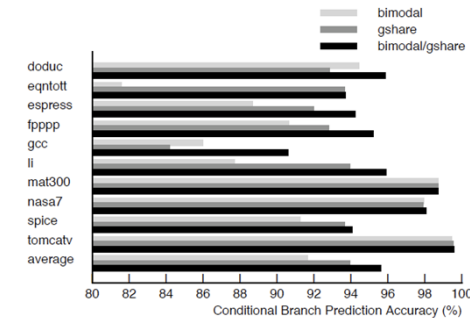


Figure 13: Combined Predictor Performance by Benchmark

62

62

## 有偏向性的分支

- 观察: 很多分支会偏向某一个方向 (比如, 99% 发生)
- 问题: 这些分支破坏了分支预测的结构 → 给分支预测表和历史寄存器造成“干扰”, 使得对其它类型的分支预测变得困难
- 解决方案: 检测这样的有偏向性的分支, 用更简单的预测器预测它们
- Chang et al., "Branch classification: a new mechanism for improving branch predictor performance," MICRO 1994.

63

63

## 回顾: 分支类型

类型	取指阶段能判断的分支方向	下一个可能地址的数量?	何时能够解析出下一个取指的地址?
条件分支	不知道	2	执行 (寄存器相关)
无条件分支	总是发生转跳	1	译码 (PC + offset)
调用	总是发生转跳	1	译码 (PC + offset)
返回	总是发生转跳	多	执行 (寄存器相关)
间接分支	总是发生转跳	多	执行 (寄存器相关)

不同类型的分支处理方式不同

64

64



## 调用和返回预测

### • 直接调用容易预测

- 总是发生, 单个目的地址
- 调用在BTB中标记, 由BTB预测目的地址

Call X  
...  
Call X  
...  
Call X  
...  
Return  
Return  
Return

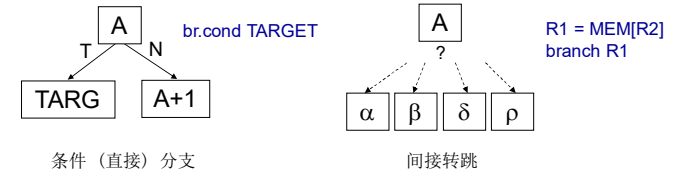
### • 返回是间接分支

- 函数可以由代码中的多个点调用
- 返回指令可能有多个目的地址
  - 相同函数的每一个调用点的下一条指令
- 观察: 通常每个返回对应一个调用
- 思路: 使用栈来预测返回地址 (返回地址栈)
  - 取到调用指令: 返回地址 (下一条指令) 压入堆栈
  - 取到返回指令: 弹出堆栈, 使用该地址作为预测的目的地址
  - 大部分时间准确: 8-entry栈 → > 95% 准确率

65

## 间接分支预测(I)

### • 寄存器间接分支有多个目标地址



### • 用来实现

- Switch-case 语句
- 虚函数调用
- 转移表 (函数指针)
- 接口调用

66

## 间接分支预测(II)

- 不需要预测方向
- 思路 1: 预测上一次解析的目标就是下次要取的地址
  - + 简单: 使用BTB 存储目标地址
  - 不准: 50% 准确率 (经验数据). 很多间接分支会在不同的目标之间切换
- 思路 2: 基于历史的目标预测
  - 比如, 用GHR XOR 间接分支PC来索引BTB
  - Chang et al., "Target Prediction for Indirect Jumps," ISCA 1997.
  - + 更准确
  - 一个间接分支会映射到 (可能很) 多个BTB表项
  - 会与其他分支发生Conflict miss (直接或间接)
  - 在分支只有极少的目标地址的情况下, 空间利用率低

67

## 分支预测中的问题(I)

### • 需要在取指结束之前识别出分支

### • 如何做到?

- BTB 命中 → 说明取的指令是一个分支
- BTB 表项包含一个分支的“类型”

### • 如果没有BTB怎么办?

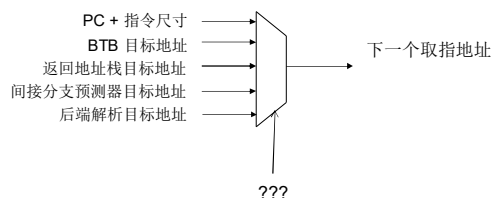
- 在流水线中加气泡直到目标地址计算出来
- 比如, IBM POWER4

68

## 分支预测中的问题(II)

### • 时延: 时延对预测很关键

- 需要为下一个周期产生取指地址
- 更大更复杂的预测器更准确但是更慢



69

69

## 超标量处理器

### • “超标量”处理器

- 尝试每个时钟周期执行超过1条指令
- 每个周期必须取多条指令

### • 考虑 2-way超标量取指的场景

(case 1) 两条指令都是未发生的控制流指令

- $nPC = PC + 8$

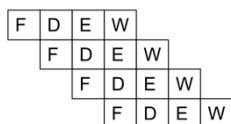
(case 2) 其中一条指令是发生的控制流指令

- $nPC =$  预测的目标地址
- \*注意\* 两条指令都可以是控制流; 基于第一条指令预测发生与否
- 如果第一条指令是预测发生的分支  
→ 使第二条指令的取指失效

70

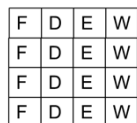
70

## 多指令取指: 概念



每周期取1条指令

缺点: Flynn 瓶颈  
如果每周期取1条指令, 那么每周期不可能完成>1条指令



每周期取>1条指令

两种可行的方法:  
1、VLIW  
编译器决定哪些指令可以并行执行, 硬件简单  
2、超标量  
硬件检测同一个周期取到的指令之间的相关性

71

71

## 回顾: 处理控制相关

### • 处理流水线中的控制相关

- 分支延迟
- 细粒度多线程
- 分支预测
  - 编译时(静态)
    - 总是不发生, 总是发生, 反向发生正向不发生, 基于分析
  - 运行时(动态)
    - Last time 预测器
    - 滞后: 2BC 预测器
    - 全局分支相关 → 两层全局预测器
    - 本地分支相关 → 两层本地预测器
- 推断执行
- 多路径执行

72

72

## 回顾：流水线设计中的问题

- 流水段的平衡
  - 需要多少段以及每一段完成什么任务
- 有影响流水的事件时，保持流水线正确、顺畅、满负荷
  - 处理相关性（冒险）
    - 数据
    - 控制
  - 处理资源争用
  - 处理长时延（多个周期）操作
- 处理异常、中断
- 更高的要求：提高流水线的吞吐
  - 使停顿最少

73

73

## 流水和精确异常：保持连续的语义

74

## 多周期执行

- 不是所有指令的“执行”时间都是一样长的
- 思路：有多个不同的功能单元，会花费不同数量的时钟周期
  - 可以流水可以不流水
  - 能够使独立的指令在不同的功能单元上，在前序的长延时指令执行完毕之前就开始执行
- 程序序的保证和精确异常

75

75

## 流水线中的问题：多周期执行

- 指令在执行阶段可能花费不同数量的时钟周期
  - 整型的 ADD vs. 浮点数的 MULT



- 这幅图有什么问题吗？
  - 如果FMUL发生了异常会怎么样？
  - ISA的连续语义是不保存的！

76

76

## 异常 vs. 中断

### • 起因

- 异常: 内部产生, 作用于运行的线程
- 中断: 外部产生, 作用于运行的线程

### • 处理的时机

- 异常: 一旦检测出 (并且已经知道非投机)
- 中断: 方便的时候
  - 除了非常高优先级的
    - 电源失效
    - 机器自检

### • 优先级: 必须处理当前进程 (异常), 不一定 (中断)

### • 处理的上下文: 进程 (异常), 系统 (中断)

77

77

## 精确异常/中断

- 当准备处理异常/中断时, 体系结构状态应该是一致的

1. 所有之前的指令必须完全回收

2. 之后的指令一律不得回收

回收(Retire) = 提交(commit) = 执行完毕并且更新体系结构状态

78

78

## 为什么希望精确异常?

### • 冯诺依曼模型ISA的语义

- 冯诺依曼模型vs. 数据流

### • 帮助软件调试

• 使异常恢复更容易, 比如页失效的恢复

• 使进程重启更容易

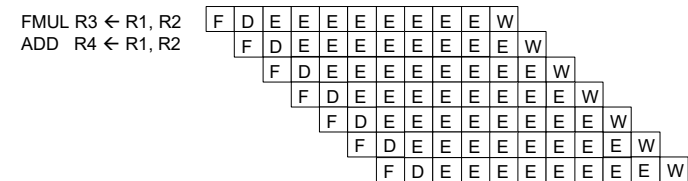
• 可以在软件中加入trap

79

79

## 流水线中确保精确异常

- 思路: 使每个操作花同样的时间



### • 缺点

- 访存操作会怎么样?
- 每个功能单元100个时钟周期?

80

80

## 解决方案

### • 重排序缓冲

• 历史缓冲

• 未来寄存器堆

• 检查点

• 推荐阅读

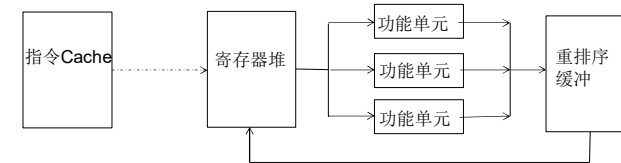
- Smith and Plezskun, “[Implementing Precise Interrupts in Pipelined Processors](#),” IEEE Trans on Computers 1988 and ISCA 1985.
- Hwu and Patt, “[Checkpoint Repair for Out-of-order Execution Machines](#),” ISCA 1987.

81

81

## 解决方案 I: 重排序缓冲 (ROB)

- 思路: 乱序执行指令, 产生体系结构状态可见的结果之前重排序
- 当指令译码时在ROB中预留一个条目
- 当指令执行完时, 将结果写入ROB中相应条目
- 当指令成为ROB中最旧的一条, 并且已经执行完 (没有异常) 时, 将结果移动到寄存器堆或者存储器



82

82