指令级并行性(1)

概念

- 指令级并行性(instruction-level parallelism, ILP)：指令之间的重叠执行。
- 开发指令级并行性的两类主要方法
  - 依靠硬件来动态地发现和利用并行性
    - 用于服务器和台式机处理器（如 Intel 和 ARM 处理器）
    - 在个人移动设备（如平板电脑和高端手机）中的处理器
    - 在 IoT 领域，成本和功耗约束是主要目标，利用较低层次的指令级并行性
  - 依靠软件在编译时静态地发现并行性
    - 上世纪 80 年代开始尝试，1999 年用于 Intel Itanium（安腾）系列
    - 特定领域环境 或 带有明显数据级并行性的科学应用
- 局限性
  - 直接导致了多核(multicore)处理器
  - 平衡指令级并行性与线程级平行性

概念

- 流水线处理器的 CPI 方程

$$CPI_{pipeline} = CPI_{Ideal\ pipeline} + Stalls_{structural} + Stalls_{data\ hazard} + Stalls_{control}$$

- 主要技术对 CPI 方程的影响

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Simple branch scheduling and prediction	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences
Loop unrolling	Control hazard stalls
Advanced branch prediction	Control stalls
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences
Hardware speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Issuing multiple instructions per cycle	Ideal CPI
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls

指令级并行性

- 挖掘指令级并行性的所有技术都利用指令之间的并行性
- 基本块(basic block)
  - 一段直线代码序列
  - 除了入口没有分支转入 且 除了出口也没有分支转出
- 并行性
  - 基本块一般长度为 3 ~ 6 条指令（分支指令的频度 15%~25%），基本块内可利用的并行度很小
  - 提高性能必须跨分支指令（或跨多个基本块）优化

相关与危害

- 程序的本性：指令间存在相关
- 流水线结构(pipeline organization)的性质：
  - 相关(dependence)是否导致检测到实际危害(hazard)
  - 危害(hazard)是实际上引起停顿(stall)
    - 停顿(stall)（阻塞、拖延、暂停）将直接造成流水线性能下降
- 确定一条指令如何依赖另一条指令是关键的，它决定着程序中
  - 存在多少并行性
  - 如何开发并行性
- 三种类型的相关
  - 数据相关(data dependences)，也称真实数据相关
  - 名称相关(name dependences)
  - 控制相关(control dependences)

数据相关与数据危害

- 指令 j 与指令 i 数据相关的条件是
  - 指令 i 产生的结果被指令 j 使用 或
  - 指令 j 与指令 k 数据相关，而指令 k 与指令 i 数据相关——相关链 (dependence chain)

- 例：数组元素加上一个标量

```
Loop:   fld     f0,0(x1)    // f0=array element
        fadd.d  f4,f0,f2    // add scalar in f2
        fsd     f4,0(x1)    // store result
        addi    x1,x1,-8    // decrement pointer 8 bytes
        bne     x1,x2,Loop  // branch x1≠x2
```

  - 后面的指令依赖于前面的指令
  - 为正确执行必须保持某种次序

数据相关

- 如果两条指令是数据相关的，则
  - 必须按序执行
  - 不能同时执行或完全重叠执行
- 数据相关表示
  - 危害的可能性
  - 必须遵循的计算顺序
  - 可能利用指令级并行性的上限
- 解决方法
  - 维持数据相关，但避免数据危害
    - 调度代码是在不改变数据相关的情况下避免数据危害的主要方法，可以由编译器和硬件来完成
  - 通过转换代码来消除数据相关
- 难点：存在于内存中的数据相关难于检测

名称相关

- 名称相关(name dependence)：两条指令使用的相同寄存器或内存位置，称为名称(name)，但关联指令之间不存在数据流。
- 假设：指令 i 位于指令 j 之前
  - 反相关(anti-dependence)：指令 j 写入和指令 i 读取的寄存器名或内存位置是相同的
  - 输出相关(output dependence)：指令 i 和指令 j 写入的寄存器名或内存位置是相同的
- 不是真正数据相关，但在调度指令（重新排序）时会发生问题。
- 通过改变指令中的名称（寄存器名或存储位置），使得指令不再冲突（危害），则名称相关的指令是可以同时执行的。
  - 使用寄存器重命名(register renaming)技术解决名称相关问题
  - 寄存器重命名技术可以通过编译器静态地完成，也可以通过硬件动态地实现。

寄存器重命名

- 例：

```
fdiv.d    f0,f2,f4
fadd.d    f6,f0,f8
fsd       f6,0(x1)
fsub.d    f8,f10,f14
fmul.d    f6,f10,f8
```

  - 反相关：fadd.d-fsub.d,fsd-fmul.d
  - 输出相关：fadd.d-f.mul.d
  - 数据相关：fdiv.d-fadd.d,fsub.d-fmul.d,fadd.d-fsd

  - 寄存器重命名：只剩下 RAW 危害

```
fdiv.d    f0,f2,f4
fadd.d    S,f0,f8
fsd       S,0(x1)
fsub.d    T,f10,f14
fmul.d    f6,f10,T
```

寄存器重命名

- 通过重命名所有目标寄存器来消除 WAR 和 WAW 危害
  - 需要充足的寄存器资源
- 寄存器
  - 逻辑寄存器⇔“名称”，编程模型中使用的寄存器
  - 物理寄存器⇔“位置”，硬件设备中存在的寄存器
- 示意图

Initial Mapping

Map Table			
R1	R2	R3	R4
P1	P2	P3	P4
P5	P2	P3	P4
P5	P2	P3	P6
P5	P2	P7	P6
P8	P2	P7	P6

ADD	R1,	R2,	R4	ADD	P5,	P2,	P4
SUB	R4,	R1,	R2	SUB	P6,	P5,	P2
ADD	R3,	R1,	R3	ADD	P7,	P5,	P7
ADD	R1,	R3,	R2	ADD	P8,	P7,	P2

数据危害

- 目标：在影响程序执行结果的地方维持程序顺序来开发并行性。
- 程序顺序(program order)：源程序所确定的指令逐条执行的顺序

- 数据危害(data hazard)依赖于指令中读和写的顺序
- 类型
  - 写后读(Read After Write, RAW)：先写后读，真正数据相关。
  - 写后写(Write After Write, WAW)：顺序写，输出相关。
    - 流水线中有多个流水段可写，或，允许指令乱序执行
  - 读后写(Write After Read, WAR)：先读后写，反相关（名称相关）。
    - 指令在流水线早期写结果，在晚期读结果；或指令被重排序
  - 读后读(Read After Read, RAR)：顺序读，非相关。

控制相关

- 控制相关(control dependence)：If 语句 then 部分中的指令对分支指令的相关

```
if p1{
      S1;
};
if p2{
      S2;
}
```

  - S1 与 p1 控制相关。
  - S2 与 p2 控制相关，但与 p1 不存在控制相关。

- 控制相关施加的两条约束
  - 与分支指令存在控制相关的指令不能移动到分支指令之前，以使得它的执行不再受分支指令的控制。
  - 与分支指令不存在控制相关的指令不能移动到分支指令之后，以使得它的执行受分支指令的控制。

控制相关

- 因为违反控制相关不一定影响程序的正确性，故
  - 控制相关不是必须维持的
  - 中断行为和数据流是必须维持的
- 中断行为：改变指令执行顺序不能改变程序中引发中断的方式。
  - 指令执行的重新排序不能在程序中引起任何新的中断
  - 例：

```
        add  x2,x3,x4
        beq  x2,x0,L1
        ld   x1,0(x2)
L1:
```

    - 没有数据相关妨碍交换 beq 指令和 ld 指令；
    - 忽略控制相关，将 ld 指令移动到 beq 指令之前，ld 指令可能引起内存保护中断。

控制相关

- 数据流：在生成结果指令和使用该结果指令间真实的数据流
  - 程序顺序(program order)：一条指令可以与它前面的多条指令存在数据相关，但这条指令只从其前面的一条指令中获取数据。
  - 通过维持控制相关来保证程序顺序。
  - 例

```
        add x1,x2,x3
        beq x4,x0,L
        sub x1,x5,x6
L:      …
        or x7,x1,x8
```

    - or 指令使用的 x1 数值取决于分支指令转移与否
    - or 指令与 add 指令和 sub 指令间存在数据相关，但仅保持顺序不足以正确执行
    - 指令执行时可以通过保持控制相关来维持正确的数据流

控制相关

- 违反控制相关可能不会影响中断行为或数据流
  - 例：假如知道转移之后不使用寄存器 x4 ，则可以将 sub 指令移动到分支指令之前。

```
        add x1,x2,x3
        beq x12,x0,skip
        sub x4,x5,x6
        add x5,x4,x9
skip:   or x7,x8,x9
```

```
        add x1,x2,x3
        sub x4,x5,x6
        beq x12,x0,skip
        add x5,x4,x9
skip:   or x7,x8,x9
```

  - 这种类型的代码调度是推测(speculation)的一种形式，称为软件推测。
  - 推测的含义就是“赌”！上述情形的赌注就是“分支不转移”。
- 通过完成对引起控制停顿(control stall)的控制危害的检测来维持控制相关
  - 控制停顿可以通过硬件或软件技术消除或减少

利用指令级并行性的基本编译技术

- 基本流水线调度
  - 为了防止流水线停顿，相关指令的执行必须与源指令相距一定的周期；
  - 相距的周期数等于指令流水线中源指令的延迟周期数。
- 编译器的调度能力取决于程序中的 ILP 和功能单元的延迟
- 浮点数单元中两条相关指令之间的延迟周期数

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

  - 假设为标准五段指令流水线，故分支转移指令的延迟为 1 个周期。
  - 假设功能单元是完全流水的或重复的，故每个时钟周期可以发出任何类型指令，且不存在结构危害(structural hazards)。

循环并行性

- 循环并行性(loop-level parallelism)：循环迭代之间的并行性
  - 增加指令级并行性的最简单且最常用的方式
  - 静态或动态的循环展开(unrolling the loop)
  - 使用 SIMD 部件（矢量处理器和图形处理器GPU）

- 例：

```
for (i=0;i<=999;i=i+1)
    x[i]=x[i]+y[i]
```

  - 在每次循环迭代中，几乎不存在重叠的机会
  - 循环的每次迭代与其它迭代之间存在着重叠

循环展开

- 例：矢量的所有分量加上同一个标量

```
for(i=999;i>=0;i=i-1)
    x[i]=x[i]+s;
```

  - 注意：每次迭代的循环体是独立的

  - 对应的 RISC V 代码

```
Loop:   fld     f0,0(x1)    // f0=array element
        fadd.d  f4,f0,f2    // add scalar in f2
        fsd     f4,0(x1)    // store result
        addi    x1,x1,-8    // decrement pointer 8 bytes
        bne     x1,x2,Loop  // branch x1≠x2
```

循环展开

- 未调度代码

```
                                        Clock cycle issued
Loop:   fld      f0,0(x1)                       1
        stall                                   2
        fadd.d   f4,f0,f2                       3
        stall                                   4
        stall                                   5
        fsd      f4,0(x1)                       6
        addi     x1,x1,-8                       7
        bne      x1,x2,Loop                     8
```

  - 指令 addi 和 bne 之间，由于向前通道(EX ⇒EX)，所以不用等待

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

循环展开

- 调度代码

```
                                        Clock cycle issued
Loop:   fld      f0,0(x1)                       1
        addi     x1,x1,-8                       2
        fadd.d   f4,f0,f2                       3
        stall                                   4
        stall                                   5
        fsd      f4,8(x1)                       6
        bne      x1,x2,Loop                     7
```

  - 将指令 addi 调度到第 2 个周期发出，替换原来的停顿
  - 每次迭代的 7 个时钟周期 = 有效指令(3 个周期) + 循环开销(4 个周期)

- 循环展开(loop unrolling)：增加有效指令的相对数量
  - 简单地多次复制循环体
  - 调整循环终止代码

循环展开

- 展开系数为 4（假设元素个数可以被 4 整除）
  - 消除不必要的指令，不重复使用任何寄存器。

```
Loop:   fld      f0,0(x1)
        fadd.d   f4,f0,f2
        fsd      f4,0(x1)       // drop addi & bne
        fld      f6,-8(x1)
        fadd.d   f8,f6,f2
        fsd      f8,-8(x1)      // drop addi & bne
        fld      f10,-16(x1)
        fadd.d   f12,f10,f2
        fsd      f12,-16(x1)    // drop addi & bne
        fld      f14,-24(x1)
        fadd.d   f16,f14,f2
        fsd      f16,-24(x1)
        addi     x1,x1,-32
        bne      x1,x2,Loop
```

  - 注意：每次迭代需要 14（指令）+ 12（停顿）时钟周期。（？）

循环展开

- Strip Mining：一种拆分循环以利用缓存技术的编译技术

- 假设：循环次数为 N，复制 k 次循环体
  - 首先执行 $N\ mod\ k$ 次原始循环体
  - 再执行 N/k 次展开循环体

循环展开与调度

- 调度+循环展开

```
Loop:   fld      f0,0(x1)
        fld      f6,-8(x1)
        fld      f10,-16(x1)
        fld      f14,-24(x1)
        fadd.d   f4,f0,f2
        fadd.d   f8,f6,f2
        fadd.d   f12,f10,f2
        fadd.d   f16,f14,f2
        fsd      f4,0(x1)
        fsd      f8,-8(x1)
        fsd      f12,-16(x1)
        fsd      f16,-24(x1)
        addi     x1,x1,-32
        bne      x1,x2,Loop
```

  - 共 14 个周期，每个循环体平均 3.5 个周期。

循环展开与调度

- 技术关键：知道何时以及如何更改指令之间的顺序
  - 这个过程必须由编译器或硬件有条不紊地完成，而不是人类。

- 步骤：
  - 通过发现循环迭代是独立的（除了循环维护代码），确定是否可以使用循环展开；
  - 使用不同的寄存器来避免不必要的约束（如，名称相关）；
  - 消除额外的测试和分支指令，并调整循环终止和迭代代码；
  - 通过观察来自不同迭代的 `load` 和 `store` 是否独立，来确定展开循环中的 `load` 和 `store` 指令是否可以互换（需要分析内存地址并发现它们访问的不是同一个地址）；
  - 调度代码，保留产生与原始代码相同结果所需的所有相关项。

循环展开与调度

- 关键需求
  - 理解一条指令是如何依赖于另一条指令的
  - 根据相关性，如何更改或重新排序指令

- 限制循环展开效果的三个因素
  - 每次展开所分摊（循环）开销的减少
  - 代码大小的限制
    - 随着代码量的增加，可能引起指令缓存不命中率的增加
  - 编译器的限制
    - 由于积极的展开和调度而造成寄存器数量的潜在不足，即：寄存器压力 (register pressure)

利用高级分支预测技术降低分支代价

- 因为需要通过分支危害和停顿来实施控制相关，所以分支指令将损害流水线性能。
- 循环展开是一种减少分支危害数目的途径
- 预测分支指令行为可以降低其对流水线性能的损害
- 随着流水线深度的增加和每个时钟周期更多指令的发出，都增加了重叠运行指令的数目，更准确地预测分支指令行为的重要性也随之增加。

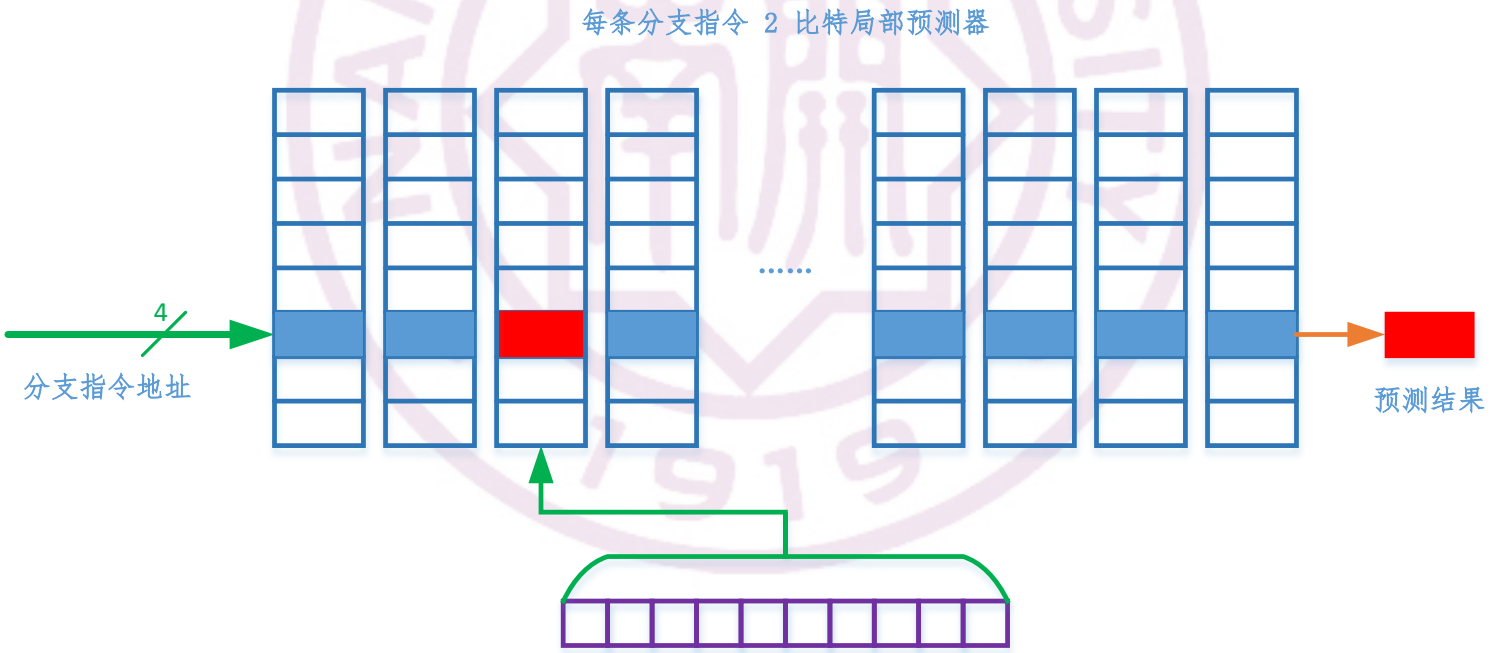
关联分支预测

- 基本 2 比特预测方法
  - 只使用单条分支指令的近期行为来预测它未来的行为
  - 对每条分支指令
    - 预测转移或不转移
    - 如果连续两次预测错误，则改变预测结果

- 关联预测方法（或两级预测方法）
  - 同时考虑其它分支指令最近的行为（相互关联）
  - (m,n)预测器：使用最近 m 条分支指令的行为来从 2^m 个分支预测器中进行选择，其中每个分支预测器都是一条分支指令的 n 位预测器。
    - 最近 m 条分支指令的全局历史记录在一个 m 位移位寄存器中，每一位记录着该分支指令是转移还是未转移。
    - 对分支预测缓存的访问由分支指令的低位地址拼接上 m 位全局历史记录得到

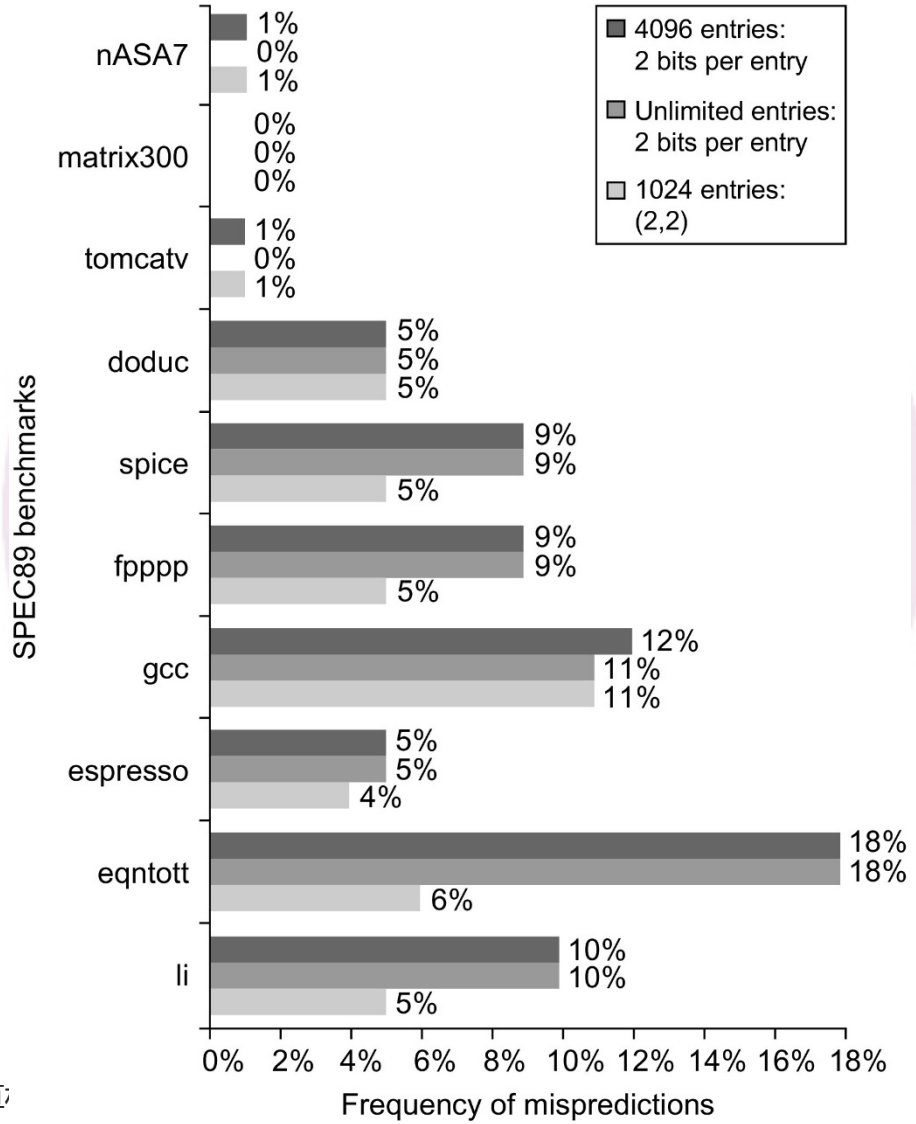
关联分支预测

- 例：$(10, 2)$ 关联预测器
  - 使用前 10 个分支指令行为从 2^{10} 个分支预测中进行选择
  - 每个预测对应单个分支指令的 2 位预测器
  - 假设：限制关联预测器空间为 $30K$
    - $2^{10} \times 2 \times EntryNumber = 30K \Rightarrow EntryNumber = 15$
    - 为了计算方便，选择使用 16 个入口，即使用低 4 位地址做为入口地址



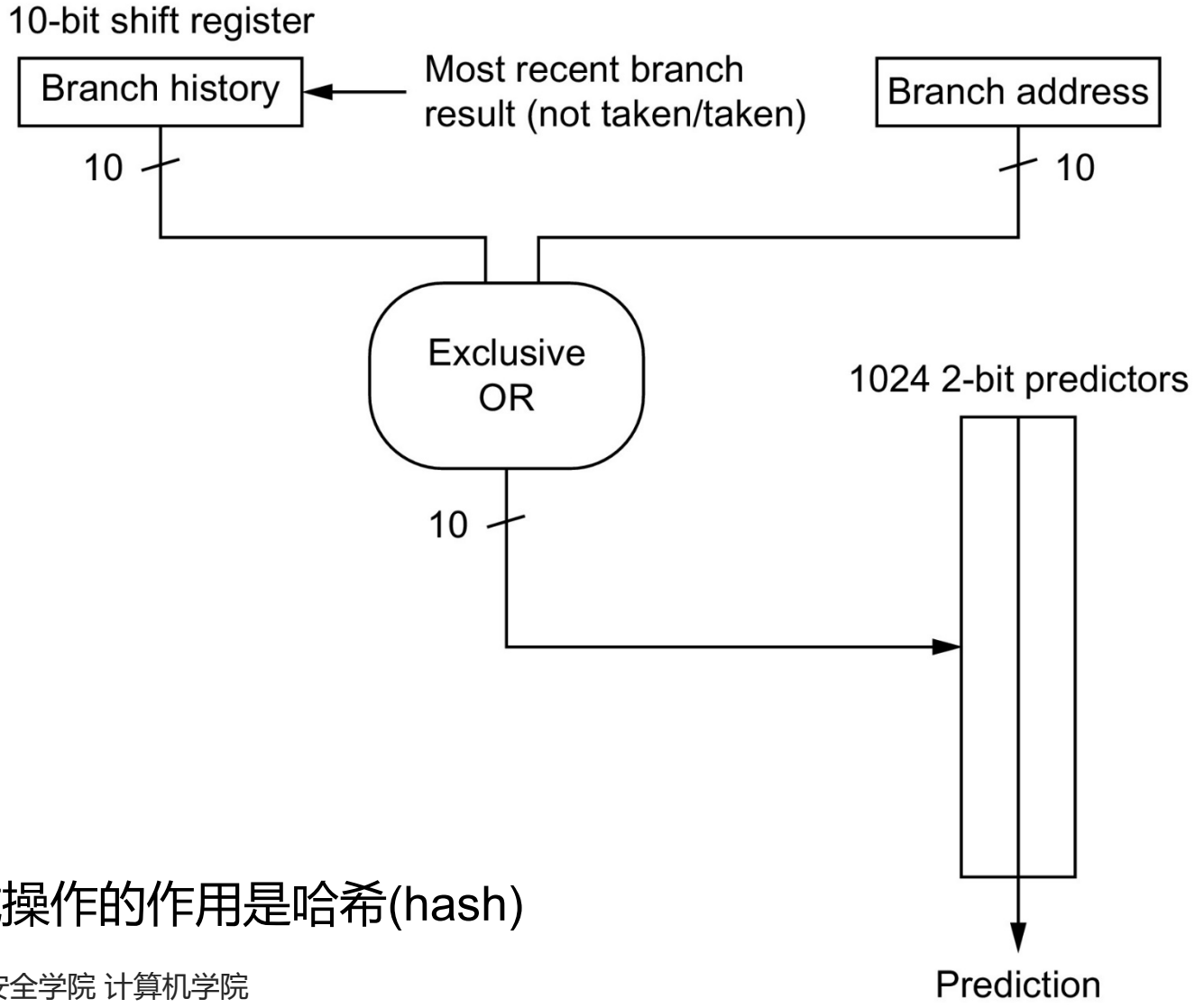
(0,2)预测器和(2,2)预测器性能比较

- 相同比特数：(0,2)预测器 4K 项，(2,2)预测器 1K 项



McFarling的 gshare 预测器

- 效果很好，常作为更加复杂预测器的比较基准。



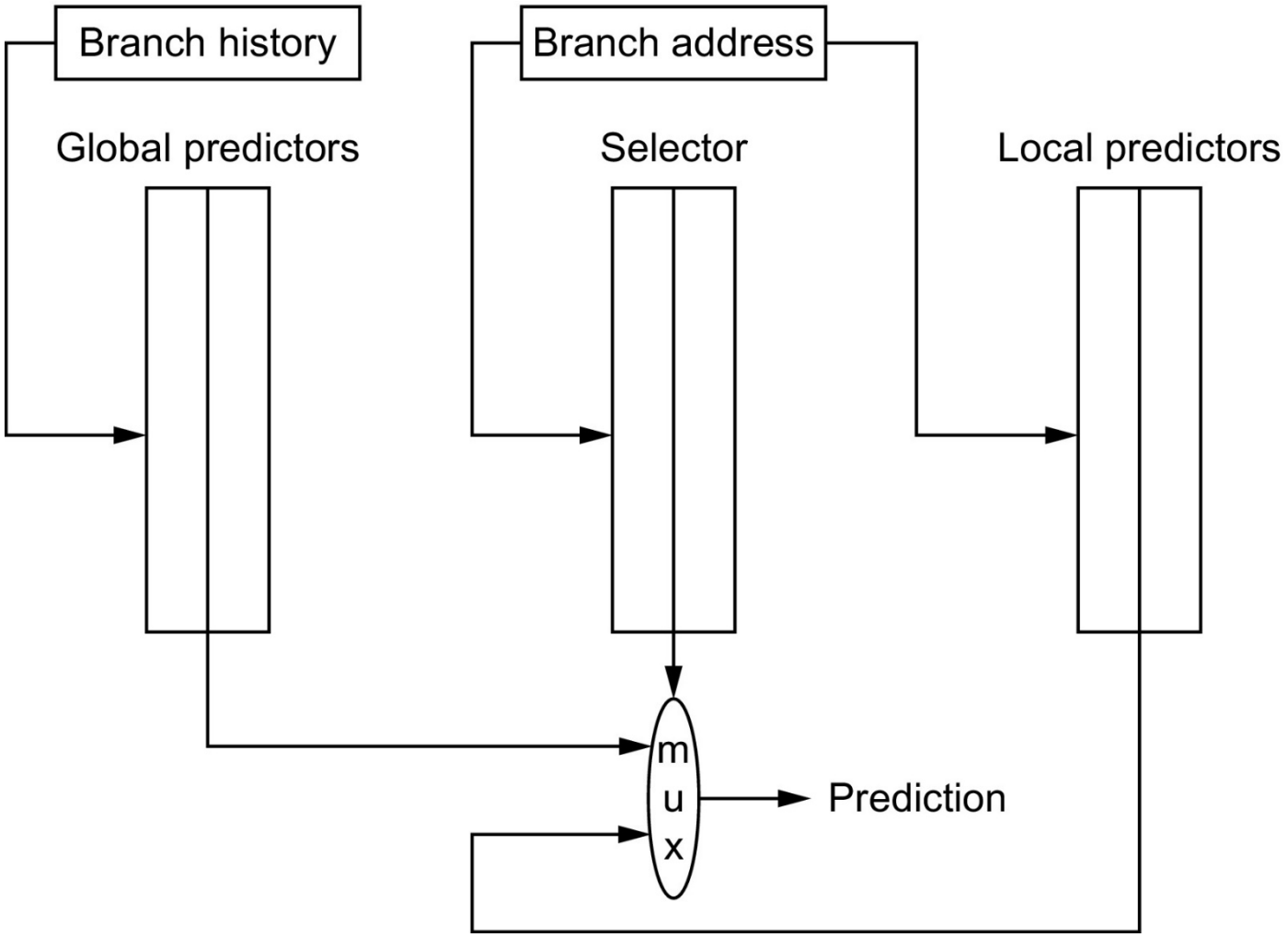
  - 异或操作的作用是哈希(hash)

锦标预测器

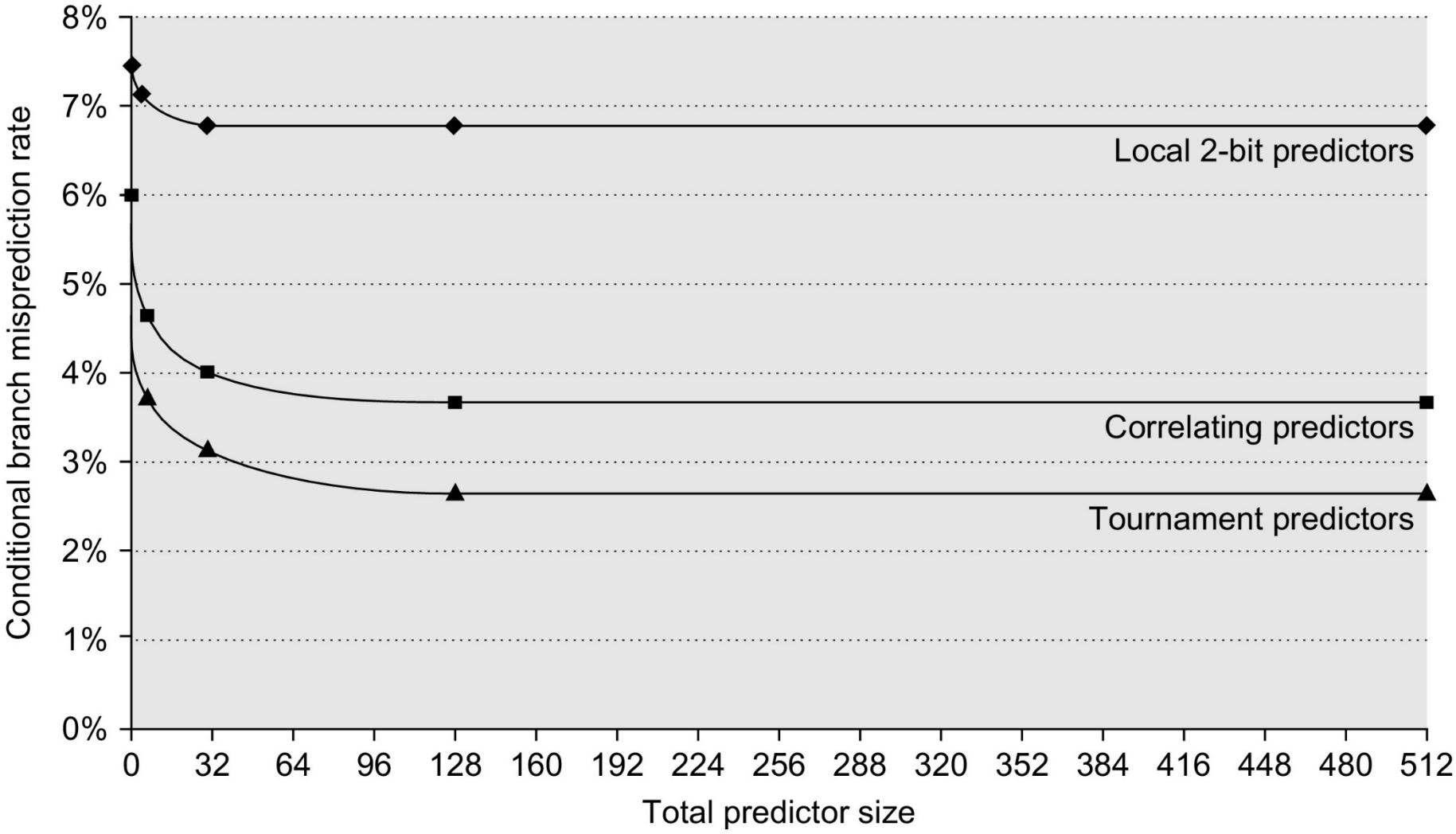
- 合金预测器(alloyed predictors)或混合预测器(hybrid predictors)：结合局部分支信息和全局分支历史的预测器
- 锦标预测器(tournament predictors)：自适应结合局部和全局预测器
  - 全局预测器(global predictor)：使用最近的分支历史来索引预测器
  - 局部预测器(local predictor)：使用分支指令地址做为索引
  - 选择器：象一个 2 比特预测器，当一行中出现两次预测错误时，对该分支地址更改首选预测器
- 问题：错误预测时需要同时改变
  - 选择器表
  - 全局预测器或局部预测器

锦标预测器

- 分支地址位数与分支历史长度相同



分支预测方法的性能


8%
7%
6%
5%
4%
3%
2%
1%
0%
Conditional branch misprediction rate
Local 2-bit predictors
Correlating predictors
Tournament predictors
0
32
64
96
128
160
192
224
256
288
320
352
384
416
448
480
512
Total predictor size

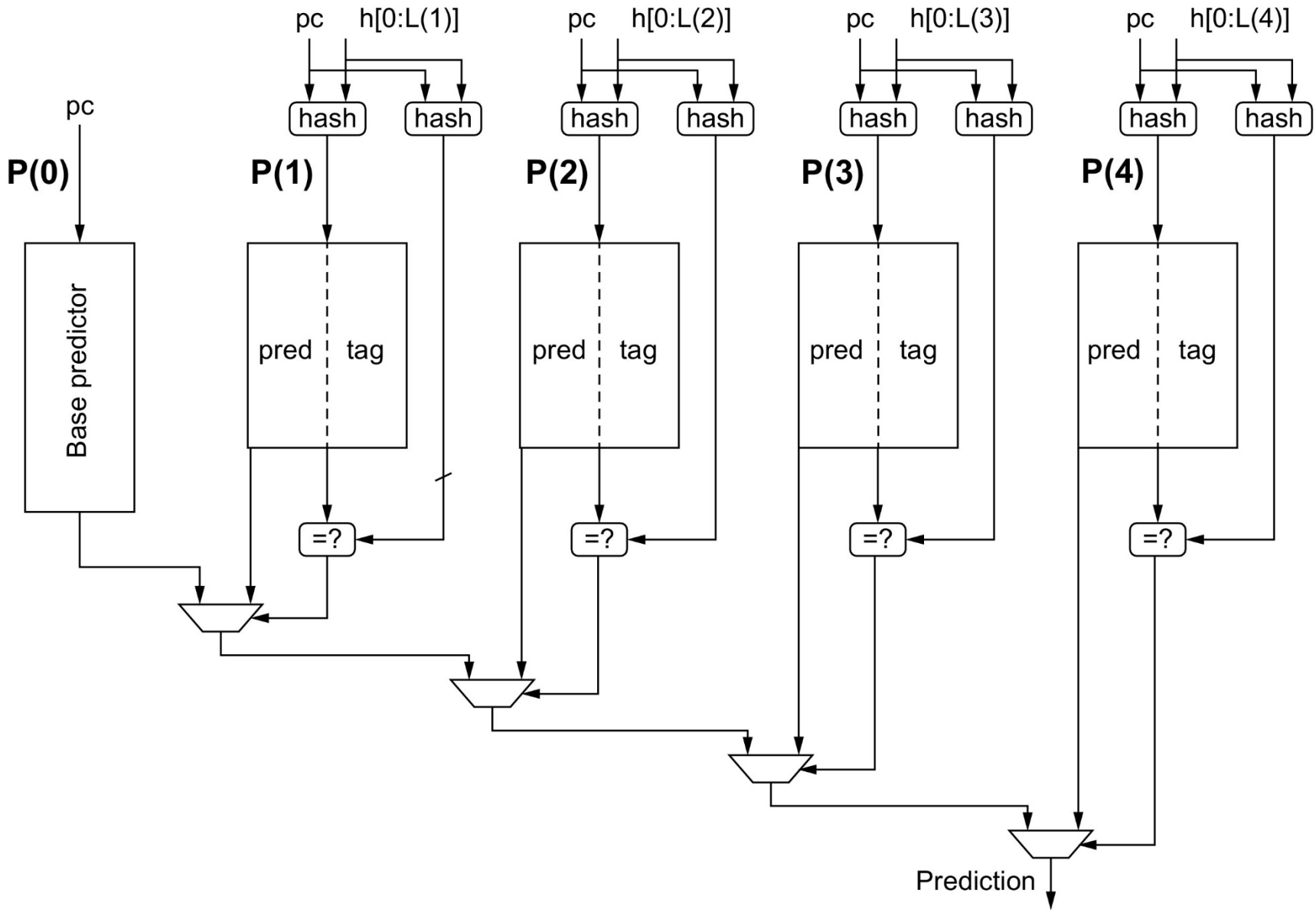
标记混合预测器

- 最好的分支预测方法是组合多个预测器
  - 多个预测器：跟踪预测是否可能与当前分支相关
  - 问题：这意味着巨大的数据表
- 一类重要的预测器基于统计压缩算法 PPM
  - PPM (Prediction by Partial Matching)：部分匹配预测法，基于历史信息预测未来行为
- 标记混合预测器(Tagged Hybrid Predictors)
  - 一类基于 PPM 思路的分支预测器
  - 使用一组使用不同长度历史索引的全局预测器

例：五分量标记混合预测器

- 五个预测表：$P(0), P(1), \ldots, P(4)$
  - 3 比特预测器（三次预测错误后改变）
- $P(i)$ 由 PC 值与最近 i 条分支指令历史的哈希值来索引
  - 历史信息保存在移位寄存器 h 中，与 gshare 相似
  - 较长的历史记录可能导致哈希冲突的概率增加，因此使用历史记录越来越短的多个数据表
- 标记：一般采用 $4 \sim 8$ 比特，因为不需要 100% 的匹配
- 只有当标记与哈希值相匹配时，才会使用 $P(1) \sim P(4)$ 的预测值
- 预测结果：与标记相匹配且具有最长分支历史的预测器
  - 当 $P(1) \sim P(4)$ 均不匹配时，$P(0)$ 做为缺省预测结果

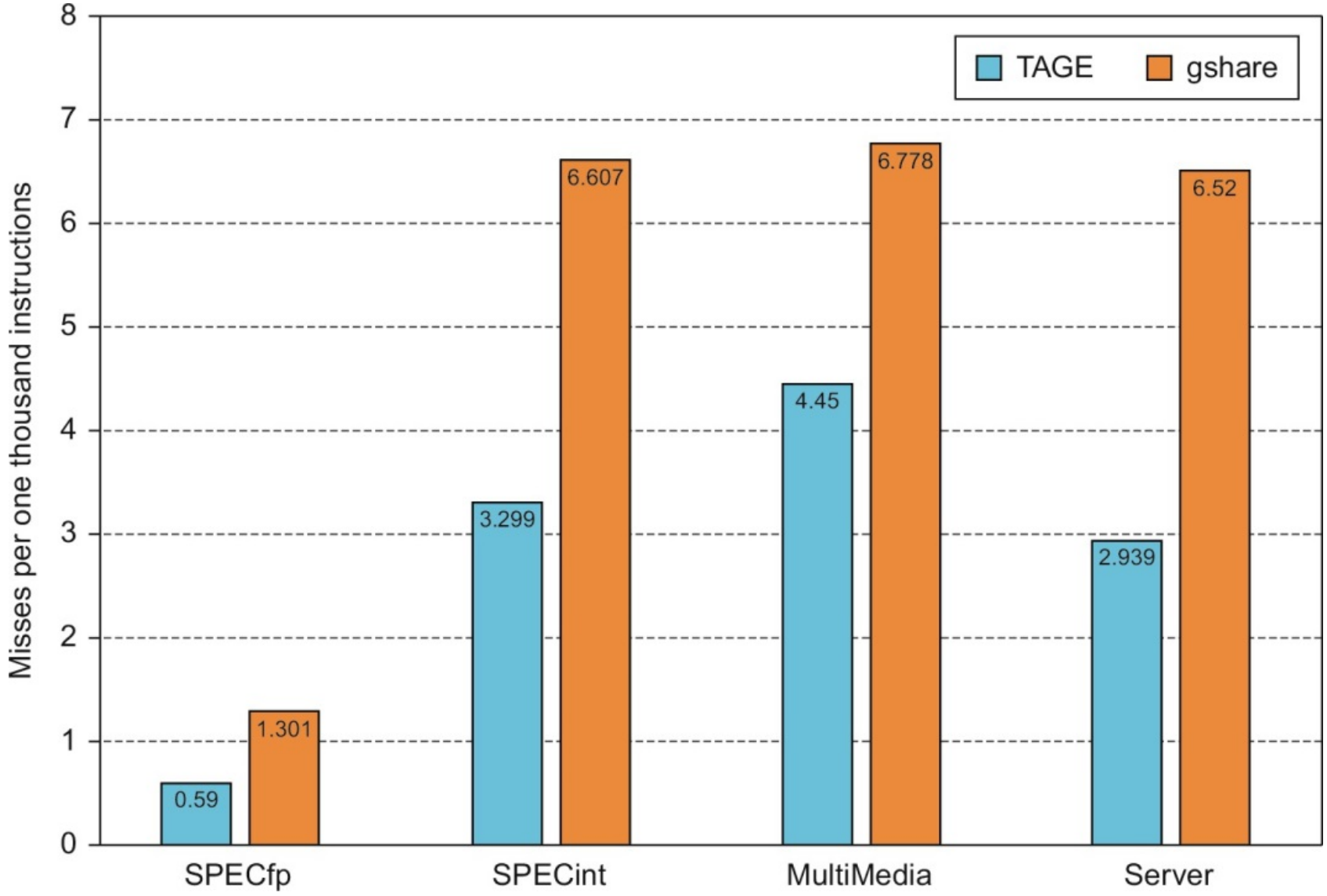
例：五分量标记混合预测器



预测器的初始化问题

- 随机初始化
  - 需要相当长的执行时间来填满有用的预测信息
- 设置有效位
  - 表示预测器对应条目处于“已设置”状态还是“未使用”状态
  - 可以使用非随机化方法初始化预测条目，如“期望”该指令是否转移
- 根据分支方向设置初始预测
  - 向前的分支指令初始化为“未转移”
  - 向后的分支指令（可能是循环分支）初始化为“转移”

标记混合预测器与 gshare 性能比较



Intel Core i7 分支预测器的演变

- Intel Core i7 处理器从 2008 年到 2016 年经历了六代。
  - 2008 年：Core i7 920
  - 2016 年：Core i7 6700
- 观察：深度流水线与单周期多指令发出相结合，导致 i7 处理器同时运行的指令最多达到 256 条，通常至少 30 条
- 结果：分支预测成为关键，是 Intel 持续努力改进的领域，并对相关细节高度保密。

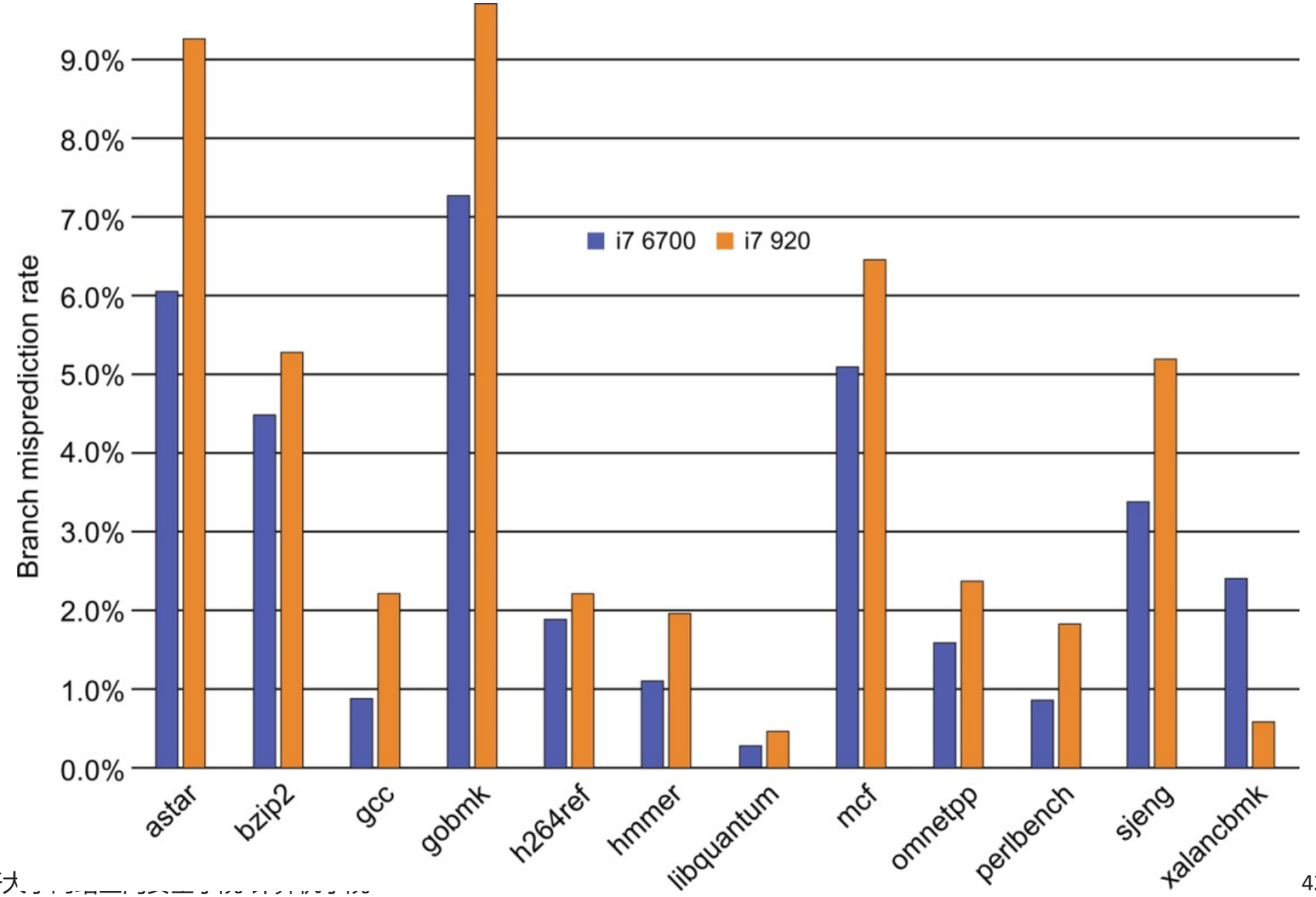
Core i7 920 分支预测器

- 使用两级预测器
  - 较小的一级预测器：满足每个时钟周期预测一条分支指令的周期约束
  - 较大的二级预测器：做为备用
- 每个预测器由三个不同的预测器组成
  - 简单的 2 比特预测器（如锦标预测器）
  - 全局历史预测器
  - 循环出口预测器：使用计数器来预测被检测为循环分支的迭代次数
- 通过跟踪每个预测器的准确率来选择最佳预测器
- 预测机制
  - 多层主预测器
  - 独立单元预测间接分支指令的目标地址
  - 堆栈预测返回地址

Core i7 6700 分支预测器

- 相关信息知之甚少
- 充分理由相信：正在使用标记混合预测器
  - 结合了早期 i7 中所有三个二级预测器的功能
- 不同历史长度的标记混合预测器包含循环出口预测器，局部和全局历史预测器，仍然使用单独的返回地址预测器。
- 指令的推测执行带来对预测器评估的挑战
  - 分支指令错误预测结果将导致另一条分支指令的预取和错误预测
- 测试结果表明：分支指令的错误预测结果导致无效的推测执行，浪费了工作时间。
  - 由下页插图与 35 页插图比较得出

Integer SPECCPU2006 预测错误率



动态调度

- 静态调度(static scheduling)
  - 常态：取入指令 ⇒ 发出指令
  - 非常态：
    - 读取指令与正在执行指令之间存在数据相关，且旁路或向前通道不能解决。
    - 危害检测硬件从使用结果指令开始停顿，直到相关被清除再取入或发出指令。
- 动态调度(dynamic scheduling)：在维持数据流和中断行为的同时，硬件重新排列指令执行顺序以减少停顿。
- 动态调度避免停顿，静态调度最小化停顿。
  - 当然，编译流水线调度也可以用于运行在动态调度流水线的代码上。

动态调度的特点

- 代码编译不需要考虑微体系结构
  - 在一条流水线下编译的代码可以在不同流水线上高效运行
- 能够处理编译时不了解相关性的一些情况
  - 内存访问或数据相关分支
  - 使用动态链接或调度的现代编程环境
- 最重要的一点：允许处理器容忍不可预测的延迟
  - 如：缓存不命中，可以通过执行其它指令等待缓存不命中问题的解决
- 代价是大幅度增加硬件的复杂性

动态调度

- 简单流水技术的主要局限：指令按序发出和执行
  - 如果一条指令被停顿，其后指令无论如何都不能继续，由此造成多个功能单元的闲置。
- 例：

```
fdiv.d   f0,f2,f4
fadd.d   f10,f0,f8
fsub.d   f12,f8,f14
```

  - fadd.d 与 fdiv.d 数据相关，故必须停顿；
  - fsub.d 与它们无关，但由于保持程序顺序，故也必须停顿。

- 动态调度意味着
  - 乱序执行(Out-of-order execution)
  - 乱序完成(Out-of-order completion)

动态调度

- 传统五段流水线的 ID 流水段
  - 检测结构危害和数据危害
  - 发出没有危害的指令
- 拆分 ID 流水段为两部分
  - 发出(issue)：指令译码，检测结构危害
  - 读操作数(read operands)：等待数据危害的消失，读取操作数
- 特点
  - 保持按序发出(in-order issue)
  - 能够乱序执行(out-of-order execution)，暗示乱序完成(out-of-order completion)
- 乱序执行：一旦指令的操作数有效则马上开始执行。

乱序执行

- 例：

```
fdiv.d   f0,f2,f4
fmul.d   f6,f0,f8
fadd.d   f0,f10,f14
```

  - fmul.d 与 fadd.d 之间的反相关 ➔ WAR 危害
  - fadd.d 与 fdiv.d 之间的输出相关 ➔ WAW 危害

- 乱序完成可能导致不精确中断(imprecise exception)
- 不精确中断：当发生中断时的处理器状态看起来不完全像指令以严格程序次序顺序执行的结果。
  - 流水线中已经完成的指令在程序次序上比引起中断的指令晚
  - 流水线中还未完成的指令在程序次序上比引起中断的指令早
- 问题：不精确中断使得中断后难于重新启动

诚信 创新 实践