高等计算机体系结构
第十三讲: 多处理器与多处理

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

1

阅读: 多处理

- 必读
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
  - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979
  - 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5.8节 (第四版)
- 推荐
  - Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966
  - Hill, Jouppi, Sohi, "Multiprocessors and Multicomputers," pp. 551-560 in Readings in Computer Architecture.
  - Hill, Jouppi, Sohi, "Dataflow and Multithreading," pp. 309-314 in Readings in Computer Architecture.
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

2

多处理器和多处理中的难题

3

回顾: Flynn的分类

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: 单指令操作单个数据元素
- SIMD: 单指令操作多个数据元素
  - 阵列处理机
  - 向量处理器
- MISD: 多指令操作单个数据元素
  - 最接近的形式: 脉动阵列处理器, 流处理器
- MIMD: 多指令操作多个数据元素 (多指令流)
  - 多处理器
  - 多线程处理器

4

为什么要有并行机?

- 并行性: 同时做多件"事情"
- "事情": 指令, 操作, 任务

- 主要目标
  - 改善性能 (执行时间或任务吞吐量)
    - 程序的执行时间由Amdahl定律控制

- 其他目标
  - 降低功耗
    - 4N个单元工作在F/4频率下的功耗低于N个单元工作在F频率下
    - 为什么?
  - 提高成本效率和可扩展性, 降低复杂性
    - 很难设计出一个复杂单元能够有N个简单单元一样的执行效果
  - 提高可靠性: 在空间上冗余执行

并行的类型

- 指令级并行
  - 一个指令流中的不同指令可以并行执行
  - 流水线, 乱序执行, 投机执行, VLIW
  - 数据流

- 数据并行
  - 数据的不同片段可以被并行的操作
  - SIMD: 向量处理, 阵列处理
  - 脉动阵列, 流处理器

- 任务级并行
  - 不同的"任务/线程"可以被并行执行
  - 多线程
  - 多处理 (多核)

任务级并行: 生成任务

- 将一个问题分割成多个相关的任务(线程)
  - 显式: 并行编程
    - 当问题中的任务能够很自然地划分时
      - Web/数据库请求
    - 当任务的边界不那么清晰时会比较困难

  - 透明/隐式: 线程级投机
    - 投机地分割单个线程

- 同时运行多个独立的任务(进程)
  - 当有多个进程时
    - 批处理的仿真, 不同用户的进程, 云计算的工作负载
  - 不能提升单个任务的性能

多处理基础

多处理器的类型

- 松耦合多处理器
  - 没有共享的全局存储地址空间
  - 多机网络
    - 基于网络的多处理器
  - 通常通过消息传递编程
    - 通过显式调用 (send, receive) 通信
- 紧耦合多处理器
  - 共享全局存储地址空间
  - 传统的多处理: 对称多处理器(SMP)
    - 现在的多核处理器, 多线程处理器
  - 编程模型与单处理器类似(多任务单处理器)，除了
    - 操作共享数据需要同步

9

9

紧耦合多处理器的主要难点

- 共享存储同步
  - 锁, 原子操作
- Cache 一致性
- 访存操作的序
  - 程序员希望硬件提供什么?
- 资源共享, 竞争和分区
- 通信: 互连网络
- 负载不均衡

10

10

基于硬件的多线程

- 粗粒度多线程
  - 基于定量
  - 基于事件
- 细粒度多线程
  - 每周期
  - Thornton, "CDC 6600: Design of a Computer," 1970.
  - Burton Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
- 同时多线程
  - 能够同时由多个线程分发指令
  - 有效提升执行单元的利用率

11

11

并行加速比

- $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$
- 假设每个操作占用1个周期, 没有通信开销, 每个操作可以在不同的处理器上执行
- 用单个处理器执行有多快?
  - 假设没有流水线或者对指令的并发执行
- 用3个处理器有多快?
  - 加速比(speedup)

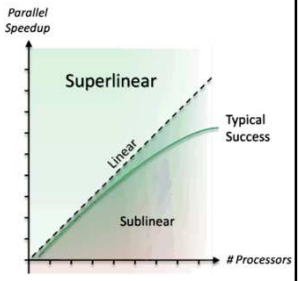
Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.

12

12

超线性加速比

- 当使用P个处理单元时能否获得大于P的加速比?
  - Cache 的影响
  - 工作集的影响
- 两种情况下:
  - 对比不公平
  - 访存的影响

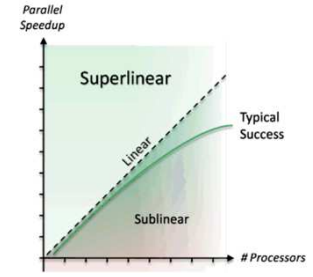


利用率, 冗余度和效率

- 常用的指标
  - 假设所有P个处理器都满负荷参与并行计算
- 利用率: 有多少处理能力被使用
  - U = (并行操作的数量) / (处理器 x 时间)
- 冗余度: 并行处理时做了多少额外的工作
  - R = (并行操作的数量) / (用最佳的单处理器算法执行的操作数)
  - R总是≥1
- 效率: 用了多少资源占能够获得多少资源的比例
  - E = (使用1个处理器花费的时间) / (处理器 x 使用P个处理器花费的时间)
  - E = U/R

真实的加速比

- 为什么真实的加速比是这样的?



$$\tau_p = \alpha \cdot \frac{\tau_1}{p} + (1\text{-}\alpha) \cdot \tau_1$$

单处理器程序中可以并行化的部分　　不可以并行化的部分

Amdahl定律

$$\text{加速比}_{P\text{个处理器}} = \frac{\tau_1}{\tau_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}$$

$$\text{加速比}_{P\to\infty} = \frac{1}{1-\alpha}$$

并行加速比的瓶颈

Amdahl定律的内涵:
1、当α<1时，增加越来越多的处理器，得到的收益(加速比)越来越少;
2、收益(加速比)不大，除非α≈1

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

Amdahl定律的启示

- Amdahl定律的另一种表示
  - f: 程序可并行化的比例
  - N: 处理器数量

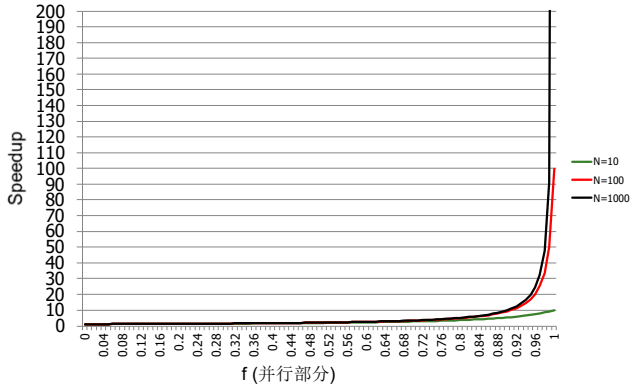
$$\text{加速比} = \frac{1}{1 - f + \frac{f}{N}}$$

  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- 最大化加速比受限于串行部分: 串行瓶颈
- 并行部分通常也不是完美的并行
  - 同步开销 (比如, 更新共享的数据)
  - 负载不均衡开销 (并行化不完美)
  - 资源共享开销 (N个处理器之间的竞争)

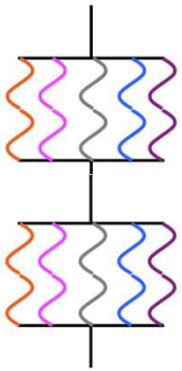
17

串行瓶颈



18

为什么串行是瓶颈?



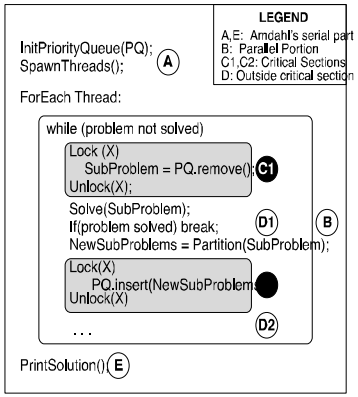
- 并行的机器有串行瓶颈
- 主要原因: 有不能并行化的数据操作(比如, 不能并行化的循环)

```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

- 数据准备是单线程的，而任务生成是并行的(通常任务本身又是串行的)

19

串行瓶颈的例子



20

并行部分的瓶颈

- 同步: 对共享数据的操作不能并行
  - 锁, 同步互斥, barrier同步
  - 通信: 任务之间可能需要互相的数据
  - 竞争共享数据时会造成线程串行
- 负载不均衡: 并行的任务可能有不同的长度
  - 由于并行化不理想或者微体系结构的影响
  - 在并行部分降低加速比
- 资源竞争: 并行任务会共享硬件资源, 互相延迟
  - 为所有资源设计冗余 (比如内存) 成本太高
  - 每个任务单独运行时并没有额外的延迟产生

并行编程的困难

- 如果存在天然的并行性不就太难
  - “高度并行” 的应用
  - 多媒体, 物理模拟, 图形图像处理
  - 大型 web 服务器, 数据库?
- 困难在于
  - 让并行程序正确运行
  - 存在瓶颈时优化性能
- **并行计算机体系结构**主要是关于
  - 如何设计计算机以克服串行和并行瓶颈，获得高性能和高效率
  - 使程序员更容易开发正确并且高性能的并行程序

多处理中访存的序

操作的序

- 操作: A, B, C, D
  - 硬件该按照什么顺序执行(报告结果)这些操作?
- 程序员和微架构之间的约定
  - 由ISA确定
- 维护一个“期望的” 序能够使程序员的人生更加美好
  - 调试; 状态恢复和处理异常
- 维护一个“期望的”序通常使硬件设计者的人生更加……
  - 尤其是需要设计高性能处理器: 乱序执行中的load-store队列

单处理器中访存的序

- 由冯诺依曼模型指定
- 顺序序
  - 硬件执行load和store操作，按照程序串行执行所指定的序
- 乱序执行不改变语义
  - 硬件提交 (向软件报告结果) load和store操作，按照程序串行执行所指定的序
- 优点: 1) 执行过程中的体系结构状态是精确的 2) 程序每次运行的体系结构状态是一致的 → 容易调试程序
- 缺点: 保持序会增加开销, 降低性能

25

25

数据流处理器中访存的序

- 当操作数准备好就可以执行访存操作
- 序由数据相关性指定
- 如果两个操作没有相关性，他们可以按照任何序执行和提交
- 优点: 很多的并行性 → 高性能
- 缺点: 同一个程序每次执行可能会出现不同的序 → 很难调试

26

26

MIMD 处理器中访存的序

- 每个处理器的访存操作按照在该处理器上运行的"线程"的顺序执行序(假设每个处理器遵循冯诺依曼模型)
- 多处理器并发执行访存操作
- 存储器看到来自所有处理器的访问的序是什么样的?
  - 换句话说, 跨不同处理器的操作的序是什么样的?

27

27

为什么这件事很重要?

- 容易调试
  - 同一个程序在不同时间的执行拥有同样的执行序是很有用的
- 正确性
  - 如果从不同处理器看到的访存操作序不同，是否会导致错误的操作呢?
- 性能和开销
  - 在实现提升性能相关技术(比如乱序执行、cache等)的同时执行严格的"顺序执行序"，对硬件设计者是一个挑战

28

28

保护共享数据

- 线程不能够并发地更新共享数据
  - 为了正确性
- 对共享数据的访问被封装在*临界区*或者通过*同步机制(锁, 信号量, 条件变量)*
- 只能有一个线程在某个确定的时间执行临界区
  - 同步互斥原则
- 多处理器需要提供同步原语的*正确*执行以确保程序员能够保护共享数据

29

29

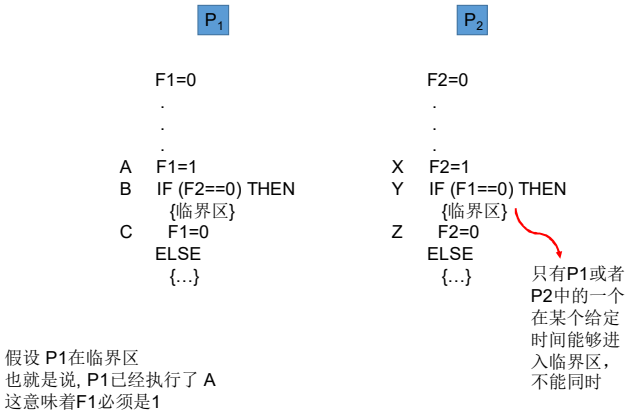
支持同步互斥

- 程序员需要确认同步互斥是否正确的实现
  - 我们假设是这样的
  - 但是, 并行编程的正确性是一个重要的主题
  - 阅读: Dijkstra, "Cooperating Sequential Processes," 1965.
    - Dekker的同步互斥算法
- 程序员依赖硬件原语支持正确的同步
  - 如果硬件原语不正确(或不确定), 程序员的人生。。。
  - 如果硬件原语是正确的, 但是不容易使用, 程序员的人生还是。。。

30

30

保护共享数据



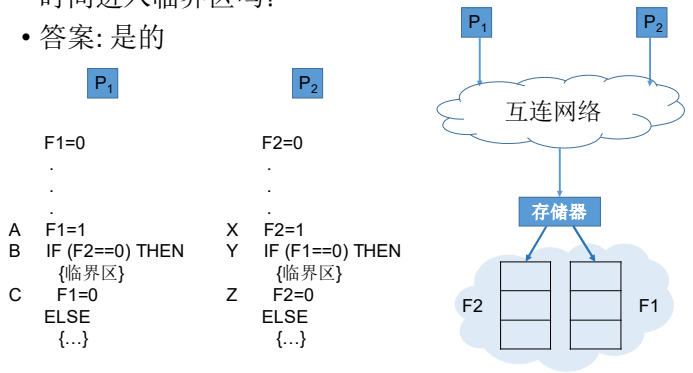
假设 P1在临界区
也就是说, P1已经执行了 A
这意味着F1必须是1
这意味着P2不能进入临界区

31

31

问题

- 两个处理器(遵循冯诺依曼模型)有可能在相同的某个时间进入临界区吗?
- 答案: 是的



两个处理器看到的是访存操作不同的序

32

32

该如何解决这个问题?

- 思路: 顺序一致性(Sequential consistency)

- 所有处理器看到相同的访存操作的序
- 即, 所有的访存操作按照一个对所有处理器都一致的序执行 (称为全局总序)

- 假设: 在全局序中，每个处理器自己的操作按顺序序执行

33

33

顺序一致性

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- 一个多处理器系统是顺序一致的，如果:
- 所有操作的结果都是相同的，就好像所有处理器的操作都按照某种顺序序执行

并且

- 每个处理器操作所呈现出的序是按照程序所指定的序

- 这是一个访存执行序模型，或者说是一个内存模型
- 由ISA指明

34

34

程序员抽象

- 内存像一个"开关"，某个时刻处理来自任何处理器的一个load或者store
- 所有处理器同时看到当前被响应的load或者store
- 每个处理器的操作按照程序序被响应

35

35

顺序一致性操作序

- 可能正确的全局序(都是正确的):

- A B X Y
- A X B Y
- A X Y B
- X A B Y
- X A Y B
- X Y A B

- 什么序 (交叉存取)会被观察到取决于具体实现和动态的延迟

36

36

顺序一致性的结果

- 推论

1. 在同一次的执行中, 所有处理器会看到同样的访存操作全局序
   → 没有正确性问题

2. 对于不同次的执行, 可能观察到不同的全局序 (每一个都是顺序一致的)
   → 调试仍然困难 (不同的执行序会变化)

37

37

顺序一致性的问题?

- 很漂亮的编程抽象, 但是有两个问题:
  - 对序的要求过于保守
  - 限制了那些性能改善技术的激进性

- 全局序的要求是否太强了?
  - 是否需要一个对所有处理器的所有操作的全局序?
  - 仅针对所有store的全局序怎么样?
    - 完全store序访存模型; 唯一store序访存模型
  - 仅仅在同步的边界处执行全局序怎么样?
    - 宽松访存模型
    - 请求-释放一致性模型

38

38

顺序一致性的问题?

- 性能提升技术会使顺序一致性实现变得困难

- 乱序执行
  - load之间乱序，load和独立的store之间乱序

- 高速缓存
  - 一个内存位置会出现在多个地方
  - 妨碍store的结果被其它处理器看到

39

39

弱的内存一致性

- 操作的序十分重要，尤其是当序会影响对共享数据的操作时 → 即, 当处理器需要同步对某个"程序区域"的执行时

- 弱一致性
  - 思路: 程序员指明在访存操作中不需要按序的程序区域
  - "内存barrier" 指令描述了这些区域
    - 所有在barrier障之前的内存操作必须在barrier被执行前完成
    - 所有在barrier之后的内存操作必须等待barrier执行完毕才能执行
    - barrier按程序序执行
  - 所有同步操作的表现就像barrier

40

40

Tradeoff: 更弱的一致性

- 优点
  - 不需要保证访存操作非常严格的序
    → 使得一些性能提升技术的硬件实现更简单
    → 可以比严格的序性能更高
- 缺点
  - 程序员(或者软件)的负担更重(需要保证“barrier” 正确)
- 程序员-微架构折衷的又一个例子

41

41

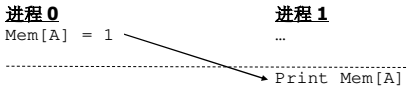
Cache一致性 (Cache Coherence)

42

42

共享存储模型

- 很多并行程序通过*共享存储*通信
- 进程 0 写某个地址，接着进程 1 读
  - 两个进程之间通信



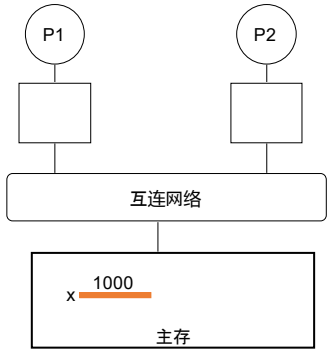
- 每一个读操作都能够收到任何人最后一次写的值
  - 这需要同步 (最后一次写是什么意思?)
- 如果Mem[A]在cache中(两个进程端都有)会怎么样?

43

43

Cache 一致性

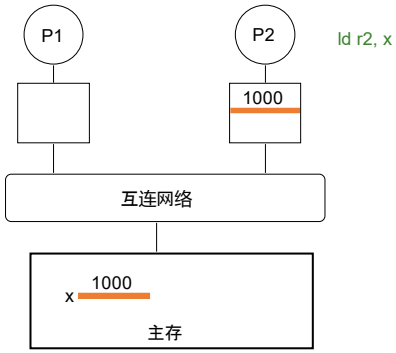
- 基本问题: 如果多处理器cache同一个块, 如何保证它们看到的是一致的状态?



44

44

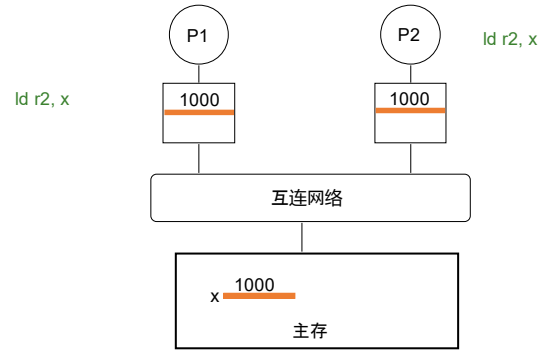
Cache一致性问题



45

45

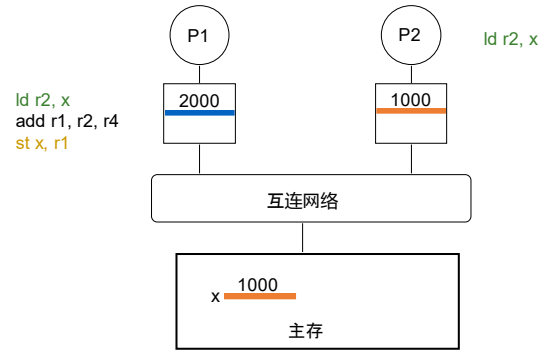
Cache一致性问题



46

46

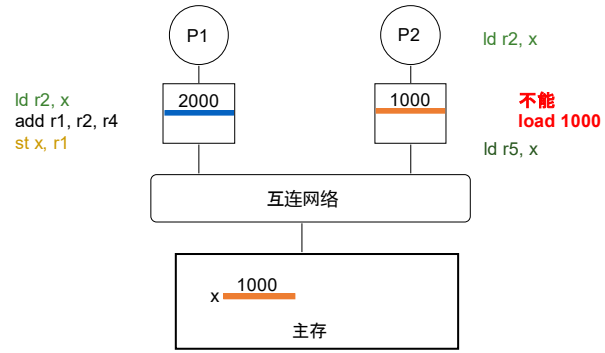
Cache一致性问题



47

47

Cache一致性问题



48

48

Cache 一致性: 该由谁负责?

- 软件
  - Cache对软件不可见，程序员还能够保证一致性吗?
  - 如果ISA提供清空cache的指令会怎么样?
    - FLUSH-LOCAL A: 清空/置无效处理器本地cache中包含地址A的一个cache块
    - FLUSH-GLOBAL A: 清空/置无效所有处理器cache中包含地址A的一个cache块
    - FLUSH-CACHE X: 清空/置无效cache X中的所有块
- 硬件
  - 简化软件的工作
  - 一个思路: 当一个处理器写某个块，将该块的所有其它拷贝全部置为无效

Cache 一致性解决方案

- 完全不依靠硬件的一致性
  - 保持cache一致性是软件的责任
  - \+ 让微架构更简单
  - -- 使程序员的工作更困难
    - 需要考虑硬件cache以保持程序的正确性?
  - -- 软件中保证一致性会带来额外开销
- 所有的处理器共享所有的cache
  - \+ 不需要一致性
  - -- 共享的cache成为瓶颈
  - -- 这种方案下极难设计即具备低延迟cache又有可扩展性的系统

保持一致性

- 需要保证所有处理器看到相同存储位置的值的一致性(一致更新)
- P0向位置A的写入应该被P1 (最终)看到, 所有对A的写应该按照某种序呈现出来
- 一致性需要提供:
  - **写的传播**: 保证更新被传播出去
  - **写的序列化**: 为所有处理器提供一致的全局序
- 需要一个全局的序列化点对写排序

硬件 Cache 一致性

- 基本思路:
  - 一个处理器/cache向所有其它处理器广播它对某个内存位置的写/更新
  - 另一个拥有这个位置的数据的cache要么更新要么置无效它的本地拷贝

一致性: 更新vs. 置为无效

- 如何 *安全的更新有多份副本的数据*?
  - 选择 1 (更新): 向所有拷贝推送一个更新
  - 选择 2 (置无效): 确保只有一个有效拷贝 (本地的), 更新它

- **读数据时:**
  - 如果本地拷贝无效, 发出请求
  - (如果另一个节点有有效拷贝, 该节点返回, 否则由内存返回)

53

53

一致性: 更新vs. 置为无效

- **写数据时:**
  - 按照之前的方法读一个块到cache

更新协议:

- 写一个块, 同时广播写的数据给该数据的共享者
- (其它节点上如果有该数据在cache中则更新cache)

置无效协议:

- 写一个块, 同时广播需要置为无效的地址给数据共享者
- (其它节点对cache块作相应处理)

54

54

更新 vs. 置无效的Tradeoffs

- 目的是什么?
  - 写的频率和共享行为都是至关重要的
- **更新**
  - \+ 如果共享者集合是常数并且更新操作不频繁，可以避免被置无效数据重新获取的开销 (广播更新模式)
  - – 如果其它核重写的数据并没有被读取, 更新就是无用的
  - – 写直达cache策略 ➔ 总线成为瓶颈
- **置无效**
  - \+ 置无效广播后, 处理器核对数据有独占访问权
  - \+ 只有在每个写之后会持续读的核会保有一份拷贝
  - – **如果写竞争度很高**, 会导致ping-pong 效应(快速的互相置无效/重取)

55

55

两种cache一致性方法

- 如何确保合适的cache被更新?

- **监听总线(Snoopy Bus)** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - 基于总线, 所有请求在单点序列化
  - 处理器观察其它处理器的动作
    - 比如: P1 在总线上发出对A的 “排他读”请求, P0 看到后将自己的A的拷贝置为无效

- **目录(Directory)** [Censier & Feautrier, IEEE ToC 1978]
  - *每个块*单点序列化, 序列化点分布在各节点
  - 处理器生成对块的显式请求
  - 目录追踪每个块的所有权 (共享者集合)
  - 目录协调置无效操作
    - 比如: P1 向目录请求排他的拷贝, 目录要求 P0 置无效相关数据, 等待应答, 然后向 P1 响应请求

56

56

基于目录的cache一致性

- 思路: 维护一个逻辑上的中心目录并保持跟踪每个cache块所在的位置, Cache查询这个目录以确保一致性
- 例子:
  - 对内存中的每个cache块, 在目录中存储 P+1 位
    - 每个cache使用1位, 显示这个块是否在那个cache中
    - 独占位: 显示某个cache拥有这个块的唯一拷贝,可以不通知其它cache而更新该块
  - 当读一个块时: 置目录中该块的对应cache位并向相应cache提供数据
  - 当写一个块时: 对所有拥有该块的其它cache执行置无效操作, 并重置它们在目录中的相应位
  - 为每一个cache中的每个块关联一个独占位

57

57

扩展目录的一些问题

- 目录该有多大?
- 如何减小目录的访问延迟?
- 如何将系统扩展到上千个节点?

58

58

监听Cache一致性

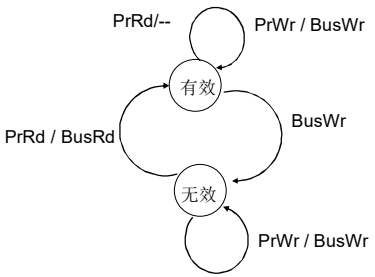
- 思路:
  - 所有cache "监听" 所有其它cache的读/写请求, 并保持cache块的一致性
  - 每个cache块在每个cache的标签存储中关联"一致性元数据"
- 当所有cache共享一个公共总线时很容易实现
  - 每个cache在总线上广播它的读/写操作
  - 对于小规模的多处理器很有效
  - 如果是1000个节点的多处理器会怎么样?

59

59

一个简单的监听cache一致性协议

- Cache "监听" (观察) 彼此的写/读操作, 如果一个处理器写一个块, 所有其它的处理器将自己cache中的相同块置为无效
- 一个简单的协议:



- 写直达, 写不分配cache
- 操作: PrRd, PrWr, BusRd, BusWr

60

60

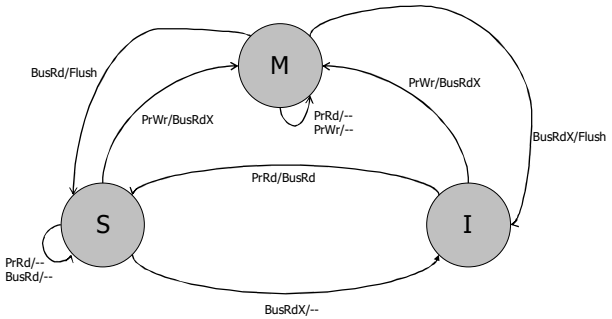
更复杂的协议: MSI

- 将每个块的单个有效位扩展为3个状态:
  - **M**(odified): cache行是唯一拷贝并且被修改过(dirty)
  - **S**(hared): cache行是一系列拷贝中的一个
  - **I**(nvalid): 无效
- 读缺失导致总线上一个*读*请求, 状态迁移到**S**
- 写缺失导致一个*排他读*请求, 状态迁移到**M**
- 当一个处理器监听到从另一个处理器发出的*排他读*请求, 它必须置无效自己的拷贝(如果有的话)

61

61

MSI 状态机



观察到的事件/操作

[Culler/Singh96]

62

62

MSI协议的问题

- 一开始任意一个块都不在cache中
- 问题: 当读一个块时, 这个块立即变为S状态，即使它可能只是cache中仅有的一个拷贝
- 为什么这是一个问题?
  - 设想一下，读这个块的cache要在某个时刻写它
  - 即使它拥有的是唯一拷贝，也需要向总线广播"置无效"!
  - 如果cache知道系统中只有它有这份拷贝, 它在写块的时候无需通知其它cache → 省去不必要的广播

63

63

解决方案: MESI

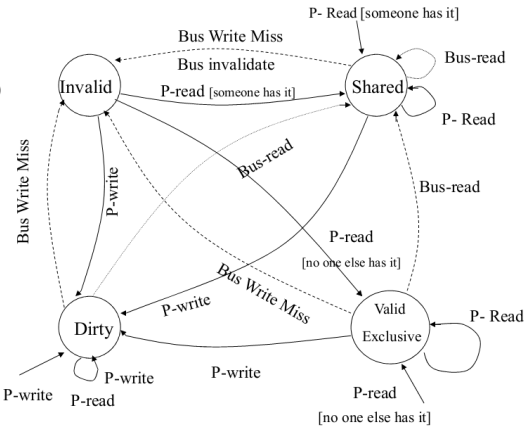
- 思路: 增加一个状态表示cache中仅有的拷贝，并且是干净的
  - *独占(Exclusive)*状态
- 如果总线读(*BusRd*)时，没有其它cache有拷贝，块就会被置为*独占(E)*状态
- 写块是可能会导致状态从*独占(E)* → *修改(M)* 的变换!
- MESI 也称为 *Illinois 协议(模式)* [Papamarcos and Patel, ISCA 1984]

64

64

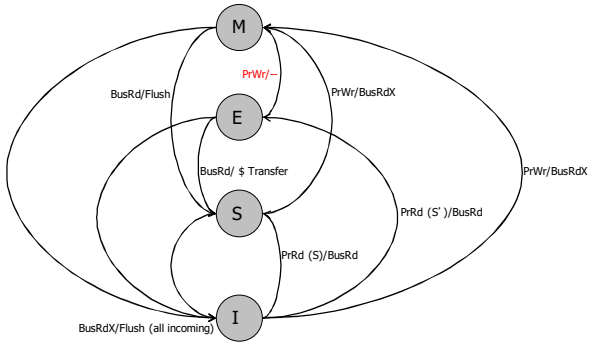
MESI(Illinois 模式)

四种状态：
M(独占拷贝，脏)
E(独占拷贝，干净)
S(共享，干净)
I (无效)



65

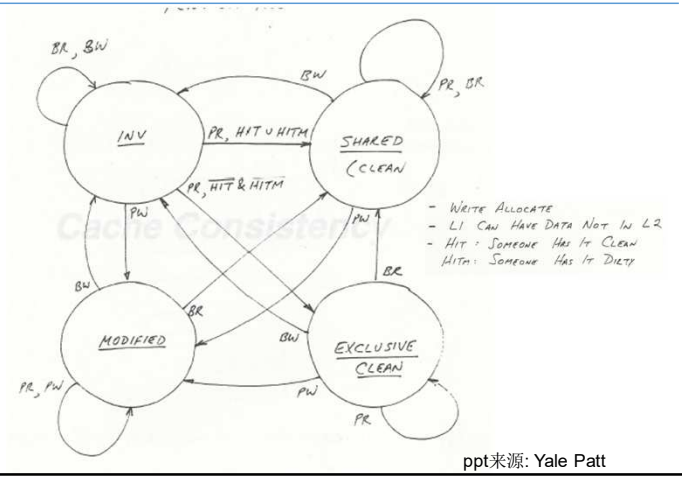
MESI 状态机



[Culler/Singh96]

66

Intel Pentium Pro



ppt来源: Yale Patt

67

MESI的问题

- 共享(S)状态需要数据是干净的
  - 即, 所有拥有这个块的cache必须都拥有最新的拷贝, 并且跟内存中的一致
- 问题: 当有总线读时, 如果块处于修改(M)状态需要将块写回内存
- 为什么这是个问题?
  - 内存可能做不必要的更新 → 其它处理器可能会在cache之后写块

68

改进MESI

- 思路 1: 总线读时不做M→S的状态转换, 将拷贝置为无效, 直接把修改过的块发给请求的处理器而不更新内存
- 思路 2: 做 M→S的转换, 但是指定一个cache作为拥有者 (O), 它负责在块被移除的时候写回
  - 这时候 “共享(S)” 意味着 “共享的并且可能是脏的”
  - 这是MOESI 协议的一个版本

69

69

复杂Cache一致性协议中的tradeoff

■协议可以通过更多的状态和预测机制优化
+ 减少不必要的置无效和块的传输

■当然, 更多的状态和优化
-- 设计和验证更困难(导致更多需要考虑的情况和条件)
-- 收益递减

70

70