

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
2020-06-12

1

提醒：作业

- 作业 6
 - 已发布，6月19日截止
 - 预取和并行

2

实验2-5

- 5月10日发布，预计7月10日截止

3

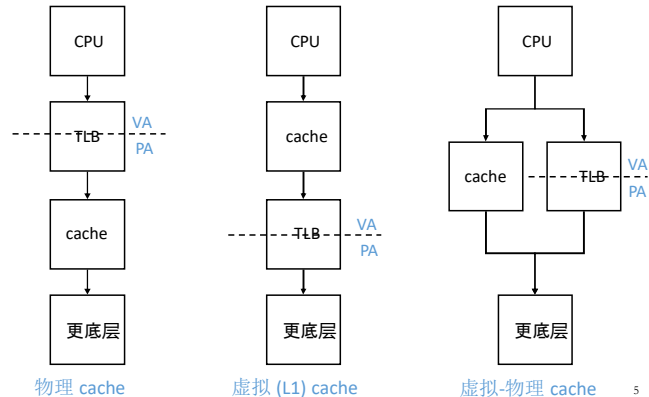
阅读材料

- 并行和多处理
- 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)
 - 第五章：5.8
- 推荐阅读
 - 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.

4

回顾：虚拟存储与cache的交互

- 什么时候需要做地址翻译？
 - 换句话说, cache是虚拟编址还是物理编址？



5

回顾：虚拟-索引,物理-标签

- 如果 $\text{Cache} \leq (\text{页大小} \times \text{相联度})$, cache索引位只来自页的偏移量部分 (在虚拟地址和物理地址中相同)
- 如果片内有cache和TLB
 - 用虚拟地址同时索引cache和TLB
 - cache检查标签(物理的)并比对TLB的输出给出结果
- 如果 $\text{Cache} > (\text{页大小} \times \text{相联度})$, cache索引位将包含VPN \Rightarrow “同义词”会引发问题
 - 同音异义词：相同的声音不同的含义
 - 同一个虚拟地址可能映射到两个不同的物理地址
 - 同义词：不同的声音相同的含义
 - 不同的虚拟地址可能映射到同一个物理地址

6

回顾：虚拟存储与DRAM的交互

- 操作系统会影响DRAM中的地址映射



- 操作系统能够控制虚页映射到哪一个bank/channel/rank
- 可以通过页着色最小化bank的冲突
- 或者最小化应用之间的干扰

7

7

回顾：访存延迟容忍

- 由长延迟指令导致的停顿
 - 满窗口停顿
 - L2 cache的缺失是导致满窗口停顿的最主要原因
- 如何容忍内存导致的停顿？
 - 两种主要方法
 - 减少/消除停顿
 - 当停顿发生时容忍它的影响
 - 四种基本技术
 - 高速缓存
 - 预取
 - 多线程
 - 乱序执行

有很多技术使这四种基本技术在容忍存储延迟时更加有效

8

8

回顾：预取

- 思路: 在程序需要使用之前取数据
- 包括预测哪个地址会是未来需要的
 - 预取时的错误预测不会影响正确性
- 现代系统中, 预取通常在cache块的粒度上实现
 - 预取技术可以减小
 - 缺失率
 - 缺失延迟
 - 预取可以在以下层面实现
 - 硬件
 - 编译器
 - 程序员

9

9

回顾：预取的四个问题(I)

- What
 - 预取什么地址
 - 预取精度 = 有用的预取/ 发出的预取
 - 基于过去访问模式的预测
 - 利用编译器关于数据结构的知识
 - 预取算法决定预取什么
- When
 - 何时发起预取请求
 - 数据被预取的时机影响一个预取器的及时性指标
 - 更加的激进: 尽量保持领先处理器访问流的幅度(硬件)
 - 在代码中更早的发起预取指令(软件)

10

10

回顾：预取的四个问题(II)

- Where
 - 预取的数据放到哪儿
 - 放到cache里
 - 放到独立的预取缓冲中
 - 很多现代系统将预取数据放入cache
 - 预取到哪个级别的cache?
 - 在cache里把预取的数据放到哪儿?
 - 需要调整替换策略以使它能够更优待按需取的块吗?
 - 硬件预取器应该放在分层存储结构的什么位置?
 - 预取器应该看到什么样的访问模式?
- How
 - 软件、硬件、基于执行.....

11

11

回顾：预取的四个问题(III)

- 软件预取
 - ISA 提供预取指令
 - 程序员或编译器插入预取指令
 - 通常只对“常规的访问模式”有效
- 硬件预取
 - 硬件监控处理器的存取
 - 记录或者发现模式
 - 自动生成预取地址
- 基于执行的预取器
 - 执行一个“线程”为主程序预取数据
 - 可以通过软件/程序员或者硬件生成

12

12

回顾：预取器性能

- **精度** (有用的预取 / 发出的预取)
- **覆盖率** (预取的缺失 / 所有的缺失)
- **及时性** (准时的预取 / 有用的预取)
- 带宽消耗
 - 有/没有预取器时，存储带宽的消耗
 - 好消息: 可以利用空闲时的总线带宽
- Cache污染
 - 由于预取放在cache中导致的额外的按需访问缺失
 - 很难量化，但是会影响性能

13

13

回顾：如何预取不规则访问模式？

- 规则模式: 跨度, 流预取器
- 不规则访问模式
 - 间接数组访问
 - 链式数据结构
 - 多跨度(1,2,3,1,2,3,1,2,3,...)
 - 随机模式?
 - 针对所有模式的通用预取器?
- 基于相关性的预取器
- 基于内容的预取器
- 基于预计算或预执行的预取器

14

14

并行

15

15

并行结构

- 为什么采用并行结构
 - 绝对性能
 - 功耗和能耗
 - 复杂度
 - 性能价格比
- 关键的使能因素
 - 半导体和互连网络技术的发展
 - 软件技术的发展

16

16

并行机

- 定义：并行机就是一系列能够通过协同和通信来快速解决大规模问题的处理单元的集合 —— Almasi and Gottlieb
- “...处理单元的集合...”
 - 多少个？每个处理单元的能力有多强？可扩展性如何？
- “...能够通信...”
 - 如何通信？(shared-memory 还是 message passing)
 - 互连网络的结构如何？(bus, crossbar, multistage...)
 - 评价指标：cost, latency, throughput, scalability, fault tolerance

17

17

并行机（续）

- “... 能够协同...”
 - 重要的问题有：synchronization, granularity, and autonomy
 - 同步允许按顺序操作保证正确性
 - 各种原语 (test&set, fetch&add...)
 - 粒度要与程序中的子任务大小相对应
 - 减小粒度 → Parallelism, Communication, Overhead increase
 - 粒度 vs 指令数
 - Program level: 10^6 instr
 - Task level: $10^3 - 10^6$ instr
 - Loop level: 10 - 1000 instr
 - Stmt/instr level: 1 - 10 instr
- 自治性：tradeoff

18

18

并行机（续）

- “... 快速解决大规模问题...”
 - 通用 vs 专用？
 - 任何机器都可以很好的解决某一类问题
 - 什么样的应用适合并行处理？
 - 高并行应用
 - 很多科学计算利用数据的并行性 (MonteCarlo)
 - 中并行应用
 - 很多工程性的应用 (有限元, VLSI-CAD)
 - 不太并行的应用
 - 需要考虑应用和机器的规模

19

19

并行机的分类

- Michael Flynn的分类
- 基于模型的分类

20

20

Flynn分类法

- Mike Flynn, “[Very High-Speed Computing Systems](#),” Proc. of IEEE, 1966
- 基于指令流、数据流的概念
- 根据硬件对指令流和数据流的支持不同来区分并行的组织
 - SISD: Single Instruction and Single Data Stream (uniprocessor) 单指令操作单个数据元素
 - SIMD: Single I and Multiple D Streams (GPUs) 单指令操作多个数据元素
 - 阵列处理机
 - 向量处理器
 - MISD: Multiple I and Single D Streams 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: Multiple I and Multiple D Streams (multicores) 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

21

21

基于模型的分类

- 共享内存 (Shared-memory)
- 消息传递 (Message-passing)
- 数据流 (Dataflow)
- 脉动阵列 (systolic)
- 数据并行 (Data parallel)

22

22

多处理器和多处理中的难题

23

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
 - 提高成本效率和可扩展性, 降低复杂性
 - 提高可靠性: 在空间上冗余执行

24

24

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)

25

25

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

26

26

多处理基础

27

27

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

28

28

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

29

29

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

30

30

Amdahl定律

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

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

并行加速比的瓶颈

Amdahl定律的内涵:

- 1、当 $\alpha < 1$ 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 $\alpha \approx 1$

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

31

31

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

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

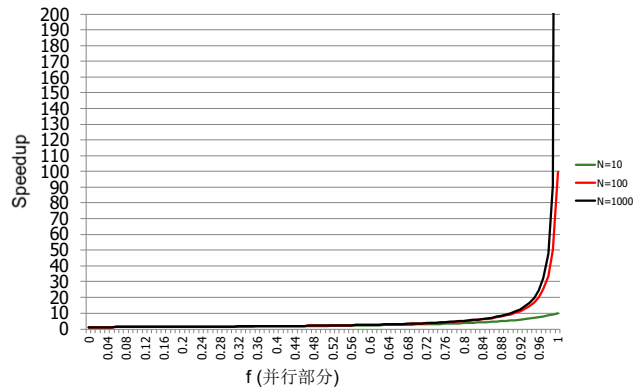
Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- 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)

32

32

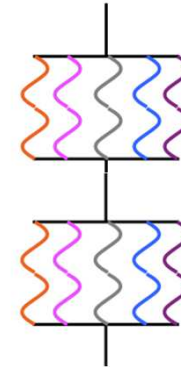
串行瓶颈



33

33

为什么串行是瓶颈?



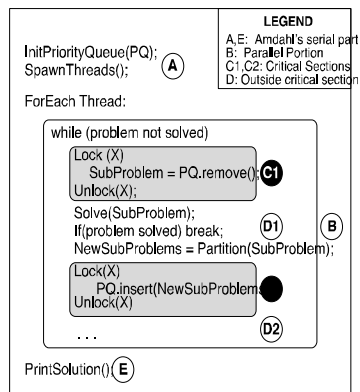
- 并行的机器有串行瓶颈
- 主要原因: 有不能并行化的数据操作(比如, 不能并行化的循环)


```
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)

34

34

串行瓶颈的例子



35

35

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

36

36

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

37

37

多处理中访存的序

38

38

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

39

39

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

40

40

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

41

41

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?

42

42

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

43

43

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

44

44

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。。。。

45

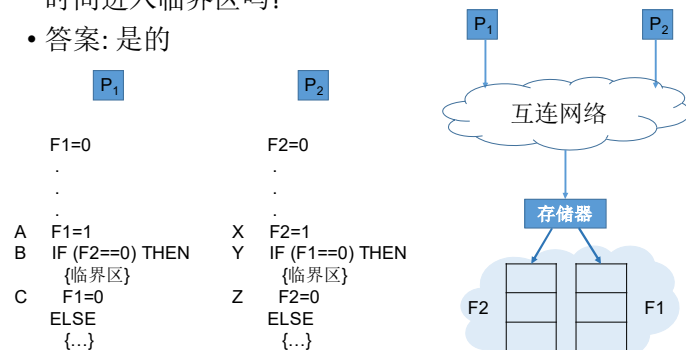
保护共享数据



46

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

47

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

48

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

49

49

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

50

50

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

51

51

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)

52

52

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

53

53

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

54

54

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

55

55

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

56

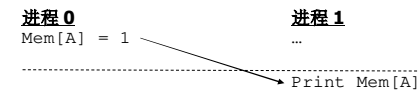
56

Cache一致性 (Cache Coherence)

57

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

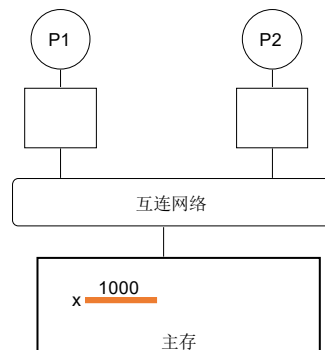


- 每一个读操作都能够收到任何人最后一次写的值
 - 这需要同步 (最后一次写是什么意思?)
- 如果Mem[A]在cache中(两个进程端都有)会怎么样?

58

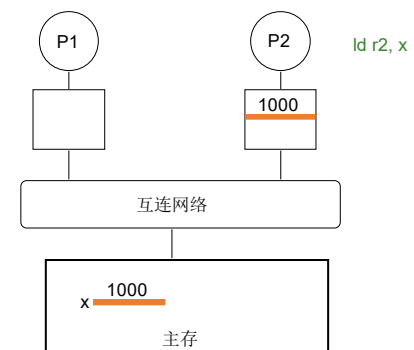
Cache 一致性

- 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?



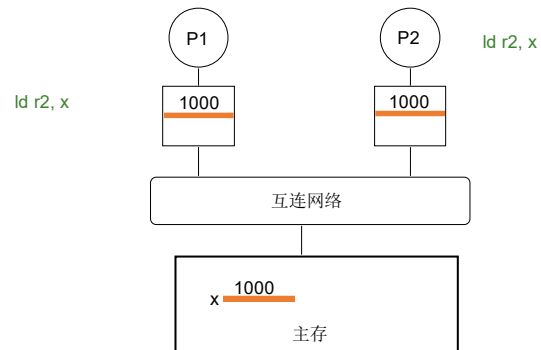
59

Cache一致性问题



60

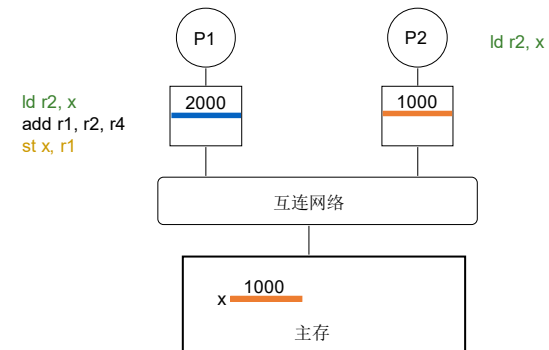
Cache一致性问题



61

61

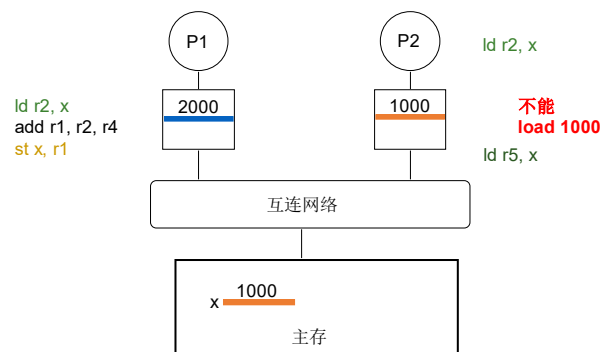
Cache一致性问题



62

62

Cache一致性问题



63

63

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

64

64

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

65

65

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

66

66

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

67

67

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)

68

68

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)

69

69

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)

70

70

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

71

71

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

72

72

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?

73

73

监听Cache一致性

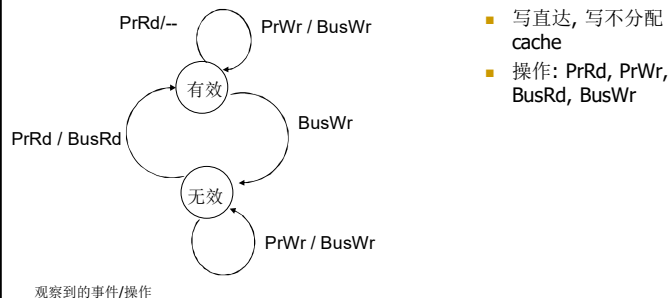
- 思路:
 - 所有cache“监听”所有其它cache的读/写请求,并保持cache块的一致性
 - 每个cache块在每个cache的标签存储中关联“一致性元数据”
- 当所有cache共享一个公共总线时很容易实现
 - 每个cache在总线上广播它的读/写操作
 - 对于小规模的多处理器很有效
 - 如果是1000个节点的多处理器会怎么样?

74

74

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:



75

75

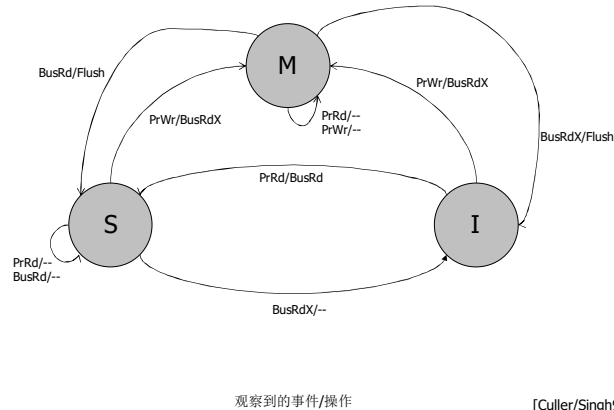
更复杂的协议: MSI

- 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)

76

76

MSI 状态机



77

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

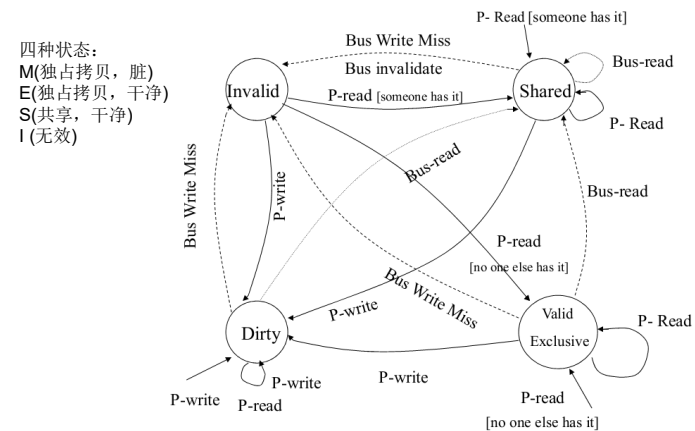
78

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*) \rightarrow 修改(*M*) 的变换!
- MESI 也称为 *Illinois 协议(模式)* [Papamarcos and Patel, ISCA 1984]

79

MESI(Illinois 模式)

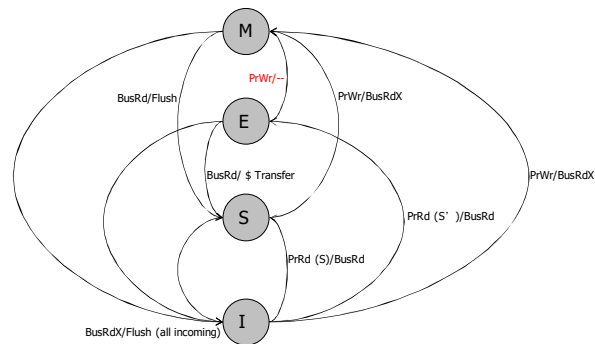


80

79

80

MESI 状态机

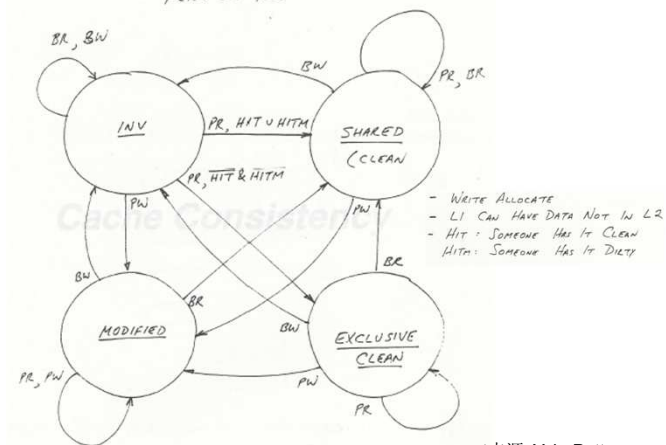


[Culler/Singh96]

81

81

Intel Pentium Pro



ppt来源: Yale Patt

82

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

83

83

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

84

84

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

85

85

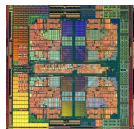
多核设计

86

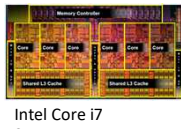
86

片上众核

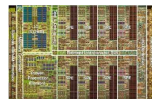
- 比一个大核更简单并且功耗更低
- 片上大规模并行



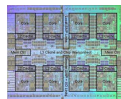
AMD Barcelona
4 cores



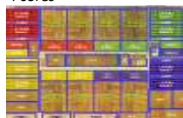
Intel Core i7
8 cores



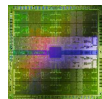
IBM Cell BE
8+1 cores



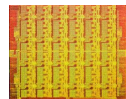
IBM POWER7
8 cores



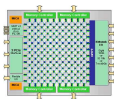
Sun Niagara II
8 cores



Nvidia Fermi
448 "cores"



Intel SCC
48 cores, networked



Tiler TILE Gx
100 cores, networked

87

87

对于片上多核

- 我们想要:
 - 当我们在N个核上并行化一个应用, 我们能获得N倍在单个核上的性能
- 我们能够得到:
 - Amdahl定律 (串行瓶颈)
 - 并行部分的瓶颈

88

88

回顾：并行性

- Amdahl定律

- f: 程序中可并行的部分
- N: 处理器个数

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

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

- 最大加速比受限于串行部分: 串行瓶颈

- 并行部分通常不完美

- 同步开销 (比如, 对共享数据的更新)
- 负载均衡开销 (并行化不完美)
- 资源共享开销 (N个处理器之间的竞争)

89

89

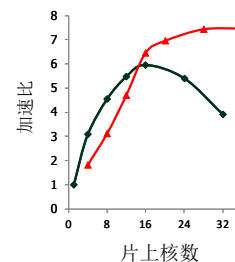
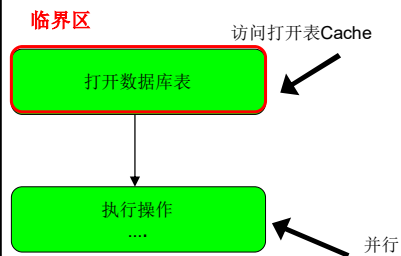
问题: 串行化的代码段

- 很多并行程序无法完全并行化
- 串行化代码段的产生
 - 连续的部分(Amdahl的“串行部分”)
 - 临界区
 - 栅障
 - 流水化程序中的受限阶段
- 串行化的代码段
 - 降低性能
 - 限制扩展性
 - 浪费能源

90

90

MySQL的例子



91

91

不同代码段的需求

- 我们想要:
- 串行代码段 → 一个强有力的“大”核
- 并行代码段 → 许多弱的“小”核
- 这两者互相冲突:
 - 如果你有一个强有力的核, 就不可能同时拥有很多核
 - 一个小核在能耗和面积消耗方面都远比一个大核更高效

92

92

“大” vs. “小” 核



- 乱序
- 宽取指
- 更深的流水线
- 激进的分支预取器
- 很多的功能单元
- 内存依赖的投机
-



- 按序
- 窄取指
- 浅的流水线
- 简单的分支预取器
- 很少的功能单元

大核的功效低:
比如, 4x的面积(功率)只能获得2x的性能

93

大核 vs. 小核

- Grochowski et al., “Best of both Latency and Throughput,” ICCD 2004.

	Large core	Small core
Microarchitecture	Out-of-order, 128-256 entry ROB	In-order
Width	3-4	1
Pipeline depth	20-30	5
Normalized performance	5-8x	1x
Normalized power	20-50x	1x
Normalized energy/instruction	4-6x	1x

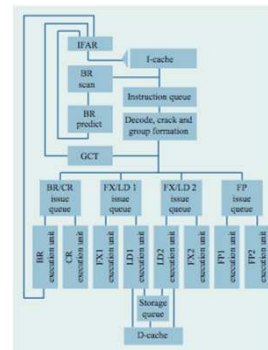
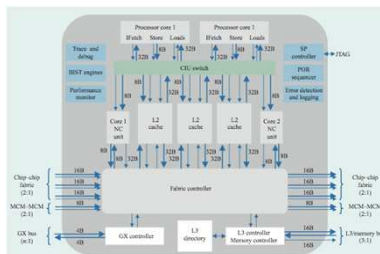
94

94

大核: IBM POWER4

- Tendler et al., “POWER4 system microarchitecture,” IBM J R&D, 2002.

- 另外一种对称多核芯片...
- 但是, 更少、更强有力的核



95

95

IBM POWER4

- 2个核, 乱序执行
- 每核100-entry的指令窗口
- 8宽度取指、发射和执行
- 大容量, 本地+全局混合分支预测器
- 1.5MB, 8路 L2 cache
- 基于流的激进预取

96

96

IBM POWER5

- Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro 2004.

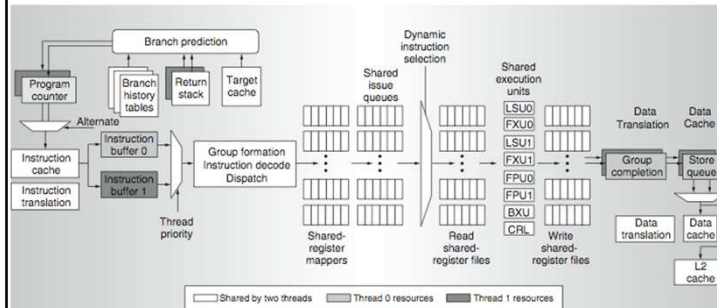


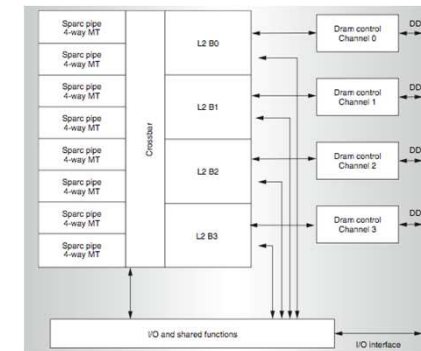
Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

97

97

小核: Sun Niagara (UltraSPARC T1)

- Kongetira et al., "Niagara: A 32-Way Multithreaded SPARC Processor," IEEE Micro 2005.

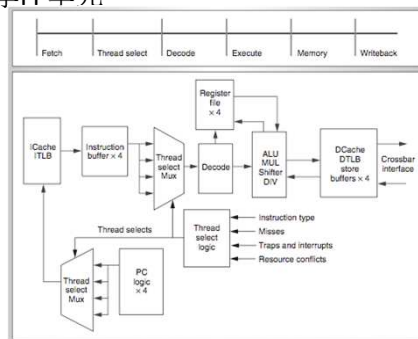


98

98

Niagara的核

- 4路细粒度多线程, 6阶段双发射按序执行
- 循环线程选择(除非发生cache缺失)
- 核间共享FP单元



99

回顾: 需求

- 我们想要:
 - 串行代码段 → 一个强有力的“大”核
 - 并行代码段 → 许多弱的“小”核
- 这两者互相冲突:
 - 如果你有一个强有力的核, 就不可能同时拥有很多核
 - 一个小核在能耗和面积消耗方面都远比一个大核更高效
- 能否两全其美?

100

100

性能 vs. 并行性

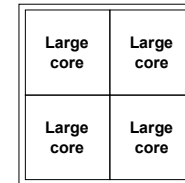
假设:

1. 小核用1个面积的预算获得1份性能
2. 大核用4个面积的预算获得2份性能

101

101

铺砌大核



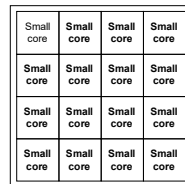
“铺砌大核”

- 铺砌少量大核
- IBM Power 5, AMD Barcelona, Intel Core2Quad, Intel Nehalem
- + 单线程、串行代码段时可获得高性能
- 并行程序段时吞吐率低

102

102

铺砌小核



“铺砌小核”

- 铺砌很多小核
- Sun Niagara, Intel Larrabee, Tiler TILE
- + 并行部分吞吐率高
- 串行部分、单线程性能低

103

103

两全其美?

- 铺砌大核
 - + 单线程、串行代码段时可获得高性能
 - 并行程序段时吞吐率低
- 铺砌小核
 - + 并行部分吞吐率高
 - 串行部分、单线程性能低, 相比现有单线程处理器性能还差
- 思路: 在一个芯片上同时集成大核和小核 → 性能不对称

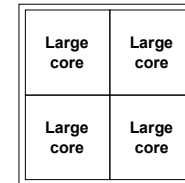
104

104

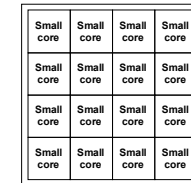
非对称多核

105

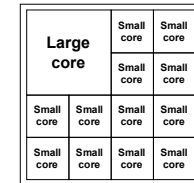
非对称片上多处理器(ACMP)



“铺砌大核”



“铺砌小核”



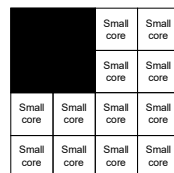
ACMP

- 提供一个大核和多个小核
- + 利用大核加速串行部分
- + 在小核和大核上执行并行部分以获得高的吞吐率

106

加速串行瓶颈

单线程 → 大核

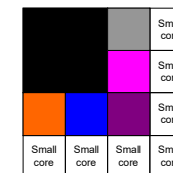


ACMP

107

加速串行瓶颈

单线程 → 大核

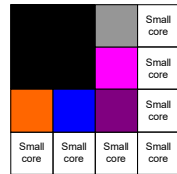
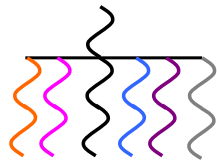


ACMP

108

加速串行瓶颈

单线程 → 大核



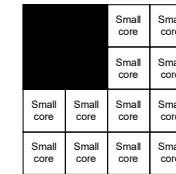
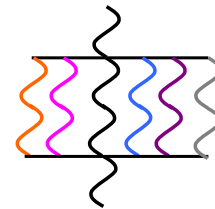
ACMP

109

109

加速串行瓶颈

单线程 → 大核



ACMP

110

110

性能 vs. 并行性

假设:

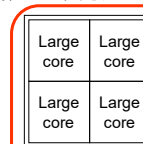
1. 小核用1个面积的预算获得 1 份性能
2. 大核用4个面积的预算获得2份性能

111

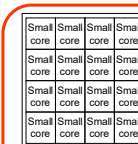
111

ACMP 性能 vs. 并行性

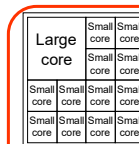
面积预算 = 16 个小核



“铺砌大核”



“铺砌小核”



ACMP

大核	4	0	1
小核	0	16	12
串行性能	2	1	2
并行吞吐	$2 \times 4 = 8$	$1 \times 16 = 16$	$1 \times 2 + 1 \times 12 = 14$

112

112

再审视：并行性

• Amdahl定律

- f: 程序中可并行的部分
- N: 处理器个数

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

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

• 最大加速比受限于串行部分: 串行瓶颈

• 并行部分通常不完美

- 同步开销 (比如, 对共享数据的更新)
- 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)

113

113

加速并行瓶颈

- 并行部分中的序列化或不均衡的执行同样可以得益于大核

• 例子:

- 临界区竞争
- 比别的阶段执行时间更长的并行阶段

- 思路: 动态判别会导致序列化执行的代码段并将它们放到大核上执行

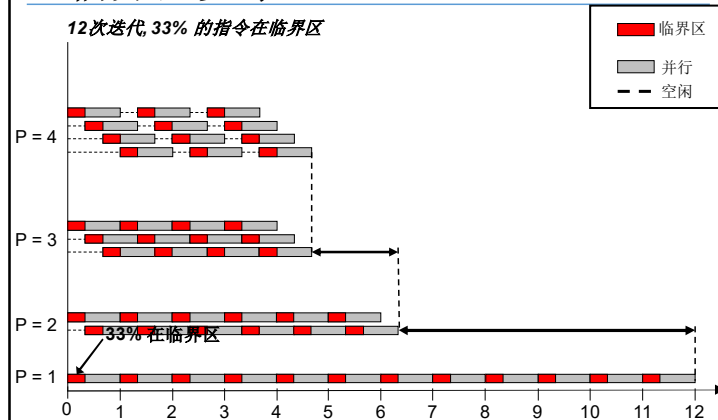
- 加速临界区
- 瓶颈识别和调度

114

114

临界区竞争

12次迭代, 33% 的指令在临界区

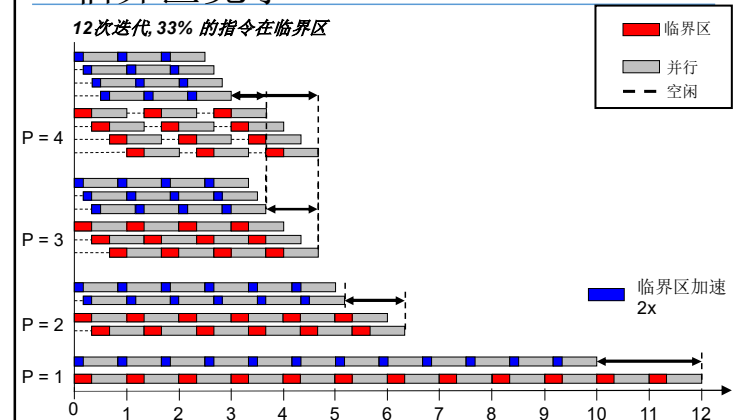


115

115

临界区竞争

12次迭代, 33% 的指令在临界区

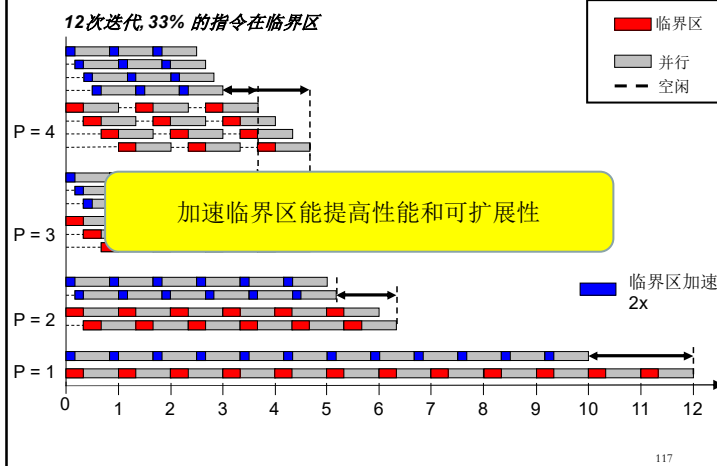


116

116

临界区竞争

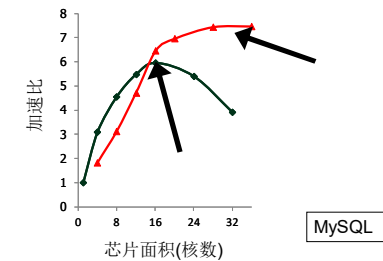
12次迭代, 33% 的指令在临界区



117

临界区对可扩展性的影响

- 临界区竞争导致并程序段中线程的串行执行(序列化)
- 临界区竞争随着线程数增加而增加, 并且限制可扩展性



118

利用不对称

- 串行部分的执行时间必须短
- 对程序员来说缩短这些串行段相当困难
 - 领域知识不够
 - 硬件平台的多样性
 - 受限的资源
- 目标: 一种不需要程序员参与的缩小串行瓶颈的机制
- 思路: 在非对称多核平台上通过将串行代码段迁移到强有力的核上来加速串行部分的执行

119

119

一个例子: 加速临界区

- 思路: 在非对称多核体系结构中将临界区迁移到大的、强有力的核上
- 好处:
 - 减少由于锁的争用带来的串行化
 - 减少无法并行化的部分对性能的影响
 - 程序员不需要(过多地)优化并行代码 → 更少的bug, 提高效率
- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro Top Picks 2010.
- Suleman et al., "Data Marshaling for Multi-Core Architectures," ISCA 2010, IEEE Micro Top Picks 2011.

120

120

多线程应用中的瓶颈

一种定义: 任何会被线程竞争的代码段

例如:

- **Amdahl的串行段**
 - 只有一个线程在执行 → 在关键路径上
- **临界区**
 - 保证互斥 → 如果有竞争很可能就在关键路径上
- **栅障**
 - 再继续推进之前保证所有线程到达该点 → 最后到达的线程在关键路径上
- **流水线阶段**
 - 一个循环迭代的不同阶段可能在不同线程上执行, 最慢的阶段会使其它阶段等待 → 在关键路径上

121

121

观察: 起作用的瓶颈随时间变化

A=满的链表; B=空的链表

repeat

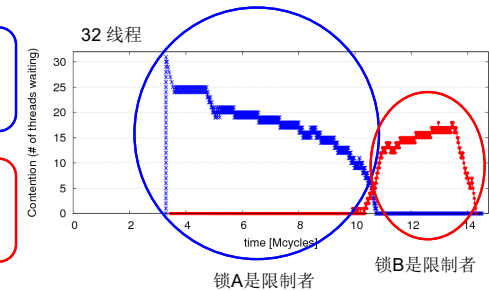
Lock A
 Traverse list A
 Remove X from A

Unlock A

Compute on X
Lock B
 Traverse list B
 Insert X into B

Unlock B

until A is empty

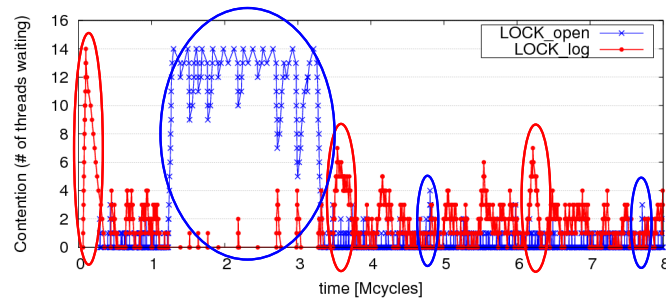


122

122

真实应用中瓶颈产生限制确实是变化的

MySQL运行Sysbench查询, 16 线程



123

123

瓶颈加速的一些研究

- 非对称片上多处理器(ACMP) [Annavaram+, ISCA'05]
[Morad+, Comp. Arch. Letters'06] [Suleman+, Tech. Report'07]
- 临界区加速 (ACS) [Suleman+, ASPLOS'09, Top Picks'10]
- 反馈指引的流水线 (FDP) [Suleman+, PACT'10 and PhD thesis'11]

- 不能加速所有类型的瓶颈
- 不能适应起作用瓶颈(瓶颈重要性)的细粒度变化

目标:

一种通用机制能够识别并加速任何类型的正在影响性能的瓶颈

Jose A. et.al "Bottleneck Identification and Scheduling in Multithreaded Applications", ASPLOS 2012.

124

124

瓶颈识别与调度

- 主要观点:

- 线程等待损害并行性并且可能降低性能
- 代码是导致大多数线程等待的原因 → 可能的关键路径

- 主要思路:

- 动态识别导致大多数线程等待的瓶颈
- 加速它们(使用ACMP中的强有力的核)

Jose A, et.al "Bottleneck Identification and Scheduling in Multithreaded Applications", ASPLOS 2012.

125