

---

# 高等计算机体系结构

## 第十六讲: 多处理与多核设计

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

2021-06-11

# 提醒： 作业

---

- 作业6
  - 已发布，6月18日截止
  - 预取和并行

# 提醒： 实验2-5

---

- 7月16日截止

# 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.

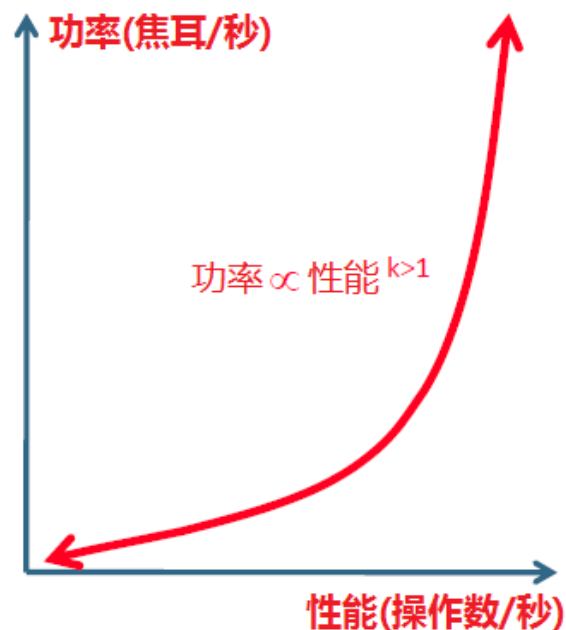
# 回顾：电子学中的能量与功率

---

- 能量(焦耳)以阻抗产生的热耗散掉
  - 每次操作需要一定量的能量，例如，加法、寄存器读/写、对节点充放电
  - 能量 $\propto$ 计算量(功)
- 功率(瓦特=焦耳/秒)是能量耗散率
  - 运算数/秒越高，焦耳/秒就越大
  - 功率 $\propto$ 性能

# 回顾：功率和性能是密不可分的

- 如果不关心性能，很容易将功耗降至最低
- 可以预期功率超线性增加将提高性能
- 推论：较低的性能会产生较低的焦耳/操作
- 总之，越慢越节能

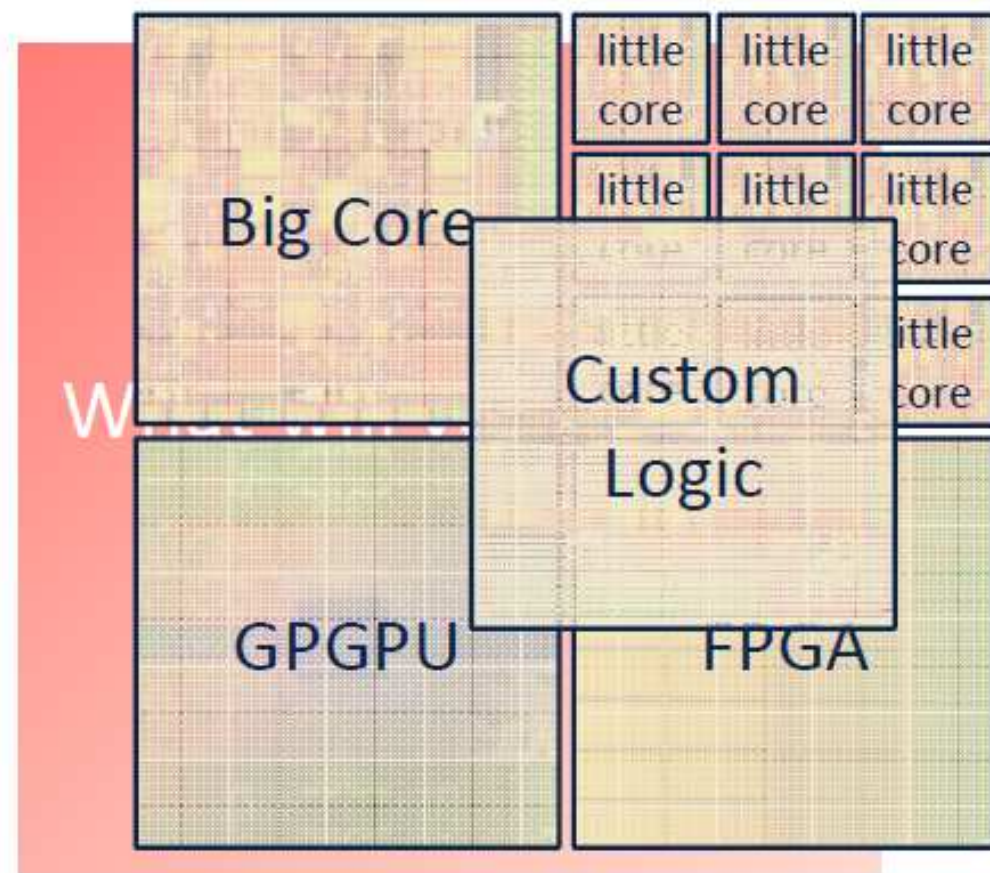


## 回顾：为什么能耗对今天的计算机体系结构如此重要？

---

- 摩尔定律→性能
  - 总的来说，我们以约2x功率获得约2x性能
- 性能效率低下
  - 为了在单线程微处理器上达到“预期”的性能目标
    - 通过增加流水线深度越来越难提高频率
    - 使用2x晶体管构建更复杂的微体系结构(cache、分支预测、超标量、乱序执行)，以使更快/更深的流水线不会停顿
- 登纳德缩放定律率先失效
  - 必须以更少的焦耳/秒完成更多的操作数/秒
- 摩尔定律通过核数的增加继续扩展
- 所以问题在哪儿？
  - 我们知道如何在芯片上封装更多的核，从而在“聚合”或“吞吐”性能方面保持摩尔定律
  - 如何使用它们？能有效地使用多少核？

# 回顾：性能/瓦特和操作数/焦耳





## 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

# 回顾：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）
- .....

# 回顾：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

- 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）

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

# 回顾：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

# 回顾：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

---

# 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



# 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

# 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?

# 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

# 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

# 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。。。


# 保护共享数据

$P_1$

```
F1=0
.
.
.
A  F1=1
B  IF (F2==0) THEN
    {临界区}
C  F1=0
   ELSE
    {...}
```

$P_2$

```
F2=0
.
.
.
X  F2=1
Y  IF (F1==0) THEN
    {临界区}
Z  F2=0
   ELSE
    {...}
```

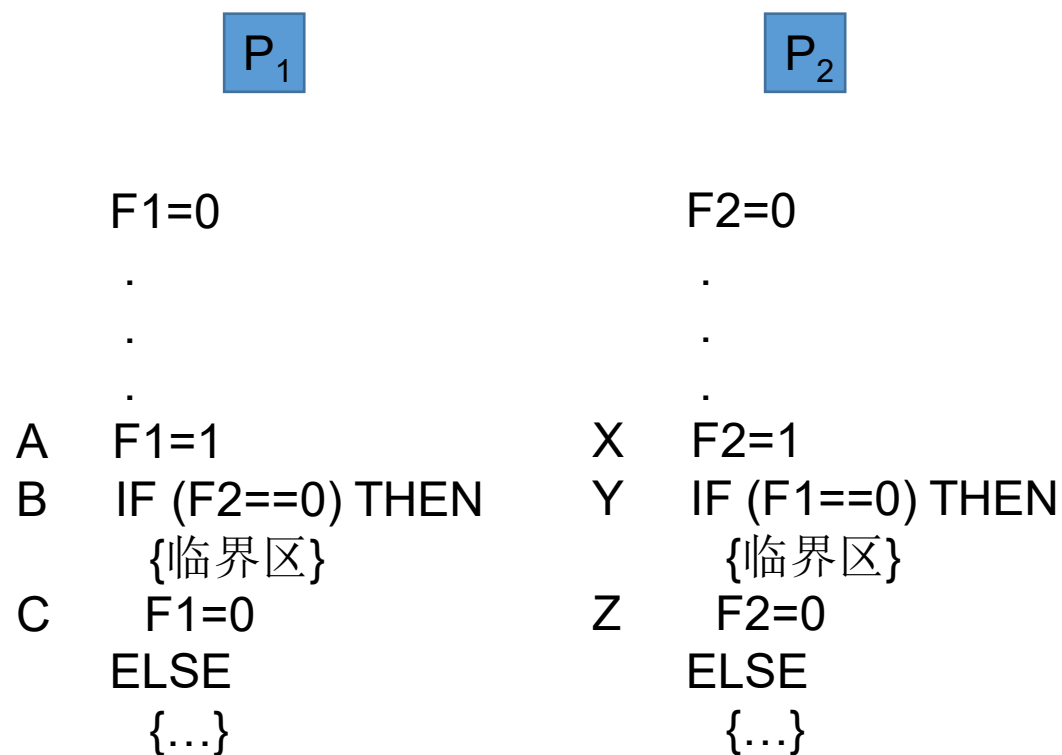


只有P1或者  
P2中的一个  
在某个给定  
时间能够进  
入临界区，  
不能同时

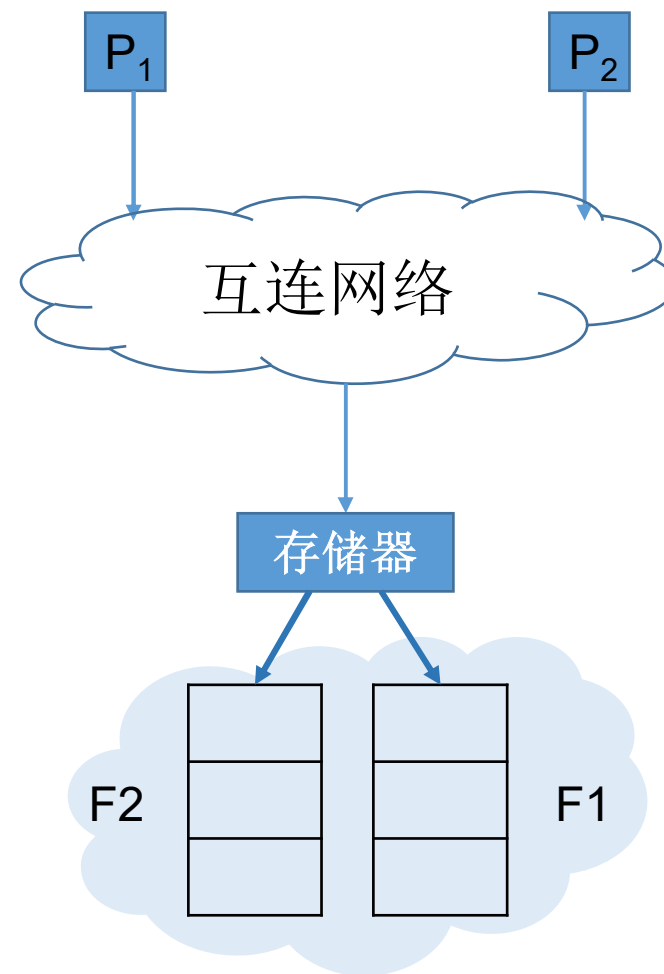
假设  $P_1$  在临界区  
也就是说,  $P_1$  已经执行了 A  
这意味着 F1 必须是 1  
这意味着  $P_2$  不能进入临界区

# 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



# 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



# 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

# 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

# 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

# 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)

# 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

# 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

# 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

# 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



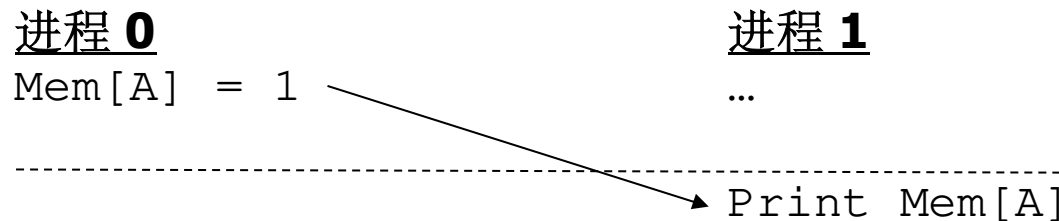
---

# Cache一致性 (Cache Coherence)

# 共享存储模型

---

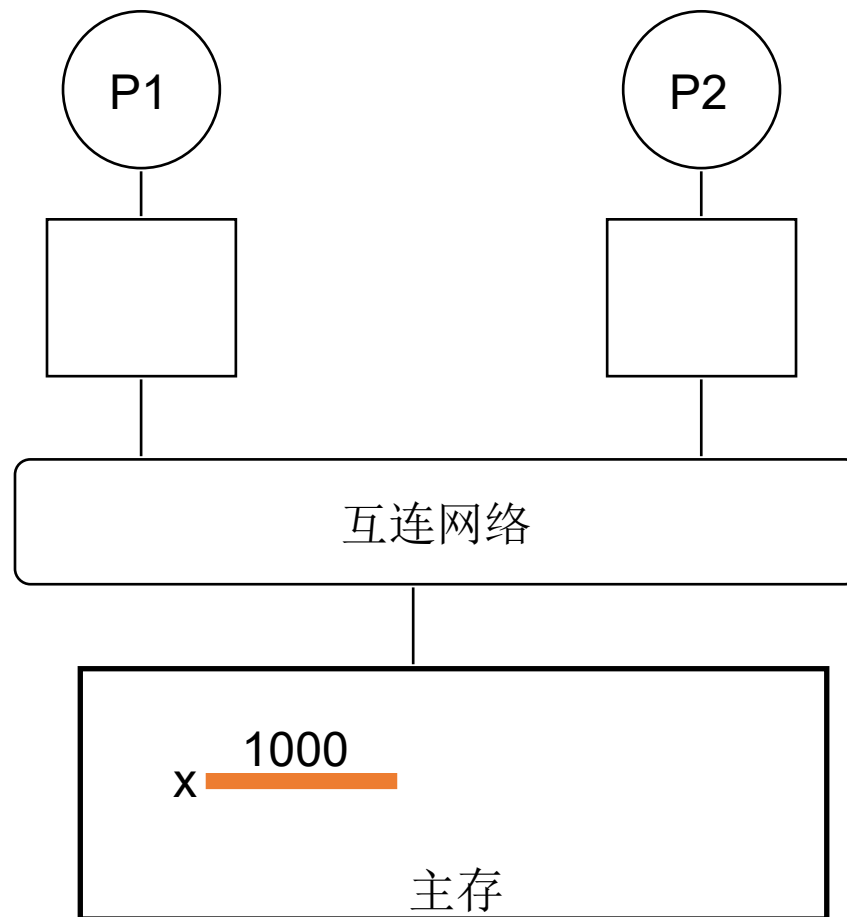
- 很多并行程序通过共享存储通信
- 进程 0 写某个地址，接着进程 1 读
  - 两个进程之间通信



- 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?

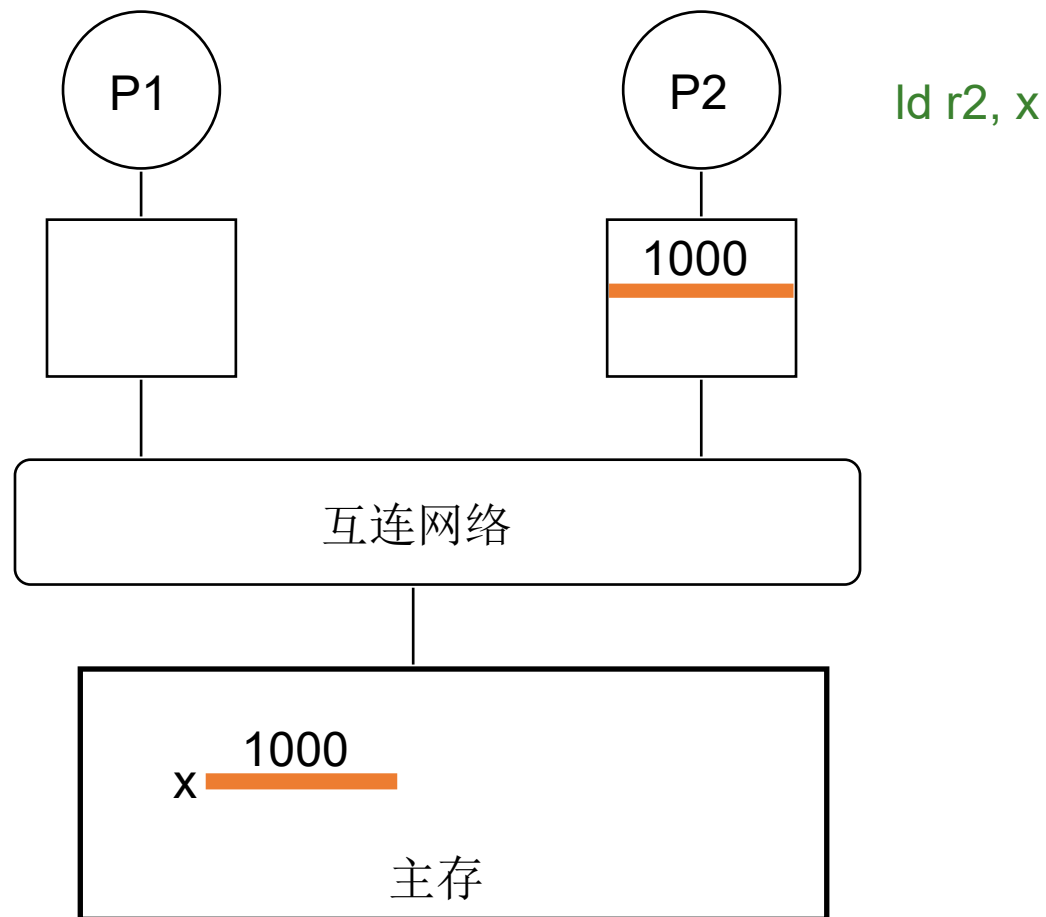
# Cache 一致性

- 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?



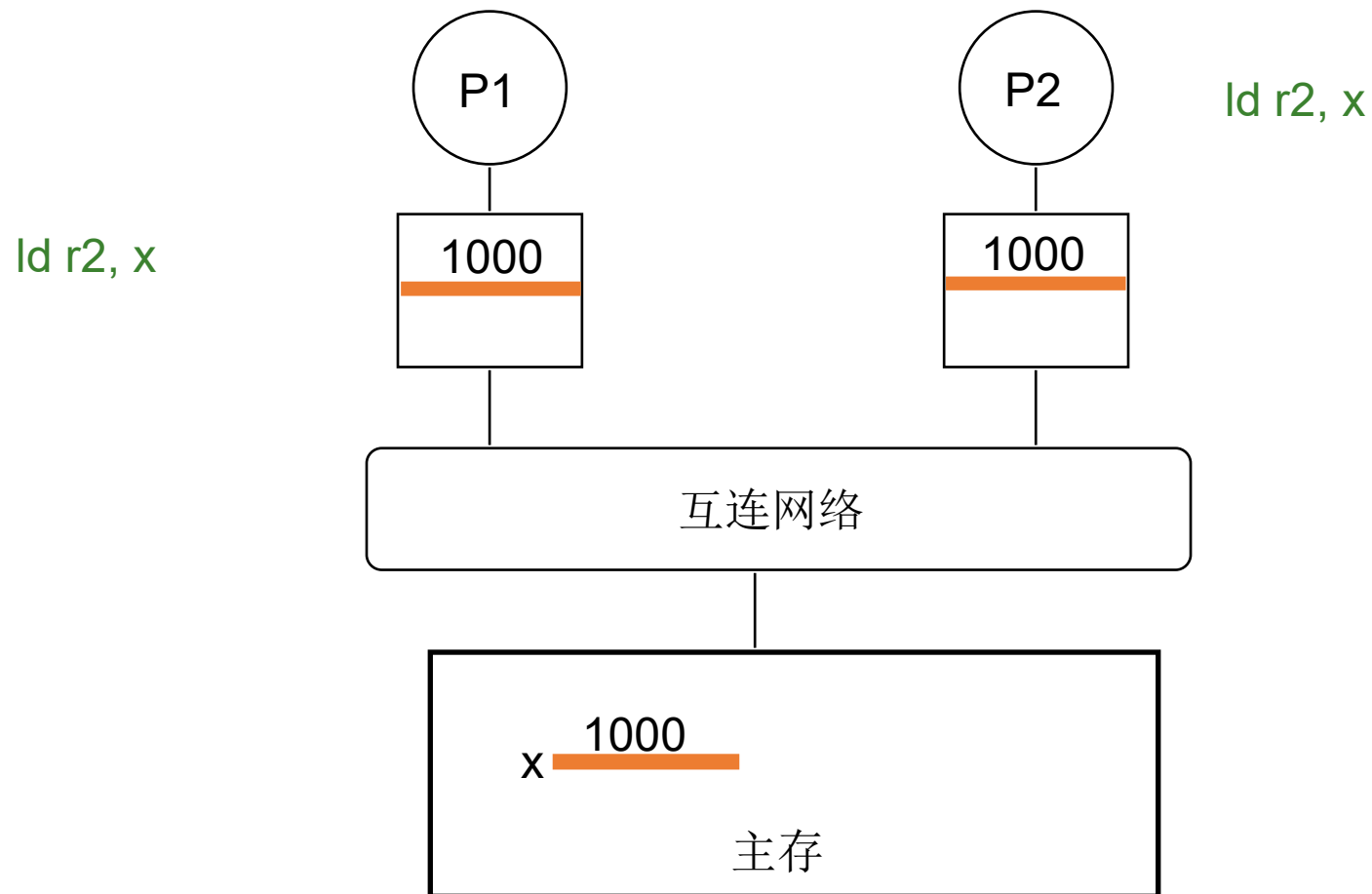
# Cache一致性问题

---

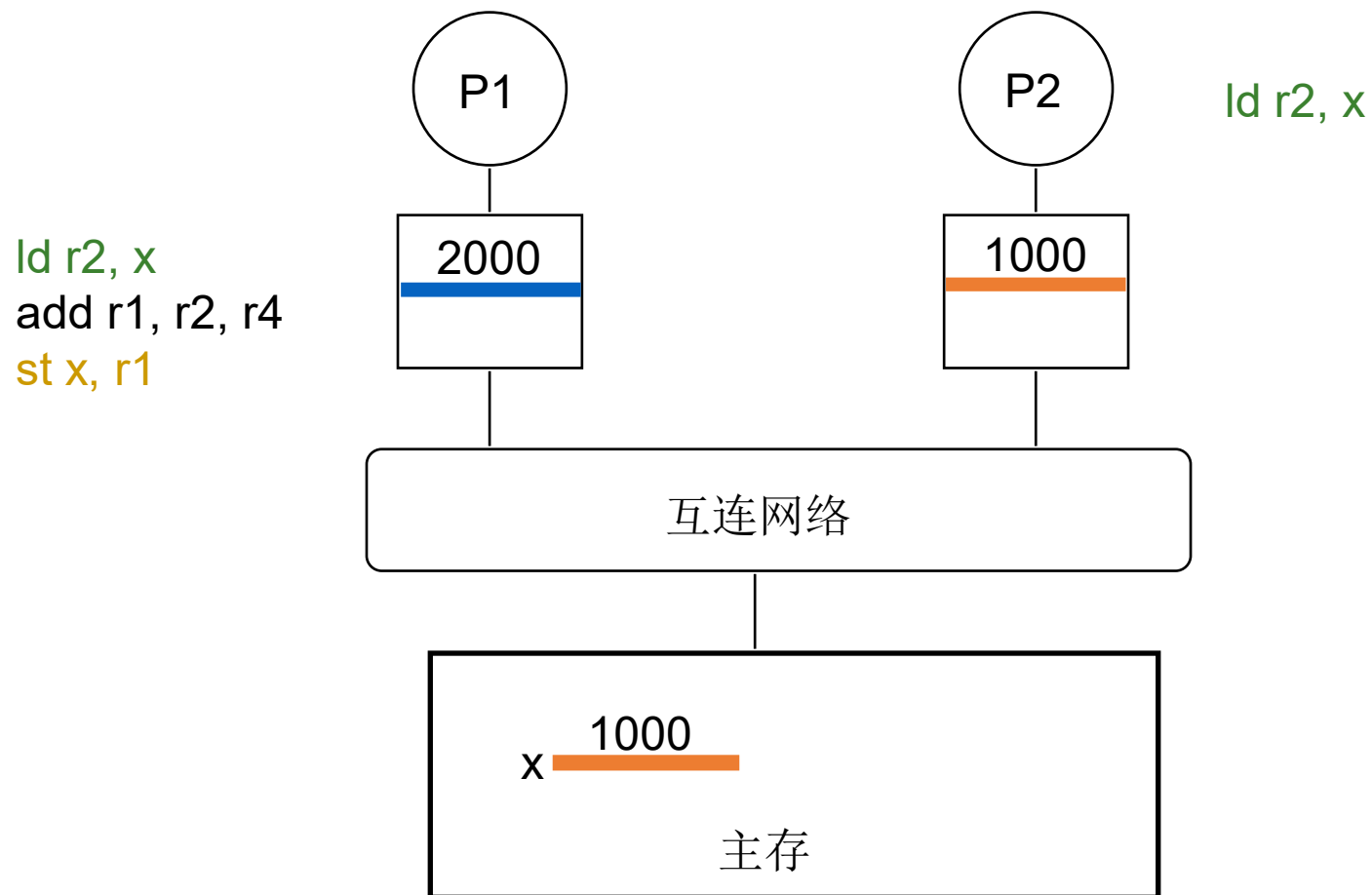


# Cache一致性问题

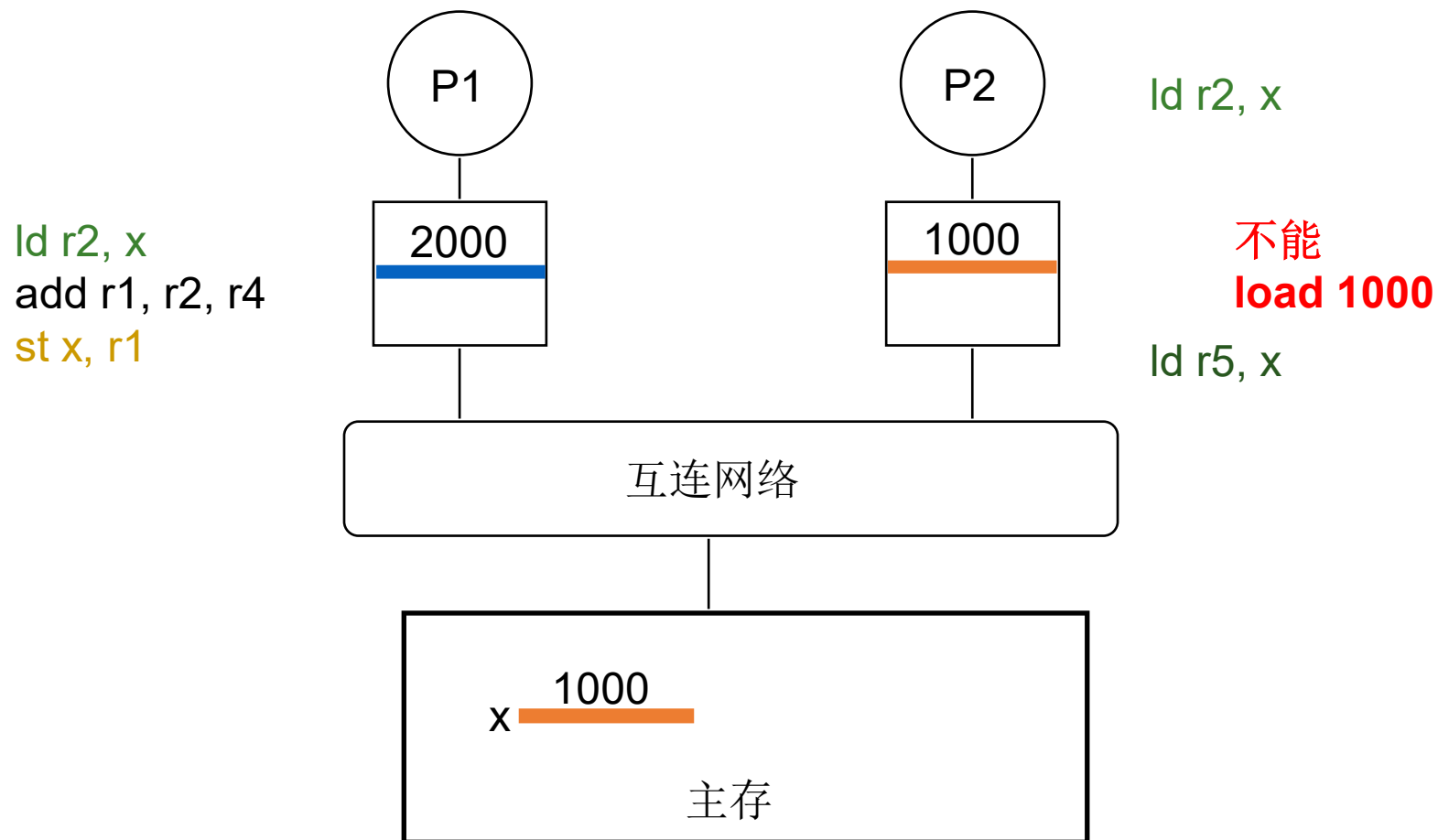
---



# Cache一致性问题



# Cache一致性问题



# 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)

# 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)

# 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)

# 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

# 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



# 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?

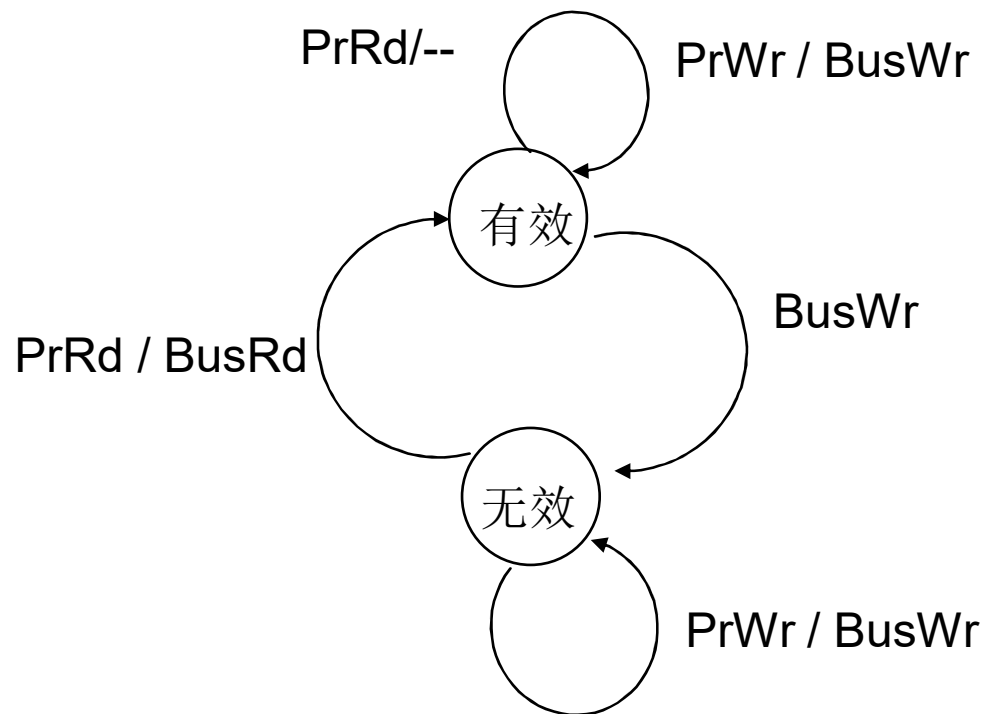
# 监听Cache一致性

---

- 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?

# 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

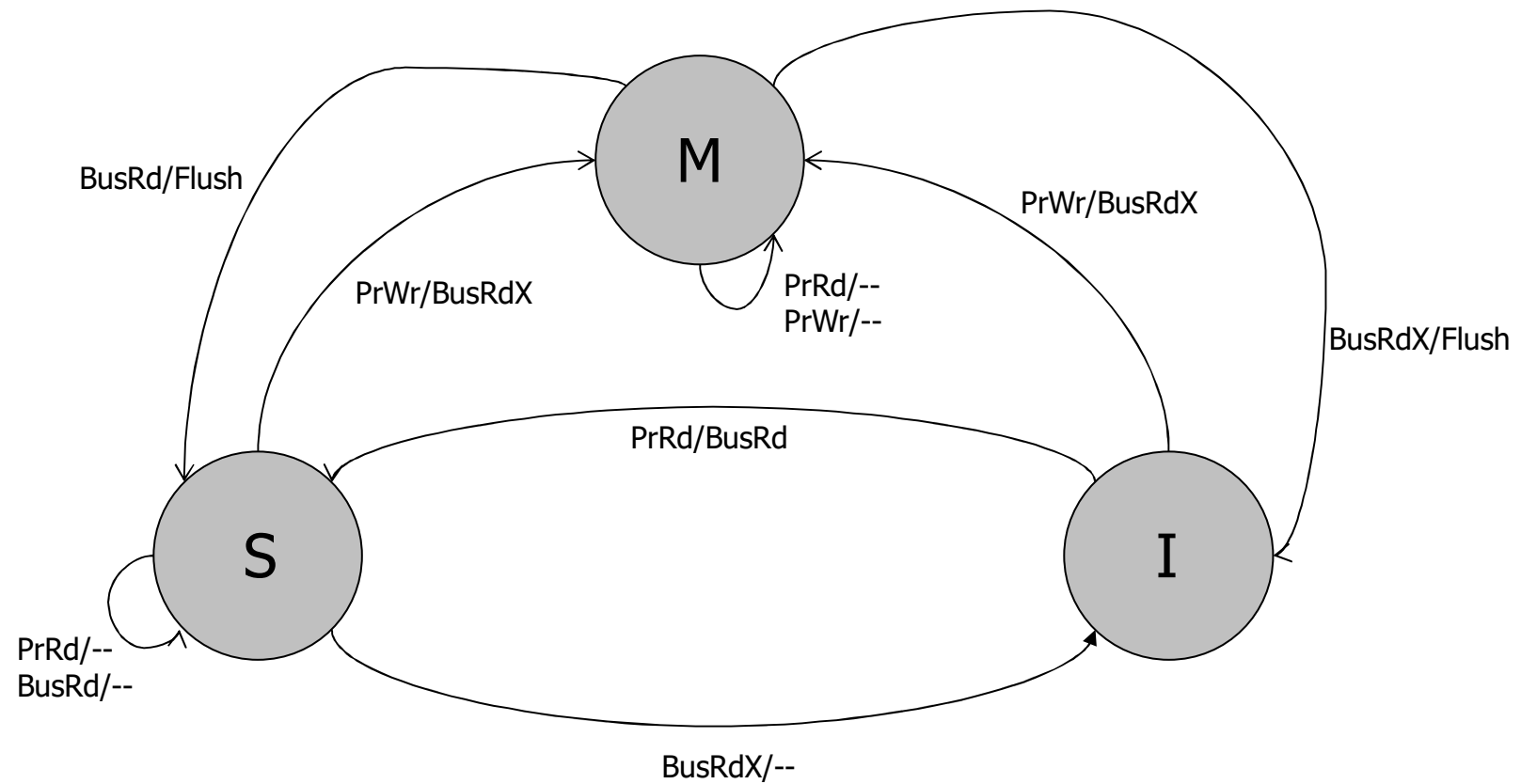
观察到的事件/操作

# 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)

# MSI 状态机



观察到的事件/操作

[Culler/Singh96]

# 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

# 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]

# MESI(Illinois 模式)

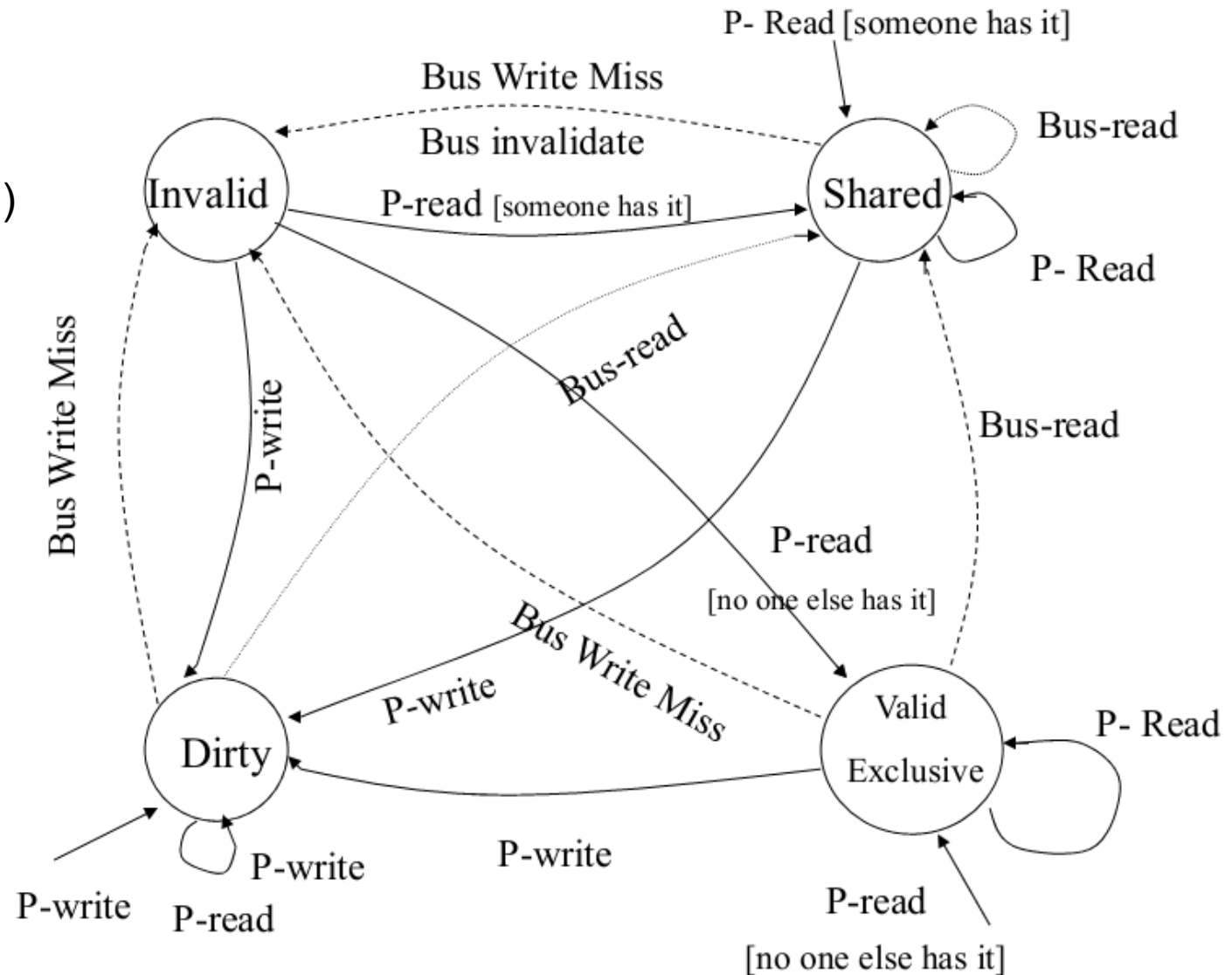
四种状态:

M(独占拷贝, 脏)

E(独占拷贝, 干净)

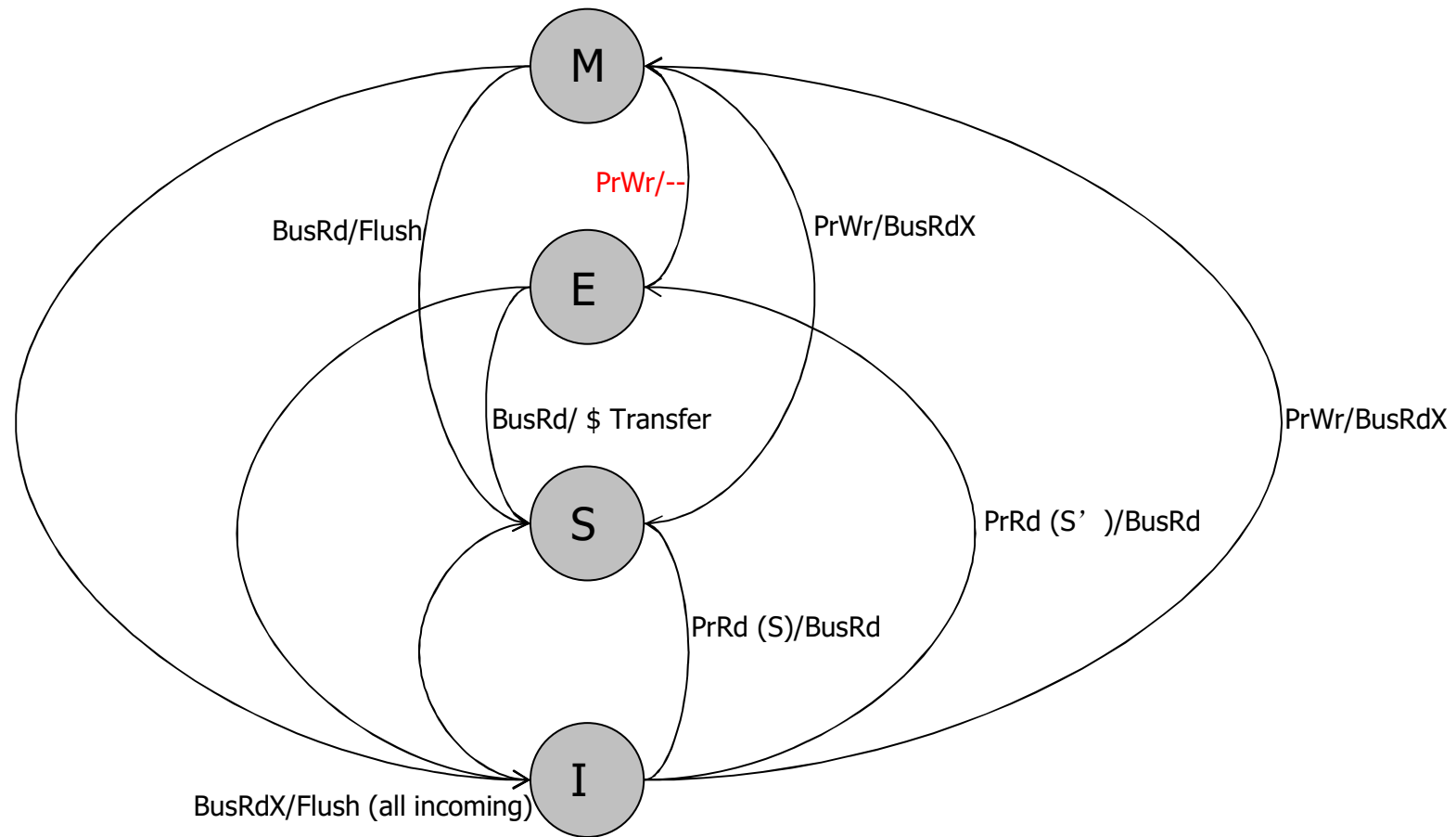
S(共享, 干净)

I(无效)



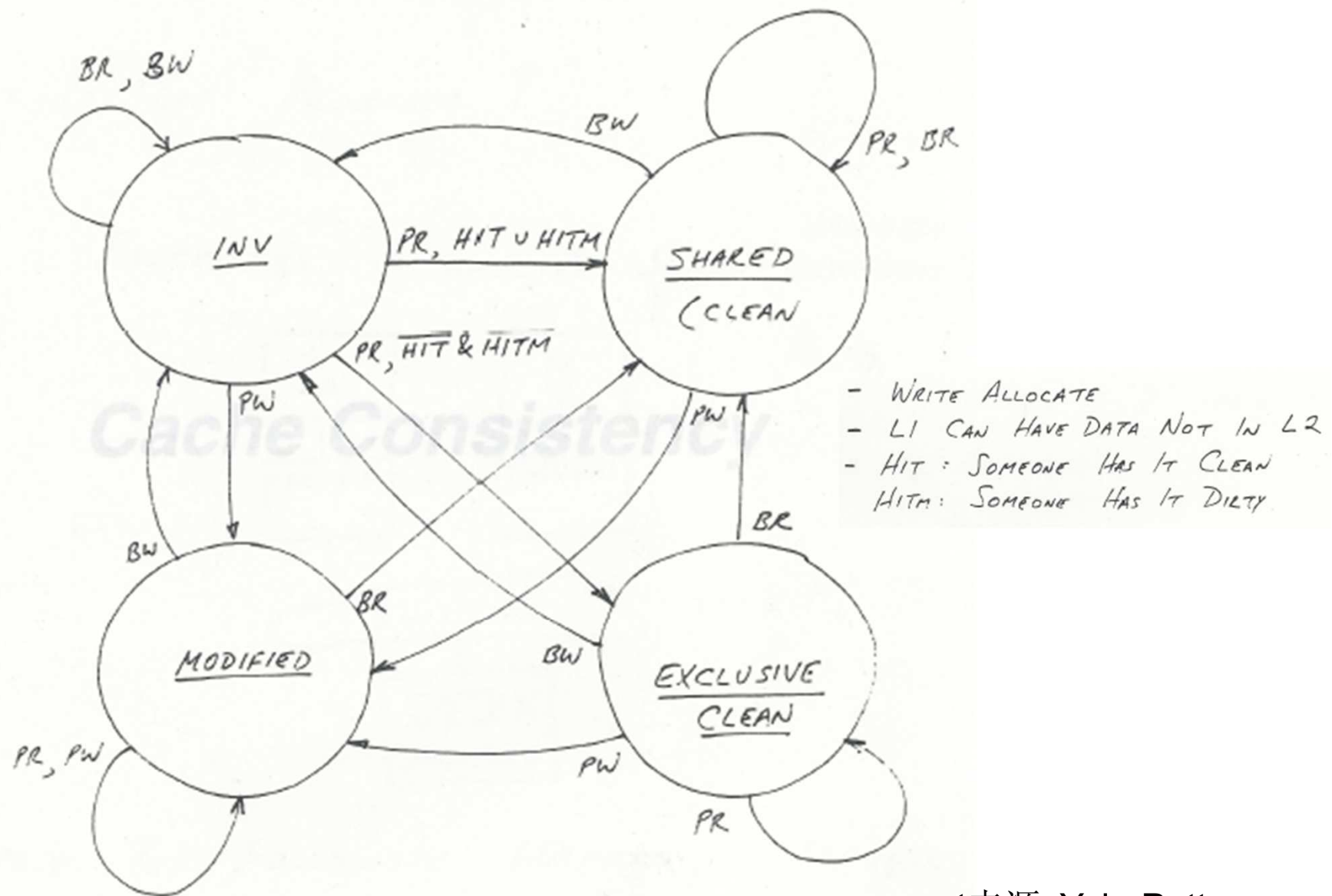


# MESI 状态机



[Culler/Singh96]

# Intel Pentium Pro



# 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

# 改进MESI

---

- 思路 1: 总线读时不做 $M \rightarrow S$ 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  $M \rightarrow S$  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

## 复杂Cache一致性协议中的tradeoff

---

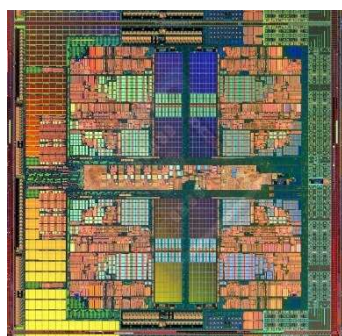
- 协议可以通过更多的状态和预测机制优化
  - + 减少不必要的置无效和块的传输
- 当然, 更多的状态和优化
  - 设计和验证更困难(导致更多需要考虑的情况和条件)
  - 收益递减

---

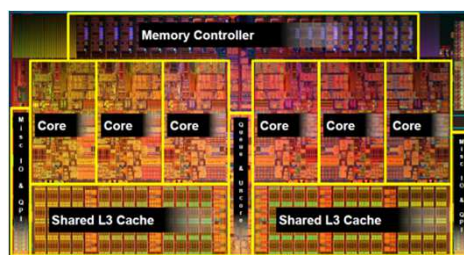
# 多核设计

# 片上众核

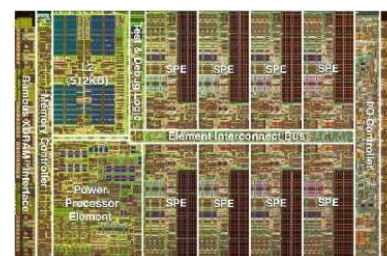
- 比一个大核更简单并且功耗更低
- 片上大规模并行



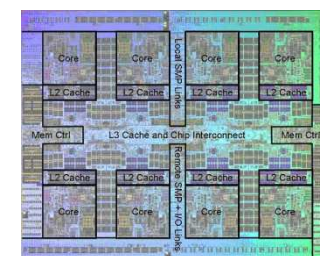
AMD Barcelona  
4 cores



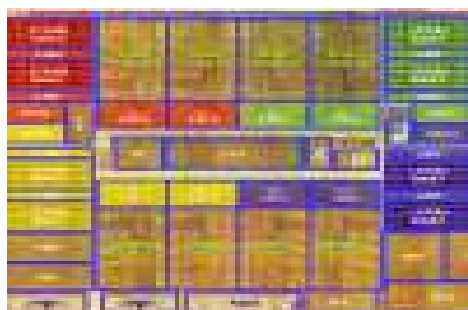
Intel Core i7  
8 cores



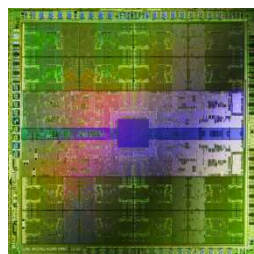
IBM Cell BE  
8+1 cores



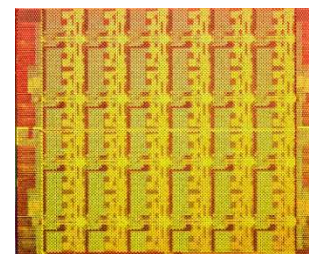
IBM POWER7  
8 cores



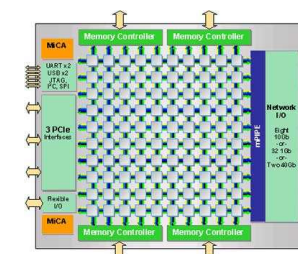
Sun Niagara II  
8 cores



Nvidia Fermi  
448 "cores"



Intel SCC  
48 cores, networked



Tiler TILE Gx  
100 cores, networked

# 对于片上多核

---

- 我们要：
  - 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



# 回顾：并行性

---

- Amdahl定律
  - f: 程序中可并行的部分
  - N: 处理器个数

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

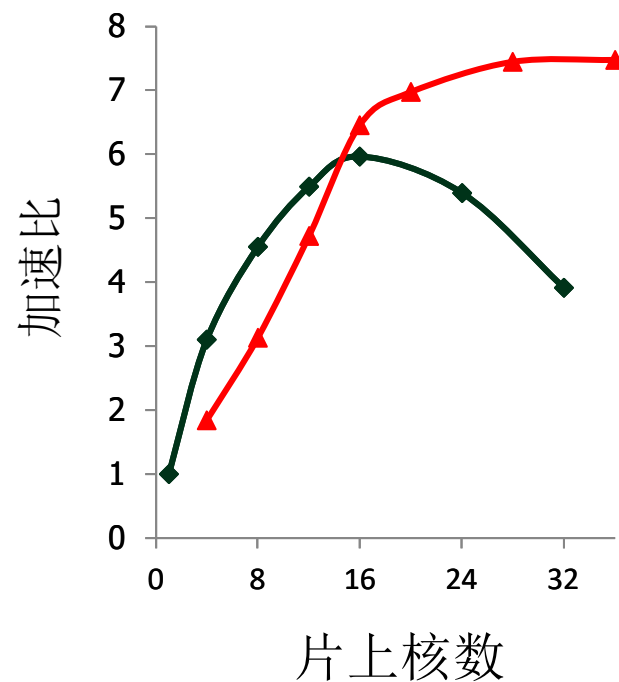
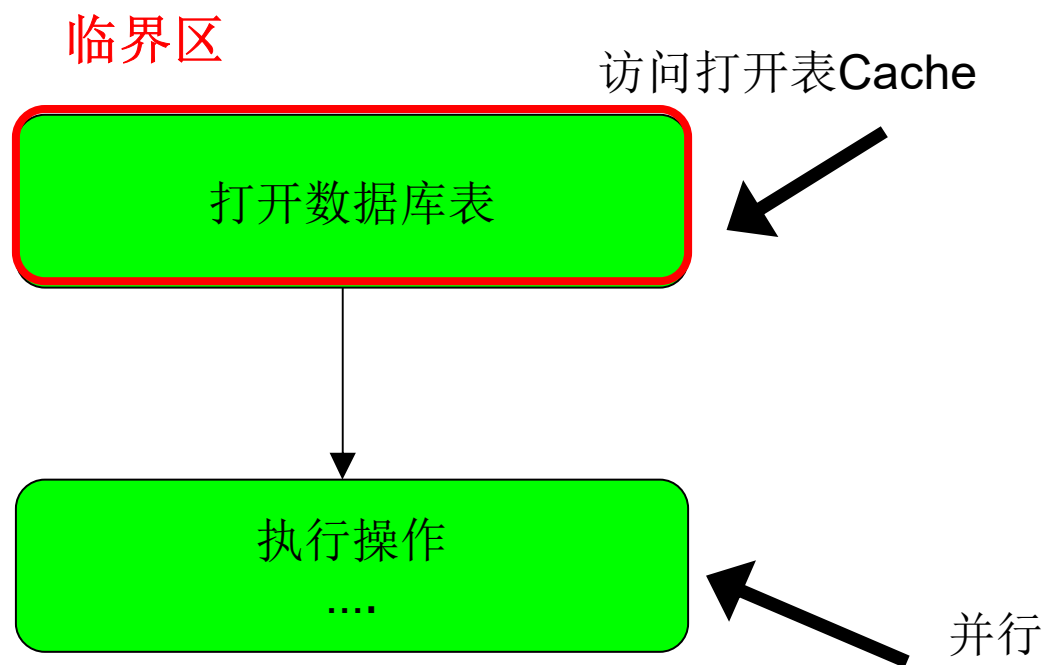
- Amdahl, “[Validity of the single processor approach to achieving large scale computing capabilities](#),” AFIPS 1967.
- 最大加速比受限于串行部分: 串行瓶颈
- 并行部分通常不完美
  - 同步开销 (比如, 对共享数据的更新)
  - 负载不均衡开销 (并行化不完美)
  - 资源共享开销 (N个处理器之间的竞争)

# 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

# MySQL的例子



# 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

# “大” vs. “小” 核

---

大核

- 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
- .....

小核

- 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

大核的能效低:  
比如, 4x的面积(功率)只能获得2x的性能

# 大核 vs. 小核

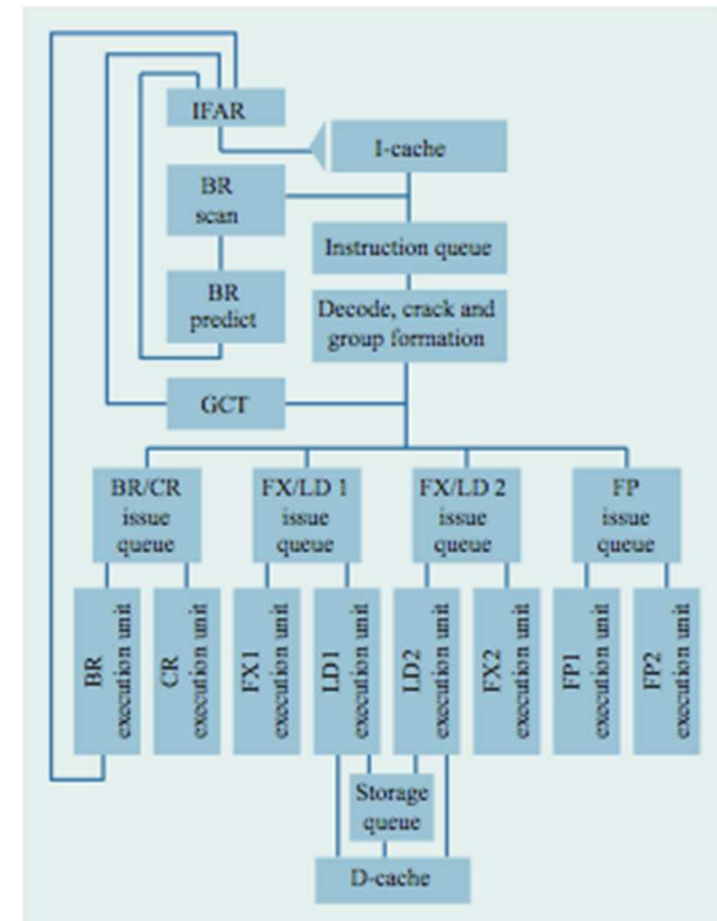
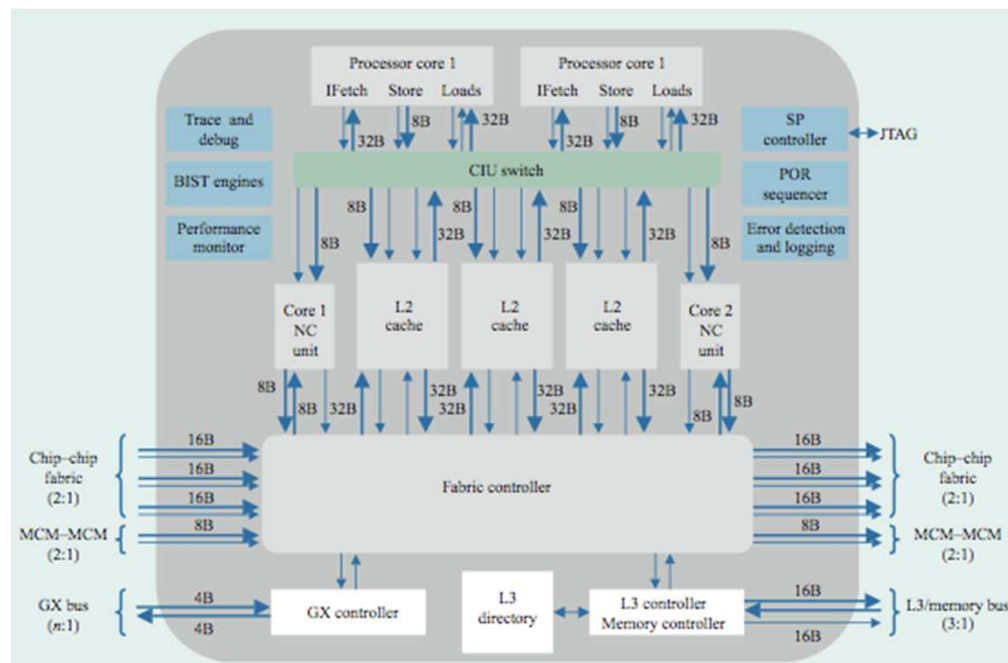
---

- 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

# 大核: IBM POWER4

- Tandler 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



# 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



# IBM POWER5

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

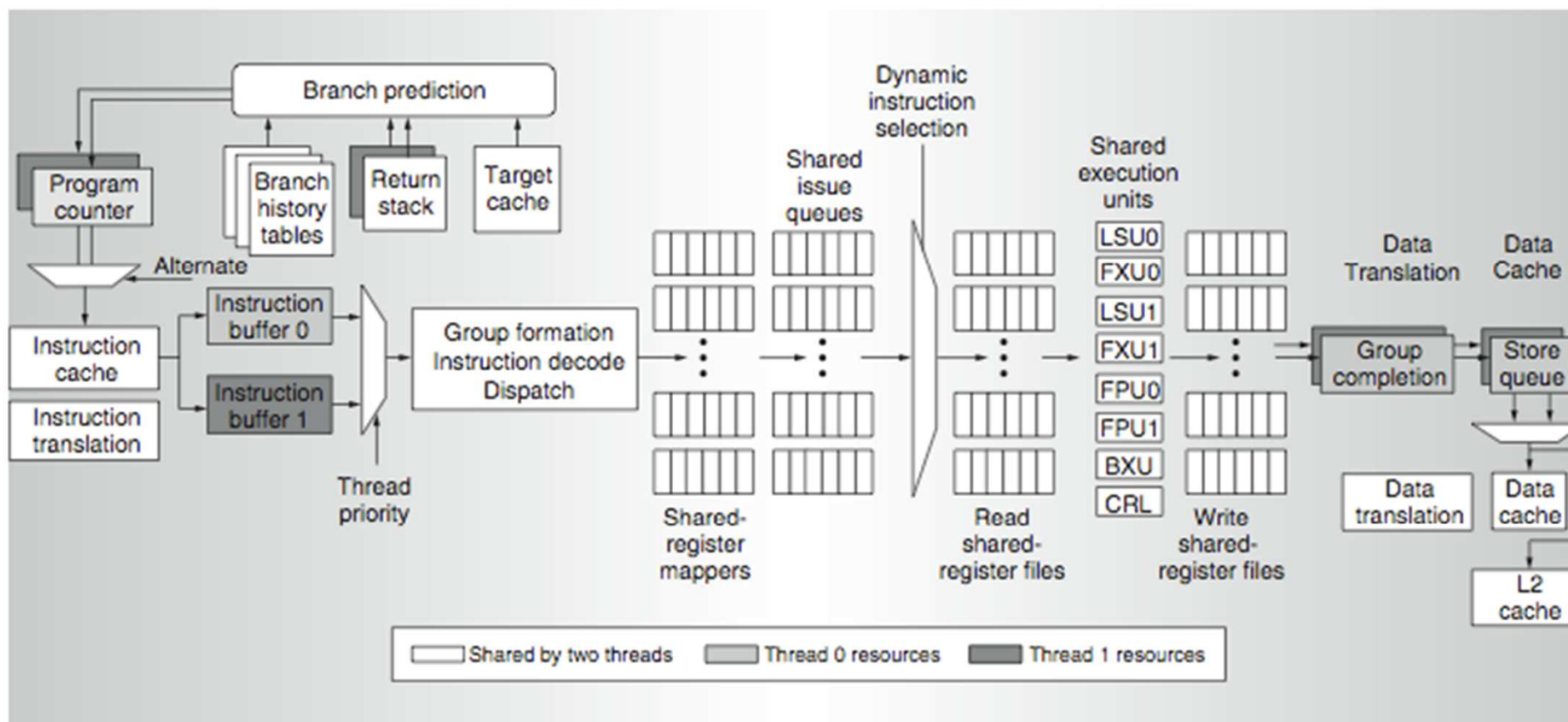
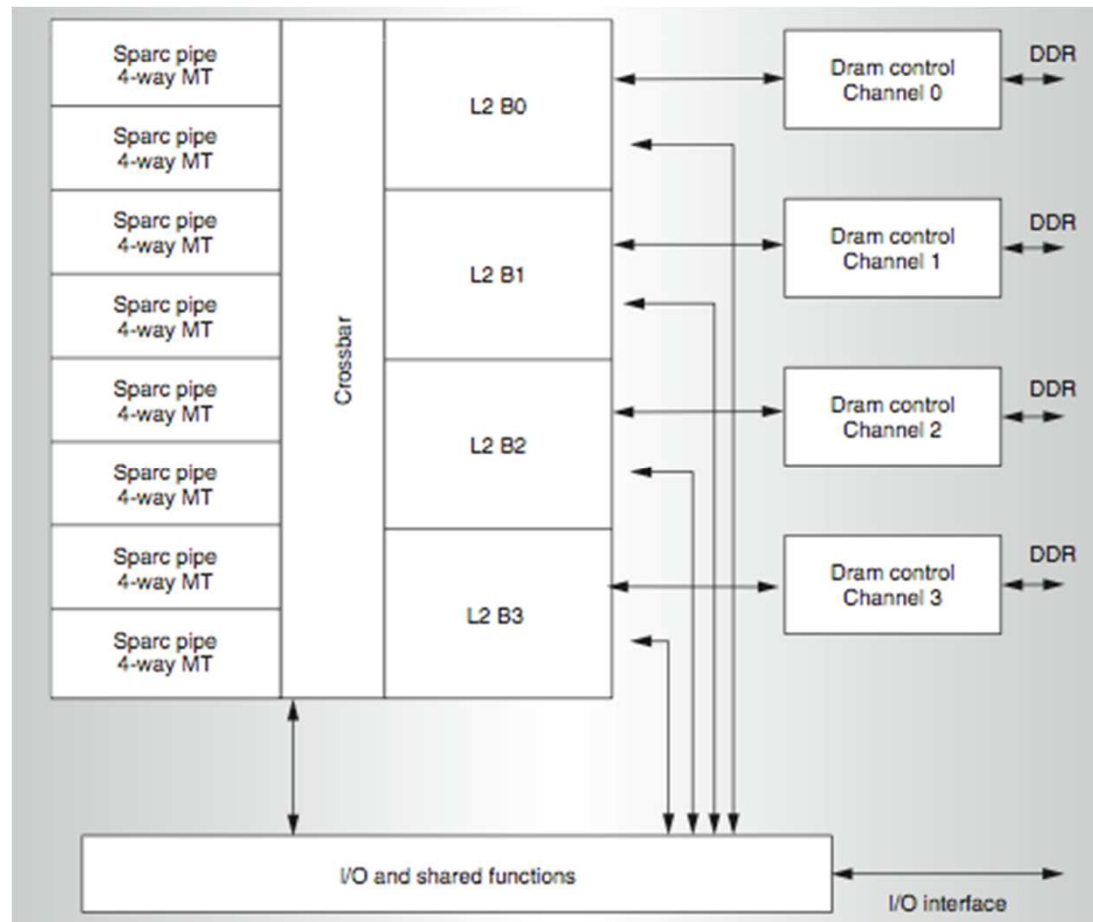


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

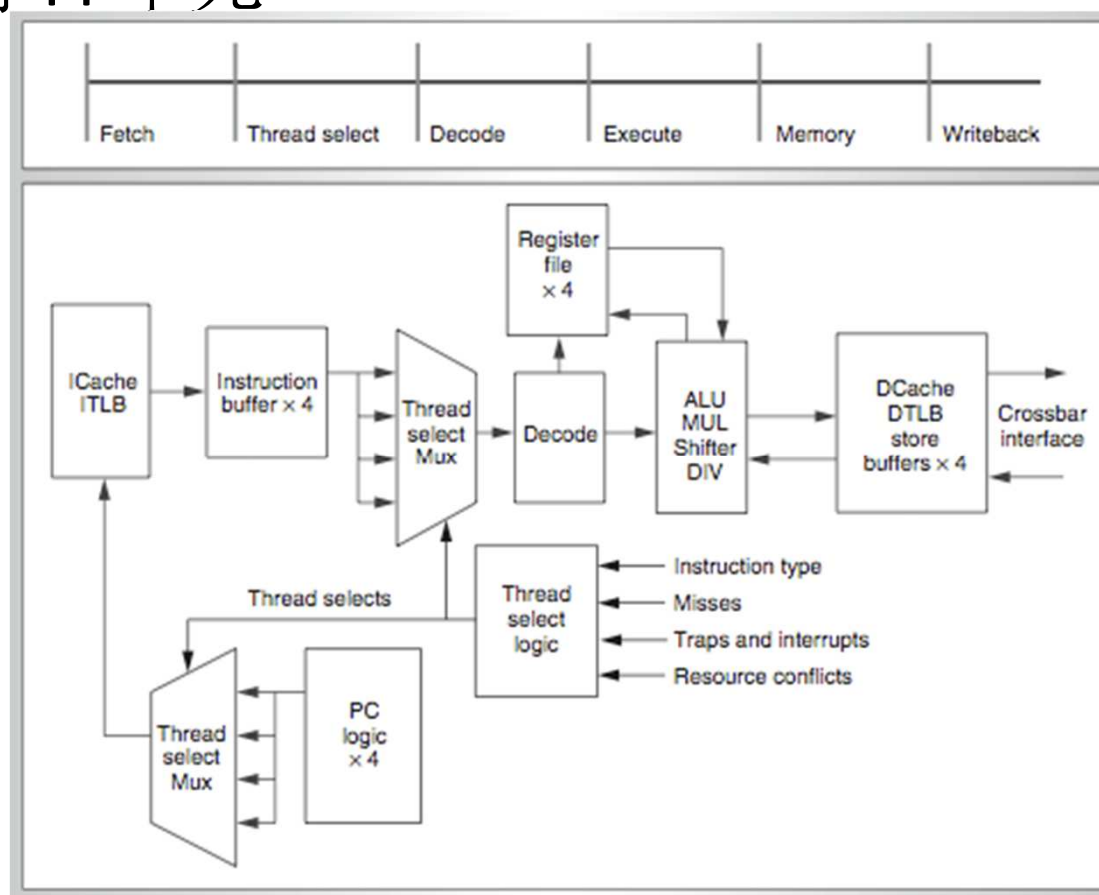
# 小核: Sun Niagara (UltraSPARC T1)

- Kongetira et al., “[Niagara: A 32-Way Multithreaded SPARC Processor](#),” IEEE Micro 2005.



# 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



# 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？

# 性能 vs. 并行性

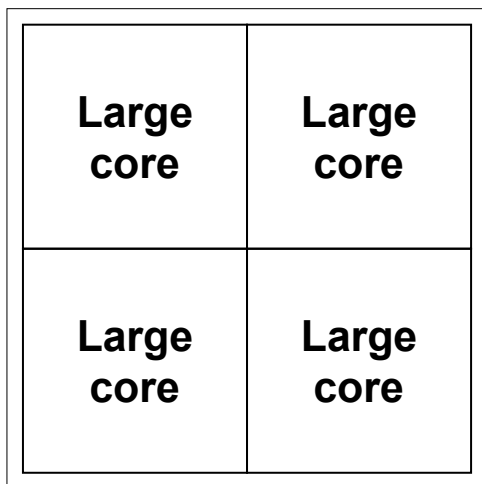
---

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

# 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

# 铺砌小核

---

Small core	<b>Small core</b>	Small core	<b>Small core</b>
<b>Small core</b>	Small core	<b>Small core</b>	Small core
Small core	<b>Small core</b>	Small core	<b>Small core</b>
<b>Small core</b>	Small core	<b>Small core</b>	Small core

“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

# 两全其美?

---

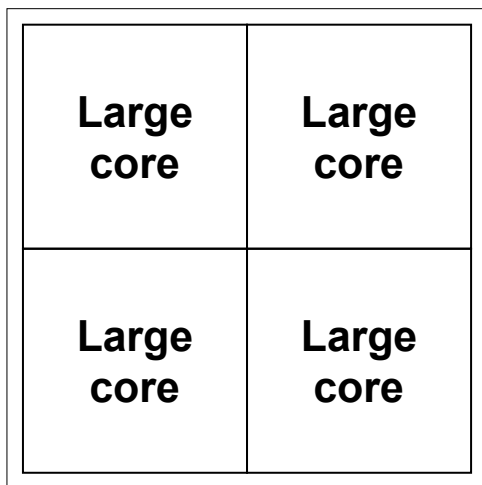
- 铺砌大核
  - + 单线程、串行代码段时可获得高性能
  - 并行程序段时吞吐率低
- 铺砌小核
  - + 并行部分吞吐率高
  - 串行部分、单线程性能低, 相比现有单线程处理器性能还差
- 思路: 在一个芯片上同时集成大核和小核 → 性能不对称



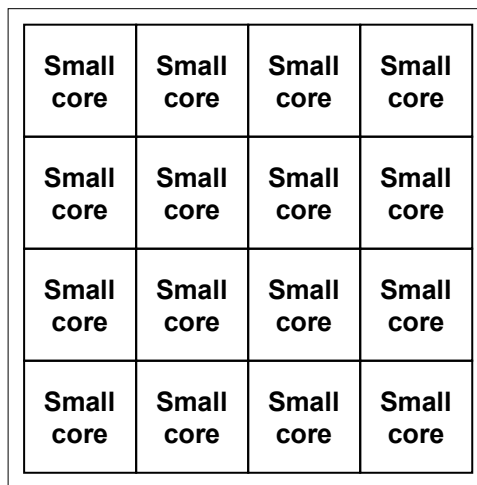
---

# 非对称多核

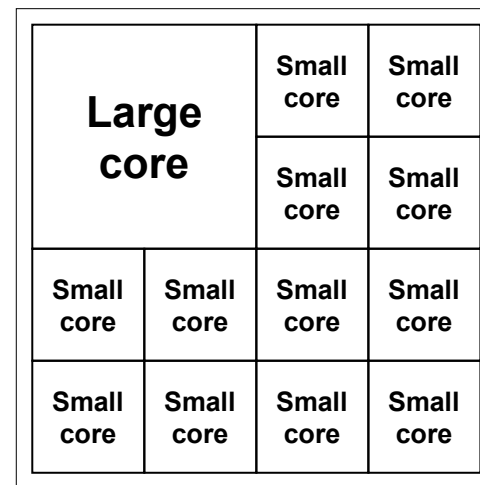
# 非对称片上多处理器(ACMP)



“铺砌大核”



“铺砌小核”



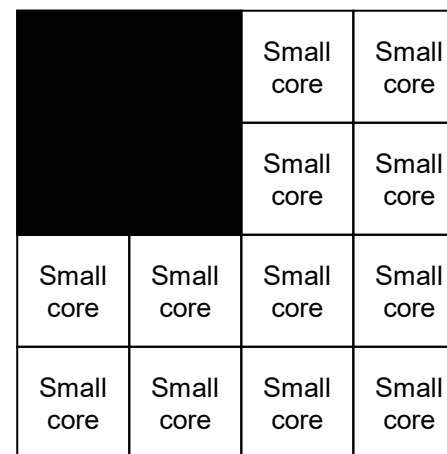
ACMP

- 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

# 加速串行瓶颈

---

单线程 → 大核

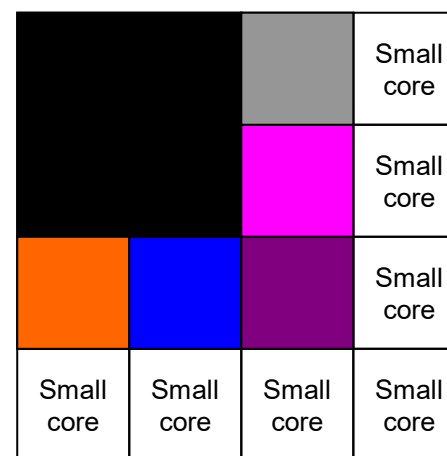
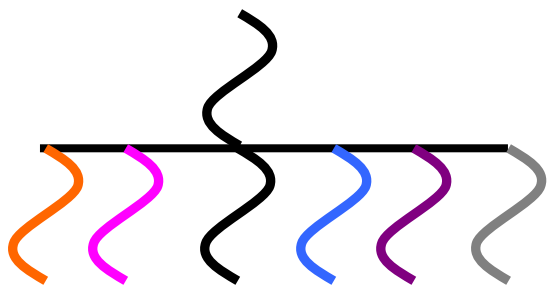


ACMP

# 加速串行瓶颈

---

单线程 → 大核

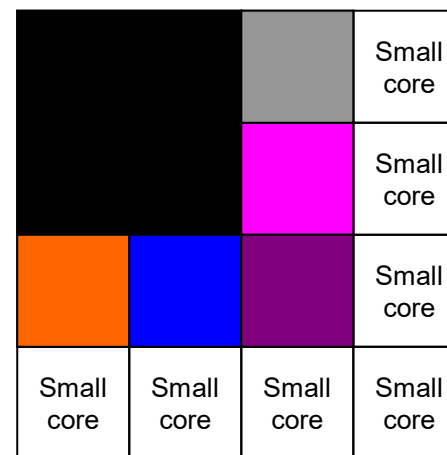
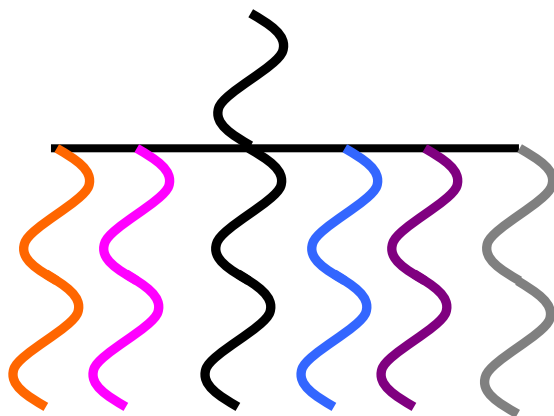


ACMP

# 加速串行瓶颈

---

单线程 → 大核

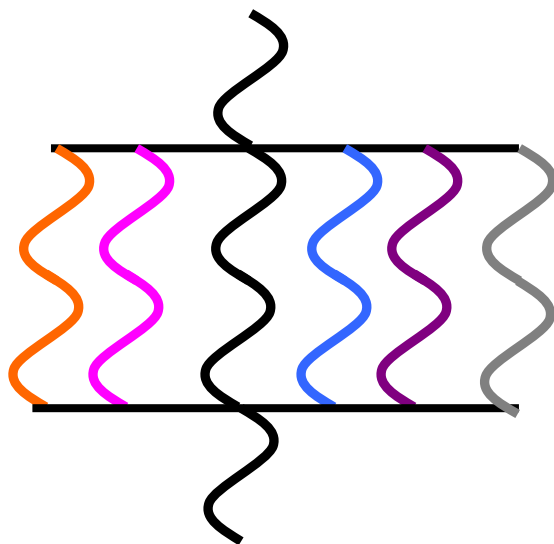


ACMP

# 加速串行瓶颈

---

单线程 → 大核



		Small core	Small core
		Small core	Small core
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core

ACMP

# 性能 vs. 并行性

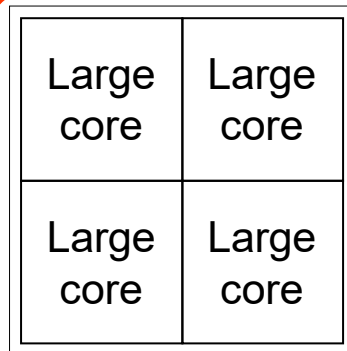
---

假设:

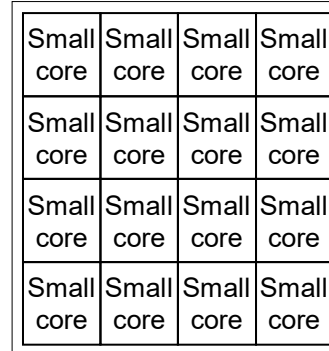
1. 小核用1个面积的预算获得1份性能
2. 大核用4个面积的预算获得2份性能

# ACMP 性能 vs. 并行性

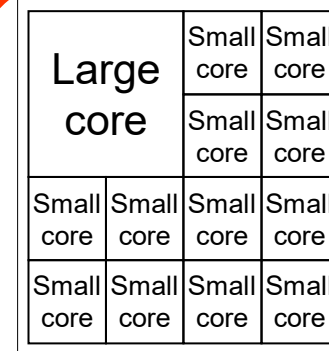
面积预算 = 16 个小核



“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	$2 \times 4 = 8$	$1 \times 16 = 16$	$1 \times 2 + 1 \times 12 = 14$



# 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)

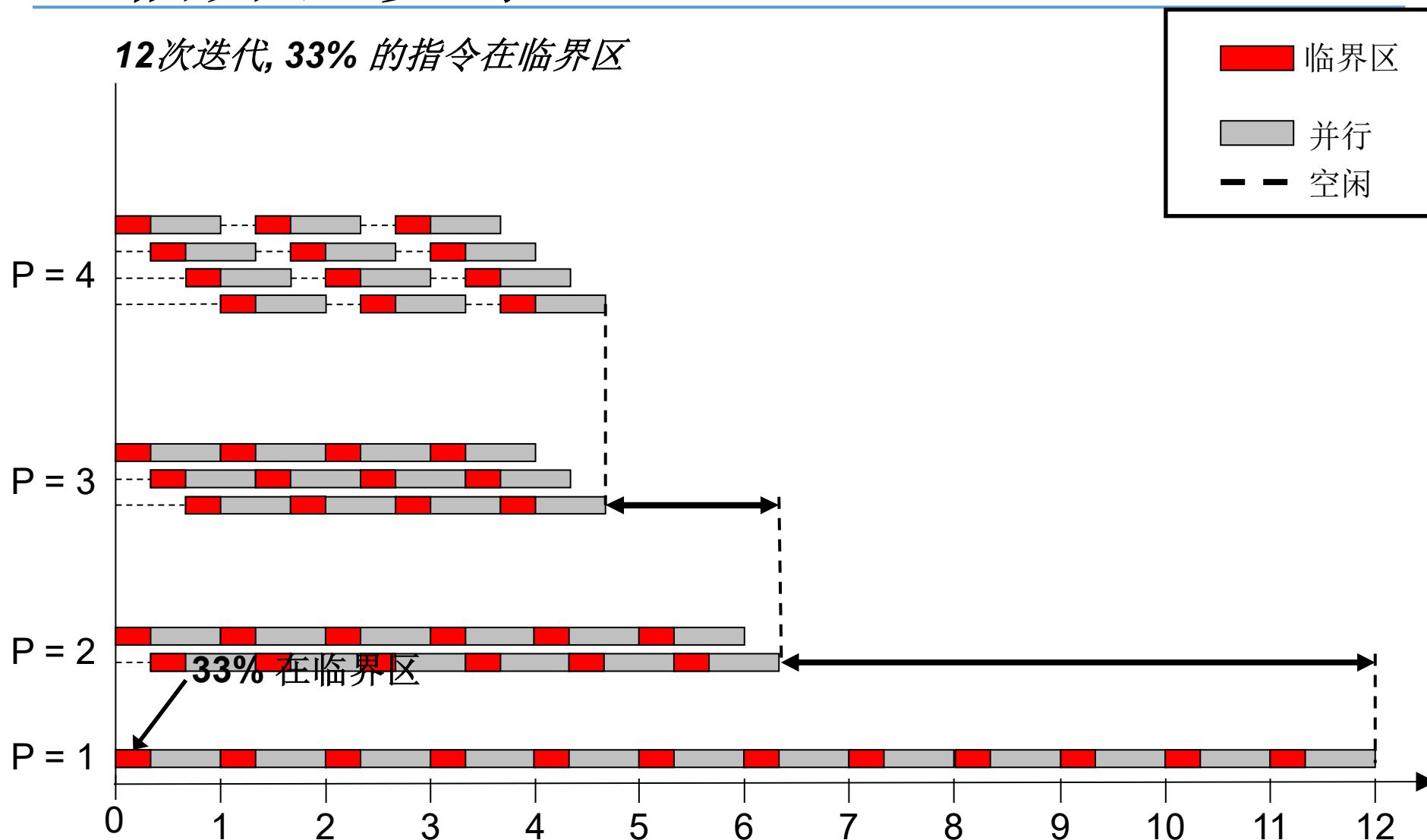
# 加速并行瓶颈

---

- 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

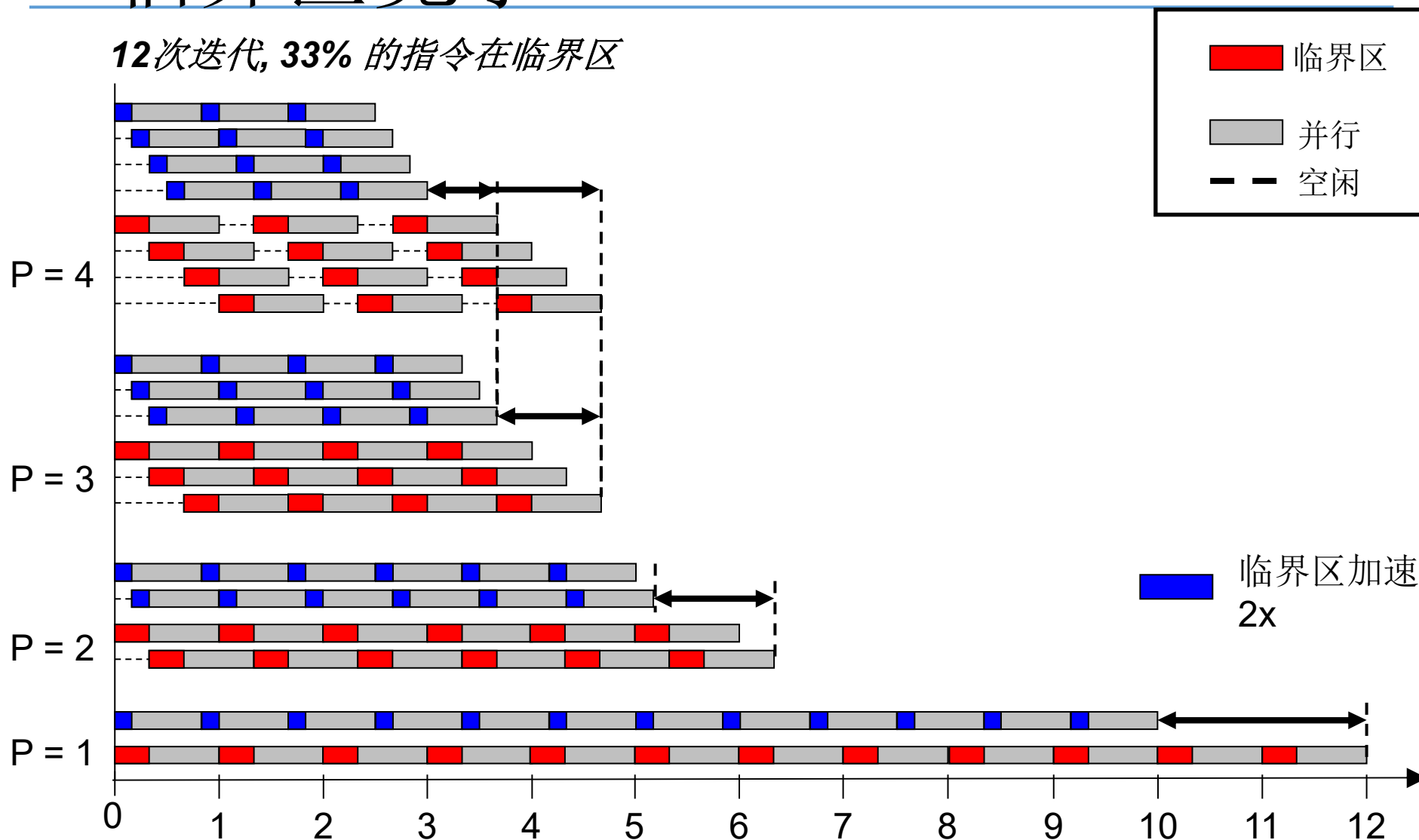
# 临界区竞争

12次迭代, 33% 的指令在临界区



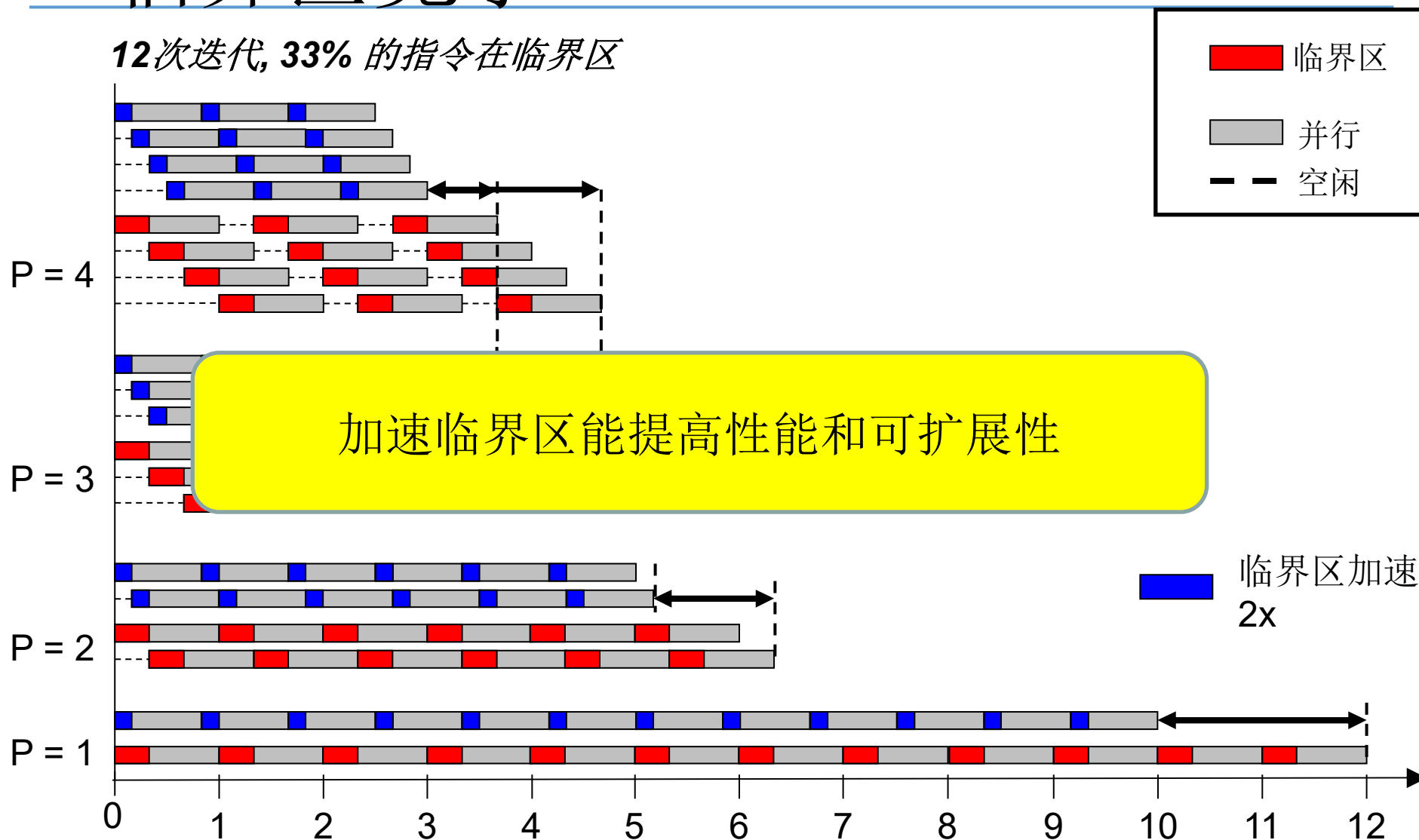
# 临界区竞争

12次迭代, 33% 的指令在临界区



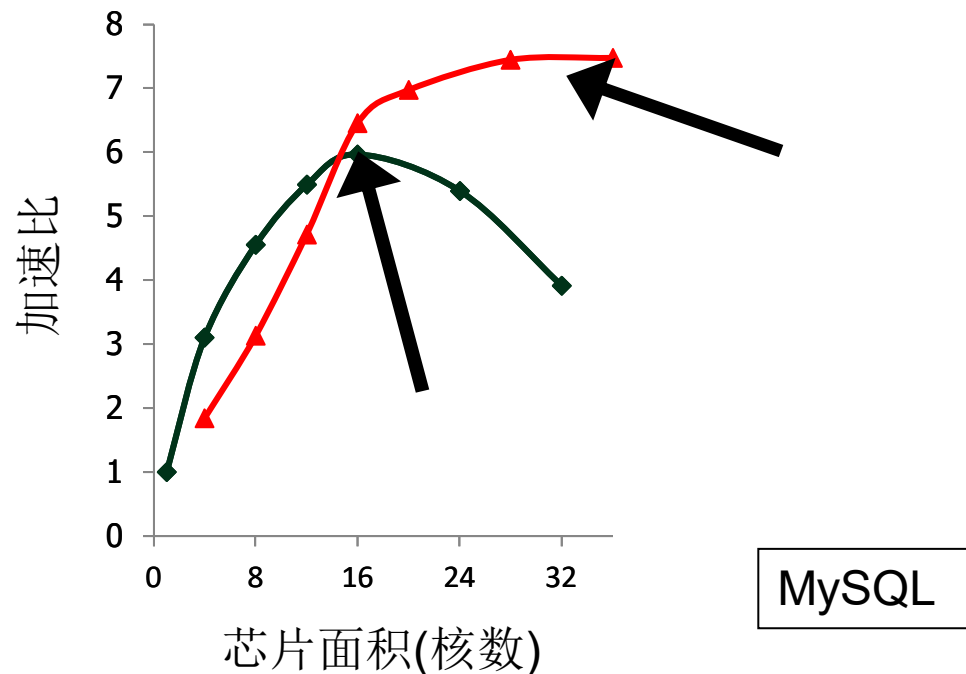
# 临界区竞争

12次迭代, 33% 的指令在临界区



# 临界区对可扩展性的影响

- 临界区竞争导致并行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



# 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

# 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.



# 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

# 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

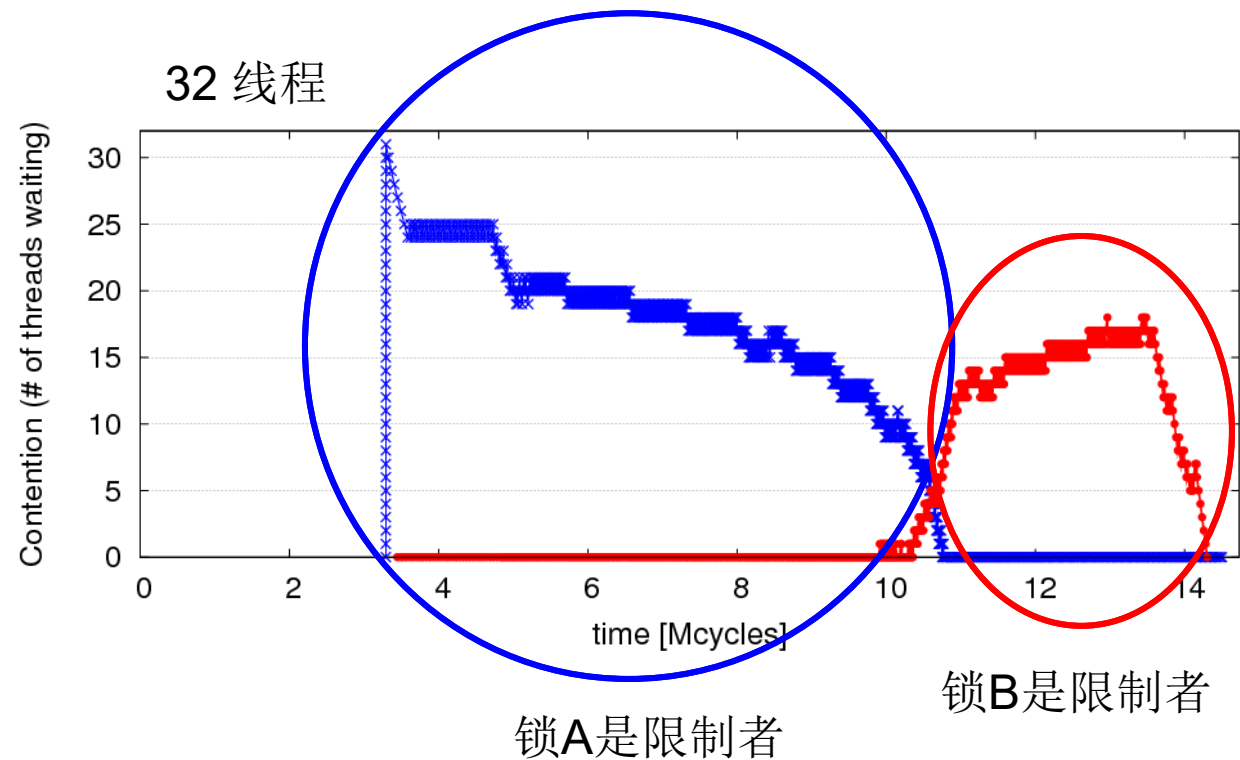
Lock B

Traverse list B

Insert X into B

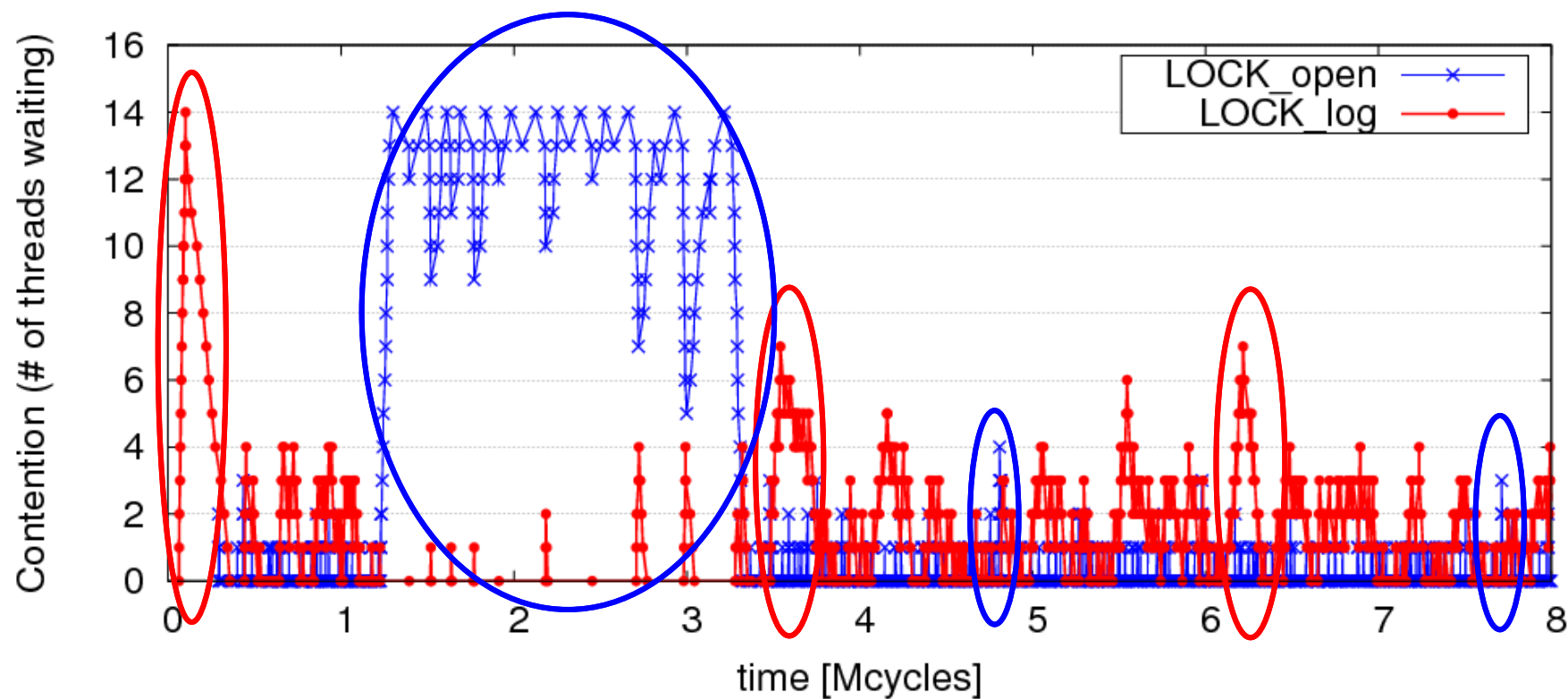
Unlock B

until A is empty



# 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



# 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.

# 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.

---

# 多核系统中的内存干扰和调度

# 单核系统的调度策略

---

- **FR-FCFS** (行缓冲优先)

1. 行命中的优先

2. 最旧的优先

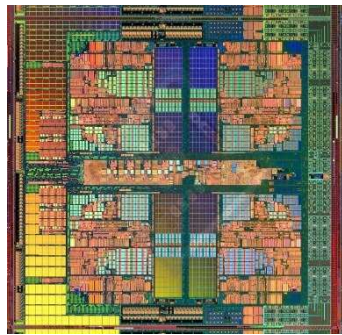
目的1: 最大化行缓冲命中率 → 最大化**DRAM**吞吐量

目的2: 优先考虑旧的请求 → 确保向前推进

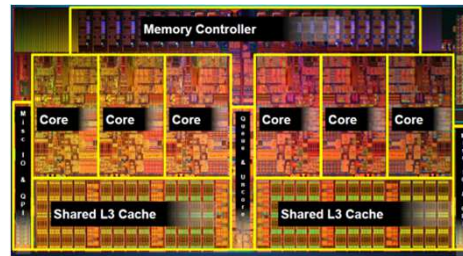
- 在多核系统中这还是个好的策略吗?

# 趋势: 片上众核(Many Core)

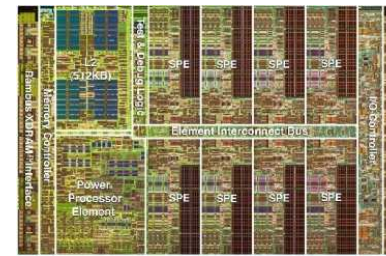
- 比一个大核更简单，功率更低
- 片上大规模并行



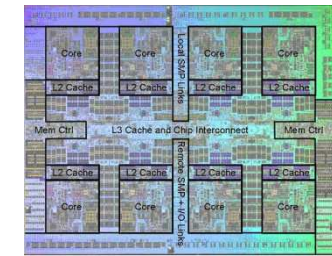
AMD Barcelona  
4 cores



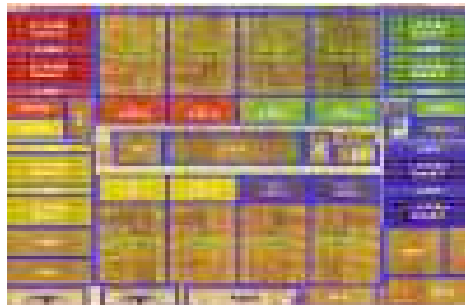
Intel Core i7  
8 cores



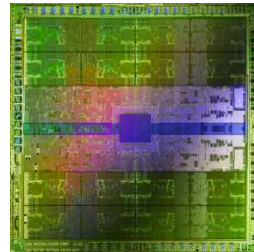
IBM Cell BE  
8+1 cores



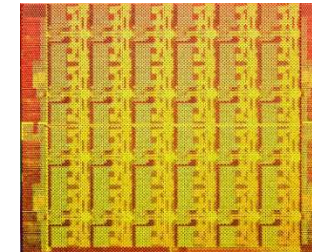
IBM POWER7  
8 cores



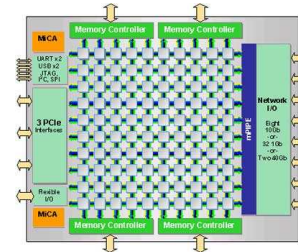
Sun Niagara II  
8 cores



Nvidia Fermi  
448 "cores"



Intel SCC  
48 cores, networked



Tiler TILE Gx  
100 cores, networked

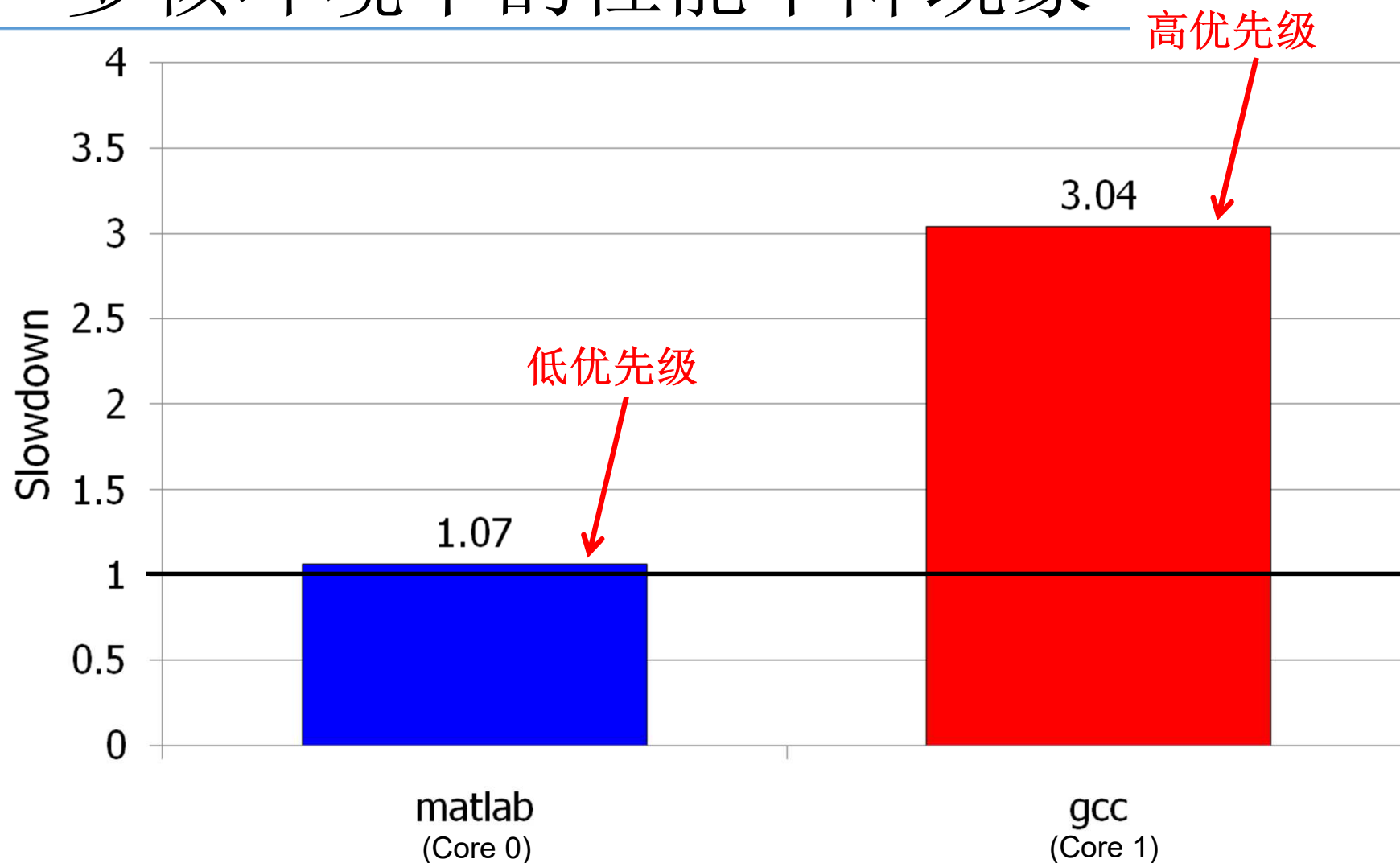


# 片上众核

---

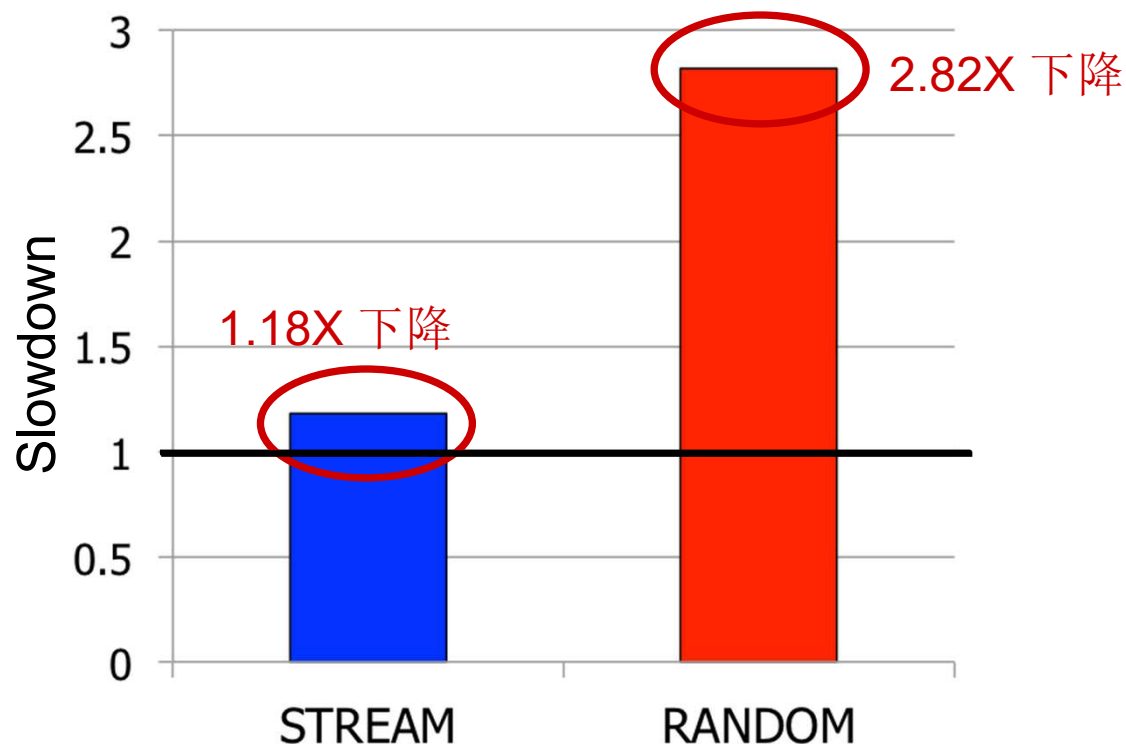
- 我们想要:
  - 用N倍的核获得N倍的系统性能
- 我们得到了什么?

# 多核环境下的性能下降现象



Moscibroda and Mutlu, “[Memory performance attacks: Denial of memory service in multi-core systems](#),” USENIX Security 2007.

# 内存性能抢占现象的影响

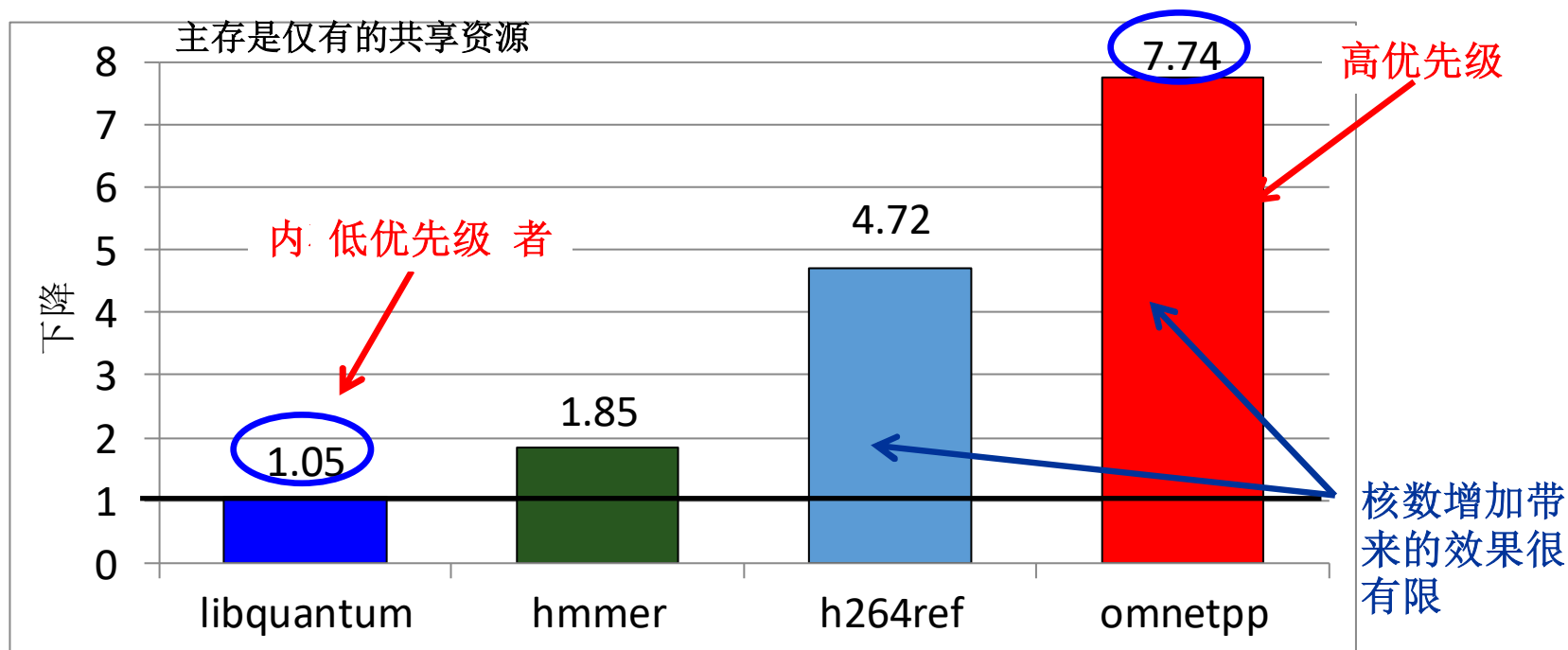


Intel Pentium D, Windows XP

(Intel Core Duo, AMD Turion, Fedora Linux, 结果类似)

Moscibroda and Mutlu, “[Memory performance attacks: Denial of memory service in multi-core systems](#),” USENIX Security 2007.

# 干扰不受控导致的问题



- 不同线程性能的下**降不公平**
- 系统性能低
- 拒绝服务的漏洞
- 优先级反转: 无法保证按优先级/SLA执行
- 糟糕的性能可预测性 (无性能隔离)

无法控制、不可预料系统

# 内存中的线程间干扰

---

- 内存控制器、管脚、内存Bank是共享的
- 管脚带宽的增加不像核数增加的那样快
  - 每核的带宽在减小
- 不同核上执行的不同线程在主存系统中会互相干扰
- 线程间由于资源竞争互相延迟：
  - Bank, 总线, 行缓冲 冲突 → 减小了 DRAM 吞吐量
- 线程还会破坏彼此的DRAM Bank访问的并行性

# DRAM中线程间干扰的影响

---

- 排队/争用导致延迟
  - Bank 冲突, 总线冲突, 通道冲突, ...
- 由于**DRAM**的约束导致额外的延迟
  - 称为 “协议开销”
  - 比如
    - 行冲突
    - 读-写和写-读延迟
- 线程内并行的丧失
  - 一个线程的并发请求被串行处理

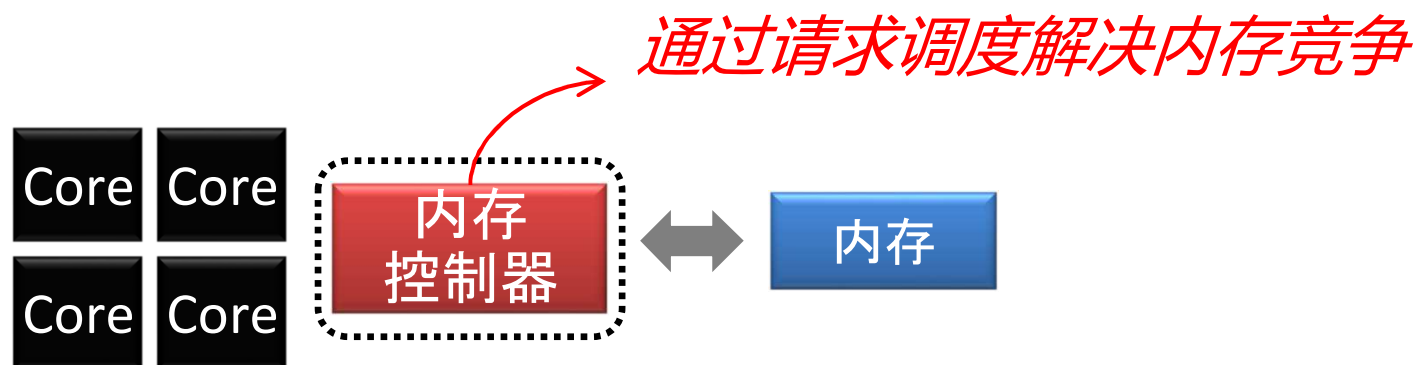
# 问题: 无法感知QoS的内存控制

---

- 现有的**DRAM**控制器无法感知**DRAM**系统中线程间的干扰
- 目标只是最大化**DRAM**的吞吐量
  - 不能感知线程, 也存在线程间的不公平
  - 无法并行地响应线程的请求
  - **FR-FCFS** 策略: 1) 行命中优先, 2) 最旧优先
    - 行缓冲的高局部性导致线程优先级的不公平
    - 受益的线程往往是内存密集型的 (大量的访存)

# 解决方案: QoS感知的内存请求调度

---



- 如何通过请求调度获得
  - 高系统性能
  - 应用的高公平性
  - 系统软件的可配置性
- 内存控制器需要能够感知线程



# 一些内存调度方法

---

- O. Mutlu et.al., “[Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors](#)”, 40th International Symposium on Microarchitecture (MICRO), pages 146-158, Chicago, IL, December 2007
- O. Mutlu et.al., “[Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems](#)”, 35th International Symposium on Computer Architecture (ISCA), pages 63-74, Beijing, China, June 2008
- Y. Kim et.al., “[Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior](#)”, 43rd International Symposium on Microarchitecture (MICRO), pages 65-76, Atlanta, GA, December 2010
- Y. Kim et.al., “[ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers](#)”, 16th International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, January 2010

---

# 其它处理干扰的方法

# 基本的干扰控制技术

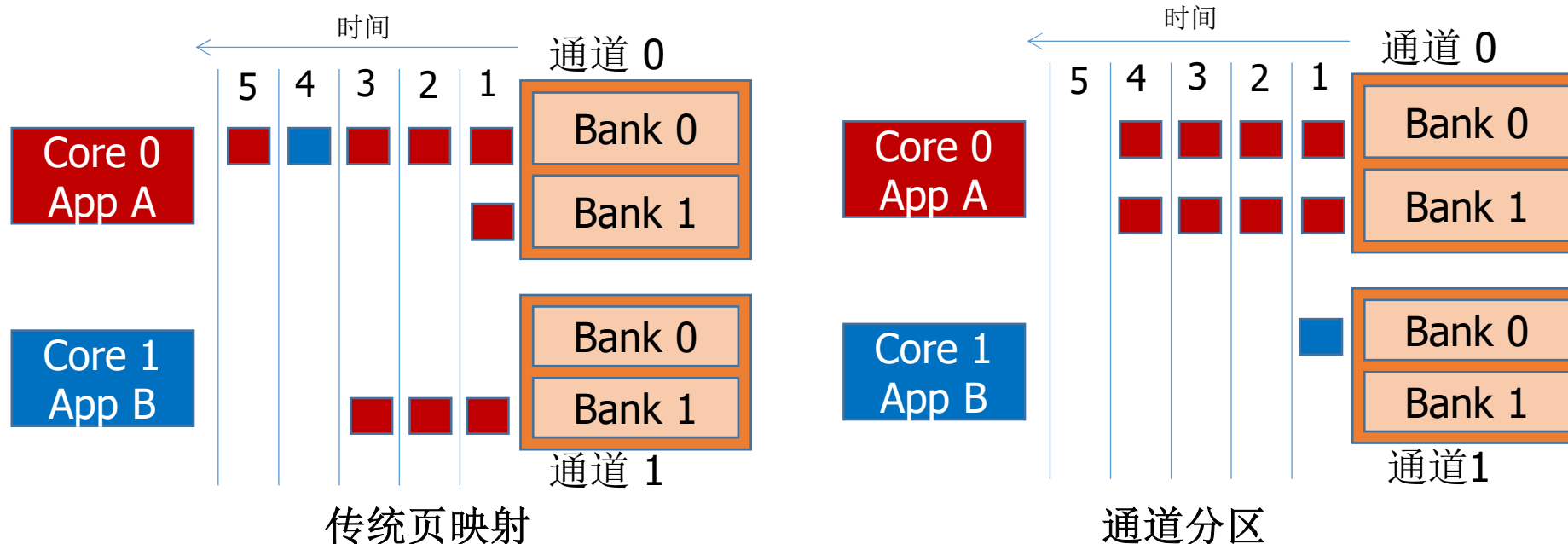
---

- 目标: 减少/控制干扰
  1. 优先级或请求调度
  2. 数据映射到Bank/通道/Rank
  3. 核/源调节
  4. 应用/线程调度

# 内存通道分区

- 内存通道分区

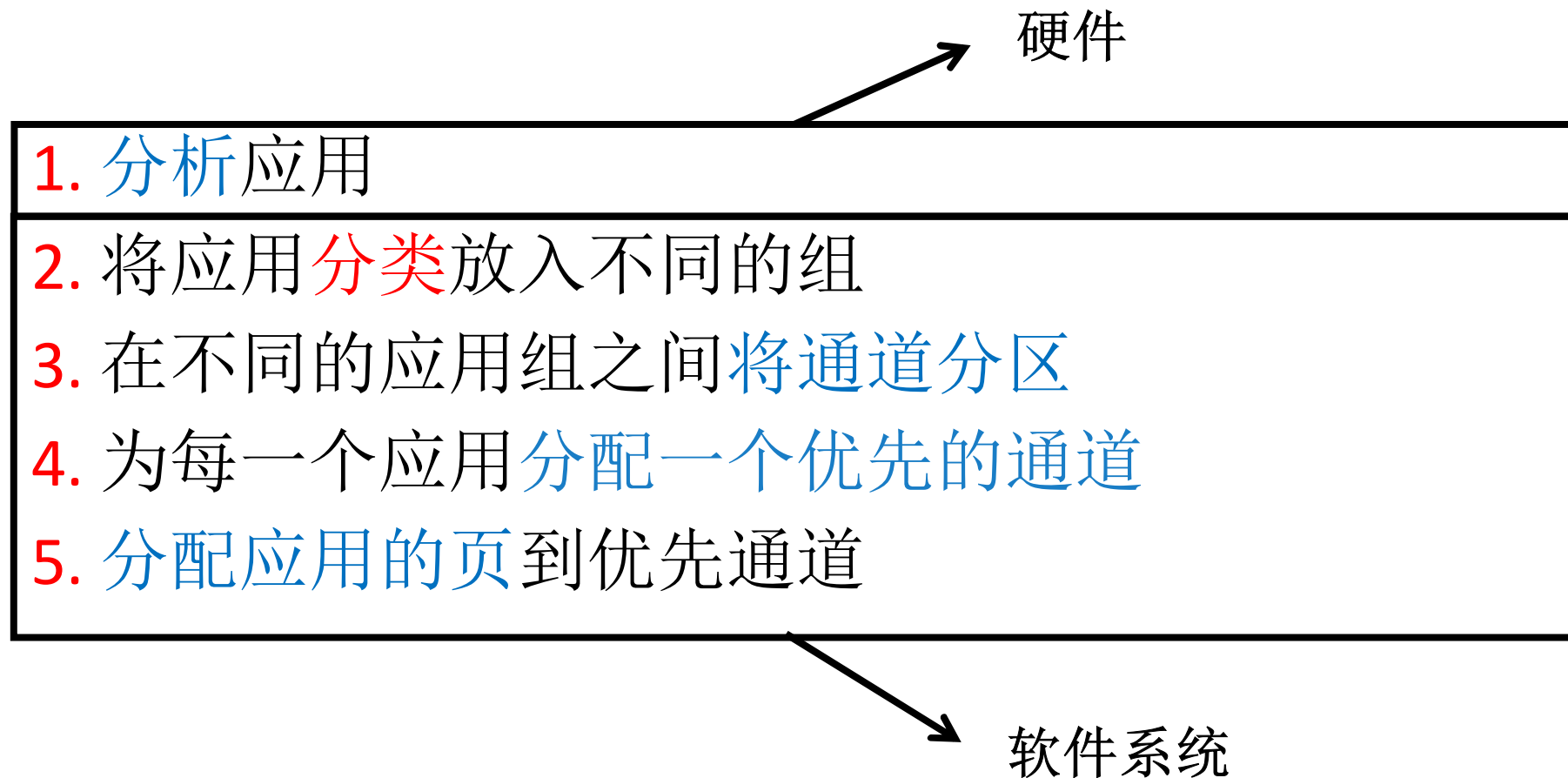
- 思路: 把干扰严重的应用的页映射到不同的通道[Muralidhara+, MICRO'11]



- 分离具有不同行局部性和内存密集程度的应用
- 对于减少具有“中等”和“重度”内存密集的线程的干扰尤其有效

# 内存通道分区(MCP)机制

---



# 观察

---

- 内存密集程度非常低的应用很少访存 → 为它们分配通道会导致宝贵的内存带宽的浪费
- 它们非常可能使它们所在的核持续繁忙 → 真的应该给它们高的优先级
- 它们极少与其它应用发生干扰 → 给它们高优先级不会损害其它应用

## 集成的内存分区和调度(IMPS)

---

- 在内存调度时总是给内存密集程度很低的应用以高优先级
- 使用内存通道分区减少其它应用之间的干扰

# 基本的干扰控制技术

---

- 目标: 减少/控制干扰
  1. 优先级或请求调度
  2. 数据映射到Bank/通道/Rank
  3. 核/源调节
  4. 应用/线程调度



## 另外一种方法: 源调节

---

- 在核(源)上管理线程间干扰, 而不是在共享资源上
- 在内存系统中动态估计公平性
- 将该信息反馈给控制器
- 相应地调节核的访存频率
  - 调节谁调节多少取决于性能目标 (吞吐量, 公平性, 每个线程的QoS, 等等)
  - 比如, 如果不公平性 > 系统软件指定的目标则调低导致不公平的核的访存频率 & 调高被不公平对待的核的访存频率
- Ebrahimi et al., “Fairness via Source Throttling,” ASPLOS’10, TOCS’12.

# 核 (源) 调节

---

- 思路: 估计由干扰造成的性能下降, 调低“肇事”线程的访存量
  - Ebrahimi et al., “Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems,” ASPLOS 2010.
- 优点
  - + 核/请求的调节容易实现: 不需要改变内存调度算法
  - + 是一种处理共享资源竞争的通用方法
- 缺点
  - 需要估计干扰/性能下降
  - 阈值优化会比较困难 → 吞吐量下降

# 基本的干扰控制技术

---

- 目标: 减少/控制干扰

1. 优先级或请求调度

2. 数据映射到Bank/通道/Rank

3. 核/源调节

4. 应用/线程调度

思路: 选择互相干扰不严重的线程一起调度到核上  
共享内存系统

# 在并行应用中处理干扰

---

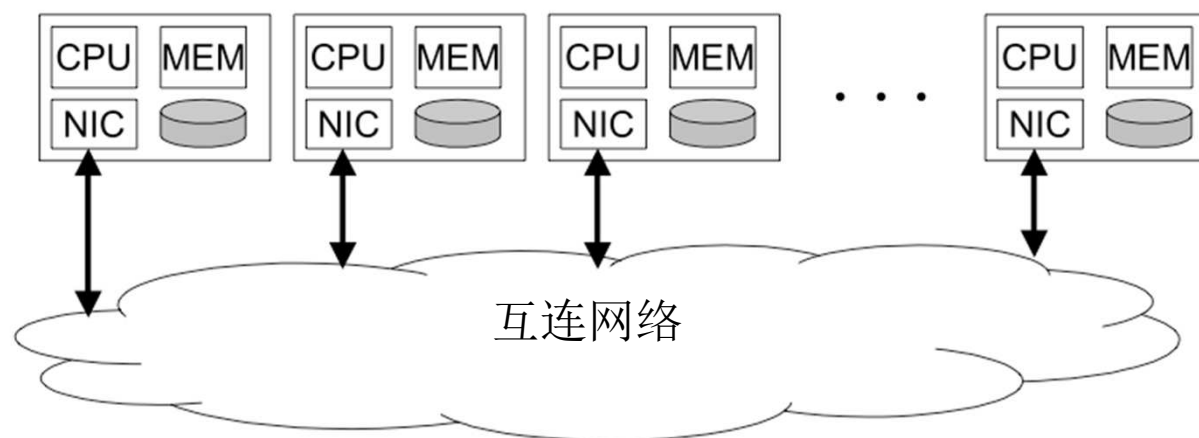
- 多线程应用中的线程是相互依赖的
  - 由于同步的原因某些线程会在关键路径上执行，有些线程不会
  - 如何调度相互依赖的线程的请求，才能最大化多线程应用的性能？
- 
- 思路: 估计可能在关键路径上的线程，优先处理它们的请求; 调整非关键线程的优先级以减小它们之间的干扰 [Ebrahimi+, MICRO'11]
  - 硬件/软件协同的关键线程估计

---

# 互连网络

# 哪里需要互连网络?

- 有组件需要互连
- 很多例子
  - 处理器和处理器
  - 处理器和内存(bank)
  - 处理器和cache (bank)
  - Cache和cache
  - I/O 设备



# 为什么这个很重要？

---

- 影响系统的可扩展性
  - 可以构建一个多大的系统？
  - 增加更多的处理器有多容易？
- 影响性能和能效
  - 处理器、cache和内存之间通信有多快？
  - 访存延迟有多大？
  - 通信消耗多少能量？

# 互连网络基本概念

---

- 拓扑
  - 指明组件连接的方式
  - 影响路由、可靠性、吞吐量、延迟
- 路由 (算法)
  - 消息如何从源到目的
  - 静态还是自适应
- 缓冲和流量控制
  - 在互连网络中存储些什么?
    - 完整的包, 分包, 其它?
  - 如何在过载时进行调节?
  - 与路由策略紧耦合



# 拓扑

---

- 总线 (最简单)
- 点到点互连 (理想方式、成本最高)
- 交叉开关 (Crossbar)
- 环
- 树
- Omega
- 超立方
- 网状网(Mesh)
- Torus
- 蝶形
- ...

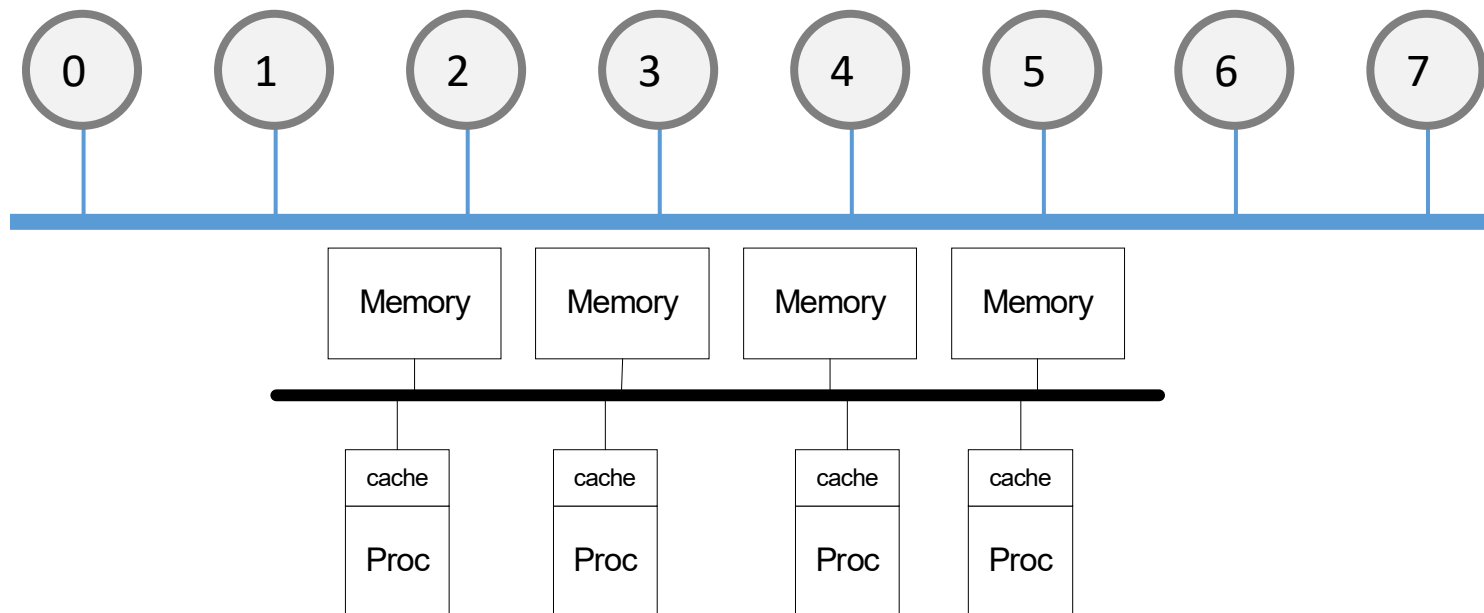
# 评价互连网络的指标

---

- 成本
- 延迟 (按跳, 单位纳秒)
- 竞争度
- 其它需要考虑的指标
  - 能耗
  - 带宽
  - 系统整体性能

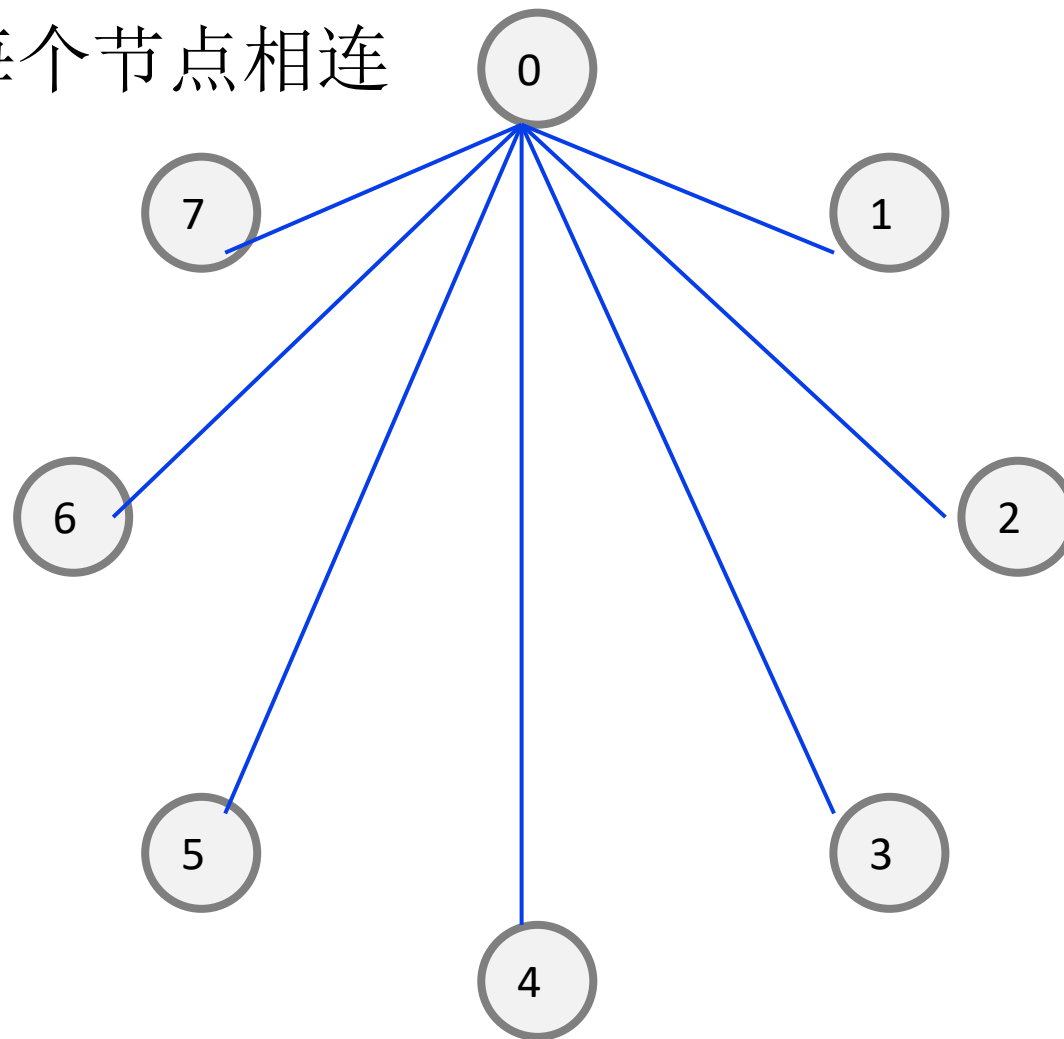
# 总线

- + 简单
- + 节点数量少时成本效率高
- + 很容易实现一致性 (监听和顺序)
- 节点数量大时没有可扩展性 (受限的带宽, 电负载 → 频率降低)
- 高竞争度 → 快饱和



# 点到点

每个节点都与其它的所有节点相连



# 点到点

每个节点都与其它的所有节点相连

- + 竞争度最低
- + 潜在的最低延迟
- + 理想(如果不差钱)

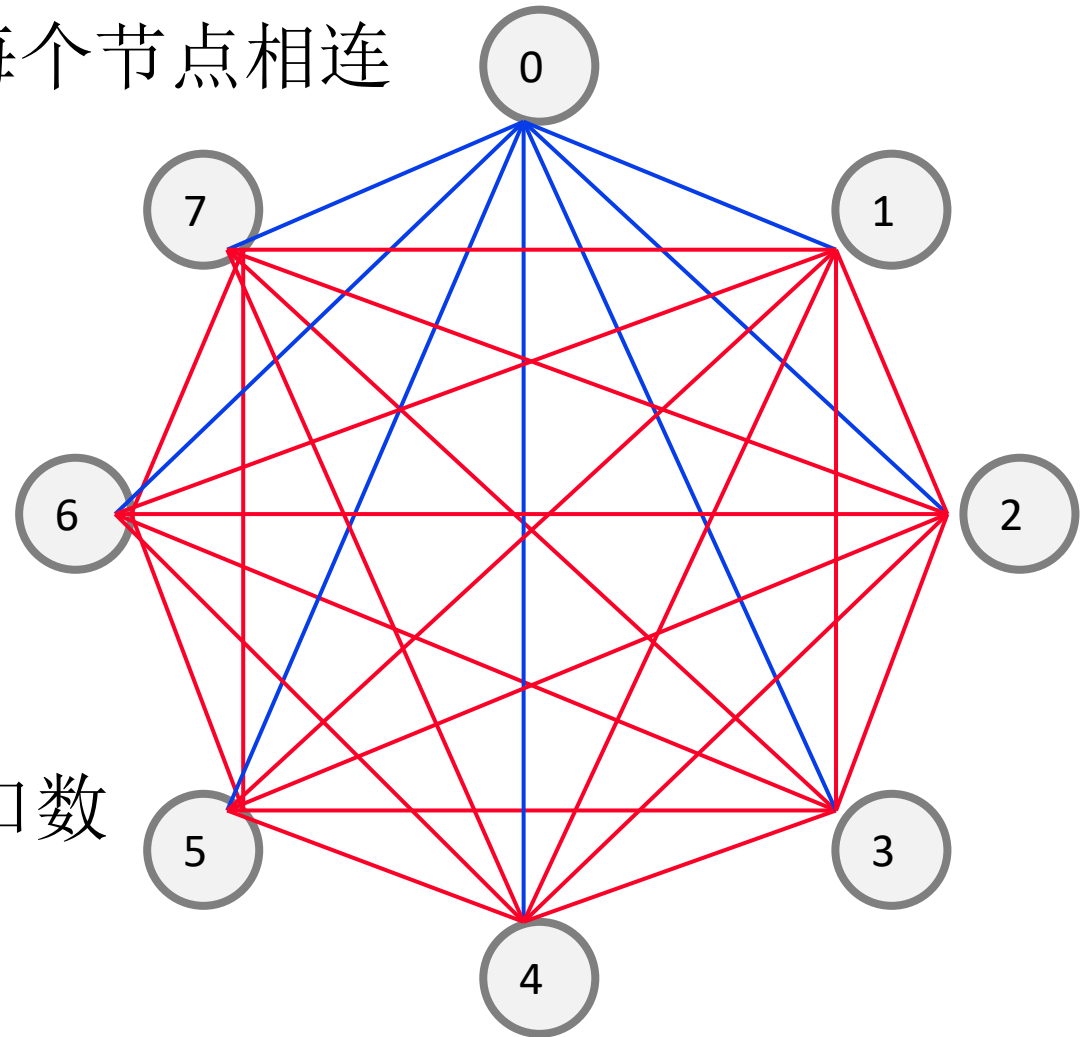
-- 成本最高

$O(N)$  连接/每节点端口数

$O(N^2)$  链路

-- 没有扩展性

-- 在芯片上如何布局?



# 交叉开关(Crossbar)

- 每个节点可连接到任何其它节点 (无阻塞)，不同的是任一节点可随时使用连接
- 允许同时向无冲突的目的节点发送
- 适用于节点数目较小的情况

+ 低延迟、高吞吐

- 昂贵

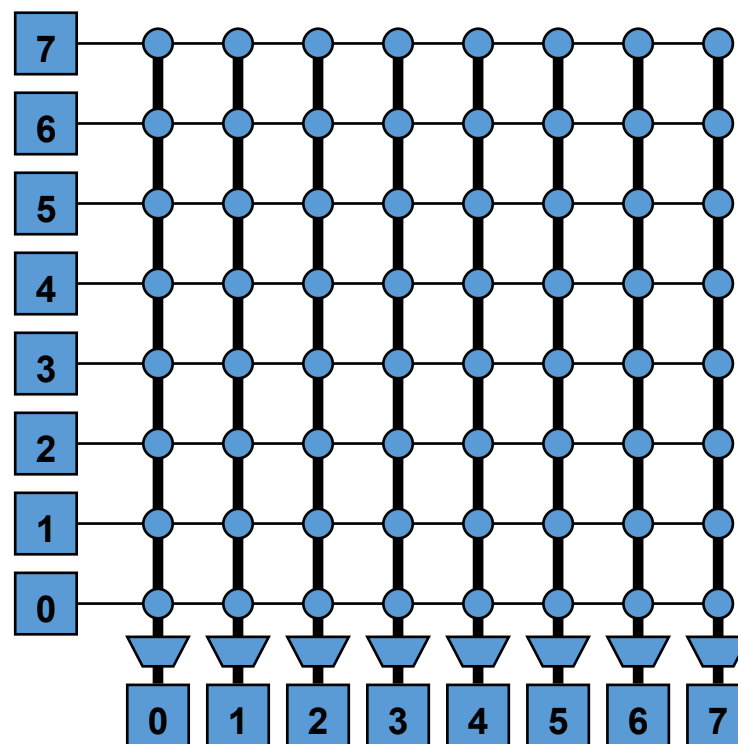
- 无可扩展性  $\rightarrow O(N^2)$  成本

- 随着N的增加仲裁越来越困难

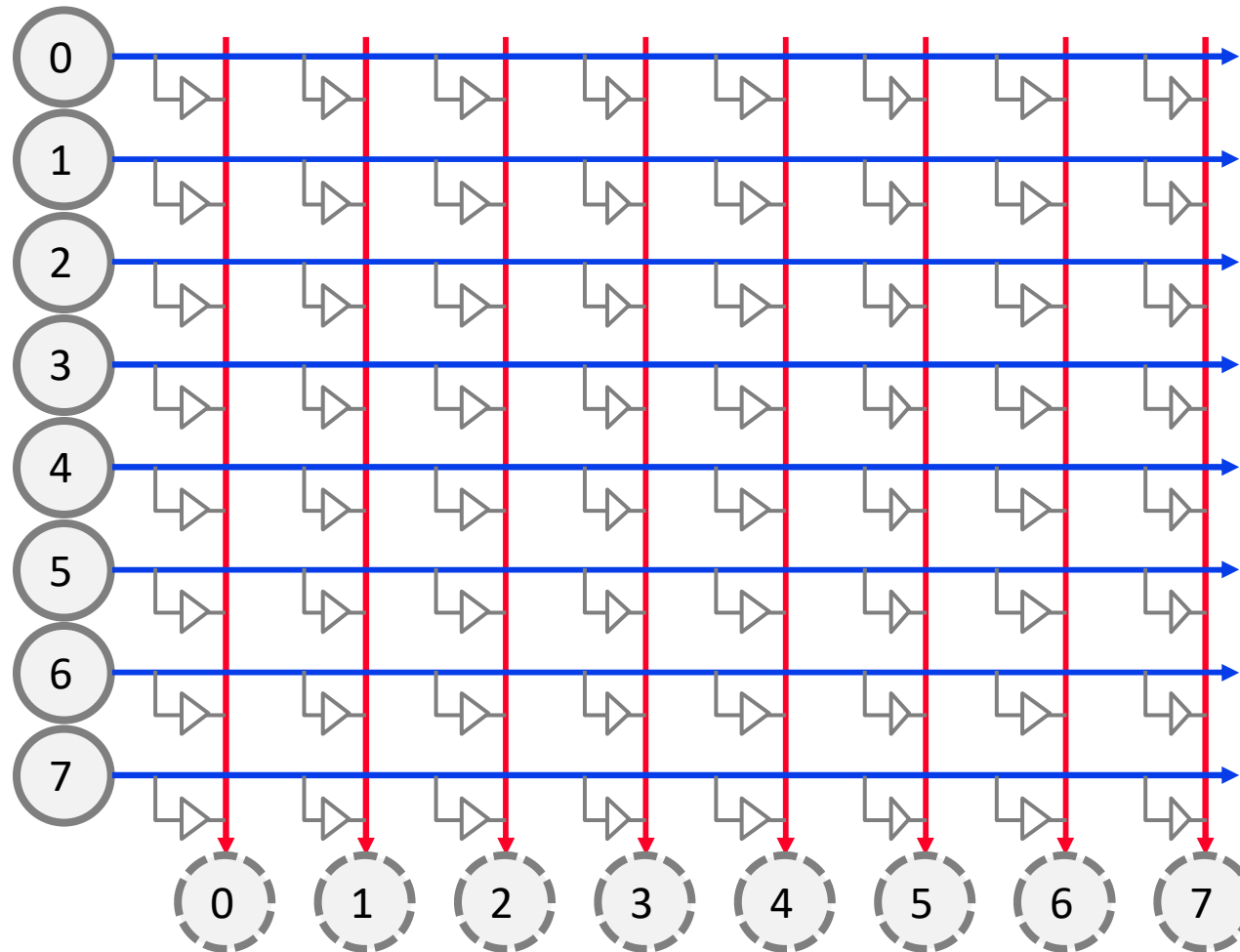
在核到cache到bank中使用

- IBM POWER5

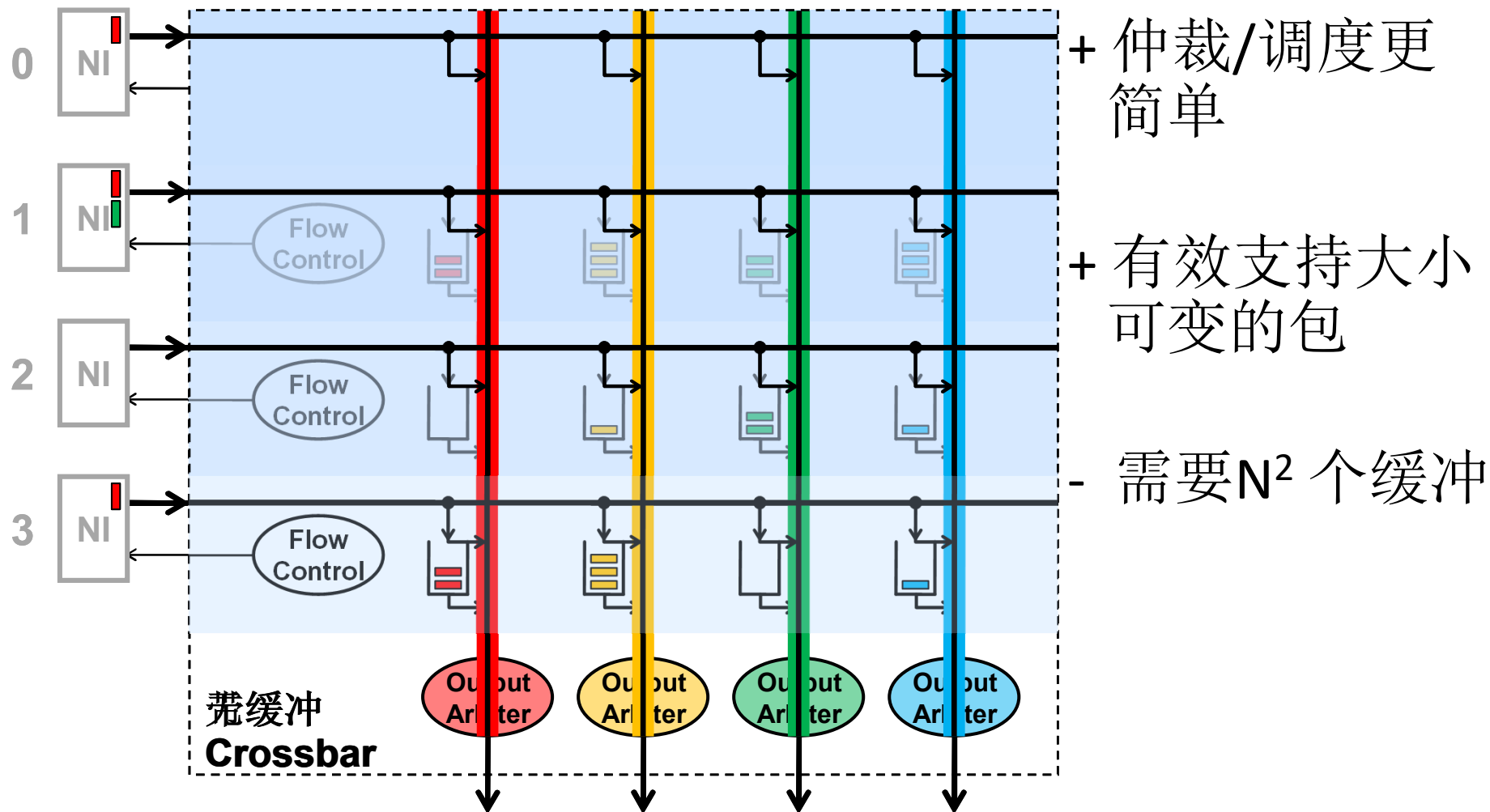
- Sun Niagara I/II



# 另一种 Crossbar 设计



# 带缓冲的 Crossbar





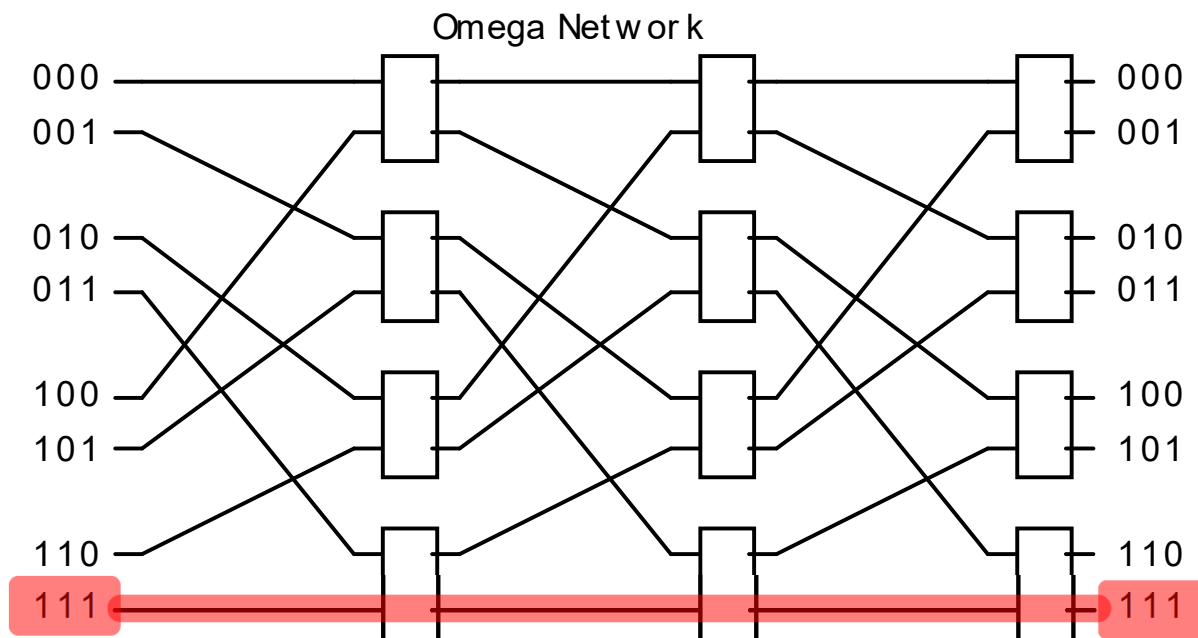
# 能比Crossbar成本更低吗?

---

- 仍能拥有低竞争度?
- 思路: 多阶段网络

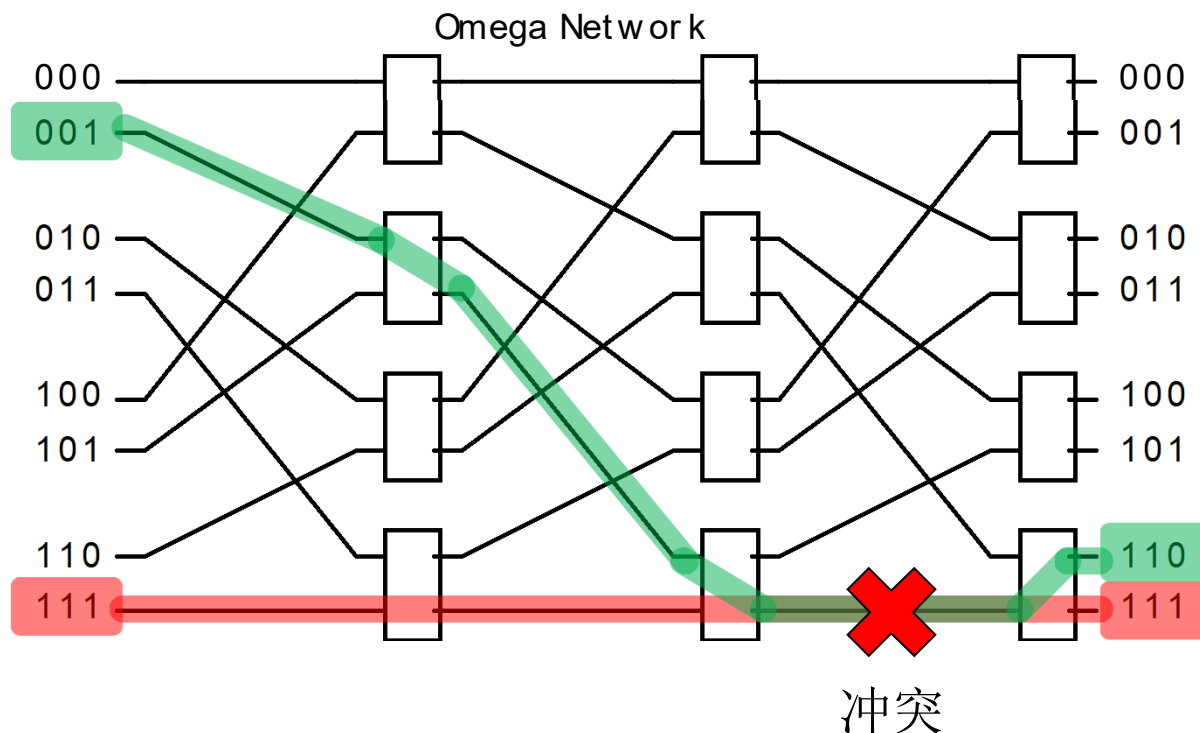
# 多级对数网络

- 思路: 在终端/节点之间通过多层交换实现间接组网
- 成本:  $O(N\log N)$ , 延迟:  $O(\log N)$
- 很多变种(Omega, 蝴蝶, Benes, Banyan, ...)
- Omega 网络:

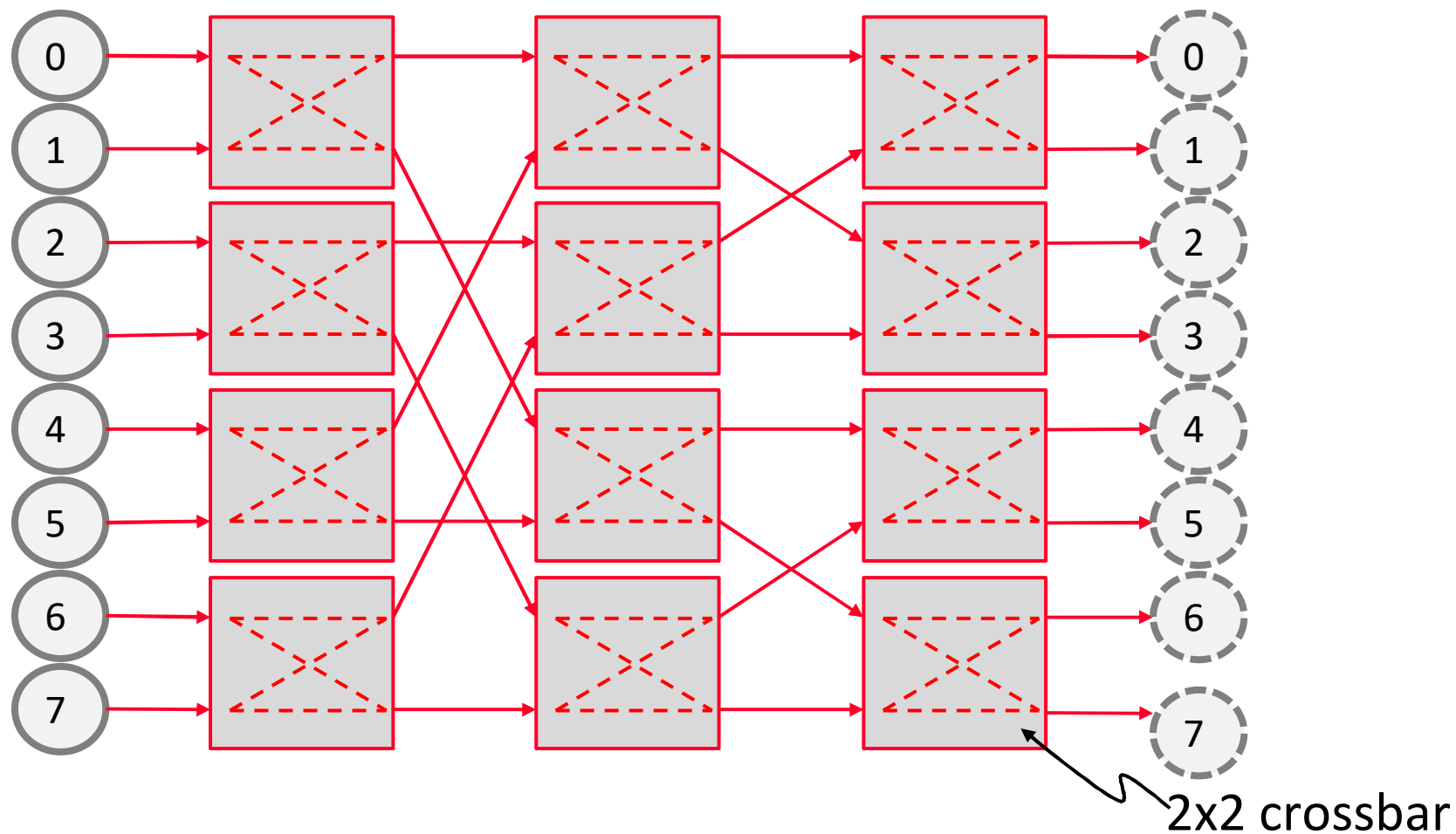


# 多级对数网络

- 思路: 在终端/节点之间通过多层交换实现间接组网
- 成本:  $O(N\log N)$ , 延迟:  $O(\log N)$
- 很多变种(Omega, 蝴蝶, Benes, Banyan, ...)
- Omega 网络:

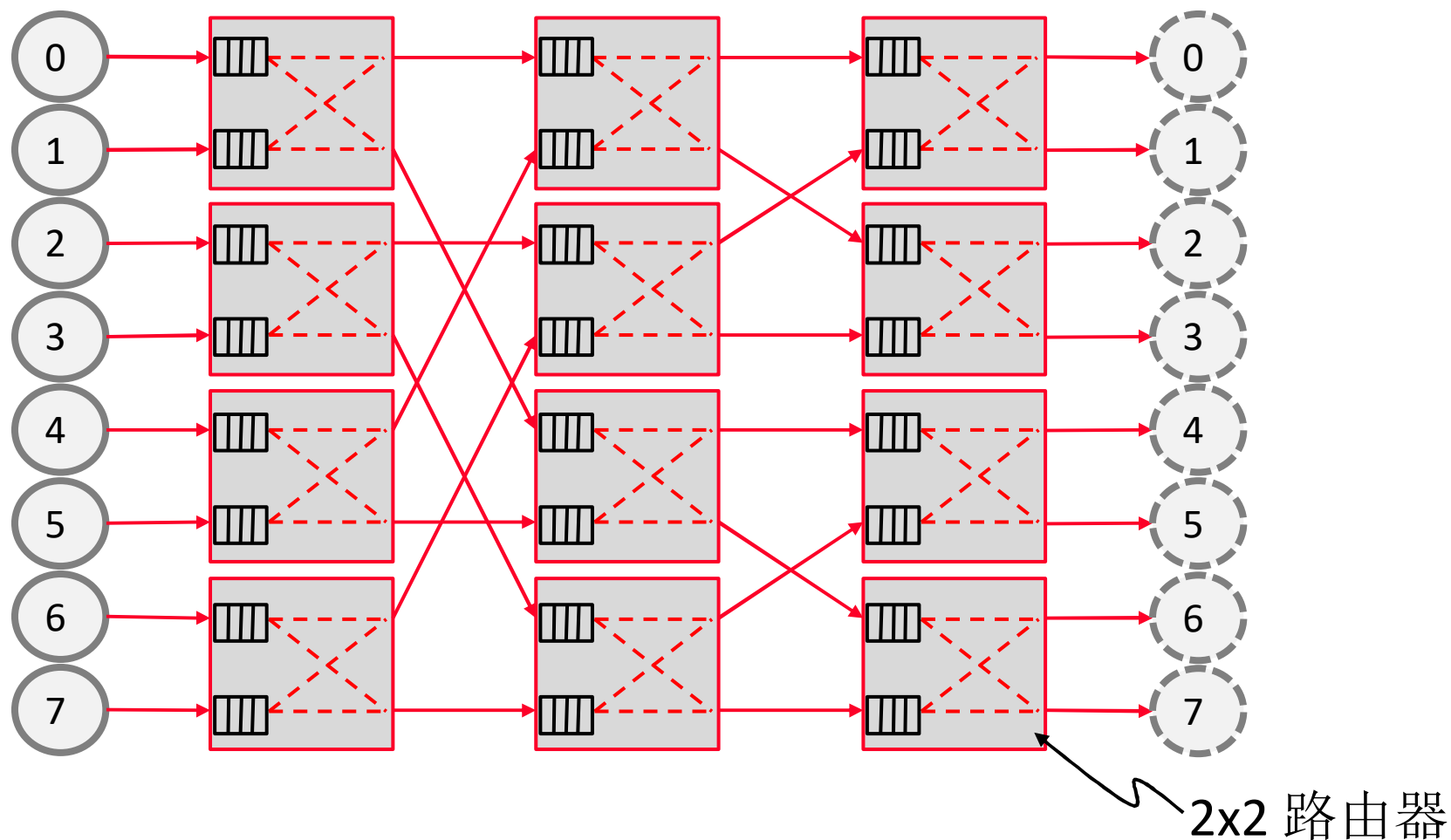


# 多级电路交换



- 对并发更严格
- 但是相对于crossbar的成本来说具有更好的可扩展性

# 多级包交换



- 包在路由器之间逐“跳”传递，等待下一跳交换机和缓冲的可用性

# 交换 vs. 拓扑

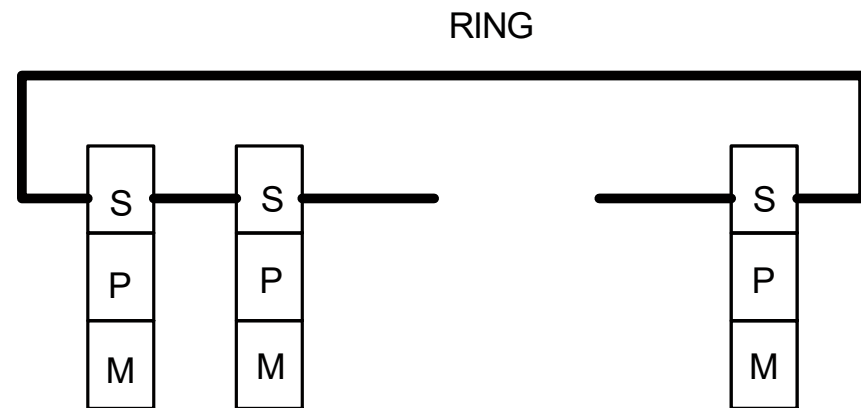
---

- 电路/包交换的选择不依赖于拓扑
- 消息如何传送至目的地依靠高层协议
- 当然, 某些拓扑确实可能更适合于电路或者包交换

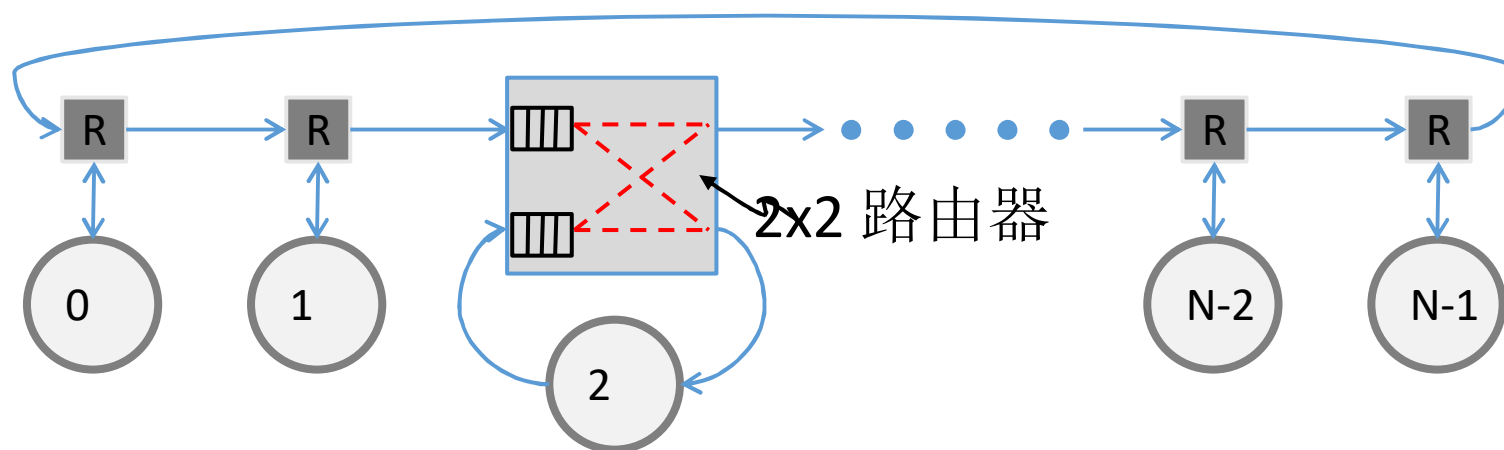
# 环

- + 便宜:  $O(N)$  成本
- 高延迟:  $O(N)$
- 不易扩展
  - 对分带宽是常数

在Intel Haswell, Intel Larrabee, IBM Cell等很多现代商业化系统中使用



# 单向环



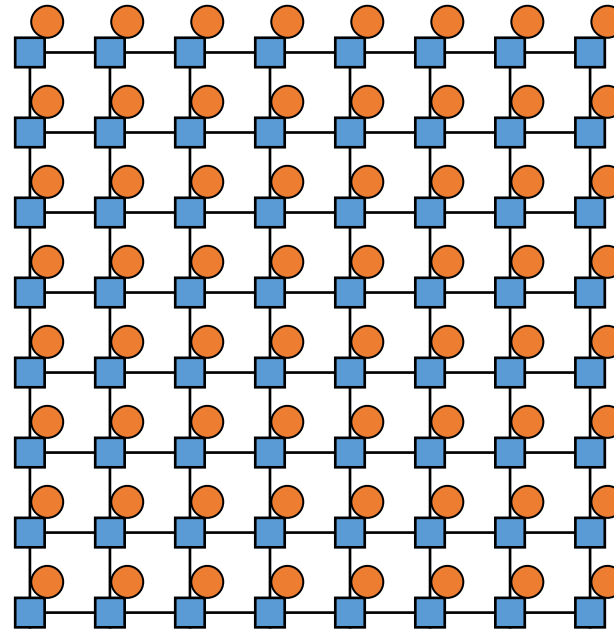
- 拓扑及实现简单
  - 如果 $N$ 和带宽及延迟要求都相对比较低, 性能比较合理
  - $O(N)$  成本
  - $N/2$  平均跳数; 延迟依赖于利用率



# 网状网(Mesh)

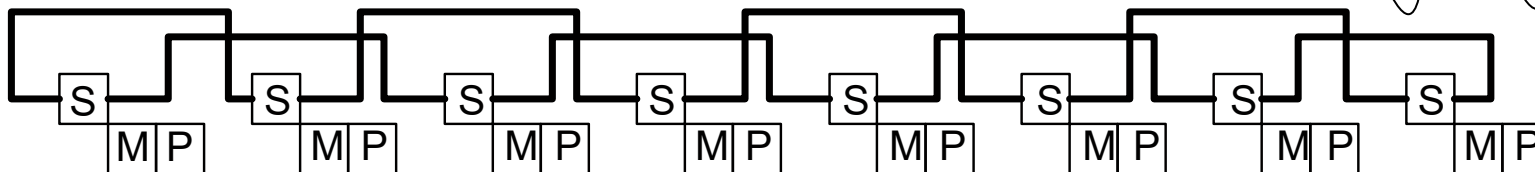
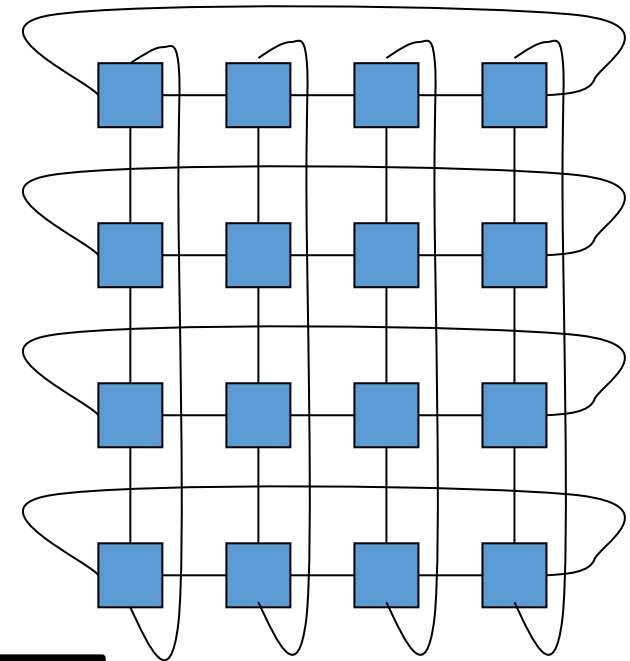
---

- $O(N)$  成本
  - 平均延迟:  $O(\sqrt{N})$
  - 容易在芯片上布局: 规则并且等长的连接
  - 路径多样性: 从一点到另一点有很多条路
- 
- Tiler 100核芯片中使用
  - 是许多片上网络的原型



# Torus

- Mesh 在边缘部分不对称: 在边缘放置任务时性能会非常敏感
- Torus 避免了这个问题
- + 比mesh更高的路径多样性(和对分带宽)
- 更高的成本
- 片上难布局
  - 链路长度不相等
- 编织节点使节点间延迟近似常数



# 树

平面、分层拓扑结构

延迟:  $O(\log N)$

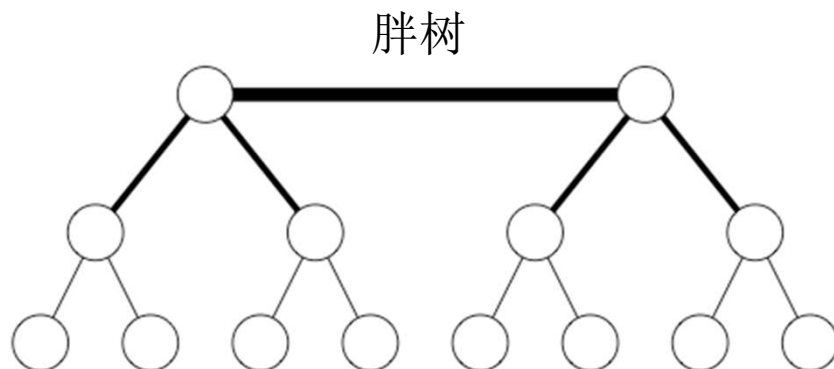
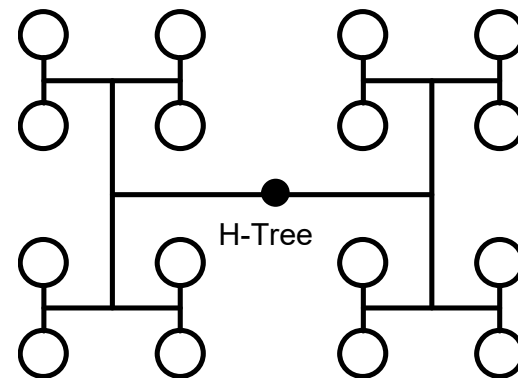
对局部流量很有效

+ 便宜:  $O(N)$  成本

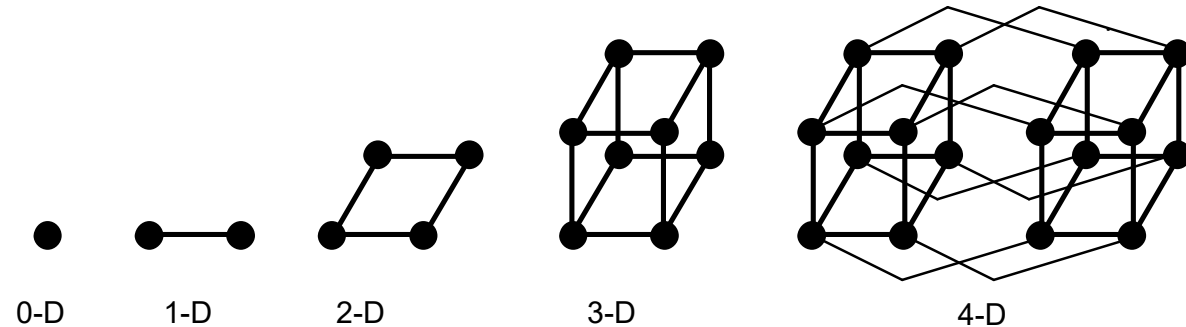
+ 容易布局

- 根会成为瓶颈

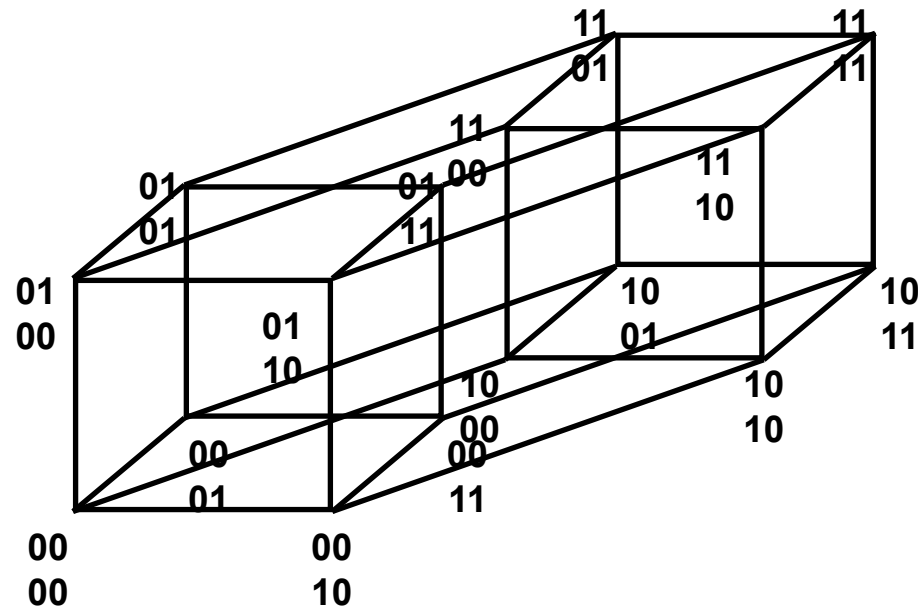
胖树可以避免这一问题(CM-5)



# 超立方

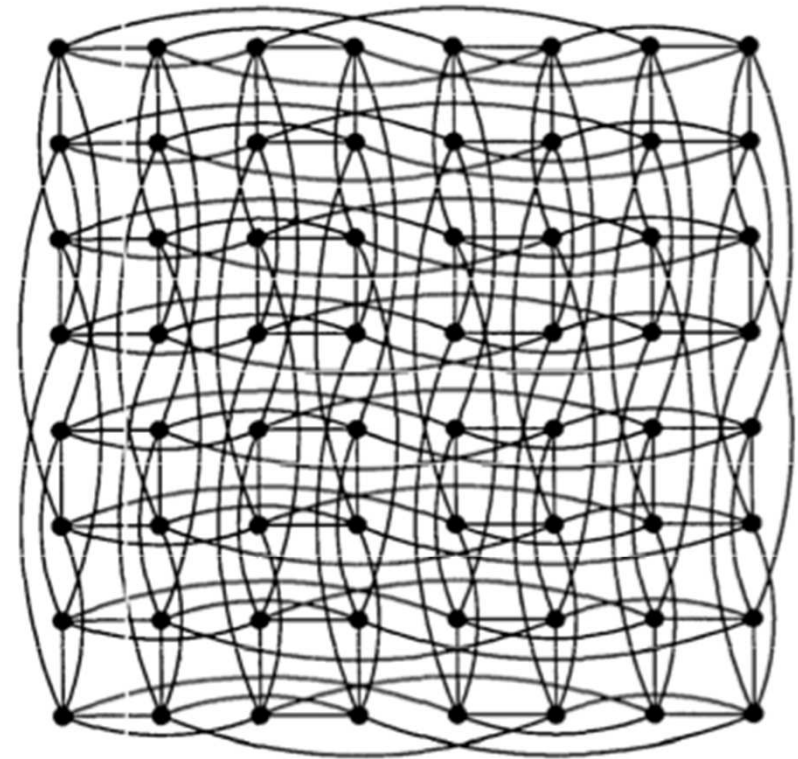
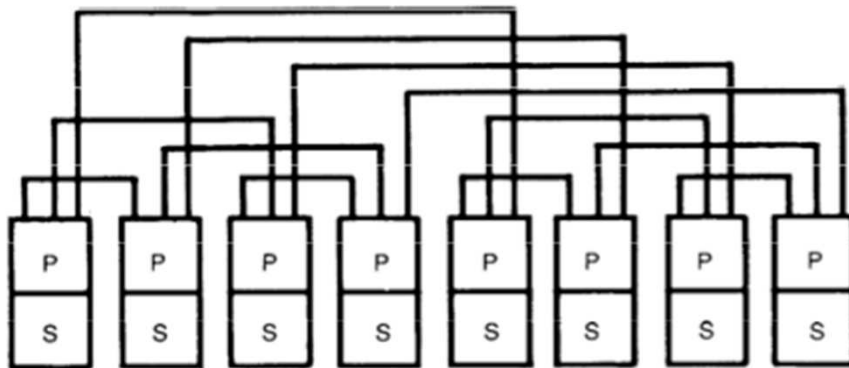


- 延迟:  $O(\log N)$
  - 基数:  $O(\log N)$
  - 连接数:  $O(N \log N)$
- + 低延迟
- 在2D/3D中难布局



# 加州理工的宇宙立方

- 64节点的消息传递机器
- Seitz, “[The Cosmic Cube](#),” CACM 1985.

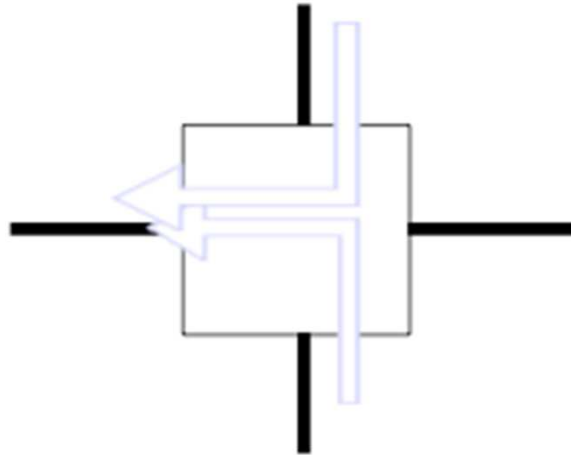


A hypercube connects  $N = 2^n$  small computers, called nodes, through point-to-point communication channels in the Cosmic Cube. Shown here is a two-dimensional projection of a six-dimensional hypercube, or binary 6-cube, which corresponds to a 64-node machine.

FIGURE 1. A Hypercube (also known as a binary cube or a Boolean  $n$ -cube)

# 处理竞争

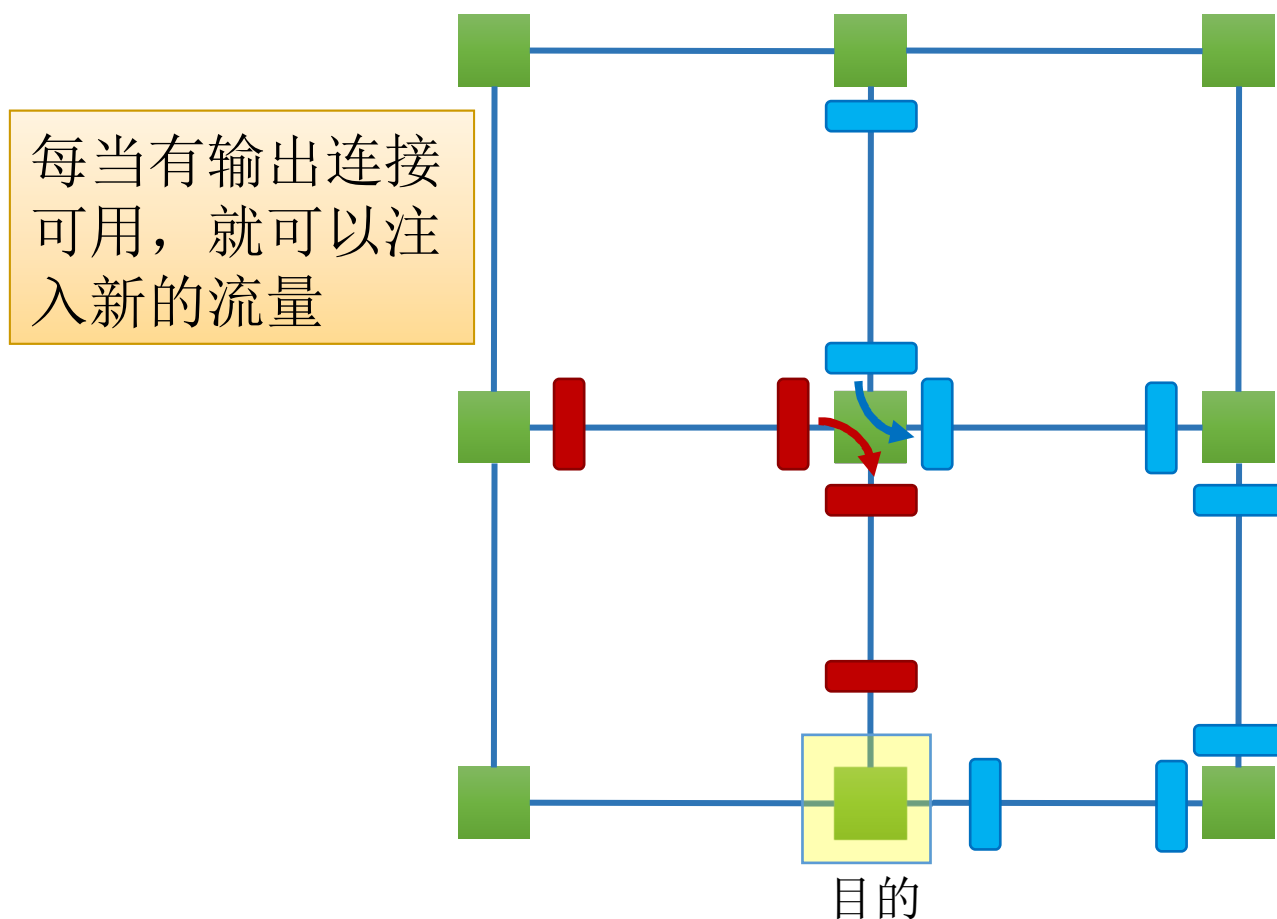
---



- 两个包试图同时使用同一个连接
- 如何处理?
  - 缓冲一个
  - 丢弃一个
  - 偏转 (错误路由) 一个
- Tradeoff?

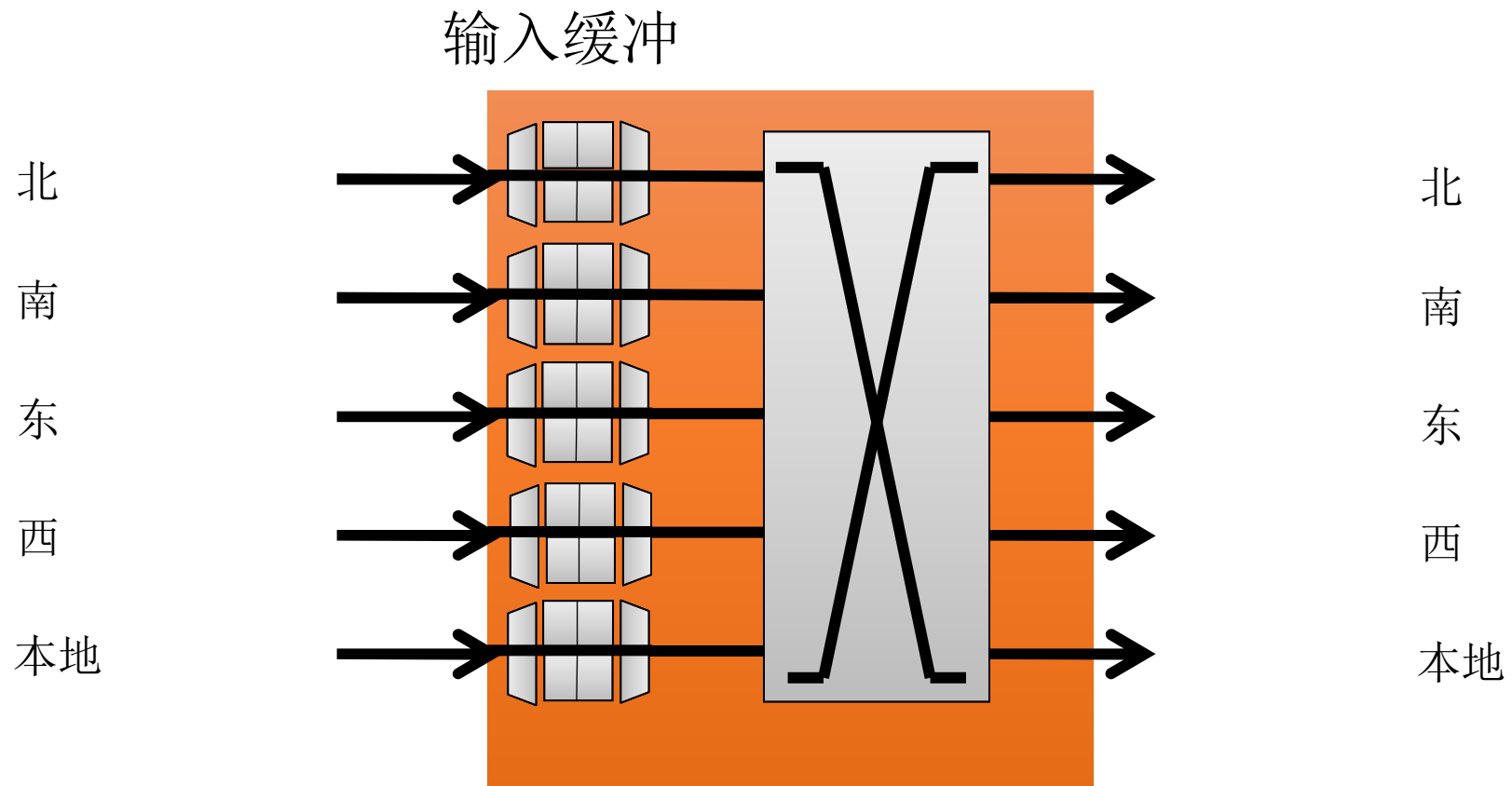
# 无缓冲偏转路由

- 核心思路: 包在网络中绝不缓冲, 当两个包争用同一连接时, **偏转** 其中一个



# 无缓冲偏转路由

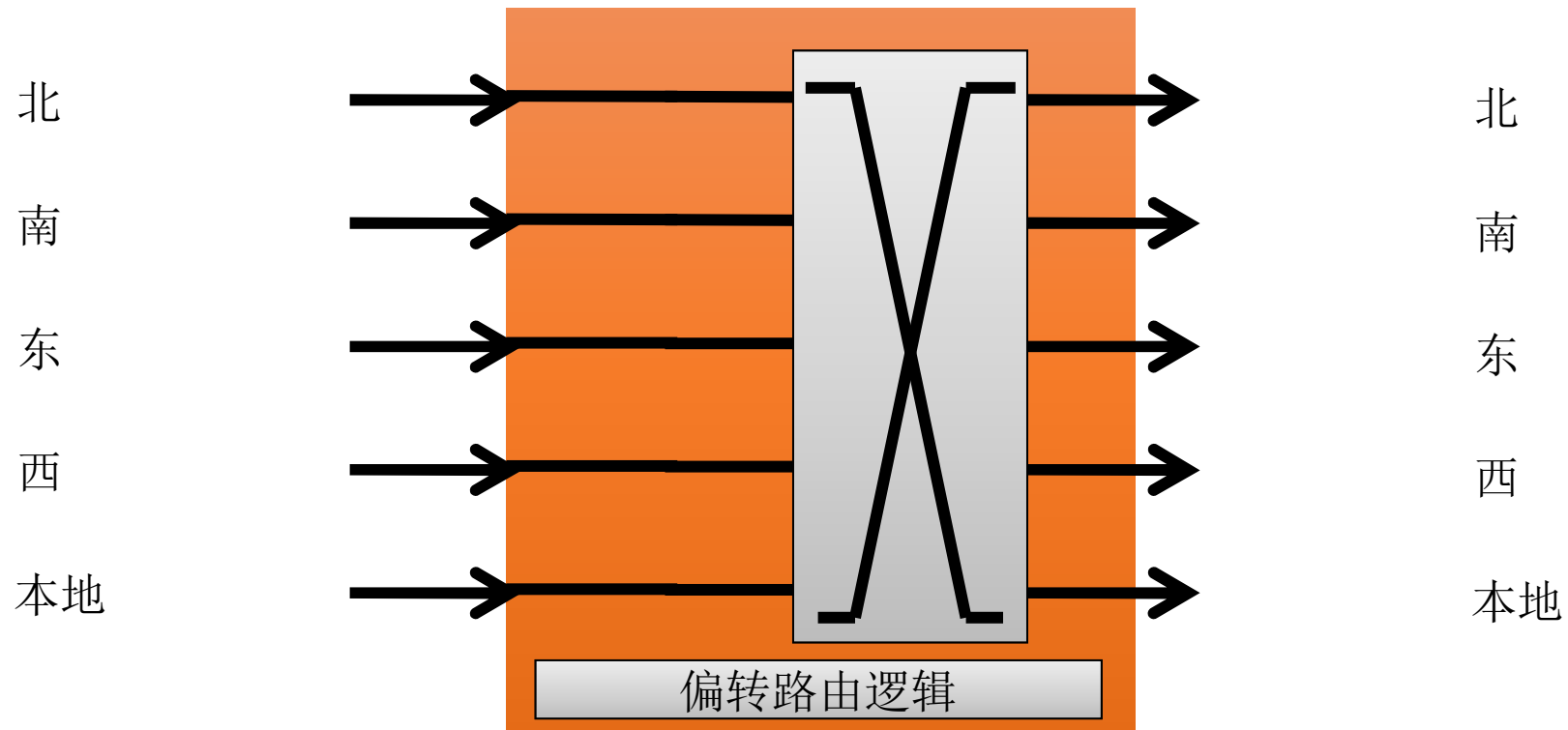
- 可以免除输入缓冲: 传输的包会通过流水线锁存器和网络连接“缓冲”





# 无缓冲偏转路由

- 可以免除输入缓冲: 传输的包会通过流水线锁存器和网络连接“缓冲”



# 路由算法

---

- 类型
  - 确定的: 总是为一个源-目的对之间的通信选择同样的路径
  - 健忘的: 选择不同的路径, 不考虑网络状态
  - 适应的: 可以选择不同的路径, 适应网络的状态
- 如何适应
  - 局部/全局反馈
  - 最小或非最小路径

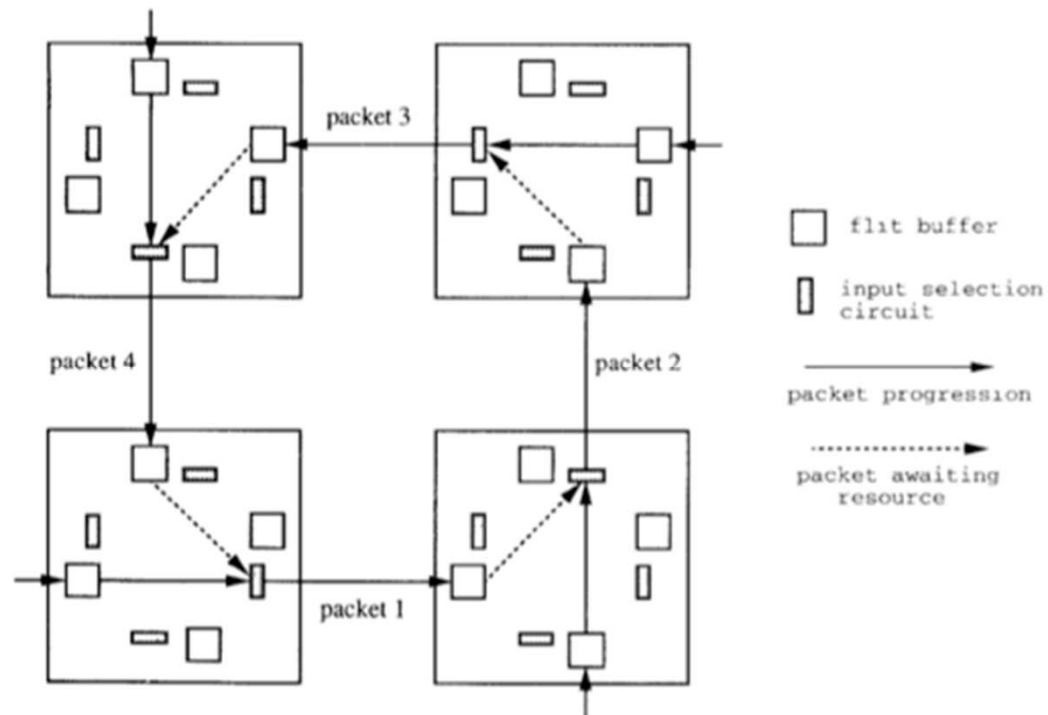
# 确定性路由

---

- 相同(源, 目的)对的包走同样的路径
  - 维度序路由
    - 比如, XY 路由(Cray T3D, 其它很多片上网络)
    - 先遍历维度 X, 再遍历维度 Y
- + 简单
- + 无死锁(资源分配不需要时钟周期)
- 可能导致高度竞争
- 没有利用路径多样性

# 死锁

- 没有进程可以向前推进
- 由对资源的循环依赖引发
- 每个包等待被下游包占据的缓冲区



# 处理死锁

---

- 避免路由中的环
  - 维度序路由
    - 不会产生循环依赖
  - 限制每个包的“轮次”
- 通过增加缓冲避免死锁(逃生路径)
- 检测并突破死锁
  - 抢占缓冲区

# 避免死锁的转向模型

- 思路
  - 分析一个网络中包可能的转移方向
  - 确定这些转向可以构成的环
  - 禁止某些转向以打破可能的环
- Glass and Ni, “[The Turn Model for Adaptive Routing](#),” ISCA 1992.

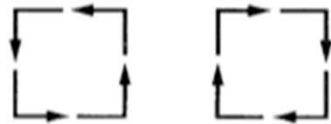


FIG. 2. The possible turns and simple cycles in a two-dimensional mesh.



FIG. 3. The four turns allowed by the *xy* routing algorithm.

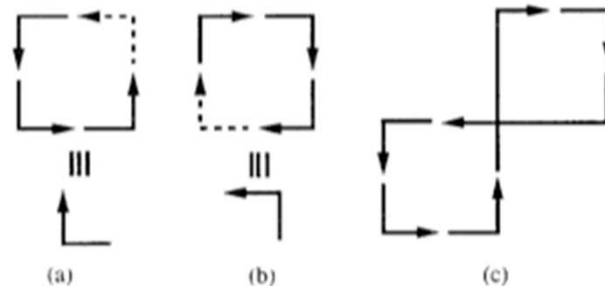


FIG. 4. Six turns that complete the cycles and allow deadlock.

# 健忘性路由: 勇士算法

---

- 健忘性算法的例子
  - 目标: 均衡网络负载
  - 思路: 随机选择一个中间目的节点, 首先路由到该节点, 接着从该节点路由到最终目的
    - 源-中间节点和中间节点-目的, 可以使用维度序路由
- + 随机的/均衡网络负载
- 非最小(包延迟可能增加)
- 优化:
    - 在高负载时使用
    - 限制中间节点

# 适应性路由

---

- 最小适应性

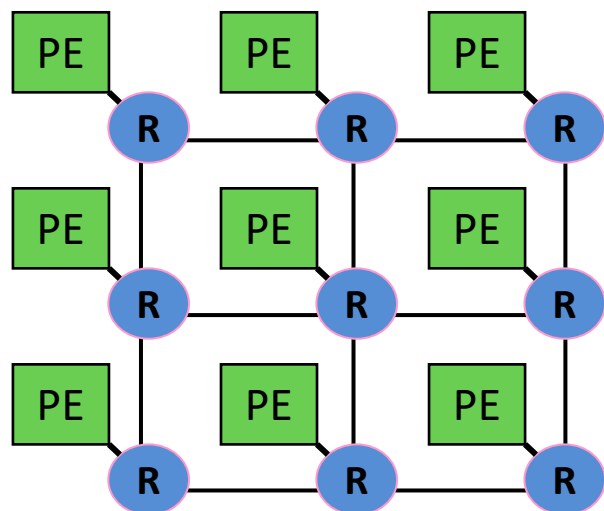
- 路由器根据网络状态(比如, 下游缓冲区的占用情况)来选择高效输出口发送包
- 高效输出口: 能使包离目的更近的端口
- + 能感知局部拥塞
- 追求最小性限制了高连接利用率的获得 (负载均衡)

- 非最小(完全) 适应性

- 根据网络状态将包“错误路由”到非高效输出口
- + 能够获得更好的网络利用率和负载均衡
- 需要保证避免活锁



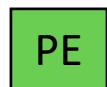
# 片上网络



- 连接核、**cache**、内存控制器等
  - 总线和交叉开关不具有可扩展性
- 包交换
- **2D mesh**: 最常用的拓扑
- 主要用来应对**cache**缺失和访存请求



路由器



处理单元

(核, L2 Bank, 内存控制器, 等等)

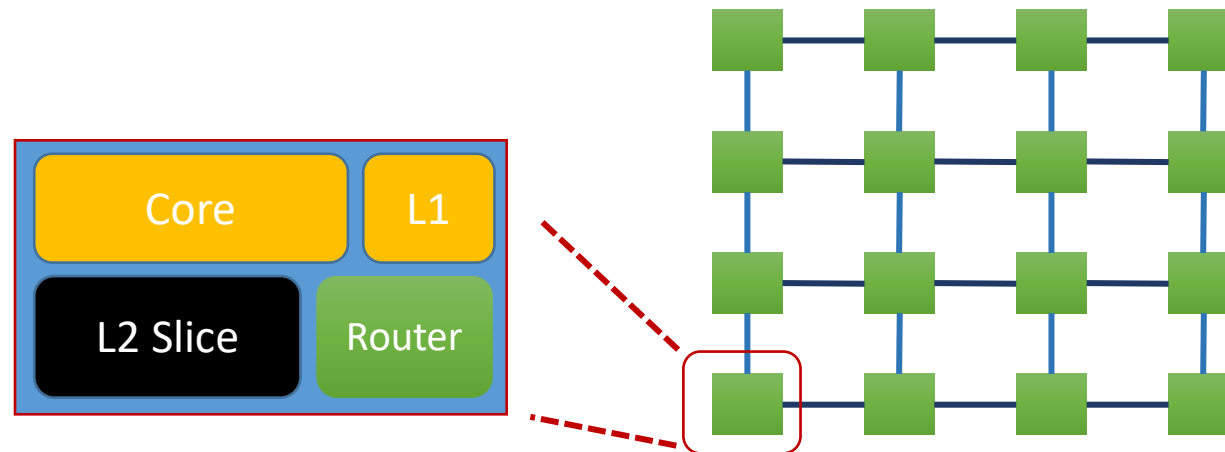
# 高效互连的动机

- 在众核芯片中, 片上互连(NoC)消耗了巨大的能量

**Intel Terascale:** ~28% 芯片功耗

**Intel SCC:** ~10%

**MIT RAW:** ~36%



- 最近的一些工作利用无缓冲偏转路由来减小功耗和芯片尺寸