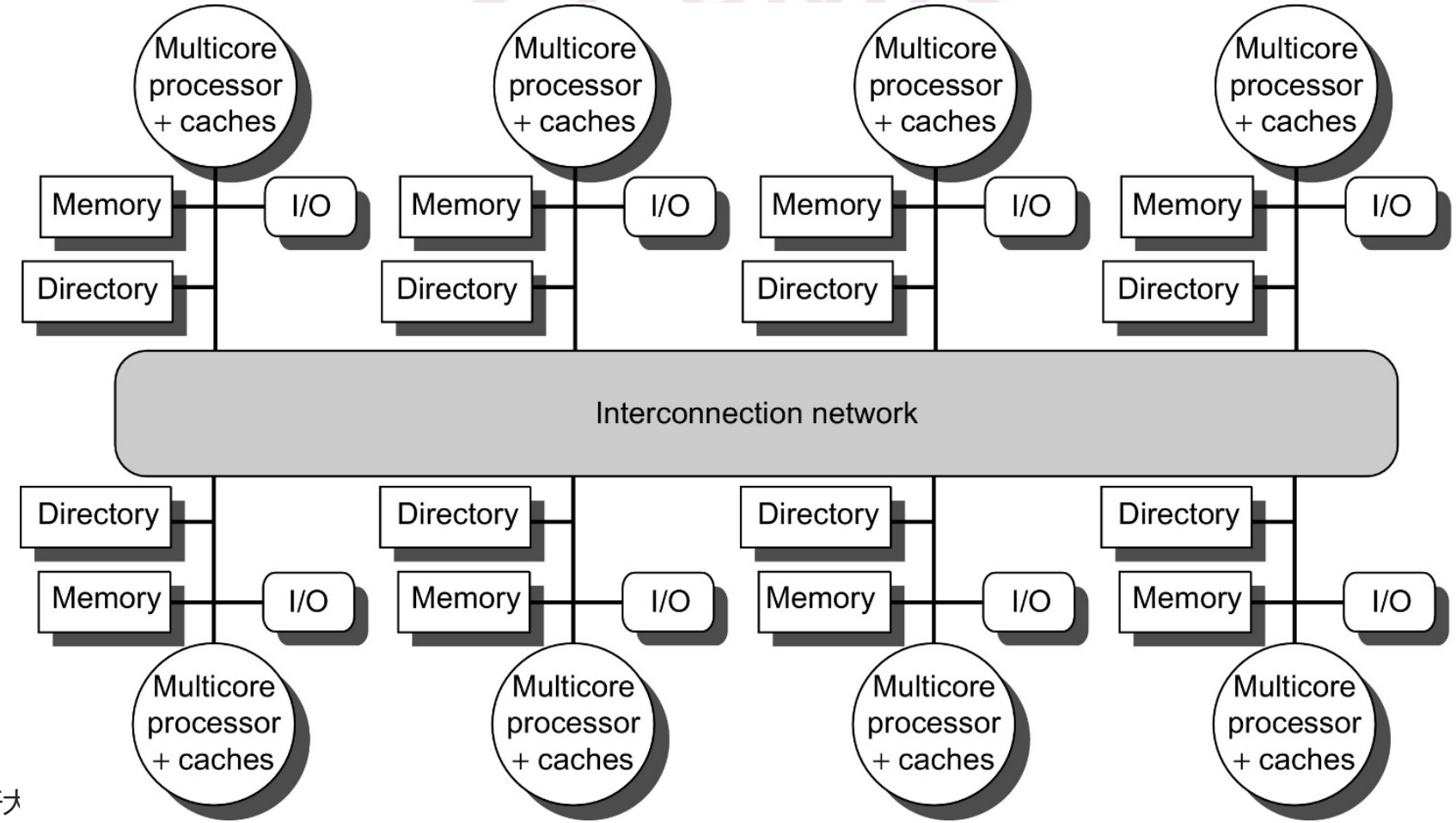
线程级并行性(2)

目录协议

- 为了保证缓存数据的一致性，监听协议在每次缓存不命中（包括写共享数据）时均要求所有缓存之间进行通信。
  - 基本优势：成本低廉
  - 致命弱点：可扩展性差
- 目录协议(directory protocol)
  - 目录保存每个缓存数据块的状态
  - 目录内容
    - 哪个缓存拥有数据块副本
    - 数据块是否处于“脏”状态
- 在共享 L3 缓存中实现
  - 每个数据块中保存比特矢量，矢量长度为处理器核的数目
  - 每个比特矢量表示是否某个私有 L2 缓存中包含 L3 数据块的副本
  - 无法扩展到共享L3之外

分布式目录

- 特点
  - 内存和目录共同分布存储，以便不同的一致性请求转到不同的目录
  - 在每个节点上添加目录以实现缓存一致性



目录协议

- 简单实现
  - 将目录中的每个条目与一个内存块相关联
  - 信息总量：内存块数目 × 节点数目
    - 节点：实现了内部数据一致性的单个多核处理器或一组处理器
- 主要操作
  - 处理读操作未命中
  - 处理对共享、干净缓存数据块的写操作
- 缓存数据块状态
  - 共享(shared)：一个或多个节点已缓存该块，且内存中数值是最新的。
    - 使用比特矢量表示拥有该共享缓存数据块副本的处理器，便于失效操作处理
  - 未缓存(uncached)：没有节点拥有该缓存数据块副本
  - 修改(modified)：只有一个节点拥有该缓存数据块副本，且已经修改该缓存数据块的内容，故内存中数值是过时的。

节点类型

- 局部节点(local node)
  - 发出请求的节点
- 主节点(home node)
  - 一条物理地址关联的内存位置和目录项所在的节点
- 远程节点(remote node)
  - 拥有缓存数据块副本，状态为独占(exclusive)或共享(shared)。
  - 远程节点也可以是局部节点或主节点

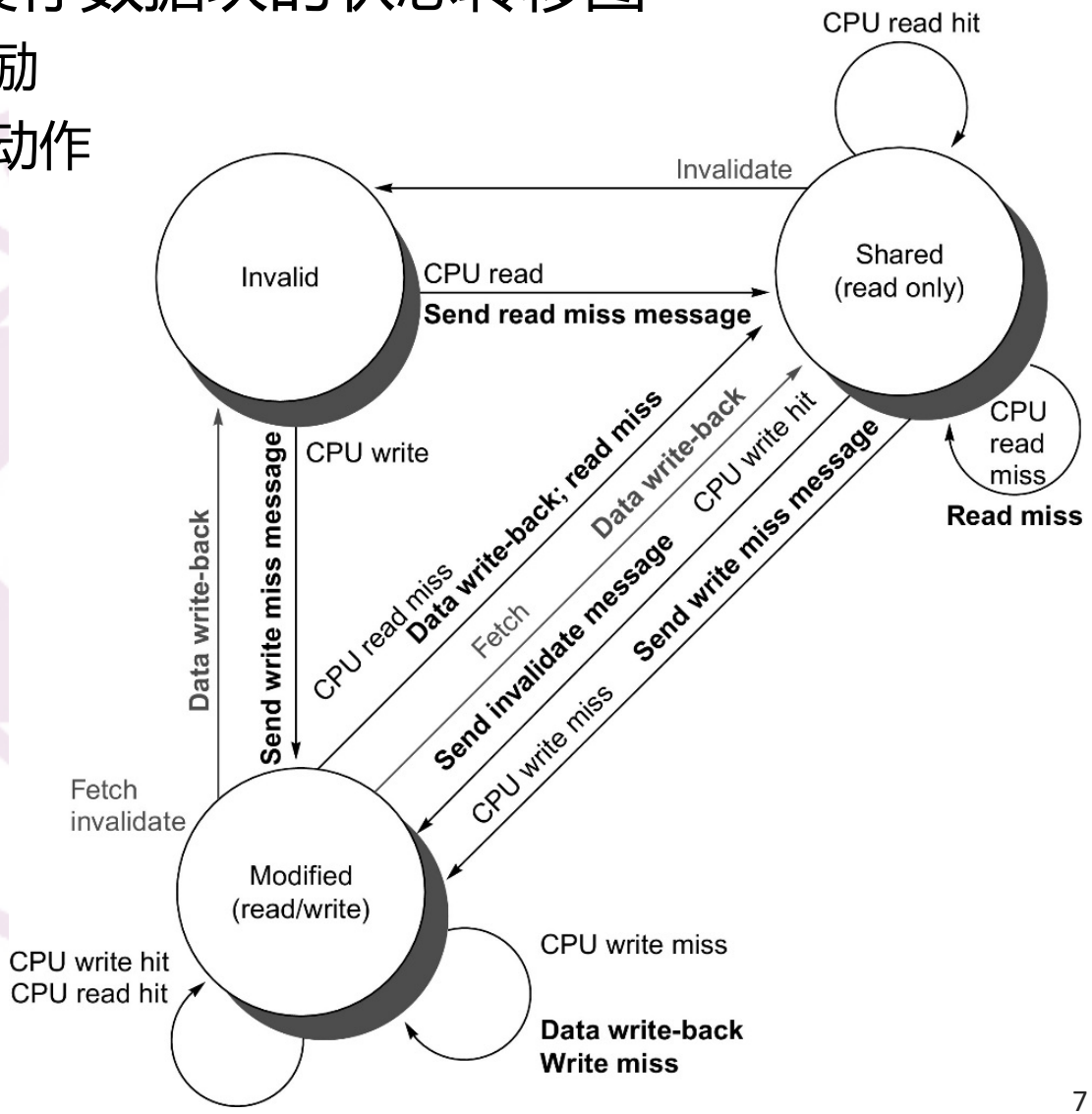
节点间传递的消息

- 为了维护数据一致性，节点间传递的信息。
  - P：请求节点号，A：请求地址，D：数据内容

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/ invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write back a data value for address A.

状态转移图

- 基于目录协议中单个缓存数据块的状态转移图
  - 正常字体：状态转移激励
  - 粗体：状态转移产生的动作
  - 灰色：外部节点的请求
  - 黑色：局部节点的请求



两种协议状态转移图之比较

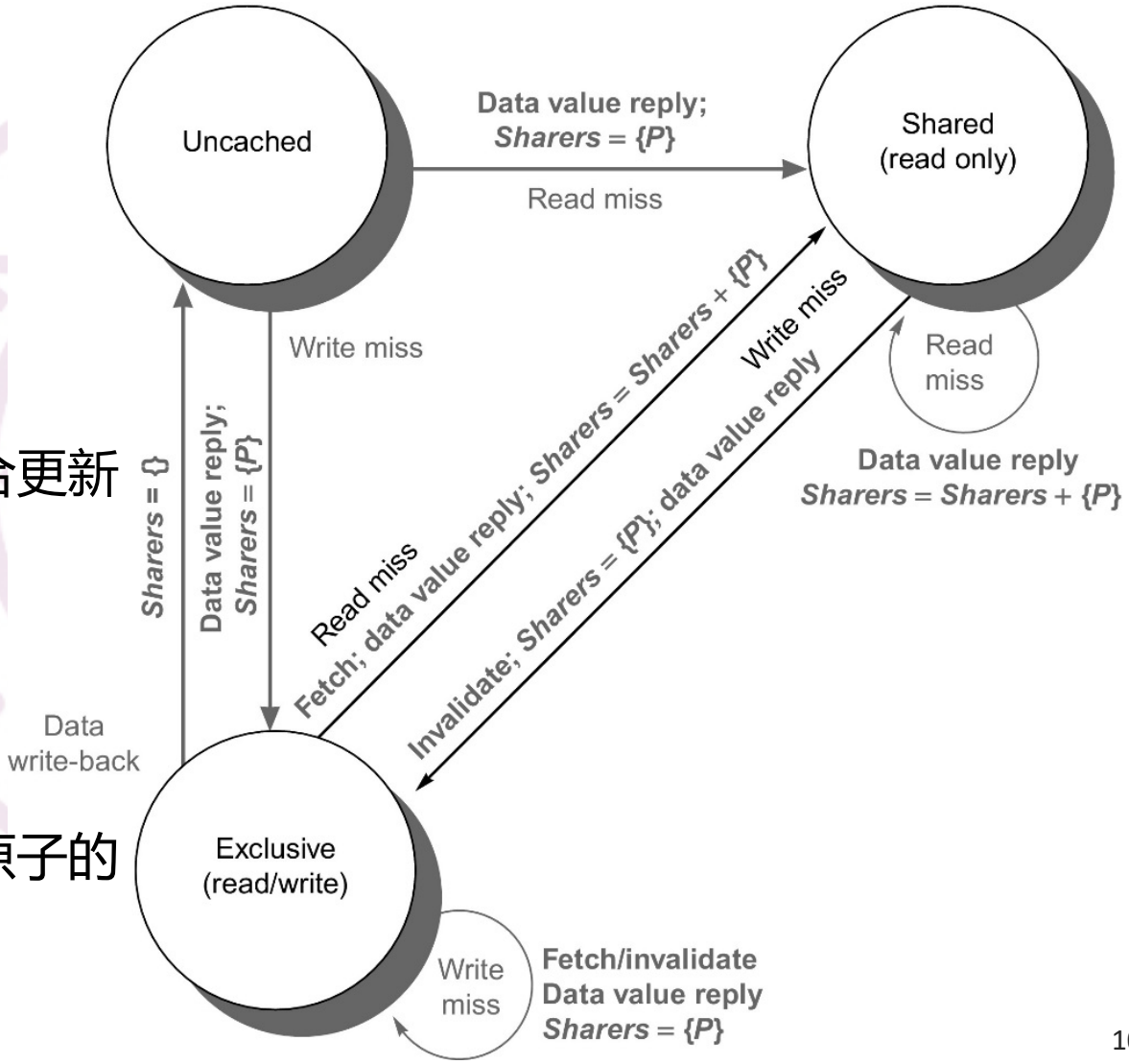
- 相似
  - 状态转移原因
    - 读操作不命中
    - 写操作不命中
    - 失效操作
    - 取数据请求
  - 读和写操作不命中需要回应数据值，在改变状态之前需要等待回应
  - 对数据块的写操作使其处于独占状态
  - 任何共享数据块在内存中必须是最新的
- 差异
  - 写操作不命中
    - 监听协议：在总线（或其它网络）上广播
    - 目录协议：目录控制器有选择地发送取数据和失效操作

目录协议

- 发送到目录消息的不同导致两种不同类型的操作
  - 更新目录状态
  - 发送附加消息以满足请求
- 目录条目（状态）表示一个数据块的三种标准状态
  - 目录状态表明内存块所有缓存副本的状态，而不是单个缓存块的状态。
- 内存块的三种标准状态
  - Uncached：未被任何节点缓存
  - Shared：被多个节点缓存且可读
  - Exclusive：被单个节点独占地缓存且可写
- 使用共享者(Sharers)集合表示拥有数据块副本的节点集合
  - 当系统小于64个节点时，共享者集合可以使用比特矢量表示。

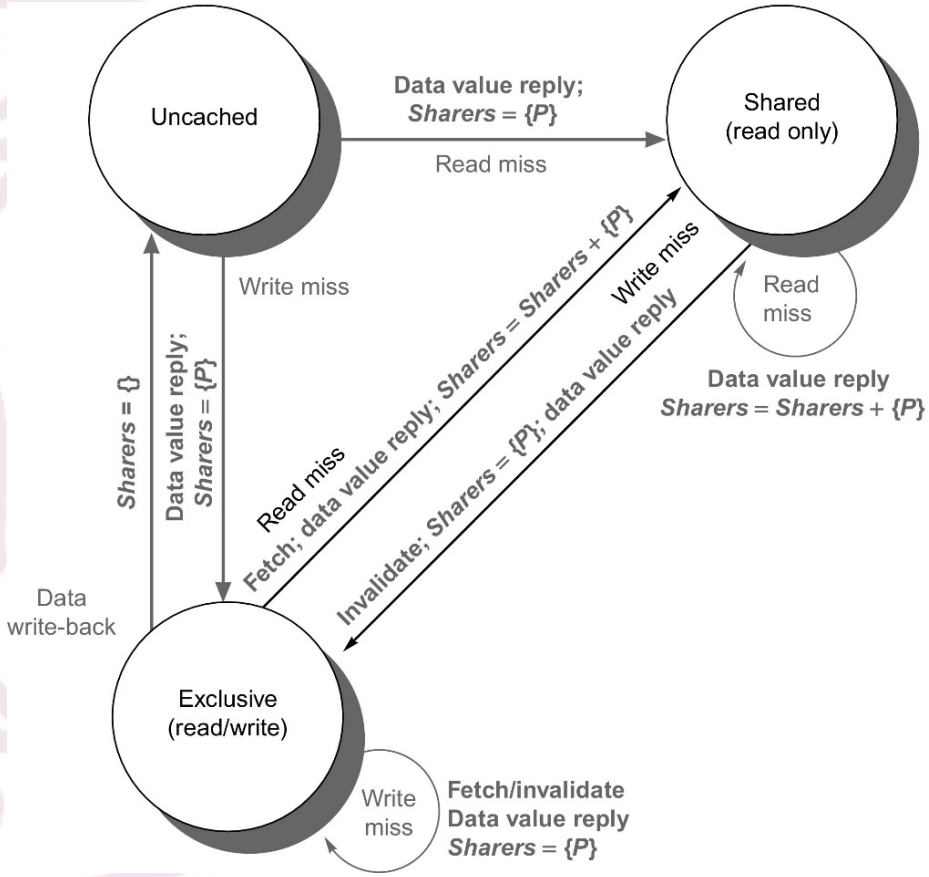
目录协议

- 目录对接收到信息做出响应动作的状态转移图
  - 接收三种不同请求
    - 读操作不命中
    - 写操作不命中
    - 数据写回
  - **粗体**：目录响应
  - ***粗斜体***：共享者集合更新
  - 灰色：激励和动作
  - 假设所有动作都是原子的



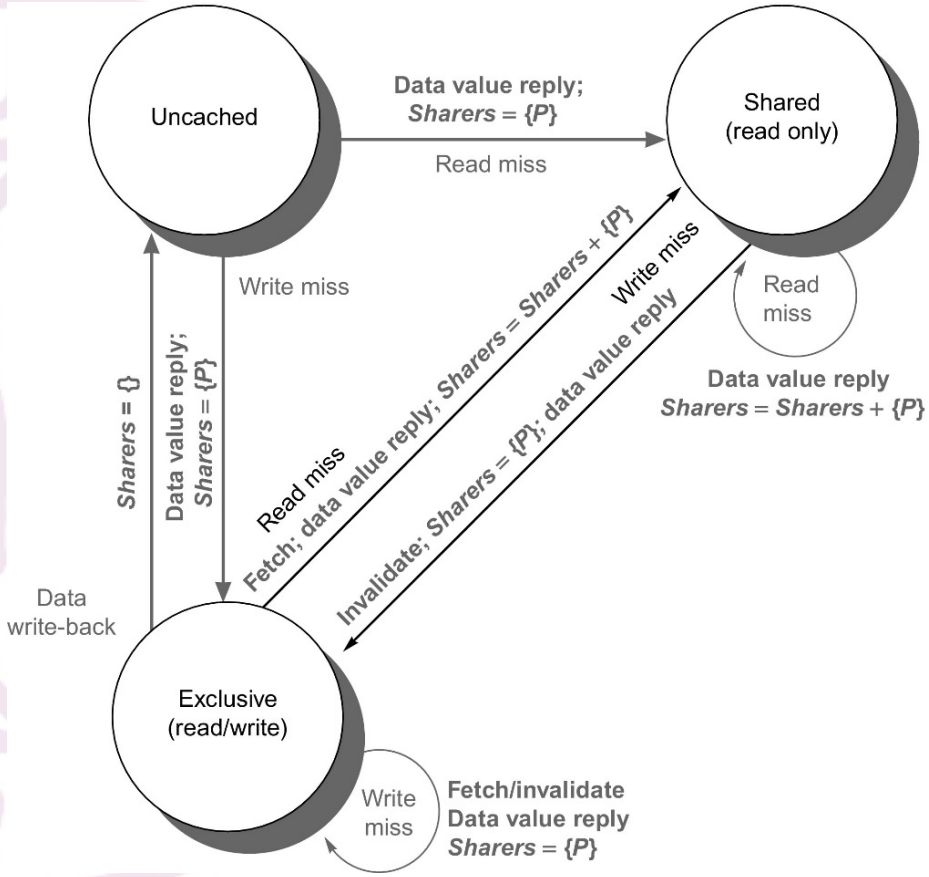
Uncached 状态

- 读未命中(read miss)
  - 将请求数据发送给请求节点
  - 请求节点成为唯一共享节点
  - 数据块为共享(shared)状态
  - 共享者集合置为所有者身份
- 写未命中(write miss)
  - 将请求数据发送给请求节点
  - 数据块为独占(exclusive)状态，以表示只有一个有效缓存副本
  - 共享者集合置为所有者身份



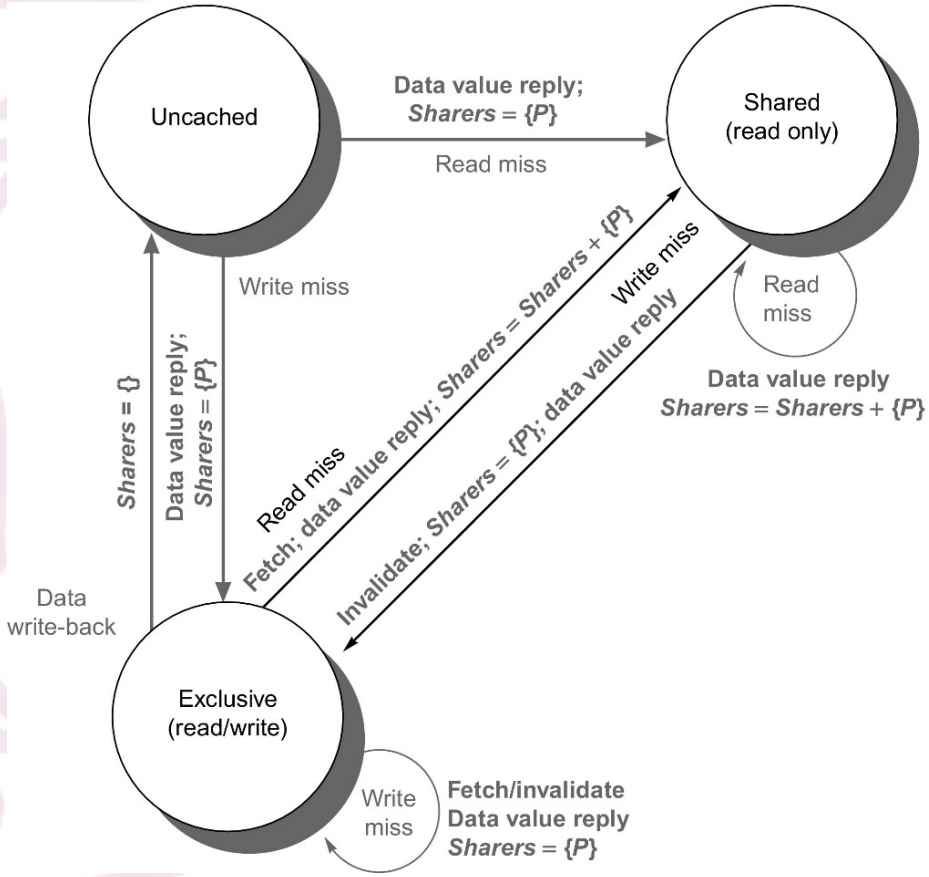
Shared 状态

- 读不命中(read miss)
  - 将请求数据发送给请求节点
  - 请求节点加入到共享者集合
- 写不命中(write miss)
  - 发送失效信息给共享者集合中的所有节点
  - 共享者集合置为请求节点身份
  - 回复请求数据给请求节点
  - 数据块为独占(exclusive)状态



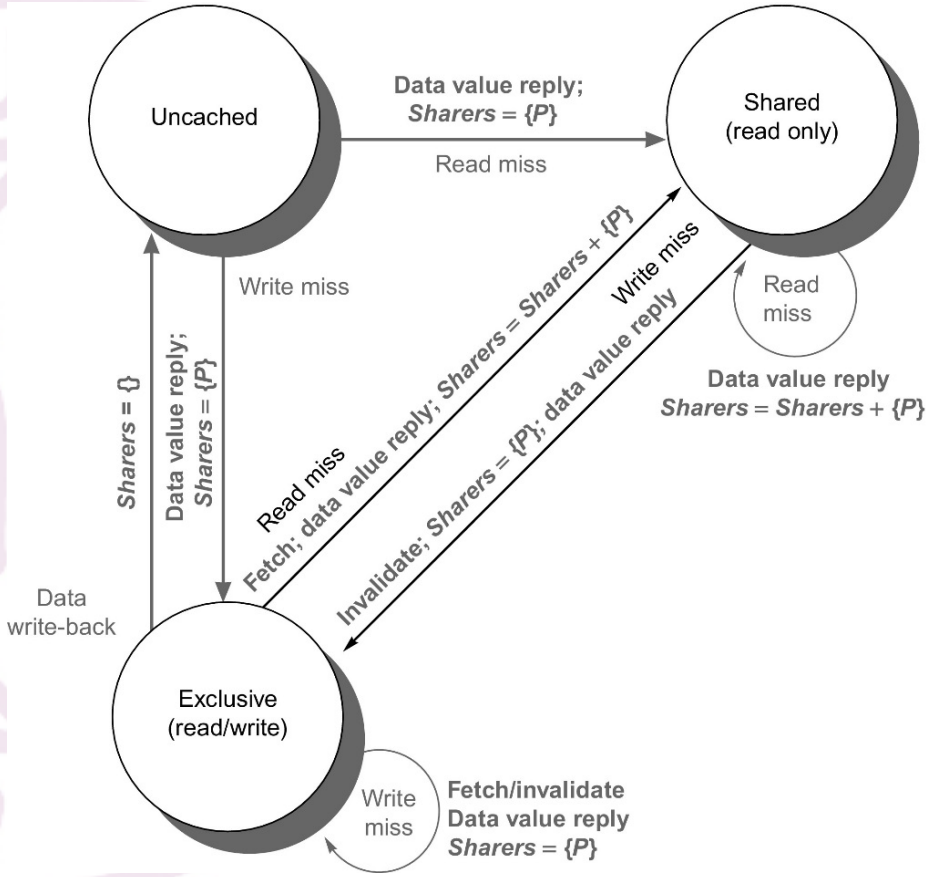
Exclusive 状态

- 读不命中(read miss)
  - 向所有者发出一条获取(fetch)数据信息
    - 所有者Cache中的数据块变为共享(shared)状态
    - 所有者将数据发送给目录，将数据写回内存，并发送给请求处理器
  - 将请求节点身份添加到共享者集合中



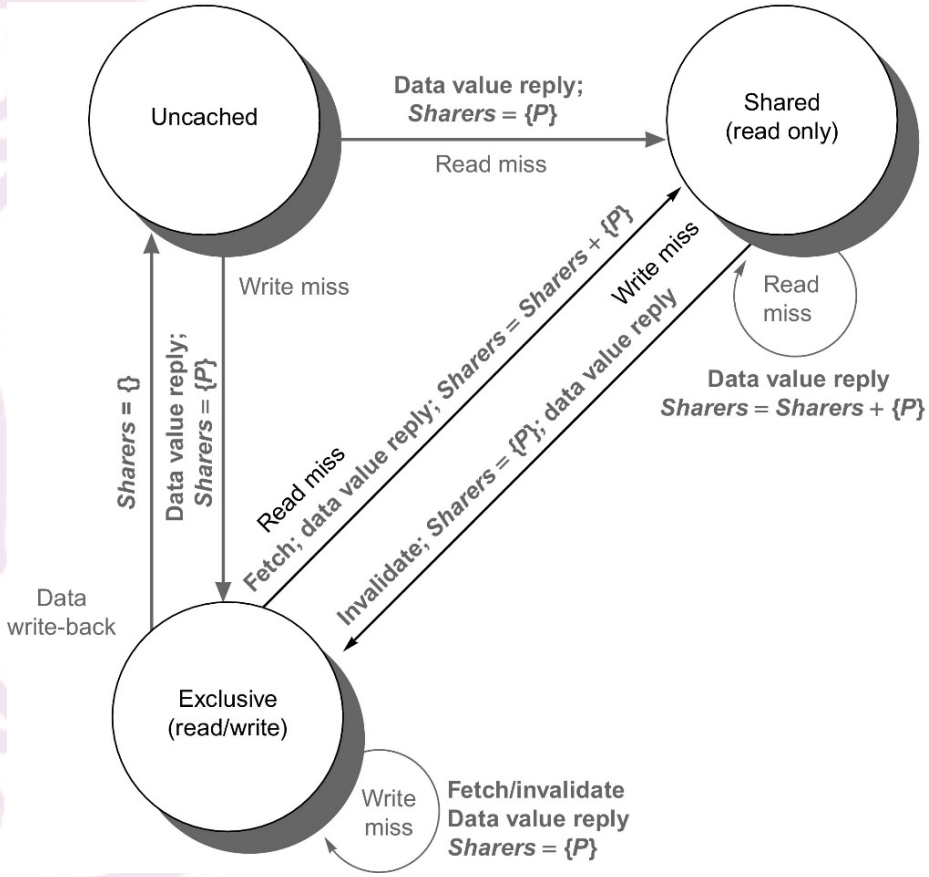
Exclusive 状态

- 数据写回(data write-back)
  - 所有者的数据块被替换，必须写回到内存
  - 内存数据是最新的，主目录成为所有者
  - 数据块为未缓存状态
  - 共享者集合置空



Exclusive 状态

- 写不命中(write miss)
  - 老拥有者将数据发送给目录，并由此发送给请求节点
  - 发送失效消息给老的拥有者，将缓存数据块失效
  - 请求节点成为新的所有者
  - 共享者集合置为请求节点
  - 数据块为独占(exclusive)状态



目录协议

- 多芯片一致性和多核一致性的组合
  - 监听-监听 (AMD)
  - 监听-目录
  - 目录-监听
  - 目录-目录

- 常用选择
  - 在单个芯片内（多核）采用监听协议
    - 如果包含最外层缓存是共享的
  - 多个芯片之间采用目录协议

同步

- 同步机制一般是由用户级软件程序构建的，而这些软件程序是以硬件提供的同步指令(synchronization instruction)为基础的。
  - 不间断的指令或指令序列，能够原子检索和更改一个数值。

- 需要一组具有原子读和修改内存内容的硬件原语(hardware primitives)
  - 这些原语是构建各种用户级同步操作的基本构件(basic building block)
  - 通常由系统程序员用来编写同步库函数，而不是一般用户使用

基本构件

- 原子交换(atomic exchange)
  - 内存位置的数值和寄存器的数值之间的交换
  - 例：
    - 处理器通过将寄存器中的数值 1 与“锁(lock)”对应内存位置的数值进行交换，以设置锁。
    - 如果其它处理器已经声明访问，则交换指令返回 1，否则返回 0。
  - 关键：交换是不可分开的——原子的
  - 两个同时的交换将由写串行机制来排序

- 测试且设置(test-and-set)
  - 如果对数值的测试通过，则置位。
  - 例：如果测试为 0，则置位为 1。类似于原子交换。

基本构件

- 读取且递增(fetch-and-increment)
  - 返回内存位置的数值且对内存位置的数值递增
- 单条不可中断指令完成内存读和写操作

基本构件

- 可选方法：指令对(pair of instructions)
  - 从第二条指令的返回值中可以推断这对指令是否像原子指令一样执行。

- 保留加载 / 条件存储(load reserved, store conditional)
  - 链接加载(load linked)或加锁加载(load locked)
  - 执行步骤
    - 保留加载指令：将 $rs1$ 指定内存地址的内容加载到 rd 中，且在该内存位置设置保留地(reservation)；
    - 条件存储指令：将 $rs2$ 中数值存储到由 $rs1$ 指定的内存位置；
    - 如果保留地内容被写入相同地址位置的其它指令破坏，则条件存储指令失败，在 $rs2$ 中写入非零数值；如果成功，则写入数值零。
  - 如果交换两条指令的上下文内容，则条件存储指令永远失败。

基本构件

- 例：使用 x4 中数值原子交换 x1 指定地址的内容

```
try:  mov x3,x4       ; mov exchange value
      lr x2,x1        ; load reserved from 0(x1)
      sc x3,0(x1)     ; store conditional
      bnez x3,try     ; branch store fails
      mov x4,x2       ; put load value in x4?
```

- 例：原子读取且递增

```
try:  lr x2,x1        ; load reserved from 0(x1)
      addi x3,x2,1    ; increment
      sc x3,0(x1)     ; store conditional
      bnez x3,try     ; branch store fails
```

使用一致性实现锁

- 自旋锁(spin locks)
  - 处理器不断尝试获取的“锁”，在循环中自旋直到成功。

- 无缓存一致性情形下（锁在内存中）的实现

```
        addi  x2,R0,#1
lockit: EXCH  x2,0(x1)    ; atomic exchange
        bnez  x2,lockit   ; already locked?
```

- 缓存一致性情形下（锁在缓存中）的实现

```
lockit: ld    x2,0(x1)    ; load of lock
        bnez  x2,lockit   ; not available-spin
        addi  x2,R0,#1    ; load locked value
        EXCH  x2,0(x1)    ; swap
        bnez  x2,lockit   ; branch if lock wasn't 0
```

内存一致性模型

- 缓存一致性保证多个处理器看到一致的内存视图(view of memory)。
  - 问题：处理器必须按照什么顺序看到另一个处理器写入的数据
- 两颗处理器运行的代码段

```
P1:    A=0                          P2:    B=0
       …                                   …
       A=1                                 B=1
L1:    if (B==0) …                  L2:    if (A==0) …
```

  - 假设：**A** 和 **B** 被两颗处理器缓存，且初值都是 0；
  - 如果写操作总是立即有效且被另一颗处理器立即看到，则两条 **IF** 语句的条件都不会为 **true**。
  - 假设写失效(write invalidate)存在延迟，而在延迟期间允许继续执行。那么，存在双方在读取数值之前均未看到对方失效的可能性。
  - 问题：是否允许上述情况？在什么条件下允许？

内存一致性模型

- 顺序一致性(sequential consistency)
  - 任何执行结果都是相同的，就好像在每颗处理器上执行的内存访问都是有序的，且不同处理器之间的内存访问是任意交错的。
  - 顺序一致性保证了上例中初始化 `IF` 语句之前必须完成赋值语句
- 最简单实现方法
  - 要求处理器延迟完成任何内存访问，直到由其引发的所有失效操作完成为止。
  - 降低了性能，特别是对大数目处理器构成的多处理器系统。
- 两个方向
  - 保留顺序一致性，利用延迟隐藏技术来减少惩罚
  - 较少约束的顺序一致性模型，允许使用更快的硬件，但影响程序员对多处理器的视图。

程序员视图

- 虽然顺序一致性模型的性能较差，但从程序员视角来看具有简单化的优点。
  - 要求：编程模型应该易于解释，但高性能执行
- 概念：程序是同步的(synchronized)
  - 如果通过同步操作，对共享数据所有访问都是有序的，则程序是同步的
  - 例：
    - 通过一对同步操作将某颗处理器对变量的写操作和其它处理器对该变量的访问（读或写操作）分开；
    - 一个同步操作（解锁）在写处理器写操作之后执行
    - 另一个同步操作（加锁）在另一颗处理器访问之前执行
  - 程序是同步的，也称为是无数据争用的(data-race-free)。
- 程序员可通过构造自己的同步机制来保证有序性（不建议）
  - 棘手的问题，容易出bug，且体系结构上不支持

松弛一致性模型

- 思路
  - 允许读和写操作乱序完成
  - 使用同步操作来强制有序化

- 规则 $X \rightarrow Y$
  - 操作 X 必须在操作 Y 工作之前完成

- 顺序一致性需要保持所有可能的顺序，共有四种：
  - $R \rightarrow W$
  - $R \rightarrow R$
  - $W \rightarrow R$
  - $W \rightarrow W$

松弛一致性模型

- 松弛模型
  - 松弛 $W \rightarrow R$：总体存储有序(total store ordering)
  - 松弛 $W \rightarrow R$ 和 $W \rightarrow W$：部分存储有序(partial store ordering)
  - 松弛所有四种顺序：弱有序(weak ordering)或释放一致性(release consistency)

- 释放一致性(release consistency)
  - S_A：获取(acquire)对共享变量访问权限的同步操作
  - S_R：释放(release)对象以允许其它处理器获取访问权限的同步操作

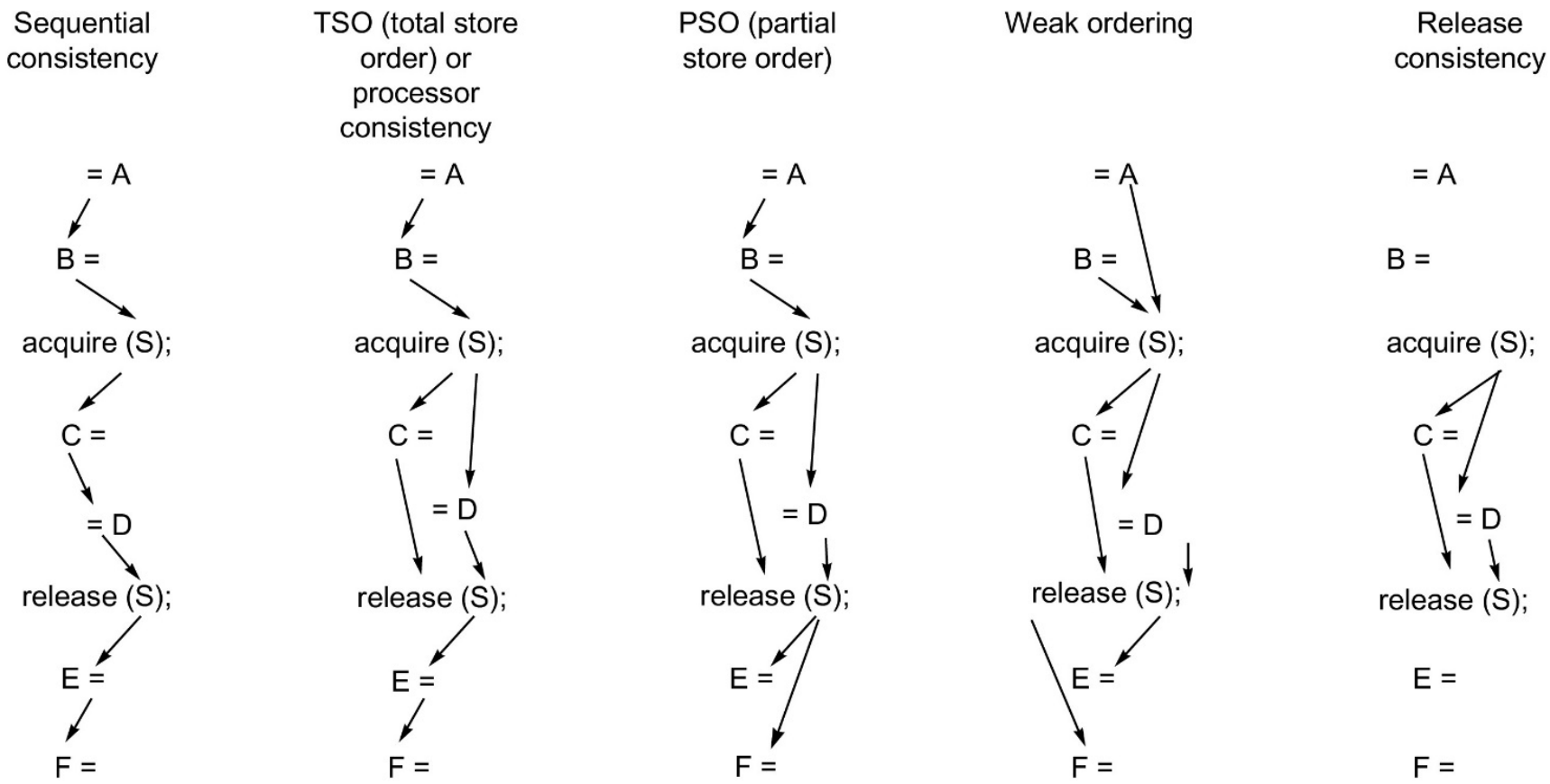
松弛一致性模型

- 各种一致性模型的强制顺序

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Partial store order	SPARC	$R \rightarrow R$, $R \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Weak ordering	PowerPC		$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Release consistency	MIPS, RISC V, Armv8, C, and C++ specifications		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

松弛一致性模型

- 五种一致性模型的示例
  - 读：= A，写：B =



几个问题

- 编译优化与一致性模型
  - 一致性模型确定了可以对共享数据进行合法编译优化的范围。
  - 没有明确定义的同步位置，编译器不能交换两个不同共享数据项的读和写操作，以免影响程序的语义。
  - 编译器是否能够从更加松弛的一致性模型中获取明显的优势仍然是个有待研究的课题。

- 在严格一致性模型中使用推测隐藏延迟
  - 推测可以用于隐藏严格一致性模型引起的延迟，给松弛一致性模型提供许多好处。
    - 如果处理器在内存访问提交之前获得内存访问失效的信息，就可以使用推测恢复来退出计算，从地址失效的内存访问处重新开始。
  - 尚待解决的问题：编译技术如何成功地优化对共享变量的内存访问

几个问题

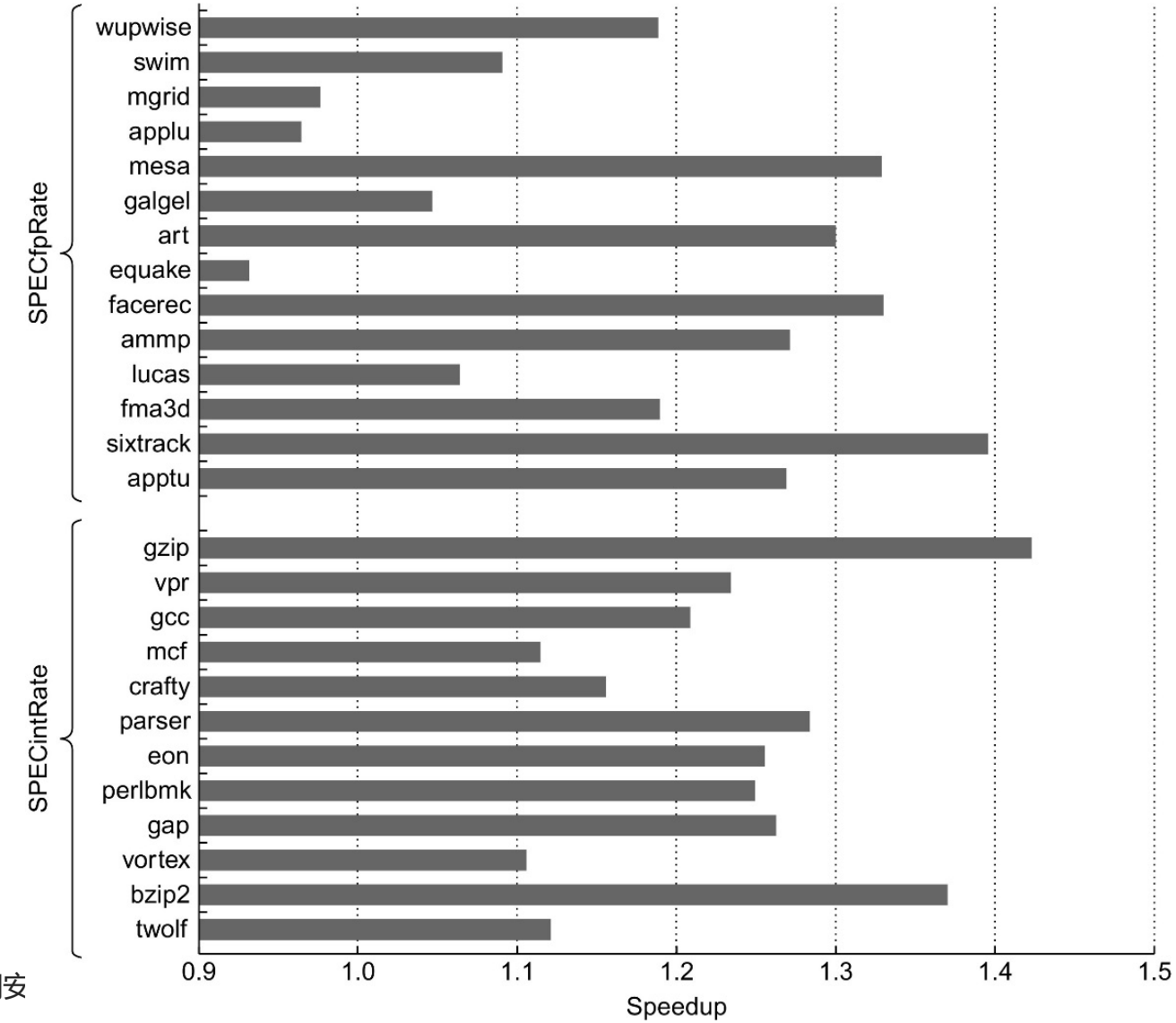
- 包含(inclusive)与它的实现
  - 所有多处理器都使用多级缓存层次(multilevel cache hierarchies)来降低对全局互联的性能需求和减少缓存不命中的延迟。
  - 多层包含(multilevel inclusive)：缓存层次中的每一层都是更远离处理器缓存层的子集。
  - 拥有多层包含的缓存可以减少一致性流量和处理器流量之间的争用。
    - 处理器流量：当监听和处理器缓存访问必须竞争缓存时发生
  - 为了维持多种块容量下的包含性，当在低层发生替换时，必须查看高层以确保在低层替换的任何数据字在高层中都是失效的。

几个问题

- 例：$L2$ 块容量是 $L1$ 的四倍，当 $L1$ 和 $L2$ 不命中时将会怎样？
  - $L1$ 和 $L2$ 采用直接映射方法，$L1$ 块容量是 b 字节，$L2$ 是 $4b$ 字节；
  - $L1$ 包含 x 和 $x+b$ 两个块，且 $x \ mod \ 4b = 0$，那么，$L2$ 中一个块包含 $L1$ 中四个块 $x, x+b, x+2b, x+3b$；
  - 处理器访问映射到块 x 中的地址 y 引起两层均不命中，$L2$ 读取 $4b$ 字节且替换 $x, x+b, x+2b, x+3b$，$L1$ 读取 b 字节且替换 x；
  - $L1$ 包含未替换的块 $x+b$，而 $L2$ 没有，包含性被破坏。

多处理和多线程带来的性能收益

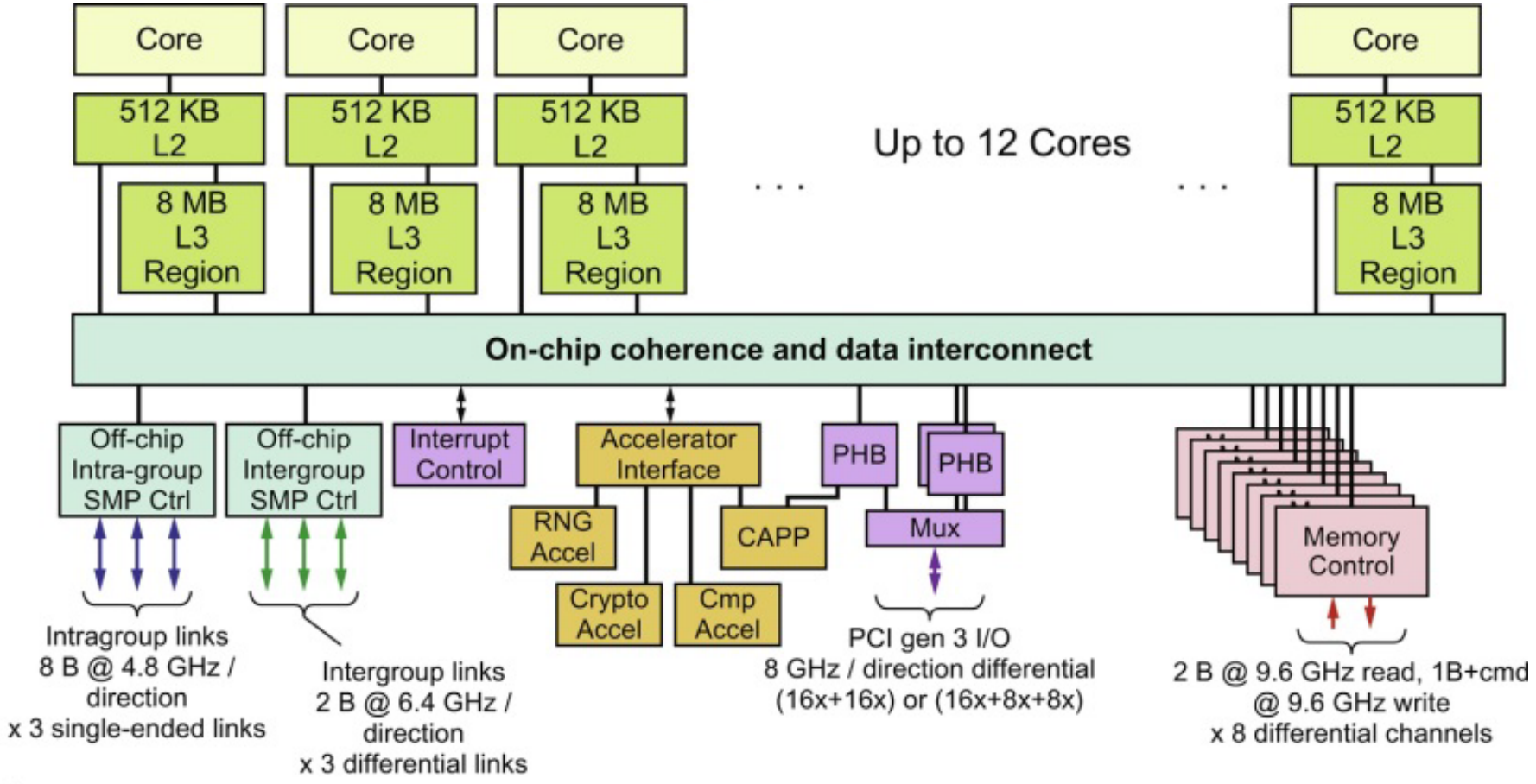
- 同步多线程与单线程的性能比较(IBM eServer p5 575)



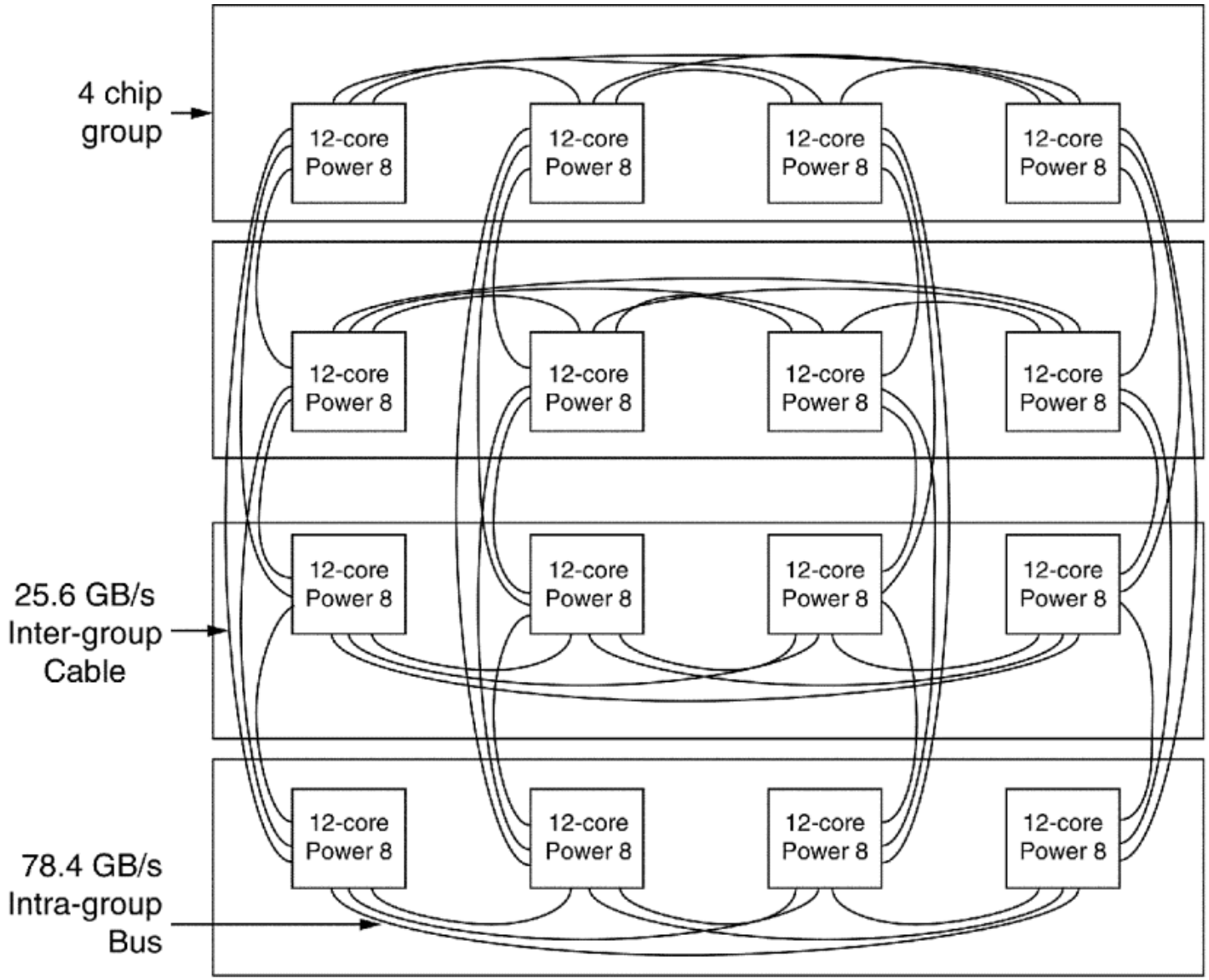
高端多核处理器的性能

Feature	IBM Power8	Intel Xeon E7	Fujitsu SPARC64 X+
Cores/chip	4, 6, 8, 10, 12	4, 8, 10, 12, 22, 24	16
Multithreading	SMT	SMT	SMT
Threads/core	8	2	2
Clock rate	3.1–3.8 GHz	2.1–3.2 GHz	3.5 GHz
L1 I cache	32 KB per core	32 KB per core	64 KB per core
L1 D cache	64 KB per core	32 KB per core	64 KB per core
L2 cache	512 KB per core	256 KB per core	24 MiB shared
L3 cache	L3: 32–96 MiB: 8 MiB per core (using eDRAM); shared with nonuniform access time	10–60 MiB @ 2.5 MiB per core; shared, with larger core counts	None
Inclusion	Yes, L3 superset	Yes, L3 superset	Yes
Multicore coherence protocol	Extended MESI with behavioral and locality hints (13-states)	MESIF: an extended form of MESI allowing direct transfers of clean blocks	MOESI
Multichip coherence implementation	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory
Multiprocessor interconnect support	Can connect up to 16 processor chips with 1 or 2 hops to reach any processor	Up to 8 processor chips directly via Quickpath; larger system and directory support with additional logic	Crossbar interconnect chip, supports up to 64 processors; includes directory support
Processor chip range	1–16	2–32	1–64
Core count range	4–192	12–576	8–1024

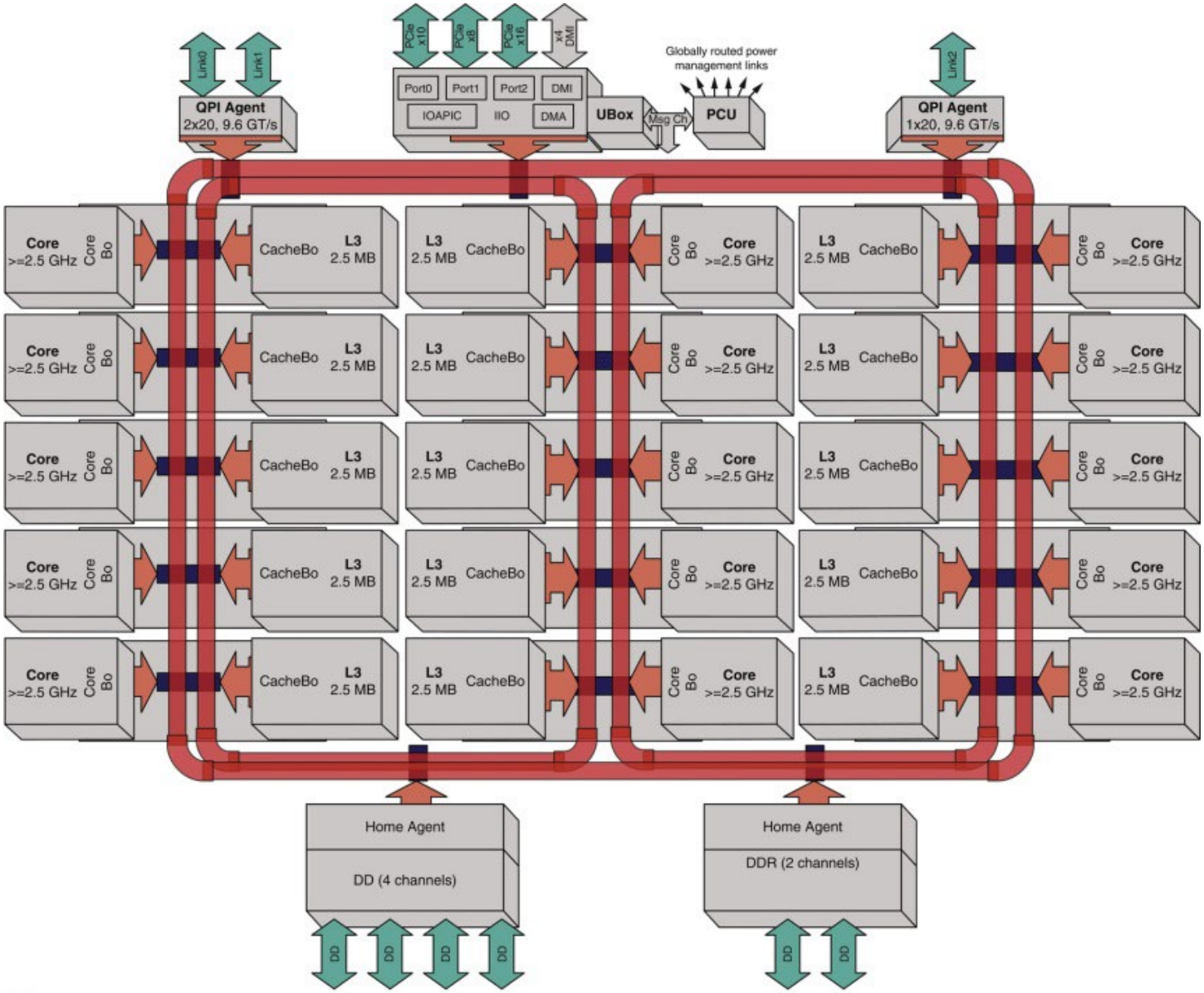
IBM Power8 芯片组成



Power8 多片互联

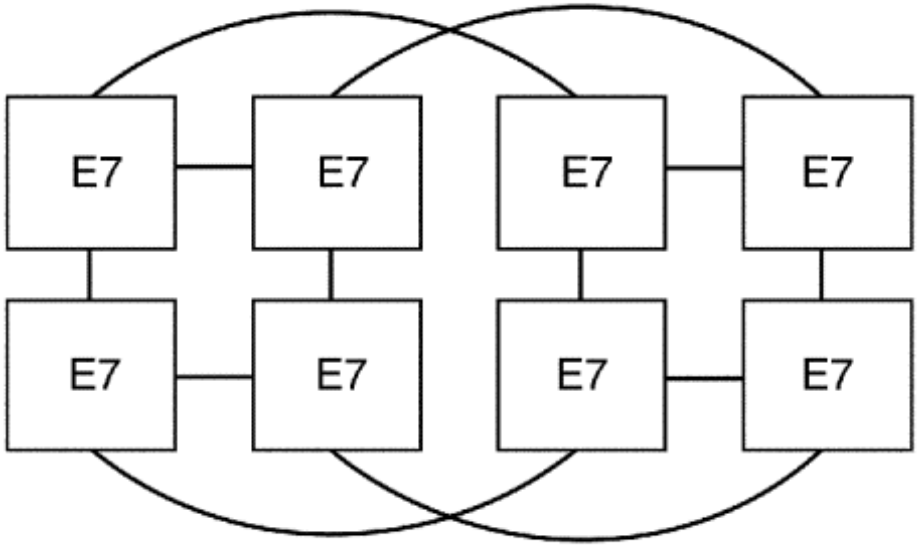

4 chip group
25.6 GB/s Inter-group Cable
78.4 GB/s Intra-group Bus
12-core Power 8

Intel Xeon E7



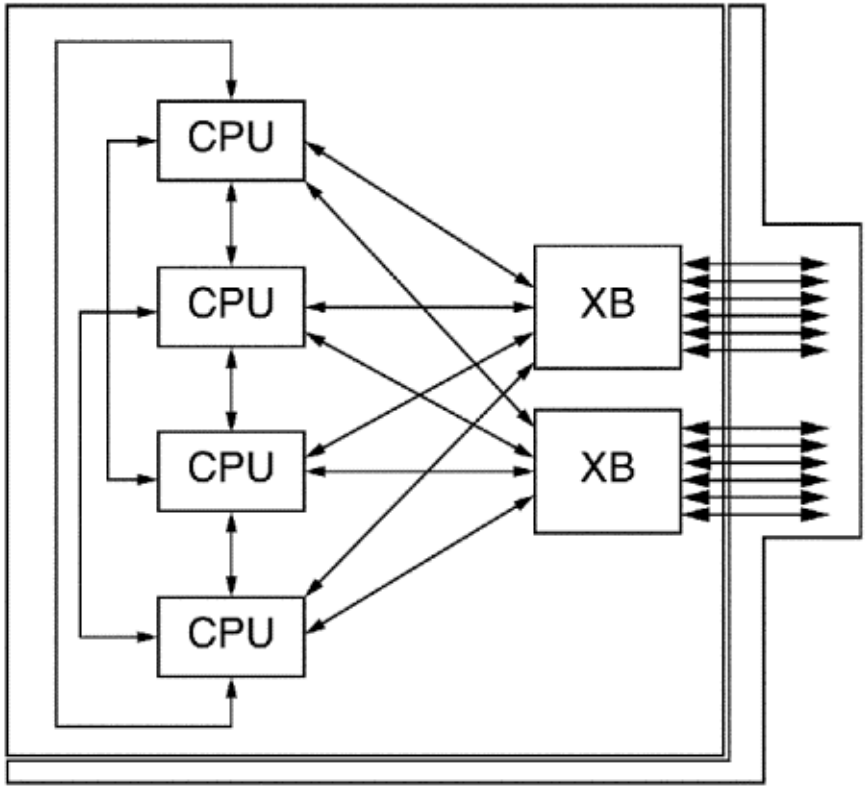
Xeon E7 多片互联

- 最新的 4 片多处理器具有128 个核
- 三个 QPI 链分别连接三个近邻，获得 4 片全连接多处理器
- 两个 QPI 链分别连接两个近邻，第三个 QPI 连接纵横交换网络，形成更大的多处理器系统



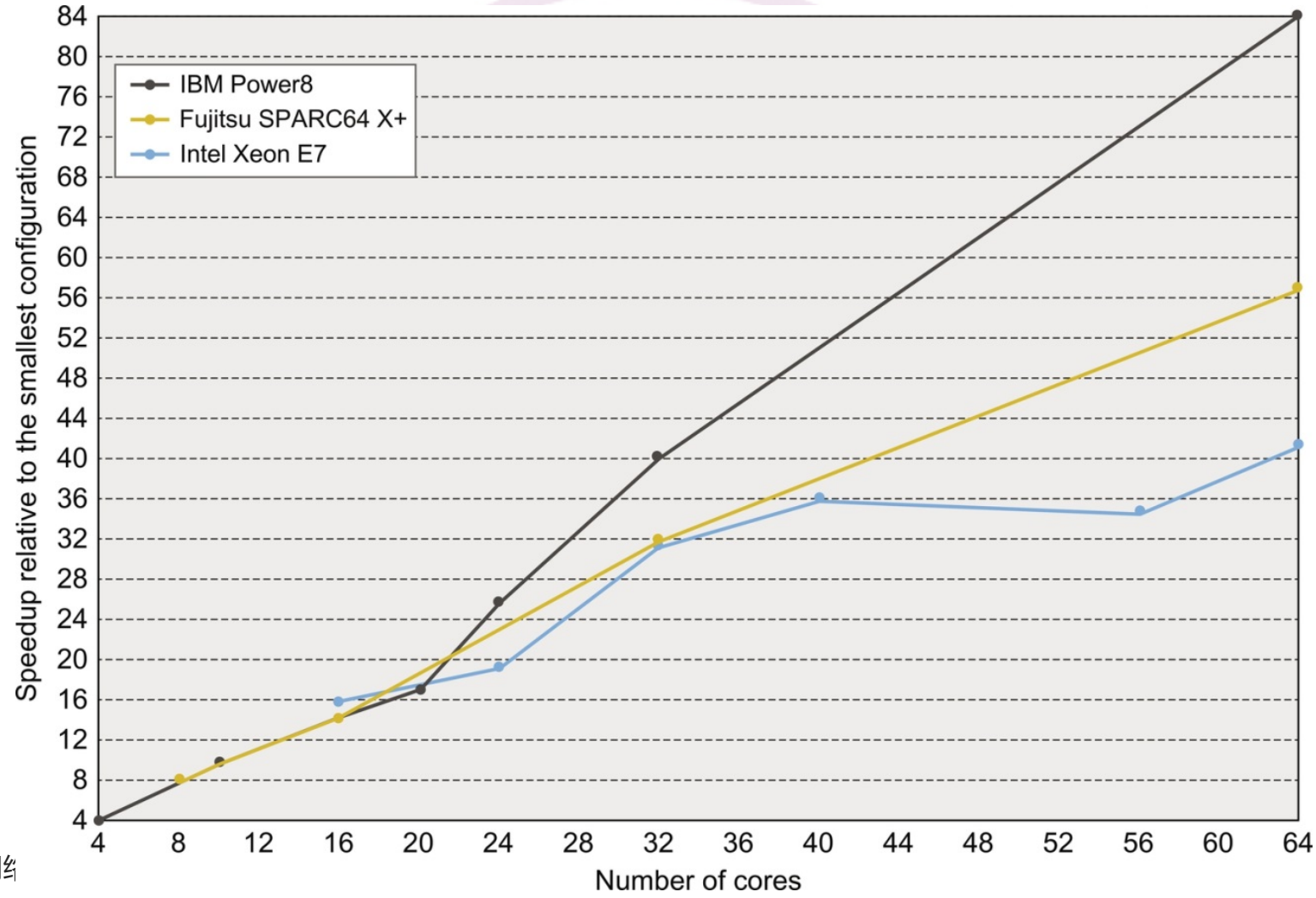
SPARC64 X+多片互联

- SPARC64 X+ 使用 4 处理器模块
  - 每颗处理器的三个链连接近邻，其它两个链连接纵横交换网络



多核处理器的性能

- 随着核数目增加，相对最小配置的加速比
  - 注：该表中三种处理器之间没有可比性（基准配置不同）



诚信 创新 实践