《计算机系统结构》课程直播
2020. 5.21

听不到声音请及时调试声音设备，可以下课后补签到

请将ZOOM名称改为“姓名”；

本节内容

- 线程级并行性
  - 高速缓存一致性
  - 存储一致性模型

From : H&P Computer Architecture: A Quantitative Approach,
Fifth Edition, （5th edition)

高速缓存一致性与假共享： 例题

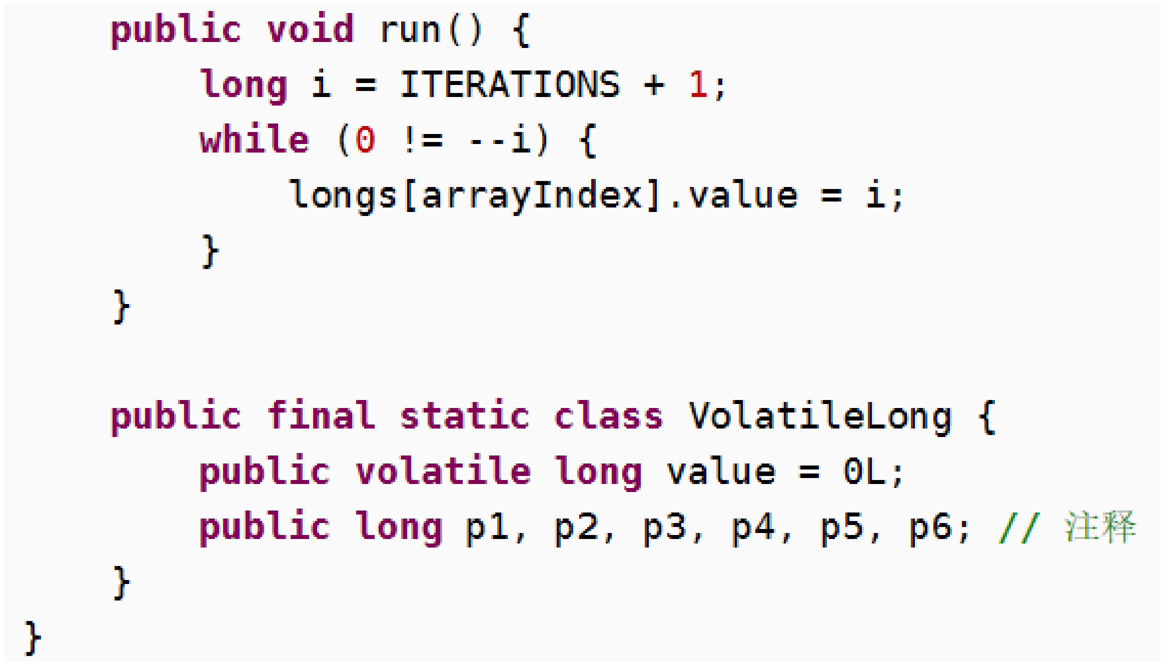
- 如下代码在SMP（shared memory multiprocessors）环境下执行，sum和sum_local是全局变量，被NUM_THREADS个线程所共享：

```
double  sum=0.0,  sum_local[NUM_THREADS];
```

```
#pragma omp parallel num_threads(NUM_THREADS)
//由NUM_THREADS个线程执行以下相同的代码段
  {  int me = omp_get_thread_num();
     sum_local[me] = 0.0;
     #pragma omp for  //并行for语句，不同线程处理部分数据
     for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];  //将结果存入对应该线程的sum_local元素中
    #pragma omp atomic        //并行原子操作，
      sum += sum_local[me];  //求总和
  }
```

例题2

- https://www.iteye.com/blog/coderplay-1486649
- 博客：从Java视角理解伪共享(False Sharing)



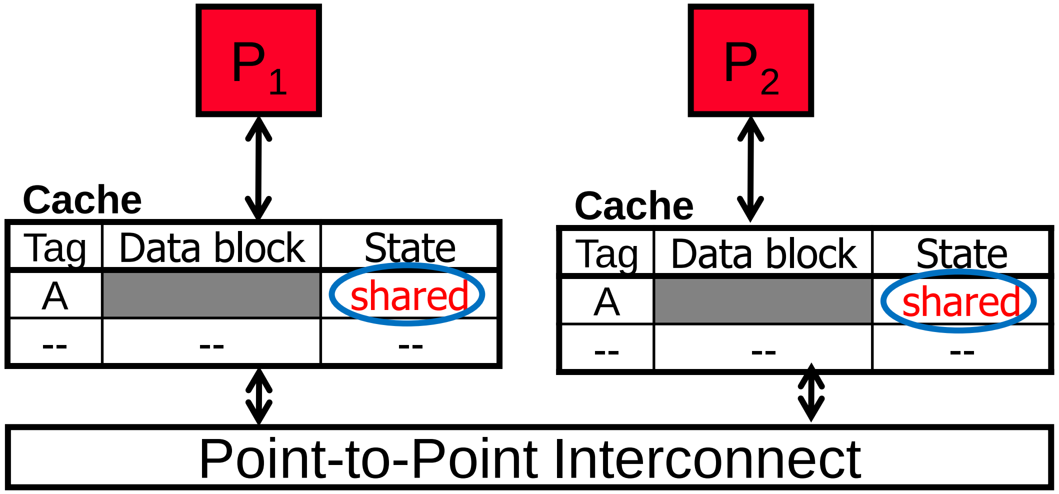
```
    public void run() {
        long i = ITERATIONS + 1;
        while (0 != --i) {
            longs[arrayIndex].value = i;
        }
    }

    public final static class VolatileLong {
        public volatile long value = 0L;
        public long p1, p2, p3, p4, p5, p6; // 注释
    }
}
```

高速缓存一致性和假共享

假设:

- P1写一个数据块内的第i个字
- P2写同一块内的第k个字

会发生什么？

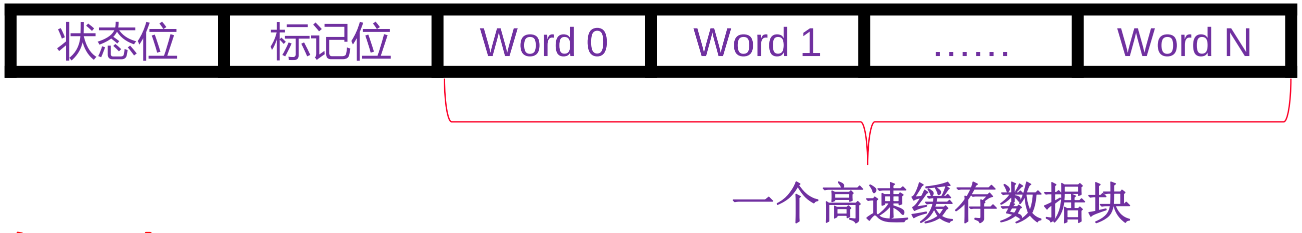


初始时，P1和P2共享一个数据块，

私有cache中的状态都是shared

高速缓存一致性和假共享

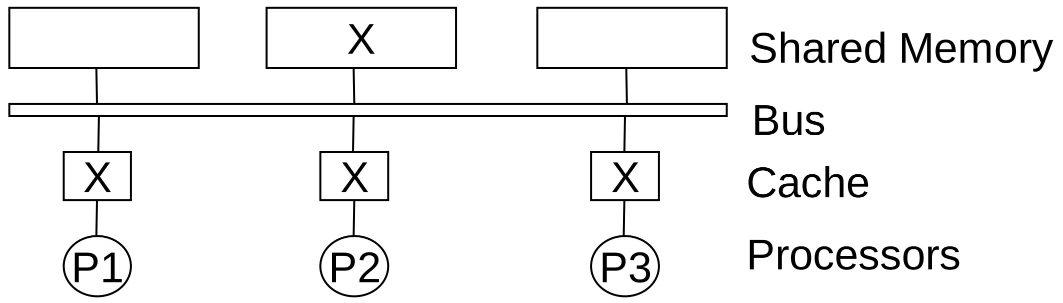
- 高速缓存一致性协议以数据块为单位，而不是以字为单位
- 一个高速缓存数据块包含的字数多于1



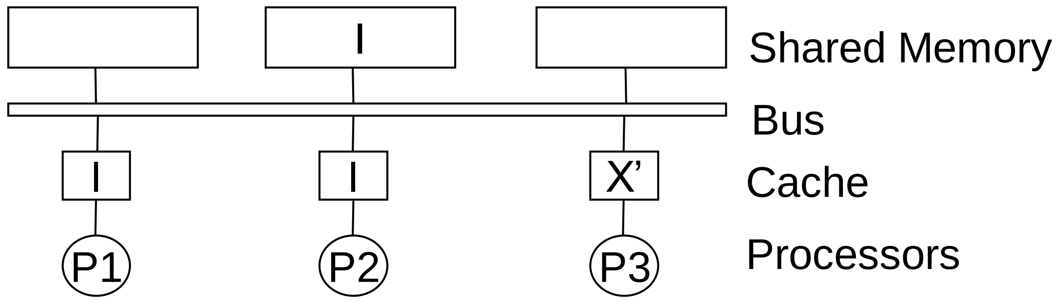
假共享:

- 当两个或更多处理器共享同一个数据块的不同部分时是假共享

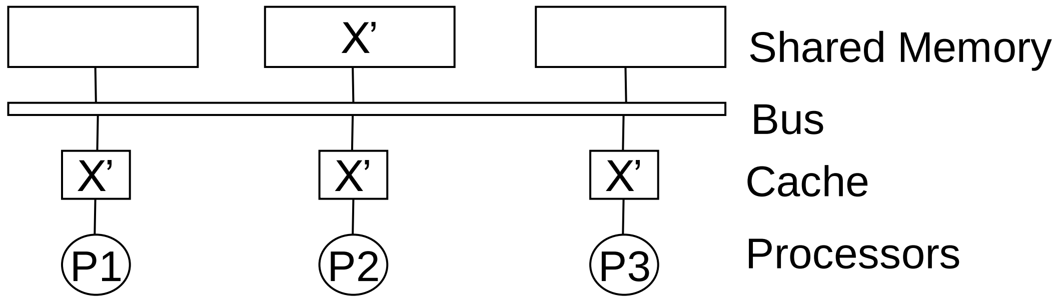
Bus Based Snooping Protocol



X is a shared variable that has a copy in all caches. Then a write occurred



For Write Invalidate all the cache copies are marked as "invalid" except the most recent one



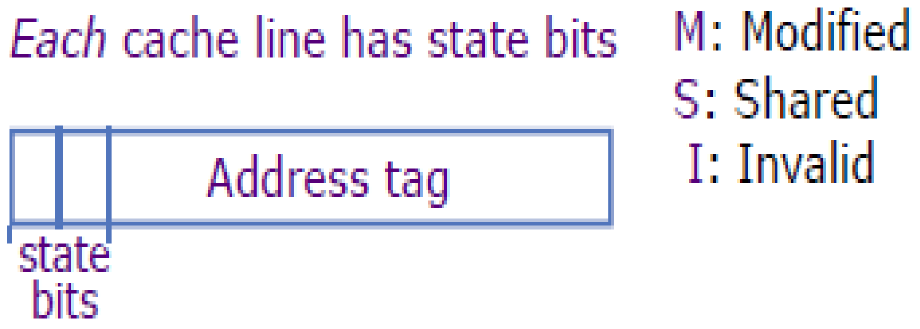
For **Write Update** all the cache copies are updated with the most recent value

Assume a write through cache protocol

The MSI Protocol

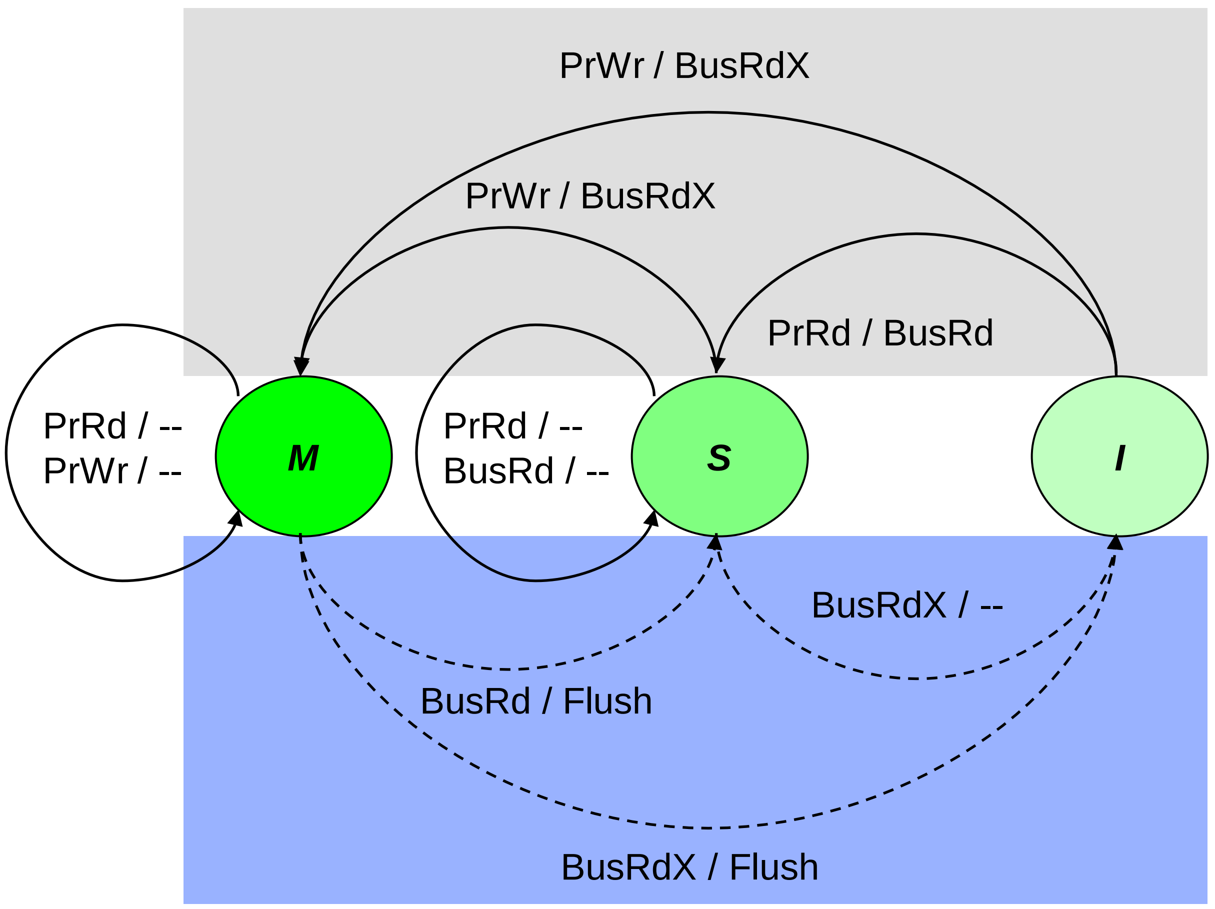
- Three states to differentiate between clean or dirty
  - Modified, Shared and Invalid
- Two types of Processor actions and Three types of bus's signals
  - Processor Writes and Reads
  - Bus Read, Bus Read Exclusive and Bus Write Back

MSI States



- Modified
  - The cached copy is the only valid copy in the system.
  - Memory is stale.
- Shared
  - The cached copy is valid and it may or may not be shared by other caches.
    - Initial state after first loaded.
  - Memory is up to date.
- Invalid
  - The cached copy is not existence.

MSI Protocol State Machine



Input / Output	
PrWr	Processor Write
PrRd	Processor Read
BusRd	Bus Read
BusRdX	Read to own
Flush	Flush to memory
--	No Action

MSI Example

Process or Action	**State P1**	**State P2**	**State P3**	**Bus Action**	**Data Supplied by**
P1 loads u	S	_	_	BusRd	Mem
P3 loads u	S	_	S	BusRd	Mem
P3 stores u	I	_	M	BusRdX	Mem
P1 loads u	S	_	S	BusRd	P3 c
P2 loads u	S	S	S	BusRd	Mem

Snoopy Coherence Protocols

Complications for the basic MSI protocol:

- Operations are not atomic
  - E.g. detect miss, acquire bus, receive a response
  - Creates possibility of deadlock and races
  - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate

Extensions:

- Add exclusive state to indicate clean block in only one cache (MESI protocol)
  - Prevents needing to write invalidate on a write

- Owned state (MOESI protocol, AMD line of multi core personal computers and servers.)

The MESI Protocol

States:

- Modified, Exclusive, Shared and Invalid

Due to Goodman [ISCA’93]

State transitions are due to:

- Processor actions: This being Write or Reads
- Bus operations caused by the former

Implemented in Intel Pentium Pro (in some modes)

MESI States

Modified

- Main Memory’s value is stale
- No other cache possesses a copy

Exclusive

- Main Memory’s value is up to date
- No other cache possesses a copy

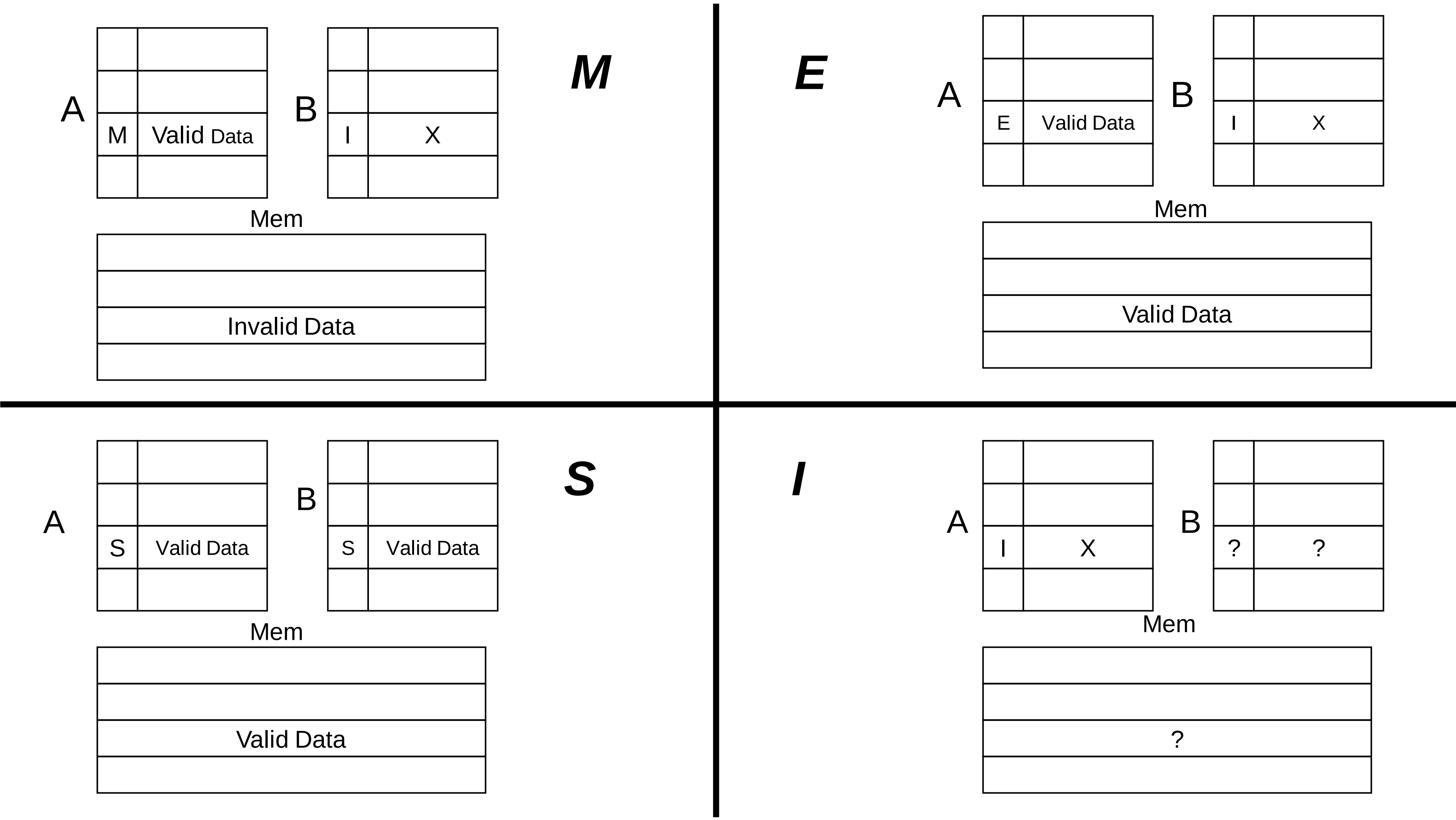
Shared

- Main Memory’s value is up to date
- Other caches have a copy of the variable

Invalid

- This cache have a stale copy of the variable

MESI States


M
A
M
Valid Data
B
I
X
Mem
Invalid Data
E
A
E
Valid Data
B
I
X
Mem
Valid Data
S
A
S
Valid Data
B
S
Valid Data
Mem
Valid Data
I
A
I
X
B
?
?
Mem
?

Example: Two Processor System

Cache 1		Cache 2		Memory Transfer
Bus	State	Bus	State	
	I		I	Load into Cache 1
	I → E		I	

P1 → Load

Cache 1		Cache 2		Memory Transfer
Bus	State	Bus	State	
	E		I	Load into Cache 2
Rd Hit	E → S		I → S	

P2 → Load

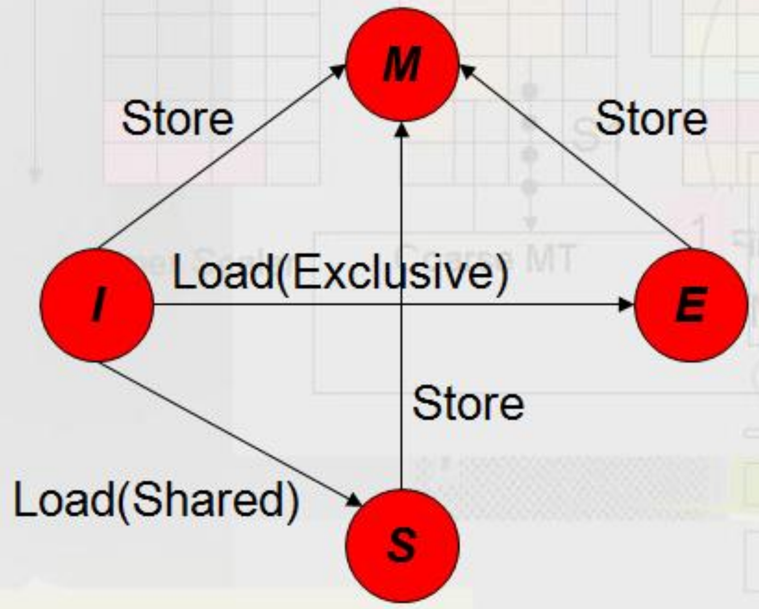
Cache 1		Cache 2		Memory Transfer
Bus	State	Bus	State	
	S		S	
	S →M	Inv	I	

P1 → Store

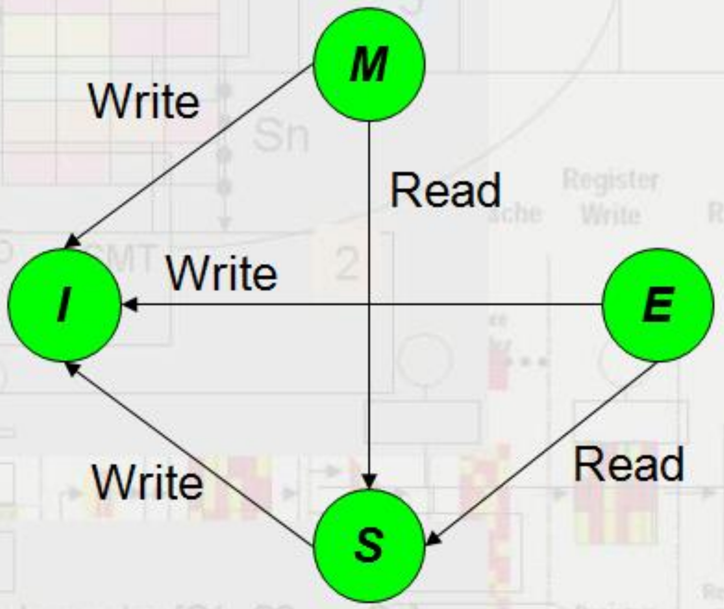
Cache 1		Cache 2		Memory Transfer
Bus	State	Bus	State	
	M		I	
	M		I	Store from Cache 1
Rd Hit	M → S		I	Load into Cache 2
	S		I → S	

P2 → Load (first abort and then try again)

Processor Activities V.S. Bus Signals



$$\begin{tikzcd}
& M & \\
I \arrow[ur, "Store"] \arrow[rr, "Load(Exclusive)"] \arrow[dr, "Load(Shared)"'] & & E \arrow[ul, "Store"'] \\
& S \arrow[uu, "Store"'] &
\end{tikzcd}$$

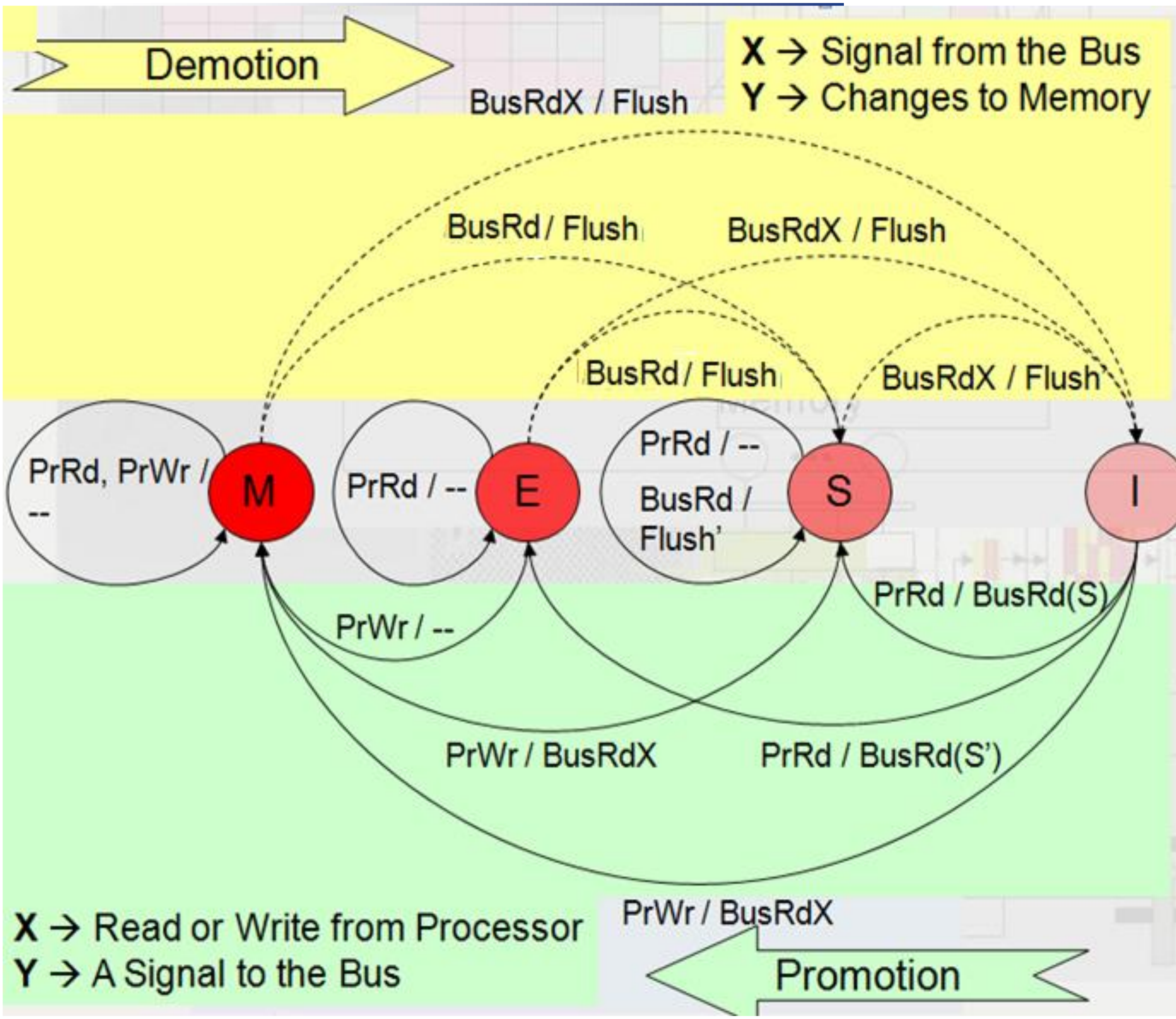


$$\begin{tikzcd}
& M \arrow[dl, "Write"'] \arrow[dd, "Read"] & \\
I & & E \arrow[ll, "Write"'] \arrow[dl, "Read"] \\
& S \arrow[ul, "Write"] &
\end{tikzcd}$$

Processor activities. Load (shard) means there are other caches that have copies of the loaded data. Load (exclusive) means this is the only copy.

Snooping activities. Read and Write are operations seen on the bus by the snooping logic.

MESI Protocol State Machine



X / Y

X is the input
Y is the Output

Inputs and Outputs

PrRd A Processor Read

PrWr A Processor Write

BusRdX
A Bus Read Exclusive. Request the data to be exclusive to this cache or demote it to a shared state

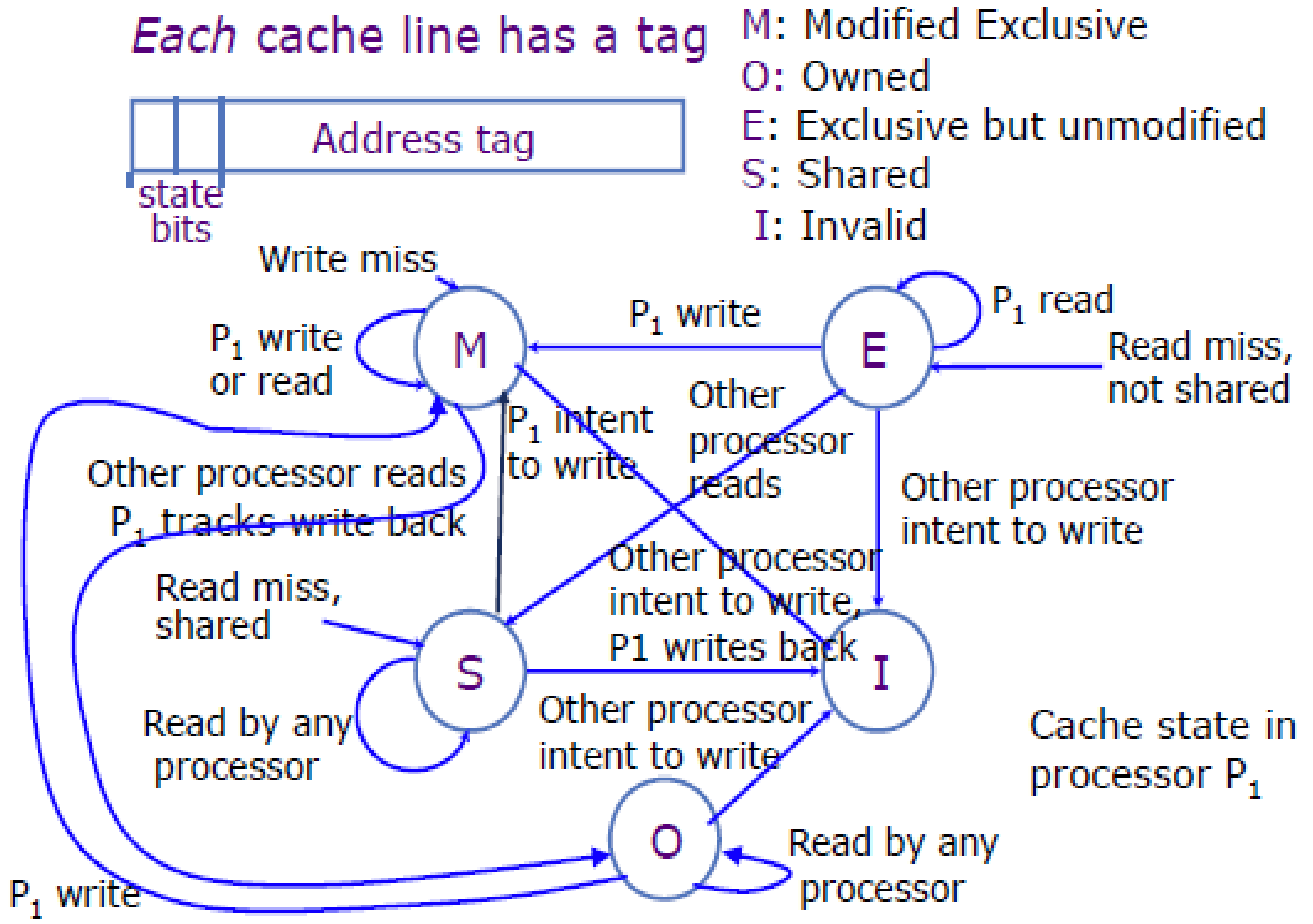
BusRd(S)
A Bus Read when the element is shared by another processor

BusRd(S')
A Bus Read when the element is not shared by another processor

Flush
Flush to either memory or a requesting processor (according to what sharing scheme is used)

--
No action or signal produced

MOESI (Used in AMD Opteron)



The Extra States Rationale

Exclusive (MSI to MESI)

–Reduce the number of busses transactions when a value is read exclusively and it may be modified in the future

Owned (MESI to MOESI)

–Reduce the number of busses transactions by delaying the update to the memory

MESI to MOESI, MESIF

	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes		Must writeback to share or replace
Exclusive	Clean	Yes	Yes	Yes	MSIF	Transitions to M on write
Shared	Clean	No	No	No	I	Does not forward
Invalid	NA	NA	NA	NA		Cannot Read
Forwarding	Clean	Yes	No	Yes	SI	Must invalidate other copies to write

	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes	O	Can share without writeback
Owned	Dirty	Yes	Yes	Yes		Must writeback to transition
Exclusive	Clean	Yes	Yes	Yes	MSI	Transitions to M on write
Shared	Either	No	No	No	I	Shared can be dirty or clean
Invalid	NA	NA	NA	NA		Cannot Read

	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes		Must writeback to share or replace
Exclusive	Clean	Yes	Yes	Yes	MSI	Transitions to M on write
Shared	Clean	No	No	Yes	I	Shared implies clean, can forward
Invalid	NA	NA	NA	NA		Cannot Read

Scalability Limitations of Snooping

- Caches
  - Bandwidth into caches
  - Tags need to be dual ported or steal cycles for snoops
  - Need to invalidate all the way to L1 cache

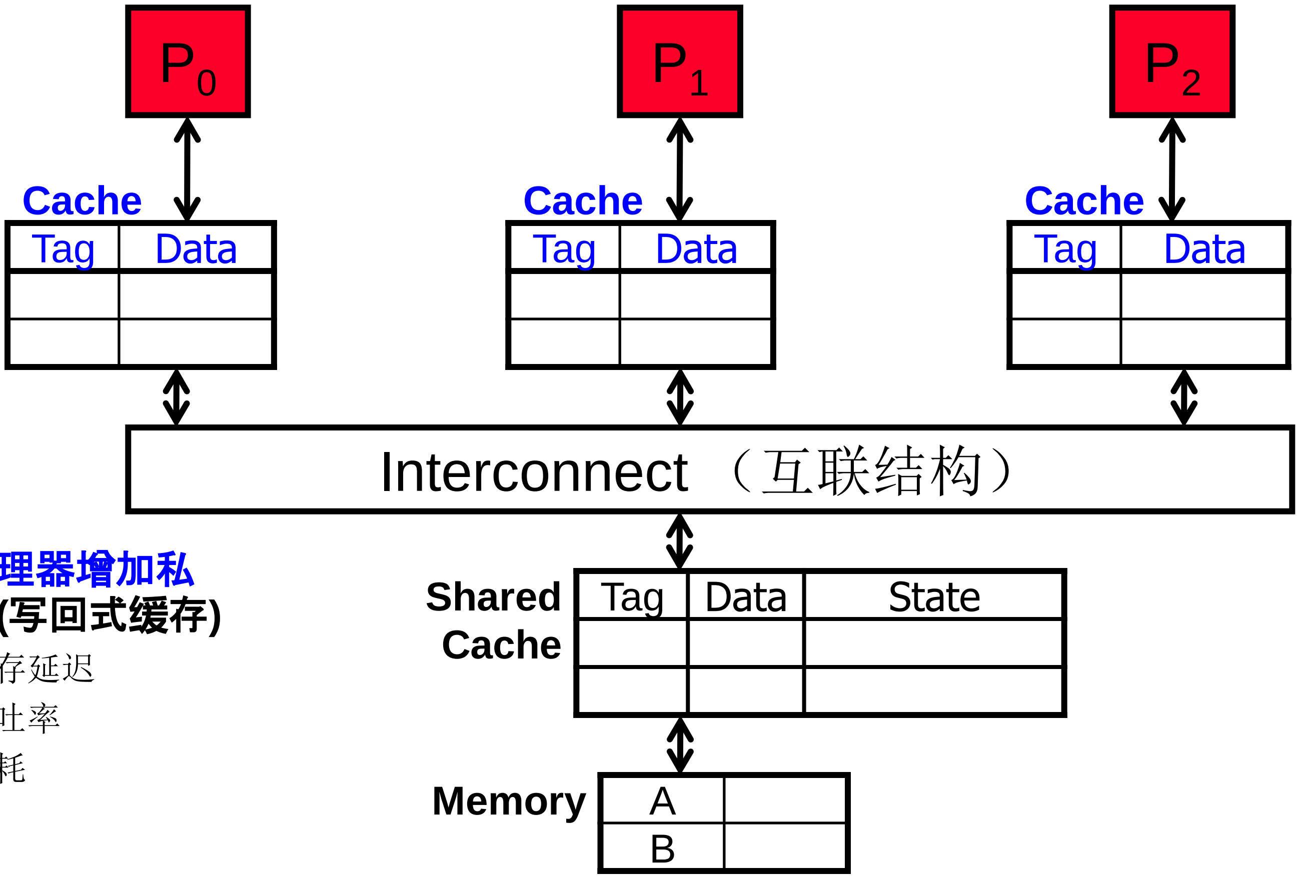
- Bus
  - Bandwidth
  - Occupancy (As number of cores grows, atomically
  - utilizing bus becomes a challenge)

Enforcing Coherence

Cache coherence protocols

- Directory based
  - Sharing status of each block kept in one location
- Snooping
  - Each core tracks sharing status of each block

增加私有高速缓存



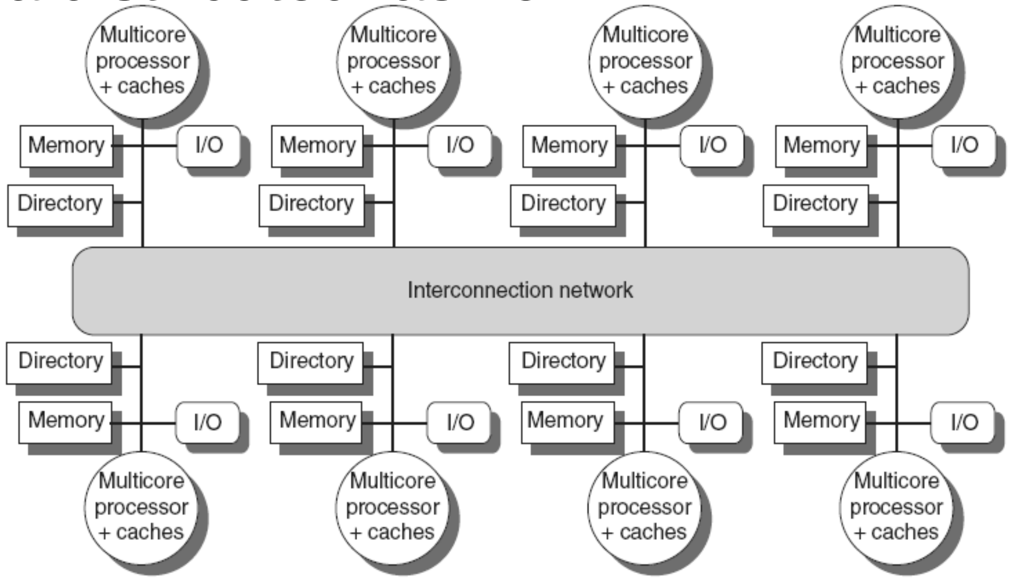
- **为每一个处理器增加私有高速缓存(写回式缓存)**
  - 降低访存延迟
  - 增加吞吐率
  - 减少能耗

Directory Protocols

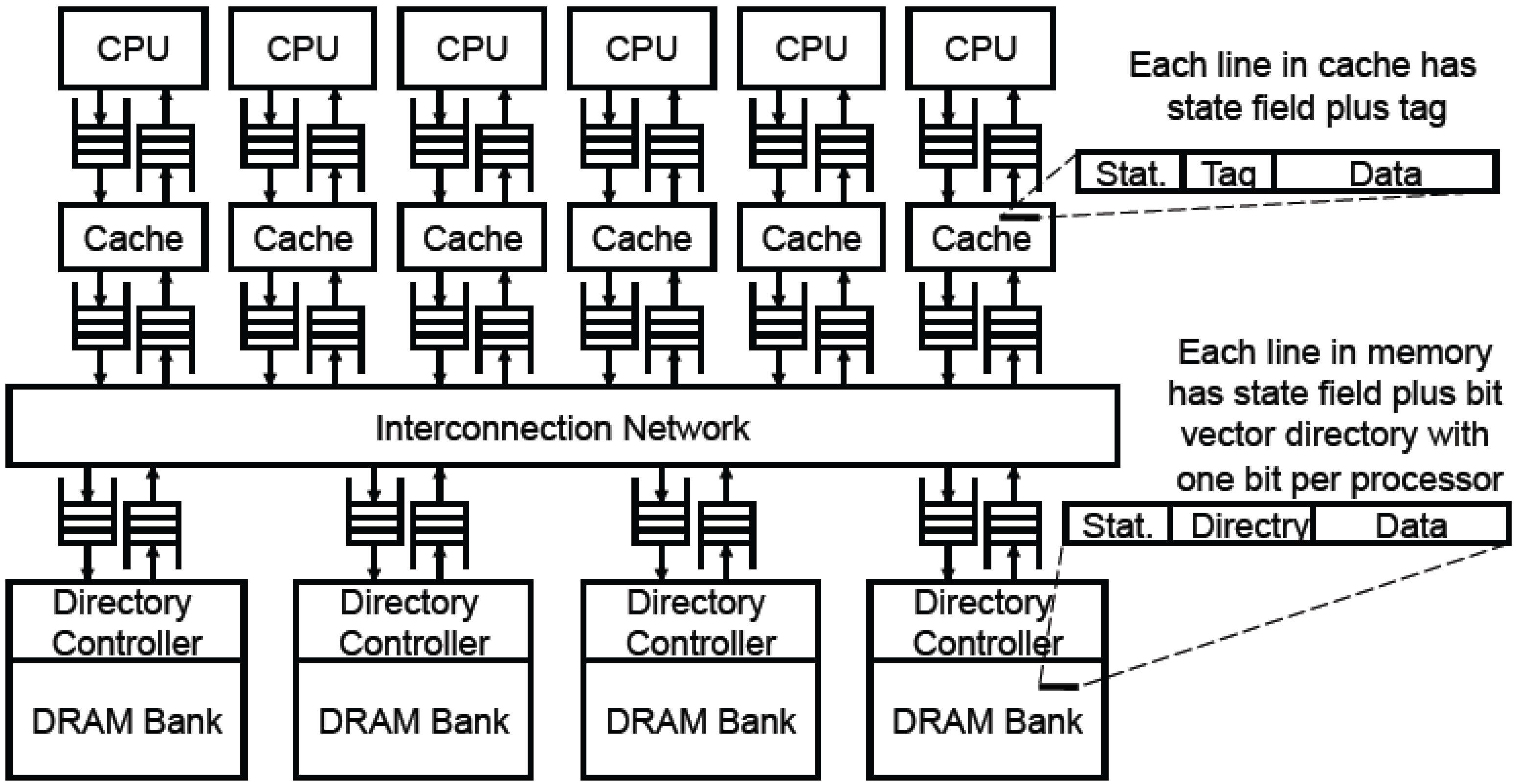
Directory keeps track of every block

- Which caches have each block
- Keep bit vector of size = # cores
- Status of each block

Implement in a distributed fashion:

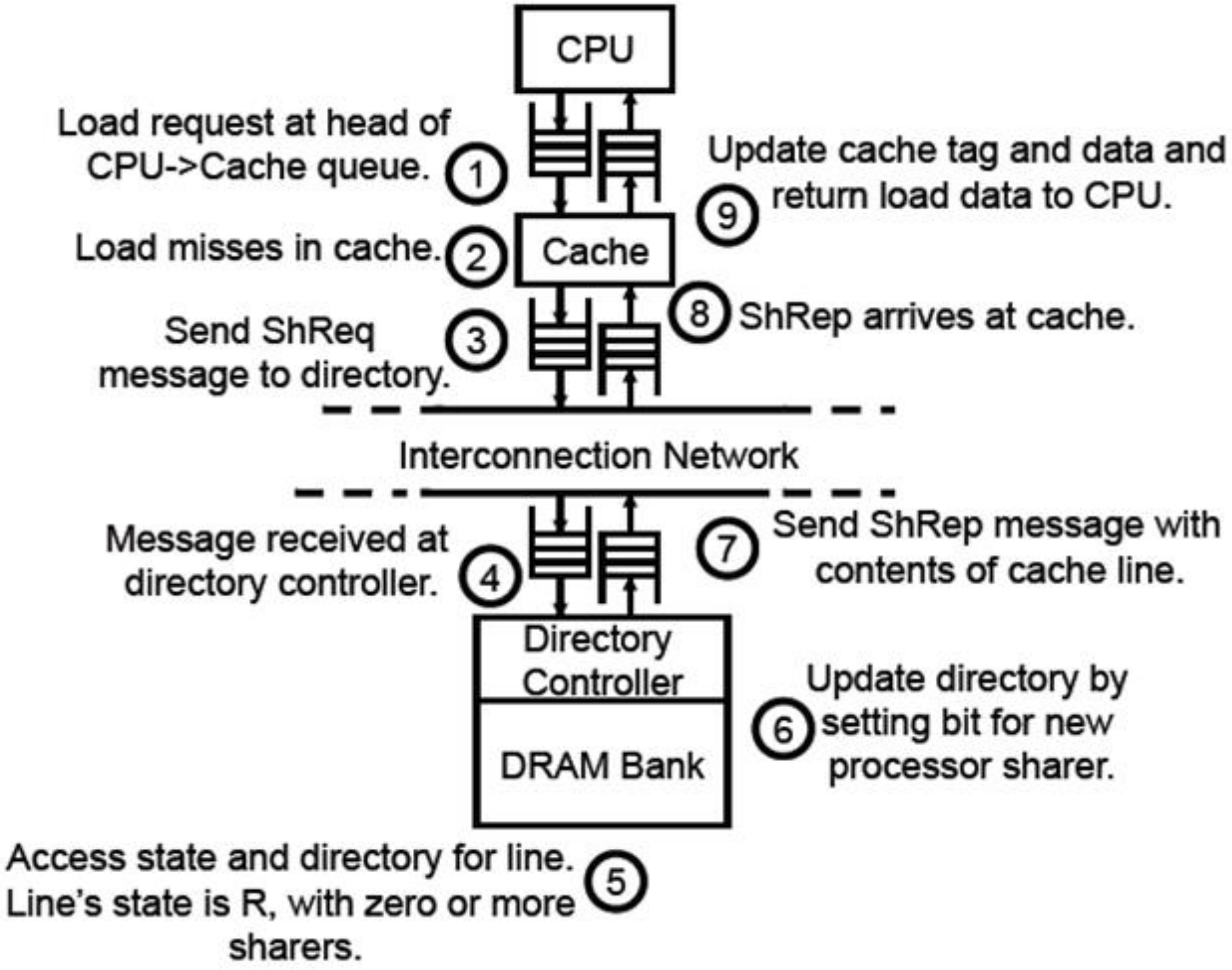


An directory example

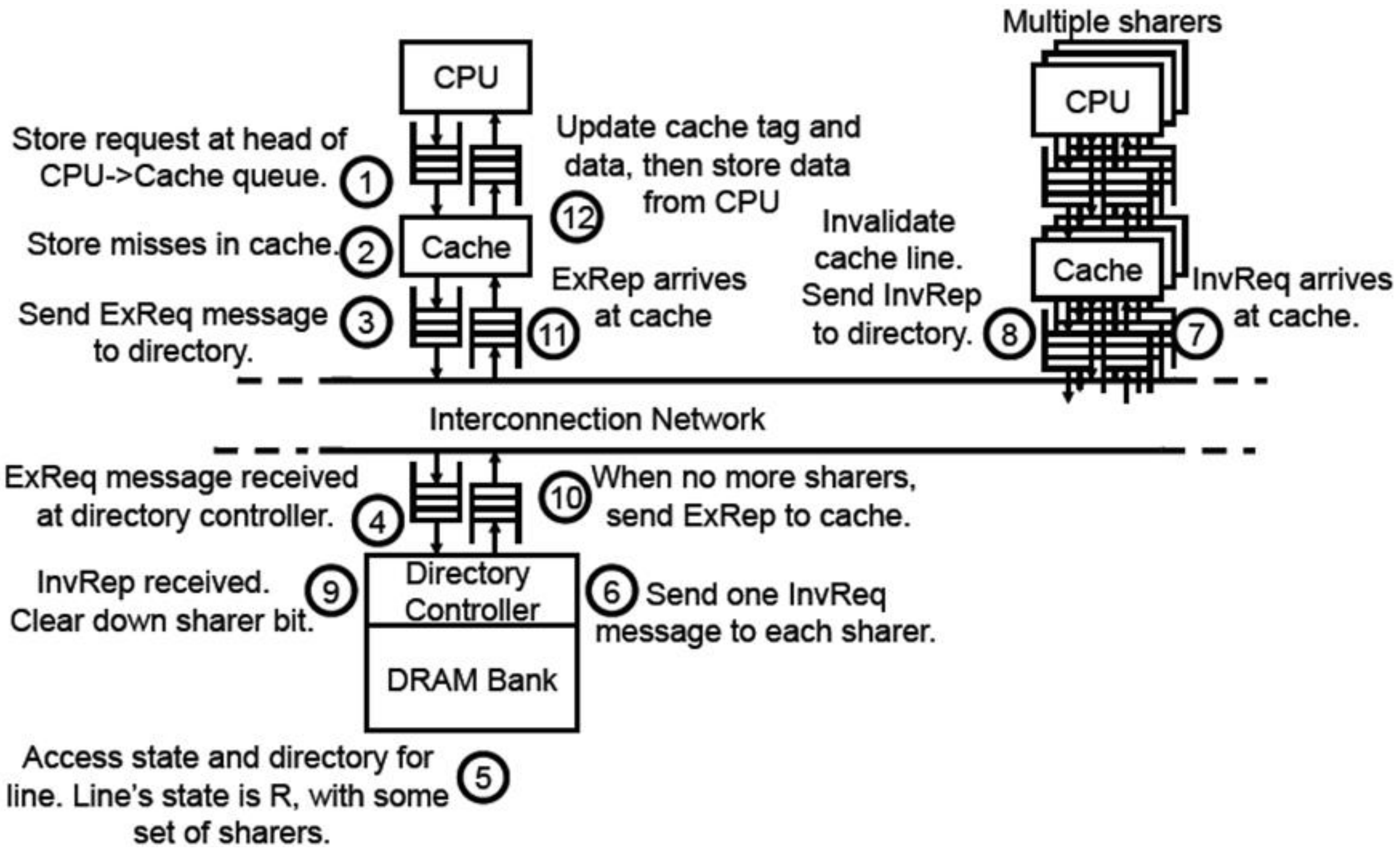


- Assumptions: Reliable network, FIFO message delivery between any given source-destination pair

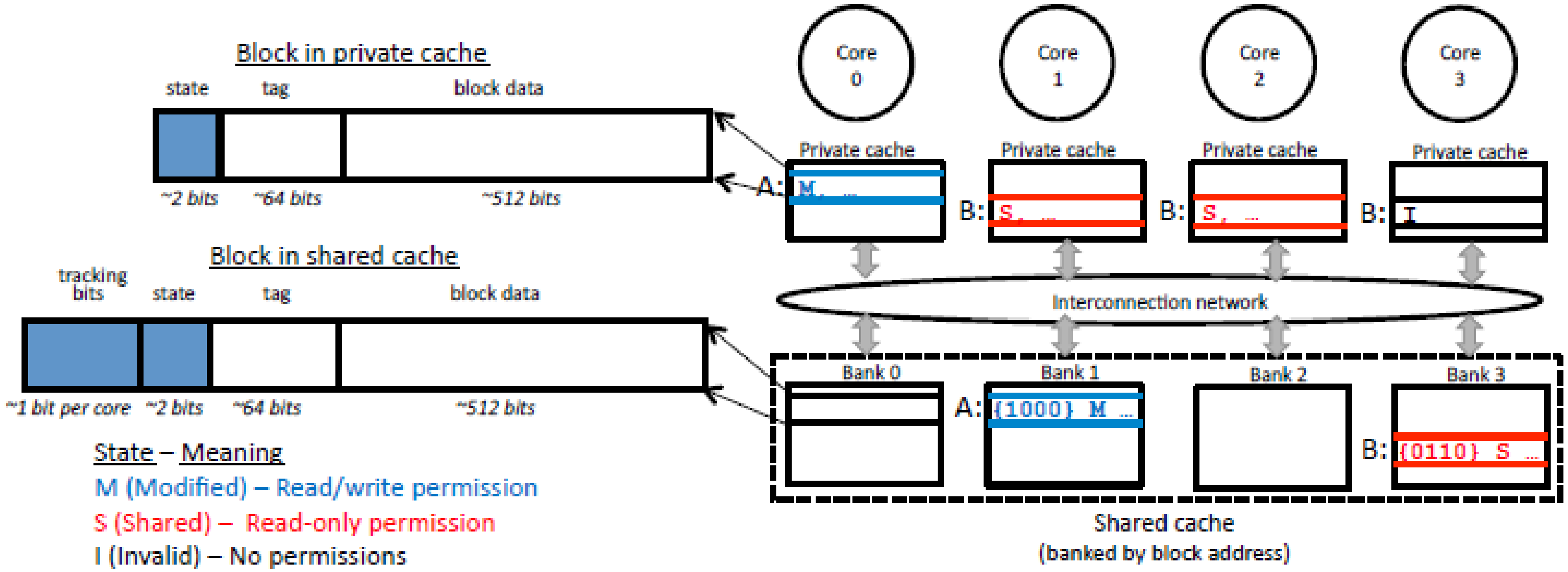
Read miss, to uncached or shared line


CPU
Load request at head of CPU->Cache queue.
1
Update cache tag and data and return load data to CPU.
9
Load misses in cache.
2
Cache
8
ShRep arrives at cache.
Send ShReq message to directory.
3
Interconnection Network
Message received at directory controller.
4
7
Send ShRep message with contents of cache line.
Directory Controller
DRAM Bank
Update directory by setting bit for new processor sharer.
6
Access state and directory for line. Line's state is R, with zero or more sharers.
5

Write miss, to read shared line


Multiple sharers
CPU
Cache
CPU
Cache
Store request at head of CPU->Cache queue. 1
Store misses in cache. 2
Send ExReq message to directory. 3
Update cache tag and data, then store data from CPU 12
ExRep arrives at cache 11
Invalidate cache line. Send InvRep to directory. 8
InvReq arrives at cache. 7
Interconnection Network
ExReq message received at directory controller. 4
When no more sharers, send ExRep to cache. 10
InvRep received. Clear down sharer bit. 9
Directory Controller
DRAM Bank
6 Send one InvReq message to each sharer.
Access state and directory for line. Line's state is R, with some set of sharers. 5

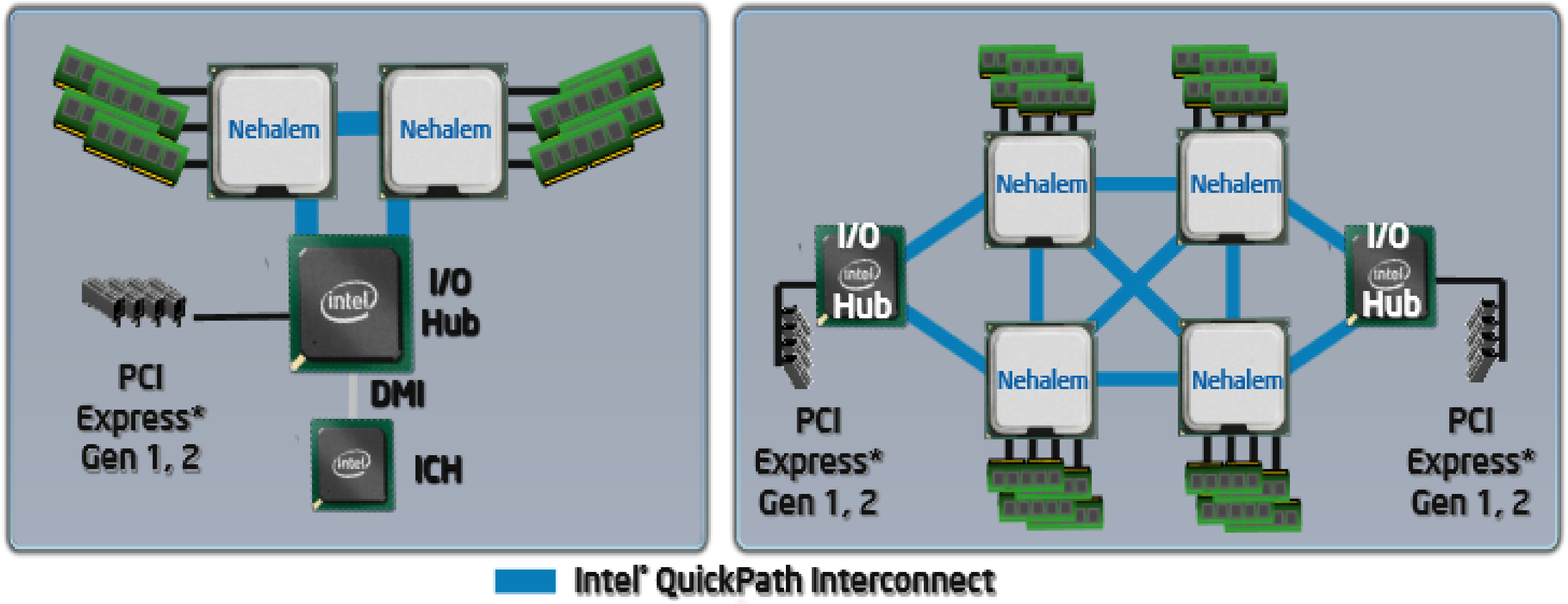
Coherence protocol implementd in Shared Cache



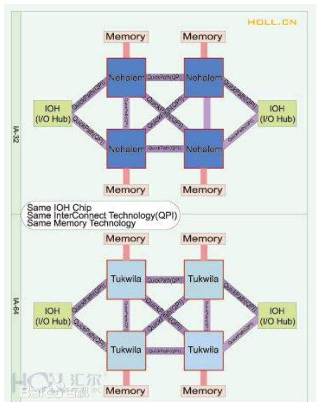
•Duplicating tags;
•Place directory in outermost cache (L3 cache)

The figure is from the paper: Why On-Chip Cache Coherence is Here to Stay

Nehalem Based System Architecture Used in Intel Core i7



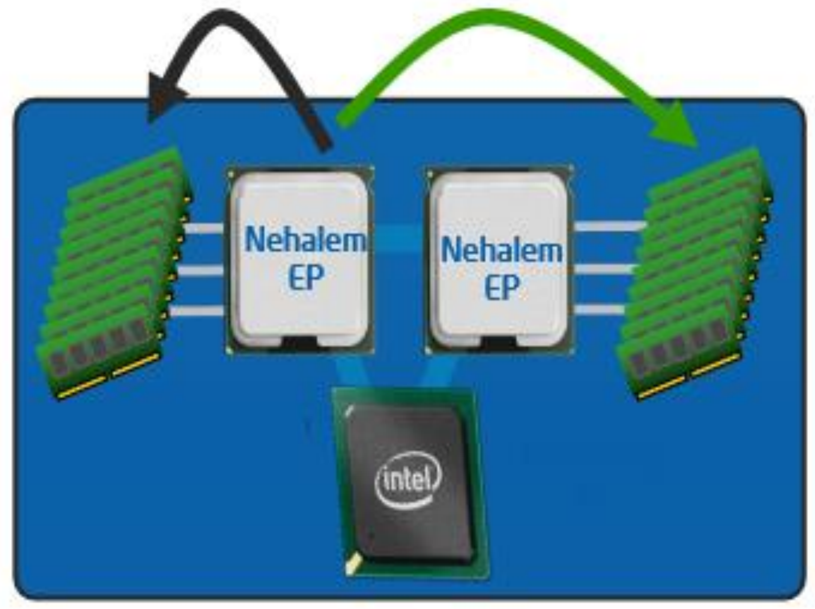
Nehalem Microarchitecture
Integrated Intel® QuickPath Memory Controller
Intel® QuickPath Interconnect
Buffered or Un-buffered Memory
PCI Express* Generation 2
Optional Integrated Graphics



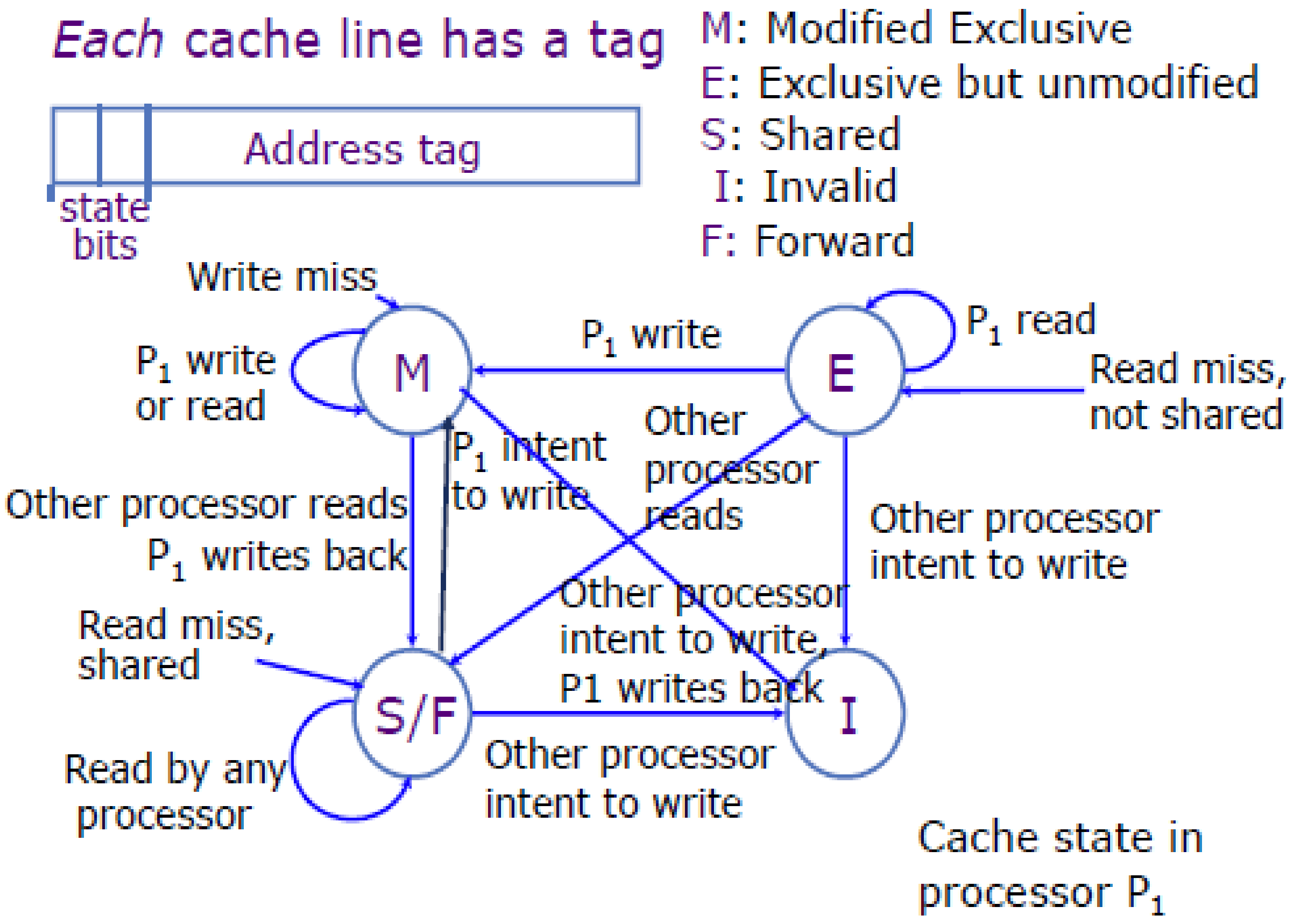
（QPI 图）

Non-Uniform Memroy Access(NUMA)

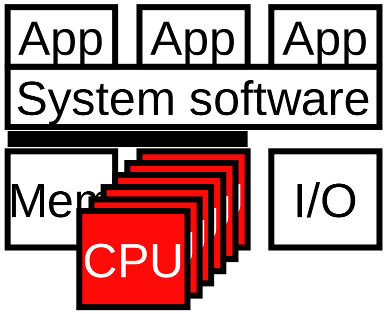
- Memory located in multiple places
- Latency to memory dependent on location
- Local memory
  - Highest BW
  - Lowest latency
- Remote Memory
  - Higher latency



MESIF (Used by Intel Core i7)


Each cache line has a tag
Address tag
state bits
M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid
F: Forward
Write miss
P1 write or read
M
P1 write
E
P1 read
Read miss, not shared
P1 intent to write
Other processor reads
Other processor reads P1 writes back
Other processor intent to write
Read miss, shared
Other processor intent to write, P1 writes back
S/F
I
Read by any processor
Other processor intent to write
Cache state in processor P1

存储器一致性模型



共享内存多处理器：例1

- **初始状态: 所有变量都为0** (即 x 为 0, y 为 0)

线程 1	线程 2

`store 1 → y`
`load x`
	`store 1 → x`
	`load y`

- 线程1读到的X，和线程2读到的Y，分别会是什么结果？
- 结果可能是 (x=0, y=0)吗?

存储器一致性

- **存储器一致性模型**
  - 共享存储器操作的语义
  - 例如，一个读操作，可能会返回什么结果
- **高速缓存一致性**
  - 关注的是同一个高速缓存块在多个处理器上的一致问题

三种存储器一致性模型

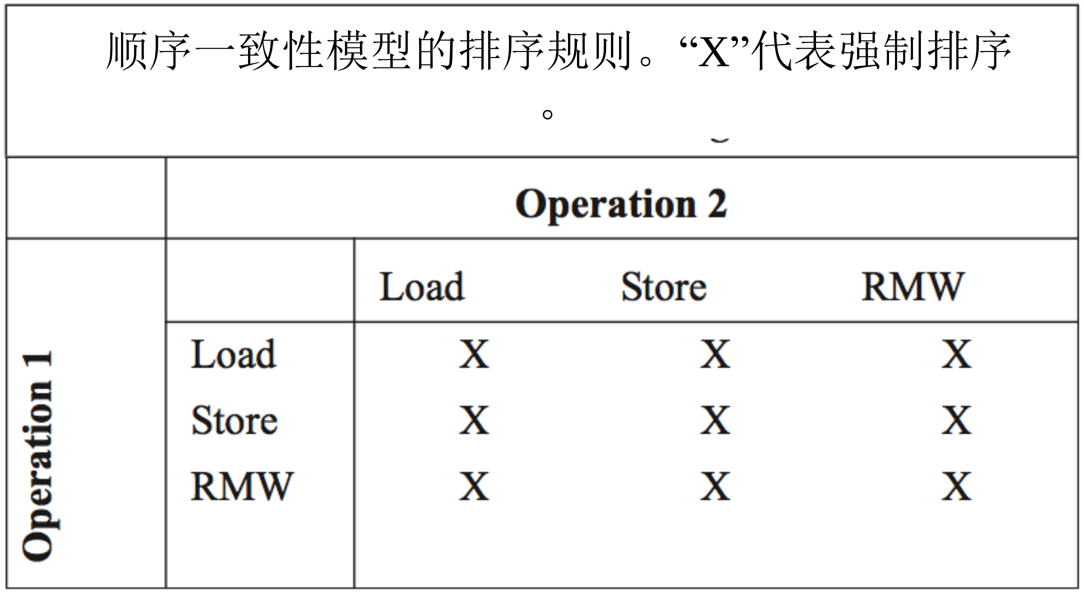
- **顺序一致性模型 (Sequential Consistency, SC)** (MIPS, PA-RISC)

  这个模型是程序员希望机器提供的，它能保证：
  - 处理器看到的自己的读内存和写内存的顺序，和程序中的一致
  - 处理器看到的别的处理器的读内存和写内存的顺序，和程序中的一致
  - 所有处理器看到的读内存和写内存的顺序是一致的

- **全存储排序模型 (Total Store Order,TSO)** (**x86**, SPARC)
  - 使用先入先出（FIFO）的store buffer
  - 写动作可以被推迟，但写入高速缓存前先经过了store buffer, 写的顺序是按序的

- **释放一致性模型 (Release Consistency,RC)** (**ARM**, Itanium, **PowerPC**)
  - 使用无序的、可合并的store buffer, store操作可以乱序
  - load操作也可以是乱序的

顺序一致性 (SC)

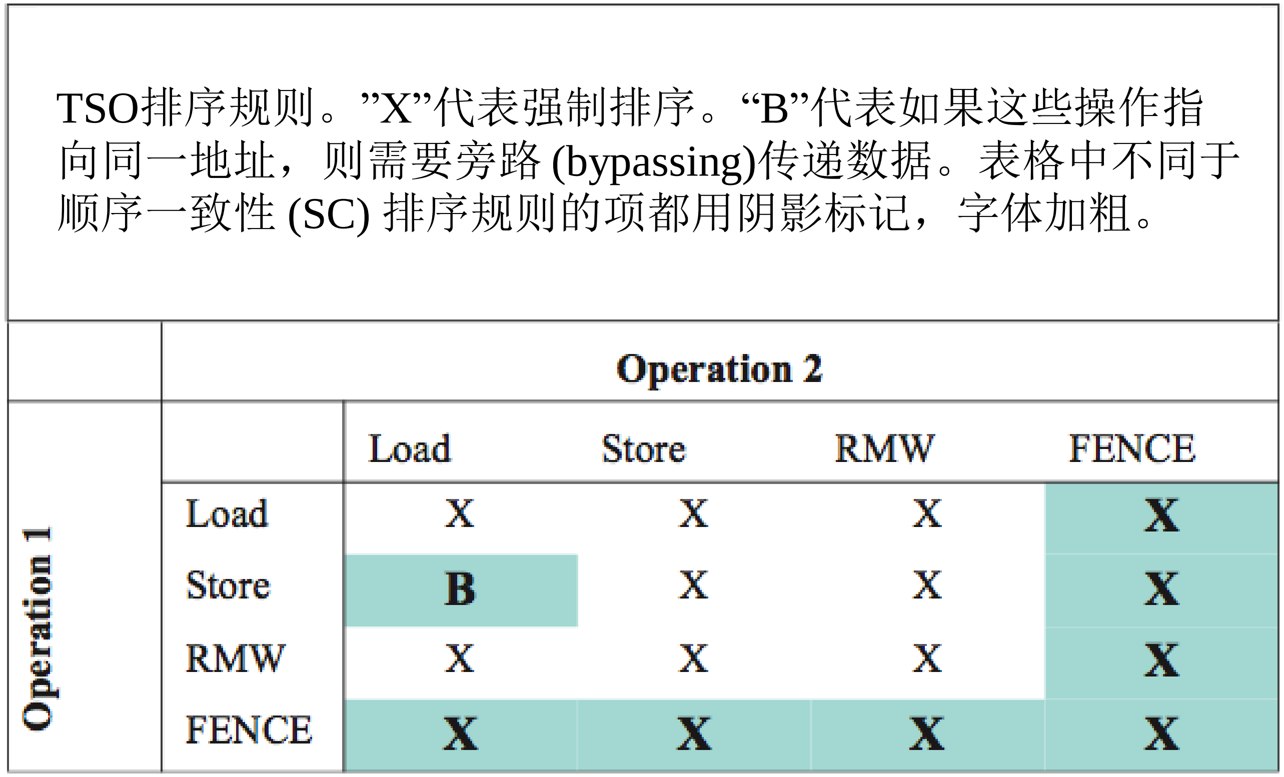


顺序一致性模型的排序规则。“X”代表强制排序。

		Operation 2		
		Load	Store	RMW
Operation 1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

From “A Primer on Memory Consistency and Cache Coherence” by Sorin, Hill and Wood

全存储排序 (TSO) (x86)



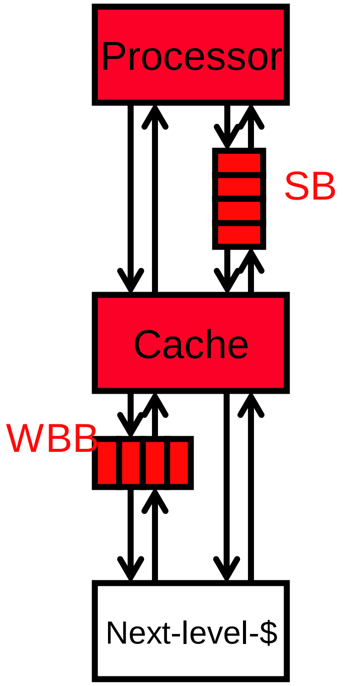
TSO排序规则。”X”代表强制排序。“B”代表如果这些操作指向同一地址，则需要旁路 (bypassing)传递数据。表格中不同于顺序一致性 (SC) 排序规则的项都用阴影标记，字体加粗。

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	X	X	X	**X**
	Store	**B**	X	X	**X**
	RMW	X	X	X	**X**
	FENCE	**X**	**X**	**X**	**X**

Memory fence，所有thread会阻塞到所有修改Memory的操作对其他thread可见

写失效与 Store Buffer

- Store buffer(SB)
  - 只需要把要写的地址和数据放入store buffer，处理器就可以继续后面的操作
  - store buffer在后台写入cache
  - load操作在读cache的同时也会读store buffer
  - (几乎可以)消除写失效引起的停顿
  - 会给共享内存的多处理器带来问题
- Store buffer vs. writeback buffer(WBB)
  - store buffer放置在数据高速缓存的前面，用于隐藏store miss（写失效)
  - write back buffer 放置在cache和下一级存储器之间，用于隐藏写回下一级存储器的开销



Store Buffer 的作用

Store Buffer 的作用:

隐藏写失效的延迟

- ❑ 对多处理器系统的影响
  - 打乱store 和load 操作的顺序 (对于不同地址的访问)
- ❑ 例:
  - 线程1和线程2中各自在store的时候都发生了写失效，把要写的数据放置在store buffer中
  - 线程1读x, 命中，线程2读y也在cache命中，但读到的旧的值

线程 1	线程 2
store 1 → y	store 1 → x
load x	load y

共享内存多处理器 例1

- **初始状态: 所有变量都为0** (即 x 为 0, y 为 0)

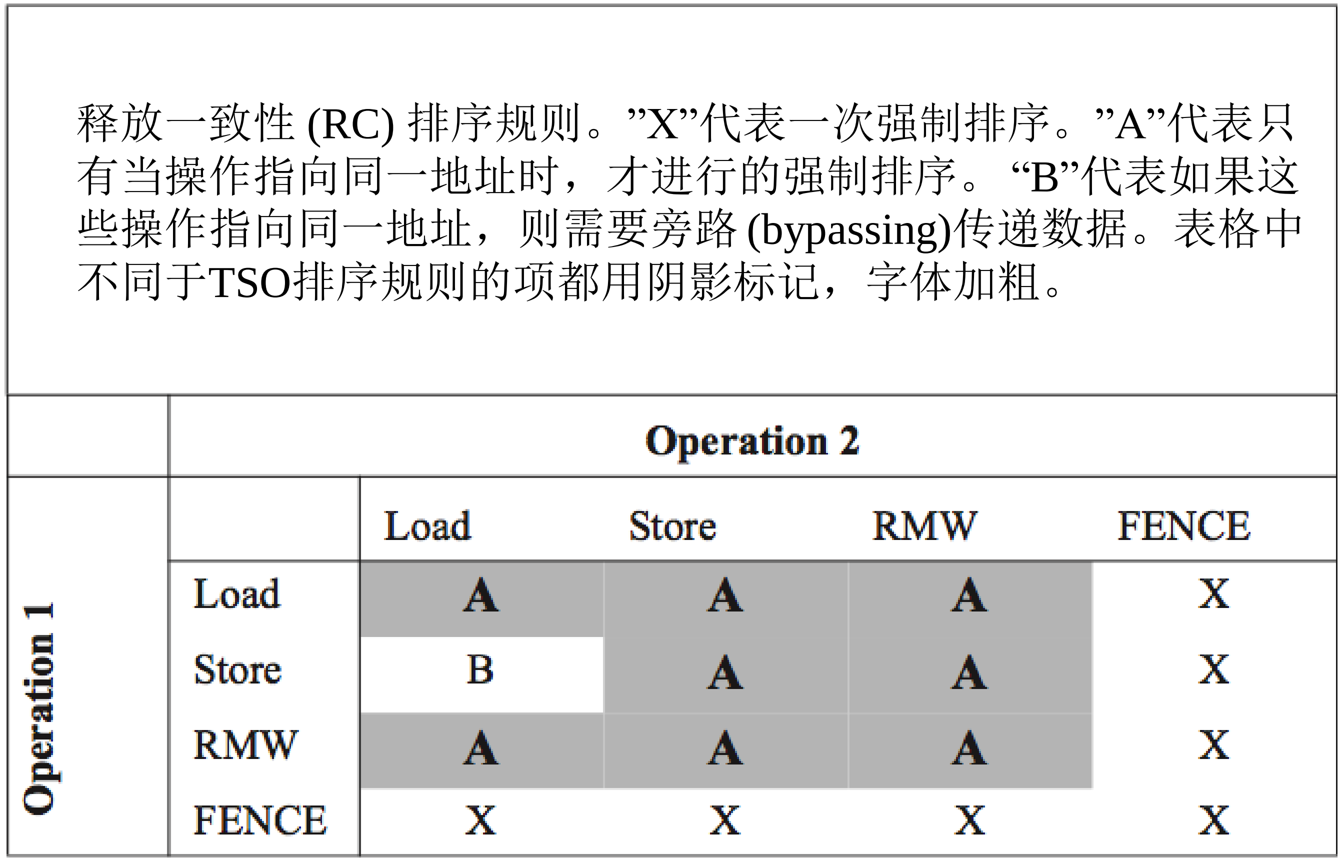
线程 1	线程 2

`store 1 → y`
`load x`
	`store 1 → x`
	`load y`

- 线程1读到的X，和线程2读到的Y，分别会是什么结果？

- 结果可能是(x=0,y=0)吗? 是！(对于x86, SPARC, ARM, PowerPC)

释放一致性



释放一致性 (RC) 排序规则。”X”代表一次强制排序。”A”代表只有当操作指向同一地址时，才进行的强制排序。“B”代表如果这些操作指向同一地址，则需要旁路 (bypassing)传递数据。表格中不同于TSO排序规则的项都用阴影标记，字体加粗。

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	**A**	**A**	**A**	X
	Store	B	**A**	**A**	X
	RMW	**A**	**A**	**A**	X
	FENCE	X	X	X	X

Why乱序？支持编译优化

- 编译优化非常重要
  - 考虑循环中有不变量的情况:

原代码

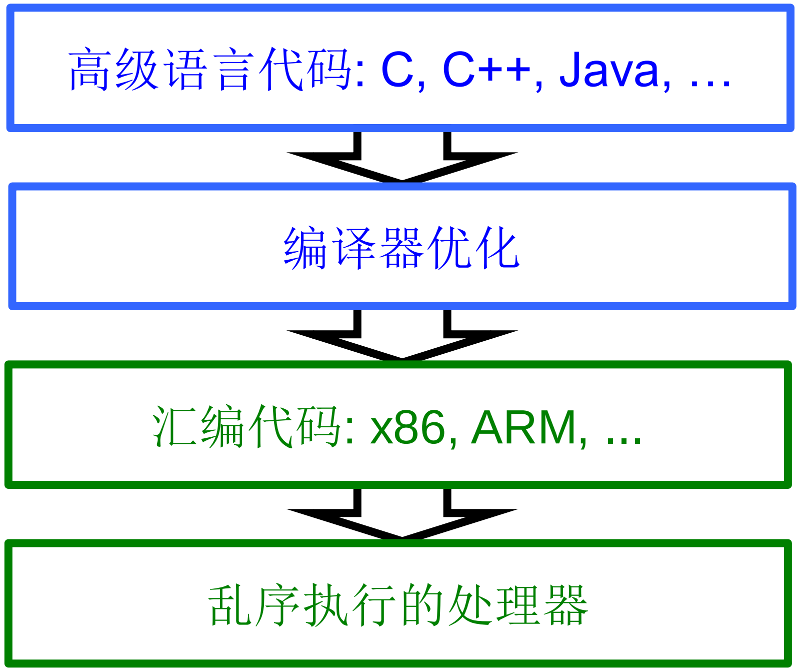
```
for (i=0; i<10; i++)
  array[i] = array2[i] + x^2;
```

优化后代码

```
tmp1 = x^2;
for (i=0; i<10; i++)
  array[i] = array2[i] + tmp1;
```

- 优化后的代码速度更快
- 优化后，读x操作在原来程序中的顺序被改变

一致性模型：层次化结构



- 如何防止代码出错？
  - **编程语言提供的一致性**
  - 依赖于编程语言中提供的工具，来保证程序在不同硬件结构上的正确性

硬件层面调整访存顺序

- 有些时候必须要规定内存操作之间的顺序 (通常不用)
- 如何规定？ 插入 **fences (memory barriers)**
  - 特殊指令，ISA的一部分
- 例如：
  - 设置一个临界区，通过同步，保证所有在临界区之前的load，store必须全部完成，才能开始执行临界区

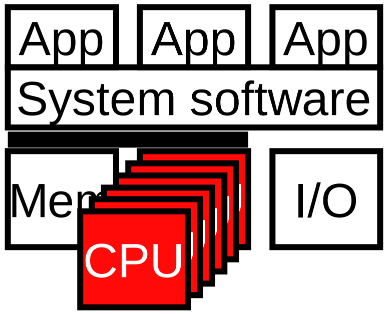
```
lock acquire
fence
"critical section"
fence
lock release
```

- fences 如何工作？
  - 在store buffer清空之前，暂停所有的访存操作
- 在各种高级语言中，都提供了同步函数库，供程序员使用

软件层面调整访存顺序

- 如何告诉编译器不要进行乱序优化呢？
  - 不同的语言提供不同的机制
- Java:
  - 用 **volatile** 关键字来修饰变量
    - 赋予所有标记 **volatile** 的位置SC语义
  - Java 编译器插入硬件级排序指令
- C/C++:
  - C++11 提供一个新的 **atomic** 关键字, 与Java的 **volatile** 类似

小结



- 高速缓存一致性模型
- 存储器一致性模型

谢　谢！