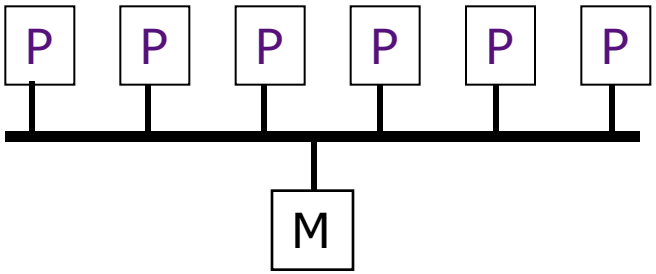
计算机体系结构

Cache Consistency

顺序同一性的存储器模型



“ A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

Leslie Lamport

Sequential Consistency =

多个进程之间的存储器操作可以任意交叉

每个进程的存储器操作按照程序序

顺序同一性的充分条件

- **多个进程可以交织执行，但顺序同一性模型没有定义具体的交织方式，满足每个进程程序序的总体执行序可能会很多。因此有下列定义：**
  - 顺序同一性的执行：如果程序的一次执行产生的结果与前面定义的任意一种可能的总体序产生的结果一致，那么程序的这次执行就称为是顺序同一的。
  - 顺序同一性的系统：如果在一个系统上的任何可能的执行都是顺序同一的，那么这个系统就是顺序同一的

顺序同一性的充分条件

- 每个进程按照程序执行序发出存储操作
- 发出写操作后，进程要等待写的完成，才能发出它的下一个操作
- 发出读操作后，进程不仅要等待读的完成，还要等待产生所读数据的那个写操作完成，才能发出它的下个操作。即：如果该写操作对这个处理器来说完成了，那么这个处理器应该等待该写操作对所有处理器都完成了。
- 第三个条件保证了写操作的原子性。即读操作必须等待逻辑上先前的写操作变得全局可见

TABLE 3.1: Should r2 Always be Set to NEW?		
Core C1	**Core C2**	**Comments**
S1: Store data = NEW; S2: Store flag = SET;	L1: Load r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: Load r2 = data;	/* Initially, data = 0 & flag ≠ SET */ /* L1 & B1 may repeat many times */

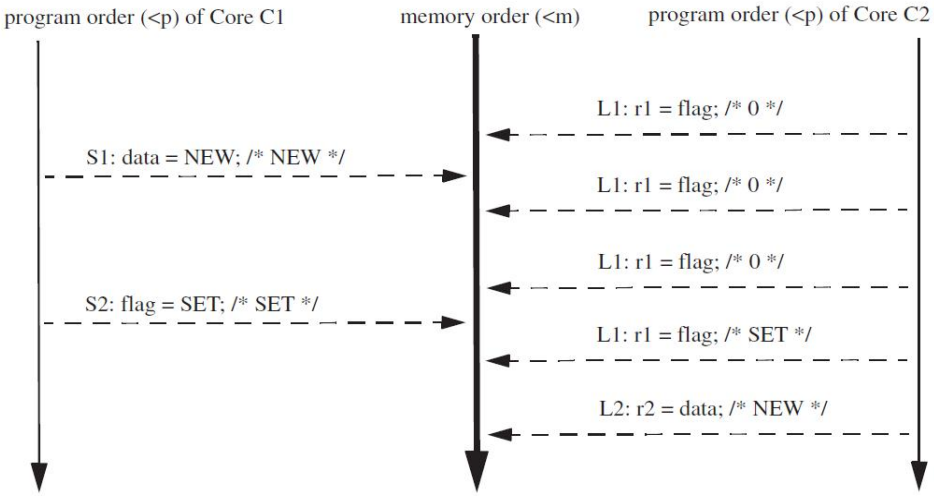


FIGURE 3.1: A Sequentially Consistent Execution of Table 3.1's Program.

(1) 所有core执行的Load/Store满足程序序

/* Load -> Load */

If L(a) <p L(b) => L(a) <m L(b)

/*Load -> Store */

If L(a) <p S(b) => L(a) <m L(b)

/* Store ->Store */

If S(a) <p S(b) => S(a) <m S(b)

/* Store -> Load */

If S(a) <p L(b) => S(a) <m L(b)

(2) 对同一存储单元的Load操作的值来源于最近一次写操作(global memory order)

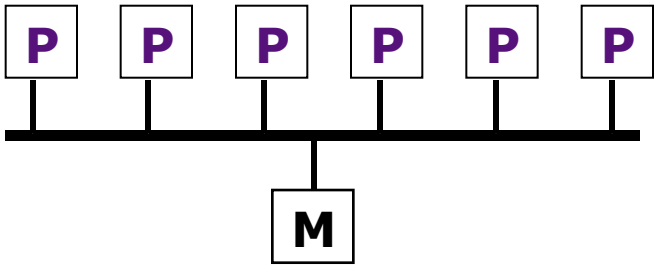
Value of L(a) = Value of $Max_{<m}${S(a) <m L(a)},

$Max_{<m}$表示最近的memory order

Sequential Consistency

- **SC 约束了所有的存储器操作的序:**
  - Write → Read
  - Write → Write
  - Read → Read
  - Read → Write
- **是有关并行程序执行的简单模型**
- **但是, 直觉上在单处理器上的合理的存储器操作的重排序会违反SC模型**
- **现代微处理器设计中一直都在应用重排序操作来获得性能提升(write buffers, overlapped writes, non-blocking reads...).**
- **Question: 如何协调性能提升与SC的约束?**

Issues in Implementing Sequential Consistency



现代计算机系统实现SC 的两个问题

- *Out-of-order execution capability*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Caches. Write buffer*

  Cache使得某一处理器的store操作不能被另一处理器即时看到

No common commercial architecture has a sequentially consistent memory model ！ ！ ！

Relaxed Consistency Models

- **Rules:**
  - X ⟶ Y :Operation X must complete before operation Y is done

Sequential consistency requires (SC) :

R ⟶ W, R ⟶ R, W ⟶ R, W ⟶ W

Relax W ⟶ R (TSO)

"Total store ordering" (X86)

Relax W ⟶ W (PSO)

"Partial store order"

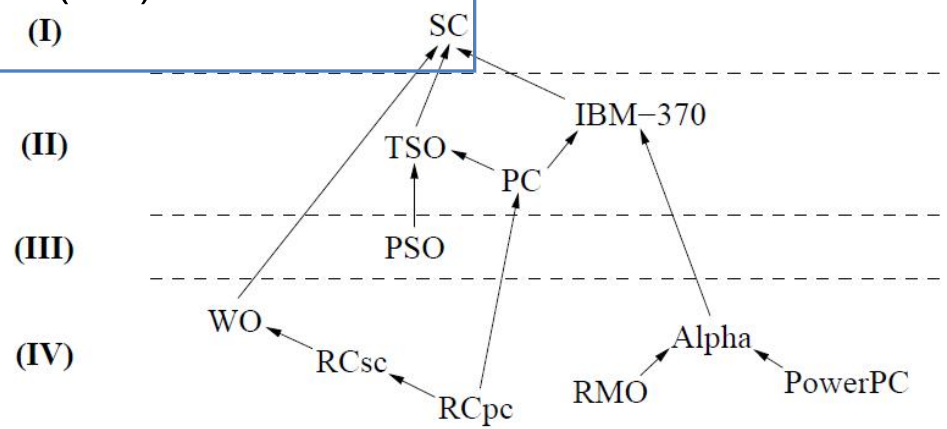


Figure 2.24: Relationship among models according to the "stricter" relation.

Relax R ⟶ W and R ⟶ R

"Weak ordering" and "release consistency"

Relax R ⟶ R , R ⟶ W , W-R, W ⟶ W (RMO)

"Release Memory Ordering"

Maintains the program order to access the same location:

W ⟶R, W ⟶ W

Simple categorization of relaxed models

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Figure 8: Simple categorization of relaxed models. A √ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The "Read Own Write Early" relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The "Read Others' Write Early" relaxation is possible and detectable with complex implementations of RCsc.

TABLE 4.1: Can Both r1 and r2 be Set to 0?		
Core C1	**Core C2**	**Comments**
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/

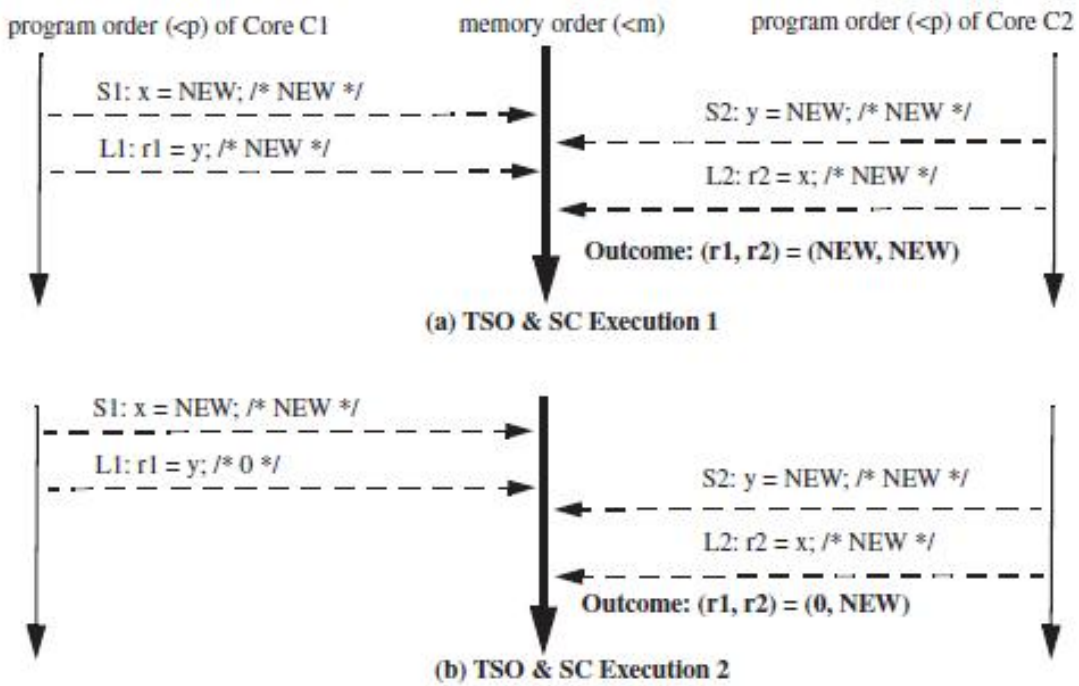
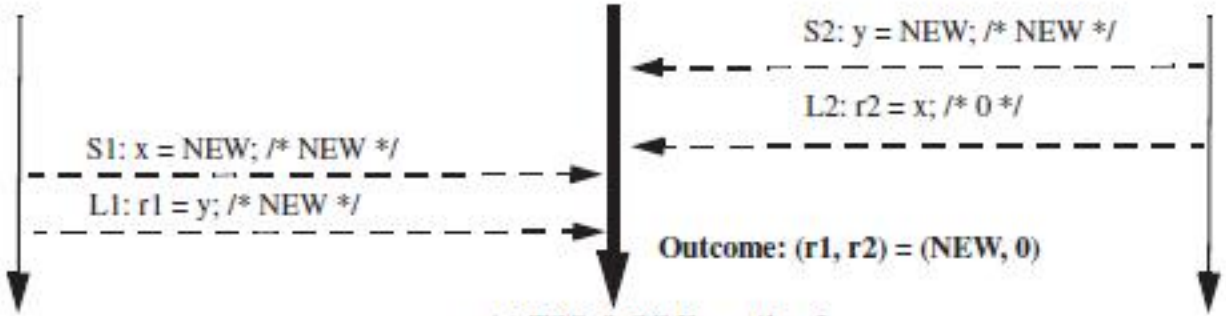
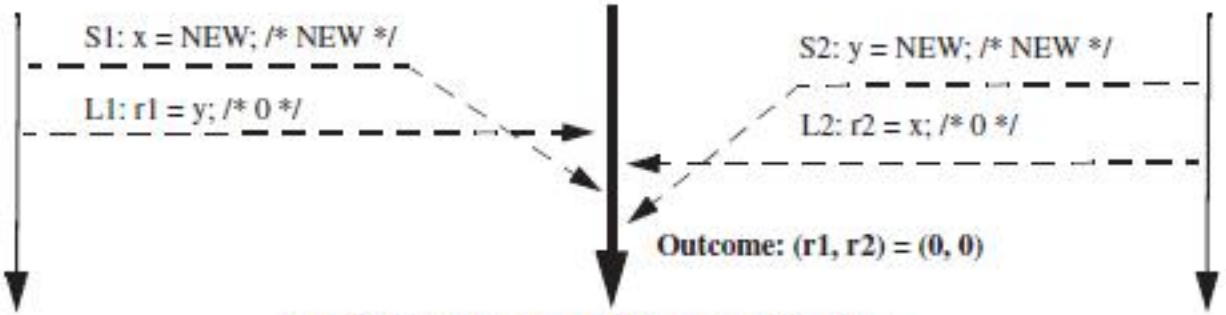


TABLE 4.1: Can Both r1 and r2 be Set to 0?

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/



(c) TSO & SC Execution 3



(d) TSO Execution, but NOT an SC Execution

RE 4.2: Four Alternative TSO Executions of Table 4.1's Program.

TABLE 4.3: Can r1 or r3 be Set to 0?		
Core C1	**Core C2**	**Comments**
S1: x = NEW; L1: r1 = x; L2: r2 = y;	S2: y = NEW; L3: r3 = y; L4: r4 = x;	/* Initially, x = 0 & y = 0*/ /* Assume r2 = 0 & r4 = 0 */

- **当(r2, r4) = (0, 0) 时， (r1, r3) 一定为 (0, 0) ？**

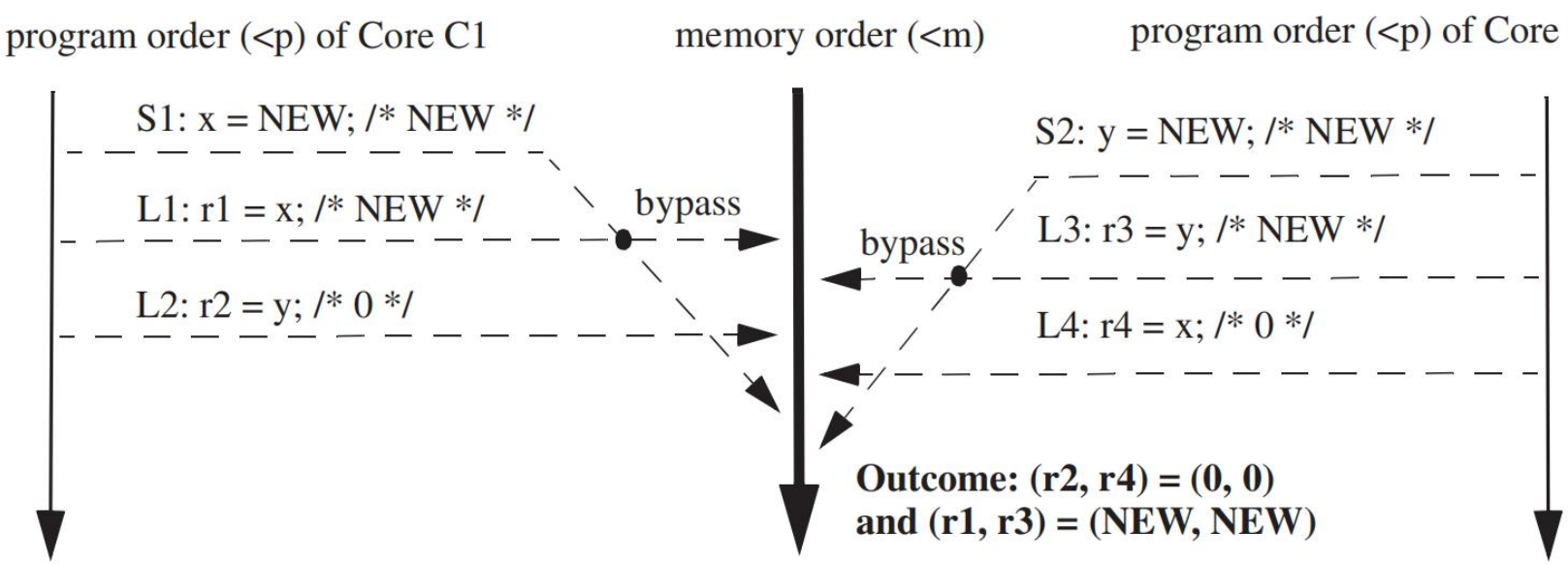


FIGURE 4.3: A TSO Execution of Table 4-3's Program (with "bypassing").

Memory Fences

Instructions to sequentialize memory accesses

实现弱同一性或放松的存储器模型的处理器（允许针对不同地址的loads 和stores操作乱序）需要提供存储器栅栏指令来强制对某些存储器操作串行化

Examples of processors with relaxed memory models:

Sparc V8 (TSO,PSO): Membar

Sparc V9 (RMO):

Membar #LoadLoad, Membar #LoadStore

Membar #StoreLoad, Membar #StoreStore

PowerPC (WO): Sync, EIEIO

ARM: DMB (Data Memory Barrier)

X86/64: mfence (Global Memory Barrier)

存储器栅栏是一种代价比较大的操作，仅仅在需要时，对存储器操作串行化

Table 1: Summary of Memory Ordering

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

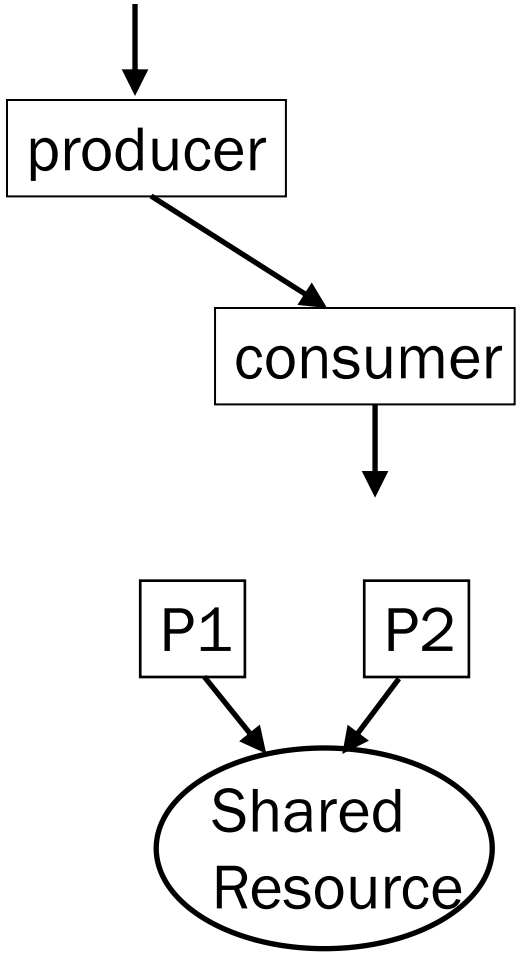
Synchronization

系统中只要存在并发进程，即使是单核系统都需要同步操作

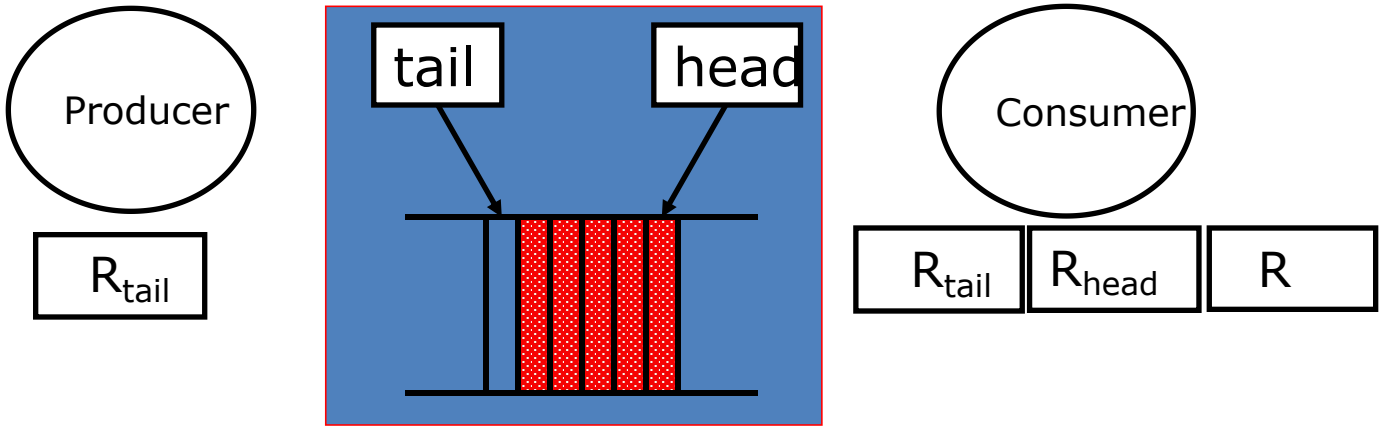
总体上存在两类同步操作问题:

生产者-消费者问题：一个 消费者进程必须等待生产者进程产生数据

互斥问题（Mutual Exclusion）：保证在一个给定的时间内只有一个进程使用共享资源（临界区）



A Producer-Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)

Store (R_{tail}), x

R_{tail}=R_{tail}+1

Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail)

if R_{head}==R_{tail} goto spin

Load R, (R_{head})

R_{head}=R_{head}+1

Store (head), R_{head}

process(R)

假设指令都是顺序执行的

Problems?

A Producer-Consumer Example *continued*

Producer posting Item x:

```
    Load R_tail, (tail)
1   Store (R_tail), x
    R_tail=R_tail+1
2   Store (tail), R_tail
```

Consumer:

```
    Load R_head, (head)
spin: Load R_tail, (tail)      3
    if R_head==R_tail goto spin
    Load R, (R_head)           4
    R_head=R_head+1
    Store (head), R_head
    process(R)
```

Can the tail pointer get updated before the item x is stored?

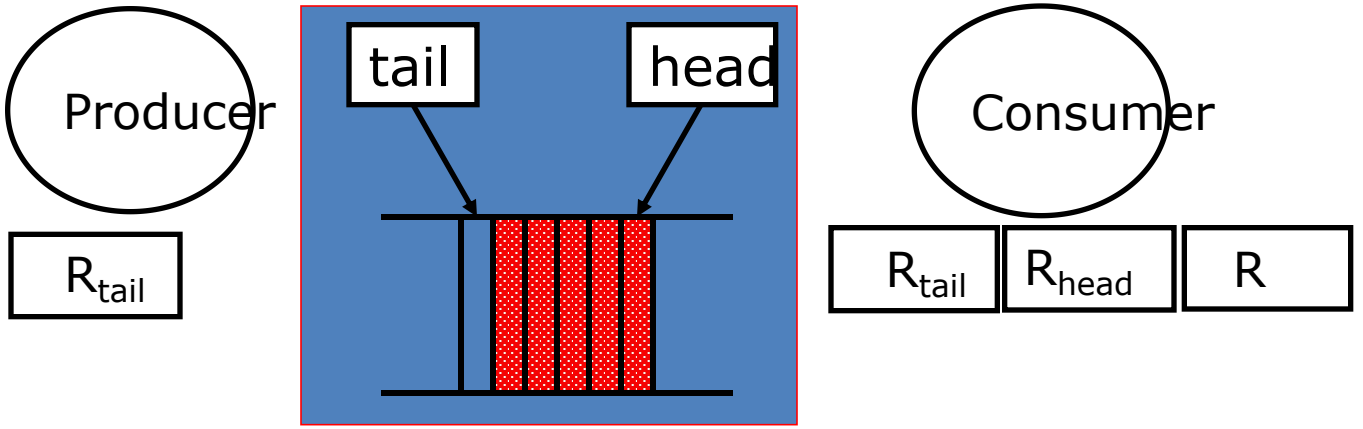
Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

2, 3, 4, 1

4, 1, 2, 3

Using Memory Fences



Producer posting Item x:

Load R_{tail}, (tail)

Store (R_{tail}), x

$Membar_{SS}$

$R_{tail}=R_{tail}+1$

Store (tail), R_{tail}

ensures that tail ptr is not updated before x has been stored

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail)

if $R_{head}==R_{tail}$ goto spin

$Membar_{LL}$

Load R, (R_{head})

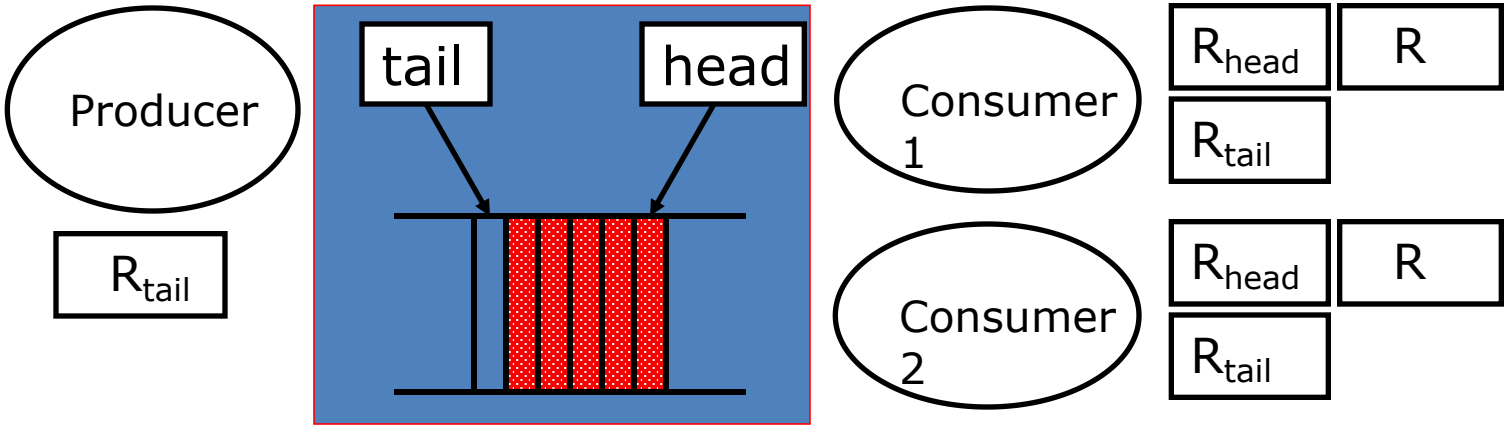
$R_{head}=R_{head}+1$

Store (head), R_{head}

process(R)

ensures that R is not loaded before x has been stored

Multiple Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
R_{tail}=R_{tail}+1
Store (tail), R_{tail}

Critical section:
Needs to be executed atomically by one consumer

Consumer:

Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if R_{head}==R_{tail} goto spin
Load R, (R_{head})
R_{head}=R_{head}+1
Store (head), R_{head}
process(R)

What is wrong with this code?

Mutual Exclusion Using Load/Store

基于两个共享变量c1和c2的同步协议。初始状态c1和c2均为0

Process 1

```
      ...
      c1=1;
L:    if c2=1 then go to L
        < critical section>
      c1=0;
```

Process 2

```
      ...
      c2=1;
L:    if c1=1 then go to L
        < critical section>
      c2=0;
```

What is wrong? *Deadlock!*

Mutual Exclusion: second attempt

Process 1

```
    ...
L:  c1=1;
    if c2=1 then
    { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
    ...
L:  c2=1;
    if c1=1 then
    { c2=0; go to L}
      < critical section>
    c2=0
```

- 为避免死锁，我们让一进程等待时放弃reservation（预订）
  - Process 1 sets c1 to 0.
- 死锁显然是没有了，但有可能会发生 活锁*(livelock)* 现象
  - C1 = 1, C2=1, Read C2, Read C1, C1 =0, C2 = 0, C1=1, C2=1, C1=0, C2=0
- 可能还会出现某个进程始终无法进入临界区 ⇒ *starvation*
  - *例如： C1=1，C2= 1，C1=0，Read C1 进入临界区， P1和P2竞争 P2始终胜出*

A Protocol for Mutual Exclusion

T. Dekker, 1966

基于3个共享变量c1, c2 和turn的互斥协议，初始状态三个变量均为0.

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
            then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
        then go to L
      < critical section>
    c2=0;
```

- turn = *i 保证仅仅进程 i 等待*
- 变量 c1 和 c2 保证n个进程互斥地访问临界区，该算法由*Dijkstra给出，相当精巧*

Locks or Semaphores

E. W. Dijkstra, 1965

信号量（A *semaphore*） 是一非负整数, 具有如下操作:

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P(s)和V(s) 必须是原子操作, i.e., 不能被中断，不能由多个处理器交叉访问s

```
Process i
    P(s)
        <critical section>
    V(s)
```

s的初始值设置为可访问临界区的最大进程数

Atomic Operations

- **顺序一致性并不保证操作的原子性**
- **原子性：存储器操作使用原子性操作，操作序列一次全部完成。例如exchange（交换操作）。**

exchange(r,M): 互换寄存器r与存储单元M的内容

```
r0 = 1;
do exchange(r0,S) while (r0 != 0);
                                    //S is memory location
 //enter critical section
 .....
 //exit critical section
 S = 0;
```

Implementation of Semaphores

在顺序同一性模型中，信号量 (mutual exclusion) 可以用常规的 Load 和 Store 指令实现，但是互斥的协议很难设计。一种简单的解决方案是提供:

atomic read-modify-write instructions

Examples: *m is a memory location, R is a register*

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
    M[m] ← 1;
```

```
Fetch&Add (m), RV, R:
    R ← M[m];
    M[m] ← R + RV;
```

```
Swap (m), R:
    Rt ← M[m];
    M[m] ← R;
    R ← Rt;
```

Multiple Consumers Example

using the Test&Set Instruction

```
P:   Test&Set (mutex),Rtemp
     if (Rtemp!=0) goto P
     Load Rhead, (head)
spin:  Load Rtail, (tail)
     if Rhead==Rtail goto spin
     Load R, (Rhead)
     Rhead=Rhead+1
     Store (head), Rhead
V:   Store (mutex),0
     process(R)
```

Critical Section

其他原子的read-modify-write 指令 (Swap, Fetch&Add, etc.) 也能实现 P(s)和 V(s)操作

Nonblocking Synchronization

```
Compare&Swap(m), Rt, Rs:
  if (Rt==M[m])
     then  M[m]=Rs;
           Rs=Rt ;
           status ← success;
     else  status ← fail;
```

status is an *implicit argument*

```
try:    Load Rhead, (head)
spin:   Load Rtail, (tail)
     if Rhead==Rtail goto spin
     Load R, (Rhead)
     Rnewhead = Rhead+1
     Compare&Swap(head), Rhead, Rnewhead
     if (status==fail) goto try
     process(R)
```

Performance of Locks

Blocking atomic read-modify-write instructions

e.g., Test&Set, Fetch&Add, Swap

vs

Non-blocking atomic read-modify-write instructions

e.g., Compare&Swap,

Load-reserve/Store-conditional

vs

Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:

degree of contention,

caches,

out-of-order execution of Loads and Stores

later ...

本课程主要内容

关键字：

并行（并发）、评估、优化

第1章

- **重点关注： 与性能度量、功耗（能耗）有关的问题**

- **性能度量**
  - 响应时间 (response time)
  - 吞吐率 (Throughput)
- **CPU 执行时间 = IC × CPI × T**
  - CPI （ Cycles per Instruction)
- **MIPS = Millions of Instructions Per Second**
- **Latency versus Bandwidth**
  - Latency指单个任务的执行时间，Bandwidth 指单位时间完成的任务量（rate）
  - Latency 的提升滞后于带宽的提升 (在过去的30年)
- **Amdahl’s Law 用来度量加速比（speedup)**
  - 性能提升受限于任务中可加速部分所占的比例
  - 应用于多处理器系统的基本假设：在给定的问题规模下，研究随着处理器数目的增加性能的变化
- **Benchmarks：指一组用于测试的程序**
  - 比较计算机系统的性能
  - SPEC benchmark：针对一组应用综合性能值采用SPEC ratios 的几何平均

Power & Energy

- **Dynamic Energy ∝**

  Capacitive Load × Voltage²

  – 从0-1-0 或 1-0-1逻辑跃迁的脉冲能量
  – Capacitive Load =输出晶体管和导线的电容负载
  – 20年来晶体管供电电压已经从5V降到1V
- **Dynamic Power ∝**

  Capacitive Load × Voltage² × Frequency Switched

第2章 指令集架构

- **重点关注：ISA设计的基本方法、RISC**

- **ISA需考虑的问题：**
  - Class of ISA; Memory addressing; Types and sizes of operands ; Operations ; Control flow instructions ; Encoding an ISA
  - ……
- **ISA的类型**
  - 通用寄存器型占主导地位
- **寻址方式**
  - 重要的寻址方式: 偏移寻址方式, 立即数寻址方式, 寄存器间址方式
    - SPEC测试表明，使用频度达到 75%--99%
  - 偏移字段的大小应该在 12 - 16 bits, 可满足75%-99%的需求
  - 立即数字段的大小应该在 8 -16 bits, 可满足50%-80%的需求
- **操作数的类型和大小**
  - 对单字、双字的数据访问具有较高的频率
  - 支持64位双字操作，更具有一般性

- **ISA的功能设计：任务为确定硬件支持哪些操作。方法是统计的方法。存在CISC和RISC两种类型**
  - CISC（Complex Instruction Set Computer）
    - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
    - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
  - RISC（Reduced Instruction Set Computer）
    - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
    - 主要手段：充分发挥流水线的效率，降低（优化）CPI
- **控制转移类指令**
- **指令编码（指令格式）**
- **MIPS ISA**
- **RISC-V ISA**

- 流水线性能评估和优化技术

- 实际吞吐率：假设k段，完成n个任务，单位时间所实际完成的任务数。
- 加速比: k段流水线的速度与等功能的非流水线的速度之比。
- 效率：流水线的设备利用率。

$$TP_{\max} = \frac{1}{\max\{\Delta t_i\}}$$

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$S = \frac{n \cdot \sum_{i=1}^{k} \Delta t_i}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$E = \frac{n \cdot \sum_{i=1}^{k} \Delta t_j}{k \cdot [\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j]}$$

$$E = TP \cdot \frac{\sum_{i=1}^{k} \Delta t_i}{k}$$

- **指令流水线通过指令重叠减小 CPI**
- **充分利用数据通路**
  - 当前指令执行时，启动下一条指令
  - 其性能受限于花费时间最长的段
  - 检测和消除相关
- **如何有利于流水线技术的应用**
  - 所有的指令都等长
  - 只有很少的指令格式
  - 只用Load/Store来进行存储器访问

流水线的加速比计算

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

- **影响流水线性能**
  - 结构相关、数据相关
  - 控制相关、异常
- **异常处理**
  - 种类与分类
  - 精确与非精确中断
- **支持浮点数操作的MIPS流水线**
  - Latency & Repeat Interval
  - 问题：结构相关（增多）；数据相关、控制相关引起的stall增多；有新的冲突源产生；定向路径增多；异常处理复杂
  - MIPS R4000 8级流水线
    - 存储器操作分阶段 – load延迟为2个cycles
    - Branch操作在EX段确定分支方向- 3个cycles的延迟
    - 多个定向源： EX/DF，DF/DS，DS/TC，TC/WB
  - MIPS R4000的浮点数操作

- **存储层次、Cache性能评估和优化、主存组织、虚拟存储（TLB）**
- **Cache基本原理 4Q**
- **Cache性能分析**

CPU time = (CPU execution clock cycles +
Memory stall clock cycles) x clock cycle time

Memory stall clock cycles =
(Reads x Read miss rate x Read miss penalty +
Writes x Write miss rate x Write miss penalty)

Memory stall clock cycles =
Memory accesses x Miss rate x Miss penalty

Different measure: AMAT
Average Memory Access time (AMAT) =
Hit Time + (Miss Rate x Miss Penalty)

Cache优化

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

第 5 章

- **提高指令级并行的硬件方法、软件方法**

- **指令级并行(ILP)：流水线的平均CPI**
  - Pipeline CPI = Ideal Pipeline CPI + Struct Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls +……
  - 提高指令级并行的方法
    - 软件方法：指令流调度，循环展开，软件流水线，trace scheduling
    - 硬件方法
- **软件方法：指令流调度-循环展开**
- **硬件方法：两种指令流动态调度方法**
  - 如何处理精确中断?
  - Out-of-order execution -> out-of-order completion!
  - 如何处理分支?
- **动态分支预测**
  - 常见的方法有哪些？评估

- **存储器访问的冲突消解**
  - Total Ordering
  - Partial Ordering
  - Load Ordering, Store Ordering
  - Store Ordering

- **Superscalar and VLIW: CPI < 1 (IPC > 1)**
  - Dynamic issue vs. Static issue
  - 同一时刻发射更多的指令 => 导致更大的冲突开销

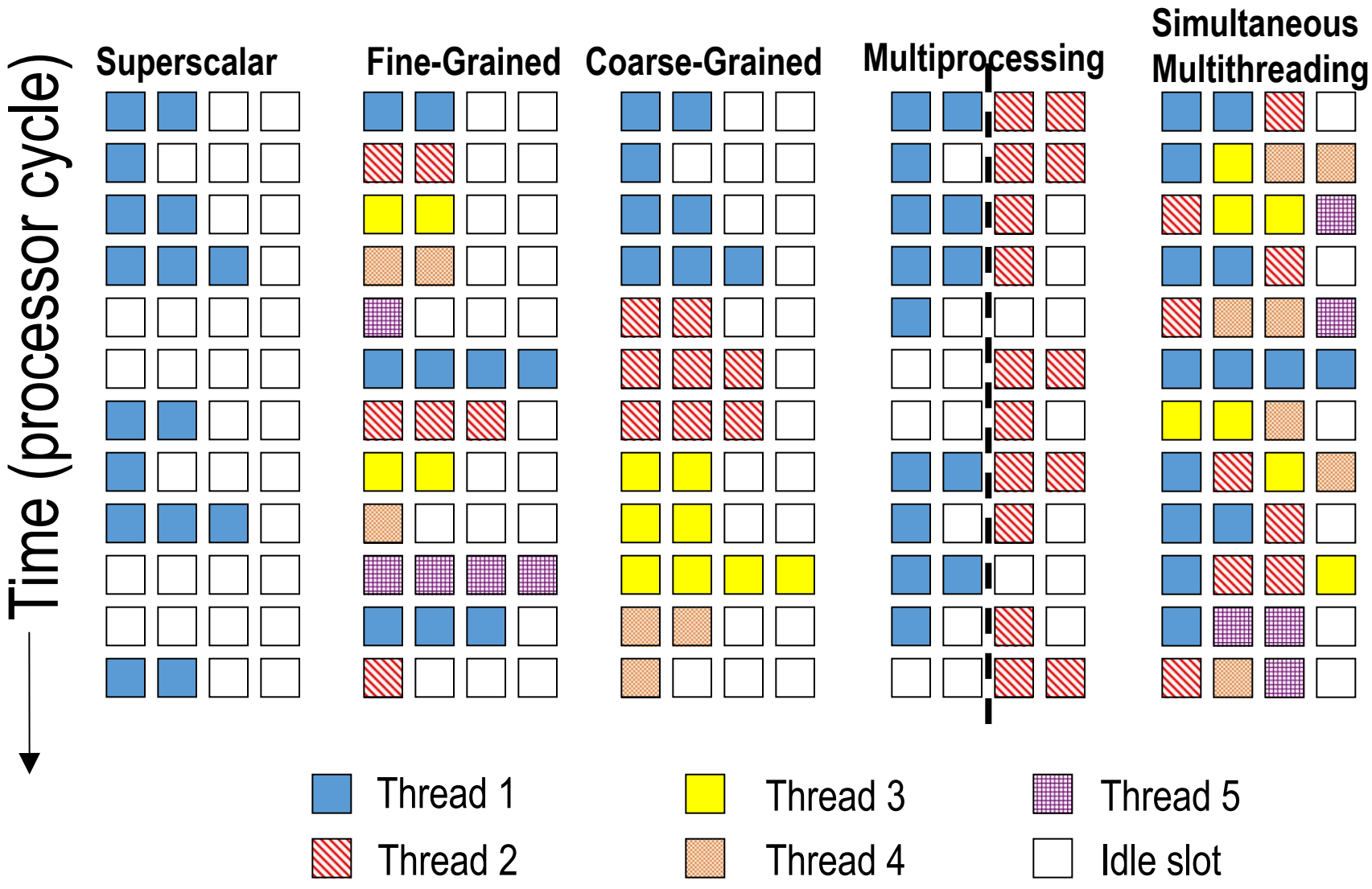
Memory Disambiguation

TABLE 6.1: Memory disambiguation schemes.

NAME	SPECULATIVE	DESCRIPTION
Total Ordering	No	All memory accesses are processed in order.
Partial Ordering	No	All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address.
Load Ordering Store Ordering	No	Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them.
Store Ordering	Yes	Stores execute in order, but loads execute completely out of order.

- **非投机方式的基本原则：当前存储器指令之前的store指令计算存储器地址后，才能执行当前的存储器操作**

Multithreaded Categories



第6章

- **向量处理机模型、ＧＰＵ**

- **向量处理机基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
  - VSIW-一条指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- **向量处理机基本结构**
  - 向量指令并行执行
  - 向量运算部件的执行方式-流水线方式
  - 向量部件结构-多“道”结构-多条运算流水线
- **向量处理机性能评估**
  - 向量指令流执行时间: Convey, Chimes, Start-up time
  - 其他指标： R_∞ , $N_{1/2}$, N_V

- **两个问题**
  - 向量处理机中的存储器访问
  - 向量处理机中的优化技术

- **向量机的存储器访问**
  - 存储器组织：独立存储体、多体交叉方式
  - Stride : 固定步长（1 or 常数），非固定步长（index）
- **分段开采技术**
- **基于向量机模型的优化**
  - 链接技术
  - 有条件执行
  - 稀疏矩阵的操作

GPU

- **GPU：多线程协处理器**
- **GPU编程模型：SPMD (Single Program Multiple Data)**
  - 使用线程 (SPMD 编程模型)，不是用SIMD指令编程
    - 每个线程执行同样的代码，但操作不同的数据元素
    - 每个线程有自己的上下文(即可以独立地启动/执行等)
  - 计算由大量的相互独立的线程(CUDA threads or microthreads) 完成，这些线程组合成线程块（thread blocks）
- **GPU执行模型：SIMT (Single Instruction Multiple Thread)**
  - 一组执行相同指令的线程由硬件动态组织成warp
  - 一个warp是由硬件形成的SIMD操作
- **GPU存储器组织**
  - Local Memory， Shared Memory, Global Memory
- **GPU分支处理（发散与汇聚）**

- **重点关注：Coherence、Consistency**

Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**