第 11 讲：Scalable Synchronization on Shared-Memory Multiprocessors

第三节：Hardware Behavior in Shared-Memory Multiprocessors – Memory Consistency

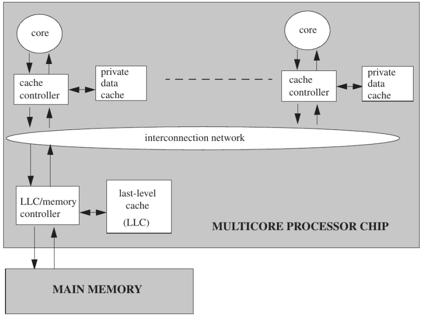
陈渝

清华大学计算机系

yuchen@tsinghua.edu.cn

2020 年 4 月 27 日

Introduction – Basic Info

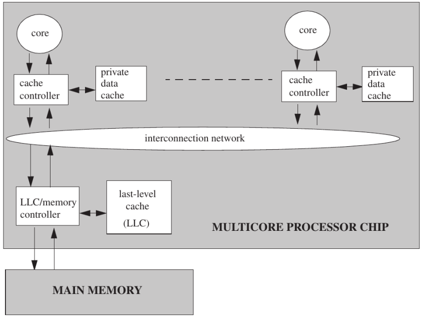


回顾：内存缓存

- 在现代的 CPU（大多数）上，所有的内存访问都需要通过层层的缓存来进行。
- 例外：比如，对映射成内存地址的 I/O 口，这些访问至少会绕开这个流程的一部分。
- CPU 的读 / 写（以及取指令）单元正常情况下甚至都不能直接访问内存——这是物理结构决定的。
- 缓存是分“段”（line）的，一个段对应一块存储空间，大小是 32、64、128 字节。

ref: Some info are from
Paul McKenney (IBM) Tom Hart (University of Toronto), Frans Kaashoek (MIT), Daniel J. Sorin ”A Primer on Memory Consistency and Cache Coherence”, Fabian Giesen ”Cache coherency primer”, Mingyu Gao(Tsinghua),Yubin Xia(SJTU)

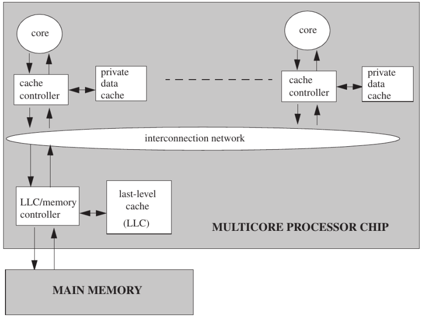
Introduction



回顾 –内存模型 (Memory Consistency Model)
Shared-Memory Multi-Core/SMP/NUMA

- All cores see a single physical address space
- Inter-core communication through loads and stores
- Data can be cached in multiple private and shared caches

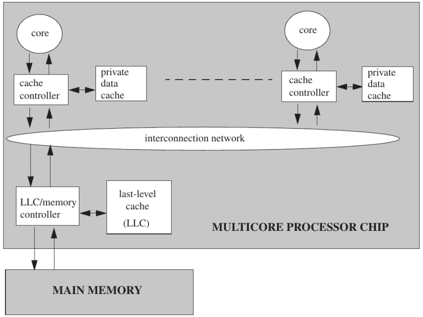
Introduction



回顾 –内存模型 (Memory Consistency Model)
Shared-Memory Multi-Core/SMP/NUMA
Coherence concerns accesses only a single memory location

- Making sure that caches & stale copies do not cause problems
- Two invariants: single-writer multiple-readers, data-value

Introduction



回顾 –内存模型 (Memory Consistency Model)

Shared-Memory Multi-Core/SMP/NUMA

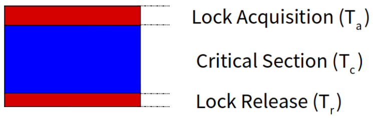
Coherence concerns accesses only a single memory location

- Making sure that caches & stale copies do not cause problems
- Two invariants: single-writer multiple-readers, data-value

Consistency concerns ordering for accesses to many locations

- Making sure that ordering does not cause problems

Synchronization

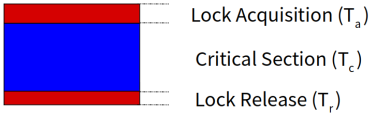


$$\text{Critical-section efficiency} = \frac{T_c}{T_c + T_a + T_r}$$

Synchronization requrires mutual exclusion (critical sections)

- Acquire (lock)
  - Try to obtain the lock or proceed past barrier, may need to wait (spin/busy wait or block)
  - Must be atomic to both read and write
- Release (unlock)
  - Allow other processes to proceed past synchronization event

Atomic Instructions



$$\text{Critical-section efficiency} = \frac{T_c}{T_c + T_a + T_r}$$

Read-modify-write: specifies an atomic operation on a memory location

- Value in memory is read into a register
- Another value is written into location
    - May or may not depend on the value read
- Load and store must happen atomically in hardware

(Memory Consistency Model Many instances

- Test & Set
- Compare & Swap
- Fetch & Add/Sub/...

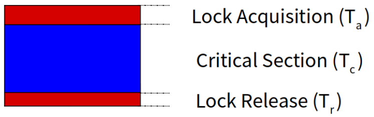
Test & Set

```
boolean test-and-set(*lock)
{
        boolean old=*lock;
        *lock=true;
        return old;
}
```

简单互斥算法

```
do
{  while(Test_And_Set_Lock(&lock)); (Memory Consistency Model
   //临界区
   lock = False;
   //剩余区
}while(True)
```

Improving Performance of Sync



$$\text{Critical-section efficiency} = \frac{T_c}{T_c + T_a + T_r}$$

Improving Performance of Sync

- Overhead: High contention, read/write miss, ...
- Rich interaction of HW-SW tradeoffs
- Must evaluate HW primitives and SW algorithms together
- Available HW primitives determine which SW algorithms perform well

Memory Consistency Model

memory consistency model

a specification of the allowed behavior of multithreaded programs executing with shared memory.

How Core or Cores Might Reorder Memory Accesses

- store-store reordering: non-FIFO write buffer
- load-load reordering: C1, C2 dynamically scheduled
- load-store reordering: Out-of-order cores
- store-load reordering: Out-of-order cores

Memory Consistency Model – Sequential Consistency

Sequential Consistency

The result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program.

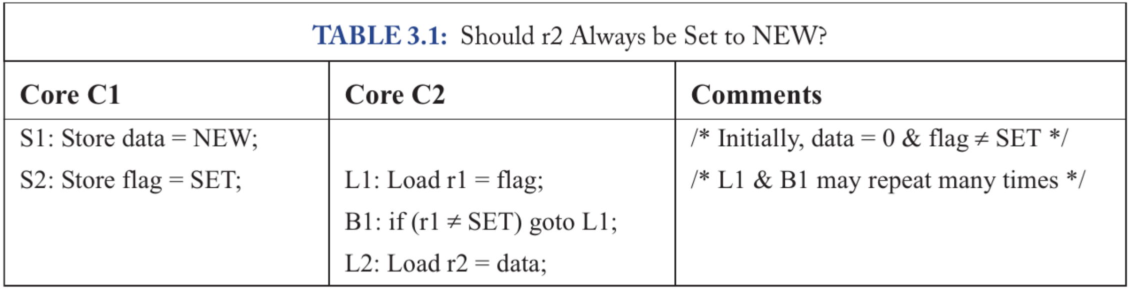


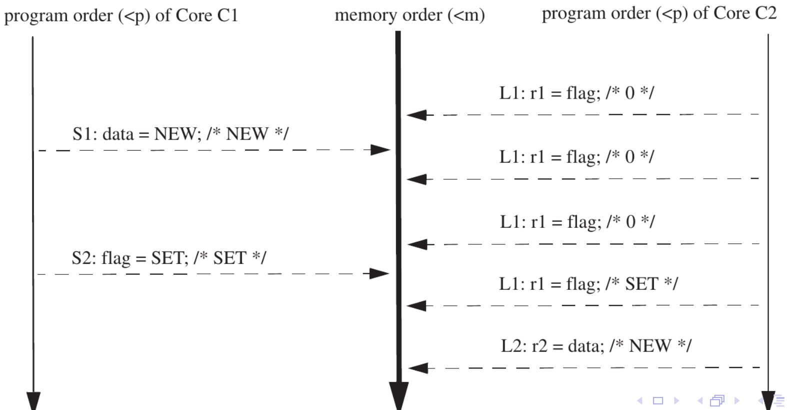
TABLE 3.1: Should r2 Always be Set to NEW?

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

Memory Consistency Model – Sequential Consistency

Sequential Consistency

The result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program.



Memory Consistency Model – Sequential Consistency

Sequential Consistency

The result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program.

- op1 <p op2 implies that op1 precedes op2 in that core' s program order
- op1 <m op2 implies that op1 precedes op2 in memory order.

Memory Consistency Model – Sequential Consistency

Sequential Consistency execution requires:

- All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses (i.e., a=b or a≠b).
  - If L(a) <p L(b) $\Longrightarrow$ L(a) <m L(b) /* Load$\longrightarrow$Load */
  - If L(a) <p S(b) $\Longrightarrow$ L(a) <m S(b) /* Load$\longrightarrow$Store */
  - If S(a) <p L(b) $\Longrightarrow$ S(a) <m L(b) /* Store$\longrightarrow$Load */
  - If S(a) <p S(b) $\Longrightarrow$ S(a) <m S(b) /* Store$\longrightarrow$Store */

Memory Consistency Model – Sequential Consistency

Sequential Consistency execution requires:

- All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses (i.e., a=b or a≠b).
  - If L(a) <p L(b) ⟹ L(a) <m L(b) /* Load⟶Load */
  - If L(a) <p S(b) ⟹ L(a) <m S(b) /* Load⟶Store */
  - If S(a) <p L(b) ⟹ S(a) <m L(b) /* Store⟶Load */
  - If S(a) <p S(b) ⟹ S(a) <m S(b) /* Store⟶Store */
- Every load gets its value from the last store before it (in global memory order) to the same address:
  Value of L(a) = Value of MAX $_{<m}$ {S(a) | S(a) <m L(a)}, where MAX $_{<m}$
  denotes “latest in memory order.”

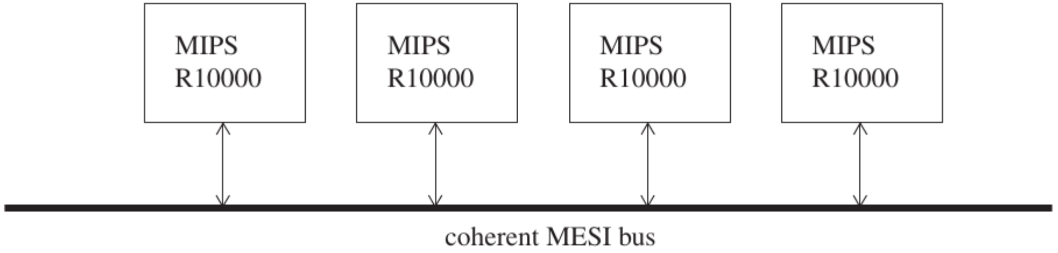
Memory Consistency Model – Sequential Consistency

Sequential Consistency execution requires:

- All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses (i.e., a=b or a≠b).
  - If L(a) <p L(b) $\Rightarrow$ L(a) <m L(b) /* Load$\longrightarrow$Load */
  - If L(a) <p S(b) $\Rightarrow$ L(a) <m S(b) /* Load$\longrightarrow$Store */
  - If S(a) <p L(b) $\Rightarrow$ S(a) <m L(b) /* Store$\longrightarrow$Load */
  - If S(a) <p S(b) $\Rightarrow$ S(a) <m S(b) /* Store$\longrightarrow$Store */
- Every load gets its value from the last store before it (in global memory order) to the same address:
  Value of L(a) = Value of MAX $_{<m}$ {S(a) | S(a) <m L(a)}, where MAX $_{<m}$
  denotes "latest in memory order."
- Atomic read–modify–write (RMW) instructions logically appear consecutively in the memory order (no interpose)

Memory Consistency Model – Sequential Consistency

The MIPS R10000 [18] provides a venerable, but clean, commercial example for a speculative microprocessor that implements SC in cooperation with a cache-coherent memory hierarchy.

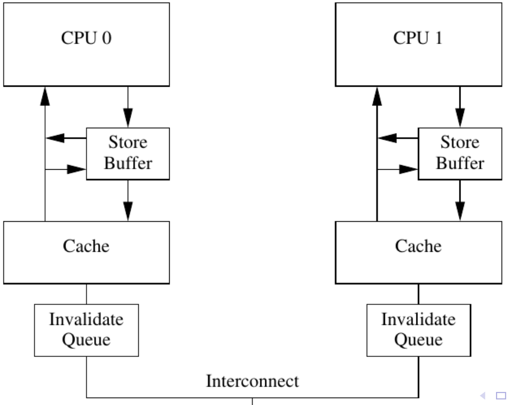


Memory Consistency Model – Total Store Order Consistency

SC 本质上要求：

- 缓存一收到总线事件，就可以在当前指令周期中迅速做出响应
- 处理器如实地按程序的顺序，把内存操作指令送到缓存，并且等前一条执行完后才能发送下一条

为了性能，现代处理器无法满足以上条件



Memory Consistency Model – Total Store Order Consistency

为了性能，现代处理器要做到：

- 缓存不会及时响应总线事件。如果总线上发来一条消息，要使某个缓存段失效，但是如果此时缓存正在处理其他事情（比如和 CPU 传输数据），那这个消息可能无法在当前的指令周期中得到处理，而会进入所谓的“失效队列（invalidation queue）”，这个消息等在队列中直到缓存有空为止。

Memory Consistency Model – Total Store Order Consistency

为了性能，现代处理器要做到::

- 缓存不会及时响应总线事件。
- 处理器一般不会严格按照程序的顺序向缓存发送内存操作指令。当然，有乱序执行（Out-of-Order execution）功能的处理器肯定是这样的。顺序执行（in-order execution）的处理器有时候也无法完全保证内存操作的顺序（比如想要的内存不在缓存中时，CPU 就不能为了载入缓存而停止工作）。

Memory Consistency Model – Total Store Order Consistency

为了性能，现代处理器要做到：

- 缓存不会及时响应总线事件。
- 处理器一般不会严格按照程序的顺序向缓存发送内存操作指令。
- 写操作尤其特殊，因为它分为两阶段操作：在写之前我们先要得到缓存段的独占权。如果我们当前没有独占权，我们先要和其他处理器协商，这也需要一些时间。同理，在这种场景下让处理器闲着无所事事是一种资源浪费。实际上，写操作首先发起获得独占权的请求，然后就进入所谓的由“写缓冲（store buffer）”组成的队列。写操作在队列中等待，直到缓存准备好处理它，此时写缓冲就被“清空（drained）”了，缓冲区被回收用于处理新的写操作。

Memory Consistency Model – Total Store Order Consistency

为了性能，现代处理器要做到：

- 缓存不会及时响应总线事件。
- 处理器不会严格按照程序的顺序向缓存发送内存操作指令。
- 写操作不会严格按照程序的顺序。

这些特性意味着，默认情况下，读操作有可能会读到过时的数据（如果对应失效请求还等在队列中没执行），写操作真正完成的时间有可能比它们在代码中的位置晚，一旦牵涉到乱序执行，一切都变得模棱两可。

Memory Consistency Model – Total Store Order Consistency

Total Store Order Consistency (TSO)

- program order v.s. memory order
  - If L(a) <p L(b) $\Longrightarrow$ L(a) <m L(b) /* Load$\longrightarrow$Load */
  - If L(a) <p S(b) $\Longrightarrow$ L(a) <m S(b) /* Load$\longrightarrow$Store */
  - If S(a) <p S(b) $\Longrightarrow$ S(a) <m S(b) /* Store$\longrightarrow$Store */
  - **NO:** *If S(a) <p L(b) $\Longrightarrow$ S(a) <m L(b) /* Store$\longrightarrow$Load */*
- Every load gets its value from the last store before it to the same address:

  ~~**Value of L(a) = Value of MAX $_{<m}$ {S(a) | S(a) <m L(a)}**~~ **/* Change 2: Need Bypassing */**

  Value of L(a) = Value of MAX $_{<m}$ {S(a) | S(a) <m L(a) **or S(a) <p L(a)**}
- **Fence** orders everything
- **Atomic operations** effectively drain the store buffer

Memory Consistency Model – Total Store Order Consistency

QUIZ:

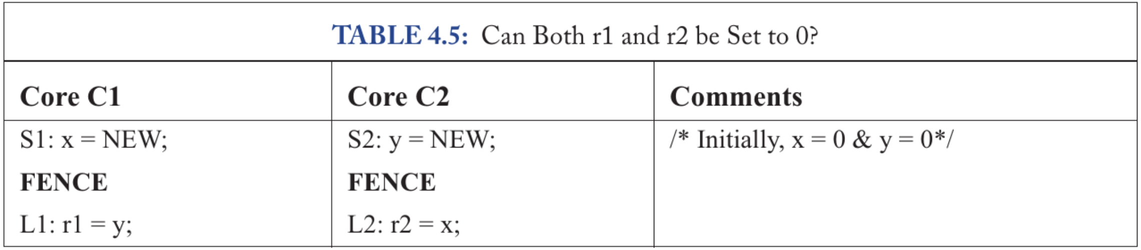


TABLE 4.5: Can Both r1 and r2 be Set to 0?

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

- (x, y) == (0, 0) ? with SC or TSO
- delete FENCE ?

Memory Consistency Model – Total Store Order Consistency

QUIZ:

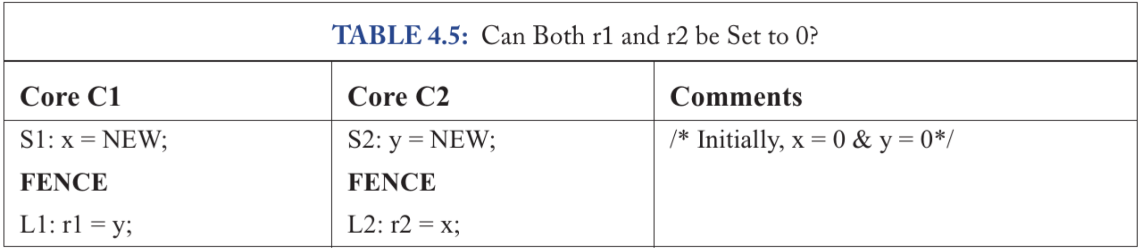


TABLE 4.5: Can Both r1 and r2 be Set to 0?

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

- (x, y) == (0, 0) ? with SC or TSO
- delete FENCE ?
- (x, y) == (0, 0) is not allowed with SC
- (x, y) == (0, 0) is possible with TSO :Load bypasses store

Memory Consistency Model – Total Store Order Consistency

TSO implementation / x86

- x86 中，所谓 RMW 即 intel 64 架构中的 locked 指令，其包括：隐式 locked 指令如 xchg，也包括别的（助记符中）以 lock 为前缀的 RMW 指令，如 LOCK ADD、LOCK CMPXCHG。这一类指令的特点正如其名，能够使用一个指令完成读取-修改-写入这样复合的行为。

Memory Consistency Model – Total Store Order Consistency

TSO implementation / x86

- x86 中，所谓 RMW 即 intel 64 架构中的 locked 指令，其包括：隐式 locked 指令如 xchg，也包括别的（助记符中）以 lock 为前缀的 RMW 指令，如 LOCK ADD、LOCK CMPXCHG。这一类指令的特点正如其名，能够使用一个指令完成读取-修改-写入这样复合的行为。
- locked 指令执行时在总线上断言 LOCK# signal，这个由处理器提供的信号将为系统总线上锁，此时其它控制该总线的处理器或总线代理将被阻塞。
- 作为优化，P6 处理器及之后的处理器型号在执行 locked 指令期间，当访问缓存在处理器内部的内存区域时，将不产生 LOCK# signal，而采用 cache locking，也即更新缓存并依赖缓存一致性算法确保同时仅有一个处理器能修改该内存区域中的数据。

Memory Consistency Model – other Consistency

Even more relaxed (less ordering)

- Weak Consistency
  - Enable higher performance: OoO write buffer, speculation, ...
  - Need more use of FENCE to enforce ordering
- Release Consistency
  - ACQUIRE and RELEASE: order memory operations in only one direction
  - ACQUIRE < load, store
  - Load, store < RELEASE
  - All ACQUIRE/RELEASE pairs are ordered