Computer Architecture

计 算 机 体 系 结 构

第19讲：TLP（5）

张献伟

xianweiz.github.io

DCS3013, 12/7/2022

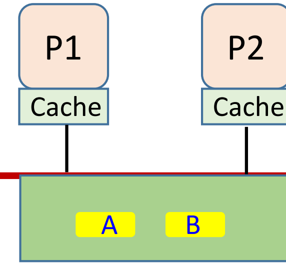
Review Questions

- Limits of snooping protocol
  Broadcast-based, hard to scale for many processors.
- For directory-based protocol, what is the entry content?
  Dirty bit + presence bits.
- Explain the storage overhead of dir-protocol?
  One entry per memory line, presence bits for all nodes/processors.
- Schemes to reduce storage overhead?
  Limited pointer scheme (P), sparse directory (M).
- What is 'home node' in dir-protocol?
  Node with memory holding the corresponding data for the line
- Intervention forwarding?
  Way to reduce #messages. Home node directly requests data from owner node, instead of involving requesting node.

Summary of Directory-base Coherence

- Primary observation: broadcast doesn't scale, but we don't need to broadcast to ensure coherence because often the number of caches containing a copy of a line is small
- Instead of snooping, just store the list of sharers in a directory and check the list when necessary
- One challenge on storage[存储]
  - Use hierarchies of processors or larger cache size
  - Limited pointer schemes: exploit fact that most processors not sharing line
  - Sparse directory schemes: exploit fact that most lines not in cache
- Another challenge on communication[通信]
  - Reduce messages sent (traffic) and parallelize trans (latency)

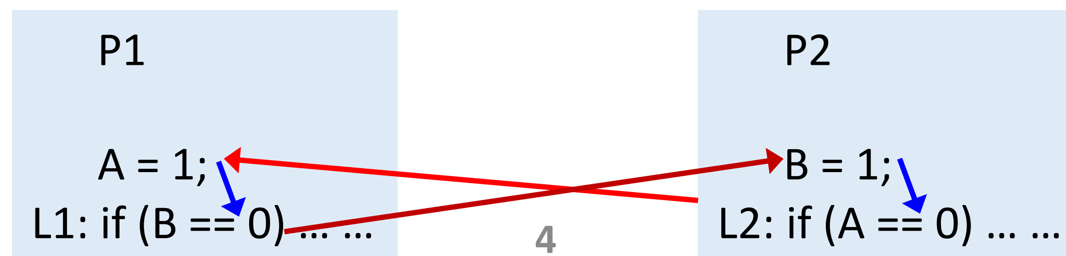
Example



- Assume that
  - Processes *P1* and *P2* are running on two different processors
  - Locations *A* and *B* are originally cached by both processors with the initial value of *0*
- If writes always take <u>immediate</u> effect and are immediately seen by other processors
  - Then impossible for both *IF* to be true

Reaching the IF means that either A or B must have been assigned the value 1 (i.e., IF is false)

- If write invalidate can be <u>delayed</u>, and the processor is allowed to continue during this delay
  - Then possible to that *P1* and *P2* haven't seen the invalidations before they attempt to read the values



Coherence vs. Consistency[对比]

- **Cache coherence** defines requirements for the observed behavior of reads and writes to the same memory location
  - Goal: to ensure that the memory system in a parallel computer behaves as if the caches were not there
    - A system without caches would have no need for cache coherence
  - Write value will be seen if sufficiently separated in time
- **Memory consistency** defines the behavior of reads and writes to different locations
  - The allowed behavior of memory should be specified whether or not caches are present
  - Coherence only guarantees that writes to address X will eventually propagate to other processors
  - Consistency deals with when writes to X propagate to other processors, relative to reads and writes to other addresses

http://15418.courses.cs.cmu.edu/tsinghua2017content/lectures/12_consistency/12_consistency_slides.pdf

Memory Consistency[内存一致性]

- Memory consistency specifies the ordering behaviors
  - What ordering behavior should be allowed?
  - Under what conditions?

- Example: a program running two threads, where A and B are initially both 0. What this program can output?
  - 01: (1)(2)(3)(4) or (3)(4)(1)(2)
  - 11: (1)(3)(2)(4) or (1)(3)(4)(2)
  - 00: intuitively, it shouldn't be possible

Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

中山大學 SUN YAT-SEN UNIVERSITY

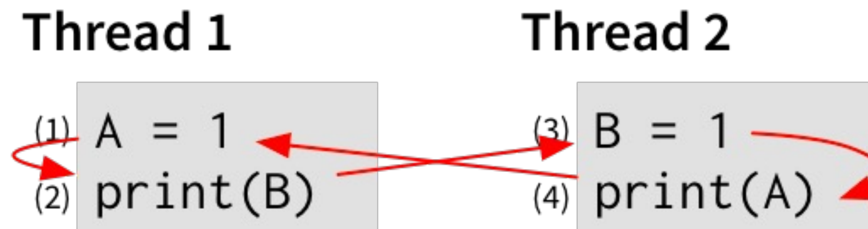
NSCC GZ

The Example

- x → y: x must happen before y
  - (2) to print 0: (2) → (3)
  - (4) to print 0: (4) → (1)
  - If each thread's events happen in order
    - (1) → (2)
    - (3) → (4)
- Start from (1), follow the edges
  - (1) → (2) → (3) → (4) → (1)
  - (1) must happen before itself ???

A=0: (4)->(1)
B=0: (2)->(3)

(4)->(1)->(2)->(3) → (4)



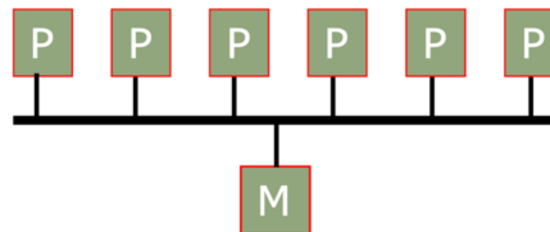
https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Memory Operation Ordering[访存先后]

- A program defines a sequence of loads and stores (this is the "program order" of the loads and stores)[程序顺序]
- Four types of memory operation orderings[4类顺序]
  - **W→R**: write to X must commit before subsequent read from Y
    - When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs
  - **R→R**: read from X must commit before subsequent read from Y
  - **R→W**: read to X must commit before subsequent write to Y
  - **W→W**: write to X must commit before subsequent write to Y
- A sequentially consistent memory system maintains all four memory operation orderings[顺序一致]
- Certain orderings can be violated ???[违背一些顺序？]

Sequential Consistency[顺序一致性]

- The most straightforward model for memory consistency
  - Intuitive idea: multiple threads running in parallel are manipulating a single main memory, and so everything must happen in order
    - But what order?
  - Intuitive order: the events in a single thread happen in the order in which they were written[程序顺序]
    - Intuitive to programmers
- **Sequential consistency** requires that the result of any execution be the same as though
  - Memory accesses executed by each proc. were kept in order
  - The accesses among different processors were arbitrarily interleaved



Sequential Consistency (cont.)

- Sequential consistency (SC)
  - Formalized by Leslie Lamport in 1979
  - "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" [看起来像。。。]
  - Defining SC is one of the many achievements that earned Lamport the Turing award in 2013
-

LESLIE LAMPORT

United States – 2013

CITATION

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

https://lamport.azurewebsites.net/pubs/time-clocks.pdf

The Examples

- With SC,
- Example-1:
  - Must delay the read of *A* or *B* (*A == 0* or B == 0) until the previous write has completed (*B = 1* or *A = 1*)
  - Cannot simply place the write in a buffer and continue with the read

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

- Example-2:
  - *print(B)*/*print(A)* cannot happen before *A = 1*/*B = 1*
    - 00 cannot be printed

Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```

Memory Consistency Model[一致性模型]

- **Memory consistency model** (or just "**memory model**") defines the allowed orderings of multiple threads on a multiprocessor
  - SC is one such model
    - E.g., orderings that print 01/11 are allowed, but not 00
- A memory consistency model is a contract between the hw and sw
  - The hw promises to only reorder operations in ways allowed by the model[硬件承诺]
  - In return, the sw acknowledges that all such reorderings are possible and that is needs to account for them[软件认可]

Thread 1

```
(1) A = 1
(2) print(B)
```

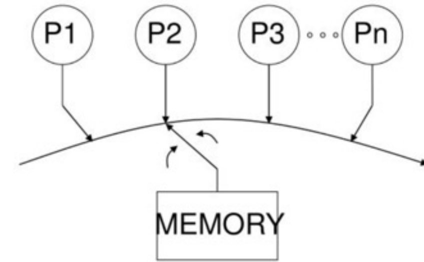
Thread 2

```
(3) B = 1
(4) print(A)
```


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Issues of SC[问题]

- SC: just like a switch to select thread to run, and runs its next event completely
  - Events happen in program order



- SC presents a simple programming paradigm
- But, SC reduces potential performance
  - Especially in a multiprocessor with a large number of processors or long interconnect delays
- Simplest way to implement SC
  - A processor delays the completion of any memory access until all the invalidations caused by that access are completed
  - Example: for a write miss, four processors share a block
    - 170 cycles for write: 50 cycles to establish ownership, then 10 cycles to issue each invalidate, and 80 cycles for an invalidate to complete and be acknowledged (50 + 40 + 80)

Optimizations[优化]

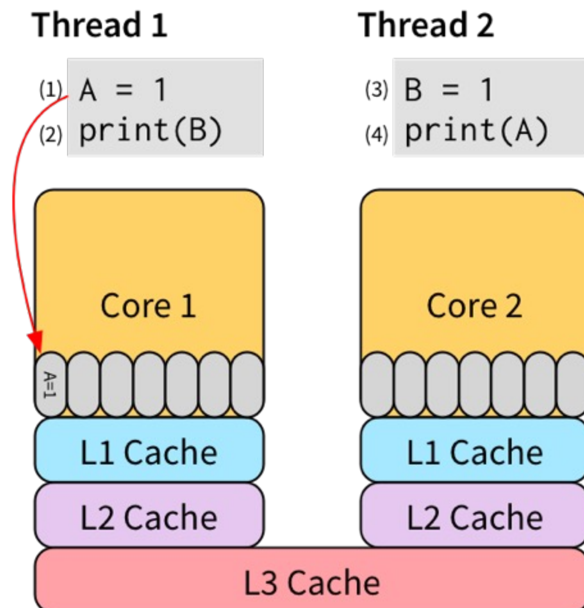
- **Goal**: develop a model that is simple to explain and yet allows a high performance implementation[好理解、高性能]

- **Solution-1**: develop ambitious implementations that preserve SC but use latency-hiding techniques to reduce the penalty[保持SC、隐藏时延]

- **Solution-2**: develop less restrictive memory consistency models that allow for faster hw[放宽顺序要求]
  - Such models can affect how the programmer sees the multiprocessor

The Example

- SC maintains a single view of memory
  - Cannot run *(2)* until *(1)* has become visible to every other thread
- No reason why *(2)* needs to wait until *(1)* completes
  - *(2)*: a read from B, *(1)*: a write to A
  - They don't interfere with each other at all
    - So should be allowed to run in parallel
  - Note that event *(1)* is very slow
    - A very high overhead
- SC greatly hurts performance
  - The model should be relaxed!!!
    - Event *(2)* should not wait for *(1)*

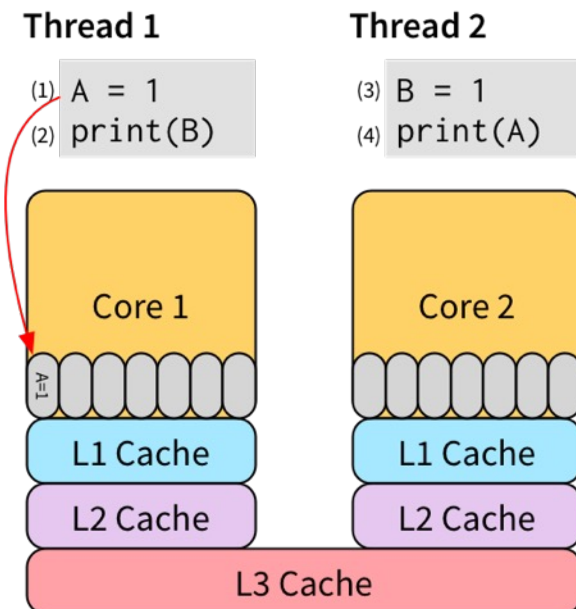


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

The Example (cont.)

- Place *write(1)* into a store buffer, rather than waiting for it to become visible
  - Then *(2)* could start immediately, rather than waiting for *(1)* to reach the L3
  - The store buffer is on-core: very fast to access
  - At some time in the future, the cache hierarchy will pull the write from the store buffer and propagate it through the L3 so that it becomes visible to other threads
- The buffer helps hide the write latency
- Preserves single-threaded behavior
  - Access: store buffer → memory

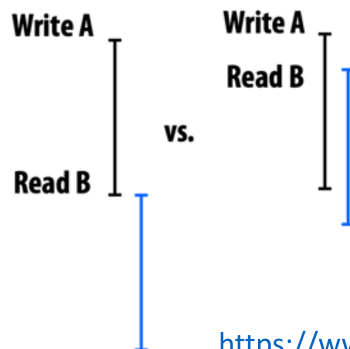
Is it still cache coherent?
Is the result correct?

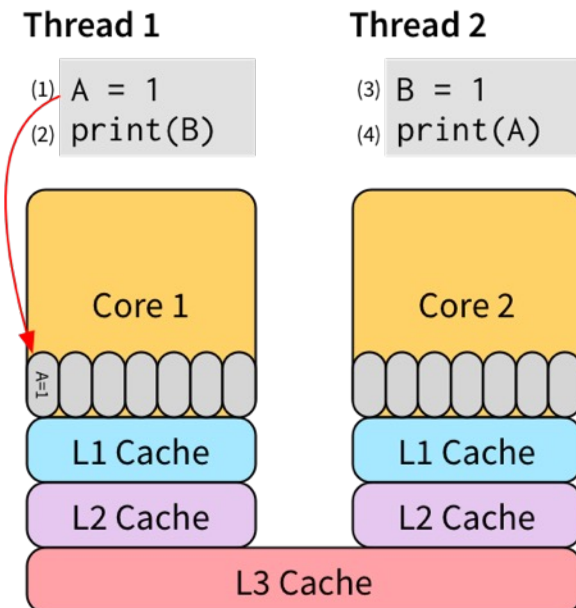


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Total Store Ordering[TSO一致性]

- TSO mostly preserves the same guarantees as SC, except that it allows the use of store buffers
  - There buffers hide write latency, making execution significantly faster
- Retains ordering among writes (that's why called 'total store ordering')[保证写顺序]
  - Relaxed only the W→R ordering
- Performance gain
  - Allow processor to hide latency of writes when later read is independent



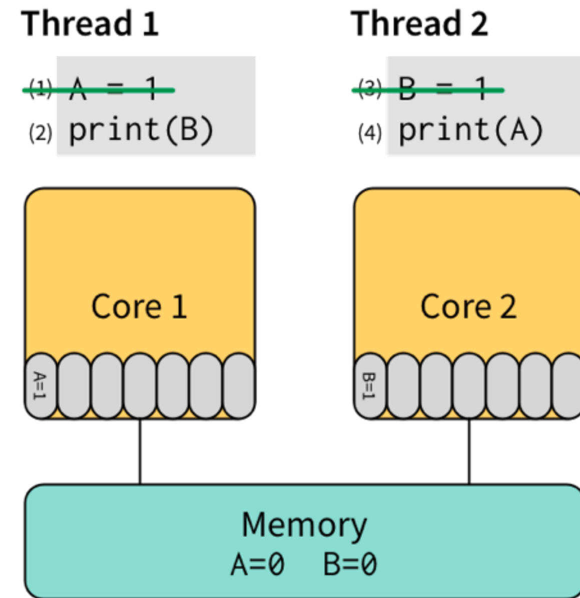


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

中山大學 SUN YAT-SEN UNIVERSITY

Total Store Ordering (cont.)

- While boosting performance, TSO allows behaviors that SC does not
  - I.e., programs running on TSO hw can exhibit behavior that programmers would find suprising
- The example: both threads first check their local store buffer, but fails to locate and then fetches from memory
  - This program can print 00
    - B=1 not in Core-1's buffer
    - A=1 not in Core-2's buffer
  - TSO cannot put into practices ???



https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Ordering on Different Architectures

- Actually, every modern architecture includes a store buffer, and so has a memory model at least as weak as TSO
  - x86 specifies a memory model that is very close to TSO
    - among the most well-behaved architectures in terms of the crazy behaviors it allows
  - ARM memory model is notoriously underspecified, but is essentially a form of *weak ordering*, gives very few guarantees
    - RISC-V: “RVWMO” (RISC-V Weak Memory Ordering)
    - Weak ordering allows almost any operation to be reordered, good for hardware optimizations but nightmare to program at the lowest levels

Architecture	Memory Model
x86_64	Total Store Order
Sparc	Total Store Order
ARMv8	Weakly Ordered
PowerPC	Weakly Ordered
MIPS	Weakly Ordered

https://www.cs.utexas.edu/~bornholt/post/memory-models.html
https://kernelgo.org/memory-model.html

Partial Store Ordering[PSO一致性]

- In TSO, only W→R order is relaxed
  - The W→W constraint still exists
    - Writes by the same thread are not reordered (they occur in program order)
- In **partial store ordering** (PSO), W → W is also relaxed

- Example: *A* and *flag* are initially 0s
  - SC: print '1' (when flag is 1, *A* must be 1 already)
  - TSO: print '1' (ditto)
  - PSO: may print '0' (when *flag* is 1, *A* can be 0 or 1)

Thread 1 (on P1)	Thread 2 (on P2)

A = 1;
flag = 1;
	while (flag == 0); // spinning if flag is 0
	print A;

http://15418.courses.cs.cmu.edu/tsinghua2017content/lectures/12_consistency/12_consistency_slides.pdf

Aggressive Memory Ordering???

- SC maintains all four memory operation orderings
- Certain orderings can be violated ???
  - W→R: store buffer to allow read execute earlier
  - W→W: reorder writes in the store buffer
    - Earlier write is a cache miss, later is a hit
  - R→W, R→R: processor may reorder independent instructions
    - Out-of-order execution
  - Note that all are valid optimizations if a program consists of a single instruction stream[对单线程都有效]

- What if we discard all four memory orderings?
  - Still a memory consistency model (**Release Consistency**)

http://15418.courses.cs.cmu.edu/tsinghua2017content/lectures/12_consistency/12_consistency_slides.pdf

Release Consistency[RC一致性]

- Release Consistency (RC)
  - Processors support special synchronization operations
  - Memory accesses before memory fence instruction must complete before the fence issues
  - Memory accesses after fence cannot begin until fence instruction is complete
  - 硬件不再对一致性做过多保证，需要软件介入以控制执行行为

reorderable reads and writes here
...
MEMORY FENCE
...
reorderable reads and writes here
...
MEMORY FENCE

Express Synchronization[同步]

- ’00’ is not allowed in SC (the example)
  - Suppose architecture is of RC model, how to get the same effect with SC (i.e., no ‘00’)?
- All modern architectures include **synchronization** operations to bring their relaxed memory models under control when necessary
  - Most common operation: barrier (or fence)

- A barrier inst forces all memory operations before it to complete before any memory operation after it can begin
  - I.e., a barrier inst effectively <u>reinstates SC at a particular point</u> in program execution

Thread 1	Thread 2
(1) A = 1	(3) B = 1
(2) print(B)	(4) print(A)

S1: Store x = NEW;	S2: Store y = NEW;
FENCE	FENCE
L1: Load r1 = y;	L2: Load r2 = x;

FENCE: S1/S2 must be completely done before L1/L2