Memory Consistency Models

Nima Honarmand

Why Consistency Models Matter

- Each thread accesses two types of memory locations
  - ***Private***: only read/written by that thread – should conform to sequential semantics
    - "Read A" should return the result of the last "Write A" in program order
  - ***Shared***: accessed by more than one thread – *what about these?*
- Answer is determined by the ***Memory Consistency Model*** of the system
- Determines the order in which shared-memory accesses from different threads can "appear" to execute
  - In other words, determines what value(s) a read can return
  - More precisely, *the set of all writes (from all threads) whose value can be returned by a read*

Coherence vs. Consistency: Example 1

{A, B} are memory locations; {r_1, r_2} are registers.
Initially, A = B = 0

Processor 1	**Processor 2**
Store A ← 1	Store B ← 1
Load r_1 ← B	Load r_2 ← A

- Assume coherent caches
- Is this a possible outcome: {r_1=0, r_2=0}?
- Does cache coherence say anything?
  - Nope, different memory locations

Coherence vs. Consistency: Example 2

{A, B} are memory locations; {r_1, r_2, r_3, r_4} are registers.
Initially, A = B = 0

Processor 1	Processor 2	Processor 3	Processor 4
Store A ← 1	Store B ← 1	Load r_1 ← A Load r_2 ← B	Load r_3 ← B Load r_4 ← A

- Assume coherent caches
- Is this a possible outcome: {r_1=1, r_2=0, r_3=1, r_4=0}?
- Does cache coherence say anything?

Coherence vs. Consistency: Example 3

{A, B} are memory locations; {r_1, r_2, r_3} are registers.
Initially, A = B = 0

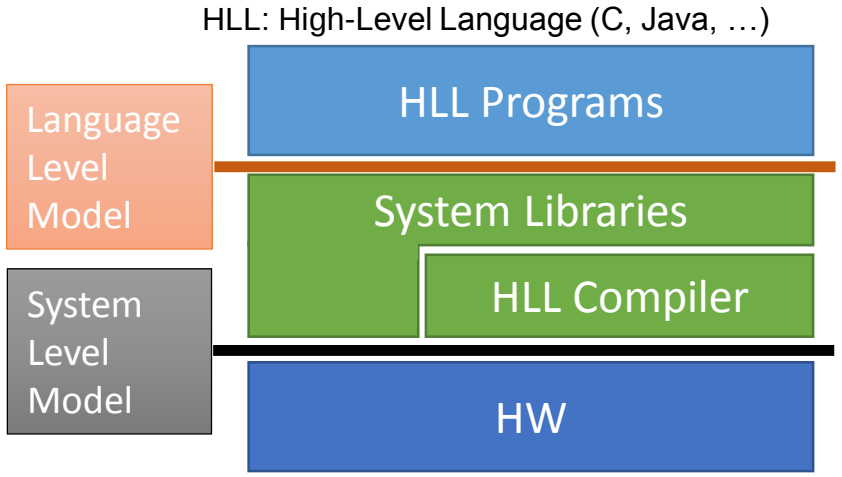
Processor 1	**Processor 2**	**Processor 3**

Store A ← 1	Load r_1 ← A
	if (r_1 == 1)
	  Store B ← 1
		Load r_2 ← B
		if (r_2 == 1)
		  Load r_3 ← A

- Assume coherent caches
- Is this a possible outcome: {r_2=1, r_3=0}?
- Does cache coherence say anything?

Memory Models at Different Levels

- Hardware implements *system-level memory model*
  - Shared-memory ordering of ISA instructions
  - Contract between hardware and ISA-level programs



- Compiler/System Libraries implement *language-level memory model*
  - Shared-memory ordering of HLL constructs
  - Contract between HLL implementation and HLL programs
- Compiler/system libraries use system-level model to implement program-level model

Who Cares about Memory Models?

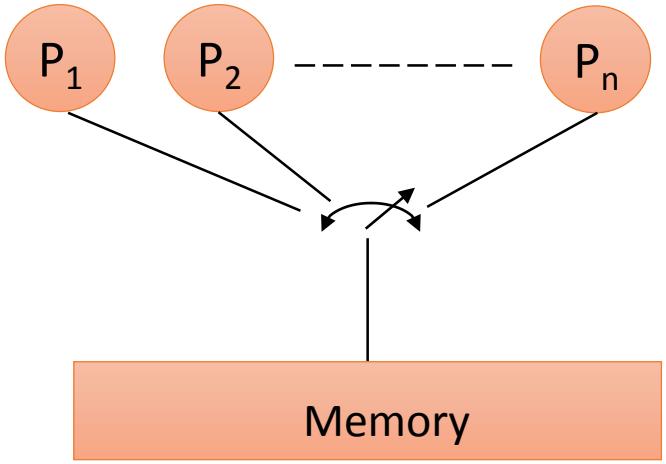
- Programmers want:
  - A framework for writing correct parallel programs
  - Simple reasoning -“principle of least astonishment”
  - The ability to express as much concurrency as possible
- Compiler/Language designers want:
  - To allow as many compiler optimizations as possible
  - To allow as much implementation flexibility as possible
  - To leave the behavior of “bad” programs undefined
- Hardware/System designers want:
  - To allow as many HW optimizations as possible
  - To minimize hardware requirements / overhead
  - Implementation simplicity (for verification)

Intuitive Model: Sequential Consistency (SC)

"A multiprocessor 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 this sequence in the order specified by its program."

-Lamport, 1979

Processors issue memory ops in program order



Each op executes atomically (at once), and switch randomly set after each memory op

Problems with SC: HW Perspective

- HW designers are not happy with SC
  - Naïve SC implementation forbids many processor performance optimizations
    - Store buffers
    - Out-of-order execution of accesses to different locations
    - Combining store buffers and MSHRs
    - Responding to remote GetS after a GetM before receiving all invalidation acks in a 3-hop protocol
    - …
- Aggressive (high-performance) SC implementation requires complex HW
  - Will see examples later

→ HW needs models that allow performance optimizations without complex hardware

Problems with SC: HLL Perspective

- SC limits many compiler optimizations on shared memory
  - Register allocation
  - Partial redundancy elimination
  - Loop-invariant code motion
  - Store hoisting/sinking
  - …
- SC is not what programmers really need
- E.g., an SC program still can have data races, making the program hard to reason about

→ HLLs need models that allow optimizations and are easier to reason about

System-Level Memory Models

Relaxed Memory Models

- To keep hardware simple and performance high, relax the ordering requirements
  → ***Relaxed Memory Models***

- SC has two ordering requirements
  - Memory operations should appear to be executed in program order
  - Memory operations should appear to be executed atomically
    - Effectively, extending the "write serialization" property of coherence to all write operations

- A relaxed memory model may relax any of these two requirements

Aspects of Relaxed Memory Models

- ***Local instruction ordering***
  - What memory operations should appear to have been sent to memory in program order?
- ***Store atomicity***
  - Can a write be observed by one processor before it's been made visible to all processors?
- ***Safety nets***
  - How to enforce orderings that are relaxed by default?
  - How to enforce atomicity for a memory op (if relaxed by default)?

Local Instruction Ordering

- Typically, defined between a pair of instructions
- Memory model specifies which orders should be preserved and which ones can be relaxed
- Typically, the ordering rules fall into three categories:
  1. Ordering requirements between normal reads and writes
     - **W→R**: a write and a following read in program order
     - **W→W**: a write and a following write in program order
     - **R→R**: a read and a following read in program order
     - **R→W**: a read and a following write in program order
  2. Ordering requirements between normal ops and special instructions (*e.g.*, ***fence*** instructions)
  3. Ordering requirements between special instructions

Local Instruction Ordering

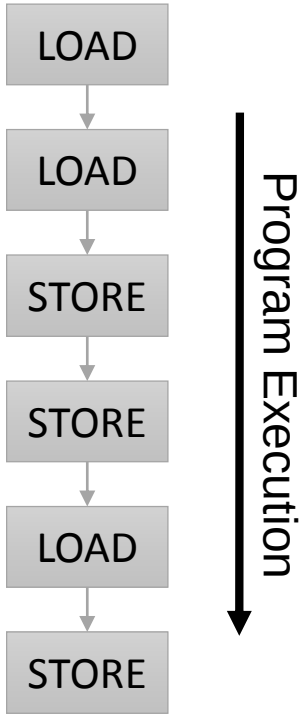
- Often there are exceptions to general rules
  - E.g., let's assume a model relaxes R→R in general
  - One possible exception: R→R not relaxed if the addresses are the same
  - Another possible exception: R→R not relaxed if the second ones address depends on the result of the first one
- Typically, it's the job of a processor core to ensure local ordering
  - Hence called "local ordering"
  - *E.g.*, if R→R should be preserved, do not send the second R to memory until the first one is complete
  - Requires the processor to know when a memory operation is ***performed*** in memory

"Performing" a memory operation
[Scheurich and Dubois 1987]

- A Load by P_i is performed with respect to P_k when new stores to same address by P_k can not affect the value returned by the load
- A Store by P_i is performed with respect to P_k when a load issued by P_k to the same address returns the value defined by this (or a subsequent) store
- An access is performed when it is performed with respect to all processors
- A Load by P_i is globally performed if it is performed and if the store that is the source of its value has been performed

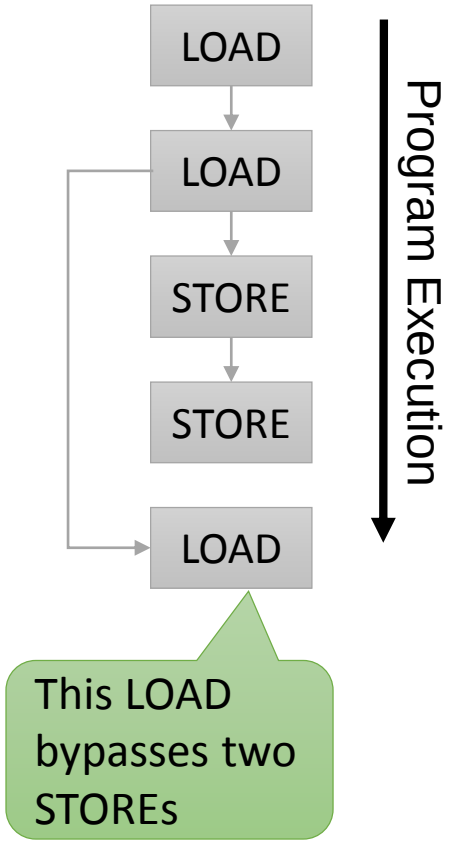
Local Ordering: No Relaxing (SC)

- Formal Requirements:
  - Before LOAD is performed w.r.t. any other processor, all prior LOADs must be globally performed and all prior STOREs must be performed
  - Before STORE is performed w.r.t. any other processor, all prior LOADs must be globally performed and all previous STORE be performed
  - Every CPU issues memory ops in program order
- SC: Perform memory operations in-program-order
  - No OoO execution for memory operations
  - Any miss will stall the memory operations behind it



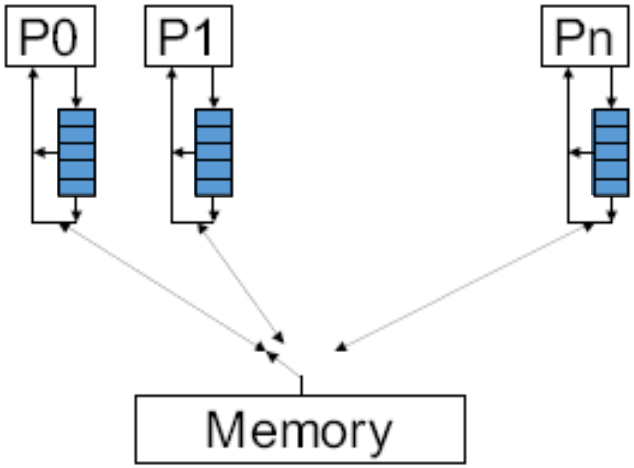
Local Ordering: Relaxing W→R

- Initially proposed for processors with in-order pipelines
  - Motivation: allow Post-retirement Store Buffers
- Later loads can bypass earlier stores to independent addresses
- Examples of memory models w/ this relaxation
  - Processor Consistency [Goodman 1989]
  - Total Store Ordering (TSO) [Sun SPARCv8]



Detour: Post-Retirement Store Buffer

- Allow reads to bypass incomplete writes
  - Reads search store buffer for matching values
  - Hides all latency of store misses in uniprocessors
- Writes are still ordered w.r.t. other writes
- Reads are still ordered w.r.t. other reads



Local Ordering: Relaxing W→W & R→RW

- In Processor Consistency and TSO, W→W and R→R are still enforced
- Naïvely enforcing R→R:
  - prevents OoO execution of independent loads
  - prevents having multiple pending load misses (lock-up free caches)
- Naïvely enforcing W→W:
  - prevents OoO execution of independent writes
  - prevents having multiple pending write misses (lock-up free caches)
  - W→W prevents "write combining" in the store buffer or MSHR
- By allowing RW→RW, we enable all conventional uni-processor optimizations for memory operations
  - Note: relaxations are for accesses to different addresses; same-addr accesses are ordered, just like uni-processors

Store Atomicity

- ***Store atomicity***: property of a memory model stating the existence of a total order of all writes

- Lack of store atomicity can result in ***non-causal*** executions
  - ***Causality:*** if I see something and tell you, you will see it too.

{A, B} are memory locations; {r_1, r_2, r_3} are registers.
Initially, A = B = 0

Processor 1	**Processor 2**	**Processor 3**

Store A ← 1	Load r_1 ← A
	if (r_1 == 1)
	  Store B ← 1
		Load r_2 ← B
		if (r_2 == 1)
		  Load r_3 ← A

- Processor 3 seeing Store B but not Store A is not a causal behavior → results in astonishing behavior

Relaxing Store Atomicity

- Relaxation comes in one of two flavors
  1. A thread can see its own write early (*i.e.*, before write is globally performed)
     - Enables store-to-load forwarding in the store buffer
  2. A thread can see another thread's write early (*i.e.*, before it is globally performed)
     - Can reduce "remote cache hit" penalty
       - "Remote cache hit": a cache miss which hits in a remote cache
       - E.g., can respond to a remote GetS before all Inv-Acks for the local GetM are received
     - Simplifies implementation of hardware-multithreading
       - Threads running on the same core can see each others' writes before the write is globally performed

Implementing Store Atomicity

- On a bus…
  - Trivial (mostly); store is globally performed when it reaches the bus
- With invalidation-based directory coherence…
  - Writer cannot reveal new value till all invalidations are ack'd
- With update-based coherence…
  - Hard to achieve… updates must be ordered across all nodes
- With SMT multiprocessors & shared caches
  - Cores that share a cache must not see one another's writes! (ugly!)

Safety Nets

- Sometimes, one need to enforce orderings that are relaxed by default
- For example, consider Dekker's algorithm
  - Works as advertised under SC
  - Can fail with relaxed W→R
    - P1 can read B before writing A to memory/cache

Processor 1

```
Lock_A:
3  A = 1;
1  if (B != 0)
       { A = 0; goto Lock_A; }
   /* critical section*/
   A = 0;
```

Processor 2

```
Lock_B:
4  B = 1;
2  if (A != 0)
       { B = 0; goto Lock_B; }
   /* critical section*/
   B = 0;
```

Safety Nets

- Solution: force ordering from the write to the read

Processor 1

```
Lock_A:
A = 1;
<drain the write>
if (B != 0)
    { A = 0; goto Lock_A; }
/* critical section*/
A = 0;
```

Processor 2

```
Lock_B:
B = 1;
<drain the write>
if (A != 0)
    { B = 0; goto Lock_B; }
/* critical section*/
B = 0;
```

- How to force the hardware to do that?
  - Use a safety net mechanism

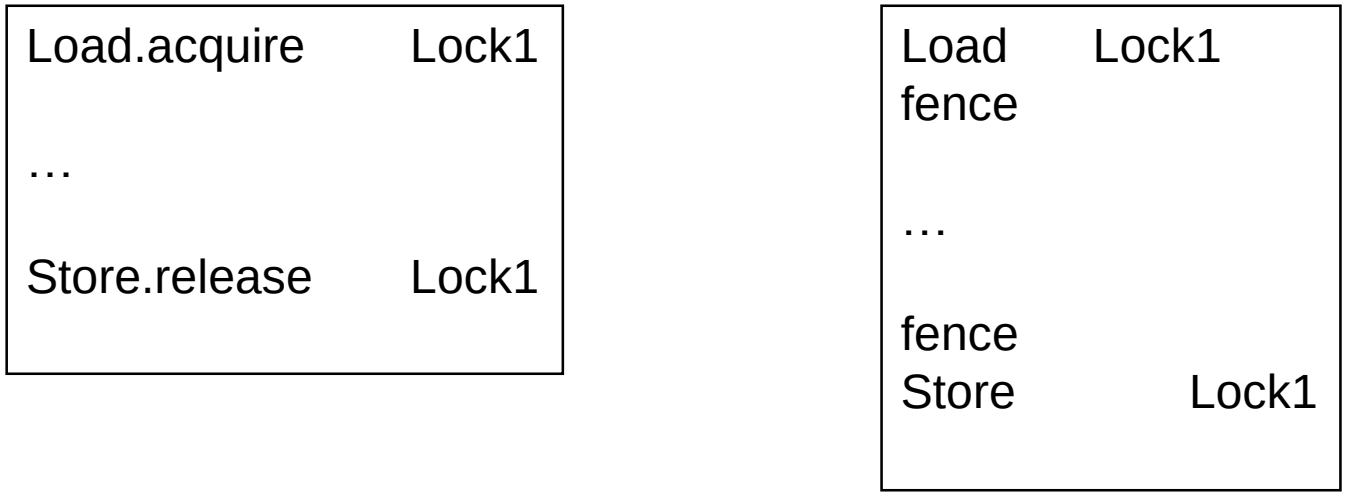
Three Approaches to Safety Nets (1/2)

- Approach 1: Using explicit ***fence*** instructions (aka memory barrier)
  - Orders instructions preceding the fence before the instructions following the fence
  - A fence can be partial: only orders certain instructions (for example LD/LD fence, ST/ST fence, etc.)
- Approach 2: Using Atomic RMW instructions
  - because they have a read and a write together
  - For example, if only W→R is relaxed, order can be enforced by making either W or R an RMW

Three Approaches to Safety Nets (2/2)

- Approach 3: Annotate loads/stores that are used for "synchronization" to enforce ordering between them and other memory operations
  - Example: a lock/unlock operation

Special load/stores vs. Fences



Mem. Model Example: **TSO**

- Total Store Ordering
  - Sun SPARC processors
  - Believed to be very similar to Intel x86 processors
- Local ordering relaxation:
  - relaxes W→R (if accessing independent addresses)
- Atomicity relaxation:
  - Can read own write early (before the write is globally performed)
  - Otherwise, there is a total order of stores
- Safety Nets: atomic RMW instructions and Fences

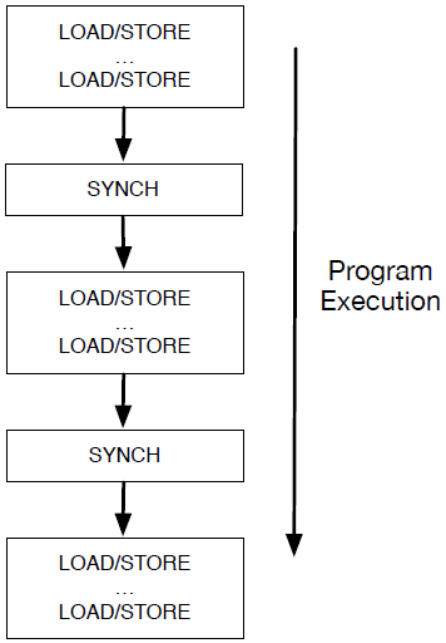
TSO: HW Perspective

- Allows a FIFO-ordered, non-coalescing store buffer
  - Typically maintains stores at word-granularity
  - Loads search buffer for matching store(s)
    - Some ISAs must deal with merging partial load matches
  - Coalescing only allowed among adjacent stores to same block
  - Must force buffer to drain on RMW and Fence
  - Often, this is implemented in same HW structure as (speculative) store queue

- Can hide store latency!
  - But, store buffer may need to be quite big
    - Stores that will be cache hits remain buffered behind misses
  - Associative search limits scalability
    - Often no more than 64 entries

Mem. Model Example: **Weak Ordering**

- Rationale: in a well-synchronized program, all reorderings inside a critical section should be allowed
  - Data-race freedom ensures that no other thread can observe the order of execution
- Mark instructions used for synchronization
- Local ordering relaxation:
  - All re-orderings allowed between "SYNCH" ops (if accessing independent addresses)
  - No re-ordering allowed across "SYNCH" ops
- Atomicity relaxation:
  - Can read own write early (before the write is globally performed)
- Safety net: SYNCH ops

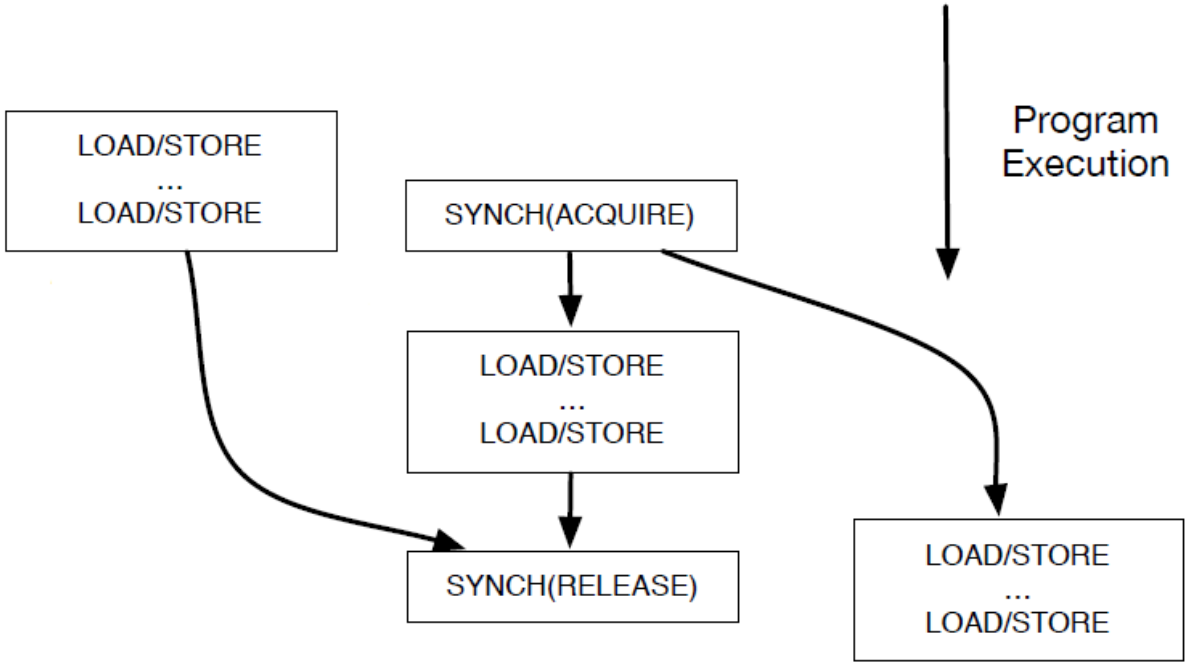


Weak Ordering

- Relaxes all orderings between ordinary operations
- Before an ordinary LOAD/STORE is allowed to perform w.r.t. any processor, all previous SYNCH accesses must be performed w.r.t. everyone
- Before a SYNCH access is allowed to perform w.r.t. any processor, all previous ordinary LOAD/STORE accesses must be performed w.r.t. everyone
- SYNCH accesses are sequentially consistent w.r.t. one another

Mem. Model Example: **Release Consistency**

- Similar to Weak Ordering but distinguishes between
  - SYNCH op used to start a critical section (Acquire)
  - SYNCH op used to end a critical section (Release)



Release Consistency

- Local ordering relaxation:
  - All reorderings allowed between SYNCH ops (if accessing independent addresses)
  - Normal ops following a RELEASE do not have to be delayed for the RELEASE to complete
  - An ACQUIRE needs not to be delayed for previous normal ops to complete
  - Normal ops between SYNCH ops do not wait for or delay Normal ops outside the critical section
- Atomicity relaxation:
  - Can read own or others' writes early
- Safety net: Acquire and Release ops

WO and RC: Hardware Perspective

- Enables all uni-processor optimizations for ordinary load/stores
- Need special support for SYNCH ops to ensure the required ordering

Enhancing Implementations of Memory Models

General Approach

- Allow accesses to partially or fully proceed even though ordering rules demand them to be delayed
- Detect and remedy cases when the early access would result in incorrect behavior
  - How to detect? Using observed coherence requests
  - How to remedy? Re-issue the access to the memory system
- Result: common case proceeds with high speed while still preserving correctness
- Two techniques
  - ***Prefetching***
  - ***Speculative Execution***

Prefetching

- Prefetching is classiffied as:
  - Binding vs non-binding
  - Hardware vs software
- Cache coherent machines can provide non-binding prefetching
- Non-binding prefetching:
  - does not affect the correctness for any consistency model

  → can be used as performance booster
- Can use:
  - for a read: read prefetch
  - for a write: read-exclusive prefetch
- Bring data into the cache and perform the operation when the memory consistency model allows

What if There Is an Intervening Access?

- After a read prefetch
  - a remote processor writes:
    - No problem
  - a remote processor writes:
    - our copy gets invalidated
    - when the local read is actually issued, it misses
- After a read-exclusive prefetch
  - a remote processor writes: same as above
  - a remote processor reads:
    - our copy loses exclusivity
    - when the local write is issued, it misses

Implementation

- Assume a processor with LD and ST queues
  - Local access kept in queues, it is delayed until it is correct to do it (per memory model)
- Hardware automatically issues:
  - Read prefetch: for reads in the buffer
  - Read-exclusive prefetch: for writes (and RMW) in the buffer
- Prefetches are buffered in a special prefetch buffer
  - Sent to memory as soon as possible
- Prefetch first checks the cache
  - If data there in the right state, then prefetch is discarded
- Prefetch response is placed into the cache
- If processor references line before prefetch has arrived, no additional request is issued to memory (combining)

Example

Ex 1

lock L	(miss)
write A	(miss)
write B	(miss)
unlock L	(hit)

Ex 2

lock L	(miss)
read C	(miss)
read D	(hit)
read E[D]	(miss)
unlock L	(hit)

- Assume: cache hit =1 cycle, cache miss=100
  - EX1: SC:301, RC:202, with prefetching (SC or RC): 103
  - EX2: SC:302, RC:203, with prefetching: 203 SC and 202 RC
- note: E[D] is not allowed to perform until reads to C and D complete (in SC) or lock access completes (in RC)

Speculative Execution

- Allow processor to consume return values out-of-order regardless of the consistency constraints
- Goal: allow speculative execution for loads
  - Loads are often sources in instruction dependency chains
  - Important to execute as early as possible
- Consider access *u* (long latency) followed by *v* (a load)
  - Assume that the consistency model requires *v* to be delayed until *u* completes
- Speculative execution:
  - the processor obtains or assumes a value for *v* before *u* completes, and proceeds
- When u completes:
  - if current value of v is as expected, speculation was successful
  - if current value is different: throw out the computation that depended on the value of v and re-execute

Required Mechanisms

- ***Speculation*** mechanism: obtain the speculated value
- ***Detection*** mechanism: how to detect incorrect speculation
- ***Correction*** mechanism: to repeat the computation if mis-speculated

Mechanisms

- Speculation mechanism: Perform the access
  - if cache hit: return immediately
  - if miss: takes longer
- Detection mechanism:
  - Naïve: repeat the access when legal and compare the value
  - Better: keep the data in cache and monitor if you received a coherence transaction for it
    - Result: cache accessed once rather than twice (as prefetch)
  - Coherence transactions: invalidation
    - false sharing and same-value update cause un-necessary mis-speculations
  - What if cache displacement?
    - Conservatively assume a mis-speculation

Mechanisms

- Correction mechanism:
  - Discard the computation that depended on the speculated value and repeat the access and computation
  - Similar mechanisms as in processors with branch prediction
    - Branches mis-speculation instructions past branch are discarded
    - Load mis-speculation: instructions past the load are discarded and the load is retried

Example

Ex 2

lock L	(miss)
read C	(miss)
read D	(hit)
read E[D]	(miss)
unlock L	(hit)

- Value of D is allowed to be used to access E[D]
- Both RC and SC complete in 104 cycles

Summary of Prefetching & Speculation

- Speculation allows out-of-order load execution
  - Naturally supported by OoO processors
  - Hardware coherence is needed to allow mis-speculation detection
- Exclusive prefetching allows out-of-order issuing of GetMs for stores
  - Hides much of the store latency
  - Again relies on hardware coherence
- Both require lock-up free caches

→ Performance of strong models (like SC and TSO) get closer to relaxed models (like RC and WO)