Advanced Computer Architecture

Memory Consistency/ Ordering

Fall 2016

Pejman Lotfi-Kamran

Adapted from slides originally developed by Profs. Hill, Hoe, Falsafi and Wenisch of CMU, EPFL, Michigan, Wisconsin

Where Are We?

Fr	Sa	Su	Mo	Tu
	27-Shahrivar		29-Shahrivar	
	3-Mehr		5-Mehr	
	10-Mehr		12-Mehr	
	17-Mehr		19-Mehr	
	24-Mehr		26-Mehr	
	1-Aban		3-Aban	
	8-Aban		10-Aban	
	15-Aban		17-Aban	
	22-Aban		24-Aban	
	29-Aban		1-Azar	
	6-Azar		8-Azar	
	13-Azar		15-Azar	
	20-Azar		22-Azar	
	27-Azar		29-Azar	
	4-Dey		6-Dey	

- This Lecture
  - Consistency (1)
- Next Lecture:
  - Consistency (2)

Review: Memory Instructions in the Processor

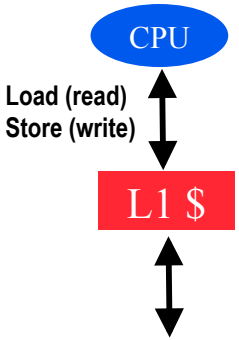
Memory instructions (loads/stores)

In simple pipelines

- Fetched and decoded in pipeline (IF/ID)
- Go through address calculation (EX)
- Access L1 with address (MEM)

In modern OoO pipelines

- Also renamed after decode
- Wait in the MEM stage in an LD/ST queue
- Calculate address when operands ready
- Access L1 while preserving order



Picture not complete: Store (or Write) Buffer

After instructions leave the CPU

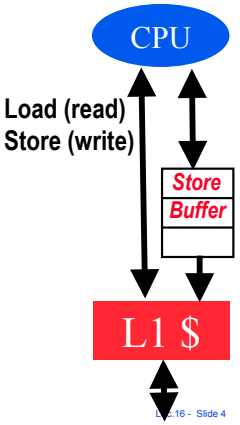
Stores go first to Store Buffer

- A few entries to buffer stores before L1
- As wide as a Word
- Store misses can wait in store buffer
- Stores can go to L1 later to open up ports for loads

When ports available, stores access L1

Loads must check store buffer

- L1 may be stale
- Must get result out of the store buffer



Store Buffer is Coalescing

Instruction order preserved in the CPU

Once stores leave CPU, no order in store buffer

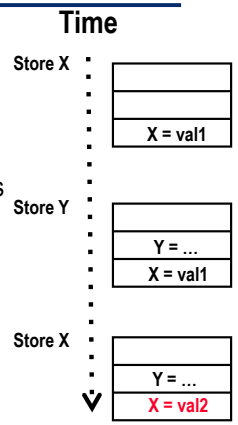
Store buffer is FIFO (to avoid) starvation

But, order is not necessary for correctness

Store values coalesce

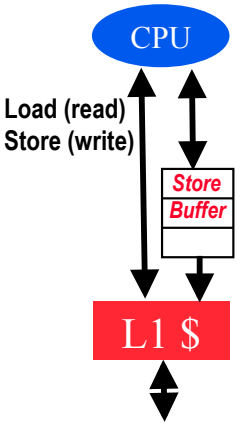
- Multiple stores to the same address
- Are merged into one entry
- Only the last value is preserved

Any load coming out of CPU will load the last value that left the CPU



Ordering instructions when they leave CPU

- Assume all instructions leave CPU in program order
- Store buffer has the latest values
- So, as long, as store buffer accessed before L1, ordering is preserved

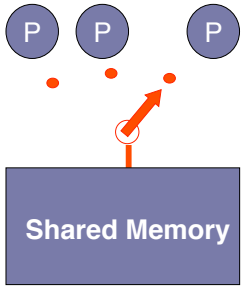


Review: Memory Consistency Models

- The ordering across addresses in a multiprocessor is called the memory consistency model
- A contract between software (at ISA level) and hardware
- Consistency != Coherence
  - Consistency has to do with ordering between accesses
  - It says nothing about caches (memory ordering is a problem with or without caches)
  - Coherence is about multiple copies of the same address

Strict memory ordering: Sequential Consistency

- SC [LAMPORT]
  - MP should behave like "multitasked" uni
- Memory should appear
  - in program order & atomic
  - e.g., critical section
    - ① lock
    - ② modify data
    - ③ unlock



Intuitive, but naïve implementations limit parallelism

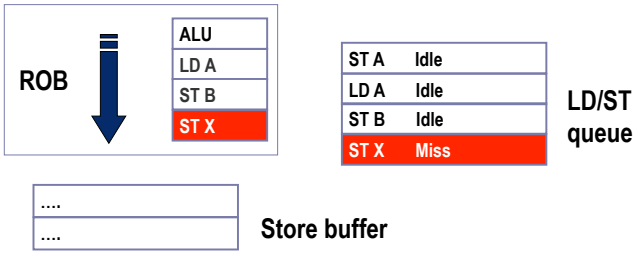
Sufficient Conditions for SC

- Every proc. issues memory ops in program order
- Memory ops happen (start and end) atomically
  - must wait for store to complete before issuing next memory op
  - after load, issuing proc waits for load to complete, before issuing next op
- Easily implemented with a shared bus

Example Code

```
ST X      ; misses in L1 & L2
ST B      ; hits in L1
LD A      ; hits in L1
ALU
ST A
LD Y      ; misses in L1 & L2
ST Z      ; misses in L1 & L2
```


Memory Ordering in Naiive SC



- ST X, LD Y, LD Z are long-latency misses
- X, Y, Z, A, B are unrelated → need not be ordered
  - CPU does not know they are unrelated

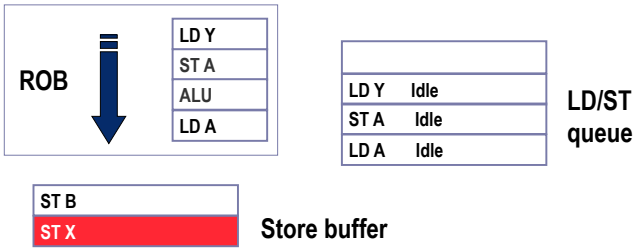
① ST X blocks CPU for hundreds of cycles

② Can not overlap LD Y & LD Z with ST X

In Naiive SC, accesses are not overlapped

```
ST X      ; misses in L1 & L2
ST B      ; does not access cache
LD A      ; does not access cache
ALU       ; waiting for LD A
ST A      ; waiting for ALU
LD Y      ; instruction not fetched
ST Z      ; instruction not fetched
```

SC + Store Buffer



Store buffer

+ Removes one pending store from the head of ROB
+ Remove back-to-back stores w/o coalescing (must preserve order)

In SC + Store Buffer

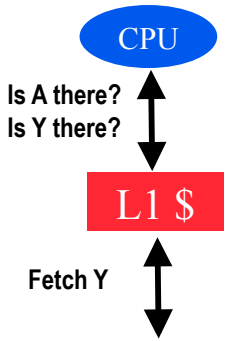
```
ST X    ; removed from pipeline (wait in SB)
ST B    ; removed from pipeline (wait in SB)
LD A    ; does not access cache
ALU     ; waiting for LD A
ST A    ; waiting for ALU
LD Y    ; does not access cache
ST Z     ; instruction not fetched
```


But, CPU can peek into L1 without loading values

- Check if blocks are in L1
- If not, fetch the blocks into L1
- But, do not load into CPU

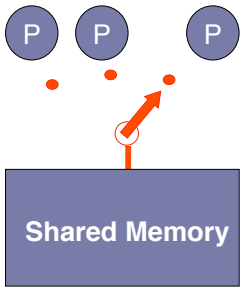
Changing SC's interpretation:

- Load/Store in-program order + atomic
- Peeking without loading/storing a value is ok!

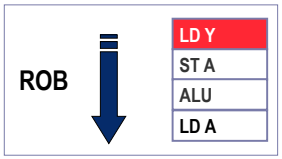


New interpretation of SC

- Memory should appear
  - in program order & atomic
- Peeking ok
  - any order
  - non-atomic/overlapped
- Meaning
  - Look for cache blocks in L1
  - If non-blocking L1, try to fetch all missing blocks



SC + Store Buffer + Lookup/Fetch blocks (into L1)



LD Y	Fetch into L1
ST A	Fetch into L1
LD A	Fetch into L1

LD/ST queue

ST B
ST X

Store buffer

Look up cache blocks but do not fetch value

+ Overlaps LD Y with ST X pending latencies

Pipeline remains completely clogged

In SC + Store Buffer + Fetch blocks

```
ST X    ; removed from pipeline (wait in SB)
ST B    ; removed from pipeline (wait in SB)
LD A    ; looks up cache but waits (no load)
ALU     ; waiting for LD A
ST A    ; looks up cache but waits (no store)
LD Y    ; looks up cache, fetches Y into L1
ST Z     ; instruction not fetched
```

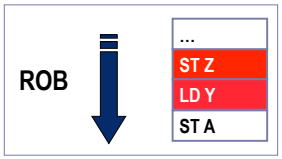

Can we relax the memory order?

- Memory accesses are mostly independent
  - Order is only necessary when entering/exiting a critical section
- Long-latency misses block the pipeline
- But, pipelines often blocked because of stores
- Stores are not needed to advance computation
  - They can be set aside (in a larger store buffer)
  - Let (unrelated) loads bypass stores

Processor Consistency

- PC
  - Was built in systems in 1970's before it was defined
- Relax the loads with respect to stores
  - Let load go if there are only pending stores ahead
- Enforce order upon an atomic instruction
  - E.g., XCHG instruction in x86 or ldstub or swap instructions in SPARC
  - Wait for store buffer to drain before proceeding
- Examples: IBM 370, Sun TSO, & Intel x86
- Relatively simple model to understand (for programmers)

PC



ST Z	Fetch into L1
LD Y	Miss
ST A	Store buffer full

LD/ST queue

ST B
ST X

Store buffer

LD A, ALU (unrelated) retire

LD Y and ST Z overlapped with ST X

- LD Y is allowed to proceed (load)

But, ST A blocks because Store buffer is full

In PC

```
ST X    ; removed from pipeline (wait in SB)
ST B    ; removed from pipeline (wait in SB)
LD A    ; retired
ALU     ; retired
ST A    ; lookup + fetch, but store buffer full
LD Y    ; miss
ST Z    ; looks up cache, fetches Z into L1
```


Performance Comparison

Assume miss takes 100 cycles, others take 1 cycle

		Naiive SC	Optimized SC	PC
ST X	; misses	100	100	100
ST B	; hits in L1	1	1	1
LD A	; hits in L1	1	1	1
ALU		1	1	1
ST A		1	1	1
LD Y	; misses	100	1 (with X)	1 (with X)
ST Z	; misses	100	100	1 (with X)
	Total:	304	205	106

But now, programming is a bit trickier

```
/* initial A = B = 0 */
P1          P2
A = 1;      B = 1
r1 = B;     r2 = A;
```

- What values in r1 and r2 are allowed
  - under SC (Naiive or Optimized)?
  - under PC?

How about this?

```
/* initial A = (volatile) flag  = 0 */
P1                  P2
A = 1;              while (flag == 0);
flag = 1;           r = A;
```

- What values in r are possible
  - under SC?
  - under PC?

How about this one?

```
/* initially all 0 */
P1                  P2
A = 1;              while (flag == 0);
B = 1;              r1 = A;
flag = 1;           r2 = B;
```

- What values of r1 and r2 are possible?
  - under SC?
  - under PC?

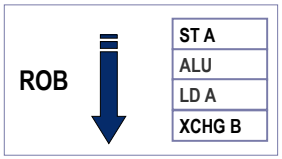
But, PC still hits a wall

- Stores in the Store buffer
  - In program order (i.e., total store order)
  - Can not coalesce
- Two reasons why pipeline may block
  1. Store buffers are small (previous example)
     - ▲Too many stores → pipeline blocks
  2. Too many atomic operations (a new example)
     - ▲Must wait for Store buffer to drain → pipeline blocks

Modified Example: Access to B is a lock!

```
ST X        ; miss in L1 & L2
XCHG B      ; hit
LD A        ; hit
ALU         ;
ST A        ; hit
LD Y        ; miss
ST Z        ; miss
```

PC when executing XCHG



ST A	Fetch into L1
LD A	Fetch into L1

LD/ST queue

ST X
...

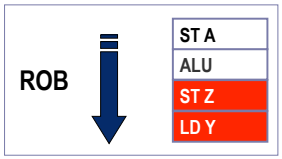
Store buffer

When XCHG reaches head of ROB, pipeline blocks!

LD Y and ST Z can not enter pipeline

- Enter after ST X is drained

PC when ST X drains



ST Z	Fetch into L1
LD Y	Miss

LD/ST queue

ST A
...

Store buffer

When XCHG retires, it's normal PC operation again

LD Y and ST Z overlapped

Can we do better?

- PC is relatively simple
- But may block often
  - Store buffer full
  - Too many lock operations (e.g., XCHG)
- Why not tell the programmer mark the beginning and end of critical sections?
  - Can relax all order within the critical section
  - Reduce pressure on Store buffer

Weak Ordering or Weak Consistency

/* initially all 0 */

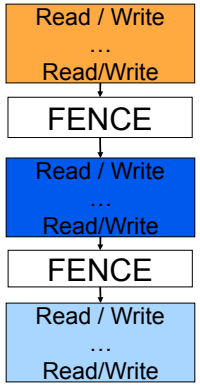
P1	P2
A = 1;	while (FENCE flag == 0);
B = 1;	r1 = A;
FENCE flag = 1;	r2 = B;

A special "FENCE" instruction in the code

No order (but uniprocessor order) is preserved between two FENCE instructions

All order is preserved across a FENCE instruction

Weak Ordering Example

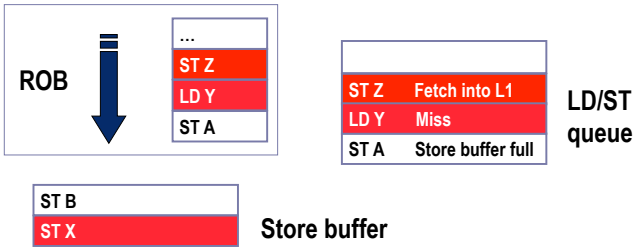


How about this?

```
/* initially all 0 */
P1                  P2
A = 1;              while (flag == 0); /* spin */
B = 1;              r1 = A;
flag = 1;           r2 = B;
```

- What values of r1 and r2 are possible?
  - under PC?
  - under WC?

Reminder: PC blocks with Store buffer full



LD A, ALU (unrelated) retire

LD Y and ST Z overlapped with ST X

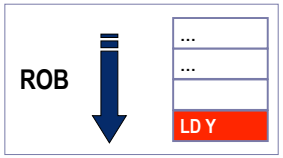
- LD Y is allowed to proceed (load)

But, ST A blocks because Store buffer is full

Reminder: In PC

```
ST X    ; removed from pipeline (wait in SB)
ST B    ; removed from pipeline (wait in SB)
LD A    ; retired
ALU     ; retired
ST A    ; lookup + fetch, but store buffer full
LD Y    ; miss
ST Z    ; looks up cache, fetches Z into L1
```

WC



LD/ST queue

Store buffer

No FENCE instructions? All ordered relaxed

LD Y and ST Z overlapped with ST X

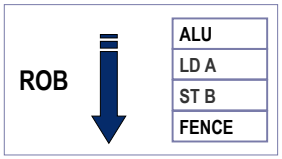
LD A, ALU, ST A all retired

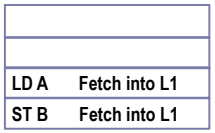
Store buffer not ordered + coalesced

Modified example: A FENCE before ST B

```
ST X     ; miss
FENCE    ;
ST B     ; hit
LD A     ; hit
ALU      ;
ST A     ; hit
LD Y     ; miss
ST Z     ; miss
```


WC with FENCE





LD/ST queue

Store buffer

When FENCE reaches head of ROB, pipeline blocks!

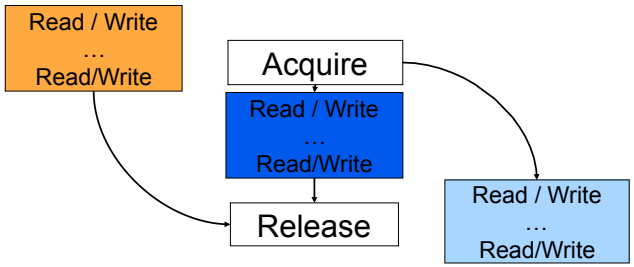
ST A, LD Y and ST Z can not enter pipeline

- Enter after ST X is drained

Release Consistency: Specialized FENCE instructions

- How about having different FENCES at entrance/exit?
  - Lets call an entry FENCE, acquire
  - Lets call an exit FENCE, release
- All loads/stores after a release are allowed to cross
- All loads/stores before an acquire are allowed to cross
- Allows for more overlap
- But, programming is super complicated

Release Consistency Example



Summary

Memory ordering in MP makes a big difference in performance

Programmers want SC

Almost all products support only relaxed models

- Example exception: IBM z990 mainframe
- Most products support PC as default
- Most SW is written assuming PC (e.g., x86, SPARC)
- The exact spec is often missing (x86 defined recently!!!)