CSE 539S, Spring 2018
Concepts in Multicore Computing

Lec 17:Hardware Memory Model

I-Ting Angelina Lee
Mar 27, 2018

Memory Model

Initially, `x = y = 0`.

Processor 0

```
mov 1, x        ;Store
mov y, %ebx     ;Load
```

Processor 1

```
mov 1, y        ;Store
mov x, %eax     ;Load
```

Q. Is it possible that Processor 0's %ebx and Processor 1's %eax both contain the value 0 after the processors have both executed their code?

A. It depends on the ***memory model***: how memory operations behave in the parallel computer system.

Sequential Consistency

> "[T]he 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."
>
> — ***Leslie Lamport [1979]***

- The sequence of instructions as defined by a thread's program are *interleaved* with the corresponding sequences defined by the other threads' programs to produce a global *linear order* of all instructions.
- A load instruction receives the value stored to that address by the most recent store instruction that precedes the load, according to the linear order.

Sequential Consistency Example

Processor 0

```
1  mov 1, x       ;Store
2  mov b, %ebx    ;Load
```

Processor 1

```
3  mov 1, y       ;Store
4  mov a, %eax    ;Load
```

	Interleavings					
	1	1	1	3	3	3
	2	3	3	1	1	4
	3	2	4	2	4	1
	4	4	2	4	2	2
%eax	1	1	1	1	1	0
%ebx	0	1	1	1	1	1

Sequential consistency implies that no execution ends with `%eax = %ebx = 0`.

Memory Models Today

- No modern-day processor implements sequential consistency.
- All implement some form of relaxed consistency.
- Hardware actively reorders instructions.
- Compilers may reorder instructions, too.

Q. Why?

A. Because most of performance is derived from a single thread's unsynchronized execution of code.

Example 1:

Processor 0

```
mov 1, x        ;Store
mov y, %ebx     ;Load
```

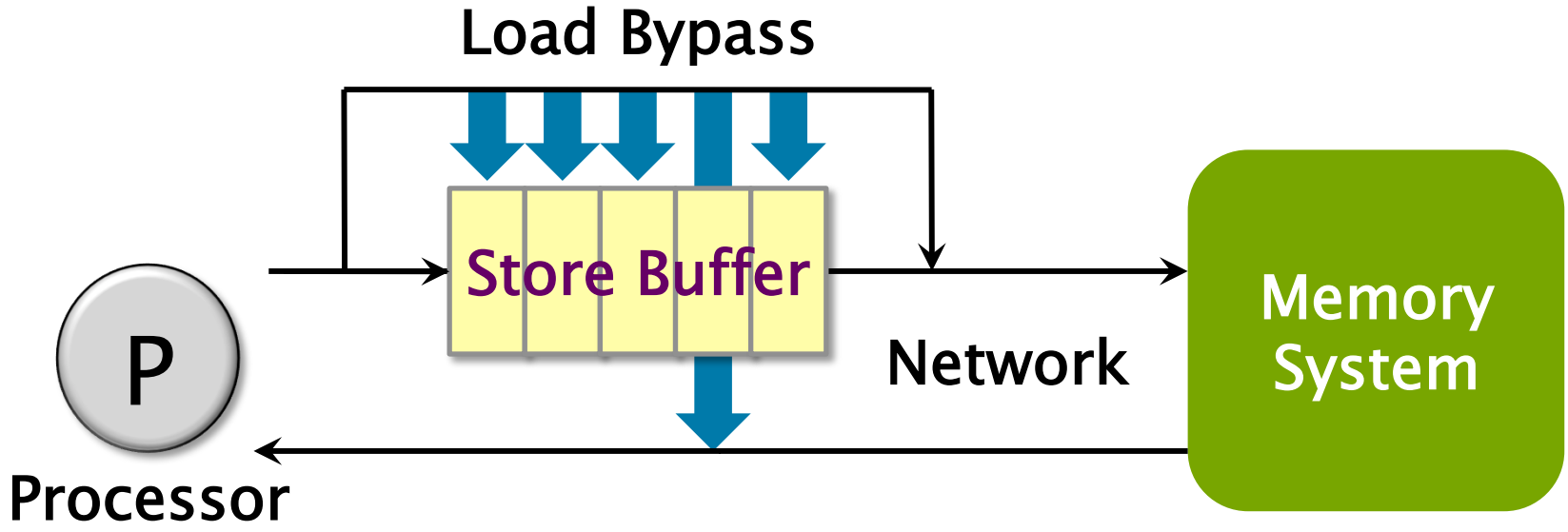
Processor 1

```
mov 1, y        ;Store
mov x, %eax     ;Load
```

Final state: %eax = 0 and %ebx = 0 allowed?

Yes for most existing architectures (AFAIK).

Hardware Reordering



- The processor can issue stores faster than the network can handle them ⇒ store buffer.
- Since a load may stall the processor until it is satisfied, loads take priority, bypassing the store buffer.
- If a load address matches an address in the store buffer, the store buffer returns the result.
- Thus, a load can bypass a store to a different address.

About Memory Model

- Why you should care:
  Reasoning about the correctness of your (or someone else's) concurrent code.

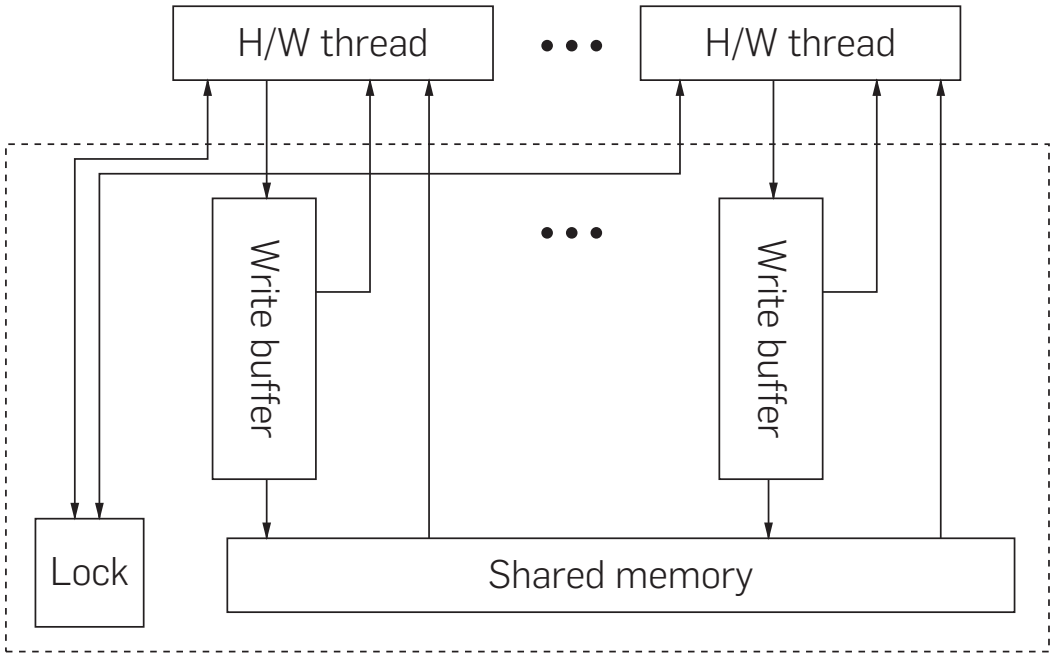
- There are a few different major flavors of hardware memory model:
  - x86-TSO: implemented by major x86 processors.
  - POWER relaxed memory model: implemented by Power and ARM processors prior to ARMv8-A.
  - ARMv8-A: implemented by ARM processors, improving upon POWER

- Today: x86-TSO

Relevant x86 Instructions

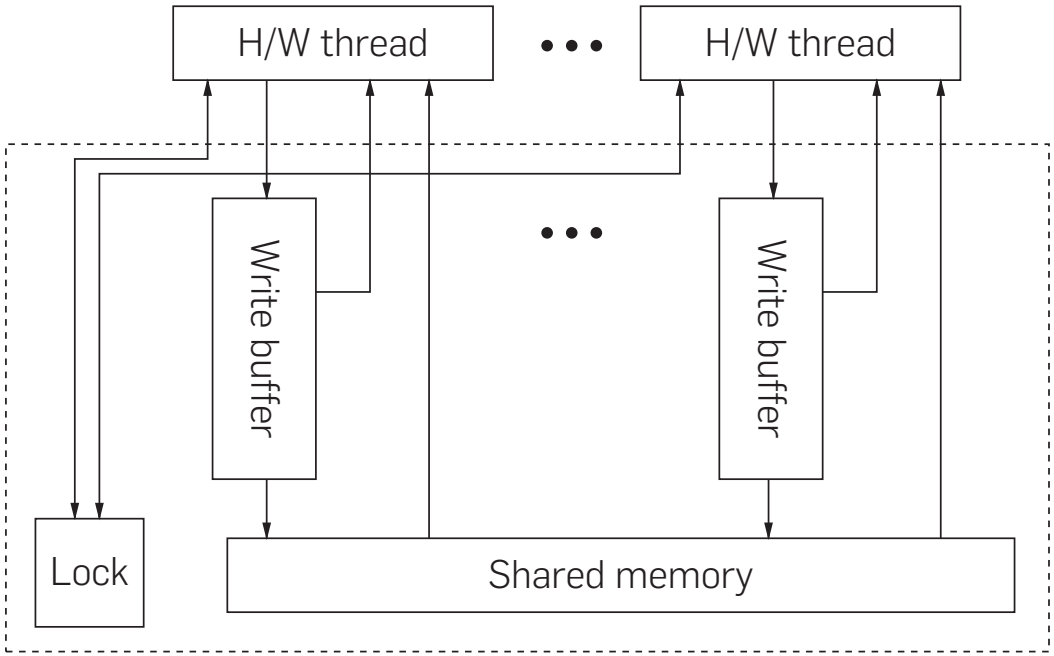
- loads and stores:
  ex: mov 2, x // store 2 into memory x
- fences:
  ex: SFENCE, LFENCE, MFENCE
- locked instructions:
  ex: lock; inc x
  ex: xchg

The Abstraction Machine Model for X86-TSO



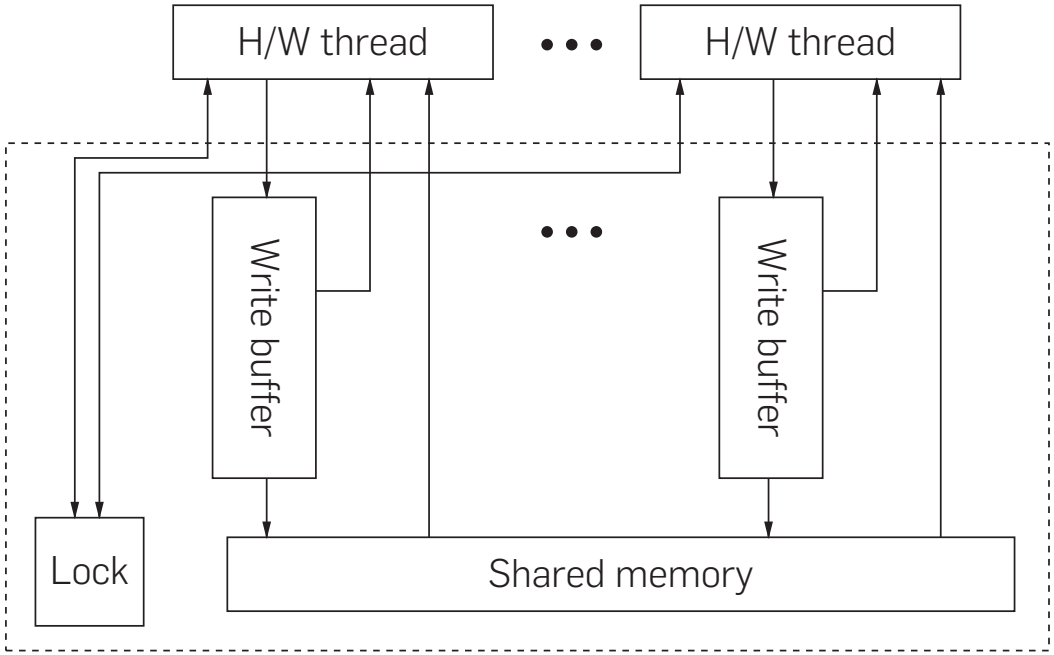
- Instructions are "committed" in order.
- Loads are served from the store buffer is the address match; otherwise served from memory.
- A load is ***committed*** if it's value is fixed.

The Abstraction Machine Model for X86-TSO



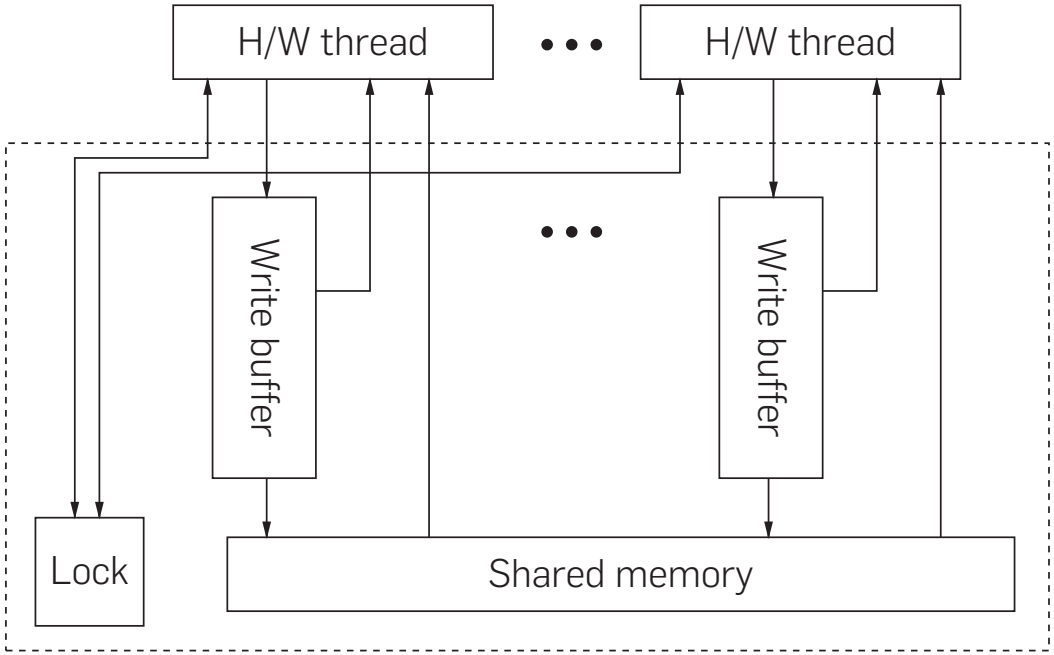
- The store is ***committed*** if it's written to the store buffer.
- The store buffers are FIFO.
- An MFENCE flushes the store buffer of that thread.

The Abstraction Machine Model for X86-TSO



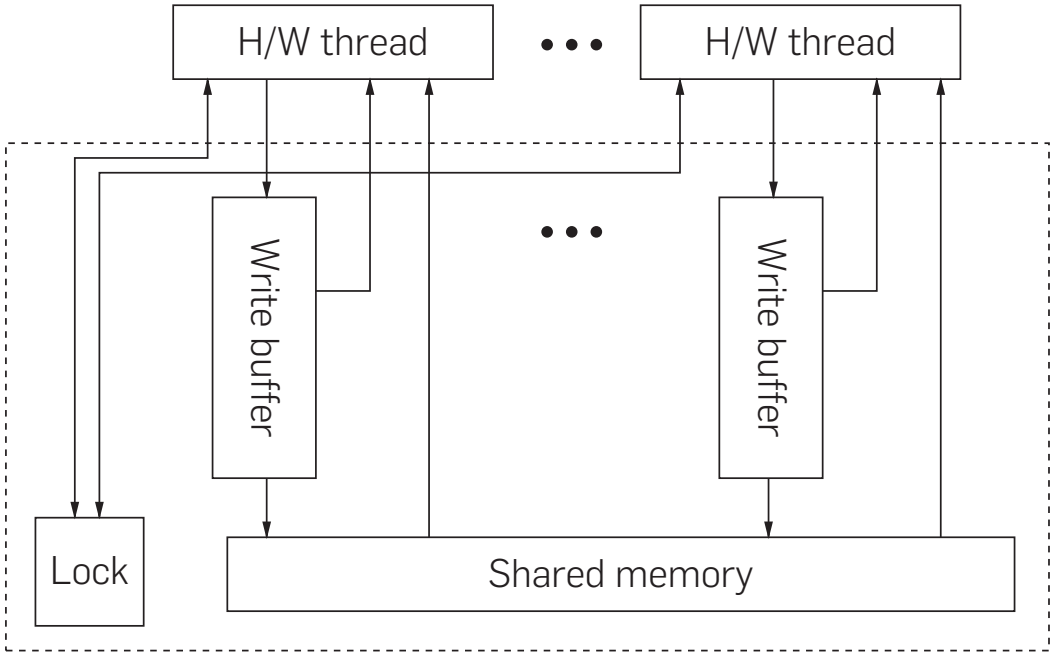
- To execute a LOCK'd instruction, a thread must first obtain the global lock. At the end of the instruction, it flushes its store buffer and relinquishes the lock.
- While the lock is held by one thread, all other threads are ***blocked***.

The Abstraction Machine Model for X86-TSO



- When a thread is blocked, it cannot read (even from its SB), and a buffered write cannot propagate to the shared memory.
- Implication: updates via locked instructions have a global total order imposed

The Abstraction Machine Model for X86-TSO



- The shared memory is ***access-atomic*** (no interleaving bits) and ***multiple-copy atomic*** (once a store hits memory, it's visible to all threads).

X86-TSO from a Thread's Perspective

1. **Loads** are *not* reordered with **loads**.
2. **Stores** are *not* reordered with **stores**.
3. **Stores** are *not* reordered with **prior loads**.
4. A **load** *may* be reordered with a **prior store** to a *different* location but *not* with a prior **store** to the *same* location.
5. **Loads** and **stores** are *not* reordered with **lock** instructions.
6. **Stores** to the same location is **coherent**.
7. **Stores** are transitively visible.
8. **Lock** instructions respect a **global total order**.

Example 1:

Initially, x = y = 0

Processor 0

```
mov 1, x        ;Store
mov b, %ebx     ;Load
```

Processor 1

```
mov 1, y        ;Store
mov a, %eax     ;Load
```

Final state: %eax = 0 and %ebx = 0 allowed?

Yes. On x86-TSO, loads can be reordered with stores.

Example 2:

Initially, x = 0

Processor 0

```
mov 1, x        ;Store
mov x, %eax     ;Load
```

Final state: %eax = 1 required.

Loads are not reordered with older stores to the same location.

Allow loading from store buffer.

Example 3:

Initially, x = y = 0

Processor 0

```
mov 1, x      ;Store
```

Processor 1

```
mov x, %eax   ;Load
mov 1, y      ;Store
```

Processor 2

```
mov y, %ebx   ;Load
mov x, %ecx   ;Load
```

Final state: %eax = 1 and %ebx = 1 and %ecx = 0 allowed?