# Lecture 9: cache coherency

cache memory hierarchy, cache coherency protocol, store-buffer, load-buffer, invalidate-queue, memory barrier, hardware memory model, weak memory model, litmus tests

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l9.pdf

# In previous episodes

Concurrency domain

- Communicating agents (threads)
- Different speed of execution and non-deterministic interleavings (OS scheduler)
- Problems with proper synchronization – race conditions, deadlocks
- Key properties: Safety (correctness), Liveness (progress), Performance

Formalization of concurrent execution

- Timeline, Events, Intervals, Precedence
- Consistency, Linearizability, Linearization points
- Progress conditions: wait-free, lock-free, obstruction-free, starvation-free, deadlock-free
- Atomic register, Snapshot, Consensus number

Practical aspects

- Read-Modify-Write operations
- Memory bus, Cache hierarchy, Cache coherence
- Lock-free algorithms and ABA problem

# Warning: lecture today is going to be like

# Warning: lecture today is going to be like

# Warning: lecture today is going to be like


# Warning: lecture today is going to be like


# Supplementary materials

Unconditional benefit

- "Memory Barriers: a Hardware View for Software Hackers"[1]
- "What Every Programmer Should Know About Memory"[2]
- "Слабые модели памяти: буферизации записи на x86"[3]

Advanced material

- "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models"[4]
- "Shared Memory Consistency Models: A Tutorial"[5]

Mechanical sympathy (better understand h/w design):

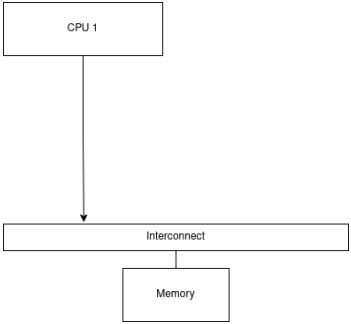
- "Digital Design and Computer Architecture" by David Money Harris, Sarah L. Harris[6]

[1] https://www.researchgate.net/publication/228824849_Memory_Barriers_a_Hardware_View_for_Software_Hackers
[2] https://people.freebsd.org/~lstewart/articles/cpumemory.pdf
[3] https://habr.com/ru/companies/JetBrains-education/articles/523298
[4] https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf
[5] https://dl.acm.org/doi/10.1109/2.546611
[6] https://www.amazon.com/Digital-Design-Computer-Architecture-Harris/dp/0123944244

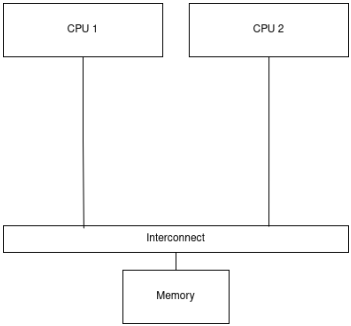
# Lecture plan

1. Preliminary discussion
2. Cache coherency
3. Hardware optimizations
   - Store buffering
   - Load buffering
   - Optional: Invalidate Queues
   - Interconnect topology
4. Hardware memory model
5. Memory barriers
6. Litmus tests
7. Summary

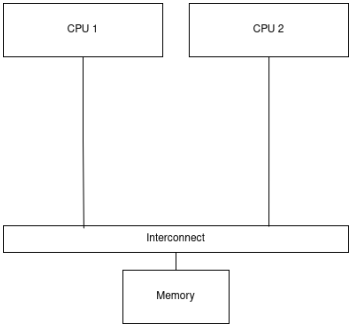
# Global memory: 1 CPU



# Global memory: 2 CPU

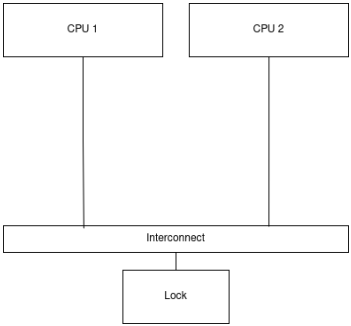


# Global memory: 2 CPU

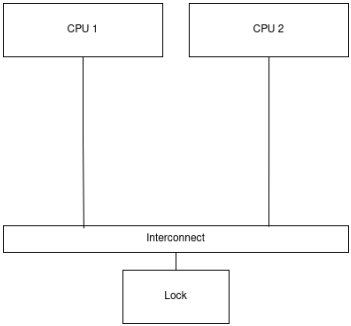



Any problem here?

# Global memory: 2 CPU

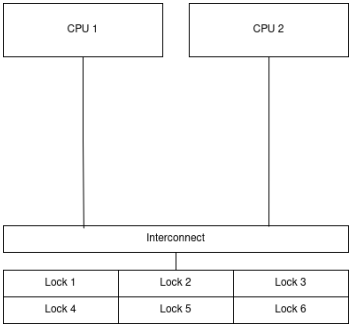


# Global memory: 2 CPU

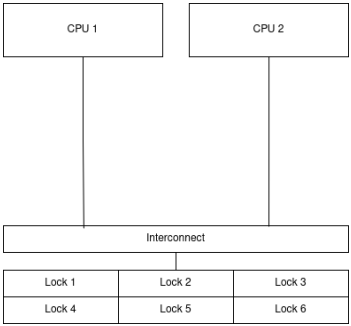



Any problem here?

# Global memory: fine-grained locking



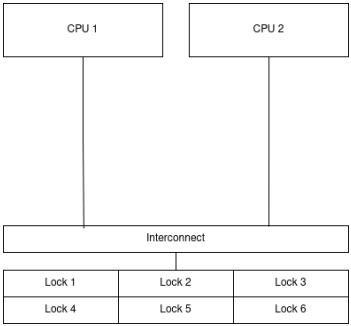
# Global memory: fine-grained locking




Cache line

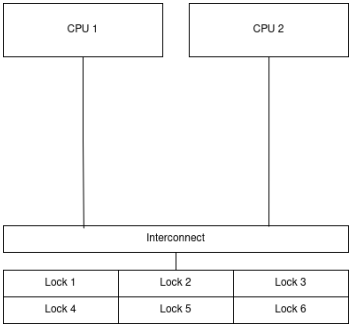
- contiguous chunk of memory (e.g 64, 128, 256 bytes)
- auxiliary data associated with the chunk

# Global memory: fine-grained locking




What is optimal cache line size?

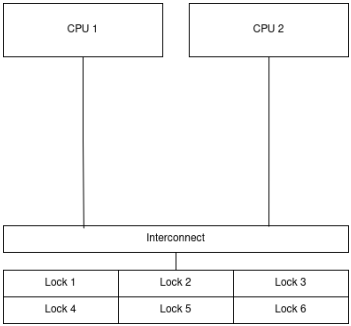
# Global memory: fine-grained locking




What is optimal cache line size?

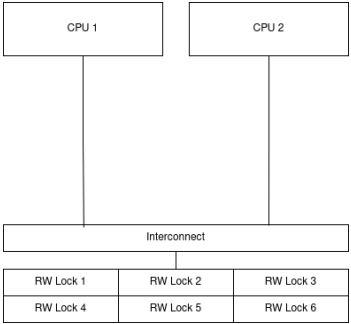
- too large – **false sharing** (access to independent data cause performance loss)
- too small – overhead for storing meta information

# Global memory: fine-grained locking

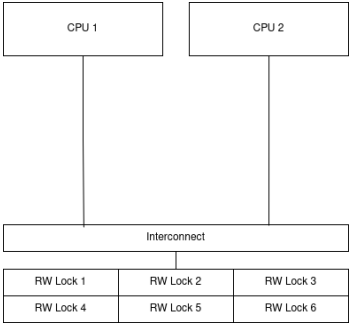



Any problem here?

# Global memory: fine-grained locking for read-mostly data

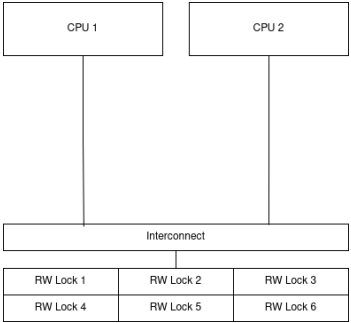


# Global memory: fine-grained locking for read-mostly data



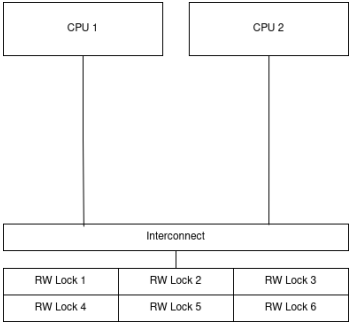

Is it consistent?

# Global memory: fine-grained locking for read-mostly data




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

# Global memory: fine-grained locking for read-mostly data




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

What?

# Coherency: CoRR1 sample

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

# Coherency: CoRR1 sample

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 1`

```
void threadA() {
      x = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = x;
}
```

# Coherency: CoRR1 sample

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 1`

```
void threadA() {
      x = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = x;
}
```

`Forbidden: r1 = 2, r2 = 1`

# Coherency: CoRR1 sample

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 1`

```
void threadA() {
      x = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = x;
}
```

`Forbidden: r1 = 2, r2 = 1`

Several other executions of the same code are allowed:

- both Thread B reads could read 1
- both Thread B reads could read 2
- first could read 1 and the second 2

# Coherency: CoRR2 sample

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

# Coherency: CoRR2 sample

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 0`

| thread1 | thread2 | thread3 | thread4 |
|---|---|---|---|
| `x = 1` | `x = 2` | `r1 = x` | `r3 = x` |
| | | `r2 = x` | `r4 = x` |

# Coherency: CoRR2 sample

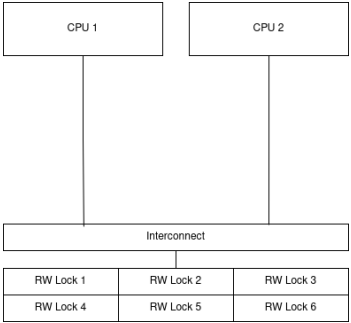
- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 0`

| thread1 | thread2 | thread3 | thread4 |
|---|---|---|---|
| `x = 1` | `x = 2` | `r1 = x` | `r3 = x` |
| | | `r2 = x` | `r4 = x` |

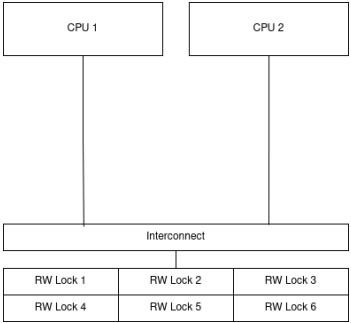
`Forbidden: r1 = 2, r2 = 1, r3 = 1, r4 = 2`

# Global memory: fine-grained locking for read-mostly data




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

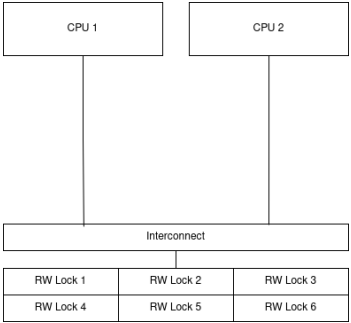
# Global memory: fine-grained locking for read-mostly data




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

What about two locations?

# Global memory: fine-grained locking for read-mostly data

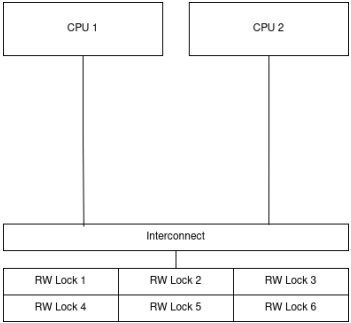



- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

What about two locations?

- No ordering required, no linearizability assumed

# Global memory: fine-grained locking for read-mostly data




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

What about two locations?

- No ordering required, no linearizability assumed

What?

# Coherency is about single memory cell

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

# Coherency is about single memory cell

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 1, y = 1`

```
void threadA() {
      x = 2;
      y = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = y;
}
```

```
void threadC() {
    int r3 = y;
    int r4 = x;
}
```

# Coherency is about single memory cell

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 1, y = 1`

```
void threadA() {
    x = 2;
    y = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = y;
}
```

```
void threadC() {
    int r3 = y;
    int r4 = x;
}
```

Allowed: `r1 = 2, r2 = 1, r3 = 2, r4 = 1`

# Coherency is about single memory cell

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 1, y = 1`

```
void threadA() {
    x = 2;
    y = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = y;
}
```

```
void threadC() {
    int r3 = y;
    int r4 = x;
}
```

`Allowed: r1 = 2, r2 = 1, r3 = 2, r4 = 1`

**Important:**

- "each location has single linear order of all writes" **does not imply** orders of independent locations are "aligned"

# Coherency is about single memory cell

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

Initial state: `x = 1, y = 1`

```
void threadA() {
    x = 2;
    y = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = y;
}
```

```
void threadC() {
    int r3 = y;
    int r4 = x;
}
```

`Allowed: r1 = 2, r2 = 1, r3 = 2, r4 = 1`

**Important:**

- "each location has single linear order of all writes" **does not imply** orders of independent locations are "aligned"
- **Coherence** is weaker than linearizability

# Idea

- **Coherence** is weaker than linearizability

# Idea

- **Coherence** is weaker than linearizability

Let's reorder independent memory operations to improve performance of the CPU!

# Idea

- **Coherence** is weaker than linearizability

Let's reorder independent memory operations to improve performance of the CPU!
Every single-threaded program will become faster!

# Idea

- **Coherence** is weaker than linearizability

Let's reorder independent memory operations to improve performance of the CPU!
Every single-threaded program will become faster!
Who cares that concurrent software will misbehave, right?

# Relaxing memory ordering guarantees

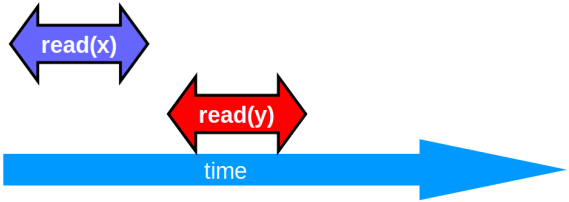
```
static int x, y;
void foo() {
  int r1 = x;
  int r2 = y;
}
```

# Relaxing memory ordering guarantees

```
static int x, y;
void foo() {
  int r1 = memory.read(x);
  int r2 = memory.read(y);
}
```

# Relaxing memory ordering guarantees

```
static int x, y;
void foo() {
  int r1 = memory.read(x);
  int r2 = memory.read(y);
}
```



# Relaxing memory ordering guarantees

```
static int x, y;
void foo() {
  Future<int> r1 = memory.read(x);
  Future<int> r2 = memory.read(y);
}
```

# Relaxing memory ordering guarantees

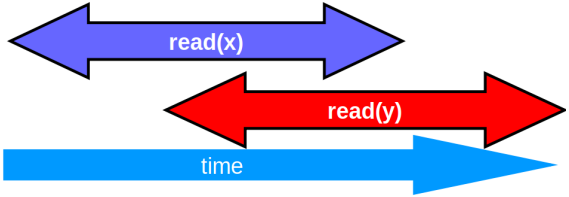
```
static int x, y;
void foo() {
  Future<int> r1 = memory.read(x);
  Future<int> r2 = memory.read(y);
  ...

  use(r2.get());
  ...
  use(r1.get());
}
```

# Relaxing memory ordering guarantees

```
static int x, y;
void foo() {
  Future<int> r1 = memory.read(x);
  Future<int> r2 = memory.read(y);
  ...

  use(r2.get());
  ...
  use(r1.get());
}
```



# Relaxing memory ordering guarantees

```
static int x, y;
void foo() {
  Future<int> r1 = memory.read(x);
  Future<int> r2 = memory.read(y);
  ...

  use(r2.get());
  ...
  use(r1.get());
}
```

- Improve efficiency and CPU utilization – faster
- Sacrifice consistency – harder to reason about correctness

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory
- Maintain consistency of single memory cell – **coherency**

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory
- Maintain consistency of single memory cell – **coherency**
- Allow to reorder operations on independent memory cells – **relaxed memory model**

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory
- Maintain consistency of single memory cell – **coherency**
- Allow to reorder operations on independent memory cells – **relaxed memory model**

Our plan:

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory
- Maintain consistency of single memory cell – **coherency**
- Allow to reorder operations on independent memory cells – **relaxed memory model**

Our plan:

- Look at simplified model: replication pattern for cache lines + MESI protocol

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory
- Maintain consistency of single memory cell – **coherency**
- Allow to reorder operations on independent memory cells – **relaxed memory model**

Our plan:

- Look at simplified model: replication pattern for cache lines + MESI protocol
- Improve it: Store buffering, Load buffering, Invalidate queue, Interconnect topology

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory
- Maintain consistency of single memory cell – **coherency**
- Allow to reorder operations on independent memory cells – **relaxed memory model**

Our plan:

- Look at simplified model: replication pattern for cache lines + MESI protocol
- Improve it: Store buffering, Load buffering, Invalidate queue, Interconnect topology
- Repair multi-cell consistency using memory barriers

# Implementing hardware memory: challenges

- Provide efficient and scalable hardware implementation of shared memory
- Maintain consistency of single memory cell – **coherency**
- Allow to reorder operations on independent memory cells – **relaxed memory model**

Our plan:

- Look at simplified model: replication pattern for cache lines + MESI protocol
- Improve it: Store buffering, Load buffering, Invalidate queue, Interconnect topology
- Repair multi-cell consistency using memory barriers
- Empirically study concurrent behaviour of existing hardware via litmus tests

# Lecture plan

# Why do we need caches?

# Why do we need caches?

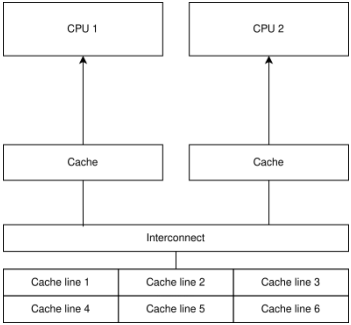
**Nanoseconds and toilet paper show**

# Why do we need caches?

**Nanoseconds and toilet paper show**

https://travisdowns.github.io/blog/2020/07/06/concurrency-costs.html
Level 2 (True Sharing)

# Using caches




Replication pattern:

- Cache line copied to processor-local cache
- Metadata associated with every replicated cache line – state of cache line

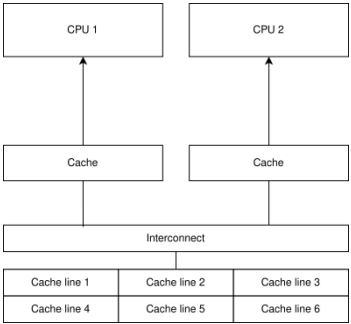
# Homework: caches and associativity

Hardware caches are quite efficient and use hardware-supported N-way associativity
"Is Parallel Programming Hard, And, If So, What Can You Do About It" (perfbook)

- Appendix B "Why Memory Barriers?" , section B.1 "Cache Structure"
- perfbook.B.1

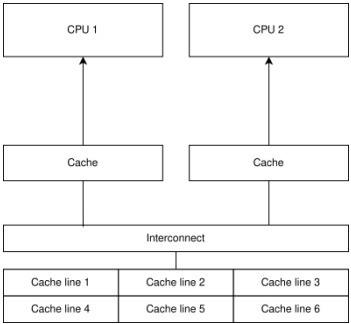
Be ready to draw and explain what is associative hardware cache.

# Distributed caching




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

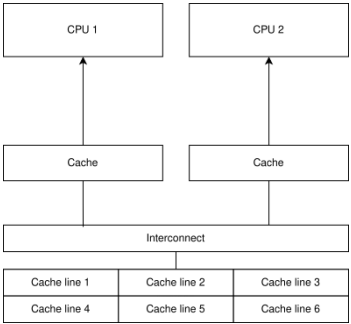
# Distributed caching




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

How to implement that without locks?

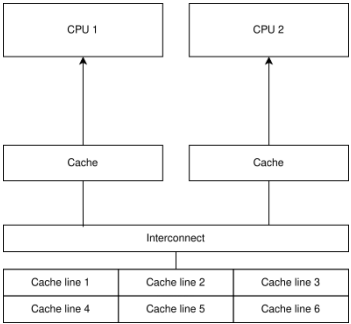
# Distributed caching




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

How to implement that without locks? And without Atomic MRMW registers?

# Distributed caching




- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads

How to implement that without locks? And without Atomic MRMW registers? Efficient?

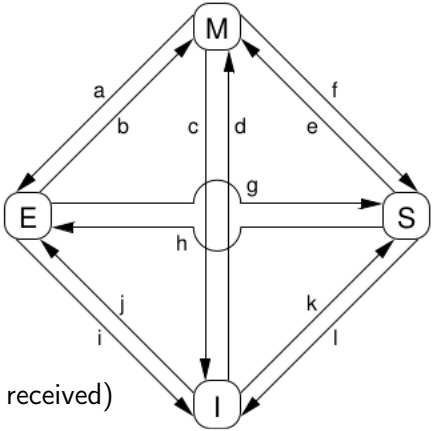
# Cache coherency: MESI

State machine:

- **Modified**
- **Exclusive**
- **Shared**
- **Invalid**

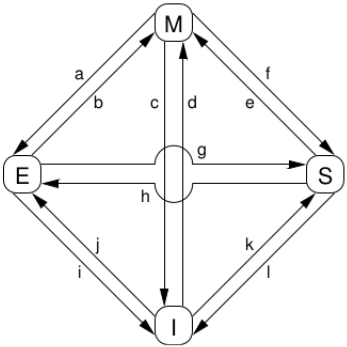
Message passing on some transitions:

- asynchronous (fire and forget)
- synchronous (cannot change state until confirmation received)



# Cache coherency: message passing[7]

- Transition **(k)**: CPU loads data in a cache line that was not in its cache. Send "read", await "read response".
- Transition **(h)**: CPU realizes that it will soon need to write data in this cache line. Send "invalidate" message, await "invalidate acknowledge".
- Transition **(b)**: CPU writes to the cache line that it already had exclusive access to. Nothing to send/receive.
- Transition **(c)**: CPU receives "read invalidate" message for a cache line that it has modified. The CPU must invalidate local copy, respond with "read response" and "invalidate acknowledge".




[7] perfbook B.2.3.

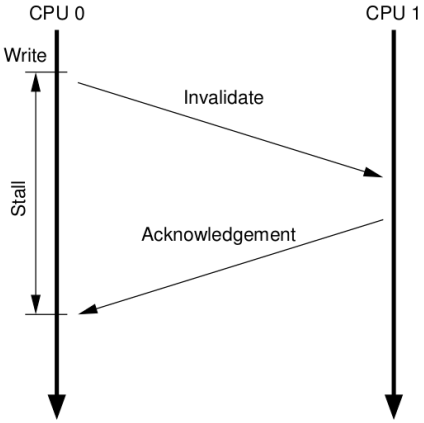
# Homework: more complicated MESI protocol example

Homework: read perfbook.B.2.4. "MESI Protocol Example"

| Sequence # | CPU # | Operation | CPU Cache | | | | Memory | |
|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 | 0 | 8 |
| 0 | | Initial State | -/I | -/I | -/I | -/I | V | V |
| 1 | 0 | Load | 0/S | -/I | -/I | -/I | V | V |
| 2 | 3 | Load | 0/S | -/I | -/I | 0/S | V | V |
| 3 | 0 | Invalidation | 8/S | -/I | -/I | 0/S | V | V |
| 4 | 2 | RMW | 8/S | -/I | 0/E | -/I | V | V |
| 5 | 2 | Store | 8/S | -/I | 0/M | -/I | I | V |
| 6 | 1 | Atomic Inc | 8/S | 0/M | -/I | -/I | I | V |
| 7 | 1 | Writeback | 8/S | 8/S | -/I | -/I | V | V |

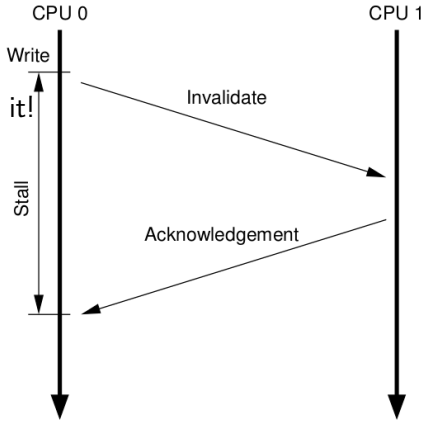
# Unnecessary stalls

- CPU 0 starts write to cache line held by CPU 1
- CPU 0 must wait for the data to arrive



# Unnecessary stalls

- CPU 0 starts write to cache line held by CPU 1
- CPU 0 must wait for the data to arrive
- Why wait? CPU 0 is going to unconditionally overwrite it!



# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)
- Number of states (MSI, MOESI, MESIF ...)

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)
- Number of states (MSI, MOESI, MESIF ...)
- Hierarchy of caches (L1/L2/L3, iCache/dCache)

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)
- Number of states (MSI, MOESI, MESIF ...)
- Hierarchy of caches (L1/L2/L3, iCache/dCache)
- Topology of interconnect: see slide 37

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)
- Number of states (MSI, MOESI, MESIF ...)
- Hierarchy of caches (L1/L2/L3, iCache/dCache)
- Topology of interconnect: see slide 37
- When to send request? To whom? How to process incoming request?

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)
- Number of states (MSI, MOESI, MESIF ...)
- Hierarchy of caches (L1/L2/L3, iCache/dCache)
- Topology of interconnect: see slide 37
- When to send request? To whom? How to process incoming request?

Challenges:

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)
- Number of states (MSI, MOESI, MESIF ...)
- Hierarchy of caches (L1/L2/L3, iCache/dCache)
- Topology of interconnect: see slide 37
- When to send request? To whom? How to process incoming request?

Challenges:

- Invalidation storms

# Cache coherency: takeaways

High-level ideas:

- Memory split into chunks (cache lines) with auxiliary data (cache line state)
- Copies of cache lines could reside in main memory and in processor-specific caches
- Processors communicate via memory bus and use cache coherency protocol
- Read/write operations change state of cache line and trigger message passing

Design choices:

- Cache line size (64, 128, 256 ...)
- Number of states (MSI, MOESI, MESIF ...)
- Hierarchy of caches (L1/L2/L3, iCache/dCache)
- Topology of interconnect: see slide 37
- When to send request? To whom? How to process incoming request?

Challenges:

- Invalidation storms
- Unnecessary stalls

# Lecture plan

# Concurrent invariants and where to violate them

- **Coherence**: in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads
- Step 1: "break" intuitive behaviour to get performance improvement
- Step 2: repair consistency

# Which parts of cache coherence should be optimized?

State machine:

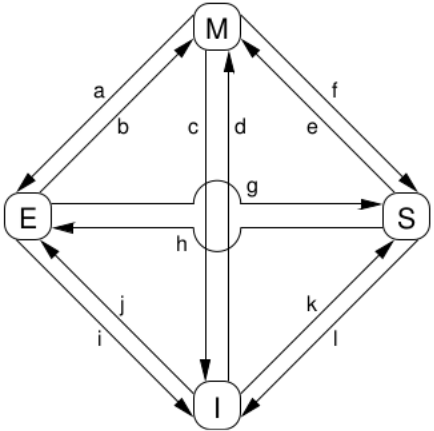
- **Modified**, **Exclusive**, **Shared**, **Invalid**

Message passing on some transitions:

- asynchronous (fire and forget)
- synchronous (await confirmation)

Optimization opportunities:

- Option 1: more states (MESIF, MOESI ...)
- Option 2: more asynchronous operations



# Which parts of cache coherence should be optimized?

State machine:

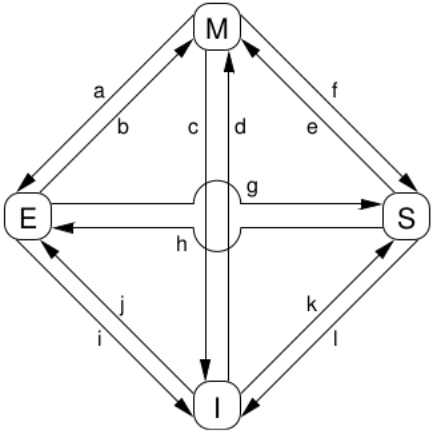
- **Modified**, **Exclusive**, **Shared**, **Invalid**

Message passing on some transitions:

- asynchronous (fire and forget)
- synchronous (await confirmation)

Optimization opportunities:

- Option 1: more states (MESIF, MOESI ...)
- **Option 2: more asynchronous operations**



# Removing unnecessary stalls

CPU execution:

- `write(x = 1)`

# Removing unnecessary stalls

CPU execution:

- `write(x = 1) , cache_line(x) is Shared`

# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is `Shared`
- Send invalidation requests

# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is `Shared`
- Send invalidation requests , Await confirmation

# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`

# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is `Shared`
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`
- `write(y = 1)`

# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`
- `write(y = 1)` , `cache_line(y)` is Shared

# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`
- `write(y = 1)` , `cache_line(y)` is Shared
- Send invalidation requests

# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`
- `write(y = 1)` , `cache_line(y)` is Shared
- Send invalidation requests , Await confirmation

# Removing unnecessary stalls

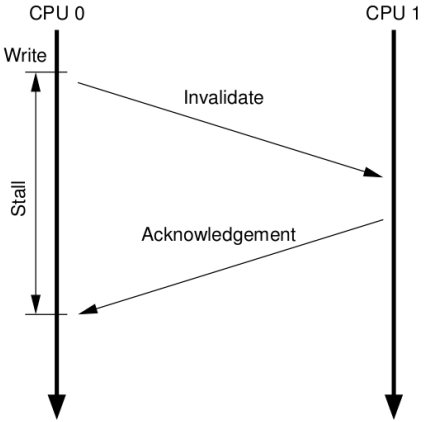
CPU execution:

- `write(x = 1)` , `cache_line(x)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`
- `write(y = 1)` , `cache_line(y)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(y)`

# Removing unnecessary stalls

CPU execution:

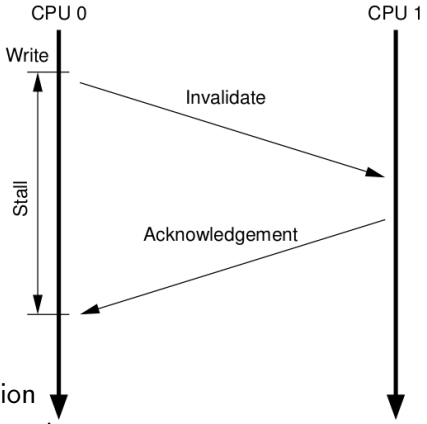
- `write(x = 1)` , `cache_line(x)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`
- `write(y = 1)` , `cache_line(y)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(y)`



# Removing unnecessary stalls

CPU execution:

- `write(x = 1)` , `cache_line(x)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(x)`
- `write(y = 1)` , `cache_line(y)` is Shared
- Send invalidation requests , Await confirmation
- Modify `cache_line(y)`




Idea: batch processing

- Buffer of "pending operations" that await confirmation
- Execute other instructions **before** invalidation confirmed
- Modification of the same location "consults" with buffer, not with cache (**store to load forwarding**)

# Lecture plan

# Store buffering

```
int x, y;
```

```
void threadA() {
        x = 1;
    int a = y;
}
```

```
void threadB() {
        y = 1;
    int b = x;
}
```

# Store buffering

```
int x, y;
```

```
void threadA() {
      x = 1;
  int a = y;
}
```

```
void threadB() {
        y = 1;
    int b = x;
}
```

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

# Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

# Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

What could we see in (EAX EBX)?

(1 1) , (0 1) , (1 0) , (0 0)

# Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

What could we see in (EAX EBX)?

(1 1) , (0 1) , (1 0) , (0 0)

Possible executions:

- A.1 -> A.2 -> B.1 -> B.2
- B.1 -> A.2 -> B.2
- B.2 -> A.2
- B.1 -> A.1 -> A.2 -> B.2
- B.2 -> A.2
- B.2 -> A.1 -> A.2

# Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

What could we see in (EAX EBX)?

(1 1) , (0 1) , (1 0) , (0 0)

Possible executions:

- A.1 -> A.2 -> B.1 -> B.2 : (0, 1)
- B.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.1 -> A.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.2 -> A.1 -> A.2 : (1, 0)

# Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

What could we see in (EAX EBX)?

Linearizable answer: (1 1) , (0 1) , (1 0)

Possible executions:

- A.1 -> A.2 -> B.1 -> B.2 : (0, 1)
- B.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.1 -> A.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.2 -> A.1 -> A.2 : (1, 0)

# Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

What could we see in (EAX EBX)?

Hardware answer: (1 1) , (0 1) , (1 0) , (0 0)

Possible executions:

- A.1 -> A.2 -> B.1 -> B.2 : (0, 1)
- B.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.1 -> A.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.2 -> A.1 -> A.2 : (1, 0)

# Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

What could we see in (EAX EBX)?

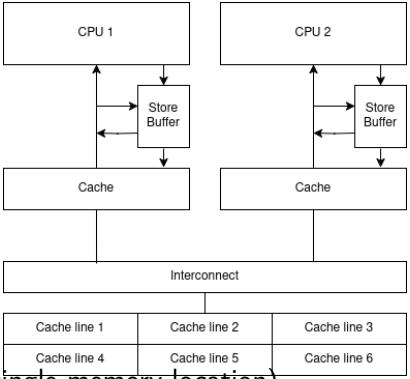
Hardware answer: (1 1) , (0 1) , (1 0) , (0 0) **WHAT?**

Possible executions:

- A.1 -> A.2 -> B.1 -> B.2 : (0, 1)
- B.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.1 -> A.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.2 -> A.1 -> A.2 : (1, 0)

# Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x]  # (B.2)
```

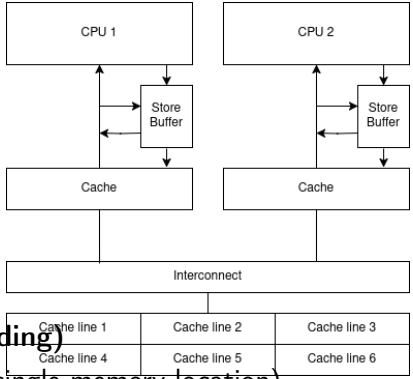



Hardware allows: (EAX=0 EBX=0)

- Looks like CPU reorders memory load and stores
- but does not break per-processor order
- and keeps cache coherency (linear order of writes for single memory location)

# Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x]  # (B.2)
```




Hardware allows: (EAX=0 EBX=0)

- Looks like CPU reorders memory load and stores
- but does not break per-processor order **(store forwarding)**
- and keeps cache coherency (linear order of writes for single memory location)

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute a = 1, a not in cache
- CPU 0: send "read invalidate"

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond
- CPU 0: start execute `b = a + 1`

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond
- CPU 0: start execute `b = a + 1`
- CPU 0: receive cache line from CPU 1 ("a is zero"), put it to cache

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond
- CPU 0: start execute `b = a + 1`
- CPU 0: receive cache line from CPU 1 ("a is zero"), put it to cache
- CPU 0: use a value from cache (`a = 0`)

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond
- CPU 0: start execute `b = a + 1`
- CPU 0: receive cache line from CPU 1 ("a is zero"), put it to cache
- CPU 0: use a value from cache (`a = 0`)
- CPU 0: commit "`write(a = 1)`" from store buffer (now cache thinks "a is one")

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond
- CPU 0: start execute `b = a + 1`
- CPU 0: receive cache line from CPU 1 ("a is zero"), put it to cache
- CPU 0: use a value from cache (`a = 0`)
- CPU 0: commit "`write(a = 1)`" from store buffer (now cache thinks "a is one")
- CPU 0: finish writing to `b` ("'b is one")

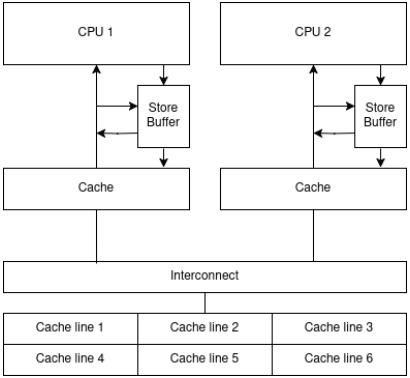
# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond
- CPU 0: start execute `b = a + 1`
- CPU 0: receive cache line from CPU 1 ("a is zero"), put it to cache
- CPU 0: use a value from cache (`a = 0`)
- CPU 0: commit "`write(a = 1)`" from store buffer (now cache thinks "a is one")
- CPU 0: finish writing to `b` ("'b is one")
- CPU 0: assert failed

# Store to load forwarding: why?

```
static int a = 0, int b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```




- CPU 0: start execute `a = 1`, a not in cache
- CPU 0: send "read invalidate"
- CPU 0: put "`write(a = 1)`" to store buffer
- CPU 1: receive "read invalidate", respond
- CPU 0: start execute `b = a + 1`
- CPU 0: receive cache line from CPU 1 ("a is zero"), put
- CPU 0: use a value from cache (`a = 0`)
- CPU 0: commit "`write(a = 1)`" from store buffer (now cache thinks "a is one")
- CPU 0: finish writing to b ("'b is one")
- CPU 0: assert failed

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

Some concurrent algorithms do not tolerate arbitrary memory reorderings (e.g. Peterson Lock).

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

Some concurrent algorithms do not tolerate arbitrary memory reorderings (e.g. Peterson Lock). How to repair them?

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

Some concurrent algorithms do not tolerate arbitrary memory reorderings (e.g. Peterson Lock).
How to repair them?
Use special CPU instruction:

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

Some concurrent algorithms do not tolerate arbitrary memory reorderings (e.g. Peterson Lock).
How to repair them?
Use special CPU instruction:

- "flush store buffer before next operation"

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

Some concurrent algorithms do not tolerate arbitrary memory reorderings (e.g. Peterson Lock).
How to repair them?
Use special CPU instruction:

- "flush store buffer before next operation"
- "ensure no reordering of **this** and **that**"

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

Some concurrent algorithms do not tolerate arbitrary memory reorderings (e.g. Peterson Lock).
How to repair them?
Use special CPU instruction:

- "flush store buffer before next operation"
- "ensure no reordering of **this** and **that**"
- "execute memory barrier instruction"

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Store buffering: useful?

- Processor extended with internal buffer to speed-up execution
- In-processor order conforms to program order (**store forwarding**)
  - Perfect single-CPU abstraction
- Other processor may see memory updates to happen in non-program order
  - Abstraction leaks[8] in multicore environment

Some concurrent algorithms do not tolerate arbitrary memory reorderings (e.g. Peterson Lock).
How to repair them?
Use special CPU instruction:

- "flush store buffer before next operation"
- "ensure no reordering of **this** and **that**"
- "execute memory barrier instruction"

We will discuss few more hardware optimizations before diving into memory barriers.

[8] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

# Lecture plan

# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```

Possible result: (EAX=1, EBX=1)

# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```

Possible result: (EAX=1, EBX=1)

Some algorithms will fail (e.g. Peterson Lock).

# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```

`Possible result: (EAX=1, EBX=1)`

Some algorithms will fail (e.g. Peterson Lock).

How to repair them?

# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```

`Possible result: (EAX=1, EBX=1)`

Some algorithms will fail (e.g. Peterson Lock).

How to repair them?

Use special CPU instruction:

# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```

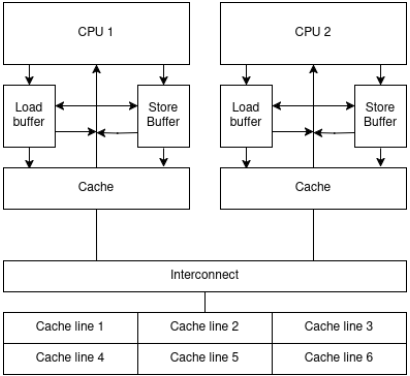
Possible result: (EAX=1, EBX=1)

Some algorithms will fail (e.g. Peterson Lock).

How to repair them?

Use special CPU instruction:

- "flush load buffer before next operation"



# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```

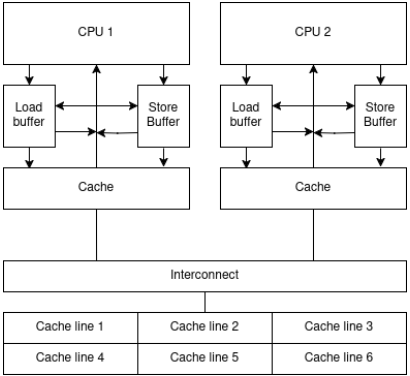
Possible result: (EAX=1, EBX=1)

Some algorithms will fail (e.g. Peterson Lock).

How to repair them?

Use special CPU instruction:

- "flush load buffer before next operation"
- "ensure no reordering of **this** and **that**"



# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```

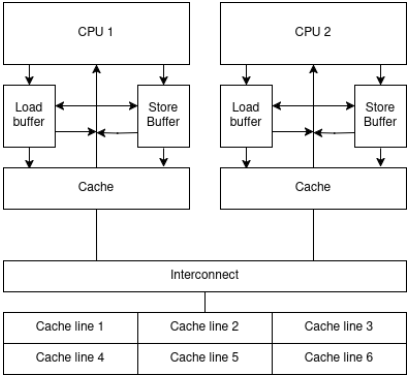
Possible result: (EAX=1, EBX=1)

Some algorithms will fail (e.g. Peterson Lock).

How to repair them?

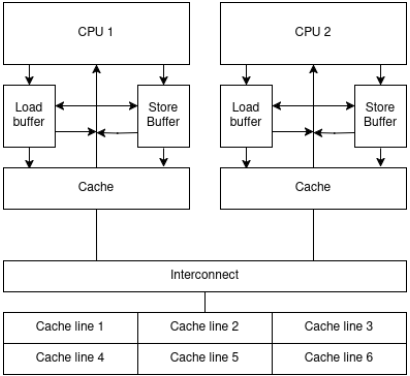
Use special CPU instruction:

- "flush load buffer before next operation"
- "ensure no reordering of **this** and **that**"
  - should we flush load buffer and store buffer simultaneously?



# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```




Possible result: (EAX=1, EBX=1)

Some algorithms will fail (e.g. Peterson Lock).

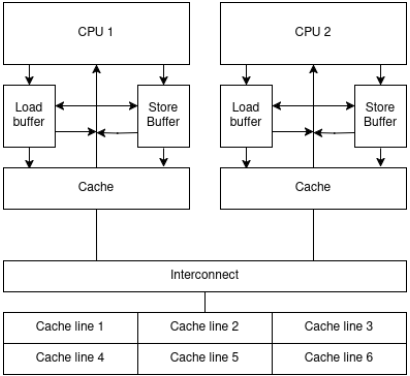
How to repair them?

Use special CPU instruction:

- "flush load buffer before next operation"
- "ensure no reordering of **this** and **that**"
  - should we flush load buffer and store buffer simultaneously?
- "execute memory barrier instruction"

# Load buffering

```
# thread A                    # thread B
mov EAX , [y] # (A.1)         mov EBX, [x] # (B.1)
mov [x] ,  1  # (A.2)         mov [y] , 1  # (B.2)
```




`Possible result: (EAX=1, EBX=1)`

Some algorithms will fail (e.g. Peterson Lock).

How to repair them?

Use special CPU instruction:

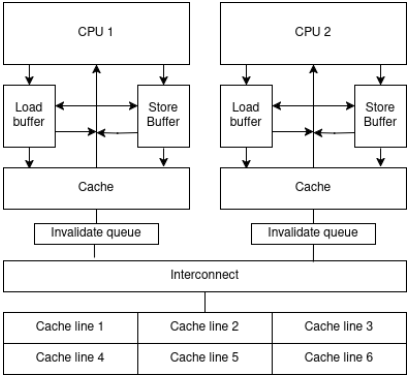
- "flush load buffer before next operation"
- "ensure no reordering of **this** and **that**"
  - should we flush load buffer and store buffer simultaneously?
- "execute memory barrier instruction"

Looks like we need different **kinds** of memory barriers.

# Lecture plan

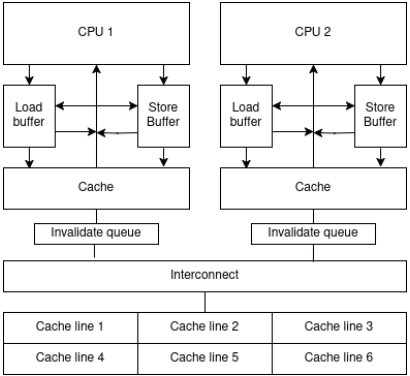
# Invalidate Queue

```
void foo(void) {
  a = 1;
  smp_mb(); // memory barrier
  b = 1;
}
void bar(void) {
  while (b == 0) continue;
  assert(a == 1); // fails with `a == 0`. WHAT?
}
```



# Invalidate Queue

```
void foo(void) {
  a = 1;
  smp_mb(); // memory barrier
  b = 1;
}
void bar(void) {
  while (b == 0) continue;
  assert(a == 1); // fails with `a == 0`. WHAT?
}
```



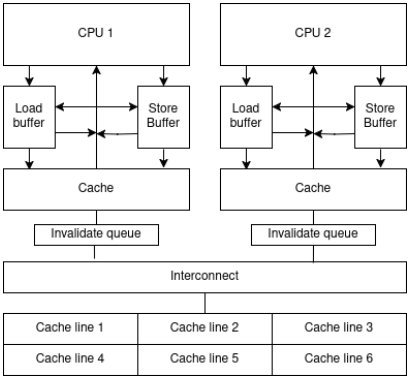

Key insight:

- Synchronization is **always** a **communication**
  - The one to initiate data exchange (writer, request sender)
  - The one to ensure data view is consistent (reader, response sender)
- "My thread executed strongest memory barrier, others will see updated data" is **NOT OK**

# Invalidate Queue

```
void foo(void) {
  a = 1;
  smp_mb(); // memory barrier
  b = 1;
}
void bar(void) {
  while (b == 0) continue;
  smp_mb(); // memory barrier
  assert(a == 1); // never fails
}
```

Optional homework: perfbook.B.4.3



# Lecture plan

# Independent Reads of Independent Writes

```
thread1      thread2      thread3      thread4
  x = 1        y = 1        r1 = x       r3 = y
                            r2 = y       r4 = x
```

Assume there is no in-CPU reordering of memory operations (no load buffering)

# Independent Reads of Independent Writes

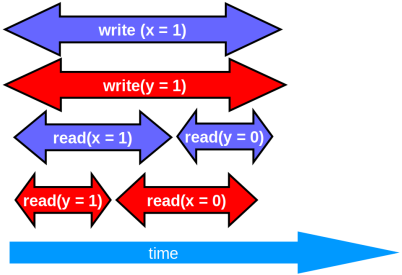
```
thread1      thread2      thread3      thread4
  x = 1        y = 1        r1 = x       r3 = y
                            r2 = y       r4 = x
```

Assume there is no in-CPU reordering of memory operations (no load buffering)
Is it possible to observe `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?

# Independent Reads of Independent Writes

```
thread1      thread2      thread3      thread4
  x = 1        y = 1        r1 = x       r3 = y
                            r2 = y       r4 = x
```

Assume there is no in-CPU reordering of memory operations (no load buffering)
Is it possible to observe `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?



# Independent Reads of Independent Writes

```
thread1      thread2      thread3      thread4
  x = 1        y = 1        r1 = x       r3 = y
                            r2 = y       r4 = x
```

Assume there is no in-CPU reordering of memory operations (no load buffering)
Is it possible to observe `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Hint: it is NOT linearizable

# Independent Reads of Independent Writes

```
thread1      thread2      thread3      thread4
  x = 1        y = 1        r1 = x       r3 = y
                            r2 = y       r4 = x
```

Assume there is no in-CPU reordering of memory operations (no load buffering)
Is it possible to observe `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Hint: it is NOT linearizable

- x86 or x86_64 (TSO): no

# Independent Reads of Independent Writes

```
thread1      thread2      thread3      thread4
  x = 1        y = 1        r1 = x       r3 = y
                            r2 = y       r4 = x
```

Assume there is no in-CPU reordering of memory operations (no load buffering)
Is it possible to observe `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Hint: it is NOT linearizable

- `x86` or `x86_64` (TSO): no
- `ARM` or `POWER`: yes[9]

[9] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, section 6.1

# Independent Reads of Independent Writes

```
thread1      thread2      thread3      thread4
  x = 1        y = 1        r1 = x       r3 = y
                            r2 = y       r4 = x
```

Assume there is no in-CPU reordering of memory operations (no load buffering)
Is it possible to observe `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Hint: it is NOT linearizable

- `x86` or `x86_64` (TSO): no
- `ARM` or `POWER`: yes[9]

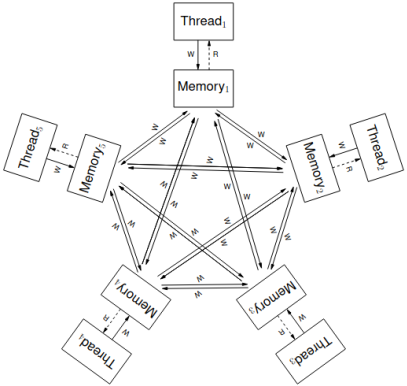
Information about cache lines could "travel" from one CPU to other CPU with different speed.

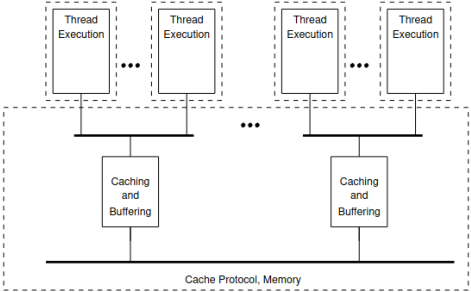
[9] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, section 6.1

# Interconnect topology

Key insight:

- Cache lines could "travel" from one CPU to other CPU with different speed



# Lecture plan

# Hardware-level reordering

In one sentence

# Hardware-level reordering

In one sentence

- Anything could be reordered with everything

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location
  - no ordering guarantees across **different** locations

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location
  - no ordering guarantees across **different** locations

How to treat non-synchronized memory accesses:

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location
  - no ordering guarantees across **different** locations

How to treat non-synchronized memory accesses:

- Happened on some processor – no guarantees it is visible on another processor

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location
  - no ordering guarantees across **different** locations

How to treat non-synchronized memory accesses:

- Happened on some processor – no guarantees it is visible on another processor
- Synchronization is always a communication

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location
  - no ordering guarantees across **different** locations

How to treat non-synchronized memory accesses:

- Happened on some processor – no guarantees it is visible on another processor
- Synchronization is always a communication
  - somebody writes and ensures data will be properly shared (memory barrier)

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location
  - no ordering guarantees across **different** locations

How to treat non-synchronized memory accesses:

- Happened on some processor – no guarantees it is visible on another processor
- Synchronization is always a communication
  - somebody writes and ensures data will be properly shared (memory barrier)
  - somebody reads and ensures data view is consistent (memory barrier)

# Hardware-level reordering

Conservative approximation

- Anything could be reordered with everything (point-of-view of some CPUs may disagree)
- Current CPU will "emulate" execution of single-threaded program "as if" in program order
- Single memory cell is **coherent**
  - linear order of writes for **particular** location
  - no ordering guarantees across **different** locations

How to treat non-synchronized memory accesses:

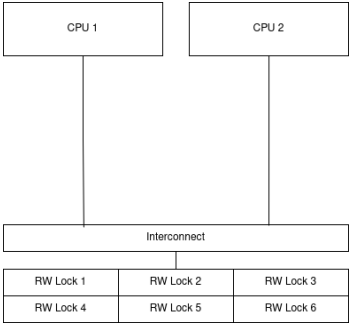
- Happened on some processor – no guarantees it is visible on another processor
- Synchronization is always a communication
  - somebody writes and ensures data will be properly shared (memory barrier)
  - somebody reads and ensures data view is consistent (memory barrier)
- "Today it works on my processor" is **NOT OK**

# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy

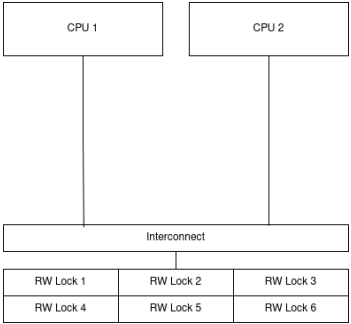
# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy



# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy




- All memory operations are guarded with read-write mutex (thus ordered)
- Read-mostly workloads scale quite well
- Performance is low due to contention and slow queries of data

# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy

- All memory operations are guarded with read-write mutex (thus ordered)
- Read-mostly workloads scale quite well
- Performance is low due to contention and slow queries of data

# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy

- All memory operations are guarded with read-write mutex (thus ordered)
- Read-mostly workloads scale quite well
- Performance is low due to contention and slow queries of data

We started to optimize our model and lost "useful" consistency guarantees

# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy

- All memory operations are guarded with read-write mutex (thus ordered)
- Read-mostly workloads scale quite well
- Performance is low due to contention and slow queries of data

We started to optimize our model and lost "useful" consistency guarantees

- Replication pattern (multiple copies of same data in different caches)

# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy

- All memory operations are guarded with read-write mutex (thus ordered)
- Read-mostly workloads scale quite well
- Performance is low due to contention and slow queries of data

We started to optimize our model and lost "useful" consistency guarantees

- Replication pattern (multiple copies of same data in different caches)
- Reordering of independent operations (store buffering, load buffering)

# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy

- All memory operations are guarded with read-write mutex (thus ordered)
- Read-mostly workloads scale quite well
- Performance is low due to contention and slow queries of data

We started to optimize our model and lost "useful" consistency guarantees

- Replication pattern (multiple copies of same data in different caches)
- Reordering of independent operations (store buffering, load buffering)
- Making message-passing asynchronous (invalidate queues)

# Weak memory model

We started with simplistic model of multiprocessor memory hierarchy

- All memory operations are guarded with read-write mutex (thus ordered)
- Read-mostly workloads scale quite well
- Performance is low due to contention and slow queries of data

We started to optimize our model and lost "useful" consistency guarantees

- Replication pattern (multiple copies of same data in different caches)
- Reordering of independent operations (store buffering, load buffering)
- Making message-passing asynchronous (invalidate queues)
- Non-uniform interconnect (faster/slower message passing)

# Weak memory model

We started to optimize our model and lost "useful" consistency guarantees

- Replication pattern (multiple copies of same data in different caches)
- Reordering of independent operations (store buffering, load buffering)
- Making message-passing asynchronous (invalidate queues)
- Non-uniform interconnect (faster/slower message passing)

# Weak memory model

We started to optimize our model and lost "useful" consistency guarantees

- Replication pattern (multiple copies of same data in different caches)
- Reordering of independent operations (store buffering, load buffering)
- Making message-passing asynchronous (invalidate queues)
- Non-uniform interconnect (faster/slower message passing)

We have less strict rules on what is guaranteed for memory operations

- **Relaxed** memory model
- **Weak** memory model

# Weak memory model

We started to optimize our model and lost "useful" consistency guarantees

- Replication pattern (multiple copies of same data in different caches)
- Reordering of independent operations (store buffering, load buffering)
- Making message-passing asynchronous (invalidate queues)
- Non-uniform interconnect (faster/slower message passing)

We have less strict rules on what is guaranteed for memory operations

- **Relaxed** memory model
- **Weak** memory model
- Interesting fact №1: stronger CPU memory model – less bugs you encounter during maintenance of large concurrent software system
- Interesting fact №2: smartphones, laptops, energy-efficient servers tend to use weak hardware

# Lecture plan

# Memory barriers: definition

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations

# Memory barriers: definition

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor

# Memory barriers: definition

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- But actually we need to prevent reordering of memory operations **as they seen by other processors**

# Memory barriers: definition

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- But actually we need to prevent reordering of memory operations **as they seen by other processors**

Do not forget to insert memory barriers on both sides of communication protocol!

# Memory barriers: definition

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- But actually we need to prevent reordering of memory operations **as they seen by other processors**

Do not forget to insert memory barriers on both sides of communication protocol!
Semantics is very architecture-specific:

- `x86_64` – `mfence`, `lock` prefix
- `arm64` – `dmb`
- `power` – `sync`, `lwsync`

# Memory barriers: definition

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- But actually we need to prevent reordering of memory operations **as they seen by other processors**

Do not forget to insert memory barriers on both sides of communication protocol!
Semantics is very architecture-specific:

- `x86_64` – `mfence`, `lock` prefix
- `arm64` – `dmb`
- `power` – `sync`, `lwsync`

Described in many pages of architecture manual.

# Memory barriers: definition

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- But actually we need to prevent reordering of memory operations **as they seen by other processors**

Do not forget to insert memory barriers on both sides of communication protocol!
Semantics is very architecture-specific:

- `x86_64` – `mfence`, `lock` prefix
- `arm64` – `dmb`
- `power` – `sync`, `lwsync`

Described in many pages of architecture manual. Inconvenient.

# Memory barriers: taxonomy

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- Should be used on both sides of communication protocol

# Memory barriers: taxonomy

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- Should be used on both sides of communication protocol

Simplified taxonomy of memory barriers:

- `Store_Store`, `Store_Load`, `Load_Store`, `Load_Load`

# Memory barriers: taxonomy

**Memory barrier**

- Hardware-specific machine instruction that helps to enforce some kind of ordering between memory operations
- Affects **current** processor
- Should be used on both sides of communication protocol

Simplified taxonomy of memory barriers:

- `Store_Store`, `Store_Load`, `Load_Store`, `Load_Load`

```
int x = static.data1;
Store_Store();
Store_Load();
int y = static.data2;
static.data3 = 17;
```

```
int x = static.data1;
Load_Load();

int y = static.data2;
static.data3 = 17;
```

# Memory barriers: rule of thumb

# Memory barriers: rule of thumb

Do not use them!

# Memory barriers: rule of thumb

Do not use them! Seriously!

# Memory barriers: rule of thumb

Do not use them! Seriously!
Every modern language supports civilized concurrency and provides plenty of useful tools:

# Memory barriers: rule of thumb

Do not use them! Seriously!

Every modern language supports civilized concurrency and provides plenty of useful tools:

- design-level abstractions – `Executor`, `Future`, `ParallelStream` ...

# Memory barriers: rule of thumb

Do not use them! Seriously!

Every modern language supports civilized concurrency and provides plenty of useful tools:

- design-level abstractions – `Executor`, `Future`, `ParallelStream` ...
- concurrent data structures – `Lock`, `Semaphore`, `CountDownLatch`, `Monitor` ...

# Memory barriers: rule of thumb

Do not use them! Seriously!
Every modern language supports civilized concurrency and provides plenty of useful tools:

- design-level abstractions – `Executor`, `Future`, `ParallelStream` ...
- concurrent data structures – `Lock`, `Semaphore`, `CountDownLatch`, `Monitor` ...
- atomic read-modify-write operations - `getAndSet`, `compareAndExchange`, `getAndAdd` ...

# Memory barriers: rule of thumb

Do not use them! Seriously!

Every modern language supports civilized concurrency and provides plenty of useful tools:

- design-level abstractions – `Executor`, `Future`, `ParallelStream` ...
- concurrent data structures – `Lock`, `Semaphore`, `CountDownLatch`, `Monitor` ...
- atomic read-modify-write operations - `getAndSet`, `compareAndExchange`, `getAndAdd` ...

Memory barriers are needed to:

- implement basics (writing your own OS, compiler, VM)

# Memory barriers: rule of thumb

Do not use them! Seriously!

Every modern language supports civilized concurrency and provides plenty of useful tools:

- design-level abstractions – `Executor`, `Future`, `ParallelStream` ...
- concurrent data structures – `Lock`, `Semaphore`, `CountDownLatch`, `Monitor` ...
- atomic read-modify-write operations - `getAndSet`, `compareAndExchange`, `getAndAdd` ...

Memory barriers are needed to:

- implement basics (writing your own OS, compiler, VM)
- get CRITICAL performance

# Memory barriers: rule of thumb

Do not use them! Seriously!

Every modern language supports civilized concurrency and provides plenty of useful tools:

- design-level abstractions – `Executor`, `Future`, `ParallelStream` ...
- concurrent data structures – `Lock`, `Semaphore`, `CountDownLatch`, `Monitor` ...
- atomic read-modify-write operations - `getAndSet`, `compareAndExchange`, `getAndAdd` ...

Memory barriers are needed to:

- implement basics (writing your own OS, compiler, VM)
- get CRITICAL performance and spend a lot of resources to maintain the code

# Memory barriers: rule of thumb

Do not use them! Seriously!

Every modern language supports civilized concurrency and provides plenty of useful tools:

- design-level abstractions – `Executor`, `Future`, `ParallelStream` ...
- concurrent data structures – `Lock`, `Semaphore`, `CountDownLatch`, `Monitor` ...
- atomic read-modify-write operations - `getAndSet`, `compareAndExchange`, `getAndAdd` ...

Memory barriers are needed to:

- implement basics (writing your own OS, compiler, VM)
- get CRITICAL performance and spend a lot of resources to maintain the code
- design useful concurrent programming language (see you at next Lecture)

# Lecture plan

# Litmus test: definition

- **Litmus test**: very small concurrent program that access few shared variables and illustrates some relaxed-memory phenomena

# Litmus test: definition

- **Litmus test**: very small concurrent program that access few shared variables and illustrates some relaxed-memory phenomena

**Coherence (CoRR1)**

Initial state: `x = 1`, Forbidden: `r1 = 2, r2 = 1`

```
void threadA() {
      x = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = x;
}
```

# Litmus test: definition

- **Litmus test**: very small concurrent program that access few shared variables and illustrates some relaxed-memory phenomena

**Coherence (CoRR1)**

Initial state: `x = 1`, Forbidden: `r1 = 2, r2 = 1`

```
void threadA() {
      x = 2;
}
```

```
void threadB() {
    int r1 = x;
    int r2 = x;
}
```

**Store buffering**

```
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

Hardware allows: (EAX=0 EBX=0)

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

- Help to get "relaxed ordering" right

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

- Help to get "relaxed ordering" right
- May be empirically checked against (new version of) hardware

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

- Help to get "relaxed ordering" right
- May be empirically checked against (new version of) hardware
- Minimize "problematic surface" of concurrency library

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

- Help to get "relaxed ordering" right
- May be empirically checked against (new version of) hardware
- Minimize "problematic surface" of concurrency library
- Could be engineered as cross-platform functions (with different implementations)

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

- Help to get "relaxed ordering" right
- May be empirically checked against (new version of) hardware
- Minimize "problematic surface" of concurrency library
- Could be engineered as cross-platform functions (with different implementations)
- Used to illustrate weak memory model related bug in complicated protocol[10,11]

[10] ... this one bite us hard and scare the %$^ out of us https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64

[11] Fix https://github.com/torvalds/linux/commit/76835b0ebf8a7fe85beb03c75121419a7dec52f0

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

- Help to get "relaxed ordering" right
- May be empirically checked against (new version of) hardware
- Minimize "problematic surface" of concurrency library
- Could be engineered as cross-platform functions (with different implementations)
- Used to illustrate weak memory model related bug in complicated protocol[10,11]
- Could be used to provide non-portable yet highly efficient algorithms[12]

[10] ... this one bite us hard and scare the %$^ out of us https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64

[11] Fix https://github.com/torvalds/linux/commit/76835b0ebf8a7fe85beb03c75121419a7dec52f0

[12] Section 4 "Linux x86 SpinLock implementation" in https://dl.acm.org/doi/pdf/10.1145/1785414.1785443

# Litmus test

Litmus tests are basic building blocks to construct reliable concurrent primitives

- Help to get "relaxed ordering" right
- May be empirically checked against (new version of) hardware
- Minimize "problematic surface" of concurrency library
- Could be engineered as cross-platform functions (with different implementations)
- Used to illustrate weak memory model related bug in complicated protocol[10,11]
- Could be used to provide non-portable yet highly efficient algorithms[12]
- Well-studied problem domain

[10] ... this one bite us hard and scare the %$^ out of us https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64

[11] Fix https://github.com/torvalds/linux/commit/76835b0ebf8a7fe85beb03c75121419a7dec52f0

[12] Section 4 "Linux x86 SpinLock implementation" in https://dl.acm.org/doi/pdf/10.1145/1785414.1785443

# Summary

Cache coherency

- Cache line: granularity, false sharing, tagging with extra info
- Cache coherency protocol: states, transitions, message passing, coherency

Hardware optimizations

- Store buffering, Load buffering, Invalidate Queues, Interconnect topology

Hardware memory model

- Reorderings of memory operations on independent memory locations
- Weak (relaxed) consistency

Memory barriers

- Architecture-specific, Affect local CPU only, Different kinds
- `{Store,Load}_{Store,Load}` taxonomy

Litmus tests

- Basic blocks for many key concurrent algorithms
- Empirical approach to building reliable concurrent software

# Summary: homework

"Is Parallel Programming Hard, And, If So, What Can You Do About It" (perfbook) Appendix B "Why Memory Barriers?"

- section B.1 "Cache Structure". Be ready to draw and explain what is associative hardware cache.
- section B.2.4 "MESI Protocol Example"