Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Addresses of word:
- \( t \) bits
- \( s \) bits
- \( b \) bits

- Tag
- Set index
- Block offset

Data begins at this offset

- \( s = 2^t \) sets
- \( E = 2^e \) lines per set
- \( E = \) associativity, \( E = 1 \): direct mapped

\( B = 2^b \) bytes per cache block (the data)
What about writes?

- What to do on a write-hit?
  - **Write-through**: write immediately to memory
  - **Write-back**: defer write to memory until replacement of line

- What to do on a write-miss?
  - **Write-allocate**: load into cache, update line in cache
  - **No-write-allocate**: writes immediately to memory

### Write-back/write-allocate (Most common)  Write-through/no-write-allocate

```
mem  mem

1: load  1: update
CPU  CPU
update  update

2: update

$  $  $
```

**Write-hit**  **Write-miss**

**Write-hit**  **Write-miss**

---

Cache Architectures

- **Multi level caches** (L1, L2, ..., common today)
- **Multi-port vs. single port**
- **Shared vs. private** (in multicore computers, see next slide)
- **Inclusive vs. exclusive** (usually inclusive = content of smaller caches is in larger caches in the hierarchy)
- **Write back vs. write through** (usually write-back)
- **Victim cache to reduce conflict misses**
- **...**
Caches in a Multicore Computer (Example)

Many Variants Possible

Problem? Coherence!
Today: Cache Coherence

- Motivation
- Terminology, snooped-based vs. directory-based
- Snooped based: MESI protocol
- MOESI
- Directory-based coherence
Write-Back Cache

1. CPU₀ reads X from memory
   ▪ loads X=0 into its cache
2. CPU₁ reads X from memory
   ▪ loads X=0 into its cache
3. CPU₀ writes X=1
   ▪ stores X=1 in its cache
4. CPU₁ writes X=2
   ▪ stores X=2 in its cache
5. CPU₁ writes back cache line
   ▪ stores X=2 in memory
6. CPU₀ writes back cache line
   ▪ stores X=1 in cache
   ▪ Later (!) store X=2 from CPU₁ lost

Requires write serialization!

Write-Through Cache

1. CPU₀ reads X from memory
   ▪ loads X=0 into its cache
2. CPU₁ reads X from memory
   ▪ loads X=0 into its cache
3. CPU₀ writes X=1
   ▪ stores X=1 in its cache
   ▪ stores X=1 in memory
4. CPU₁ reads X from its cache
   ▪ loads X=0 from its cache
   ▪ Incoherent value for X on CPU₁

Requires write propagation!
Another Example

- The issue in the previous examples was shared access including writes
- Assume C99:

```
struct twoint {
    int a;
    int b;
}
```

- Two threads:
  - Thread 0: writes to a
  - Thread 1: writes to b
  - The timeline is not specified
- Assume non-coherent write-back cache
  - What may end up in main memory?

- False sharing: two threads access different data in the same cache line

---

Cache Coherence

- Basic problem: threads on different cores with private caches access the same data, including with writes
- Cache coherence requirements
  - A memory system is coherent if it guarantees the following:
    - Write propagation: updates are eventually visible to all readers
    - Write serialization: writes to the same location must be observed in order

  *Everything else: memory model issues (later)*
Cache Coherence Protocol

- Programmer cannot deal with unpredictable behavior!
- Hardware mechanism to ensure coherence: Cache controller executes protocol to maintain coherence

Fundamental Mechanisms
- Snooping
  - Shared bus or (broadcast) network
- Directory-based
  - Record (in a memory) information necessary to maintain coherence
  - *E.g., owner and state of a line etc.*

Cache Coherence Approaches

- Snooping
  - Shared bus or (broadcast) network
  - Cache controller snoops all transactions
  - Monitors and changes state data in cache
  - Works at small scale, challenging at large-scale
  - Example: Intel Xeon etc.

- Directory-based
  - Record information necessary to maintain coherence
  - *E.g., owner and state of cache lines*
  - Central or distributed directory
  - Scalable but more complex/expensive
  - Example: Intel Xeon Phi

Source: Intel
Cache Coherence Parameters

- **Concerns/Goals**
  - Performance
  - Implementation cost (chip space)
  - Correctness
    - (Memory model side effects)

- **Issues**
  - Detection (when does a controller need to act)
  - Enforcement (how does a controller guarantee coherence)
  - Granularity of block sharing (typically cache block size)

Cache Coherence

- **Basic problem:** threads on different cores with private caches access the same data, including with writes

- **Cache coherence requirements**
  A memory system is coherent if it guarantees the following:
  - *Write propagation:* updates are eventually visible to all readers
  - *Write serialization:* writes to the same location must be observed in order
    - *Everything else: memory model issues (later)*
An Engineering Approach: Empirical start

- **Problem 1: stale reads**
  - Cache 1 holds value that was already modified in cache 2
  - **Solution:**
    - *Disallow this state*
    - *Invalidate all remote copies before allowing a write to complete*

- **Problem 2: lost update**
  - Incorrect write back of modified line writes main memory in different order from the order of the write operations or overwrites neighboring data
  - **Solution:**
    - *Disallow more than one modified copy*

Invalidation vs. update

- **Invalidation-based:**
  - Write to a shared line has to invalidate copies in other caches
  - Subsequent writes by the same thread to the same cache line are efficient

- **Update-based:**
  - Local write updates copies in other caches
  - All sharers continue to hit cache line after one core writes
  - Supports producer-consumer pattern well
  - Many writes cause many updates

- Hybrid forms are possible!
- MESI (next): Invalidation-based
MESI Cache Coherence

- Most common hardware implementation of coherence aka. “Illinois protocol”

Each cache line has one of the following states (added as bits to line):

- **Modified (M)**
  - Local copy has been modified, no copies in other caches
  - Memory is stale

- **Exclusive (E)**
  - No copies in other caches
  - Memory is up to date

- **Shared (S)**
  - Unmodified copies *may* exist in other caches
  - Memory is up to date

- **Invalid (I)**
  - Line is not in cache

- Draw 4 cores with private caches, memory, bus
Terminology

- **Clean line:**
  - Content of cache line and main memory is identical (memory is up to date)
  - Can be evicted without write-back

- **Dirty line:**
  - Content of cache line and main memory differ (memory is stale)
  - Needs to be written back eventually
  
  *Time depends on protocol details*

- **Bus transaction:**
  - A signal on the bus that can be observed by all caches
  - Usually blocking (only one signal at a time)

- **Local (private) read/write:**
  - A load/store operation originating at a core connected to the cache

Transitions in Response to Local Reads

- **State is M**
  - No bus transaction, same state

- **State is E**
  - No bus transaction, same state

- **State is S**
  - No bus transaction, same state

- **State is I**
  - Generate bus read (BusRd)
    
    *May force other cache operations (see later)*
  - Other cache(s) signal “shared” if they hold a copy
  - If “shared” was signaled, go to state S
  - Otherwise, go to state E
Transitions in Response to Local Writes

- **State is M**
  - No bus transaction

- **State is E**
  - No bus transaction
  - Go to state M

- **State is S**
  - Line already local & clean
  - There may be other copies
  - Generate bus read for upgrade to exclusive ownership (BusRdX*)
    - *May force other cache operations (see later)*
  - Go to state M

- **State is I**
  - Generate bus read for exclusive ownership (BusRdX)
    - *May force other cache operations (see later)*
  - Go to state M

Transitions in Response to Snooped BusRd

- **State is M**
  - Write cache line back to main memory
  - Signal “shared”
  - Go to state S

- **State is E**
  - Signal “shared”
  - Go to state S

- **State is S**
  - Signal “shared”

- **State is I**
  - Ignore
Transitions in Response to Snooped BusRdX

- **State is M**
  - Write cache line back to memory
  - Go to I (discard line)
- **State is E**
  - Go to I (discard line)
- **State is S**
  - Go to I (discard line)
- **State is I**
  - Ignore

- BusRdX* is handled like BusRdX!

 MESI State Diagram (FSM)

Additional detail: **FlushOpt** = processor (may) send its copy of cache line on bus for possible faster read by other processor. On writebacks the cacheline is always on the bus for possible reading.
## Small Exercise

- Initially: all in I state

<table>
<thead>
<tr>
<th>Action</th>
<th>P1 state</th>
<th>P2 state</th>
<th>P3 state</th>
<th>Bus action</th>
<th>Data from</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 reads x</td>
<td></td>
<td></td>
<td></td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>P2 reads x</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>BusRd</td>
<td>Memory or Cache of P1 (FlushOpt)</td>
</tr>
<tr>
<td>P1 writes x</td>
<td>M</td>
<td>I</td>
<td>I</td>
<td>BusRdX*</td>
<td>Cache</td>
</tr>
<tr>
<td>P1 reads x</td>
<td>M</td>
<td>I</td>
<td>I</td>
<td>-</td>
<td>Cache</td>
</tr>
<tr>
<td>P3 writes x</td>
<td>I</td>
<td>I</td>
<td>M</td>
<td>BusRdX</td>
<td>Memory or Cache of P1 (FlushOpt)</td>
</tr>
</tbody>
</table>
Related Protocols: MOESI (AMD)

- Extended MESI protocol (What could be improved?)
- *Cache-to-cache transfer* of modified cache lines
  - Cache in M or (new) O state always transfers cache line to requesting cache
  - No need to contact (slow) main memory
- Avoids write back when another process accesses cache line
  - Good when cache-to-cache performance is higher than cache-to-memory
    *E.g., shared last level cache!*

Related Protocols: MOESI (AMD)

- Modified (M): Modified Exclusive
  - No copies in other caches, local copy dirty
  - Memory is stale, *cache supplies copy* (reply to BusRd*)
- Owner (O): Modified Shared
  - Exclusive right to make changes
  - Other S copies may exist ("dirty sharing")
  - Memory is stale, *cache supplies copy* (reply to BusRd*)
- Exclusive (E):
  - Same as MESI (one local copy, up to date memory)
- Shared (S):
  - Unmodified copy may exist in other caches
  - Memory is up to date unless an O copy exists in another cache
- Invalid (I):
  - Same as MESI
Multi-level caches

- Most systems have multi-level caches (here assume 2)
  - Problem: only “last level cache” is connected to bus or network
  - Yet, snoop requests are relevant for inner-levels of cache (L1)
  - Modifications of L1 data may not be visible at L2 (and thus the bus)

- L1/L2 modifications
  - On BusRd check if line is in M state in L1
    *It may be in E or S in L2!*
  - On BusRdX(*) send invalidations to L1
  - Everything else can be handled in L2

Directory-based cache coherence

- Snooping does not scale
  - Bus transactions must be *globally* visible
  - Implies broadcast

- Typical solution: tree-based (hierarchical) snooping
  - Root becomes a bottleneck

- Directory-based schemes are more scalable
  - Directory (one entry for each cache line) keeps track of all owning caches
  - Point-to-point update to involved processors
    *No broadcast*
  - *Can use specialized (high-bandwidth) network, e.g., HT, QPI …*
Basic Scheme (Sketch)

- System with N processors $P_i$
- For each memory block (size: cache block) maintain a directory entry
  - N presence bits
    - i-th bit set = block is in cache of $P_i$
  - 1 dirty bit (red)
- First proposed by Censier and Feautrier (1978)

```
P_0  P_1  P_2
  Cache   Cache   Cache
      X = 0       X = 7
Main Memory

Directory
Presence bits
Dirty bit
X = 7
```

Cache block containing x (and adjacent data)!

Directory-based CC: Read miss

- $P_0$ intends to read, misses
- If dirty bit (in directory) is off
  - Read from main memory
  - Set presence[i]
  - Supply data to reader

```
Read X  P_1  P_2
  Cache   Cache   Cache
      X = 7
Main Memory

Directory
Presence bits
X = 7
```
Directory-based CC: Read miss

- \( P_0 \) intends to read, misses

- If dirty bit is on
  - Recall cache line from \( P_j \) (determine by presence[])
  - Update memory
  - Unset dirty bit, block is shared
  - Set presence[]
  - Supply data to reader

Directory-based CC: Write miss

- \( P_0 \) intends to write, misses

- If dirty bit (in directory) is off
  - Send invalidations to all \( P_j \) with presence[] turned on
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[], owner \( P_i \)
Directory-based CC: Write miss

- \( P_0 \) intends to write, misses

- If dirty bit is on
  - Recall cache line from owner \( P_j \), invalidate there
  - Write to cache
  - Unset presence\([j]\)
  - Set presence\([i]\)
  - Dirty bit stays on
  - Acknowledge to writer

Read hit and Write hit

- Not shown, think about it at home
Discussion

- **Scaling of memory bandwidth**
  - No centralized memory

- **Directory-based approaches scale with restrictions**
  - Require presence bit for each cache
  - Number of bits determined at design time
  - Directory requires memory (size scales linearly)
  - Shared vs. distributed directory

- **Software emulation**
  - Distributed shared memory (DSM)
  - Emulate cache coherence in software (e.g., TreadMarks)

---

**Example Fast Fourier Transform**

Each stage has 8 butterflies

**butterfly**: one add, one sub, one mult (not shown)
Example: Fast Fourier Transform

Six-step FFT

Input (16 values)

Output

Assume: Cache line can hold 2 values

Multi-core FFT

Open Problems (for projects, theses, research)

- Tune algorithms to cache-coherence schemes
  - What is the optimal parallel algorithm for a given scheme?
  - Parameterize for an architecture

- Measure and classify hardware
  - Read Maranget et al. “A Tutorial Introduction to the ARM and POWER Relaxed Memory Models” and have fun!
  - RDMA consistency is not well understood!
  - GPU memories are not well understood!
  - Huge potential for new insights!

- Can we program (easily) without cache coherence?
  - How to fix the problems with inconsistent values?
  - Compiler support (issues with arrays?)