cache coherence in NUMA systems
overview

- caching introduction
- coherence and consistency
- coherence protocols
- cache only memory architecture (COMA)
- appendix: implementation details
why caching?

- access to main memory is slow
- faster memory is available but expensive
- caches are trade-off (cost / speed)
- even multiple levels of caching (usually one to three)
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td>invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111111</td>
<td>invalid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000001</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000010</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>1111111</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
</tbody>
</table>
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0000010</td>
<td>001011</td>
<td>invalid</td>
</tr>
</tbody>
</table>

invalid -> cache miss

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000001</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000010</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111111</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
</tbody>
</table>
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>00000001</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000010</td>
<td>101</td>
<td>0xFF ... 0x53</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111111</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
</tbody>
</table>
**how does it work?**

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000001</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000010</td>
<td>101</td>
<td>0xFF ... 0x53</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111111</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
</tbody>
</table>

**valid + tag correct -> cache hit**
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>0000000</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0000001</td>
<td>0000001</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0000010</td>
<td>101</td>
<td>0xFF ... 0x53</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111111</td>
<td>1111111</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>0000000</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0000001</td>
<td>0000001</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0000010</td>
<td>101</td>
<td>0xFF ... 0x53</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111111</td>
<td>1111111</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

definition of cache line

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>00000001</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>00000010</td>
<td>110</td>
<td>0xAB ... 0xC3</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111111</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
</tbody>
</table>

Tag  Index  Offset
110  0000010  101101

valid, but tag incorrect -> cache miss
replace cache line, although most of cache is empty

direct mapped
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000001</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000010</td>
<td>1010</td>
<td>0xFF ... 0x53</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1000010</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>000010</td>
<td>101101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64B)</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000001</td>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td>0000010</td>
<td>1010</td>
<td>0xFF ... 0x53</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1000010</td>
<td>1100</td>
<td>0xAB ... 0xC3</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2-way set associative
how does it work?

cache-size: 8KB, block-size: 64B, byte-addressable, 16 bit addresses

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>0000000</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0000001</td>
<td>0000001</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0000010</td>
<td>1010000010</td>
<td>0xFF ... 0x53</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1000010</td>
<td>1100000010</td>
<td>0xAB ... 0xC3</td>
<td>valid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
cache associativity

- direct mapped
  - one cache block per index
  - **fastest hit times**, but higher miss rates

- fully associative
  - every address can be stored in every cache block
  - **lowest miss rates**, but slower hit times

- n-way set associative
  - n cache blocks need to be checked per lookup

- current implementations
  - 4-8-way set associative (L1) / 16-way set associative (L2)
  - `grep . /sys/devices/system/cpu/cpu0/cache/index*/*`
modifying cached data

- only applies to data caches, since instruction caches are read only

- requires an additional modified flag per cache line

- several write policies
  - write-back: write changes to cache first
  - write-through: write changes directly to main memory
  - write-allocate: write to non-cached block will load block
replacement strategies

- only applies to non-direct mapped caches
- first-in-first-out (FIFO): oldest entry
- least recently used (LRU): entry with least recent access
- least frequently used (LFU): “rarely” used entry
- CLOCK: additional clock bit (set on access, reset on replacement)
- random
what about multiple cores?

- caching becomes inherently harder when applied to multi-core systems
  - every system with multiple cores that have access to some shared memory
  - not only NUMA, but also SMP

- what properties can programmers expect?
- what guarantees can hardware deliver?
consistency vs coherence

- coherence :: for any memory location and any given time there is either
  - a single core that can read and write it
  - multiple cores that can only read it

- coherence protocols maintain this invariant

- coherence makes caches “invisible”
coherence is not enough

what is the value of \texttt{var} after execution?

P1
\begin{verbatim}
data = 'new'
flag = 1
\end{verbatim}

P2
\begin{verbatim}
while (flag != 1) {} 
var = data
\end{verbatim}
coherence is not enough

P1

```
data = 'new'
flag = 1
```

P2

```
while (flag != 1) {} 
var = data
```

depends on execution order
coherence is not enough

P1

data = 'new'
flag = 1

P2

while (flag != 1) {}
var = data

'new', correct
coherence is not enough

\begin{align*}
P1 & \\
\text{flag} &= 1 \\
\text{data} &= \text{‘new’} \\
\end{align*}

\begin{align*}
P2 & \\
\text{while (flag} &\neq 1) \\
\text{var} &= \text{data} \\
\end{align*}

maybe unexpected, but not (necessarily) wrong
consistency vs coherence

- consistency :: specification of **allowed behavior** of **multi-threaded** programs with **shared memory**
  - what can programmers expect?
  - what optimizations can hardware do?
consistency vs coherence

- consistency :: specification of **allowed behavior** of **multi-threaded** programs with **shared memory**
- what can programmers expect?
- what optimizations can hardware do?

\[
\begin{align*}
\text{P1} & : S1: x = 'new' \\
& : L1: r1 = y \\
\text{P2} & : S2: y = 'new' \\
& : L2: r2 = x
\end{align*}
\]
serial consistency

- result of execution is the same as if instructions had been executed in program order
  - no reordering allowed
  - most intuitive model
serial consistency

P1

S1: x = 'new'
L1: r1 = y

P2

S2: y = 'new'
L2: r2 = x

result (r1='new', r2='new')
relaxed consistency - total storage order

- some reorderings are allowed
  - load → load / store → load / load → store / store → store

- relaxed sequential consistency
  - store → load must not longer be ordered
  - can be implemented using FIFO write buffer
  - common implementation (x86 and AMD64)
relaxed consistency - total storage order

P1

S1: x = 'new'
L1: r1 = y

P2

S2: y = 'new'
L2: r2 = x

result (r1=0, r2=0)
relaxed consistency - total storage order

now either store instruction has to be completed first
weak consistency

- all memory accesses can be reordered
- only FENCE instructions protect order
why should I care?

- correctness
  - caches are transparent only on cache coherent systems
  - on non-cc systems
    - do not use same memory from different cores
    - disable caching
    - implement cache coherence in software
      - manually flush caches
why should I care?

- performance
  - caches work on a cache block granularity (for example 64B)
  - problem: frequently writing to variables in same cach block from different cores
    - cache line will be repeatedly moved / invalidated
    - huge performance drop
  - not apparent from code \( \rightarrow \) false sharing
  - modern compilers “spread” variables
how do we achieve coherence?

- coherence protocols
  - snoopy based
    - each CPU is responsible for local cache
    - listens to all transactions on memory bus
  - directory based
    - central directory handles coherence
    - home node stores location of data in bit vector
- hybrid approaches
  - snoopy for local nodes
  - directory between distant node clusters
**MSI (modified, shared, invalid)**

- **snoopy protocol, write-back**

- **three states per cache line**
  - invalid: not yet used or invalidated
  - shared: one or more valid copies
  - modified: only valid copy is local

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

allowed states for any two caches
MSI (modified, shared, invalid)

- two processor operations
  - PrRd: processor reads data
  - PrWr: processor writes data
- three bus signals
  - BusRd: some processor reads
  - BusRdX: some processor reads exclusively
  - BusFlush: write data to bus
MSI (modified, shared, invalid)

allowed states for any two caches
MSI (modified, shared, invalid)

Allowed states for any two caches:

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
MSI (modified, shared, invalid)

Allowed states for any two caches:

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
MSI (modified, shared, invalid)

- **PrRd/BusRd**: State where the cache is read from and updated back to the main memory.
- **PrRd/-**: State where the cache is read from but not updated.
- **PrWr/-**: State where the cache is written to but not updated.
- **BusRd/-**: State where the bus reads from the cache.
- **BusRdX/-**: State where the bus reads from the cache with exclusive access.
- **PrWr/BusRdX**: State where the cache is written to and updated back to the main memory.
- **PrRd/-PrWr/-**: State where the cache is read from and written to, but not updated.
- **PrWr/BusRdX**: State where the cache is written to and updated back to the main memory.
- **BusRd/Flush**: State where the bus updates the cache with a flush operation.
- **BusRdX/Flush**: State where the bus updates the cache with an exclusive flush operation.

**Allowed states for any two caches**

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

These states represent the possible interactions between two caches in an MSI system.
MSI (modified, shared, invalid)

allowed states for any two caches
MSI (modified, shared, invalid)

PrRd $\rightarrow$ BusRd

Allowed states for any two caches:

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

PrRd/BusRd
PrRd/-PrRd/-
BusRdX/-
BusRdX/Flush
PrWr/BusRdX
PrWr/BusRd
PrWr/-PrWr/-
BusRd/Flush
MSI (modified, shared, invalid)

PrRd → BusRd
PrRd → BusRd

allowed states for any two caches
MSI (modified, shared, invalid)

PrRd → BusRd
PrRd → BusRd
PrWr → BusRdX

allowed states for any two caches
**MSI (modified, shared, invalid)**

Allowed states for any two caches:

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M</strong></td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

- **PrRd → BusRd**
- **PrRd → BusRd**
- **PrWr → BusRdX**
- **PrWr → BusRdX Flush**
MSI (modified, shared, invalid)

- **PrRd** → **BusRd**
- **PrRd** → **BusRd**
- **PrWr** → **BusRdX**
- **PrWr** → **BusRdX Flush**
- **PrRd** → **BusRd Flush**

Allowed states for any two caches:
CC-NUMA

- cache coherent NUMA systems
- most snoopy protocols rely on bus system
  - where global order of bus signals exists
- modern NUMA systems do not use buses but point-to-point links
- idea: replace bus with point-to-point links and snoop
cache only memory architecture (COMA)

- increasing number of cores causes increasing overhead
- treat all main memory as cache
  - memory address no longer has a home node
  - memory moves when needed on a different core
Intel single chip cloud computer (SCC)

- research vehicle for COMA computing
- 48 Pentium-I processors (arranged on a 4x6 mesh)
- per tile: 2 cores, 16KB message passing buffer, router, caches
- up to 64GB RAM
- no cache coherence implemented
Intel single chip cloud computer (SCC)

- memory organization
  - private DRAM region per core (off-die)
  - larger shared DRAM region (with caches disabled, off-die)
  - shared message passing buffer (on-die)
programming by shared memory
- via SCC’s native communications library (RCCE)
- RCCE_shmalloc() returns pointer to new shared region
- RCCE_put() / RCCE_get() used to share MPB regions
  - secured by test and set
programming by **message passing**

- RCCE_send() / RCCE_recv()
- local put / remote get
- blocking functions
- pipelining for large messages
private DRAM

core0

page table

private DRAM

core17

page table

message passing buffer (8KB)
private DRAM

core0

message passing buffer (8KB)

private DRAM

core17

1. access violation

page table

page table
private DRAM

core0

page table

message passing buffer (8KB)

private DRAM

core17

page table

1. access violation

2. send request
1. access violation

2. send request

3. flush and clear access permission

message passing buffer (8KB)
1. access violation

2. send request

3. flush and clear access permission

4. send frame

message passing buffer (8KB)
private DRAM

- 1. access violation
- 2. send request
- 3. flush and clear access permission
- 4. send frame
- 5. update access permissions

message passing buffer (8KB)

private DRAM

- 1. access violation
- 2. send request
- 3. flush and clear access permission
- 4. send frame
- 5. update access permissions

core0

page table

core17

page table
questions?
“A Primer on Memory Consistency and Coherence”  
D. J. Sorin,  M. D. Hill,  D. A. Wood (Stanford University)  
https://class.stanford.edu/c4x/Engineering/CS316/asset/A_Primer_on_Memory_Consistency_and_Coherence.pdf

“Cache Coherence” from Computer Architecture lecture  
D. J. Sorin (Duke University)  

from Computer Design and Organization lecture  
Luke McDowell (University of Washington)  
https://courses.cs.washington.edu/courses/cse471/00au/Lectures/luke_directories.pdf  
https://courses.cs.washington.edu/courses/cse471/00au/Lectures/luke_snooping.pdf

“Cache Coherence Techniques For Multicore Processors”, dissertation  
Michael R. Marty (University of Wisconsin–Madison)  
http://research.cs.wisc.edu/multifacet/theses/michael_marty_phd.pdf
sources & references

“Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures”
P. Stenstrom, T. Joe, A. Gupta (Berkley University)

“Using Intel’s Single-Chip Cloud Computer”
Tim Mattson (Microprocessor and Programming Lab, Intel Corporation)

“A Life Without Cache Coherence”
Dr. rer. nat. S. Lankes, Dr.-Ing. C. Clauß (RWTH Aachen)

“Source Snooping Cache Coherence Protocols”
J. R. Goodman (University of Auckland)
http://parlab.eecs.berkeley.edu/sites/all/parlab/files/20091029-goodman-ssccp.pdf

“MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects”
J. R. Goodman (University of Auckland), H.H.J. Hum (Intel Corporation)
appendix

- performance optimizations on MSI
  - MESI
  - MESIF

- implementation on NUMA system (without bus)
MSI (modified, shared, invalid)

- unnecessary large amount of bus transactions
  - if only one core has data in shared state, there is no need for BusRdX

- solution: add additional state
  - exclusive: data is unmodified, but only read by one core
MESI (modified, exclusive, shared, invalid)

allowed states for any two caches
MESI (modified, exclusive, shared, invalid)

allowed states for any two caches
MESI (modified, exclusive, shared, invalid)

Allowed states for any two caches:

- M (Modified)
- E (Exclusive)
- S (Shared)
- I (Invalid)

States:
- PrRd/BusRd(s)
- PrRd/BusRd(!s)
- PrRd/-
- PrWr/BusRdX
- PrWr/-
- BusRd/-
- BusRdX/-
- BusRdX/Flush
MESI (modified, exclusive, shared, invalid)

allowed states for any two caches

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>E</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>E</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
**MESI (modified, exclusive, shared, invalid)**

Allowed states for any two caches:

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>E</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>E</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>I</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

- **I** (Invalid): A cache is invalid, and its lines are flushed from other caches.
- **S** (Shared): A cache holds a shared copy of the data.
- **E** (Exclusive): A cache is exclusive, and its data is not shared by other caches.
- **M** (Modified): A cache holds a modified copy of the data.

Transitions between states:
- **PrRd/BusRd(s)**: Privileged read to bus read shared.
- **PrRd/BusRd(1)s**: Privileged read to bus read exclusive.
- **PrWr/BusRdX**: Privileged write to bus read shared.
- **PrRd/-**: Privileged read to invalid.
- **PrWr/-**: Privileged write to invalid.
- **PrWr/BusRdX**: Privileged write to bus read shared.
- **BusRd/-**: Bus read to invalid.
- **BusRdX/-**: Bus read with flush to invalid.
- **BusRdX/Flush**: Bus read with flush to shared.
- **BusRd/Flush**: Bus read to shared.
- **BusRdX/Flush**: Bus read with flush to exclusive.
- **BusRd/Flush**: Bus read to exclusive.

**Note:** The diagram and table illustrate the state transitions and allowed states for any two caches in the MESI protocol.
MESI (modified, exclusive, shared, invalid)

- still more bus transactions than necessary
  - if PrRd is issued for data in S state, it will either:
    - be served from main memory
    - be served by all cores holding it in S state
  - solution: add a new state (MESIF)
    - forward (F): same as S, but responsible for answering PrRd requests
    - forward state is passed on to least recent requester of data
point-to-point instead of a bus

- most snoopy protocols rely on bus system
  - where global order of bus signals exists
- modern NUMA systems do not use buses but point-to-point links
- idea: replace bus with point-to-point links and snoop
  - but: no intervention, no ordering
point-to-point instead of a bus

- system model
  - number of nodes A, B, C
  - home node H (owner of the memory)
  - pairwise point-to-point links (may be indirect)
  - MESIF protocol
reading uncached line
reading uncached line
reading uncached line
reading uncached line
reading uncached line
reading shared line
reading shared line

Diagram with arrows indicating connections between points labeled A, B, C, H, and MC.
reading line with conflict
reading line with conflict
reading line with conflict
reading line with conflict
reading line with conflict
reading line with conflict

conflict
reading line with conflict
reading line with conflict
reading line with conflict
reading line with conflict
reading line with conflict
reading line with conflict
complex conflicts can hardly be resolved this way

single point (home) is required to sort out conflicts
  winner determined by “first come, first served”
  home cares about loosers
conflict resolution (MESIF)

- home receives cache requests, but does not respond
- all nodes must report conflicts
- second message to home (cancelling or confirming)
- in case of conflict
  - winner is instructed to forward data to looser
  - looser receives acknowledgment but no data
reading shared line (MESIF)
reading shared line (MESIF)
reading shared line (MESIF)
conflict resolution (QPI)

- home receives all responses
- data is directly send to requester
- home sends acknowledgement to requester
  - including data, if not already delivered
reading shared line (QPI)
reading shared line (QPI)
reading shared line (QPI)
reading shared line (QPI)
MESIF vs QPI

- QPI specification is not public (hard to compare)
- both allow for cache-to-cache responses
- QPI requires one less request for common / simple cases
- MESIF requires less messages in case of conflicts