# Stale data, or how we (mis-)manage modern caches

Mark Rutland <`mark.rutland@arm.com`>

Embedded Linux Conference 2016

# Scope

The behaviour of caches is surprisingly complex!

- Depends on architecture, implementation, and integration
- Commonly misunderstood by software engineers
- Potential for subtle, non-repeatable software bugs

This talk is about the general behaviour in ARMv8-A

- Largely (but not entirely) applicable to ARMv7-A
- Focus on architectural guarantees and requirements
- The Architecture Reference Manual ("ARM ARM") is authoritative

# Warning

Examples in this presentation:

- describe non-architectural details
- assume specific potential implementations
- assume specific runtime configurations
- act as intuitive existence proofs
- do not describe all problems

These do not define the architectural envelope!

# Today: CPUs

Modern CPUs, even simple ones, gain efficiency and performance by many techniques:

- Automatic prefetching
- Store buffering
- Out-of-order execution
- Speculation

These have a non-deterministic impact on cache behaviour!

CPUs are likely to become more aggressive over time, making cache behaviour less deterministic.

# Today: Cache coherence protocols

Modern (SMP) systems support **cache coherence**:

> Accesses to a location act is if using the same copy of that location
> (e.g. a load returns the value of the last store)

Cache coherence protocols are becoming more advanced:

- Reduced memory traffic (e.g. fewer writebacks)
- Scalable to larger systems (e.g. shared lines)
- Fewer incidental coherence guarantees
- More stringent maintenance requirements in practice

# Today: Topology

Modern systems typically have many CPUs

- More CPUs means more non-determinism
- Also means more caches

Cache-coherent DMA masters are more common

- Yet more non-determinism
- Erroneous programming may break coherence!

Shared **system caches** becoming popular

- Typically much larger; can hold data for longer

# Today: looking forward

Systems are more complex and more varied than they used to be!

... and likely to become more so.

We cannot predict what future systems will look like.

... but we know they must follow architectural rules!

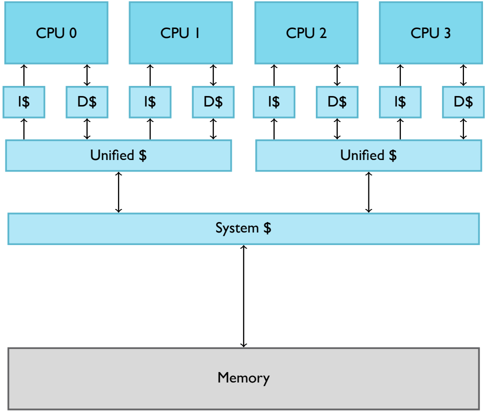
# ARM architecture: cache basics

Modified Harvard architecture:

- 0 to $\infty$[*] levels of instruction caches (I$)
- 0 to $\infty$[*] levels of data caches (D$)
- 0 to $\infty$[*] levels of unified caches (U$)

Many implementations permitted:

- Scales to power/performance/area points
- May be asymmetric (e.g. big.LITTLE)

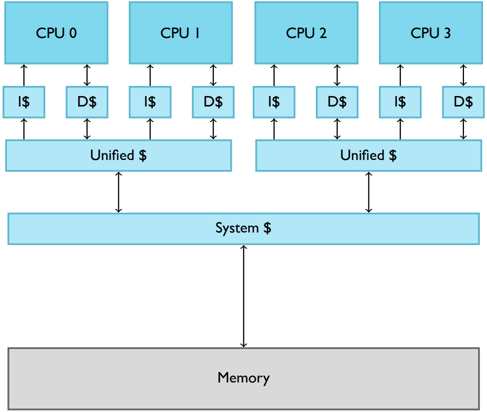


[*] CLIDR_EL1 lists 0 to 7 levels, may not include all caches!

# ARM architecture: cache coherence

Coherence in the ARM architecture:

- D$ - D$ coherence ensured by hardware*
- D$ - Memory coherence not ensured
- D$ - I$ coherence not ensured
- I$ - I$ coherence not ensured



Where coherence is required, but not ensured, explicit cache maintenance is necessary.

*So long as consistent *memory attributes* are used.

# ARM architecture: cache coherence

Every memory access has associated **memory attributes**:

- Memory type
- Cacheability
- Shareability

Coherence of a location **requires** consistent use of memory attributes.

Inconsistent usage (Mismatched memory attributes) can result in long-term loss of coherence!

# ARM architecture: memory types

**Device**:

- Accesses may have side-effects (e.g. MMIO)
- Subsumes Strongly Ordered from ARMv7-A
- No cacheability

**Normal**:

- Accesses do not have side effects (e.g. DRAM)
- redundant accesses permitted
- Several cacheability options possible

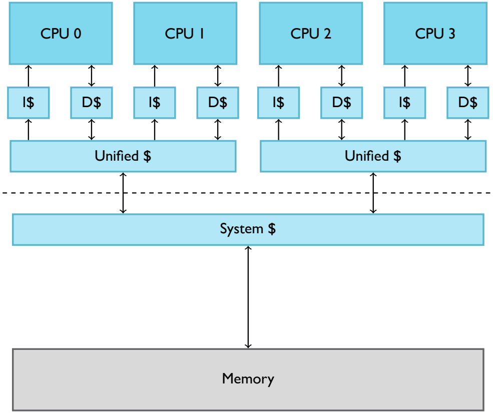
# ARM architecture: Cacheability

Many cacheability options for Normal memory:

- Non-Cacheable
- Write-Through (Non-)Transient
- Write-Back (Non-)Transient

Controlled Separately for Inner and Outer caches

- Inner caches are closest to CPU
- Outer Caches are furthest from CPU
- Boundary is IMPLEMENTATION DEFINED

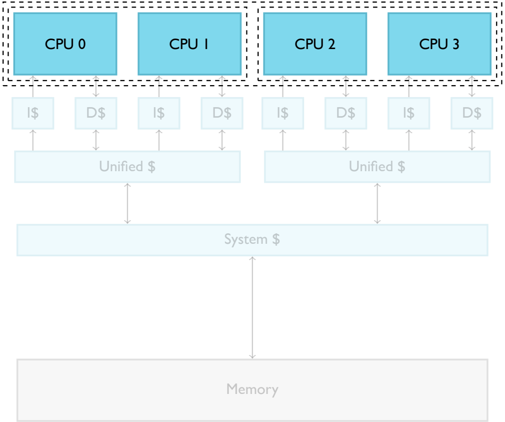


# ARM architecture: Shareability domains (1)

Hierarchical **shareability domains** contain CPUs:

- Non-Shareable
- Inner Shareable
- Outer Shareable
- System*

Coherence of a location can be limited to a particular domain, affecting caches.

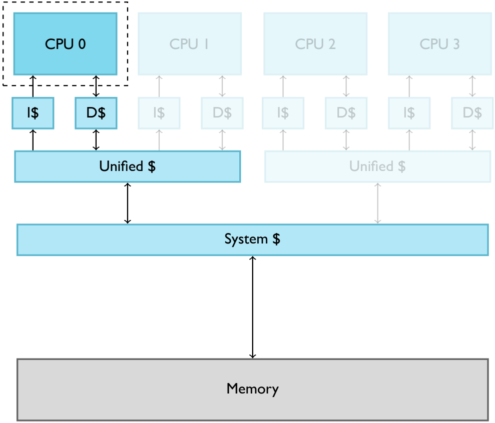


*A location cannot be System shareable, but barriers can target the System domain

# ARM architecture: Shareability domains (2)

**Non-Shareable** domain:

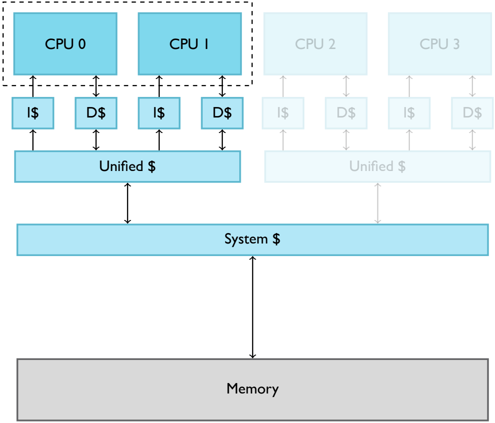
- Covers a single CPU.
- Not guaranteed to be coherent with any other CPUs.



# ARM architecture: Shareability domains (3)

**Inner Shareable** domain:

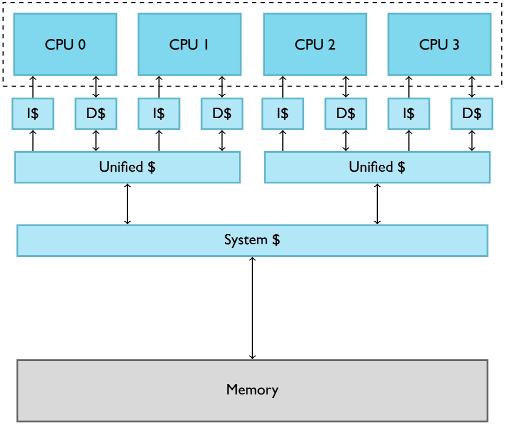
- Covers CPUs intended for use by a single OS or hypervisor.
- May cover some devices



# ARM architecture: Shareability domains (4)

**Outer Shareable** domain:

- Covers the largest set of CPUs which can be coherent.
- May cover some devices



# ARM architecture: cache states

The ARM architecture does not mandate a specific cache coherence protocol.

Commonly, protocols allow for a location to be:

- **Invalid**: No data in any caches
- **Clean**: Copied from memory, unchanged  
  Present in some caches
- **Dirty**: Has been written to  
  Present in some caches

# ARM architecture: general cache behaviour

Caches may allocate clean lines at any time for cacheable locations

- Due to speculation, prefetching, etc
- Impossible to prevent

Caches may write back dirty lines at any time

- To make space for new allocations
- Even if MMU is off
- Even if Cacheable accesses are disabled (caches are never 'off')

# Cache maintenance

Sometimes we need coherence that the hardware doesn't guarantee:

- Non-coherent DMA
- Modifying instructions
- Changing memory attributes for a location

We can ensure coherence for these cases with cache maintenance.

# Cache maintenance: terminology

The ARM architecture defines three maintenance operations:

- **Clean**: Write *dirty* data back, marking cached copies clean
- **Invalidate**: delete data from caches
- **Clean+Invalidate**: Clean followed by Invalidate

**No "flush" is defined** - may mean any of the above.

# Cache maintenance: Set/Way

Instructions for `IMPLEMENTATION DEFINED` power-up and power-down cache management:

- `DC ISW`: Data Cache Invalidate by Set/Way
- `DC CSW`: Data Cache Clean by Set/Way
- `DC CISW`: Data Cache Clean+Invalidate by Set/Way

These instructions **cannot** be used to ensure coherence:

- Only affect caches local to a CPU (not other CPUs or system caches)
- Not atomic: race with usual behaviour of caches
- Misuse may result in a loss of coherence!

# Cache maintenance: VA

Cache maintenance by VA can be used to ensure coherence:

- Affects all caches for shareability domain of VA
- ... including system caches (since ARMv8-A*)
- No race with usual cache behaviour
- Posted: batches completed with memory barriers

Maintenance by VA operates to two conceptual points:

- Point-of-Coherency (PoC)
- Point-of-Unification (PoU)

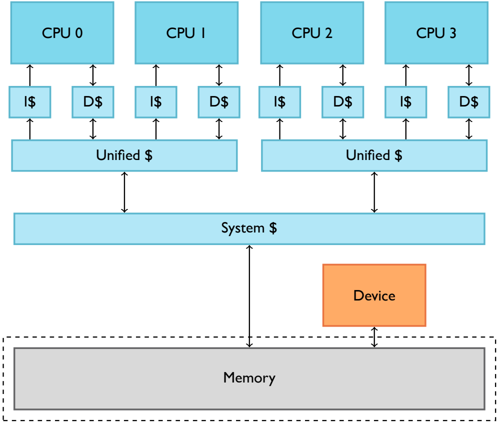
*In ARMv7-A, IMPLEMENTATION DEFINED maintenance may be necessary

# ARM architecture: PoC

Point-of-Coherency (PoC):

The point at which all accesses to a memory location see the same copy of that location. (i.e. where all accesses are coherent).

Typically the PoC is main memory, but *invisible* caches may exist between the PoC and memory.



# Cache maintenance: VA to PoC

Instructions for ensuring coherence with the PoC:

- `DC CVAC`: Data Cache Clean by VA to the PoC
- `DC IVAC`: Data Cache Invalidate by VA to the PoC
- `DC CIVAC`: Data Cache Clean+Invalidate by VA to the PoC

Useful for:

- Non-coherent DMA
- Changing memory attributes*
- Changing instructions (in some cases)

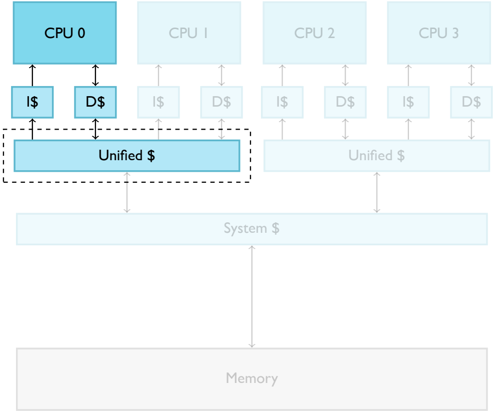
*requires care to avoid races with usual cache behaviour

# ARM architecture: PoU

Point-of-Unification (PoU):

The point at which the instruction caches and data caches of a particular CPU see the same copy of a memory location.

Each CPU has its own PoU, which may be shared with some other CPUs in the system.



# Cache maintenance: VA to PoU

Instructions for ensuring coherence with a set of PoUs:

- `DC CVAU`: Data Cache Clean by VA to the PoU
- `IC IVAU`: Instruction Cache Invalidate by VA to the PoU

Affects all caches for shareability domain of VA, prior to the set of PoUs for that domain.

# Cache maintenance: all (instruction)

Instructions for invalidating entirety of instruction cache(s)

- `IC IALLU`: Instruction Cache Invalidate All to PoU (local)
- `IC IALLUIS`: Instruction Cache Invalidate All to PoU (Inner Shareable)

Requires prior D$ maintenance to PoU or PoC!

There are no all data/unified maintenance instructions

# Cache maintenance: dodgy DMA

1. CPU allocates buffer
2. CPU invalidates buffer by VA to PoC
3. Non-coherent DMA into buffer
4. CPU reads buffer

CPU reads stale data. Why?

# Cache maintenance: dodgy DMA

1. CPU allocates buffer
2. CPU invalidates buffer by VA to PoC
3. Non-coherent DMA into buffer
4. CPU reads buffer

CPU reads stale data. Why?

Prefetching or speculation can occur between steps (1) and (4)!

# Cache maintenance: dodgy DMA

1. CPU allocates buffer
2. Non-coherent DMA into buffer
3. CPU invalidates buffer by VA to PoC
4. CPU reads buffer

CPU **still** reads stale data. Why?

# Cache maintenance: dodgy DMA

1. CPU allocates buffer
2. Non-coherent DMA into buffer
3. CPU invalidates buffer by VA to PoC
4. CPU reads buffer

CPU **still** reads stale data. Why?

Buffer had dirty lines which were evicted between steps (2) and (3)!

# Cache maintenance: dodgy DMA

1. CPU allocates buffer
2. CPU invalidates buffer by VA
3. Non-coherent DMA into buffer
4. CPU invalidates buffer by VA
5. CPU reads buffer

Does CPU read the DMA'd data?

# Cache maintenance: dodgy DMA

1. CPU allocates buffer
2. CPU invalidates buffer by VA
3. Non-coherent DMA into buffer
4. CPU invalidates buffer by VA
5. CPU reads buffer

Does CPU read the DMA'd data?

Yes! Step (2) avoids eviction of dirty lines, and step (4) removes lines allocated by prefetching or speculation.

# Closing

The behaviour of caches can be surprising, and careful management is required.

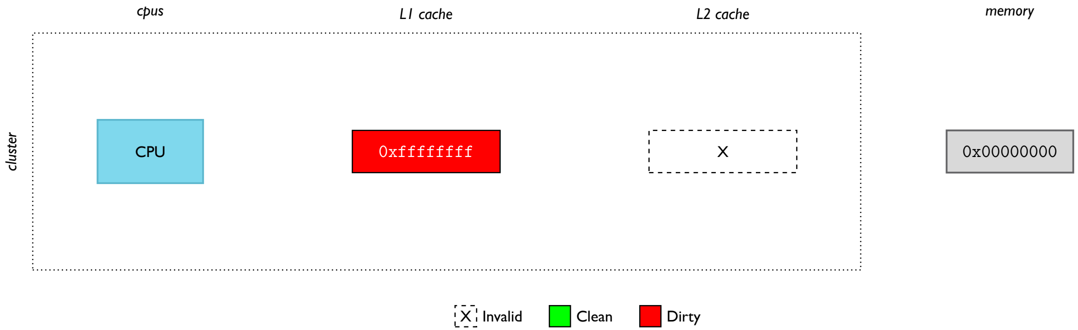
We can have fewer surprises if we think in terms of architecture.

The ARM architecture provides a simple, scalable cache model.

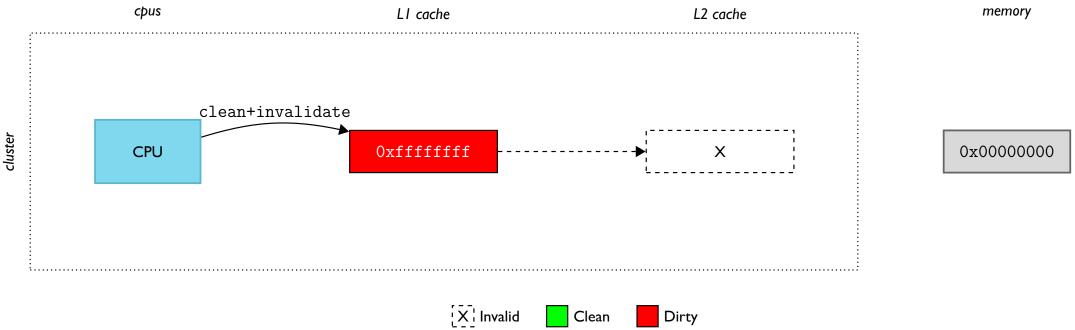
# Thank You

*The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.*

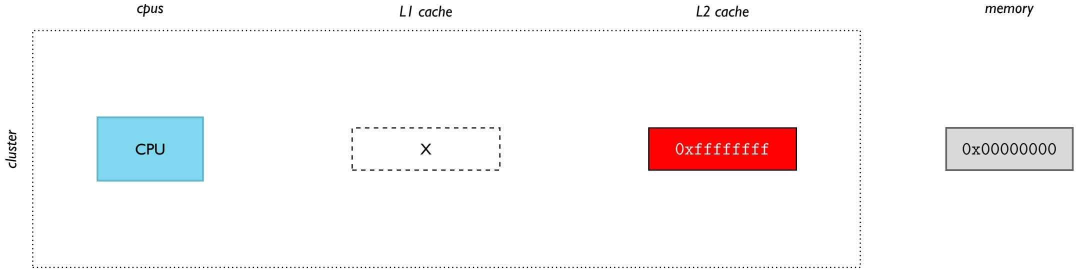
# Set/Way maintenance: example (intuition)



# Set/Way maintenance: example (intuition)

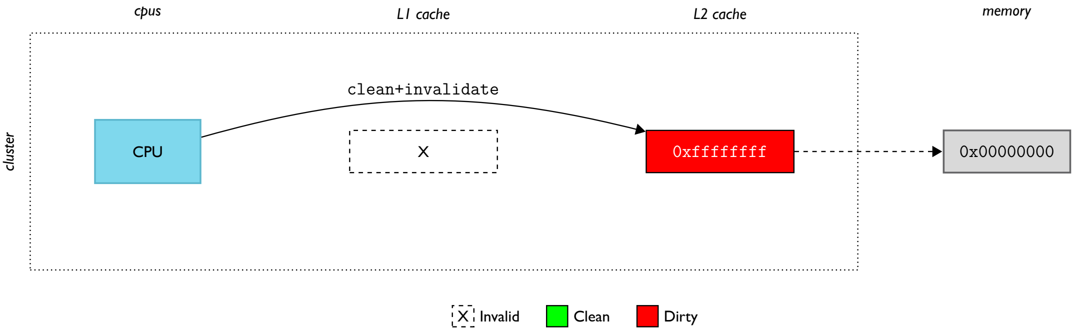


# Set/Way maintenance: example (intuition)

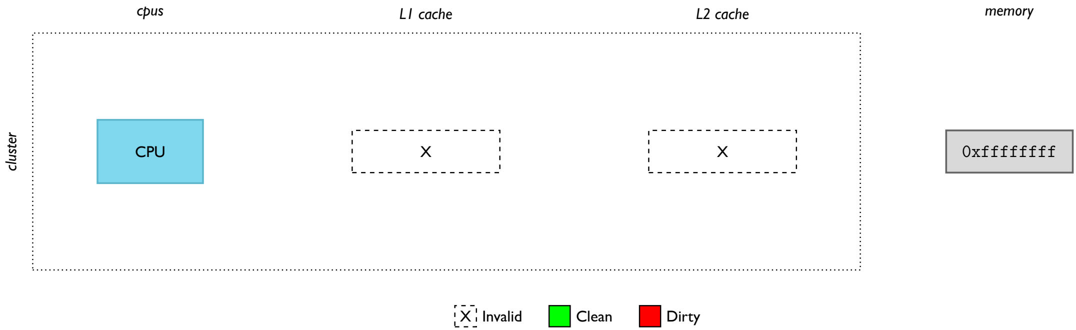


X Invalid 🟩 Clean 🟥 Dirty

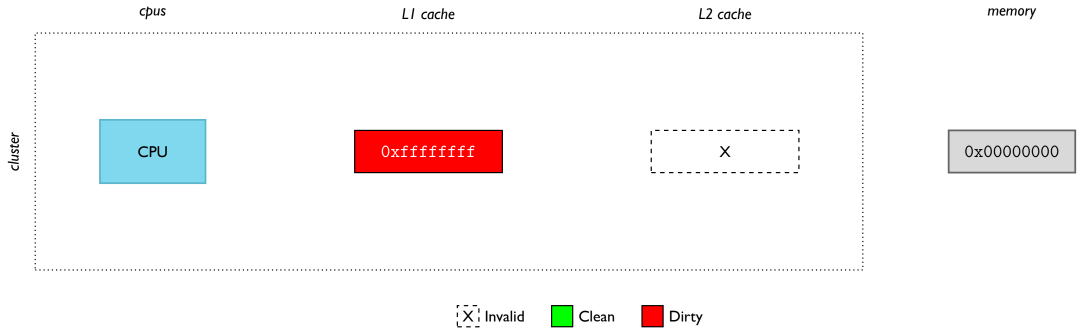
# Set/Way maintenance: example (intuition)



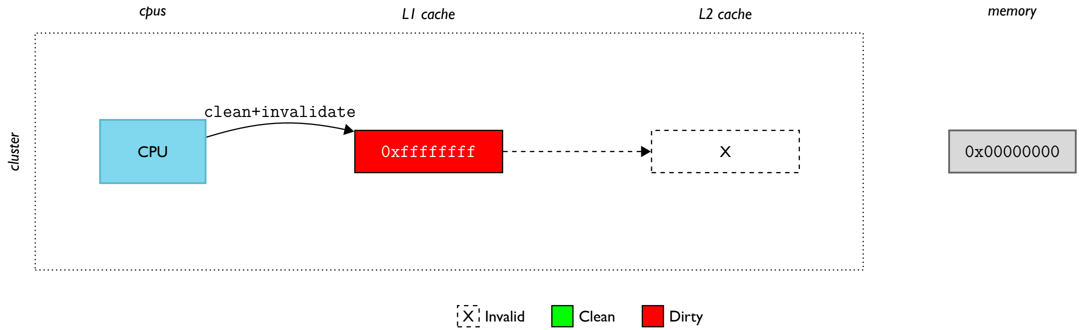
# Set/Way maintenance: example (intuition)



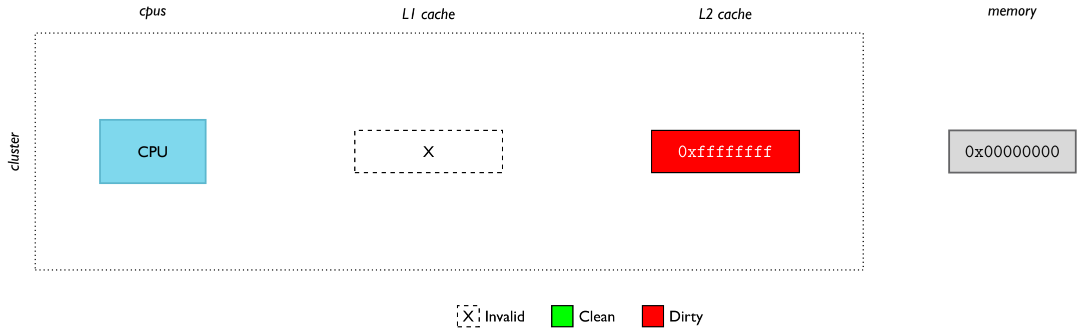
# Set/Way maintenance: example (speculation)



# Set/Way maintenance: example (speculation)



# Set/Way maintenance: example (speculation)



*cpus* — CPU

*L1 cache* — X

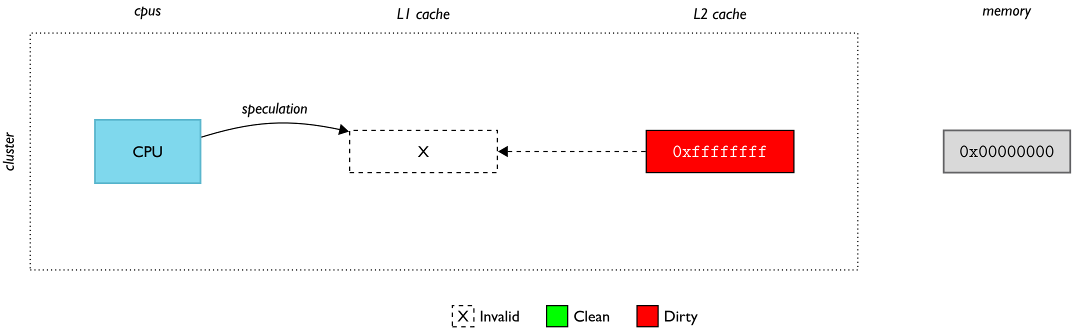
*L2 cache* — 0xffffffff

*memory* — 0x00000000

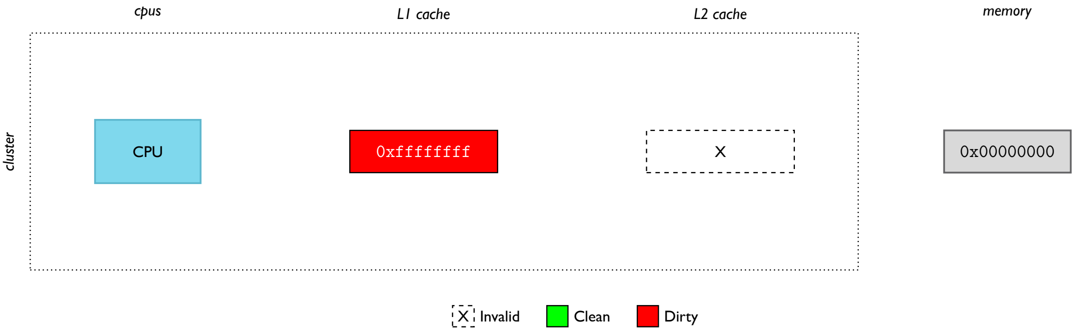
*cluster*

X Invalid 🟩 Clean 🟥 Dirty

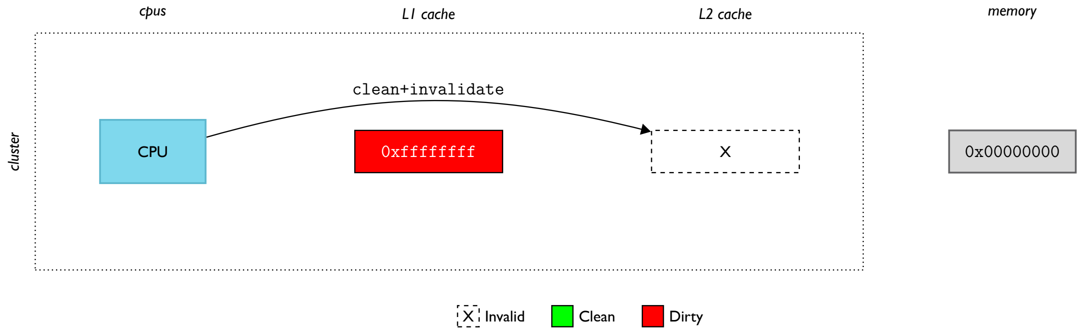
# Set/Way maintenance: example (speculation)



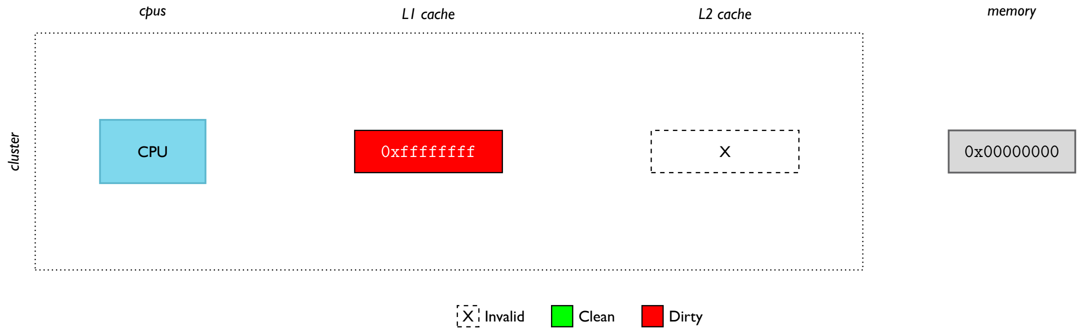
# Set/Way maintenance: example (speculation)



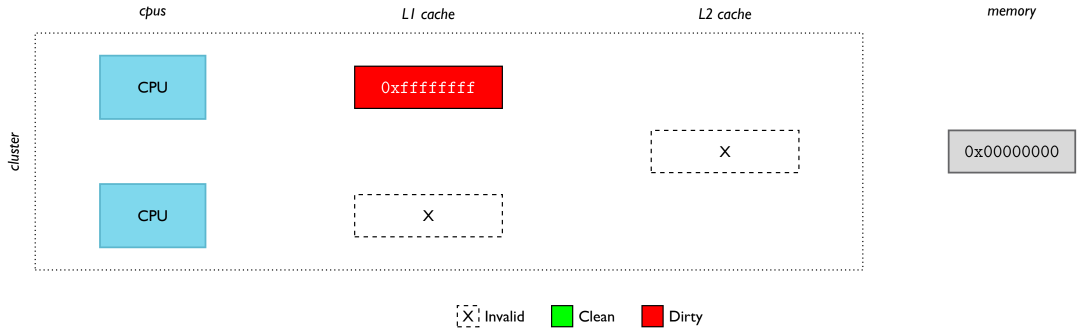
# Set/Way maintenance: example (speculation)



# Set/Way maintenance: example (speculation)



# Set/Way maintenance: example (migration)



*cpus*

*L1 cache*

*L2 cache*

*memory*

*cluster*

CPU

0xffffffff

X

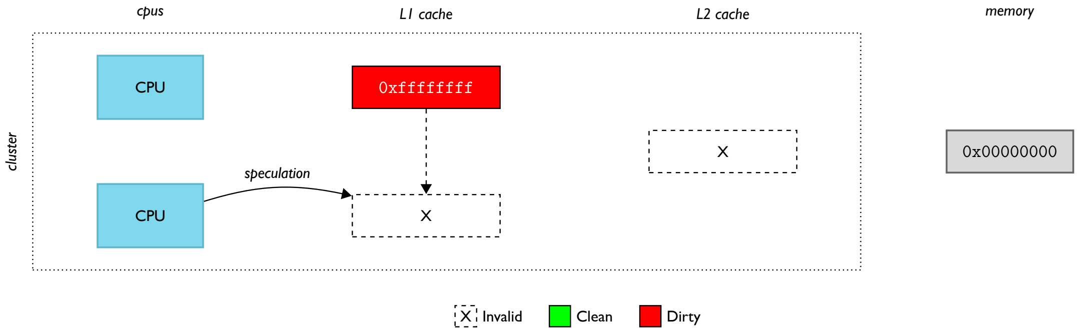
0x00000000

CPU

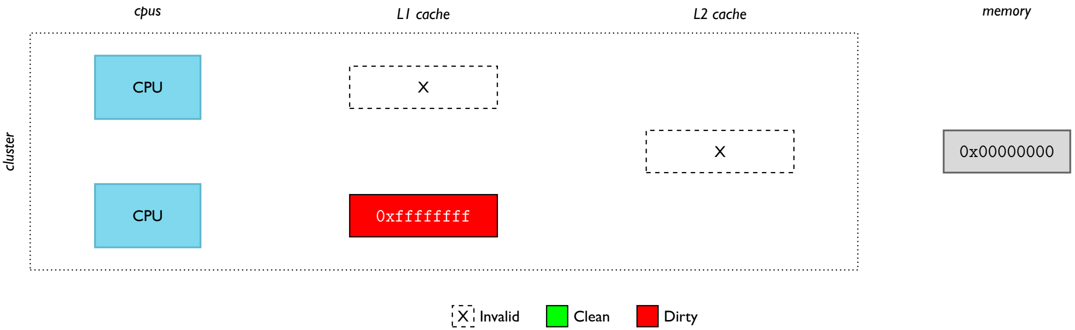
X

X Invalid 🟩 Clean 🟥 Dirty

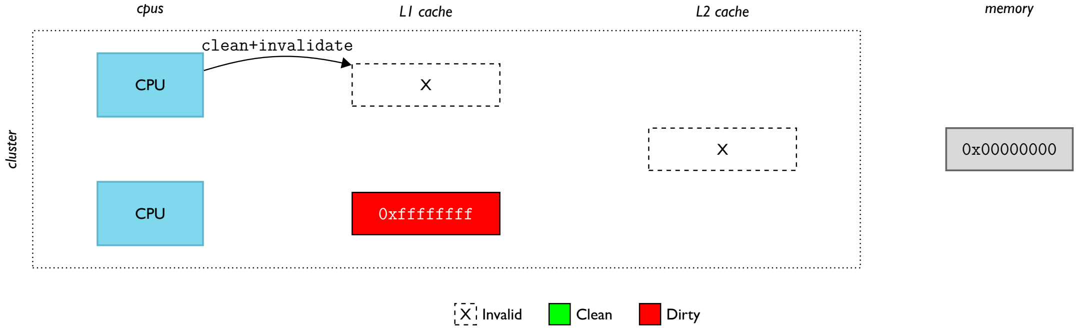
# Set/Way maintenance: example (migration)



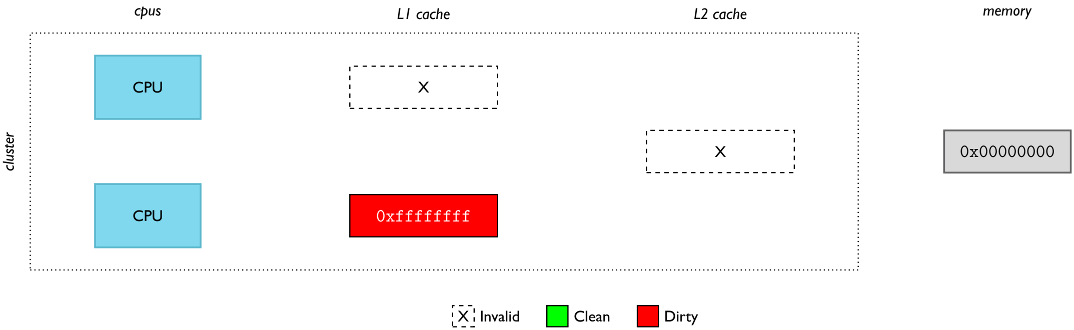
# Set/Way maintenance: example (migration)



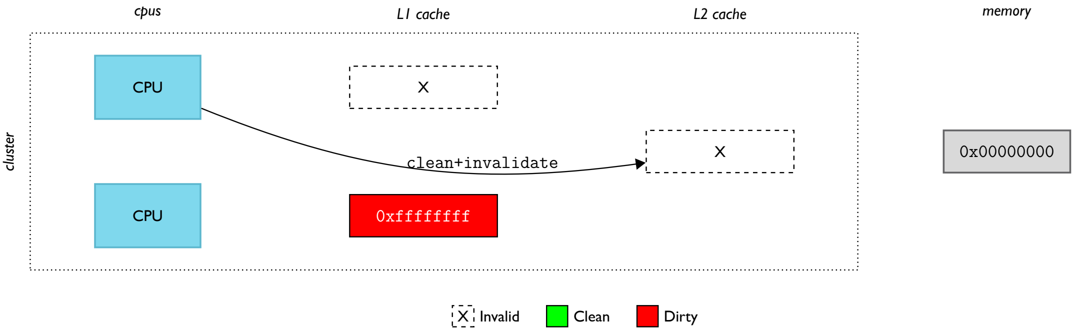
# Set/Way maintenance: example (migration)



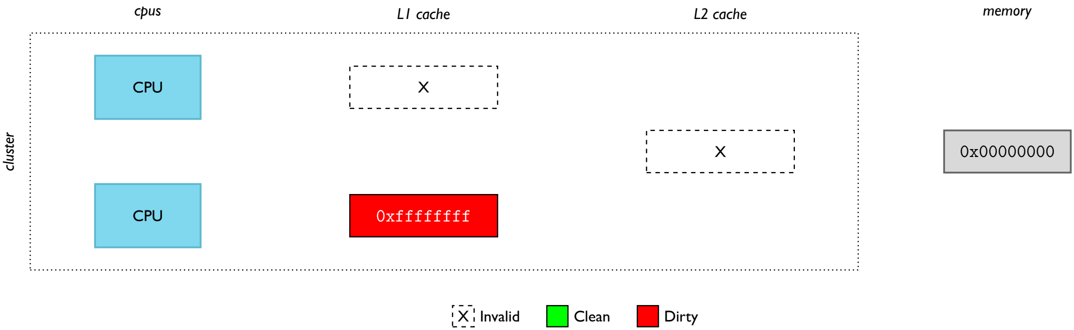
# Set/Way maintenance: example (migration)



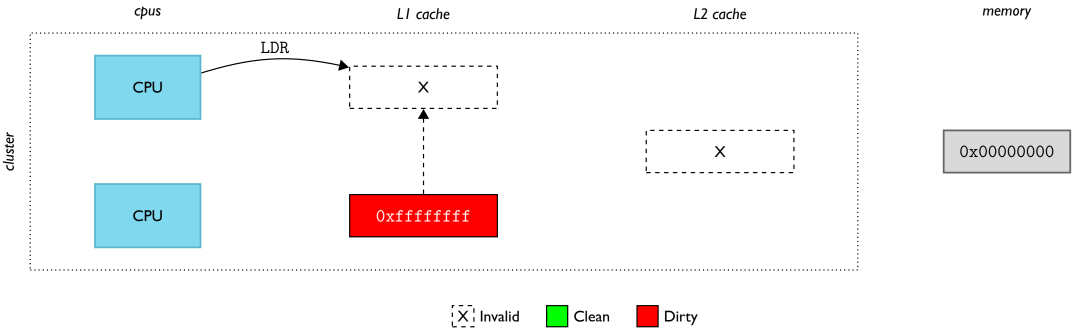
# Set/Way maintenance: example (migration)



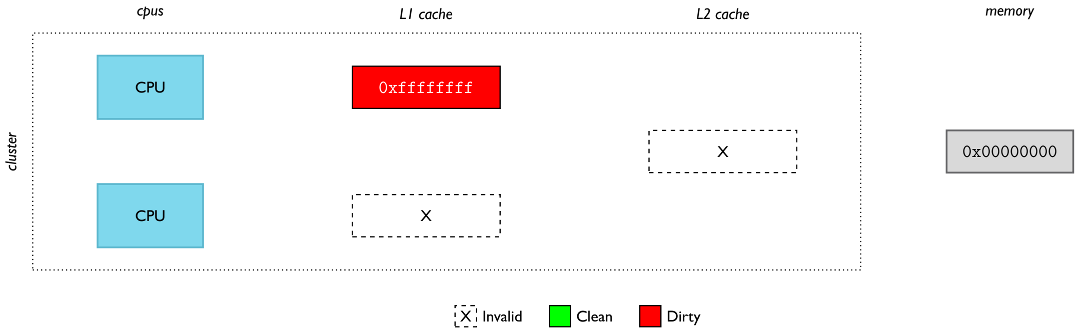
# Set/Way maintenance: example (migration)



# Set/Way maintenance: example (migration)



# Set/Way maintenance: example (migration)



*cpus*

*L1 cache*

*L2 cache*

*memory*

*cluster*

CPU

0xffffffff

CPU

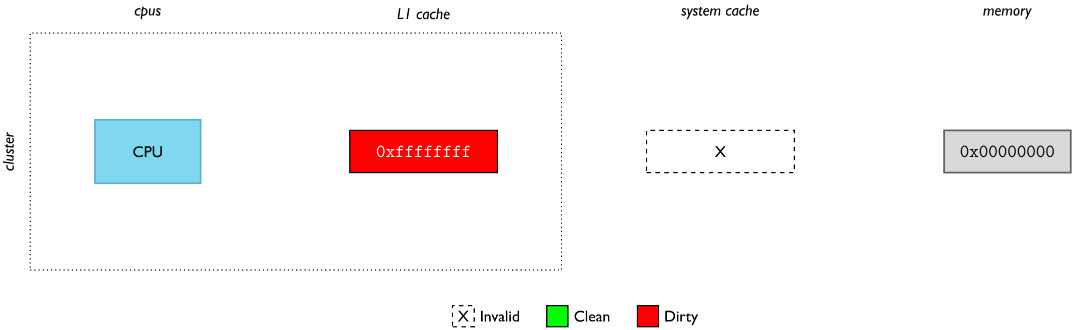
X

X

0x00000000

X Invalid 🟩 Clean 🟥 Dirty

# Set/Way maintenance: example (system caches)



cpus

L1 cache

system cache

memory

cluster

CPU

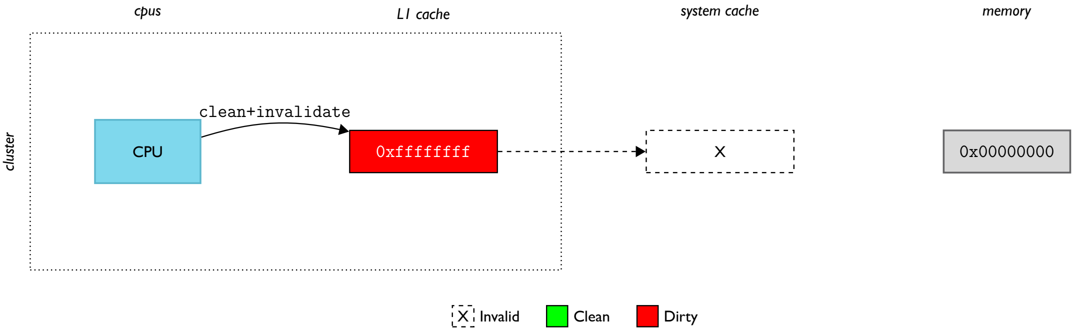
0xffffffff

X

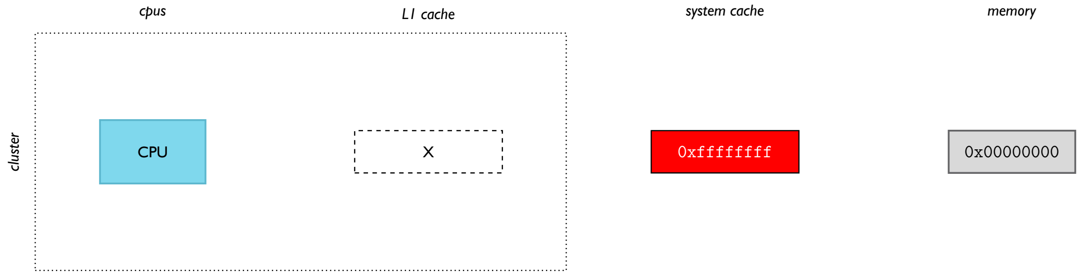
0x00000000

X Invalid 🟩 Clean 🟥 Dirty

# Set/Way maintenance: example (system caches)

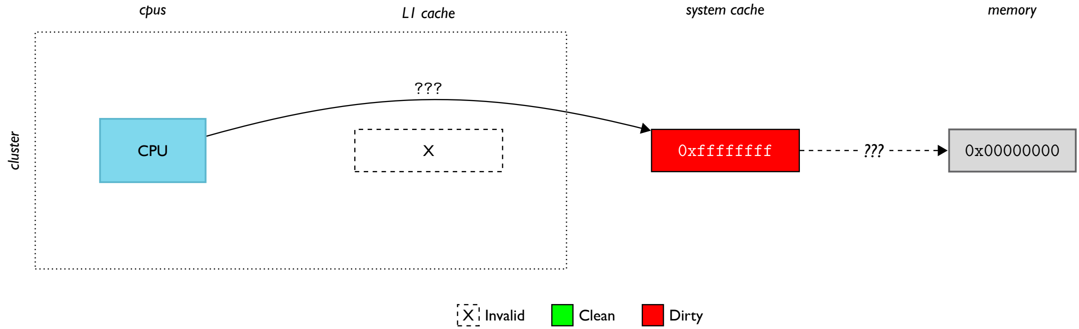


# Set/Way maintenance: example (system caches)

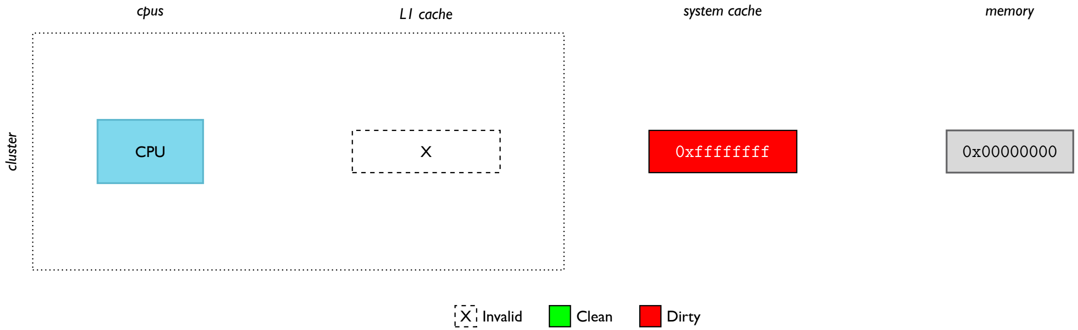


X Invalid 🟩 Clean 🟥 Dirty

# Set/Way maintenance: example (system caches)



# Set/Way maintenance: example (system caches)



*cpus*

*L1 cache*

*system cache*

*memory*

*cluster*

CPU

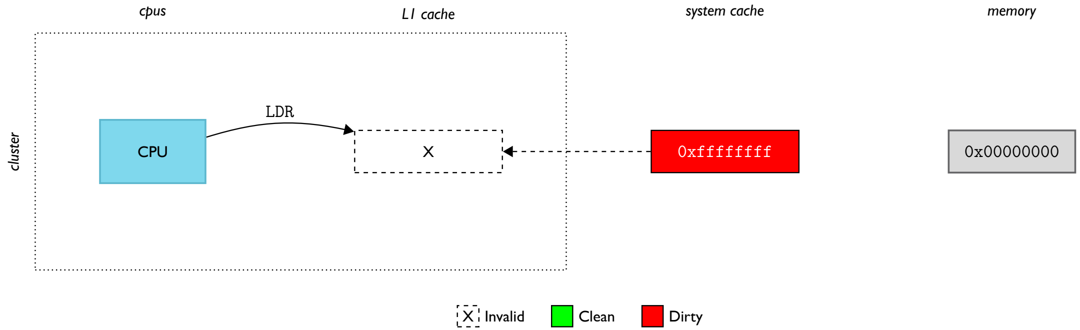
X

0xffffffff

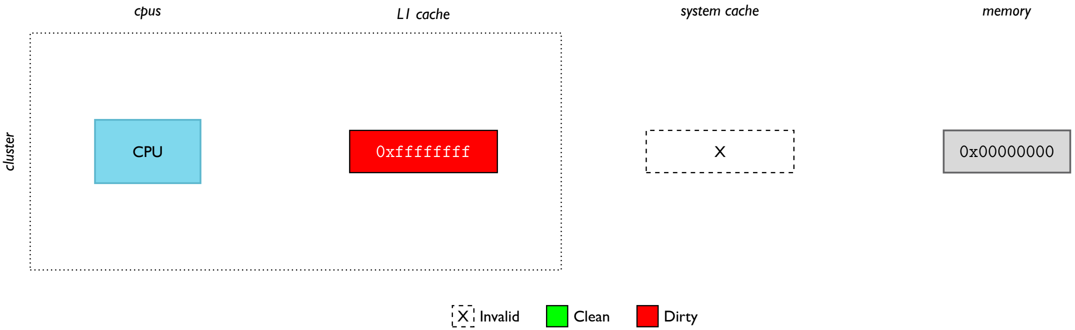
0x00000000

X Invalid 🟩 Clean 🟥 Dirty

# Set/Way maintenance: example (system caches)



# Set/Way maintenance: example (system caches)


cpus
L1 cache
system cache
memory
cluster
CPU
0xffffffff
X
0x00000000
X Invalid
🟩 Clean
🟥 Dirty