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 as 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
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*
**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
**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
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

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

Does CPU read the DMA'd data?
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
Set/Way maintenance: example (intuition)
Set/Way maintenance: example (intuition)

The diagram illustrates the process of setting and maintaining cache sets and ways.

- **CPU**: The CPU is shown at the start of the diagram, indicating the location of the data block.
- **0xffffffff**: This value represents the address of the data block, which is being accessed by the CPU.
- **X**: The symbol `X` denotes an invalid cache state, which indicates that the cache block is not present in the cache.
- **0x00000000**: This value represents the clean state of the cache block.

The process involves setting the cache block to the clean state and then invalidating it. This ensures that the data block is kept consistent across the system cache:

- **clean+invalidate**: This indicates the action performed on the cache block to set it to clean and invalidate it.
- **L1 cache**: The L1 cache is where the data block is initially located.
- **L2 cache**: The L2 cache is where the data block is eventually located.
- **memory**: The memory location where the data block is stored.
Set/way maintenance: example (intuition)

Cluster:
- CPUs
- L1 cache
- L2 cache
- Memory

Legend:
- X: Invalid
- Clean
- Dirty

CPU: 0xffffffff
Memory: 0x00000000
Set/Way maintenance: example (intuition)

CPU

0xffffffff

0x00000000

Invalid

Clean

Dirty

cluster

cpus

L1 cache

clean+invalidate

L2 cache

memory

X

ARM
Set/Way maintenance: example (intuition)

```
CPU

X
0xffffffff
Invalid
Clean
Dirty

cluster

memory

cpus
L1 cache
L2 cache

40

system cache

L2 cache

memory

ARM
```
Set/Way maintenance: example (speculation)

- CPU: 0xffffffff
- L1 cache: Invalid
- L2 cache: Clean
- Memory: 0x00000000

Legend:
- X: Invalid
- Clean
- Dirty
Set/Way maintenance: example (speculation)
Set/way maintenance: example (speculation)
Set/Way maintenance: example (speculation)

- CPUs
- L1 cache
- L2 cache
- Memory

- Cluster
- speculation

- CPU
- 0xffffffff
- 0x00000000

- X: Invalid
- Clean
- Dirty
Set/Way maintenance: example (speculation)

CPU

L1 cache

0xffffffff

L2 cache

×

memory

0x00000000

Cluster

cpus

Invalid

Clean

Dirty

L1 cache

L2 cache

memory

CPU

45

ARM
Set/Way maintenance: example (speculation)

![Diagram showing set/way maintenance example](image)
Set/Way maintenance: example (speculation)
Set/Way maintenance: example (migration)
Set/Way maintenance: example (migration)

- **CPU**: 0xffffffff
- **L1 cache**: Invalid
- **L2 cache**: Invalid
- **X**: Invalid
- **memory**: 0x00000000

Diagram:
- Cluster with CPUs
- L1 cache with 0xffffffff
- L2 cache with X
- Specification: Invalid, Clean, Dirty

**Legend**:
- ![Invalid](image)
- ![Clean](image)
- ![Dirty](image)
Set/Way maintenance: example (migration)

Cluster: CPUs

- CPU
- CPU

L1 cache:
- 0xffffffff

L2 cache:
- 0xffffffff

Memory:
- 0x00000000

Legend:
- X: Invalid
- Clean
- Dirty
Set/Way maintenance: example (migration)

- CPU
- L1 cache
  - clean+invalidate
  - 0xffffffff
- L2 cache
  - Invalid
- Memory
  - 0x00000000

Cluster: cpus

CPU

L1 cache

CPU

L2 cache

Invalid
Clean
Dirty
Set/Way maintenance: example (migration)

![Diagram showing CPU and memory states with Set/Way maintenance example.]

- **Cluster**: Contains CPUs and memory regions.
- **Cpus**: Individual processors within the cluster.
- **L1 cache**: Local cache memory for each CPU.
- **L2 cache**: Secondary cache memory.
- **Memory**: Address space for data storage.

- **Invalid**: States where data is not valid for use.
- **Clean**: States where data is clean and ready for use.
- **Dirty**: States where data has been modified and needs to be updated.

Key States:
- **0xffffffff**: Represents a state variable.
- **0x00000000**: Represents a memory address initialized to zero.

Legend:
- **X**: Depicts invalid or dirty states.
- **Clean** and **Dirty** colors indicate data status.

This diagram illustrates how memory and cache states are managed during Set/Way maintenance, focusing on the transition and validation of data in the memory hierarchy.
Set/Way maintenance: example (migration)
Set/Way maintenance: example (migration)
Set/Way maintenance: example (migration)

- CPU
- L1 cache
- CPU
- L2 cache
- memory

Cluster cpus

LDR

0xffffffff

0x00000000

Invalid
Clean
Dirty
Set/Way maintenance: example (migration)

- **CPUs**: Cluster organization with CPU units.
- **L1 cache**: Contains the value `0xffffffff`.
- **L2 cache**: Denoted by `X`.
- **Memory**: Shows `0x00000000`.

Legend:
- `X`: Invalid
- Green: Clean
- Red: Dirty

Diagram illustrative of cache sets and ways in a cluster layout, highlighting maintenance scenarios in ARM architecture.
Set/Way maintenance: example (system caches)

- **CPU**: 0xffffffff
- **L1 cache**: 0x00000000
- **System cache**: Invalid
- **Memory**: 0x00000000

Legend:
- ✗: Invalid
- 🟢: Clean
- 🔴: Dirty

Cluster: cpus

Cluster: L1 cache

Cluster: System cache

Cluster: Memory
Set/Way maintenance: example (system caches)

- **CPU**: 0xffffffff
- **system cache**: Invalid
- **memory**: 0x00000000

Diagram showing the flow of clean+invalidate from CPU to system cache to memory.
Set/Way maintenance: example (system caches)

Cluster
cpus

CPU

L1 cache

X

system cache

0xffffffff

memory

0x00000000

×: Invalid
green: Clean
red: Dirty

memory

59
Set/Way maintenance: example (system caches)
Set/Way maintenance: example (system caches)
Set/Way maintenance: example (system caches)
Set/way maintenance: example (system caches)