Computer Organization and Architecture

CPU Caching

Dr Russ Ross
Utah Tech University—Department of Computing

Spring 2026

The memory wall

CPU pipelines can execute at GHz rates, but useful work pauses when loads miss in cache.

At 3 GHz, one cycle is about 0.33 ns. If every load waited for a slower level, the whole machine would feel much slower.

Access	Typical latency	3 GHz equivalent
L1 hit	1 ns	about 1.0 GHz
L2 hit	4 ns	about 250 MHz
L3 hit	14 ns	about 71 MHz
DRAM hit	90 ns	about 11 MHz

The equivalent-frequency column is a rough intuition tool, not a precise performance model.

The cache hierarchy exists to absorb most accesses at the top levels.

- Small, very fast private caches near each core
- Larger mid-level caches with slightly higher latency
- Last-level shared cache before DRAM
- DRAM for capacity, not speed

When the hit rate is high, average memory access time stays close to L1 behavior.

Why locality makes caching effective

Programs are not random address generators; they cluster accesses in time and space.

Temporal locality (same data reused soon)

Read-modify-write loops repeatedly hit the same line while updating values.

```
for (int i = 0; i < n; i++) {
  int old = counter[i];
  int next = old + delta;
  counter[i] = next;
}
```

`counter[i]` is read and written close together, so recently loaded lines are useful again.

Spatial locality (nearby data reused soon)

Struct fields and adjacent array elements often share a line.

```
typedef struct {
  int id;
  int credits;
  int flags;
} Student;

sum += s[k].credits;
if (s[k].flags & 1) { ... }
```

Once `s[k]` arrives, nearby fields are already in the fetched line.

Cache lines and refill path

Caches transfer data one line at a time, and misses fall back through lower levels.

A 64-byte line means one miss can serve multiple nearby loads.

Line	Address range	Typical use
line 0	`0x1000–0x103F`	`a[0]..a[15]` in a sequential loop
line 1	`0x1040–0x107F`	`a[16]..a[31]`, brought by next miss
line 2	`0x1080–0x10BF`	next chunk, often prefetched

Load at `0x1058` pulls line 1, so `0x105C`, `0x1060`, and `0x1064` are likely hits.

The lookup path is conceptually:

```
L1 → L2 → L3 → DRAM
```

- L1 hit: serve immediately
- L1 miss, L2 hit: fill L1 from L2
- L1/L2 miss, L3 hit: fill upper levels on return path
- DRAM hit: fill LLC, then upper levels

Many CPUs install returned data into L1 for the requesting core; exact inclusion policy is microarchitecture-specific.

Direct-mapped placement rule

A direct-mapped cache has exactly one destination line per memory block. This keeps hardware simple but can amplify conflicts.

For 16 KiB and 64-byte lines:

- Lines: $\frac{16384}{64} = 256$
- Offset bits: $\log_2(64) = 6$
- Index bits: $\log_2(256) = 8$
- Tag bits: $32 - 8 - 6 = 18$

Tag	Index	Offset
18 bits	8 bits	6 bits

Bit ranges are `tag[31:14]`, `index[13:6]`, `offset[5:0]`.

```
Memory block number → single cache line

0   → 0
1   → 1
...
255 → 255
256 → 0
257 → 1
```

The modulo behavior (`block mod 256`) is exactly where conflict misses originate.

Example in direct-mapped bits

The three addresses below differ in high bits, but their index bits are identical.

Example

32-bit addresses, 16 KiB cache, 64-byte lines.

Symbol	Address	Offset bits [5:0]
A	0x0000_10C0	0b000000
B	0x0000_50C0	0b000000
C	0x0000_90C0	0b000000

Direct-mapped field split: `tag[31:14] | index[13:6] | offset[5:0]`

```
A = 00000000000000001 | 00001101 | 000000
B = 00000000000000101 | 00001101 | 000000
C = 00000000000001001 | 00001101 | 000000
      tag (18 bits)      index       offset
```

All three map to index `0b00001101` (line `0x0D`), so each new one evicts the previous one.

Whiteboard walkthrough: direct-mapped thrash

Example

32-bit addresses, 16 KiB cache, 64-byte lines.

Symbol	Address	Offset bits [5:0]
A	`0x0000_10C0`	`0b000000`
B	`0x0000_50C0`	`0b000000`
C	`0x0000_90C0`	`0b000000`

Use `A, B, C, A, B, C` in the worksheet at right.

Direct-mapped extraction reminder:

```
A: t=0x00000, i=0x0D, o=0x00
B: t=0x00001, i=0x0D, o=0x00
C: t=0x00002, i=0x0D, o=0x00
```

Step	Access	line[0x0D] tag after access
1	A	
2	B	
3	C	
4	A	
5	B	
6	C	

Expected result: 6 misses, 0 hits.

Fully associative placement rule

At the other extreme, a fully associative cache lets any block occupy any line. This reduces conflict misses but increases lookup cost.

For the same cache capacity:

- Offset bits: 6
- Index bits: 0
- Tag bits: $32 - 6 = 26$

Tag	**Offset**
26 bits	6 bits

Each access compares against many tags in parallel (CAM-like behavior).

```
Example legal placements

A → line 5
B → line 201
C → line 17
```

Because there is no fixed index, A/B/C no longer force each other out solely due to address mapping.

For a 256-line cache, each lookup compares roughly:

- direct-mapped: 1 tag
- 4-way set-assoc: 4 tags
- fully associative: 256 tags

Example in fully associative

The left panel stays fixed; only interpretation changes with organization.

Example

32-bit addresses, 16 KiB cache, 64-byte lines.

Symbol	Address	Offset bits [5:0]
A	0x0000_10C0	0b000000
B	0x0000_50C0	0b000000
C	0x0000_90C0	0b000000

Fully associative field split: `tag[31:6] | offset[5:0]`

For this organization, the same sequence `A, B, C, A, B, C` behaves as:

- First `A, B, C`: compulsory misses
- Second `A, B, C`: hits (assuming no unrelated eviction)

Metric	Value	Reason
Misses	3	first touch
Hits	3	all three can coexist

A replacement policy still matters here; if unrelated blocks flood the cache, these hits can disappear.

Set-associative as the practical middle

Set associativity balances hit-rate improvement against hardware complexity.

A cache with associativity N has these endpoints:

- $N = 1$: direct-mapped
- $N =$ total lines: fully associative

Typical CPU choices are moderate values ($N = 2, 4, 8, 12, 16$ depending on level).

Each block maps to one set, then chooses one of N ways inside that set.

This gives hardware a bounded search space while still reducing pathological collisions.

For a 4-way version of our running cache:

- Total lines: 256
- Sets: $\frac{256}{4} = 64$
- Offset bits: 6
- Set bits: $\log_2(64) = 6$
- Tag bits: $32 - 6 - 6 = 20$

Tag	**Set**	**Offset**
20 bits	6 bits	6 bits

Representative pattern in real CPUs:

Level	**Common associativity**
L1D	4-way to 8-way
L2	8-way to 16-way

Example in 4-way set-associative

The same addresses still collide on set index, but multiple ways absorb that pressure.

Example

32-bit addresses, 16 KiB cache, 64-byte lines.

Symbol	Address	Offset bits [5:0]
A	0x0000_10C0	0b000000
B	0x0000_50C0	0b000000
C	0x0000_90C0	0b000000

4-way split: `tag[31:12] | set[11:6] | offset[5:0]`

All three addresses map to the same set, with different tags.

```
set[0x03]
+--------+--------+
| way 0  | way 1  |
+--------+--------+
| way 2  | way 3  |
+--------+--------+
```

`A`, `B`, and `C` fit simultaneously, so the second half of `A,B,C,A,B,C` hits.

After access	Ways occupied in set 0x03	Hit/miss
A	1/4	miss
B	2/4	miss
C	3/4	miss

Same sequence, three organizations

Holding the access pattern fixed isolates the effect of organization.

Organization	**Misses**	**Hits**	**Conflict pressure**	**Hardware cost**
Direct-mapped	6	0	high	low
Fully associative	3	3	very low	high
4-way set-assoc	3	3	low	moderate

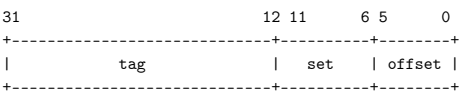
Set-associative designs are widely used because they preserve most of the hit-rate benefit without fully associative lookup cost.

Reading address fields like hardware

The address decoder always performs field extraction before tag comparison.

For 4-way set-associative in this lecture:

- `offset[5:0]` selects byte position in line
- `set[11:6]` selects one set
- `tag[31:12]` is compared against tags in all ways of that set



Example with `0x1234_56A8`:

- offset = lower 6 bits
- set = next 6 bits
- tag = upper 20 bits

This logic is simple in concept but heavily optimized in physical layout and timing paths.

Miss classes and performance cliffs

Different misses have different causes, so mitigations differ too.

Miss class	**Cause**
Compulsory	first touch
Capacity	working set $>$ cache
Conflict	mapping collisions

Conflict misses are exactly what associativity is designed to reduce.

As working set or stride increases, throughput often drops in steps:

- L1-dominated region
- L2-dominated region
- L3-dominated region
- DRAM-dominated region

This step pattern is the main intuition behind the memory-mountain benchmark.

Loop order and locality

Two loops can do identical arithmetic but induce very different memory behavior.

Row-major traversal aligns with contiguous memory in C-like layouts.

```
for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
    sum += a[i][j];
  }
}
```

This preserves spatial locality and helps hardware prefetchers.

Column-wise traversal creates large strides on row-major data.

```
for (int j = 0; j < cols; j++) {
  for (int i = 0; i < rows; i++) {
    sum += a[i][j];
  }
}
```

This tends to increase miss rate and memory-level stalls.

Representative cache structures (recent mainstream hardware)

Modern chips vary in details, but the hierarchy pattern is remarkably consistent.

Hardware	**L1**	**L2**	**L3 / LLC**	**Comment**
Intel Core i9-14900K	split L1 per core	2 MB per P-core, 4 MB per E-cluster	36 MB shared	desktop hybrid core design
AMD Ryzen 9 9950X	80 KB split L1 per core	1 MB per core	64 MB shared	Zen 5 desktop
NVIDIA H100	up to 256 KB/SM L1+shared	—	50 MB L2	GPU memory hierarchy

Private upper levels hide latency; shared lower levels reduce DRAM traffic and provide inter-core data sharing.

Typical line sizes and associativity in practice

The specific parameters differ by vendor and level, but a few trends are common.

- 64-byte lines remain common in general-purpose CPUs.
- L1D is often around 4-way to 8-way.
- L2/L3 are often more associative than L1.
- True LRU is rare in large caches; pseudo-LRU variants are common.

When you benchmark real code, these parameters surface as hit-rate and bandwidth inflection points.

Why writes need separate policies

Reads are straightforward, but writes must define both update timing and allocation behavior.

Two questions define write behavior:

1. On a write hit, do we update lower memory immediately?
2. On a write miss, do we bring the line into cache first?

Those choices produce the familiar policy pairs.

Common pairings in real systems:

- write-through + no-write-allocate (often in simpler designs)
- write-back + write-allocate (common in high-performance cores)

Coherence, bandwidth, and verification complexity all influence the choice.

Write-through vs write-back

Both policies are correct; they optimize for different system goals.

Policy	Behavior	Typical tradeoff
Write-through	update cache and lower level on each write	simpler state, higher bandwidth use
Write-back	mark line dirty, defer lower-level write until eviction	lower bandwidth, more control complexity

If a loop writes the same hot line 100 times:

- write-through may send about 100 lower-level writes
- write-back often sends about 1 write on eviction

That bandwidth difference is why write-back is common in performance-oriented CPUs.

Write-allocate vs no-write-allocate

Allocation policy on a write miss determines whether future writes can hit locally.

`Write-allocate` pulls the line into cache, then writes it.

This is usually better when software will reuse nearby data or repeatedly update the same line.

`No-write-allocate` (write-around) writes straight to lower level.

This is often better for streaming output that is unlikely to be read again soon.

Key points to carry forward

Caching decisions are design tradeoffs, and software behavior determines how those tradeoffs play out.

- Locality is the fundamental reason caches work.
- Address fields (`tag`, `set/index`, `offset`) define lookup behavior.
- Increasing associativity reduces conflicts but raises hardware cost.
- Set-associative caches are usually the practical sweet spot.
- Write policy choices strongly affect bandwidth and coherence complexity.

Quick check questions

Use these to close the lecture or open discussion.

1. Why can two arrays far apart in address space still conflict in a direct-mapped cache?
2. If line size doubles, which address field must grow first?
3. When is no-write-allocate better than write-allocate?
4. Why does fully associative lookup generally cost more energy than set-associative lookup?