Caching
A cache is a block of memory for temporary storage of data likely to be used again. The CPU and hard drive frequently use a cache, as do web browsers.

Caches are usually formed from SRAM and are located very close to the CPU to provide maximum operational speed.
Caching was invented to solve a significant problem. In the early decades of computing, main memory was extremely slow and incredibly expensive — but CPUs weren’t particularly fast, either. Starting in the 1980s, the gap began to widen very quickly. Microprocessor clock speeds took off, but memory access times improved far less dramatically. As this gap grew, it became increasingly clear that a new type of fast memory was needed to bridge the gap.

![Graph: Processor vs Memory Performance]

- **1980:** no cache in microprocessor;
- **1995:** 2-level cache
The goal of the cache system is to ensure that the CPU has the next bit of data it will need already loaded into cache by the time it goes looking for it (also called a cache hit). A cache miss, on the other hand, means the CPU has to go scampering off to find the data elsewhere. This is where the L2 cache comes into play — while it’s slower, it’s also much larger. Some processors use an inclusive cache design (meaning data stored in the L1 cache is also duplicated in the L2 cache) while others are exclusive (meaning the two caches never share data). If data can’t be found in the L2 cache, the CPU continues down the chain to L3 (typically still on-die), then L4 (if it exists) and main memory (DRAM).
Definitions

**Set** – A block of data in a cache composed of the tag plus the cache line.

**Way** – An indication of the associativity of the system.

**Associative** – a memory where each address is mapped to a certain set of cache locations.

**Cache line** – An amount of data returned that may be from one or more addresses

**Write-through** - Data written by the CPU to memory or to disk is also written into the cache. Write performance is not improved with this method. However, if a subsequent read operation needs that same data, read performance is improved, because the data are already in the high-speed cache.

**Write-back** - A caching method in which modifications to data in the cache aren't copied to the cache source until absolutely necessary. Write-back caching yields somewhat better performance than write-through caching because it reduces the number of write operations to main memory.
Definitions

**Direct Mapped Cache**: The simplest way to allocate the cache to the system memory is to determine how many cache lines there are (16,384 in our example) and just chop the system memory into the same number of chunks. Then each chunk gets the use of one cache line. This is called direct mapping. So if we have 64 MB of main memory addresses, each cache line would be shared by 4,096 memory addresses (64 M divided by 16 K).

**Fully Associative Cache**: Instead of hard-allocating cache lines to particular memory locations, it is possible to design the cache so that any line can store the contents of any memory location. This is called fully associative mapping.

**N-Way Set Associative Cache**: "N" here is a number, typically 2, 4, 8 etc. This is a compromise between the direct mapped and fully associative designs. In this case the cache is broken into sets where each set contains "N" cache lines, let's say 4. Then, each memory address is assigned a set, and can be cached in any one of those 4 locations within the set that it is assigned to. In other words, within each set the cache is associative, and thus the name. This design means that there are "N" possible places that a given memory location may be in the cache. The tradeoff is that there are "N" times as many memory locations competing for the same "N" lines in the set. Let's suppose in our example that we are using a 4-way set associative cache. So instead of a single block of 16,384 lines, we have 4,096 sets with 4 lines in each. Each of these sets is shared by 16,384 memory addresses (64 M divided by 4 K) instead of 4,096 addresses as in the case of the direct mapped cache. So there is more to share (4 lines instead of 1) but more addresses sharing it (16,384 instead of 4,096).
Every CPU contains a specific type of RAM called tag RAM. The **tag RAM is a record of all the memory locations that can map to any given block of cache**. If a cache is fully associative, it means that any block of RAM data can be stored in any block of cache. The advantage of such a system is that the hit rate is very high, but the search time is extremely long — the CPU has to look through its entire cache to find out if the data is present before searching main memory.

At the opposite end of the spectrum we have direct-mapped caches. A **direct-mapped cache is a cache where each cache block can contain one and only one block of main memory**. This type of cache can be searched extremely quickly, but since it maps 1:1 to memory locations, it has a low hit rate. In between these two extremes are **n-way associative caches**. A 2-way associative cache (Piledriver’s L1 is 2-way) means that each main memory block can map to one of two cache blocks. An eight-way associative cache means that each block of main memory could be in one of eight cache blocks.

---

**Graph: Hit Rate v. Associativity & Cache Size**

- **X-axis:** Cache Size (1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1M)
- **Y-axis:** Hit Rate (0 to 100)
- **Legend:**
  - 1
  - 2
  - 4
  - 8
  - 16

*(L1 cache, Running GCC)*
Definitions

cache coherence is the uniformity of shared resource data that ends up stored in multiple local caches.

Cache Coherency phases:

1. Request - During the request phase the agent performing a read or write drives request and address pins to indicate what data it will be accessing.

2. Snoop - The snoop phase gives any agent sharing the bus the chance to check whether its own cache is affected. The address from the request phase must be compared with all the cache tags to determine if the data being accessed is in the cache. If so, the snooping processor can provide the data itself. If not, during the response phase the intended receiver can signal that it is ready to send or receive data.

3. Response - The responder might signal that it must defer the transaction. It does not have the data needed for a read or is not ready to accept a write. The responder will create a new transaction at a later time to satisfy this request.

4. Data - If the responder can handle the transaction, then during the data phase, the actual data is exchanged over the bus. Even when processors are idle and not executing any instructions, they must continue to snoop the bus to keep all the caches synchronized.
Between the CPU and the main memory DRAM, a fast cache SRAM is provided. It holds the frequently accessed data and delivers it very quickly. The process is controlled by a cache controller which may implement various write strategies, such as write-through or write-back.
(a) Fully-associative cache. Any memory block can map to any cache line.

(b) Direct-mapped cache. All addresses mod C map to the same cache line.

(c) Two-wayset-associative cache. A memory block maps into a specific line in either set.
Memory Address, Cache Entry and 4-Way Cache Directory

32-bit Memory Address

- A31
- A16
- A15
- A12
- A11
- A4
- A3
- A0

Tag Address: (30-bits)
Set Address: (4-bits)
Byte Addr.: (4-bits)

Cache Directory Entry

- Address
- 20-bit Tag Address
- 1 Write-protect Bit
- 1 Bit “valid”
- LRU

3 Bits

Cache Memory Entry

- Data: 16-byte Cache Line
- DW: 3 Bits

Set

Cache Directory

- Way 0
- Way 1
- Way 2
- Way 3
- LRU

Set Address: A4-A11

- Set 0
- Set 1
- Set 247
- Set 248
- Set 249
- Set 250
- Set 251
- Set 252
- Set 253
- Set 254
- Set 255

- 20 Bits
- 1 1
Determining Cache Hits
Associative Cache Organization
This chart is useful because it actually illustrates the performance impact of adding a huge (128MB) L4 cache as well as the conventional L1/L2/L3 structures. Each stair step represents a new level of cache. The red line is the chip with an L4 — note that for large file sizes, it’s still almost twice as fast as the other two Intel chips.

It might seem logical, then, to devote huge amounts of on-die resources to cache — but it turns out there’s a diminishing marginal return to doing so. Larger caches are both slower and more expensive. At six transistors per bit of SRAM (6T), cache is also expensive (in terms of die size, and therefore dollar cost). Past a certain point, it makes more sense to spend the chip’s power budget and transistor count on more execution units, better branch prediction, or additional cores.
Cache Consistency by Inclusion

All addresses in the on-chip CPU cache are also present in the L2-cache.
Cache Replacement policies

- Least recently used (LRU)
- Most recently used (MRU)
- First-in-first-out
- Random
MESI

* The **MESI protocol** is a widely used cache coherency and memory coherence protocol, which was later introduced by Intel in the Pentium processor to "*support the more efficient write-back cache in addition to the write-through cache previously used by the Intel 486 processor*".

Every cache line is marked with one of the four following states (coded in two additional *bits*):

**M** - Modified: The cache line is present only in the current cache, and is *dirty*; it has been modified from the value in *main memory*. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state.

**E** - Exclusive: The cache line is present only in the current cache, but is *clean*; it matches main memory.

**S** - Shared: Indicates that this cache line may be stored in other caches of the machine.

**I** - Invalid: Indicates that this cache line is invalid.

* From Wikipedia
MESI Rules

A cache may satisfy a read from any state except Invalid. An Invalid line must be fetched (to the Shared or Exclusive states) to satisfy a read.

A write may only be performed if the cache line is in the Modified or Exclusive state. If it is in the Shared state, all other cached copies must be invalidated first. This is typically done by a broadcast operation known as Read For Ownership (RFO).

A cache may discard a non-Modified line at any time, changing to the Invalid state. A Modified line must be written back first.

A cache that holds a line in the Modified state must snoop (intercept) all attempted reads (from all of the other CPUs in the system) of the corresponding main memory location and insert the data that it holds. This is typically done by forcing the read to back off (i.e. to abort the memory bus transaction), then writing the data to main memory and changing the cache line to the Shared state.

A cache that holds a line in the Shared state must also snoop all invalidate broadcasts from other CPUs, and discard the line (by moving it into Invalid state) on a match.

A cache that holds a line in the Exclusive state must also snoop all read transactions from all other CPUs, and move the line to Shared state on a match.

The Modified and Exclusive states are always precise: i.e. they match the true cacheline ownership situation in the system. The Shared state may be imprecise: if another CPU discards a Shared line, and this CPU becomes the sole owner of that cacheline, the line will not be promoted to Exclusive state (because broadcasting all cacheline replacements from all CPUs is not practical over a broadcast snoop bus).

In that sense the Exclusive state is an opportunistic optimization: If the CPU wants to modify a cache line that is in state S, a bus transaction is necessary to invalidate all other cached copies. State E enables modifying a cache line with no bus transaction.

* From Wikipedia
# The Four MESI States

<table>
<thead>
<tr>
<th>MESI State</th>
<th>Cache Line Valid?</th>
<th>Values in Memory are</th>
<th>Is There a Copy in Another Cache?</th>
<th>Write Access Refers to&lt;sup&gt;1)&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Yes</td>
<td>Invalid</td>
<td>No</td>
<td>Cache</td>
</tr>
<tr>
<td>E</td>
<td>Yes</td>
<td>Valid</td>
<td>No</td>
<td>Cache</td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
<td>Valid</td>
<td>Possibly</td>
<td>Cache/Memory Subsystems</td>
</tr>
<tr>
<td>I</td>
<td>No</td>
<td>Unknown</td>
<td>Possibly</td>
<td>Memory Subsystem</td>
</tr>
</tbody>
</table>

<sup>1)</sup> Memory subsystem means main memory or LS-cache.
Transitions of the MESI States Into Another State
Update Policies – write through

- Ensures consistency between cache contents and main memory contents at all times
- Write traffic can dominate performance
- Write buffers are used to hide the latency of memory writes by overlapping writes with useful work
Update Policies – write back

- Write operations take place in the cache and modified cache lines are marked
  - Modified or “dirty” cache lines written back on replacement
- Locality of writes impacts memory traffic
- Writes occur at the speed of a cache
- Complexity of cache management is increased
- Cache may be inconsistent with main memory
ECE-684
Cache Performance
Memory Hierarchy example:

- **L0**: CPU registers hold words retrieved from cache memory.
- **L1**: on-chip L1 cache (SRAM) holds cache lines retrieved from the L2 cache.
- **L2**: off-chip L2 cache (SRAM) holds cache lines retrieved from memory.
- **L3**: main memory (DRAM) holds disk blocks retrieved from local disks.
- **L4**: local secondary storage (local disks) holds files retrieved from disks on remote network servers.
- **L5**: remote secondary storage (distributed file systems, Web servers) holds files retrieved from disks on remote network servers.

**Storage Devices**

- **L0**: Smaller, faster, and costlier (per byte) storage devices
- **L1**: Larger, slower, and cheaper (per byte) storage devices
Examples of caching in the hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What Cached</th>
<th>Where Cached</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-byte word</td>
<td>CPU registers</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32-byte block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>32-byte block</td>
<td>Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware+OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
Memory Hierarchy

Temporal Locality
- Keep recently referenced items at higher levels
- Future references satisfied quickly

Spatial Locality
- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly
Locality of reference

Principle of Locality:

– Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.

– **Temporal locality:** Recently referenced items are likely to be referenced in the near future.

– **Spatial locality:** Items with nearby addresses tend to be referenced close together in time.

Locality Example:

• **Data**
  – Reference array elements in succession (spatial)

• **Instructions**
  – Reference instructions in sequence (spatial)
  – Cycle through loop repeatedly (temporal)

```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```
Back to caches:

Cache Misses and Performance

- Miss penalty operations:
  - Detect miss: 1 or more cycles
  - Find victim (replace line): 1 or more cycles
    • Write back if dirty
  - Request line from next level: several cycles
  - Transfer line from next level: several cycles
    • (block size) / (bus width)
  - Fill line into data array, update tag array: 1+ cycles
  - Resume execution

- In practice: varies from 6 cycles to 100s of cycles
Cache Miss Rate

• Determined by:
  – Program characteristics
    • Temporal locality
    • Spatial locality
  – Cache organization
    • Block size, associativity, number of sets
The performance impact of adding a CPU cache is directly related to its efficiency or hit rate; repeated cache misses can have a catastrophic impact on CPU performance. The following example is vastly simplified but should serve to illustrate the point.

Imagine that a CPU has to load data from the L1 cache 100 times in a row. The L1 cache has a 1ns access latency and a 100% hit rate. It therefore takes our CPU 100 nanoseconds to perform this operation.

Now, assume the cache has a 99% hit rate, but the data the CPU actually needs for its 100th access is sitting in L2, with a 10-cycle (10ns) access latency. That means it takes the CPU 99 nanoseconds to perform the first 99 reads and 10 nanoseconds to perform the 100th. A 1% reduction in hit rate has just slowed the CPU down by 10%.

In the real world, an L1 cache typically has a hit rate between 95% and 97%, but the performance impact of those two values in our simple example isn’t 2% — it’s 14%. Keep in mind, we’re assuming the missed data is always sitting in the L2 cache. If the data has been evicted from the cache and is sitting in main memory, with an access latency of 80-120ns, the performance difference between a 95% and 97% hit rate could nearly double the total time needed to execute the code.
Improving Locality

• Instruction placement
  – Profile program, place unreferenced or rarely referenced paths “elsewhere” [done by compiler]
    • Maximize temporal locality
  – Eliminate taken branches
    • Fall-through path has spatial locality
Improving Locality

- Data placement, access order
  - Arrays: “block” loops to access subarray that fits into cache
    - Maximize temporal locality
  - Structures: pack commonly-accessed fields together
    - Maximize spatial, temporal locality
  - Trees, linked lists: allocate in usual reference order
    - Heap manager usually allocates sequential addresses
    - Maximize spatial locality

- Hard problem, not easy to automate:
  - C/C++ disallows rearranging structure fields
  - OK in Java
Cache Miss Rates: Caused by the 3 C’s

- **Compulsory miss**
  - First-ever reference to a given block of memory

- **Capacity**
  - Working set exceeds cache capacity
  - Useful blocks (with future references) displaced

- **Conflict**
  - Placement restrictions (not fully-associative) cause useful blocks to be displaced
  - Think of as capacity within set
Improving Miss Rates:

- Number of blocks (sets x associativity)
  - Bigger is better: fewer conflicts, greater capacity
- Associativity
  - Higher associativity reduces conflicts
  - Very little benefit beyond 8-way set-associative
- Block size
  - Larger blocks exploit spatial locality
  - Usually: miss rates improve until 64B-256B
  - 512B or more miss rates get worse
    - Larger blocks less efficient: more capacity misses
    - Fewer placement choices: more conflict misses
Miss rate versus cache size using SPEC – CINT2000.
Improving Miss Rates: victim cache

- Reduce the conflict misses of a direct mapped cache
  - Effective backup
- Capture a slightly larger cache footprint
  - Capture recent discards
Cache Miss Rate

- Subtle tradeoffs between cache organization parameters
  - Large blocks reduce compulsory misses but increase miss penalty
    - \#compulsory = \frac{\text{working set}}{\text{block size}}
    - \#transfers = \frac{\text{block size}}{\text{bus width}}
  - Large blocks increase conflict misses
    - \#blocks = \frac{\text{cache size}}{\text{block size}}
  - Associativity reduces conflict misses
  - Associativity increases access time
- Q2 - Can associative cache ever have higher miss rate than direct-mapped cache of same size?
Cache Miss Rates: 3 C’s

- Vary size and associativity
  - Compulsory misses are constant
  - Capacity and conflict misses are reduced
Cache Miss Rates: 3 C’s

- Vary size and block size
  - Compulsory misses drop with increased block size
  - Capacity and conflict can increase with larger blocks
Cache Misses and Performance

- How does this affect performance?
- Performance = Time / Program

\[
\text{Performance} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Cache organization affects cycle time
  - Hit latency
- Cache misses affect \text{CPI}
Cache Misses and CPI

\[
CPI = \frac{cycles_{\text{inst}}}{cycles_{hit}} + \frac{cycles_{\text{miss}}}{cycles_{\text{inst}}}
\]

\[
= \frac{cycles_{hit}}{cycles_{\text{inst}}} + \frac{cycles_{\text{miss}}}{cycles_{\text{inst}}} \times Miss_{\text{rate}}
\]

- Cycles spent handling misses are strictly additive.
- Miss_penalty is recursively defined at next level of cache hierarchy as weighted sum of hit latency and miss latency.
Cache Misses and CPI

\[ CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l \]

- \( P_l \) is miss penalty at each of \( n \) levels of cache
- \( MPI_l \) is miss rate per instruction at each of \( n \) levels of cache
- Miss rate specification:
  - Per instruction: easy to incorporate in CPI
  - Per reference: must convert to per instruction
    - Local: misses per local reference
    - Global: misses per ifetch or load or store
Cache Performance Example

• Assume following:
  – L1 instruction cache with 98% per instruction hit rate
  – L1 data cache with 96% per instruction hit rate
  – Shared L2 cache with 40% local miss rate
  – L1 miss penalty of 8 cycles
  – L2 miss penalty of:
    • 10 cycles latency to request word from memory
    • 2 cycles per 16B bus transfer, 4x16B = 64B block transferred
    • Hence 8 cycles transfer plus 1 cycle to fill L2
    • Total penalty 10+8+1 = 19 cycles
Cache Performance Example

\[
CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l
\]

\[
CPI = 1.15 + \frac{8 \text{cycles}}{\text{miss}} \times \left( \frac{0.02 \text{miss}}{\text{inst}} + \frac{0.04 \text{miss}}{\text{inst}} \right)
\]

\[
+ \frac{19 \text{cycles}}{\text{miss}} \times \frac{0.40 \text{miss}}{\text{ref}} \times \frac{0.06 \text{ref}}{\text{inst}}
\]

\[
= 1.15 + 0.48 + \frac{19 \text{cycles}}{\text{miss}} \times \frac{0.024 \text{miss}}{\text{inst}}
\]

\[
= 1.15 + 0.48 + 0.456 = 2.086
\]
Cache Misses and Performance

• CPI equation
  – Only holds for misses that cannot be overlapped with other activity
  – Store misses often overlapped
    • Place store in store queue
    • Wait for miss to complete
    • Perform store
    • Allow subsequent instructions to continue in parallel
  – Modern out-of-order processors also do this for loads
    • Cache performance modeling requires detailed modeling of entire processor core
L3 Cache

At its simplest level, an **L3 cache is just a larger, slower version of the L2 cache.** Back when most chips were single-core processors, this was generally true. The first L3 caches were actually built on the motherboard itself, connected to the CPU via the backside bus. When AMD launched its K6-III processor family, many existing K6/K-2 motherboards could accept a K6-III as well. Typically these boards had 512K-2MB of L2 cache — when a K6-III, with its integrated L2 cache was inserted, these slower, motherboard-based caches became L3 instead.

By the turn of the century, slapping an additional L3 cache on a chip had become an easy way to improve performance — Intel’s first consumer-oriented Pentium 4 “Extreme Edition” was a repurposed Gallatin Xeon with a 2MB L3 on-die. Adding that cache was sufficient to buy the Pentium 4 EE a 10-20% performance boost over the standard Northwood line.

**As multicore processors became more common, L3 cache started appearing more frequently on consumer hardware.** These new chips, like Intel’s Nehalem and AMD’s K10 (Barcelona) used L3 as more than just a larger, slower backstop for L2. In addition to this function, the **L3 cache is often shared between all of the processors on a single piece of silicon.** That’s in contrast to the L1 and L2 caches, both of which tend to be private and dedicated to the needs of each particular core. (AMD’s Bulldozer design is an exception to this — Bulldozer, Piledriver, and Steamroller all share a common L1 instruction cache between the two cores in each module).

**Intel’s latest Haswell-E, for example, has eight separate cores that all back up to a common L3 cache.**
Caches Summary

• Four parameters
  – Placement
    • Direct-mapped, set-associative, fully-associative
  – Identification
    • Tag array used for tag check
  – Replacement
    • LRU, FIFO, Random
  – Write policy
    • Write-through, writeback
Cache structure and design are still being fine-tuned as researchers look for ways to squeeze higher performance out of smaller caches.

There’s an old rule of thumb that we add roughly one level of cache every 10 years, and it appears to be holding true into the modern era — Intel’s Haswell and Broadwell chips offer certain SKUs with an enormous L4, thereby continuing the trend.

It’s an open question at this point whether AMD will ever go down this path. The company’s emphasis on HSA and shared execution resources appears to be taking it along a different route, and AMD chips don’t currently command the kind of premiums that would justify the expense.

Regardless, cache design, power consumption, and performance will be critical to the performance of future processors, and substantive improvements to current designs could boost the status of whichever company can implement them.
Caches Summary

\[
CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l
\]

- **Hit latency**
  - Block size, associativity, number of blocks
- **Miss penalty**
  - Overhead, fetch latency, transfer, fill
- **Miss rate**
  - 3 C’s: compulsory, capacity, conflict
  - Determined by locality, cache organization