Caches

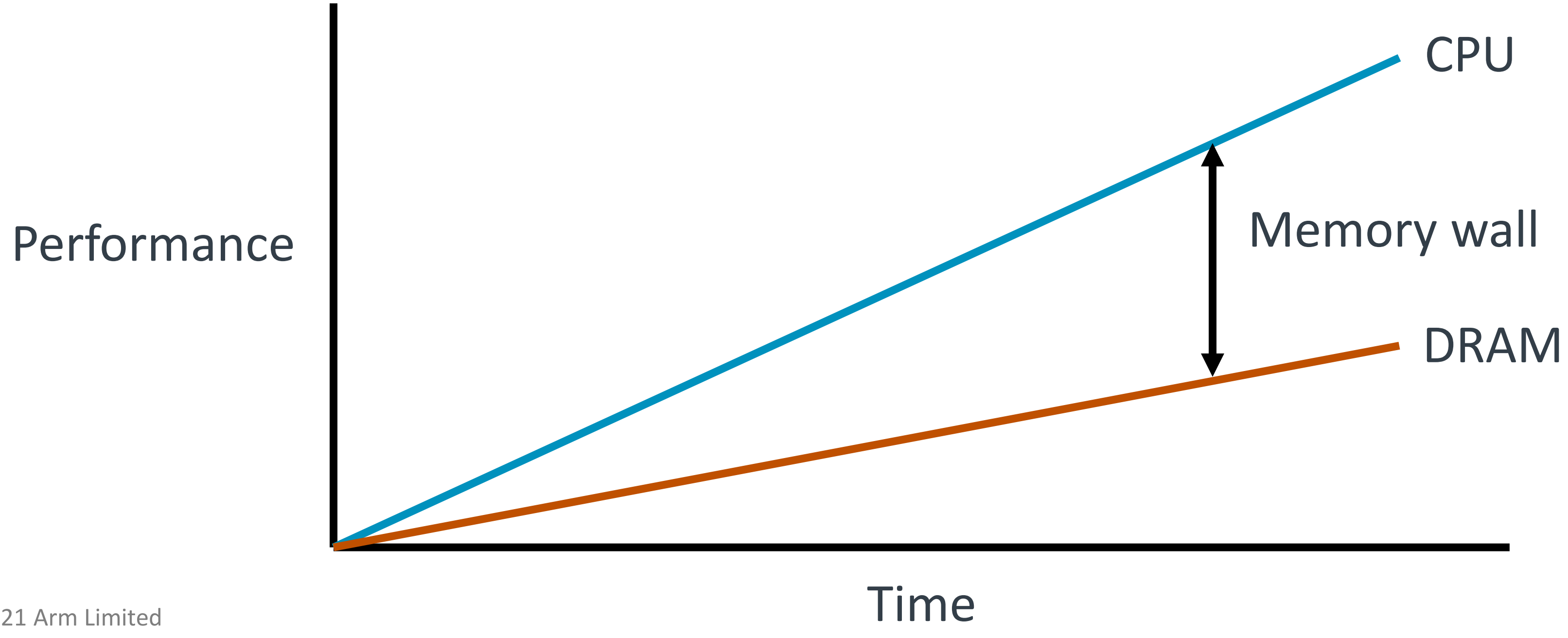
Module 7

Module Syllabus

- Why do we need caches?
- Cache designs
  - Direct-mapped cache
  - Set-associative cache
  - Fully associative cache
- Cache policies
- Multi-level caches
- Cache performance
  - Reducing cache misses
- Case study

Motivation – Why Do We Need Caches?

- For CPUs to reach maximum performance, they need fast access to memory.
  - Both to read instructions and to read and write data
- However, historically, processor clock speeds have increased far faster than in dynamic RAM (DRAM).
  - In addition to the differences in the speed of the underlying process technologies



Motivation – Why Do We Need Caches?

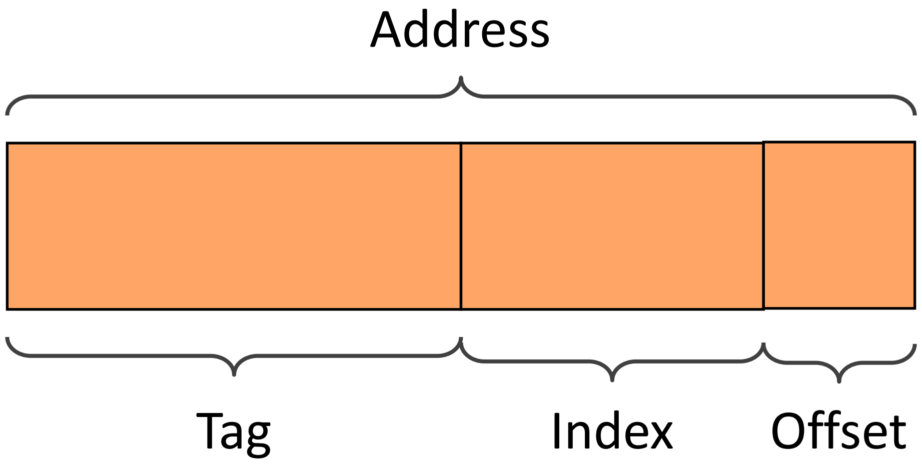
- Fortunately, most programs don’t need access to all memory all of the time.
    - Accesses tend to exhibit locality of reference.
    - Temporal locality – if an address is accessed, it is likely to be accessed again soon.
    - Spatial locality – if an address is accessed, its neighbors are likely to be accessed soon.
    - Therefore, only a small number of addresses are likely to be accessed in the near future.
- Small memories are quick to access and can be placed near to the CPU.
    - If we can identify these locations likely to be accessed soon, then we can keep them in these memories.
- A cache stores copies of some memory locations for fast access when required.

Cache Entries

- The cache operates as follows:
  - Whenever the CPU needs to read from a location residing in the main memory, it first checks the cache for any matching entries.
  - If the location exists in the cache, it is simply returned directly to the CPU; this is known as a cache hit.
  - If the location doesn't exist in the cache, also known as a cache miss, the cache allocates a new entry for the location, copies the contents from the main memory, and then fulfills the request from the contents in the cache.
- If the CPU needs to write some data, then it also checks the cache first and writes on a hit.
  - What happens on a miss is governed by the cache's polices, described in later slides.
- The proportion of accesses that result in hits, as opposed to misses, is known as the hit rate and is a useful measure of the effectiveness of the cache.
- The cache stores data in blocks to take advantage of spatial locality.
  - For example, a block may be 32B or 64B long whereas each data item is typically only 8B in size.

Accessing the Cache

- To identify whether data are in a cache, we need an identifier to map to a cache block.
  - The data's address is easily used for this.
- We split the address into separate parts.
  - Tag – the unique identifier for the data, compared to tags stored within the cache
  - Index – used to select the cache blocks to do the tag comparison with
  - Offset – position of the data within the cache block



Cache Designs

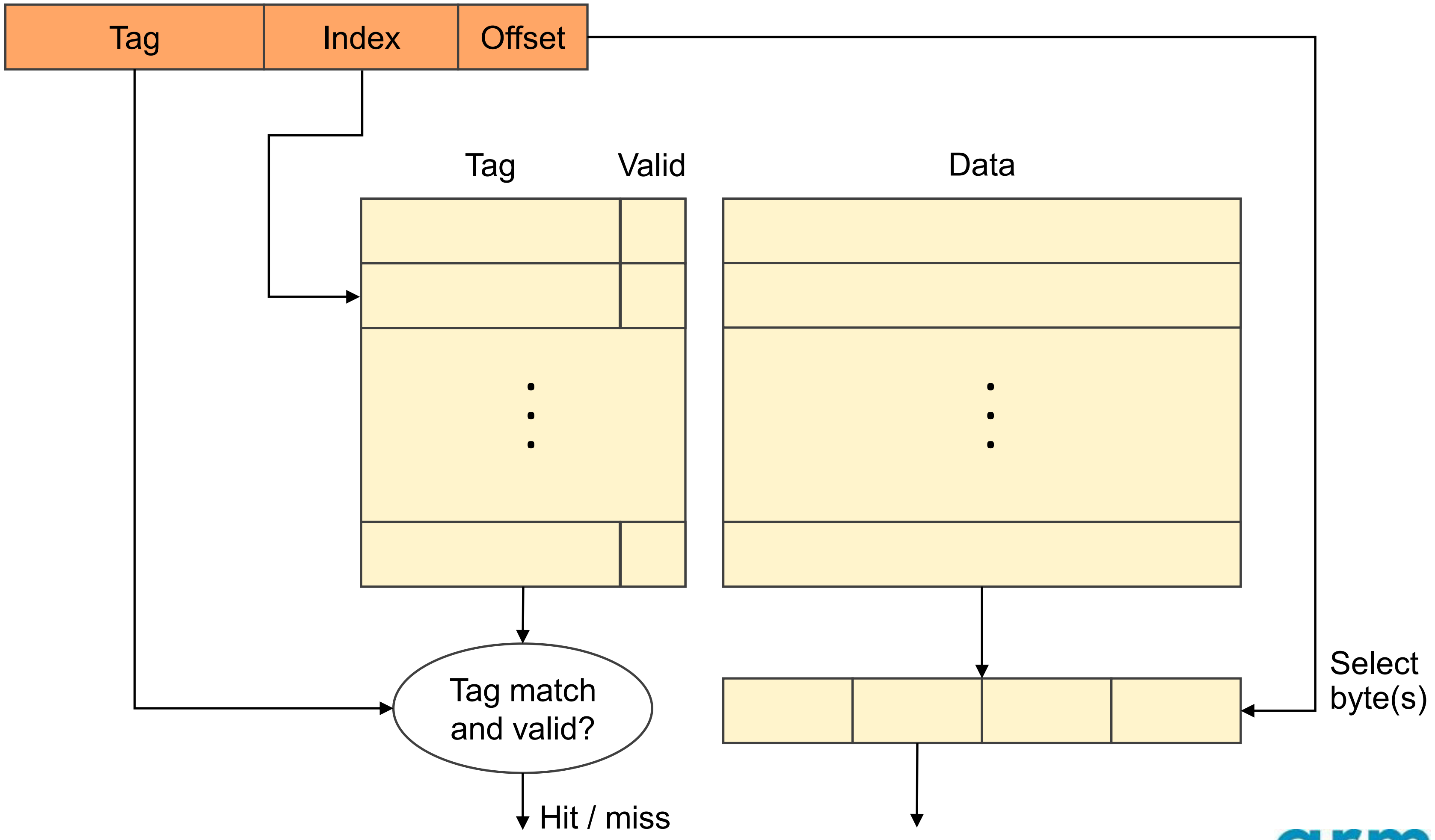
Direct-mapped Cache

- A simple design because each memory location only maps to one cache block
- However, this often leads to contention.
  - When several locations with the same cache index are repeatedly accessed

- Pros:
  - Simple design, therefore inexpensive
  - Quick to search
- Cons:
  - Low hit rate when there is contention

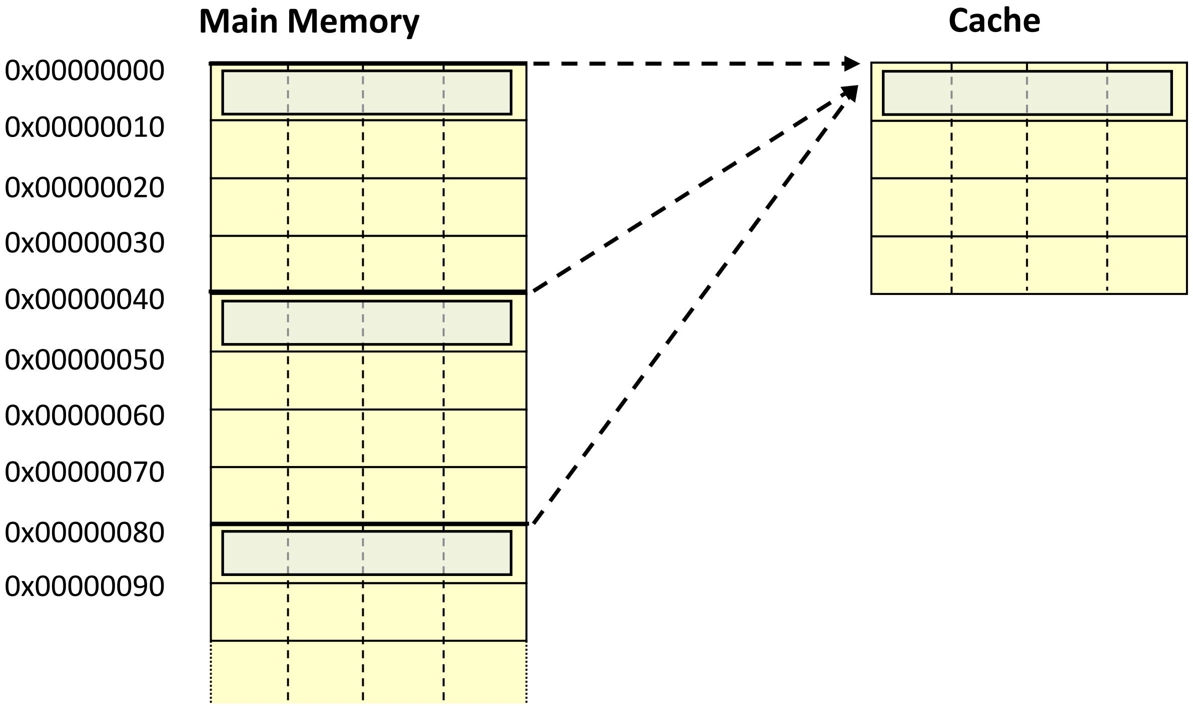
Direct-mapped Cache

- Index used to select a single cache block
- Tags compared
  - If valid and tags match, then hit
- On hit, offset chooses starting byte from data array.



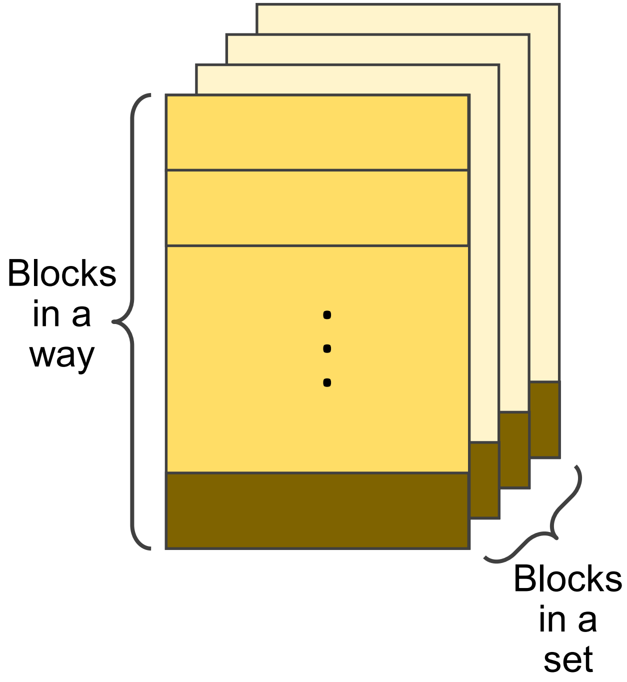
Direct-mapped Cache

- The downside is that each address in memory has only one location in the cache that it maps to.
- Multiple memory locations could contend for the same cache line.



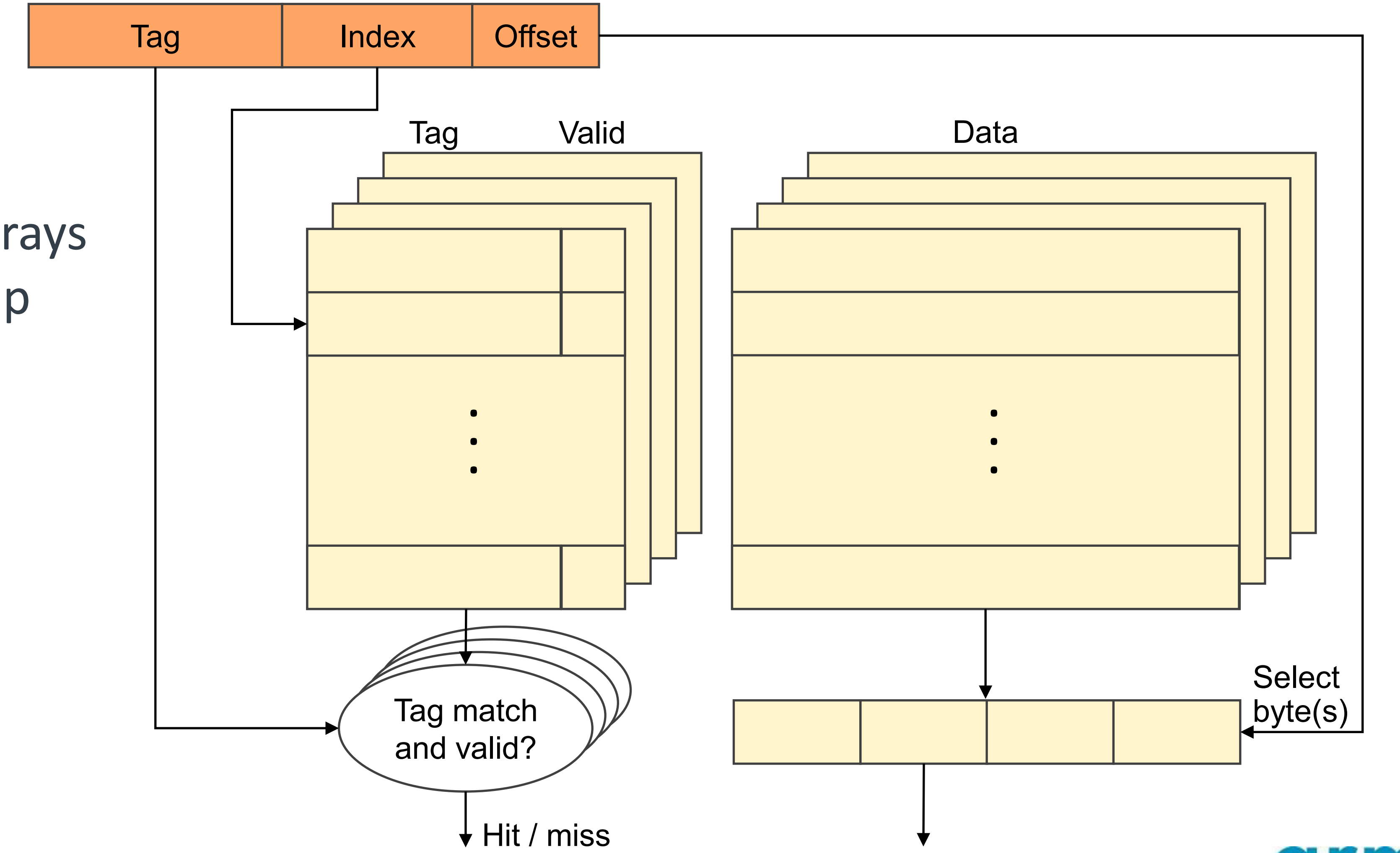
Set-associative Cache

- Each memory location maps to N cache blocks.
  - Each group of N cache blocks is called a set – hence, N-way set associative.
  - A group of cache blocks in the same array but with different indices is called a way.
- Improved hit rate compared to a direct-mapped cache
  - Less contention when there are several memory locations with the same cache index
- Pros:
  - Combines the speed of a direct-mapped cache with improved flexibility in placement
- Cons:
  - Finding blocks within a set requires more complex hardware.



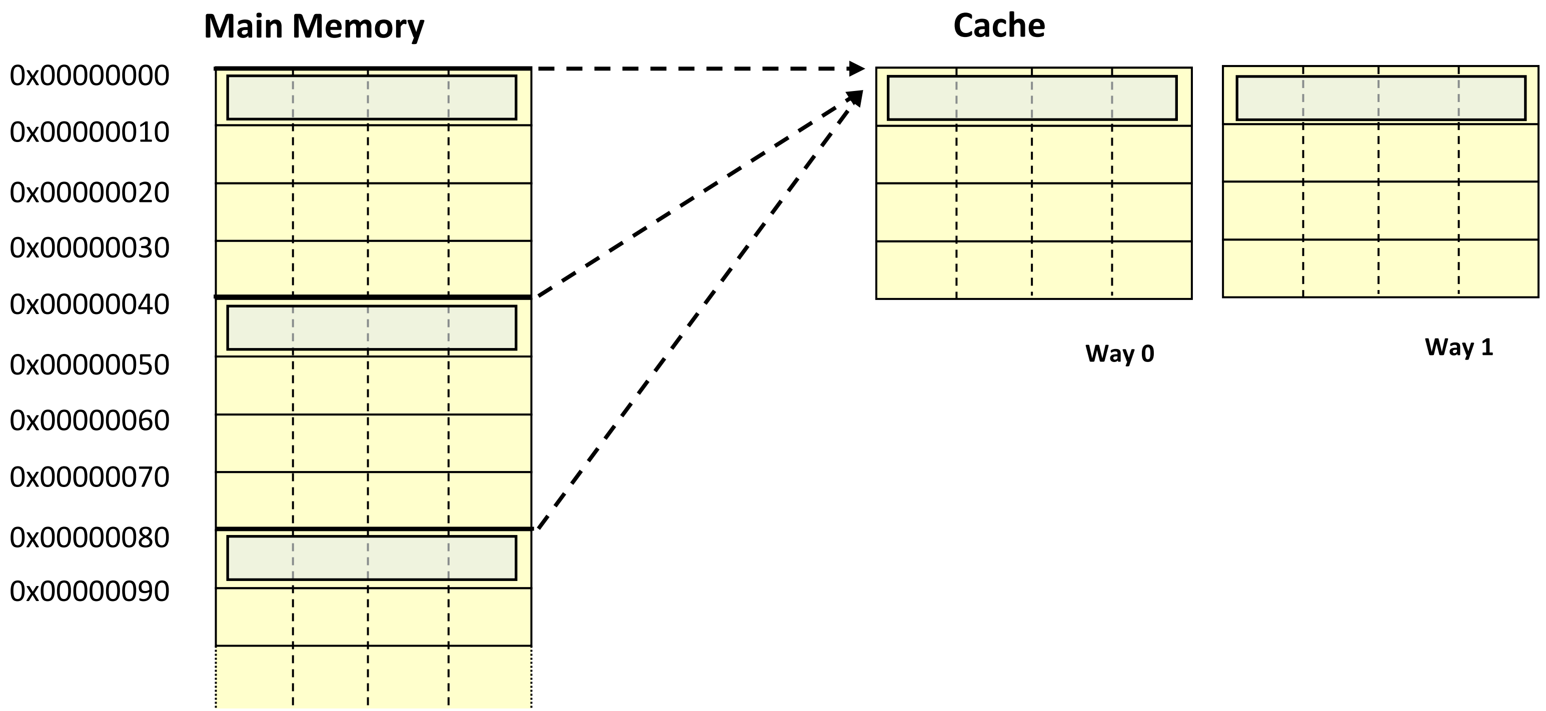
Set-associative Cache

- Instead of just one tag and data array, there are multiple.
- The address is split as before, but multiple arrays in parallel are looked up and have their tags checked.



Set-associative Cache

- Now, each address in memory maps to multiple cache locations (ways), reducing contention.



Fully Associative Cache

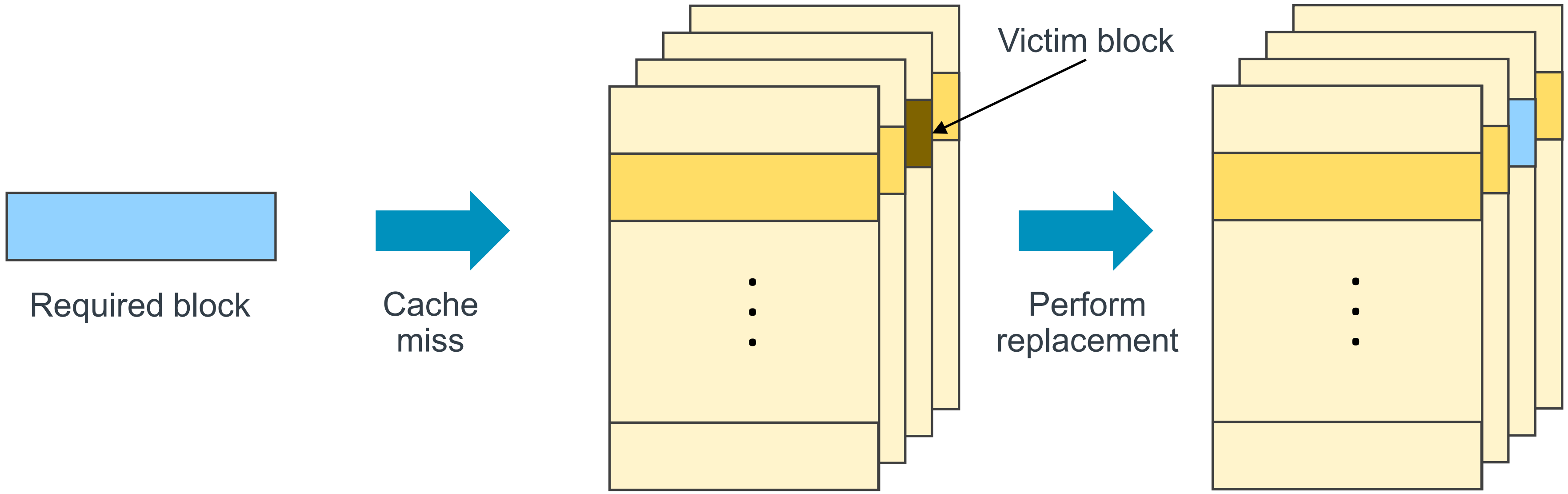
- In a fully associative cache, a block can be placed in any available location in the cache.
- This makes the cache more flexible, increasing the hit rate.
- But it is also more complex, as searching for a block involves comparing against all blocks.

- Pros:
  - Good flexibility: greater hit rate
- Cons:
  - Searching for a match can be expensive, power-hungry, and slow.

Cache Policies

Replacement Policies

- On a cache miss, a new memory location that isn't already in the cache is accessed.
- We generally want to cache this new block, to take advantage of its locality.
- Therefore, we need to choose a block within the cache to replace, called the victim.
- The replacement policy determines which block we choose.



Replacement Policies

By cache type

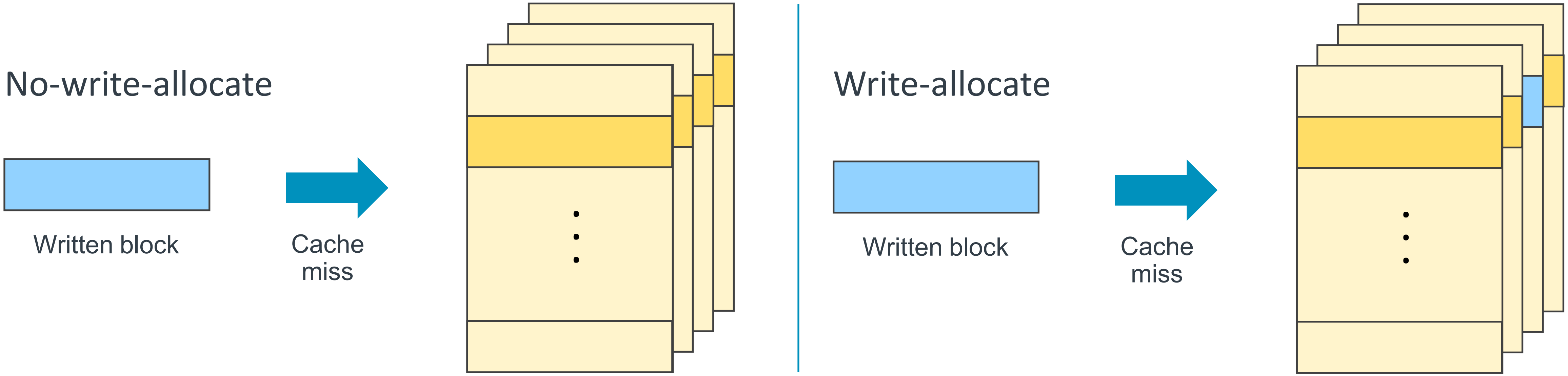
- In a direct-mapped cache, there is only one possible victim for each block.
- Fully associative cache blocks only need to be replaced when the cache is full, and then they need to choose a victim.
- Set-associative cache blocks need to use a policy to choose and replace a victim when the set is full.

Associative policies

- Round robin (first in, first out)
  - Cycle round the ways in a set
  - Simple, but doesn’t maximize locality
- Least-recently used (LRU)
  - Track order blocks have been accessed.
  - Requires extra logic, usually pseudo-LRU used
- Random
  - Simple to implement

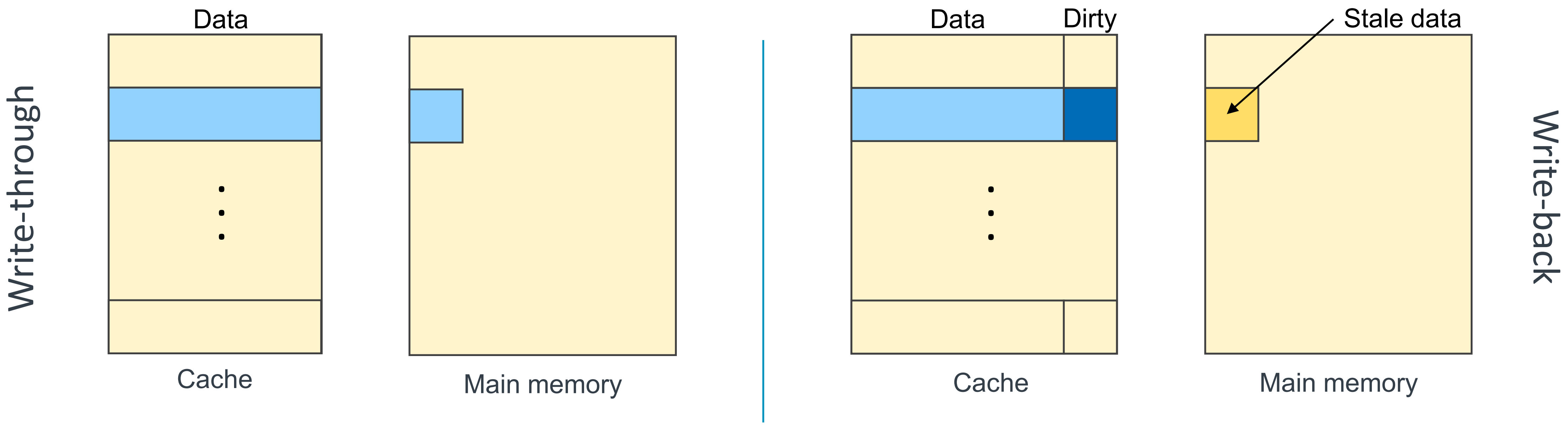
Cache Policies

- Other policies determine the operation of the cache.
- The allocation policy controls when new data are loaded into the cache.
  - A no-write-allocate policy only allocates new data on a read miss.
  - A write-allocate policy also allocates on a write miss.



Cache Policies

- The cache write policy controls what happens when a write operation hits in the cache.
  - A write-through cache updates external memory in parallel with itself.
  - A write-back cache does not update external memory until it is required to and marks modified blocks as dirty.
    - For example, on eviction of the written-to cache block

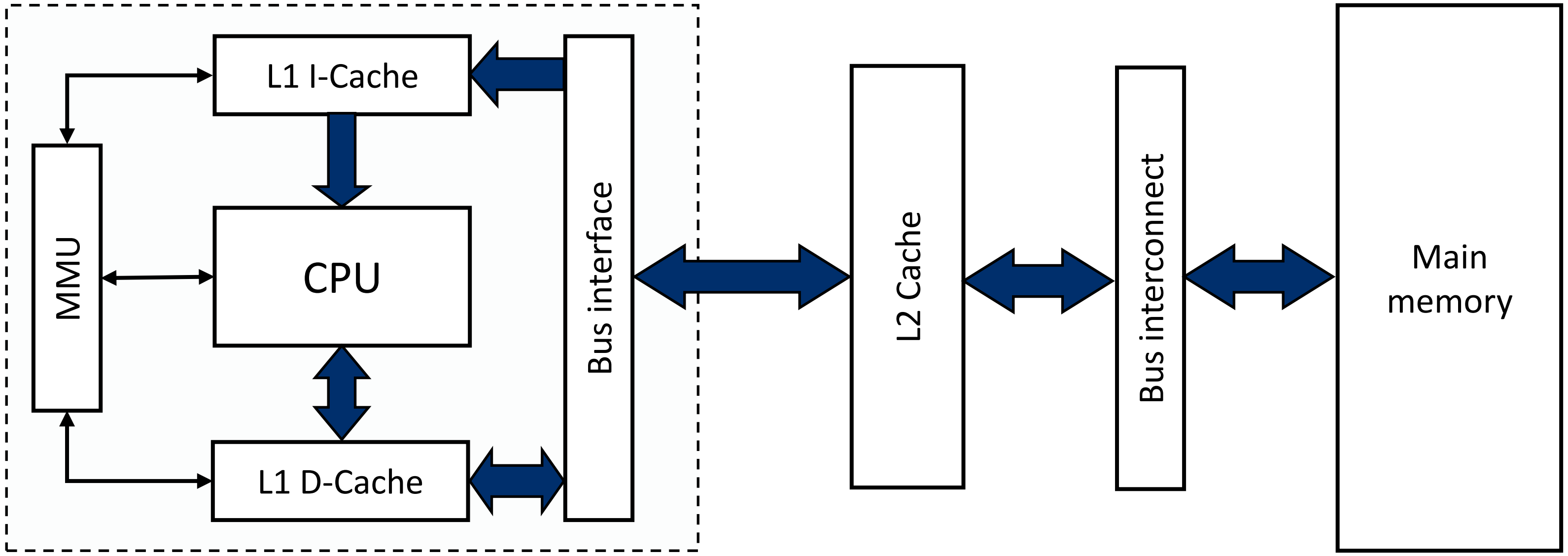


Multi-level Caches

Multi-level Caches

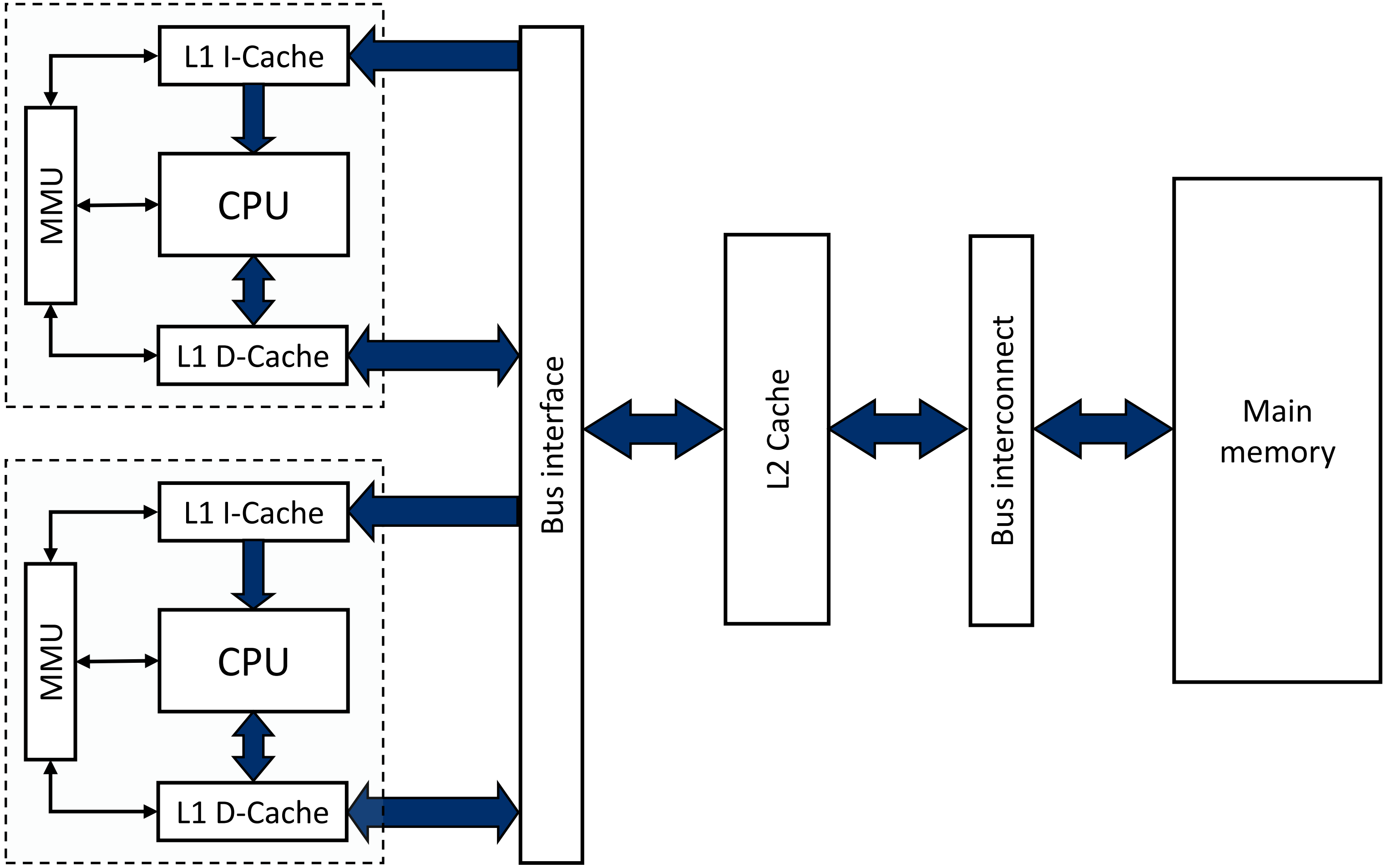
Modern systems provide support for multi-level caches.

- L1 caches smaller and closer to CPU
  - Usually integrated into the processor
  - Usually separate data and instruction caches
- L2 caches larger but further from CPU
  - On the same die
  - Usually unified instruction and data



Cache Sharing

- Caches can also be shared by several processors in the system.
  - L1 caches are typically private.
  - Sharing often occurs at L2 or L3 caches.
  - If at L3, both L1 and L2 caches are private.



Cache Performance Metrics

Cache Performance Terminology

Reducing cache misses to improve performance

Cache hits and misses

- If the data to be accessed are present in the cache, it is called a hit; otherwise, it is a miss.
- If a cache miss occurs, a block of data containing the requested data are copied into the cache.

Cache miss rate and miss penalty

- The miss rate is the fraction of cache misses in relation to the total number of memory accesses.
- The miss penalty is the amount of extra time taken to load the requested data.

Breaking Down Cache Misses

The three categories of cache miss

Compulsory miss

- The memory location being accessed has never existed in the cache; the first access to any new block generates a compulsory miss.

Capacity miss

- There is not enough space in the cache to hold all the data required; therefore, some of it must be evicted and reloaded, and this generates capacity misses.

Conflict miss

- Too many memory locations map to the same set, so some blocks have to be evicted and reloaded; this generates conflict misses.
- Conflict misses only occur in direct-mapped and set-associative caches.

Cache Performance

- Cache hit and miss rates give an indication of cache performance.
  - But they fail to capture the impact of the cache on the overall system.
- We therefore prefer to incorporate timing into the cache performance.
  - For example, including the time taken to access the cache
  - And the time taken to service a miss
- This can give us a value for the average memory access time (AMAT).
- First, we need to define the metrics that we will use.

Cache Performance

Metrics useful for measuring performance

Memory

- Memory cycle time
  - Minimum delay between two memory accesses
- Memory access time
  - Elapsed time between the start and finish of one memory access

Bandwidth

- The data throughput
  - Bit rate * number of bits

Cache

- Cache hit time
  - Elapsed time between starting access to the cache and retrieving the data
- Cache miss penalty
  - Time required to retrieve data from memory on a cache miss

Cache Performance

- From the CPU's point of view, we want to reduce the average memory access time (AMAT).
  - This is the average time it takes to load data.
  - Including a cache in the system should lead to reducing AMAT; otherwise, it is doing more harm than good!

- AMAT = Cache hit time + Cache miss rate * Cache miss penalty

- For example:
  - An L1 cache with 1 ns hit latency and 5% miss rate
  - Combined with an L2 cache with a 10 ns hit latency and 1% miss rate
  - And 100 ns main memory latency

- AMAT = 1 + 0.05 * (10 + 0.01 * 100) = 6.5 ns

Techniques for Reducing Cache Misses

Larger cache blocks

- Pros:
  - Better spatial locality
  - Reduces number of tags
- Cons:
  - Increases miss penalty
  - Increases capacity misses
  - Increases conflict misses

Bigger caches

- Pros:
  - Reduces capacity misses
- Cons:
  - Increases hit time
  - More expensive
  - Consumes more power

Higher associativity

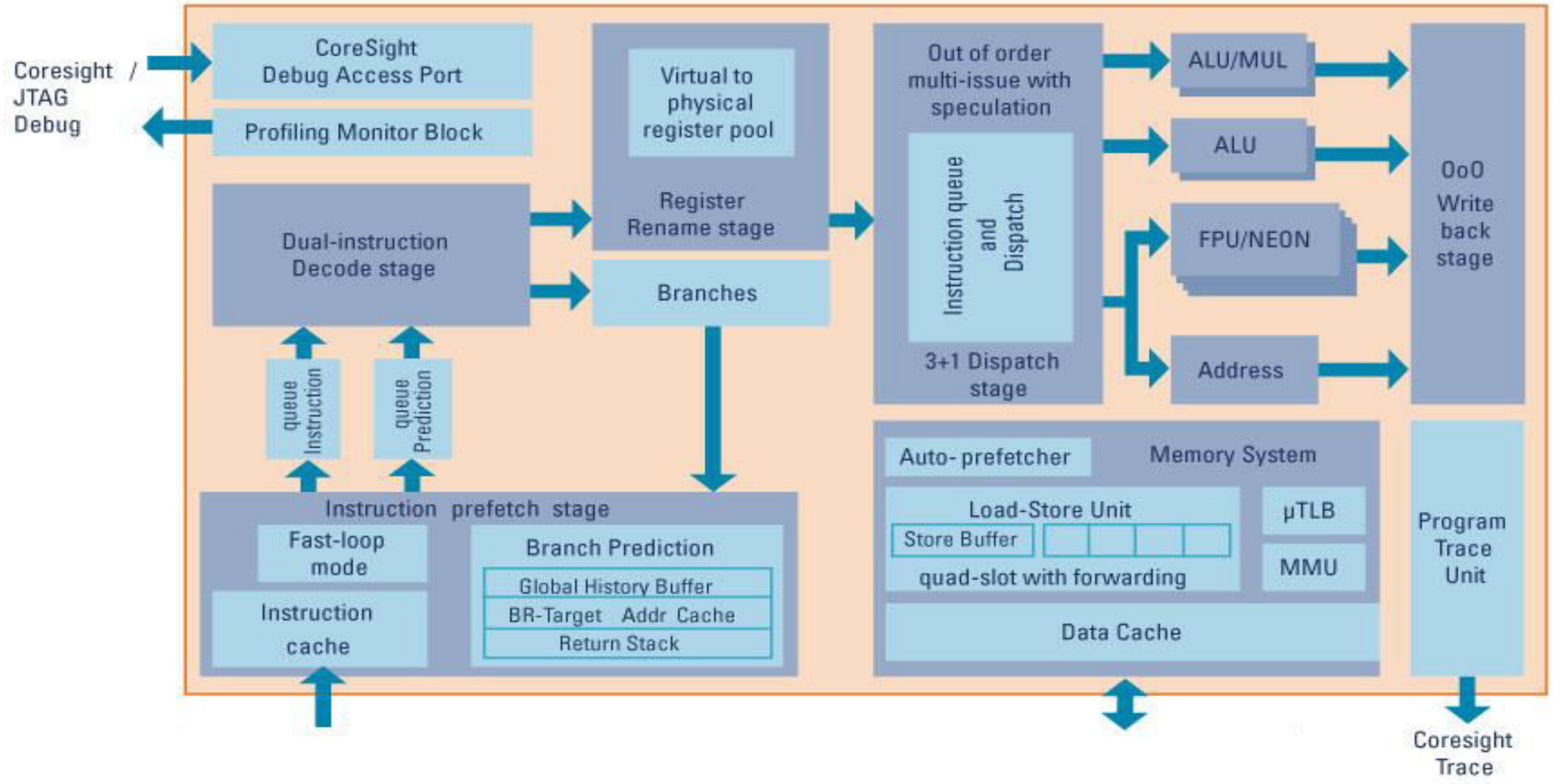
- Pros:
  - Reduces conflict misses
- Cons:
  - Increases hit time
  - Consumes more power

Speculative Prefetch

- Many CPUs support speculative data prefetching to L1 caches.
  - Monitors for sequential access and automatically requests subsequent blocks
    - E.g., access to 0x8000, 0x8100, 0x8200 will result in prefetch of 0x8300
  - More advanced prefetching schemes are also possible.
- Some L2 caches also have speculative prefetch.
  - Separate from L1 cache prefetch
  - Will fetch instructions or data (since the L2 cache is unified)
- ISAs often include instructions for software to perform data prefetching, too.
- In addition, some CPUs will detect memset-like operations (e.g., zeroing out memory) and change the cache policy from write-allocate to read allocate.
  - Avoids polluting cache (with lines of zeroes, for example)

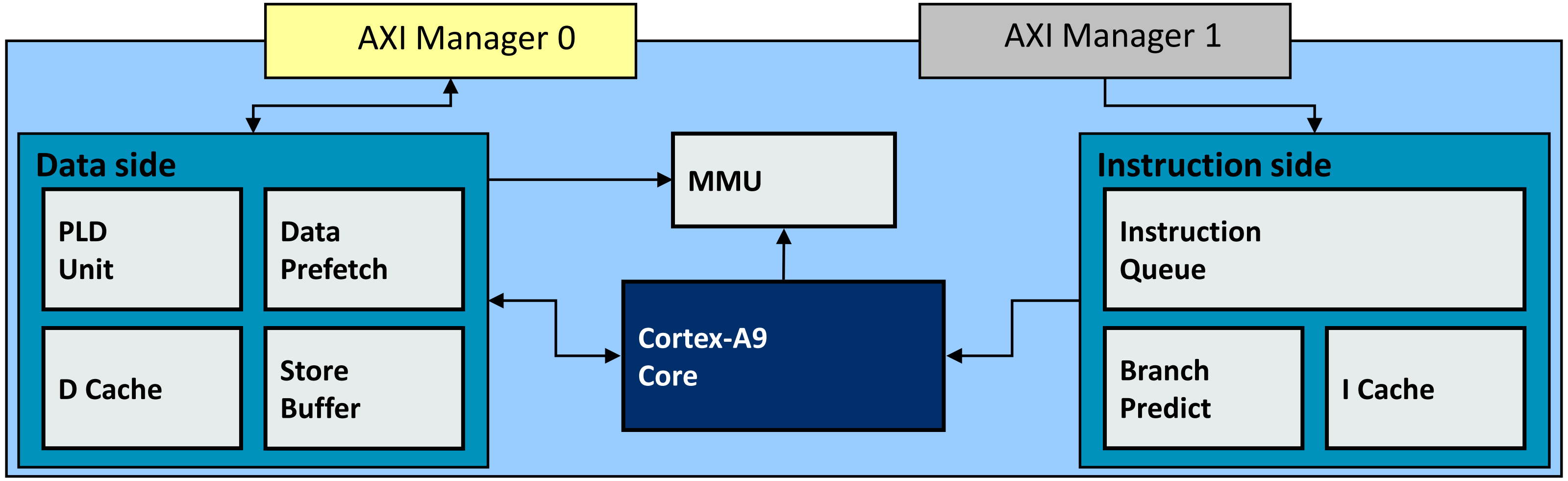
Case Study

Case Study: Cortex-A9



Case Study: Cortex-A9

- 16-64KiB, 4-way set-associative caches
- Block size of 32B
- Caches closely coupled with parts of the core
  - Store buffer for data
  - Branch predictor for instructions



Conclusions

- CPU performance historically outpaced main memory, leading to a memory wall.
- Caches provide a solution, but storing copies of recently used data near the CPU
- Caches have many different parameters.
  - Direct-mapped or set-associative or fully associative
  - Different policies for choosing a victim on a miss when replacing data
  - Write through to the next level of memory or write-back only when required
  - Allocate a cache block on a write miss or only on a read miss
- Multi-level caches provide a hierarchy of small-yet-fast caches, with large-yet-slow ones.
- Caches can be shared by cores or other caches within this hierarchy.