

CSE 321a

Computer Organization (1)

تنظيم الحاسبات (1)



3rd year, Computer Engineering
Fall 2016

Lecture #6



Dr. Hazem Ibrahim Shehata

Dept. of Computer & Systems Engineering

Credits to Dr. Ahmed Abdul-Monem Ahmed for the slides

Administrivia

- Assignment #1:
 - Solution was posted last week.
- Assignment #2:
 - To be released later on this week.

Website: <http://hshehata.github.io/courses/zu/cse321a>

Office hours: Sunday 12:00pm-1:00pm

Chapter 4. Cache Memory (*Cont.*)

Cache Memory – Design

1. Mapping function
2. Replacement algorithm
3. Read/Write policies
4. Number of caches
5. Addresses
6. Size
7. Block/line size

2. Replacement Algorithms

Direct mapping

- No choice.
- Each block only maps to one line.
- Replace that line.

2. Replacement Algorithms

Associative and Set Associative

- Hardware implemented algorithms!
- Most common ones:
 1. Least Recently Used (LRU)
 - Replace the block that has not been recently accessed.
 2. First In First Out (FIFO)
 - Replace the oldest block.
 3. Least Frequently Used
 - Replace the block which has had the fewest hits.
 4. Random
 - Replace any block!

Cache Memory – Design

1. Mapping function
2. Replacement algorithm
3. Read/Write policies
4. Number of caches
5. Addresses
6. Size
7. Block/line size

3. Read/Write Policies

Read Policy (Hit & Miss)

- Reads dominate processor cache accesses.
 - All instructions need to be fetched → reads.
 - Some instructions may read one or more operands.
 - Most instructions do not write to memory!
- What if CPU wants to read a word from MM?
 - Read-hit policy**: if read hits in cache (i.e., container block is in cache), **just read word from cache!**
 - Read-miss policy**: if read misses in cache (i.e., container block is in not cache),
 1. **Read-through**: read word from MM, and bring container block to cache.
 2. **No-read-through**: bring container block to cache, and then read word from cache.

3. Read/Write Policies

Write Policy (Hit)

- What if CPU wants to write a word to memory?
 - **Write-hit policy**: if write hits (i.e., container block is in cache)
 1. **Write-through**: write word to both cache and MM
 - + **Pros**: easier to implement, MM is always consistent with cache, read miss never results in writes to MM.
 - + **Cons**: Every write needs a MM access, slower writes, more MM bandwidth consumption.
 2. **Write-back**: write word to cache only, do not write word to MM until cached version of container block is replaced.
 - + In addition to tag bits, each cache line stores a “dirty bit”.
 - + **Pros**: multiple writes within block require only one write to MM, faster writes, less MM bandwidth consumption.
 - + **Cons**: harder to implement, MM is not always consistent with cache, reads may cause writes of dirty blocks to MM.

3. Read/Write Policies

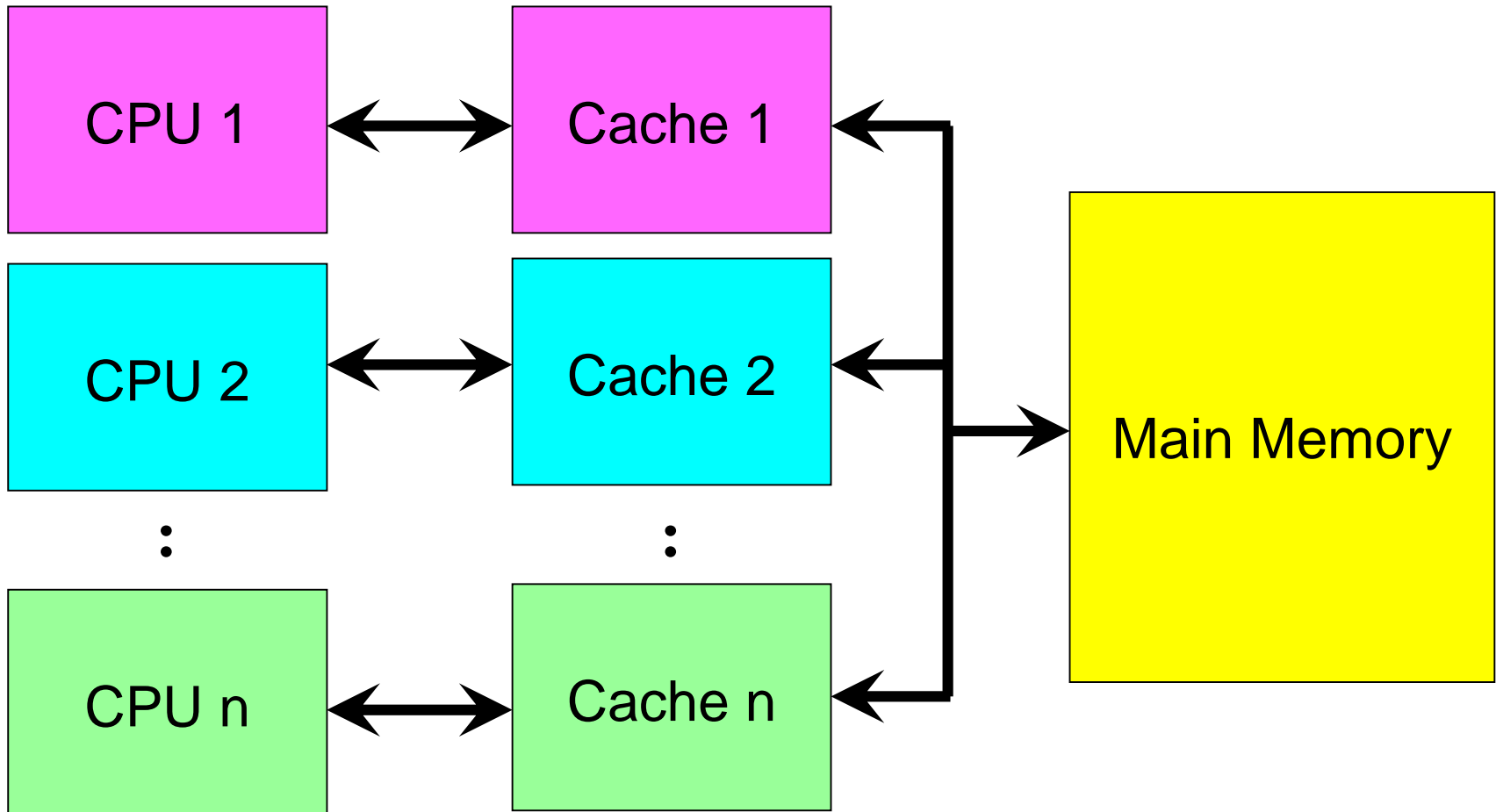
Write Policy (Miss)

- ... (Cont.)
 - **Write-miss policy**: if write misses (i.e., container block is in not cache)
 1. **Write-allocate**: bring container block to cache, and then act according to write-hit policy.
 2. **No-write-allocate**: write word to container block in MM, and do not bring block to cache.
- **Note**: either write-miss policy could be used with write-through or write-back, however:
 - Write-back caches generally use write-allocate
 - Hoping that subsequent writes to block get captured by cache
 - Write-through caches often use no-write-allocate
 - since subsequent writes block will still have to go to MM

3. Read/Write Policy

Cache Coherency ترابط او التهام w/t Multi-CPUs

- Consider a multiple-CPU organization:
 - Several CPUs share same MM
 - Each CPU has its own cache.



3. Read/Write Policy

Cache Coherency ترابط او التهام w/t Multi-CPU's

- Problem: If data is altered in one cache → this invalidates this word in MM and in other caches.
- This is true even if write-through is used.
- **Cache coherency**: maintaining data in all caches consistent.
- Some solutions:
 1. **Bus watching with write-through**: Cache controller monitors the bus and invalidates the word in the cache if another master (e.g., CPU) updates it in MM.
 2. **Hardware transparency**: Additional h/w makes sure that if one CPU updates a word in its cache, it is written to MM and updated in other caches.
 3. **Non-cacheable memory**: Only a portion of MM is shared, and that portion is non-cacheable!

Cache Memory – Design

1. Mapping function
2. Replacement algorithm
3. Read/Write policies
4. Number of caches
5. Addresses
6. Size
7. Block/line size

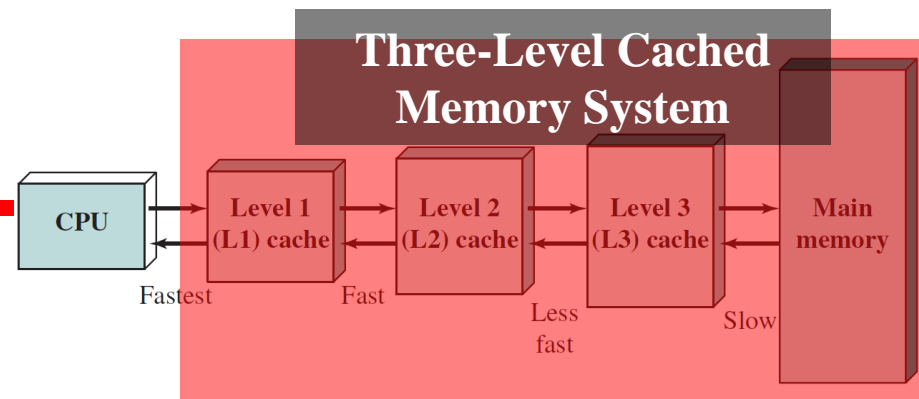
4. Number of Caches

Split vs. Unified & Dedicated vs. Shared

- Contemporary multi-core systems use **multiple caches** organized in a variety of ways:
 1. **Split cache vs. unified cache**
 - Split: one cache for instructions and another for data.
 - + No contention between fetch unit & execute unit.
 - Unified: only one cache for both.
 - + Higher hit rate than split since it balances load between instruction and data fetches.
 2. **Dedicated cache vs. shared cache**
 - Dedicated: one cache for each CPU
 - + No contention between CPUs
 - Shared: one cache shared by multiple CPUs
 - + Better load balancing

4. Number of Caches

Multi-Level Caches



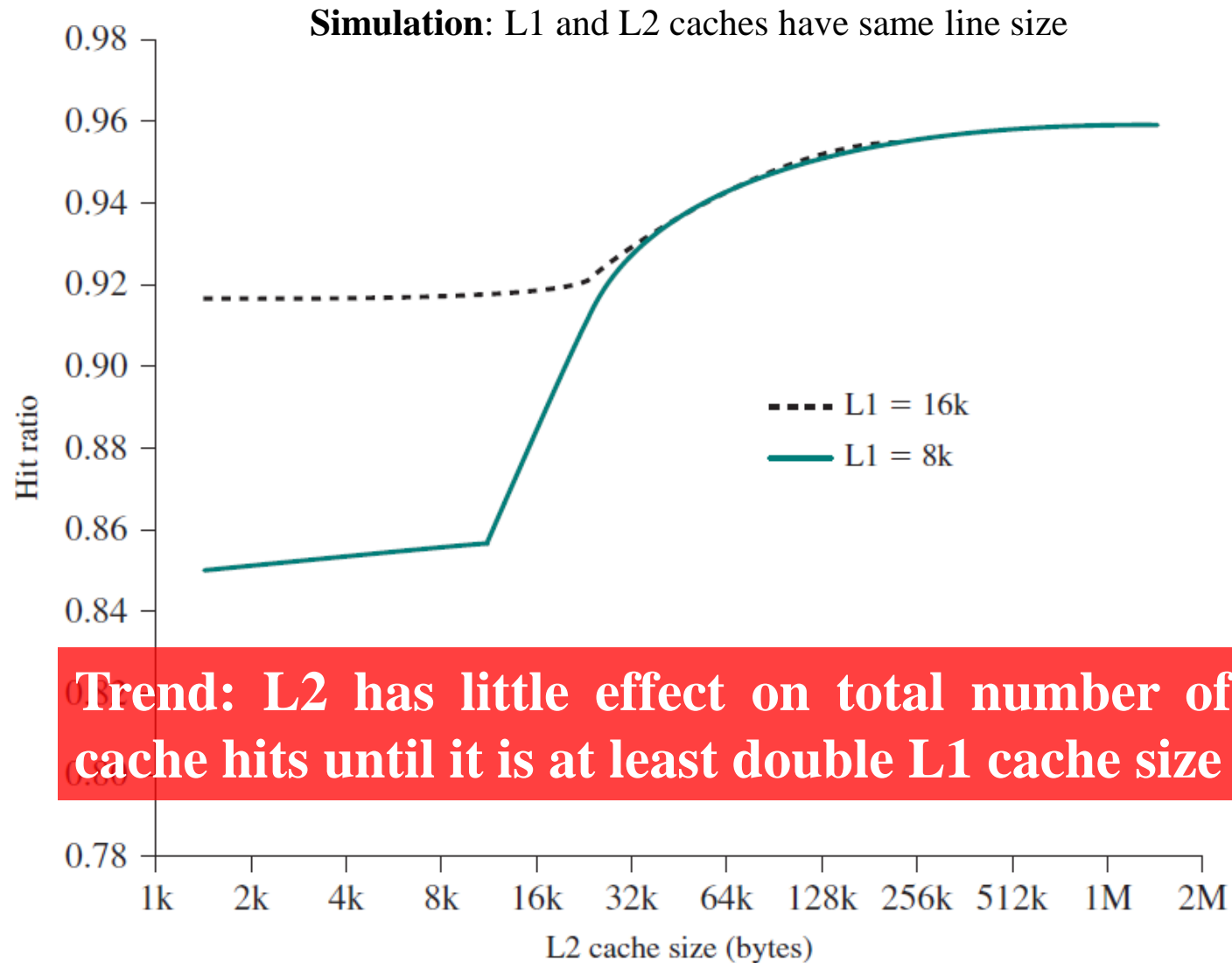
- ... (Cont.)

3. Multi-Level Caches

- Multiple levels of caches located between CPU and MM.
- Level 1 (L1) cache: closest to CPU, fastest, smallest.
- Level 2 (L2) cache: 2nd closest to CPU, 2nd fastest, 2nd smallest.
- Level 3 (L3) cache: ...
- Ex. - Operation of two-level cache: CPU looks for the requested word in L1 cache first. If not there, it fetches it from L2 cache. If not there, it is fetched from MM.
- Higher cache levels used to be off-chip, not anymore!!

4. Number of Caches

Total Hit Ratio of Two-Level Cache



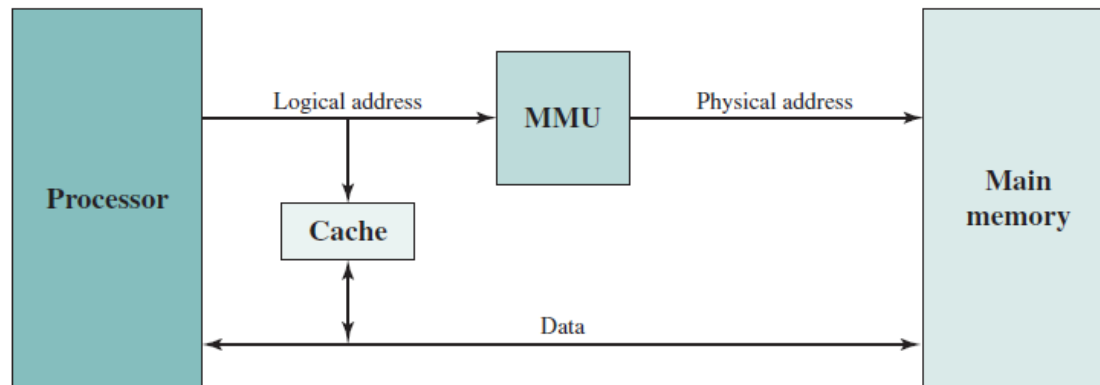
Cache Memory – Design

1. Mapping function
2. Replacement algorithm
3. Read/Write policies
4. Number of caches
5. Addresses
6. Size
7. Block/line size

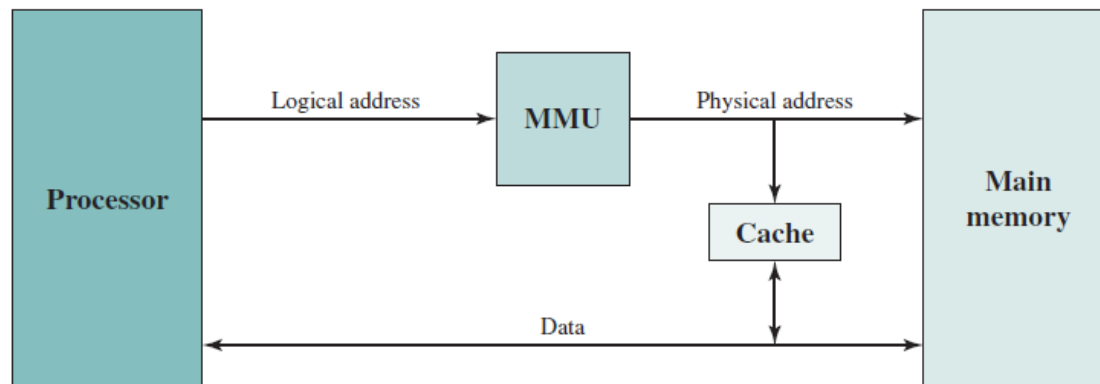
5. Cache Addressing

Location relative to MMU

- Where does cache sit relative to the memory management unit (MMU)?
 - Between CPU & MMU (MMU) → **Virtual (logical) Cache.**



- Between MMU & MM → **Physical Cache.**



5. Cache Addressing


Virtual vs. Physical Cache

- Virtual (logical) cache

- Uses virtual addresses of MM blocks in cache mapping.

- CPU accesses cache directly (not through MMU).


-  No address translation to access cache → Faster access.


-  Different applications have similar virtual addresses → Must flush cache on each context switch.

- Physical cache

- Uses physical addresses of MM blocks in cache mapping.

- CPU accesses cache through MMU.

-  Need address translation to access cache → Slower access.

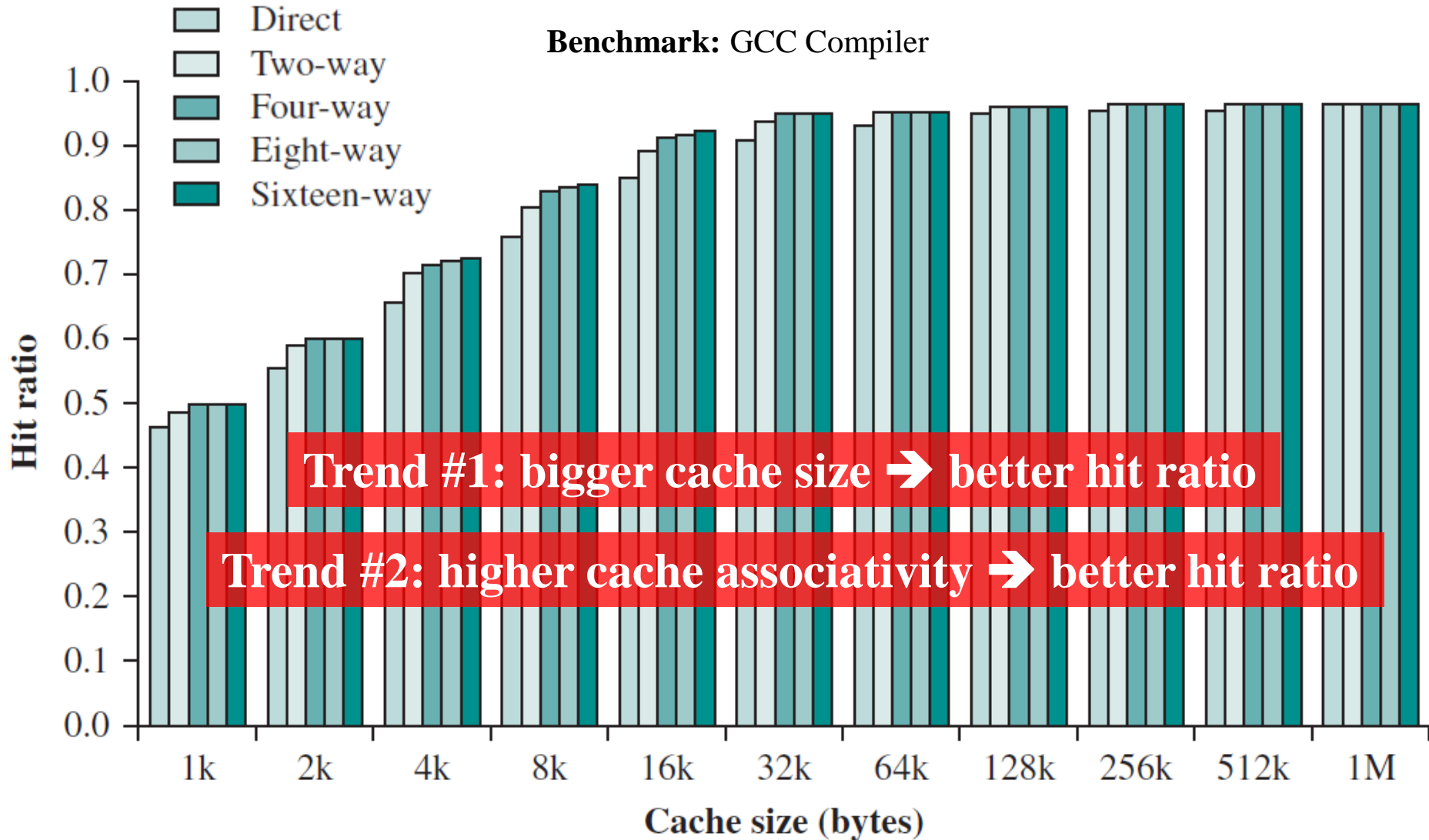
-  Different applications have different physical addresses → No need to flush cache on context switch.

Cache Memory – Design

1. Mapping function
2. Replacement algorithm
3. Read/Write policies
4. Number of caches
5. Addresses
6. Size
7. Block/line size

6. Cache Size

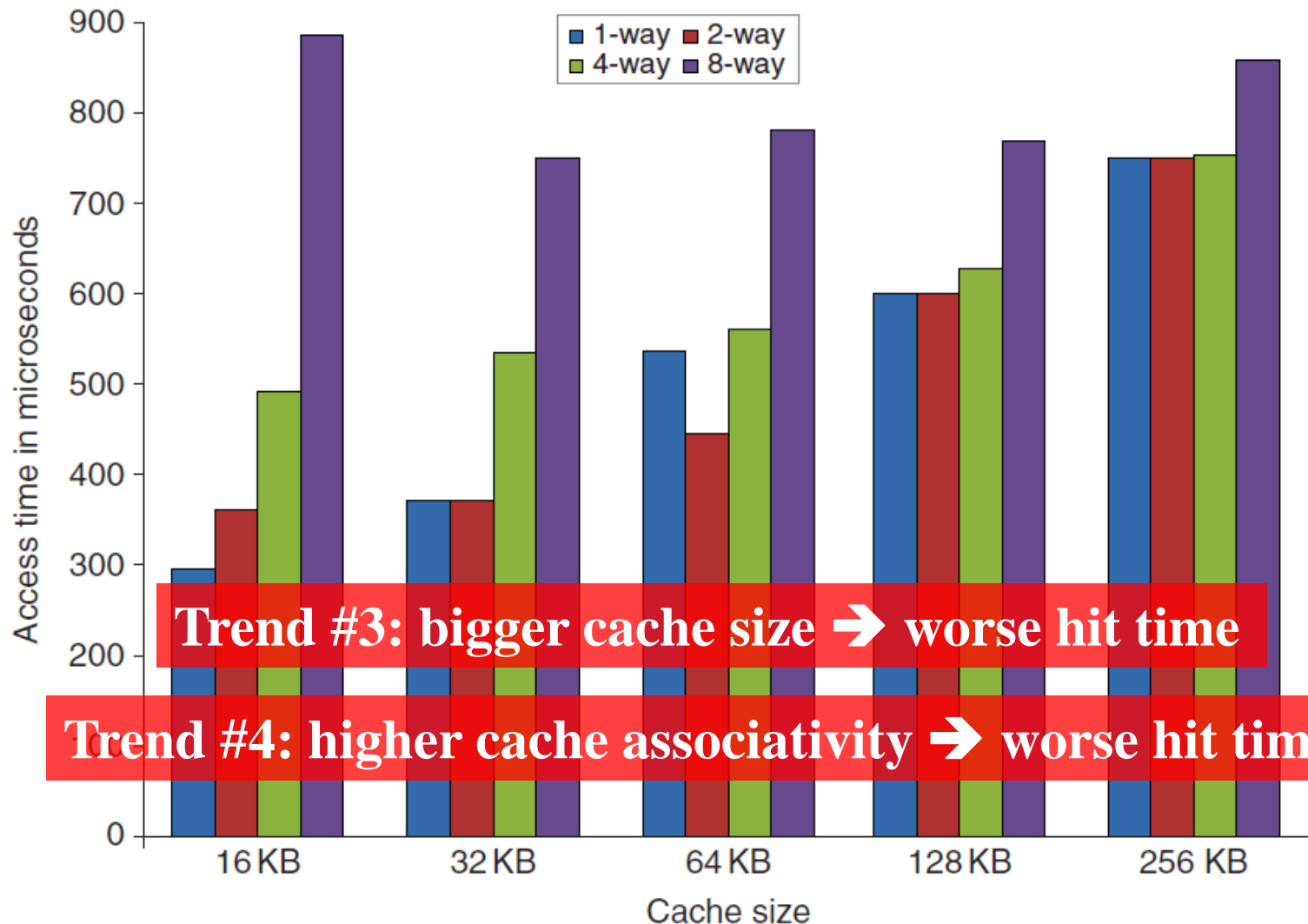
Cache Size/Associativity vs. Hit ratio



6. Cache Size

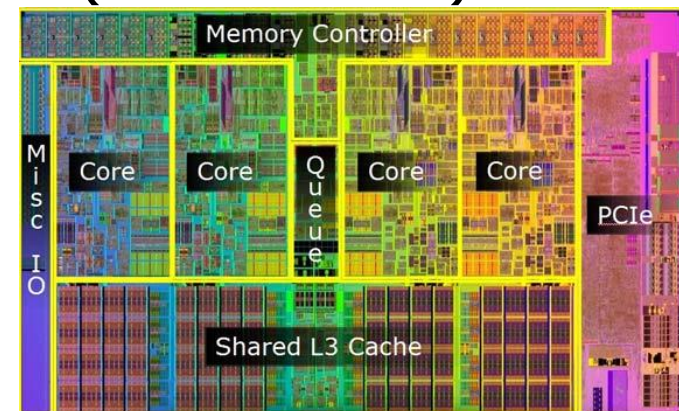
Cache Size/Associativity vs. Hit Time

Cache model: CACTI, 40-nm, single-bank, 64-byte block



6. Cache Size Summary

- Trend: **Cache size** $\uparrow \rightarrow$ **Hit ratio** $\uparrow \rightarrow$ **Hit time** \uparrow .
- Motivations for bigger cache:
 - Better hit ratio \rightarrow average access time of (cache + MM) is close to cache alone.
- Motivations for smaller cache:
 - Less gates involved in addressing \rightarrow faster cache.
 - Less cost \rightarrow average cost per bit of (cache + MM) is close to MM alone.
 - Fit the chip/board area budget.
- Reality: Cache performance depends on workload \rightarrow No single “optimum” cache size exists!

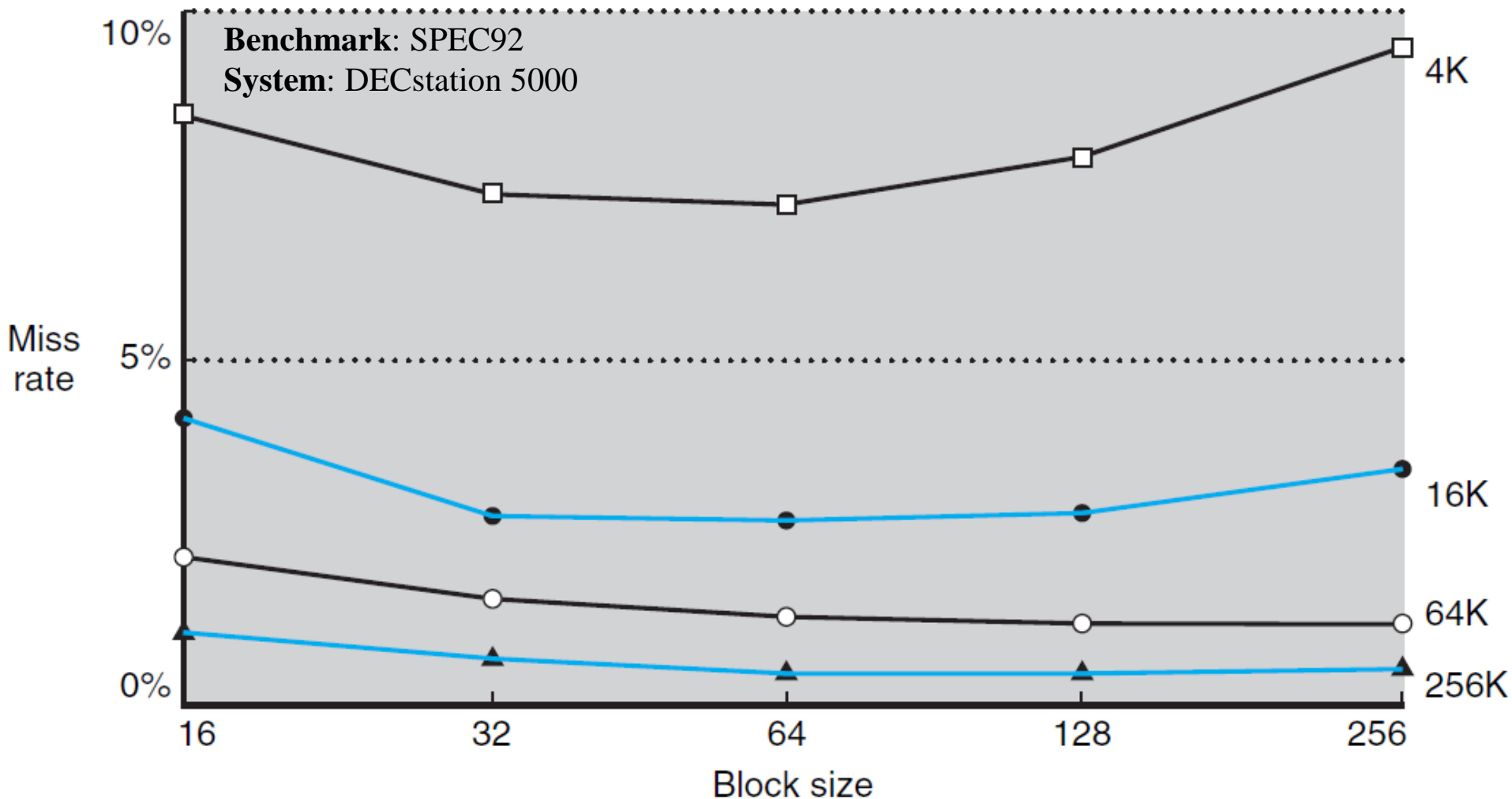


Cache Memory – Design

1. Mapping function
2. Replacement algorithm
3. Read/Write policies
4. Number of caches
5. Addresses
6. Size
7. Block/line size

7. Block/Line Size

Block Size vs. Miss Rate



7. Block/Line Size

Summary

- Block size increases → hit ratio increases at first (locality of reference).
- Block becomes too big → probability of using the fetched info becomes less than that of reusing info to be replaced → hit ratio begins to decrease.
- Two effects come into play due to larger blocks:
 1. Less blocks could be fit into cache → data gets overwritten shortly after being fetched.
 2. More words become farther from requested ones → not likely to be referenced.
- Typical block size: 8-64B (PC) & 64-128B (HPC).

Example

AMD Bulldozer Cache Organization

- **Three-level** cache organization: L1, L2, and L3.
 - **L1 → split cache**
 - L1 Code: 64KB, 2-way, 64B-line, shared (1 per 2 cores).
 - L2 Data: 16KB, 4-way, 64B-line, dedicated (1 per core).
 - Access Latency: 3-4 clock cycles.
 - **L2 → unified cache**
 - 1-2MB, 16-way, 64B-line, shared (1 per 2 cores).
 - Access Latency: 21 clock cycles.
 - **L3 → unified cache**
 - 0-8MB, 64-way, 64B-line, shared (1 per all cores).
 - Access Latency: 87 clock cycles.

Reading Material

- Stallings, Chapter 4:
 - Pages 137 – 141