CSED311 Computer Architecture – Lecture 14

Memory Hierarchy & Cache II

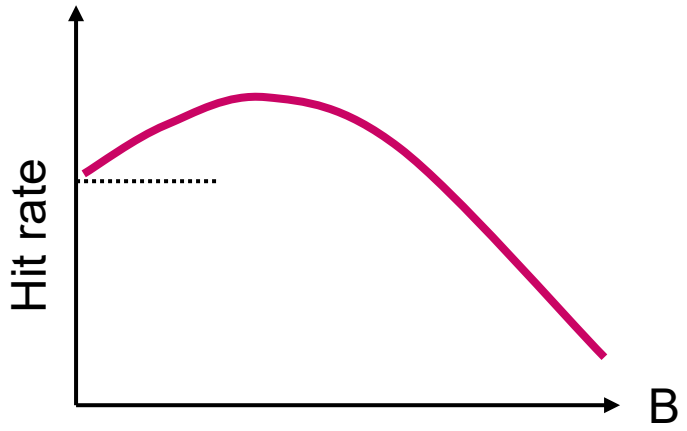
Eunhyeok Park

Department of Computer Science and Engineering
POSTECH

Disclaimer: Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Mutlu, and Wenisch @ Carnegie Mellon University, University of Michigan, Purdue University, University of Pennsylvania, University of Wisconsin.

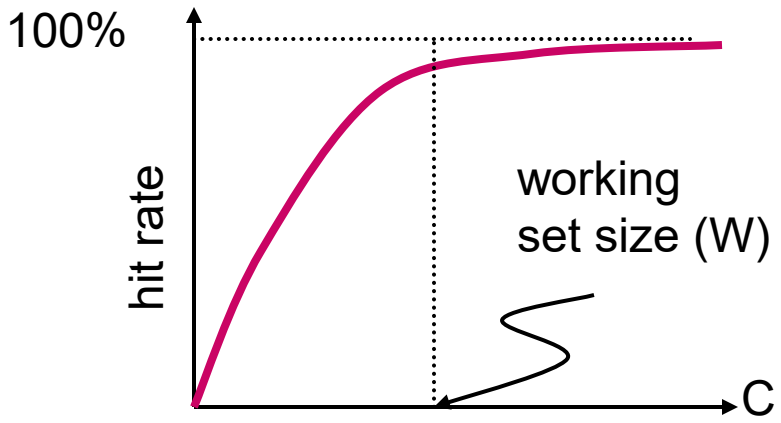
Classification of Cache Misses: Cold Miss

- Cold (or Compulsory) miss (design factor: B and prefetch)
  - **First** reference to an address (block) always results in a miss
  - Subsequent references to the same block should hit unless evicted (then, capacity miss or conflict miss)
- Dominates when locality is poor
  - For example, in a "strided" data access pattern where many addresses are visited, but each is visited exactly once → Little reuse to amortize this cost



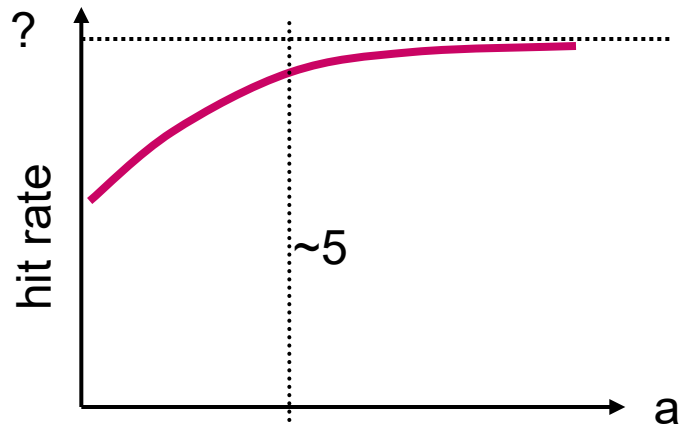
Classification of Cache Misses: Capacity Miss

- Capacity miss (design factor: C)
  - Cache is too small to hold everything needed
  - Defined as the misses that would occur even in a fully-associative cache of the same capacity due to eviction
- Dominates when C < W (Working set size)
  - For example, the L1 cache can never be made big enough due to clock frequency tradeoff



Classification of Cache Misses: Conflict Miss

- Conflict miss (design factor: a)
  - Data displaced by collision under direct-mapped or set-associative allocation (i.e., limited number of ways)
  - Defined as any miss that is neither a compulsory nor a capacity miss
- Dominates when C ≈ W or when C/B is small



More Advanced Issues

Cache "Write" Issues

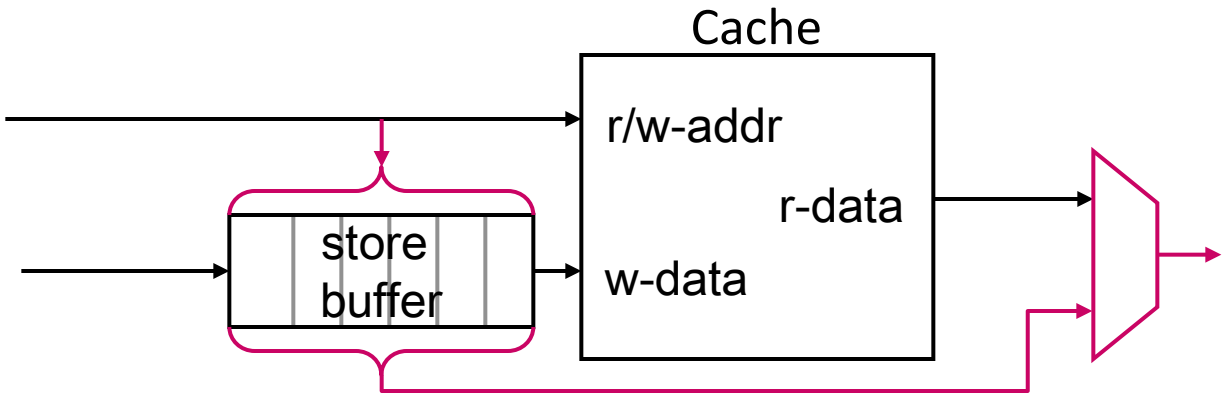
- In modern CPUs, the tag and data bank accesses are decoupled in scheduling
  - On read, schedule simultaneous access to tag and data banks as early as possible – why?
    - Reads are often at the critical path
  - On write, access tag bank first, with the data bank access possibly many cycles later – why?
    -
    -

- "**partial-word write**" issue
  - The data bank must be read first to retrieve the unmodified bytes before writing back a complete word

- Conversely, on a partial-word-read, we pick out the bytes we want and ignore the rest

Store Buffer

- If store takes multiple cycles (e.g., tag → data)
  - And it is immediately followed by another memory instruction
    → structural hazard stall occurs: back-to-back cache ops
  - Delayed load is usually bad for performance
- You can delay "data update" to the cache
  - After checking the tag bank for write-hit, buffer write-data until next free data bank cycle
  - But, must make sure the cache line is not replaced by others
- Memory forwarding
  - Later loads must check against pending store addresses in the store buffer for RAW dependence



Blocking Miss or Non-Blocking Miss?

- Does the CPU need to wait for a miss to be resolved?
- While a cache miss is being handled, should reads and writes to other addresses be allowed?
  - Important in high-clock-rate processors to avoid losing too many instruction issue opportunities.
  - But, must take care of the ordering and dependency while memory operations are completed out-of-order

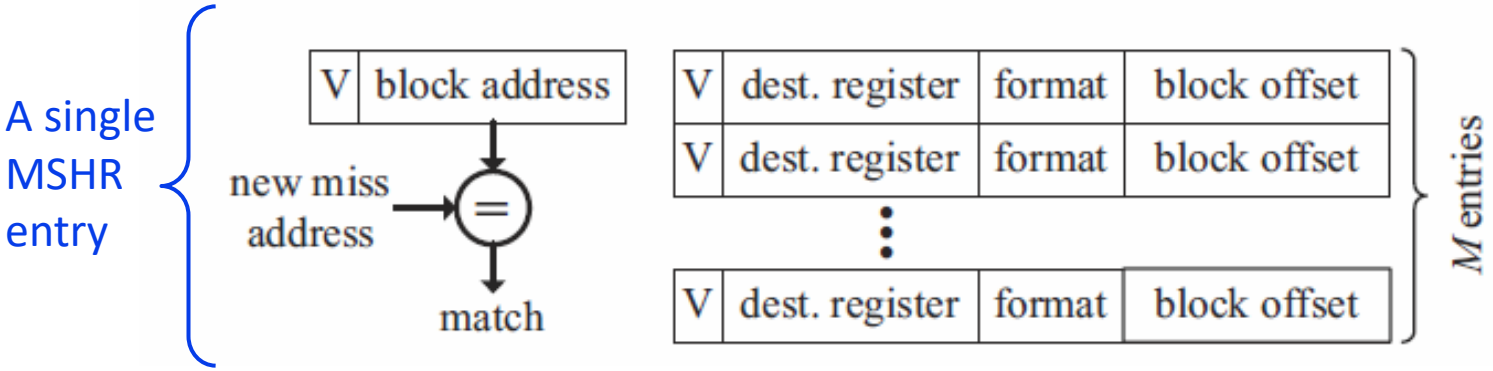
- Even in an In-Order pipeline, non-blocking write miss can be useful
  - The pipeline does not have to stall just because a STORE hasn't completed all the way into the cache
  - But on a RAW-dependent LOAD, must either stall or forward

Non-Blocking Cache using MSHR

- Blocking Cache
  - Stop the pipeline on a cache miss

- Non-blocking Cache (a.k.a. “lockup-free” Cache)
  - Keep executing instructions while servicing a missing event
  - Microarchitecture to make non-blocking cache
    - **MSHR (Miss Status/information Holding Registers):** Each entry keeps track of status of outstanding memory requests to a cache miss being serviced
  - Miss definition
    - Primary miss : Miss to a cache line
    - Secondary miss : Another miss to the cache line, while the primary miss is still being serviced
    - Structural miss : All MSHRs being used

MSHR (Miss Status Handling Register)



- **Dest. register**: destination register of loads
- **Format**: size (byte, word, …), sign- or zero-extended, offset inside the word (for byte-sized load), etc.

- MSHR
  - On a miss, allocate one entry per cache line that missed
  - Prevent the pipeline from stalling due to other cache misses
  - New cache miss will check against MSHR
    - On a match, we know this cache is being serviced due to miss
    - All instructions that missed to be notified when miss is completed
  - The pipeline stalls when a miss occurs but all MSHRs are used

Write-Hit Policy #1: Write-Through Cache

- On a write-hit in L_i, update **also** L_{i+1}

  (i.e., write through to lower-level hierarchy)

- Write-through
  - Simple policy
  - Search can be done only by looking at the lower-level hierarchy
    - Good for I/O devices: accessing lower-level memory directly will see values consistent with the upper-level caches
  - Not a good option for high-performance processors today

    3.6GHz, IPC=2, 10% stores, ~4byte/store → ~3GByte/sec

L1 write-through to L2 is still doable if L1 and L2 are in the same core.

But, (per-core, private) L2 → (shared) L3 will be expensive!

Write-Hit Policy #2: Write-Back Cache

- On a write-hit in L_i, update **only** L_i and set "**Dirty**" bit
  - Reads and writes hit in the cache until later replacement
  - On later replacement, the cached copy must be **written back** to lower hierarchy, L_{i+1}, which presumably has its older version

    Reduce required bandwidth at lower hierarchies

- "**Dirty**" bit
  - Keep a status bit per block to record if a block has ever been modified since brought into L_i
  - If not dirty (i.e., "clean"), no write-back necessary on replacement

    What if an I/O device wants to check the most recent update?

Write-Miss Policy: Write-alloc. vs Write no-alloc.

- On a write miss, where will the block be located?

- Write-Allocate
  - On a write miss in L_i, allocate the block in L_i
  - **Makes more sense for Write-Back cache**
    - On a write-miss, the block is brought in and updated there
  - Can reduce write traffic if there is good locality

- Write-No-Allocate
  - Do you expect high locality between back-to-back store and load?
  - On a write miss in L_i, do not allocate the block in L_i and keep in lower cache or memory
  - **Makes more sense for Write-Through cache**
    - You don't want to keep updating the entire memory hierarchy

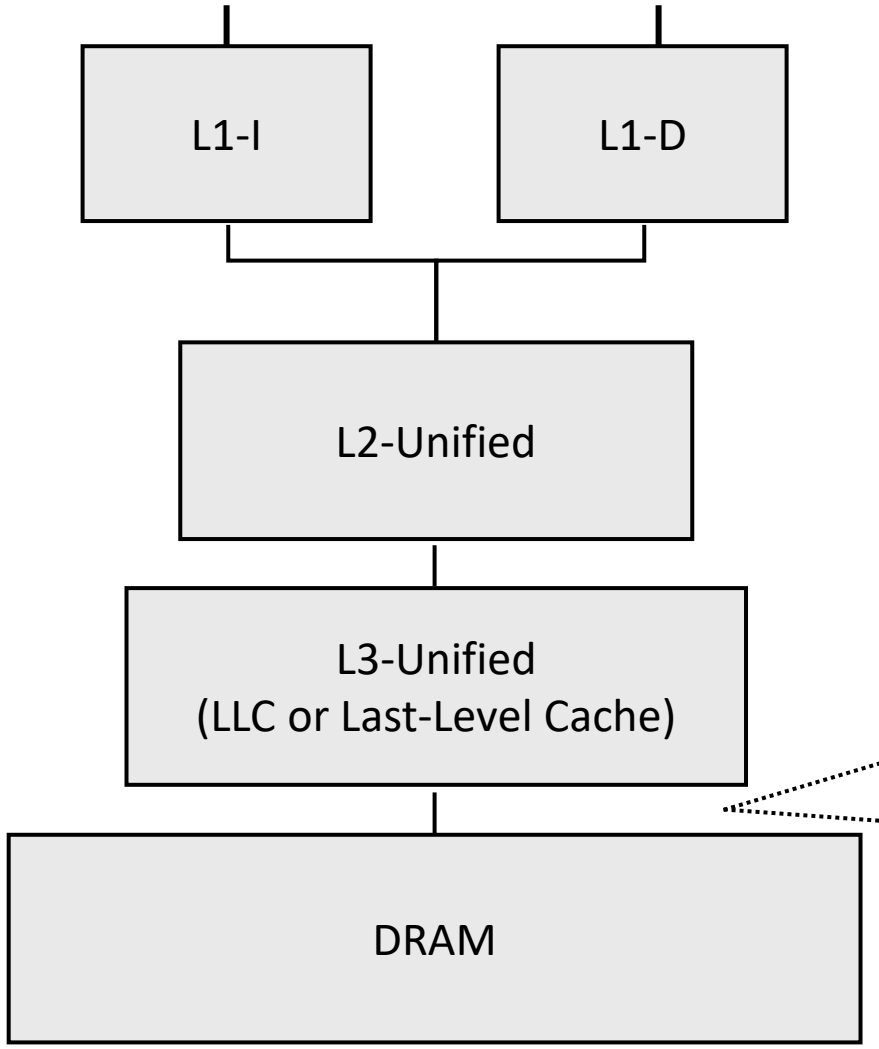
Unified vs Split I/D

- Modern high-performance processors typically use split (and sometimes asymmetric) L1 caches for instruction and data
  - Instruction and data memory footprints typically disjoint
  - Instruction: smaller footprint, higher-spatial locality and (mostly) read-only
  - Split L1 caches provide free doubled bandwidth, no-cross pollution, and separate design customizations

- L2 and L3 are typically unified

Multi-Level Caches

$T_i = t_i + m_i \cdot T_{i+1}$



- A few pclk (processor clk) latency
- Many GB/sec (even for random access)

Intermediate hierarchy is a cheaper way to reduce T_1 than using a faster or bigger L1,

(off-chip) L4 in-between?

- Hundreds of processor clk latency
- ~GB/sec (sequential)

Remember, memory hierarchy is also about memory bandwidth

Cache Hierarchy in Modern Server CPUs

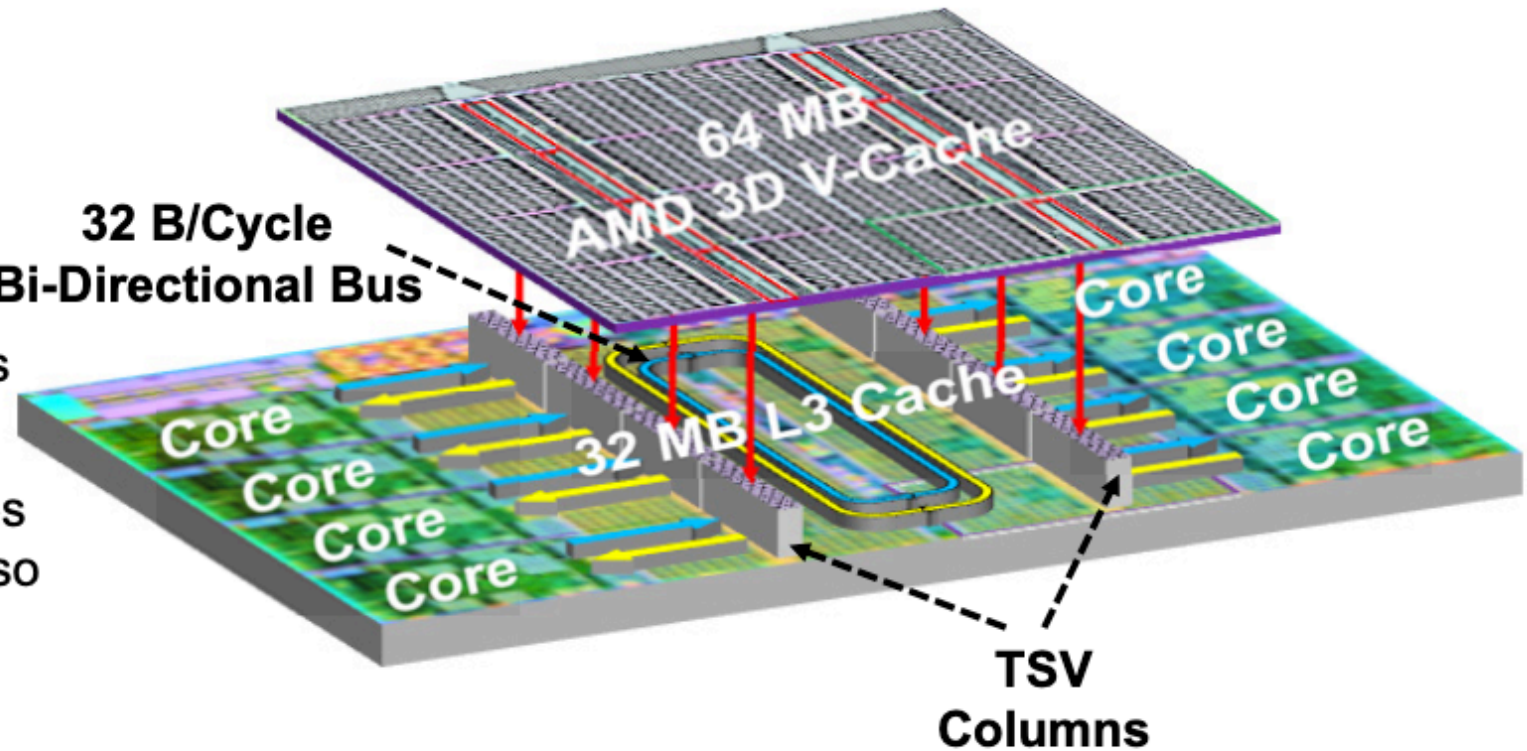
Memory Hierarchy	AMD EPYC 7742 (@ 3.4GHz)	Intel Xeon 8280 (@ 2.7GHz)
L1 Cache	32KB 4 cycles (1.18 ns)	32KB 4 cycles (1.48 ns)
L2 Cache	512KB 13 cycles (3.86 ns)	1024KB 14 cycles (5.18 ns)
L3 Cache	16MB / CCX (4C) 256MB Total ~34 cycles (~10.27 ns)	38.5MB / (28C) ~46 cycles (~17.5 ns)
DRAM	~122 ns	~89 ns

Source: https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen/7

Note: these are load-to-use latencies (i.e., include latencies through all upper layers due to misses)

AMD 3D V-Cache Ready

- Two columns of TSVs on left/right side of the L3 cache
- AMD 3D V-Cache extends L3 Cache capacity by 64MB (3x)
- Total inter-die bandwidth: >2 TB/s
- All control and routing to the cores is implemented on the base die, so AMD 3D V-Cache can be completely focused on density



*768 MB L3 cache for high-end server CPUs

Multi-Level Cache Design

- Upper Hierarchies
  - Small C: upper-bound by SRAM access time
  - Smallish B: upper-bound by C/B effects and the benefits of fine-grain spatial locality
  - a: some degree required to counter C/B effects

- Lower Hierarchies
  - Large C: upper-bounded by chip area
  - Large B: to reduce tag storage overhead and to take advantage of coarse-grain spatial locality
  - a: upper-bounded by the implementation complexity
  - Modern on-chip L2/L3s can be highly associative

Very large off-chip caches (e.g., L4 cache) are either direct-mapped or use on-chip SRAM tag + (e)DRAM

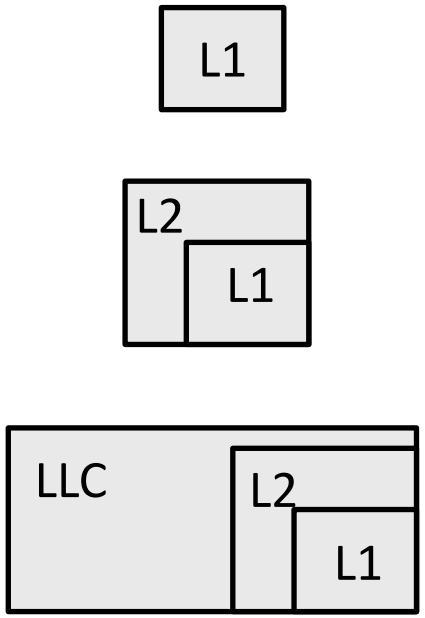
Inclusive vs. Exclusive

Inclusive cache hierarchy

- LLC is the superset of all caches
- A block in L1 is also present in L2 and LLC

Exclusive cache hierarchy

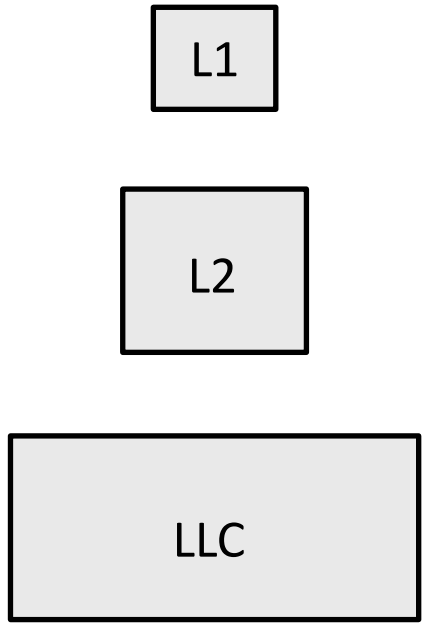
- A Cache block is present only in one level
- A block in L1 is never present in L2 and LLC



Inclusive Hierarchy

Which is better in terms of effective capacity?

Why would one use the other?



Exclusive Hierarchy

Slide credit: Gaur et al., ISCA'11

Back Invalidation Problem in Inclusive $

- **Back-invalidation**: When a block is evicted from the LLC, inclusion is enforced by invalidating that block from all the caches in the upper levels
- Example:
  - Core keeps accessing "A" in L1
  - L2 does not see accesses to "A"
    - "A" is placed at LRU (Least Recently Used) position
  - "A" is evicted from L2
    - Can we keep "A" in L1?

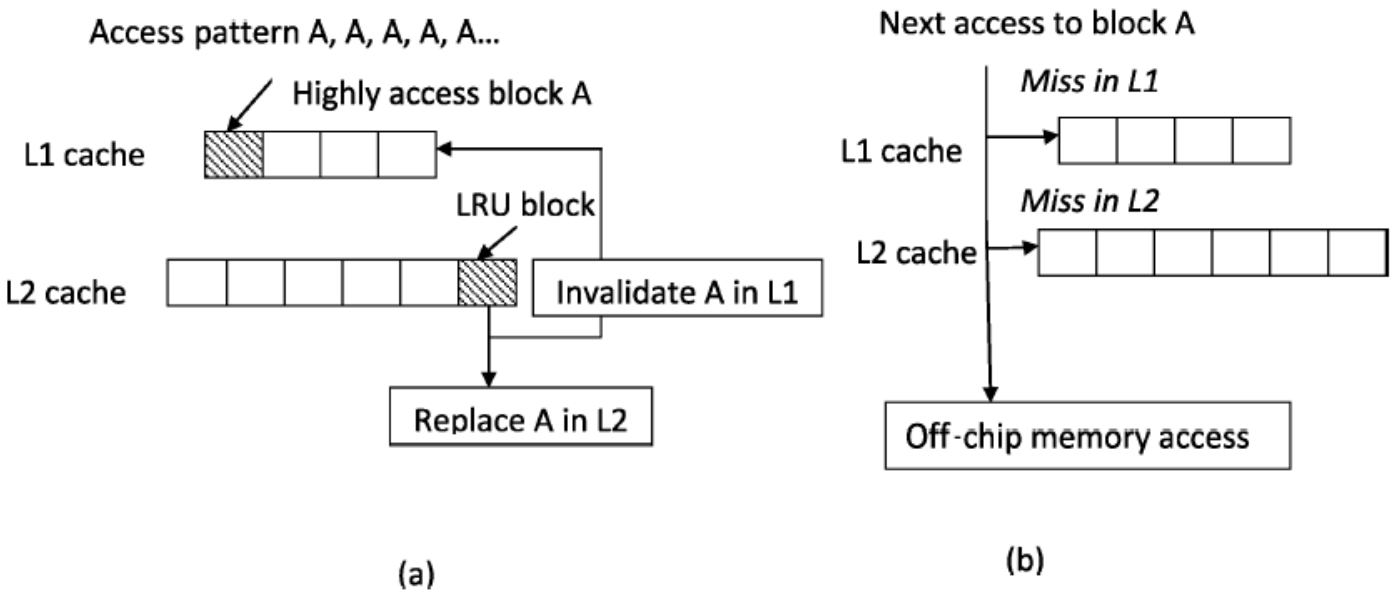
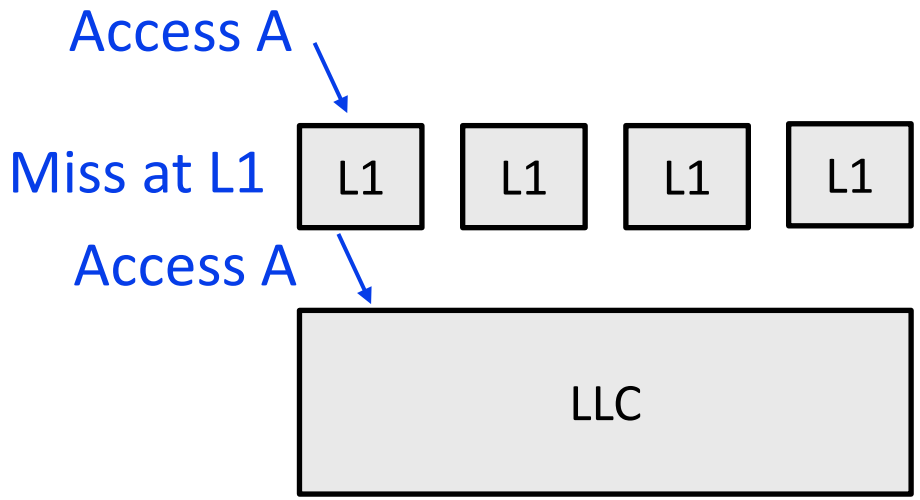


Fig. 2. An example of inclusion problem.

Figure from Tian et al., Temporal-based multilevel correlating inclusive cache replacement. ACM Trans. Archit. Code Optim. 10, 4, Article 33

Cache Coherence Complexity in Exclusive $



Miss at LLC → What happens now?

Inclusive cache

It is guaranteed that the block "A" is not present in other L1s
→ Access DRAM

Exclusive cache

The block "A" may be present in other L1s
→ Need to check all other L1s (or keep all the tags of the blocks in all L1s at L2) before accessing memory

We will learn more about coherence in a later lecture

Reverse Engineering Cache Hierarchy

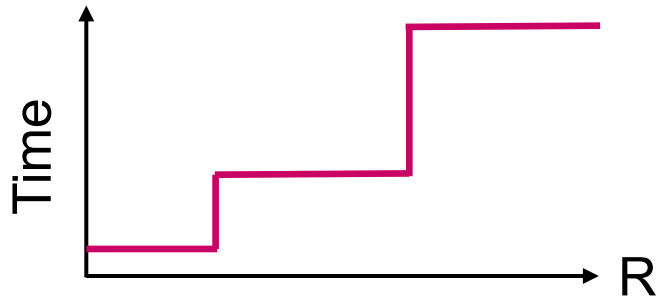
What caches are in your computer?

- Would it be possible to figure out the cache configuration in your computer?
  - Why not just look at the specification?
    - Detailed specification may not be provided by the manufacturer
    - As a CPU architect, you may want to validate your chip
  - Example configs: Capacity (C), Associativity (a), and Block-size (B), Number of levels, etc.

- Caches are supposed to be transparent to the programmer
  - The presence or lack of a cache should not be detectable by the functional behavior of software
  - But you could tell it when you measure timing behavior (e.g., execution time) which infers the number of cache misses

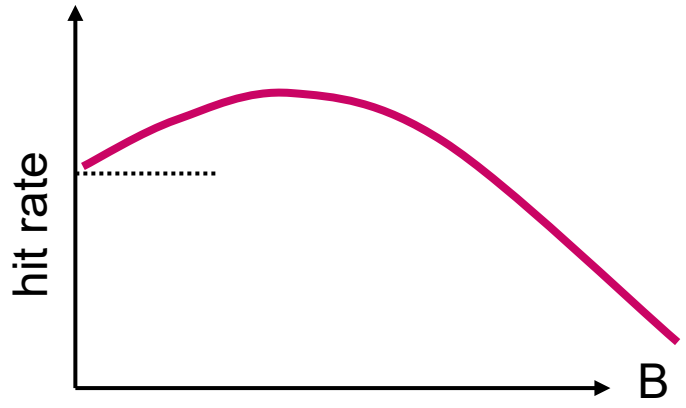
Capacity Experiment (for D-cache)

- Assume C is a 2-power number
- For increasing values of R, where R is a 2-power
  - Allocate a buffer of size R
  - **Read every memory location in R** in sequence, and repeat
- For small R<=C, we expect to hit in the cache
- For large R>C, we expect to miss in the cache and experience a noticeable jump in memory access time
- By continuing to increase R, we can look for the step-function-like increase in access time when the buffer size spill out of the next cache level.



Block Size Experiment (with known C)

- Allocate a buffer of size R that is a multiple of C
- For increasing S, read just every S'th memory location in the buffer, and repeat
- Since R>C, we expect to miss on the first access to each cache block
- When S>=B, we only use one word per cache block
- When S<B, we expect improving average memory access time for smaller S since we read more words per cache block miss



Associativity Experiment (with known C)

- For increasing values of R, where R is a multiple of C
  - Allocate a buffer of size R
  - Read every C'th memory location in sequence, and repeat
- All R/C referenced addresses map to the same set
- When a>=R/C, we expect the references to hit in the cache since all referenced addresses fit within the set
- When a<R/C, we expect at least some misses since all referenced addresses cannot fit simultaneously
  - We expect 100% cache miss if LRU is used.

What else could you tell?

- Write-through vs. Write-back
- Write-allocate
- Unified vs. split design
- I-cache C, B, a
- Latency and bandwidth
- Replacement policy
- . . . .
- Lmbench is a widely known tool for this purpose
  - *"lmbench: Portable Tools for Performance Analysis,"* McVoy and Staelin, USENIX ATC 1996
  - http://lmbench.sourceforge.net/
- Caution: experiments based on our simplified understanding of caches will NOT predict behaviors of modern CPUs exactly, because virtual memory, complex hierarchies, index hashing, and prefetchers (which you can partially turn off),

But it can still tell you lots. Try it!

Question?

Announcements

- Reading: P&H Ch. 5.8 – 5.9, 5.12 – 5.17