CSED311 Computer Architecture – Lecture 13

Memory Hierarchy & Cache

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.

Memory: "Ideal" vs. Reality

- "Ideal" memory
  - Each program sees a contiguous 2^{32} B (32-bit arch.) or 2^{64} B (64-bit arch.) memory
  - We can access anywhere in memory in 1 processor cycle

> 4.1. Ideally one would desire an indefinitely large memory c
> pacity such that any particular aggregate of 40 binary digits,
> *word* (cf. 2.3), would be immediately available—i.e. in a tin
>
> *-- Burks, Goldstein, von Neumann, 1946*

- In reality, …
  - Can't afford and don't need 2^{64} B memory even in a 64-bit architecture
  - You can't find memory technology that is large and fast

- Solution: Create "magic" memory with a hierarchy of (fast but small) caches and (large but slow) main memory

The Law of Storage

- Bigger is slower (and have lower bandwidth)
  - SRAM 512 B @ sub-nsec
  - SRAM KB~MB @ nsec
  - DRAM GB @ ~50 nsec
  - SSD/Hard disk TB @ µs to ms

- Faster is more expensive (dollars and chip area)
  - SRAM ~$10K per GB
  - DRAM ~$10 per GB
  - SSD ~$1 per GB

Note these sample values scale with time

How to make memory Bigger, Faster, and Cheaper?

Memory Hierarchy Analogy

Capacity:	Low	Medium	Huge
Latency:	3 sec	30 sec	30 min

Basic Principles

Locality

- One's recent past is a very good predictor of his/her near future.
- Temporal Locality: If you had lunch at student cafeteria today, it is very likely that you will go there again soon
- Spatial Locality: If you had lunch at student cafeteria, it is very likely you will go to a nearby shop in Jigok Community Center

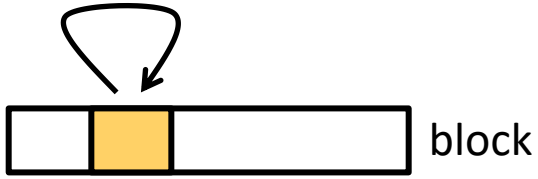
Memory Locality

- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

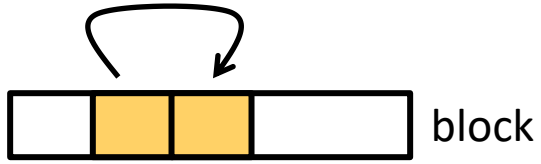
*** typical programs are composed of “loops” ***

```
sum = 0;
for (i = 0; i < MAX; i++)
   sum = sum + Arr[i];
```

- Temporal Locality:
  - Recently referenced items are *likely* to be **referenced again** in the near future
  - E.g., instructions within a loop, data with loop-carried dependence



- Spatial Locality:
  - Items with **nearby addresses** *are likely* to be referenced close together in time
  - E.g., instructions within a basic block, array/data structure references



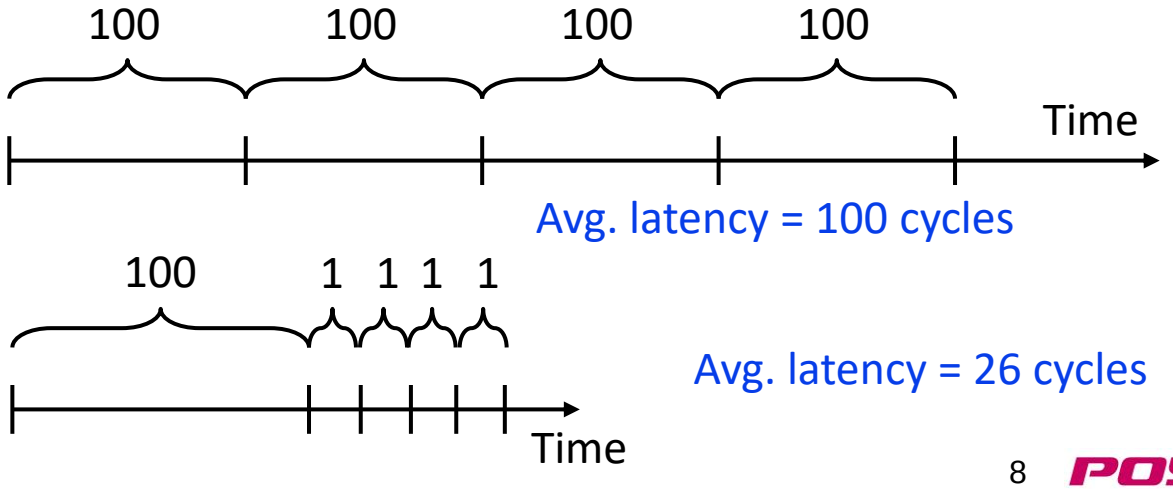
Cost Amortization

- Total-cost = One-time-overhead + Per-unit-cost x *N*
- Average cost = Total-cost / *N*

  = (One-time-overhead / *N*) + Per-unit-cost

- It is often okay to have a high overhead cost if the cost can be distributed over a large number of units → low average cost

- E.g., accessing a word four times from memory
  - Assume 100-cycle DRAM latency, 1-cycle cache latency
  - **Without cache**
    (Four separate DRAM accesses)
  - **With cache**



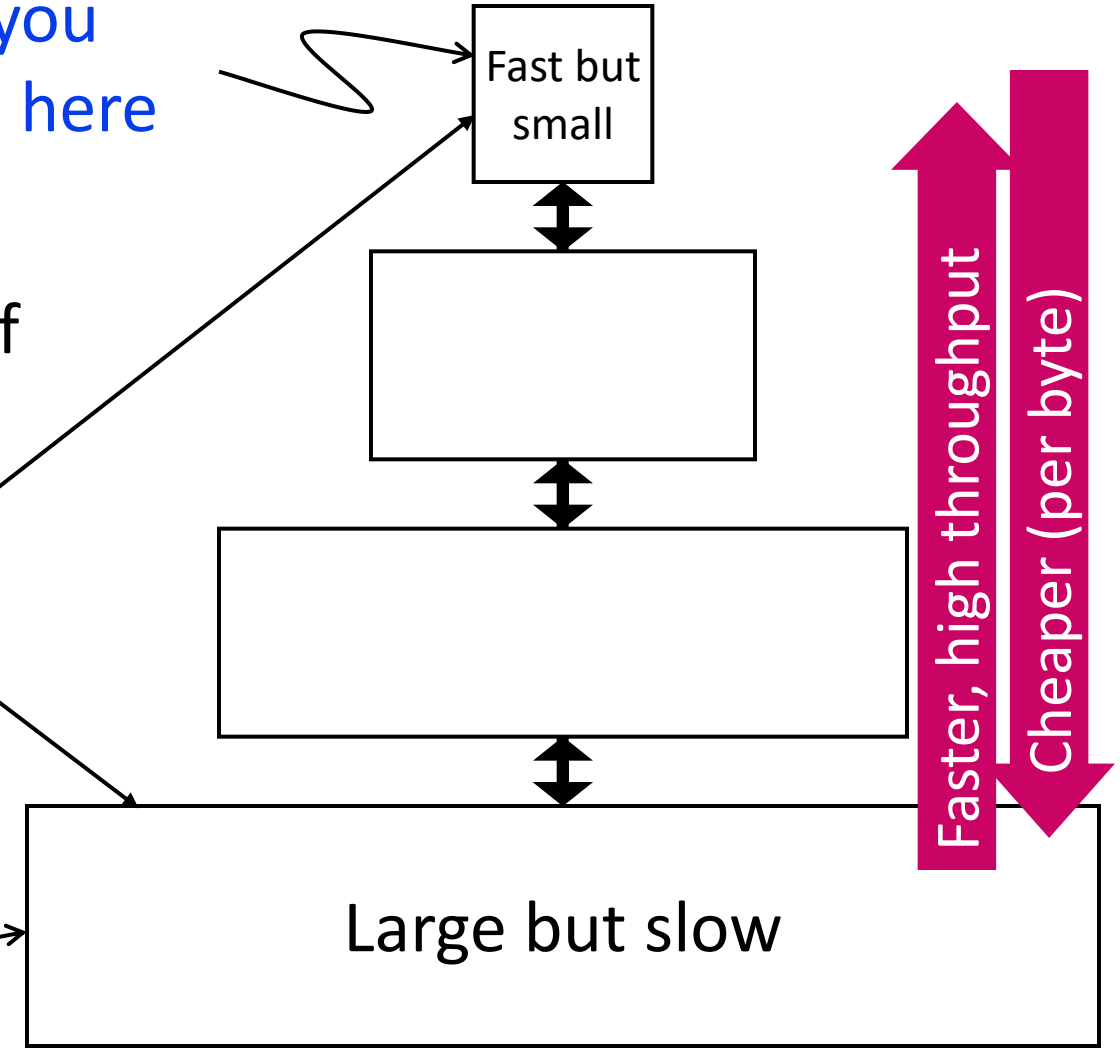
Putting the principles to work

Memory Hierarchy

Keep what you actively use here

With good locality of reference, memory appears as fast as and as large as

Backup everything here



Cache

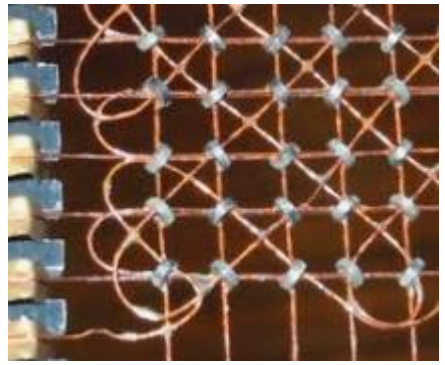
- English definition: a hidden storage space for provisions, weapons, and/or treasures

- CSE definition: computer memory with short access time used for the storage of frequently or recently used instructions or data (I-cache and D-cache)

  More generally,
  used to optimize data transfers between system elements with different characteristics (e.g., cache in CPU, cache servers for internet services, etc.)

Managing Memory Hierarchy

- You could manage data movement across hierarchies **manually**
  - Already discussed in von Neumann paper
  - "Core" (small, fast) vs. "Drum" (large, slow) memory in the 50's
  - Too painful for programmers on substantial programs
    - Imagine you have to designate the location of every variable in a C code
  - Still done in GPUs and special-purpose processors (e.g., Google TPU) (on-chip scratchpad SRAM)



Core memory

Image: Rhode Island Computer Museum

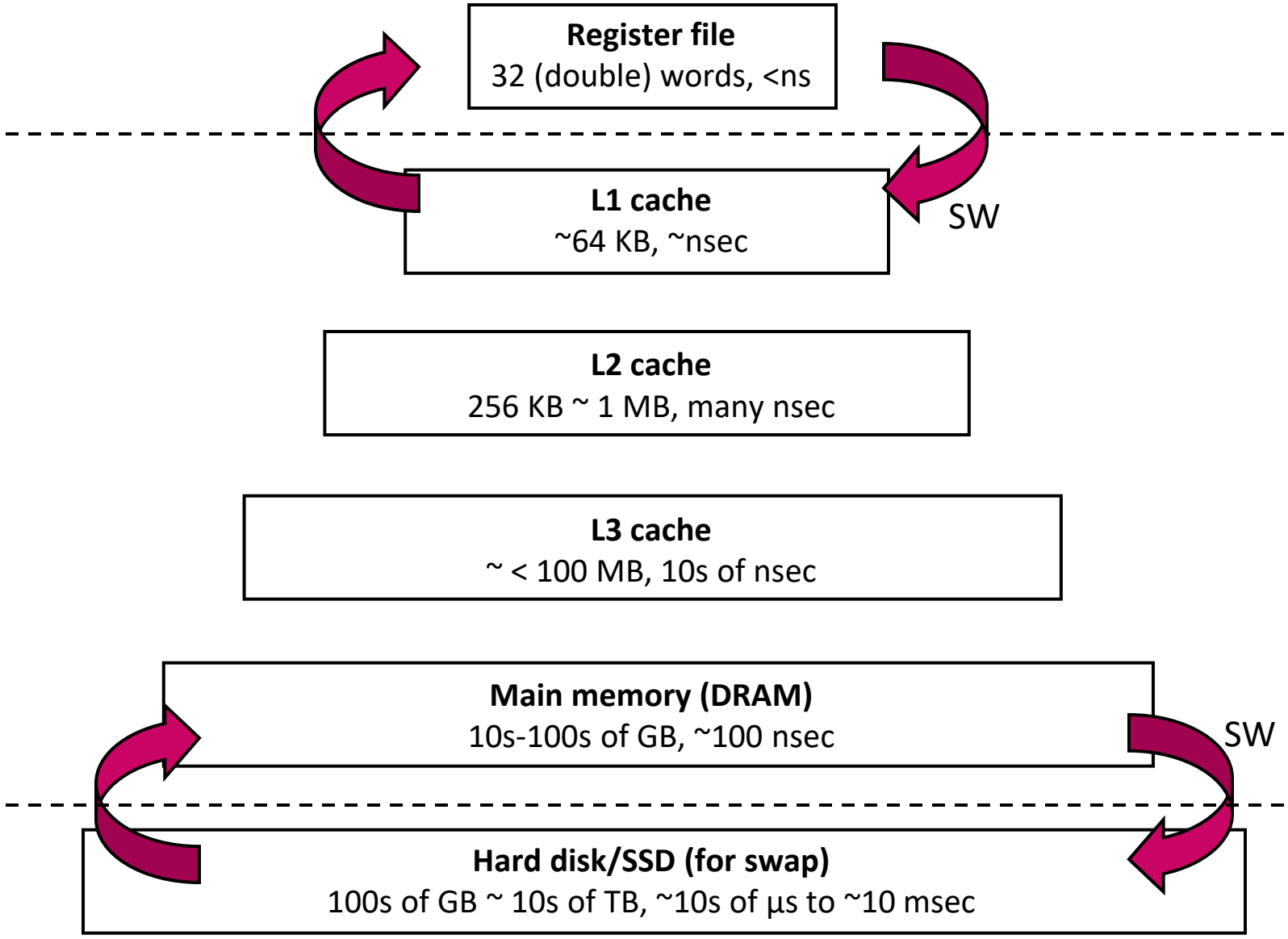
- **Automatic** management across hierarchies
  - Simple heuristic: **keep "most recently used" items in fast memory**
  - Dates back to ATLAS, 1962
  - Used in every modern servers/PCs/smartphones these days
  - Transparent to the average programmer

You can write a "correct" program without knowing cache size
You might need to know it to write a "faster" program, though

Drum memory

Image: Gregg Tavares

Modern Memory Hierarchy



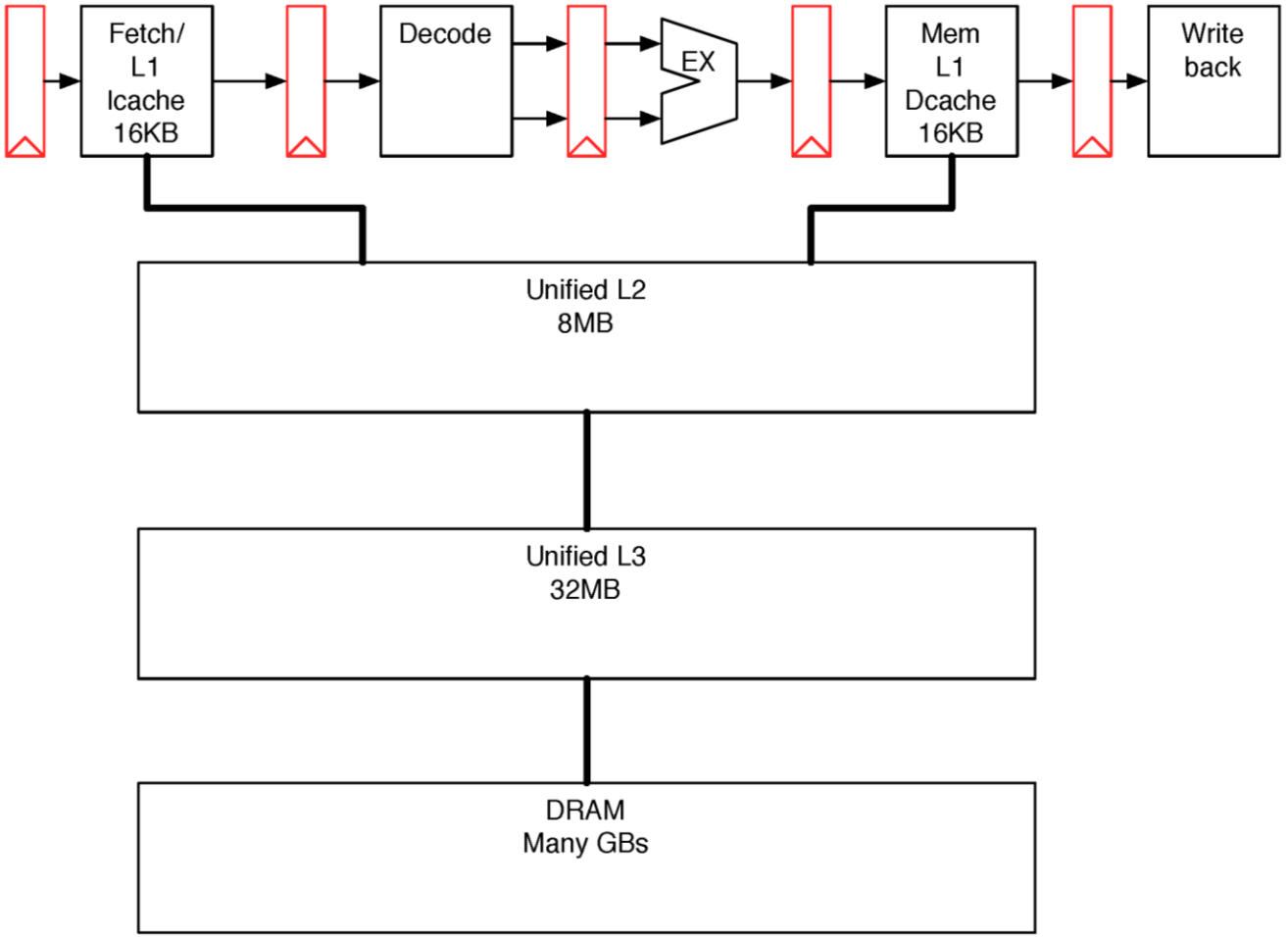
Register spilling
(by compiled code)

Automatic cache
management (by HW)

Automatic demand
paging (by OS)

Memory Hierarchy with a 5-stage Pipeline

- Separate I-cache and D-cache at L1
- Unified cache for both instruction and data at L2 and lower levels



Hierarchical Performance Analysis

- A given memory hierarchy level i has a technology-intrinsic access time t_i
- The perceived access time T_i can be longer than t_i due to misses
- Except for the very last level in the hierarchy, when looking for a given address there is
  - A chance (hit-rate h_i) you "hit" and access time is t_i
  - A chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus,

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$
$$= t_i + \underbrace{m_i \cdot T_{i+1}}$$

"miss penalty"

Keep in mind, h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at $Level_{i-1}$ (i.e., not including the hits and misses in upper levels)

Cache as Request Filter

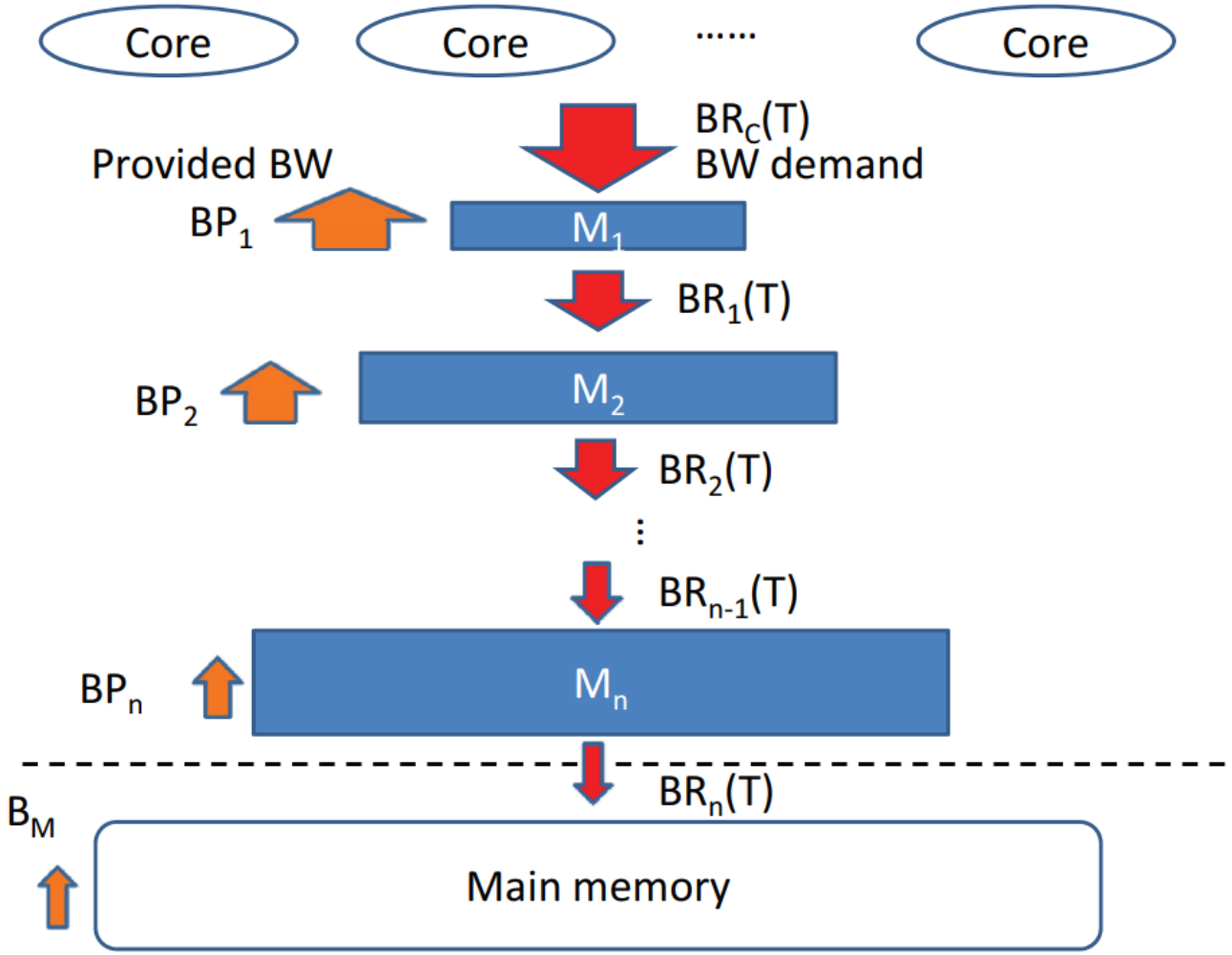


Figure from "Moguls: A model to explore the memory hierarchy for bandwidth improvements", Sun et al., ISCA'11

Hierarchy Design Compromises

- Recursive latency equation – $T_i = t_i + m_i \cdot T_{i+1}$
- Goal: achieve desired T_1 (or AMAT, Average Memory Access Time) within allowed cost
- $T_i \approx t_i$ is desirable but not necessary (processors can tolerate some amount of latency)

(1) To keep m_i low

- Increasing capacity C_i lowers m_i but increases t_i
  (imagine you have bigger backpack with more stuff in it)
- Lower m_i by smarter allocation
  - Replacement :: evict what you don't need (soon)
  - Prefetching :: get what you will need in advance

(2) To keep T_{i+1} low

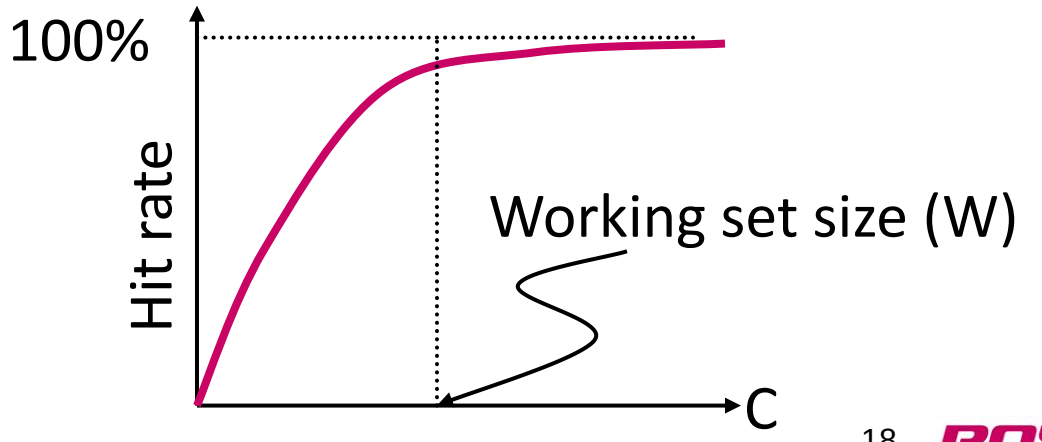
- Faster memory at lower level, but be aware of increasing cost
- Introduce intermediate hierarchies as a compromise (e.g., add L2 and L3 caches)

Hierarchy Design Considerations

- DRAM (Dynamic RAM: used for large memory)
  - Optimized for capacity/$
  - T_{DRAM} is essentially determined by timing parameters, for a given technology generation (it can still vary depending on the locality within DRAM – to be discussed later)

- SRAM (Static RAM: used for caches and register file)
  - Optimization considerations: capacity, latency, capacity/$
  - Different compromises between capacity and latency for different cache levels

- **Working set**: a collection of data that is being actively used by a process
  - Ideally, you want to keep most of the working set in caches



Intel P4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16KB
  - t_1 = 4 cycle (int) / 9+ cycle (fp)
- L2 D-cache
  - C_2 =1024 KB
  - t_2 = 18 cycle
- Main memory
  - t_3 = ~ 50ns (or 180 cycle)
- Notice
  - Even L1 access takes longer than 1 cycle
  - The worst-case access latency are into 300+ cycles, depending exactly what happens

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

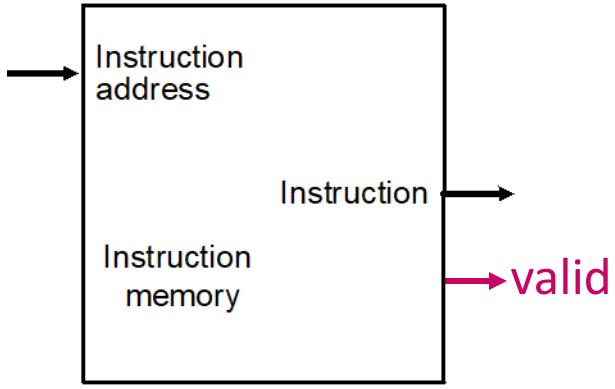
if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

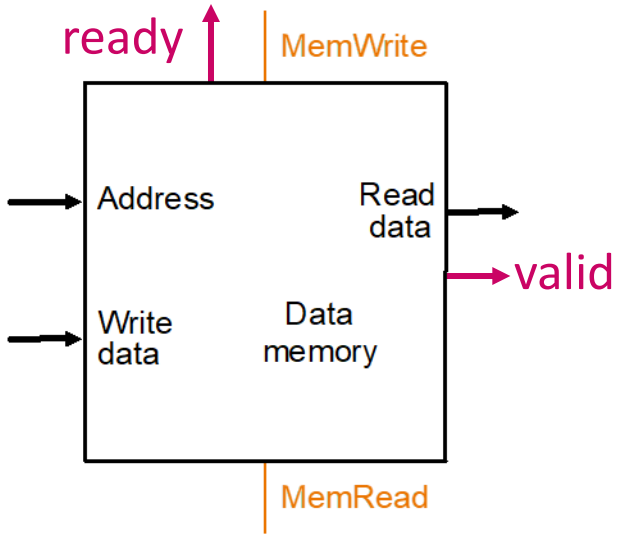
if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

Cache Design Basics

Cache Interface





- Like the magic memory we assumed so far
  - Present address, command, etc.
  - Most of the time, result or side-effect valid after a short/fixed latency
- Except, caches may not be valid/ready on every cycle
  - The cache eventually must become valid/ready
  - What happens to the pipeline until then? The instruction stalled!

The Basic Problem

- Potentially $M=2^m$ bytes of memory, how to keep the most frequently used ones in C bytes of fast storage where $C << M$

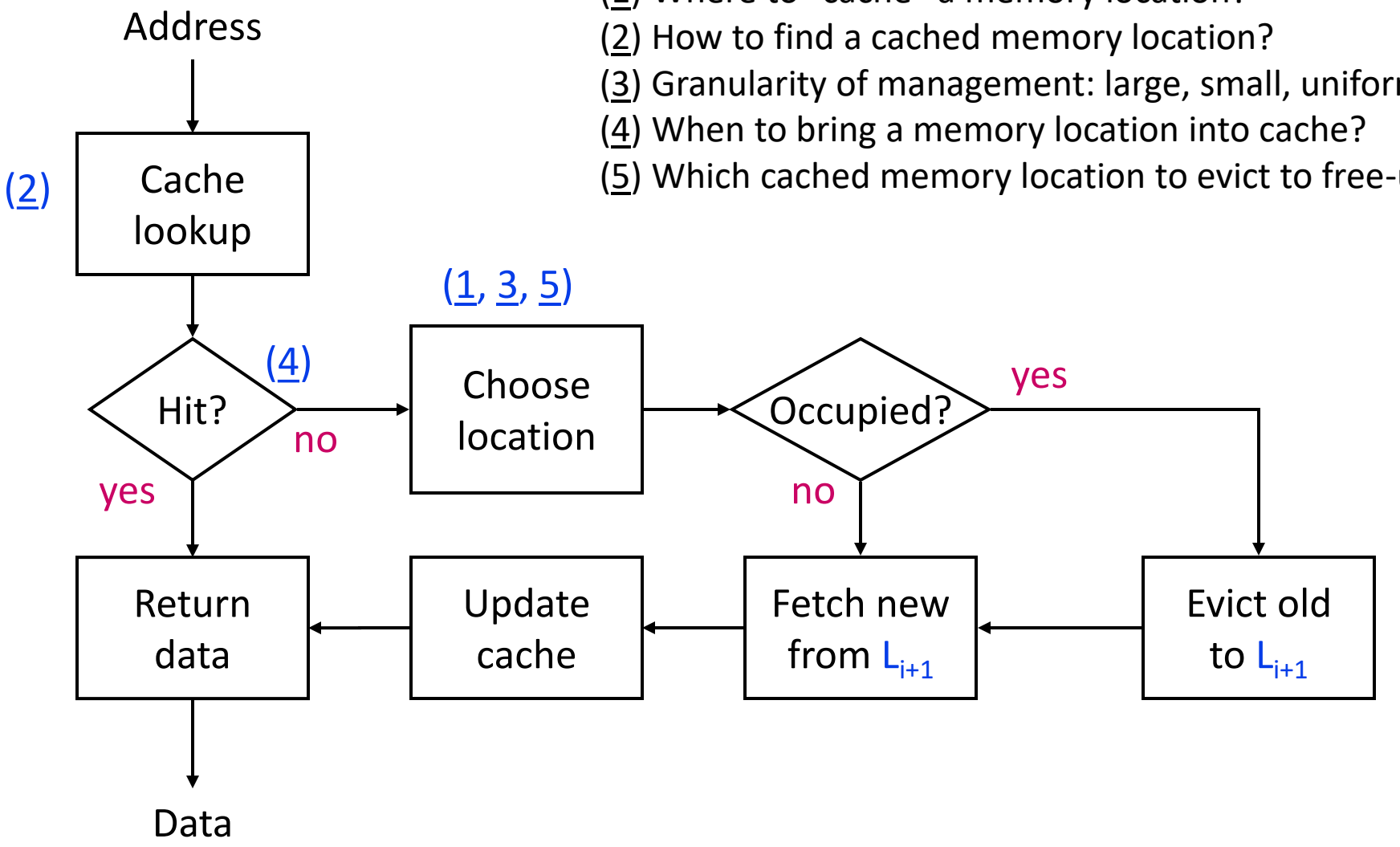
- Basic issues
  - (1) Where to “cache” a memory location?
  - (2) How to find a cached memory location?
  - (3) Granularity of management: large, small, uniform?
  - (4) When to bring a memory location into cache?
  - (5) Which cached memory location to evict to free-up space?

- Optimizations

Basic Operation

Basic issues

(1) Where to "cache" a memory location?
(2) How to find a cached memory location?
(3) Granularity of management: large, small, uniform?
(4) When to bring a memory location into cache?
(5) Which cached memory location to evict to free-up space?



(4): Memory location brought into cache "on-demand"

Basic Cache Parameters

- Let $M = 2^m$ be the size of the address space in bytes

  sample values: 2^{32}, 2^{64}

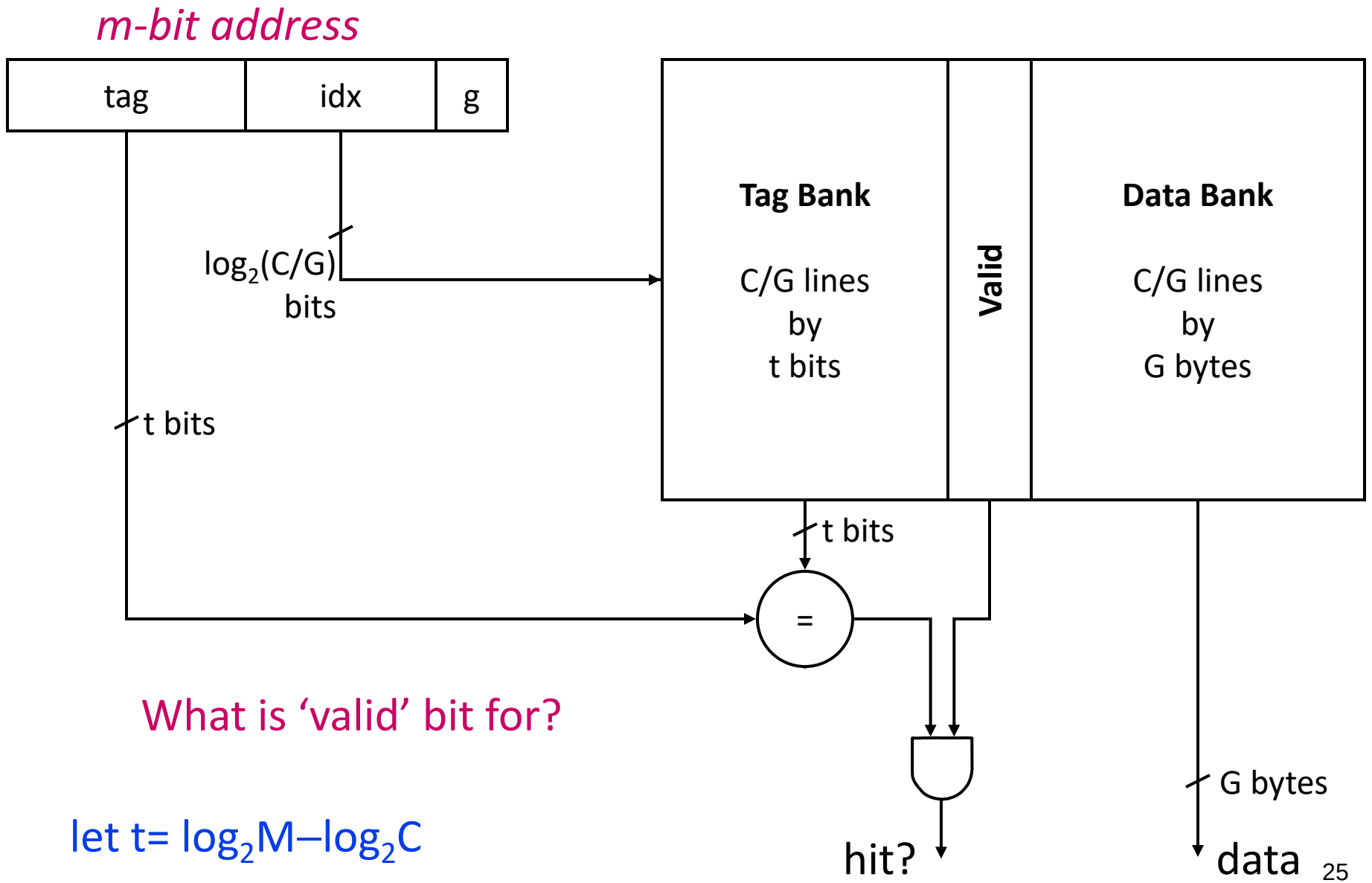
- Let $G = 2^g$ be the cache access granularity in bytes

  sample values: 4, 8 for LW (32-bit arch), LD (64-bit arch)

- Let C be the "capacity" of the cache in bytes

  sample values: 32 KB (L1), 1 MB (L2)

Direct-Mapped Cache (v1)

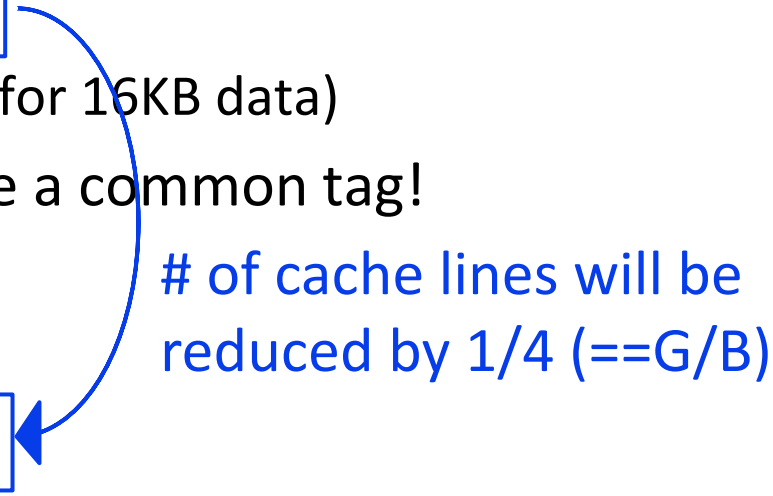


What is 'valid' bit for?

let t= $\log_2 M - \log_2 C$

"Tag" Storage Overhead

- For each cache block of G bytes, must also store additional "t+1" bits, where $t=\log_2 M - \log_2 C$
  - If $M=2^{32}$, G=4 (word), $C=16K=2^{14}$
    - → (32-14) + 1 bits for each 4-byte block
      60% storage overhead (i.e., 9.6KB tag for 16KB data)
- **Solution**: let multiple G-byte words share a common tag!
  - Each B-byte block holds B/G words
  - If $M=2^{32}$, B=16 (block), G=4, C=16K
    - → (32-14) + 1 bits for each 16-byte block
      15% storage overhead

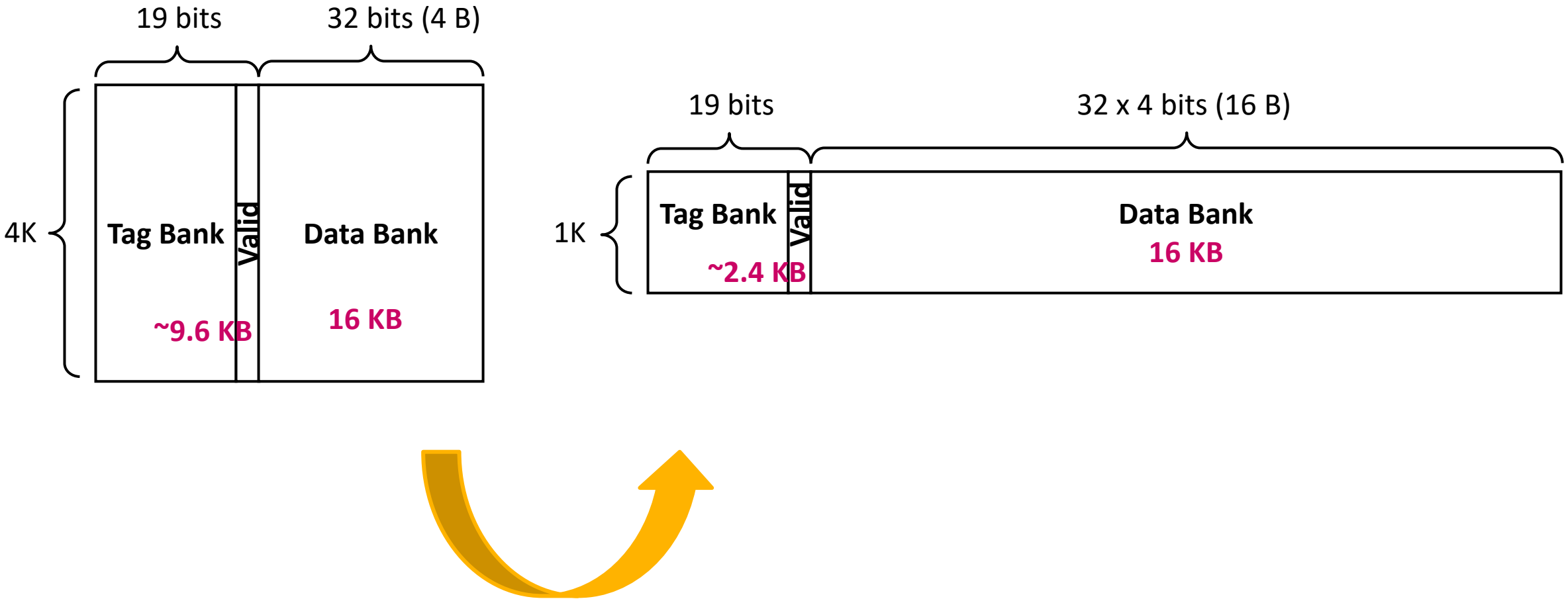


15% of 32 KB L1$ is small (4.8 KB), but 15% of 1 MB L2$ is NOT small (152 KB)

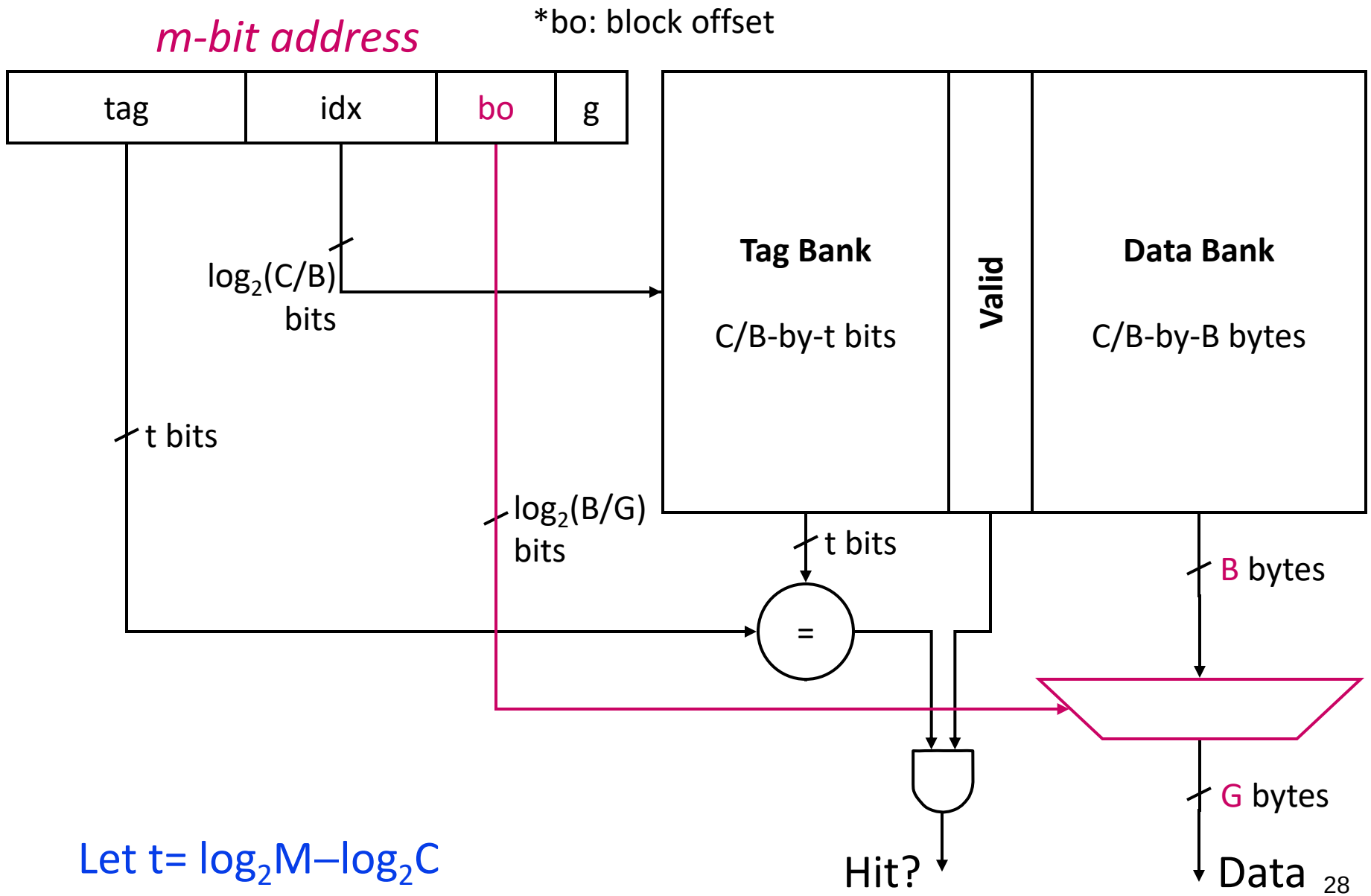
→ Consider using larger block size for lower/larger hierarchies

Hmm. Should I keep increasing B?

Increasing Cache Block Size



Direct-Mapped Cache (Final)



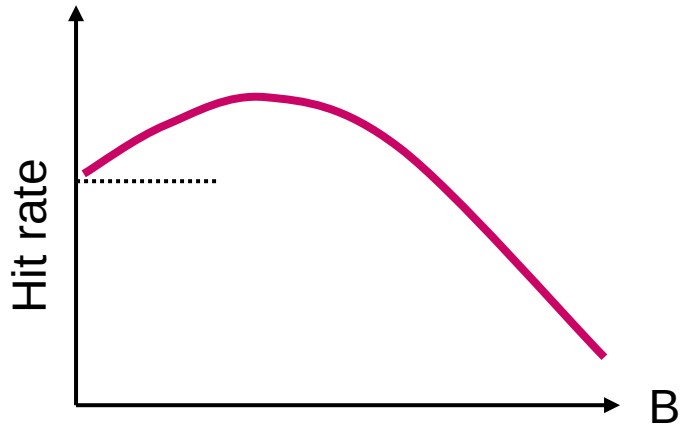
Let t= $\log_2 M - \log_2 C$

Block Size and m_i (miss-rate)

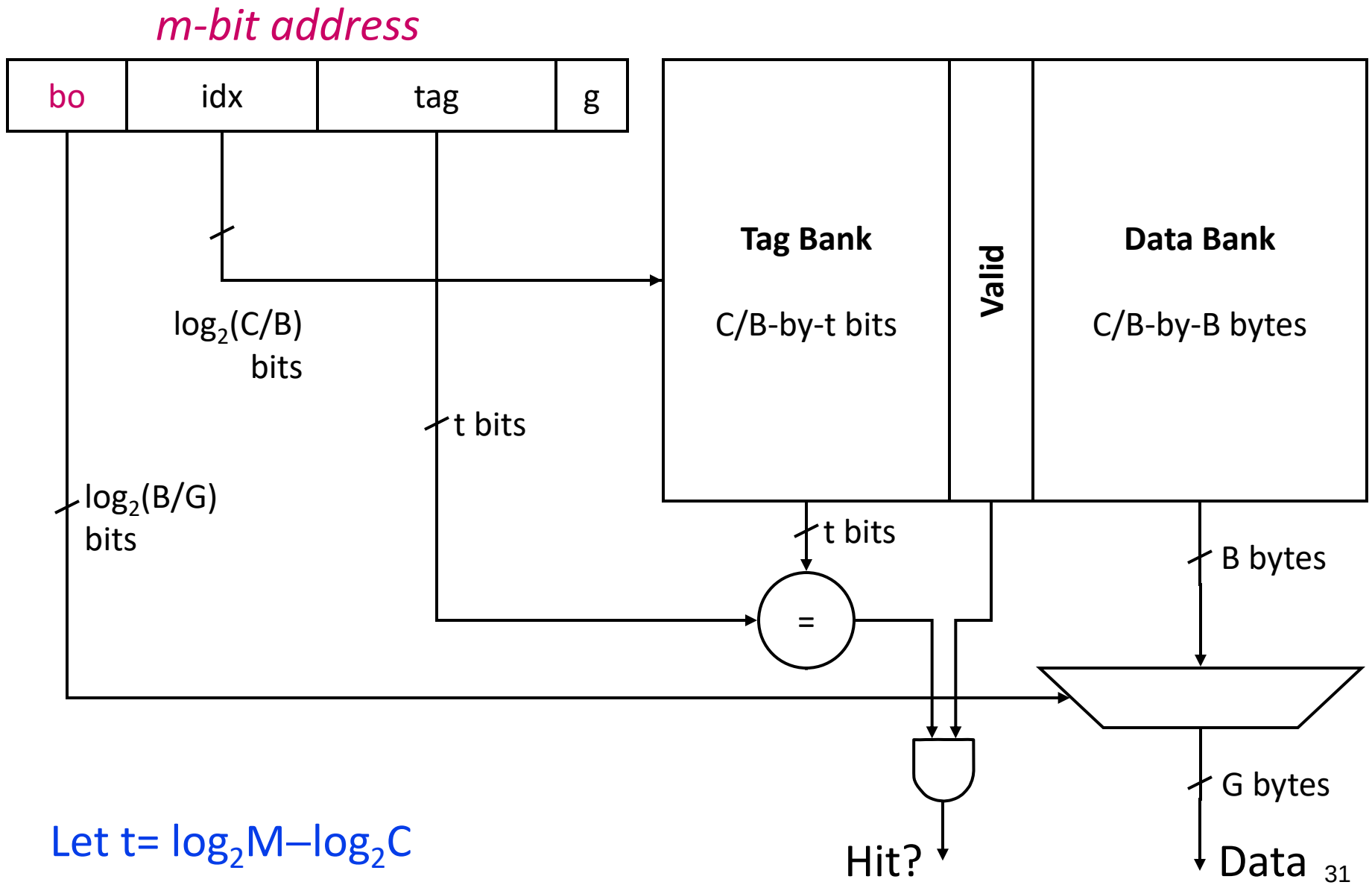
- Bytes that share a common tag are all-in or all-out
- What are the effects of increasing block size (while holding C constant)?
- Good: loading a multi-word block at a time has the effect of prefetching (better exploit spatial locality)
  - Pay miss penalty only once per block
  - Works especially well in instruction caches and regular data access patterns (e.g., sequential array access)
  - Effective up to the limit of spatial locality

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



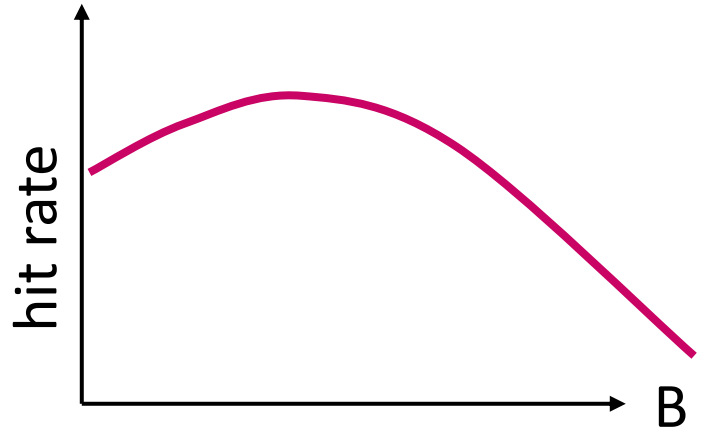
Test yourself: Is this a good design?



Let t= $\log_2 M - \log_2 C$

Block Size and m_i (miss-rate), cont'd

- Bytes that share a common tag are all-in or all-out
- What are the effects of increasing block size (while holding C constant)?
- Good: loading a multi-word block at a time has the effect of prefetching (better exploit spatial locality)
  - Pay miss penalty only once per block
  - Works especially well in instruction caches and regular data access patterns (e.g., sequential array access)
  - Effective up to the limit of spatial locality
- Bad:
  - Can waste capacity
  - Can increase latency and bandwidth
  - Reduces the number of blocks
    → Increases possibility for conflict

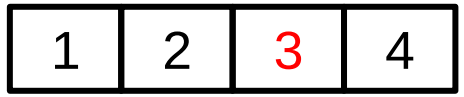


Block Size and T_{i+1}(total access time)

- Loading a large block can increase T_{i+1}
  - If I want the last word on a block, I have to wait for the entire block to be loaded

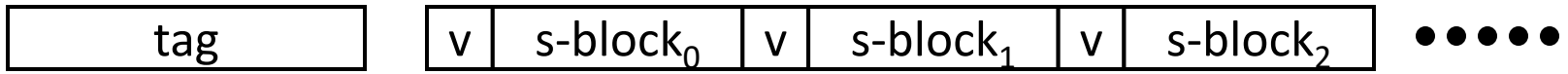
- **Solution 1**: Critical-word first reload
  - L_{i+1} returns the requested word first then rotate around the complete block
  - Supply requested word to pipeline as soon as available (early restart)

Cache block



return order: 3 → 4 → 1 → 2

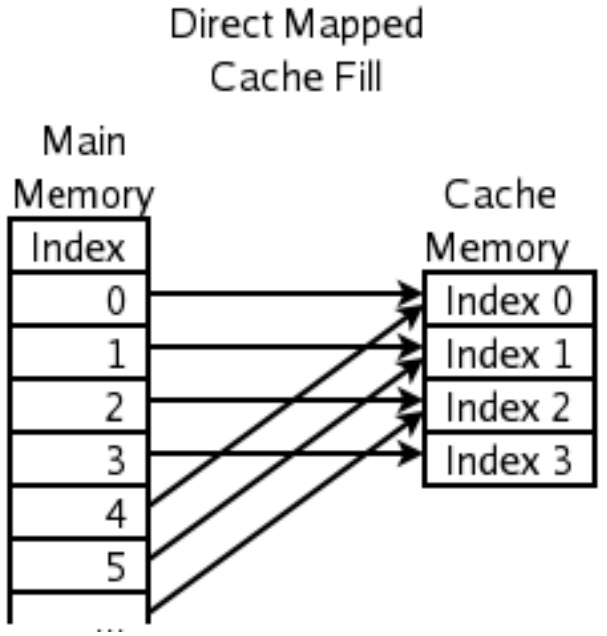
- **Solution 2**: Sub-blocking reload (a.k.a., sectored cache)
  - Individual valid bits for different sub-blocks
  - Reload only requested sub-block on demand
  - Note: all sub-blocks share common tag

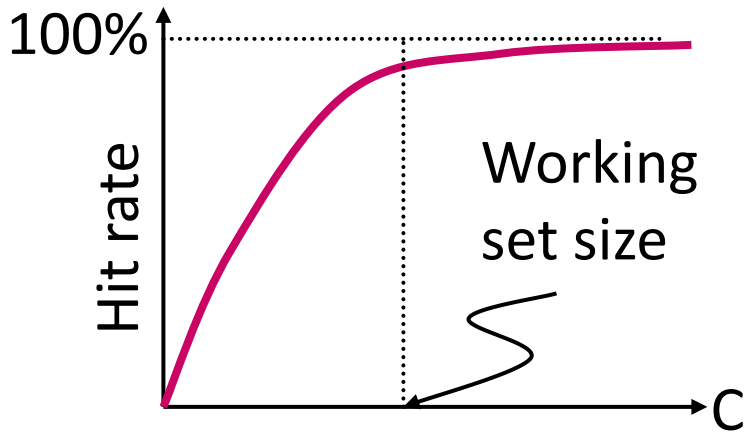


Conflicts in Direct-Mapped Cache

- C bytes of storage divided into C/B blocks
- A block of memory is mapped to one particular cache block according to the address' block index field
- All addresses with the same block index map to the same cache block
  - 2^t such addresses; can cache only one such block at a time
  - Even if C > working set size, collision is possible
  - Given 2 random addresses, chance for collision is 1/(# of cache blocks) = 1/(C/B)

    Notice likelihood for collision decreases with increasing # of cache blocks (C/B)

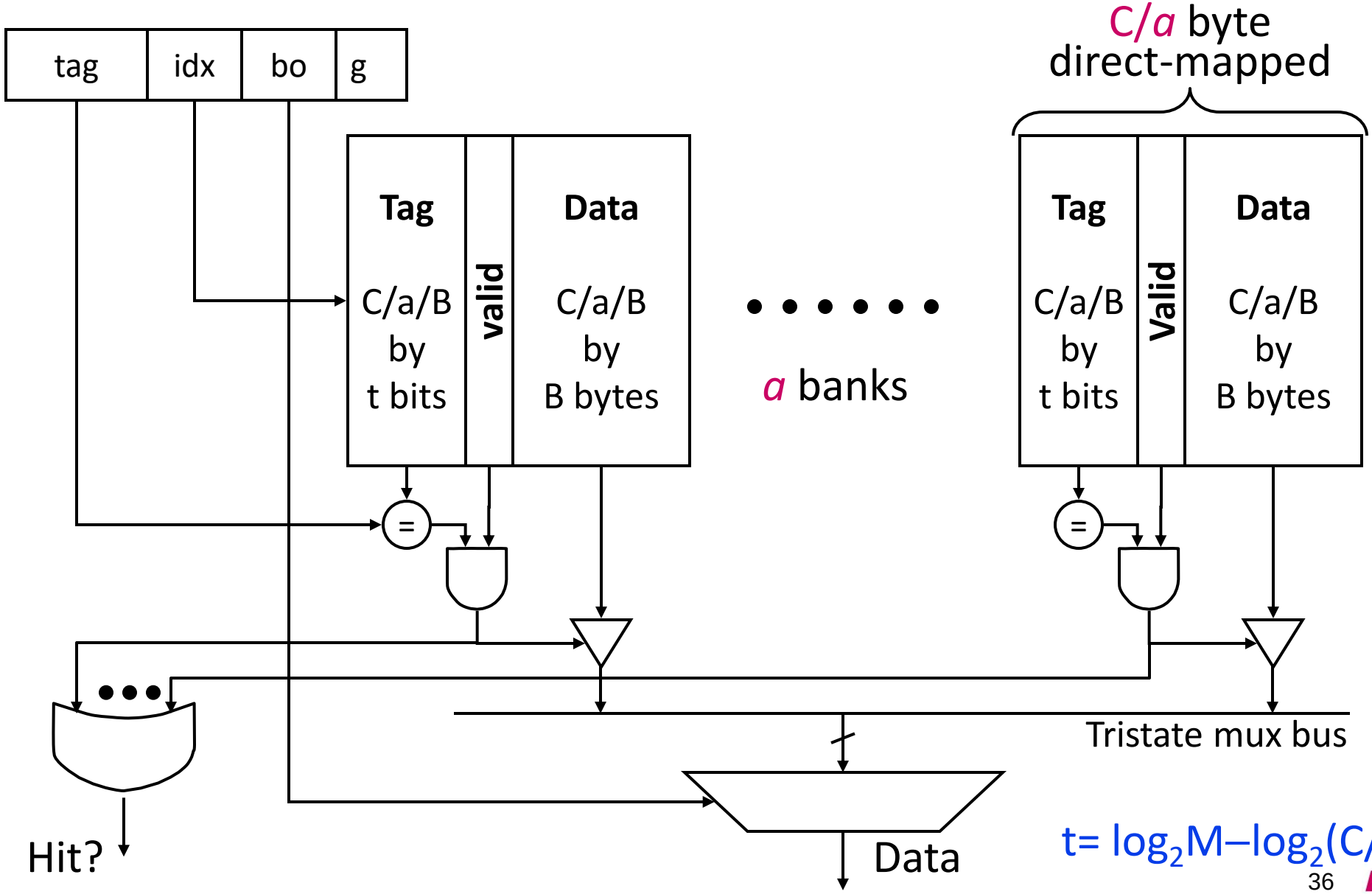




Many is better than one.

How about using multiple direct-mapped caches together?

"*a*"-way Set-Associative Cache



$t = \log_2 M - \log_2 (C/a)$

"*a*"-way Set-Associative Cache

- C bytes of storage divided into *a* banks each with C/*a*/B blocks
  (can be thought of as a multiple (*a*) direct-mapped caches but each is smaller)

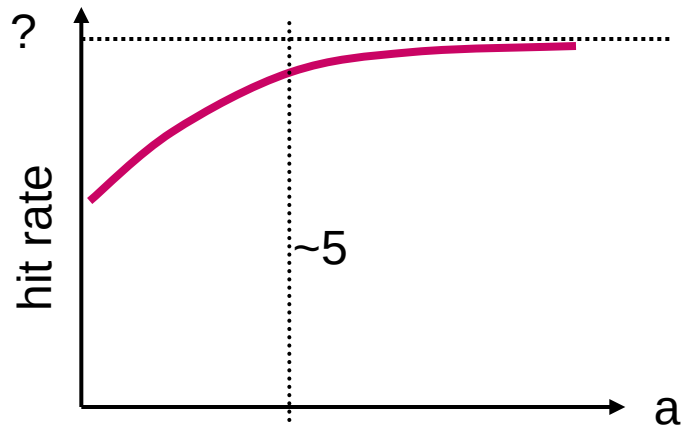
- Requires *a* comparators and *a*-to-1 multiplexer

- An address is mapped to a particular block in a bank according to its block index field, but there are *a* such banks (together known as a "set")

- All addresses with the same block index field map to a common "set" of cache blocks
  - 2^t such addresses; can contain *a* such blocks at a time
  - Higher-degree of associatively → Fewer conflicts

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 (W: working set) or when C/B is small

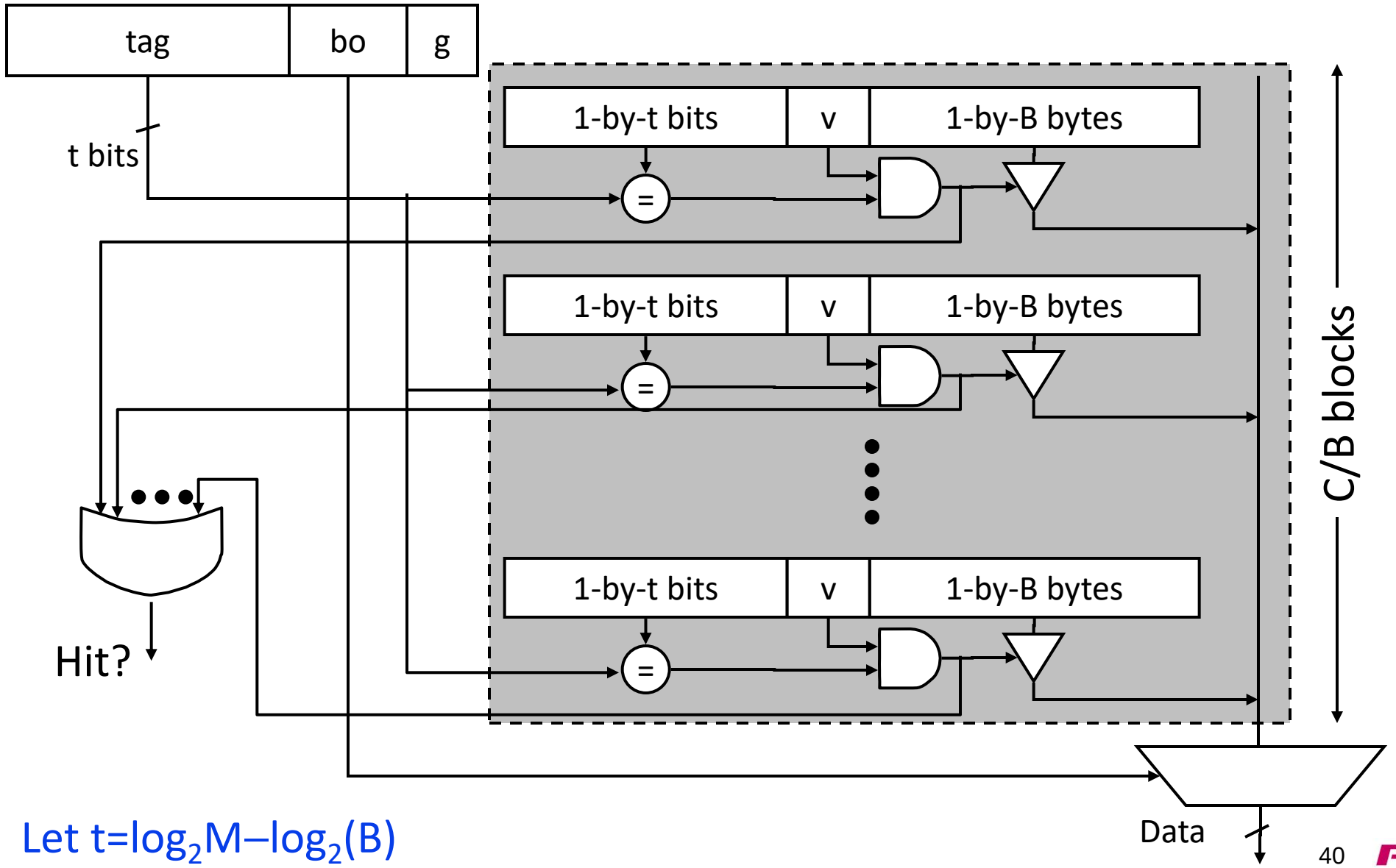


Replacement Policy for Set Associative Caches

- A new cache block can evict any of the cached memory block in the same set, which one?
  - **Pick the one that is Least Recently Used (LRU)**

    simple function for a=2, complicated for a>2 (why?)
  - Pick any one except the most recently used
  - ~~Pick the most recently used one~~
  - ~~Pick one based on some part of the address bits~~
  - Pick the one that you will need again furthest in the future (but how?)
  - Pick a (pseudo) random one
- But, replacement policy only has a second-order effect
  - If you actively use less than a blocks in a set, any sensible replacement policy will quickly converge (= become similar)
  - If you actively use more than a blocks in a set, no replacement policy can help you

Fully Associative Cache: $a \equiv C/B$



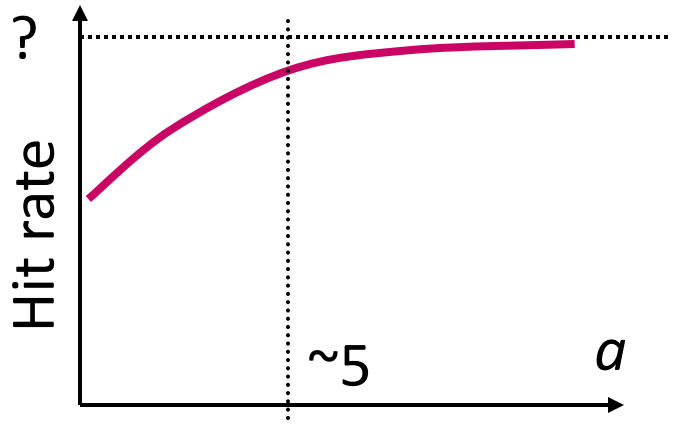
Let $t = \log_2 M - \log_2(B)$

Fully Associative Cache: $a \equiv C/B$

- So, it is a "**Content-addressable (CAM)**" memory
  - Not your regular SRAM. But, critical for some uArchitecture (e.g., TLB in a later lecture)
  - Present "tag" → Search ALL entries → If found, return "data"
  - No index bits used in lookup (=**search every index simultaneously**)
- Any address can go into any of the C/B blocks
  - if C > working set size, no collisions
- Requires one comparator per cache block, a huge multiplexer, and many long wires

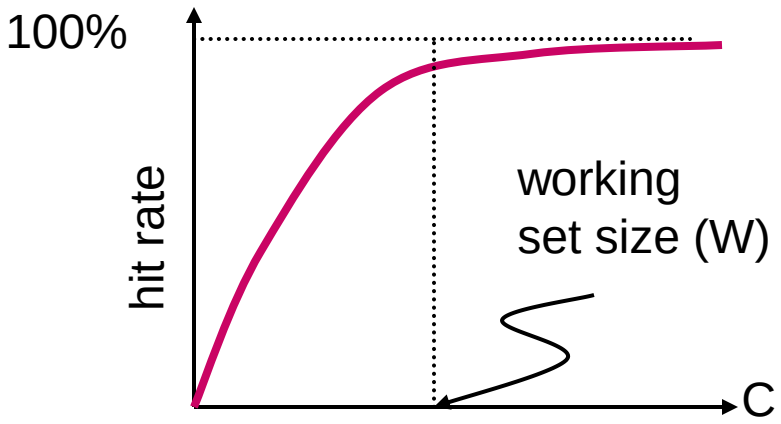
Too expensive/difficult for more than 32~64 blocks at L1 latencies

- Fortunately, there is little reason for very large fully associative caches
- For any reasonably large values of C/B, a=4~5 is as good as a=C/B typically



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



Recap: Basic Cache Parameters

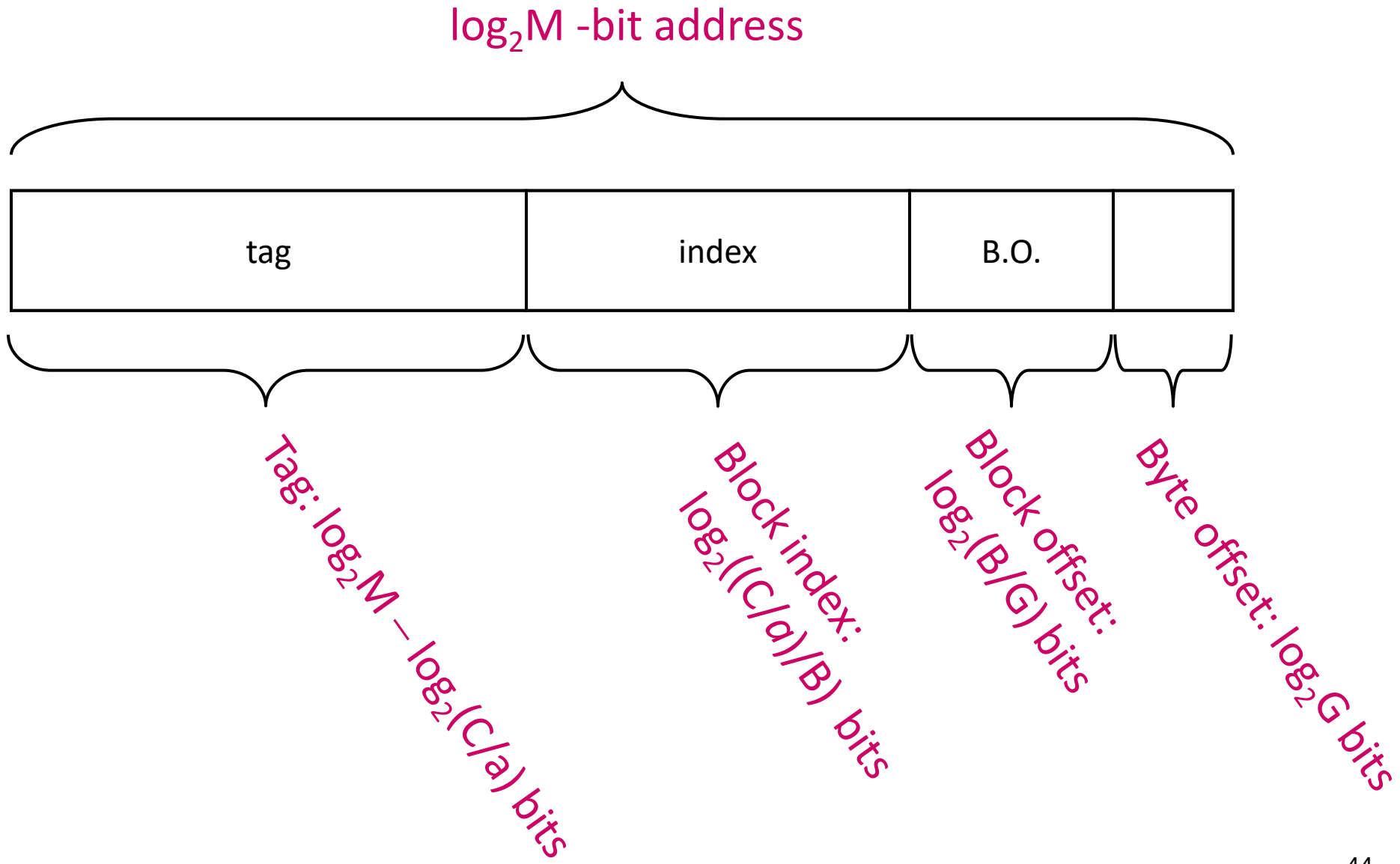
ISA

- Let $M = 2^m$ be the size of the address space in bytes
  sample values: 2^{32}, 2^{64}
- Let $G = 2^g$ be the cache access granularity in bytes
  sample values: 4 (integer), 8 (double)

Implementation

- Let C be the "capacity" of the cache in bytes
  sample values: 32 KB (L1), 1 MB (L2)
- Let $B = 2^b$ be the "block size" of the cache in bytes
  sample values: 16, 64 (typical in modern CPUs), 128
- Let a be the "associativity" of the cache
  sample values: 1, 2, 4, 5(?),... "C/B"

Recap: Address Fields



Recap: Cache Misses – The 3 C's

- **Compulsory (or cold)** miss: item has never been in the cache
- **Capacity** miss: item has been in the cache, but space was tight and it was forced out
- **Conflict** miss: item was in the cache, but the cache was not associative enough, so it was forced out
- Think about how these changes to cache design affects different types of misses
  - Larger or smaller cache size
  - Larger or smaller block size
  - Higher or lower set-associativity

Adapted from CSE410 at U. of Washington

Question?

Announcements

- Textbook reading: P&H Ch. 5.1 – 5.4
- (Optional) paper reading: “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” N. P. Jouppi, ISCA’90