The Memory Hierarchy

Silvina Hanono Wachman
Computer Science & Artificial Intelligence Lab
M.I.T.

Memory Technologies

	Capacity	Latency	Cost/GB
Register	100s of bits	20 ps	$$$$
SRAM	~10 KB-10 MB	1-10 ns	~$1000
DRAM	~10 GB	80 ns	~$10
Flash*	~100 GB	100 us	~$1
Hard disk*	~1 TB	10 ms	~$0.1

Processor Datapath

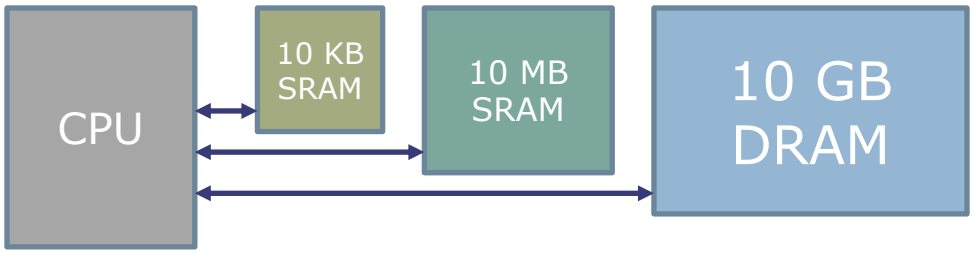
Memory Hierarchy

I/O subsystem

* non-volatile (retains contents when powered off)

- Different technologies have vastly different tradeoffs
- Size is a fundamental limit, even setting cost aside:
  - Small + low latency or
  - Large + high-latency
- Can we get best of both worlds? (large, fast, cheap)

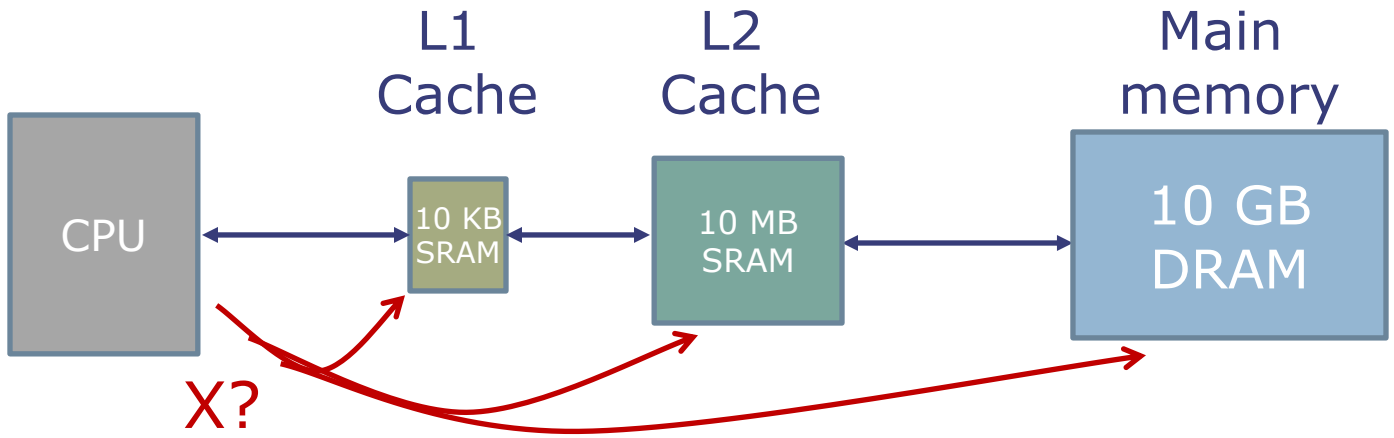
Exposed Memory Hierarchy



- Attach a variety of storage alternatives (registers, SRAM, and DRAM) of varying sizes to the CPU
- Tell programmers: "Use them cleverly"
- Implies that you either:
  1. Modify ISA to provide different instructions for accessing the different memory elements.
  2. Allocate specific regions of address space for each type of memory.
- What happens if you want to increase the size of your SRAM?

Not really useful in practice

Implicit Memory Hierarchy

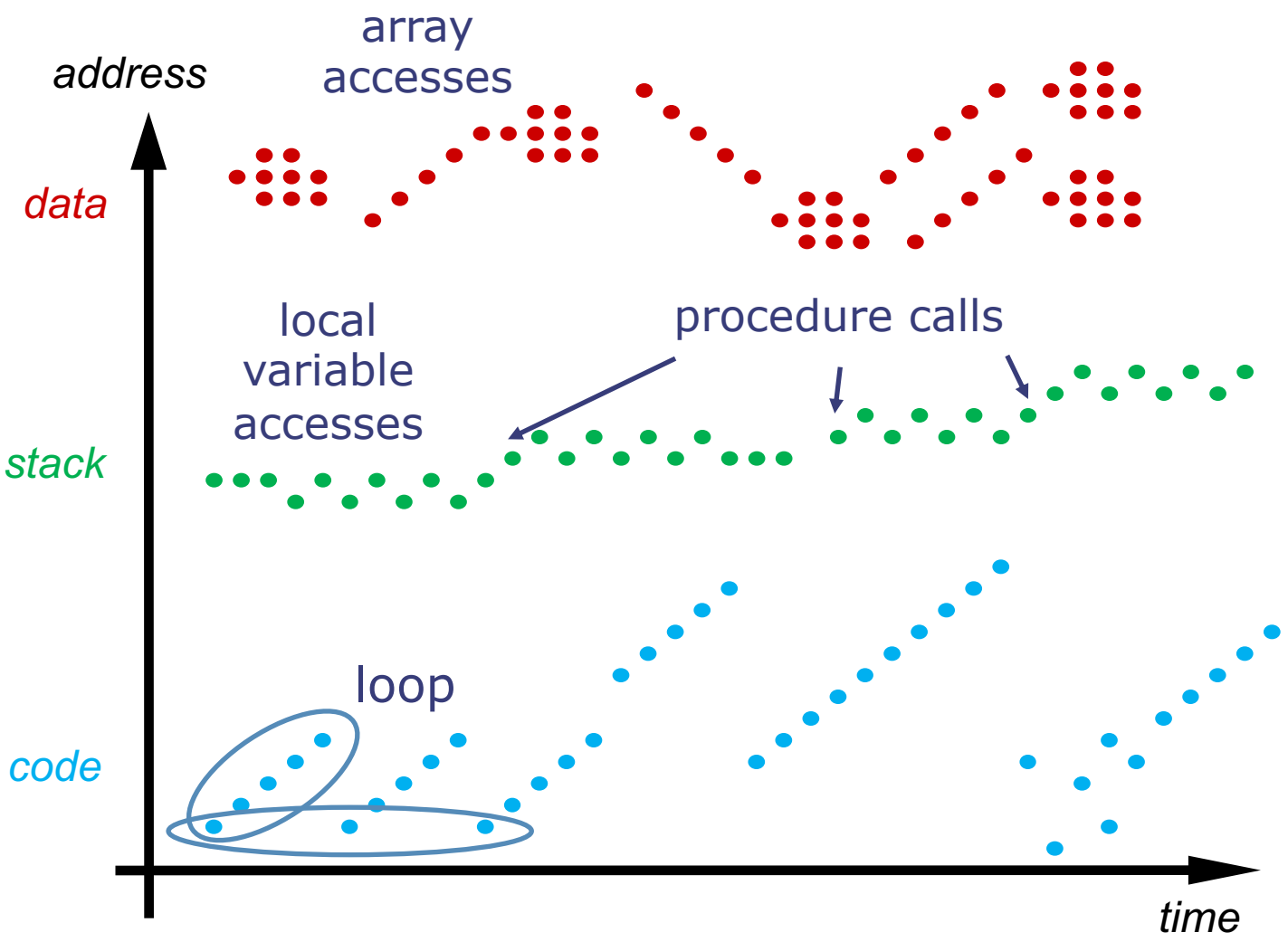


- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns
- CPU effectively sees large, fast memory if values are found in cache most of the time.

Why Caches Work

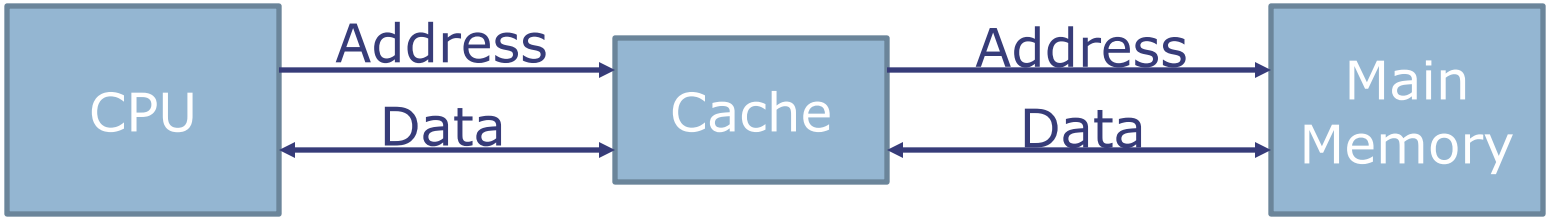
- Two predictable properties of memory accesses:
  - Temporal locality: If a location has been accessed recently, it is likely to be accessed (reused) in the near future
  - Spatial locality: If a location has been accessed recently, it is likely that nearby locations will be accessed in the near future

Typical Memory Access Patterns



Caches

- Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations



- Processor sends accesses to cache. Two options:
  - Cache hit: Data for this address in cache, returned quickly
  - Cache miss: Data not in cache
    - Fetch data from memory, send it back to processor
    - Retain this data in the cache (replacing some other data)

Processor must deal with variable access-time of memory

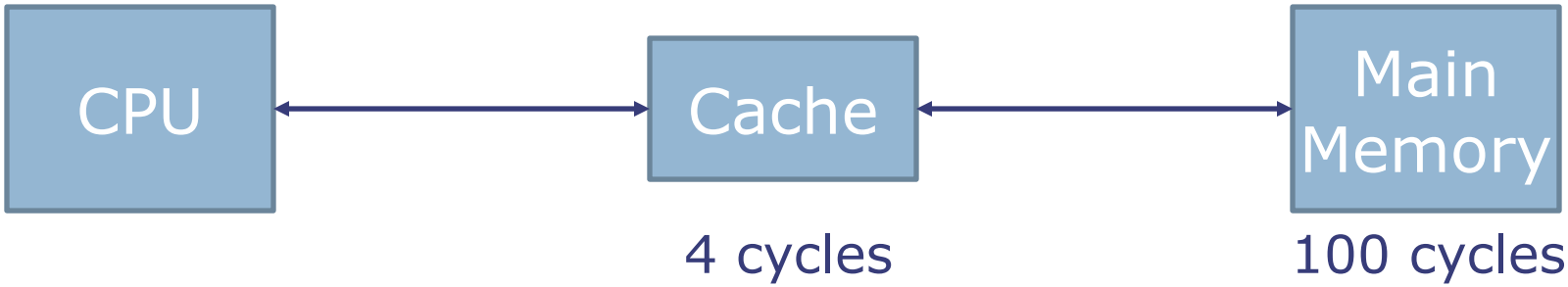
Cache Metrics

- Hit Ratio: $HR = \frac{hits}{hits + misses} = 1 - MR$
- Miss Ratio: $MR = \frac{misses}{hits + misses} = 1 - HR$
- Average Memory Access Time (AMAT):

AMAT = HitTime + MissRatio × MissPenalty

Cache design is all about reducing AMAT

Example: How High of a Hit Ratio?



AMAT without a cache = 100 cycles

Latency with cache: Hit = 4 cycles; Miss = 104 cycles

What hit ratio do we need to break even?

$$100 = 4 + (1 - HR) \times 100 \Rightarrow HR = 4\%$$

should be easy to achieve

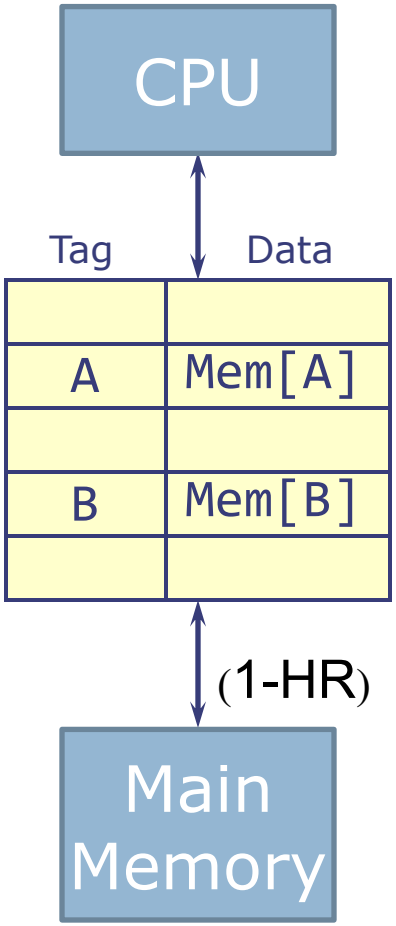
AMAT for different hit ratios:

$$HR=50\% \Rightarrow AMAT = 4 + (1 - .50) \times 100 = 54$$
$$HR=90\% \Rightarrow AMAT = 4 + (1 - .90) \times 100 = 14$$
$$HR=99\% \Rightarrow AMAT = 4 + (1 - .99) \times 100 = 5$$

Can we achieve such high HR?

With high HR caches can dramatically improve AMAT

Basic Cache Algorithm (Reads)



On reference to Mem[X], look for X among cache tags

HIT: X = Tag(i) for some cache line i

Return Data(i)

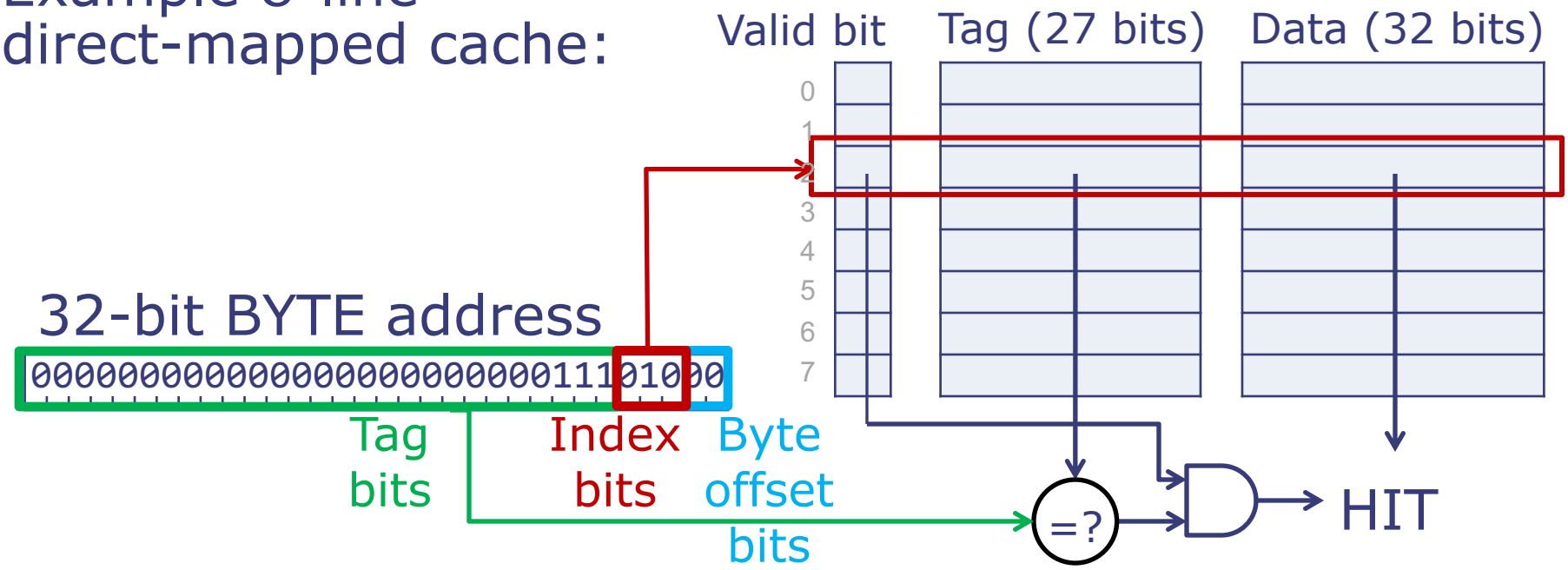
MISS: X not found in Tag of any cache line

1. Read Mem[X]
2. Return Mem[X]
3. Select a line k to hold Mem[X]
4. Write Tag(k)=X, Data(k) = Mem[X]

Q: How do we "search" the cache?

Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with 2^W lines):
  - Index into cache with W address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100

TAG: 0x40
INDEX: 0x3
OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

INDEX: 0x2 → tag mismatch
→ MISS

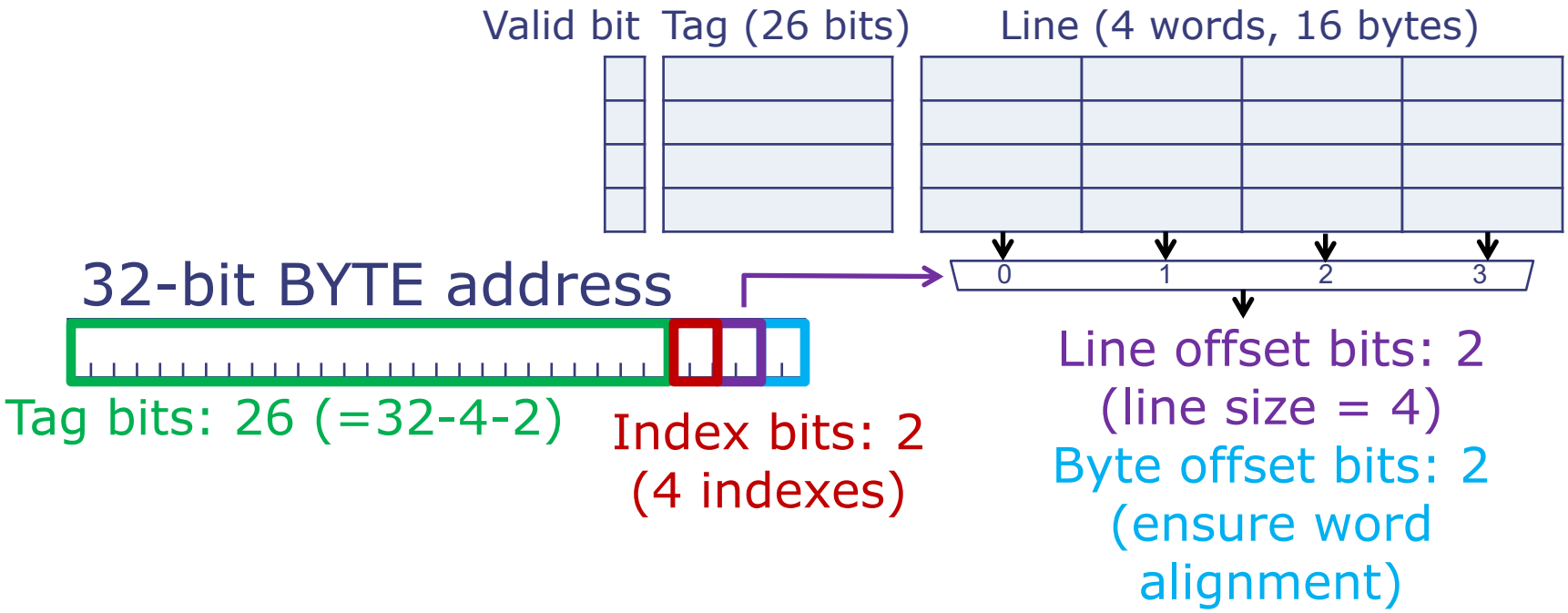
	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	1	0x000058	0x00000007
3	1	0x000040	0x42424242
4	0	0x000007	0x6FBA2381
	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

Part of the address (index bits) is encoded in the location
Tag + Index bits unambiguously identify the data's address

Further exploiting spatial locality

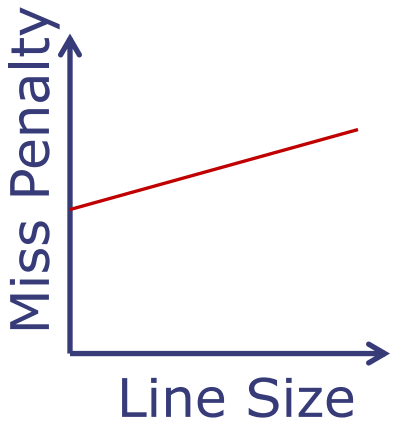
- Store multiple words per data line
  - Main advantage: Exploit spatial locality
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer lines in the cache (more conflicts)

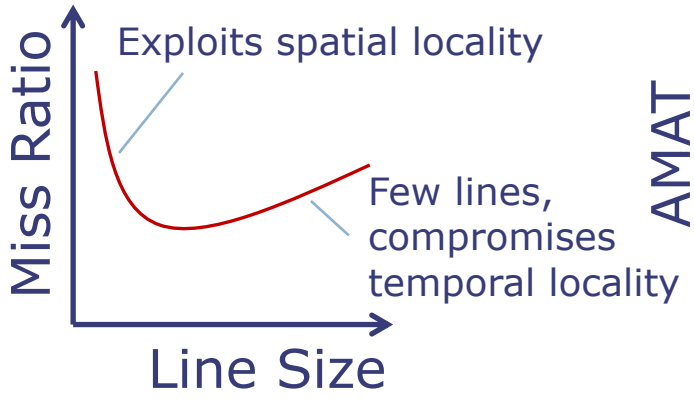
- Example: 4-word line, 16-word direct-mapped cache

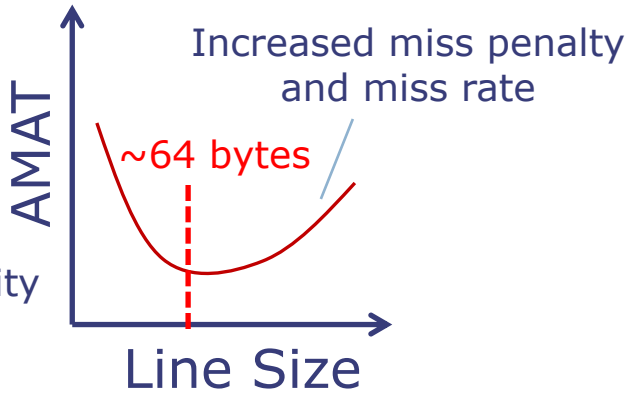


Line Size Tradeoffs

- Larger line sizes…
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the line from memory
  - Can increase the average hit time and miss ratio
- AMAT = HitTime + MissPenalty*MissRatio







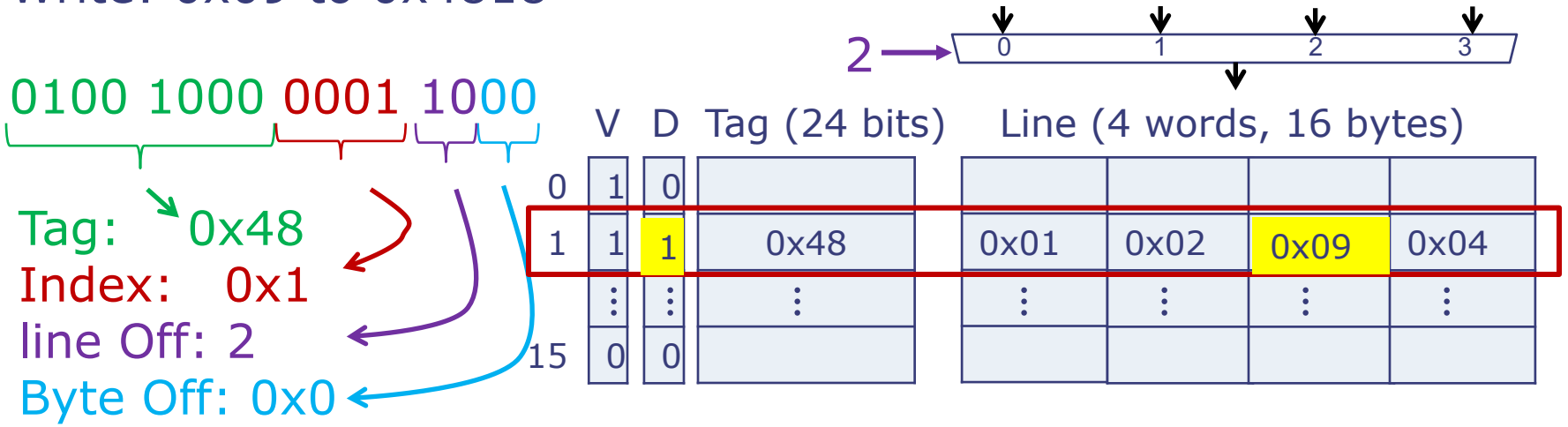
Write Policy

1. Write-through: CPU writes are cached, but also written to main memory immediately; Memory always holds current contents
2. Write-back: CPU writes are cached, but not written to main memory until we replace the line. Memory contents can be "stale"
   - Upon replacement, a modified cache line must first be written back to memory before loading the new cache line
   - To avoid unnecessary writebacks, a Dirty bit is added to each cache line to indicate if the value has been modified since it was loaded from memory
3. No cache write on a Write-miss: On a cache miss, write is sent directly to the memory without a cache write

Write-back is the most commonly used policy, because it saves cache-memory bandwidth

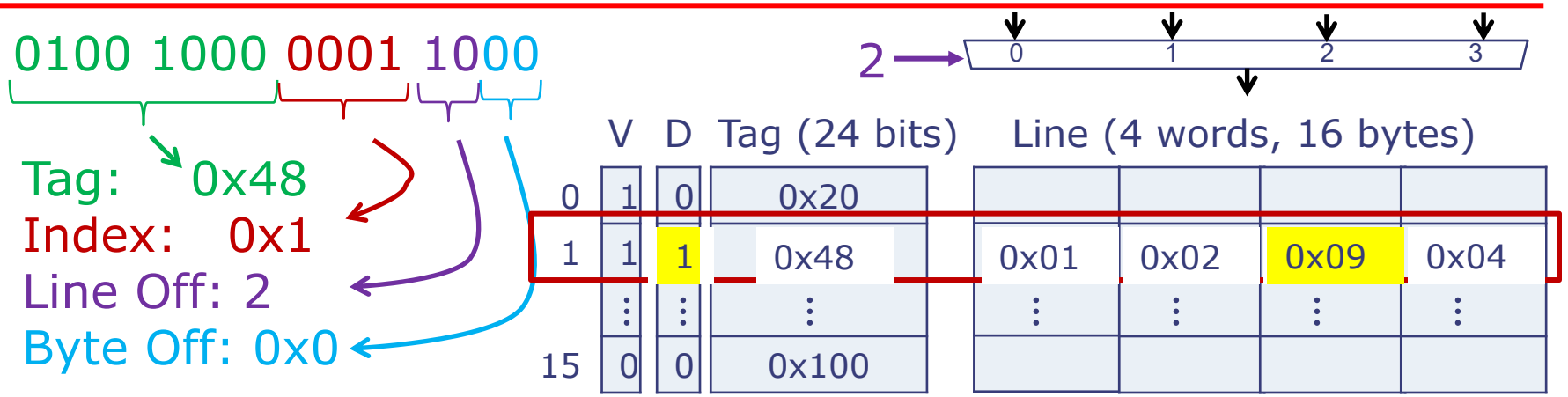
Example: Cache Write-Hit

16-line direct-mapped cache → 4 index bits
line size = 4 → 2 line offset bits
Write Policy = Write Back
Write: 0x09 to 0x4818



D=1: cache contents no longer match main memory so write back line to memory upon replacement

Example: Cache Write-Miss



Write: 0x09 to 0x4818

1. Tags don't match -> Miss
   - D=1: Write cache line 1 (tag = 0x280: addresses 0x28010-0x2801C) back to memory
   - If D=0: Don't need to write line back to memory.

2. Load line (tag = 0x48: addresses 0x4810-0x481C) from memory

3. Write 0x09 to 0x4818 (line offset 2), set D=1.

Direct-Mapped Cache Problem: Conflict Misses

Loop A: Code at 1024, data at 37

Word Address	Cache Line index	Hit/ Miss
1024	0	HIT
37	37	HIT
1025	1	HIT
38	38	HIT
1026	2	HIT
39	39	HIT
1024	0	HIT
37	37	HIT
…		

Assume:

- 1024-line DM cache
- line size = 1 word
- Consider looping code, in steady state
- Assume WORD, not BYTE, addressing

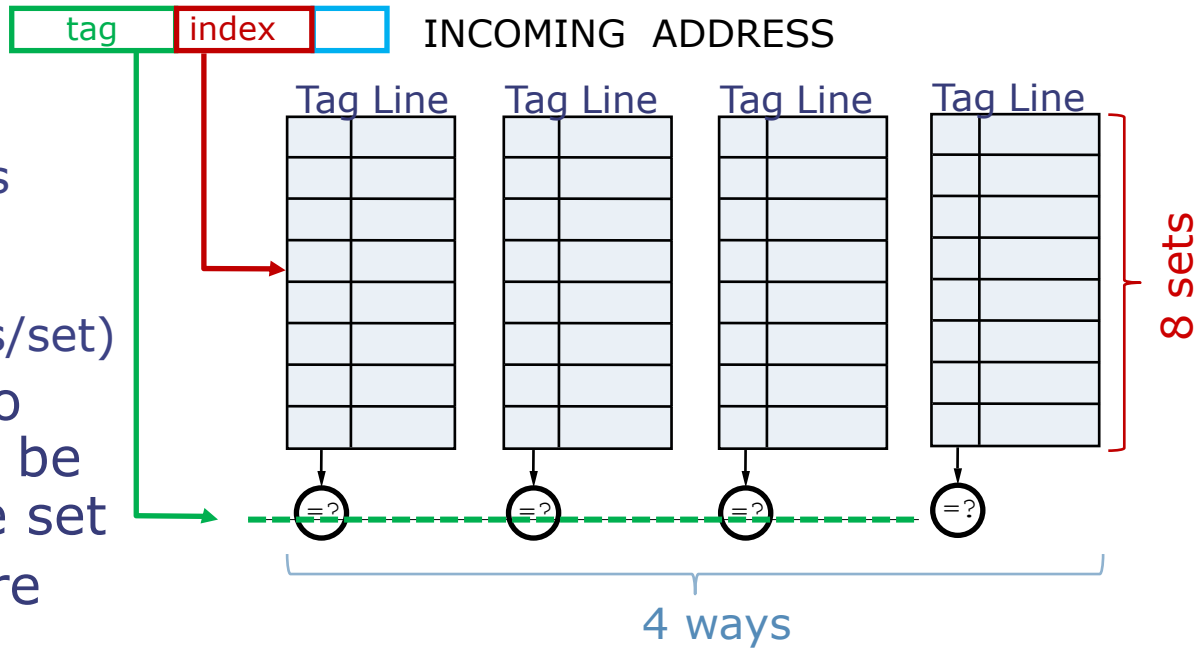
Loop B: Code at 1024, data at 2048

1024	0	MISS
2048	0	MISS
1025	1	MISS
2049	1	MISS
1026	2	MISS
2050	2	MISS
1024	0	MISS
2048	0	MISS
...		

Inflexible mapping (each address can only be in one cache location) → Conflict misses!

N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
- Nomenclature:
  - # Rows = # Sets
  - # Columns = # Ways
  - Set size = #ways = "set associativity" (e.g. 4-way → 4 lines/set)
- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel

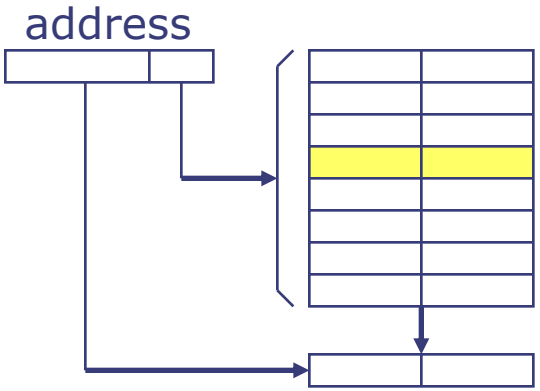


- Fully-associative cache: Number of ways = Number of lines
  - Any address can be in any line → No conflict misses, but expensive

Associativity Implies Choices

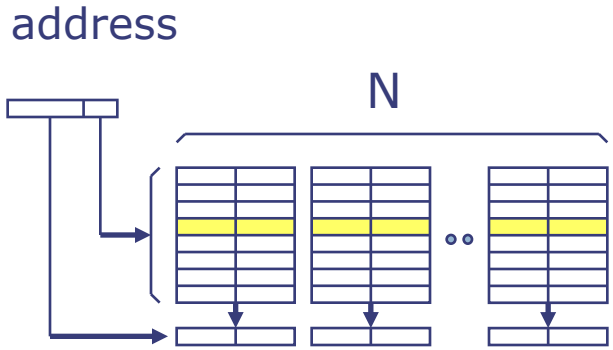
Issue: Replacement Policy

Direct-mapped



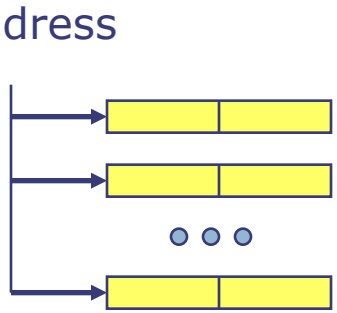
- Compare addr with only one tag
- Location A can be stored in exactly one cache line

N-way set-associative



- Compare addr with N tags simultaneously
- Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

Fully associative



- Compare addr with each tag simultaneously
- Location A can be stored in any cache line

Replacement Policies

- Least Recently Used (LRU): Replace the line that was accessed furthest in the past
  - Works well in practice
  - Need to keep ordered list of N items → N! orderings
    → $O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" + complex logic
  - Caches often implement cheaper approximations of LRU

- Other policies:
  - First-In, First-Out (least recently replaced)
  - Random: Choose a candidate at random
    - Not very good, but does not have adversarial access patterns

Summary: Cache Tradeoffs

Caches allow memory to appear like a large, fast, and cheap memory.

$$AMAT = HitTime + MissRatio \times MissPenalty$$

Design tradeoffs can be made in cache size, line size, associativity, replacement policy, write policy.

Thank you!
This material will be included in Quiz 2

Next lecture: Implementing Caches

Take home

- Compare the hit rates for the following 3 caches: Direct Mapped, 2-Way Set Associative, and Fully Associative. Assume each has 8 (4 byte) words. Assume that the access pattern is repeatedly: 0, 16, 4, 36.