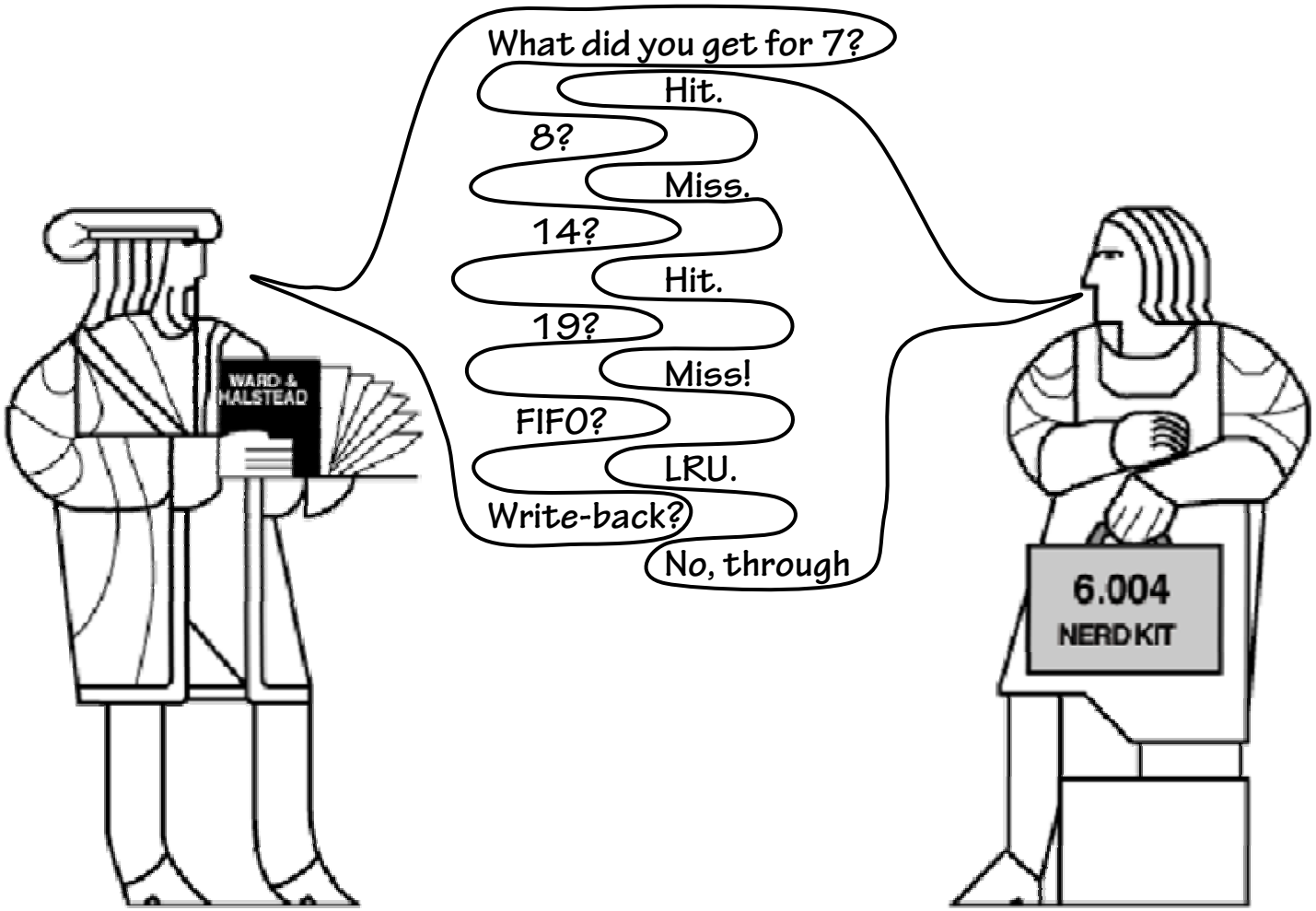
Cache Issues



General Cache Principle

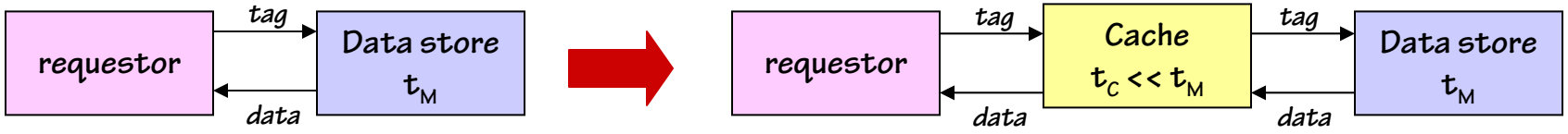
SETUP:

- Requestor making a stream of lookup requests to a data store.
- Some observed predictability – e.g. locality – in reference patterns.

TRICK:

- Small, fast *cache* memory between requestor & data store
- Cache holds <tag, value> pairs most likely to be requested

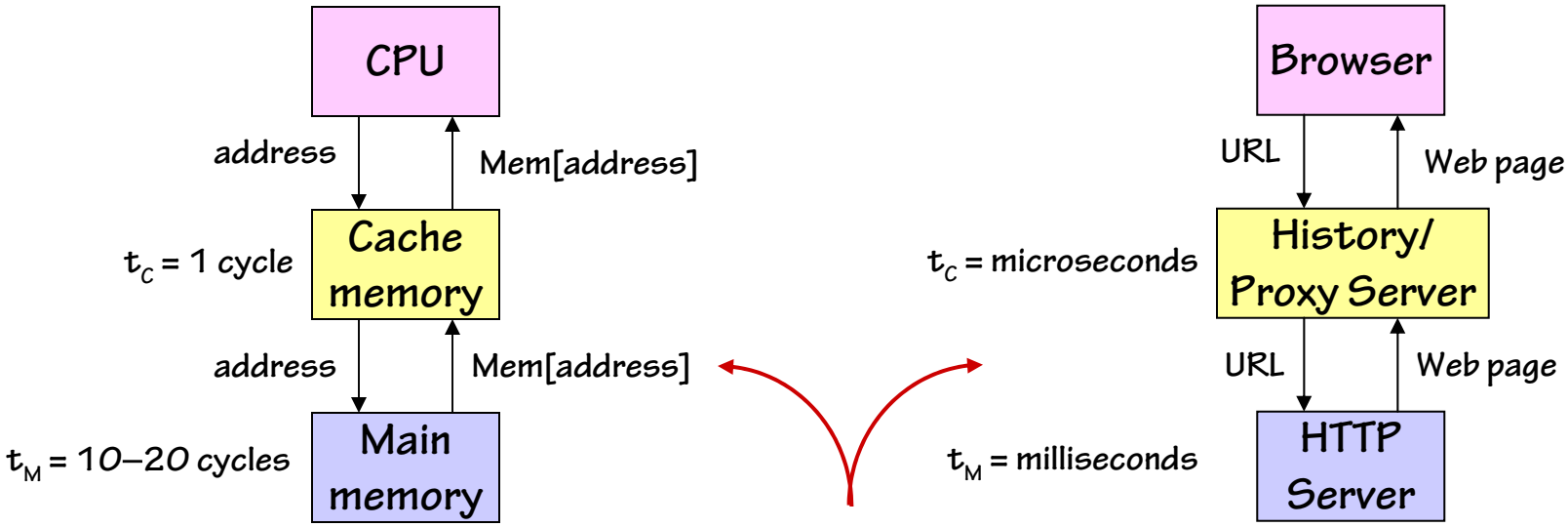
This request is made only when tag can't be found in cache, i.e., with probability $1-\alpha$, where α is the probability that the cache has the data (a "hit")



Access time: t_M

Average access time: $t_C + (1-\alpha)t_M$

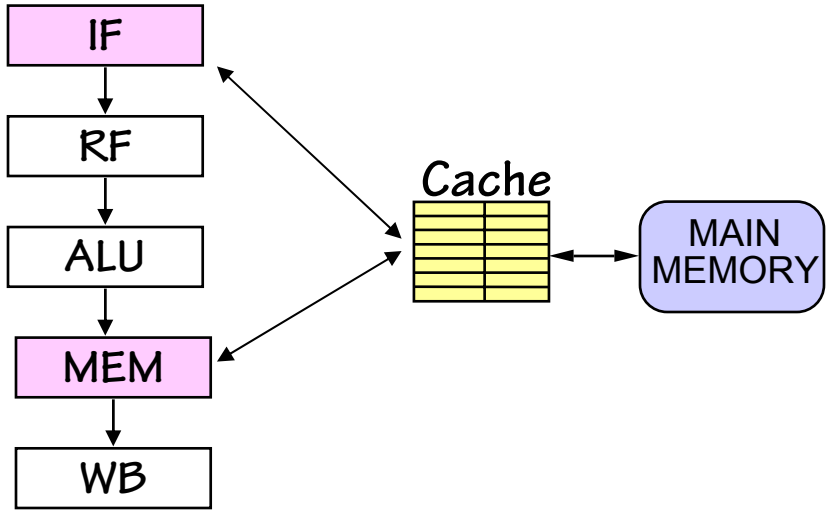
Cache Applications



Memory system: assuming a hit ratio of 95% and t_M=20 cycles, average access time is 2 cycles (urk!). Cache is worthwhile, but doesn't eliminate problem.

World-Wide Web: links, back/forward navigation give good predictability. Bonus: good cache performance reduces effective t_M by reducing contention.

Memory System Caches



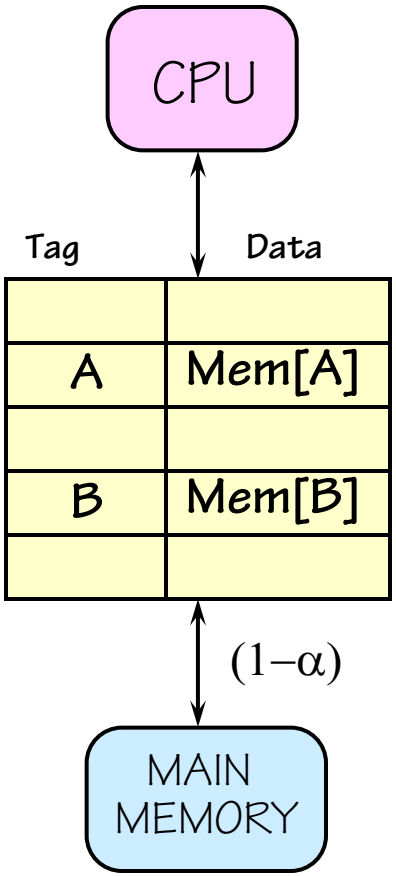
Variants:
- Separate I & D caches
- Level 1 cache on CPU chip, Level 2 cache off-chip.

Problem: Memory access times (IF, MEM) limit Beta clock speed.

Solution: Use cache for both instruction fetches and data accesses:

- assume HIT time for pipeline clock period;
- STALL pipe on misses.

Basic Cache Algorithm



ON REFERENCE TO Mem[X]:

Look for X among cache tags...

HIT: *X = TAG(i) , for some cache line i*

- READ: return DATA(i)
- WRITE: change DATA(i); Start Write to Mem(X)

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X] (Allocation)
- READ: Read Mem[X]
  Set TAG(k)=X, DATA(K)=Mem[X]
- WRITE: Start Write to Mem(X)
  Set TAG(k)=X, DATA(K)= new Mem[X]

Cache Design Issues

Associativity – a basic tradeoff between

- Parallel Searching (expensive) vs
- Constraints on which addresses can be stored where

Block Size:

- Amortizing cost of tag over multiple words of data

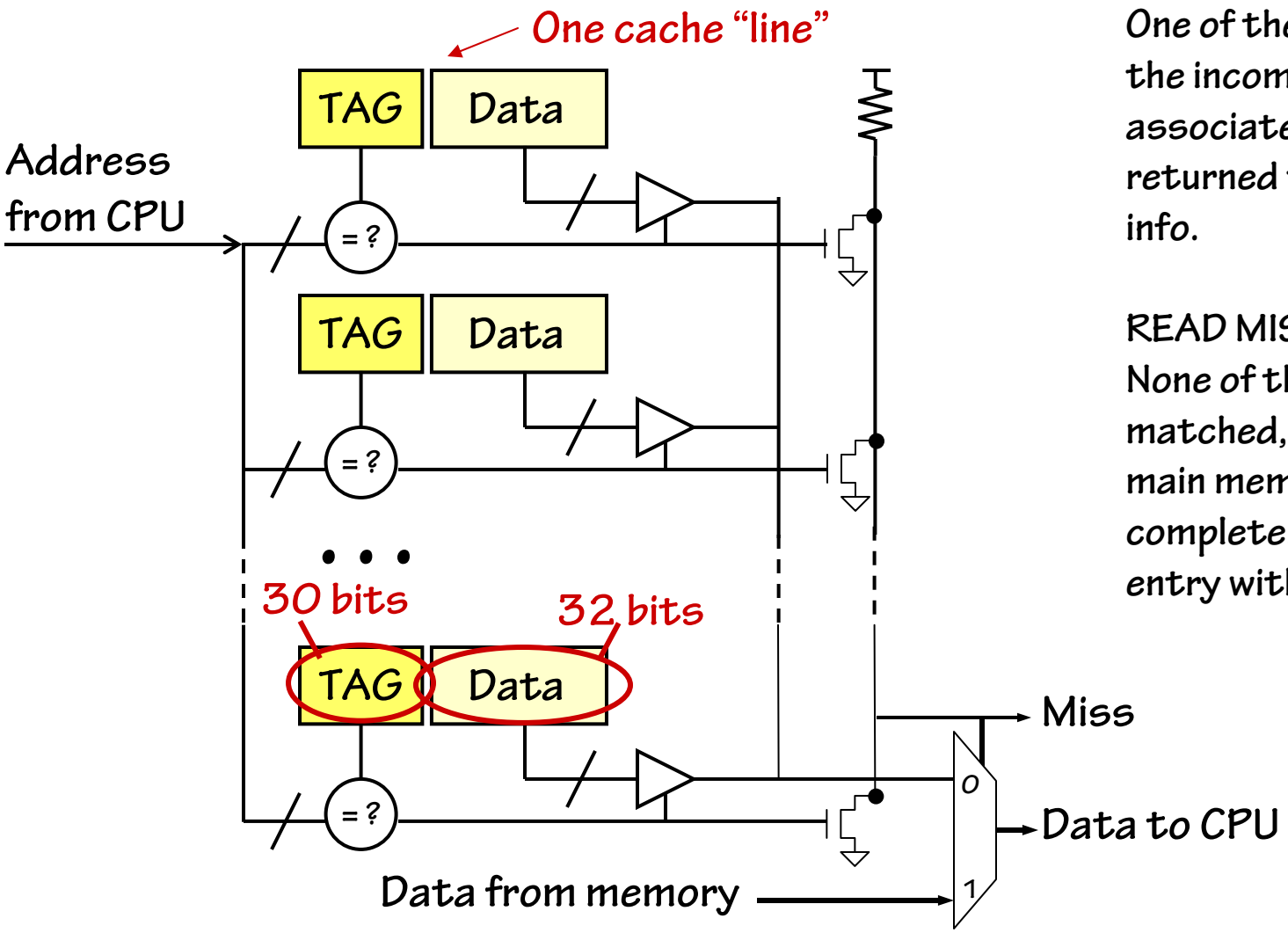
Replacement Strategy:

- OK, we've missed. Gotta add this new address/value pair to the cache. What do we kick out?
  - *Least Recently Used:* discard the one we haven't used the longest.
  - Pausible alternatives, (e.g. random replacement.

Write Strategy:

- When do we write cache contents to main memory?

Fully Associative Cache



READ HIT:
One of the cache tags matches the incoming address; the data associated with that tag is returned to CPU. Update usage info.

READ MISS:
None of the cache tags matched, so initiate access to main memory and stall CPU until complete. Update LRU cache entry with address/data.

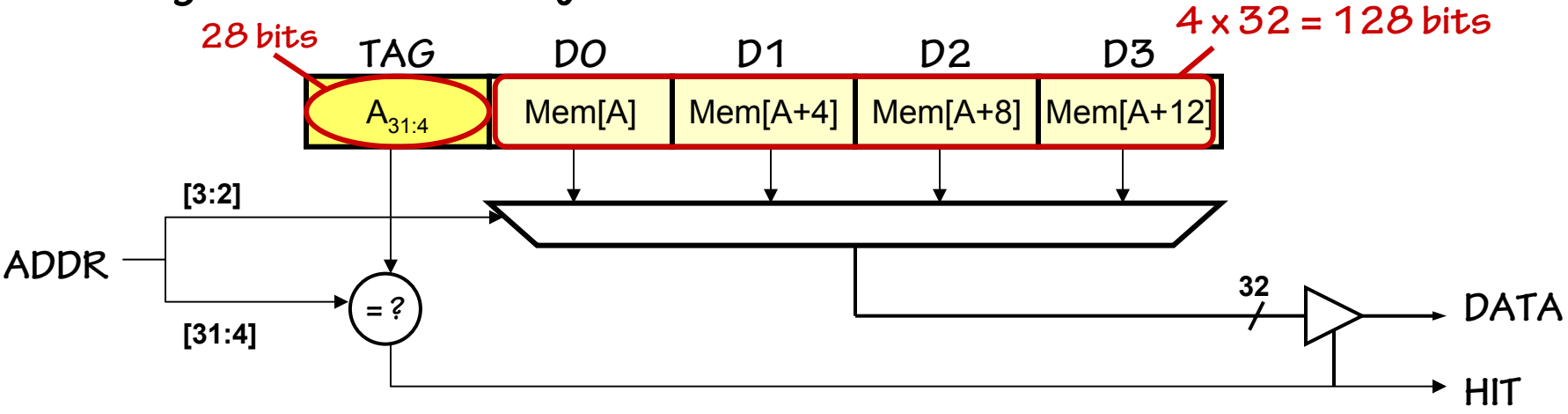
Issue: For every data word in the cache we need nearly a word of memory to store the tag!

Increasing Block Size

More Data/Tag

Overhead < ¼ bit of Tag per bit of data

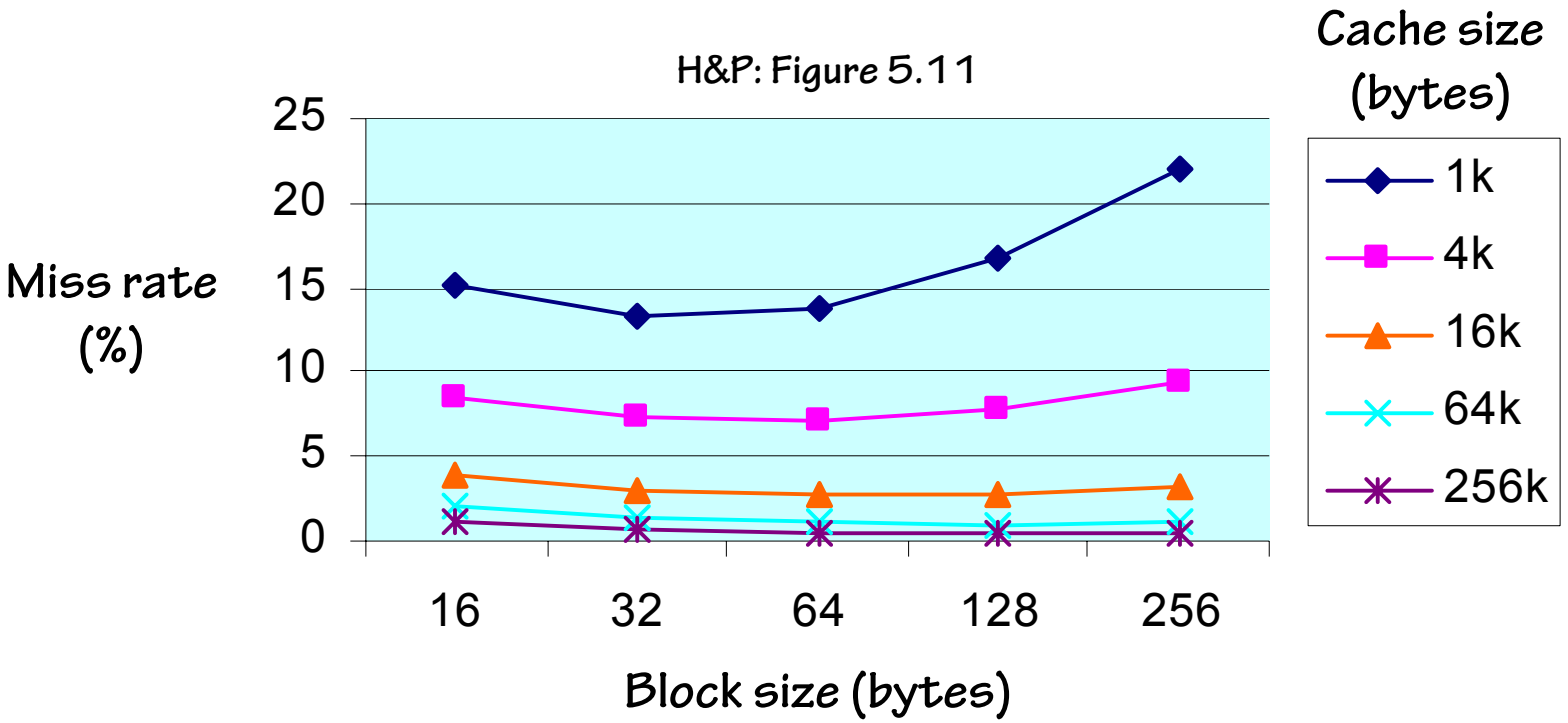
Enlarge each line in fully-associative cache:



- blocks of 2^B words, on 2^B word boundaries
- always read/write 2^B word block from/to memory
- locality: access on word in block, others likely
- cost: some fetches of unaccessed words

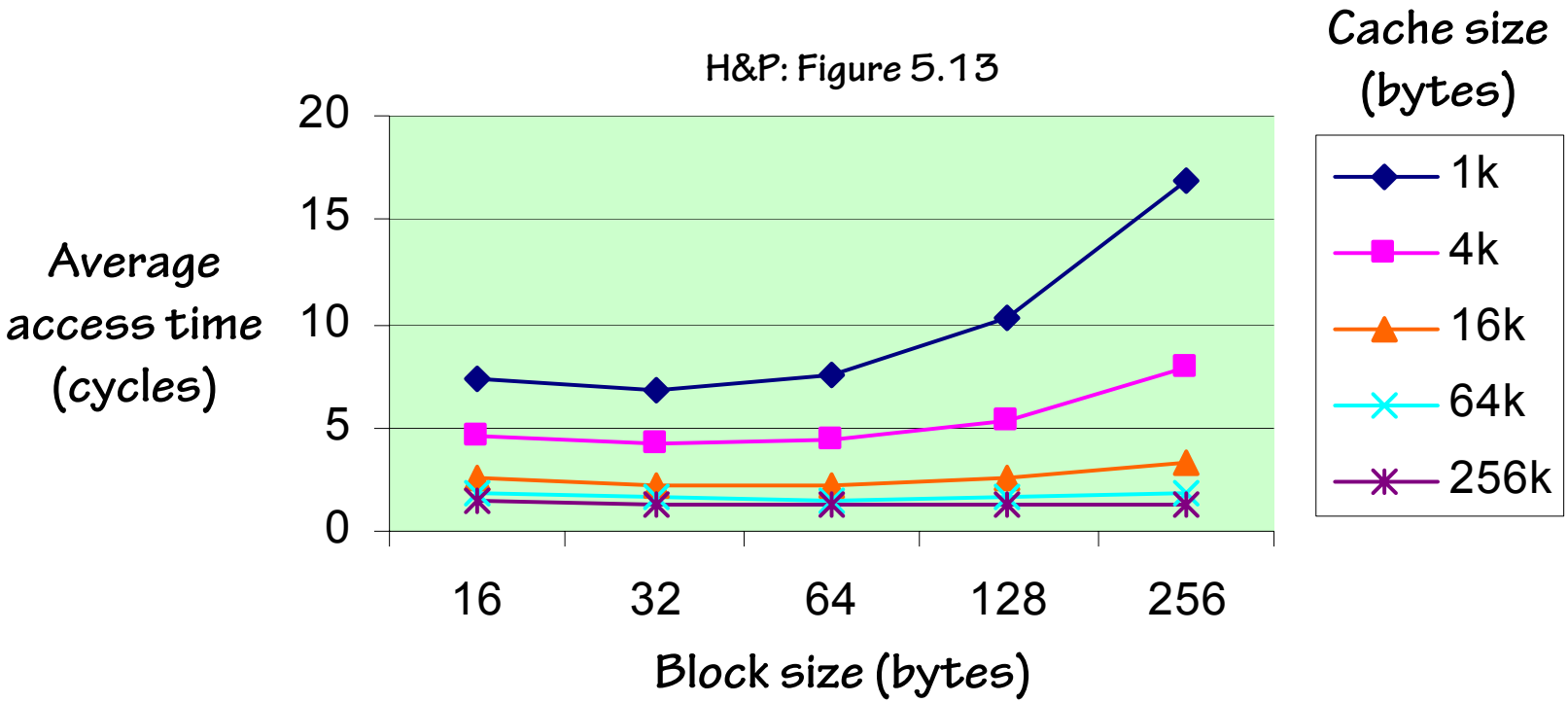
BIG WIN if there is a wide path to memory

Block size vs. Miss rate



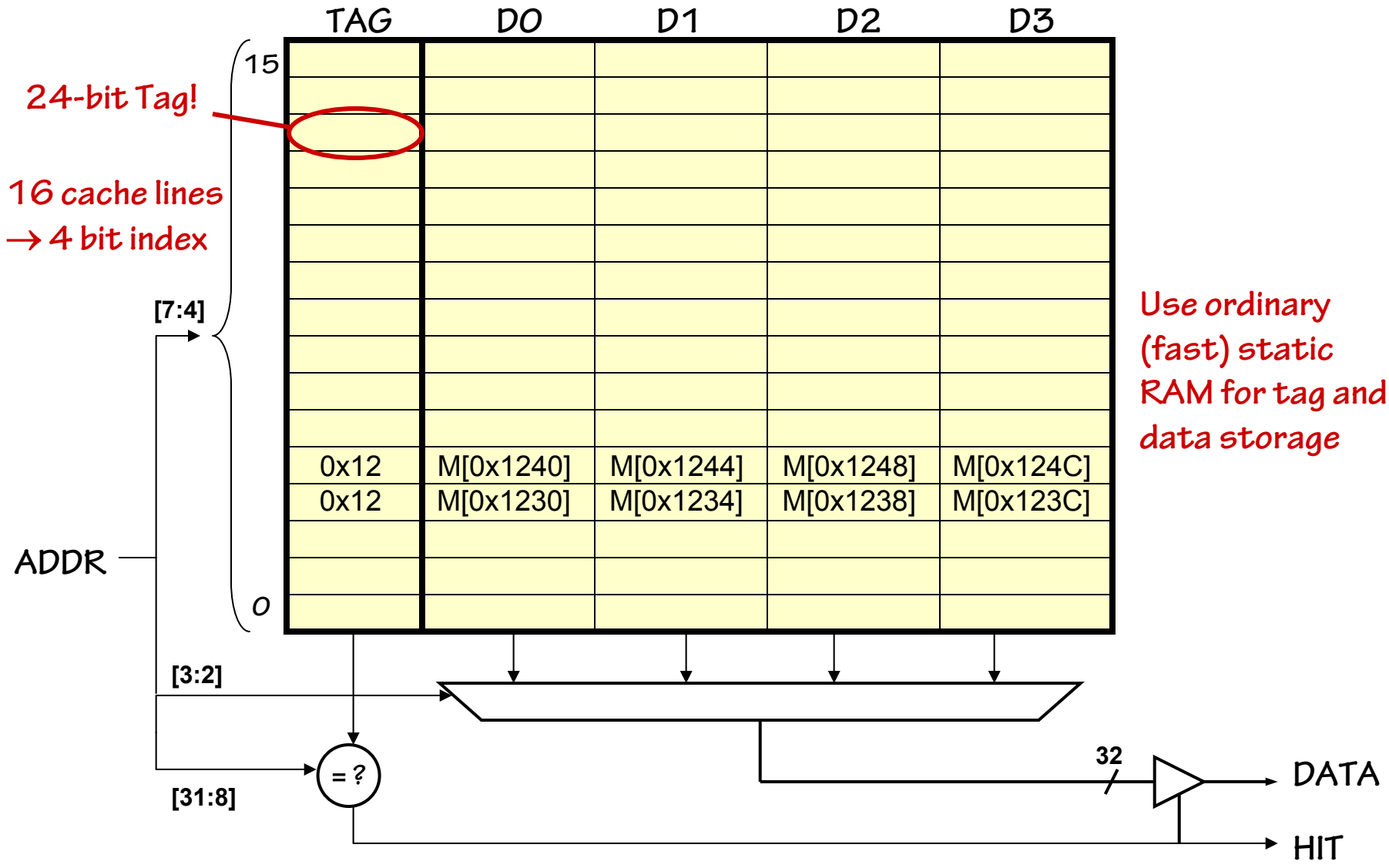
- locality: larger blocks → reduce miss rate – to a point!
- fixed cache size: larger blocks
  - → fewer lines in cache
  - → higher miss rate, especially in small caches
- Speculative prefetch has its limits… and costs!

Block size vs. Access time



- Average access time = 1 + (miss rate)*(miss penalty)
- assumed miss penalty (time it takes to read block from memory):
  40 cycles latency, then 16 bytes every 2 cycles

Direct-Mapped Cache



Only one comparator for entire cache!

Fully-assoc. vs. Direct-mapped

Fully-associative N-line cache:

- N tag comparators, registers used for tag/data storage ($$$)
- Location A might be cached in any one of the N cache lines; no "collisions" possible
- Replacement strategy (e.g., LRU) used to pick which line to use when loading new word(s) into cache

Direct-mapped N-line cache:

- 1 tag comparator, SRAM used for tag/data storage ($)
- Location A is cached in a specific line of the cache determined by its address; address "collisions" possible
- Replacement strategy not needed: each word can only be cached in one specific cache line

Cache Benchmarking

Suppose this loop is entered with R3=4000:

ADR:	Instruction	I	D
400:	LD(R3,0,R0)	400	4000+...
404:	ADDC(R3,4,R3)	404	
408:	BNE(R0,400)	408	

GOAL: Given some cache design, simulate (by hand or machine) execution well enough to determine hit ratio.

1. Observe that the sequence of memory locations referenced is

   400, 4000, 404, 408, 400, 4004, ...

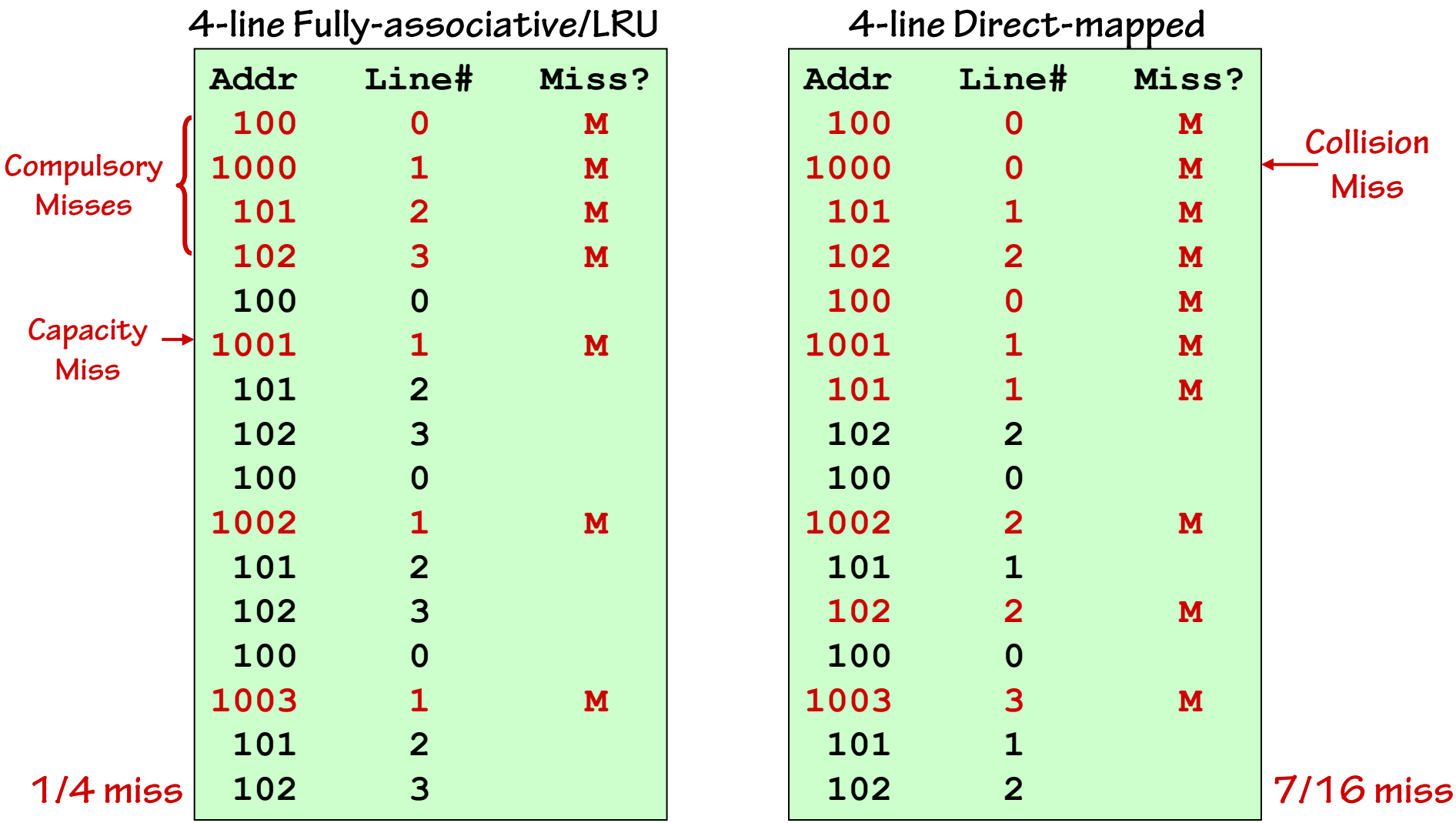
   We can use this simpler reference string, rather than the program, to simulate cache behavior.

2. We can make our life easier in many cases by converting to *word* addresses: 100, 1000, 101, 102, 100, 1001, ...

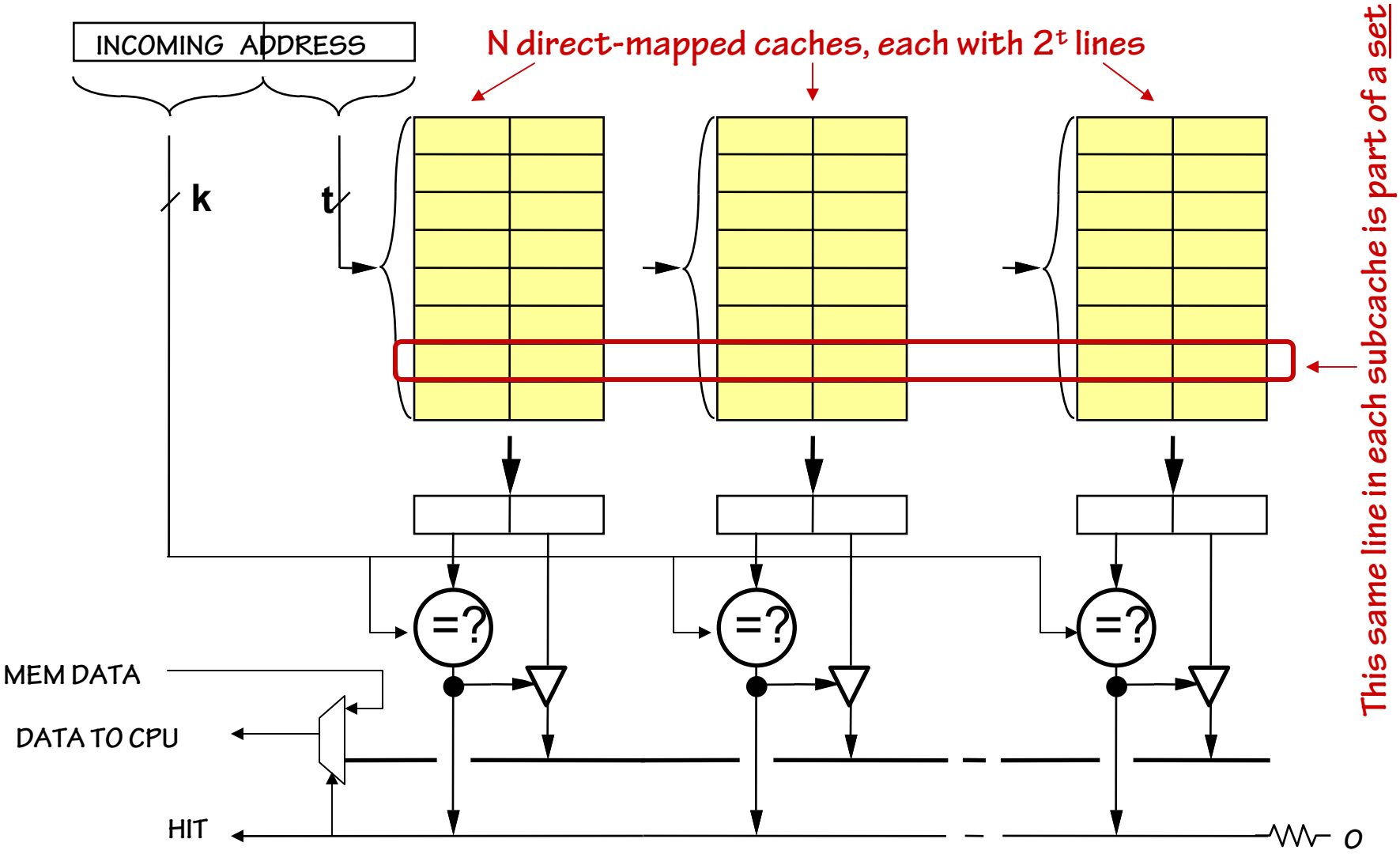
   (Word Addr = (Byte Addr)/4)

Cache Simulation

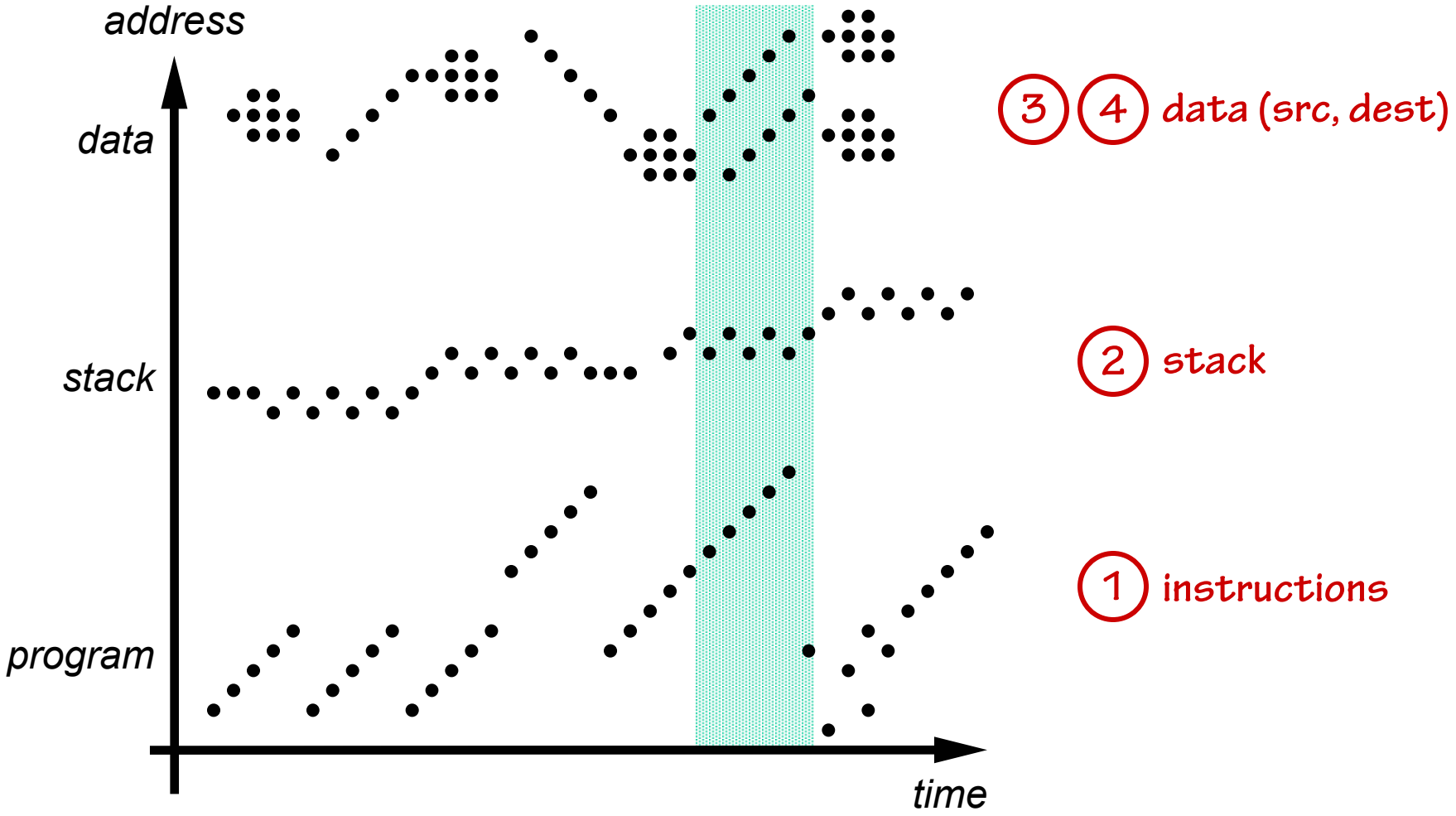
Is there anything between fully-associative and direct-mapped?



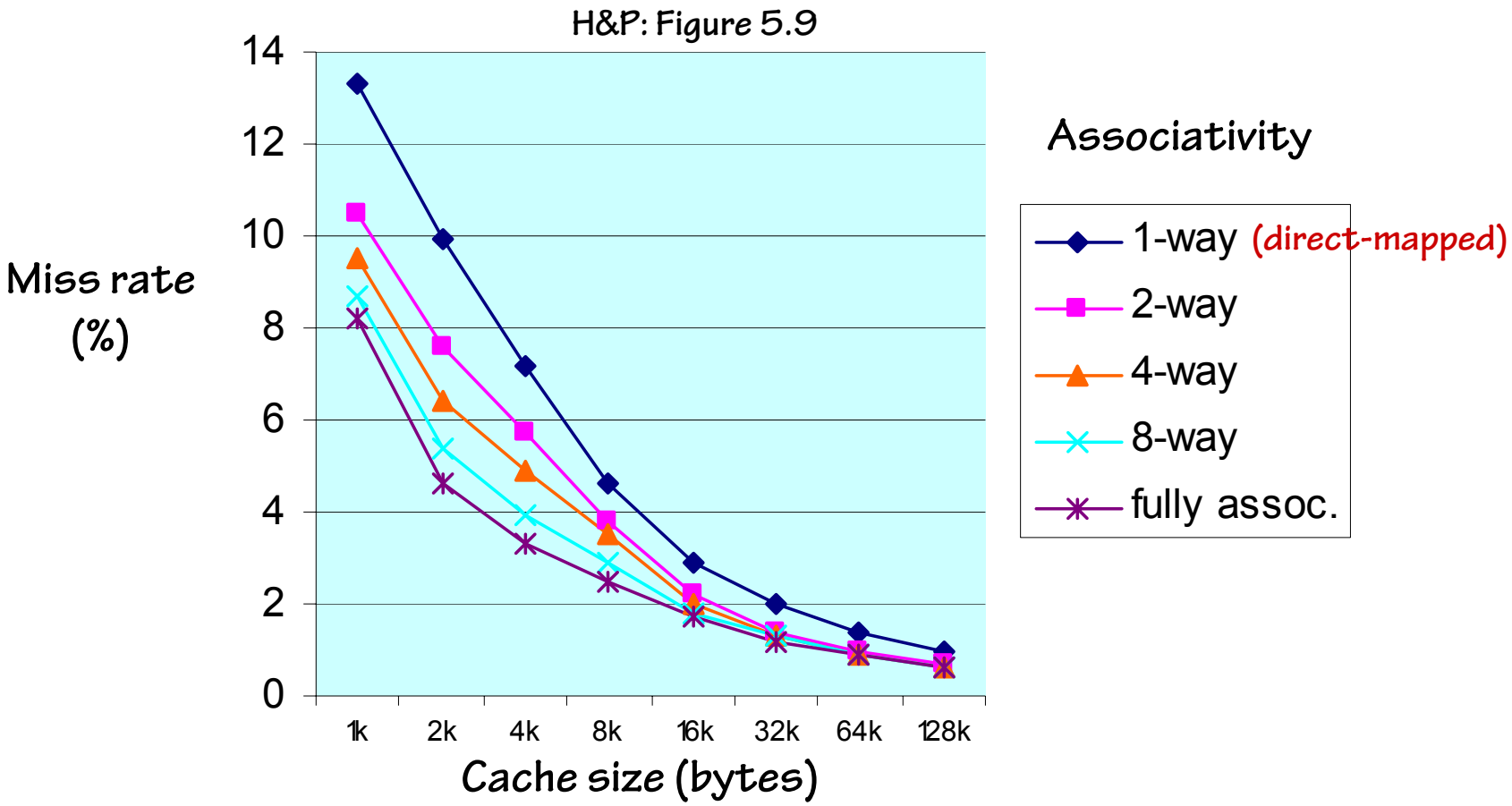
N-way Set-Associative Cache



How many lines in a set?



Associativity vs. miss rate



- 8-way is (almost) as effective as fully-associative
- rule of thumb: N-line direct-mapped == N/2-line 2-way set assoc.

Associativity: Full vs 2-way

8-line Fully-associative, LRU

Addr	Line#	Miss?
100	0	M
1000	1	M
101	2	M
102	3	M
100	0	
1001	4	M
101	2	
102	3	
100	0	
1002	5	M
101	2	
102	3	
100	0	
1003	6	M
101	2	
102	3	

1/4 miss

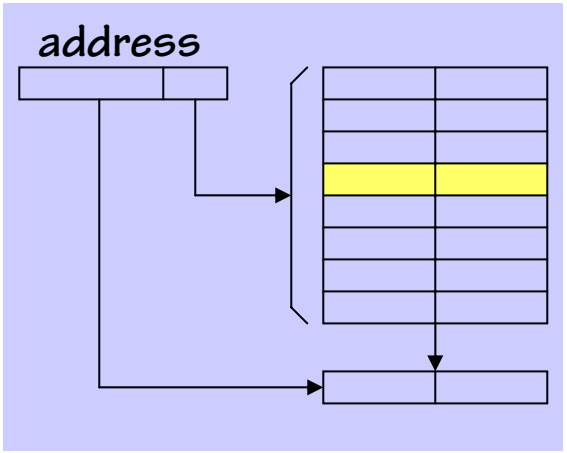
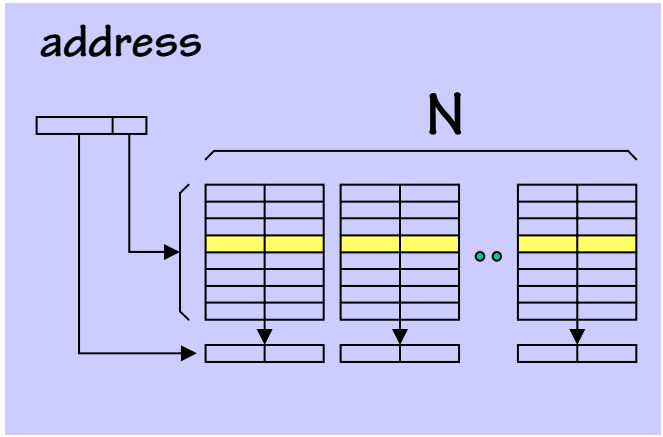
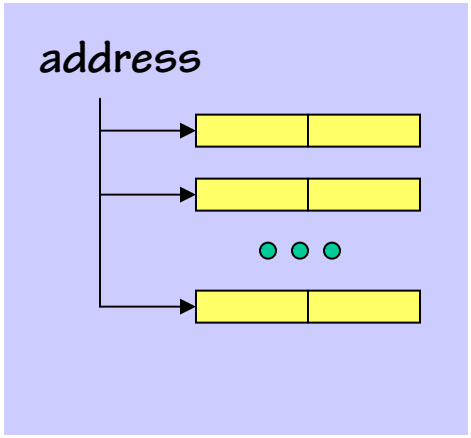
2-way, 8-line total, LRU

Addr	Line/N	Miss?
100	0,0	M
1000	0,1	M
101	1,0	M
102	2,0	M
100	0,0	
1001	1,1	M
101	1,0	
102	2,0	
100	0,0	
1002	2,1	M
101	1,0	
102	2,0	
100	0,0	
1003	3,1	M
101	1,0	
102	2,0	

1/4 miss

Associativity implies choices…

ISSUE: Replacement Strategy

Direct-mapped	N-way set associative	Fully associative
		
• compare addr with only one tag • location A can be stored in exactly one cache line	• 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	• compare addr with each tag simultaneously • location A can be stored in any cache line

Replacement Strategy

(0,1,2,3) Hit 2 -> (2,0,1,3)
(2,0,1,3) Hit 1 -> (1,2,0,3)
(1,2,0,3) Miss -> (3,1,2,0)
(3,1,2,0) Hit 3 -> (3,1,2,0)

LRU (Least-recently used)

- keeps most-recently used locations in cache
- need to keep ordered list of N items → N! orderings
  → $O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" + complex logic

Overhead is $O(N \log_2 N)$ bits/set

FIFO/LRR (first-in, first-out/least-recently replaced)

- cheap alternative: replace oldest item (dated by access time)
- within each set: keep one counter that points to victim line

Overhead is $O(\log_2 N)$ bits/set

Random (select replacement line using random, uniform distribution)

- no "pathological" reference streams causing wost-case results
- use pseudo-random generator to get reproducible behavior;
- use *real* randomness to prevent reverse engineering!

Overhead is $O(\log_2 N)$ bits/cache!

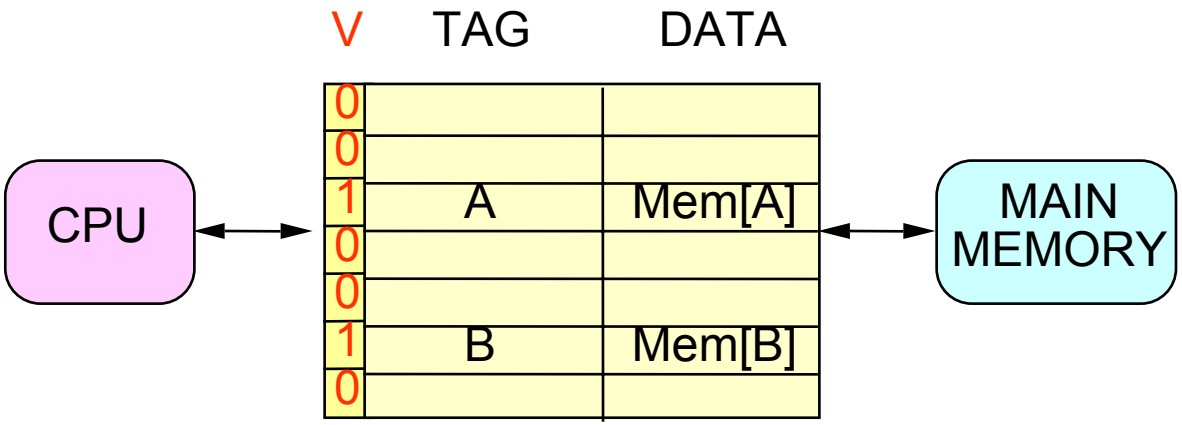
Replacement Strategy vs. Miss Rate

H&P: Figure 5.4

Size	Associativity					
	2-way		4-way		8-way	
	LRU	Random	LRU	Random	LRU	Random
16KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

- FIFO was reported to be worse than random or LRU
- little difference between random and LRU for larger-size caches

Valid bits



Problem:

- Ignoring cache lines that don't contain anything of value... e.g., on
  - - start-up
  - - "Back door" changes to memory (eg loading program from disk)

Solution:

- Extend each TAG with VALID bit.
- • Valid bit must be set for cache line to HIT.
- • At power-up / reset : clear all valid bits
- • Set valid bit when cache line is first *replaced.*
- • Cache Control Feature: *Flush* cache by clearing all valid bits, Under program/external control.

Handling of WRITES

Observation: Most (90+%) of memory accesses are *READs*. How should we handle writes? Issues:

- Write-through: CPU writes are cached, but also written to main memory (stalling the CPU until write is completed). Memory always holds "the truth".
- Write-behind: CPU writes are cached; writes to main memory may be *buffered*, perhaps pipelined. CPU keeps executing while writes are completed (in order) in the background.
- Write-back: CPU writes are cached, but not immediately written to main memory. Memory contents can be "stale".

Our cache thus far uses write-through.

Can we improve write performance?

Write-through

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X == TAG(i) , for some cache line i*

- READ: return DATA[I]
- WRITE: change DATA[I]; Start Write to Mem[X]

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X]
- READ: Read Mem[X]
  - Set TAG[k] = X, DATA[k] = Mem[X]
- WRITE: Start Write to Mem[X]
  - Set TAG[k] = X, DATA[k] = new Mem[X]

Write-back

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

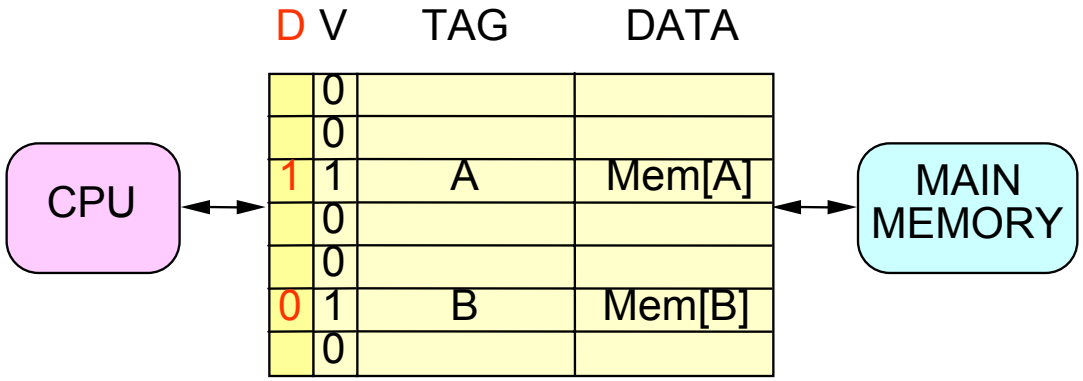
- READ: return DATA(i)
- WRITE: change DATA(i); ~~Start Write to Mem[X]~~

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X]
  - Write Back: Write Data(k) to Mem[Tag[k]]
- READ: Read Mem[X]
  - Set TAG[k] = X, DATA[k] = Mem[X]
- WRITE: ~~Start Write to Mem[X]~~
  - Set TAG[k] = X, DATA[k] = new Mem[X]

Is write-back worth the trouble? Depends on (1) cost of write; (2) consistency issues.

Write-back w/ "Dirty" bits



ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i), for some cache line I*

- READ: return DATA(i)
- WRITE: change DATA(i); ~~Start Write to Mem[X]~~ D[i]=1

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X]
  - If D[k] == 1 (Write Back) Write Data(k) to Mem[Tag[k]]
- READ: Read Mem[X]; Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0
- WRITE: ~~Start Write to Mem[X]~~ D[k]=1
  - Set TAG[k] = X, DATA[k] = new Mem[X]

Cache Analysis

Given a cache design, find pathological reference string:

- Simulate cache by hand; maximize misses
- On miss, note replaced address. Make that the next reference!

Comparing caches A and B:

- Usually can find pathological cases for each, making the other look better…
- Whenever caches make different replacement decisions, you can devise a next reference to make either cache look better!

Dominance:

- Cache A *dominates* cache B if, for every reference string, A contains every value contained by B. Thus A won't miss unless B misses as well.
- Example: Given 2 otherwise identical fully-associative, LRU caches, a bigger one dominates a smaller one.

Caches: Summary

Associativity:

- Less important as size increases
- 2-way or 4-way usually plenty for typical program clustering; BUT additional associativity
  - Smooths performance curve
  - Reduces number of select bits (we'll see shortly how this helps)
- TREND: Invest in RAM, not comparators.

Replacement Strategy:

- BIG caches: any sane approach works well
- REAL randomness assuages paranoia!

Performance analysis:

- Tedious hand synthesis may build intuition from simple examples, BUT
- Computer simulation of cache behavior on REAL programs (or using REAL trace data) is the basis for most real-world cache design decisions.