Caches 2

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

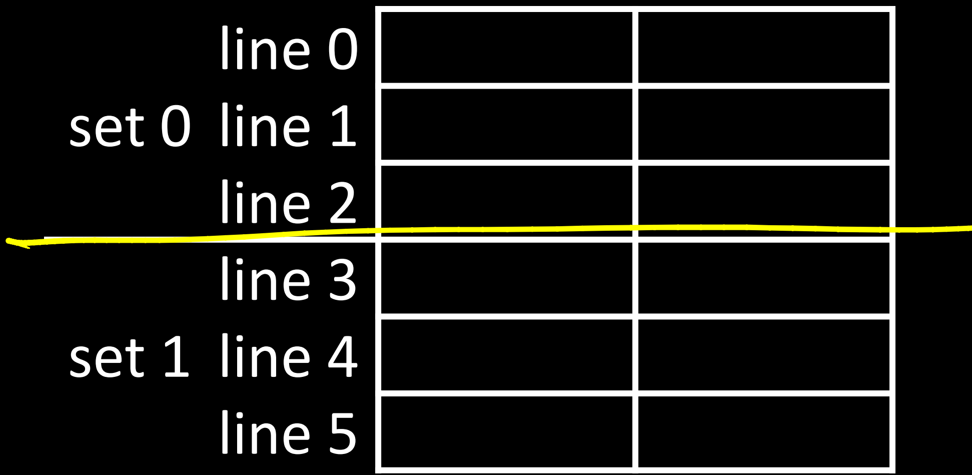
P & H Chapter 5.2 (writes), 5.3, 5.5

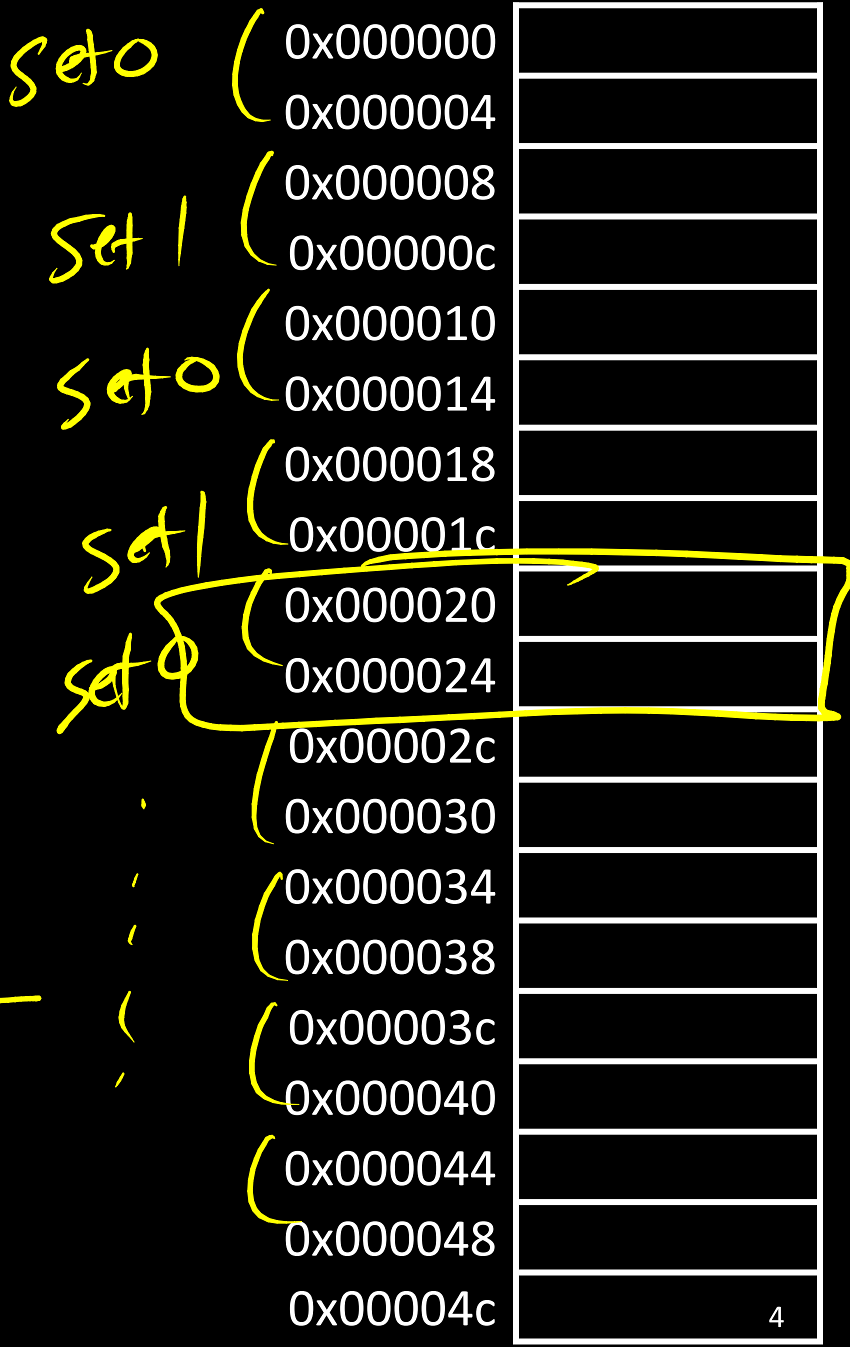
Direct Mapped		Fully Associative
+ Smaller	Tag Size	Larger –
+ Less	SRAM Overhead	More –
+ Less	Controller Logic	More –
+ Faster	Speed	Slower –
+ Less	Price	More –
+ Very	Scalability	Not Very –
– Lots	# of conflict misses	(?) Zero +
– Low	Hit rate	High +
– Common	Pathological Cases?	?

Set Associative Caches

Set Associative Cache

- Each block number mapped to a single cache line set index
- Within the set, block can go in any line





Set Associative Cache

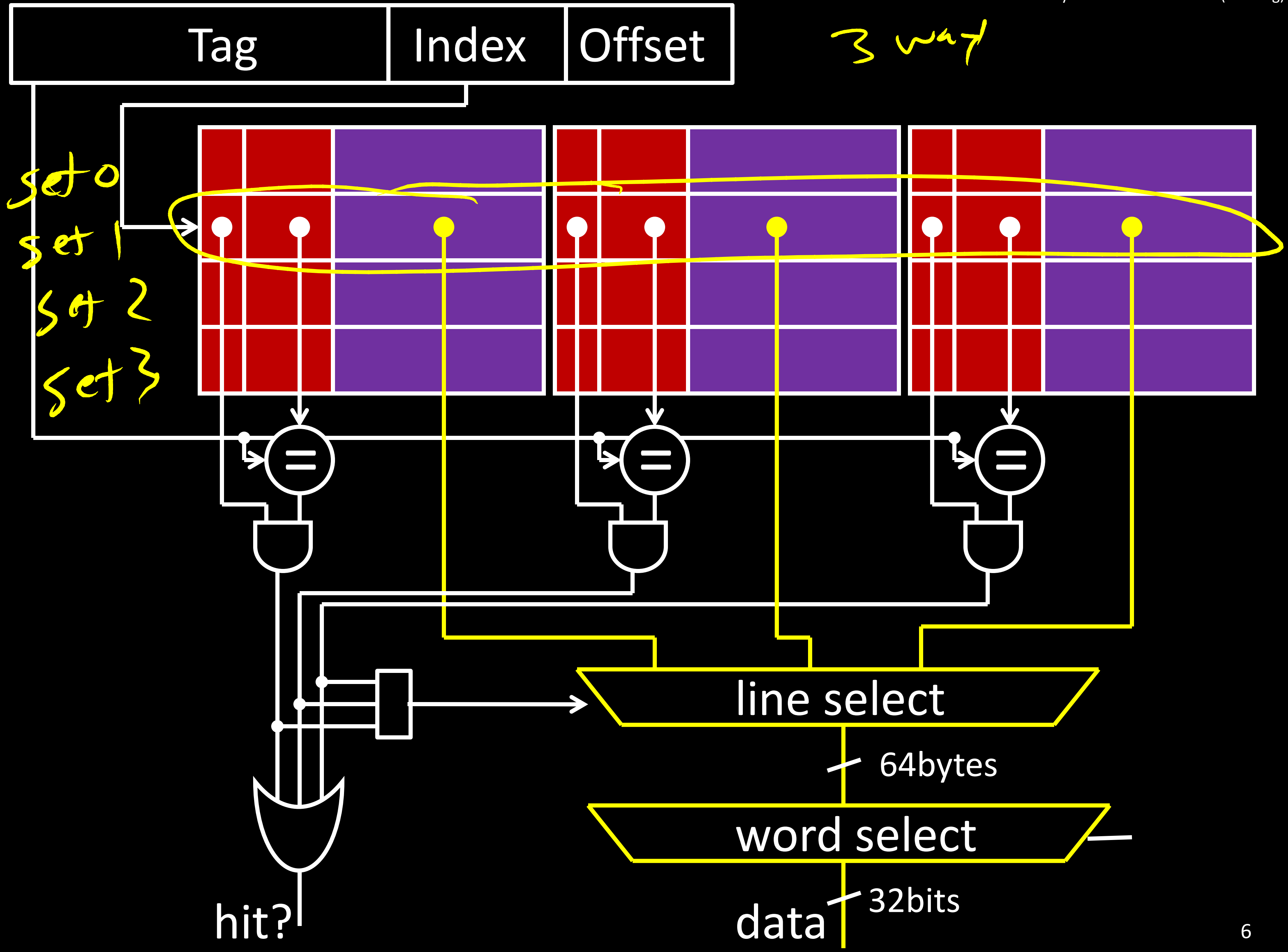
Like direct mapped cache

- Only need to check a few lines for each access...
  so: fast, scalable, low overhead

Like a fully associative cache

- Several places each block can go...
  so: fewer conflict misses, higher hit rate

3-Way Set Associative Cache (Reading)



A Simple 2-Way Set Associative Cache

Using **byte addresses** in this example! Addr Bus = 5 bits

Processor

lb $1 ← M[1] M
lb $2 ← M[13] M
lb $3 ← M[0] H
lb $3 ← M[6] M
lb $2 ← M[5]
lb $2 ← M[6]
lb $2 ← M[10]
lb $2 ← M[12]

$1	
$2	
$3	
$4	

2-Way Set Associative Cache

A = 0 0 1 0 1

tag index offset

	V	tag	data	
set 0	1	000	101	103
	1	011		
set 1	1	001	131	137
	0			

Hits: Misses:

Memory

0	101
1	103
2	107
3	109
4	113
5	127
6	131
7	137
8	139
9	149
10	151
11	157
12	163
13	167
14	173
15	179
16	181

A Pathological Case

Processor

lb $1 ← M[1] M
lb $2 ← M[8] M
lb $3 ← M[1] H
lb $3 ← M[8] H
lb $2 ← M[1] H
lb $2 ← M[16] M
lb $2 ← M[1] H
lb $2 ← M[8] M

$1	
$2	
$3	
$4	

Direct Mapped

8 Misses

			1

2-Way Set Associative

4 Misses

		0	1
		16	17

Fully Associative

3 Misses

		0	1
		8	9
		16	17

Memory

0	101
1	103
2	107
3	109
4	113
5	127
6	131
7	137
8	139
9	149
10	151
11	157
12	163
13	167
14	173
15	179
16	181

To Do:

- Evicting cache lines
- Picking cache parameters
- Writing using the cache

Q: Which line should we evict to make room?

For direct-mapped?

A: no choice, must evict the indexed line

For associative caches?

FIFO: oldest line (timestamp per line)

LRU: least recently used (ts per line)

LFU: (need a counter per line)

MRU: most recently used (?!) (ts per line)

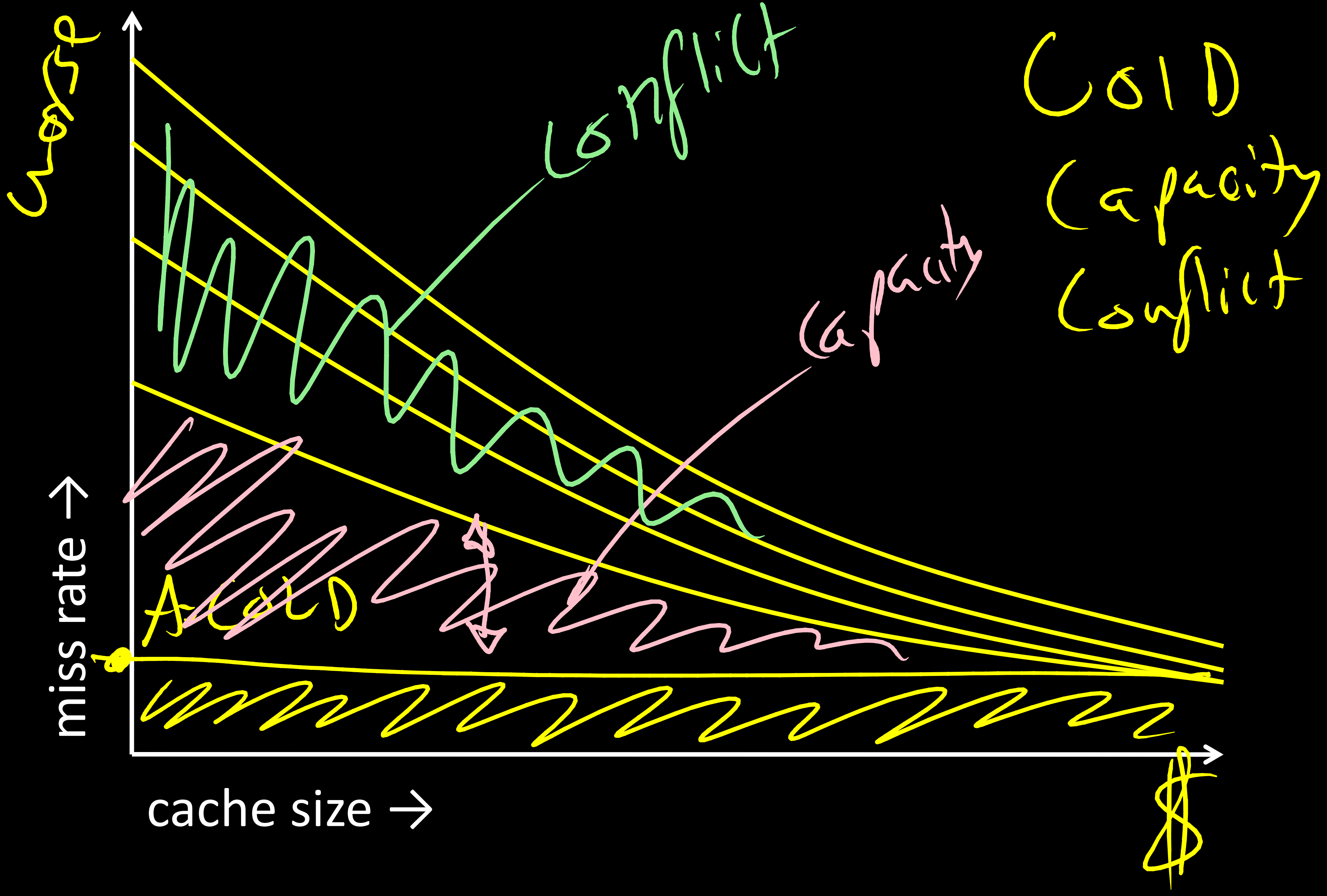
RR: round-robin (need a finger per set)

RAND: random (free!)

Belady’s: optimal (need time travel)

Cache Parameters

direct mapped, 2-way, 8-way, fully associative



Need to determine parameters:

- Cache size
- Block size (aka line size)
- Number of ways of set-associativity (1, N, ∞)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy

A Real Example

Dual-core 3.16GHz Intel (purchased in 2009)

```
> dmidecode -t cache
Cache Information
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Back
        Installed Size: 128 KB
        Error Correction Type: None
Cache Information
        Configuration: Enabled, Not Socketed, Level 2
        Operational Mode: Varies With Memory Address
        Installed Size: 6144 KB
        Error Correction Type: Single-bit ECC
> cd /sys/devices/system/cpu/cpu0; grep cache/*/*
cache/index0/level:1
cache/index0/type:Data
cache/index0/ways_of_associativity:8
cache/index0/number_of_sets:64
cache/index0/coherency_line_size:64
cache/index0/size:32K
cache/index1/level:1
cache/index1/type:Instruction
cache/index1/ways_of_associativity:8
cache/index1/number_of_sets:64
cache/index1/coherency_line_size:64
cache/index1/size:32K
cache/index2/level:2
cache/index2/type:Unified
cache/index2/shared_cpu_list:0-1
cache/index2/ways_of_associativity:24
cache/index2/number_of_sets:4096
cache/index2/coherency_line_size:64
cache/index2/size:6144K
```

A Real Example

Dual-core 3.16GHz Intel (purchased in 2009)

Dual 32K L1 Instruction caches

- 8-way set associative
- 64 sets
- 64 byte line size

Dual 32K L1 Data caches

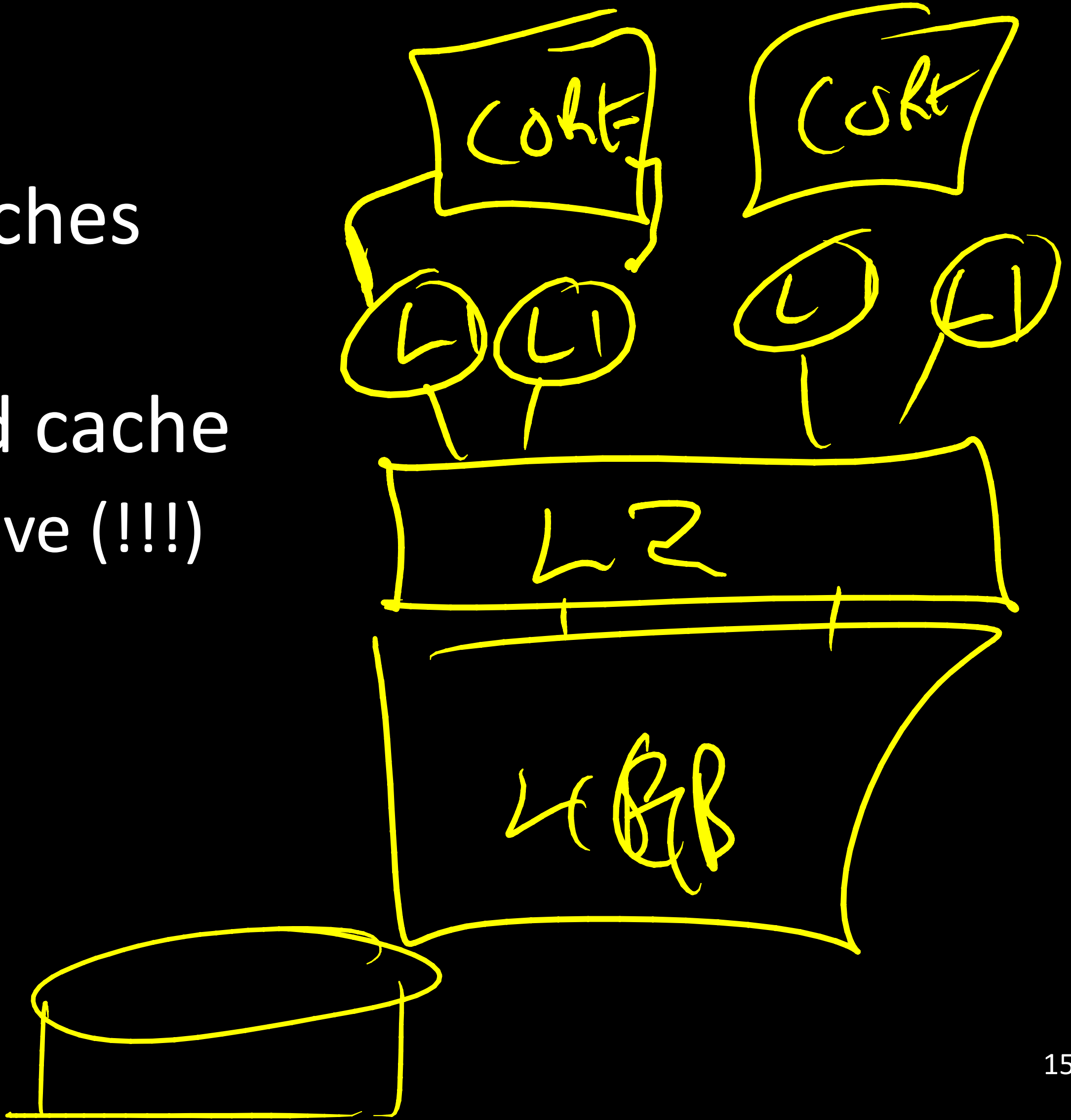
- Same as above

Single 6M L2 Unified cache

- 24-way set associative (!!!)
- 4096 sets
- 64 byte line size

4GB Main memory

1TB Disk

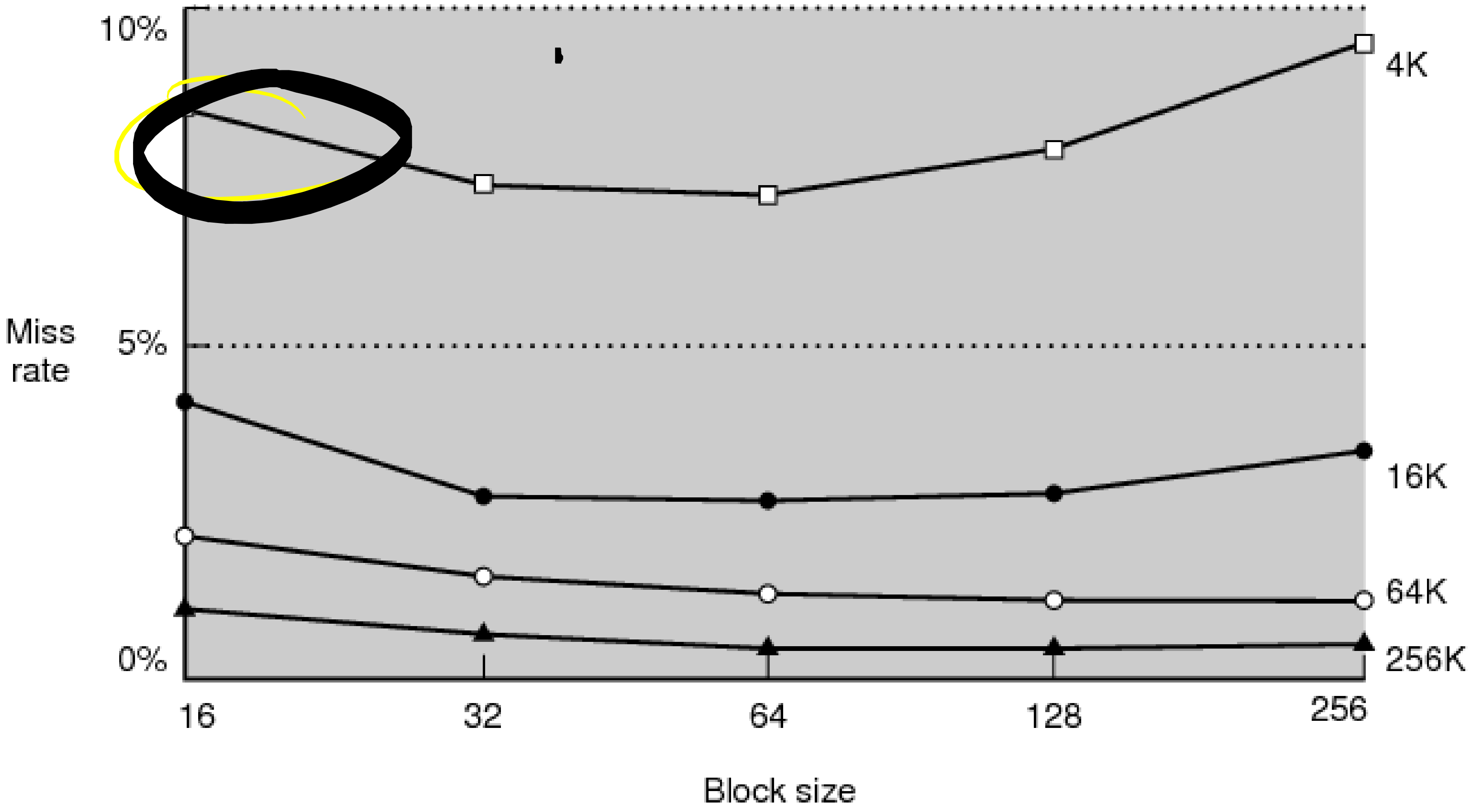


Q: How to decide block size?

A: Try it and see

But: depends on cache size, workload, associativity, …

Experimental approach!


10%
5%
0%
Miss rate
16
32
64
128
256
Block size
4K
16K
64K
256K

For a given total cache size,

larger block sizes mean….

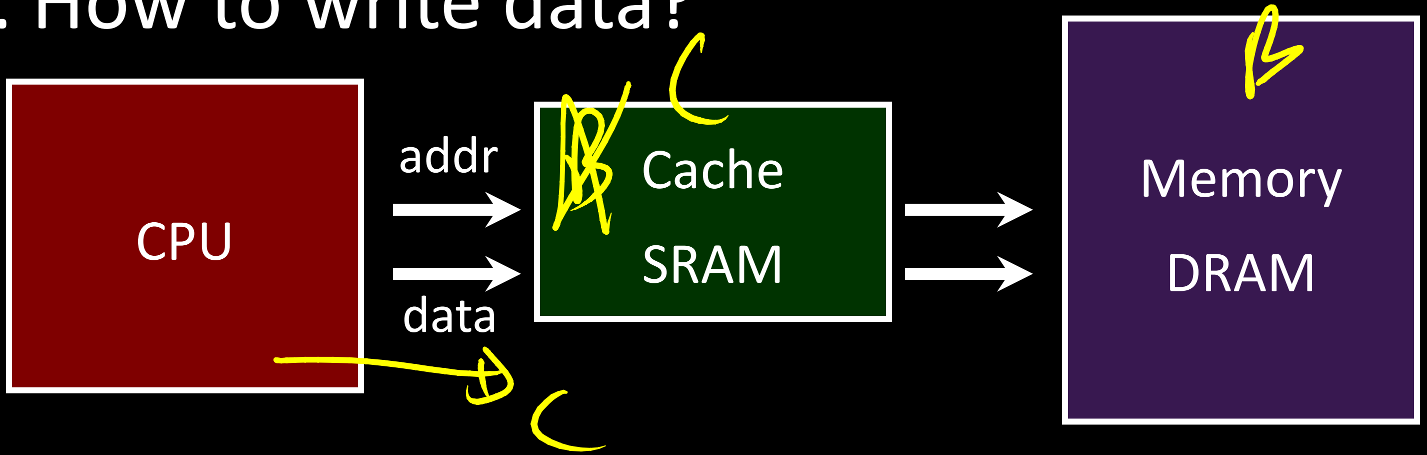
- fewer lines
- so fewer tags (and smaller tags for associative caches)
- so less overhead
- and fewer cold misses (within-block “prefetching”)

But also…

- fewer blocks available (for scattered accesses!)
- so more conflicts
- and larger miss penalty (time to fetch block)

Writing with Caches

Q: How to write data?



If data is already in the cache…

No-Write

- writes invalidate the cache and go directly to memory

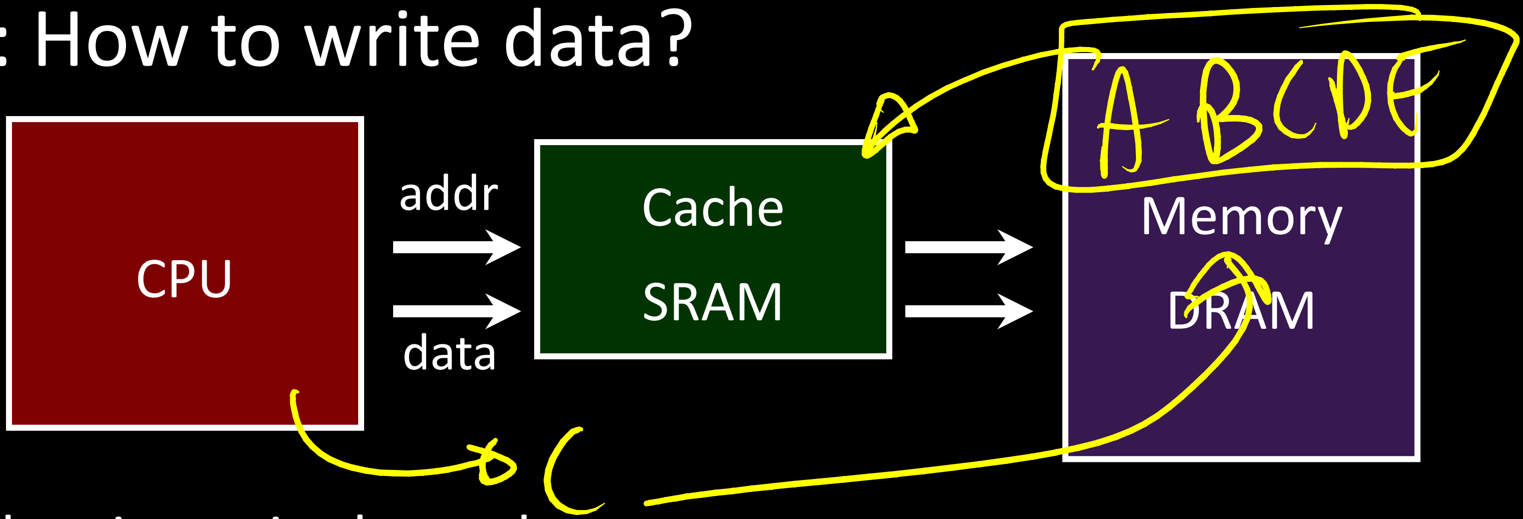
Write-Through

- writes go to main memory and cache

Write-Back

- CPU writes only to cache
- cache writes to main memory later (when block is evicted)

Q: How to write data?



If data is not in the cache…

Write-Allocate

- allocate a cache line for new data (and maybe write-through)

No-Write-Allocate

- ignore cache, just go to main memory

Using **byte addresses** in this example! Addr Bus = 5 bits

Processor

lb $1 ← M[1] ✓

lb $2 ← M[7] ✓

sb $2 → M[0]

sb $1 → M[5]

lb $2 ← M[9]

sb $1 → M[5]

sb $1 → M[0]

$1	
$2	
$3	
$4	

Direct Mapped Cache + Write-through + Write-allocate

V	tag	data	
		~~101~~ 137	103
		113	~~127~~ 103
		131	137

Hits: Misses:

Memory

0	~~101~~ 137
1	103
2	107
3	109
4	113
5	~~127~~ 103
6	131
7	137
8	139
9	149
10	151
11	157
12	163
13	167
14	173
15	179
16	181

Write-through performance

Each miss (read or write) reads a block from mem

- 5 misses → 10 mem reads

Each store writes an item to mem

- 4 mem writes

Evictions don’t need to write to mem

- no need for dirty bit

Using **byte addresses** in this example! Addr Bus = 5 bits

Processor

lb $1 ← M[1]
lb $2 ← M[7]
sb $2 → M[0]
sb $1 → M[5]
lb $2 ← M[9]
sb $1 → M[5]
sb $1 → M[0]

$1	
$2	
$3	
$4	

Direct Mapped Cache + Write-back + Write-allocate

V	D	tag	data	

Hits: Misses:

Memory

0	101
1	103
2	107
3	109
4	113
5	127
6	131
7	137
8	139
9	149
10	151
11	157
12	163
13	167
14	173
15	179
16	181

Write-back performance

Each miss (read or write) reads a block from mem

- 5 misses → 10 mem reads

Some evictions write a block to mem

- 1 dirty eviction → 2 mem writes
- (+ 2 dirty evictions later → +4 mem writes)
- need a dirty bit

Write-Back Meta-Data

V	D	Tag	Byte 1	Byte 2	... Byte N
1	1	—		*	

V = 1 means the line has valid data

D = 1 means the bytes are newer than main memory

When allocating line:

- Set V = 1, D = 0, fill in Tag and Data

When writing line:

- Set D = 1

When evicting line:

- If D = 0: just set V = 0
- If D = 1: write-back Data, then set D = 0, V = 0

Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

WT → n w

```
for (i=1; i<n; i++)
     A[0] += A[i];
```

```
for (i=0; i<n; i++)
     B[i] = A[i]
```

Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

```
for (i=1; i<n; i++)
      A[0] += A[i];
```

WT → n writes to MEM

WB → block size write

```
for (i=0; i<n; i++)
      B[i] = A[i]
```

WT = WB = n writes

Q: Hit time: write-through vs. write-back?

A: Write-through slower on writes.

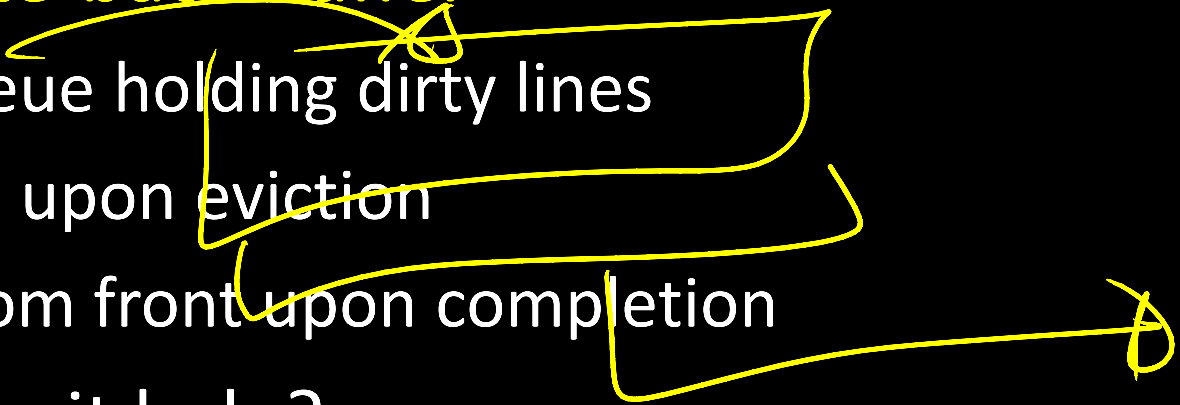
Q: Miss penalty: write-through vs. write-back?

A: Write-back slower on evictions.

Q: Writes to main memory are **slow!**

A: Use a write-back buffer

- A small queue holding dirty lines
- Add to end upon eviction
- Remove from front upon completion



Q: What does it help?

Miss

A: short bursts of writes (but not sustained writes)

- evict a line

A: fast eviction reduces miss penalty

- read a line

Q: Writes to main memory are **slow!**

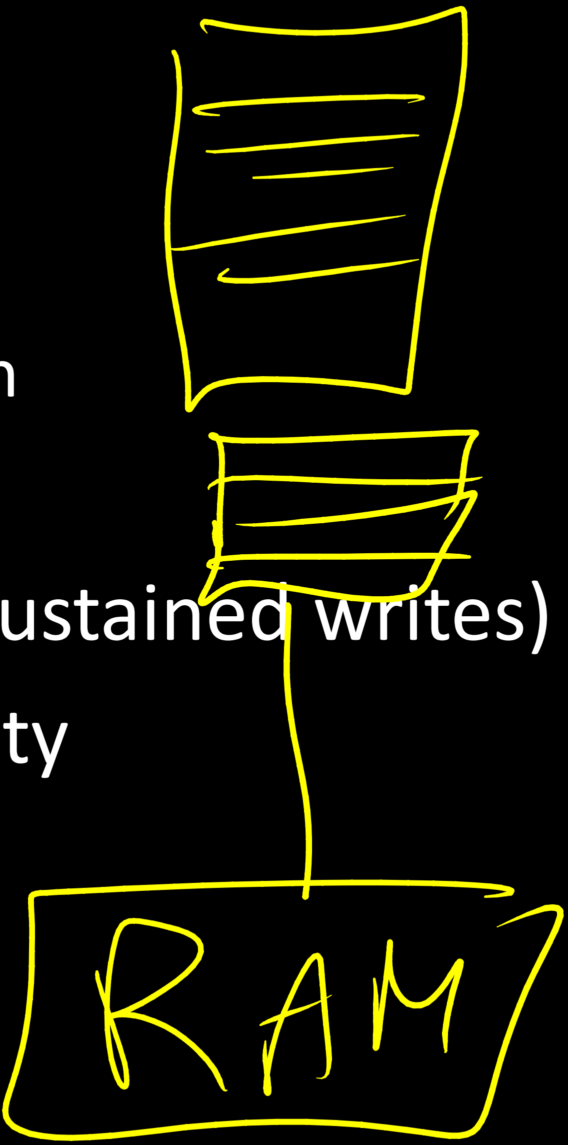
A: Use a write-back buffer

- A small queue holding dirty lines
- Add to end upon eviction
- Remove from front upon completion

Q: What does it help?

A: short bursts of writes (but not sustained writes)

A: fast eviction reduces miss penalty



Write-through is slower

- But simpler (memory always consistent)

Write-back is almost always faster

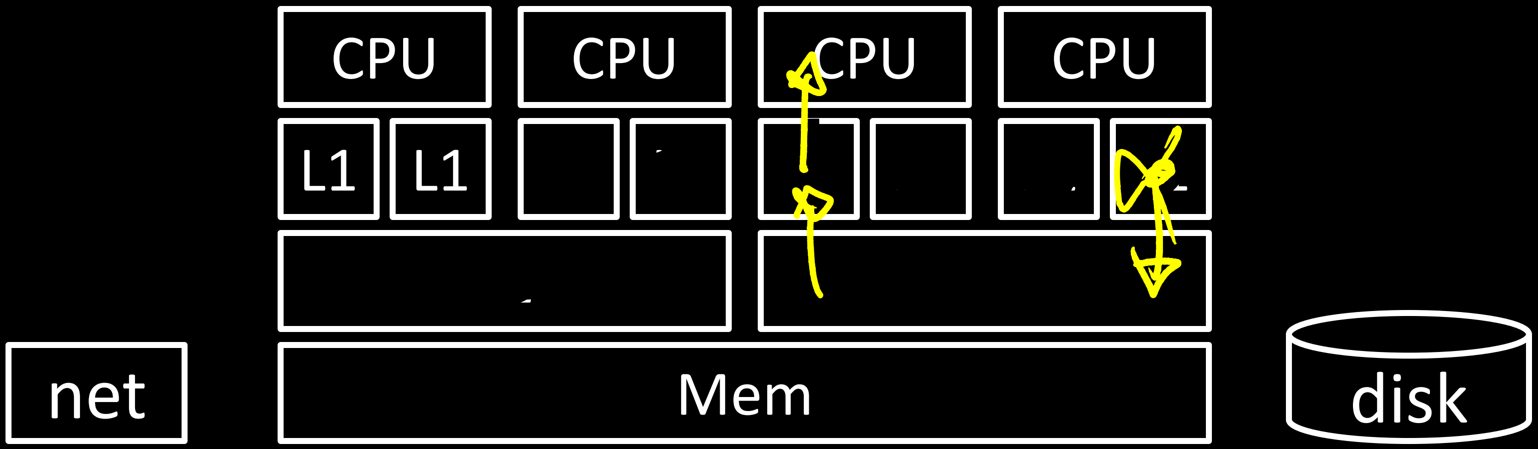
- write-back buffer hides large eviction cost
- But what about multiple cores with separate caches but sharing memory?

Write-back requires a cache coherency protocol

- Inconsistent views of memory
- Need to "snoop" in each other's caches
- Extremely complex protocols, very hard to get right

Q: Multiple readers and writers?

A: Potentially inconsistent views of memory



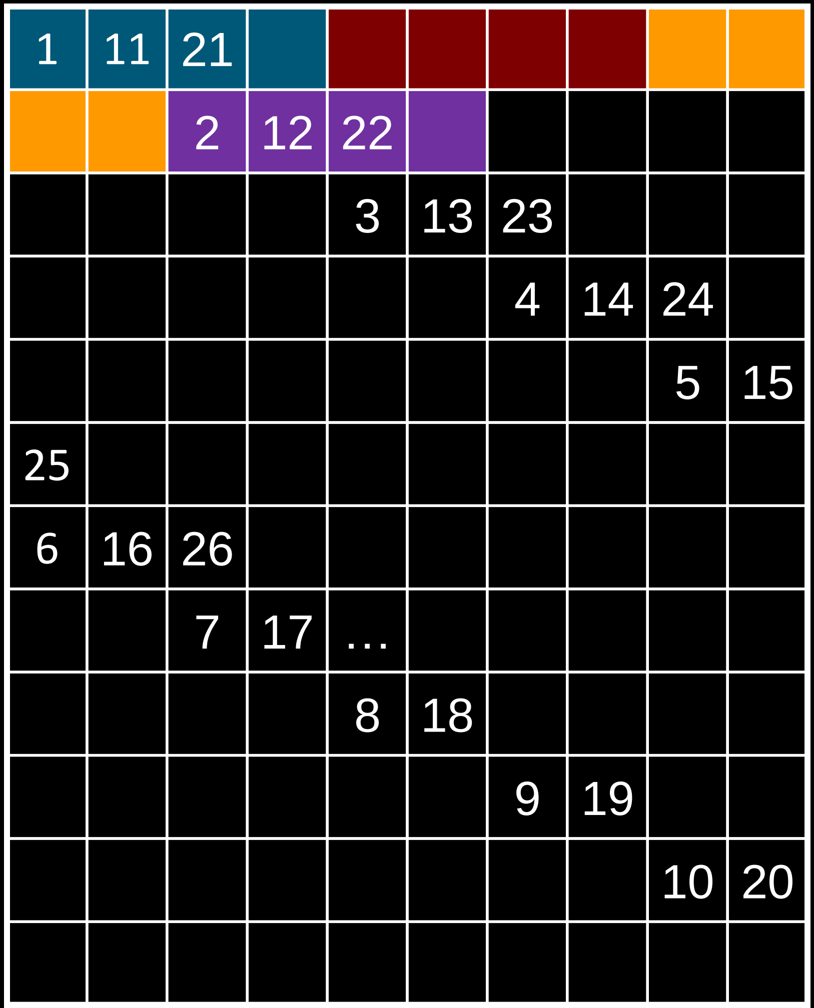
Cache coherency protocol

- May need to snoop on other CPU’s cache activity
- Invalidate cache line when other CPU writes
- Flush write-back caches before other CPU reads
- Or the reverse: Before writing/reading…
- Extremely complex protocols, very hard to get right

Cache Conscious Programming

```
// H = 12, W = 10
int A[H][W];

for(x=0; x < W; x++)
   for(y=0; y < H; y++)
      sum += A[y][x];
```



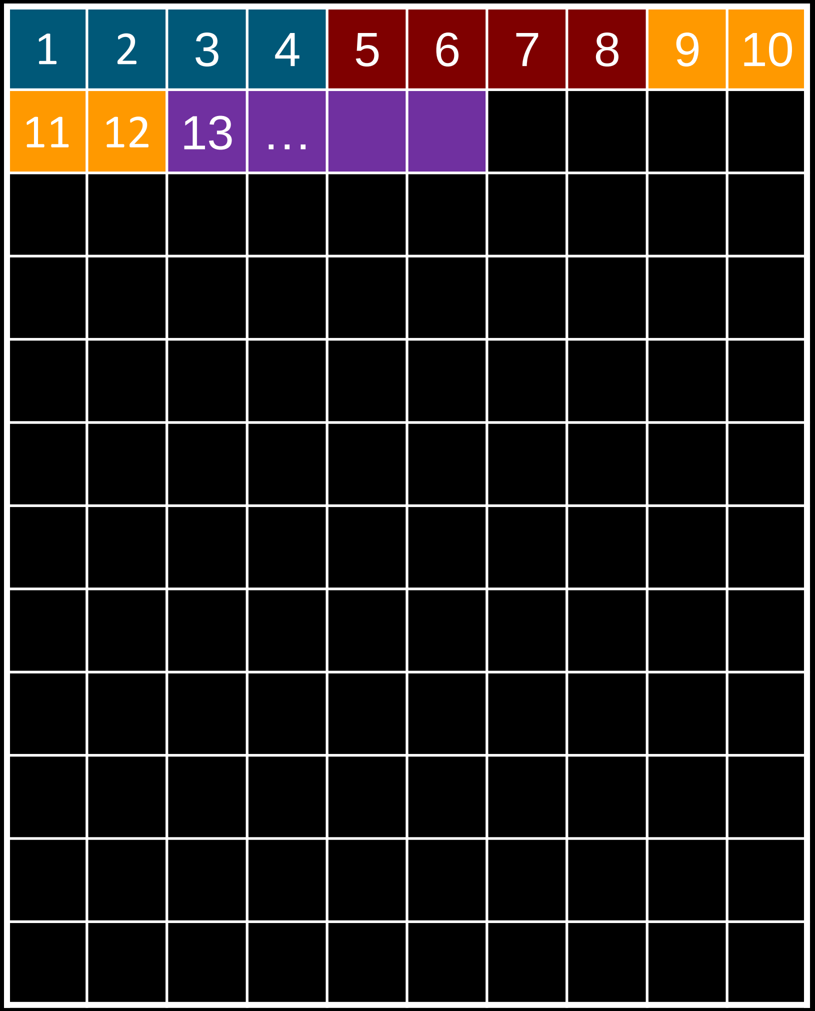
1	11	21							
		2	12	22					
				3	13	23			
						4	14	24	
								5	15
25									
6	16	26							
		7	17	…					
				8	18				
						9	19		
								10	20

Every access is a cache miss!

(unless *entire* matrix can fit in cache)

```
// H = 12, W = 10
int A[H][W];

for(y=0; y < H; y++)
   for(x=0; x < W; x++)
      sum += A[y][x];
```



Block size = 4 → 75% hit rate

Block size = 8 → 87.5% hit rate

Block size = 16 → 93.75% hit rate

And you can easily prefetch to warm the cache.

Caching assumptions

- small working set: 90/10 rule
- can predict future: spatial & temporal locality

Benefits

- (big & fast) built from (big & slow) + (small & fast)

Tradeoffs:

associativity, line size, hit cost, miss penalty, hit rate

Memory performance matters!

- often more than CPU performance
- … because it is the bottleneck, and not improving much
- … because most programs move a LOT of data

Design space is huge

- Gambling against program behavior
- Cuts across all layers:
  users → programs → os → hardware

Multi-core / Multi-Processor is complicated

- Inconsistent views of memory
- Extremely complex protocols, very hard to get right