CSCI-564 Advanced Computer Architecture

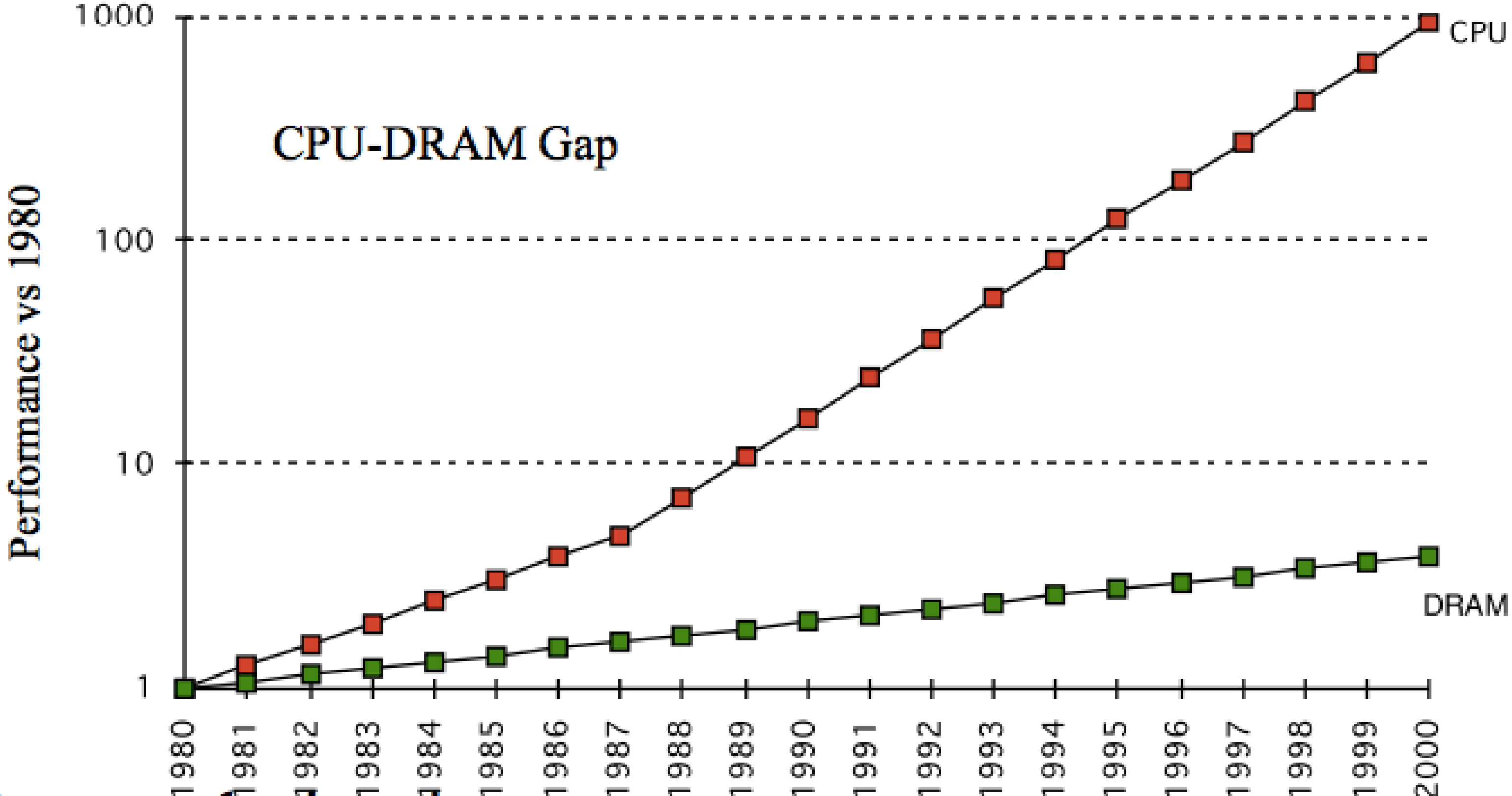
Lecture 4: Review of Memory Hierarchy

Bo Wu

Colorado School of Mines

Why do we need memory hierarchy?

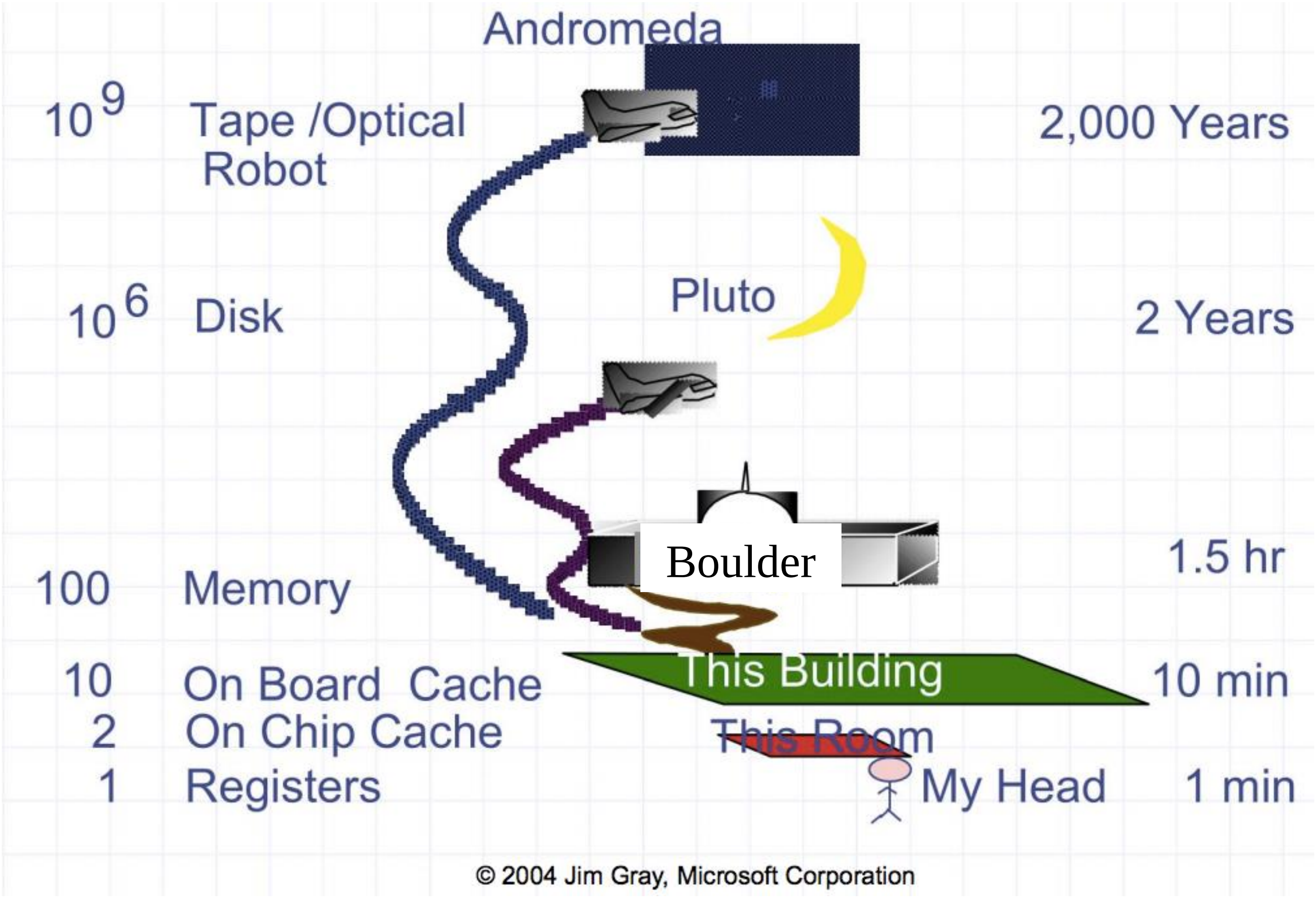
- Processor vs Memory Performance



1980: no cache in microprocessor;

1995 2-level cache

Really, how bad can it be?



Memory's impact

M = % mem ops
Mlat (cycles) = average memory latency
BCPI = base CPI with single-cycle data memory

CPI =

Memory's impact

M = % mem ops

Mlat (cycles) = average memory latency

TotalCPI = BaseCPI + M*Mlat

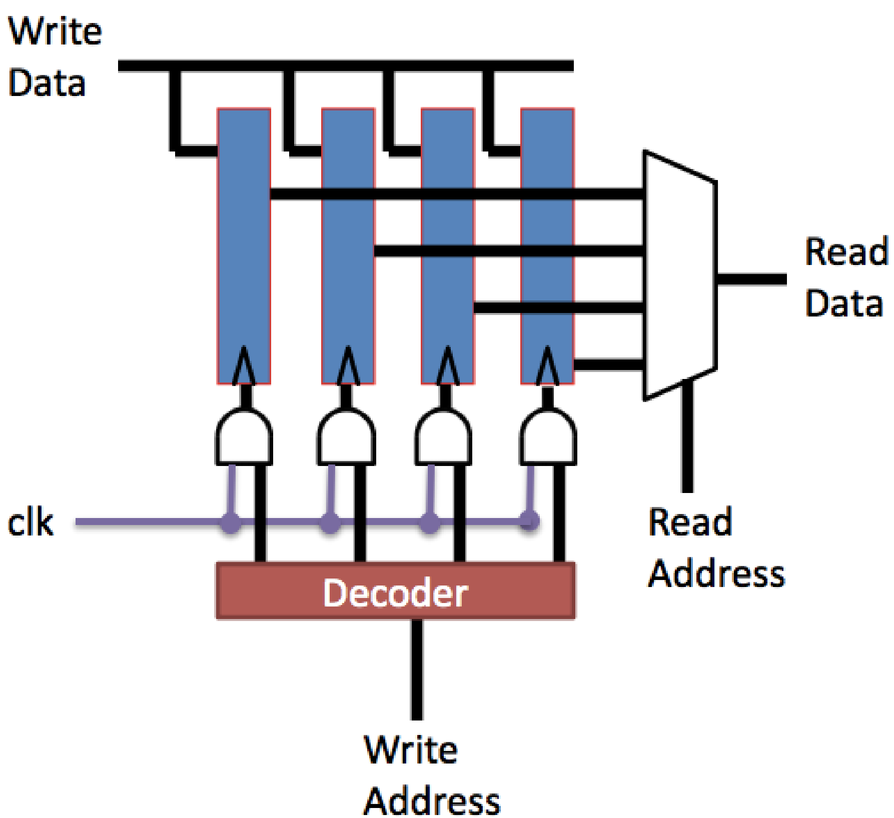
Example:

- BaseCPI = 1; M = 0.2; Mlat = 240 cycles
- TotalCPI = 49
- Speedup = 1/49 = 0.02 => 98% drop in performance

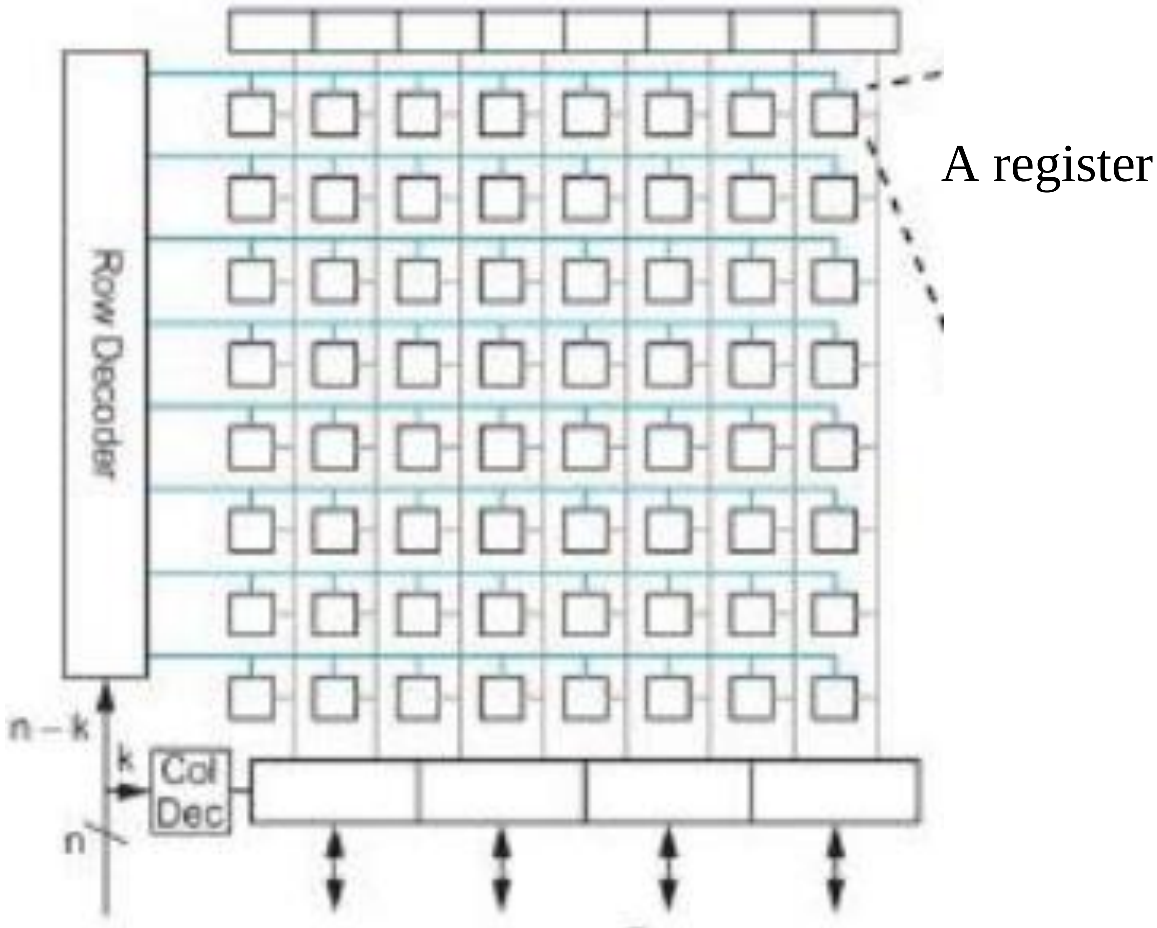
Remember!: Amdahl's law does not bound the slowdown. Poor memory performance can make your program arbitrarily slow.

More deeply into register file

Naive Register File



More deeply into register file



Why is cache fast?

- Registers and cache are built on SRAM (Static Random Access Memory) technology
  - not dense
  - Bandwidth
    - registers — 324 GB/S
    - L1 cache — 128 GB/S
- Main memory is built on DRAM (Dynamic Random Access Memory) technology
  - need refreshing to keep data
  - very dense
  - Bandwidth of DDR3 — 16 GB/S per DIMM (dual in-line memory module)

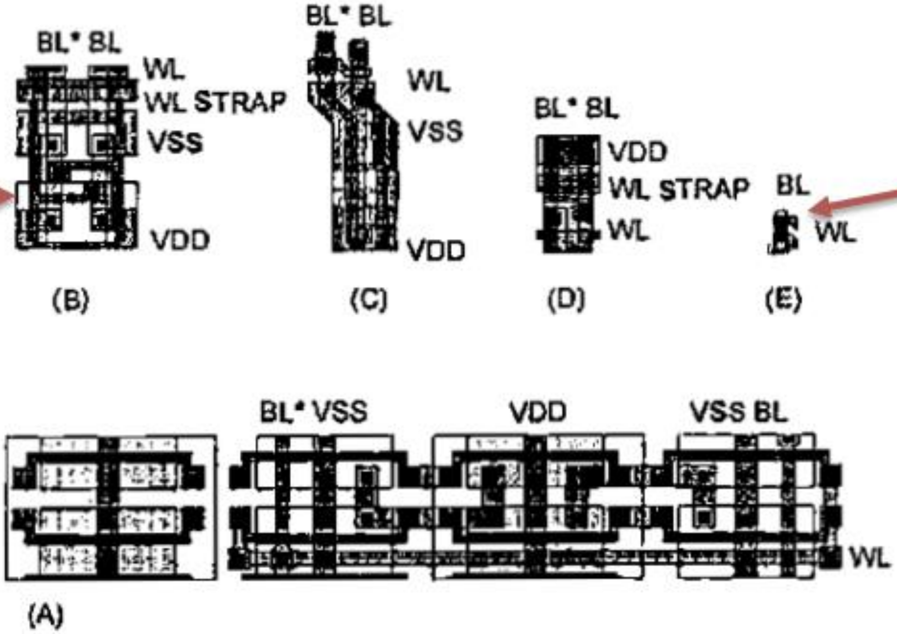
OK, now I understand cache is fast, but why don’t we build SRAM main memory?

SRAM is much more expensive than DRAM!

What does “dense” mean?

On-Chip SRAM on logic chip

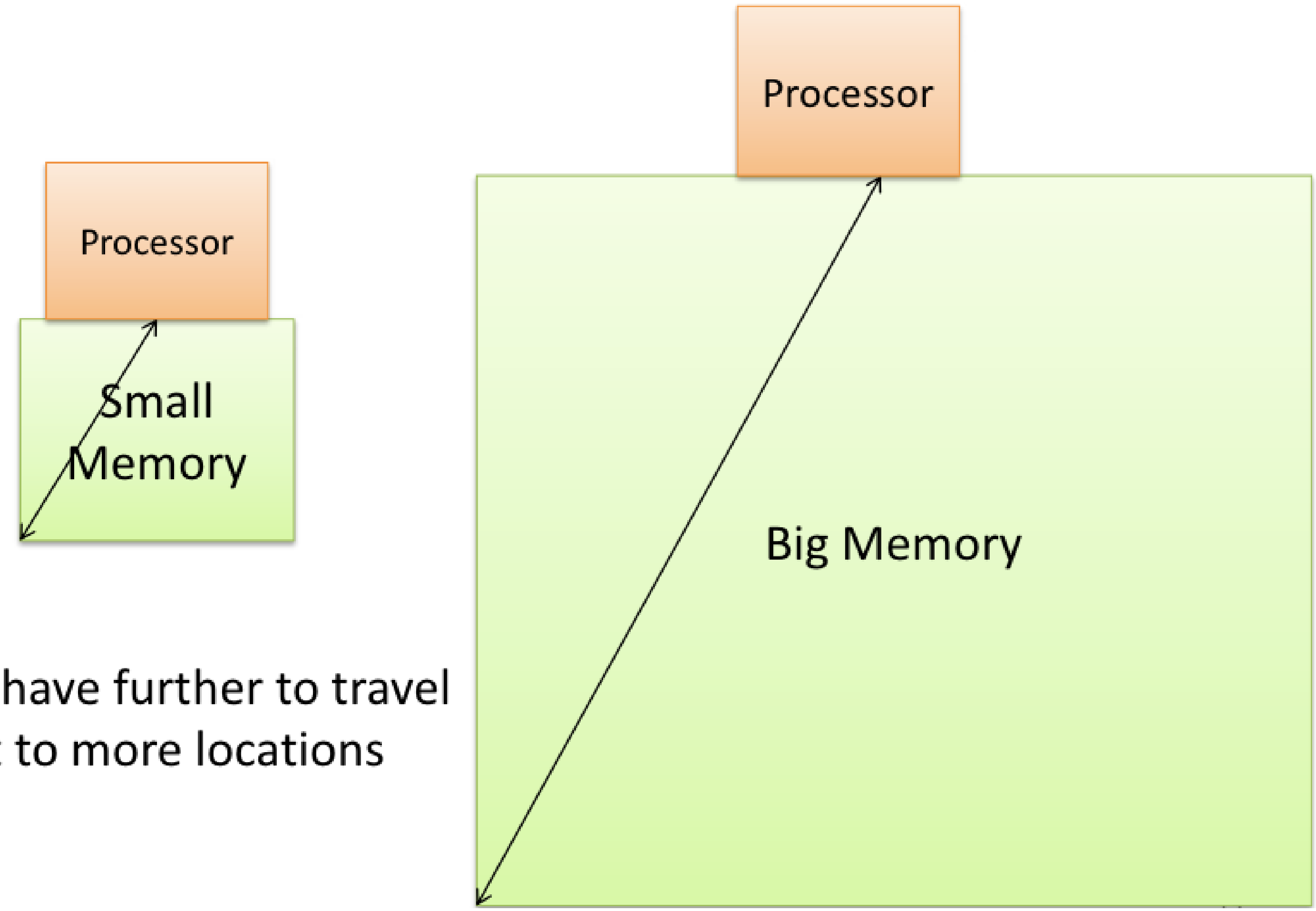
DRAM on memory chip



1 Memory cell in 0.5µm processes
a) Gate Array SRAM
b) Embedded SRAM
c) Standard SRAM (6T cell with local interconnect)
d) ASIC DRAM
e) Standard DRAM (stacked cell)

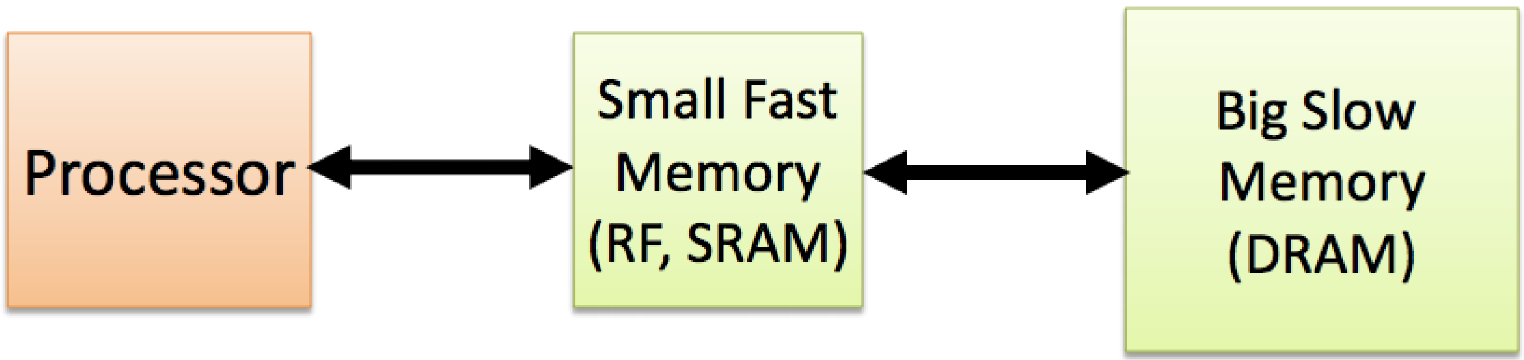
[From Foss, R.C. “Implementing Application-Specific Memory”, ISSCC 1996]

Why is cache fast?



- Signals have further to travel
- Fan out to more locations

Memory hierarchy



- Capacity: Register << SRAM << DRAM
- Latency: Register << SRAM << DRAM
- Bandwidth: on-chip >> off-chip
- On a data access:
  - if data is in fast memory -> low-latency access to SRAM
  - if data is not in fast memory -> long-latency access to DRAM
- Memory hierarchies only work if the small, fast memory actually stores data that is reused by the processor

Cache Terminology

- Hit: accessed data found at current level
  - hit rate: fraction of accesses that finds the data
  - hit time: time to access data on a hit
- Miss: accessed data NOT found at current level
  - miss rate: 1 – hit rate
  - miss penalty: time to get block from lower level

hit time << miss penalty

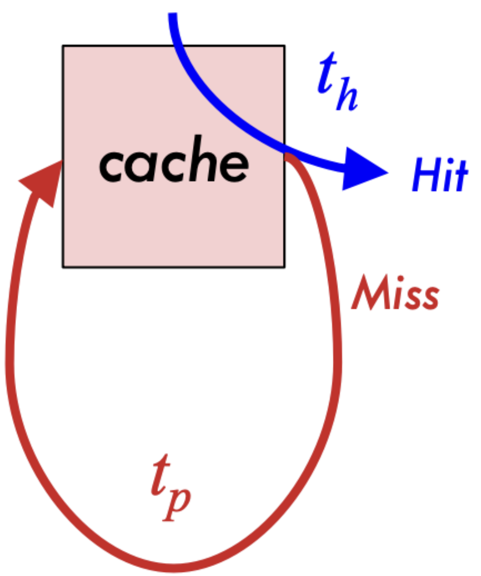
Average Memory Access Time (AMAT)

Outcome	Rate	Access Time
Hit	r_h	t_h
Miss	r_m	$t_h + t_p$

$$r_h = 1 - r_m$$

$$AMAT = r_h t_h + r_m (t_h + t_p)$$

$$AMAT = t_h + r_m t_p$$



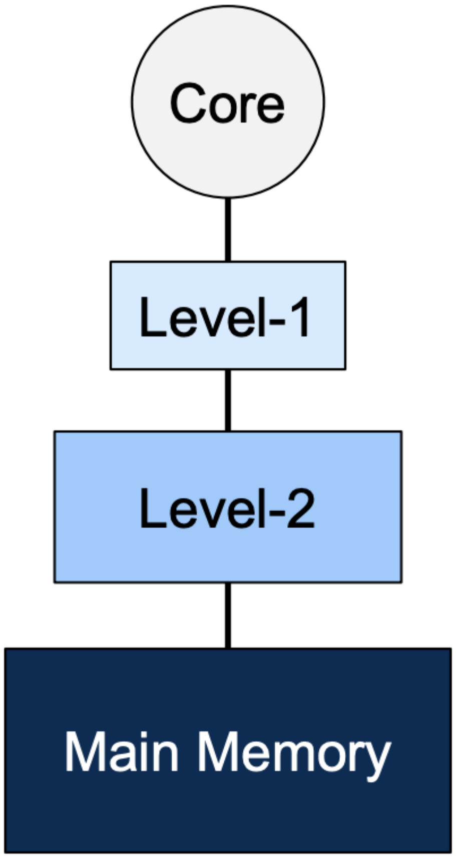
- Example: hit rate is 90%; hit time is 2 cycles; and accessing the lower level takes 200 cycles; find the average memory access time

AMAT = 2 + 0.1x200 = 22 cycles

Example problem

- Assume that the miss rate for instructions is 5%; the miss rate for data is 8%; the data references per instruction is 40%; and the miss penalty is 20 cycles; find performance relative to perfect cache with no misses
  - misses/instruction = 0.05 + 0.08 x 0.4 = 0.082
  - Assuming hit time =1
    - AMAT = 1 + 0.082 x 20 = 2.64
    - Relative performance = 1/2.64

Cache's Impact



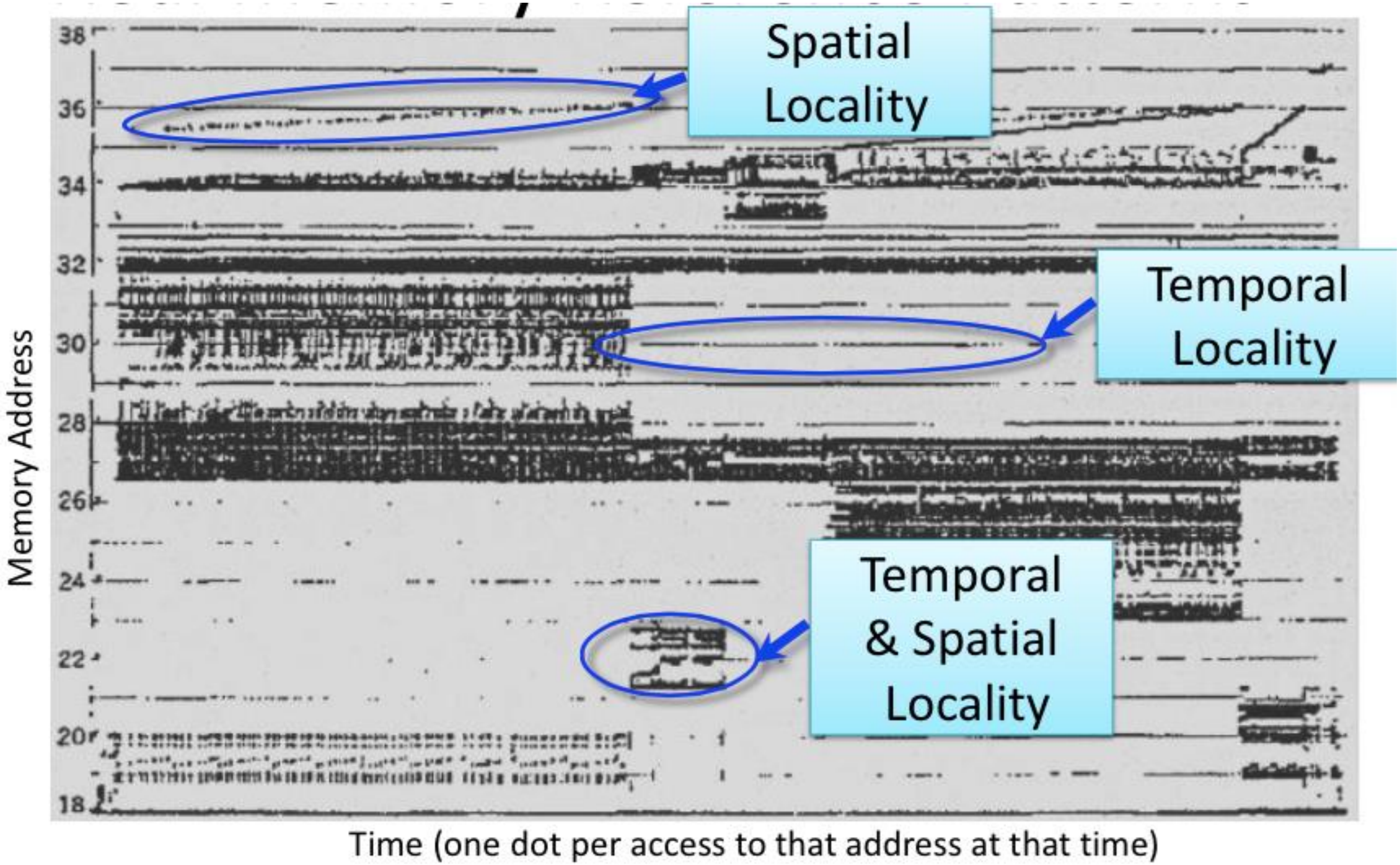
- Main memory access time: 300 cycles
- Two level cache
  - L1: 2 cycles hit time; 60% hit rate
  - L2: 20 cycles hit time; 70% hit rate
- What is the average mem access time?

$$AMAT = t_{h1} + r_{m1}\, t_{p1}$$
$$t_{p1} = t_{h2} + r_{m2}\, t_{p2}$$
$$AMAT = 46$$

The principle of locality

- “Locality” is the tendency of data access to be predictable. There are two kinds:
    - Spatial locality: The program is likely to access data that is close to data it has accessed recently
    - Temporal locality: The program is likely to access the same data repeatedly.

Evidence of locality



Locality in action

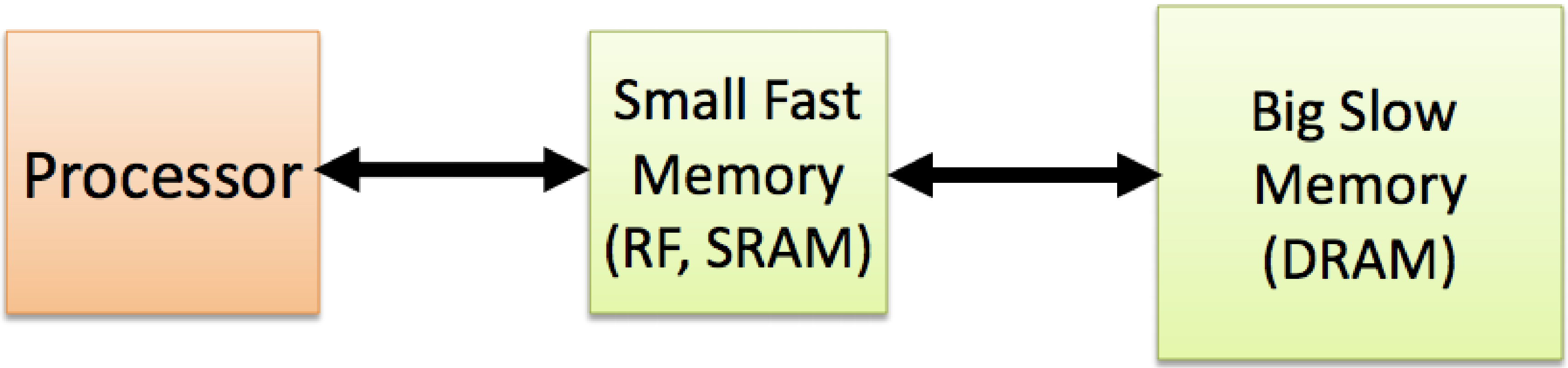
- Label each access with whether it has temporal or spatial locality or neither
  - 1
  - 2
  - 3
  - 10
  - 4
  - 1800
  - 11
  - 30

- 1
- 2
- 3
- 4
- 10
- 190
- 11
- 30
- 12
- 13
- 182
- 1004

Locality in action

- Label each access with whether it has temporal or spatial locality or neither
  - 1 n
  - 2 s
  - 3 s
  - 10 n
  - 4 s
  - 1800 n
  - 11 s
  - 30 n
  - 1 t
  - 2 s, t
  - 3 s,t
  - 4 s,t
  - 10 s,t
  - 190 n
  - 11 s,t
  - 30 t
  - 12 s
  - 13 s
  - 182 n?
  - 1004 n

Caches exploit both types of locality

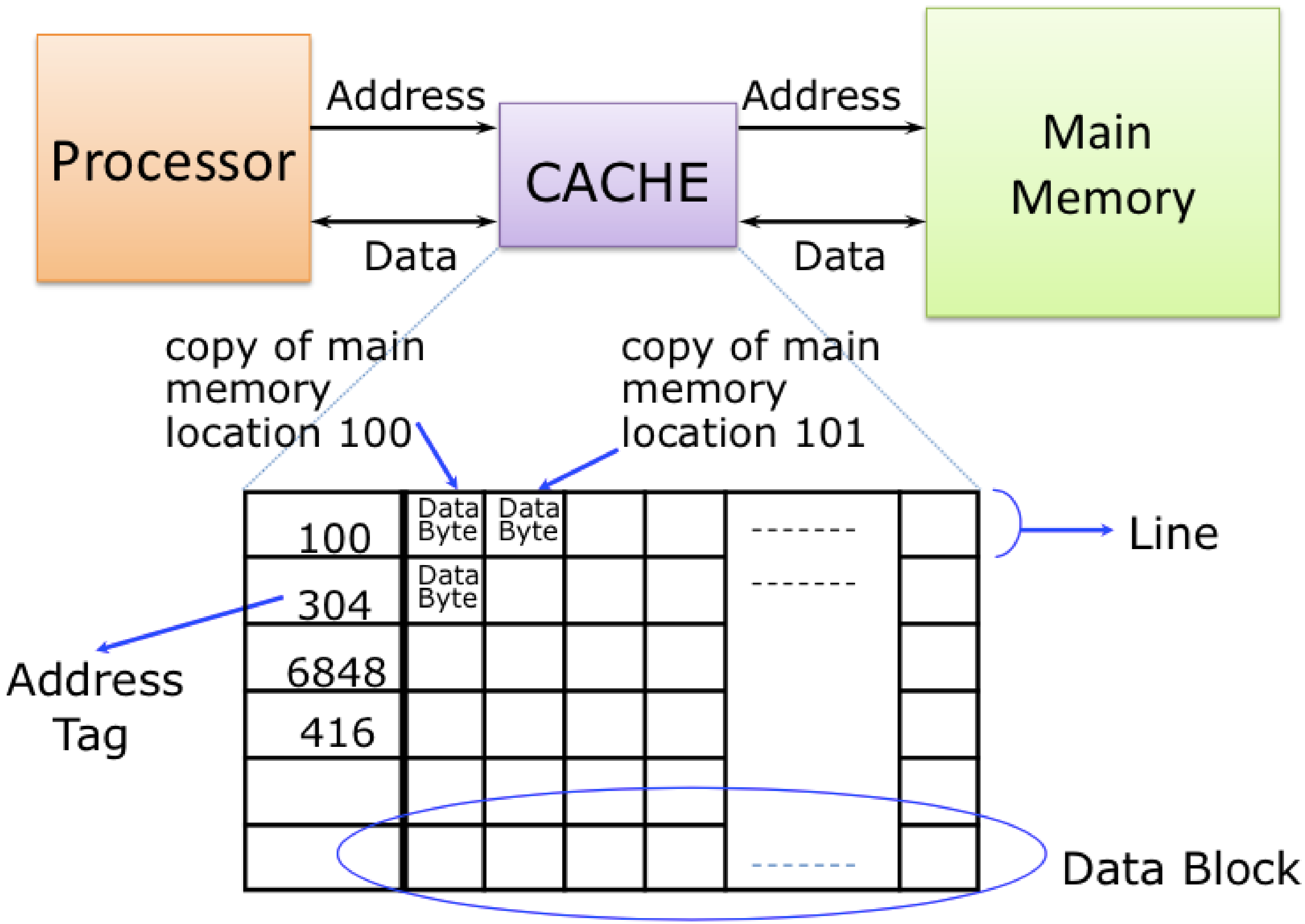


- Exploit temporal locality by remembering the contents of recently accessed locations
- Exploit spatial locality by fetching blocks of data around recently accessed locations

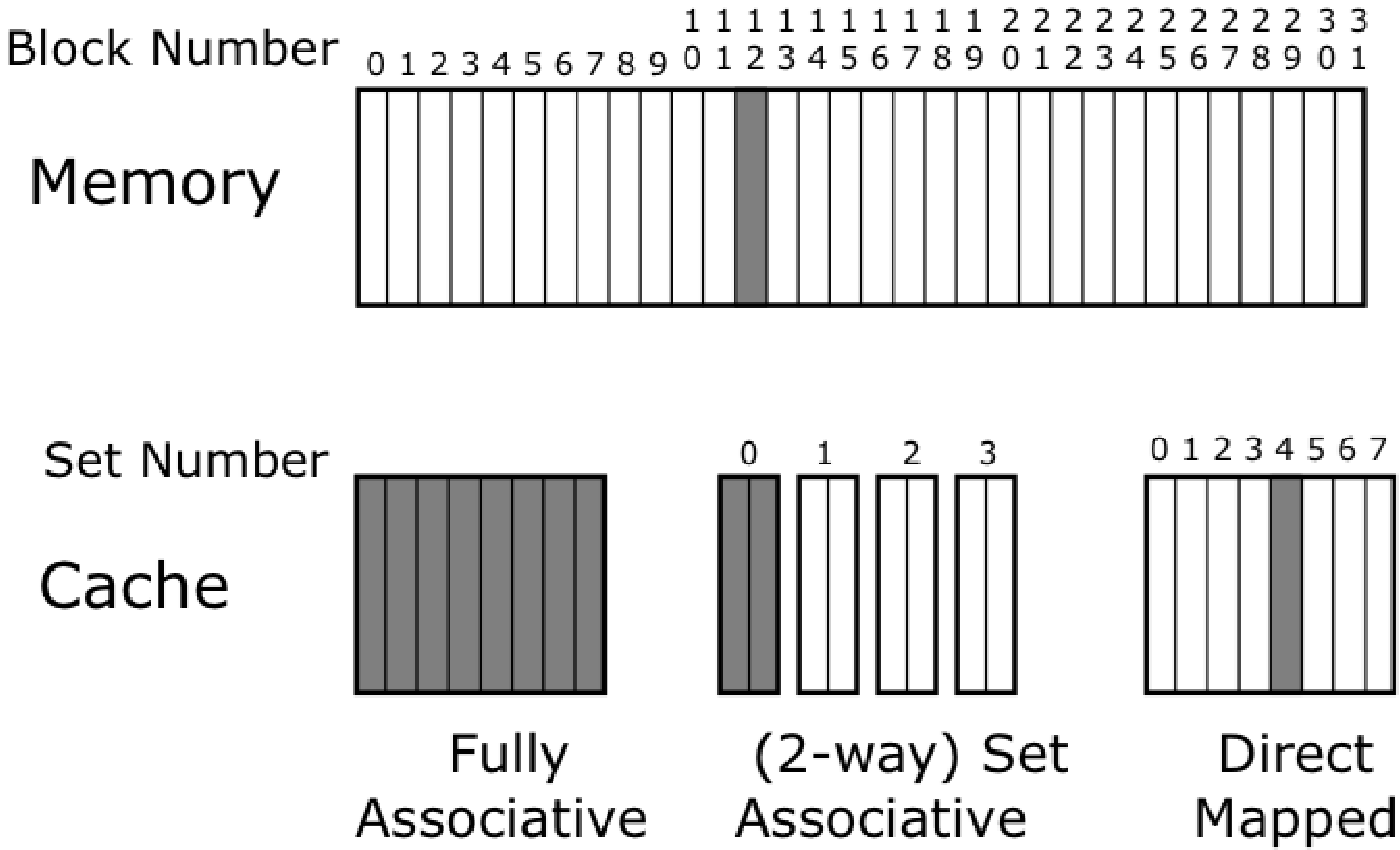
Basic problems in caching

- A cache holds a small fraction of all the cache lines, yet the cache itself may be quite large (i.e., it might contains 1000s of lines)
- Where do we look for our data?
- How do we tell if we've found it and whether it's any good?

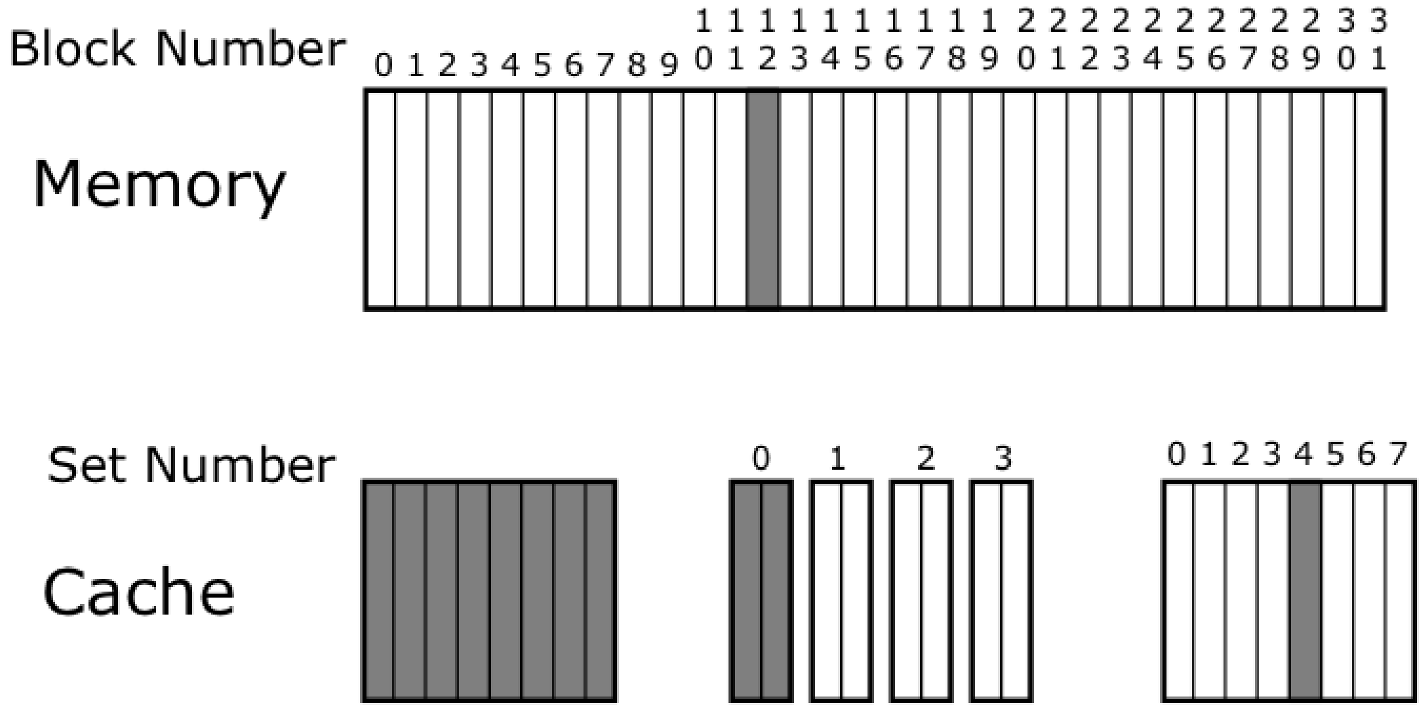
Basic cache organization



Where to place data in cache?



Where to place data in cache?



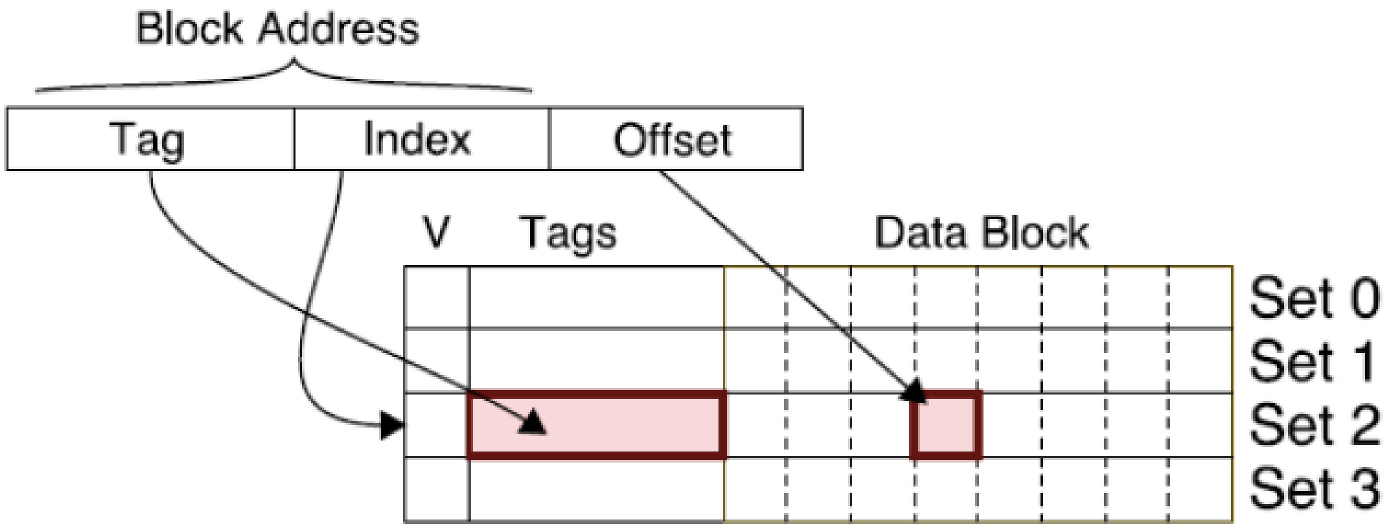
block 12 can be placed

Fully Associative
anywhere

(2-way) Set Associative
anywhere in set 0
(12 mod 4)

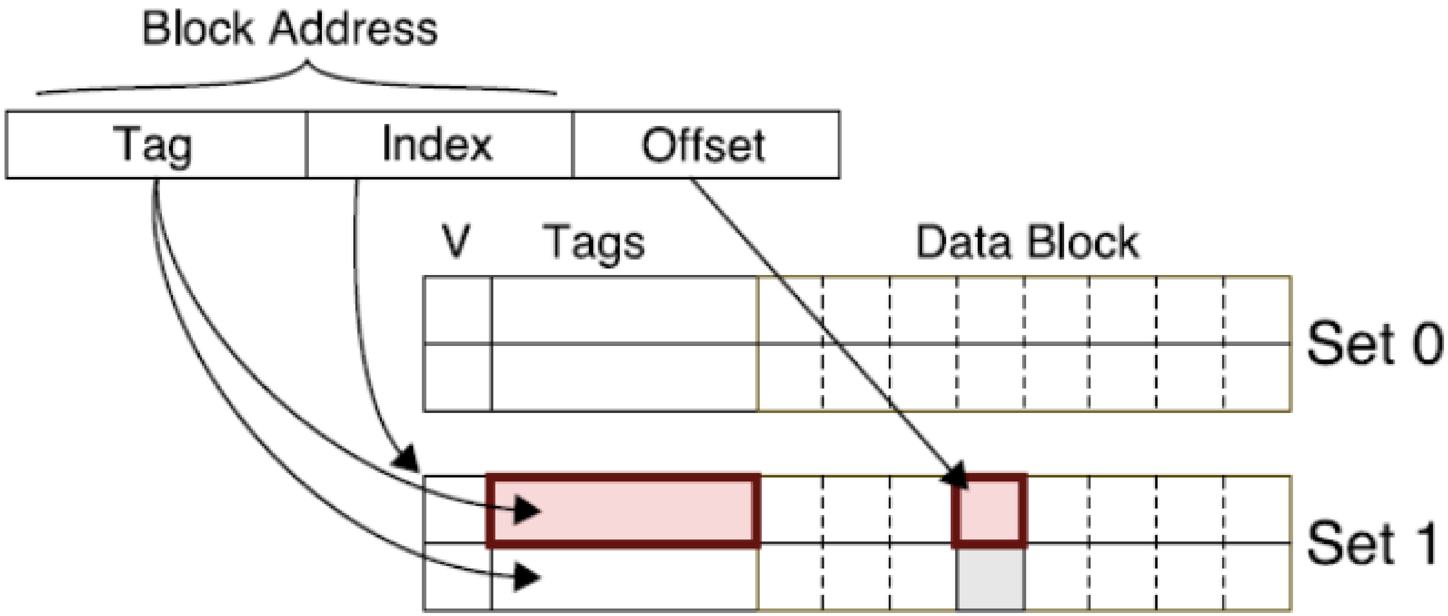
Direct Mapped
only into block 4
(12 mod 8)

How to find block in cache?



- Cache uses index and offset to find potential match, then checks tag
- Tag check only includes higher order bits
- In this example (Direct-mapped, 8B block, 4 line cache)

How to find block in cache?



- Cache checks all potential blocks with parallel tag check
- In this example (2-way associative, 8B block, 4 line cache)

The cost of associativity

- Increased associativity requires multiple tag checks
  - N-Way associativity requires N parallel comparators
  - This is expensive in hardware and potentially slow.
- This limits associativity L1 caches to 2-8.
- Larger, slower caches can be more associative.
- Example: Nehalem
  - 8-way L1
  - 16-way L2 and L3.
- Core 2’s L2 was 24-way

New cache geometry calculations

- Addresses break down into: tag, index, and offset.
- How they break down depends on the "cache geometry"

- Cache lines = L
- Cache line size = B
- Address length = A (32 bits in our case)
- Associativity = W

- Index bits = $\log_2(L/W)$
- Offset bits = $\log_2(B)$
- Tag bits = A - (index bits + offset bits)

Practice

- 32KB, 2048 Lines, 4-way associative.

- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits: 19
- Offset bits: 4

Write through vs. write back

- When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
- If we *do not* forward the write, the cache is "Write back", since the data must be written back when it's evicted (i.e., the line can be dirty)
- If we *do* forward the write, the cache is "write through." In this case, a cache line is never dirty.
- Write back advantages

Fewer writes farther down the hierarchy. Less bandwidth. Faster writes

- Write through advantages

No write back required on eviction.

Write allocate/no-write allocate

- If the cache allocates cache lines on a write miss, it is *write allocate*, otherwise, it is *no write allocate*.
- Write Allocate advantages

Exploits temporal locality. Data written will be read soon, and that read will be faster.

- No-write allocate advantages

Fewer spurious evictions. If the data is not read in the near future, the eviction is a waste.

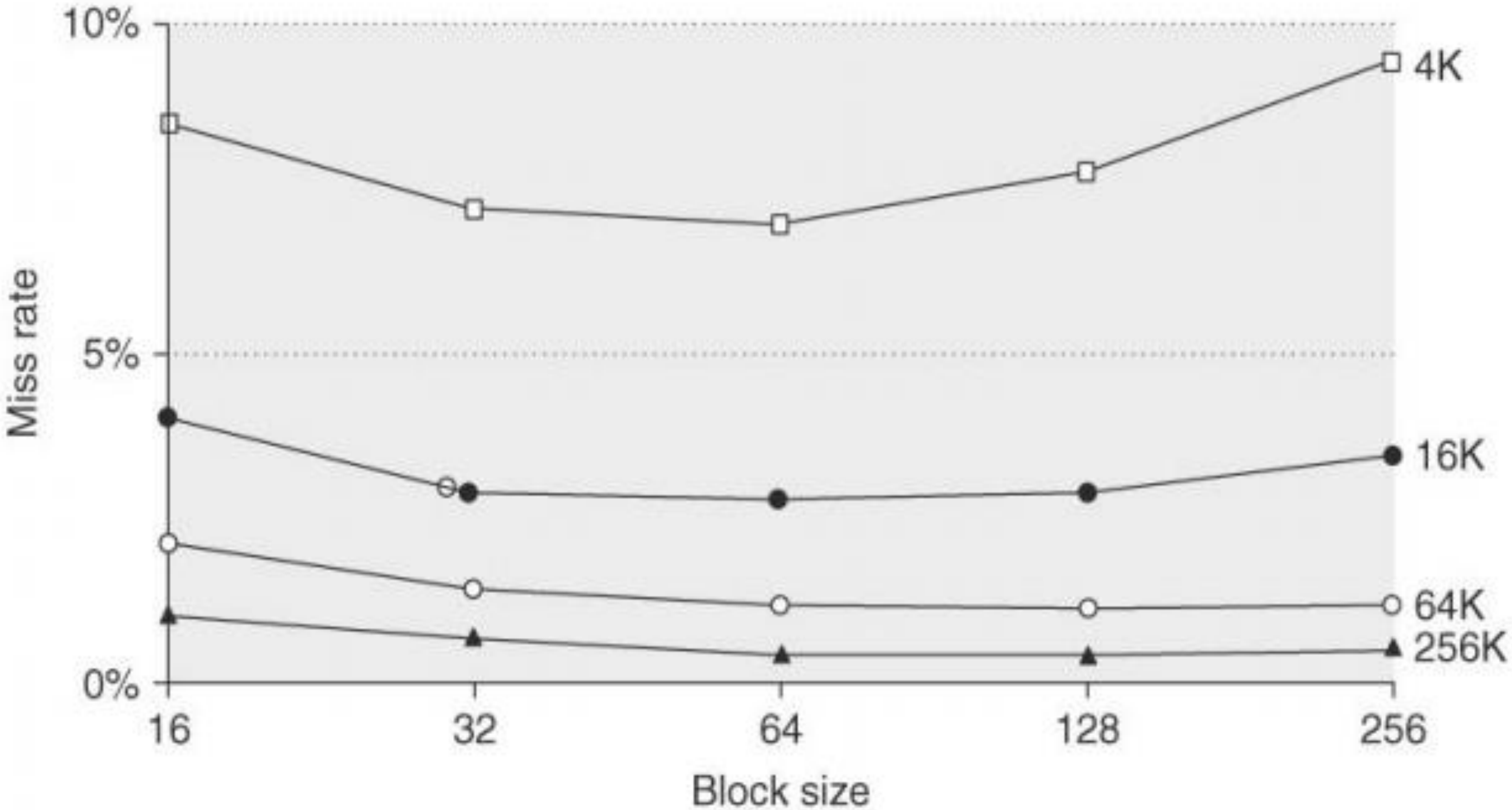
Eviction in associative caches

- We must choose which line in a set to evict if we have associativity
- How we make the choice is called *the cache eviction policy*
  - Random -- always a choice worth considering.
  - Least recently used (LRU) -- evict the line that was last used the longest time ago.
  - Prefer clean -- try to evict clean lines to avoid the write back.
  - Farthest future use -- evict the line whose next access is farthest in the future. This is provably optimal. It is also impossible to implement.

Cache line size

- How big should a cache line be?
- Why is bigger better?
  - Exploits more spatial locality.
  - Large cache lines effectively *prefetch* data that we have not explicitly asked for.
- Why is smaller better?
  - Focuses on temporal locality.
  - If there is little spatial locality, large cache lines waste space and bandwidth.
- In practice 32-64 bytes is good for L1 caches were space is scarce and latency is important.
- Lower levels use 128-256 bytes.

Cache line size



Data vs. instruction cache

- Why have different I and D caches?
  - Different areas of memory
  - Different access patterns
    - I-cache accesses have lots of spatial locality. Mostly sequential accesses.
    - I-cache accesses are also predictable to the extent that branches are predictable
    - D-cache accesses are typically less predictable
  - Not just different, but often across purposes.
    - Sequential I-cache accesses may interfere with the data the D-cache has collected.
    - This is “interference” just as we saw with branch predictors
  - At the L1 level it avoids a structural hazard in the pipeline
  - Writes to the I cache by the program are rare enough that they can be slow (i.e., self modifying code)