CSC 2224: Parallel Computer Architecture and Programming Memory Hierarchy & Caches

Prof. Gennady Pekhimenko

University of Toronto

Fall 2019

*The content of this lecture is adapted from the lectures of Onur Mutlu @ CMU and ETH*

Reviews: Cache Compression

- Review:

  - Pekhimenko et al., "**Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches**," PACT 2012

Project Proposal Deadline

- Deadline is on October 1st

- Send emails with your proposals (PDFs) to csc2224arch@gmail.com

Memory (Programmer's View)

Store ⟶ [Memory]

Load ⟵ [Memory]
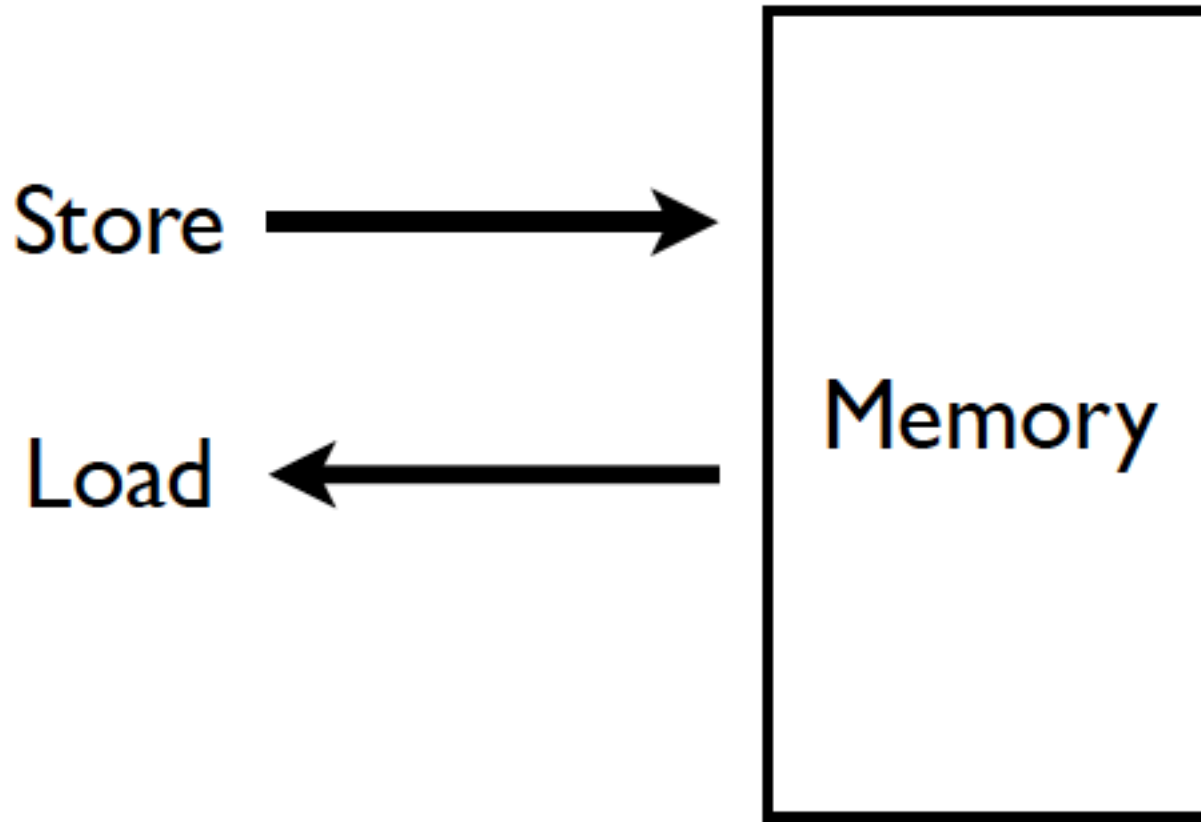
Virtual vs. Physical Memory

- Programmer sees virtual memory
  - Can assume the memory is "infinite"
- Reality: Physical memory size is much smaller than what the programmer assumes
- The system (system software + hardware, cooperatively) maps virtual memory addresses to physical memory
  - The system automatically manages the physical memory space transparently to the programmer

+ Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer

-- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff

5

(Physical) Memory System

- You need a larger level of storage to manage a small amount of physical memory automatically
  → Physical memory has a backing store: disk

- We will first start with the physical memory system

- For now, ignore the virtual → physical indirection

- We will get back to it when the needs of virtual memory start complicating the design of physical memory…

Idealism

Instruction Supply		Pipeline (Instruction execution)		Data Supply
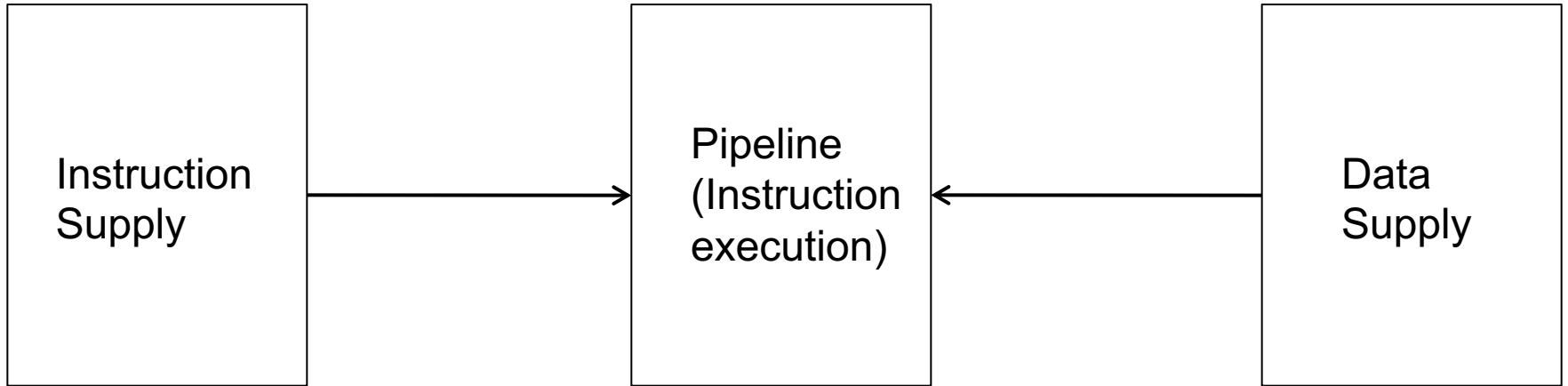
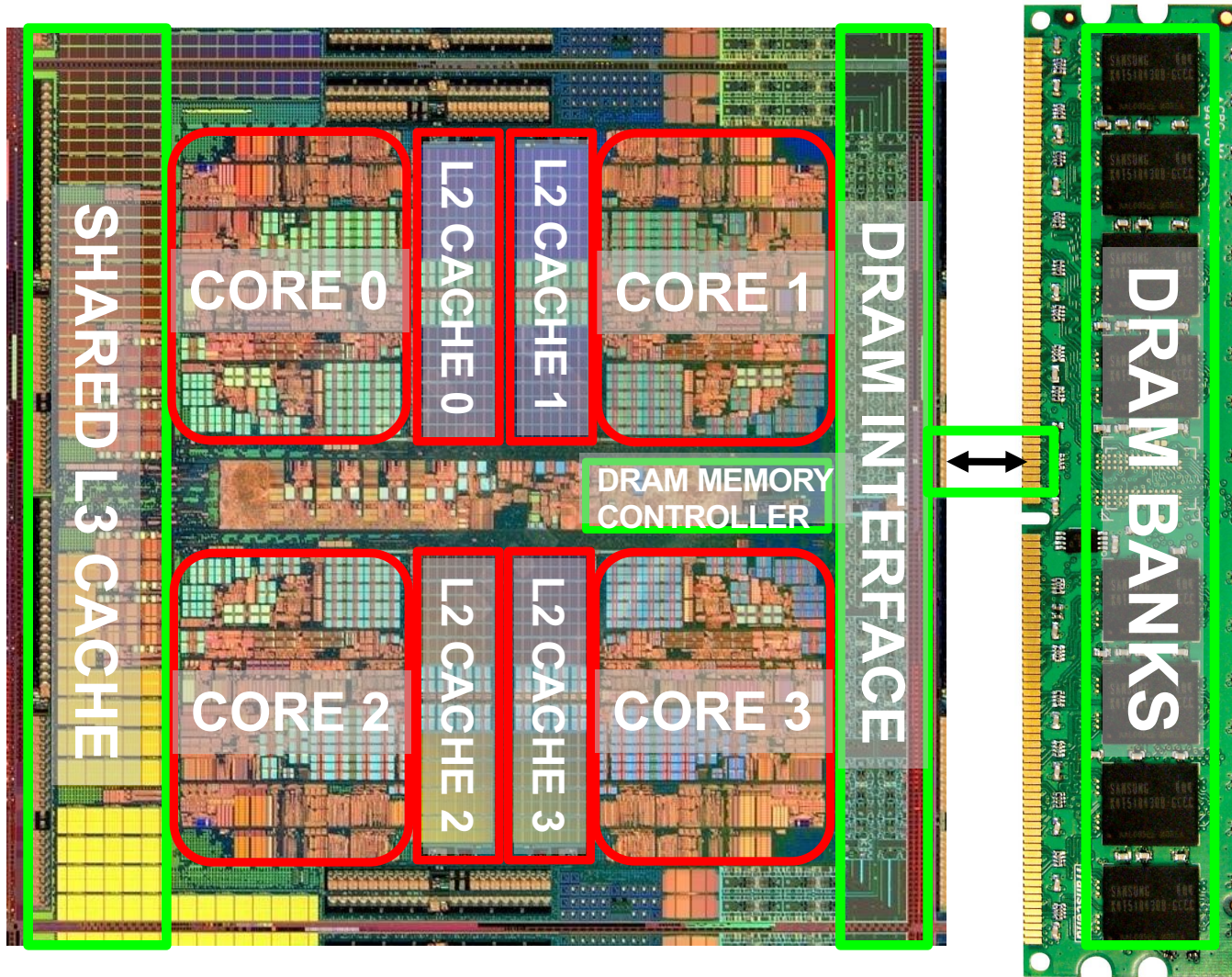- Zero latency access

- Infinite capacity

- Zero cost

- Perfect control flow

- No pipeline stalls

-Perfect data flow
  (reg/memory dependencies)

- Zero-cycle interconnect
  (operand communication)

- Enough functional units

- Zero latency compute

- Zero latency access

- Infinite capacity

- Infinite bandwidth

- Zero cost

The Memory Hierarchy

# Memory in a Modern System

Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
 - Need more banks, more ports, higher frequency, or faster technology

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor

- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed
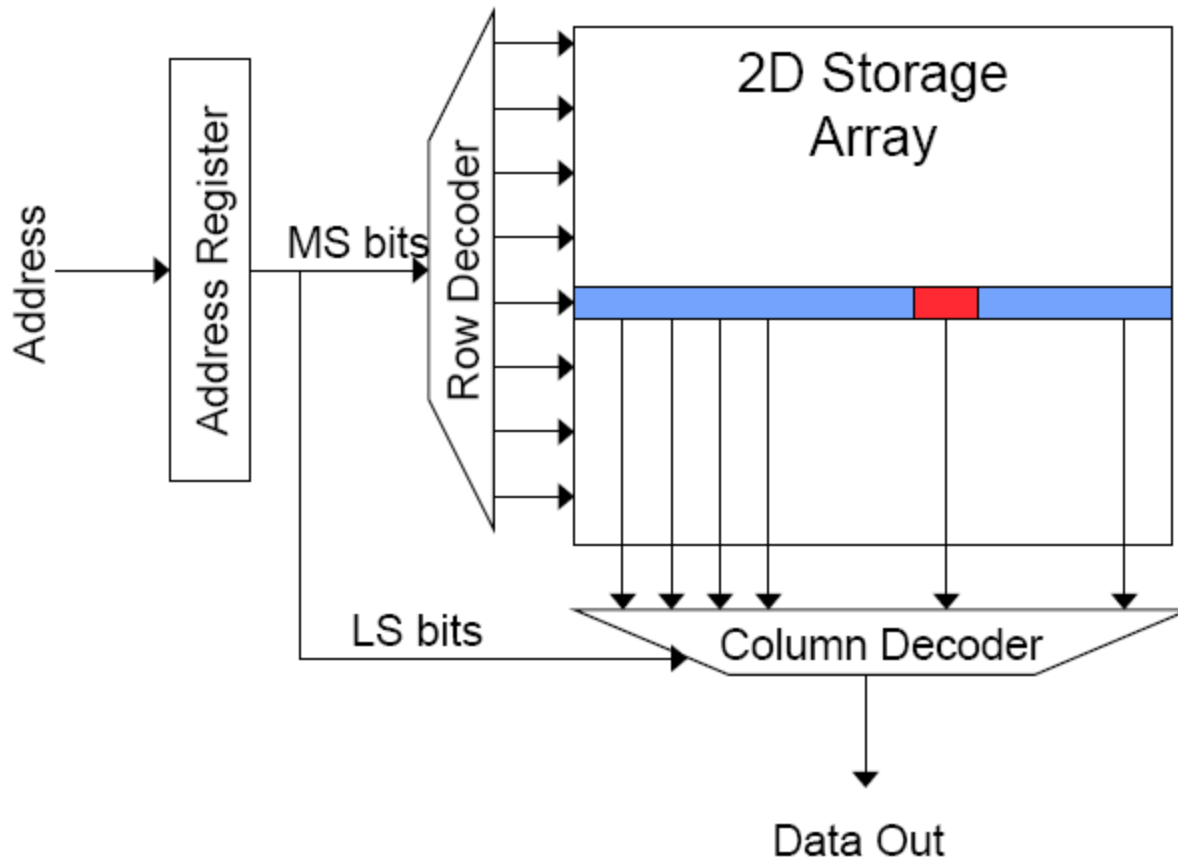
row enable

_bitline
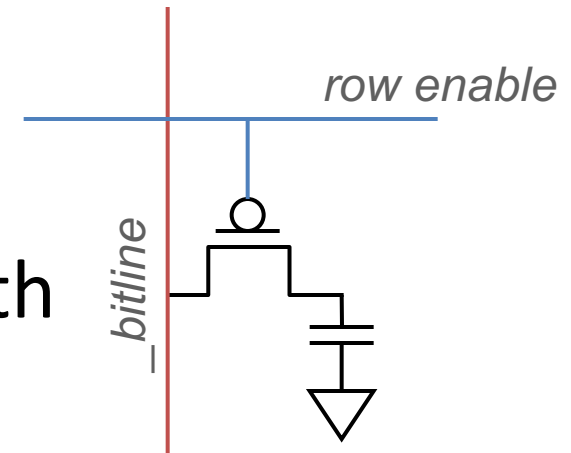
Memory Technology: SRAM

- Static random access memory

- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the "cell"
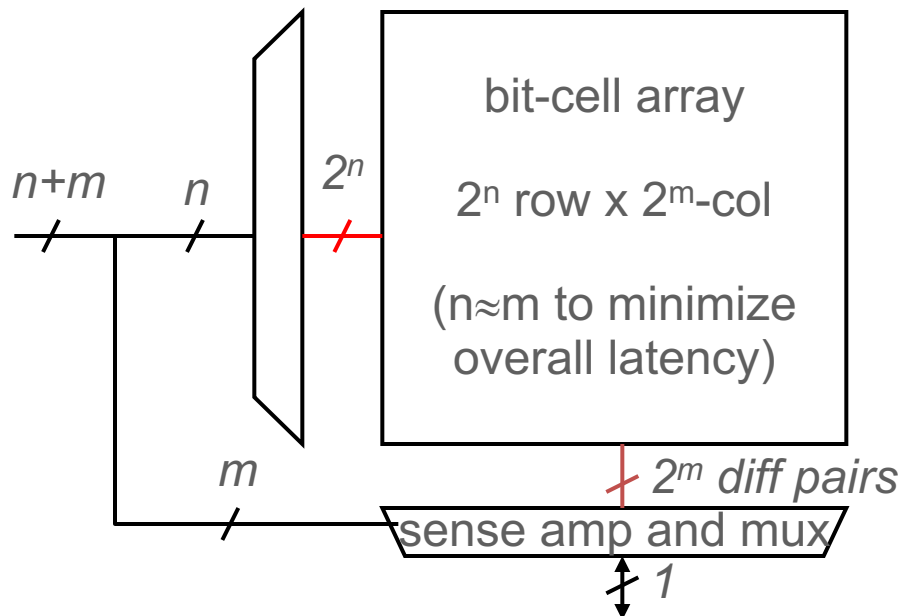 - 4 transistors for storage
  - 2 transistors for access
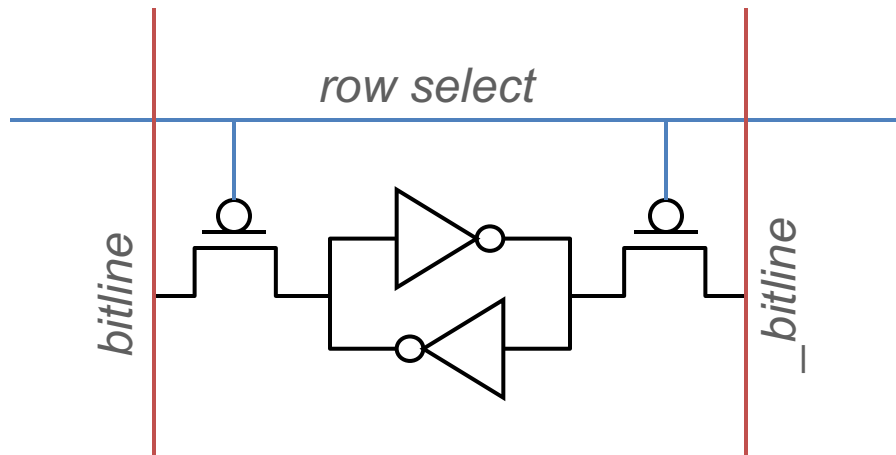
Memory Bank Organization and Operation

- Read access sequence:

1. Decode row address & drive word-lines

2. Selected bits drive bit-lines
   - Entire row read

3. Amplify row data

4. Decode column address & select subset of row
   - Send to output

5. Precharge bit-lines
   - For next access

SRAM (Static Random Access Memory)
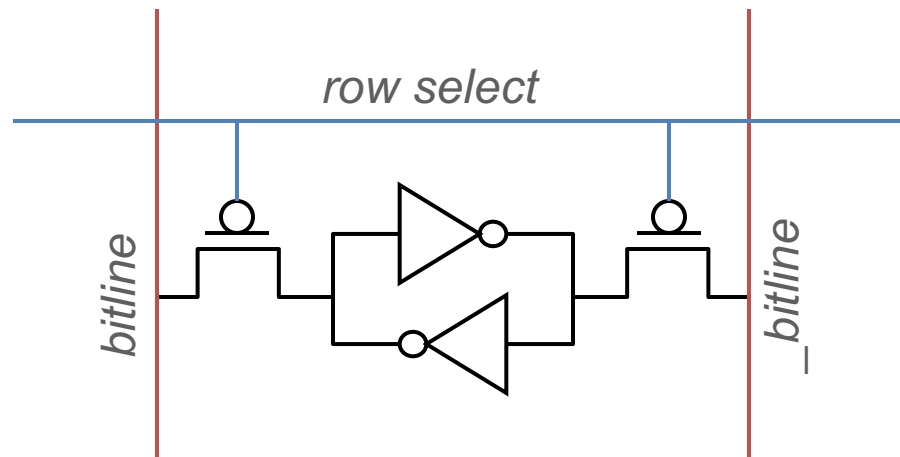
Read Sequence

1. address decode
2. drive row select
3. selected bit-cells drive bitlines
   (entire row is read together)
4. differential sensing and column select
   (data is ready)
5. precharge all bitlines
   (for next read or write)

Access latency dominated by steps 2 and 3

Cycling time dominated by steps 2, 3 and 5

- step 2 proportional to 2^m
- step 3 and 5 proportional to 2^n

15

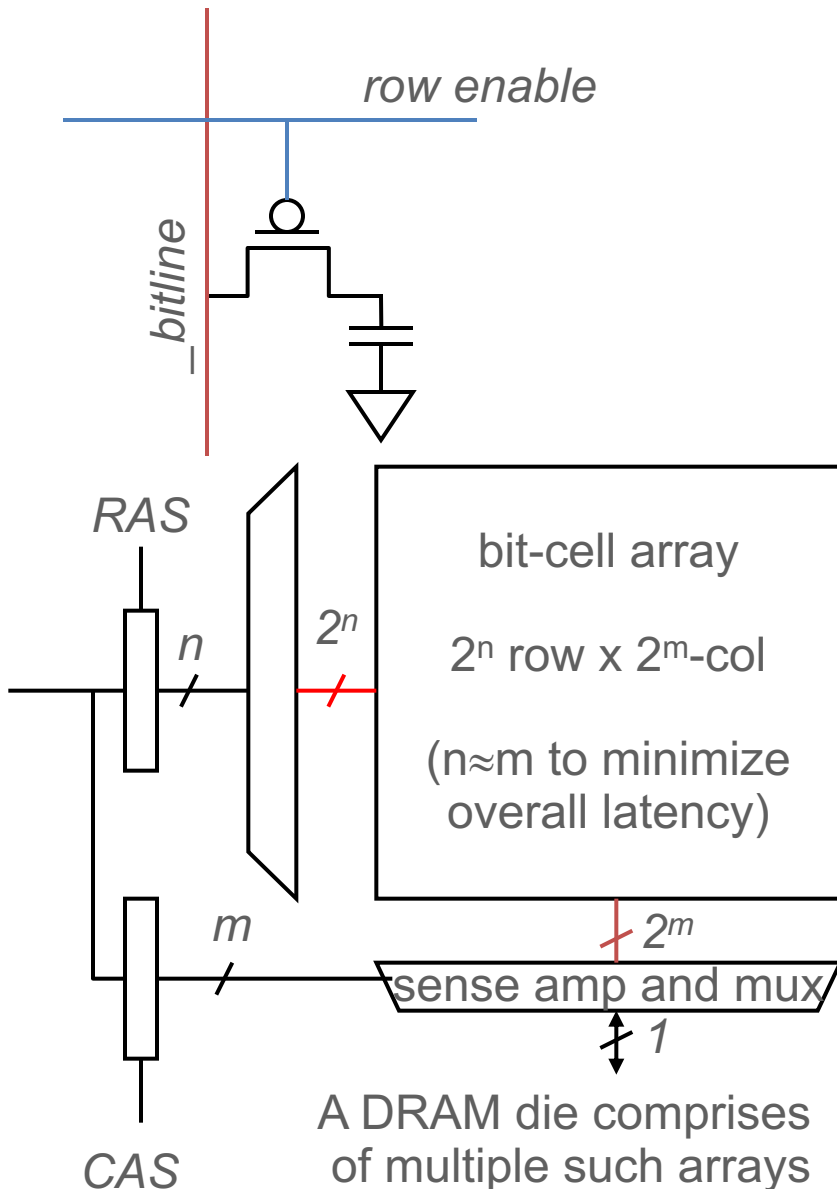# DRAM (Dynamic Random Access Memory)

row enable

bitline

bit-cell array

2^n row x 2^m-col

(n≈m to minimize overall latency)

RAS

n

2^n

m

2^m

sense amp and mux

1

CAS

A DRAM die comprises of multiple such arrays

Bits stored as charges on node capacitance (non-restorative)

- bit cell loses charge when read
- bit cell loses charge over time

Read Sequence

1~3 same as SRAM

4. a "flip-flopping" sense amp amplifies and regenerates the bitline, data bit is mux'ed out

5. precharge all bitlines

Destructive reads

Charge loss over time

Refresh: A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) such that charge is restored

16

DRAM vs. SRAM

- DRAM
  - Slower access (capacitor)
  - Higher density (1T 1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

- SRAM
  - Faster access (no capacitor)
  - Lower density (6T cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)

The Problem (data from 2011)

- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec

- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values (circa ~2011) scale with time

- Other technologies have their place as well
  - Flash memory, PC-RAM, MRAM, RRAM (not mature yet)

The Problem (Modern)

- Faster is more expensive (dollars and chip area)
  - SRAM, $5000 per GB
  - DRAM, < $100 per GB
  - **SSD, < $0.50 per GB**
  - Hard Disk < $0.04 per GB
  - **NVDIMM < $10 per GB**

Why Memory Hierarchy?

- We want both fast and large

- But we cannot achieve both with a single level of memory

- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

The Memory Hierarchy

move what you use here

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

backup everything here

fast small

big but slow

faster per byte

cheaper per byte
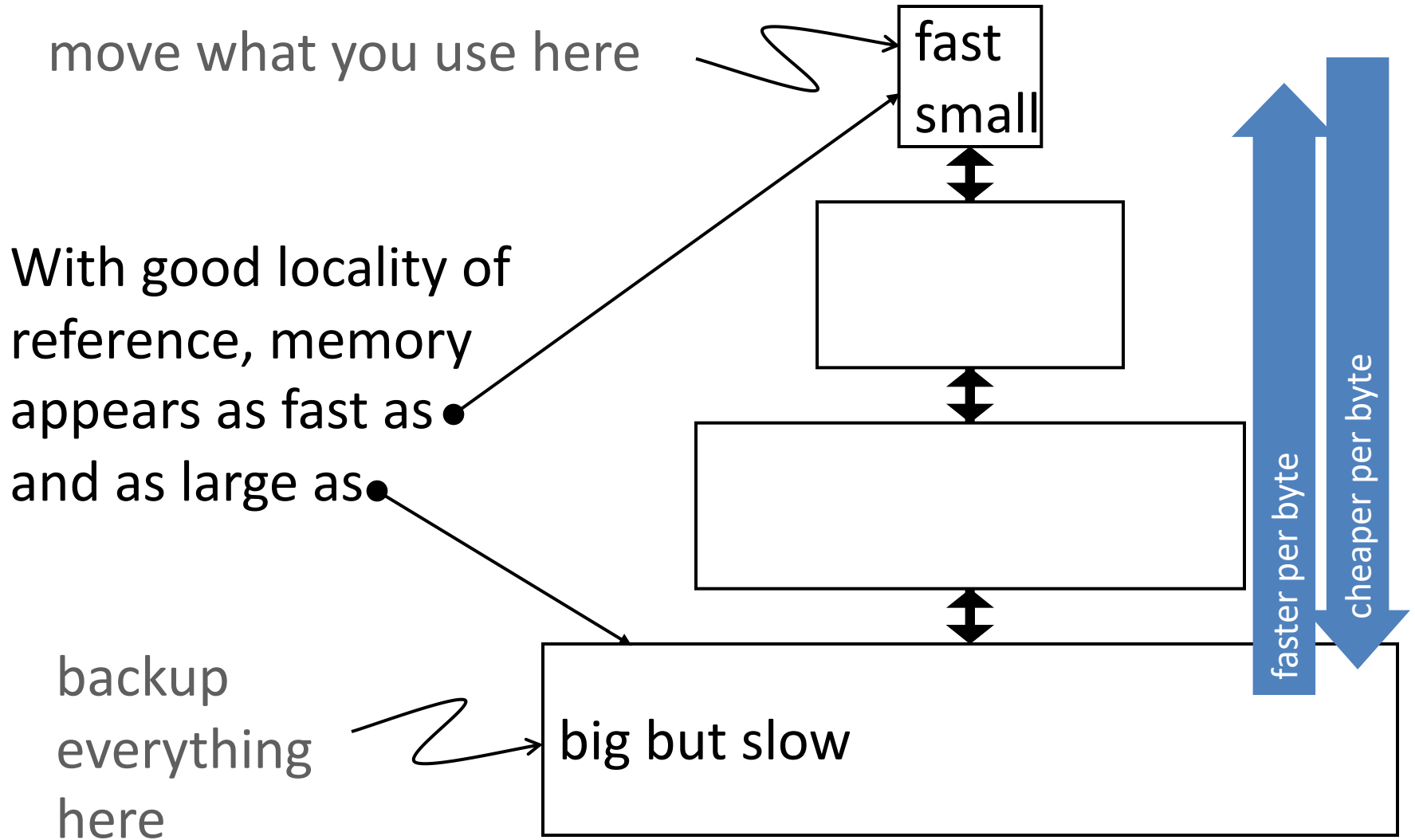
Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy

| CPU RF | Cache | | Main Memory (DRAM) | Hard Disk |
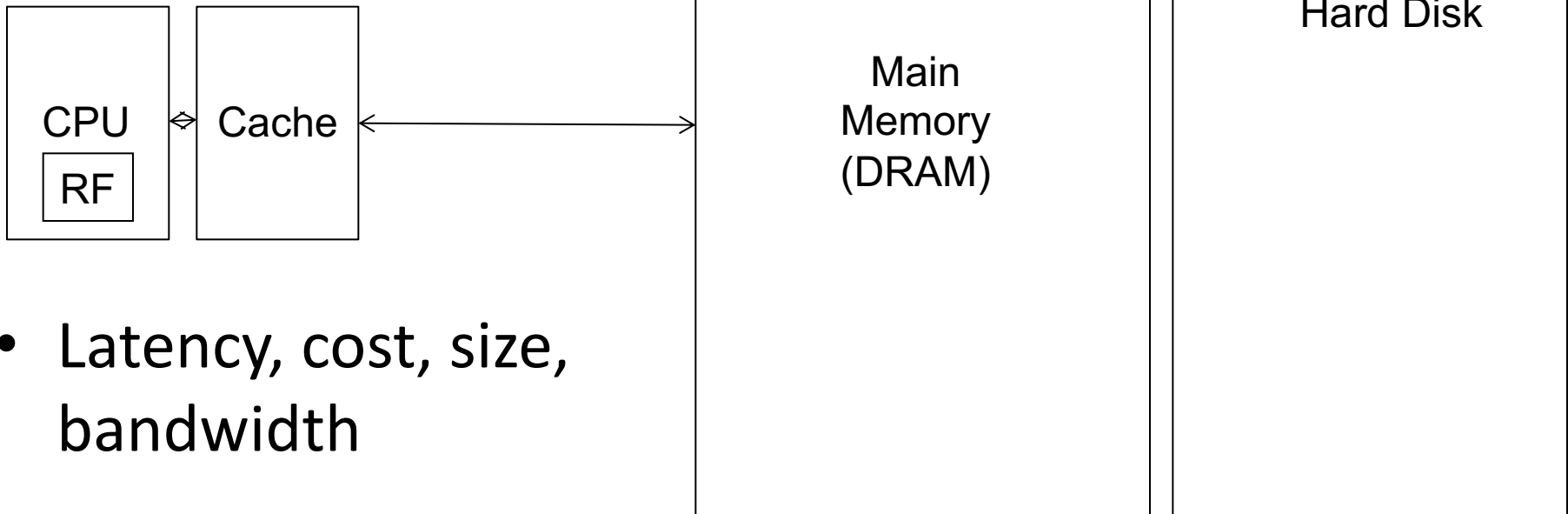
- Latency, cost, size, bandwidth

Locality

- One's recent past is a very good predictor of his/her near future.

- Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly

- Spatial Locality: If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people

Memory Locality

- A "typical" program has a lot of locality in memory references
  - typical programs are composed of "loops"

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    1. instruction memory references
    2. array/data structure references

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon

- Temporal locality principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
    - "The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory."

Caching Basics: Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon

- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, "Structural aspects of the System/360 Model 85 II: the cache," IBM Systems Journal, 1968.
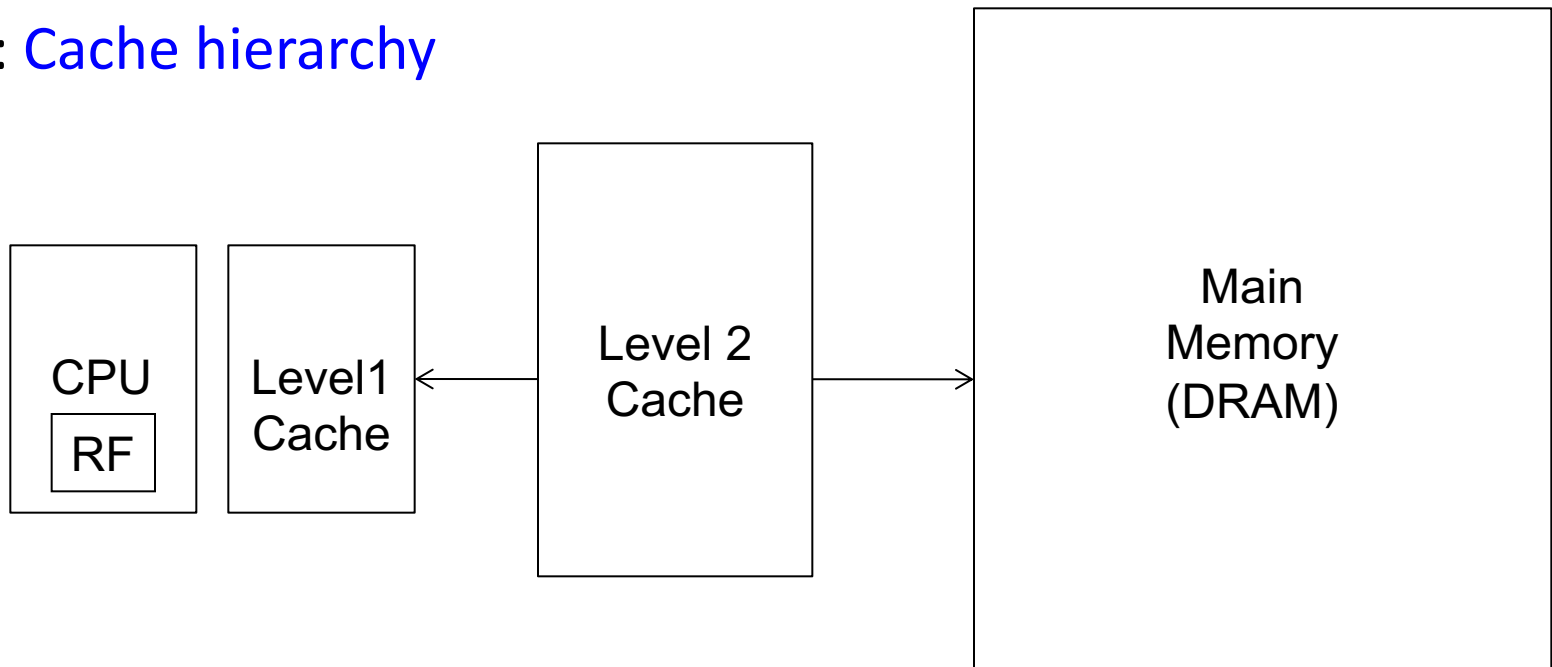
The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf

Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy

```
┌─────────┐  ┌────────┐      ┌──────────┐         ┌──────────────┐
│ CPU     │  │ Level1 │ ◄──  │ Level 2  │   ──►    │ Main         │
│ ┌─────┐ │  │ Cache  │      │ Cache    │          │ Memory       │
│ │ RF  │ │  │        │      │          │          │ (DRAM)       │
│ └─────┘ │  │        │      │          │          │              │
└─────────┘  └────────┘      └──────────┘         └──────────────┘
```

A Note on Manual vs. Automatic Management

- Manual: Programmer manages data movement across levels

  -- too painful for programmers on substantial programs

  – "core" vs "drum" memory in the 50's

  – still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache) and GPUs (called "shared memory")

- Automatic: Hardware manages data movement across levels, transparently to the programmer

  ++ programmer's life is easier

  – the average programmer doesn't need to know about it

    - You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

A Modern Memory Hierarchy

| Register File 32 words, sub-nsec | manual/compiler register spilling |

Memory Abstraction

Register File
32 words, sub-nsec

L1 cache
~32 KB, ~nsec

manual/compiler
register spilling

L2 cache
512 KB ~ 1MB, many nsec

Automatic
HW cache
management

L3 cache,
.....

Main memory (DRAM),
GB, ~100 nsec

automatic
demand
paging

Swap Disk
100 GB, ~10 msec
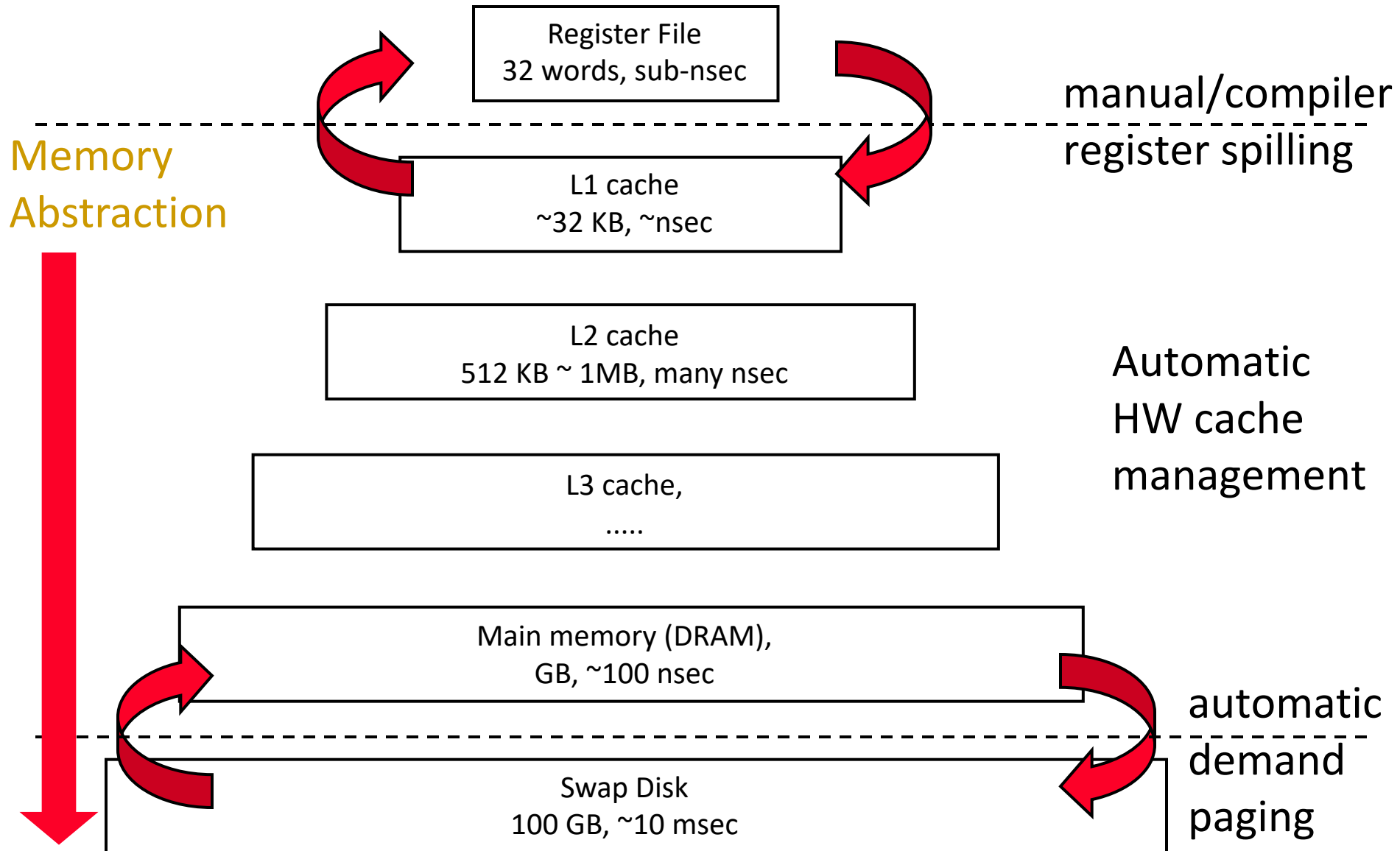
Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i, The perceived access time T_i is longer than t_i
- Except for the outer-most 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

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

h_i and m_i are defined to be the hit-rate
and miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Considerations

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired T_1 within allowed cost
- $T_i \approx t_i$ is desirable
- Keep m_i low
  - increasing capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)
- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Intel Pentium 4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16K
  - t_1 = 4 cyc int / 9 cycle fp
- L2 D-cache
  - C_2 =1024 KB
  - t_2 = 18 cyc int / 18 cyc fp
- Main memory
  - t_3 = ~ 50ns or 180 cyc
- Notice
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

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
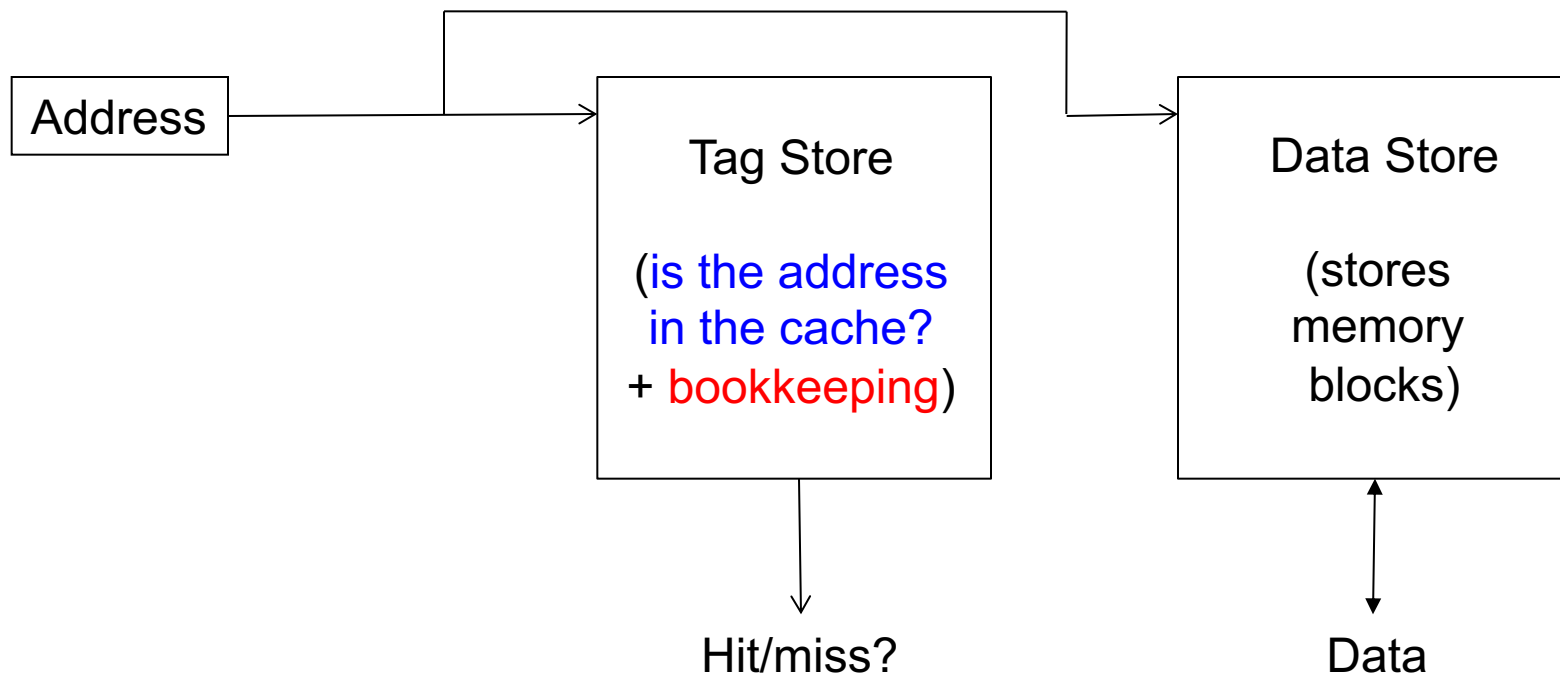 T_1=5.08, T_2=108

Cache Basics and Operation

Cache

- Generically, any structure that "memoizes" frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

- Most commonly in the on-die context: an automatically-managed memory hierarchy based on SRAM
  - memoize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency

Caching Basics

- Block (line): Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache

- On a reference:
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it

- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large or small blocks? Subblocks?
  - Write policy: what do we do about writes?
 - Instructions/data: do we treat them separately?

Cache Abstraction and Metrics

| Address | | Tag Store

(is the address
in the cache?
+ bookkeeping) | | Data Store

(stores
memory
blocks) |

Hit/miss?   Data

- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Aside: *Can reducing AMAT reduce performance?*

A Basic Hardware Cache Design

- We will start with a basic hardware cache design

- Then, we will examine a multitude of ideas to make it better
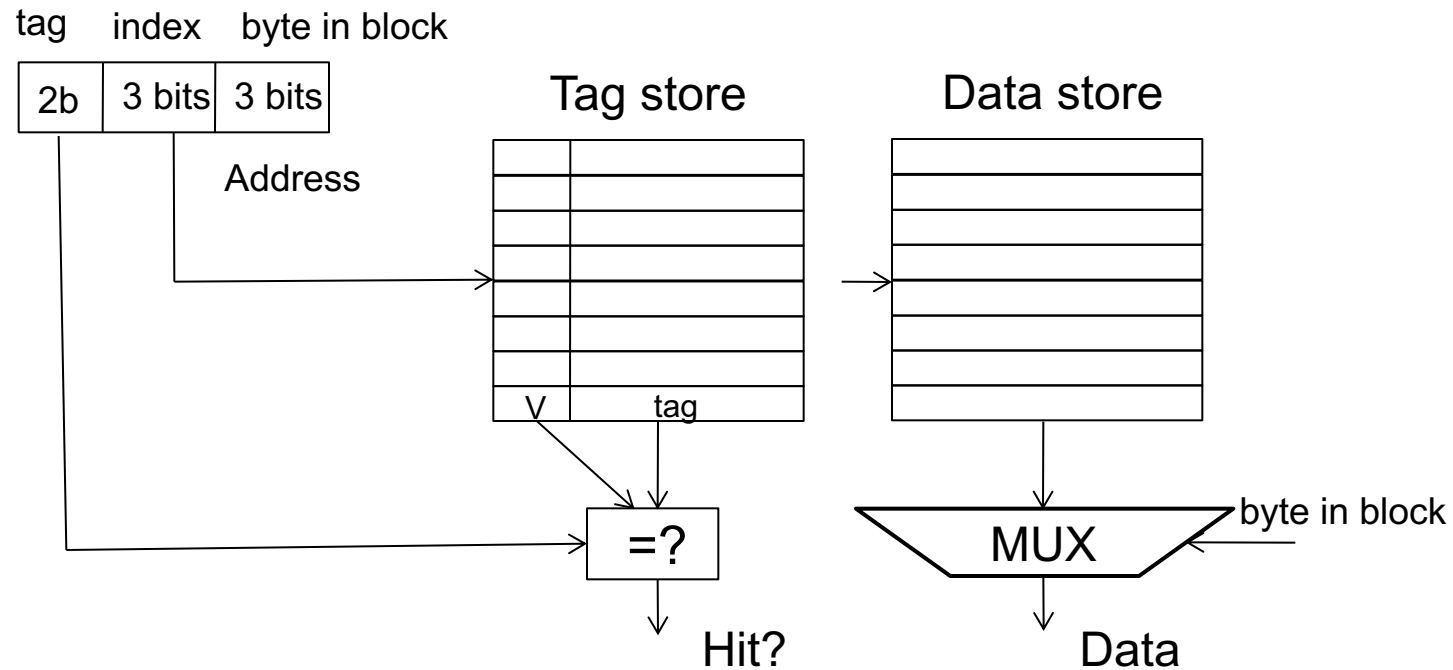
Blocks and Addressing the Cache

■ Memory is logically divided into fixed-size blocks

■ Each block maps to a location in the cache, determined by the index bits in the address

  ❏ used to index into the tag and data stores

tag	index	byte in block
2b	3 bits	3 bits

  8-bit address

■ Cache access:

  1) index into the tag and data stores with index bits in address
  2) check valid bit in tag store
  3) compare tag bits in address with the stored tag in tag store

■ If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

39
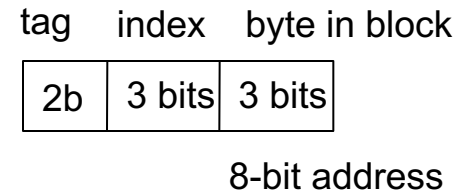
Direct-Mapped Cache: Placement and Access
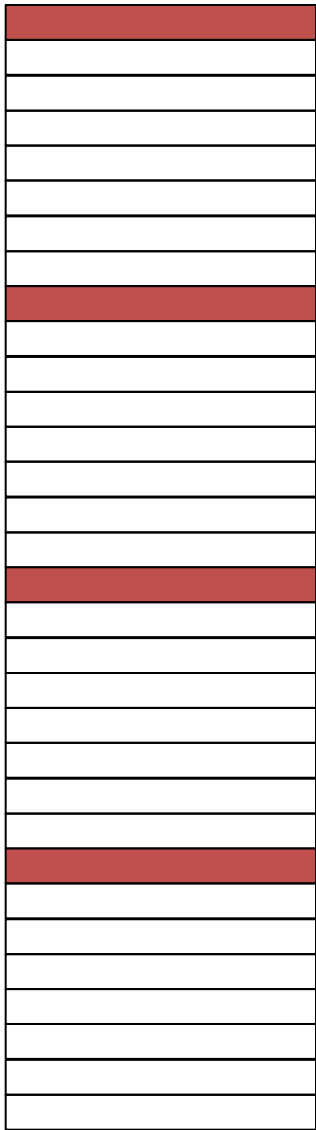
- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks

- Assume cache: 64 bytes, 8 blocks
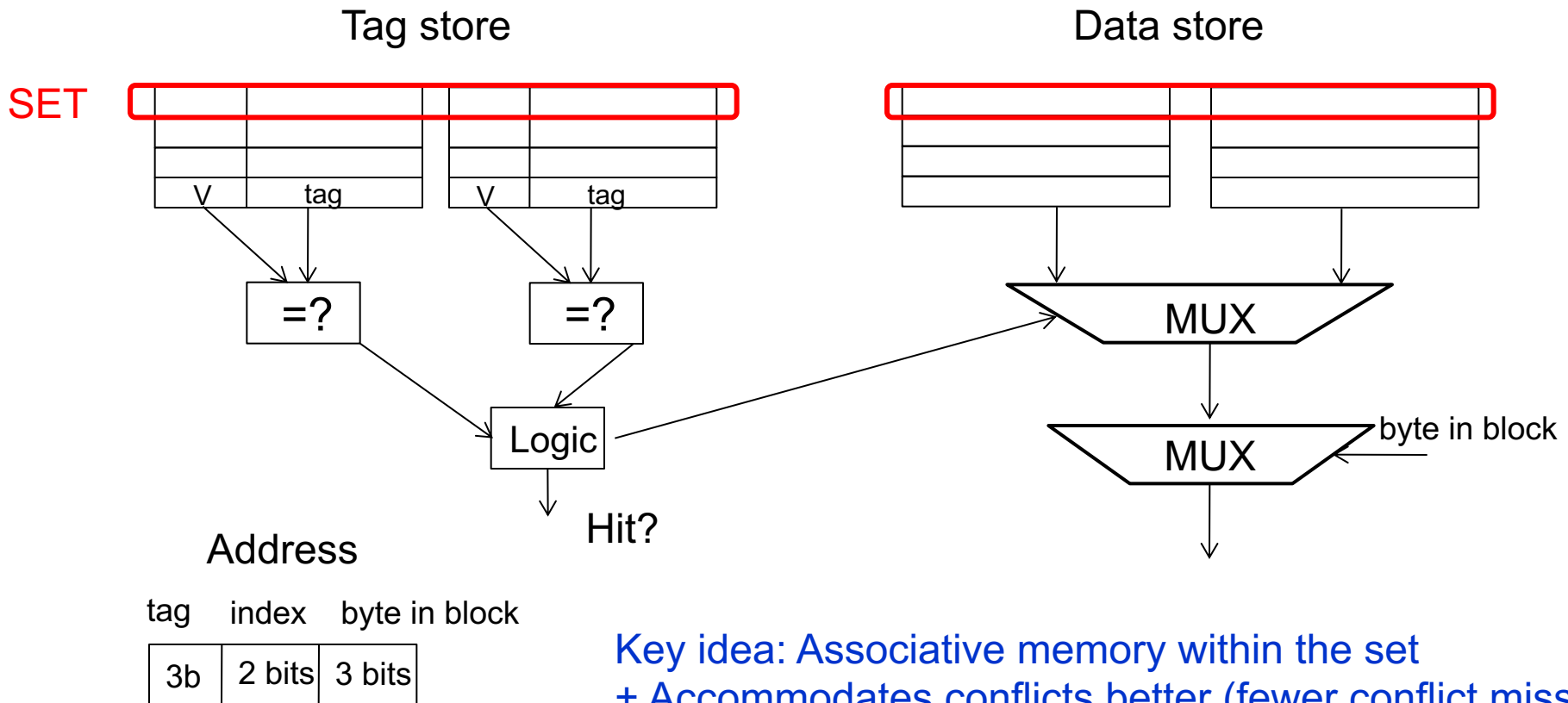 – Direct-mapped: A block can go to only one location

tag   index   byte in block

| 2b | 3 bits | 3 bits |

Address

Tag store   Data store

V   tag

=?   MUX ← byte in block

Hit?   Data

  – Addresses with same index contend for the same location: Cause conflict misses

40

Direct-Mapped Caches

- Direct-mapped cache: Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index → one entry

- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → conflict in the cache index
 - All accesses are conflict misses

Set Associativity

- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks

Tag store   Data store

SET

| V | tag | V | tag | |

=?   =?

Logic → Hit?

MUX

MUX → byte in block

Address

tag	index	byte in block
3b	2 bits	3 bits

Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store

42

Higher Associativity

- 4-way

Tag store

=? =? =? =?

Logic → Hit?

Data store
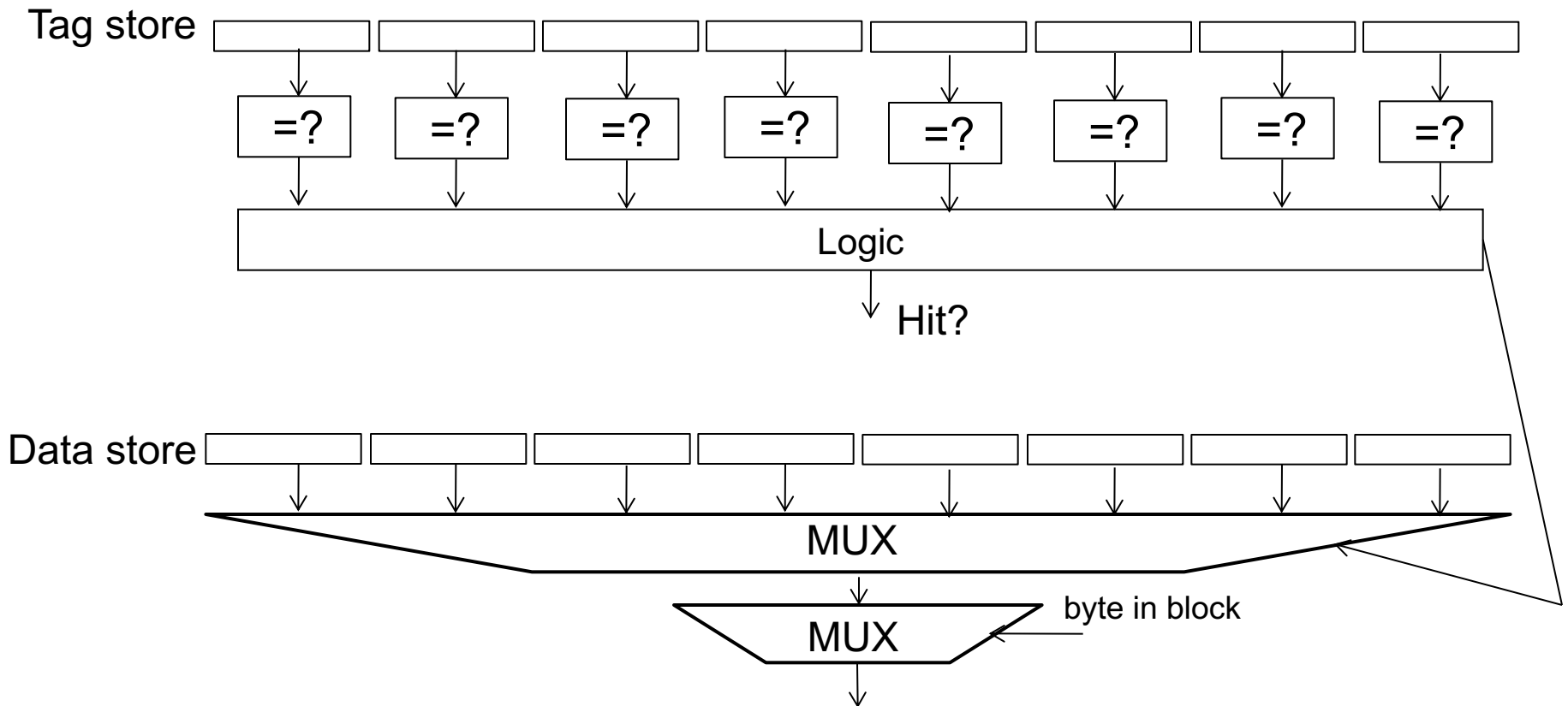
MUX

MUX ← byte in block
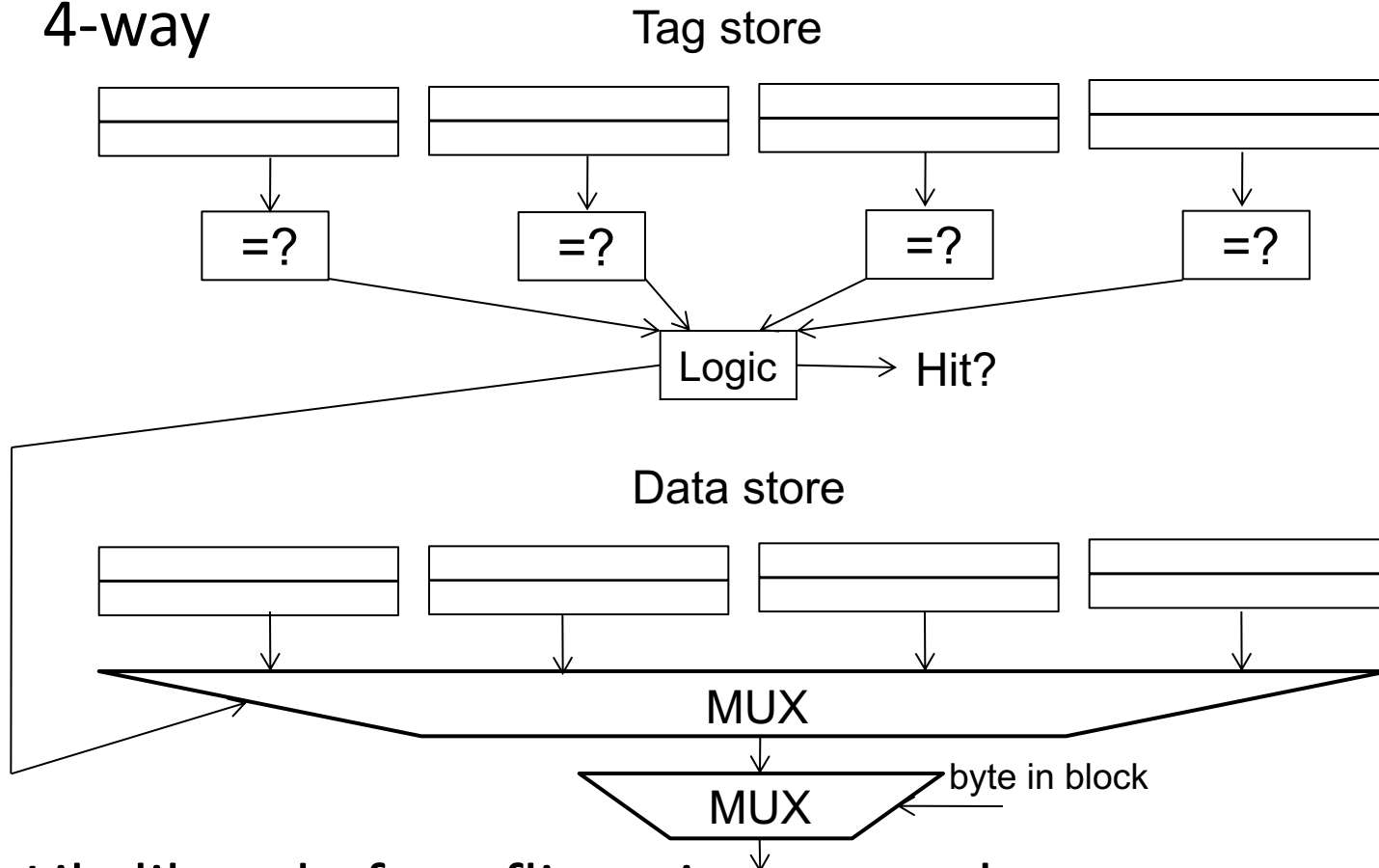
+ Likelihood of conflict misses even lower

-- More tag comparators and wider data mux; larger tags

43

Full Associativity

- Fully associative cache
  - A block can be placed in any cache location

Tag store

| =? | =? | =? | =? | =? | =? | =? | =? |

Logic

Hit?

Data store

MUX

MUX   byte in block

Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can map to the same index (or set)?

- Higher associativity

  ++ Higher hit rate

  -- Slower cache access time

  (hit latency and data access latency)

  -- More expensive hardware

  (more comparators)

hit rate

- Diminishing returns from higher associativity
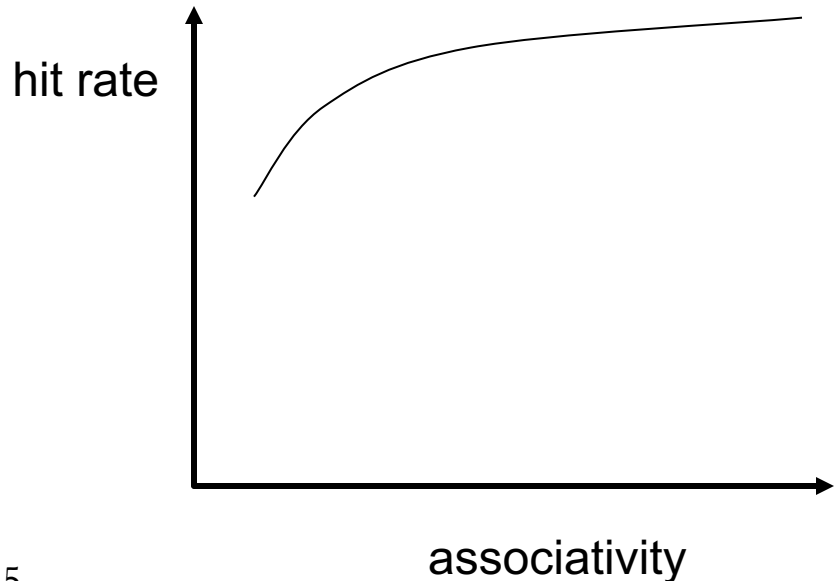
associativity

Issues in Set-Associative Caches

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)
- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

Eviction/Replacement Policy

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Least costly to re-fetch?
      - Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?
  - How many different orderings possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?

Approximations of LRU

- Most modern processors do not implement "true LRU" (also called "perfect LRU") in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)

- Examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the N-way set into M "groups", track the MRU group and the MRU way in each group
  - Victim-NextVictim Replacement: Only keep track of the victim and the next victim

Hierarchical LRU (not MRU)

- Divide a set into multiple groups

- Keep track of *only* the MRU group

- Keep track of *only* the MRU block in each group

- On replacement, select victim as:
  - A not-MRU block in one of the not-MRU groups (randomly pick one of such blocks/groups)

Cache Replacement Policy: LRU or Random

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar

- Best of both Worlds: Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

What Is the Optimal?

- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

What's In A Tag Store Entry?

- Valid bit

- Tag

- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches

Handling Writes (I)

- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the block is evicted

- Write-back
  + Can combine multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy
  -- Need a bit in the tag store indicating the block is "dirty/modified"

- Write-through
  + Simpler
  + All levels are up to date. Consistency: Simpler cache coherence because no need to check close-to-processor caches' tag stores for presence
  -- More bandwidth intensive; no combining of writes

Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss
  - \+ Can combine writes instead of writing each of them individually to next level
  - \+ Simpler because write misses can be treated the same way as read misses
  - \-\- Requires (?) transfer of the whole cache block

- No-allocate
  - \+ Conserves cache space if locality of writes is low (potentially better cache hit rate)

Handling Writes (III)

- What if the processor writes to an entire block over a small amount of time?

- Is there any need to bring the block into the cache from memory in the first place?

- Ditto for a *portion* of the block, i.e., subblock
 - E.g., 4 bytes out of 64 bytes

Cache Performance

Cache Parameters vs. Miss/Hit Rate

- Cache size

- Block size

- Associativity

- Replacement policy

- Insertion/Placement policy

Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often

- Working set: the whole set of data the executing application references
  - Within a time interval


59
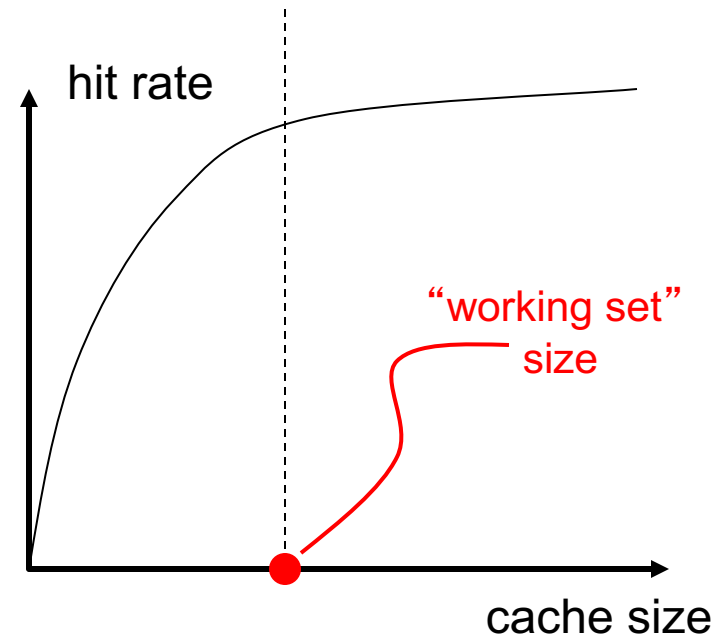
Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each with V bit)
      - Can improve "write" performance

- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total # of blocks → less
    temporal locality exploitation
  - waste of cache space and
  bandwidth/energy:
    if spatial locality is not high

hit rate

block size

60
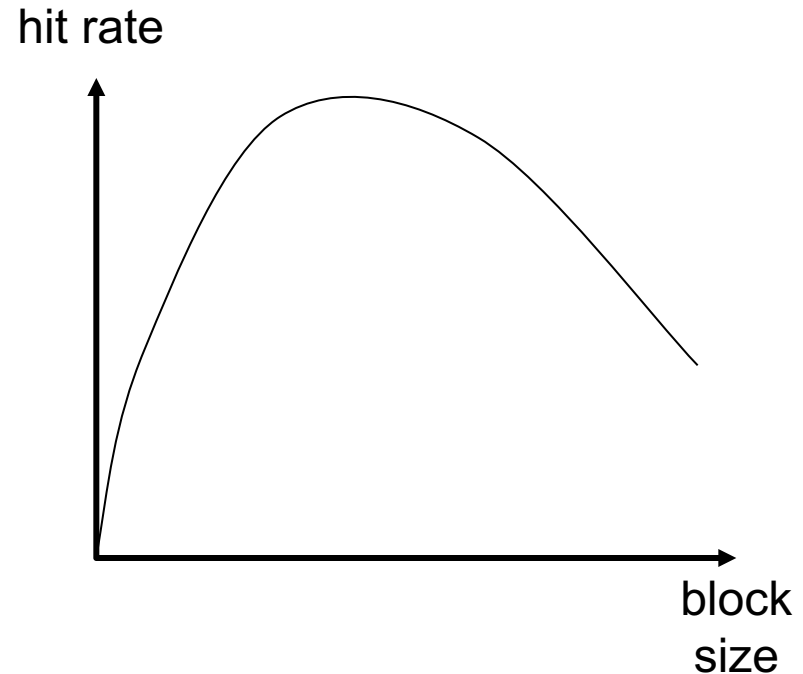
Paper Review #1: Summary

- Contents and merits are summarized well
- Main problems are in weaknesses and future work parts
- Unjustified claims or too high-level
  - "I don't believe ... is a realistic assumption"
  - "the paper should provide possible future research direction"
  - We can list more specific examples on what can be done differently/better

Paper Review #1: Grades Distribution
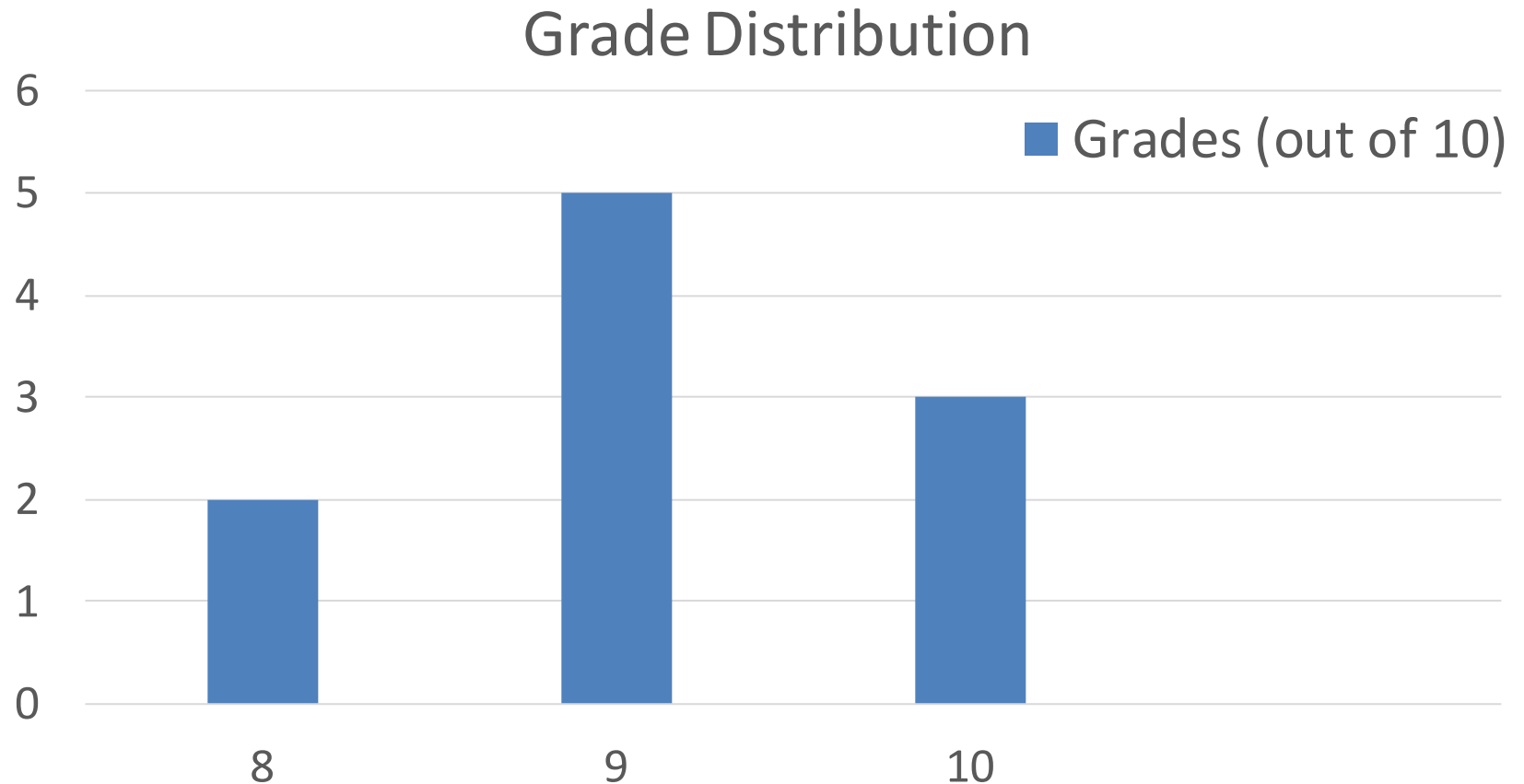
Review #3: Cache Compression

- Pekhimenko et al., "**Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches**," PACT 2012

CSC 2224: Parallel Computer Architecture and Programming Memory Hierarchy & Caches

Prof. Gennady Pekhimenko

University of Toronto

Fall 2019

The content of this lecture is adapted from the lectures of Onur Mutlu @ CMU and ETH