上海科技大学
ShanghaiTech University

CS211
Advanced Computer Architecture

L06 Memory I

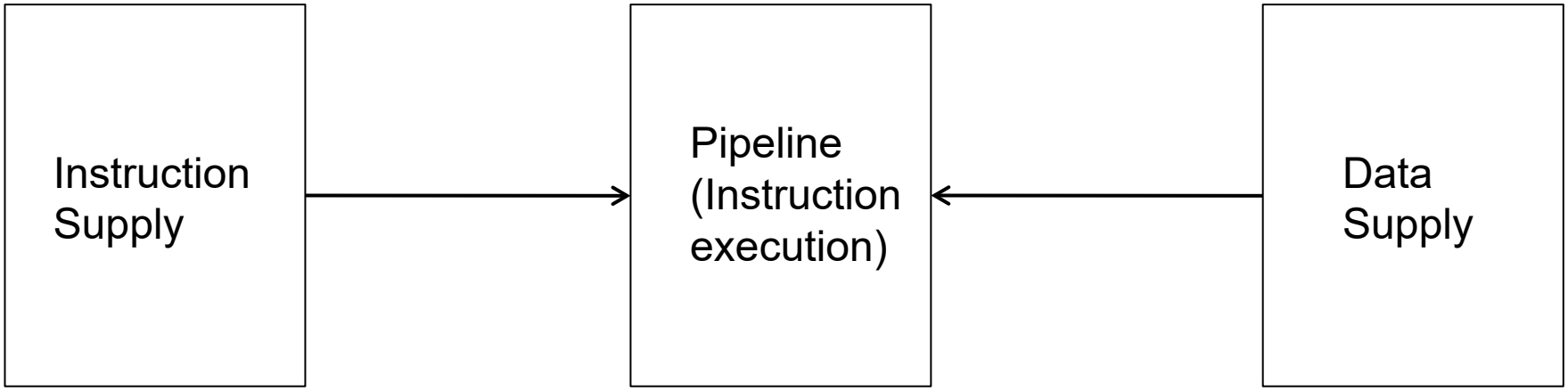
Chundong Wang

September 27th, 2020

Before 1st October

- Quiz 2
  - Today!
  - To collect a slip of questions from TA in the break
  - To submit it to TA at the end of today's class
- HW 1
  - Due at 23:59:59 (UTC+8), Monday, 28th September, 2020
  - Submitted to **Gradescope**
- Paper reading 1
  - Due at 23:59:59 (UTC+8), Monday, 12th October, 2020
- Tomorrow
  - Paper reading 3, HW 2, and Lab 1

A perfect world

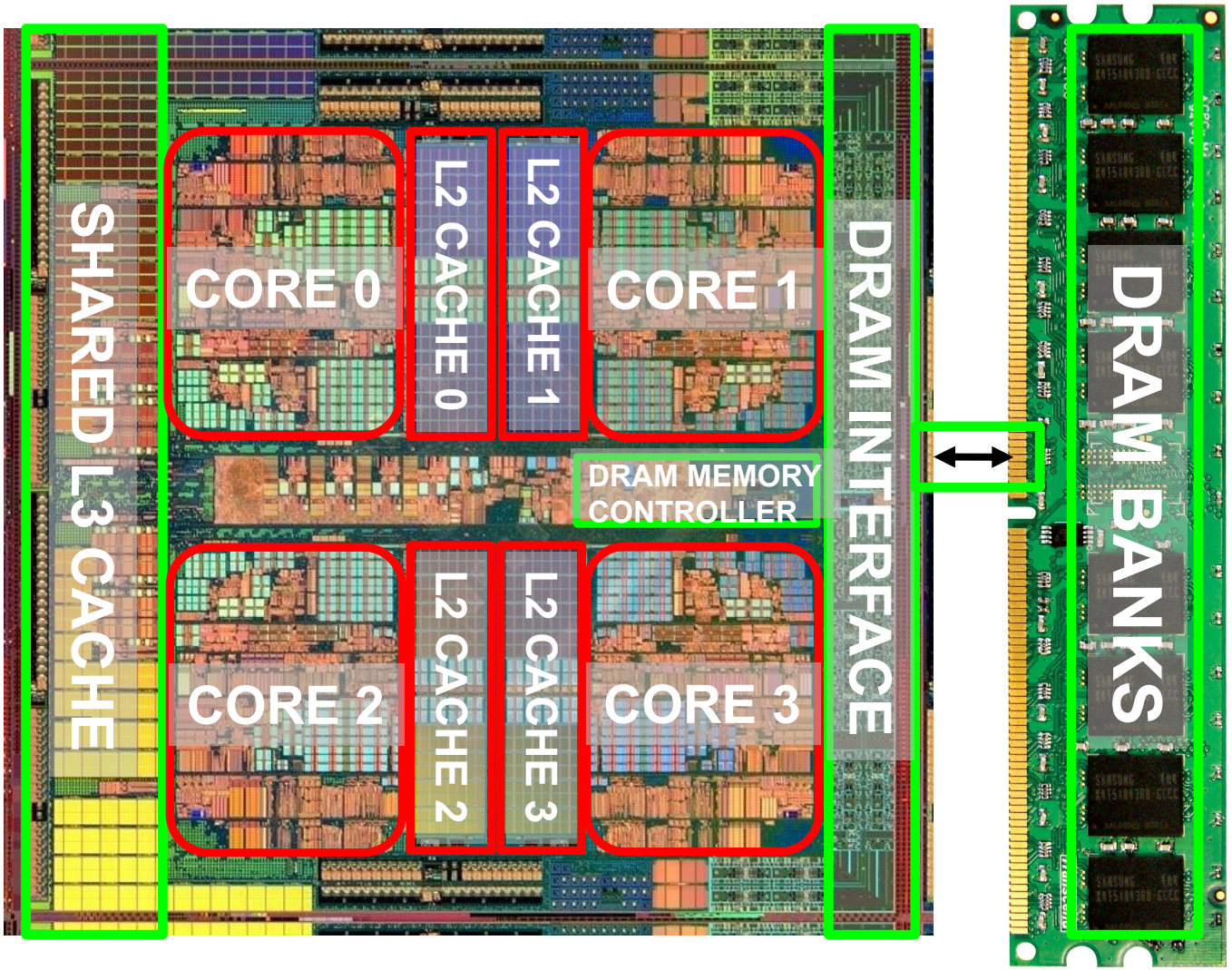


- Zero-cycle latency
- Infinite capacity
- Zero cost
- Perfect control flow

- No pipeline stalls
- Perfect data flow (reg/memory dependencies)
- Zero-cycle interconnect (operand communication)
- Enough functional units
- Zero latency compute

- Zero-cycle latency
- Infinite capacity
- Infinite bandwidth
- Zero cost

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
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

DRAM vs. SRAM

- DRAM
  - Slower access (capacitor)
  - Higher density (1 Transistor/1 Capacitor, 1T-1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together
- SRAM
  - Faster access (no capacitor)
  - Lower density (6-Transistor cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)

The Problem

- 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 scale with time
- Other technologies have their place as well
  - Flash memory, Intel Optane memory, STT-MRAM, ReRAM, etc.

Why Memory Hierarchy?

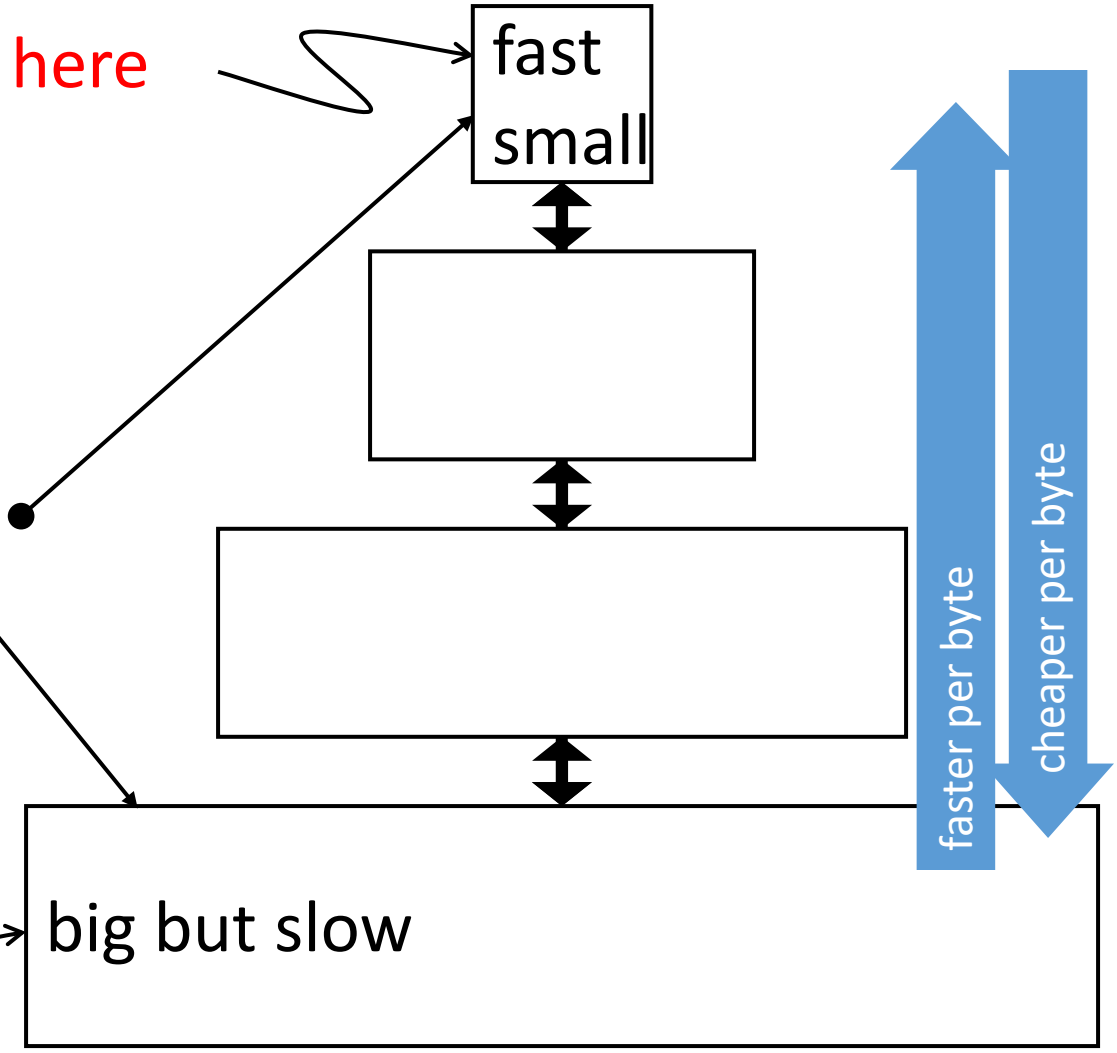
- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

The Memory Hierarchy

move what you use here

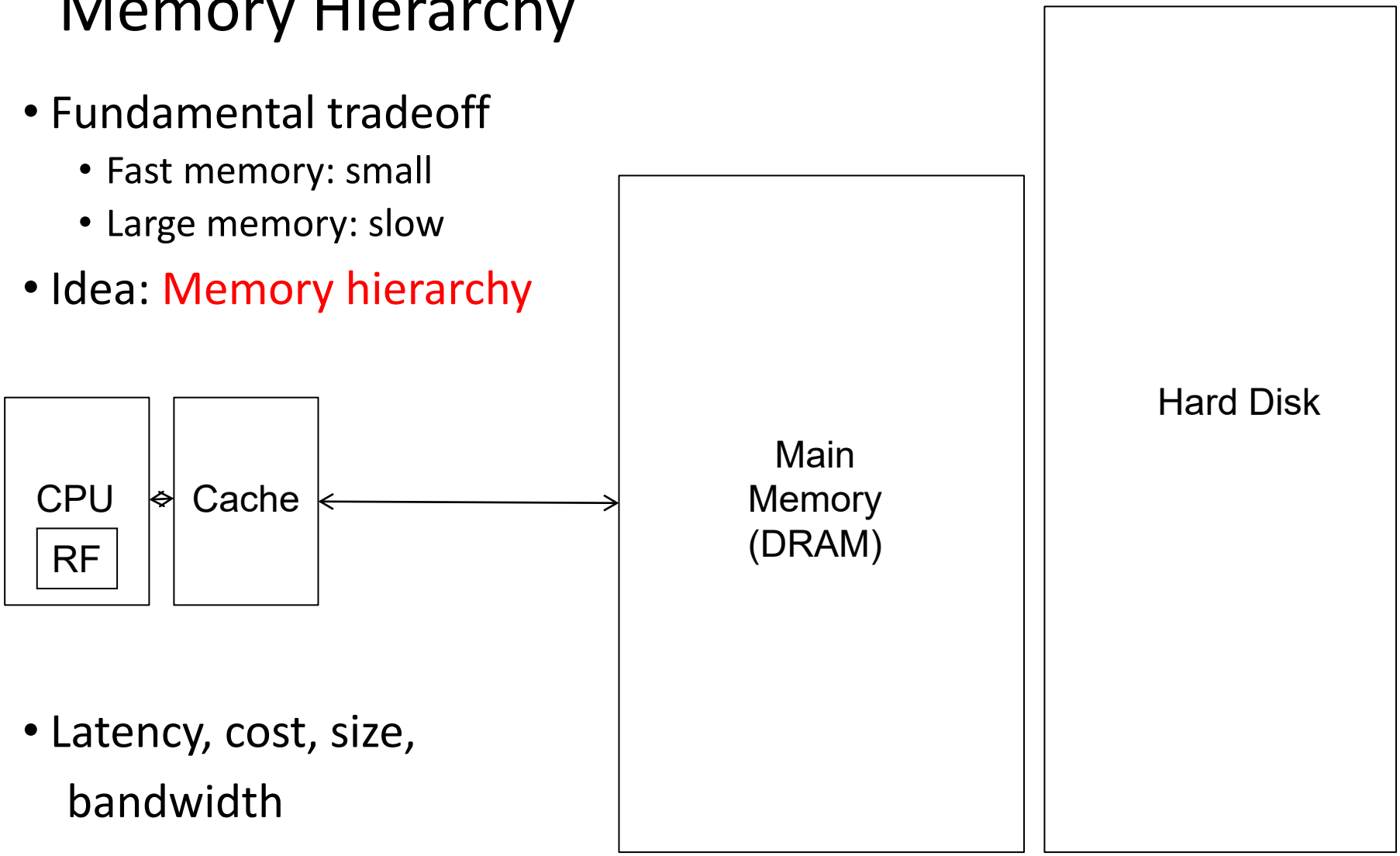
With good locality of reference, memory appears as fast as and as large as

backup everything here



Memory Hierarchy

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



- 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."

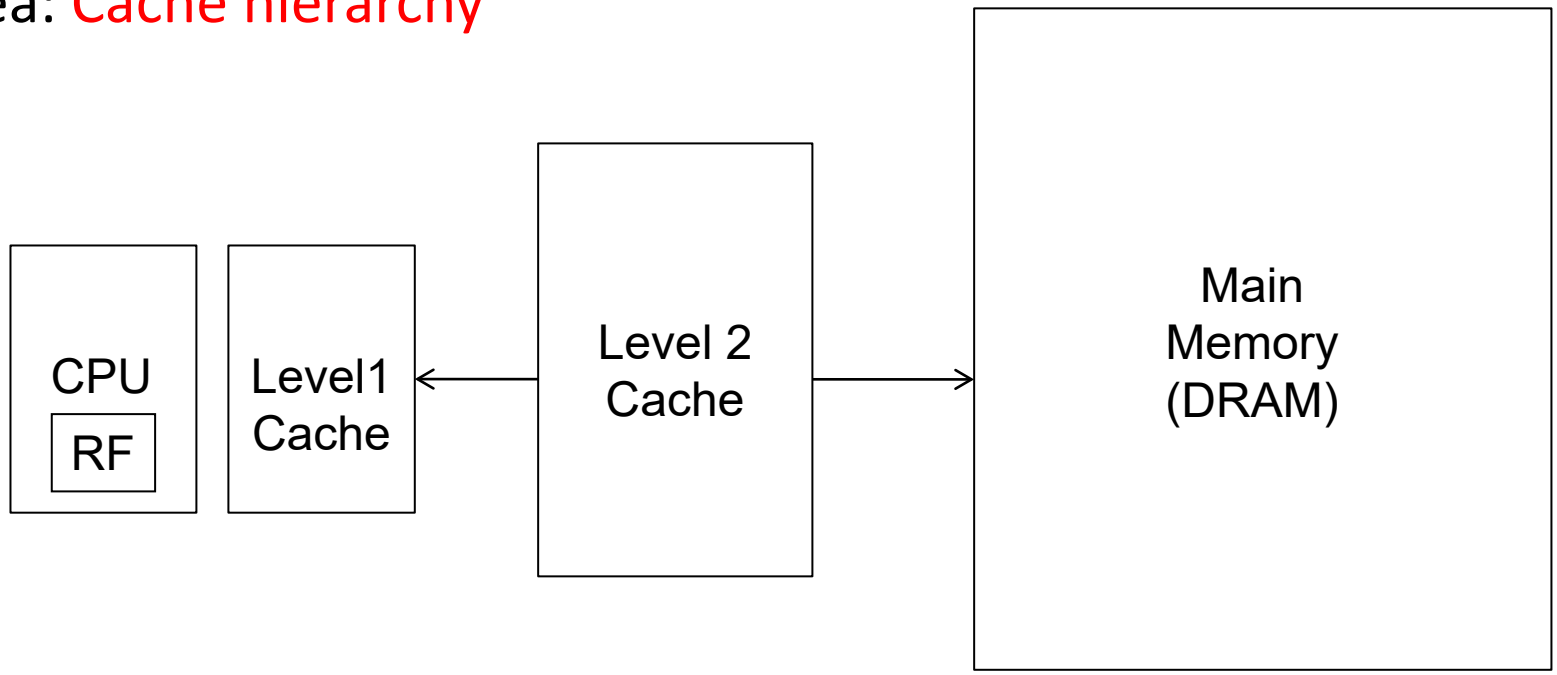
Caching Basics: Exploit Spatial Locality

- 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.

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

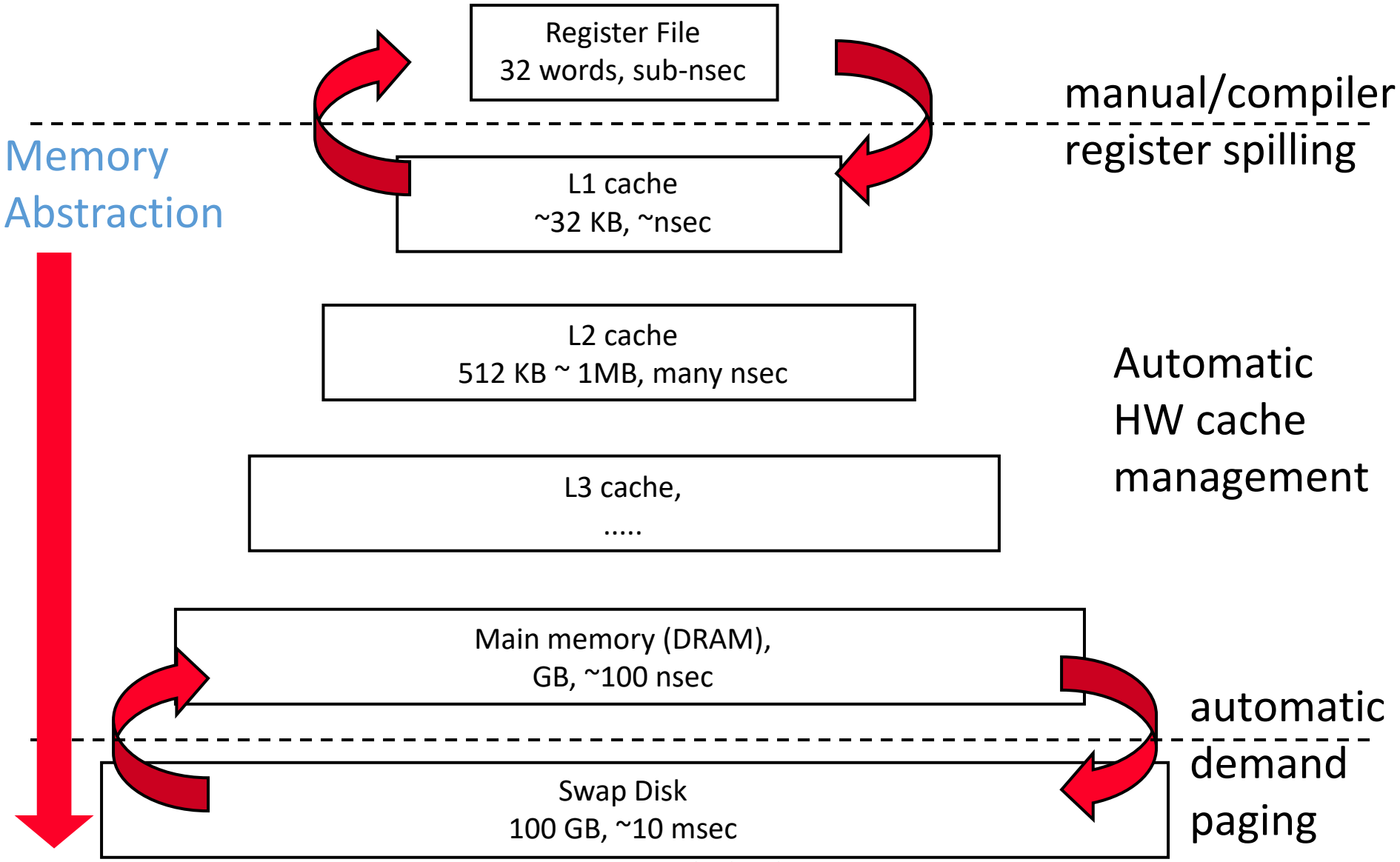


A Note on Manual vs. Automatic Management

- Manual: Programmer manages data movement across levels
  - -- too painful for programmers on substantial programs
  - Magnetic “core” vs “drum” memory in the 50’s
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)

- Automatic: Hardware manages data movement across levels, transparently to the programmer
  - ++ programmer’s life is easier
  - simple heuristic: keep most recently used items in cache
  - 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



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 $t_i + T_{i+1}$
  - $h_i + m_i = 100\%$
- Thus

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

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

keep in mind, h_i and m_i are defined to be the hit-rate and miss-rate, respectively, 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=**5.8**, 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

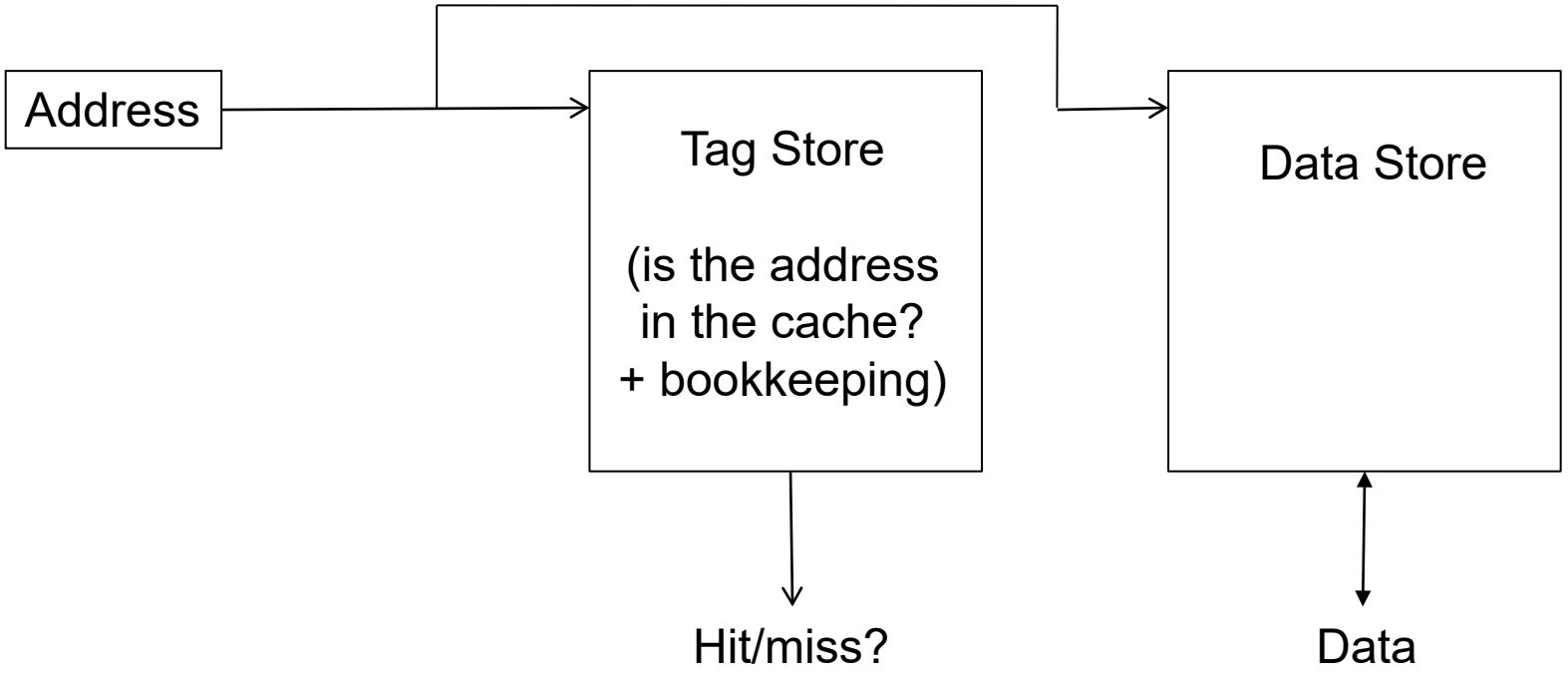
- Generically, any structure that “memoizes” frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g., a web cache

- Most commonly, 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
- When data referenced
  - 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, small, uniform blocks?
  - Write policy: what do we do about writes?
  - Instructions/data: Do we treat them separately?

Cache Abstraction and Metrics



- 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?*

Associativity in L02

9527 

Direct mapped

(Block address) % (# of blocks in cache)

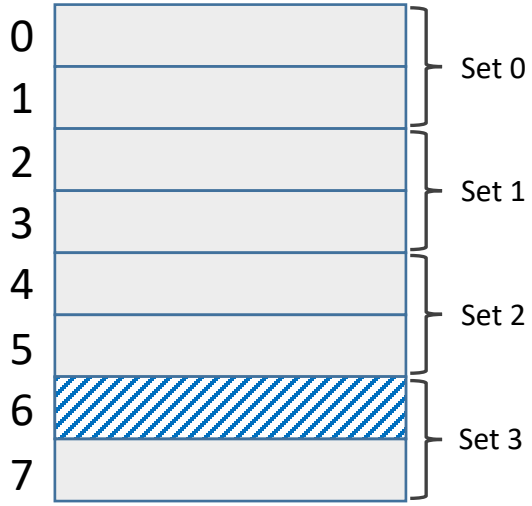
9527 % 8 = 7

Set associative

Cache partitioned into multiple sets
A block can be placed in anywhere in a set

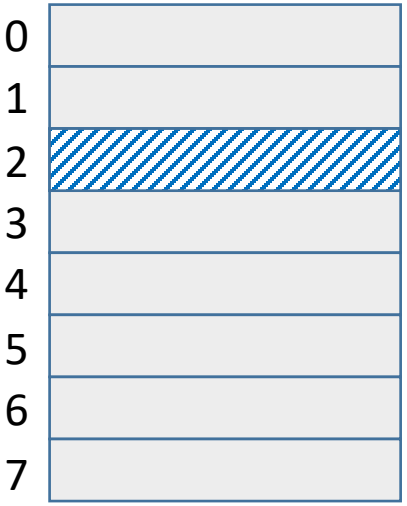
9527 % 4 = 3

Four sets, two ways



Fully associative

A block can be placed anywhere in the cache



A direct mapped cache can be viewed as a set associative cache with *N* sets and one way.
A fully associative cache can be viewed as set associative cache with one set and *N* ways.

Replacement

- How do we choose victim?
  - Verbs: Victimize, evict, replace, cast out
- Many policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used), pseudo-LRU
  - LFU (least frequently used)
  - NMRU (not most recently used)
  - NRU
  - Pseudo-random
  - Optimal
  - …

Optimal Replacement Policy?

- Evict block with longest reuse distance
  - i.e., next reference to block is farthest in future
  - Requires knowledge of the future
- Can't build it, but can model it with trace
  - Process trace in reverse
- Useful, since it reveals opportunity
  - (X,A,B,C,D,X): LRU 4-way set-associative cache, 2nd X will miss

Rabin A. Sugumar, Santosh G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization . In Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems, 1993.

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?

- Question: 4-way set associative cache:
  - 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

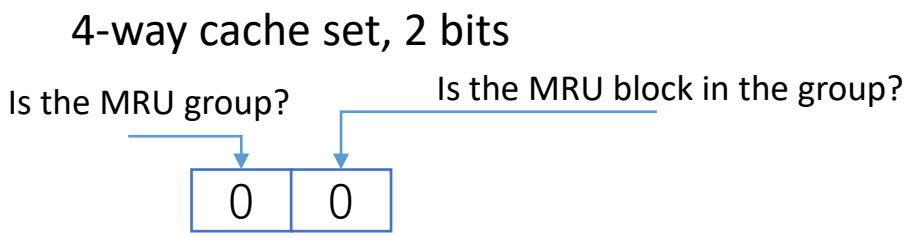
- Most modern processors do not implement "true LRU" in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible replacement policy)
  - Trashing: temporal reuse exists but LRU fails
    - Reuse distance > the number of ways in a set
- Examples:
  - Not MRU (not most recently used)
    - When the number of ways per set is 2, LRU is equivalent to NMRU
  - Hierarchical LRU: divide the 4-way set into 2-way "groups", track the MRU group and the MRU way in each group
  - NRU (Not Recently Used)

Hierarchical LRU (not MRU)

- Divide a set into multiple groups
- Keep track of the MRU group
- Keep track of the MRU block in each group

- On replacement, select victim as:
  - A not-MRU block in one of the not-MRU groups



Not Recently Used (NRU)

- Keep NRU state in 1 bit/block
  - Bit is set to 0 when installed (assume reuse)
  - Bit is set to 0 when referenced (reuse observed)
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0
    - Eviction forces all blocks in set to NRU=1
    - Picks one as victim (can be pseudo-random, or rotating, or fixed left-to-right)
  - Simple, similar to virtual memory clock algorithm
  - Provides some scan and thrash resistance
    - Relies on “randomizing” evictions rather than strict LRU order
  - Used by Intel Itanium, SPARC T2
- Variants of NRU
  - NRR: Not Recently Reused: Exploiting reuse locality!
  - RRIP: Re-reference Interval Prediction
    - [Jaleelet al. ISCA 2010]

Replacement Policy

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

Aside: Cache versus Page Replacement

- Physical memory (DRAM) is a cache for disk
  - Usually managed by system software via the virtual memory subsystem
- Page replacement is similar to cache replacement
- Page table is the "tag store" for physical memory data store
- What is the difference?
  - Hardware versus software
  - Number of blocks in a cache versus physical memory
  - "Tolerable" amount of time to find a replacement candidate

What's In a Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits

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

Handling Writes (Stores)

- 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 consolidate 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 "modified"

- Write-through

  + Simpler

  + All levels are up to date. Consistency: Simpler cache coherence because no need to check lower-level caches

  -- More bandwidth intensive; no coalescing of writes

Handling Writes (Stores)

- 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 consolidate 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)

Sectored Caches

- Divide a block into subblocks (or sectors)
  - Have separate valid and dirty bits for each sector
  - When is this useful? (Think writes…)
  - How many subblocks do you transfer on a read?

++ No need to transfer the entire cache block into the cache

(A write simply validates and updates a subblock)

++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)

-- More complex design

-- May not exploit spatial locality fully when used for reads

Instruction vs. Data Caches

- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split I and D caches)
  - -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  - -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?
- First level caches are almost always split
  - Mainly for the last reason above
- Second and higher levels are almost always unified

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
  - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially
- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter

Conclusion

- Memory hierarchy
- Cache
  - Basics
  - Replacement

Acknowledgements

- These slides contain materials developed and copyright by:
  - Prof. Onur Mutlu (ETHZ)
  - Prof. Mikko Lipasti (UW–Madison)
  - Prof. Krste Asanović (UC Berkeley)
  - Dr. Jorge Albericio (Cerebras Systems)