

CENG 5410
Advanced Computer Architecture
Lecture 10: Cache

Zhengrong Wang
CSE Department, CUHK
zhengrongwang@cuhk.edu.hk



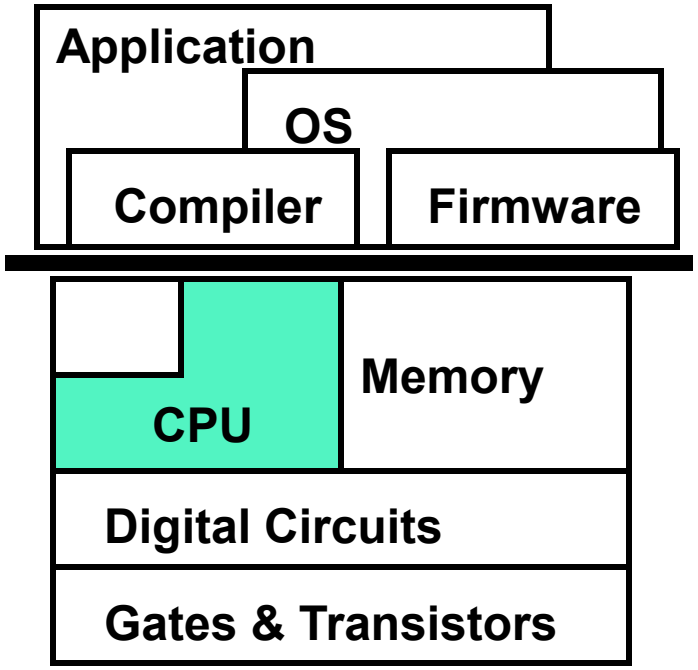
Simplifying Assumptions thus far

- Loads/Stores:
  - Fast if there's a cache hit
  - Slow, variable latency otherwise

- Addresses are "real" addresses:
  - Only one address space, and its physical
  - No need for translation
  - (real machines have virtual memory: each process pretends it has the whole address space)



This Unit: Caches



- Memory hierarchy concepts
- Cache organization
- High-performance techniques
- Low power techniques
- Some example calculations

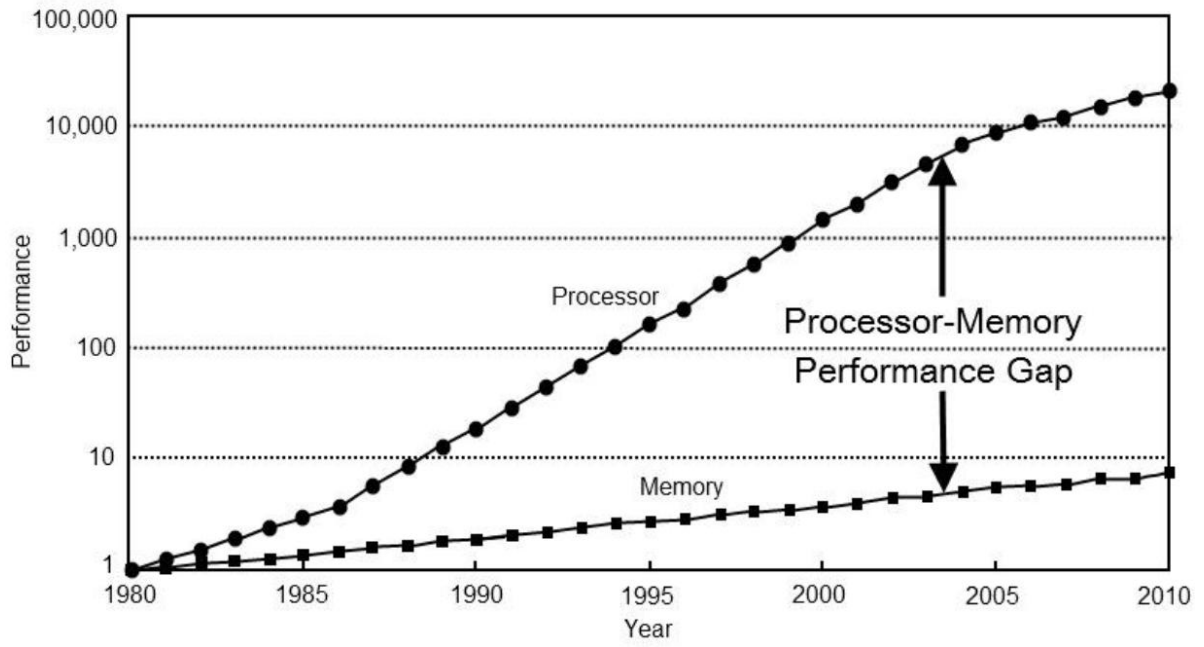


Motivation

- Processor can compute only as fast as memory
  - A 3Ghz processor can execute an "add" operation in 0.33ns
  - Today's "Main memory" latency is 50-100ns
  - Naïve implementation: loads/stores can be 300x slower than other operations
- Unobtainable goal:
  - Memory that operates at processor speeds
  - Memory as large as needed for all running programs
  - Memory that is cost effective
- Can't achieve all of these goals at once

The "Memory Wall"

- Processors are get faster more quickly than memory (note log scale)
- Year-by-year trends (Until Recently)
  - Processor speed improvement: 35% to 55%
  - Memory latency improvement: 7%



Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau



Types of Memory

- Static RAM (SRAM)
  - 6 transistors per bit
  - Optimized for speed (first) and density (second)
  - Fast (sub-nanosecond latencies for small SRAM)
    - Speed proportional to its area
  - Mixes well with standard processor logic
- Dynamic RAM (DRAM)
  - 1 transistor + 1 capacitor per bit
  - Optimized for density (in terms of cost per bit)
  - Slow (>40ns internal access, >100ns pin-to-pin)
  - Different fabrication steps (does not mix well with logic)
- Nonvolatile storage: Magnetic disk, Flash RAM



Storage Technology

- Cost - what can $300 buy today? (very approx.)
  - SRAM - 200MB (very approx)
  - DRAM - 200,000MB (200GB) --- 1000x cheaper than SRAM
  - Flash SSD- 6,000,000MB (6000GB) --- 30x cheaper than DRAM
  - Disk – 24,000,000MB (24TB) --- 4x cheaper than Flash
- Latency (random page read)
  - SRAM - <1ns (on chip)
  - DRAM - ~100ns --- 100x or more slower
  - Flash/NVM SSD - ~ 100,000ns – 1000x slower than dram
  - Disk - 10,000,000ns or 10ms --- 100x slower (mechanical)
- Bandwidth (sequential read)
  - SRAM – 10s-100s TB/sec (~106 TB/s H100 GPU)
  - DRAM – 10s-100s GB/sec (400GB/s in Xeon PHI, 3.2 GB/s H100)
  - Flash SSD – 500 MBps
  - Disk - 150MB/sec (0.15 GB/sec)



Known From the Beginning

"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, VonNeumann

"Preliminary discussion of the logical design of an electronic computing instrument"

IAS memo 1946



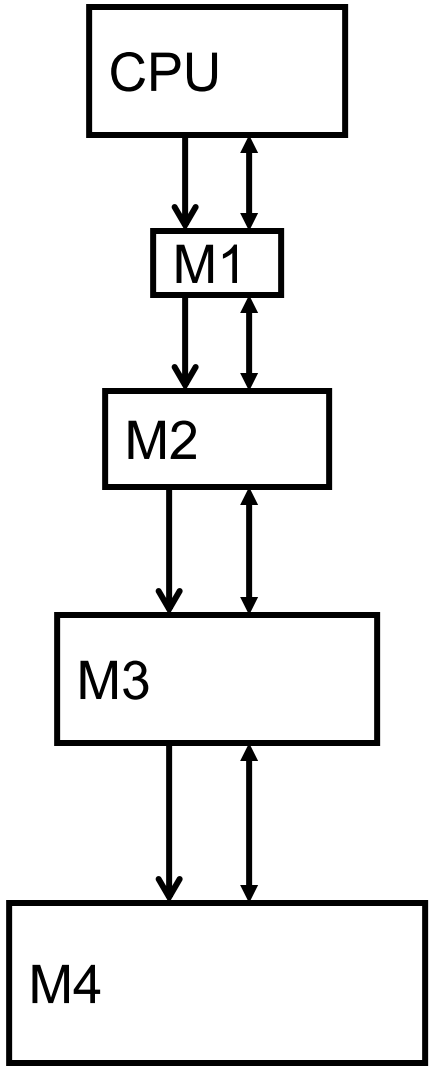
Locality to the Rescue

- Locality of memory references
  - Related data is accessed in a short time window
  - Property of real programs, few exceptions
- Temporal locality
  - Recently referenced data is likely to be referenced again soon
- Spatial locality
  - More likely to reference data "near" recently referenced data
- **Reactive**: reuse data that's been recently used
  - E.g. caches
- **Proactive**: fetch data that will be used in the near future
  - E.g. prefetch
- Holds for data and instructions



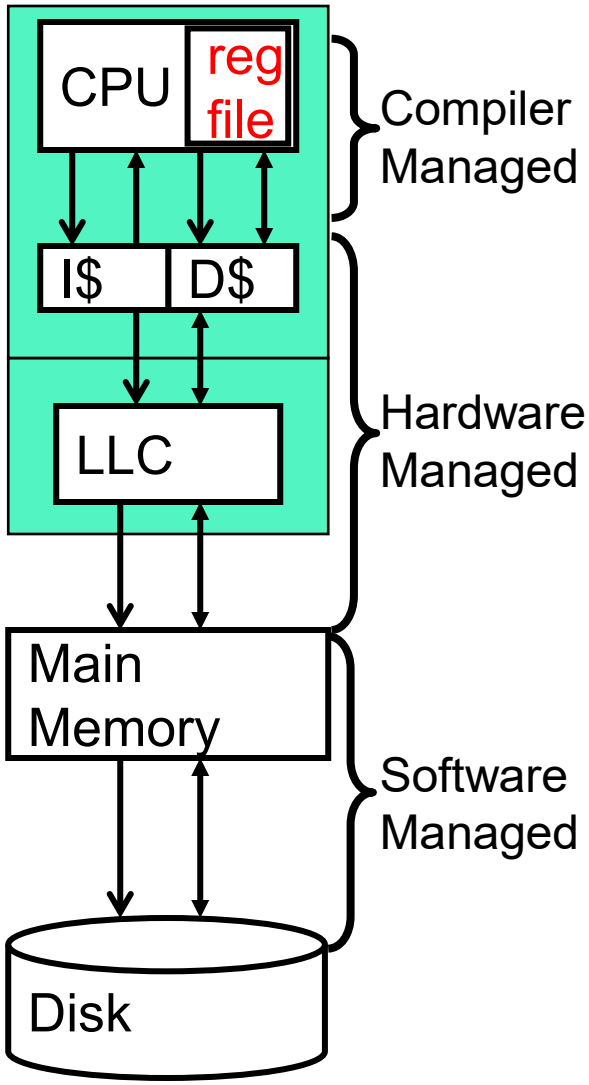
Exploiting Locality: Memory Hierarchy

- Hierarchy of memory components
  - Upper components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by network (bus/crossbar/etc.)
  - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - Move data up-down hierarchy
- Optimize average access time
  - $latency_{avg} = latency_{hit} + \%_{miss} * latency_{miss}$
  - Attack each component

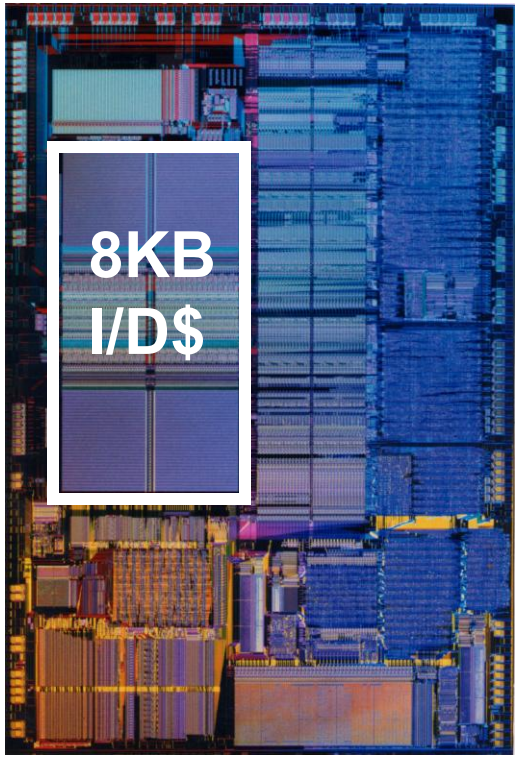


Concrete Memory Hierarchy

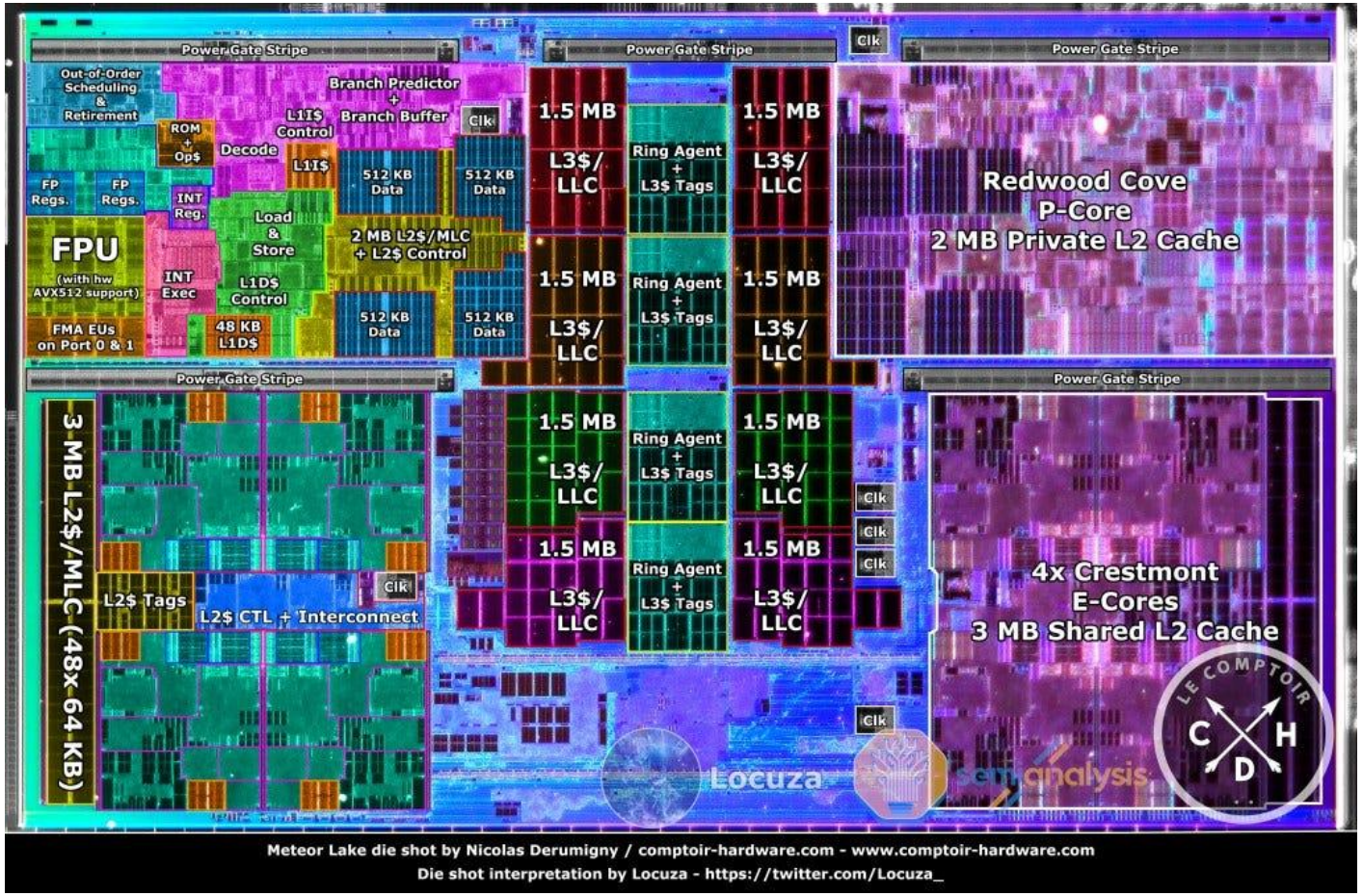
- 0th Level: **Registers**
- 1st level: **Private Caches**
  - Split instruction (I$) and data (D$) (8-64KB)
  - Secondary "L2" caches (128-1MB)
  - Made of SRAM (same circuit type as CPU)
- 2nd level: **Shared Cache** (LLC)
  - On-die, shared between cores (with CPU)
  - Typically 4-64MB Total (Epyc: 768MB Vcache)
- 3rd level: **main memory**
  - Made of DRAM
  - Typically GBs -> TBs (phone to server)
- 4th level: **disk (swap and files)**
  - SSDs or Magnetic Disks



Evolution of Cache Hierarchies



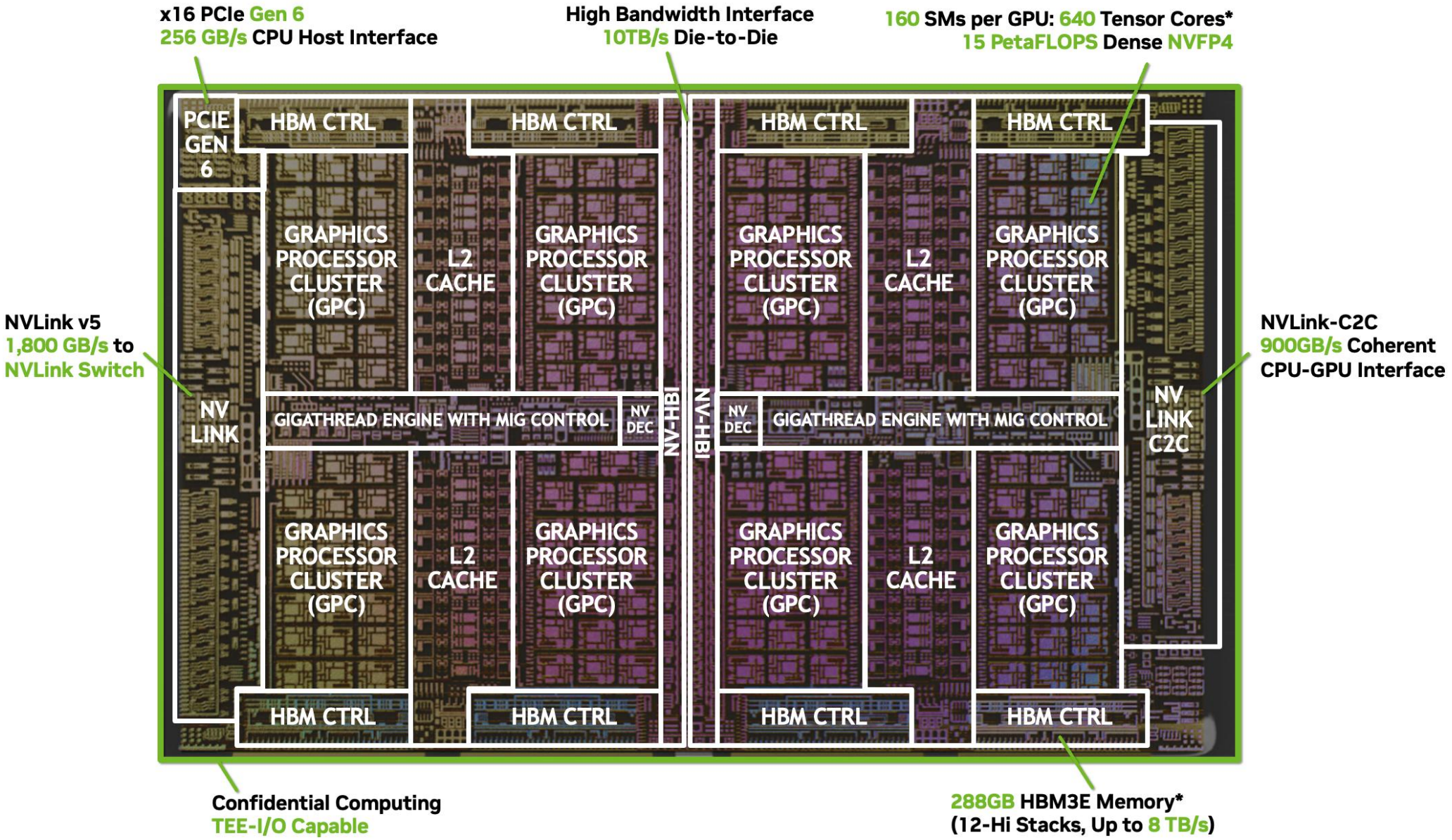
Intel 486 (1989)



Meteor Lake (2023)

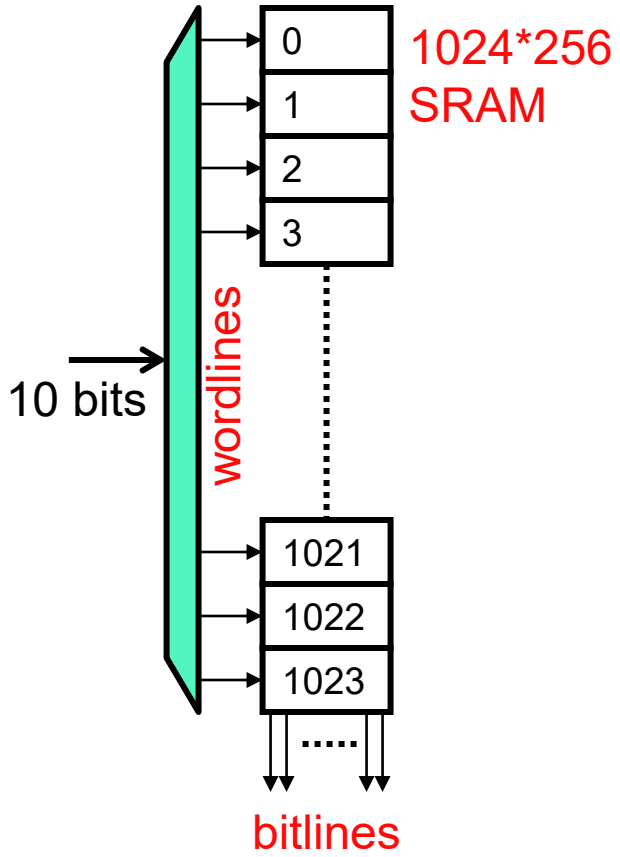
- Chips today are 30–70% cache by area (or reg-file if GPU)

NVIDIA Blackwell Ultra (B300)



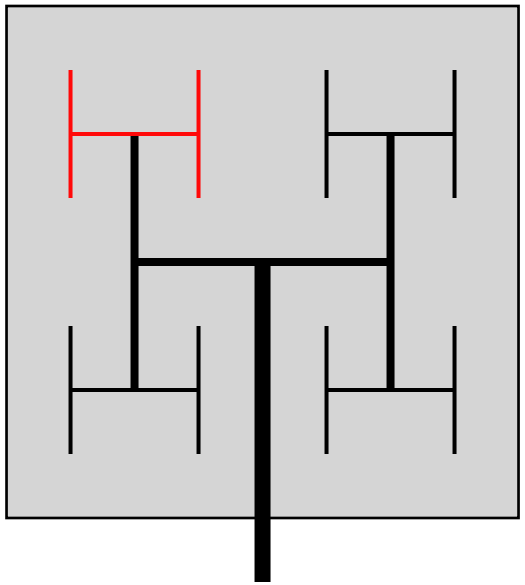
Circuit Primitive: Memory Array

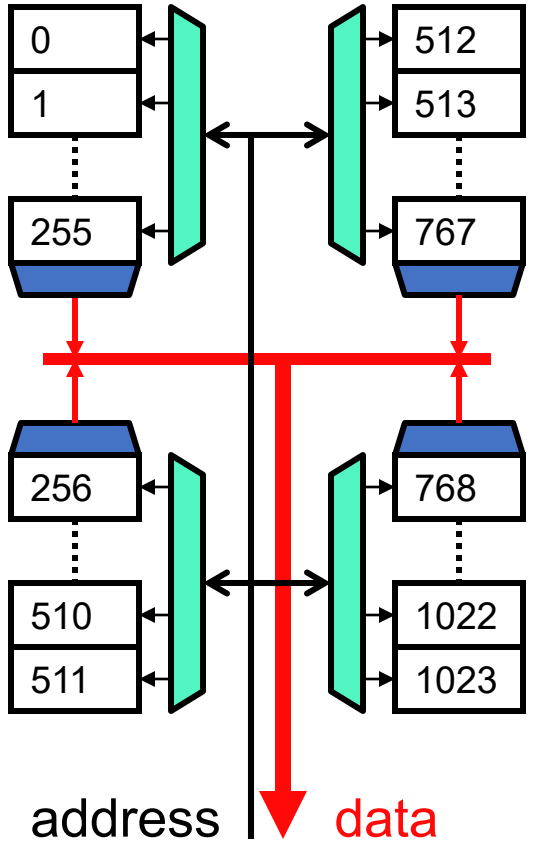
- Number of entries (number of wordlines)
  - 2^n, where n is number of address bits
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
  - One-bit address travels on "wordlines"
- Size of entries (number of bitlines)
  - Width of data accessed
  - Data travels on "bitlines"
  - 256 bits (32 bytes) in example



Physical Cache Layout

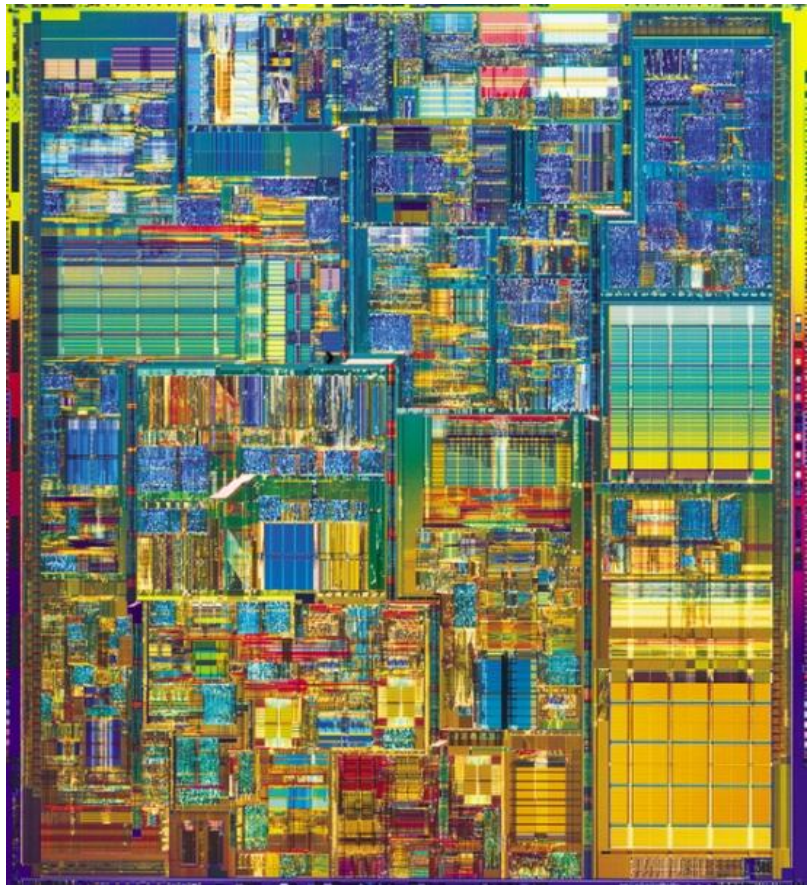
- Logical layout
  - Arrays are vertically contiguous
- Physical layout - roughly square
  - Vertical partitioning to minimize wire lengths
  - **H-tree**: horizontal/vertical partitioning layout
    - Applied recursively
    - Each node looks like an H





Physical Cache Layout

- Arrays and h-trees make caches easy to spot in μgraphs

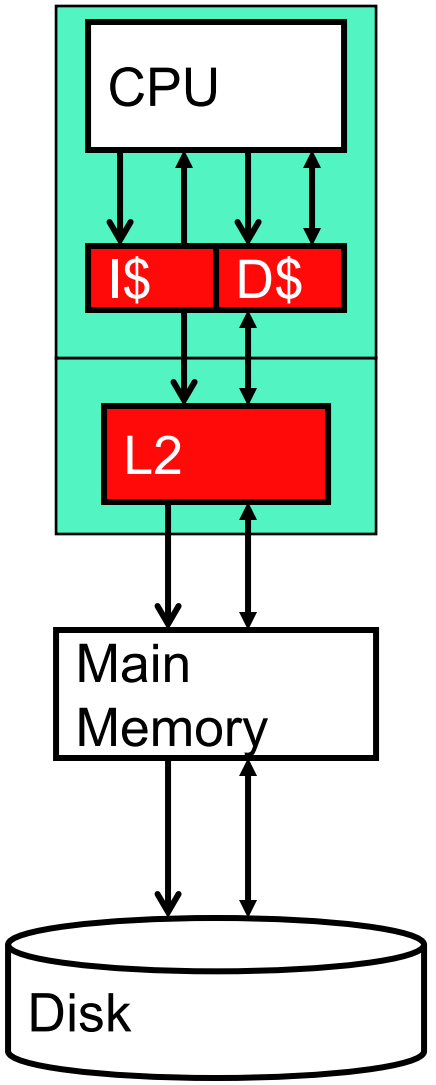


Pentium 4 ~2000



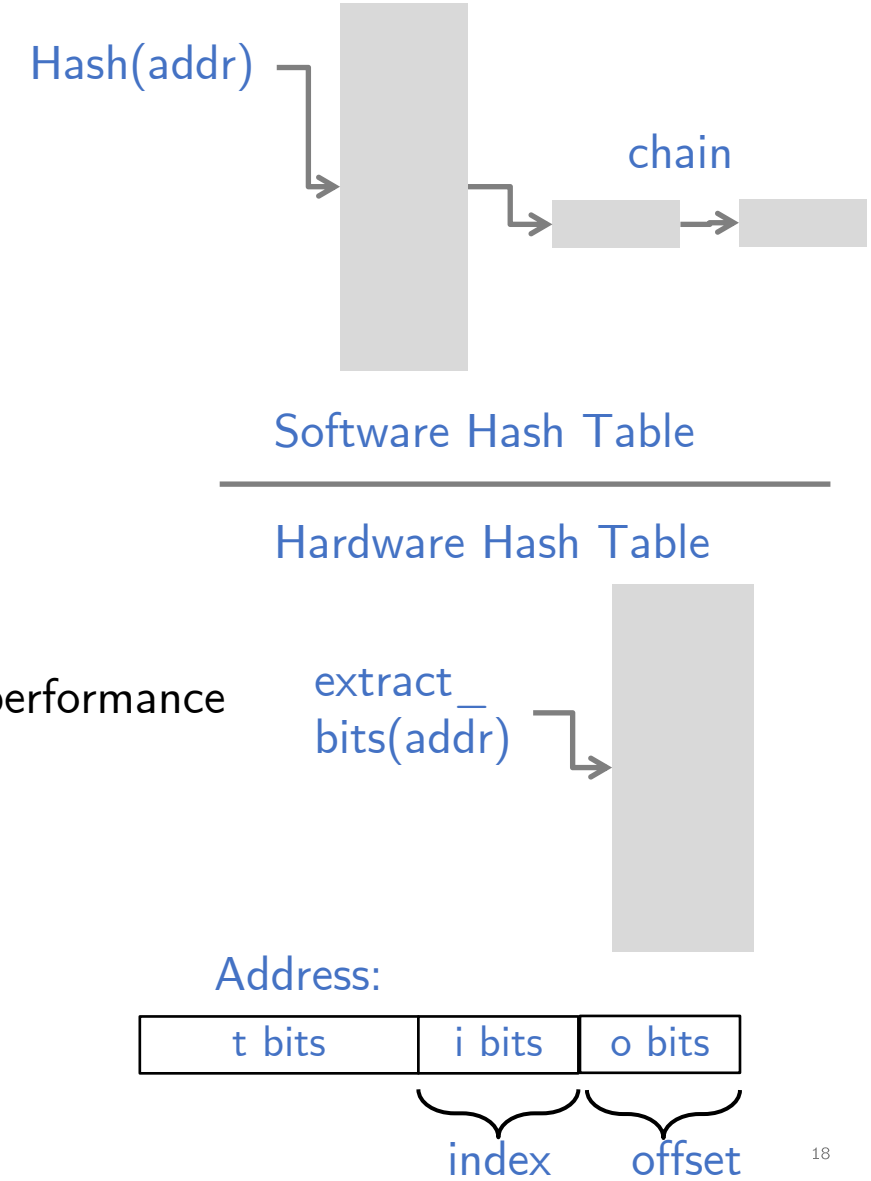
This Unit: Caches

- Cache organization
  - ABC
  - Miss classification
- High-performance techniques
  - Reducing misses
  - Improving miss penalty
  - Improving hit latency
- Low-power techniques
- Some example performance calculations



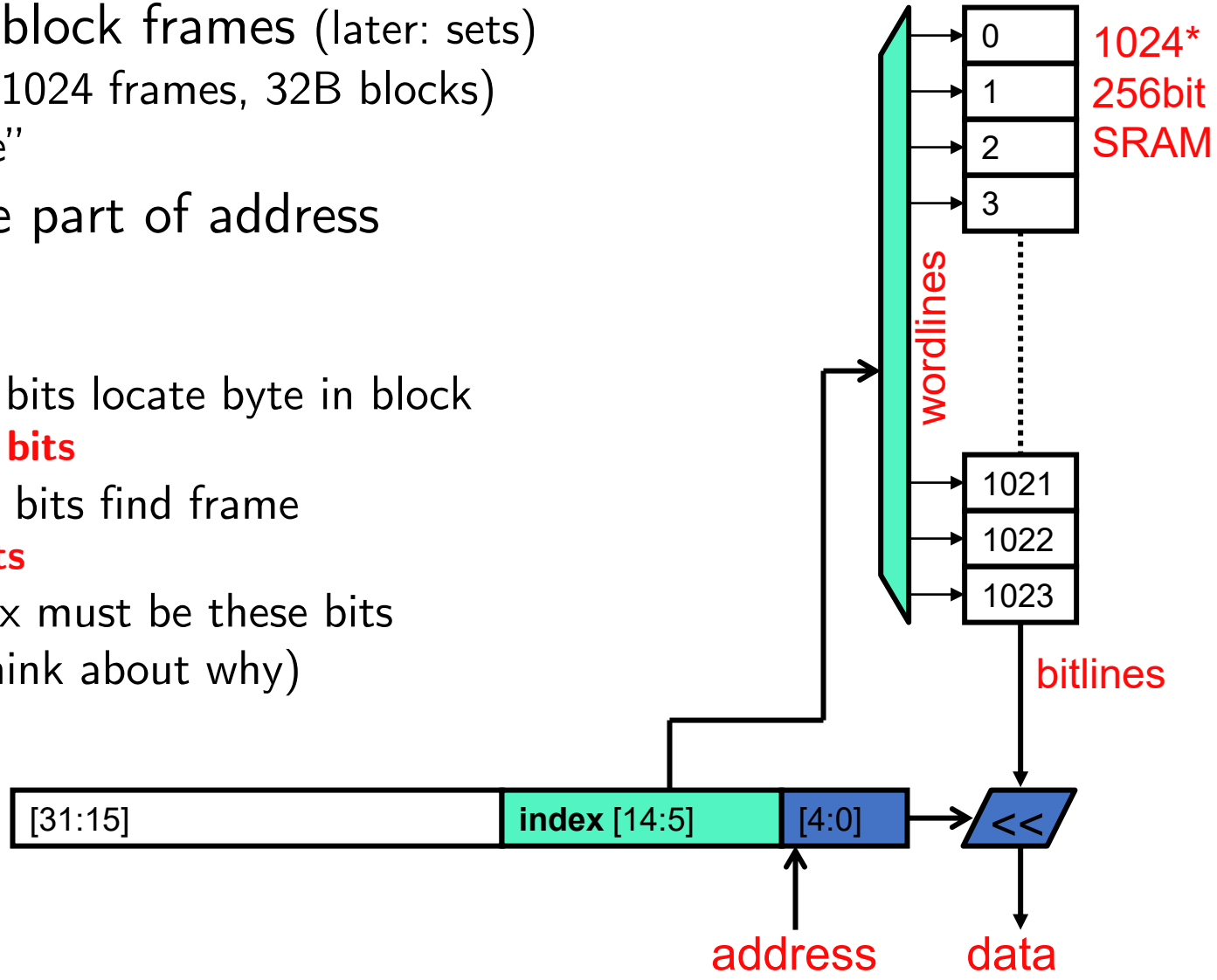
Borrow data structure from software?

- What about a hash table?
  - Hash address to find a data value
  - Each entry of the hash table stores (addr,data) pair
  - Constant time lookup and insert
- Problems:
  - Problem 1: need simple hash function
    - Just use some bits of the address
  - Problem 2: Accessing chain takes too long
    - Insight: we don't need to cache all elements
    - Just drop elements that don't fit .. maybe just lose a little performance
  - Problem 3: Storing arbitrary addr is costly
    - Amortize over Block of addresses



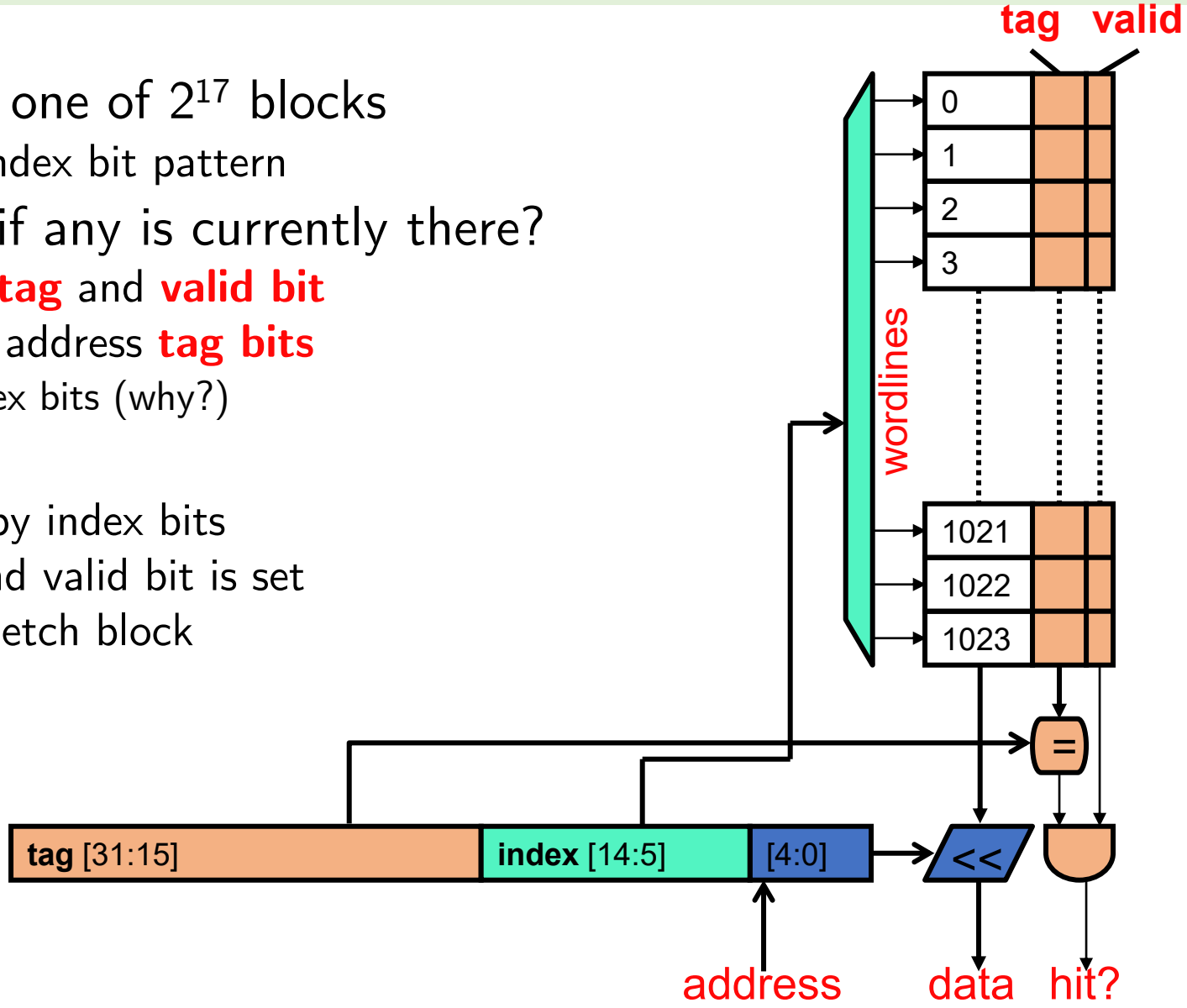
Basic Cache Structure

- Basic cache: array of block frames (later: sets)
  - Example: 32KB cache (1024 frames, 32B blocks)
  - "Hash table in hardware"
- To find frame: decode part of address
  - Which part?
  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block
    - These are called **offset bits**
  - 1024 frames → next 10 bits find frame
    - These are the **index bits**
  - Note: nothing says index must be these bits
  - But these work best (think about why)



Basic Cache Structure

- Each frame can hold one of 2^{17} blocks
  - All blocks with same index bit pattern
- How to know which if any is currently there?
  - To each frame attach **tag** and **valid bit**
  - Compare frame tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read frame indicated by index bits
  - "Hit" if tag matches and valid bit is set
  - Otherwise, a "miss". Fetch block





Calculating Tag Overhead

- "32KB cache" means cache holds 32KB of data
  - Called **capacity**
  - Tag storage is considered overhead
- Tag overhead of 32KB cache with 1024 32B frames
  - 32B frames → 5-bit offset
  - 1024 frames → 10-bit index
  - 32-bit address – 5-bit offset – 10-bit index = 17-bit tag
  - (17-bit tag + 1-bit valid)* 1024 frames = 18Kb tags = 2.2KB tags
  - ~6% overhead
- What about 64-bit addresses?
  - Tag increases to 49bits, ~20% overhead



Cache Performance Simulation

- Parameters: 8-bit addresses, 32B cache, 4B blocks
  - Nibble notation (base 4)
  - Initial contents: 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130



Cache contents (prior to access)	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	
	3030	
	2100	
	0012	
	0020	
	0030	
	0110	
	0100	
	2100	
	3020	



Conflicts

- What about pairs like 3030/0030, 0100/2100?
  - These will **conflict** in any sized cache (regardless of block size)
    - Will keep generating misses
- Can we allow pairs like these to simultaneously reside?
  - Yes, reorganize cache to do so



Cache contents (prior to access)	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	Miss
0000, 0010, 3020, 0030, 0100, 0110, 0120, 0130	**3030**	Miss
0000, 0010, 3020, **3030**, 0100, 0110, 0120, 0130	2100	Miss
0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130	0012	Hit
0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130	0020	Miss
0000, 0010, 0020, 3030, 2100, 0110, 0120, 0130	**0030**	Miss
0000, 0010, 0020, **0030**, 2100, 0110, 0120, 0130	0110	Hit

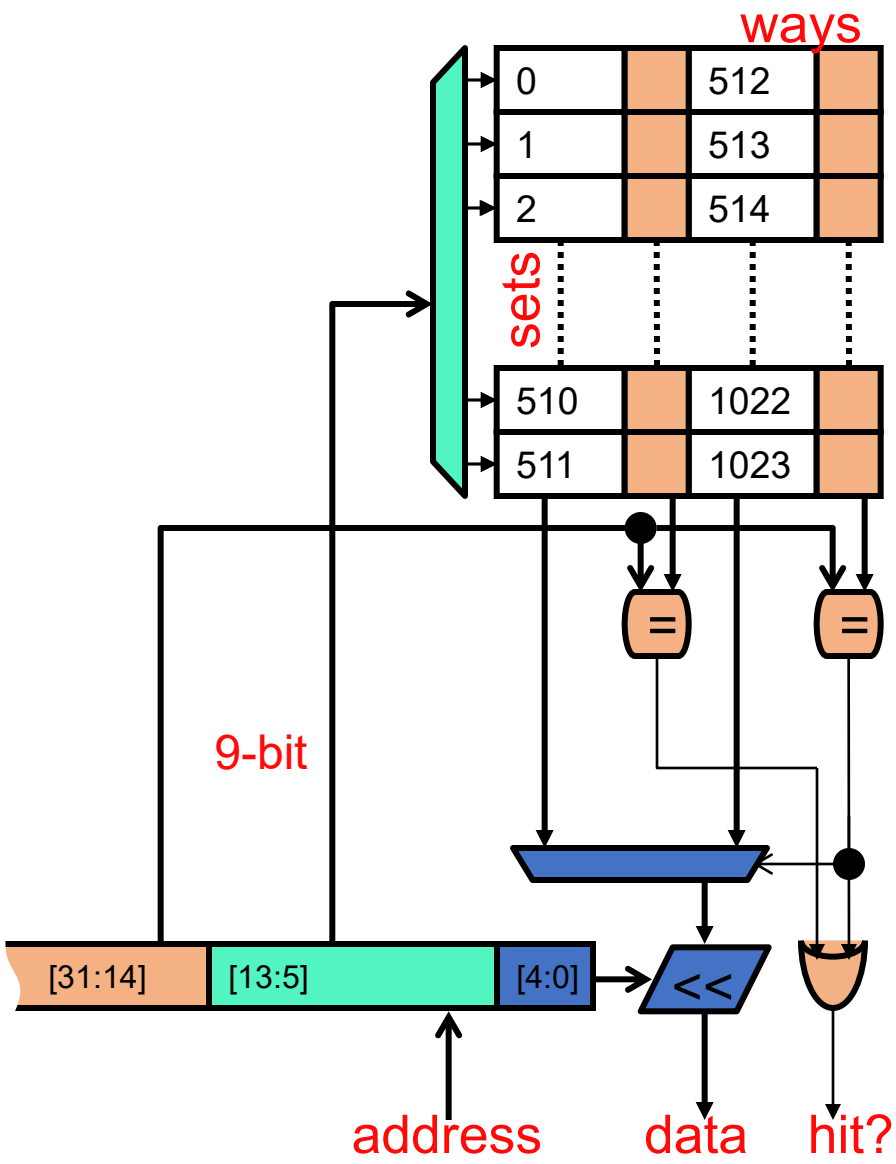
: _ (

Set-Associativity

- **Set-associativity**
  - Block can reside in one of few frames
  - Frame groups called ***sets***
  - Each frame in set called a ***way***
  - This is **2-way set-associative (SA)**
  - 1-way → **direct-mapped (DM)**
  - 1-set → **fully-associative (FA)**

  +Reduces conflicts
  –Increases latency$_{hit:}$ additional muxing

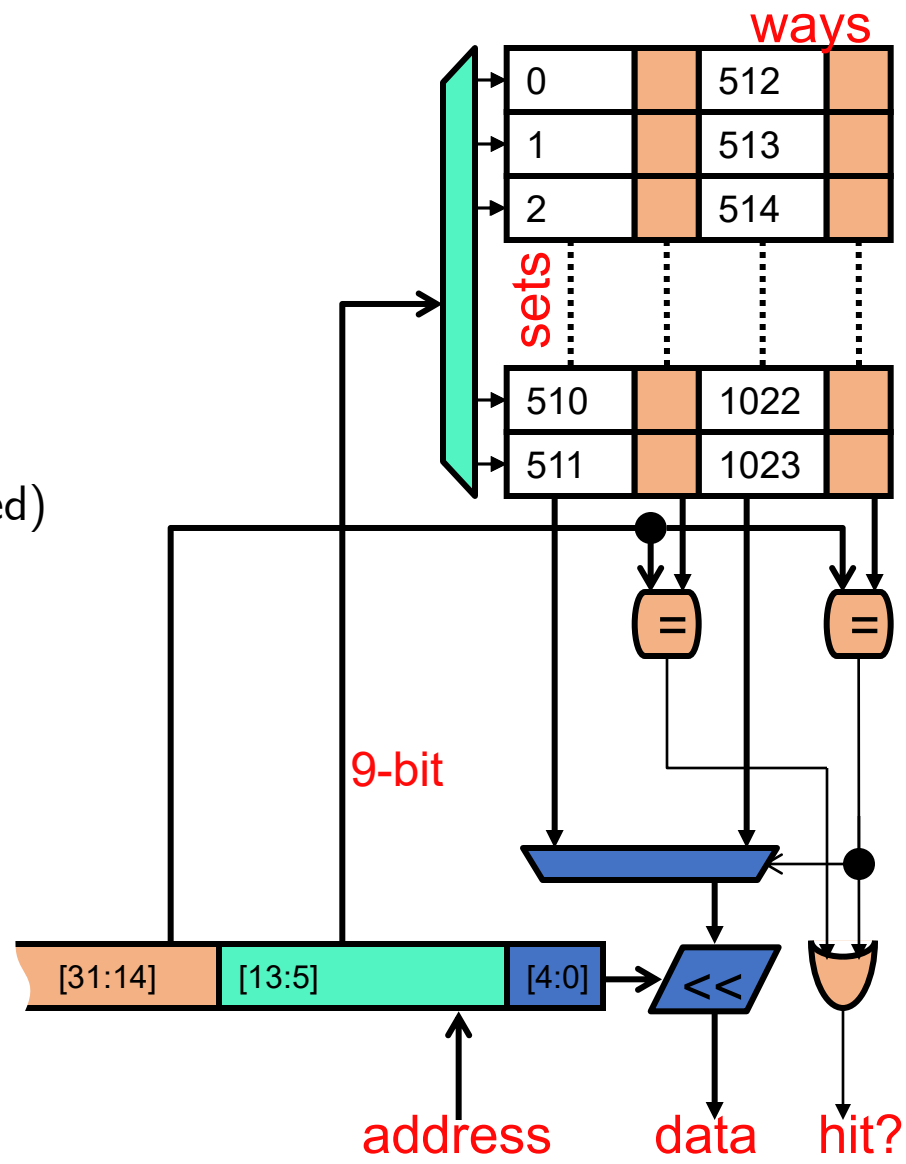
  - Note: valid bit not shown



Set-Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames in parallel
  - **Any** (match and valid bit), Hit

  - Notice tag/index/offset bits
    - Only 9-bit index (versus 10-bit for direct mapped)
  - Notice block numbering



Associativity and Performance

- Parameters: 32B cache, 4B blocks, **2-way set-associative**
  - Initial contents : 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130



Cache contents	Address	Outcome
[0000,0100], [0010,0110], [0020,0120], [0030,0130]	3020	Miss
[0000,0100], [0010,0110], [0120,**3020**], [0030,0130]	3030	Miss
[0000,0100], [0010,0110], [0120,3020], [0130,**3030**]	2100	Miss
[0100,**2100**], [0010,0110], [0120,3020], [0130,3030]	0012	Hit
[0100,2100], [0110,**0010**], [0120,**3020**], [0130,3030]	0020	Miss
[0100,2100], [0110,0010], [3020,**0020**], [0130,3030]	0030	Miss
[0100,2100], [0110,0010], [3020,0020], [3030,**0030**]	0110	Hit
[**0100**,2100], [0010,**0110**], [3020,0020], [3030,0030]	0100	**Hit (avoid conflict)**
[**2100,0100**], [0010,0110], [3020,0020], [3030,0030]	2100	**Hit (avoid conflict)**
[0100,**2100**], [0010,0110], [**3020**,0020], [3030,0030]	3020	**Hit (avoid conflict)**



Hill's 3C Miss Rate Classification

- Compulsory
  - Miss caused by initial access
- Capacity
  - Miss caused by finite capacity
  - I.e., would not miss in infinite cache
- Conflict
  - Miss caused by finite associativity (or bad policy)
  - I.e., would not miss in a (perfect) fully-associative cache

- Coherence (4th C, added by Jouppi)
  - Miss caused by invalidation to enforce coherence
  - Coherence: caches are invisible –
    maintain single writer multiple reader invariant



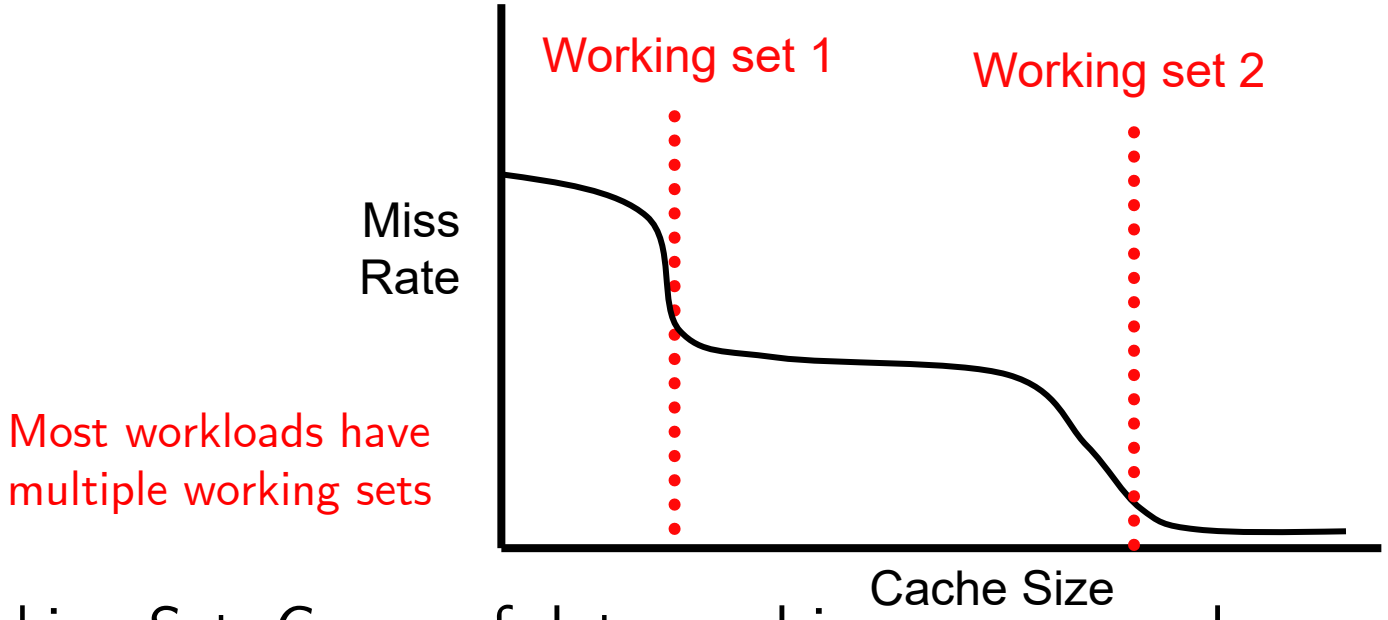
Miss Rate: ABC

- **Capacity**
  - \+ Decreases capacity misses
  - – Increases $\text{latency}_{\text{hit}}$
- **Block size**
  - – Increases conflict misses (fewer frames)
  - \+ Decreases compulsory misses (spatial prefetching)
  - • No effect on $\text{latency}_{\text{hit}}$
  - - May increase $\text{latency}_{\text{miss}}$
- **Associativity**
  - \+ Decreases conflict misses
  - – Increases $\text{latency}_{\text{hit}}$

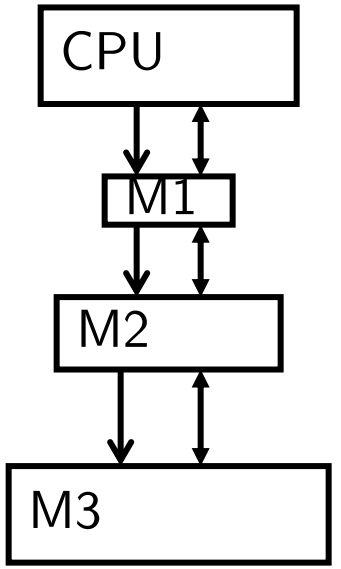
These Tradeoffs explained over the next slides!

Increase Cache Size

- Biggest caches always have better miss rates
  - However $latency_{hit}$ increases
- Diminishing returns



- Working Set: Group of data used in a program phase
  - Tile in a matrix multiply, clusters of nodes in a graph algo. etc.

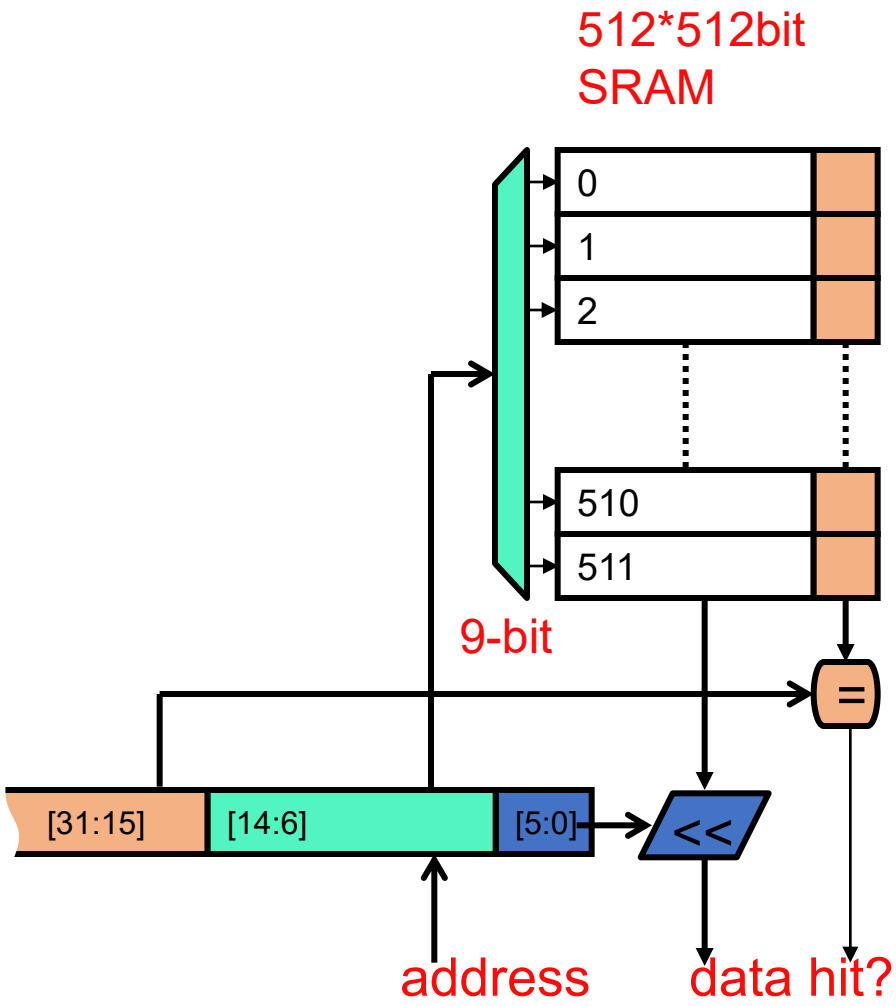


Corollary: working set size affects which level a given data item resides at



Block Size

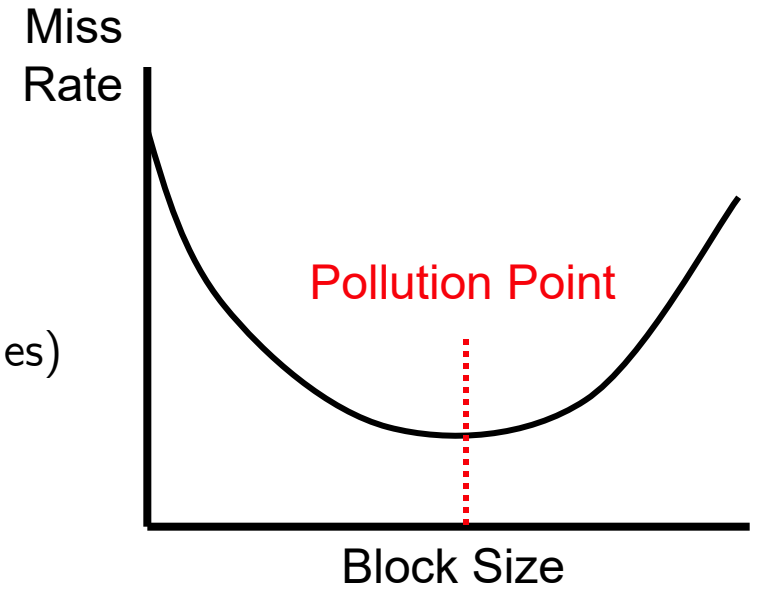
- Given capacity, manipulate $\%_{miss}$ by changing organization
- One option: increase **block size**
  - More offset, less index bits
  - Tag size remains the same
- Ramifications
  - \+ Exploit **spatial locality**
    - Caveat: past a certain point...
  - \+ Reduce tag overhead (why?)
  - – Useless data transfer (needs more bandwidth)
    - – Premature replacement of useful data





Effect of Block Size on Miss Rate

- Two effects on miss rate
  - +**Spatial prefetching (good)**
    - For blocks with adjacent addresses
    - Turns miss/miss into miss/hit pairs
  - –**Interference (bad)**
    - For blocks with non-adjacent addresses (but in adjacent frames)
    - Turns hits into misses by disallowing simultaneous residence
- Both effects always present
  - Spatial prefetching dominates initially
    - Depends on size of the cache
  - Good block size is 32–128B
    - Program dependent





Block Size and Tag Overhead

- Tag overhead of 32KB cache with 1024 32B frames
  - 32B frames → 5-bit offset
  - 1024 frames → 10-bit index
  - 32-bit address – 5-bit offset – 10-bit index = 17-bit tag
  - (17-bit tag + 1-bit valid) * 1024 frames = 18Kb tags = 2.2KB tags
  - ~6% overhead

- Tag overhead of 32KB cache with 512 64B frames
  - 64B frames → 6-bit offset
  - 512 frames → 9-bit index
  - 32-bit address – 6-bit offset – 9-bit index = 17-bit tag
  - (17-bit tag + 1-bit valid) * **512 frames** = 9Kb tags = 1.1KB tags

  +~3% overhead



Block Size and Performance

- Parameters: 8-bit addresses, 32B cache, **8B blocks**
  - Initial contents : 0000(0010), 0020(0030), 0100(0110), 0120(0130)

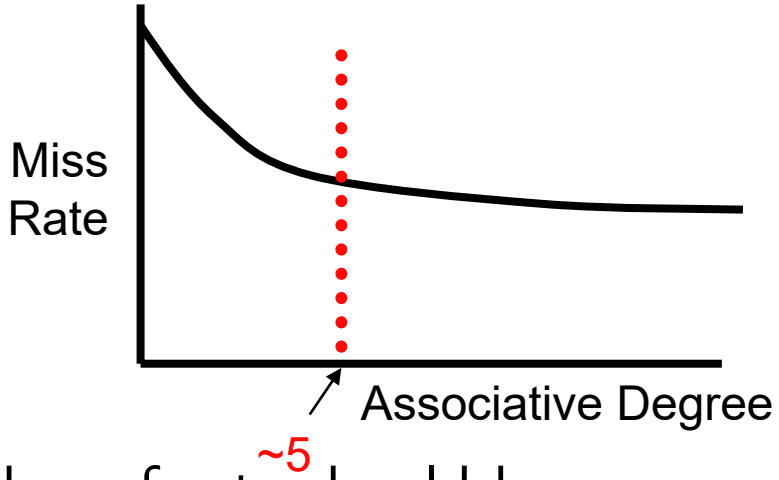


Cache contents -- prior to access (implicit block)	Address	Outcome
0000(0010), 0020(0030), 0100(0110), 0120(0130)	3020	Miss
0000(0010), **3020(3030)**, 0100(0110), 0120(0130)	3030	**Hit (spatial locality)**
0000(0010), 3020(3030), 0100(0110), 0120(0130)	2100	Miss
0000(0010), 3020(3030), **2100(2110)**, 0120(0130)	0012	Hit
0000(0010), 3020(3030), 2100(2110), 0120(0130)	0020	Miss
0000(0010), **0020(0030)**, 2100(2110), 0120(0130)	0030	**Hit (spatial locality)**
0000(0010), 0020(0030), 2100(2110), 0120(0130)	0110	**Miss (conflict)**
0000(0010), 0020(0030), **0100(0110)**, 0120(0130)	0100	**Hit (spatial locality)**
0000(0010), 0020(0030), 0100(0110), 0120(0130)	2100	Miss
0000(0010), 0020(0030), **2100(2110)**, 0120(0130)	3020	Miss



Increase Associativity

- Higher associative caches have better miss rates
  - However $\text{latency}_{\text{hit}}$ increases
- Diminishing returns



- Block-size and number of sets should be powers of two
  - Makes indexing easier (just rip bits out of the address)
  - 3-way set-associativity? No problem
- **New Question: Where to put each block?**

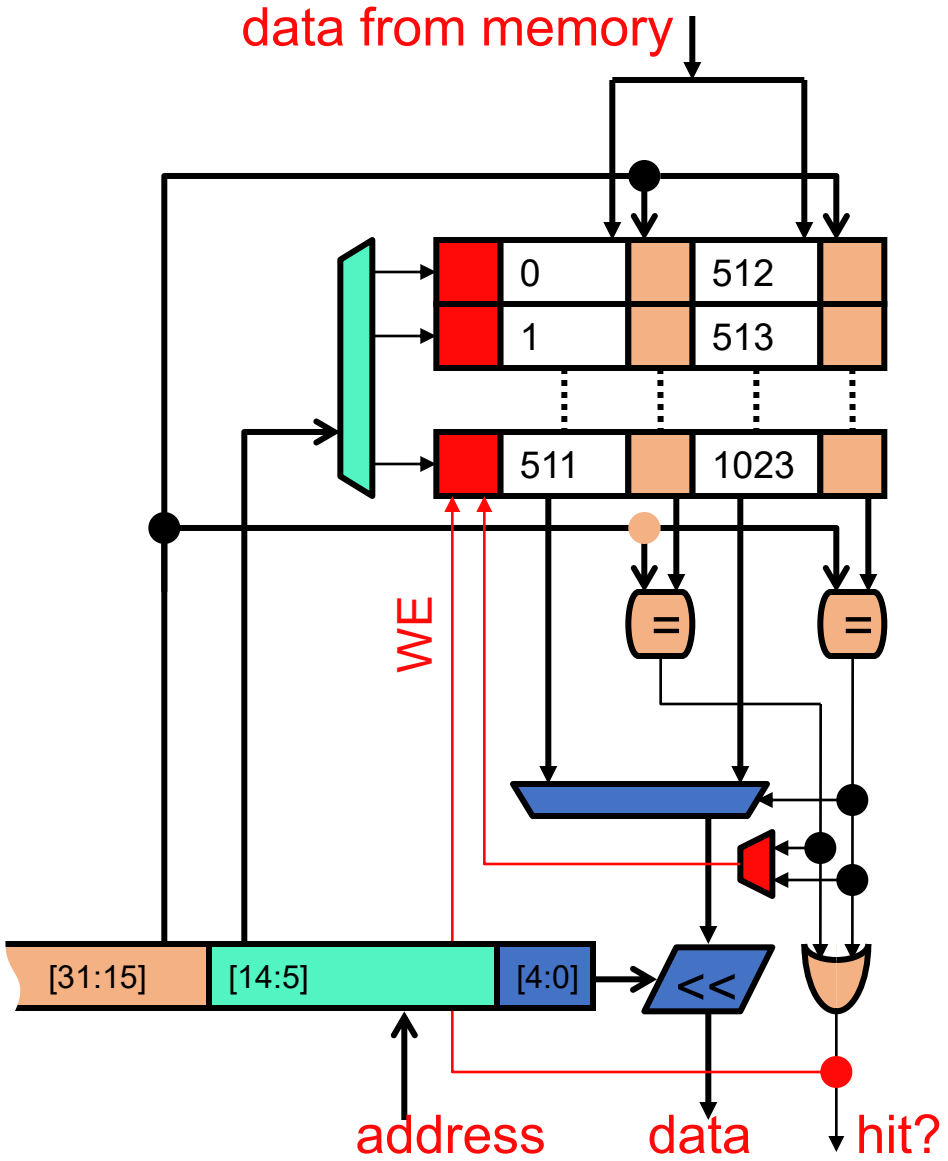


Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Some options
  - **Random**
  - **FIFO (first-in first-out)** (where would this be good?)
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU ~= least likely to be used in future
  - **NMRU (not most recently used)**
    - Remember index of most recently used – don't pick that one
    - An easier to implement approximation of LRU (and is LRU for 2-way)
  - **NRU (not recently used)**
    - One bit per block today whether it's recently used (randomly clear these)
  - **RRIP**: Predict the re-reference interval for a cache block
    - Counter set to 0 on
  - Unachievable optimum?
    - **Belady's**: replace block that will be used furthest in future
    - **Hawkeye[ISCA2016]**: predictor that learns the optimal policy from Belady's

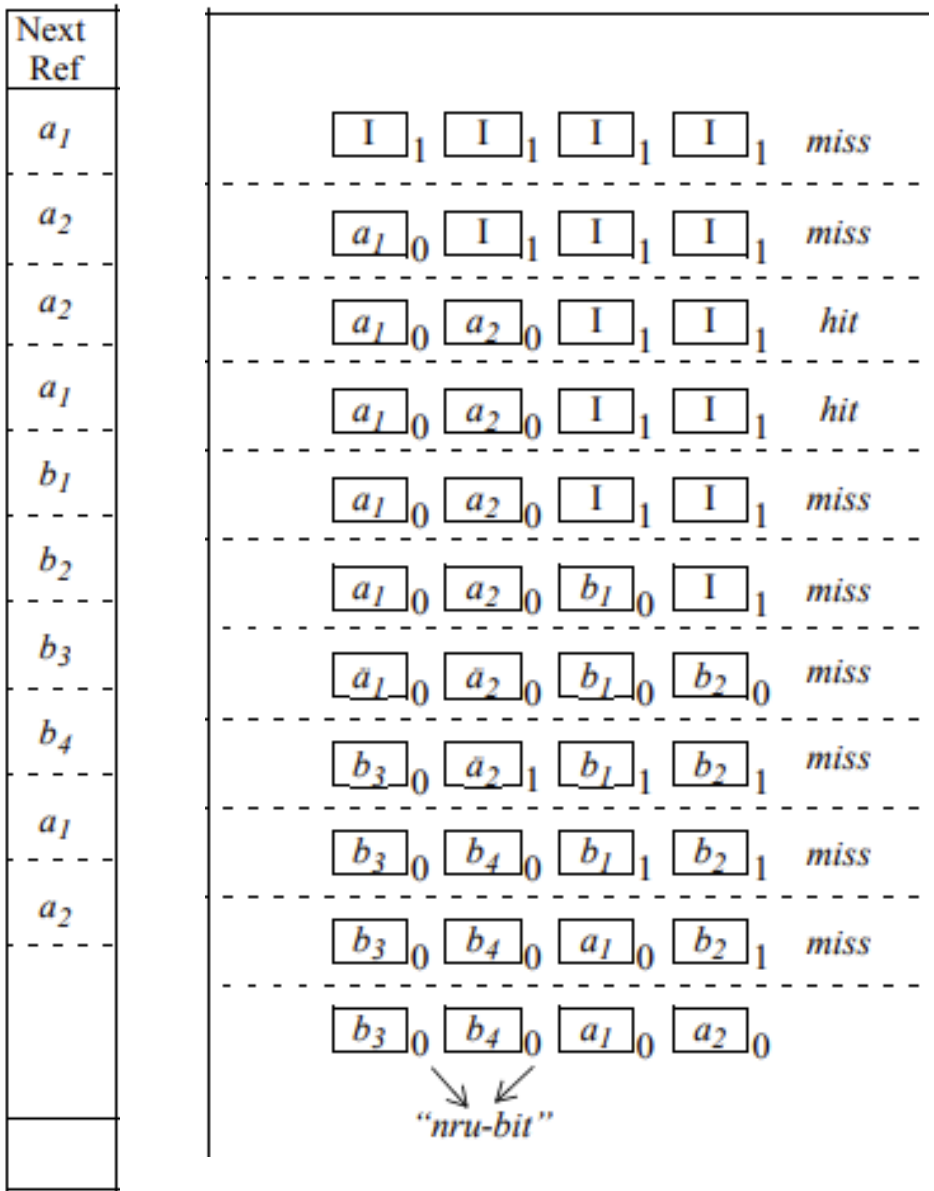
NMRU and Miss Handling

- Add **MRU** field to each set
  - MRU data is encoded "way"
  - Hit? update MRU
- MRU/LRU bits updated on each access
- Cheap version of LRU



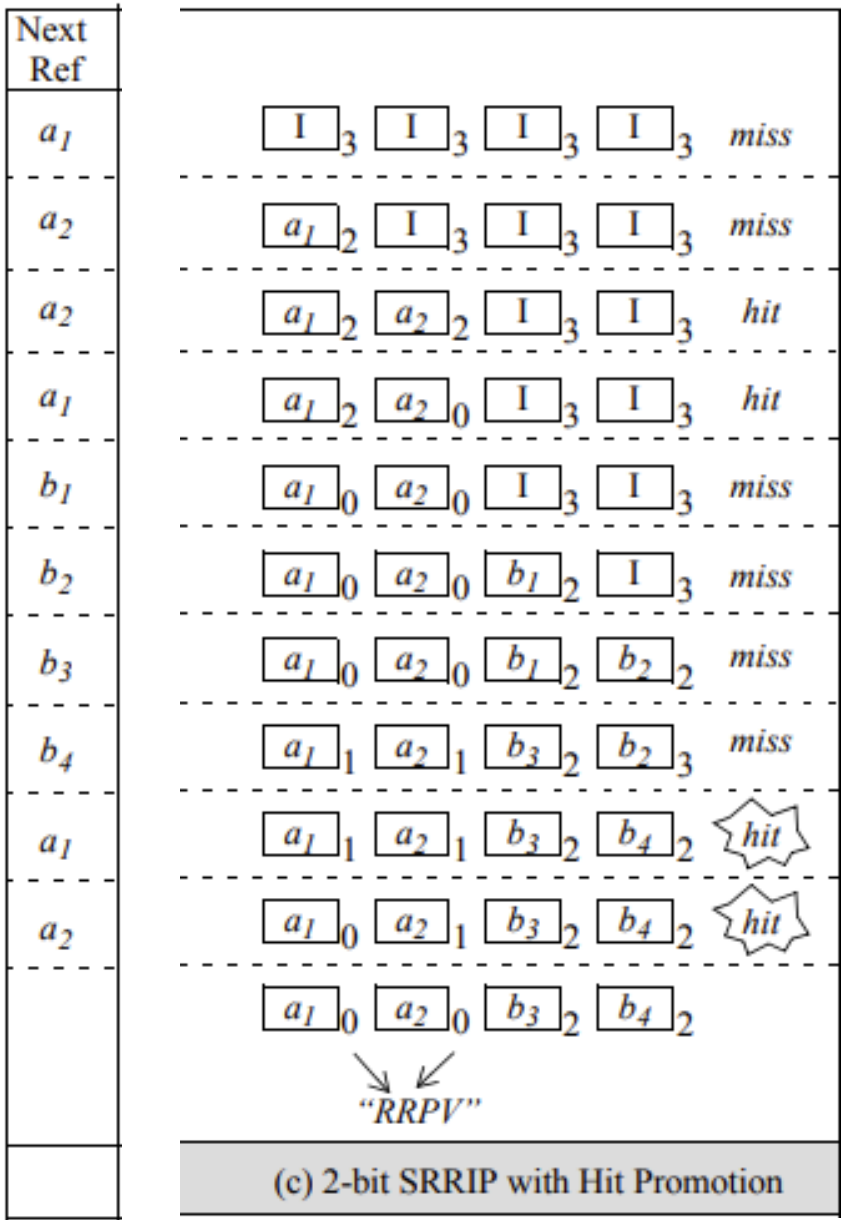
NRU Replacement

- Cache Hit:
  - set nru-bit of block to '0'
- Cache Miss:
  1. search for first '1' from left
  2. if '1' found go to step 5
  3. set all nru-bits to '1'
  4. goto step1
  5. replace block and set nru-bit to '1
- Cheap version of rrip



RRIP Replacement

- Cache Hit:
  - set RRPV of block to '0'
- Cache Miss:
  1. search for first '3' from left
  2. if '3' found go to step 5
  3. increment all RRPVs
  4. goto step 1
  5. replace block and set RRPV to '2'



(c) 2-bit SRRIP with Hit Promotion



Miss Rate: ABC

- **Capacity**
  - \+ Decreases capacity misses
  - – Increases $\text{latency}_{\text{hit}}$
- **Block size**
  - – Increases conflict misses (fewer frames)
  - \+ Decreases compulsory misses (spatial prefetching)
  - • No effect on $\text{latency}_{\text{hit}}$
  - - May increase $\text{latency}_{\text{miss}}$
- **Associativity**
  - \+ Decreases conflict misses
  - – Increases $\text{latency}_{\text{hit}}$



Classifying Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
    - Identify? easy
  - **Capacity**: miss caused ONLY because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? (assume you must classify each block)
      - Problem: want to be neutral to the replacement policy…
    - Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames in cache)
  - **Conflict**: miss caused because associativity/replacement
    - Identify? **All other misses**
    - *Captures specific problems with your cache policies*

  - Who cares? Different techniques for attacking different misses



Cache Performance Simulation

- Parameters: 8-bit addresses, 32B cache, 4B blocks
  - Initial contents : 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130
  - Initial blocks accessed in increasing order

Cache contents	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	Miss
0000, 0010, **3020**, 0030, 0100, 0110, 0120, 0130	3030	Miss
0000, 0010, 3020, **3030**, 0100, 0110, 0120, 0130	2100	Miss
0000, 0010, 3020, 3030, **2100**, 0110, 0120, 0130	0012	Hit
0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130	0020	Miss
0000, 0010, **0020**, 3030, 2100, 0110, 0120, 0130	0030	Miss
0000, 0010, 0020, **0030**, 2100, 0110, 0120, 0130	0110	Hit
0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130	0100	Miss
0000, 1010, 0020, 0030, **0100**, 0110, 0120, 0130	2100	Miss
1000, 1010, 0020, 0030, **2100**, 0110, 0120, 0130	3020	Miss

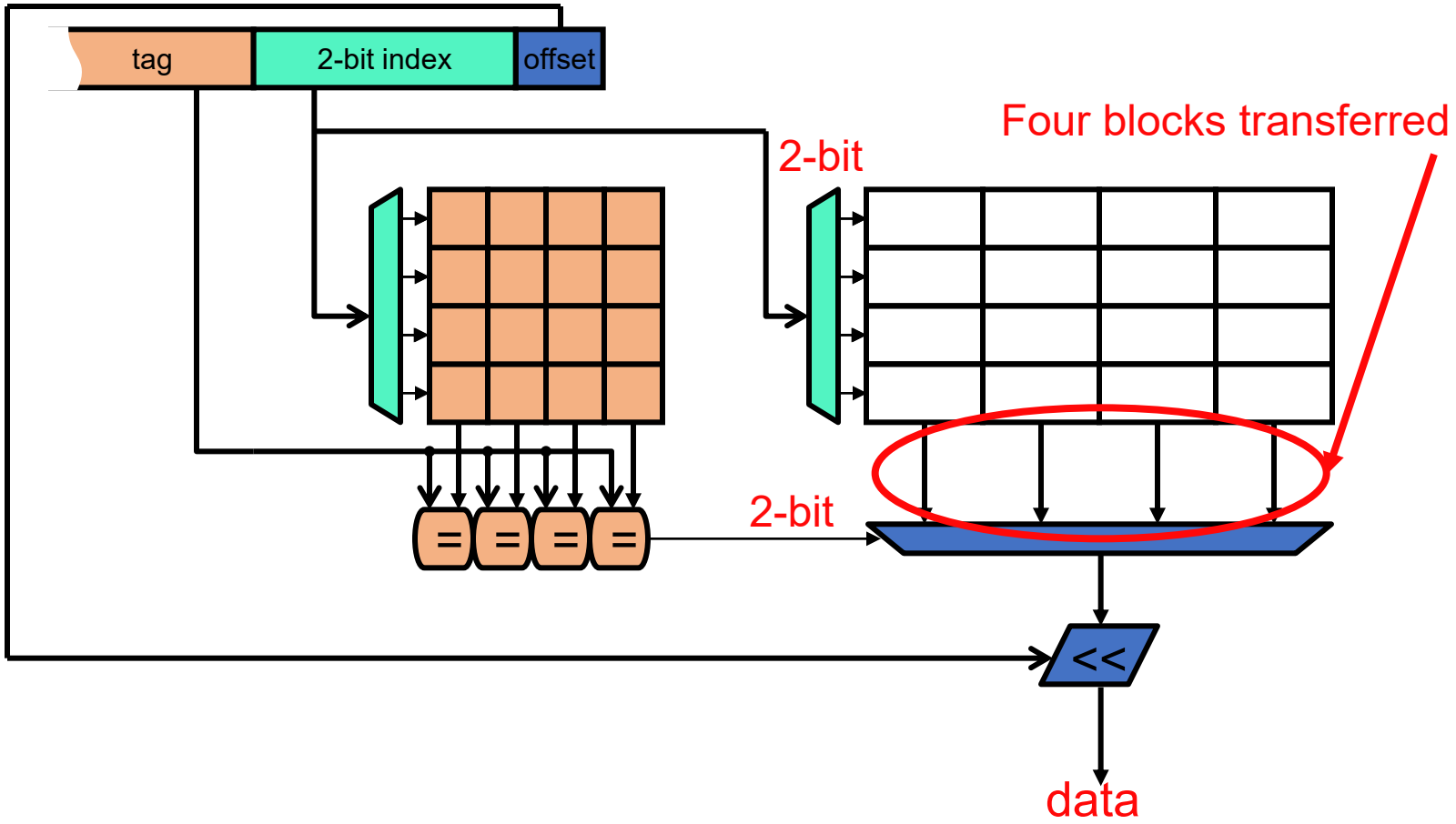


Summary so far...

- Memory hierarchy concepts
- ABC's of Cache organization
  - Associativity (reduce conflict misses)
  - Block Size (reduce compulsory misses)
  - Capacity (reduce capacity misses)
  - Miss Classification
- Practical Concerns
  - Latency Considerations
  - What about Writes?
- Beyond ABCs
  - Advanced Conflict Avoidance
  - Data layout transformations
  - Prefetching (hardware/software)
- Low power techniques

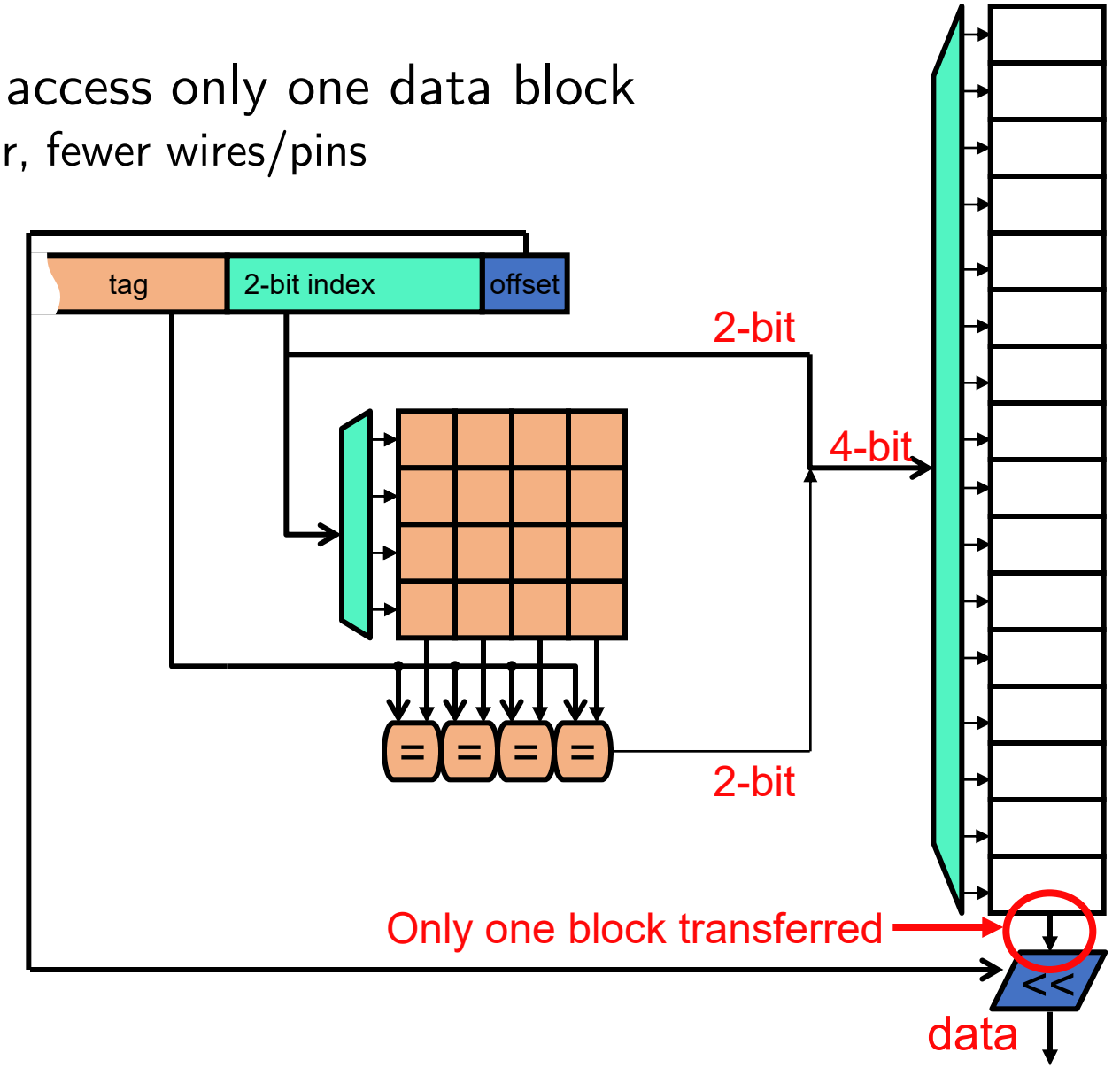
Parallel or Serial Tag Access?

- Note: data and tags actually physically separate
  - Split into two different arrays
- Parallel access example:



Serial Tag Access

- Tag match first, then access only one data block
  - Advantages: lower power, fewer wires/pins
  - Disadvantages: slow





Question

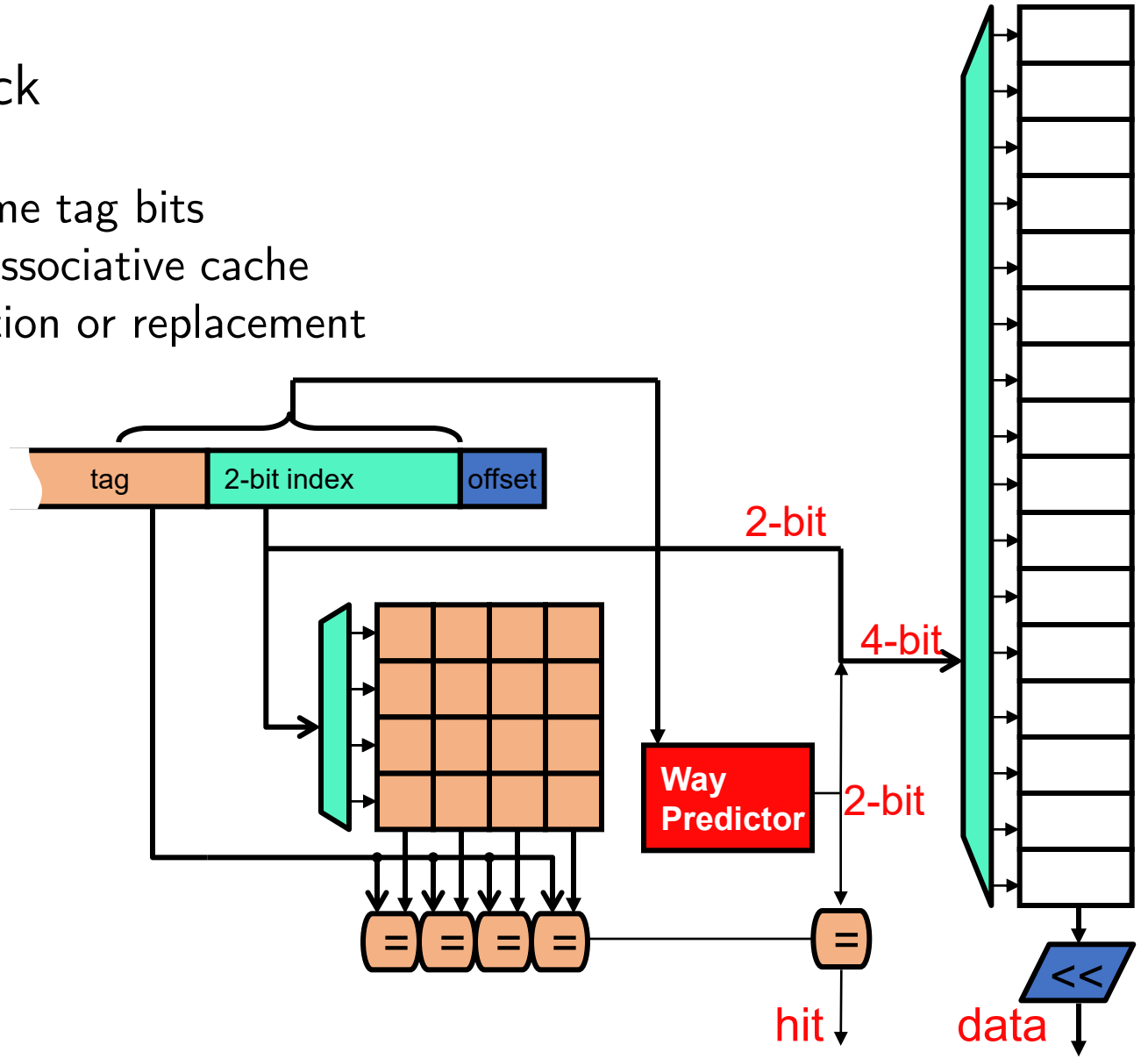
Get best of both worlds?
(high associativity + low hit time)

(Hint, prediction?)

Best of Both? Way Prediction

- Predict "way" of block
  - Just a "hint"
  - Use the index plus some tag bits
  - Table of n-bit for 2^n associative cache
  - Update on mis-prediction or replacement

- Advantages
  - Fast
  - Low-power
- Disadvantage
  - More "misses"





What about miss cost?



Miss Cost: Critical Word First/Early Restart

- Observation: Bus between levels of the memory hierarchy are sometimes narrower than the block size
- $\text{latency}_{\text{miss}} = \text{latency}_{\text{access}} + \text{latency}_{\text{transfer}}$
  - $\text{latency}_{\text{access}}$: time to get first word
  - $\text{latency}_{\text{transfer}}$: time to get rest of block
  - Implies whole block is loaded before data returns to CPU

- Optimization
  - **Early restart**: send requested word to CPU as soon as it arrives in the cache
    - Get rest of block load into cache in parallel
  - **Critical word first**: return requested word first
    - Must arrange for this to happen (bus, memory must cooperate)
  - $\text{latency}_{\text{miss}} = \text{latency}_{\text{access}}$



Large Blocks and Subblocking

- Large cache blocks can waste bus bandwidth if block size is larger than spatial locality
  - divide a block into subblocks
  - associate separate valid bits for each subblock
- Sparse access patterns can use 1/S of the cache
  - S is subblocks per block
- Why would you do this?
  - Sparse access pattern? Only save sublocks you need
  - Latency of fill may be long – could we service some sublocks sooner than others?





Miss Cost: Lockup Free Cache

- **Lockup free**: allows other accesses while miss is pending
  - Consider: Load [r1] -> r2; Load [r3] -> r4; Add r2, r4 -> r5
  - Makes sense for processors that can go ahead despite D$ miss (out-of-order)
  - Implementation: **miss status holding register (MSHR)**
    - Remember: miss address, chosen frame, requesting instruction
    - When miss returns know where to put block, who to inform
  - How many MSHRs?
    - Simplest scenario: "hit under miss" (one MSHR)
    - More common: "miss under miss" (multiple MSHRs)
  - **Coalescing MSHR**: Combine misses to same cache line
    - Request hits in MSHR, then combine it
  - More Bandwidth: Split-transaction bus/interconnect
    - Separate bus for request and response



So what about writes?



Write Issues

- So far we have looked at reading from cache (loads)
- What about writing into cache (stores)?

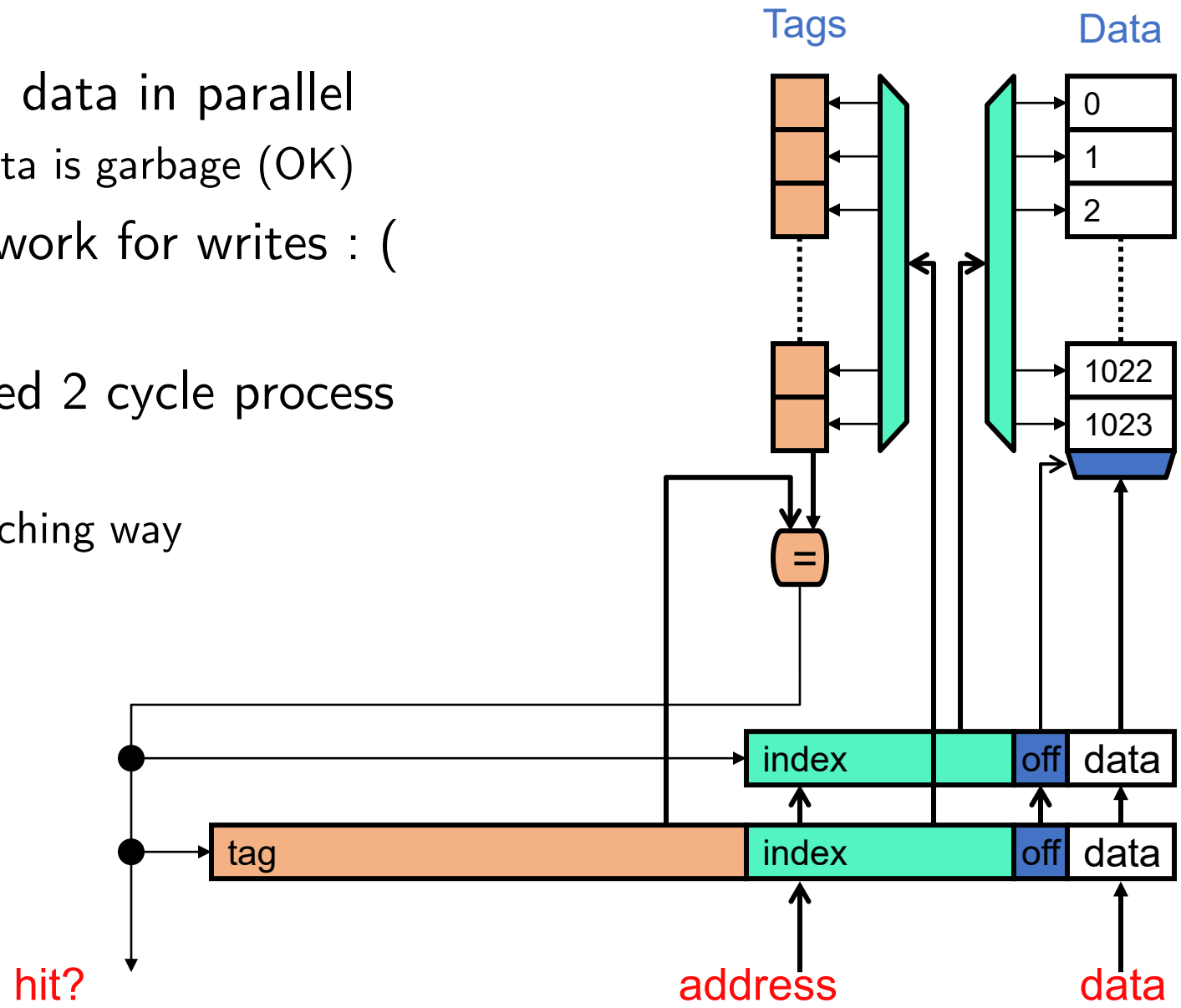
- Few new issues
  - Tag/data access
  - What to do on write hit? (Write-through vs. write-back)
  - ... or miss? (Write-allocate vs. write-not-allocate)

- Buffers
  - Store buffers (queues)
  - Write buffers
  - Writeback buffers

Tag/Data Access

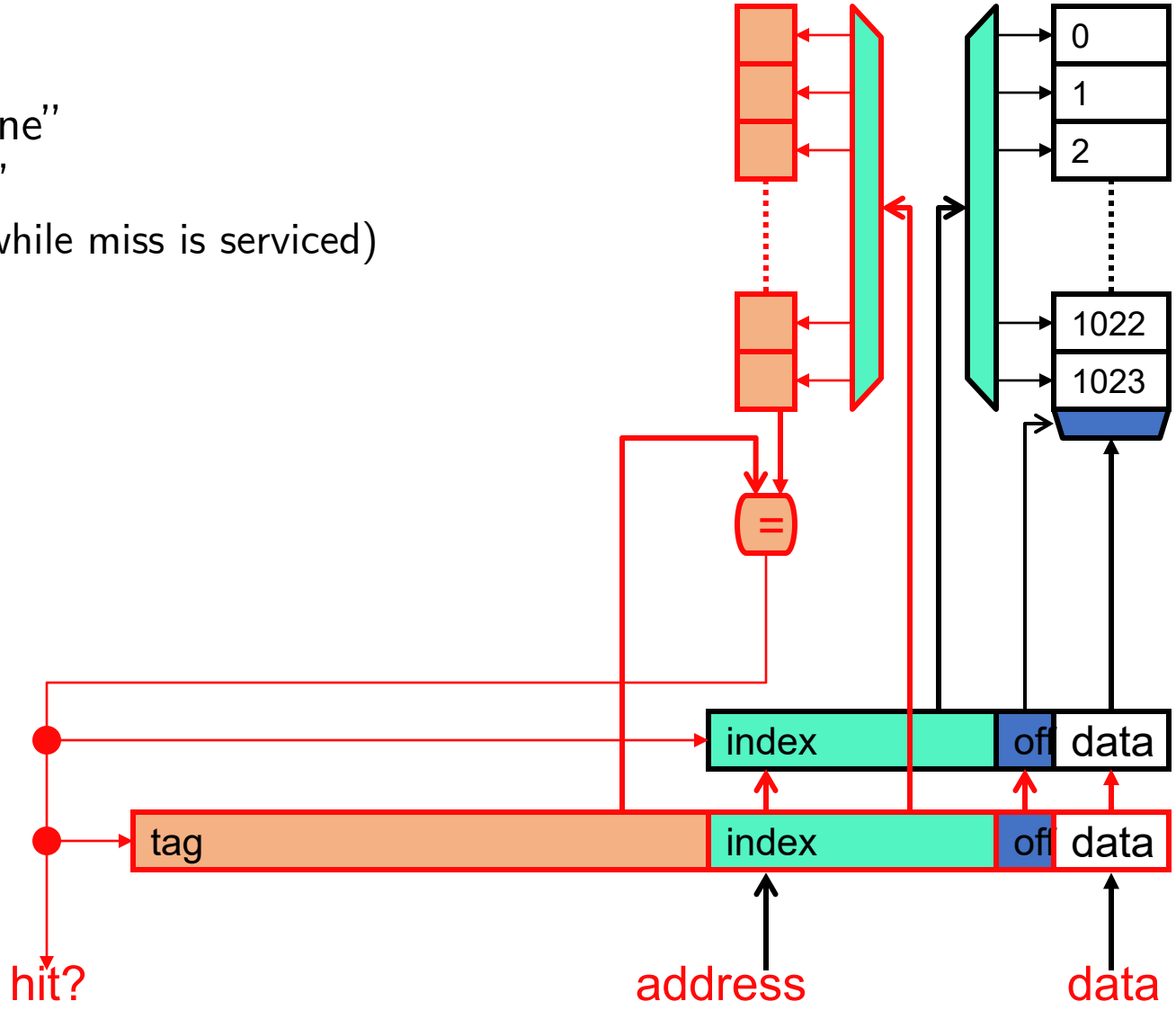
- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK)
- Same trick doesn't work for writes : (

- Writes are a pipelined 2 cycle process
  - Cycle 1: match tag
  - Cycle 2: write to matching way



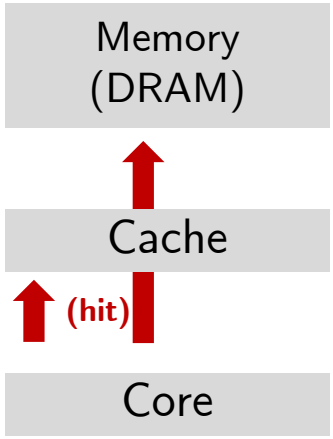
Tag/Data Access

- Cycle 1: check tag
  - Hit? Advance "store pipeline"
  - Miss? Stall "store pipeline"
    - (or remove from pipeline while miss is serviced)
- Cycle 2: write data



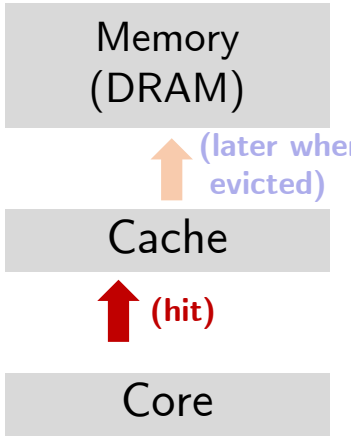
Writes: To defer or not to defer?

What to do on a write-hit?



Write to all levels, in spite of hit.

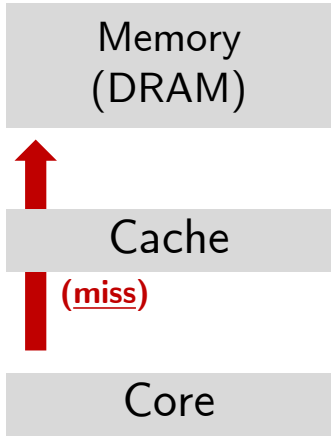
Write Through



Write to cache, evict later

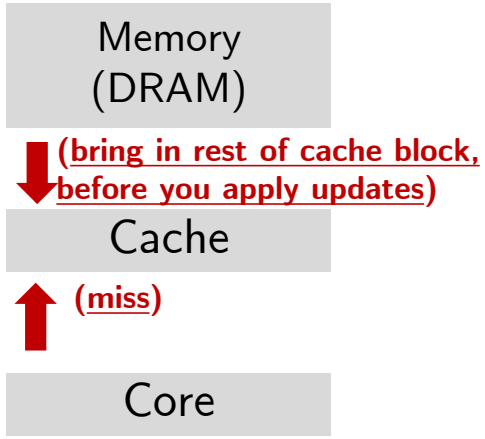
Write-Back

What to do on a write-miss?



Just write to memory, skip cache

Write-No-Allocate



Bring in rest of cache line, write to cache

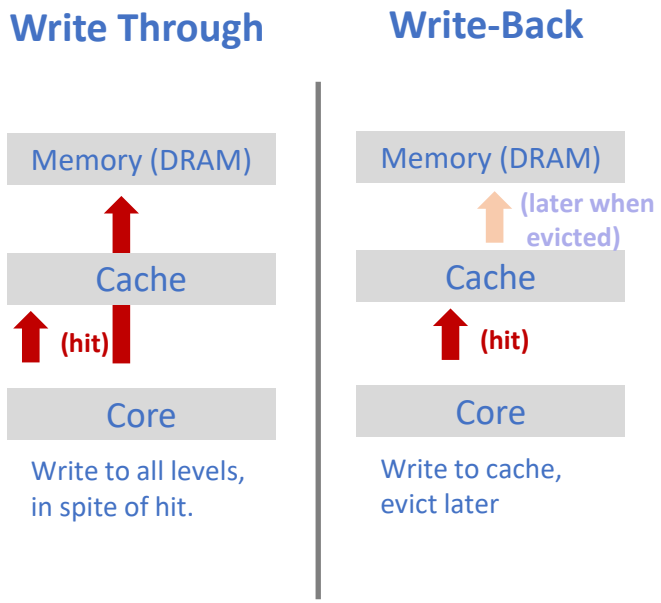
Write-Allocate

Optimization: keep a byte mask of dirty values to avoid line fetch



Write-Through vs. Write-Back

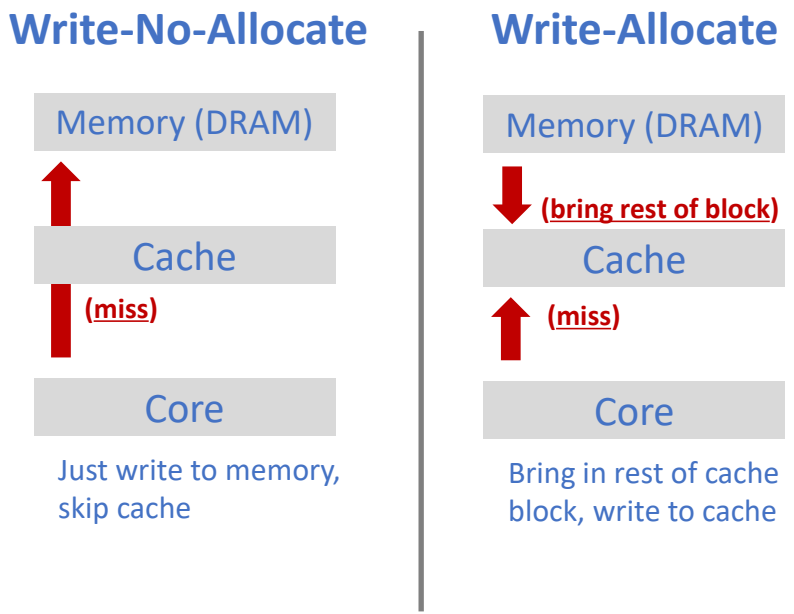
- When to propagate new value? (write hit)
  - **Write-through**: immediately
    - +Conceptually simpler
    - +When replacing a cache line, free to drop any line (no need to writeback a dirty line)
    - –Requires additional bus bandwidth
  - **Write-back**: when block is replaced
    - Requires additional "**dirty**" bit per block
    - +Lower bus bandwidth for large caches
      - Only writeback dirty blocks
    - –Non-uniform miss latency
      - Clean miss: one transaction with lower level (fill)
      - Dirty miss: two transactions (writeback + fill)
      - Writeback buffer: fill, then writeback (later)





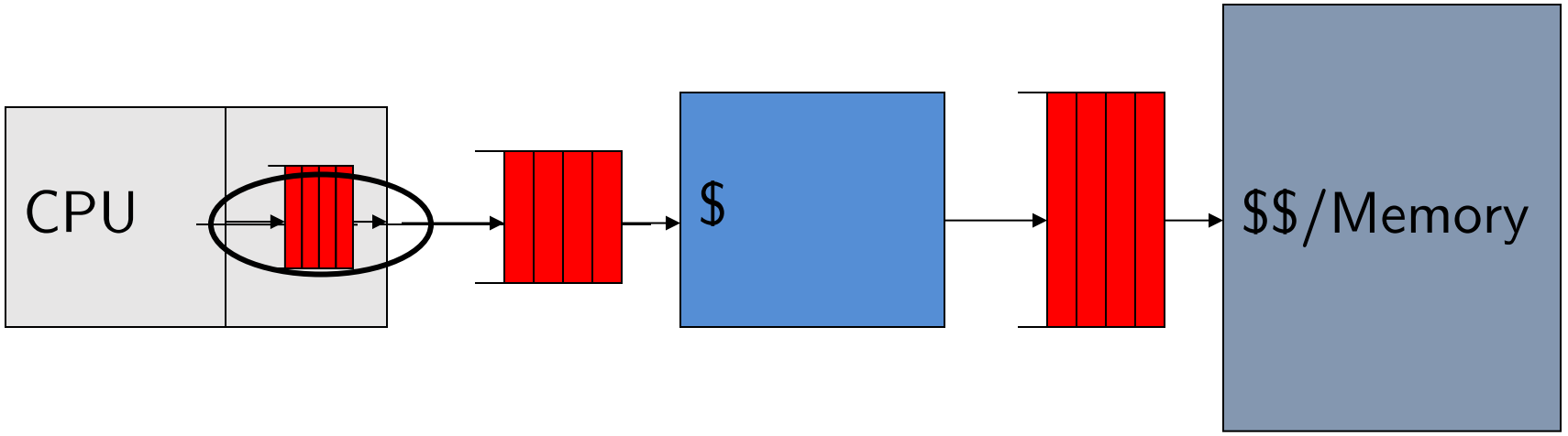
Write-allocate vs. Write-non-allocate

- When to propagate new value? (write miss)
  - **Write-non-allocate**: just write to next level
    - –Potentially more read misses
    - +Uses less bandwidth for low locality
    - Used mostly with write-through
  - **Write-allocate**: read block from lower level, write value into it
    - +Decreases read misses
    - +Uses less bandwidth for high locality
    - –Could evict useful data
    - Used mostly with write-back
- Write allocate is common for write-back
  - Write-non-allocate for write through





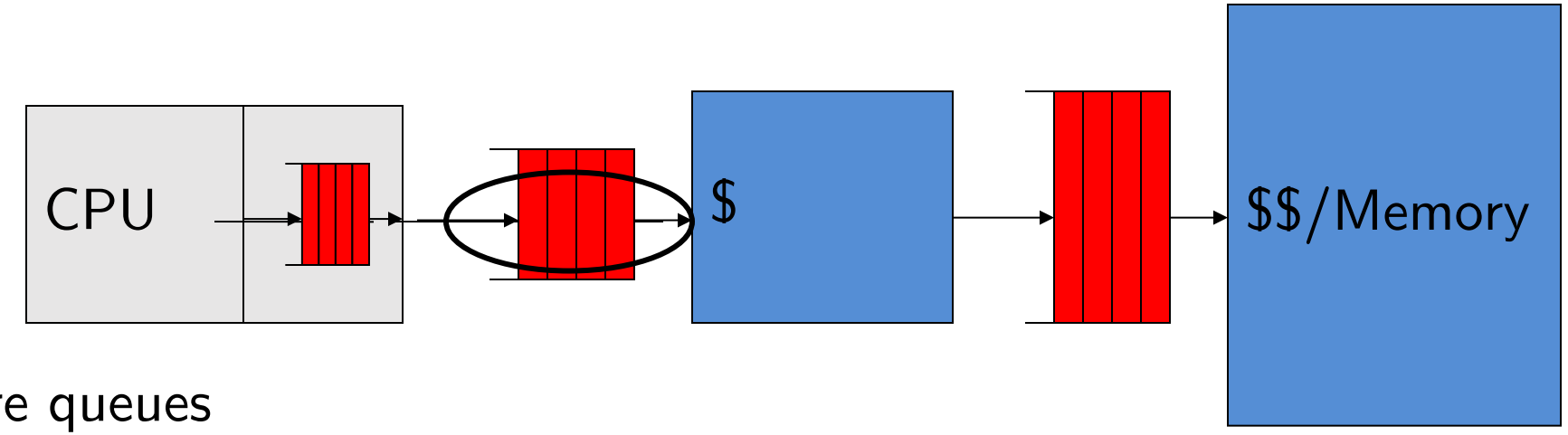
Buffering Writes 1 of 3: Store Queues



- (1) Store queues
  - Part of speculative processor; transparent to architecture
  - Hold speculatively executed stores
  - May rollback store if earlier exception occurs
  - Used to track load/store dependences
- (2) Write buffers
- (3) Writeback buffers



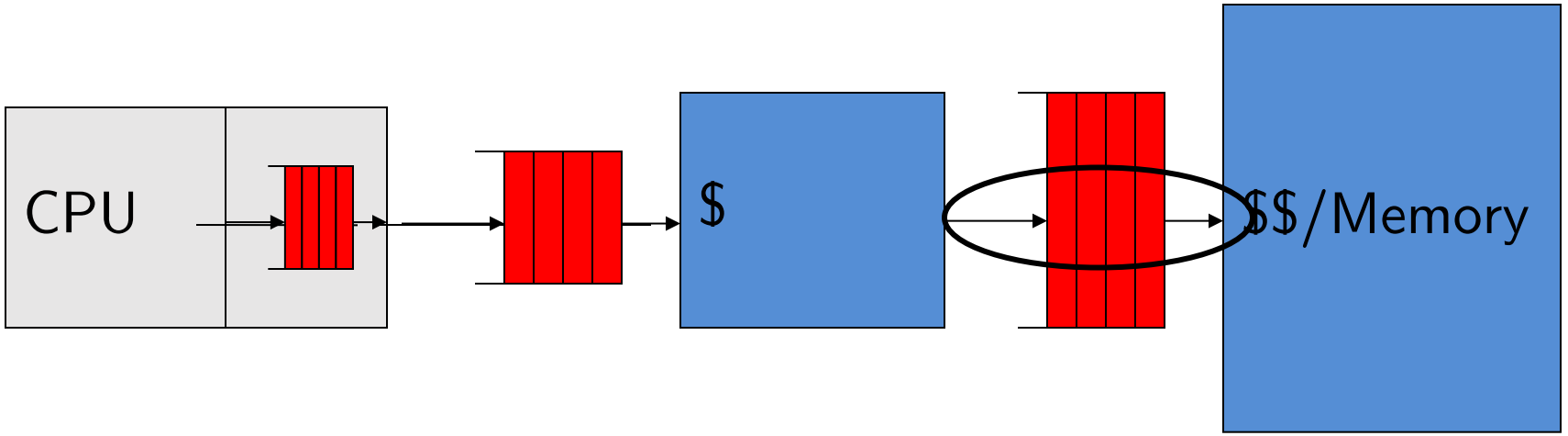
Buffering Writes 2 of 3: Write Buffer



- (1) Store queues
- (2) Write buffers
  - Holds committed architectural state
    - Transparent to single thread
    - May affect memory consistency model
  - Hides latency of memory access or cache miss
  - Bypass values to later loads (or stall)
  - Store queue & write buffer may be in same physical structure
- (3) Writeback buffers



Buffering Writes 3 of 3: Writeback Buffer



- (1) Store queues
- (2) Write buffers

- (3) Writeback buffers
  - Transparent to architecture
  - Holds victim block(s) so miss/prefetch can start immediately
  - (Logically part of cache for multiprocessor coherence)

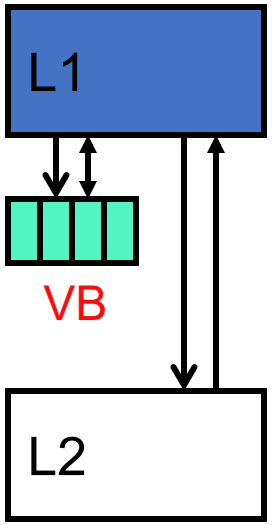


Improving Hit Rate Beyond the ABCs



Conflict Misses: Victim Buffer

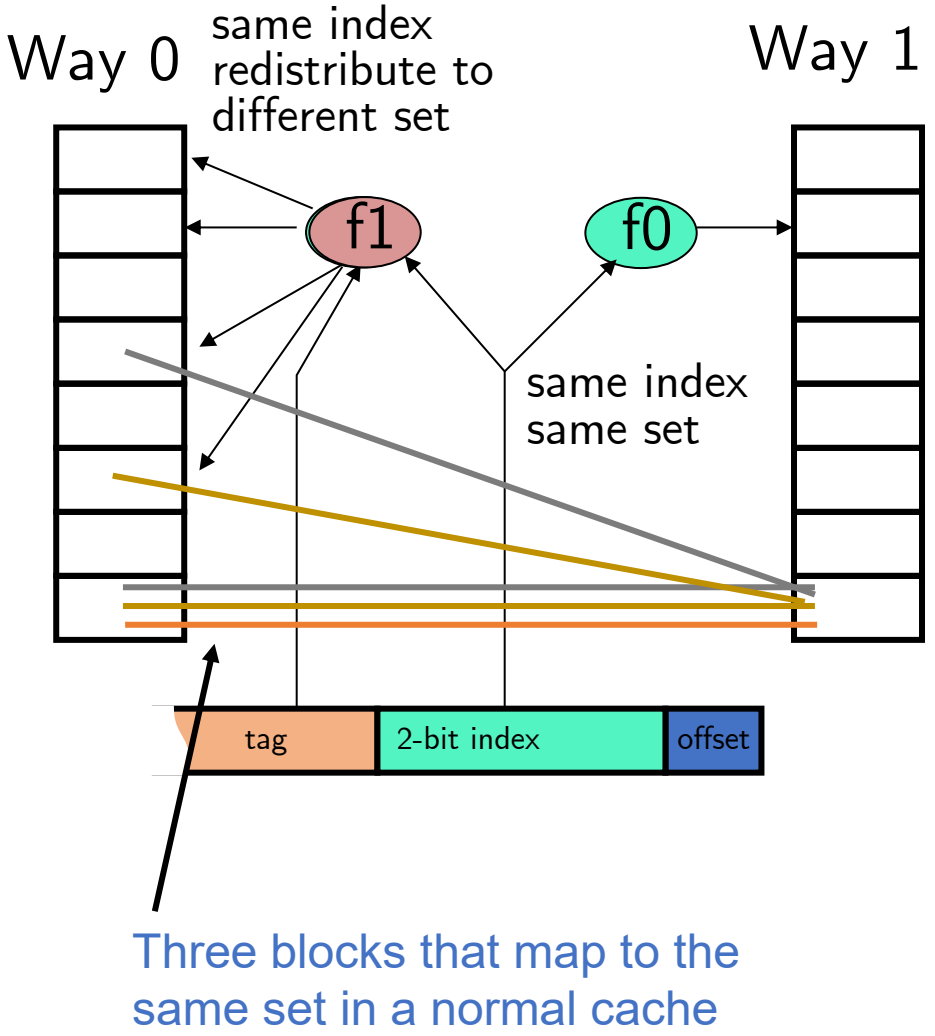
- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (ABC)*
- **Victim buffer (VB)**: small fully-associative cache
  - Sits on L1 fill path
    - Blocks kicked out of L1 placed in VB
    - On miss, check VB: hit? Place block back in L1
  - Intuitively: 8 extra ways, shared among all sets
    - +Only a few sets will need it at any given time
  - Does VB reduce **$\%_{miss}$** or **$latency_{miss}$**?





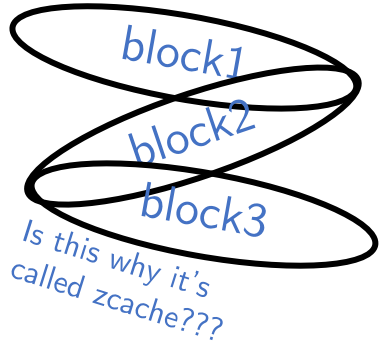
Seznec's Skewed-Associative Cache [1993]

- Principle: Index each way with a different hash function
  - Block only exists in one location in each way, but…
- Blocks that conflict on one way **do not** conflict on another way!
- Benefit: Lower conflict misses and higher utilization than a set-associative cache with the same number of ways
- Drawback?
  - Harder for programmer to reason about what can be stored in cache – maybe that doesn't matter???



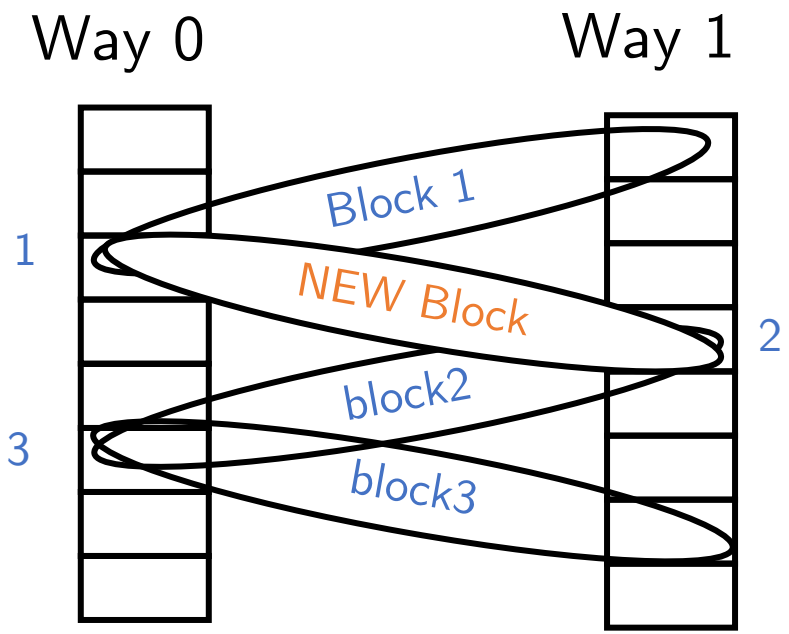


Sanchez Zcache [Sanchez Micro10]



- Zcache (Skewed-assoc + Cuckoo hashing)
- Basic Approach:
  - Hash function to each way +
  - Block only exists in one location in each way
  - Important part: Don't need to writeback the evicted block from the set that you allocate into – if re-use is likely, just move it to some other way to find a less recently used block :)
- What if that block is still valuable? Keep on going!
  - Decoupled ways and associativity: Benefits of very highly associative caches with few physical ways... how?
  - Their insight: benefit of "associativity is not determined by the number of locations that a block can reside in, but by the number of replacement candidates on an eviction."
- Hash function for hit time + Miss is complicated ... probably not for l1 cache

Overly Simplified Example



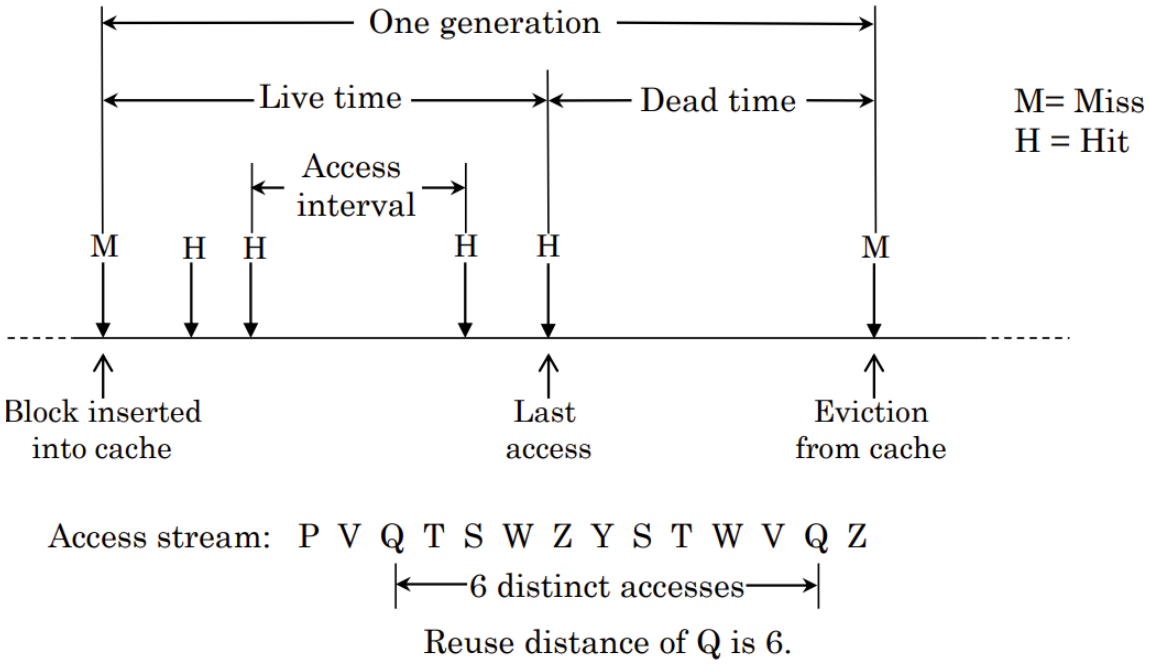
How do they choose which block to replace?

With a third way, it becomes a tree search, why?



Cache Bypassing

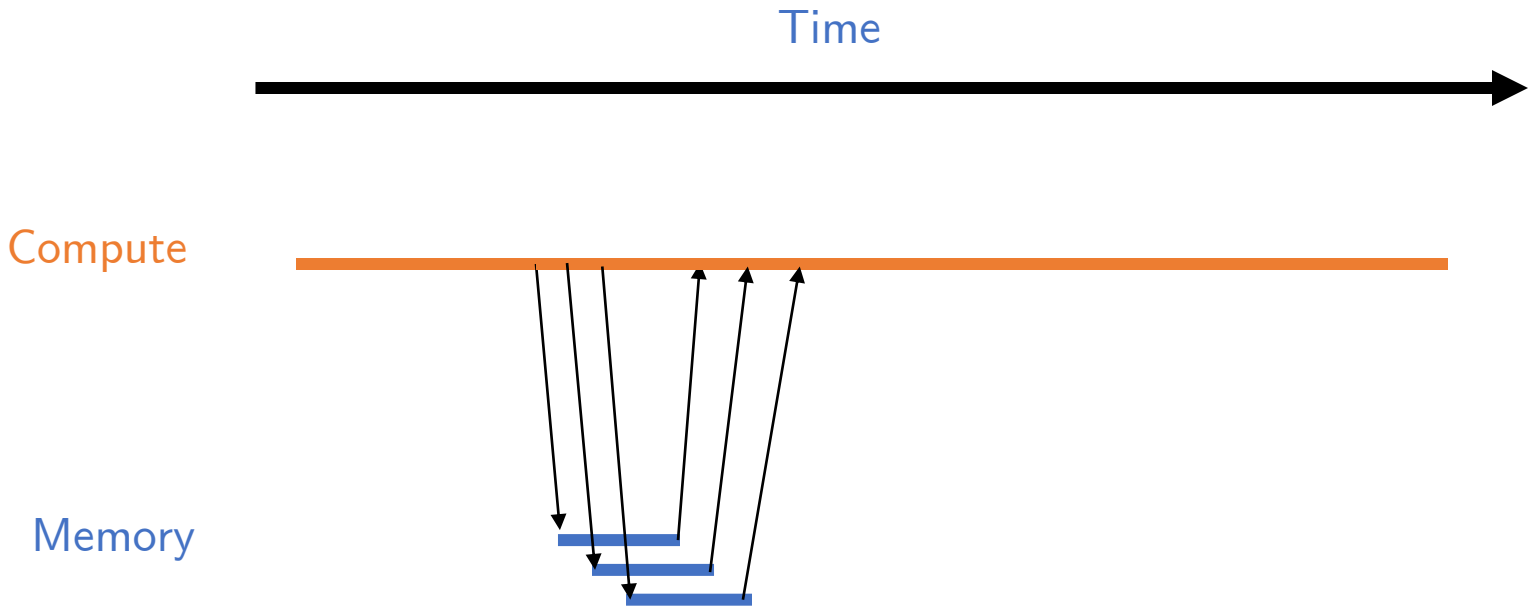
- Lots of data is "Dead On Arrival"
- No hits before evictions
- Idea: Don't cache this data!
- Do it in software:
  - "non-temporal load" – load that is not stored at some cache level
- Do it in hardware:
  - "Counter-based cache replacement and bypassing algorithms 2008"
  - Simplified idea: keep track of which loads' values aren't reused in the cache





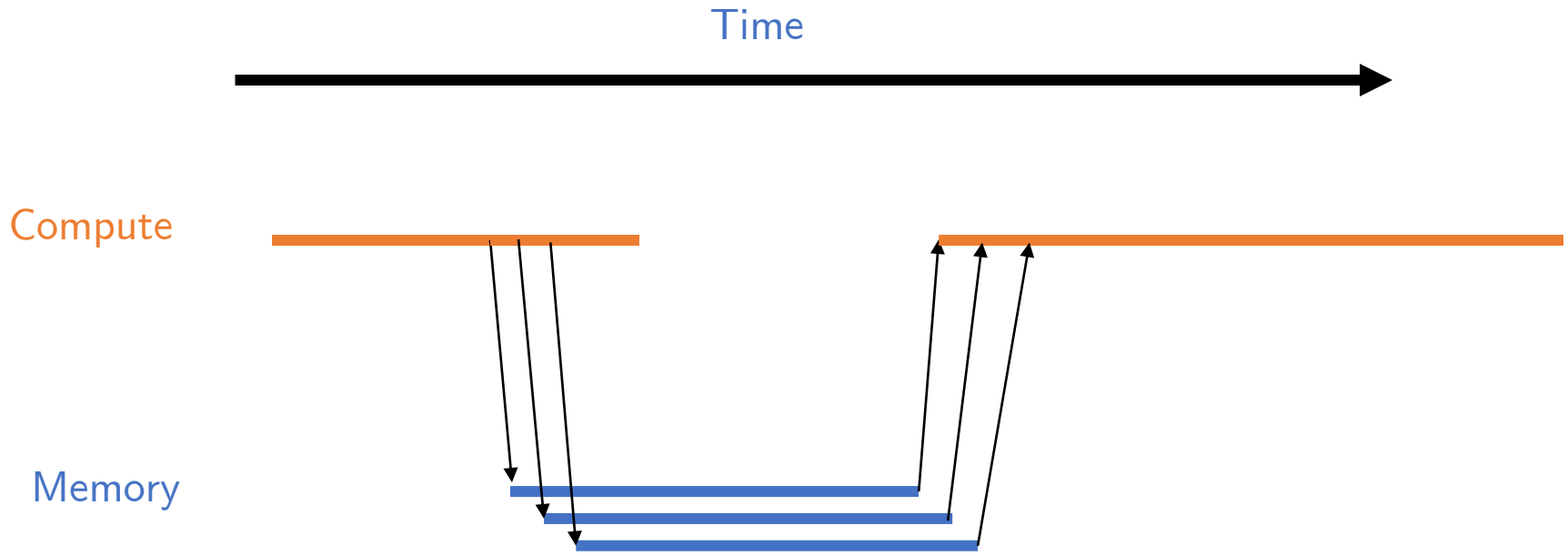
Making misses vanish...
Prefetching data before even requested

Good case: Out-of-order Core can hide misses





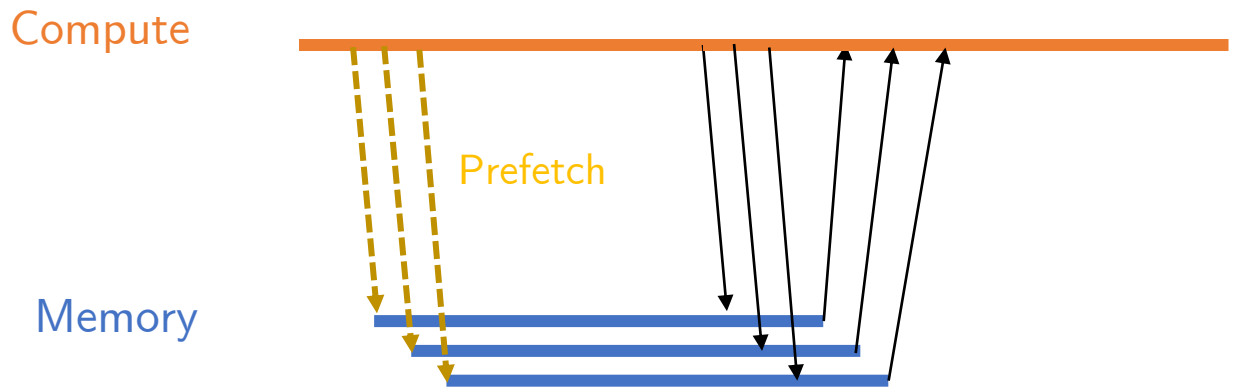
Bad case: Misses can't be hidden





New Idea: Prefetching

- Caches are basically buffers that we can load with any data we want
- Load them with the data we need before we request it? (prefetching)

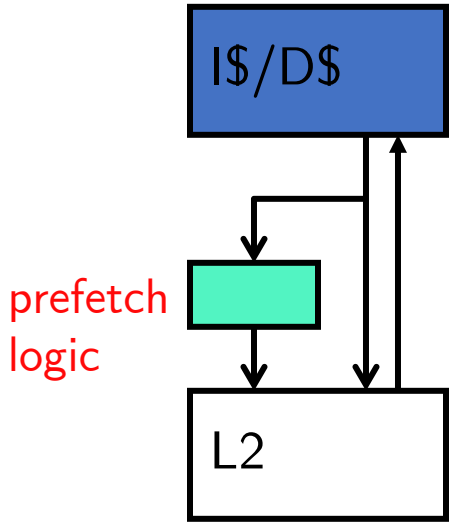




Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
- Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware

- Simple example: **next block prefetching**
  - Miss on address **X** → anticipate miss on **X+block-size**
  - +Works for insns: sequential execution
  - +Works for data: arrays



- **Timeliness**: initiate prefetches sufficiently in advance
- **Coverage**: prefetch for as many misses as possible
- **Accuracy**: don't pollute with unnecessary data



Software Prefetching

- Software prefetching: two kinds
  - **Binding**: prefetch into register (e.g., software pipelining)
    - +No ISA support needed, use normal loads (non-blocking cache)
    - –Need more registers, and what about faults?
  - **Non-binding**: prefetch into cache only
    - –Need ISA support: non-binding, non-faulting loads
    - +Simpler semantics
  - Example

```
for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j+=BLOCK_SIZE) {
      prefetch(&X[i][j]+BLOCK_SIZE);
      for (jj=j; jj<j+BLOCK_SIZE-1; jj++)
         sum += x[i][jj];
   }
```



Stream-prefetching [ISCA 2019,Wang/Nowatzki]

- Idea: "Semi-binding" prefetch a pattern specified at compile time
  - Configure a pattern at the head of loop, data is buffered in a FIFO
  - Accessed through pseudo-registers, which are renamed into FIFO positions
  - Software figures out pattern, hardware figures out when to prefetch (prefetch distance)

```
for i = 0 to N
   if (…)
      … = a[i]
   else
      … // don't use
a[i]
```

```
stream P = a[0:N]
for i = 0 to N
   if (…)
      … = P
   else
      … // don't use
a[i]
step P
```

Instruction to configure pattern for pseudoregister P

- Step Instruction: Increments the arch view of where the stream is
  - Decoupling step & usage of FIFO values enables conditional stream use
- Semi-binding: Only take exceptions when you use values
- Mispeculation: Walk backwards through steps to work out stream position
- LSQ is extended to track prefetched values to detect aliasing



Hardware Prefetching

- What to prefetch?
  - One block ahead
    - How much latency do we need to hide (Little's Law)?
    - Can also do N blocks ahead to hide more latency
    - +Simple, works for sequential things: insns, array data
  - **Address-prediction**
    - Needed for non-sequential data: lists, trees, etc.

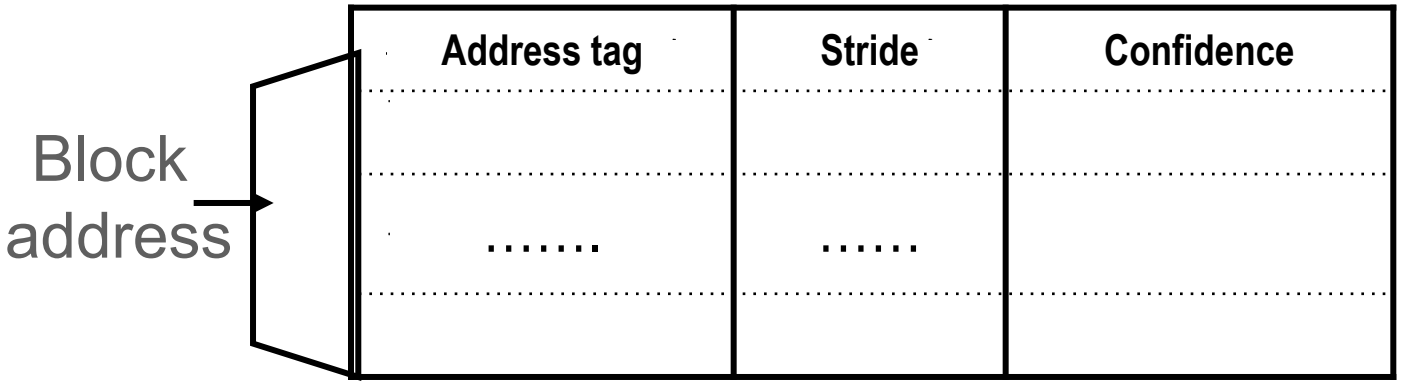
- When to prefetch?
  - On every reference?
  - On every miss? (typical)
    - +Interesting: similar to increasing block size
    - +Works better – don't always have to prefetch
  - One approach: wait until resident block becomes dead (avoid useful evictions)
    - –How to know when that is? ["Dead-Block Prediction", ISCA'01]
      - – Use the Program context to predict when addr. is last used



Address Based Stride Prefetching

- Match on some bits of the address
  - If address matches and you don't have a stride, compute the stride
  - If the address and stride match, increase the confidence and do a prefetch
  - Address: N * stride + miss address
- If no match, add a new entry with empty stride/confidence

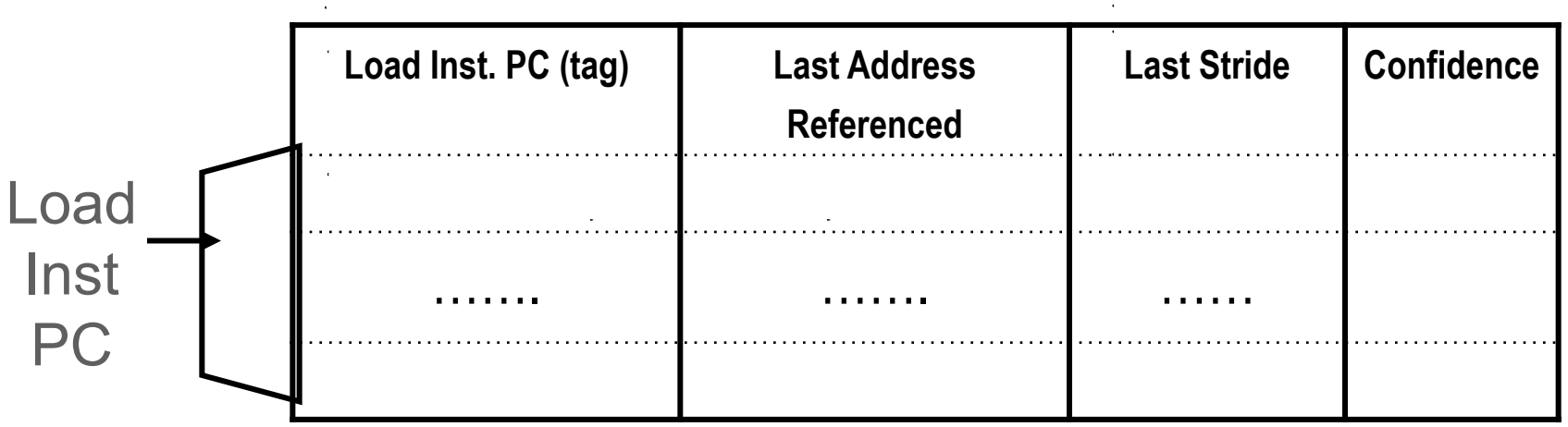
- Any possible problems?





Instruction Based Stride Prefetching

- Keep a cache of PC to stride-match table (using the PC is called **localization**)
  - For a miss, see if an entry exists in the table
  - If no match in cache, record PC + address referenced
  - If match but no stride, then compute the stride based on the current address
- If match and stride: (predictor is ready)
  - Check if stride is correctly predicted – if yes, increase confidence, otherwise decrease
  - If confidence is high enough, If match and stride prefetch N * stride + miss address
- Need to dynamically adjust N to get a timely prefetch





Irregular Programs

- What about irregular programs?
  (those with "ugly" access patterns)

- Insight: just like branch prediction, maybe prior *addresses* can be used to predict future addresses!

- Simple case: doing the same set of accesses over and over.



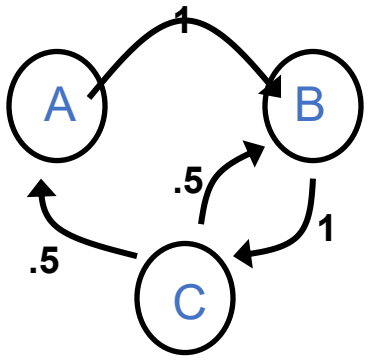
Correlating Prefetchers

- **Correlating Prefetchers**
  - Large table stores (miss-addr → next-miss-addr) pairs
  - On miss, access table to find out what will miss next
    - It's OK for this table to be large and slow
- Example: **Markov Prefetching** (Joseph and Grunwald - ISCA '97)
  - Uses global memory addresses as states in the Markov graph
  - Correlation Table *approximates* Markov graph

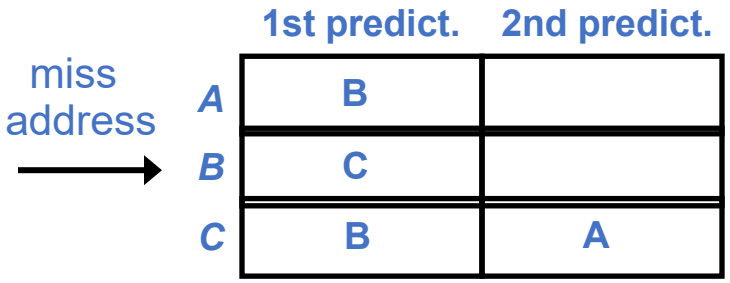
Miss Address Stream

A B C A B C B C . . .

Markov Graph

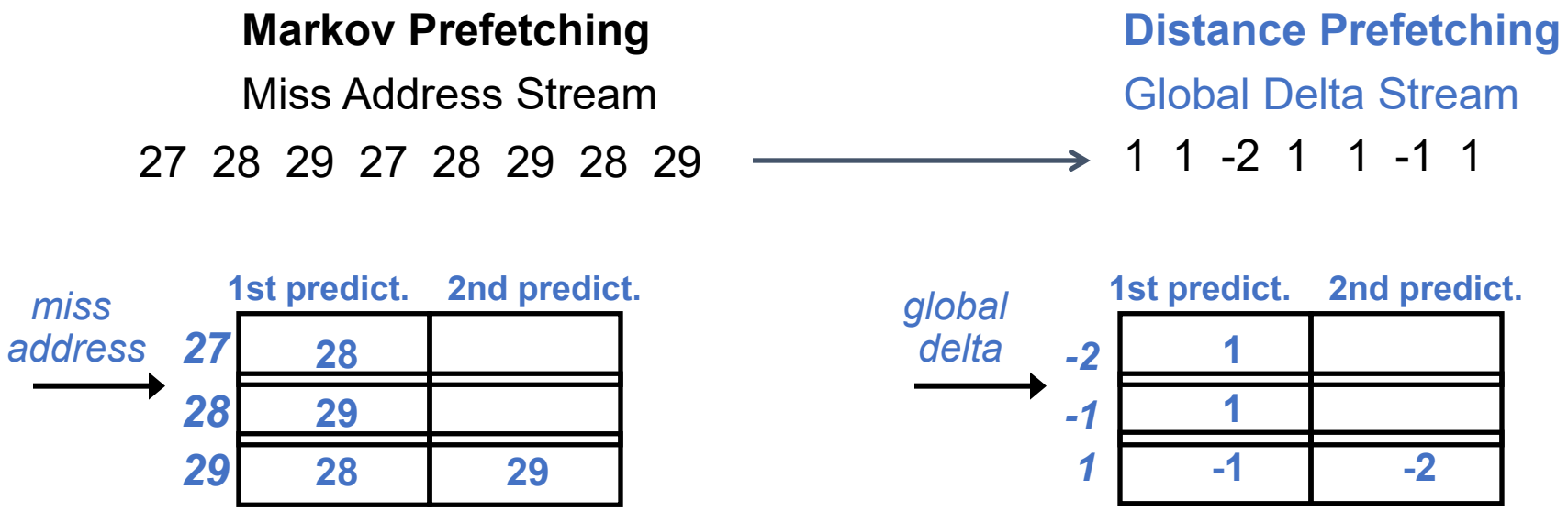


Correlation Table



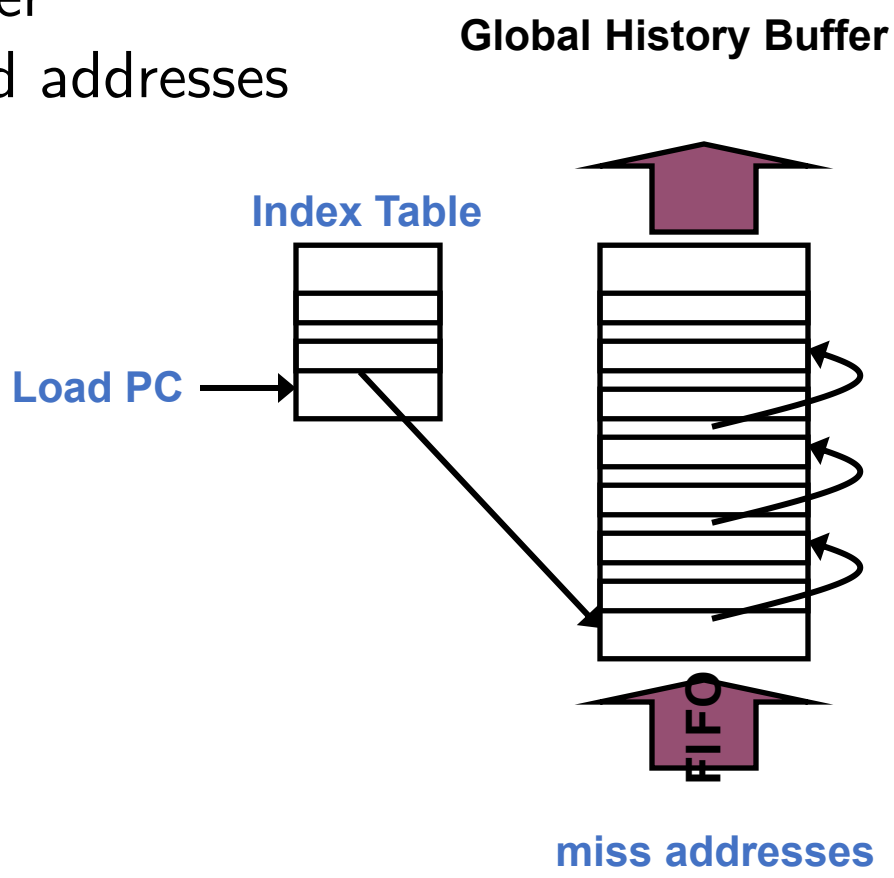
Generalizing and Compacting!

- Distance Prefetching forms *delta* correlations
  - Kandiraju and Sivasubramaniam (ISCA '02)
- Delta-based prefetching leads to much smaller table than "classical" Markov Prefetching
- Delta-based prefetching can remove compulsory misses



Global History Buffer (GHB)

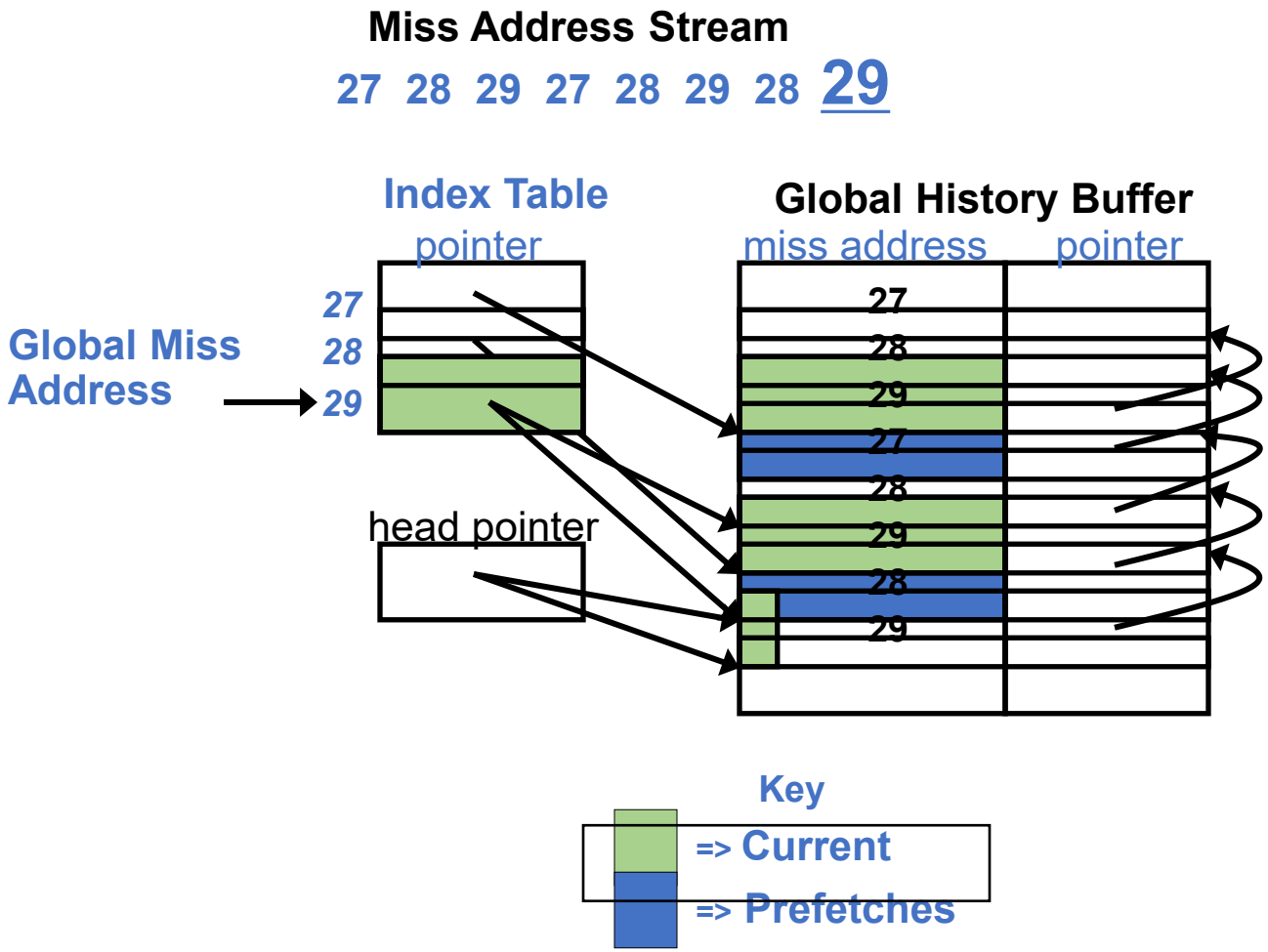
- Holds miss address history in FIFO order
- Linked lists within GHB connect related addresses
  - Same static load
  - Same global miss address
  - Same global delta



- Linked list walk is short compared with L2 miss latency



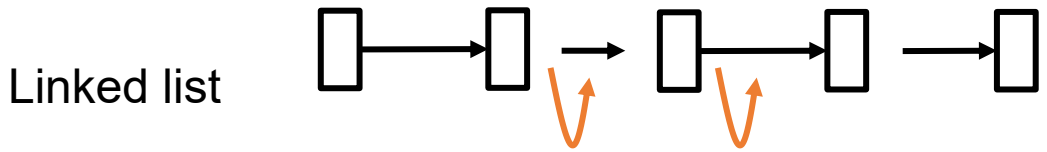
GHB - Example


Miss Address Stream
27 28 29 27 28 29 28 29
Index Table
pointer
Global History Buffer
miss address
pointer
Global Miss Address
27
28
29
head pointer
27
28
29
27
28
29
28
29
Key
=> Current
=> Prefetches



Other Fun Prefetching Ideas

- Content-directed or dependence-based prefetching
  - Greedily chases pointers from fetched blocks
- Jump pointers
  - Augment data structure with prefetch pointers



- Too many different prefetchers?
  - Set deuling: apply two different policies to different sets, see which one wins!
    - Why it works: typically uniform behavior across sets
    - Can be used for lots of cache policies actually (prefetching, replacement policies, etc.)



Making misses vanish...
Data layout transformations



Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
  - Several code restructuring techniques to improve both
  - –Compiler must know that restructuring preserves semantics

- **Loop interchange**: spatial locality
  - Example: row-major matrix: `X[i][j]` followed by `X[i][j+1]`
  - Poor code: `X[i][j]` followed by `X[i+1][j]`

```
for (j = 0; j<NCOLS; j++)
  for (i = 0; i<NROWS; i++)
    sum += X[i][j];    // non-contiguous accesses
```

  - Better code

```
for (i = 0; i<NROWS; i++)
  for (j = 0; j<NCOLS; j++)
    sum += X[i][j];    // contiguous accesses
```

When is this particularly bad for cache misses?



Software Restructuring: Data

- **Loop blocking**: Shrink working set for free
  - Increase temporal locality
  - Poor code

```
for (k=0; k<NITERATIONS; k++)
    for (i=0; i<NELEMS; i++)
        sum += X[i];     // say X is bigger than the cache!
```

  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

```
for (i=0; i<NELEMS; i+=CACHE_SIZE)
    for (k=0; k<NITERATIONS; k++)
        for (ii=0; ii<i+CACHE_SIZE-1; ii++)
            sum += X[ii];
```

  - Assumes you know **CACHE_SIZE**, do you?
  - Loop fusion: similar, but for multiple consecutive loops



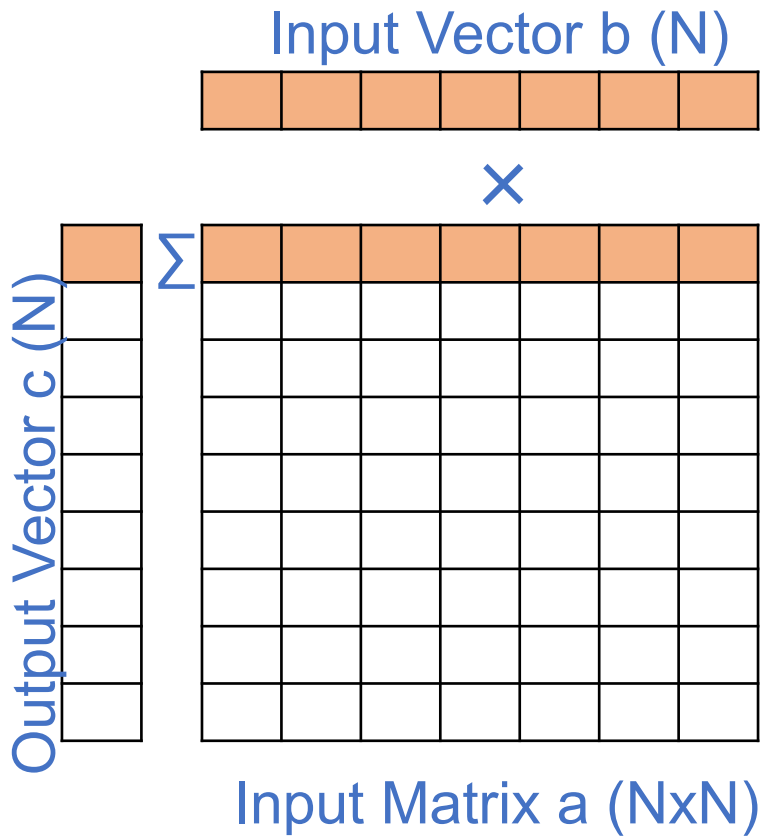
Matrix-Vector Multiply (loop ordering)

```
double sum,c[N],a[N][N],b[N];
for(int j = 0; j < N; ++j)
  for(int i = 0; i < N; i++)
    c[i] += a[i][j] * b[j];
```

1/8 8/8 0/8

```
for(int i = 0; i < N; ++i)
  for(int j = 0; j < N; ++j)
    c[i] += a[i][j] * b[j];
```

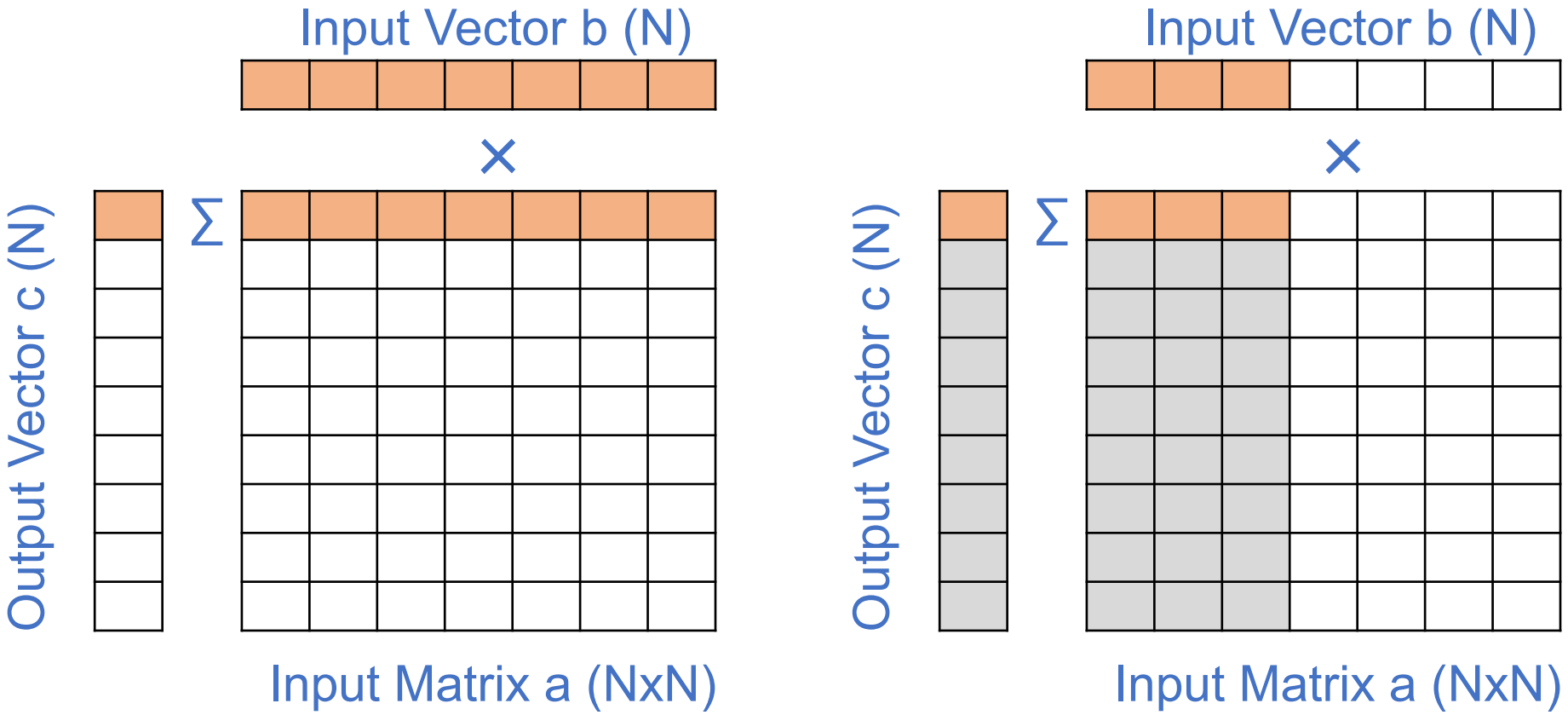
0/8 1/8 1/8



Assume 64-byte cache lines -- double is 8 bytes -- N * 8 >> cache size

What are the cache miss rates?

Matrix Blocking



Assume N >> cache size, 64-byte lines, … what is miss rate of each access?

```
c[i] += a[i][j] * b[j];
```

0/8 1/8 0/8

Half as many misses!



Blocked version in code

```
double sum, c[N], a[N][N], b[N];
#define BLOCK (8) //should divide N evenly

for(j_block=0; j_block<N; j_block+=BLOCK){
  for (i = 0; i < N; i++) {
    sum = c[i]; //get partial sum
    for (j=j_block; j<j_block+BLOCK;j++){
      sum += a[i][j] * b[j];
    }
    c[i] = sum;
  }
}
```



Restructuring Loops

- Loop Fusion
  - Merge two independent loops
  - Increase reuse of data
  - In ML, fusion is kindof a big deal …
    - E.g. fuse mat-mul with elementwise ops)
- Loop Fission
  - Split loop into independent loops
  - Reduce contention for cache resources

Fusion Example:

```
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    a[i][j] = 1/b[i][j]*c[i][j];
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    d[i][j] = a[i][j]+c[i][j];
```

Fused Loop:

```
for (i=0; i < N; i++)
  for (j=0; j < N ;j++) {
     a[i][j] = 1/b[i][j]*c[i][j];
     d[i][j] = a[i][j]+c[i][j];
}
```



Power Optimizations



Low-Power Caches

- Caches consume significant power
  - 15% in Pentium4
  - 45% in StrongARM
  - (hmm, less than the fraction of area they consume!)

- Three techniques
  - Way prediction (already talked about)
  - Dynamic resizing
  - Drowsy caches



Low-Power Access: Dynamic Resizing

- **Dynamic cache resizing**
  - Observation I: data, tag arrays implemented as many small arrays
  - Observation II: many programs don't fully utilize caches

  - Idea: dynamically turn off unused arrays
    - Turn off means disconnect power (V_{DD}) plane
    - +Helps with both dynamic and static power
  - There are always tradeoffs
    - –Flush dirty lines before powering down → costs power↑
    - –Cache-size↓ → $\%_{miss}$↑ → power↑ (dynamic – off-chip access)
      execution time↑ (static)



Dynamic Resizing: When to Resize

- Use $\%_{miss}$ feedback
  - $\%_{miss}$ near zero? Make cache smaller (if possible)
  - $\%_{miss}$ above some threshold? Make cache bigger (if possible)

- Aside: how to track miss-rate in hardware?
  - Maybe track miss-rate over some window?
  - Another scheme, track if $\%_{miss}$ higher than threshold (e.g. 5%)
    - Hit? counter −= 1
    - Miss? counter += 19
    - Counter positive? More than 1 miss per 19 hits ($\%_{miss} > 5\%$)
  - Maybe also want threshold the counter to allow for changing program phases...



Dynamic Resizing: How to Resize?

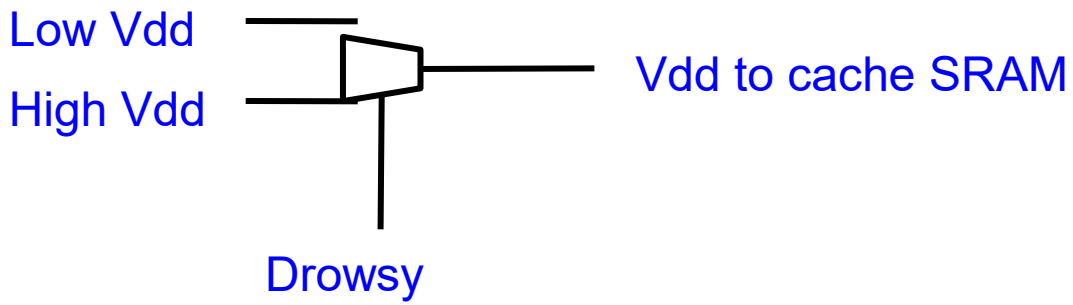
- **Reduce ways**
  - ["Selective Cache Ways", Albonesi, ISCA-98]
  - +Resizing doesn't change mapping of blocks to sets → simple
  - –Lose associativity
- **Reduce sets**
  - ["Resizable Cache Design", Yang+, HPCA-02]
  - –Resizing changes mapping of blocks to sets → tricky
    - When cache made bigger, need to relocate some blocks
    - Actually, just flush them
  - Why would anyone choose this way?
    - +More flexibility: number of ways typically small
    - +Lower $\%_{miss}$: for fixed capacity, higher associativity better



Drowsy Caches

- Circuit technique to reduce leakage power
  - Lower Vdd → Much lower leakage
  - But too low Vdd → Unreliable read/destructive read

- Key: Drowsy state (low Vdd) to hold value w/ low leakage
- Key: Wake up to normal state (high Vdd) to access
  - 1-3 cycle additional latency





Summary

- **Average access time** of a memory component
  - $latency_{avg} = latency_{hit} + \%_{miss} * latency_{miss}$
  - Hard to get low $latency_{hit}$ and $\%_{miss}$ in one structure → hierarchy
- **Memory hierarchy**
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- Cache ABCs (**capacity, associativity, block size**)
  - 3C miss model: compulsory, capacity, conflict
- **Performance optimizations**
  - $\%_{miss}$: victim buffer, prefetching, skew-assoc., zcache
  - $latency_{miss}$: critical-word-first/early-restart, lockup-free design
- **Power optimizations**: way prediction, dynamic resizing
- **Write issues**
  - Write-back vs. write-through/write-allocate vs. write-no-allocate



Simplifying Assumptions thus far

- Loads/Stores:
  - Fast if there's a cache hit
  - Slow, variable latency otherwise

- Addresses are "real" addresses:
  - Only one address space, and its physical
  - No need for translation
  - (real machines have virtual memory: each process pretends it has the whole address space)