CS/ECE 752:
Advanced Computer Architecture I

Professor Matthew D. Sinclair

Cache Architecture

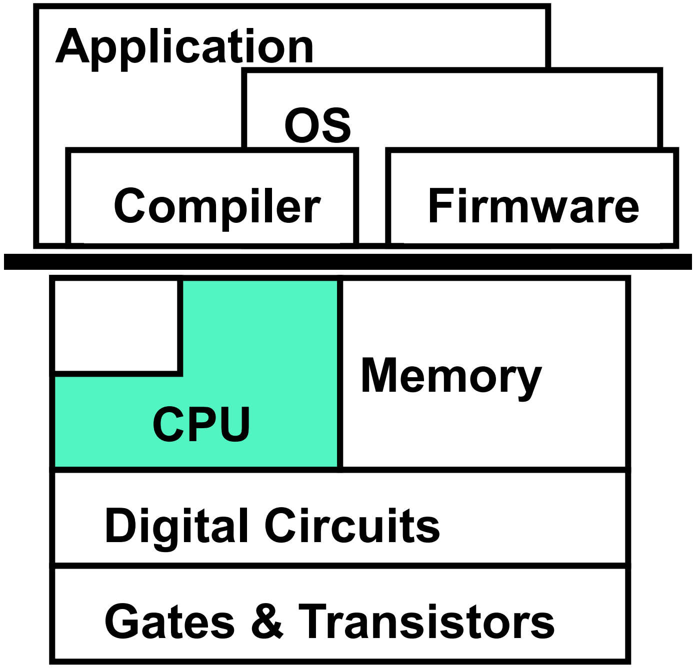
Slide History/Attribution Diagram:

Simplifying Assumptions thus far

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

- Addresses are "real" addresses:
  - Only one (physical) address space
  - 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

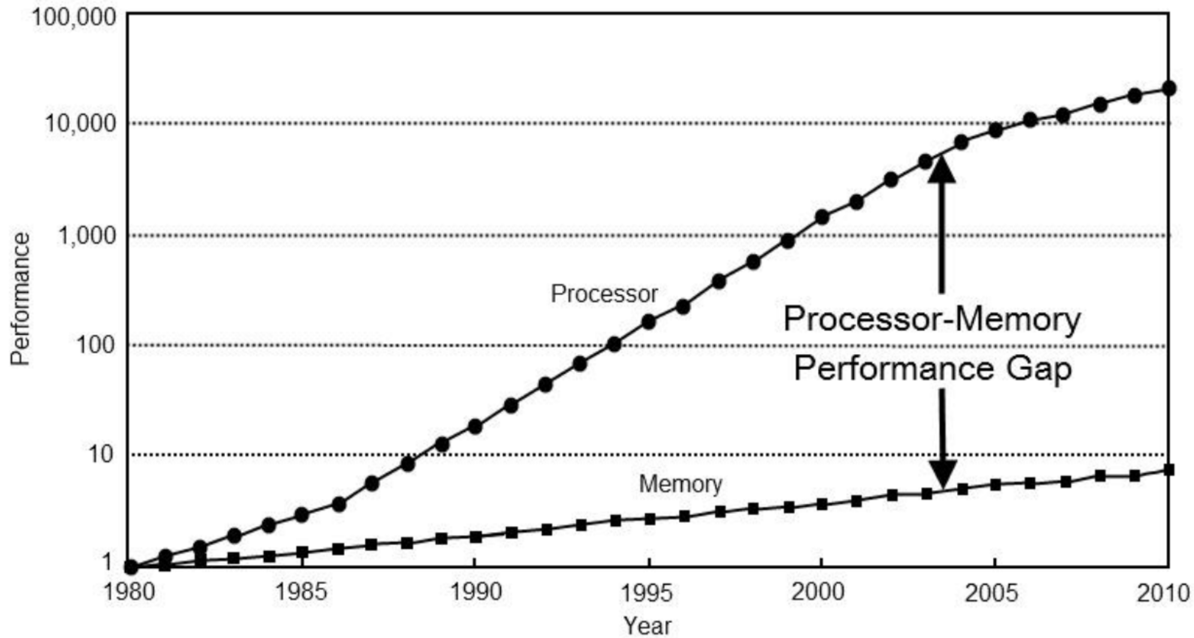
Motivation

- 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"



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

- Processors getting faster more quickly than memory (note: log scale)
  - Processor speed improvement: 35% to 55%
  - Memory latency improvement: 7%

Concepts from Previous Courses

- Determining Tag, Index, and Offset Bits
- Cache ABCs
  - Set Associativity
  - Block Size
  - Capacity
- Miss Rate Calculations (e.g., AMAT)
- Basic Cache Replacement Policies (e.g., LRU)
- Basic Write Policies (e.g., WB, WNA, WT)
- Identifying Memory Sizes through Microbenchmarking

- Our Focus: Optimizations Beyond These
  - See backup slides if you want to brush up on these

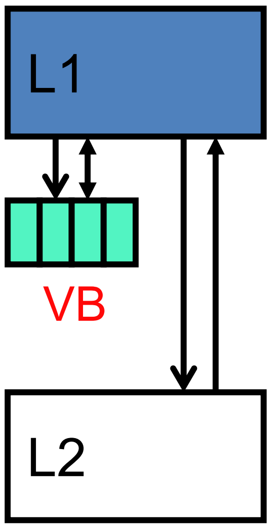
Beyond the ABCs: Avoiding Conflicts

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)*

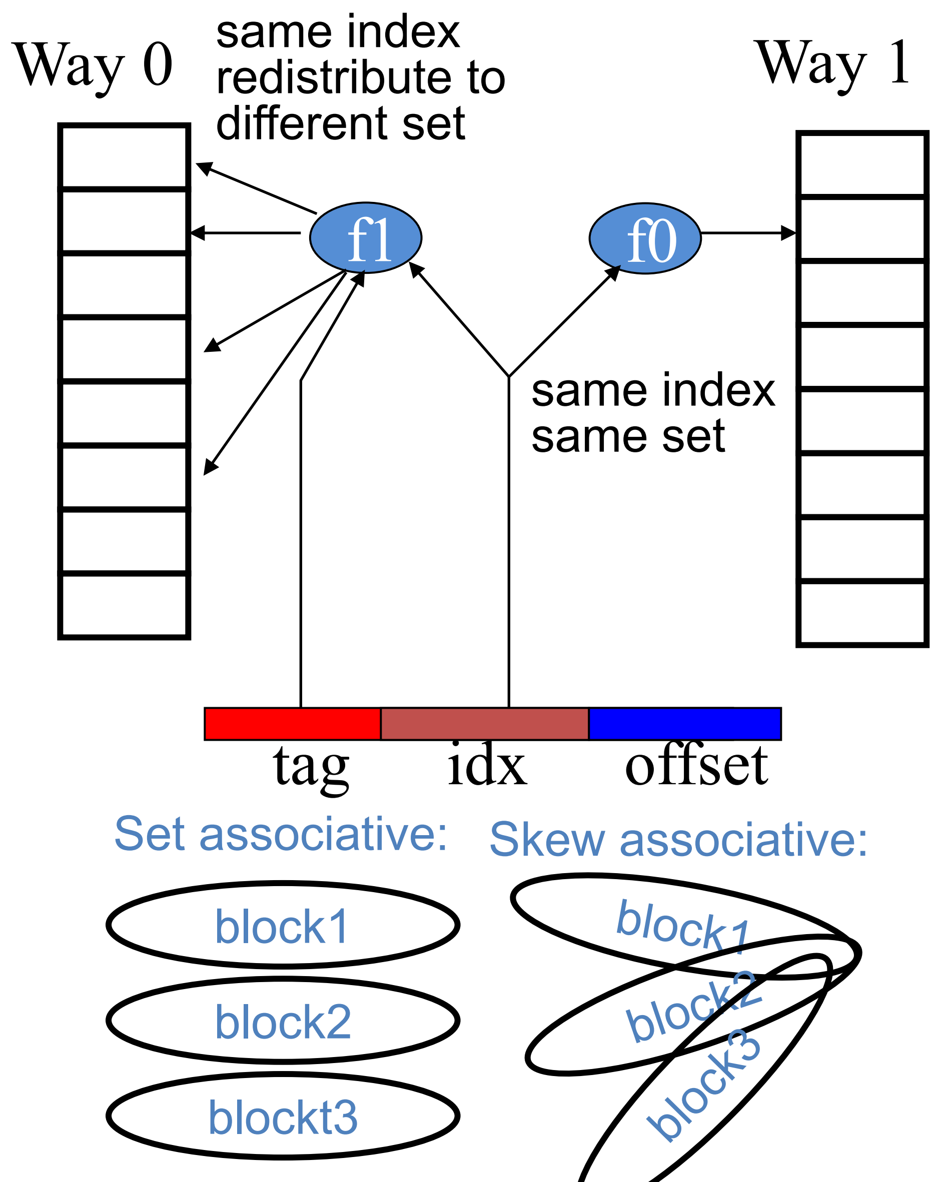
- **Victim buffer (VB)**: small fully-associative cache
  - Sits on L1 fill path
    - Blocks kicked out of L1 placed in VB
    - On miss, check VB: hit? Place block back in L1
  - Intuitively: 8 extra ways, shared among all sets
    - + Only a few sets will need it at any given time

  - Does VB reduce **$\%_{miss}$** or **$latency_{miss}$**?

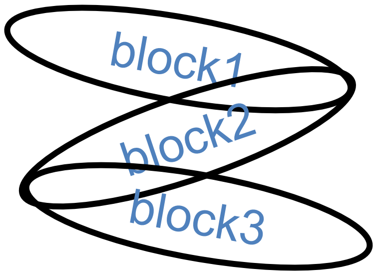


Seznec’s Skewed-Associative Cache [1993]

- 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

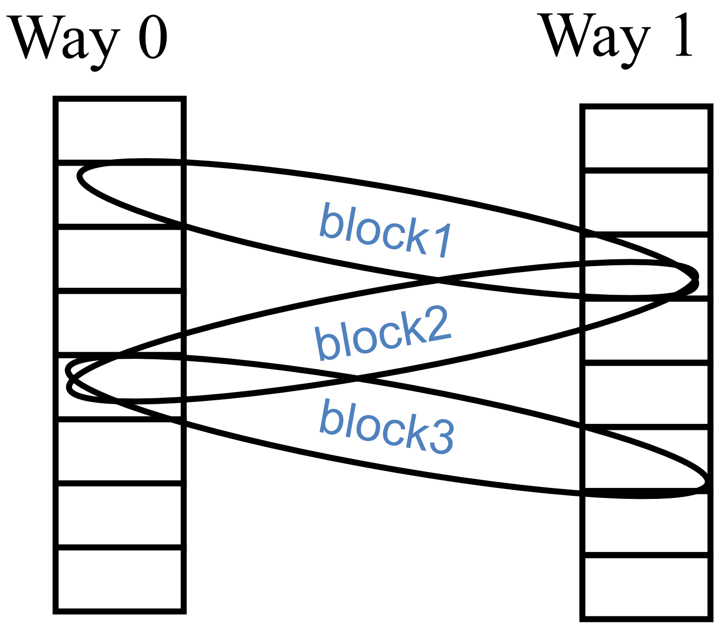


Sanchez Zcache [Sanchez Micro10]



- 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, move to some other way to find a less recently used block
- Decoupled ways and associativity
- Insight:
  - "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."

Overly Simplified Example



Beyond the ABCs:
Evicting the “Right” Stuff

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
  - I.e., would not miss in a fully-associative cache

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

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 because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? (assume you must classify each block)
    - 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 cache associativity is too low
    - Identify? **All other misses**

  - Who cares? Different techniques for attacking different misses

Large Blocks and Subblocking – Sectors

- 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?
  - Save tag space for regular access
  - Latency of fill may be long – could we service some sublocks sooner than others?

v	subblock	v	subblock	• • • •	v	subblock	tag

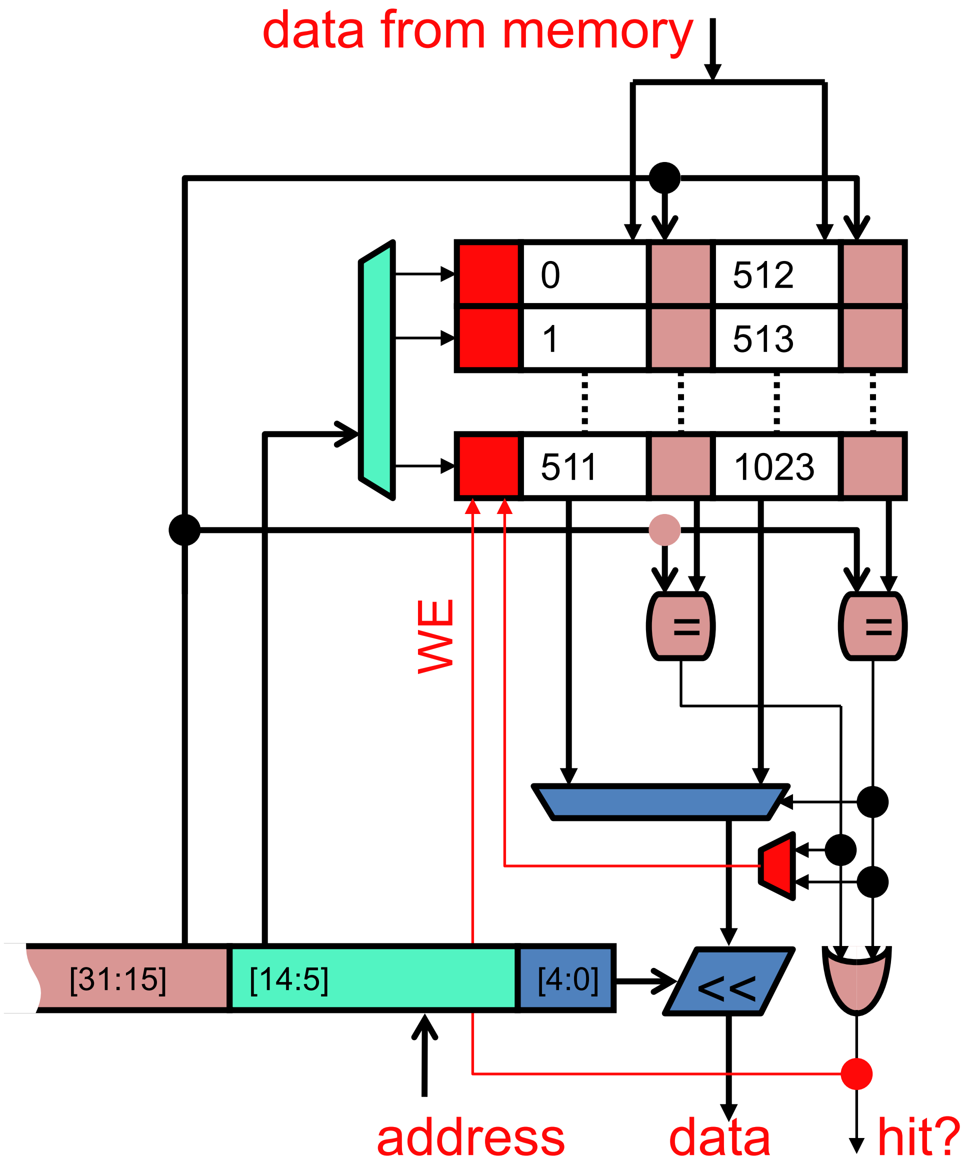
Replacement Policies

- 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)**
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU = least likely to be used in future
  - **NMRU (not most recently used)**
    - An easier to implement approximation of LRU
    - Is LRU for 2-way set-associative caches
  - **Tree-PLRU**
    - Approximate **LRU** with counter tree
  - Unachievable optimum?
    - **Belady's**: replace block that will be used furthest in future

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



Fancier Modern Replacement Policies

- What if apps have complex access patterns?
  - Reuse is far part (distance "re-reference" interval)
  - Large working sets > cache size → can cause thrashing
  - Streaming ("scans") phases → Don't want to cache these
  - And what if applications have a mix of these patterns?
  - Ideally, want **realistic** cache replacement policy that handles all

- Sidenote: can have "tournament" cache replacement policies like in branch predictor ("set dueling")

Fancier Modern Replacement Policies

- Re-Reference Interval Prediction [Jaleel ISCA '10]
  - Key Idea: Track how time to "re-reference" an entry in the cache
  - Insight: LRU is always evicting "near immediate" re-references
  - Solution:
    - n-bit counter/cache line to track/predict reuse
    - Example: 1 bit (NRU, approximates LRU/NMRU)
      - 0: "near immediate" reuse
      - 1: "distant" reuse
    - On evict, NRU favors all "1" entries + sets all NRU bits to 1
    - On insertion and cache hit, NRU predicts/sets to "0"
  - Issues:
    - How to tell multiple entries with same NRU value apart? → NRU picks a static cache location to start from
    - Static policy can't distinguish scans from reuse in mixed apps

Fancier Modern Replacement Policies

- Static RRIP (SRRIP)
  - Need: more than 1-bit counters – finer granularity predictions
    - Helps differentiate near-term (~0) vs. scans vs. distant (2^M-1)
  - Insertion: assume **long** reuse (2^M-2 where M is max counter value)
    - Prevents **distant** reuse cache blocks from polluting cache
    - Why **long** vs. **distant**? Give block a chance to be "promoted" to more likely reuse prediction (favor evicting distant)
  - Cache Hit (Hit Promotion Policy):
    - Update RRIP counter to be smaller value (nearer term)
  - Cache Miss (Victim Selection Policy):
    - Search for first location with largest RRIP value – evict it
    - Repeat for next largest RRIP value(s) if needed
    - If didn't find max value (distant), increment all counter values
  - Re-reference predictions **statically** determined by hits and misses
  - Scan resistant if counters big enough for working set

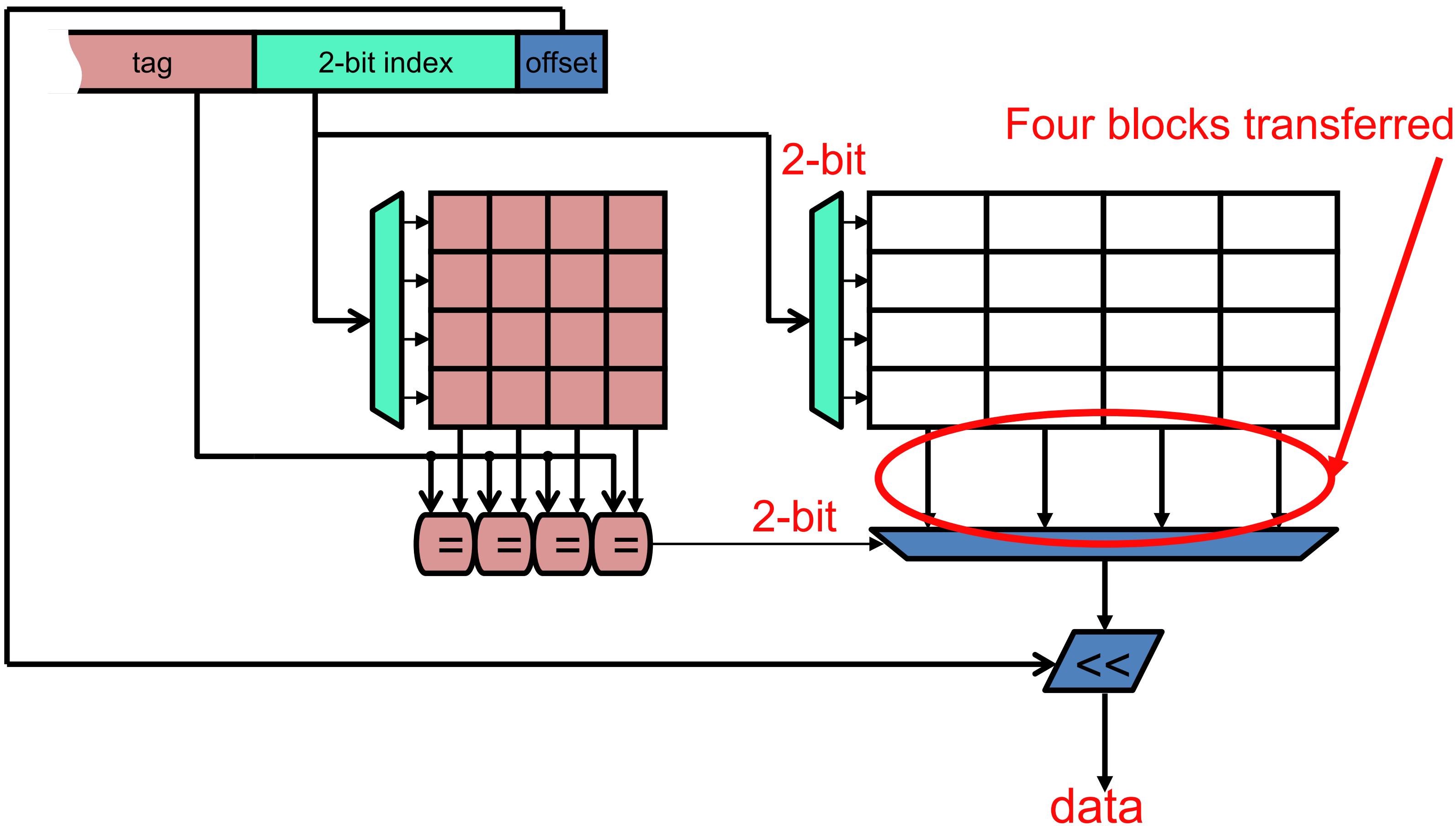
Fancier Modern Replacement Policies

- Bimodal RRIP (BRRIP)
  - Insight: SRRIP is bad if working set is larger than M → thrashing, few hits
  - Solution: on insertion:
    - **Most** blocks get **distant** re-reference interval (2^M-1)
    - Infrequently block gets **long** re-reference interval (2^M-2)
    - Helps preserve some of working set, reduces thrashing
    - But if no thrashing, BRRIP may hurt performance
- Dynamic RRIP (DRRIP) – "tournament RRIP"/"set dueling"
  - Decide if BRRIP or SRRIP is best for a given cache line
  - Add a set dueling monitor to decide which is best for a given line
    - Permanently dedicate a few sets to each policy to help train

Beyond the ABCs:
Lower Power Tag Access

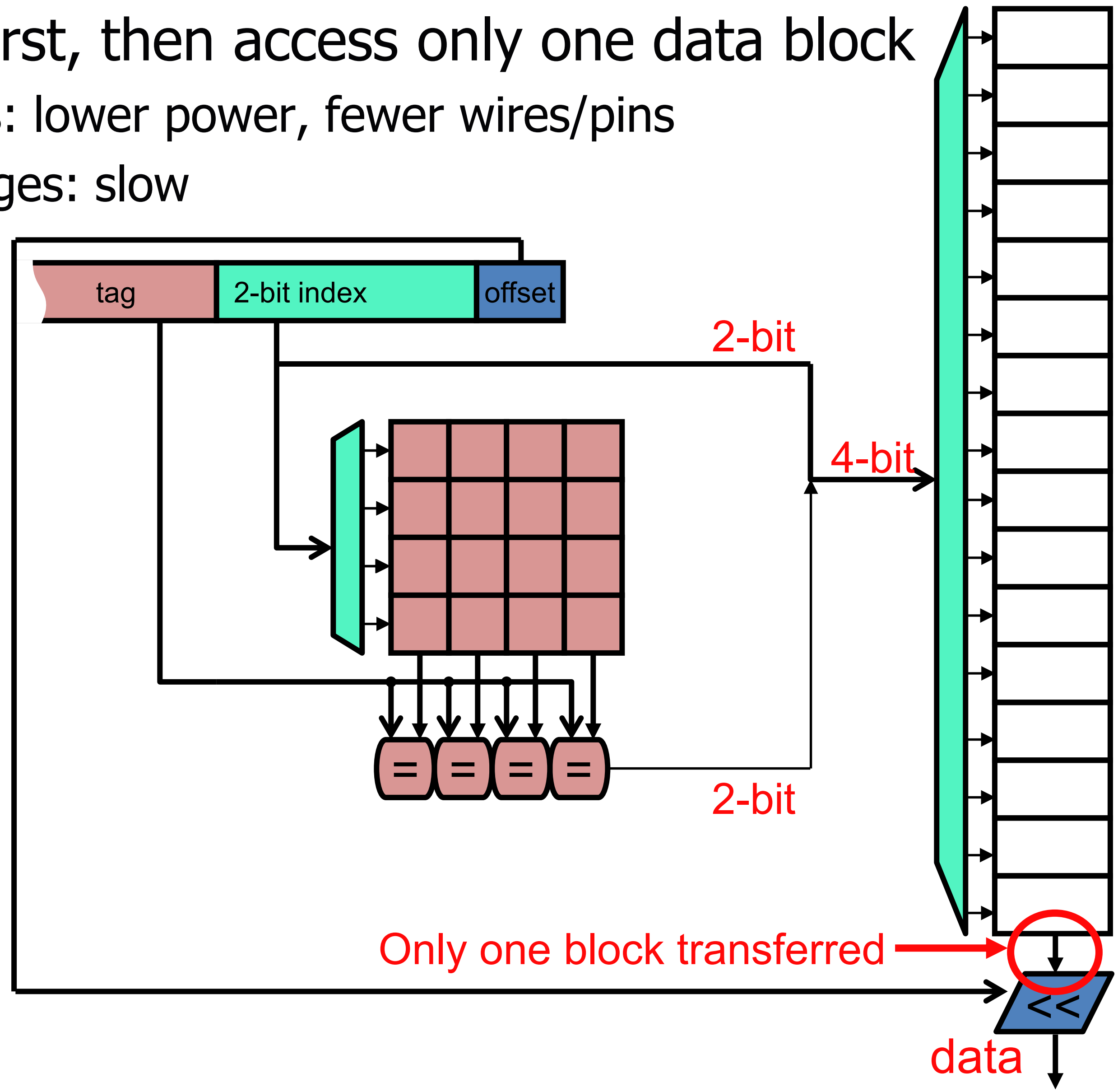
Parallel or Serial Tag Access?

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



Serial Tag Access

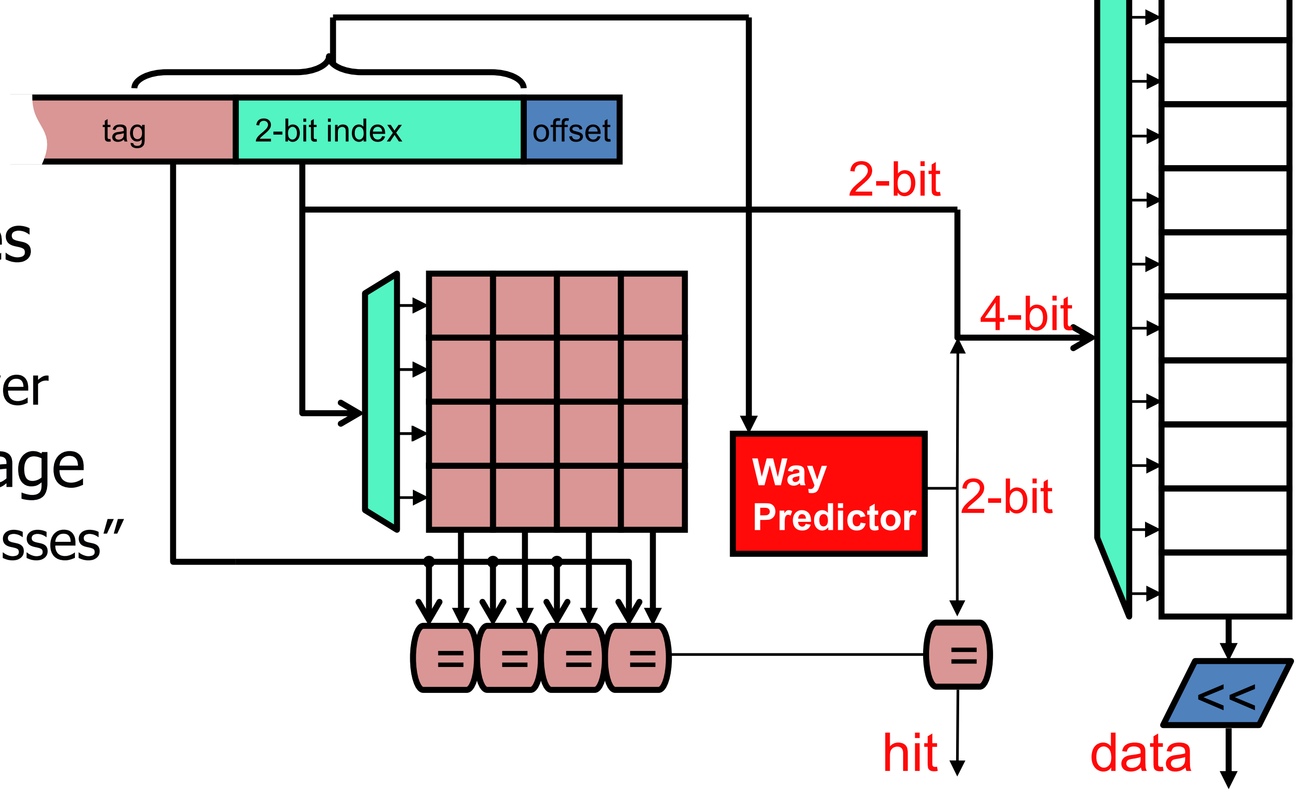
- Tag match first, then access only one data block
  - Advantages: lower power, fewer wires/pins
  - Disadvantages: slow



Question – get best of both worlds?
(high associativity + low hit time)

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"

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
- Beyond ABCs
  - Advanced Conflict Avoidance
  - Data layout transformations
  - Prefetching (hardware/software)
  - Reducing Miss Cost
- Low power techniques
- Simple models

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
```

Software Restructuring: Data

- **Loop blocking**: temporal locality
  - Poor code

```
for (k=0; k<NITERATIONS; k++)
    for (i=0; i<NELEMS; i++)
        sum += X[i];    // say
```

  - 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=i; ii<i+CACHE_SIZE-1; ii++)
            sum += X[ii];
```

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

Restructuring Loops

- Loop Fusion
  - Merge two independent loops
  - Increase reuse of data
- Loop Fission
  - Split loop into independent loops
  - Reduce contention for cache resources

Fusion Example:

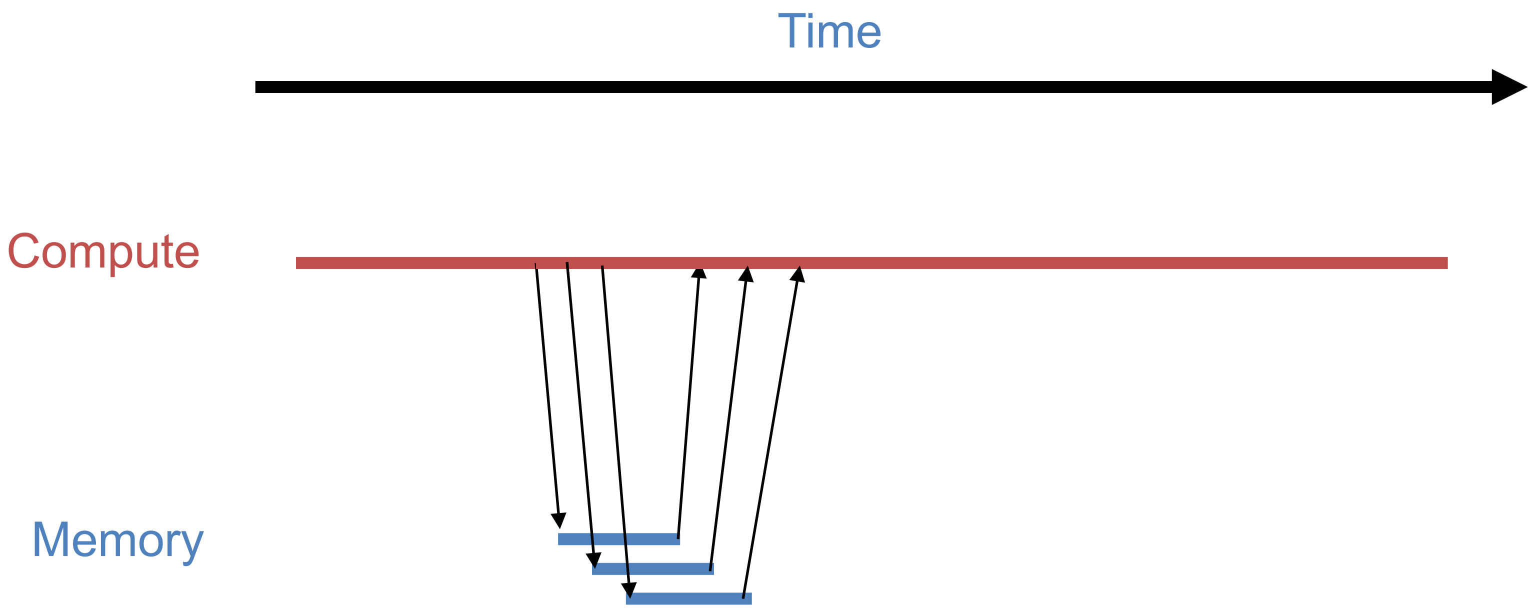
```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        a[i][j] = 1/b[i][j]*c[i][j];
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        d[i][j] = a[i][j]+c[i][j];
```

Fused Loop:

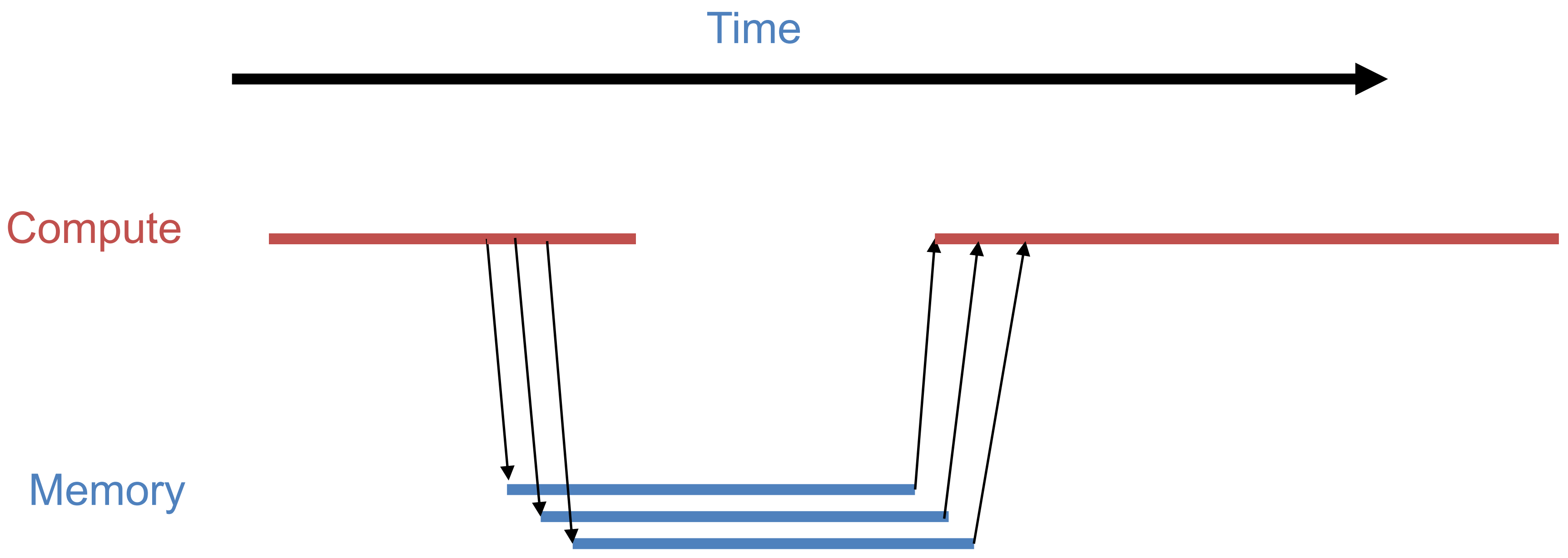
```
for (i=0; i < N; i++)
    for (j=0; j < N ;j++)
    {
        a[i][j] = 1/b[i][j]*c[i][j];
        d[i][j] = a[i][j]+c[i][j];
    }
```

Making misses vanish…
Prefetching data before even requested

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

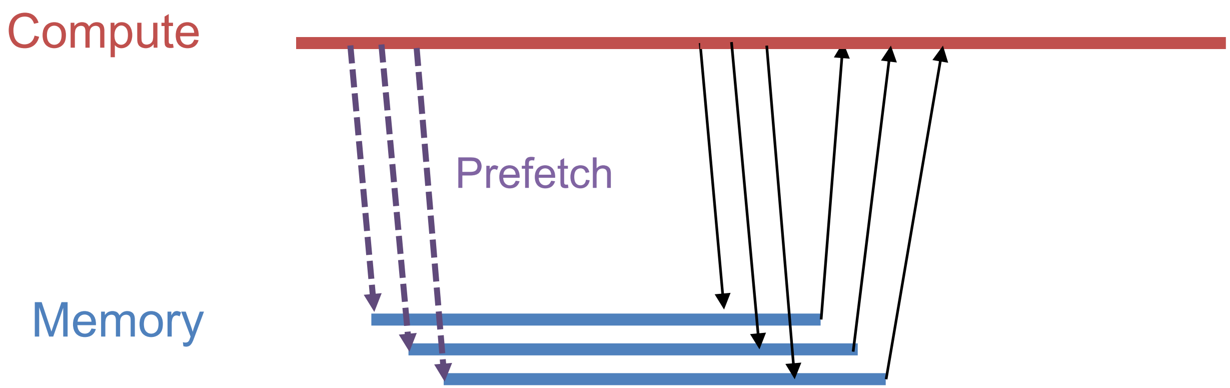


Bad case: Misses can't be hidden



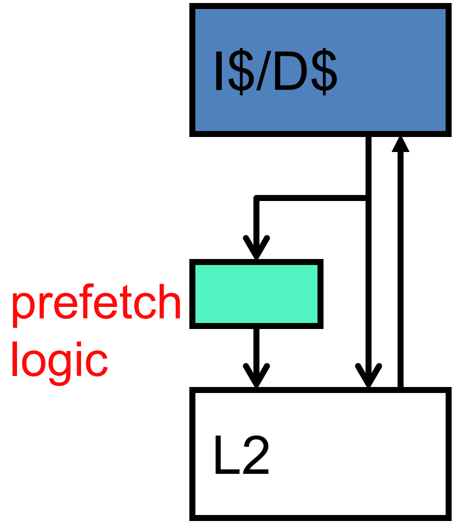
New Idea: Prefetching

- 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
  - It evicts useful 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];
    }
```

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.

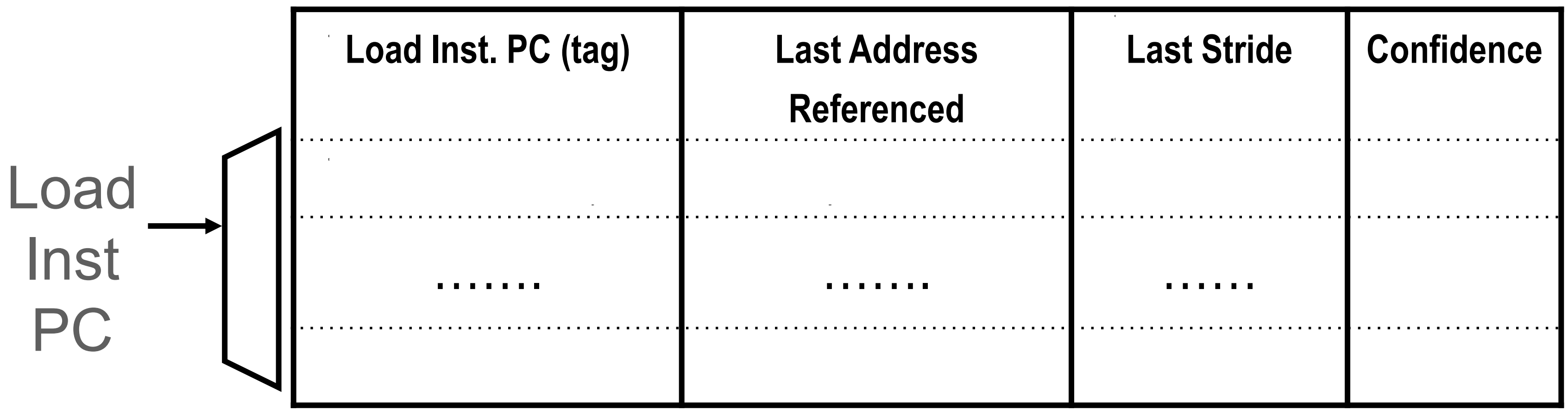
- When to prefetch?
  - On every reference?
  - On every miss?

    \+ 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)

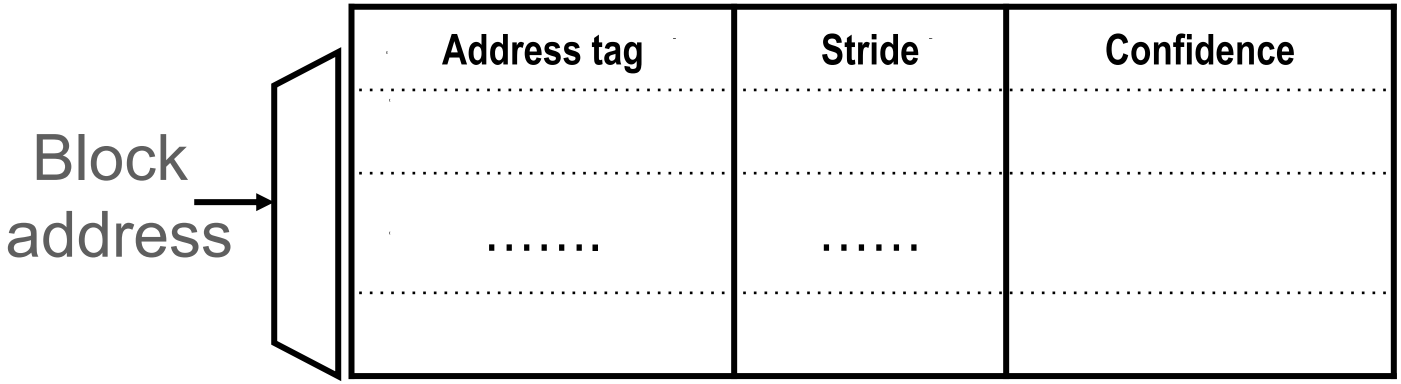
    – How to know when that is? ["Dead-Block Prediction", ISCA'01]

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

Can also just use block address



- Match on some bits of the address, works basically the same way
- What's the possible disadvantage?

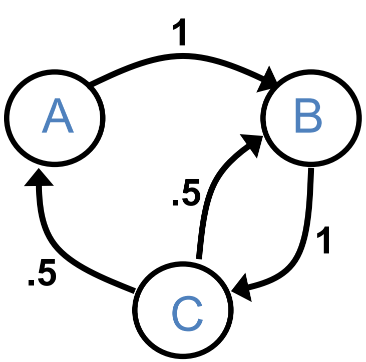
Correlating Prefetchers

- **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

miss address →

	1st predict.	2nd predict.
A	B	
B	C	
C	B	A

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

Markov Prefetching

Miss Address Stream

27 28 29 27 28 29 28 29

miss address

	1st predict.	2nd predict.
27	**28**	
28	**29**	
29	**28**	**29**

Distance Prefetching

Global Delta Stream

1 1 -2 1 1 -1 1

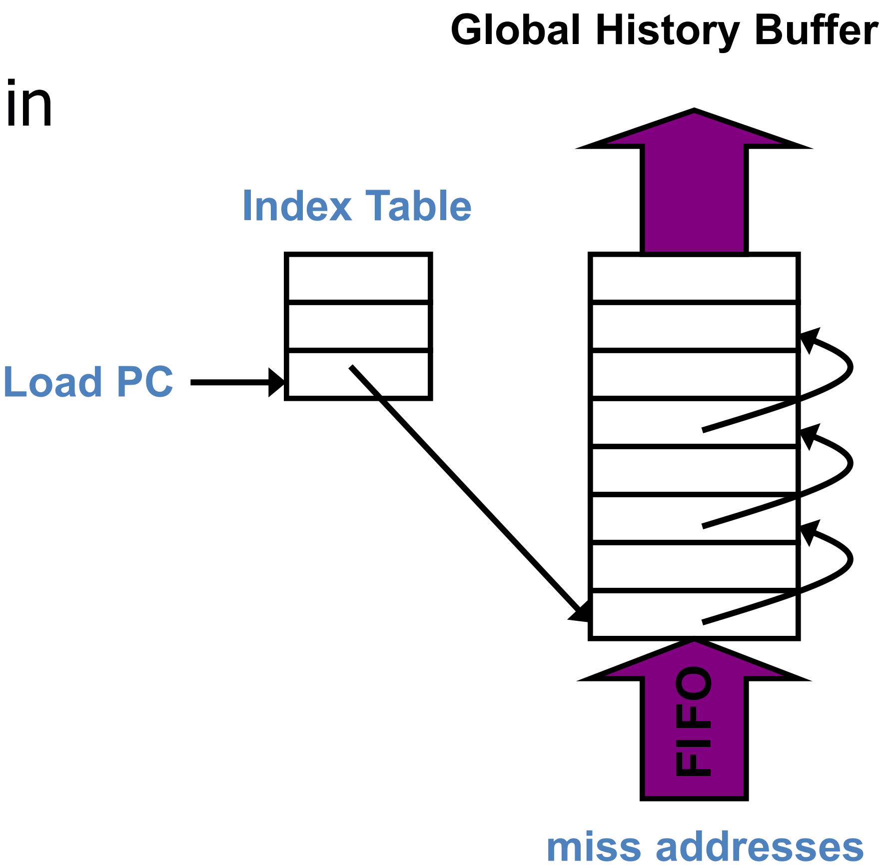
global delta

	1st predict.	2nd predict.
-2	**1**	
-1	**1**	
1	**-1**	**-2**

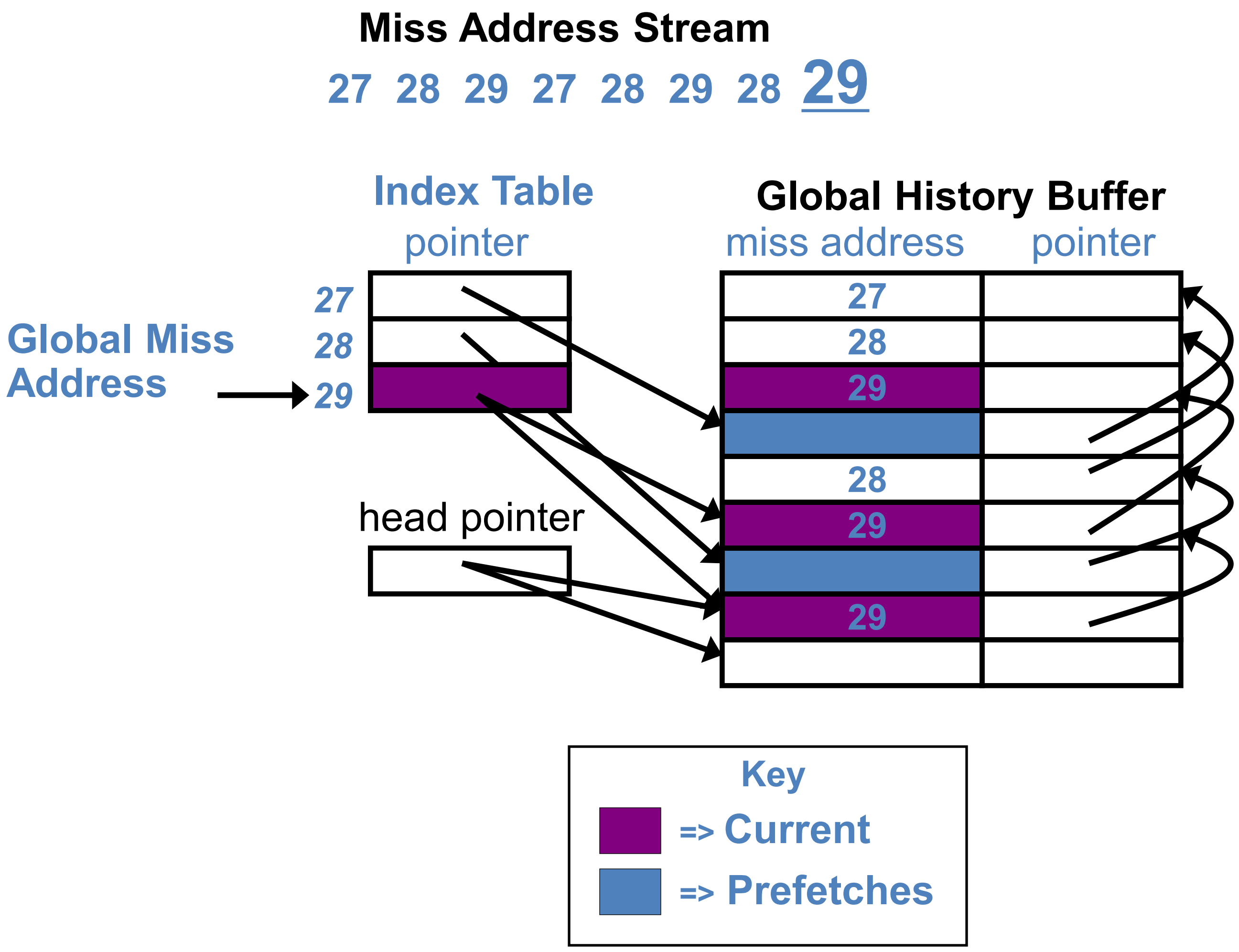
Global History Buffer (GHB)

- 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



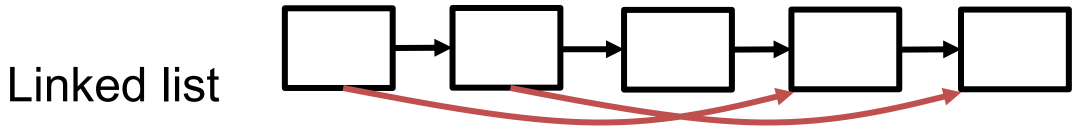
GHB - Example



Miss Address Stream
27 28 29 27 28 29 28 29
Index Table
pointer
Global History Buffer
miss address
pointer
27
28
29
Global Miss Address
27
28
29
28
29
29
head pointer
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



- An active area of research

What about miss cost?

Miss Cost: Lockup Free Cache (Non-Blocking)

- **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
  - Simplest scenario: "hit under miss"
    - Handle hits while miss is pending
    - Easy for OoO cores
  - More common: "miss under miss"
    - A little trickier, but common anyway
    - Requires split-transaction bus/interconnect
    - Requires multiple MSHRs: search to avoid frame conflicts

Miss Cost: Critical Word First/Early Restart

- Observation: Bus between levels of the memory hierarchy are typically 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 immediately
    - 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)
  - $\text{latency}_{\text{miss}} = \text{latency}_{\text{access}}$

So what about writes?

Write Issues

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

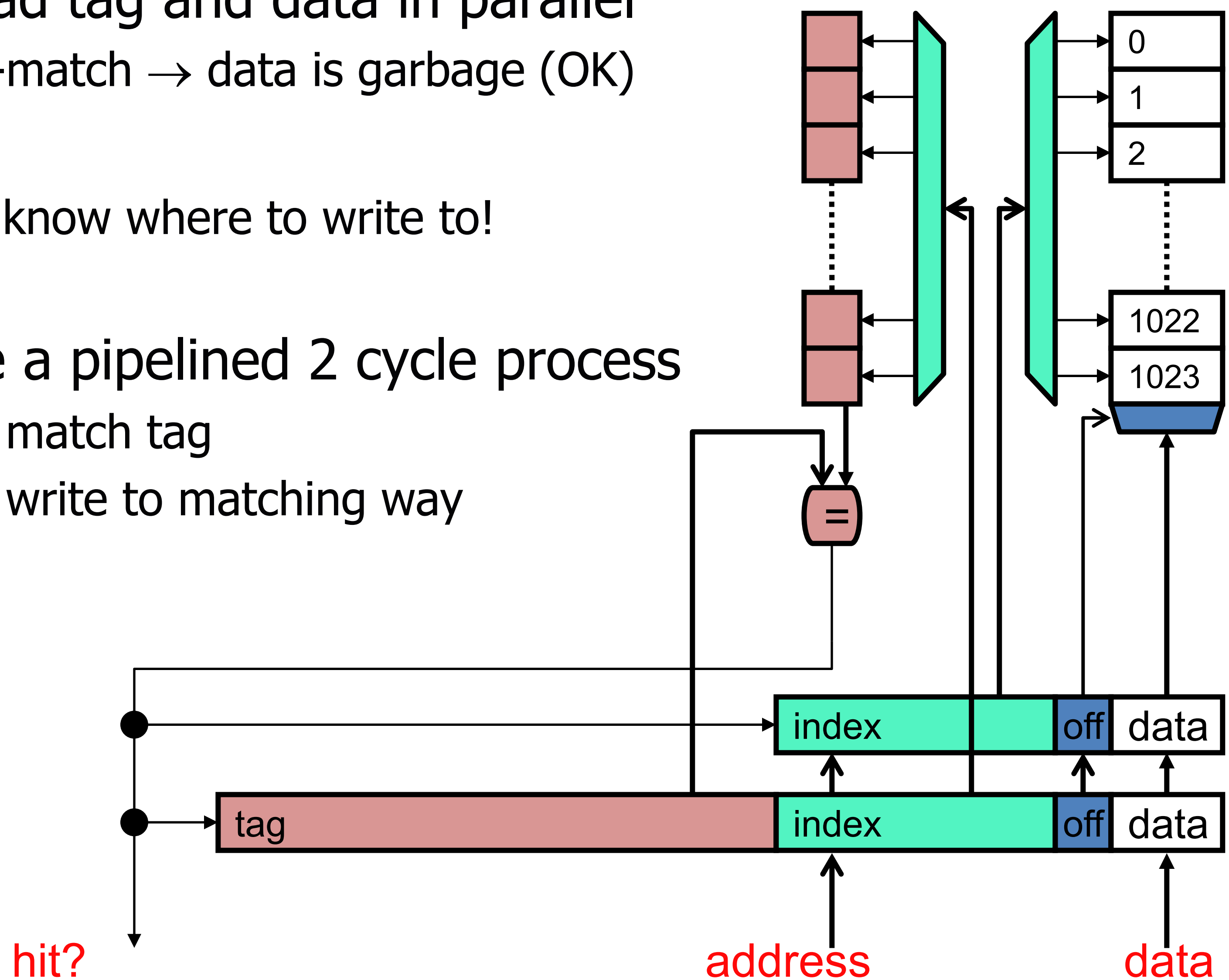
- Few new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate

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

Tag/Data Access

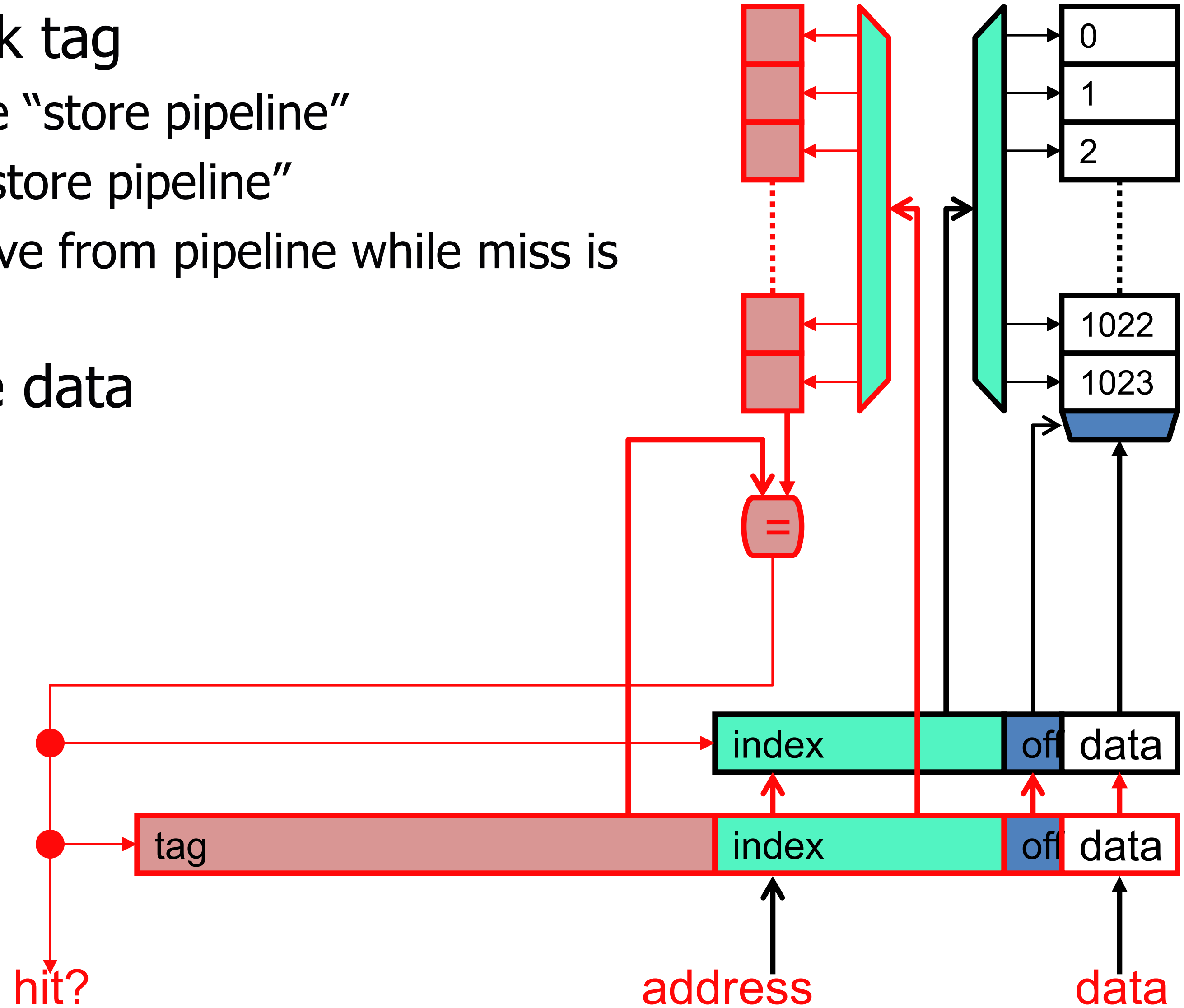
- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK)
- Writes:
  - Need to know where to write to!

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



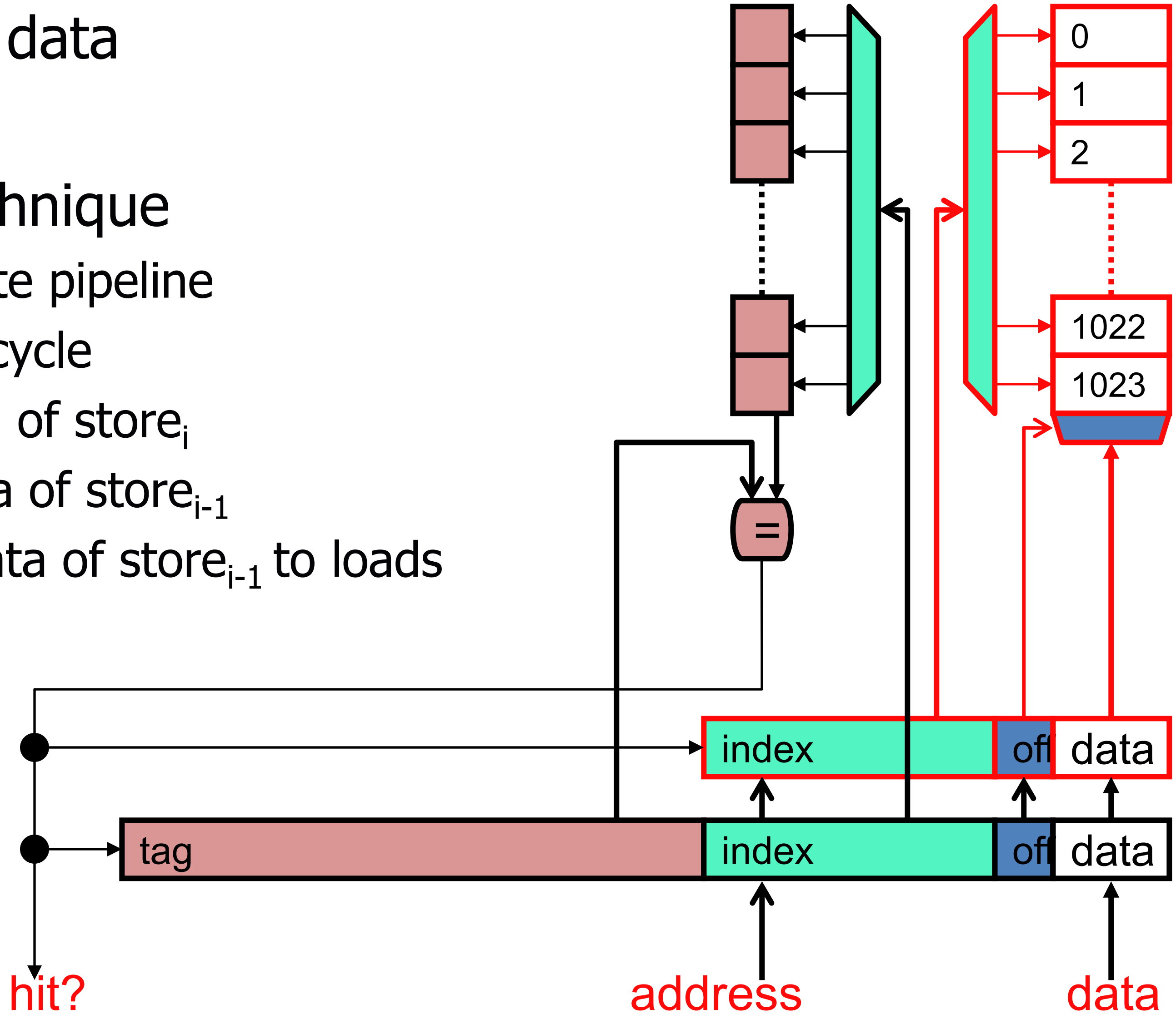
Tag/Data Access

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

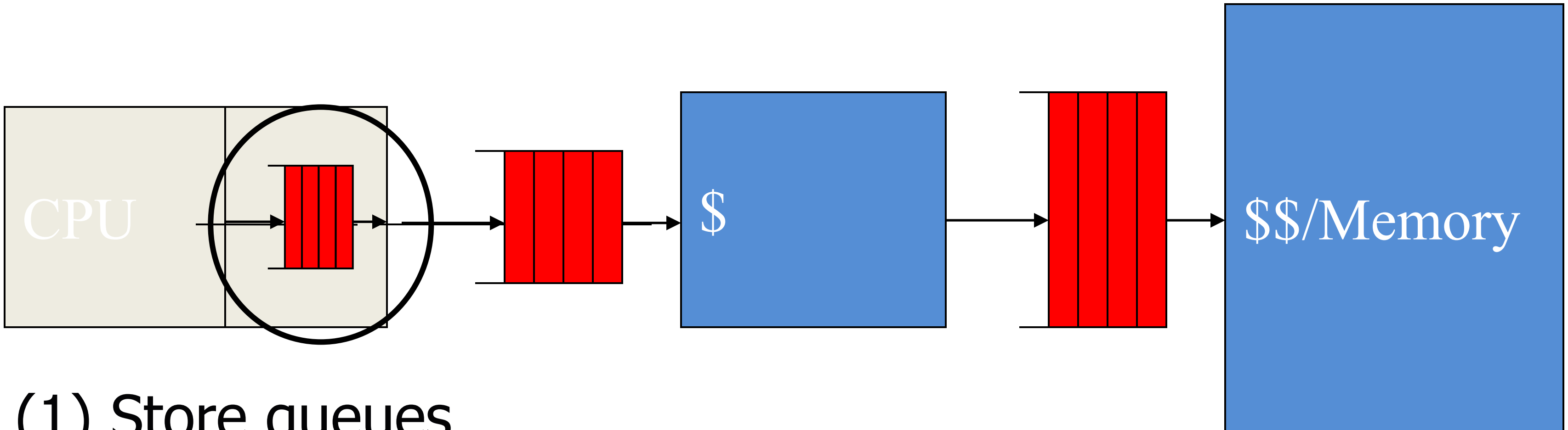


Tag/Data Access

- Cycle 2: write data
- Advanced Technique
  - Decouple write pipeline
  - In the same cycle
    - Check tag of $store_i$
    - Write data of $store_{i-1}$
    - Bypass data of $store_{i-1}$ to loads



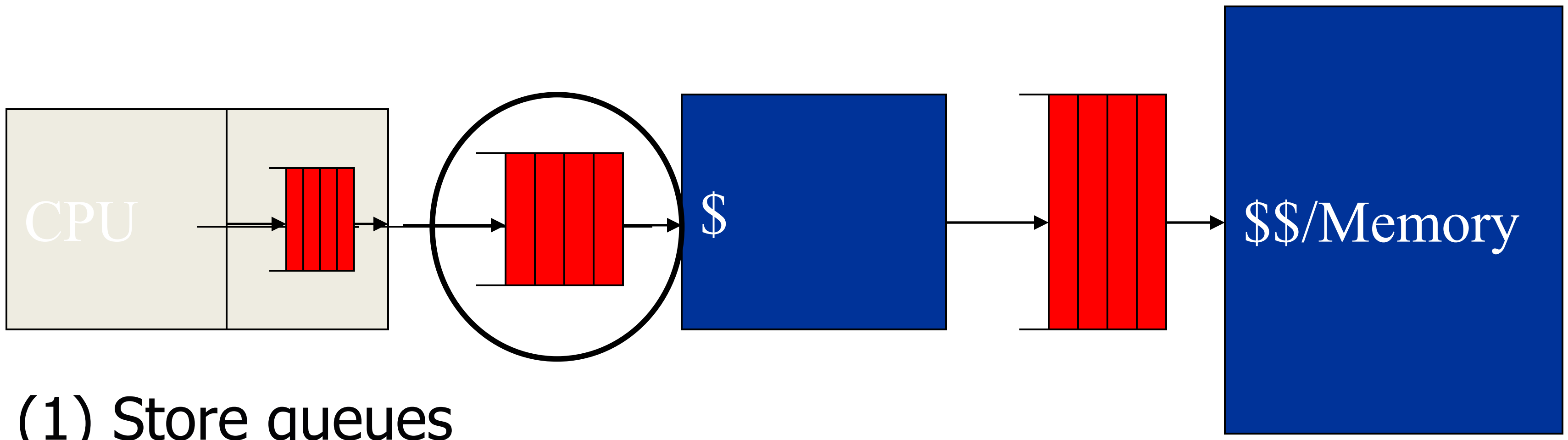
Buffering Writes 1 of 3: Store Queues



- (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

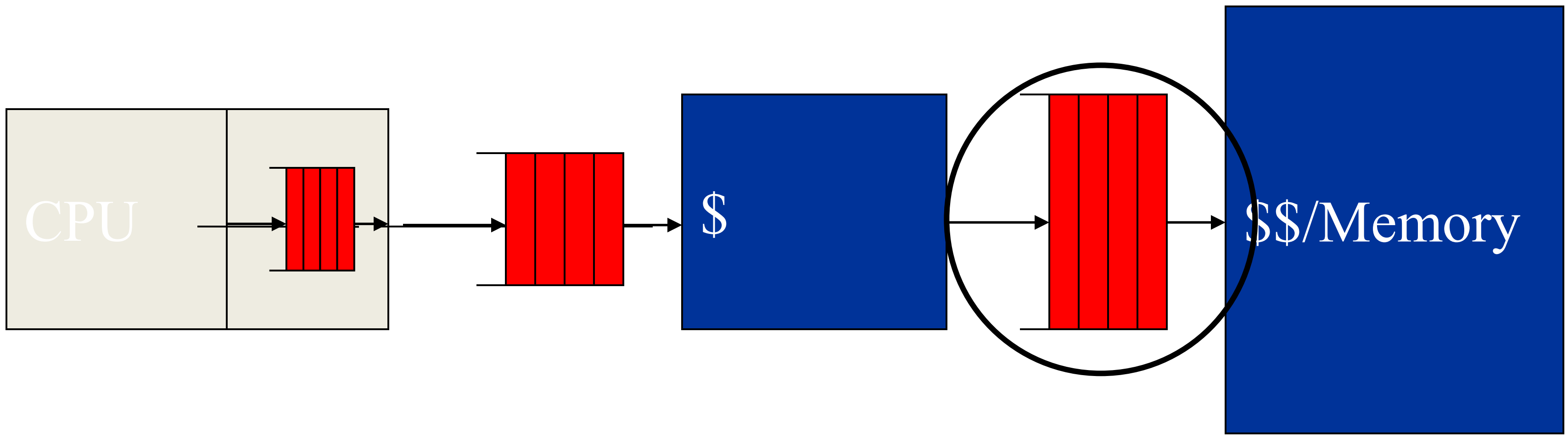
- (2) Write buffers
- (3) Writeback buffers

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
  - May 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 (Special case of Victim Buffer)
  - Transparent to architecture
  - Holds victim block(s) so miss/prefetch can start immediately
  - (Logically part of cache for multiprocessor coherence)

Increasing Cache Bandwidth

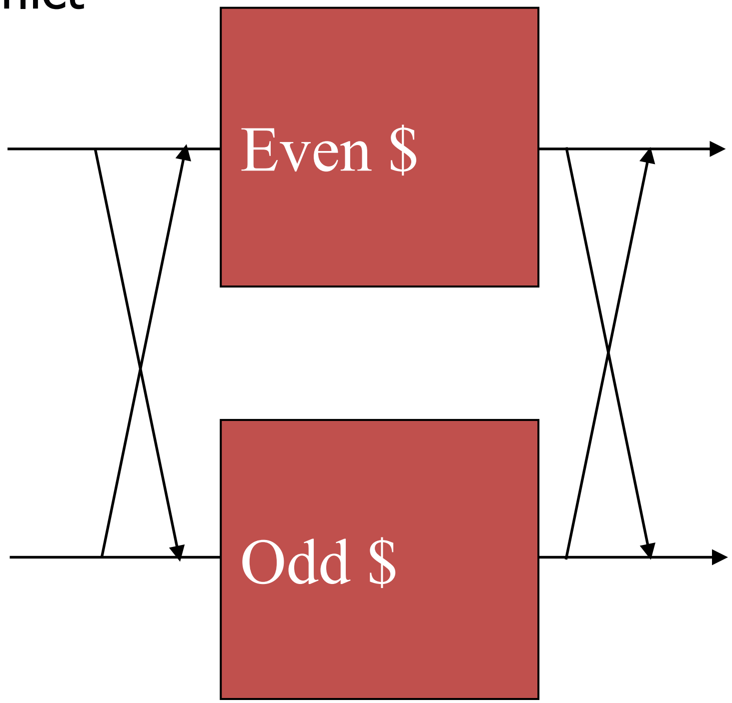
- What if we want to access the cache twice per cycle?
- Option #1: multi-ported cache
  - Same number of six-transistor cells
  - Double the decoder logic, bitlines, wordlines
    - Areas becomes “wire dominated” -> slow
  - OR, time multiplex the wires
- Option #2: banked cache
  - Split cache into two smaller “banks”
  - Can do two parallel access to different parts of the cache
  - Bank conflict occurs when two requests access the same bank
- Option #3: replication
  - Make two copies (2x area overhead)
  - Writes both replicas (does not improve write bandwidth)
  - Independent reads
  - No bank conflicts, but lots of area
  - Split instruction/data caches is a special case of this approach

Multi-Banking (Interleaving) Caches

- Address space is statically partitioned and assigned to different caches *Which addr bit to use for partitioning?*

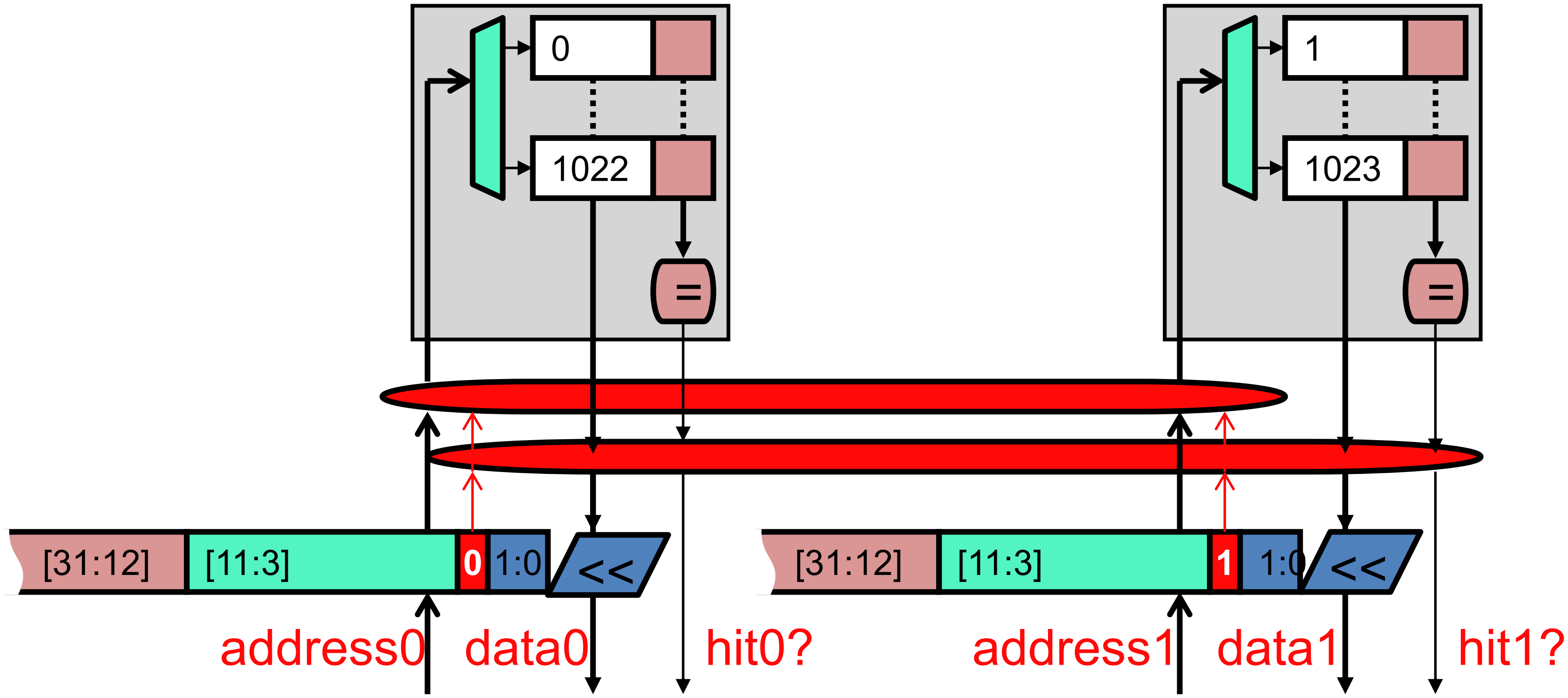
- A compromise (e.g. Intel P6, MIPS R10K)
  - multiple references per cyc. if no conflict
  - only one reference goes through
    if conflicts are detected
  - the rest are deferred
    (bad news for scheduling logic)

- Most helpful is compiler knows about the interleaving rules (esp. for in-order)



A Banked Cache

- **Banking** a cache
  - Simple: bank SRAMs
  - Which address bits determine bank? LSB of index
  - **Bank network** assigns accesses to banks, resolves conflicts
    - – Adds some latency too



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 $\rightarrow$ costs power$\uparrow$
    - – Cache-size$\downarrow$ $\rightarrow$ $\%_{miss}\uparrow$ $\rightarrow$ power$\uparrow$ (dynamic – off-chip access)
      execution time$\uparrow$ (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?
  - Hard, easier to track miss-rate vs. some threshold
  - Example: is $\%_{miss}$ higher than 5%?
    - N-bit counter (N = 8, say)
    - 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?

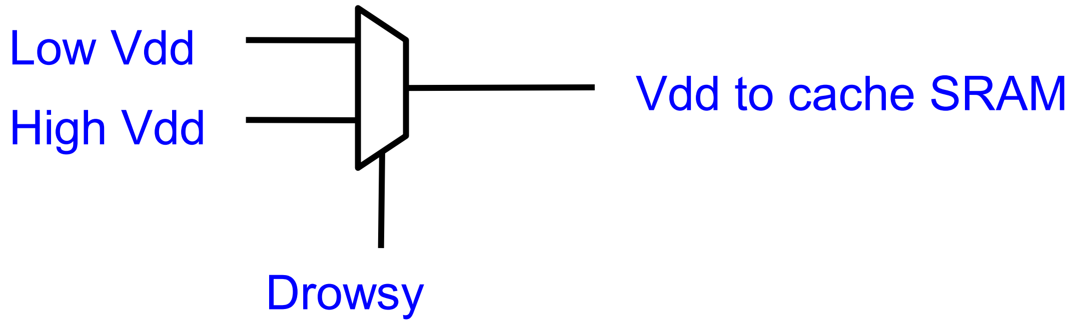
- **Reduce ways**
  - ["Selective Cache Ways", Albonesi, ISCA-98]
  - \+ Resizing doesn't change mapping of blocks to sets → simple
  - – Lose associativity

- **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



Simple Power Modeling

An Energy Calculation

- Parameters
  - 2-way set associative D$
  - 10% miss rate
  - 5μW/access tag way, 10μW/access data way
- What is power/access of parallel tag/data design?
  - Parallel: each access reads both tag ways, both data ways
    - Misses write additional tag way, data way (for fill)
  - [2 * 5μW + 2 * 10μW] + [0.1 * (5μW + 10μW)] = 31.5 μW/access
- What is power/access of serial tag/data design?
  - Serial: each access reads both tag ways, one data way
    - Misses write additional tag way (actually…)
  - [2 * 5μW + 10μW] + [0.1 * 5μW] = 20.5 μW/access

Energy Calculation Details

- What is power/access of parallel tag/data design?
  - Parallel: each access reads both tag ways, both data ways

- Breakdown:
  - Hit (90% of time):
    1. Access both tags – 2 * 5μW (one hits)
    2. In parallel access both data arrays – 2 * 10μW (one hits)
  - Miss (10% of time):
    1. Access both tags – 2 * 5μW (both miss)
    2. In parallel access both data arrays – 2 * 10μW (no hit, ignore)
    3. When fill returns, update tag & data arrays for 1 way – 5μW + 10μW
  - [2*5μW+2*10μW] + [0.1*(5μW+10μW)] = 31.5 μW/acc.

Energy Calculation Details

- What is power/access of serial tag/data design?
  - Serial: each access reads both tag ways, one data way
    - Misses write additional tag way (actually…)
- Breakdown
  - Hit (90% of the time)
    1. Access both tag arrays – 2 * 5μW
    2. Based on result (one hits), access 1 data array – 10μW
  - Miss (10% of the time)
    1. Access both tag arrays – 2 * 5μW
    2. No need to fill, just update tag and data array – 5μW + 10μW
  - = 2 * 5μW*(0.9 + 0.1) + 0.9*10μW + [0.1 * (5μW + 10μW)]
  - = 2 * 5μW + 1.0*10μW + [0.1 * 5μW]
  - = [2 * 5μW + 10μW]*1 + [0.1 * 5μW] = 20.5 μW/access

Bonus (mostly review)

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

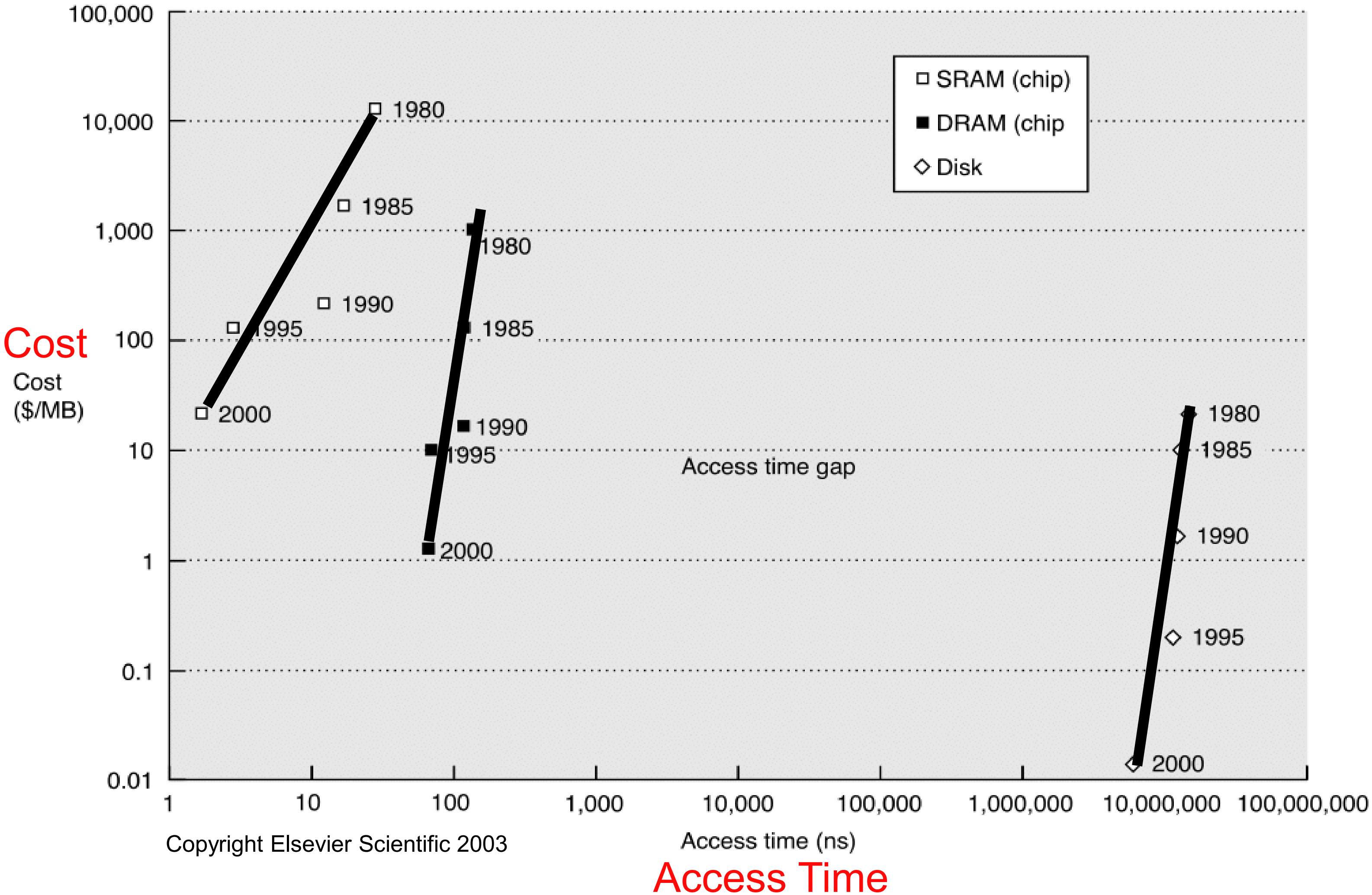
- Dynamic RAM (DRAM)
  - 1 transistor + 1 capacitor per bit
  - Optimized for density (in terms of cost per bit)
  - Slow (>40ns internal access, >100ns pin-to-pin)
  - Different fabrication steps (does not mix well with logic)

- Nonvolatile storage: Magnetic disk, Flash RAM

Storage Technology

- Cost - what can $300 buy today? (very approx.)
  - SRAM - 50MB (maybe a little more?...)
  - DRAM - 50,000MB (50GB) --- 1000x cheaper than SRAM
  - Flash SSD- 3,000,000MB (3000GB) --- 60x cheaper than DRAM
  - Disk – 12,000,000MB (12TB) --- 4x cheaper than Flash
- Latency (random page read)
  - SRAM - <1ns (on chip)
  - DRAM - ~100ns --- 100x or more slower
  - Flash SSD - ~ 100,000ns – 1000x slower than dram
  - Disk - 10,000,000ns or 10ms --- 100x slower (mechanical)
- Bandwidth (sequential read)
  - SRAM – 10s-100s TB/sec (~60 TB/s Volta GPU)
  - DRAM – 10s-100s GB/sec (400GB/s in Xeon PHI, 900 GB/s Volta)
  - Flash SSD – 500 MBps
  - Disk - 150MB/sec (0.15 GB/sec)
- Aside: New non-volatile mem. (b/t dram and FLASH)
  - Crosspoint: 300GB for $300 + 5000ns random access latency

Storage Technology Trends



Locality to the Rescue

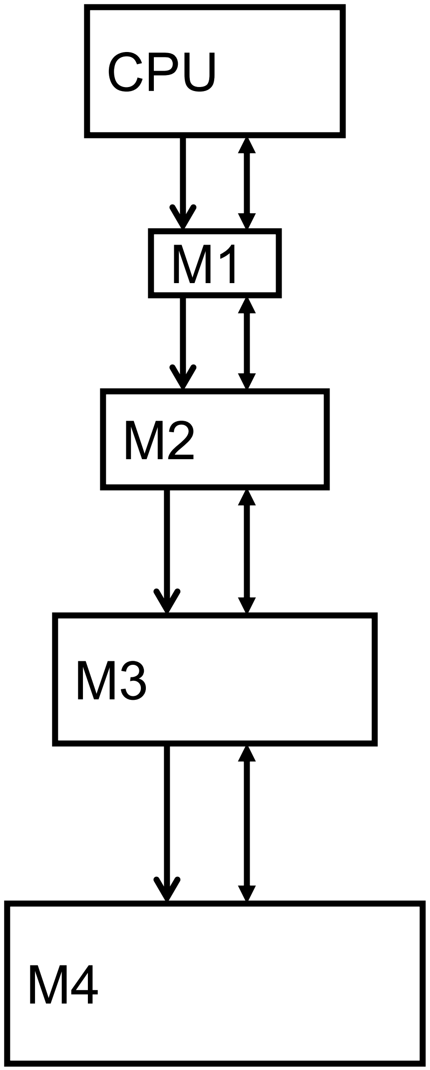
- Locality of memory references
  - Property of real programs, few exceptions
  - Books and library analogy

- Temporal locality
  - Recently referenced data is likely to be referenced again soon
  - **Reactive**: cache recently used data in small, fast memory

- Spatial locality
  - More likely to reference data near recently referenced data
  - **Proactive**: fetch data in large chunks to include nearby data

- 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

Known From the Beginning

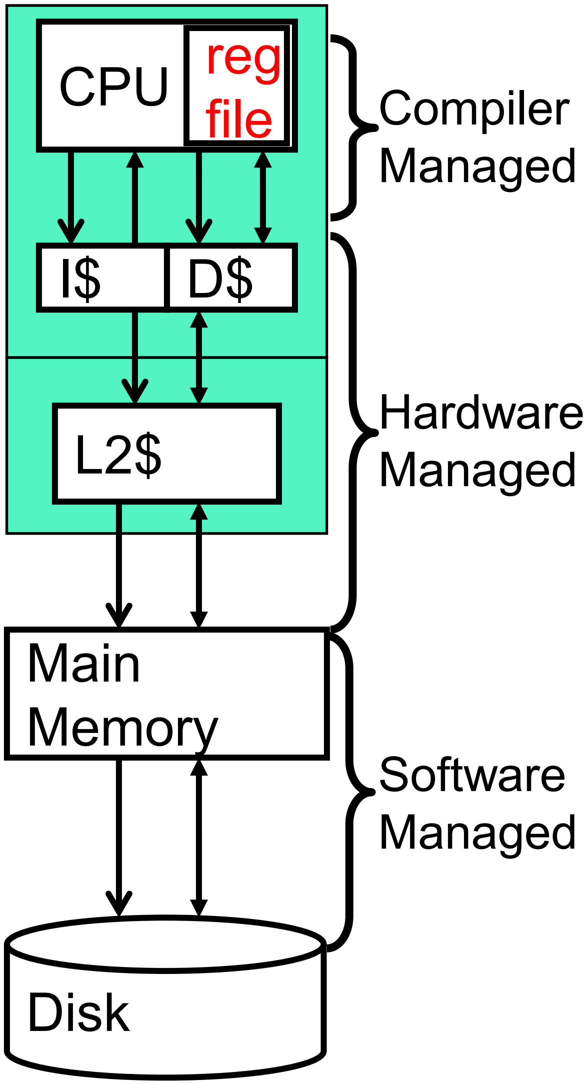
"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstein, VonNeumann

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

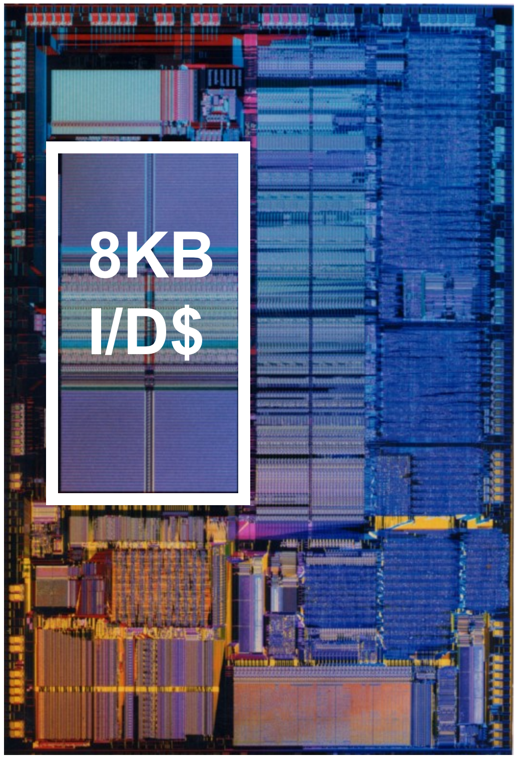
IAS memo 1946

Concrete Memory Hierarchy

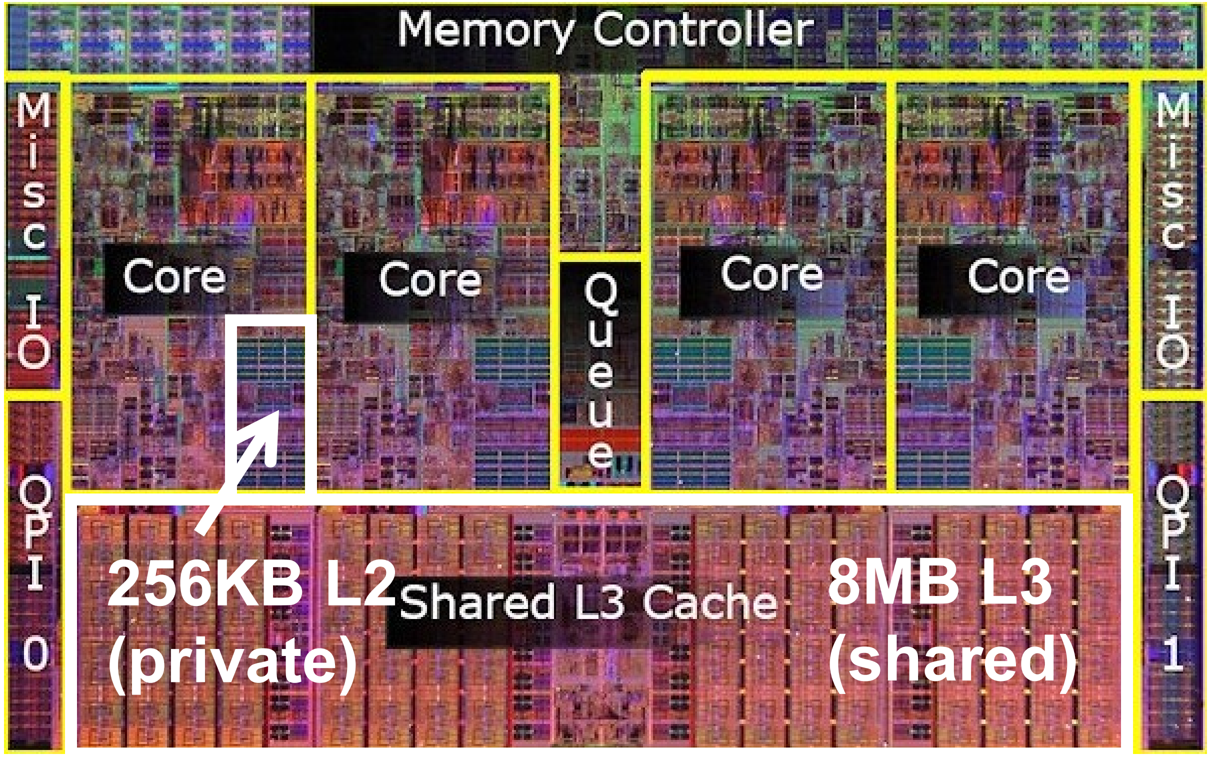


- 0^{th} Level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8-64KB each
- 2nd level: **Second-level cache** (L2$)
  - On-die (with CPU)
  - Made of SRAM (same circuit type as CPU)
  - Typically 4-20MB Total
  - Split between Private vs Shared
- 3rd level: **main memory**
  - Made of DRAM
  - Typically >4-64GBs (phone to PC)
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - SSDs or Magnetic Disks

Evolution of Cache Hierarchies



Intel 486

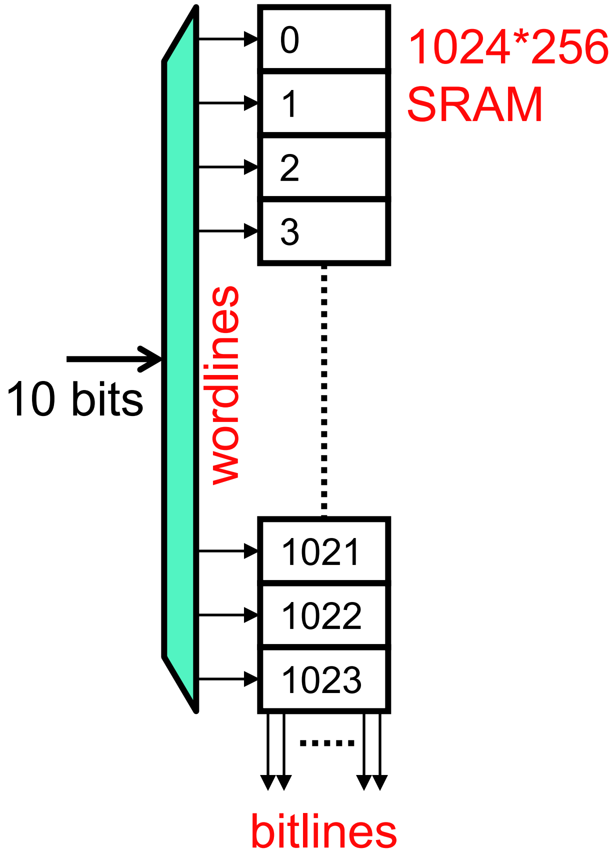


Intel Core i7 (quad core)

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

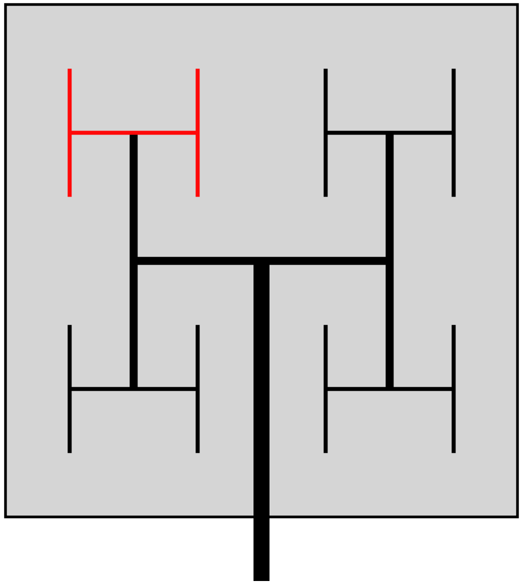
Basic Memory Array Structure

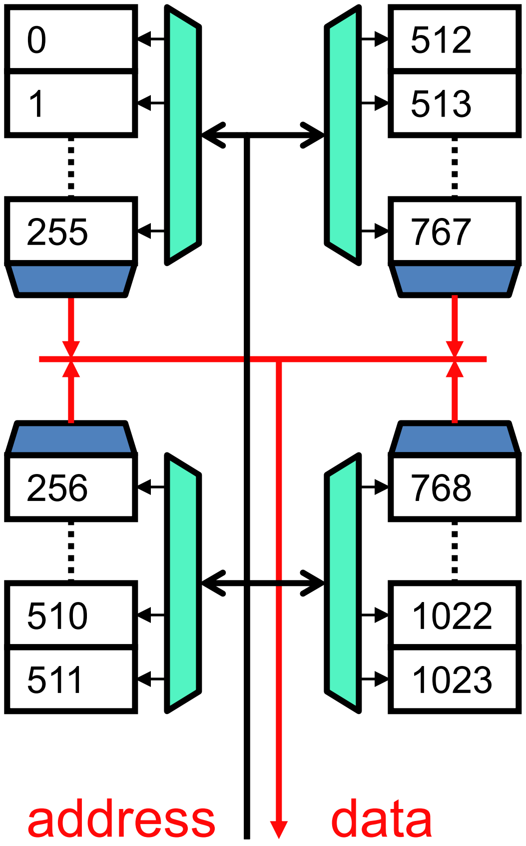
- Number of entries
  - 2^n, where n is number of address bits
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
  - One-bit address travels on "wordlines"
- Size of entries
  - Width of data accessed
  - Data travels on "bitlines"
  - 256 bits (32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



Physical Cache Layout

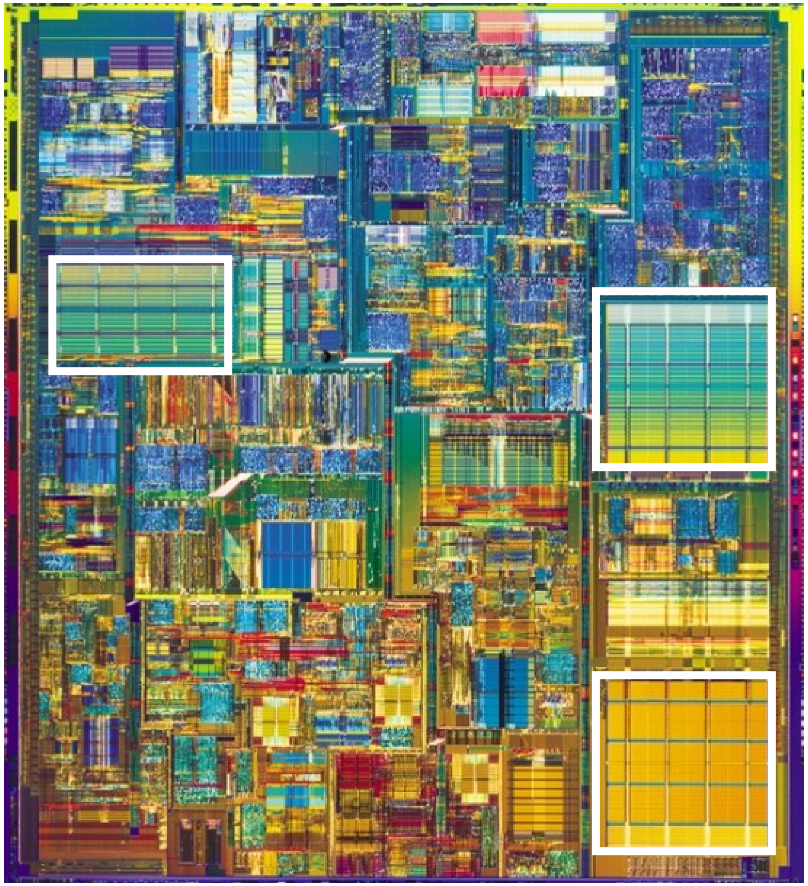
- Logical layout
  - Arrays are vertically contiguous
- Physical layout - roughly square
  - Vertical partitioning to minimize wire lengths
  - **H-tree**: horizontal/vertical partitioning layout
    - Applied recursively
    - Each node looks like an H





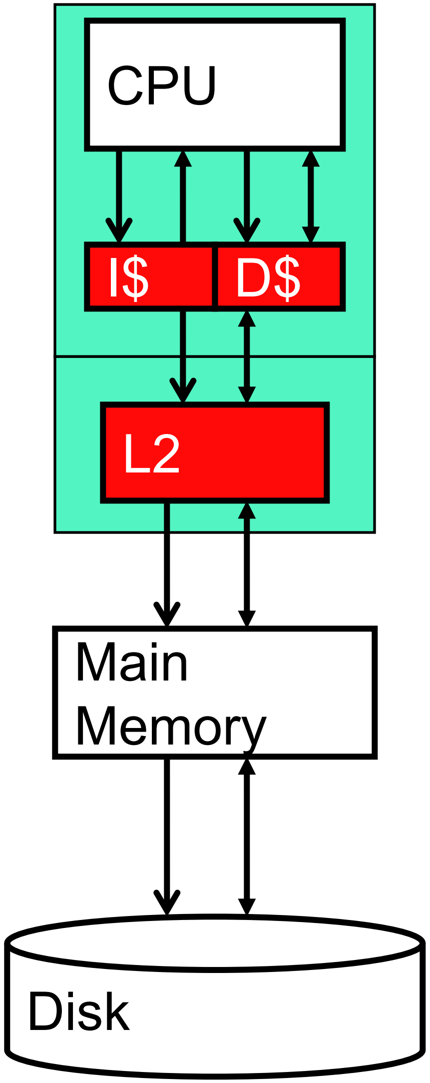
Physical Cache Layout

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



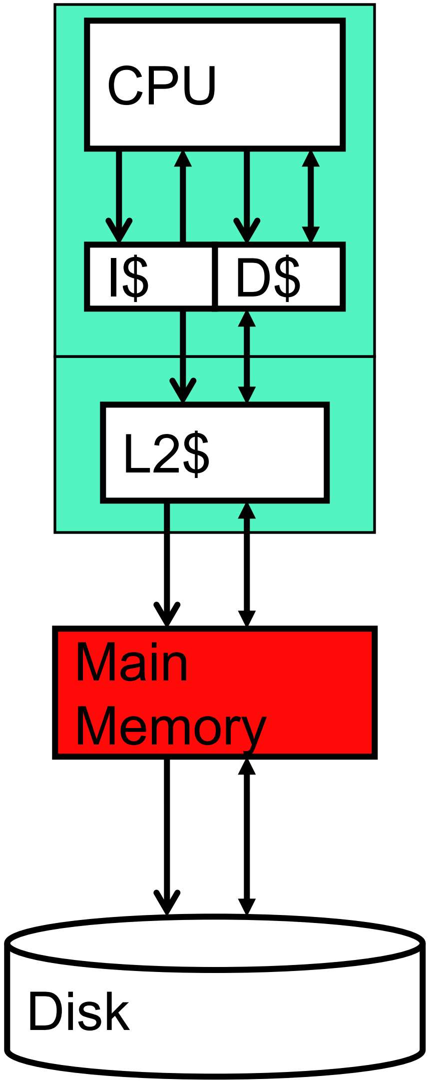
Pentium 4

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

Looking forward: Memory



- Main memory
  - Virtual memory
  - DRAM-based memory systems

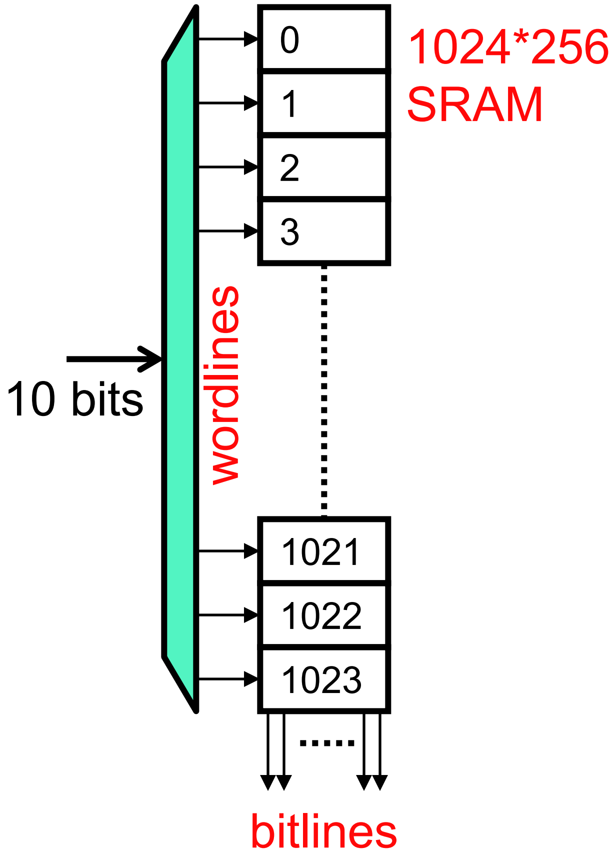
Warmup

- What is a “hash table”?
  - What is it used for?
  - How does it work?

- Short answer:
  - Maps a “key” to a “value”
    - Constant time lookup/insert
  - Have a table of some size, say N, of “buckets”
  - Take a “key” value, apply a hash function to it
  - Insert and lookup a “key” at “hash(key) modulo N”
    - Need to store the “key” and “value” in each bucket
    - Need to check to make sure the “key” matches
  - Need to handle conflicts/overflows somehow (chaining, re-hashing)

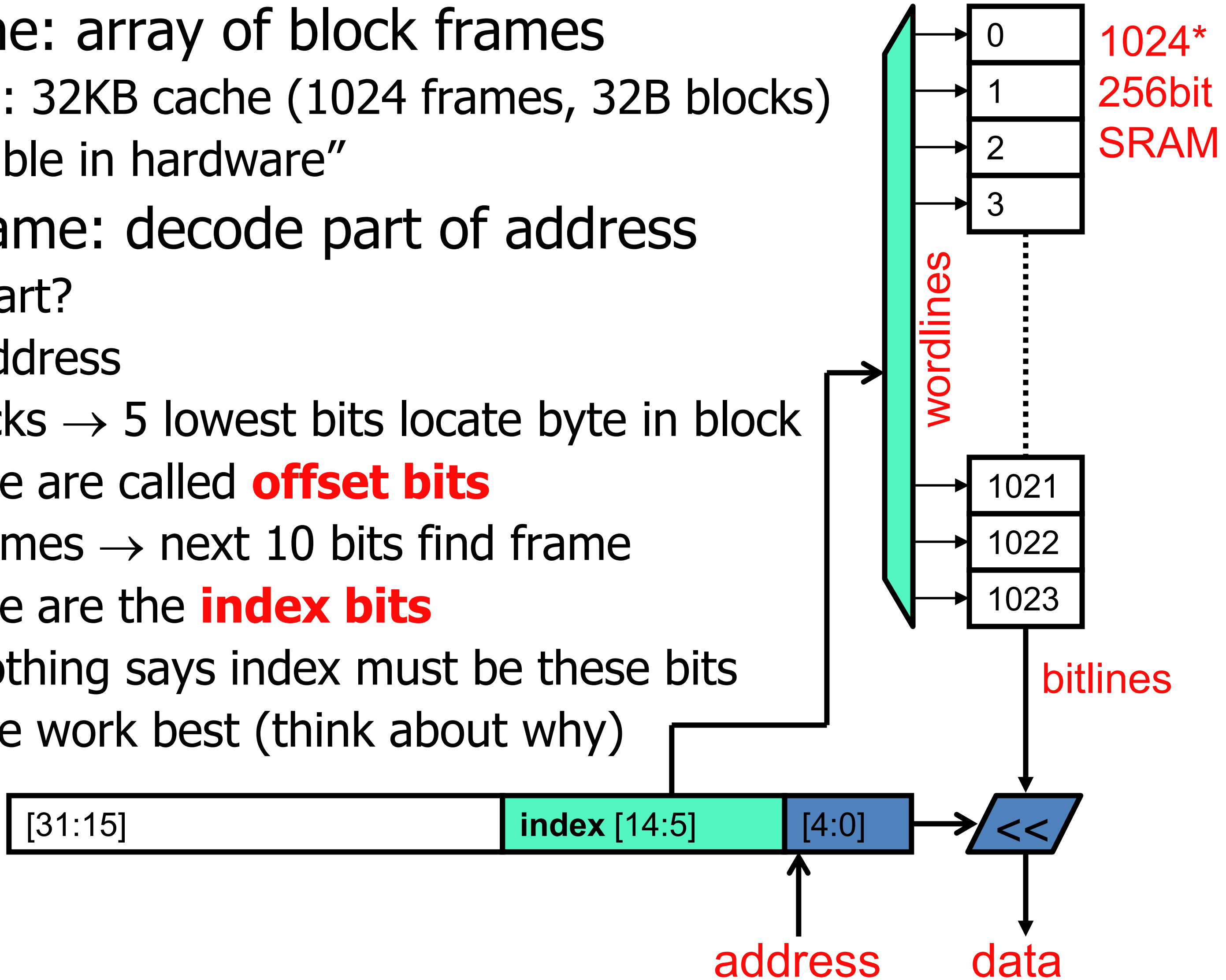
Basic Memory Array Structure

- Number of entries
  - 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
  - Width of data accessed
  - Data travels on "bitlines"
  - 256 bits (32 bytes) in example



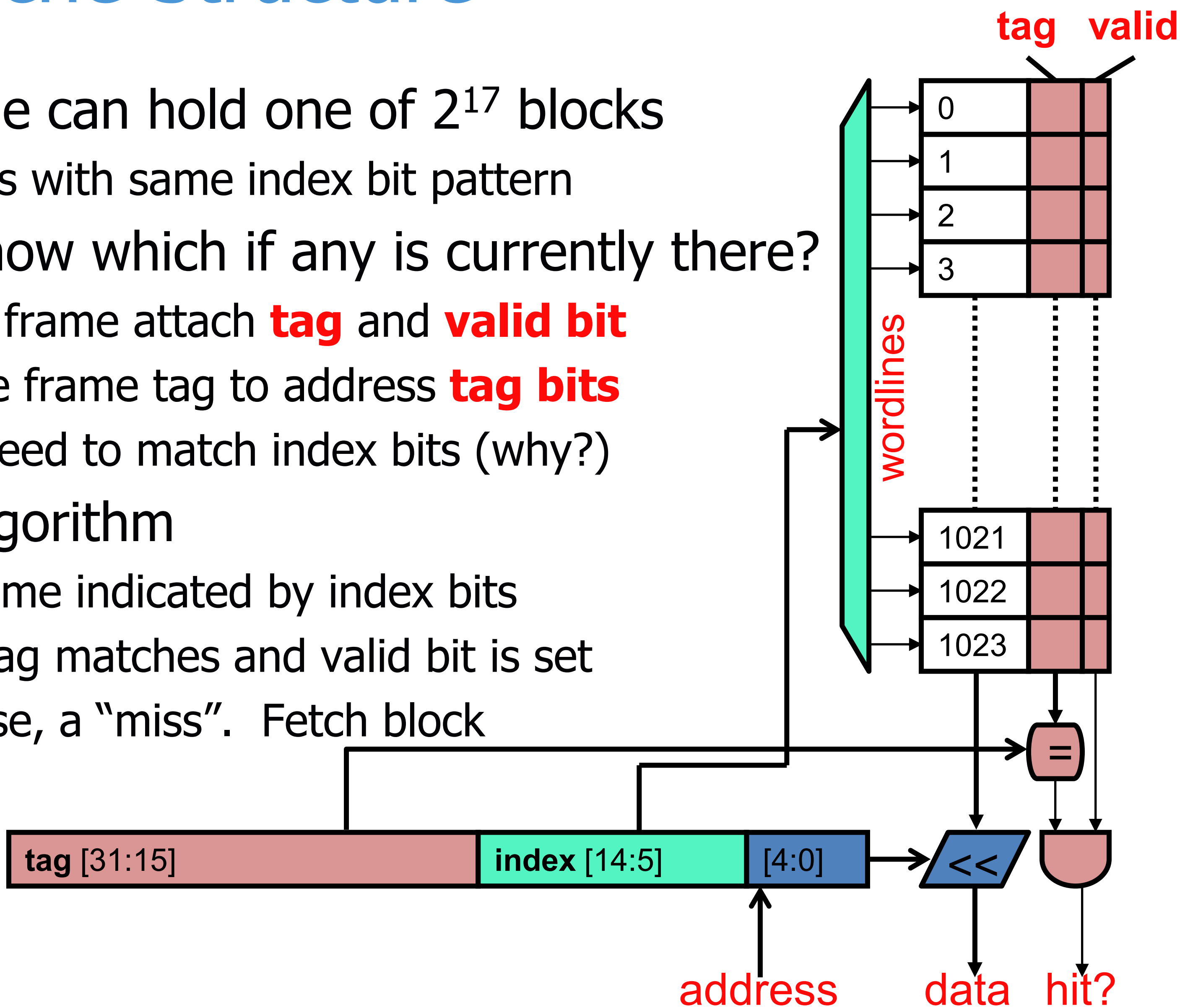
Basic Cache Structure

- Basic cache: array of block frames
  - 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

tag (3 bits)	**index** (3 bits)	2 bits

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	

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

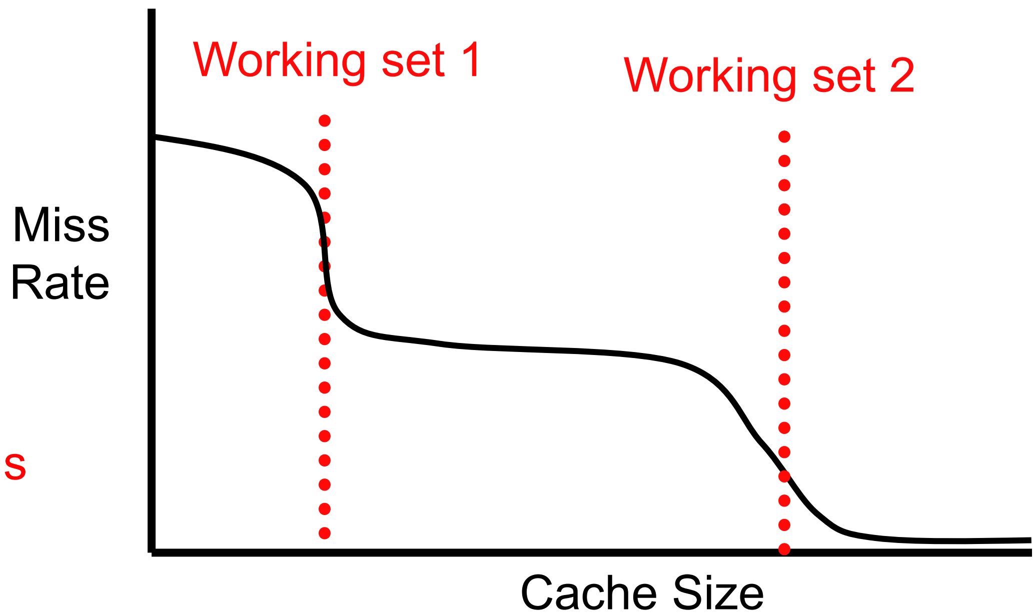
Cache contents	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	Miss
0000, 0010, **3020**, 0030, 0100, 0110, 0120, 0130	3030	Miss
0000, 0010, 3020, **3030**, 0100, 0110, 0120, 0130	2100	Miss
0000, 0010, 3020, 3030, **2100**, 0110, 0120, 0130	0012	Hit
0000, 0010, 3020, 3030, 2100, 0110,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

Miss Rate: ABC

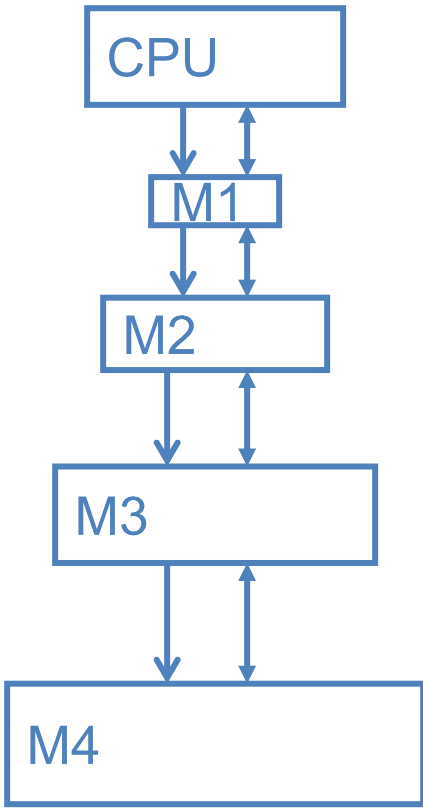
- **Capacity**
  - + Decreases capacity misses
  - – Increases latency_{hit}
- **Associativity**
  - + Decreases conflict misses
  - – Increases latency_{hit}
- **Block size**
  - – Increases conflict misses (fewer frames)
  - + Decreases compulsory misses (spatial prefetching)
  - No effect on latency_{hit}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

Increase Cache Size

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



Most workloads have multiple working sets

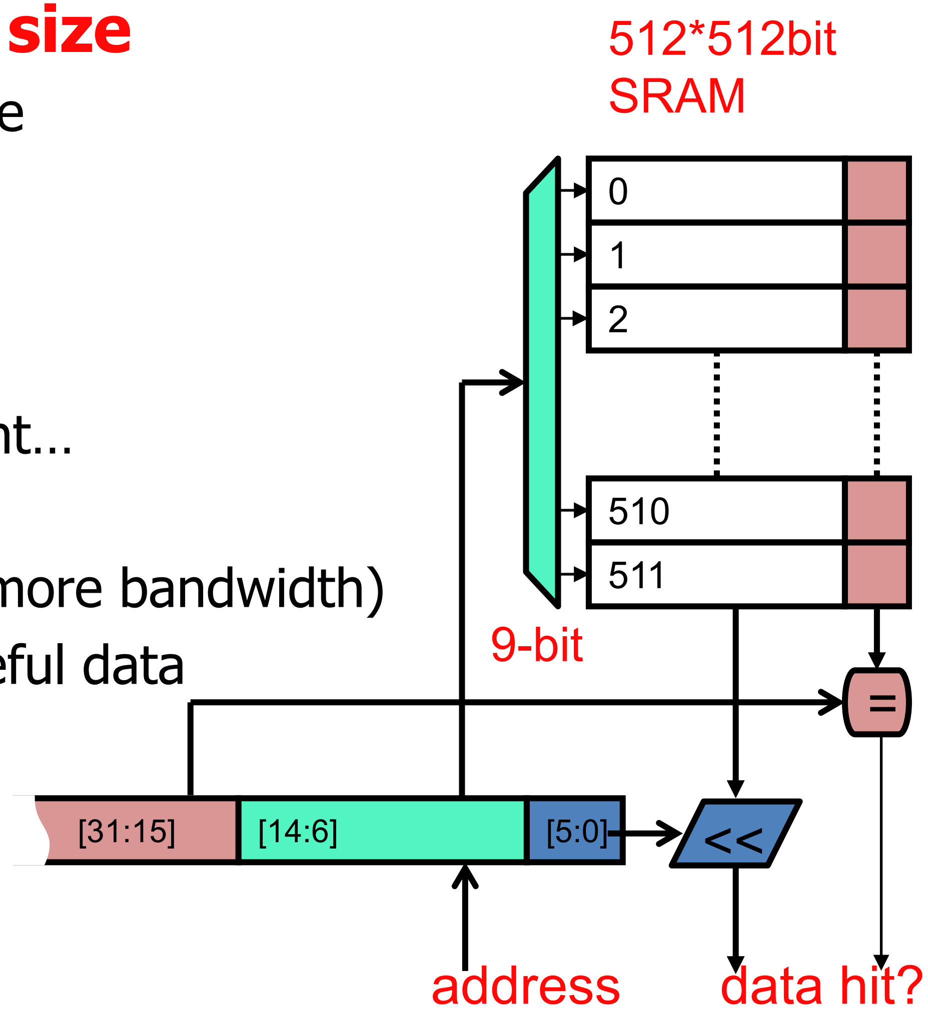


Aside: working set size affects which level a given data item resides at

- Given capacity, manipulate $\%_{\text{miss}}$ by changing **organization**

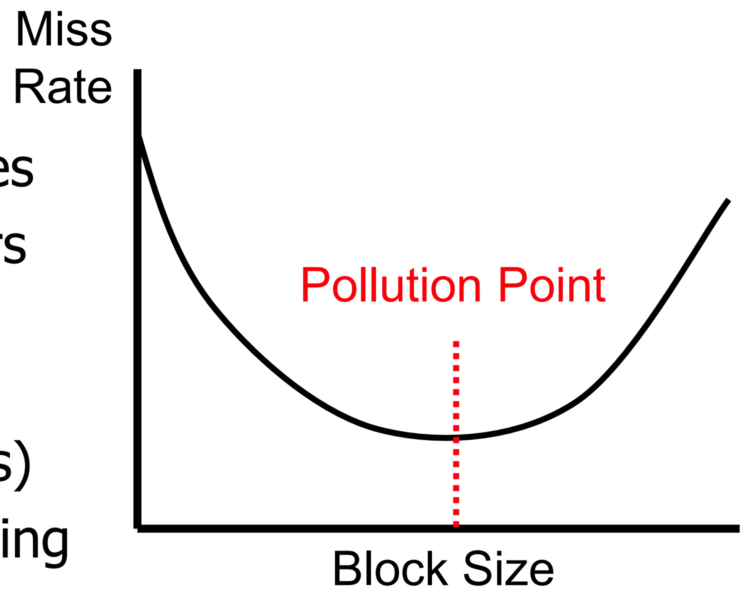
Block Size

- Given capacity, manipulate $\%_{miss}$ by changing organization
- One option: increase **block size**
  - Notice index/offset bits change
  - Tag remain 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
  – Fragmentation



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 16–128B
    - 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)

tag (3 bits)	**index** (2 bits)	3 bits

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

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

tag (3 bits)	**index** (3 bits)	2 bits

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

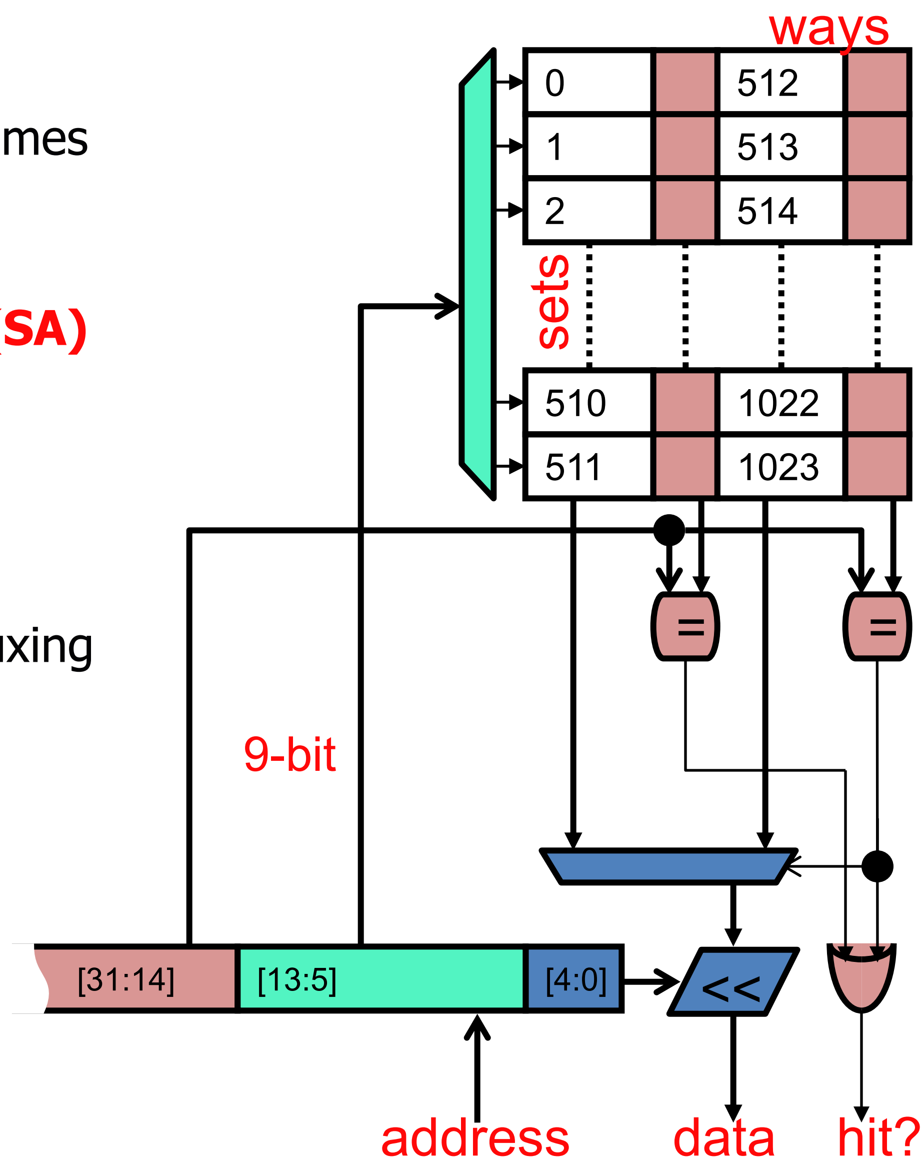
: (

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)**

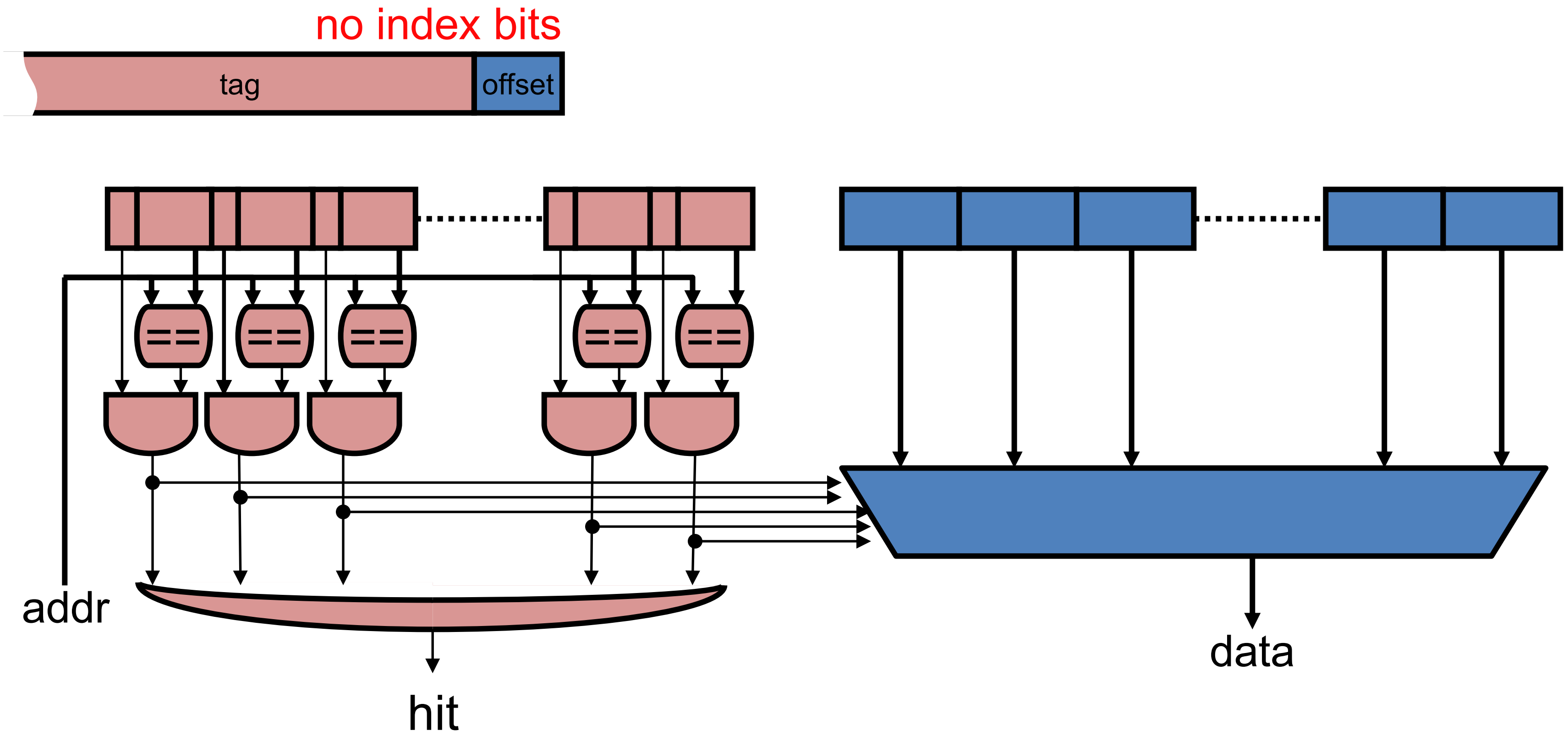
+ Reduces conflicts
– Increases latency$_{\text{hit:}}$ additional muxing

- Note: valid bit not shown



High (Full) Associative Caches

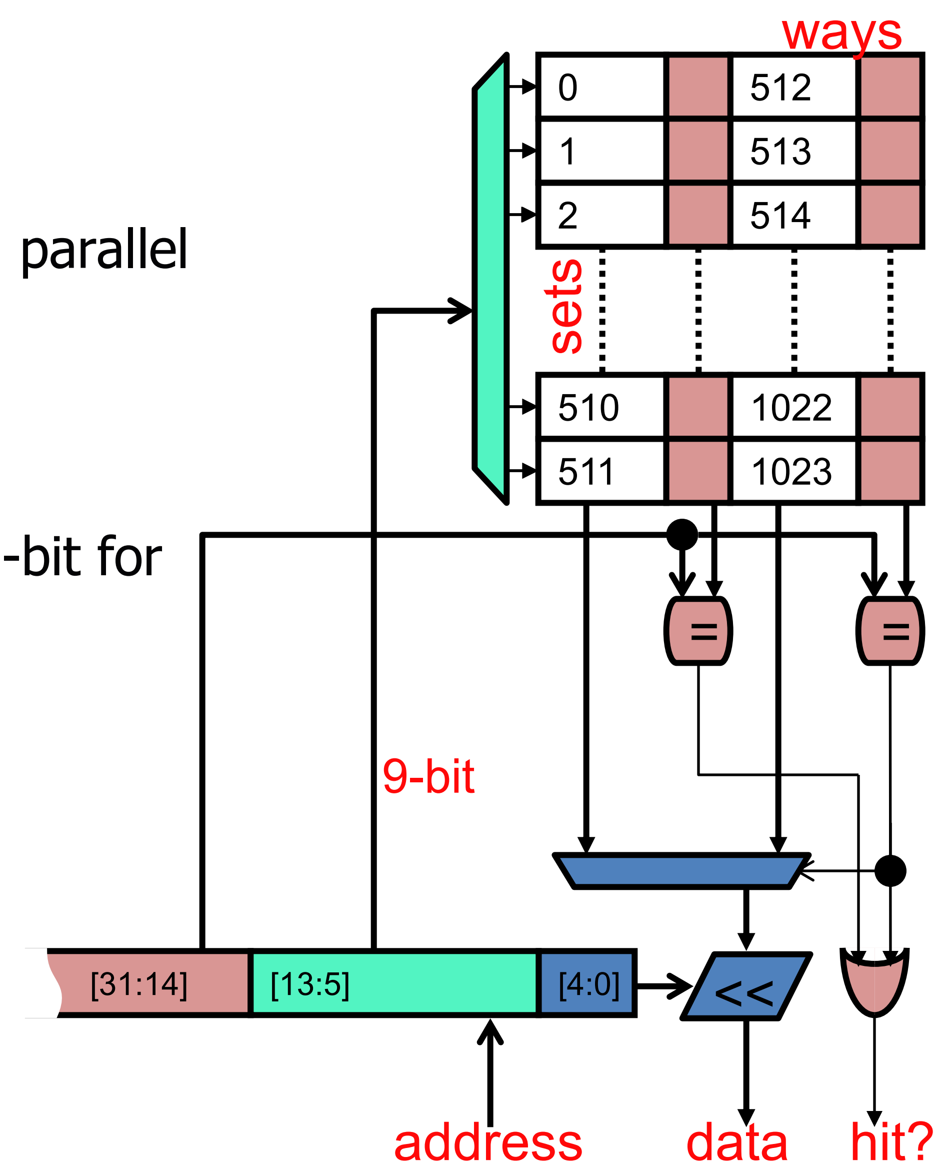
- How to implement full (or at least high) associativity?
  - **This way is terribly inefficient**
  - Matching each tag is needed, but not reading out each tag



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

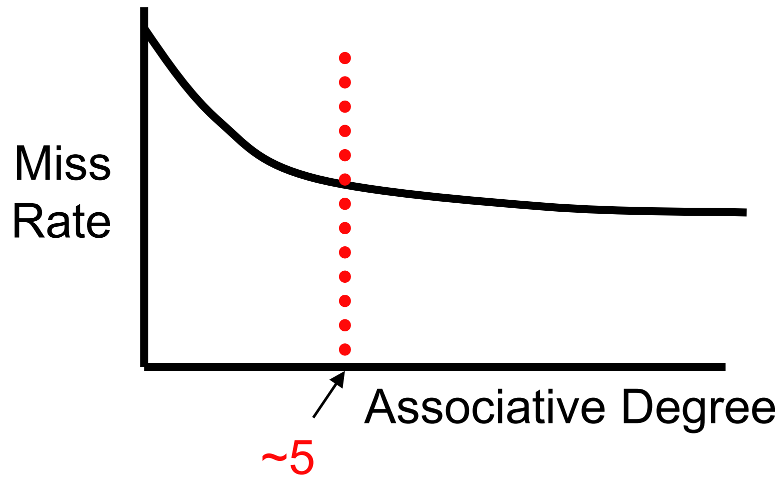
- Parameters: 32B cache, 4B blocks, **2-way set-associative**
  - Initial contents : 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130

tag (4 bits)	**index** (2 bits)	2 bits

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)**

Increase Associativity

- Higher associative caches have better miss rates
  - However $latency_{hit}$ increases
- Diminishing returns (for a single thread)



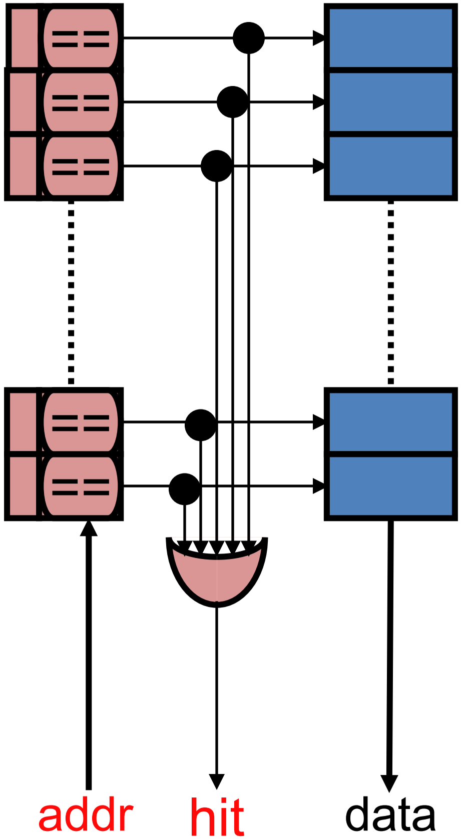
- Block-size and number of sets should be powers of two
  - Makes indexing easier (just rip bits out of the address)
  - 3-way set-associativity? No problem
- **New Question: Where to put each block?**

High (Full) Associative Caches with CAMs

- **CAM**: content addressable memory
  - Array of words with **built-in comparators**
  - **No separate "decoder" logic**
  - Input is value to match (tag)
  - Generates 1-hot encoding of matching slot

- Fully associative cache
  - Tags as CAM, data as RAM
  - Effective but somewhat expensive
    - But cheaper than any other way
  - Used for high (16-/32-way) associativity
  - – No good way to build 1024-way associativity
  - \+ No real need for it, either

- CAMs are used elsewhere (tag match)



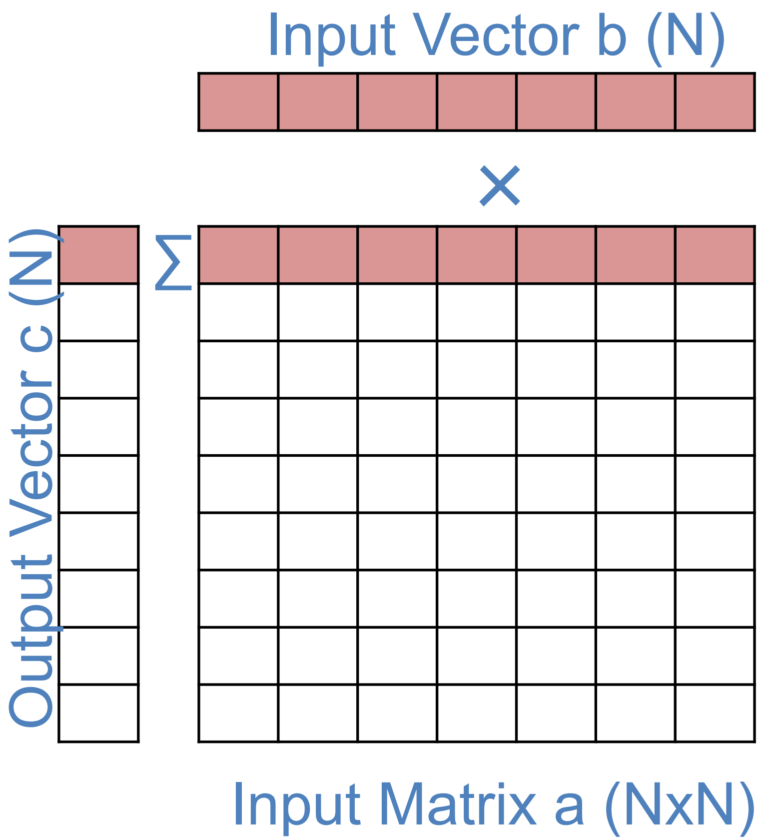
Matrix-Vector Multiply (loop ordering)

```
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

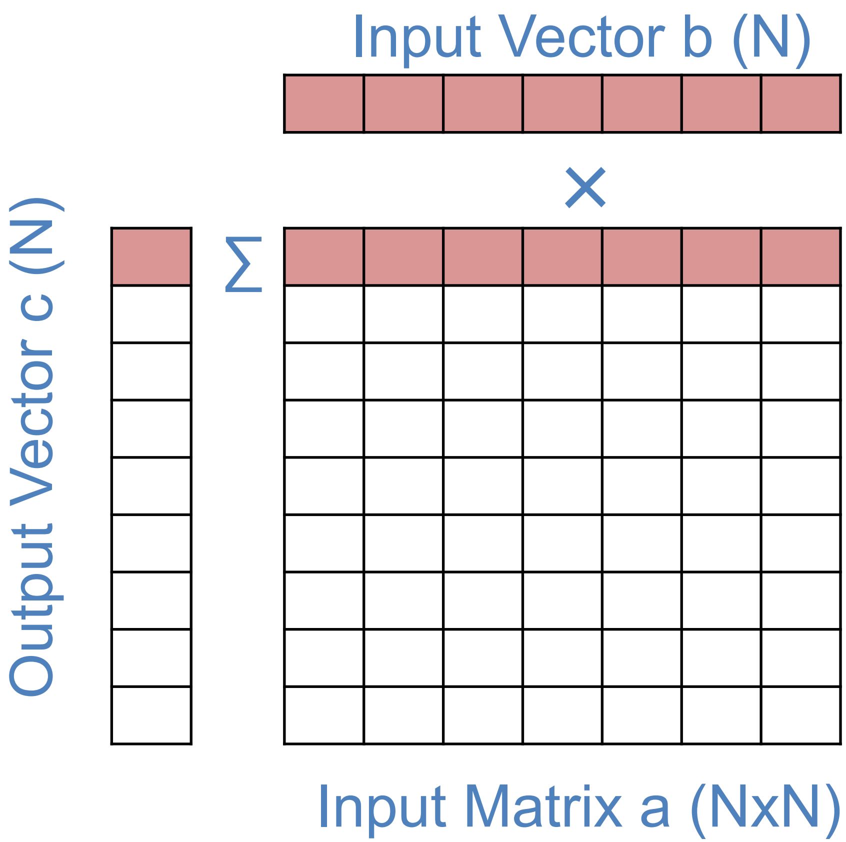
```
for(int i = 0; i < N; ++i)
  for(int j = 0; j < N; ++j)
    c[i] += a[i][j] * b[j];
```

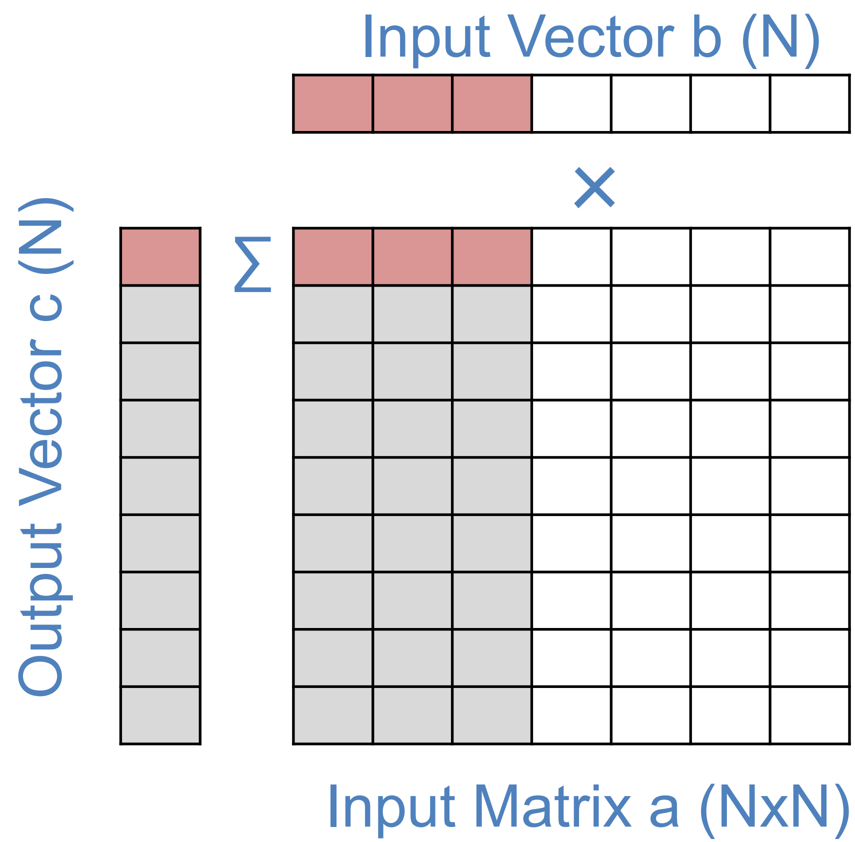
0/8 1/8 1/8



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

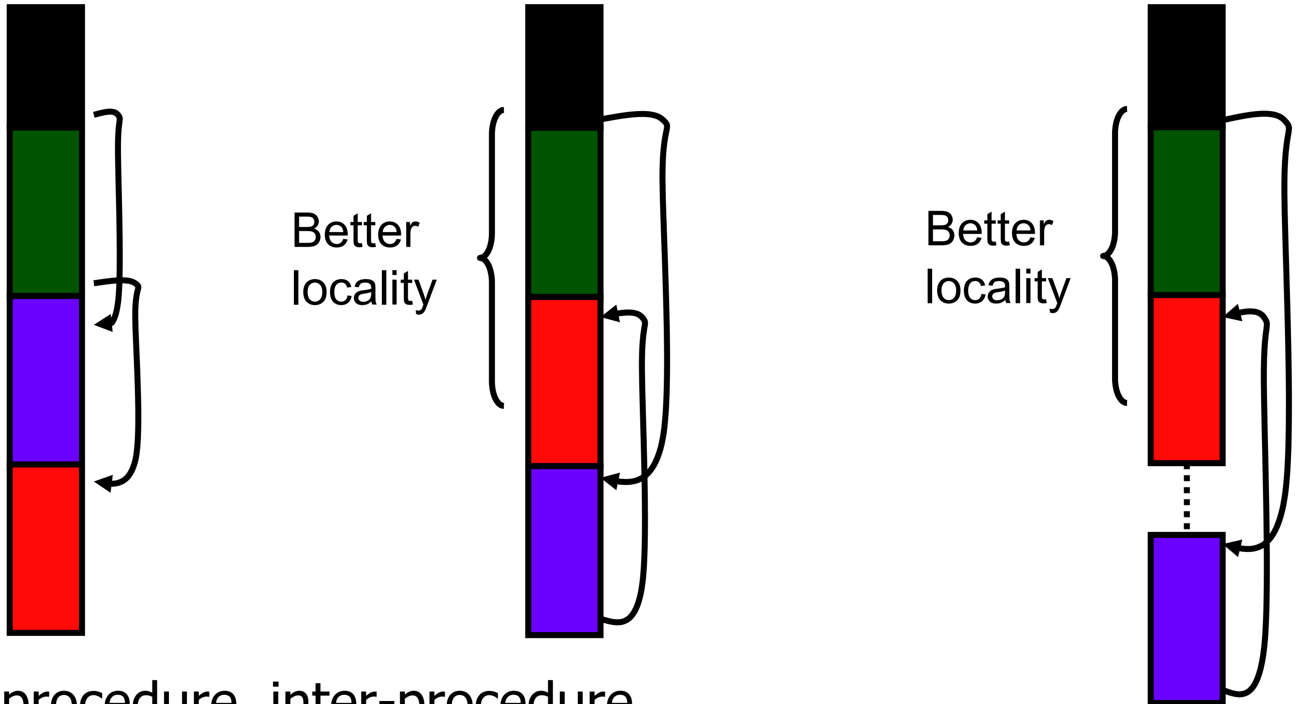
```
int b[N];
int a[N][N];
int c[N];

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

Software Restructuring: Code

- Compiler lays out code for temporal and spatial locality
  - If (a) { **code1;** } else { **code2;** } **code3;**
  - ***But, code2 case never happens (say, error condition)***



- Intra-procedure, inter-procedure
- Related to trace scheduling

Write-Through vs. Write-Back

- When to propagate new value to (lower level) memory? (What to do on a write hit?)
  - **Write-through**: immediately
    - \+ Conceptually simpler
    - \+ Uniform latency on misses **(its okay to overwrite any frame)**
    - – 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)

- Common design: Write through L1, write-back L2/L3

Write-allocate vs. Write-non-allocate

- What to do on a write miss?
  - **Write-non-allocate**: just write to next level
    - – Potentially more read misses
    - + Uses less bandwidth
    - Used mostly with write-through
  - **Write-allocate**: read block from lower level, write value into it
    - + Decreases read misses
    - – Requires additional bandwidth
    - Used mostly with write-back

- Write allocate is common for write-back
  - Write-non-allocate for write through

Superscalar processor + multicore -> More Bandwidth

(Going to skip because we talked about this earlier)

Increasing Cache Bandwidth

- What if we want to access the cache twice per cycle?
- Option #1: multi-ported cache
  - Same number of six-transistor cells
  - Double the decoder logic, bitlines, wordlines
    - Areas becomes “wire dominated” -> slow
  - OR, time multiplex the wires
- Option #2: banked cache
  - Split cache into two smaller “banks”
  - Can do two parallel access to different parts of the cache
  - Bank conflict occurs when two requests access the same bank
- Option #3: replication
  - Make two copies (2x area overhead)
  - Writes both replicas (does not improve write bandwidth)
  - Independent reads
  - No bank conflicts, but lots of area
  - Split instruction/data caches is a special case of this approach

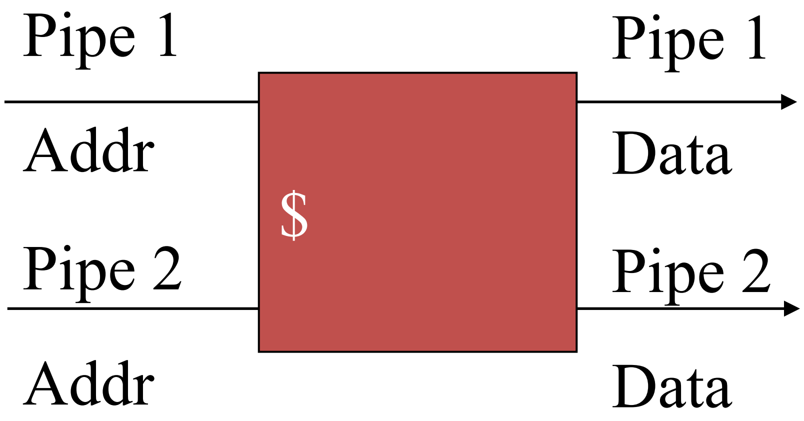
Multi-Port Caches

- Superscalar processors requires multiple data references per cycle

- Time-multiplex a single port (double pump)
  - need cache access to be faster than datapath clock
  - not scalable

- Truly multi-ported SRAMs are possible, but
  - more chip area
  - slower access

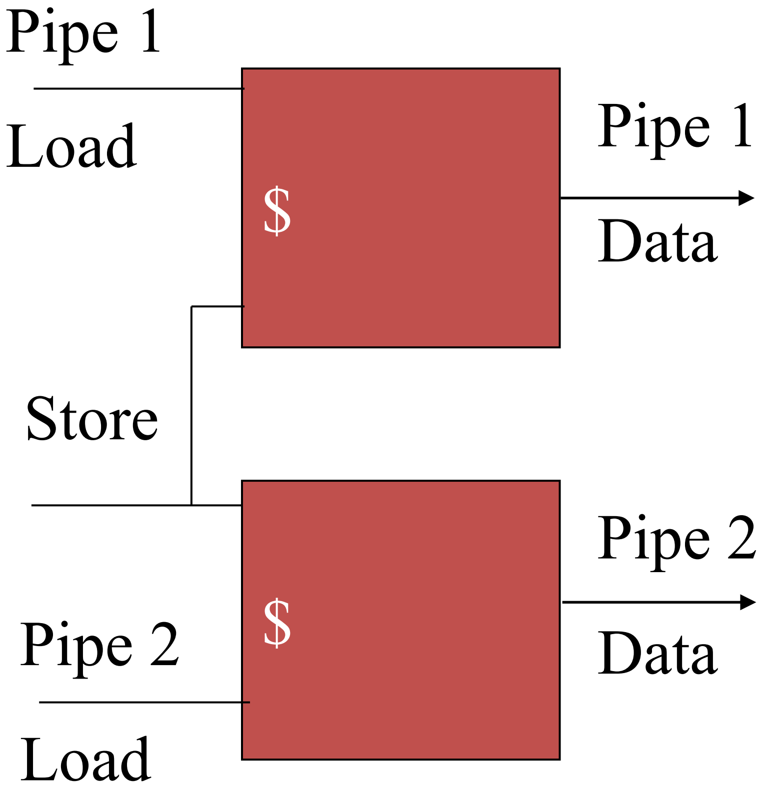
    (very undesirable for L1-D)



Multiple Cache Copies: e.g. Alpha 21164

- Independent fast load paths
- Single shared store path

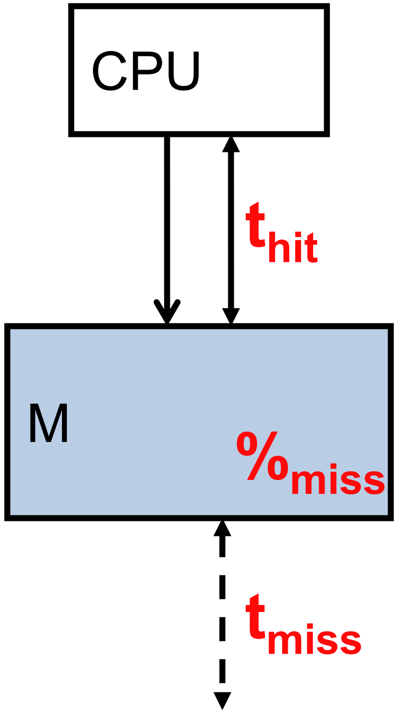
- Not a scalable solution
  - Store is a bottleneck
  - Doubles area



Simple Perf/Power Modeling

(Likely skip for time)

Memory Performance Equation



- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M

  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{hit}: time to read data from (write data to) M
  - t_{miss}: time to read data into M

- Performance metric
  - t_{avg}: average access time

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

Performance Calculation I

- Parameters
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - Main memory: t_{hit} = 50ns

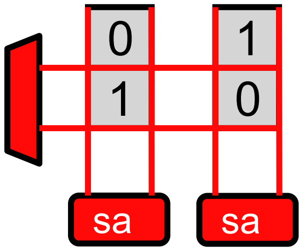
- What is $t_{avgD\$}$
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{hitM}$ = 1ns+(0.05*50ns) = 3.5ns

Performance Calculation II

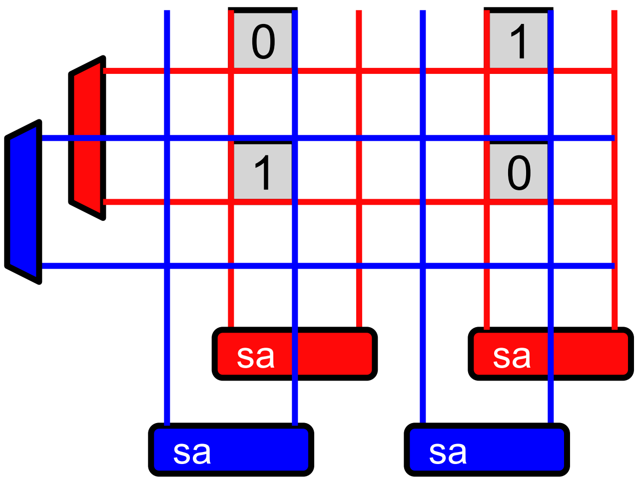
- In a pipelined processor, I$/D$ t_{hit} is "built in" (effectively 0)
- Parameters
  - Base pipeline CPI = 1
  - Instruction mix: 30% loads/stores
  - I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles
- What is new CPI?
  - Assumption: miss time cannot be overlapped (is this right?)
  - $CPI_{I\$} = \%_{missI\$} * t_{miss} = 0.02*10$ cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{memory} * \%_{missD\$} * t_{missD\$} = 0.30*0.10*10$ cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$} = 1+0.2+0.3 = 1.5$

Multi-Ported Cache-Style SRAM Latency

- Previous calculation had hidden constant
  - Number of ports **P**
- Recalculate latency components
  - Decoder: $\propto \log_2 M$ (unchanged)
  - Wordlines: $\propto$ 2NL**P** (cross 2N**P** bitlines)
  - Bitlines: $\propto$ ML**P** (cross M**P** wordlines)
  - Muxes + sense-amps: constant (unchanged)



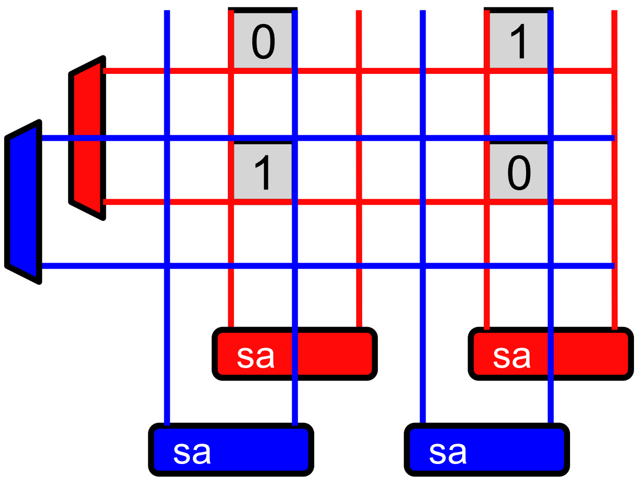
  - Latency: $\propto$ (2N+M)L**P**
  - **Latency: $\propto$ √#bits * #ports**



- How does latency scale?

Multi-Ported Cache-Style SRAM Power

- Same four components for power
  - $P_{dynamic} = \mathbf{C} * V_{DD}^2 * f$, what is C?
  - Decoder: $\propto \log_2 M$
  - Wordlines: $\propto$ 2NLP
    – Huge C per wordline (drives 2N gates)
    + But only one ever high at any time (overall consumption low)
  - Bitlines: $\propto$ MLP
    – C lower than wordlines, but large
    + $V_{swing} << V_{DD}$ $(C * V_{swing}^2 * f)$
  - Muxes + **sense-amps**: constant
  - 32KB SRAM: sense-amps are 60–70%
- How does power scale?



Multi-Porting an SRAM

- Why multi-porting?
  - Multiple accesses per cycle
- **True multi-porting** (physically adding a port) not good
  - \+ Any combination of accesses will work
  - – Increases access latency, energy $\propto$ P, area $\propto$ P^2
- Another option: **pipelining**
  - Timeshare single port on clock edges (wave pipelining: no latches)
  - \+ Negligible area, latency, energy increase
  - – Not scalable beyond 2 ports
- Yet another option: **replication**
  - Don’t laugh: used for register files, even caches (Alpha 21164)
  - Smaller and faster than true multi-porting $2*P^2 < (2*P)^2$
  - \+ Adds read bandwidth, any combination of reads will work
  - – Doesn’t add write bandwidth, not really scalable beyond 2 ports

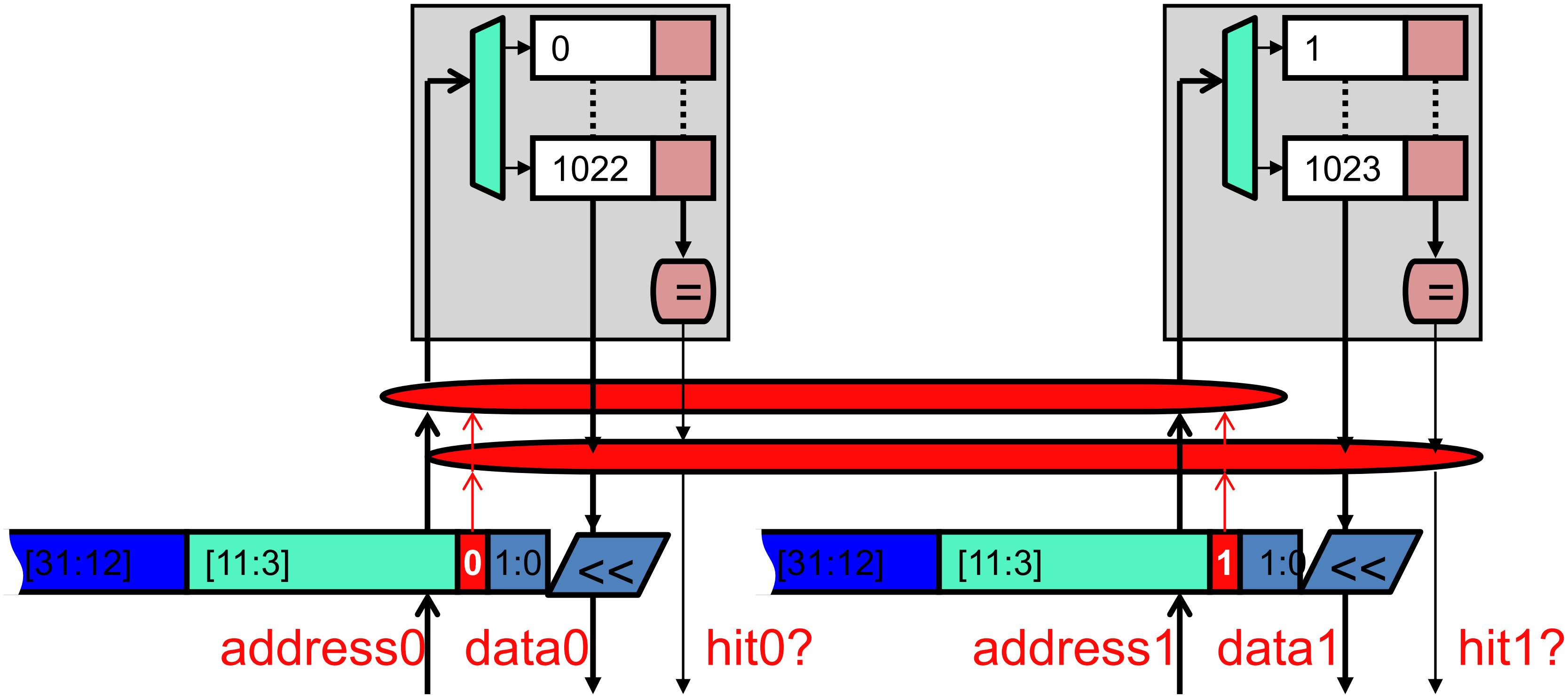
Banking an SRAM

- Still yet another option: **banking (inter-leaving)**
  - Divide SRAM into banks
  - Allow parallel access to different banks
  - Two accesses to same bank? **bank-conflict**, one waits
  - Low area, latency overhead for routing requests to banks
  - Few bank conflicts given sufficient number of banks
    - Rule of thumb: N simultaneous accesses $\rightarrow$ 2N banks

  - How to divide words among banks?
    - **Round robin**: using address LSB (least significant bits)
    - Example: 16 word RAM divided into 4 banks
    - **b0**: 0,4,8,12; **b1**: 1,5,9,13; **b2**: 2,6,10,14; **b3**: 3,7,11,15
    - Why? Spatial locality

A Banked Cache

- **Banking** a cache
  - Simple: bank SRAMs
  - Which address bits determine bank? LSB of index
  - **Bank network** assigns accesses to banks, resolves conflicts
    – Adds some latency too

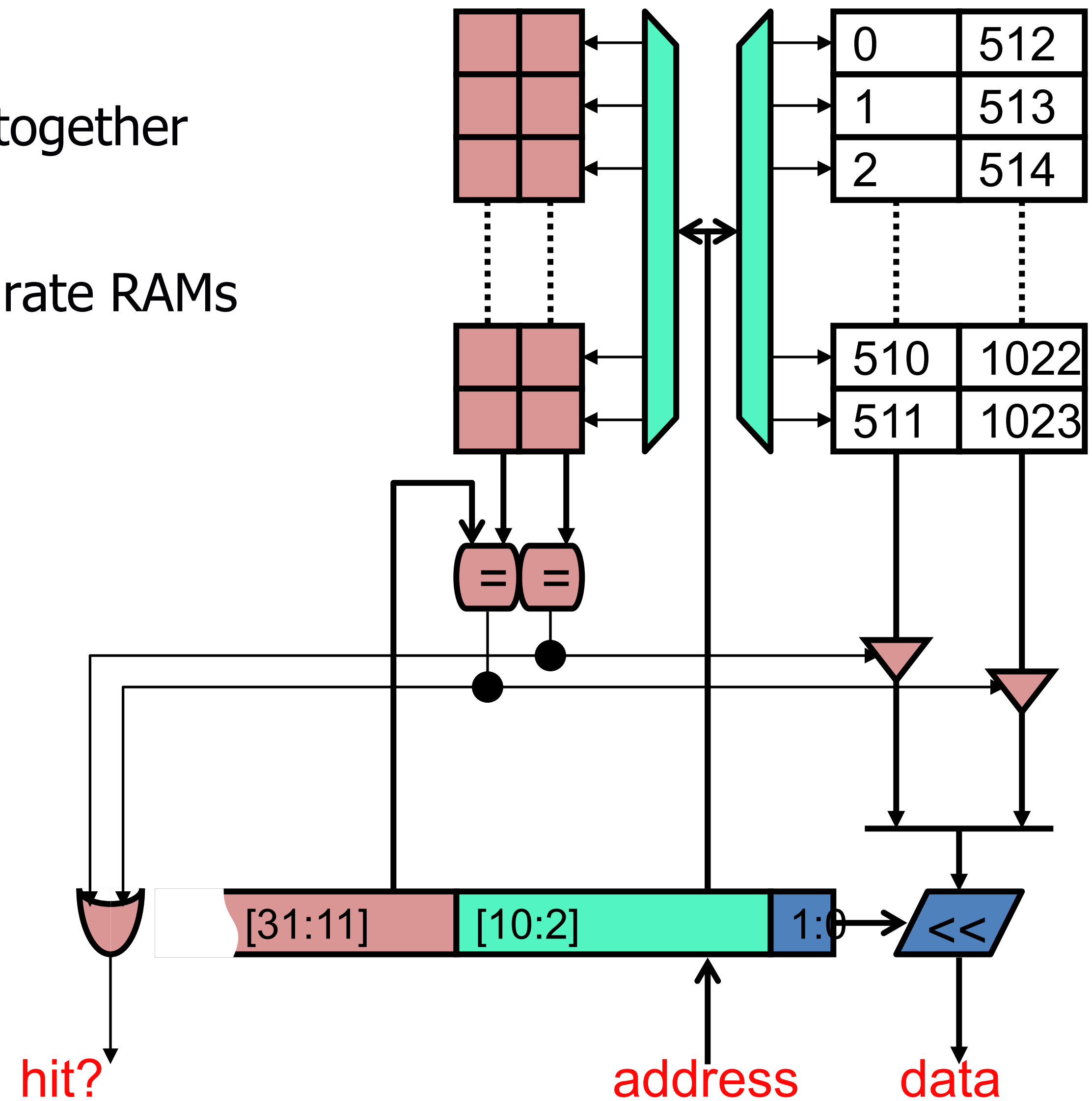


SRAM Summary

- Large storage arrays are not implemented “digitally”
- SRAM implementation exploits analog transistor properties
  - Inverter pair bits much smaller than latch/flip-flop bits
  - Wordline/bitline arrangement gives simple “grid-like” routing
  - Basic understanding of read, write, read/write ports
    - Wordlines select words
    - Overwhelm inverter-pair to write
    - Drain pre-charged line or swing voltage to read
  - Latency proportional to $\sqrt{\#bits} * \#ports$

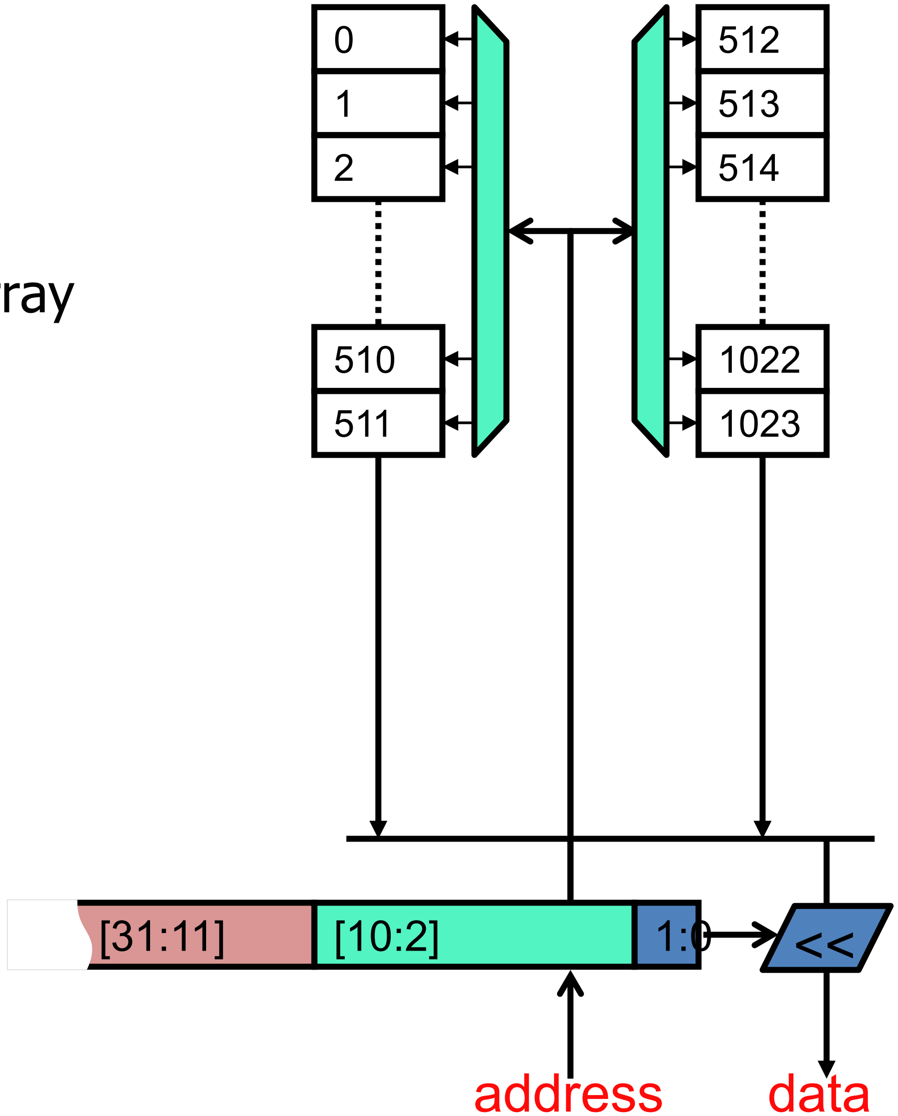
Aside: Physical Cache Layout I

- Logical layout
  - Data and tags mixed together
- Physical layout
  - Data and tags in separate RAMs



Physical Cache Layout II

- Logical layout
  - Data array is monolithic
- Physical layout
  - Each data "way" in separate array

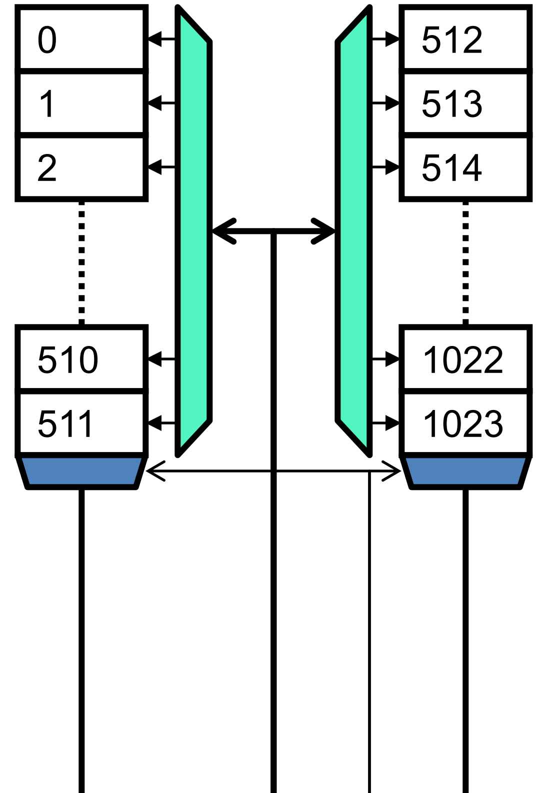


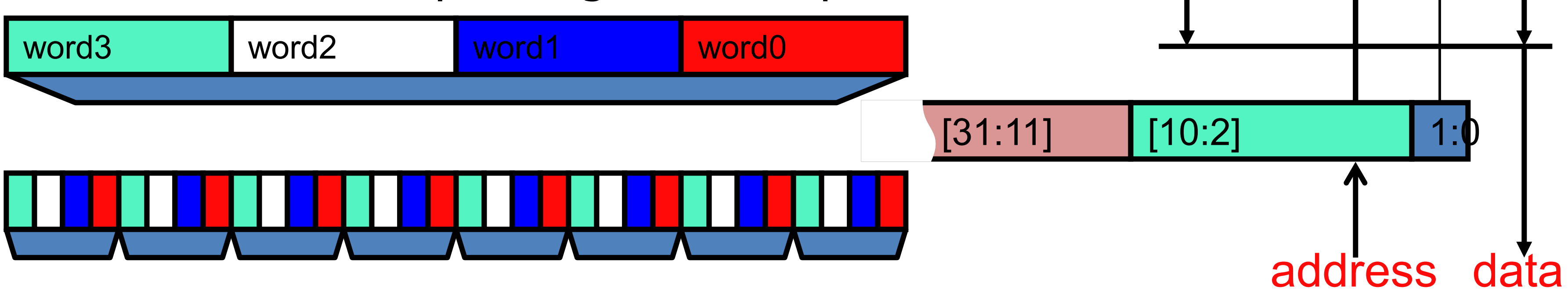
Physical Cache Layout III

- Logical layout
  - Data blocks are contiguous
- Physical layout
  - Only if full block needed on read
    - E.g., I$ (read consecutive words)
    - E.g., L2 (read block to fill D$,I$)
  - For D$ (access size is 1 word)…
  - Words in same data blocks are bit-interleaved
    - $\text{Word}_0.\text{bit}_0$ adjacent to $\text{word}_1.\text{bit}_0$

    \+ Builds word selection logic into array

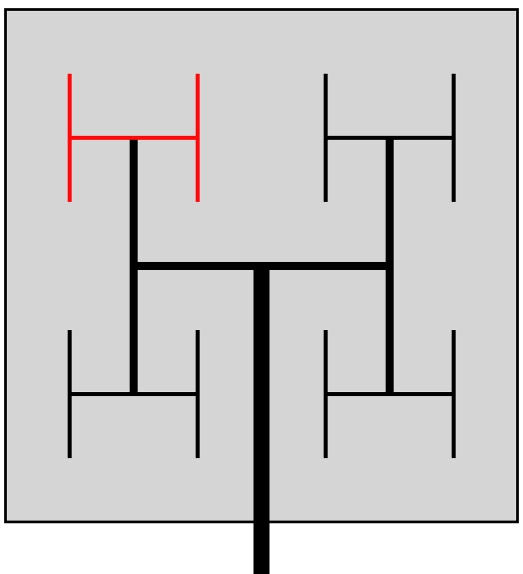
    \+ Avoids duplicating sense-amps/muxes

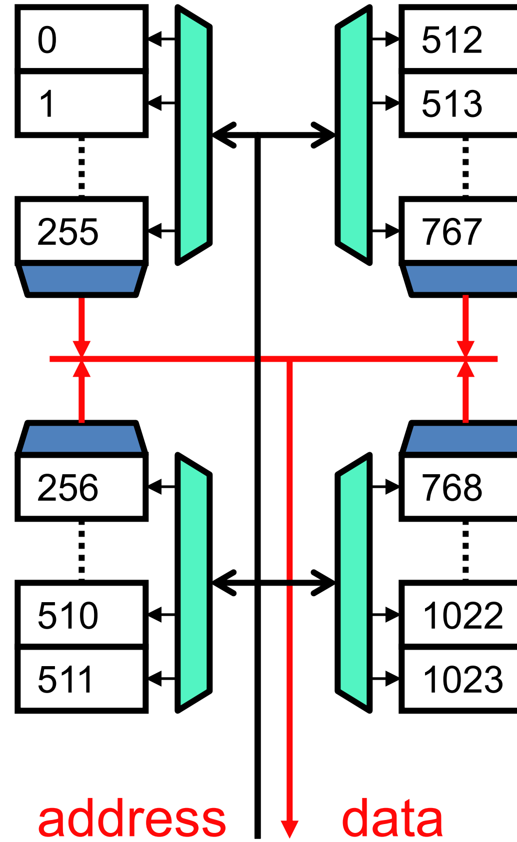




Physical Cache Layout IV

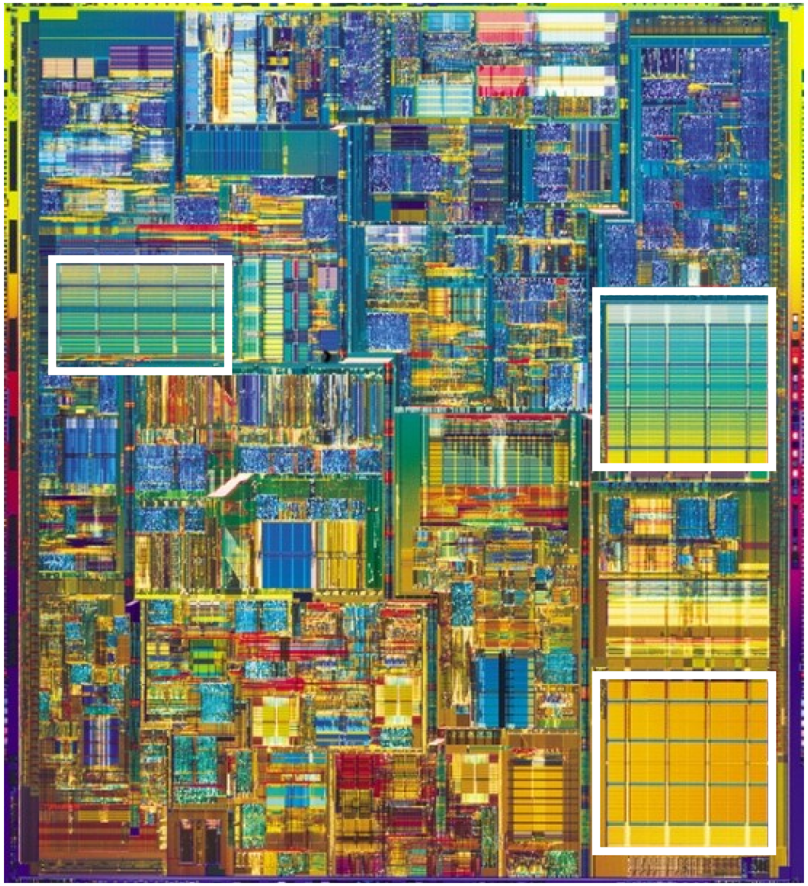
- Logical layout
  - Arrays are vertically contiguous
- Physical layout
  - Vertical partitioning to minimize wire lengths
  - **H-tree**: horizontal/vertical partitioning layout
    - Applied recursively
    - Each node looks like an H



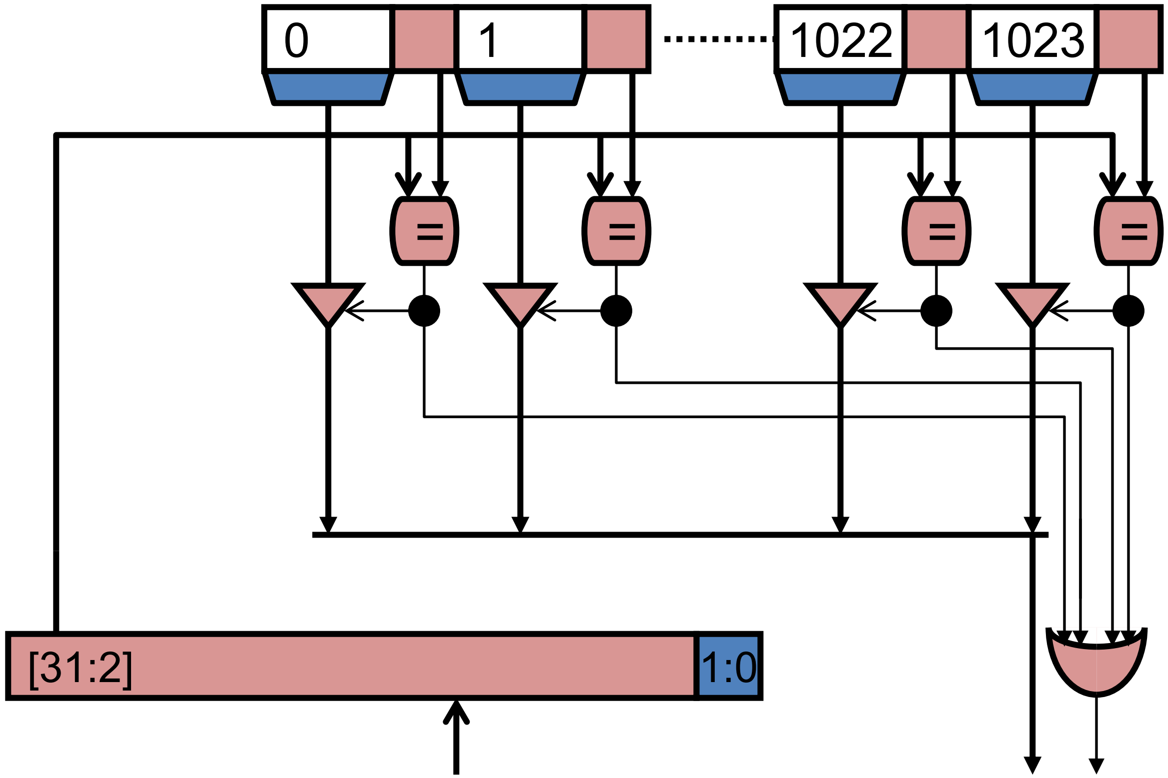


Physical Cache Layout

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



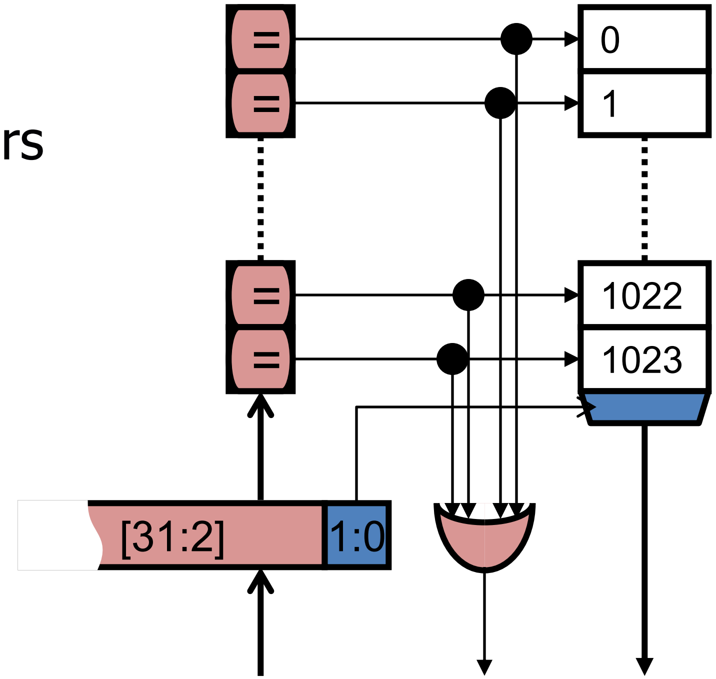
Full-Associativity



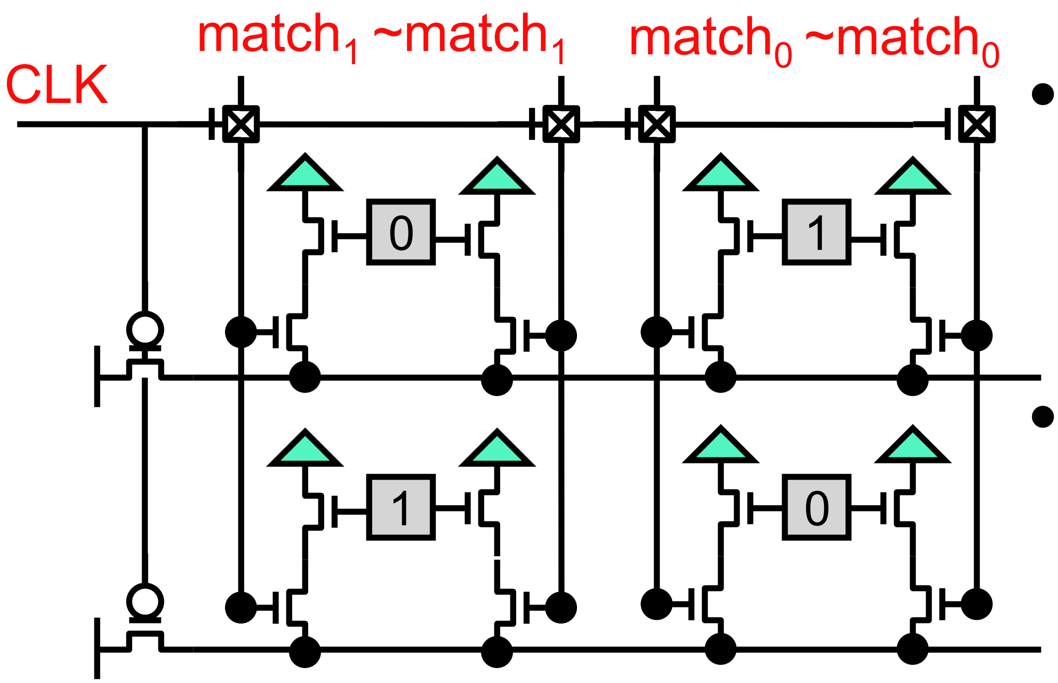
- How to implement full (or at least high) associativity?
  - 1K tag matches? unavoidable, but at least tags are small
  - 1K data reads? Terribly inefficient

Full-Associativity with CAMs

- **CAM**: content associative memory
  - Array of words with built-in comparators
  - Matchlines instead of bitlines
  - Output is “one-hot” encoding of match
- FA cache?
  - Tags as CAM
  - Data as RAM
- **Hardware is not software**
  - No such thing as software CAM

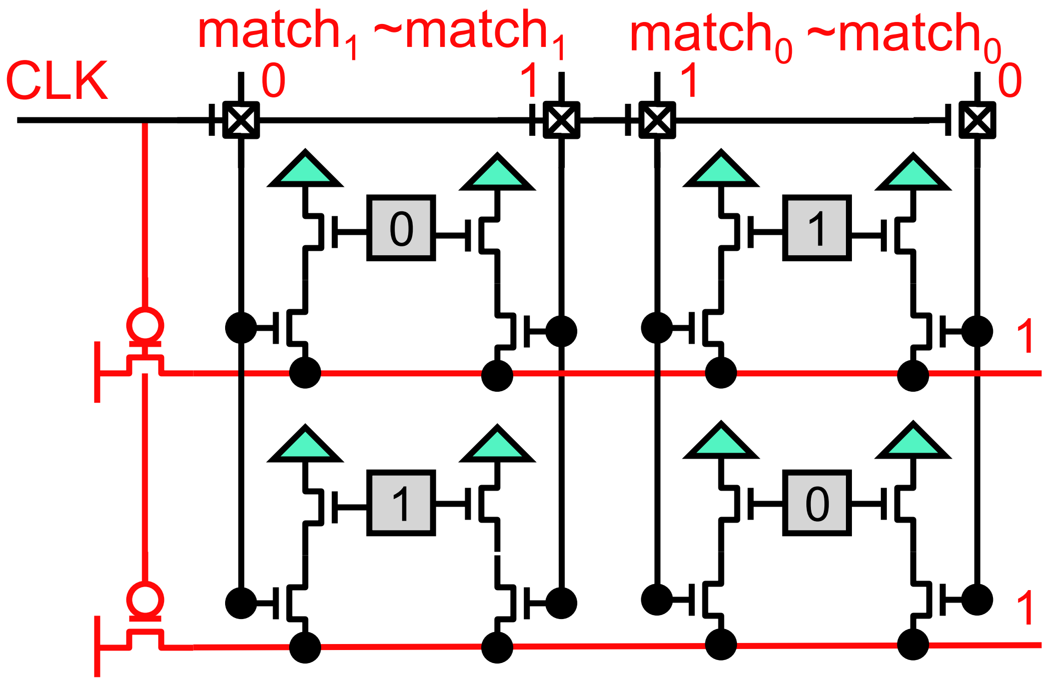


CAM Circuit



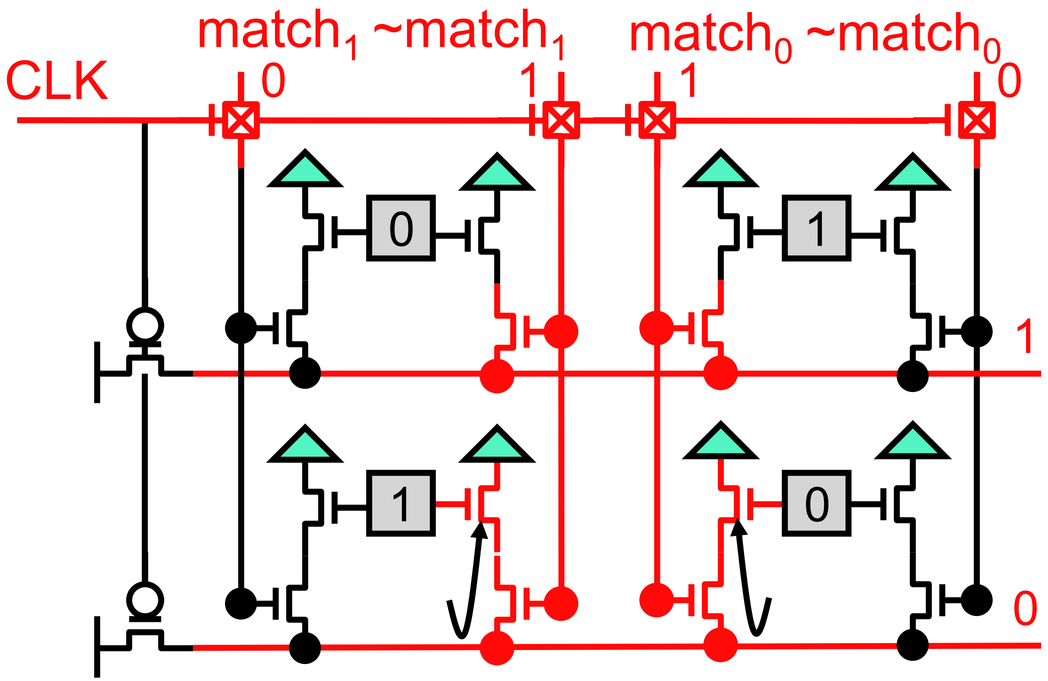
- CAM: reverse RAM
  - Bitlines are inputs
    - Called **matchlines**
  - Wordlines are outputs
- Two phase match
  - Phase I: clk=0
    - Pre-charge wordlines
  - Phase II: clk=1
    - Enable matchlines
    - Non-matching bits dis-charge wordlines

CAM Circuit In Action: Phase I



- Phase I: clk=0
  - Pre-charge wordlines

CAM Circuit In Action: Phase II

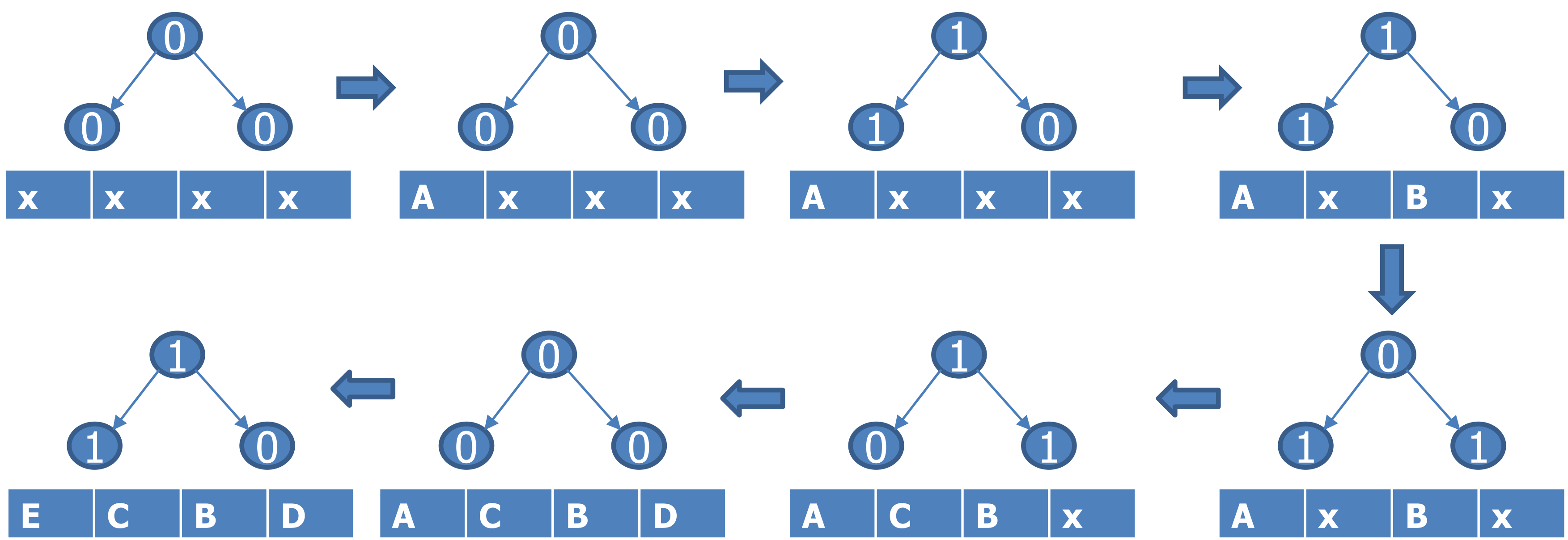


- Phase II: clk=1
  - Enable matchlines
    - Note: bits flipped
  - Non-matching bit discharges wordline
    - ANDs matches
    - NORs non-matches
  - Similar technique for doing a fast OR for hit detection

tree-PLRU

Access A – all 0’s, so place in left subtree, left entry

Because A in left subtree, left entry, update to 1 → 1 so next request goes to right subtree, and next request to left subtree goes to right entry

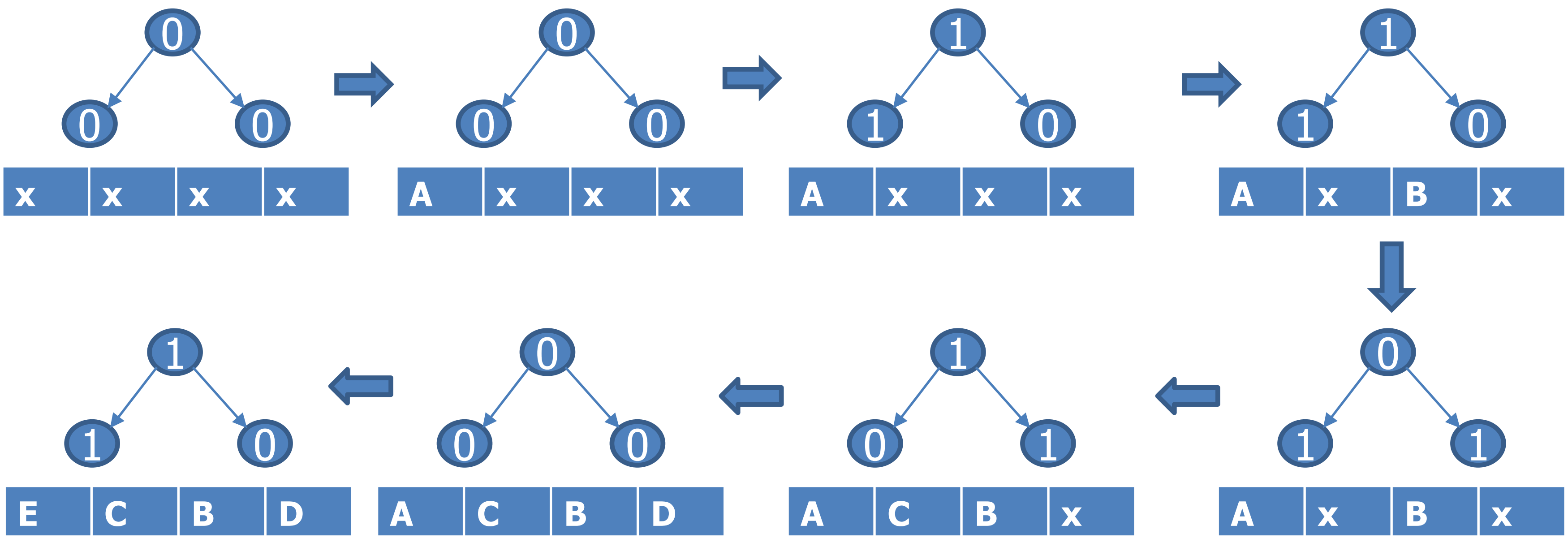


Access D → go to right subtree, because root == 1, subtree == 1, so place in right entry + update state

Access C → go to left subtree, because root == 0, subtree == 1, place in right entry + update state

Access B → go to right subtree, because right subtree == 0, place in left entry (subtree → 1) ; next request to left (root → 0)

tree-PLRU



Access E → go to left subtree, because root == 0, subtree == 0, place in left entry (evict A) + update state

Fancier Modern Replacement Policies

- **Least Frequently Used (LFU)**
  - Tracks which cache lines are frequently accessed (counter per line)
    - "Access Frequency"
  - Evict least frequently accessed line
    - **Scan Resistant: Avoids caching streaming data**
  - No MRU/LRU data, only frequency → **not thrash resistant**
  - May be expensive to determine which line to evict if not clever

Fancier Modern Replacement Policies

- **Dynamic Insertion Policy (DIP)**
  - Goal: predict when a cache line won't be reused "anymore"
  - Dynamically decide to change insertion policy between:
    - Always inserting head of RRIP chain (next slide)
    - Inserting majority of blocks at tail of RRIP chain
  - Switching to head retains some of working set → **thrash resistant**
  - Can't tell scans apart from other accesses → **not scan resistant**

Can we get the best of both worlds?

SRRIP Example

3 I	**2 a1**	**2 a1**	**2 a1**	**2 a1**	**0 a1**	**0 a1**	**1 a1**	**1 a1**	**0 a1**
3 I	3 I	2 a2	0 a2	0 a2	0 a2	0 a2	1 a2	1 a2	1 a2
3 I	3 I	3 I	3 I	3 I	2 b1	2 b1	2 b3	2 b3	2 b3
3 I	3 I	3 I	3 I	3 I	3 I	2 b2	3 b2	2 b4	2 b4
a1	a2	a2	a1	b1	b2	b3	b4	a1	
MISS	MISS	HIT	HIT	MISS	MISS	MISS	MISS	HIT	

Evaluation Methods

Evaluation Methods

- The three system evaluation methodologies
  1. Analytic modeling
  2. Software simulation
  3. Hardware prototyping and measurement

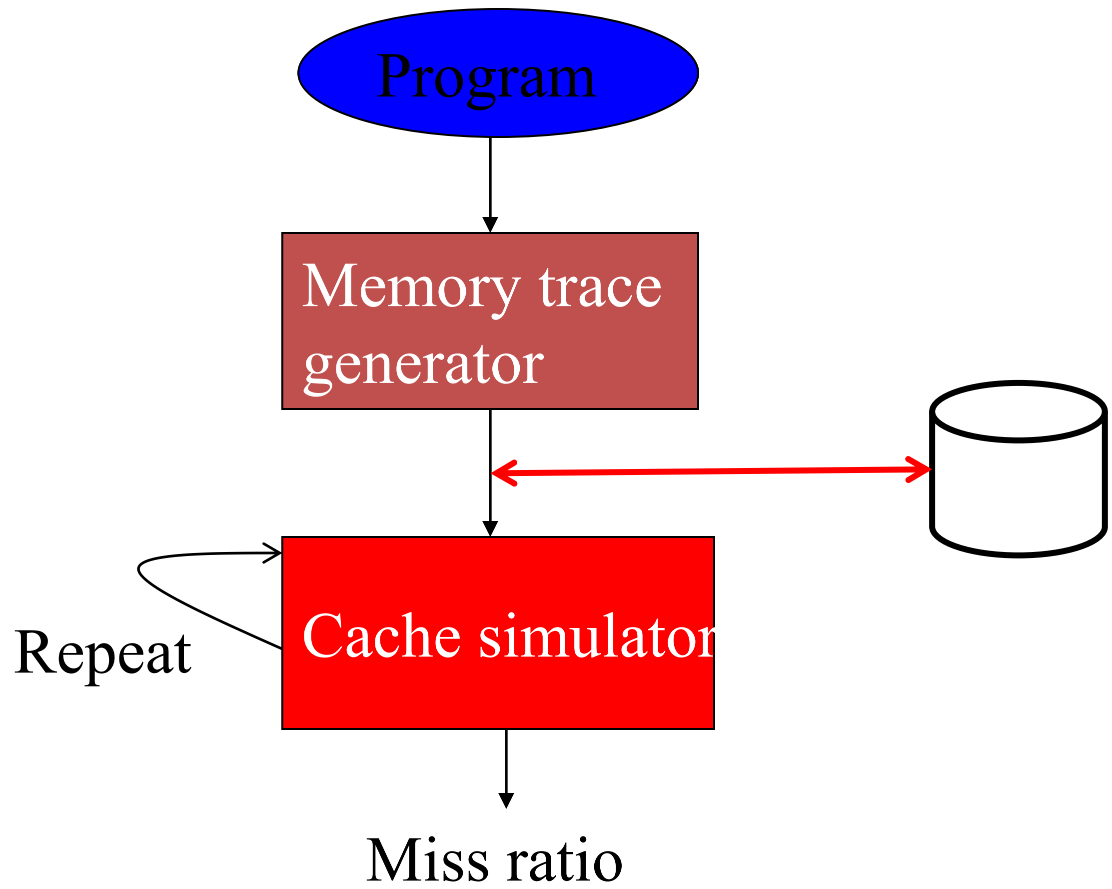
Methods: Hardware Counters

- See Clark, TOCS 1983
  - ✓ accurate
  - ✓ realistic workloads, system + user + others
  - ✗ difficult, why?
  - ✗ must first have the machine
  - ✗ hard to vary cache parameters
  - ✗ experiments not deterministic
    - ✗ use statistics!
      - ✗ take multiple measurements
      - ✗ compute mean and confidence measures
- Most modern processors have built-in hardware counters

Methods: Analytic Models

- Mathematical expressions
  - ✓ insightful: can vary parameters
  - ✓ fast
  - ✗ absolute accuracy suspect for models with few parameters
  - ✗ hard to determine parameter values
  - ✗ difficult to evaluate cache interaction with system
  - ✗ bursty behavior hard to evaluate

Methods: Trace-Driven Simulation



Methods: Trace-Driven Simulation

- ✓ experiments repeatable
- ✓ can be accurate
- ✓ much recent progress
- ✗ reasonable traces are very large (gigabytes?)
- ✗ simulation is time consuming
- ✗ hard to say if traces are representative
- ✗ don't directly capture speculative execution
- ✗ don't model interaction with system

- ✗ Widely used in industry

Methods: Execution-Driven Simulation

- Simulate the program execution
  - simulates each instruction's execution on the computer
  - model processor, memory hierarchy, peripherals, etc.
  - ✓ reports execution time
    - ✓ accounts for all system interactions
  - ✓ no need to generate/store trace
  - ✗ much more complicated simulation model
  - ✗ time-consuming but good programming can help
  - ✗ multi-threaded programs exhibit variability
- Very common in academia today

- Watch out for repeatability in multithreaded workloads

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