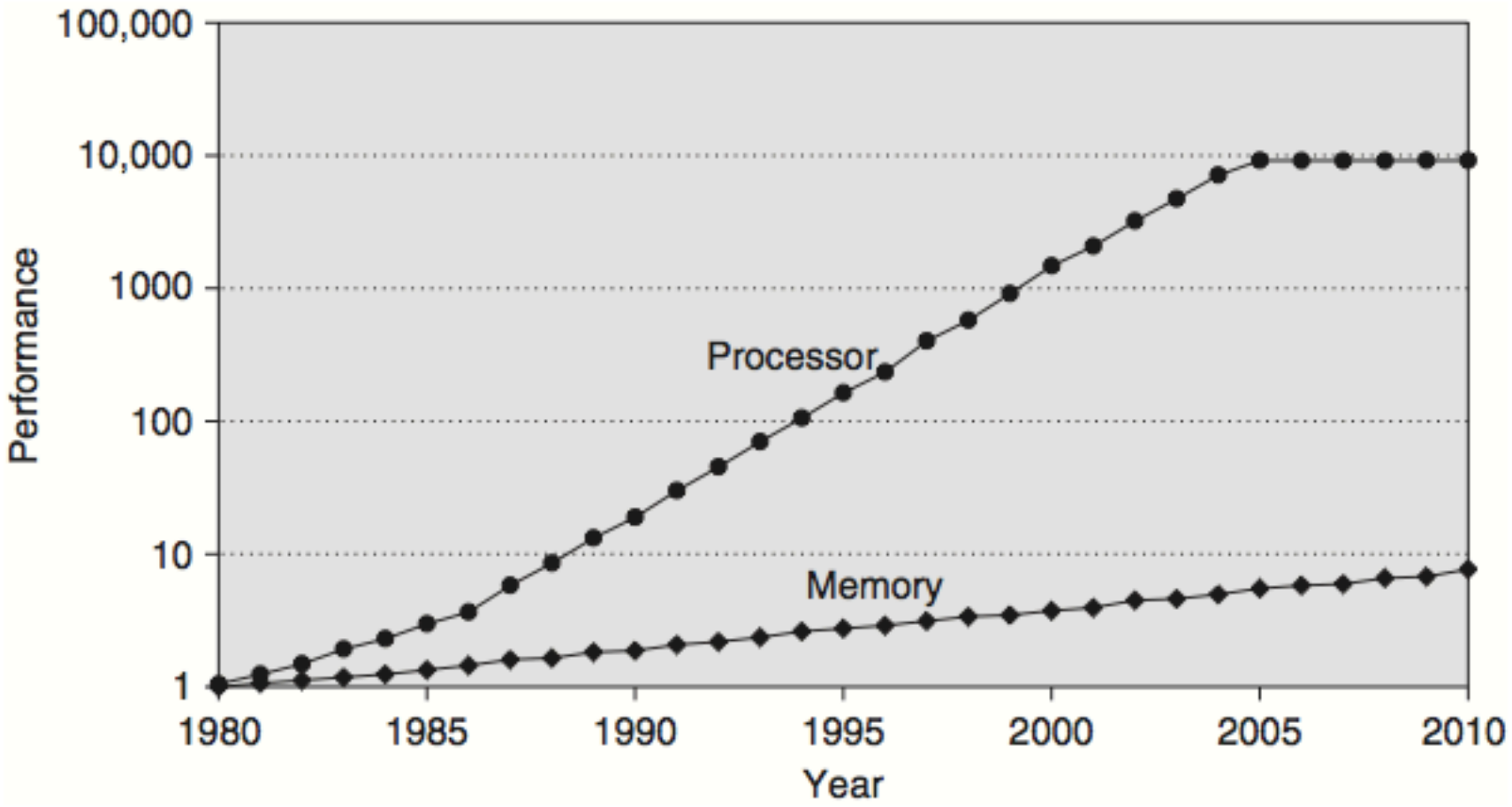
CSE 548: Computer Systems Architecture

Memory Hierarchy

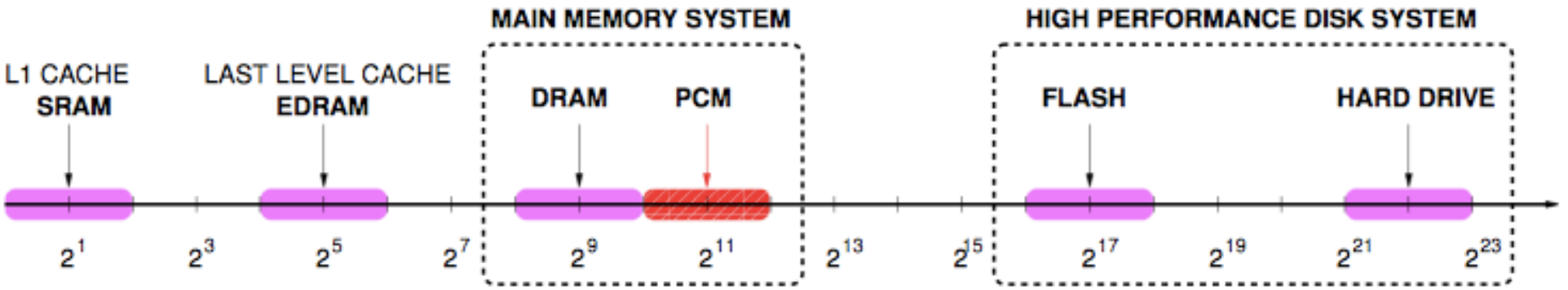
Luis Ceze, Spring 2017

based on slides from friends at UPenn, UIUC, UW, MIT, CMU.

Ooops --- The “Memory Wall”



- Yet, need to get data in-and-out of processors!



Typical Access Latency (in terms of processor cycles for a 4 GHz processor)

Bandwidth

SRAM - 10-1000GB/sec
DRAM - ~10GB/sec
Disk - 100MB/sec (0.1 GB/sec) - sequential access only

Known From the Beginning

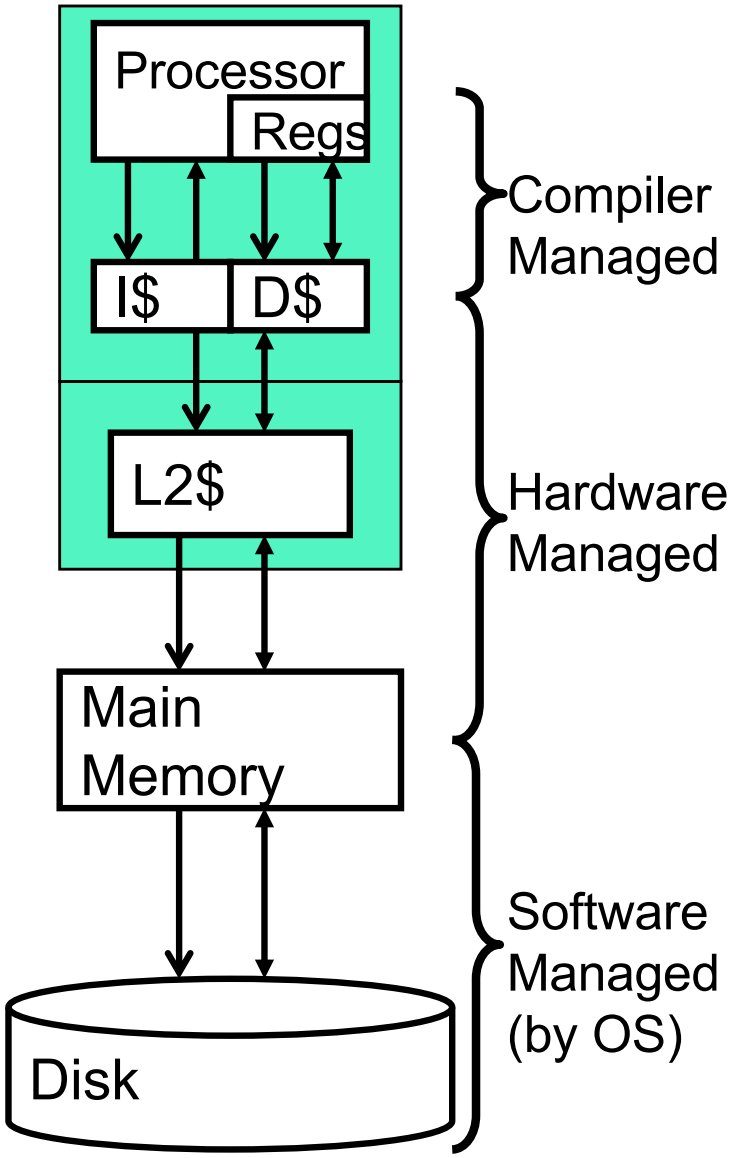
"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, VonNeumann

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

IAS memo 1946

Memory Hierarchy



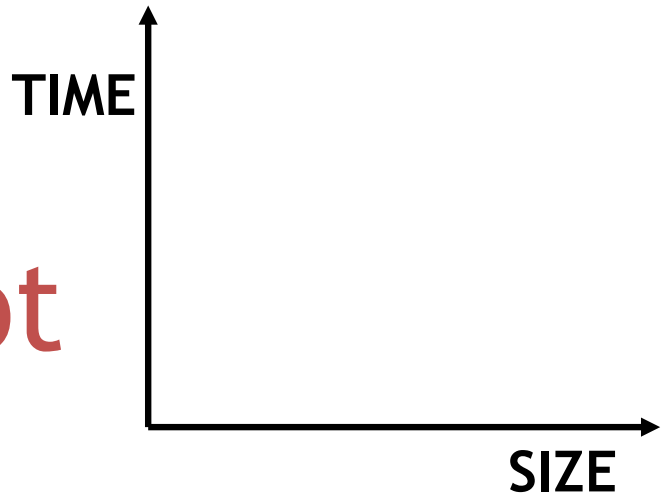
- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$) (SMC?)
  - Typically 8KB to 64KB each
- 2nd level: **Second-level cache** (L2$)
  - On-chip, certainly on-package (with CPU)
  - Made of SRAM (same circuit type as CPU)
  - Typically 512KB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
- 4th level: **disk (swap and files)**

How does execution time grow with SIZE?

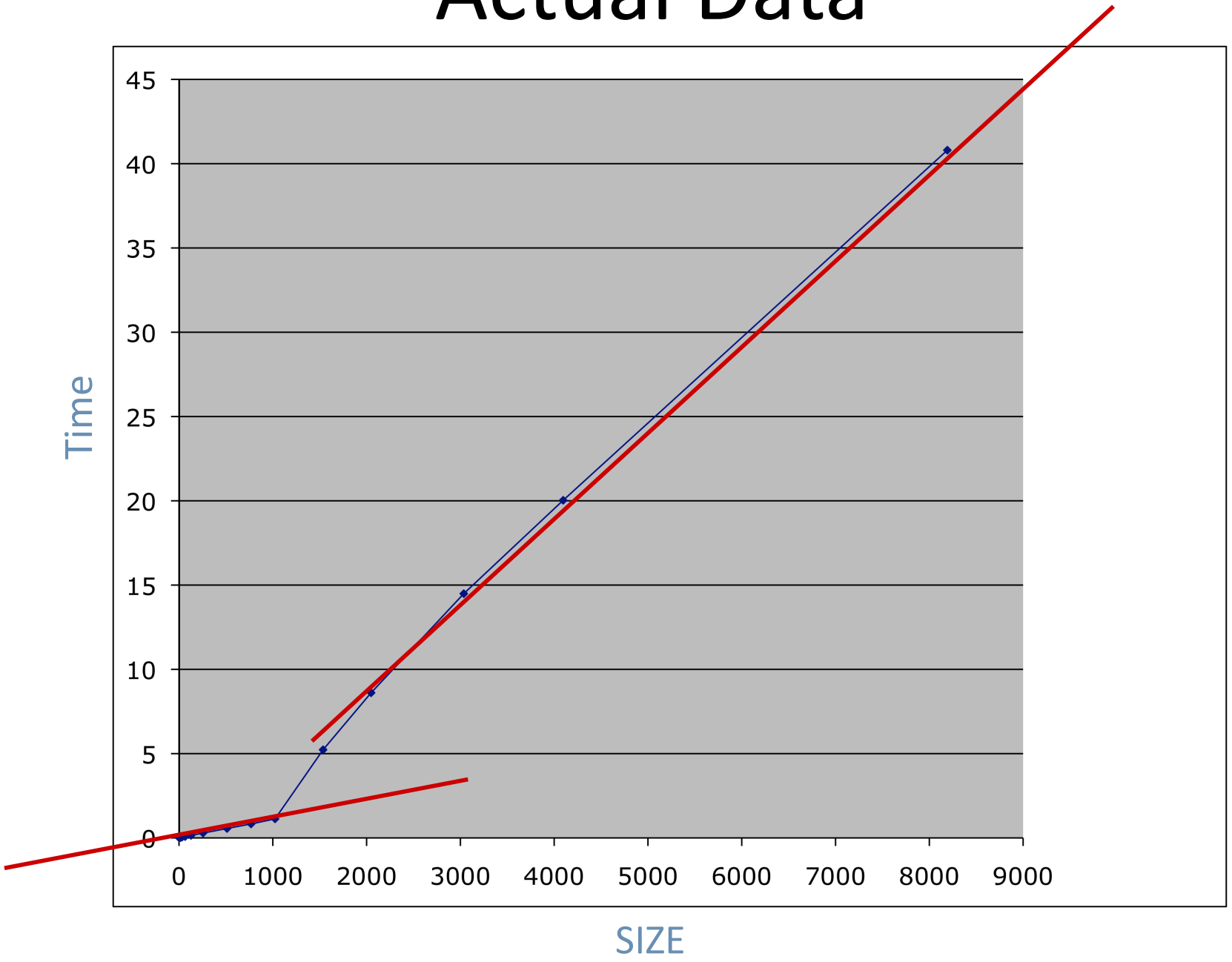
```
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
   for (int j = 0 ; j < SIZE ; ++ j) {
    A += array[j];
   }
}
```

Plot



Actual Data


45
40
35
30
25
20
15
10
5
0
Time
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
SIZE

How do programs use the memory hierarchy?

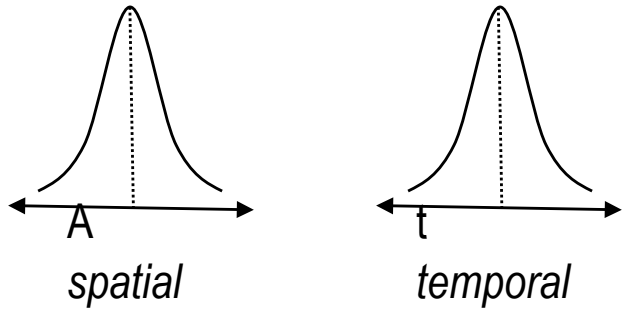
Locality to the Rescue

- **Locality of memory references**
  - Property of real programs, few exceptions

- **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, how?

- *Holds for data and instructions.*
- *Why? Where does it come from?*



Example: Locality?

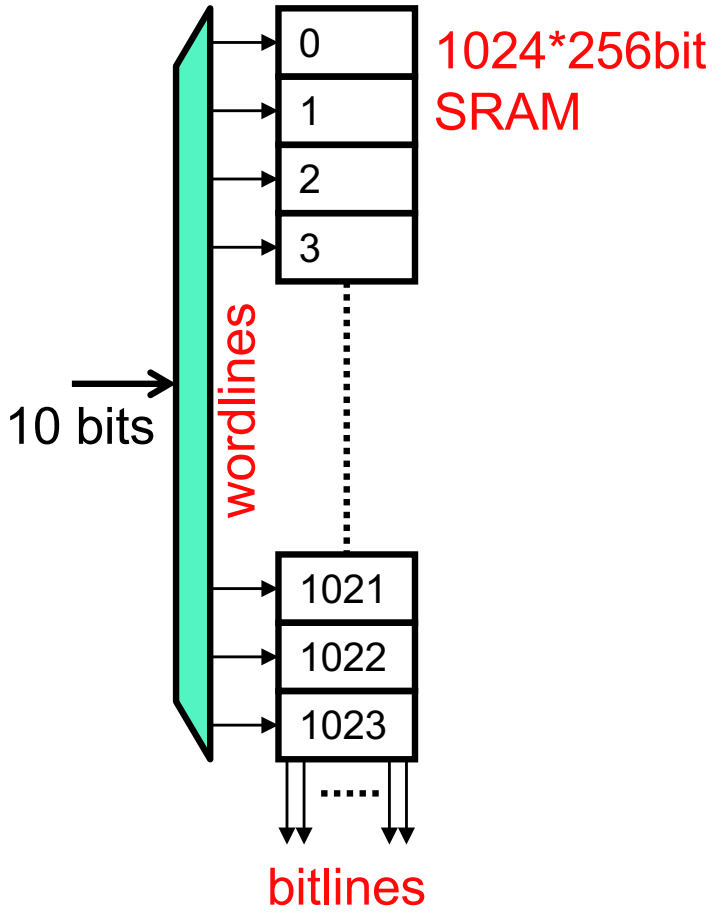
```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

- Data:
  - Temporal: **`sum`** referenced in each iteration
  - Spatial: array **`a[]`** accessed in stride-1 pattern
- Instructions:
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

- Being able to assess the locality of code is a crucial skill for a programmer

Basic Memory Array Structure

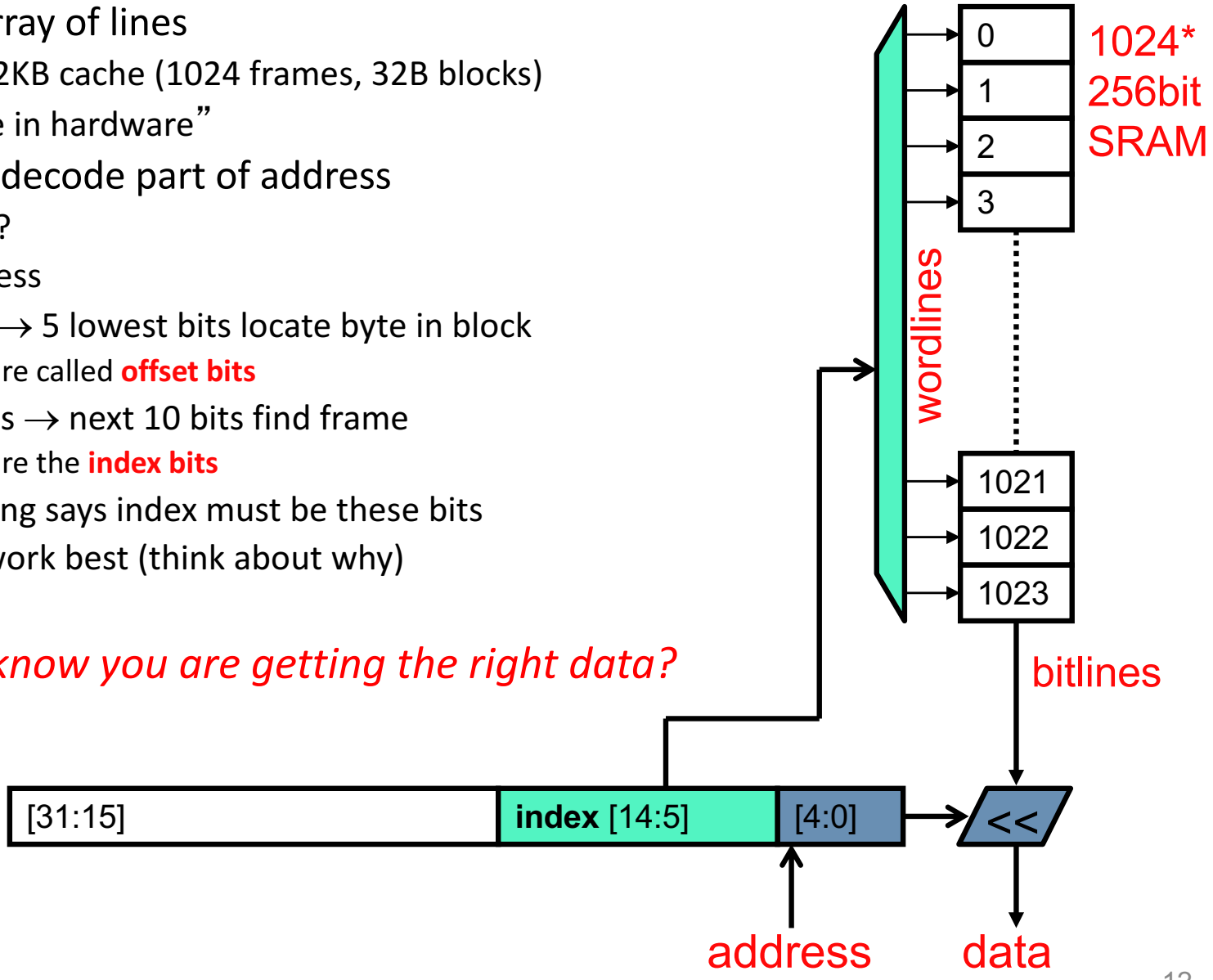
- Number of entries
  - 2^n, where n is number of address bits
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
  - One-bit address travels on "wordlines"
- Size of entries
  - Width of data accessed
  - Data travels on "bitlines"
  - 256 bits (32 bytes) in example
- How do we use a cache entry?
  - Divide up memory in blocks too.



Caches: Finding Data via Indexing

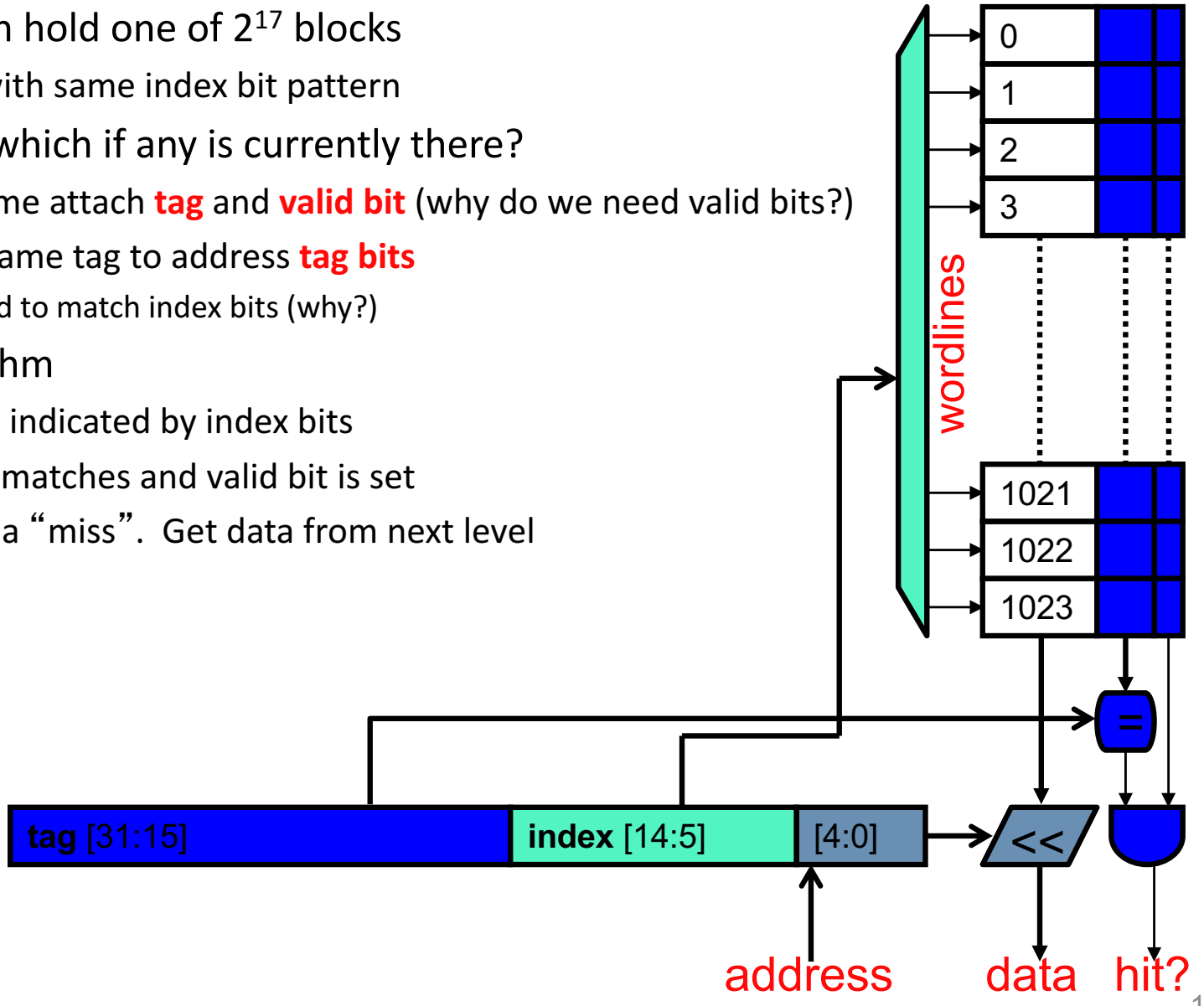
- Basic cache: array of lines
  - Example: 32KB cache (1024 frames, 32B blocks)
  - "Hash table in hardware"
- To find frame: decode part of address
  - Which part?
  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block
    - These are called **offset bits**
  - 1024 frames → next 10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

- *How do you know you are getting the right data?*



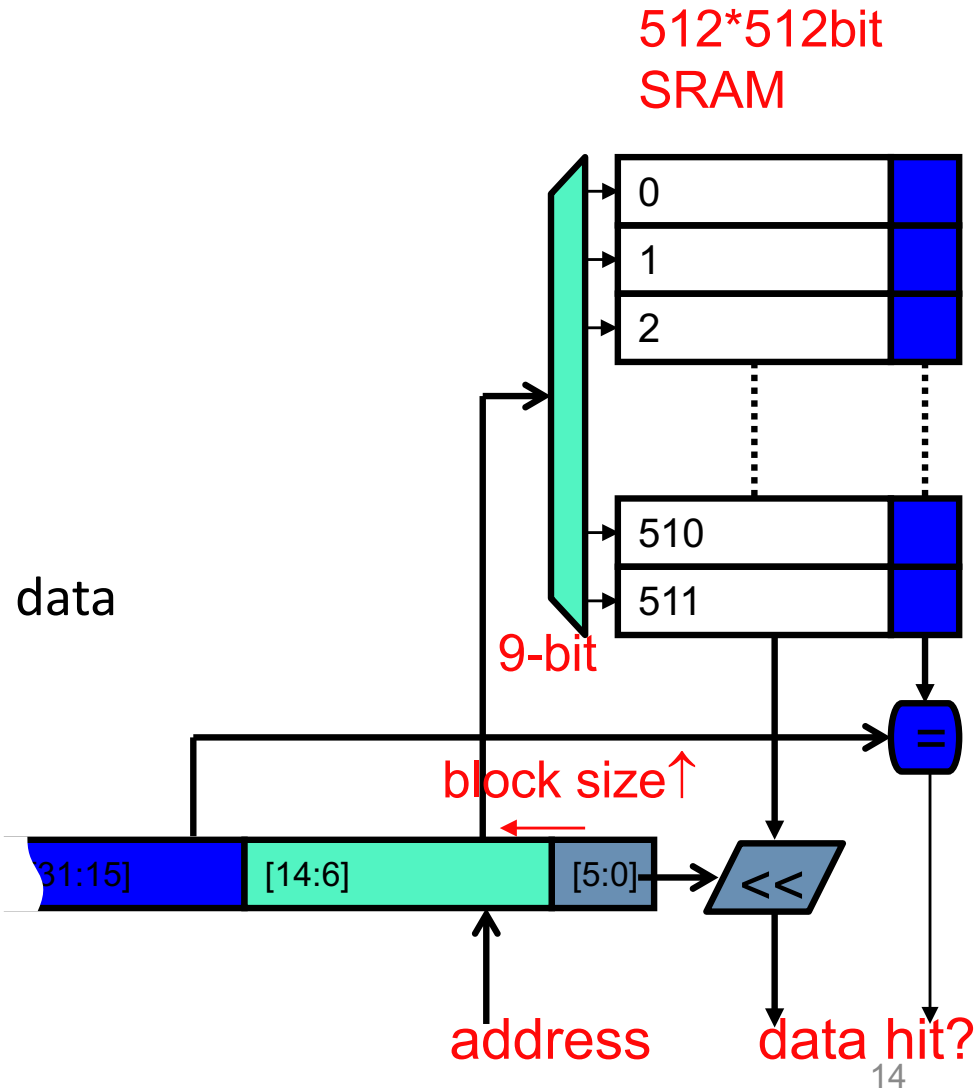
Knowing that You Found It: Tags

- 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** (why do we need valid bits?)
  - 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". Get data from next level



Block Size

- Increasing **block size**
  - Exploit **spatial locality**
  - Notice index/offset bits change
  - Tag remain the same
- Ramifications
  - \+ Reduce $\%_{miss}$
  - \+ Reduce tag overhead *(why?)*
  - – Potentially useless data transfer
  - – Premature replacement of useful data



A puzzle.

- What can you infer from this:

- Cache starts *empty*
- Access (addr, hit/miss) stream:

- (10, miss), (11, hit), (12, miss)

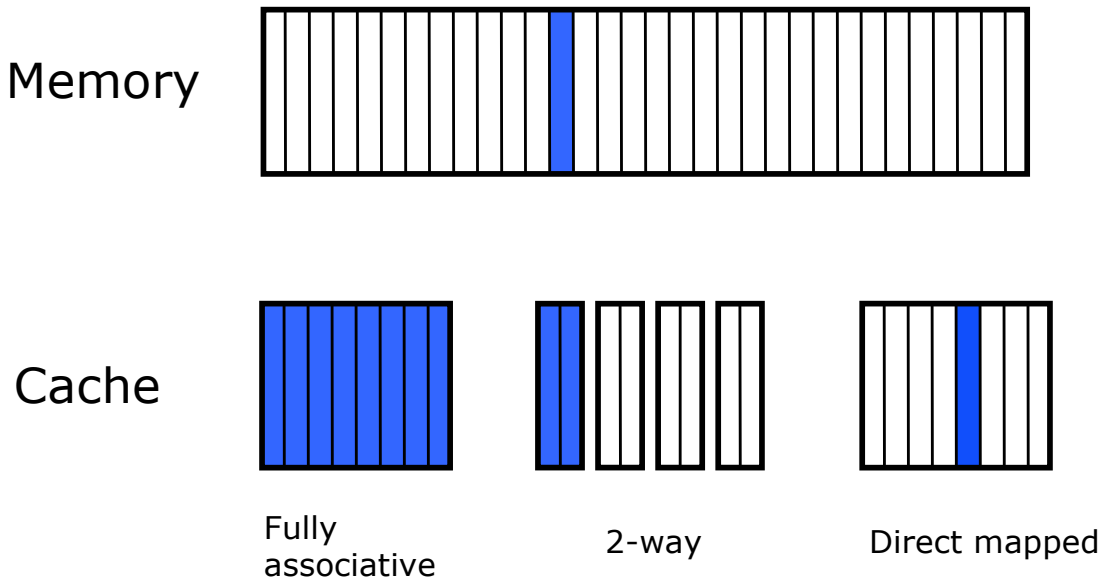
block size >= 2 bytes

block size < 8 bytes

Where can data go ?

- The cache is essentially a table with a tag.
- Where can data go in this table?

Anywhere? That would be nice!

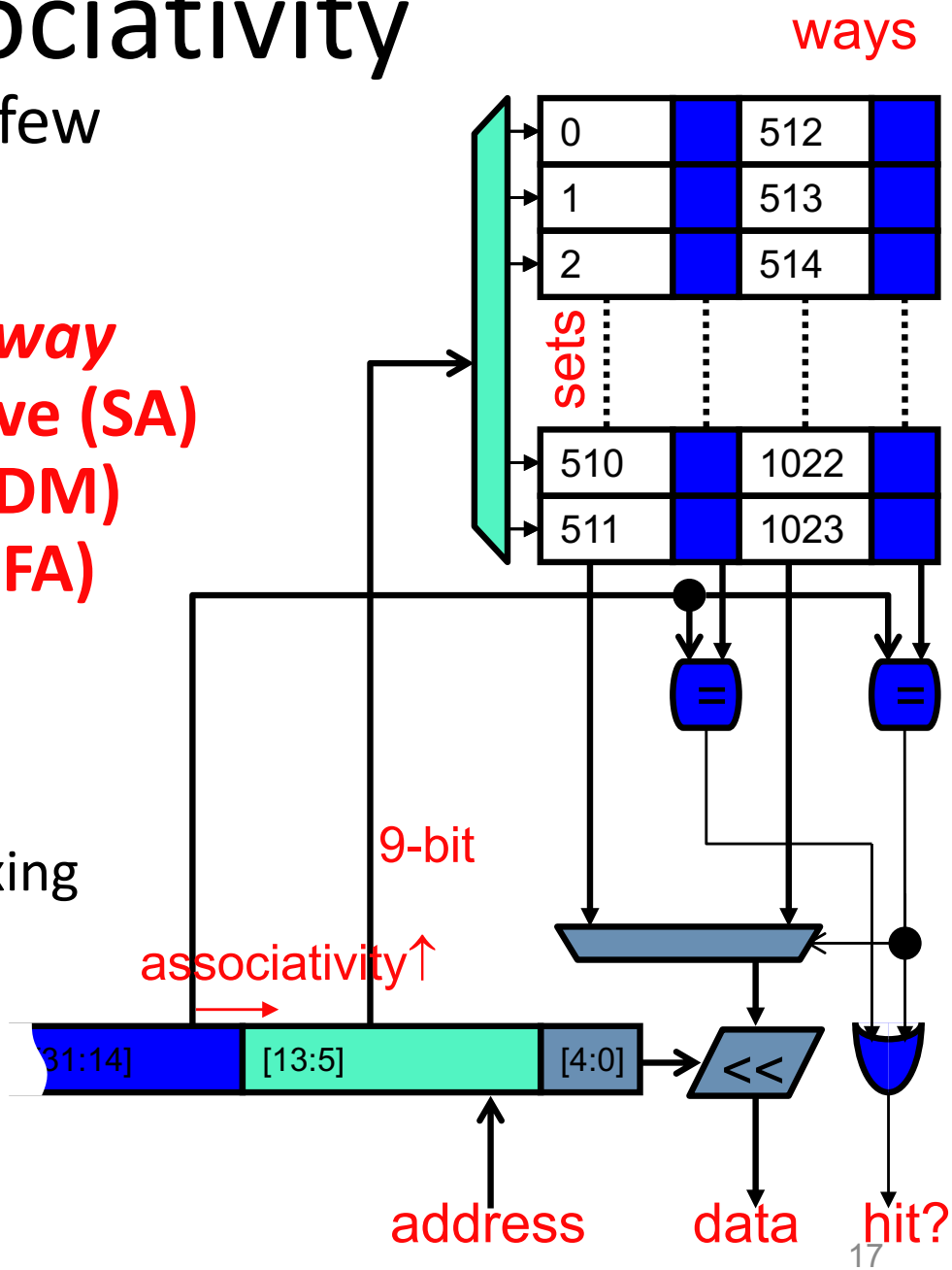


Set-Associativity

- 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 tag match & muxing

- Note: valid bit not showr



Cache (or set) is full, now what?

- **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
- **Belady's**: replace block that will be used furthest in future
    - Unachievable optimum

Another puzzle.

- What can you infer from this:

- Cache starts *empty*
- Access (addr, hit/miss) stream

- (10, miss); (12, miss); (10, miss)

12 is not in the same block as 10

12's block replaced 10's block

direct-mapped cache

Impact of Cache and Block Size

- Cache Size
  - Effect on miss rate?
  - Effect on hit time?
- Block Size
  - Effect on miss rate?
  - Effect on miss penalty?

Block Size and Miss Penalty

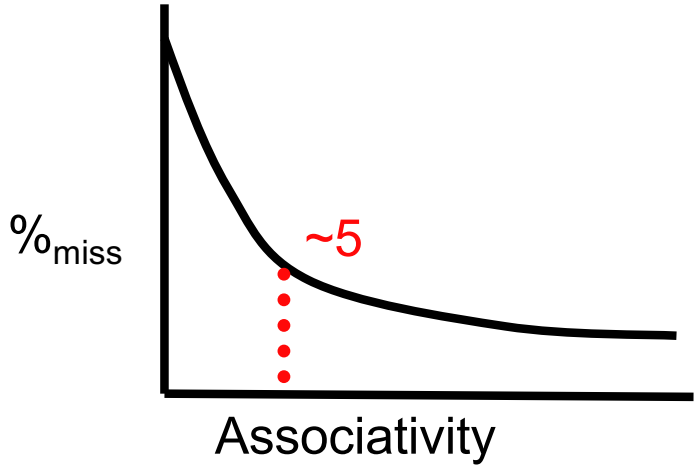
- Does increasing block size increase t_{miss}?
  - Don't larger blocks take longer to read, transfer, and fill?
  - They do, but…
- t_{miss} of an isolated miss is not affected
  - **Critical Word First / Early Restart (CRF/ER)**
  - Requested word fetched first, pipeline restarts immediately
  - Remaining words in block transferred/filled in the background
- t_{miss}'es of a cluster of misses will suffer
  - Reads/transfers/fills of two misses can't happen at the same time
  - Latencies can start to pile up
  - This is a bandwidth problem

Impact of Associativity

  - Direct-mapped, set associative, or fully associative?
- Total Cache Size (tags+data)?
- Miss rate?
- Hit time?
- Miss Penalty?

Associativity And Performance

- Higher associative caches
  - + Have better (lower) $\%_{miss}$
    - Diminishing returns
  - – However t_{hit} increases
    - The more associative, the slower
  - – What about t_{avg}?



- Block-size and number of sets should be powers of two
  - *why?*
- What about set-associativity (e.g., 3, 5-way) ?

Classifying Misses: 3C Model (Hill)

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
  - **Conflict**: miss caused because cache associativity is too low
  - **(Coherence)**: miss due to external invalidations
    - Only in shared memory multiprocessors (later)
- Calculated by multiple simulations
  - Simulate infinite cache, fully-associative cache, normal cache
  - Subtract to find each count

Miss Rate: ABC

- Why do we care about 3C miss model?
  - So that we know what to do to eliminate misses
  - If you don't have conflict misses, increasing associativity won't help

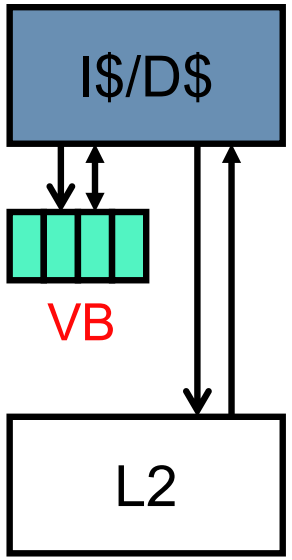
- **Associativity**
  - \+ Decreases conflict misses
  - – Increases $\text{latency}_{\text{hit}}$
- **Block size**
  - – Increases conflict/capacity misses (fewer frames)
  - \+ Decreases compulsory/capacity misses (spatial locality)
  - – No significant effect on $\text{latency}_{\text{hit}}$
- **Capacity**
  - \+ Decreases capacity misses
  - – Increases $\text{latency}_{\text{hit}}$

Hmm, what if we had "on-demand" associativity? How?

Reducing 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 (XYZ)+

- **Victim buffer (VB)**: small fully-associative cache
  - Sits on I$/D$ miss path
  - Small so very fast (e.g., 8 entries)
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit? Place block back in I$/D$
  - 8 extra ways, shared among all sets
    + Only a few sets will need it at any given time
  + Very effective in practice
  - Does VB reduce **$\%_{miss}$** or **$latency_{miss}$**?



Tolerating Latencies Again

- What would happen if you could have only one outstanding cache miss?
  - Inorder, single-issue
  - Out-of-order, wide-issue
- How about not even hits?

Overlapping Misses: Lockup Free Cache (aka Memory-Level Parallelism)

- **Lockup free**: allows other accesses while miss is pending
  - Consider: Load [r1] -> r2; Load [r3] -> r4; Add r2, r4 -> r5
  - **Handle misses in parallel**
    - "memory-level parallelism"
  - 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
  - Common scenario: "hit under miss"
    - Handle hits while miss is pending
    - Easy
  - Less common, but common enough: "miss under miss"
    - A little trickier, but common anyway
    - Requires multiple MSHRs: search to avoid frame conflicts

Is There A Performance Difference?

```
int x[NROWS][NCOLS];

for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j++)
      x[i][j] = 0;


for (j = 0; j<NCOLS; j++)
   for (i = 0; i<NROWS; i++)
      x[i][j] = 0;
```

- Is there a performance in these loops difference? If so, why?

Is There A Performance Difference?

```
int x[NROWS][NCOLS];

for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j++)
      x[i][j] = 0;
```

```
for (j = 0; j<NCOLS; j++)
   for (i = 0; i<NROWS; i++)
      x[i][j] = 0;
```

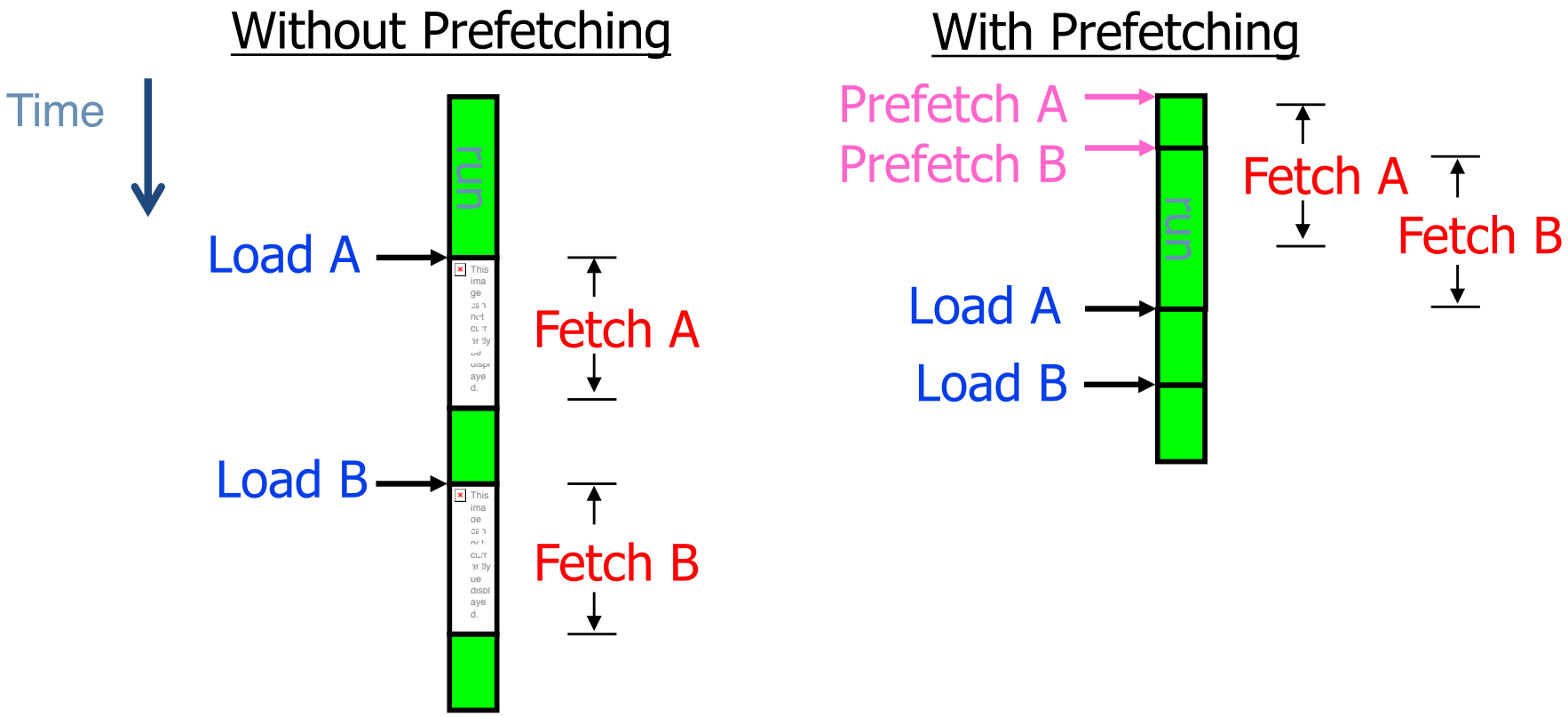
~5X slower!

Tolerating Cache Miss Latencies

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    sum += A[i][j];
```

- What can the *hardware* do to avoid miss penalties in this case?

Prefetching: Start moving data close to the processor *before* it is needed



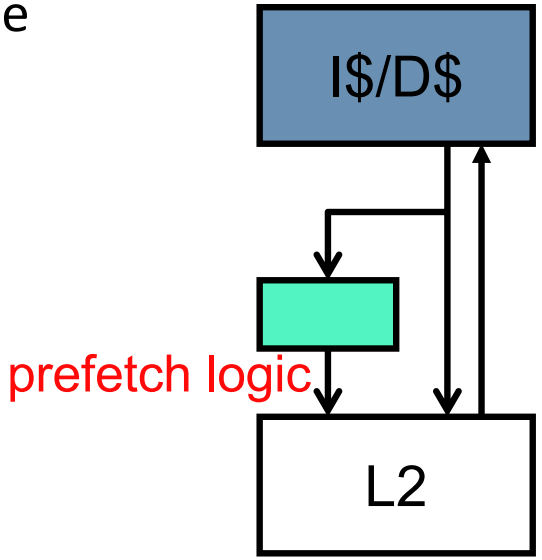
- Prefetching allows cache misses to be overlapped with:
  - computation, and
  - other cache misses.

Prefetching

- 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

- Use a special "prefetch" instruction
  - Tells the hardware to bring in data, doesn't actually read it
  - Just a hint
- Inserted by programmer or compiler
- 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];
    }
```

- Multiple prefetches bring multiple blocks in parallel
  - Using lockup-free caches
  - "Memory-level" parallelism

Prefetches vs. Memory Loads?

- What is similar?
- What is the different?

Prefetches vs. Memory Loads?

- Similarities:
  - both are given a data address as an argument
  - if that location is not in the L1 data cache, then a cache miss is triggered, and the data is moved into the cache

- Differences:
  - prefetches do not have a register destination
    - Hence they are "non-binding"
  - prefetches are non-blocking
    - i.e. the processor does not stall: it keeps executing
  - prefetches do not trigger memory exceptions
    - it is ok to prefetch invalid memory addresses

Hardware Prefetching

- What to prefetch?
  - **Stride-based sequential prefetching**
    - Can also do N blocks ahead to hide more latency
    + Simple, works for sequential things: insns, array data
    + Works better than doubling the block size
  - **Address-prediction**
    - Needed for non-sequential data: lists, trees, etc.
    - Use a hardware table to detect strides, common patterns
  - Other ideas?

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

More Advanced Address Prediction

- "Next-block" prefetching is easy, what about other options?
- **Correlating predictor**
  - 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
- Content-directed or dependence-based prefetching
  - Greedily chases pointers from fetched blocks
- Jump pointers
  - Augment data structure with prefetch pointers
- Make it easier to prefetch: cache-conscious layout/malloc
  - Lays lists out serially in memory, makes them look like array
- Active area of research

Waittaminute, What about writes?

Write Propagation

- When to propagate new value to (lower level) memory?
- **Option #1: Write-through**: immediately
  - On hit, update cache
  - Immediately send the write to the next level
- **Option #2: Write-back**: when block is replaced
  - Requires additional "dirty" bit per block
    - Replace **clean** block: **no extra traffic**
    - Replace **dirty** block: **extra "writeback" of block**

- What are the trade-offs?

Write Propagation Comparison

- **Write-through**
  - – Requires additional bus bandwidth
    - Consider repeated write hits
  - – Next level must handle small writes (1, 2, 4, 8-bytes)
  - \+ No need for valid bits in cache
  - \+ No need to handle "writeback" operations
    - Simplifies miss handling (no WBB)
  - – Sometimes used for L1 caches (for example, by IBM)

- **Write-back**
  - \+ Key advantage: uses less bandwidth
  - – Reverse of other pros/cons above
  - – Used by Intel and AMD
  - – Second-level and beyond are generally write-back caches

Write Miss Handling

- Should we bring the data to the cache on a write miss?

Write Miss Handling

- How is a write miss actually handled?

- **Write-allocate**: fill block from next level, then write it
  - \+ Decreases read misses (next read to block will hit)
  - – Requires additional bandwidth
  - – Commonly used (especially with write-back caches)

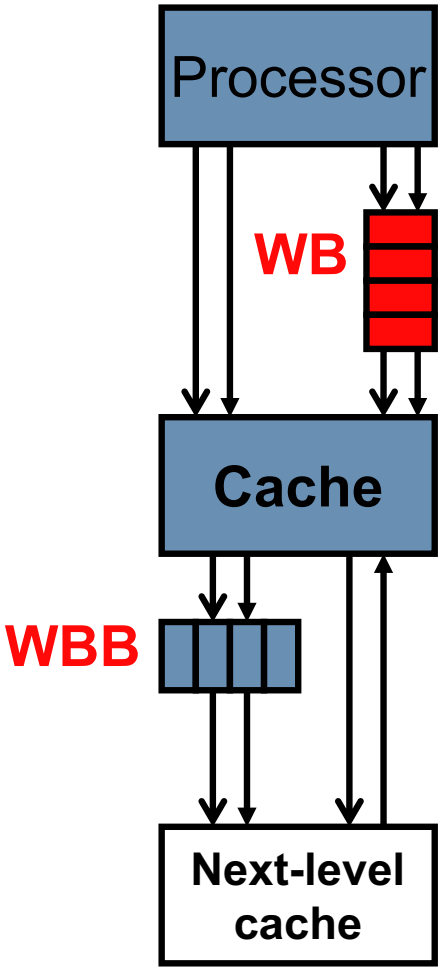
- **Write-non-allocate**: just write to next level, no allocate
  - – Potentially more read misses
  - \+ Uses less bandwidth
  - – Use with write-through

Write misses latency

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - What happens to the store instruction when it reaches the head of the ROB?
  - What can we do about it?

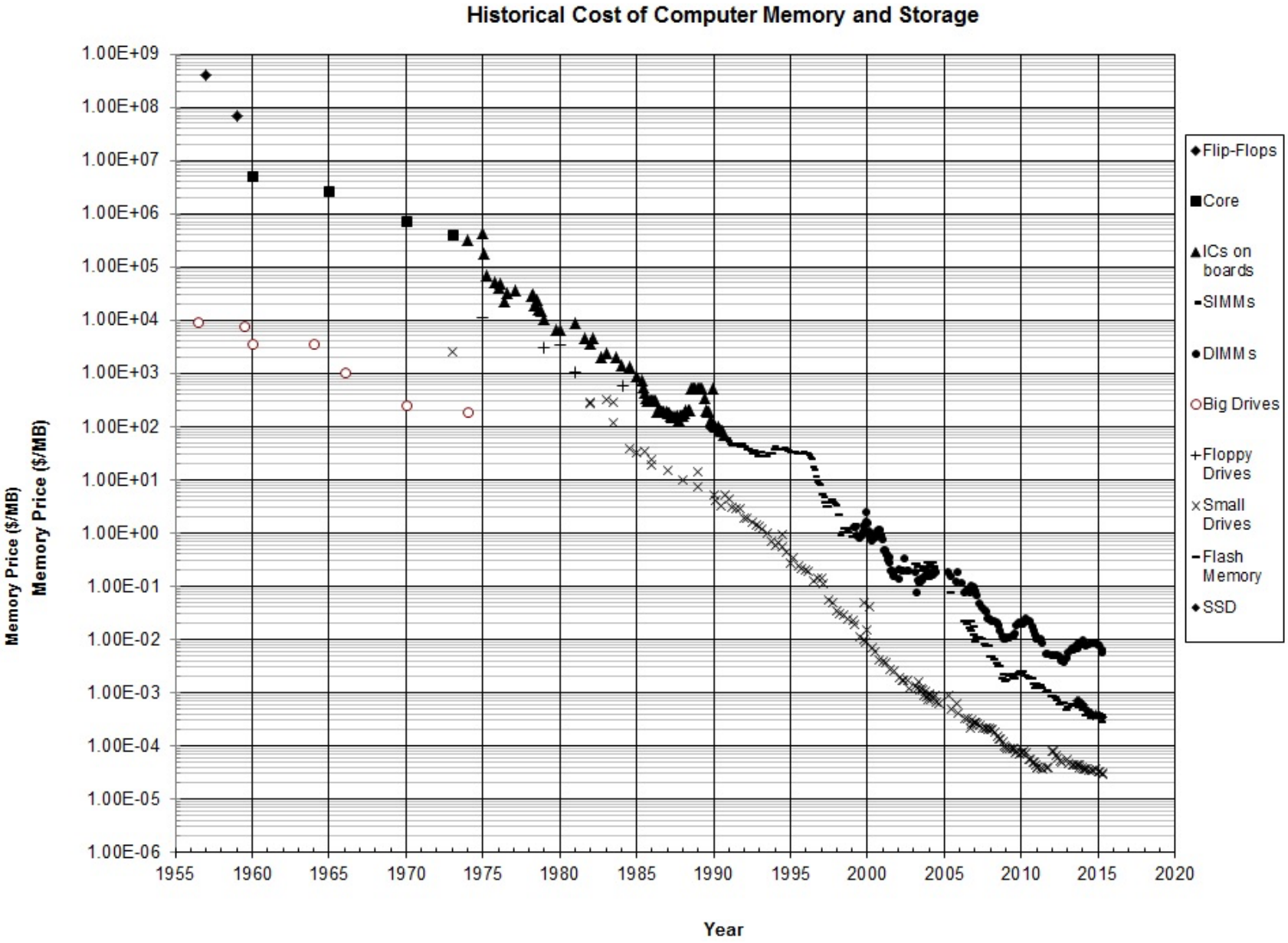
Write Misses and Write Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?
- **Write buffer**: a small buffer
  - How does it help?
- Write buffer vs. writeback-buffer
  - Write buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks
- Forwarding again?? ☺



Local vs Global Miss Rates

- Local hit/miss rate:
  - Percent of references to cache hit (e.g, 90%)
  - Local miss rate is (100% - local hit rate), (e.g., 10%)
- Global hit/miss rate:
  - Misses per instruction (1 miss per 30 instructions)
  - Instructions per miss (3% of instructions miss)
  - Above assumes loads/stores are 1 in 3 instructions
- Consider second-level cache hit rate
  - L1: 2 misses per 100 instructions
  - L2: 1 miss per 100 instructions
  - L2 "local miss rate" -> 50%


Historical Cost of Computer Memory and Storage
Memory Price ($/MB)
Memory Price ($/MB)
1.00E+09
1.00E+08
1.00E+07
1.00E+06
1.00E+05
1.00E+04
1.00E+03
1.00E+02
1.00E+01
1.00E+00
1.00E-01
1.00E-02
1.00E-03
1.00E-04
1.00E-05
1.00E-06
1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
Year
Flip-Flops
Core
ICs on boards
SIMMs
DIMMs
Big Drives
Floppy Drives
Small Drives
Flash Memory
SSD