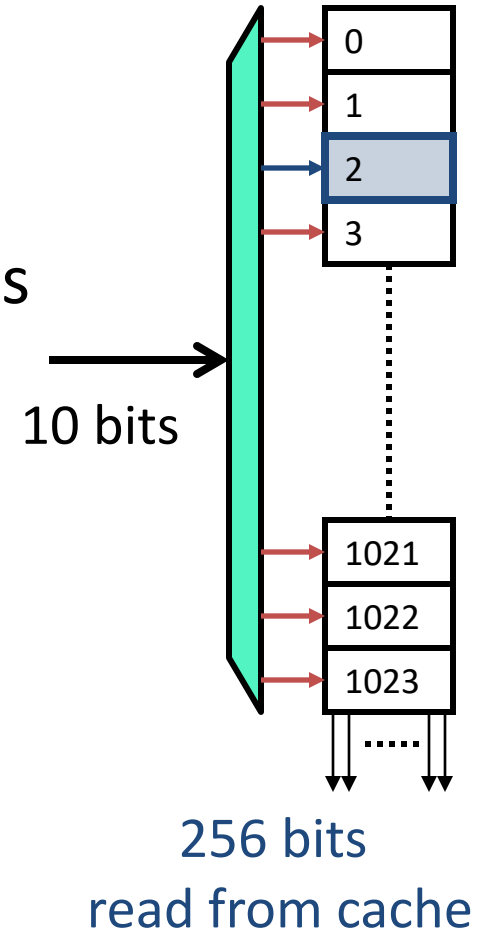
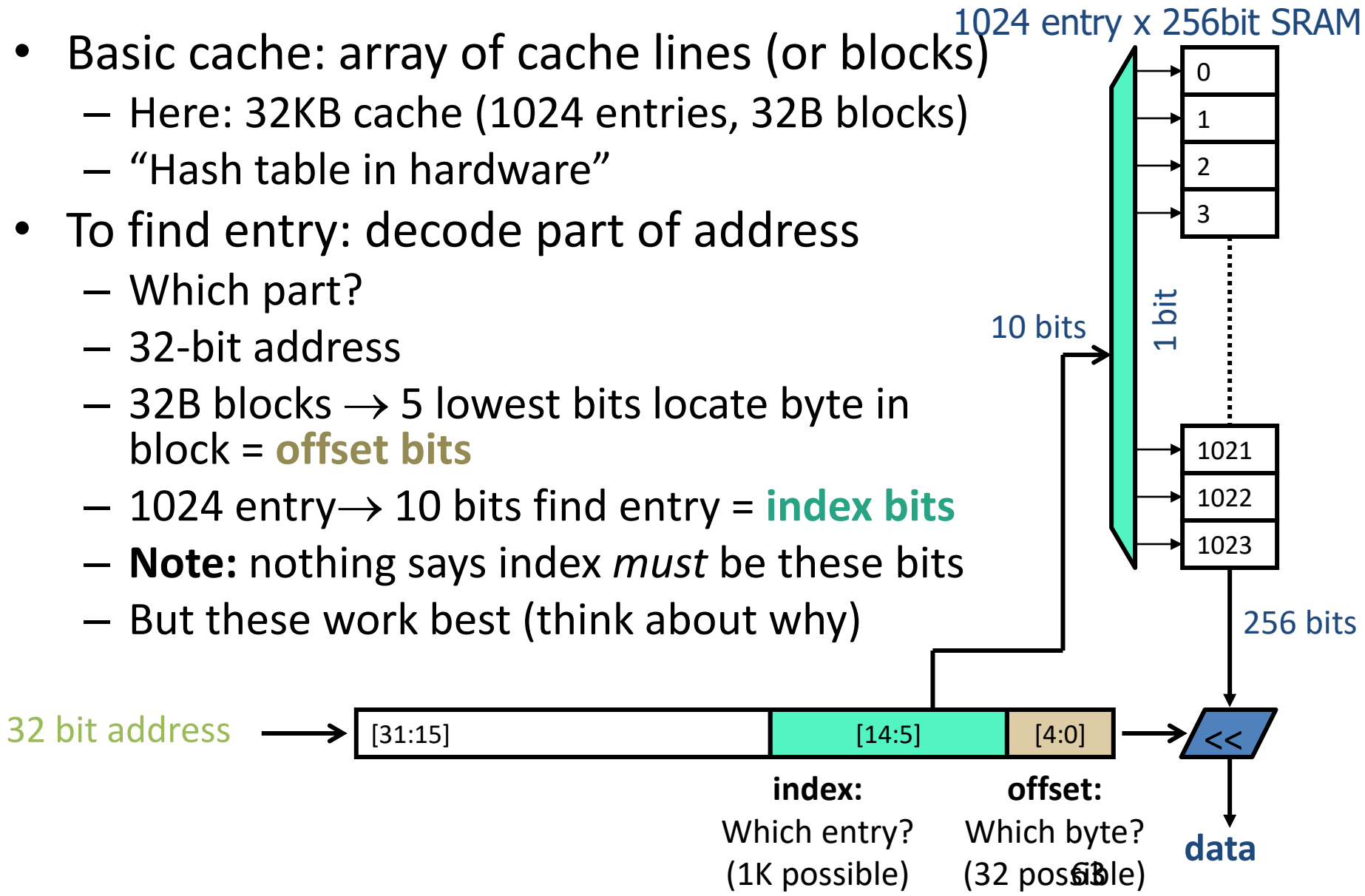
Basic Memory Array Structure

- Number of entries
  - n bits for lookup → 2^n entries
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
- Size of entries
  - Width of data accessed
  - Here: 256 bits (32 bytes)



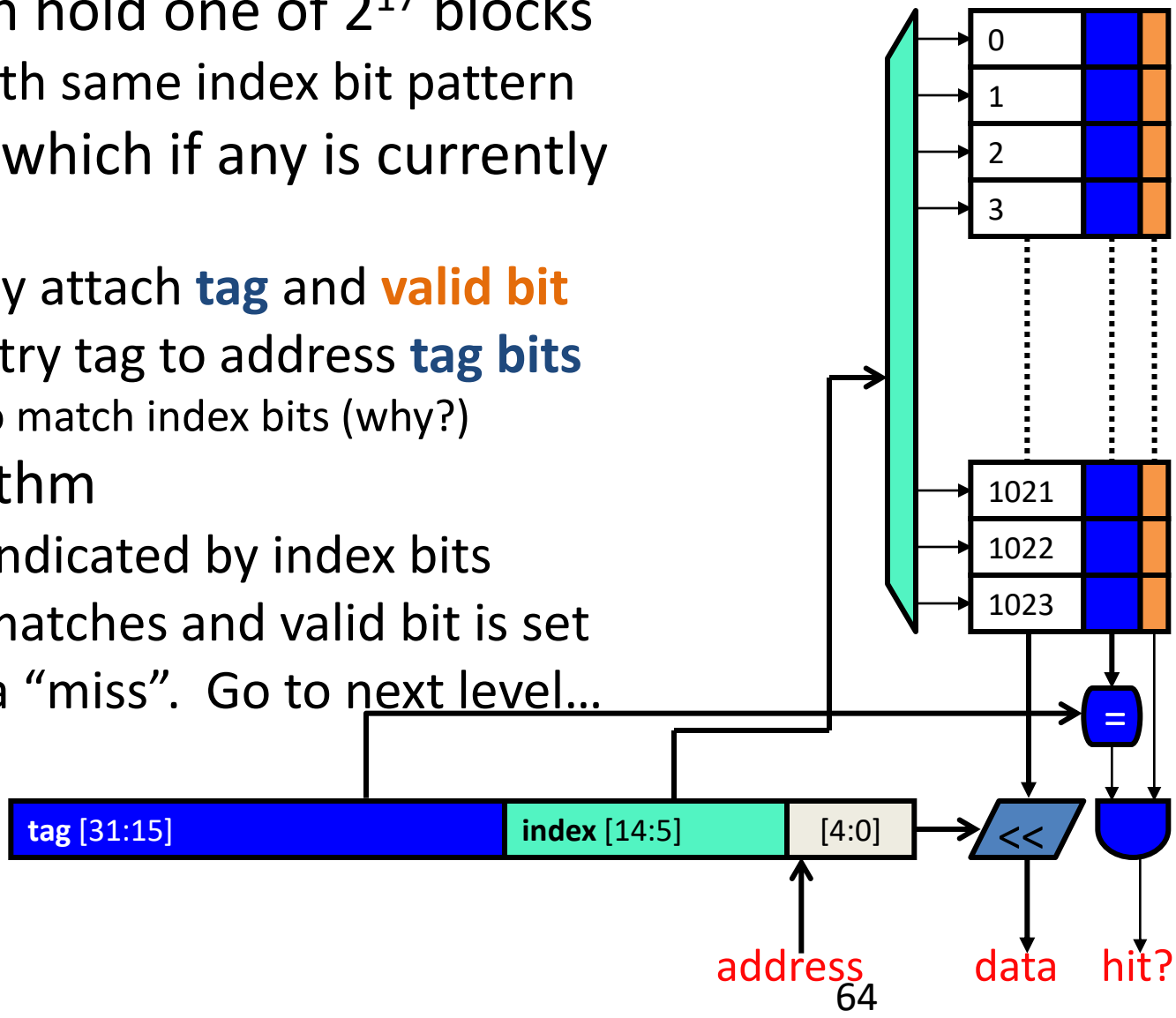
Caches: Finding Data via Indexing

- Basic cache: array of cache lines (or blocks)
  - Here: 32KB cache (1024 entries, 32B blocks)
  - "Hash table in hardware"
- To find entry: decode part of address
  - Which part?
  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block = **offset bits**
  - 1024 entry→ 10 bits find entry = **index bits**
  - **Note:** nothing says index *must* be these bits
  - But these work best (think about why)



Knowing that You Found It: Tags

- Each entry 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 entry attach **tag** and **valid bit**
  - Compare entry tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read entry indicated by index bits
  - "Hit" if tag matches and valid bit is set
  - Otherwise, a "miss". Go to next level…



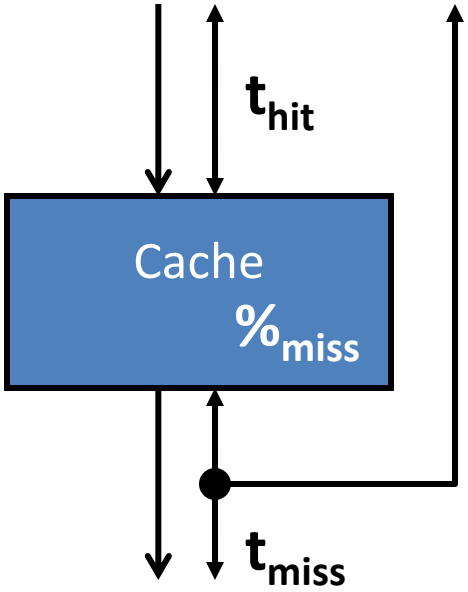
Handling a Cache Miss

- What if requested data isn't in the cache?
  - How does it get in there?

- **Cache controller**: finite state machine
  - Remembers miss address
  - Accesses next level of memory
  - Waits for response
  - Writes data/tag into proper locations

  - All of this happens on the **fill path**
  - Sometimes called **backside**

Cache Performance Equation



- **Access**: read or write to cache
- **Hit**: desired data found in cache
- **Miss**: desired data not found in cache
  - Must get from another component
  - No notion of "miss" in register file
- **Fill**: action of placing data into cache

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

- Performance metric: average access time

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

CPI Calculation with Cache Misses

- **Parameters**
  - Simple pipeline with base CPI of 1
  - Instruction mix: 30% loads/stores
  - I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

- **What is new CPI?**
  - $CPI_{I\$}$ =
  - $CPI_{D\$}$ =
  - CPI_{new} =

CPI Calculation with Cache Misses

- **Parameters**
  - Simple pipeline with base CPI of 1
  - Instruction mix: 30% loads/stores
  - I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

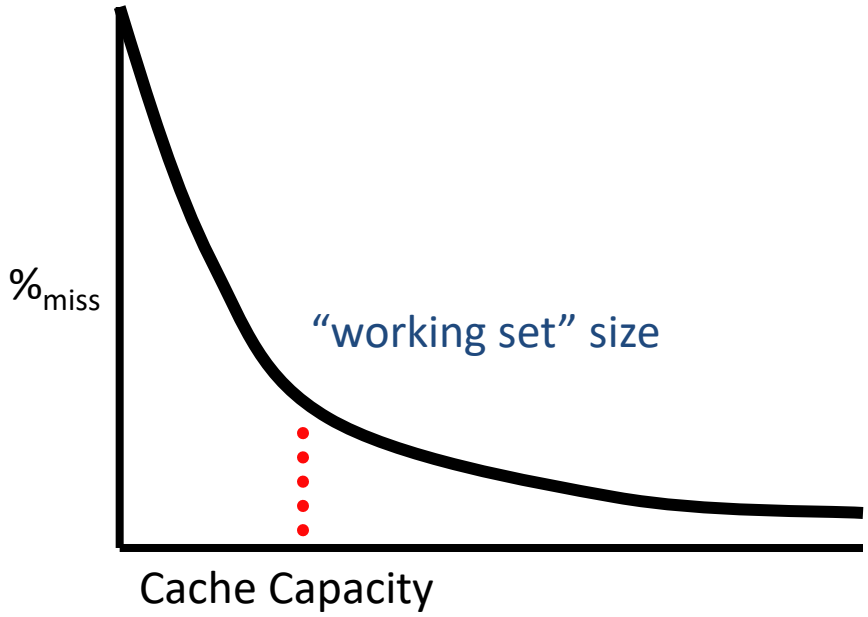
- **What is new CPI?**
  - $CPI_{I\$} = \%_{missI\$} \times t_{miss} = 0.02 \times 10 \text{ cycles} = 0.2 \text{ cycle}$
  - $CPI_{D\$} = \%_{load/store} \times \%_{missD\$} \times t_{missD\$} = 0.3 \times 0.1 \times 10 \text{ cycles} = 0.3 \text{ cycle}$
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$} = 1+0.2+0.3 = 1.5$

Measuring Cache Performance

- Ultimate metric is t_{avg}
  - Cache capacity and circuits roughly determines t_{hit}
  - Lower-level memory structures determine t_{miss}
  - Measure $\%_{miss}$
    - Hardware performance counters
    - Simulation

Capacity and Performance

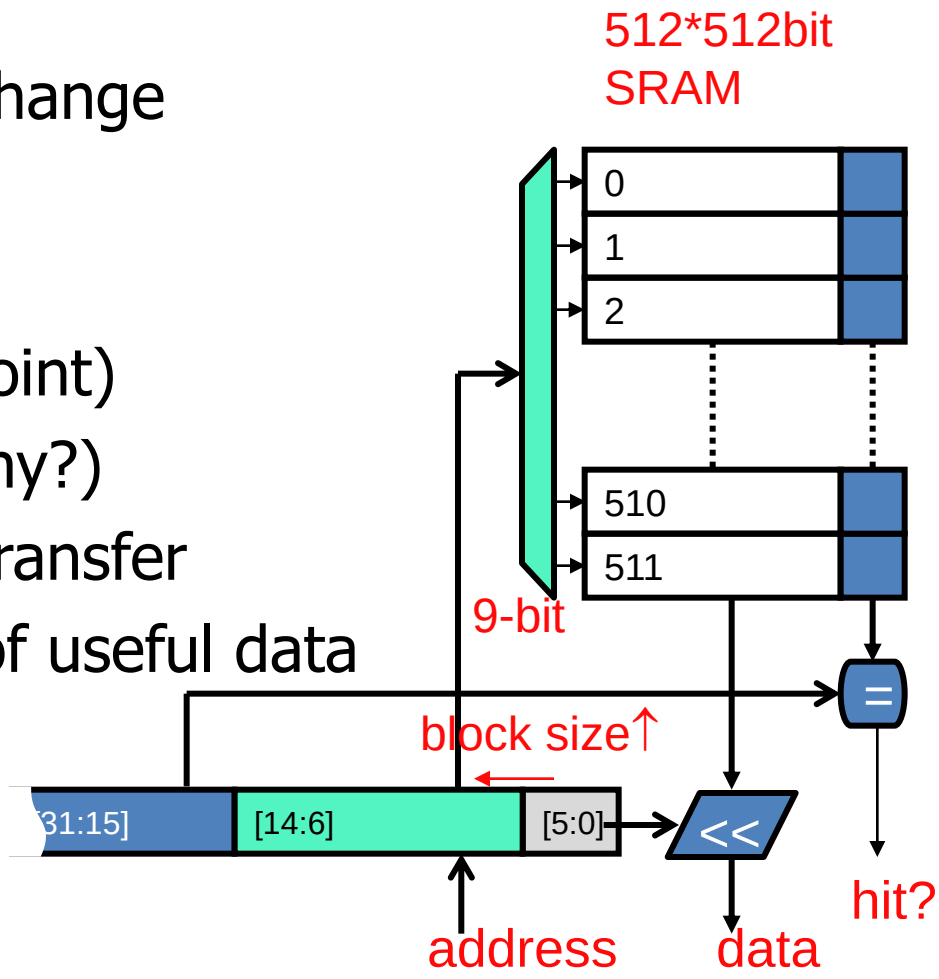
- Simplest way to reduce $\%_{miss}$: increase capacity
  - + Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
    - Diminishing returns
  - – However t_{hit} increases
    - Latency proportional to sqrt(capacity)
  - – t_{avg} ?



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

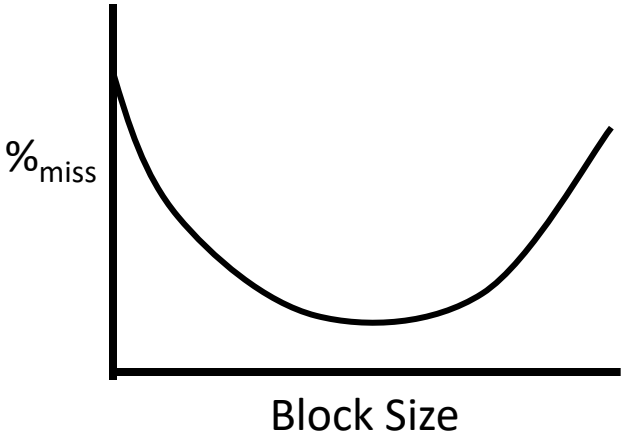
Block Size

- Given capacity, manipulate $\%_{miss}$ by changing organization
- One option: increase **block size**
  - Exploit **spatial locality**
  - Notice index/offset bits change
  - Tag remain the same
- Ramifications
  - \+ Reduce $\%_{miss}$ (up to a point)
  - \+ Reduce tag overhead (why?)
  - – Potentially useless data transfer
  - – 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 entries)
    - Turns hits into misses by disallowing simultaneous residence
    - Consider entire cache as one big block
- 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 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 (more later)

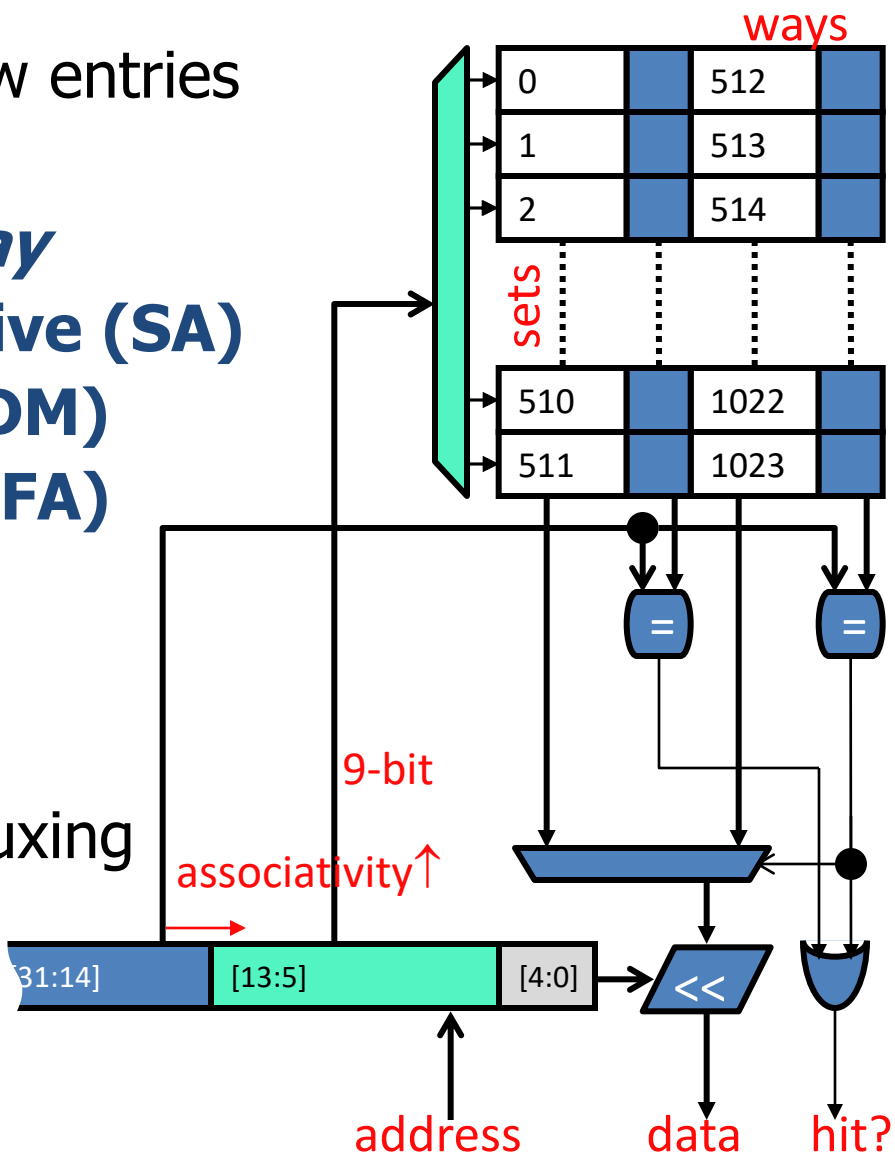
Set-Associativity

- Block can reside in one of few entries
- Entry groups called ***sets***
- Each entry 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 shown

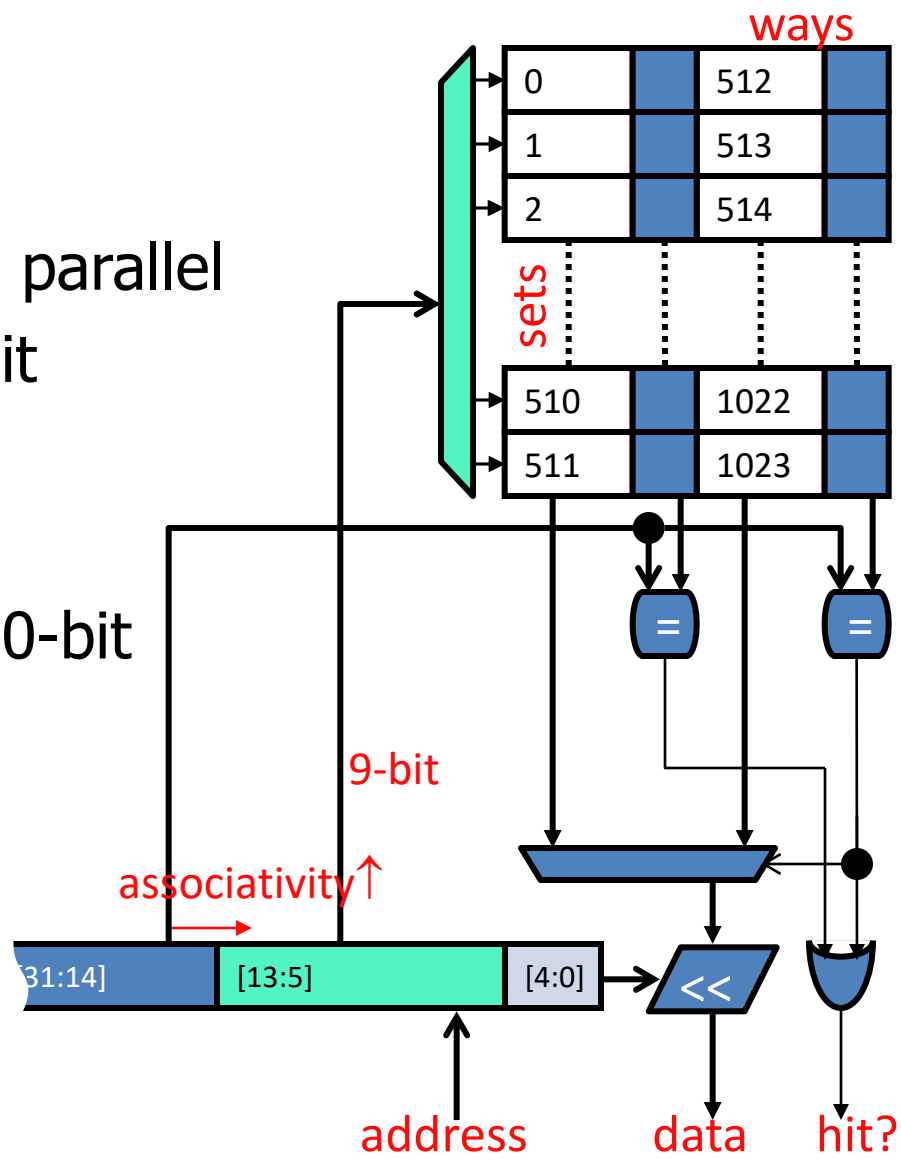


Set-Associativity

Lookup algorithm

- Use index bits to find set
- Read data/tags in all ways 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

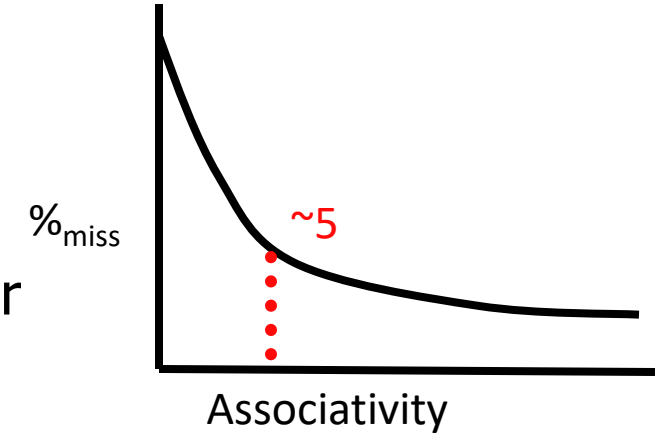


Replacement Policies

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

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
  - Makes indexing easier (just rip bits out of the address)
- 3-way set-associativity? No problem

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**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of entries in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**
  - **(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

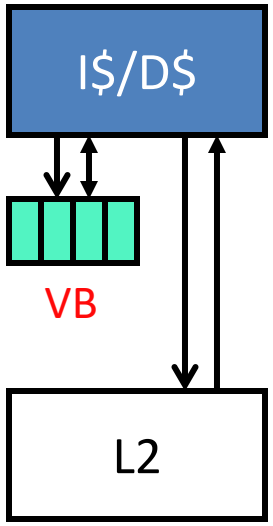
Miss Rate: ABC

- 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 entries)
  - \+ Decreases compulsory/capacity misses (spatial locality)
  - No significant effect on $\text{latency}_{\text{hit}}$
- **Capacity**
  - \+ Decreases capacity misses
  - – Increases $\text{latency}_{\text{hit}}$

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 **$latency_{miss}$**?



Overlapping Misses: Lockup Free Cache

- **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 entry, 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

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];    // say
```

- Better code

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

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
```

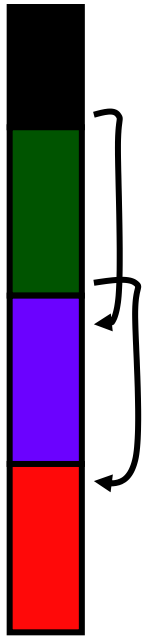
  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

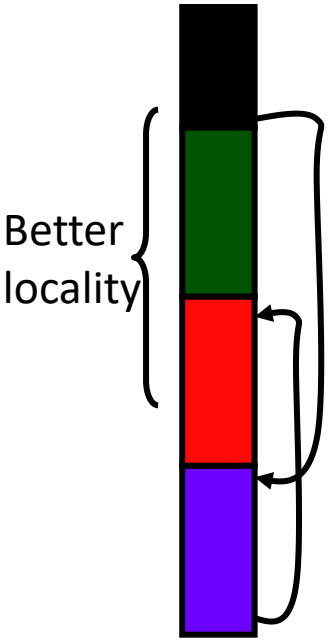
```
for (i=0; i<NELEMS; i+=CACHE_SIZE)
    for (k=0; k<NITERATIONS; k++)
        for (ii=0; ii<i+CACHE_SIZE-1; ii++)
            sum += X[ii];
```

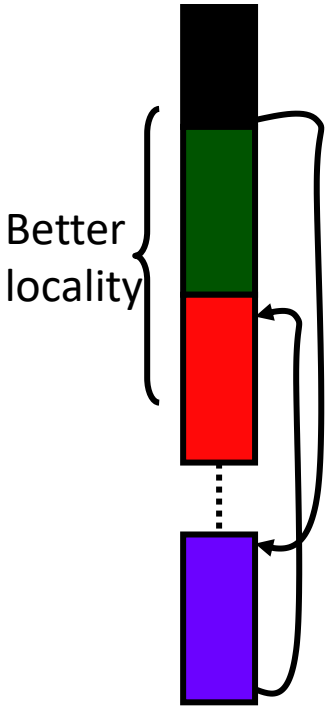
  - Assumes you know **CACHE_SIZE**, do you?
  - Loop fusion: similar, but for multiple consecutive loops

Software Restructuring: Code

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







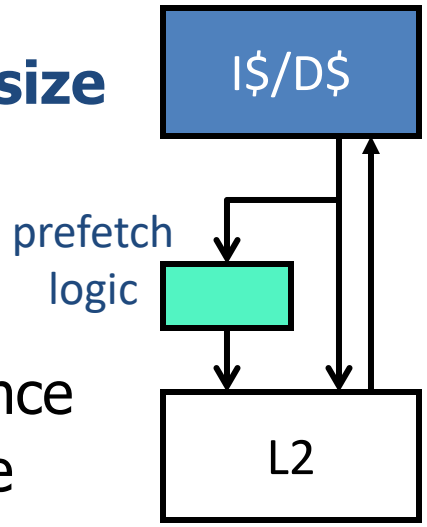
- Fewer taken branches, too
- Intra-procedure, inter-procedure

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 addr **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) {
        __builtin_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

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
- 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 $\rightarrow$ 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, so they look like arrays
- Active area of research

Write Issues

- So far we have looked at reading from cache
  - Instruction fetches, loads
- What about writing into cache
  - Stores, not an issue for instruction caches (why they are simpler)

- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
  - Hiding write miss latency

Tag/Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK, stall until good data arrives)

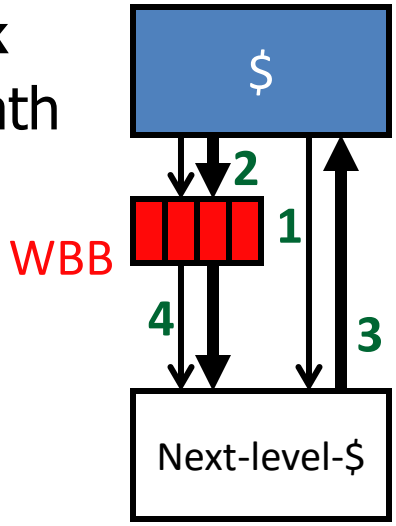
- Writes: read tag, write data in parallel?
  - Tag mis-match → clobbered data (oops)
  - For associative caches, which way was written into?

- Writes are a pipelined two step (multi-cycle) process
  - Step 1: match tag
  - Step 2: write to matching way
  - Bypass (with address check) to avoid load stalls
  - May introduce structural hazards

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**
  - \+ **Writeback-buffer (WBB)**: keep it off critical path
    1. Send "fill" request to next-level
    2. While waiting, write dirty block to buffer
    3. When new blocks arrives, put it into cache
    4. Write buffer contents to next-level



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 dirty bits in cache
  - + No need to handle "writeback" operations
    - Simplifies miss handling (no write-back buffer)
  - Sometimes used for L1 caches (for example, by IBM)

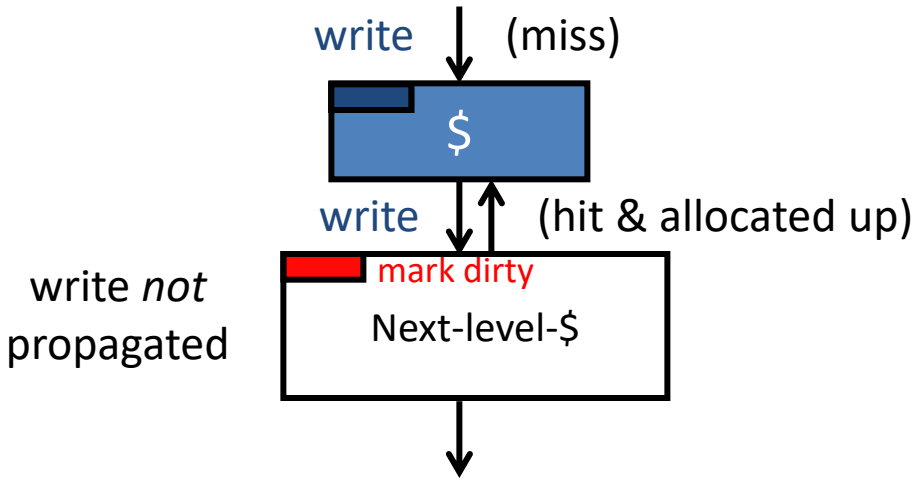
- **Write-back**
  - + Key advantage: uses less bandwidth
  - Reverse of other pros/cons above
  - Used by Intel and AMD
  - 2nd-level and beyond are generally write-back caches

Write Miss Handling

How is a write miss handled?

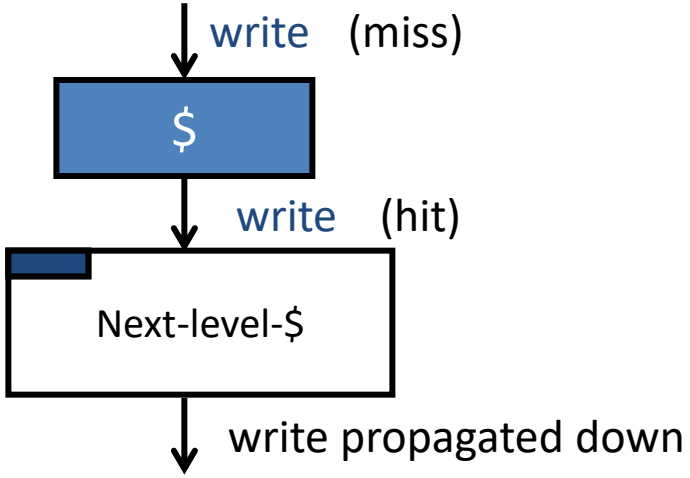
Write-allocate: fill block from next level, then write it

+ Decreases read misses (next read to block will hit)
– Requires additional bandwidth
• Commonly used (especially with **write-back** caches)

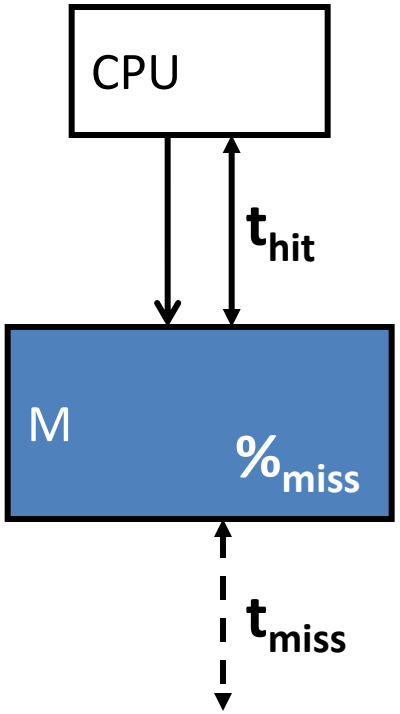


Write-non-allocate: just write to next level, no allocate

– Potentially more read misses
+ Uses less bandwidth
• Use with **write-through**



Memory Performance Equation



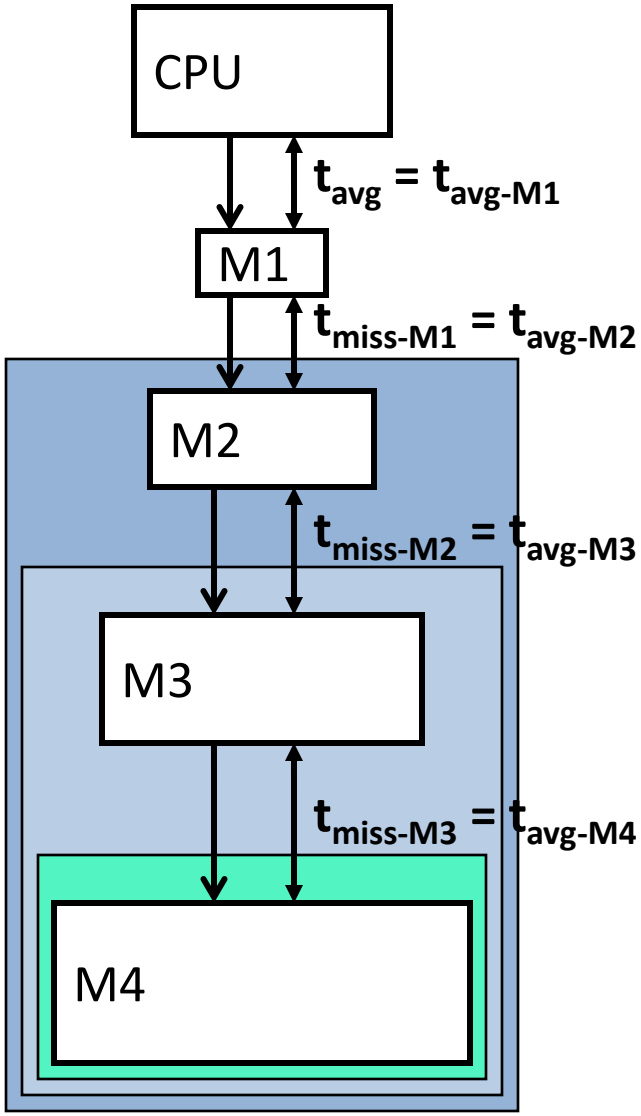
- **Access**: read or write to M
- **Hit**: desired data found in M
- **Miss**: desired data not found in M
  - Must get from another (slower) component
- **Fill**: action of placing data in M

- $\%_{miss}$ (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

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

Hierarchy Performance



t_{avg} =

t_{avg-M1}

$t_{hit-M1} + (\%_{miss-M1} \times t_{miss-M1})$

$t_{hit-M1} + (\%_{miss-M1} \times t_{avg-M2})$

$t_{hit-M1} + (\%_{miss-M1} \times (t_{hit-M2} + (\%_{miss-M2} \times t_{miss-M2})))$

$t_{hit-M1} + (\%_{miss-M1} \times (t_{hit-M2} + (\%_{miss-M2} \times t_{avg-M3})))$

...

Performance Calculation with $ Hierarchy

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$}$ =
  - $t_{avgD\$}$ =
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$}$ =
  - t_{avgL2} =
  - $t_{avgD\$}$ =

Performance Calculation with $ Hierarchy

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} \times t_{hitM}$ = 1ns+(0.05x50ns) = 3.5ns
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$} = t_{avgL2}$
  - $t_{avgL2} = t_{hitL2} + \%_{missL2} \times t_{hitM}$ = 10ns+(0.2x50ns) = 20ns
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} \times t_{avgL2}$ = 1ns+(0.05x20ns) = 2ns

Designing a Cache Hierarchy

- For any memory component: t_{hit} vs. $\%_{miss}$ tradeoff

- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access $\rightarrow$ t_{hit} important
  - t_{miss} is not bad $\rightarrow$ $\%_{miss}$ less important
  - Low capacity/associativity (to reduce t_{hit})
  - Small-medium block-size (to reduce conflicts)

- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access $\rightarrow$ t_{hit} less important
  - t_{miss} is bad $\rightarrow$ $\%_{miss}$ important
  - High capacity/associativity/block size (to reduce $\%_{miss}$)

Memory Hierarchy Parameters

Parameter	I$/D$	L2	L3	Main Memory
t_{hit}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-8GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	1–4	4–16	4-16	NA

- Some other design parameters
  - Split vs. unified insns/data
  - Inclusion vs. exclusion vs. nothing
  - On-chip, off-chip, or partially on-chip?

Split vs. Unified Caches

Split I$/D$: insns and data in different caches

- To minimize structural hazards and t_{hit}
- Larger unified I$/D$ would be slow, 2nd port even slower
- Optimize I$ for wide output (superscalar), no writes

Unified L2, L3: insns and data together

- To minimize $\%_{miss}$

\+ Fewer capacity misses: unused insn capacity used for data

– More conflict misses: insn/data conflicts

  - A much smaller effect in large caches

- Insn/data structural hazards are rare: simultaneous I$/D$ miss
- Go even further: unify L2, L3 of multiple cores in a multi-core

Hierarchy: Inclusion versus Exclusion

- **Inclusion**
  - A block in the L1 is always in the L2
  - Good for write-through L1s (why?)

- **Exclusion**
  - Block is either in L1 or L2 (never both)
  - Good if L2 is small relative to L1
    - Example: AMD's Duron 64KB L1s, 64KB L2

- **Non-inclusion**
  - No guarantees

Summary

- **Average access time** of a memory component
  - $latency_{avg} = latency_{hit} + \%_{miss} \times latency_{miss}$
  - low $latency_{hit}$ and $\%_{miss}$ in one structure = hard → hierarchy
- **Memory hierarchy**
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- Cache ABCs (**a**ssociativity, **b**lock size, **c**apacity)
  - 3C miss model: compulsory, capacity, conflict
- **Performance optimizations**
  - $\%_{miss}$: prefetching
  - $latency_{miss}$: victim buffer, critical-word-first, lockup-free design
- **Write issues**
  - Write-back vs. write-through
  - write-allocate vs. write-no-allocate