CSE 530
Fundamentals of Computer Architecture
Spring 2021

Caches

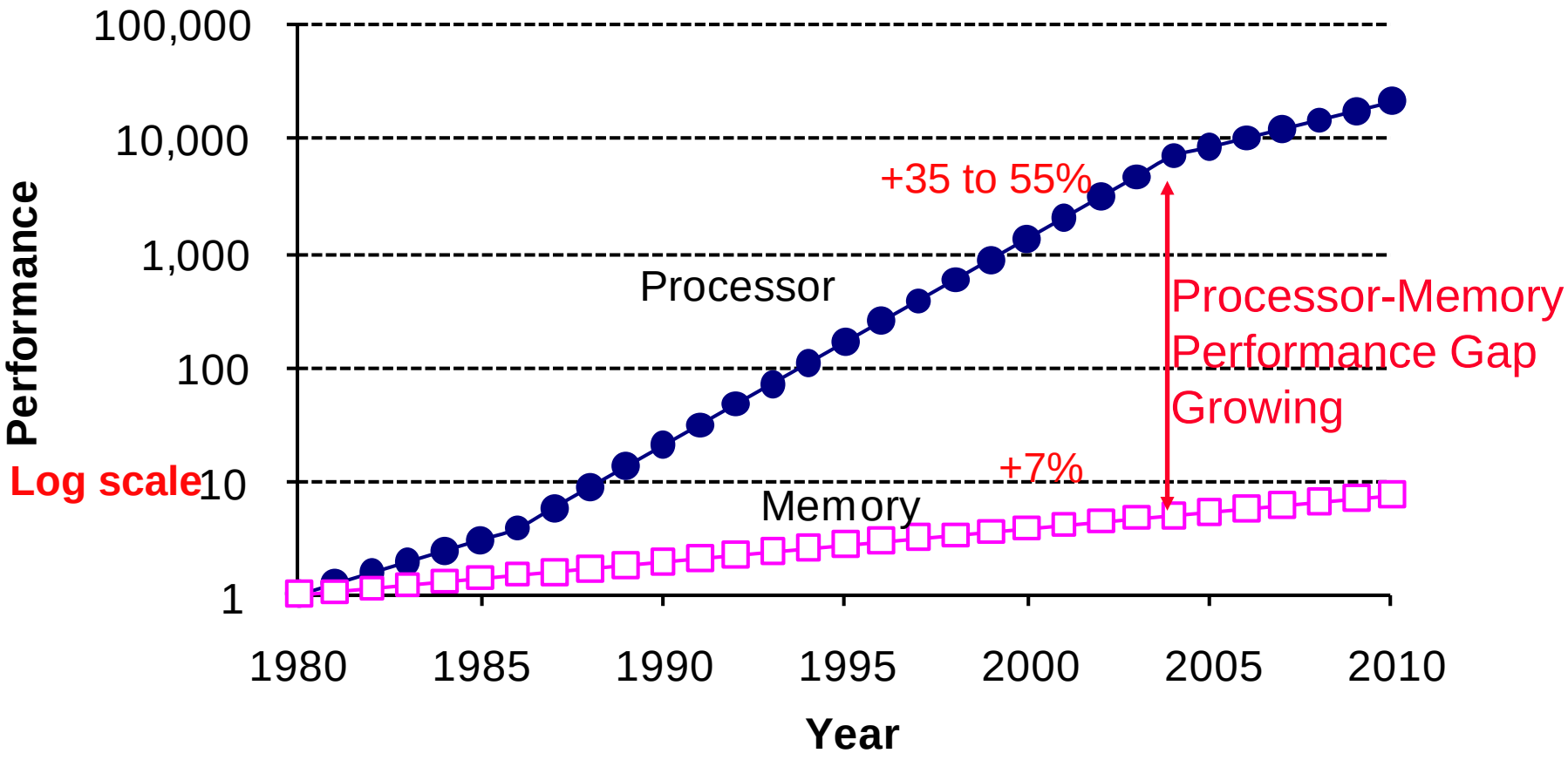
John (Jack) Sampson

Course material on CANVAS: canvas.psu.edu

[Adapted in part from Mary Jane Irwin, V. Narayanan, A. Kolli @PSU, and includes materials originally developed by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin. Material flow organized in part around *Computer Architecture: A Quantitative Approach* by Hennessey and Patterson]

The memory "wall"

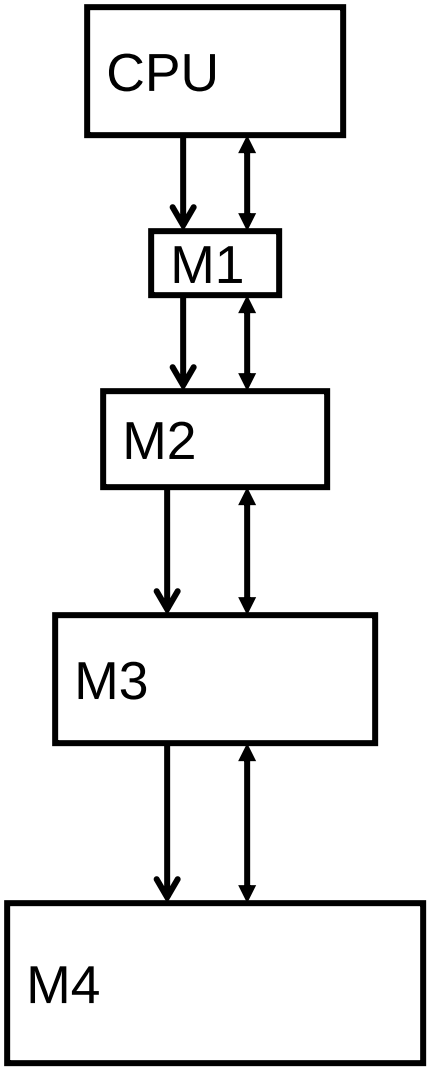
- Processors are getting faster at a rate faster than memories are getting faster



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
- Holds for both data and instructions

Exploiting locality: The memory hierarchy



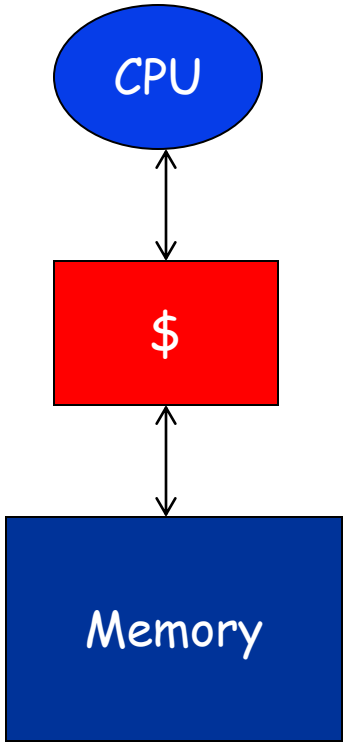
- Hierarchy of memory components
  - Upper components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by "controllers" and "buses"
  - Which also have latency and bandwidth issues
- Keep most frequently accessed data in M1
  - Next most frequently accessed in M2, etc.
  - Move data up-down hierarchy
- Optimize **average** memory access time

  $latency_{avg} = latency_{hit} + \%_{miss} * latency_{miss}$
  - Will attack **each** component of AMAT

Caches

cache is hardware managed

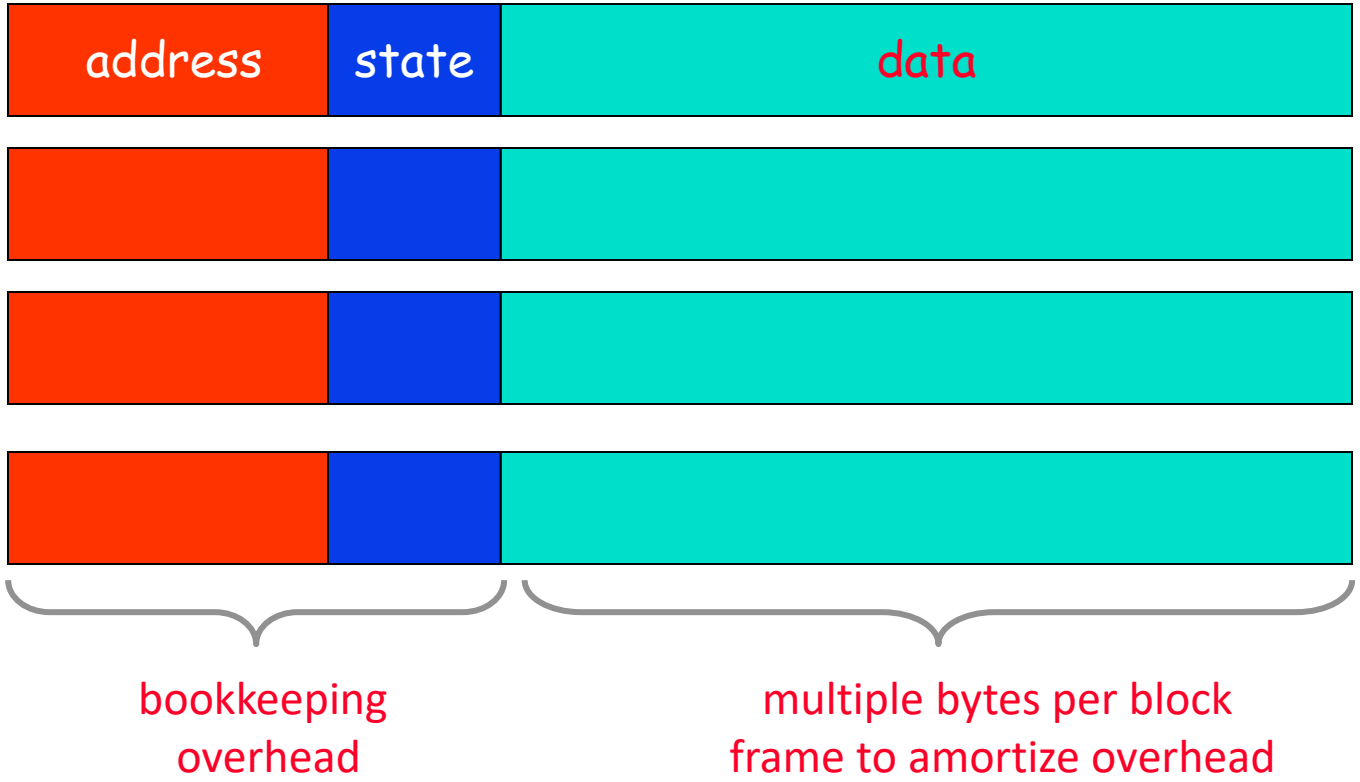
- An automatically managed hierarchy
- *"A hiding place, esp. of goods, treasure, etc." -- OED*
- Keep recently accessed block
  - temporal locality
- Break memory into blocks (several bytes) and transfer data to/from cache in blocks
  - spatial locality
- *Many architectures opt for software managed scratch-pad memory instead e.g. Cray-1, embedded processors, GPUs (partially) Why??*



because the locality is so predictable, that it makes sense to use a software managed cache

Cache (Abstractly)

- Keep recently accessed block in “block frame”
  - state (e.g., valid)
  - address tag
  - data



Cache (Abstractly)

- On memory read

  if incoming address corresponds to one of the stored address tags then

  - HIT
  - return data

  else

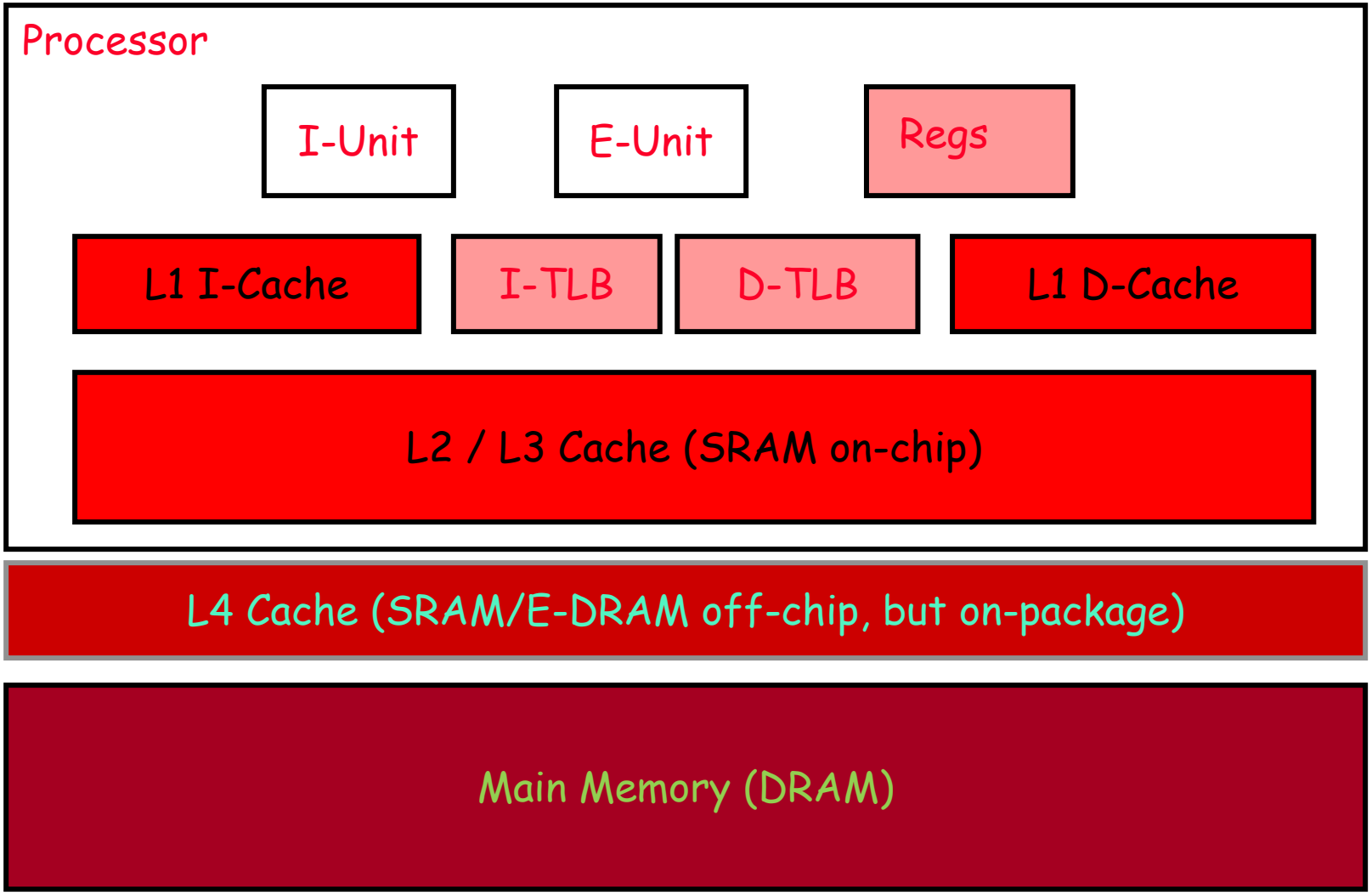
  - MISS
  - choose & displace a current block (potentially one in use
  - fetch new (referenced) block from memory into frame
  - return data

- Where and how to look for a block? (Block placement)
- Which block is replaced on a miss? (Block replacement)
- What happens on a write? Write strategy (Later)
- What is kept? (Bookkeeping, data)

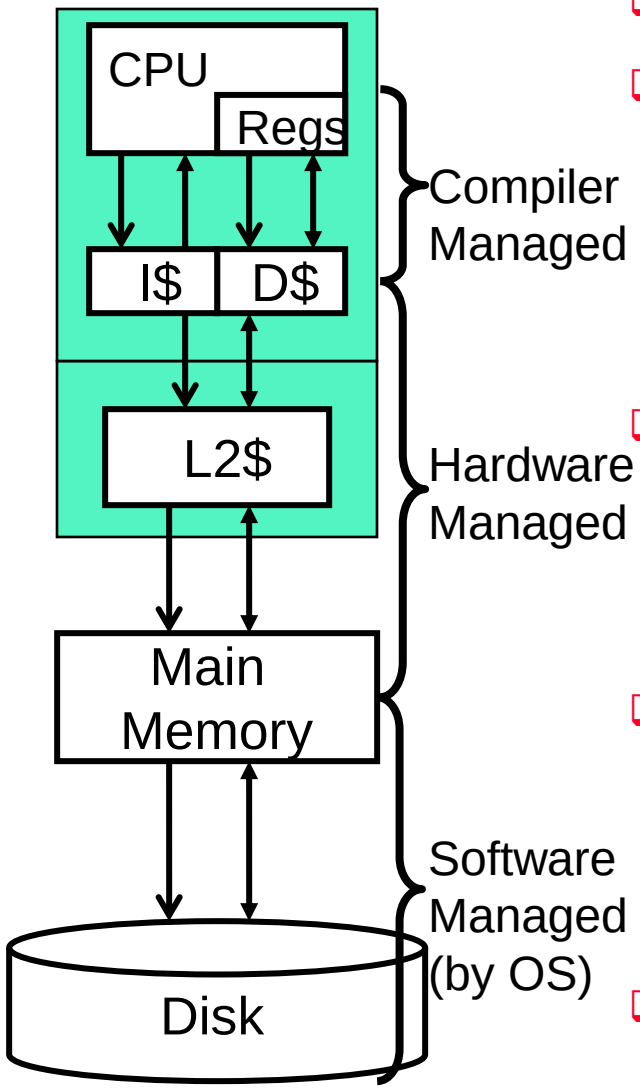
Terminology

- frame – space for a minimum unit of allocation in a cache
- block (cache line) — minimum data unit that may be present (in a frame)
- hit — block is found in the cache
- miss — block is not found in the cache
- miss ratio — fraction of references that miss
- hit time — time to access the cache
- miss penalty
  - time to replace block in the cache + deliver to upper level
  - access time — time to get (at least the) first word
  - transfer time — time for remaining words

Processor/Memory Boundaries



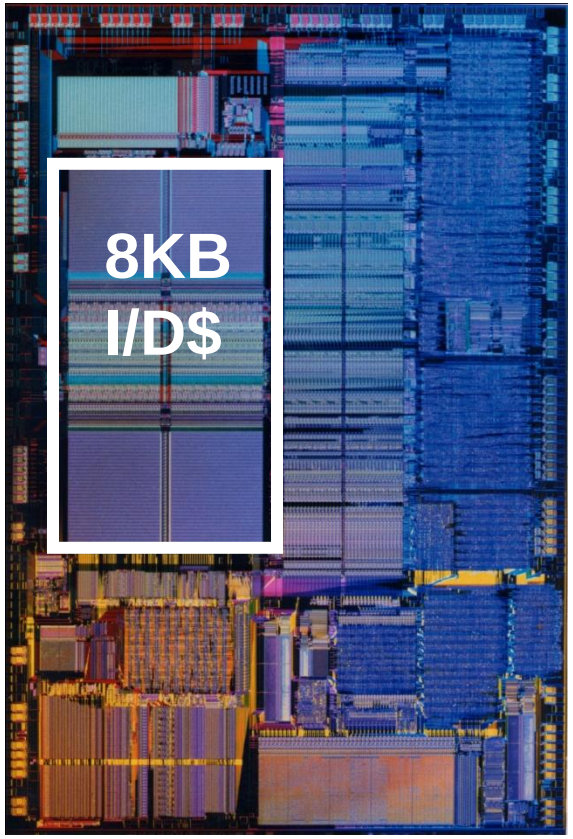
A concrete memory hierarchy example



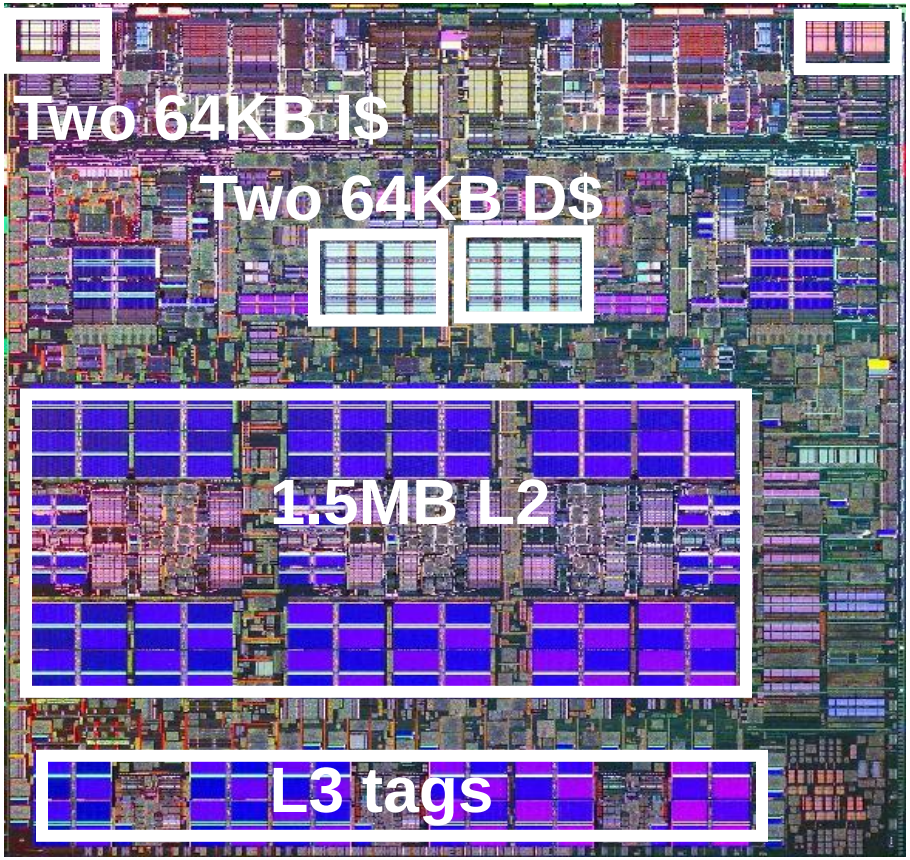
- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$); typically 8KB to 64KB each
  - On-chip (with CPU), made of SRAM (same circuit type as CPU)
- 2nd level: **Second-level cache** (L2$)
  - On-chip (with CPU), SRAM
  - Unified (holds both I & D); typically 512KB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**

Evolution of cache hierarchies

- Processors today are 30 to 70% cache by area



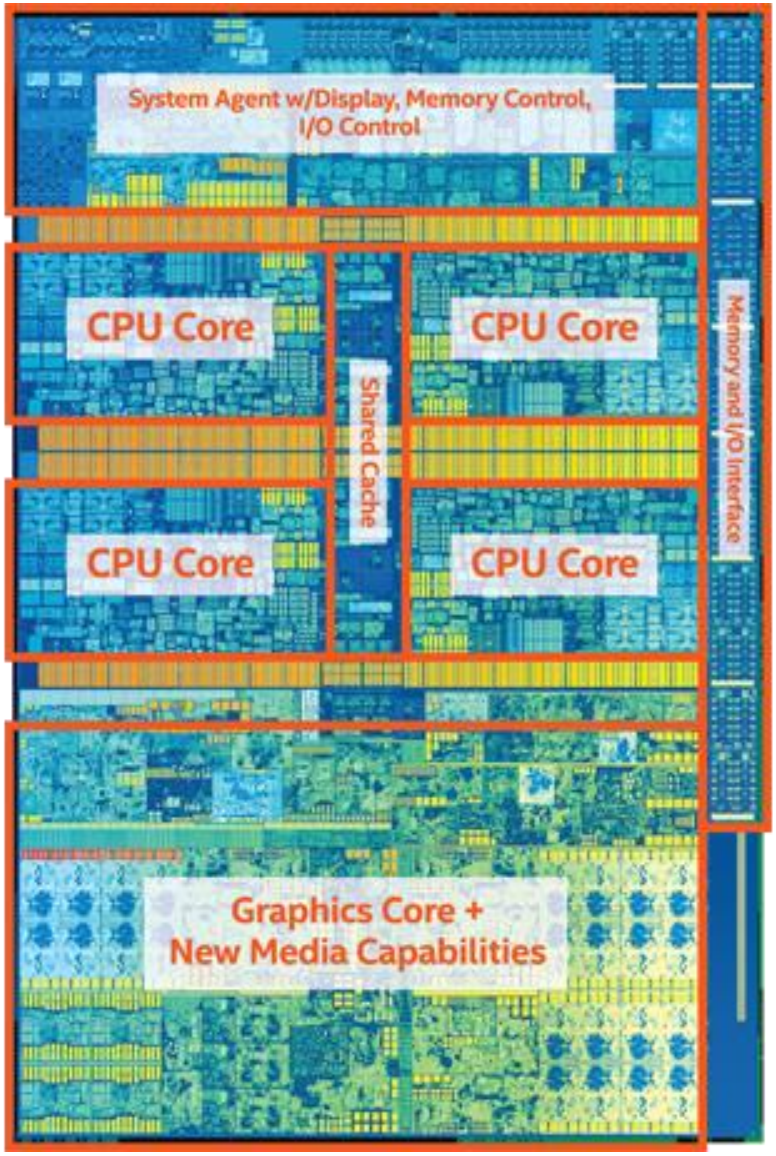
Intel 486



IBM Power5 (dual core)

Evolution of Memory Hierarchies

- Modern chips, however
  - Are SoCs – many components
  - Now have system agent logic on-chip
  - Now often have graphics/media on-chip
- But…
  - Many of these components also contain substantial memories!
  - ... although not necessarily in the primary hierarchy



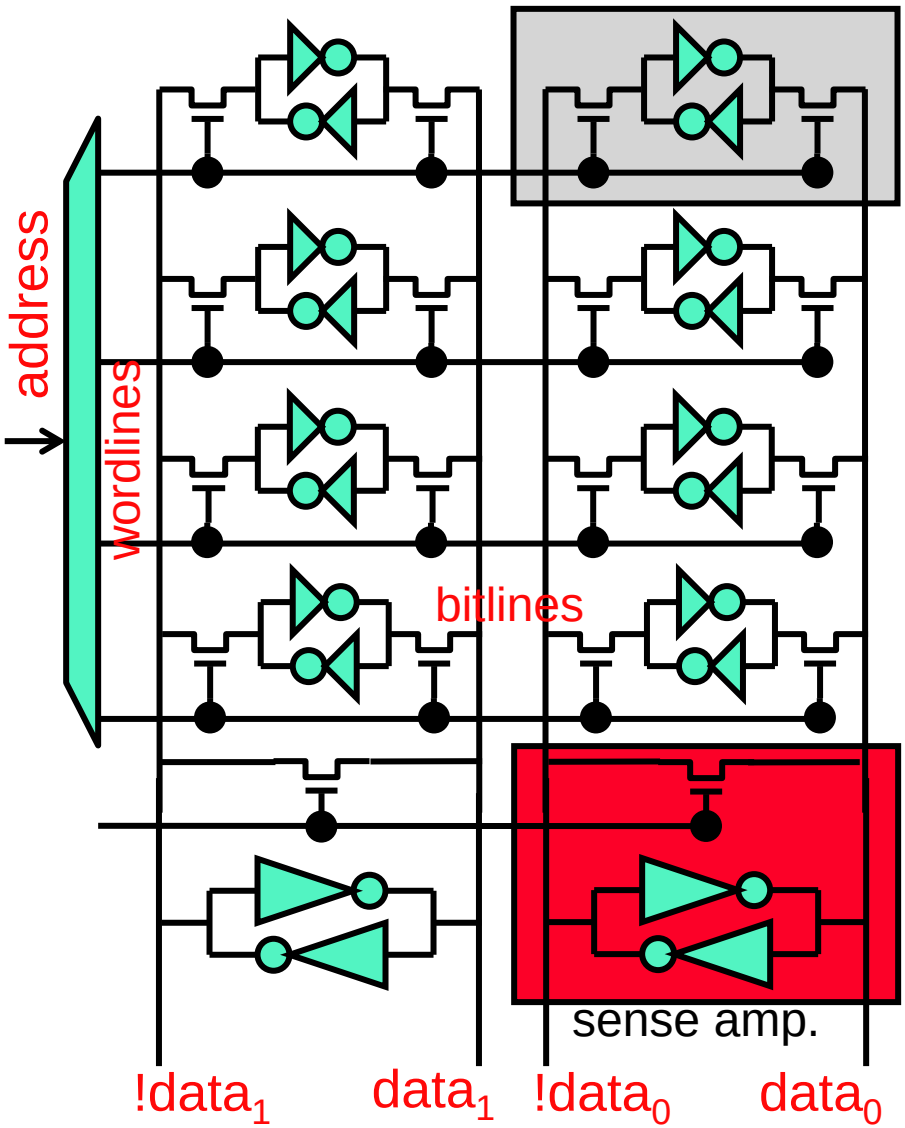
Aside: Types of memory

- **Static RAM (SRAM)**
  - 6 transistors per bit (now 8 – why?) to make it more robust
  - Optimized for speed (first) and density (second)
  - Fast (sub-nanosecond latencies for small SRAM)
    - Speed proportional to its area
  - Integrates well with standard processor logic

- **Dynamic RAM (DRAM)**
  - 1 transistor + 1 capacitor per bit cheaper, so only one
  - 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, STT, Re-RAM, PCM, etc.

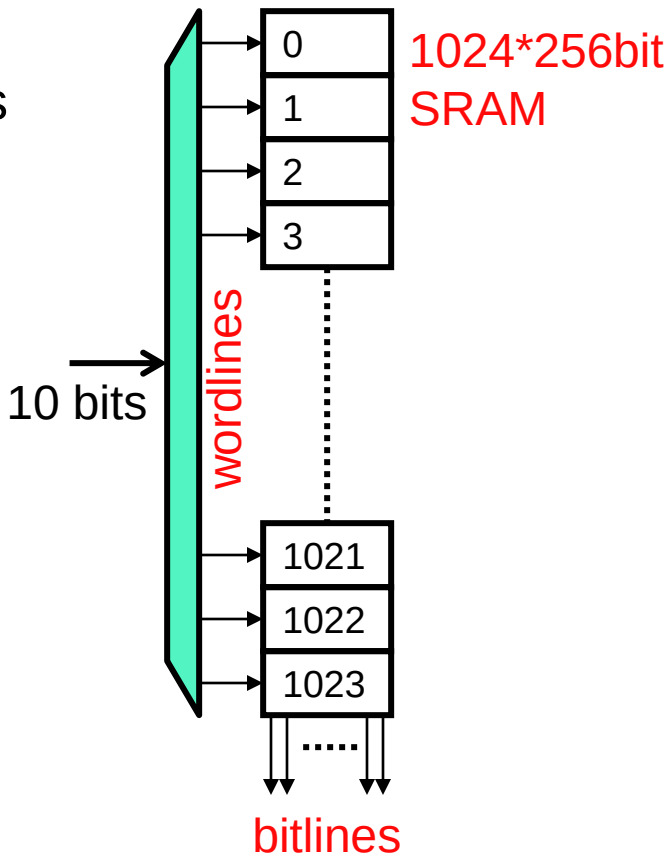
Aside: SRAM circuit implementation



- SRAM:
  - Six transistors (6T) cells
  - 4 for the cross-coupled inverters
  - 2 access transistors
- **"Static"**
  - Cross-coupled inverters hold state
- To read
  - Equalize, swing, amplify
- To write
  - Overwhelm

Basic (cache) memory array structure

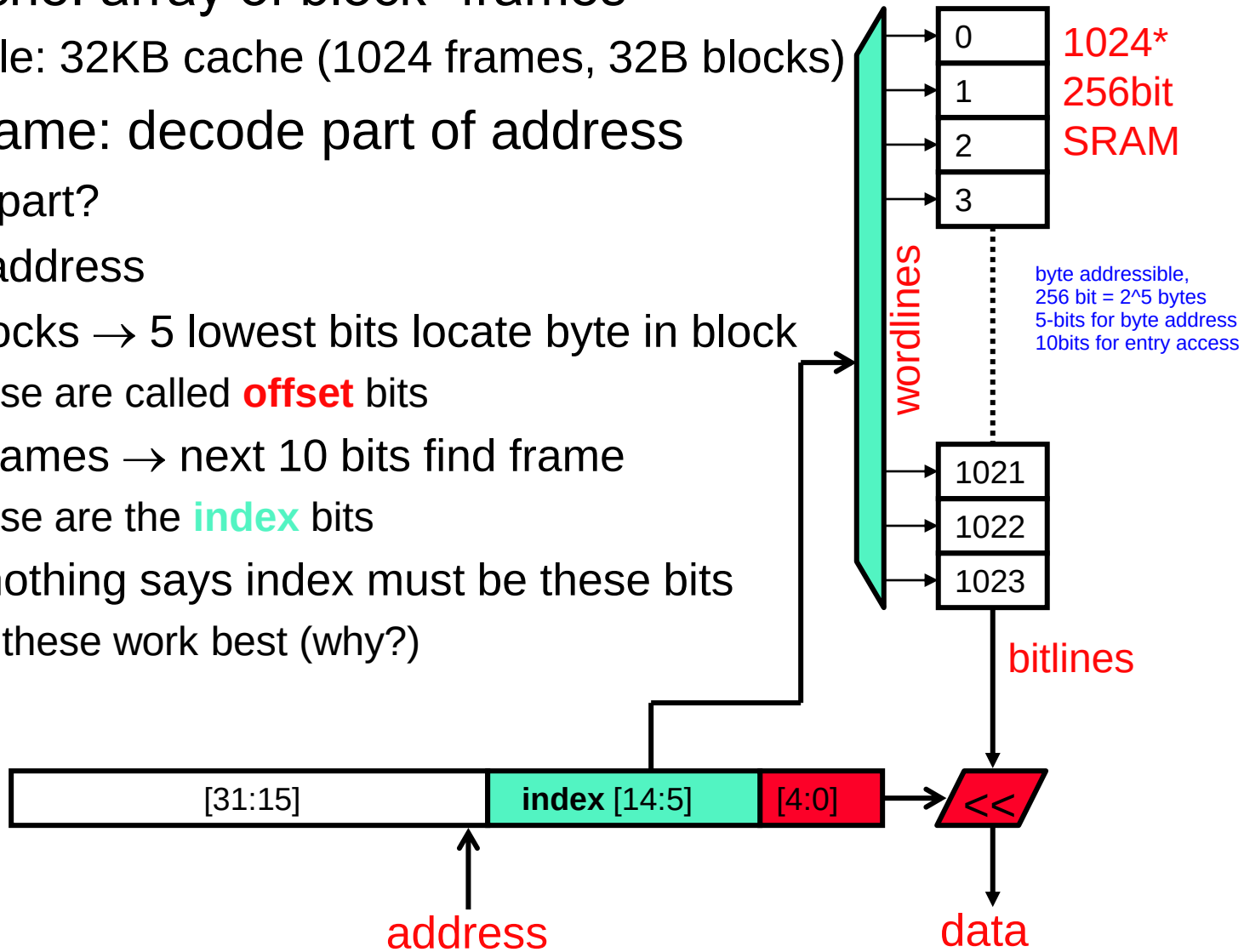
- Number of entries ("frames")
  - 2^n, where n is number of address bits
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
  - Address signals travel (horizontally) on "wordlines"



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

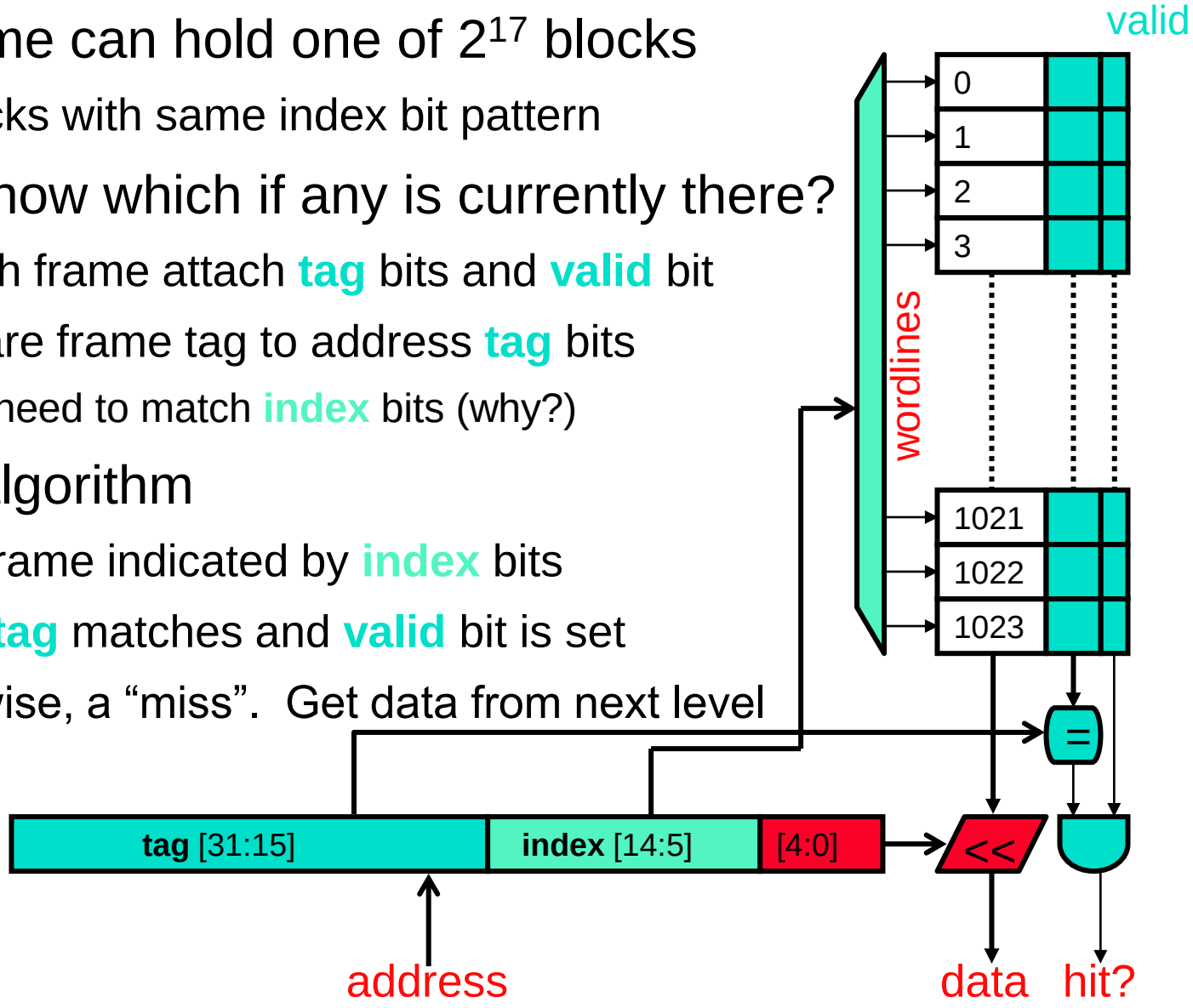
Finding data via indexing

- Basic cache: array of block "frames"
  - Example: 32KB cache (1024 frames, 32B blocks)
- 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 (why?)



Knowing that you have a hit: 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** bits 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". Get data from next level



Aside: 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 49 bits, ~20% overhead (worst case)

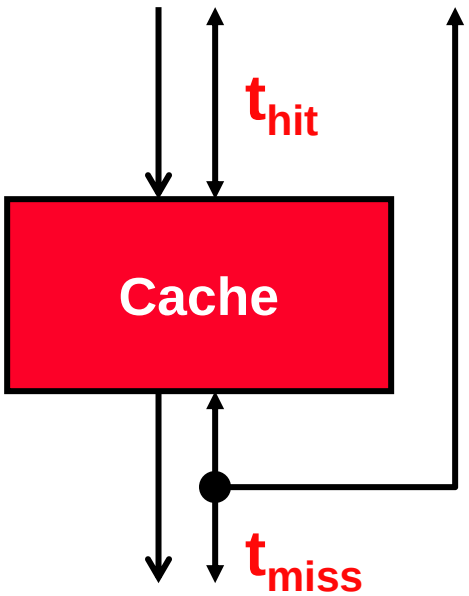
Handling a cache miss

- 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 and tag into proper locations

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

Cache performance equation



- For a cache
  - **Access**: read or write to cache
  - **Hit**: desired data found in cache
  - **Miss**: desired data not found in cache
    - Must get from another level in the hierarchy
    - No notion of "miss" in register file
  - **Fill**: action of placing data into cache
  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{hit} (hit-time): time to read data from (write data to) cache
  - t_{miss} (miss-penalty): time to read data into cache
- Performance metric: average access time

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

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?
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02 * 10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{load/store} * \%_{missD\$} * t_{missD\$}$ = 0.3 * 0.1*10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1 + 0.2 + 0.3 = 1.5
- Ultimate metric is AMAT = $t_{avg} = t_{hit} + \%_{miss} * t_{miss}$
  - Cache capacity and circuits roughly determine t_{hit}
  - Lower-level memory structures determine t_{miss}
  - How to measure $\%_{miss}$
    - Hardware performance counters
    - Simulation (e.g., SimpleScalar)
    - Paper simulation (next)

Cache miss paper simulation

- 4-bit addresses → 16B memory
- 8B cache, 2B blocks
  - Figure out number of cache frames (aka sets): 4 (capacity / block-size)
  - Figure out how address splits into tag/index/offset bits
    - Offset: least-significant $\log_2$(block-size) = $\log_2(2) = 1$ → `000`**`0`**
    - Index: next $\log_2$(number-of-sets) = $\log_2(4) = 2$ → `0`**`00`**`0`
    - Tag: rest = 4 – 1 – 2 = 1 → **`0`**`000`
- Cache diagram
  - `0000|0001` are addresses of bytes in this block, cache content values don't matter

Cache contents (in addresses)				Accessed Address	Outcome (Hit/Miss?)
Set00	Set01	Set10	Set11		
`0000|0001`	`0010|0011`	`0100|0101`	`0110|0111`		

Cache miss paper simulation

- 8B cache, 2B blocks

tag (1 bit)	**index** (2 bits)	1 bit

Cache contents (prior to access)				Address	Outcome
Set00	Set01	Set10	Set11		
0000|0001	0010|0011	0100|0101	0110|0111	**1100**	
				1110	
				1000	
				0011	
				1000	
				0000	
				1000	

- How to reduce $\%_{miss}$? And hopefully t_{avg}?

Cache miss paper simulation

- 8B cache, 2B blocks

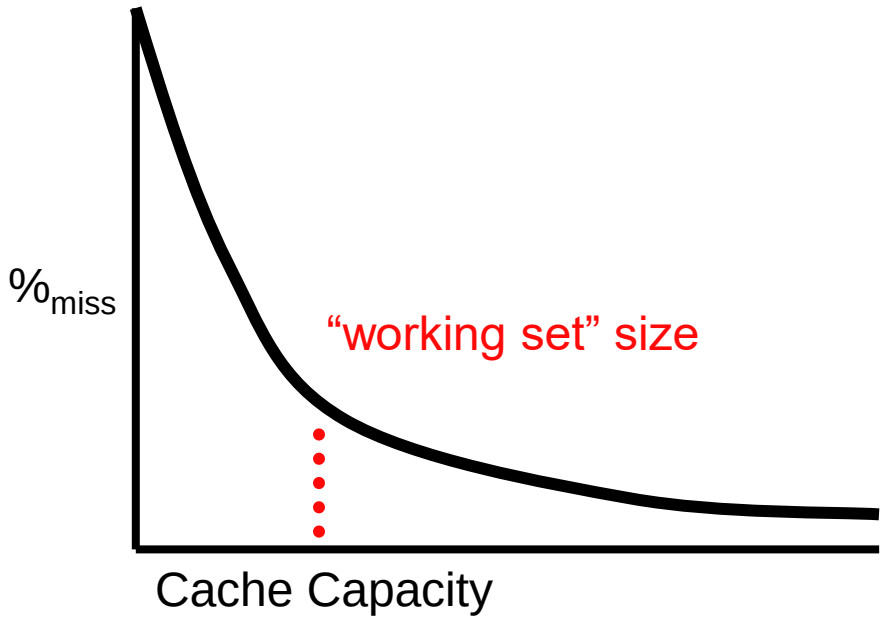
tag (1 bit)	**index** (2 bits)	1 bit

Cache contents (prior to access)				Address	Outcome
Set00	Set01	Set10	Set11		
0000|0001	0010|0011	0100|0101	0110|0111	**1100**	Miss
0000|0001	0010|0011	**1100|1101**	0110|0111	**1110**	Miss
0000|0001	0010|0011	1100|1101	**1110|1111**	**1000**	Miss
1000|1001	0010|0011	1100|1101	1110|1111	**0011**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**1000**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**0000**	Miss
0000|0001	0010|0011	1100|1101	1110|1111	**1000**	Miss

- How to reduce $\%_{miss}$? And hopefully t_{avg}?

Capacity and performance

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

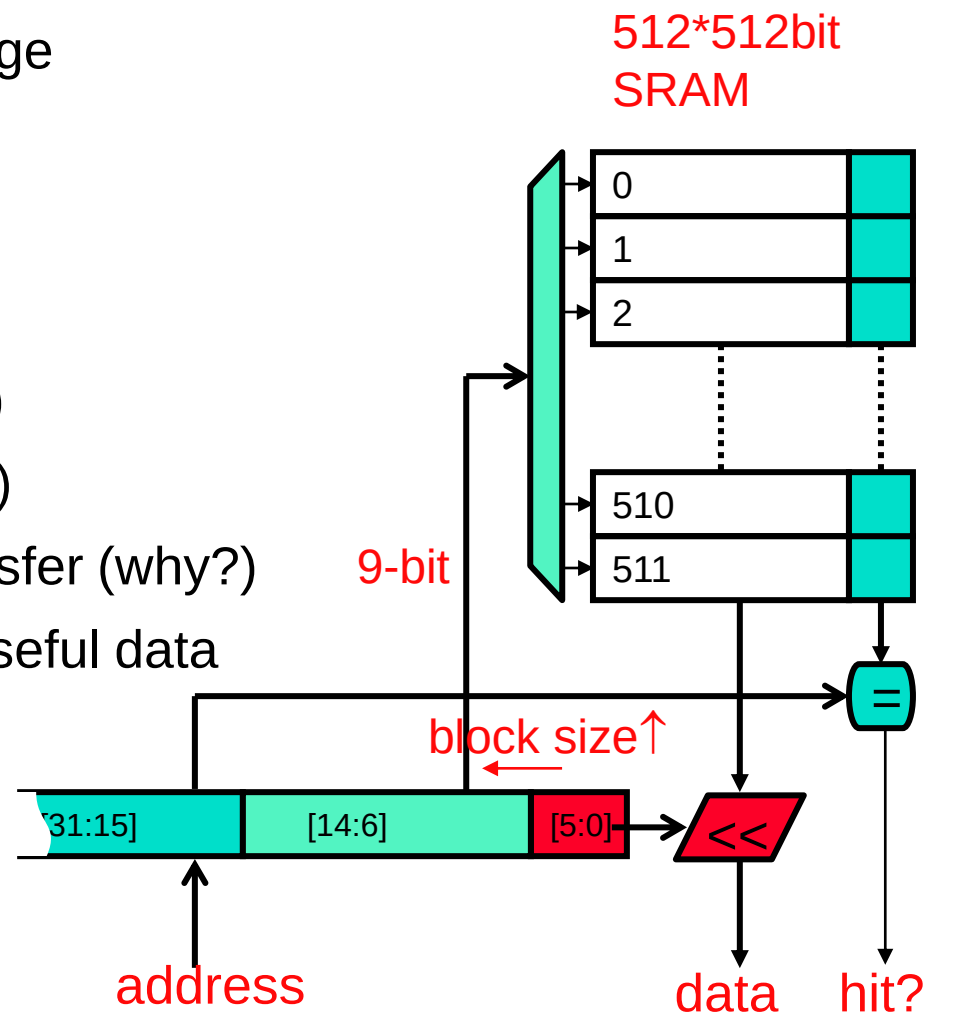


- For a given capacity, manipulate $\%_{miss}$ by changing the cache **organization**

Improving $\%_{miss}$: Increasing block size

- 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 (why?)
  - – Premature replacement of useful data



Note: valid bit not shown

Aside: 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 cache miss paper simulation

- 8B cache, **4B blocks**

tag (1 bit)	**index** (1 bit)	2 bits

Cache contents (prior to access)		Address	Outcome
Set0	Set1		
0000|0001|0010|0011	0100|0101|0110|0111	**1100**	
		1110	
		1000	
		0011	
		1000	
		0000	
		1000	

Block size cache miss paper simulation

- 8B cache, **4B blocks**

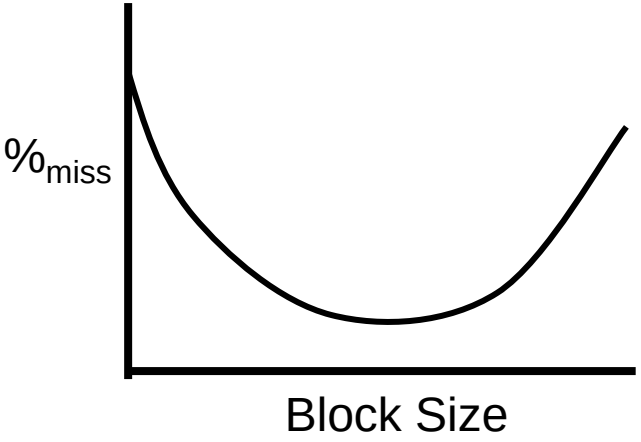
tag (1 bit)	**index** (1 bit)	2 bits

Cache contents (prior to access)		Address	Outcome
Set0	Set1		
0000|0001|0010|0011	0100|0101|0110|0111	**1100**	Miss
0000|0001|0010|0011	**1100|1101|1110|1111**	**1110**	**Hit (spatial locality)**
0000|0001|0010|0011	1100|1101|1110|1111	**1000**	Miss
1000|1001|1010|1011	1100|1101|1110|1111	**0011**	**Miss**
0000|0001|0010|0011	1100|1101|1110|1111	**1000**	**Miss**
1000|1001|1010|1011	1100|1101|1110|1111	**0000**	Miss
0000|0001|0010|0011	1100|1101|1110|1111	**1000**	Miss

+ **Spatial "prefetching"**: miss on `1100` brought in `1110`
– **Conflicts**: miss on `1000` kicked out `0011`

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 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)

(Ping-pong) Conflicts

- 8B cache, 2B blocks

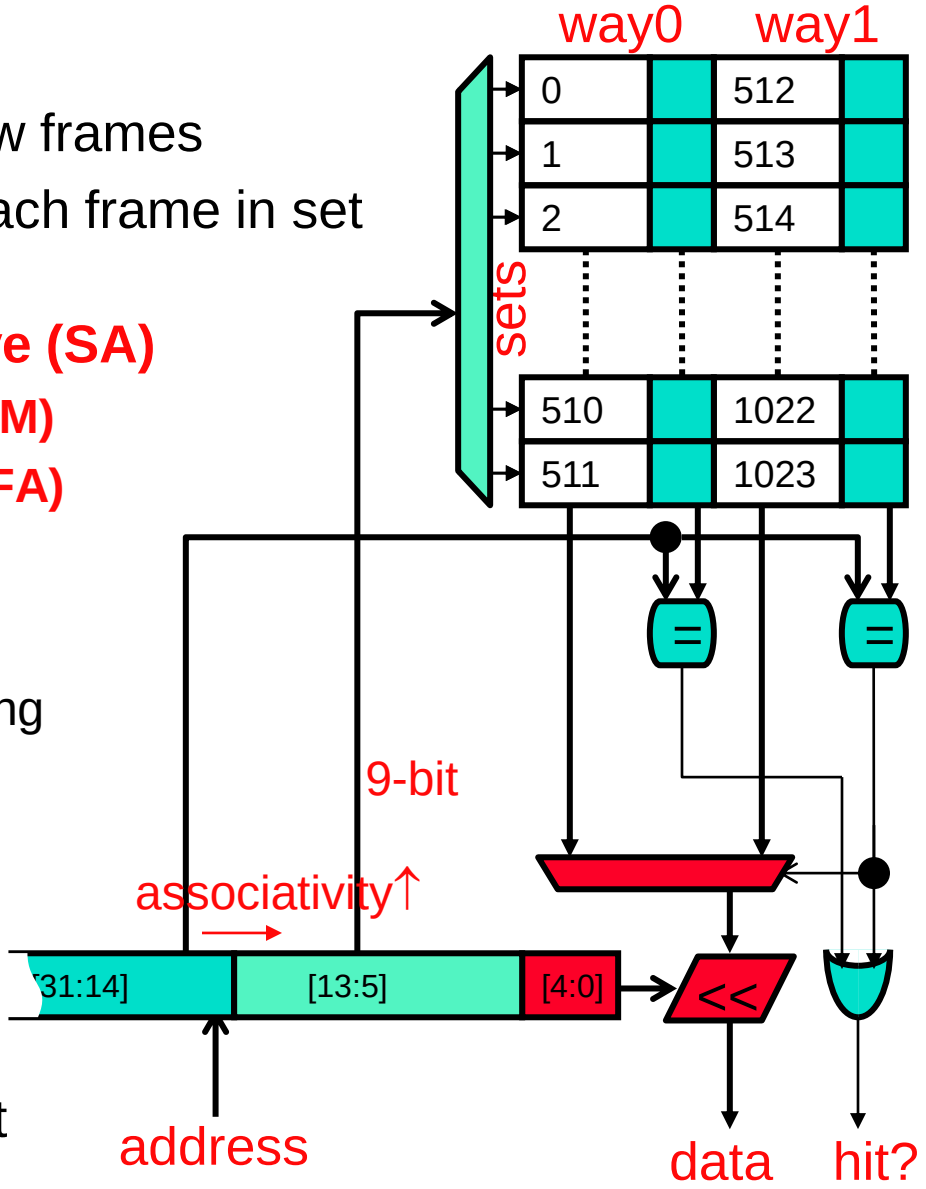
tag (1 bit)	**index** (2 bits)	1 bit

Cache contents (prior to access)				Address	Outcome
Set00	Set01	Set10	Set11		
0000|0001	0010|0011	0100|0101	0110|0111	**1100**	Miss
0000|0001	0010|0011	**1100**|1101	0110|0111	**1110**	Miss
0000|0001	0010|0011	1100|1101	**1110**|1111	**1000**	Miss
1000|1001	0010|0011	1100|1101	1110|1111	**0011**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**1000**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**0000**	Miss
0000|0001	0010|0011	1100|1101	1110|1111	**1000**	Miss

- Pairs like 0000/1000 **conflict**
  - Regardless of block-size (assuming capacity < 16)
  - Q: can we allow pairs like these to simultaneously reside?
  - A: yes, reorganize cache to do so

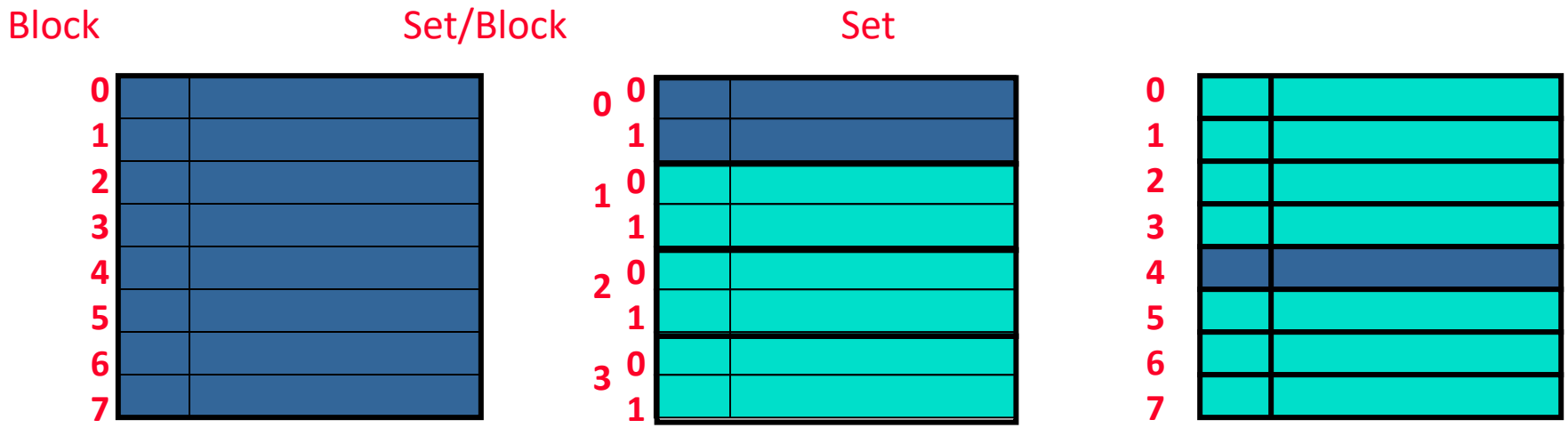
Set-Associativity

- **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 $t_{hit:}$
    - additional tag match & muxing
- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames in that set in parallel
  - **Any** (match and valid bit), Hit



All caches are set associative – some are degenerate

Where does block 12 (b'1100) go?



<u>Fully-associative</u>
block goes in any frame

<u>Set-associative</u>
a block goes in any frame in exactly one set

<u>Direct-mapped</u>
block goes in exactly one frame

(think all frames in 1 set) ⟷ (frames grouped into sets) ⟷ (think 1 frame per set)

Associativity and miss manual simulation

tag (2 bits)	index (1 bit)	1 bit

- 8B cache, 2B blocks, **2-way set-associative**

Cache contents (prior to access)				Address	Outcome
Set0.Way0	Set0.Way1	Set1.Way0	Set1.Way1		
0000|0001	0100|0101	0010|0011	0110|0111	**1100**	Miss
1100|1101	0100|0101	0010|0011	0110|0111	**1110**	Miss
1100|1101	0100|0101	**1110|1111**	0110|0111	**1000**	Miss
1100|1101	**1000|1001**	1110|1111	0110|0111	**0011**	**Miss (new conflict)**
1100|1101	1000|1001	1110|1111	**0010|0011**	**1000**	Hit
1100|1101	1000|1001	1110|1111	0010|0011	**0000**	Miss
0000|0001	1000|1001	1110|1111	0010|0011	**1000**	**Hit (avoid conflict)**

+ **Avoid conflicts**: 0000 and 1000 can **both** be in (different ways of) set 0
– **Introduce some new conflicts**: notice address re-arrangement
  - Happens, but conflict avoidance usually dominates

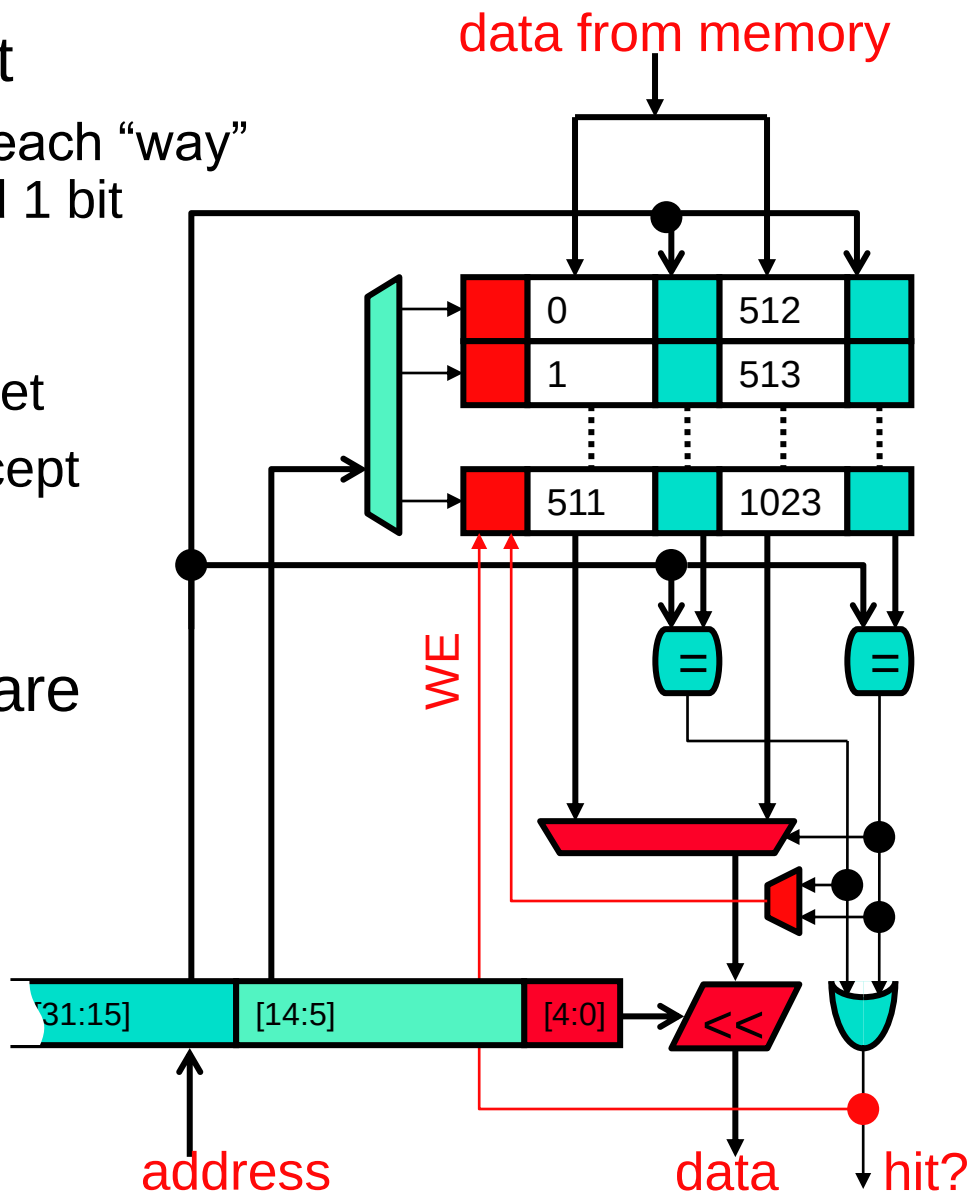
Replacement policies

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

  - Which policy is simulated in previous example?

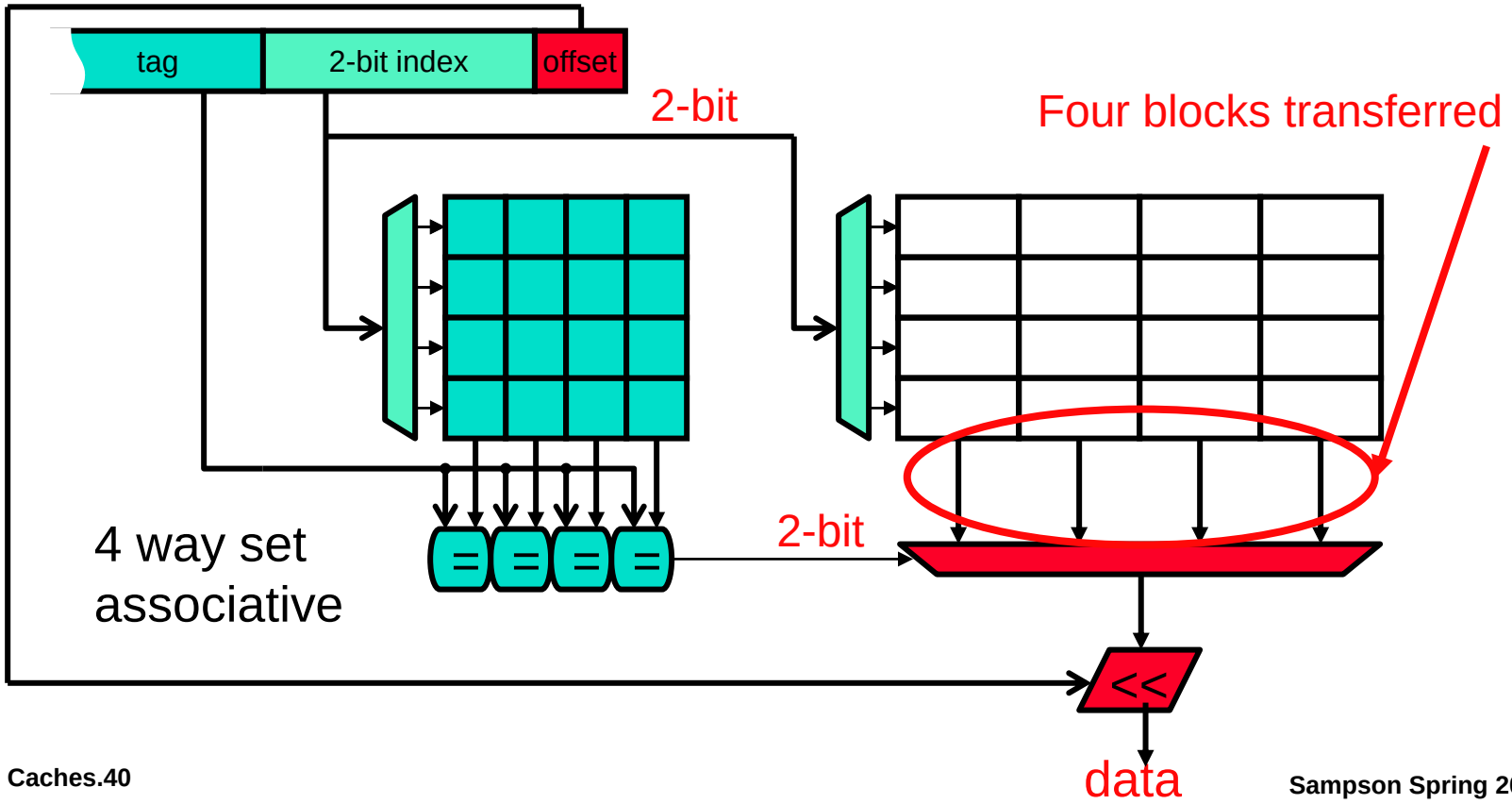
NMRU and miss handling

- Add **MRU** bits to each set
  - MRU data is encoded for each "way" (e.g., for 2 ways only need 1 bit (which way was MRU), 4 ways need 2 bits, etc.)
  - Hit? update MRU way in set
  - Miss, replace any way except the MRU one
- All the MRU bits in a set are updated on each access to that set



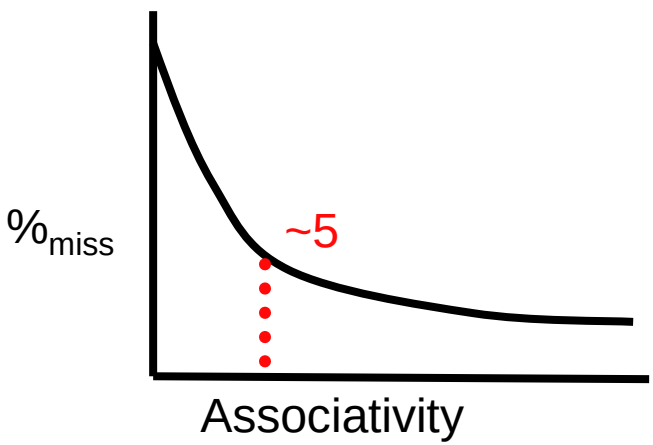
Parallel or serial tag access?

- Note: data and tags actually physically separate
  - So can split into two different arrays
- So can access the tag array and data array in parallel
  - But there is still more logic in the critical path than direct mapped caches (an additional multiplexor) so slower t_{hit} time



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)
- Number of ways don't have to be a power of two … e.g., 3-way set-associativity? No problem

Write issues

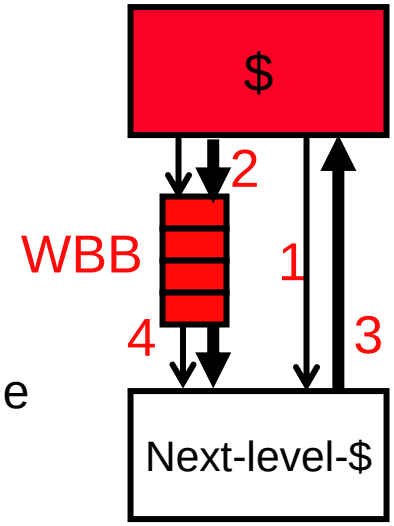
- So far we have focused on reading from cache
  - Instruction fetches, loads
- What about writing into cache
  - Stores, not an issue for instruction caches (why they are simpler/smaller (in area))
- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
  - 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 good data (oops)
  - For associative caches, which way was written into?
- Writes are a pipelined, two stage (multi-cycle) process
  - Stage 1: match tag
  - Stage 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)**: keeps writes off the the critical path
    - Step#1: Send "fill" request to next-level
    - Step#2: While waiting, write dirty block to buffer
    - Step#3: When new blocks arrives, put it into cache
    - Step#4: Write buffer sends contents to next-level

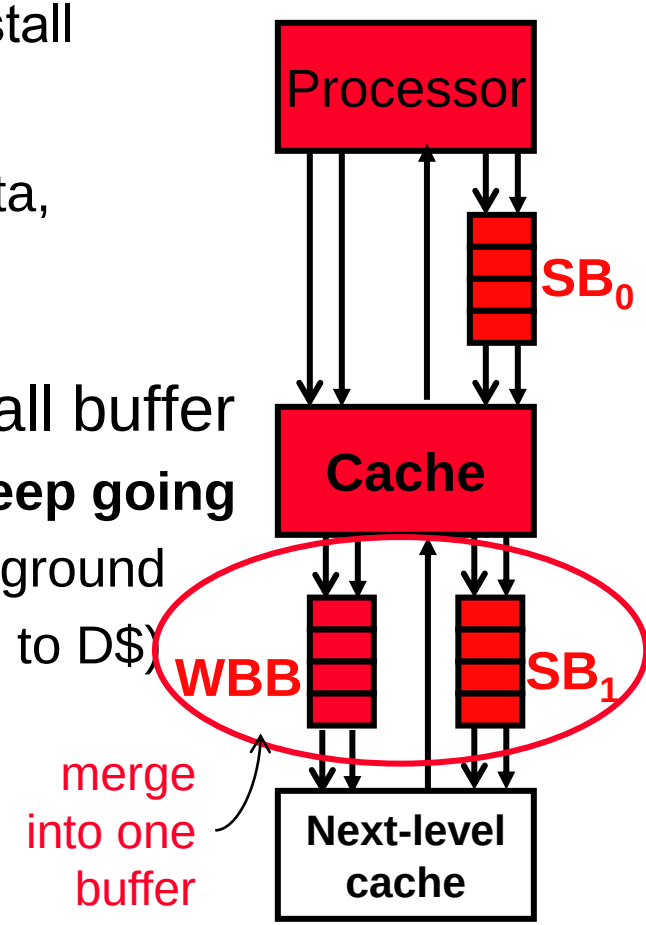


Write propagation comparison

- **Write-through**
  - Requires additional bus bandwidth
    - Consider repeated write hits (like a loop index counter)
  - Without a write buffer, must wait for writes to complete to memory
  - \+ Easier to implement, no need for dirty bits in cache
  - \+ Don't have to deal with coherence traffic at this cache level
  - \+ No need to handle "writeback" operations on replacement
    - Simplifies miss handling (no write-back buffer step)
  - Sometimes used for L1 caches (for example, by IBM)
- **Write-back**
  - \+ Key advantage: uses less bandwidth since some writes don't go to memory (so also saves power)
  - Reverse of other pros/cons above
  - Used by Intel and AMD
  - Second-level and beyond are generally write-back caches

Write misses and store buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, so why stall?
- **Store Buffer** (aka **Write Buffer**): a small buffer
  - Stores put address/value to store buffer, **keep going**
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer (in addition to D$)
  - \+ Eliminates stalls on write misses (mostly)
- Store buffer vs. writeback-buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding write backs



Write miss handling

- How is a write miss actually handled?

- **Write-allocate**: allocate a frame in the cache for the miss data
  - + Decreases read misses (next read to block will hit)
  - ▫ Commonly used (especially with write-back caches)

- **No-write-allocate**: just write to next level, no need to allocate a cache frame for the miss data
  - – Potentially more read misses
  - + Doesn't use a frame in the cache (increases capacity?)
  - ▫ Use with write-through

Classifying misses: 3C (+1) 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 frames 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 and multicores (more later)

- Calculated by multiple simulations
  - Simulate infinite cache, fully-associative cache, normal cache
  - Subtract to find each count

ABC's of miss rates

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

Basic-1: Larger block size

- \+ Decreases compulsory misses (spatial locality)
- – Increases conflict misses (fewer frames)
- – Can increase t_{miss} (now reading more bytes from the next level)
- No significant effect on t_{hit}

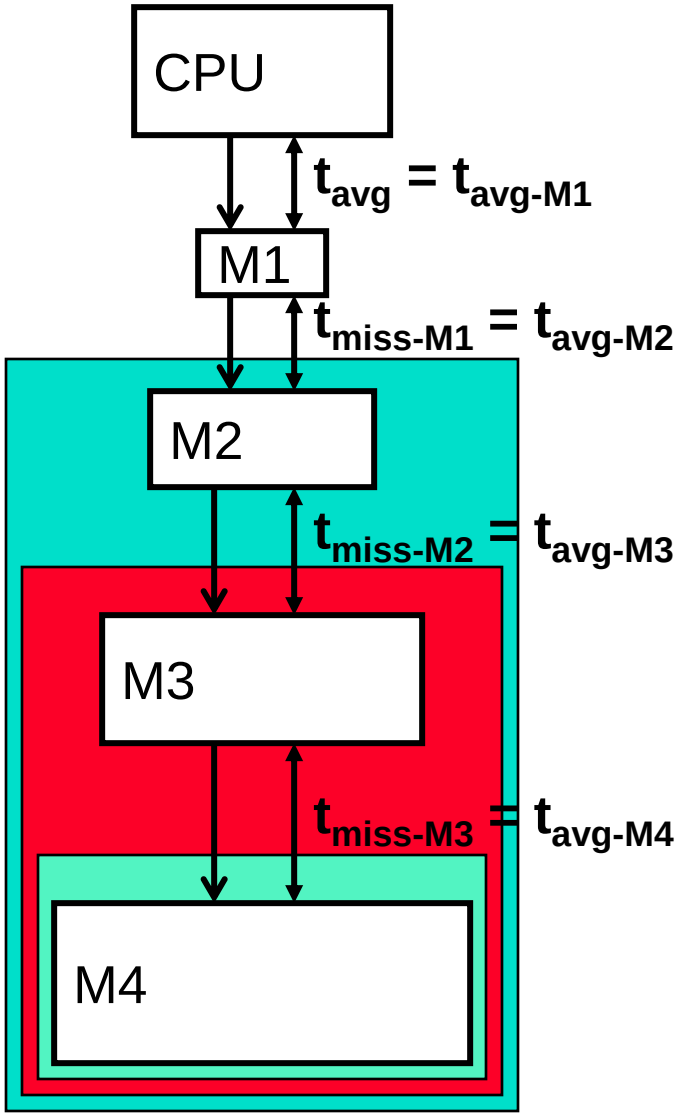
Basic-2: Bigger caches

- \+ Decreases capacity misses
- – Increases t_{hit}

Basic-3: Higher associativity

- \+ Decreases conflict misses
- – Increases t_{hit}

Basic-4: Multilevel caches



- L1 cache can be small enough to have a fast hit time while L2 cache can be large enough to help reduce capacity misses

t_{avg}

$= t_{avg-M1}$

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

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

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

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

…

Split vs. unified caches

- **Split I$/D$**: instr's and data in different caches at L1
  - To minimize structural hazards and t_{hit}
    - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ for wide output (superscalar) and no writes
  - Why is 486 I/D$ unified?

- **Unified L2, L3**: instr's and data together
  - To minimize $\%_{miss}$
  - \+ Fewer capacity misses: unused instr capacity can be used for data
  - – More conflict misses: instr/data conflicts
    - A much smaller effect in large caches
  - Instr/data structural hazards are rare: simultaneous I$/D$ miss

Designing a cache hierarchy

- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → t_{hit} important
    - Low capacity/associativity (to reduce t_{hit})
  - t_{miss} is not bad → $\%_{miss}$ less important
    - Small-medium block size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{hit} less important
  - t_{miss} is bad → $\%_{miss}$ important
    - High capacity/associativity/block size (to reduce $\%_{miss}$)

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

Designing a cache hierarchy

- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → t_{hit} important
    - Low capacity/associativity (to reduce t_{hit})
  - t_{miss} is not bad → $\%_{miss}$ less important
    - Small-medium block size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{hit} less important
  - t_{miss} is bad → $\%_{miss}$ important
    - High capacity/associativity/block size (to reduce $\%_{miss}$)

NOT A CACHE

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

Designing a cache hierarchy

- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → t_{hit} important
    - Low capacity/associativity (to reduce t_{hit})
  - t_{miss} is not bad → $\%_{miss}$ less important
    - Small-medium block size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{hit} less important
  - t_{miss} is bad → $\%_{miss}$ important
    - High capacity/associativity/block size (to reduce $\%_{miss}$)

~~NOT A CACHE~~
(VM+ paging: sort of a cache)

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

Designing a cache hierarchy

- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → t_{hit} important
    - Low capacity/associativity (to reduce t_{hit})
  - t_{miss} is not bad → $\%_{miss}$ less important
    - Small-medium block size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{hit} less important
  - t_{miss} is bad → $\%_{miss}$ important
    - High capacity/associativity/block size (to reduce $\%_{miss}$)

SSDs change things by several orders of magnitude!

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

Local vs global miss rates

- Local hit/miss rate:
  - Percent of references to this cache that hit (e.g, 90%)
  - Local miss rate is (100% - local hit rate), (e.g., 10%)
    - # misses / total # of accesses to **this** cache
- Consider second-level cache hit/miss rate
  - L1: 2 misses per 100 references
    - L1 "local miss rate" → 2%
  - L2: 1 miss per 100 references
    - L2 "local miss rate" → 50%
- Global hit/miss rate:
  - # misses / total # of memory references
    - Global miss rate for L1 is just $\%_{miss\text{-}L1}$
    - Global miss rate for L2 is $\%_{miss\text{-}L1} \times \%_{miss\text{-}L2}$

AMAT without & with a L2 cache

- 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 $t_{avgD\$}$ (in ns) without an L2?
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{hitM}$ = 1ns+(0.05*50ns) = 3.5ns
- What is $t_{avgD\$}$ (in ns) with an L2?
  - $t_{missD\$} = t_{avgL2}$
  - $t_{avgL2} = t_{hitL2} + \%_{missL2} * t_{hitM}$ = 10ns+(0.2*50ns) = 20ns
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{avgL2}$ = 1ns+(0.05*20ns) = 2ns

Inclusive versus Exclusive caches

- **Inclusive caches**
  - A block in the L1 is always in the L2
  - Good for write-through L1s (why?)
  - What to do if block size of L2 is larger than L1
    - A miss in L2 replaces data equivalent to multiple L1 blocks
  - Coherency traffic only affects the L2 cache (if write-through)
- **Exclusive caches**
  - Block is either in L1 or L2 (never in both) – if L1 misses and L2 hits, block in L2 is exchanged with that in L1
    - L1 and L2 must have the same block size (more L2 tag overhead?)
    - Exchanging more work than just copying a line from L2 to L1 (as in inclusive caches)
  - Coherence traffic must check both L1 and L2
  - Holds more data, especially if L2 is small relative to L1
    - Example: AMD's Duron 64KB L1s, 64KB L2

The Inclusion Property

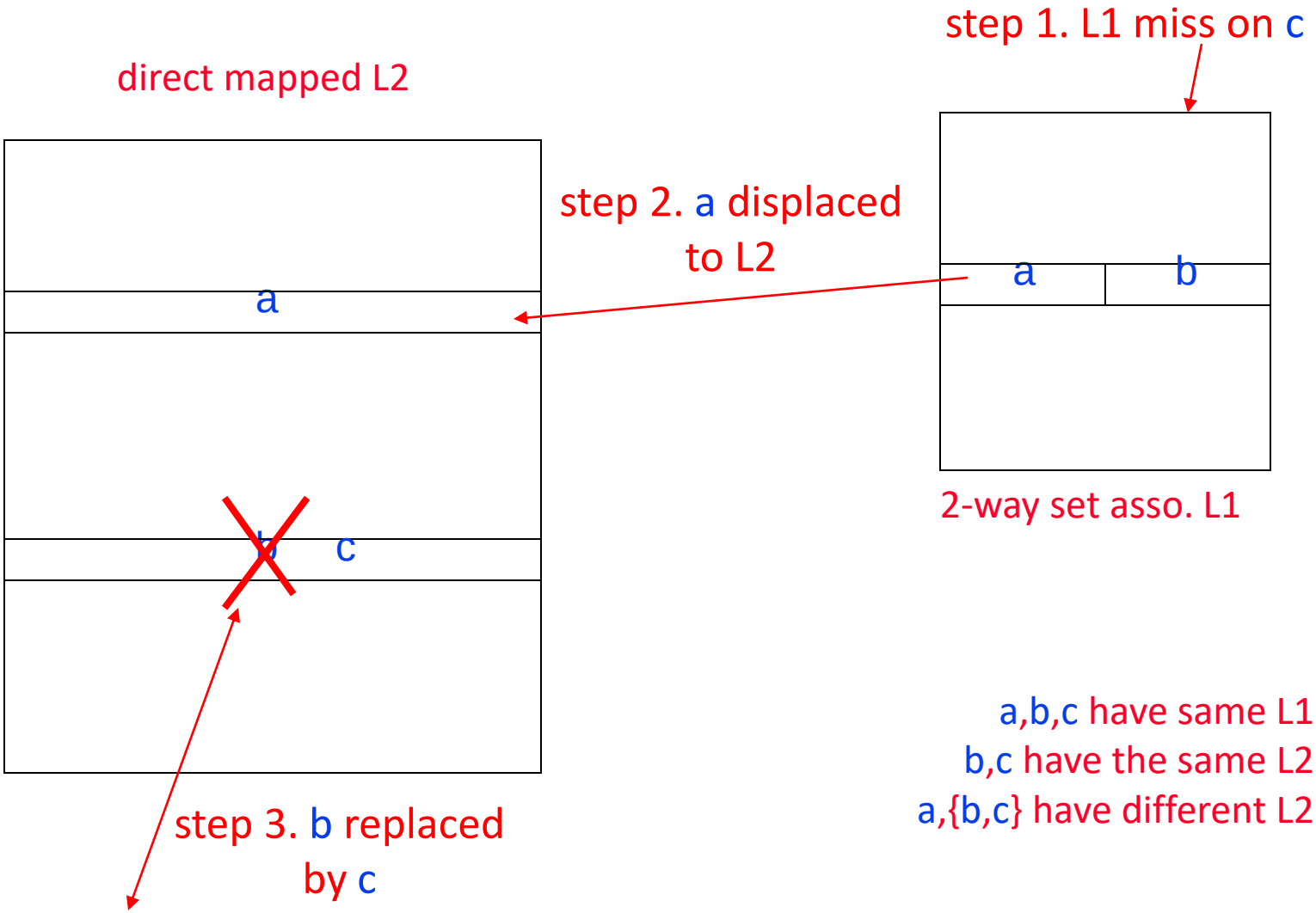
- Inclusion means L2 is a superset of L1 (ditto for L3…)
- Why?
  - if an addr is in L1, then it must be frequently used
  - makes L1 writeback simpler
  - L2 can handle external coherence checks without L1
- **Inclusion takes effort to maintain**
  - L2 must track what is cached in L1
  - On L2 replacement, must flush corresponding blocks from L1

How can this happen?

Consider:

1. *L1 block size < L2 block size*
2. *different associativity in L1*
3. *L1 filters L2 access sequence; affects LRU replacement order*

Possible Inclusion Violation



a,b,c have same L1 idx bits
b,c have the same L2 idx bits
a,{b,c} have different L2 idx bits

Multi-Level Inclusion (Cont.)

- Interesting interaction between L1I, L1D, and L2
  - Example a matrix multiply program for a large matrix
  - What happens if you maintain inclusion for L1I?

Basic-5: Give reads priority over writes

- Serve reads before write-backs have been completed
  - The read must check the contents of the WBB (since it could hold the data value) on a read miss
    - If there are is no match and if the memory system is available, let the read miss continue ahead of the write-back
  - How big to make the WBB ?
    - bigger is better so can hold more write-back blocks, but bigger would mean slower match checks

```
SW   R3,512(R0)
LW   R1,1024(R0)
LW   R2,512(R0)
```

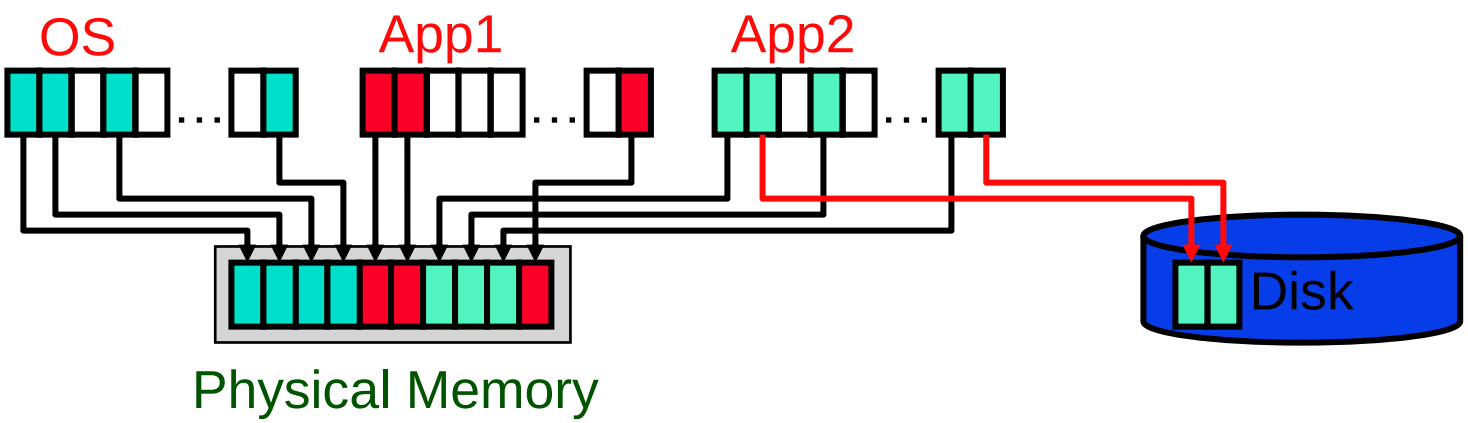
If 512 and 1024 map to the same cache block:

- 1st load misses, cache writes dirty block to WBB
- 2nd load misses, if WBB hasn't completed, load will get old data value from memory

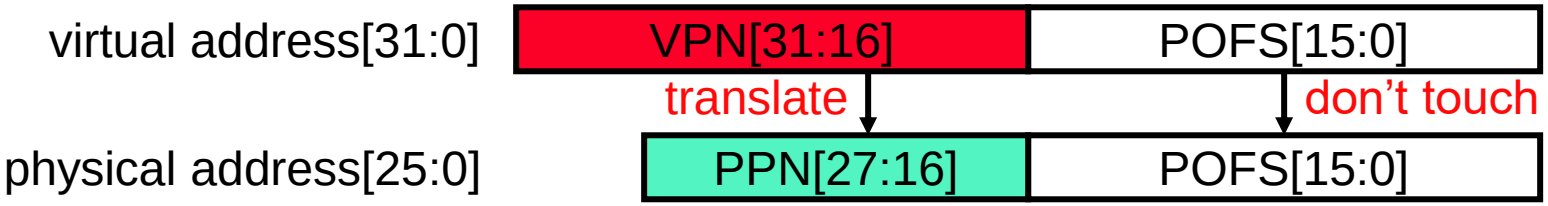
- Reduces write costs in a write-back cache
  - If the read miss will replace a dirty block – write the dirty block to WBB, then read memory, and *then* write the WBB to memory
    - Read will finish sooner (processor is probably waiting for it)

Virtual memory review

- Programs use **virtual addresses (VA)**
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit)
- Memory uses **physical addresses (PA)**
  - PA size (M) typically M<N, especially if N=64
  - 2^M is the most physical memory that a machine supports
- VA→PA mapped at **page** granularity (VP→PP)
  - Mapping need not preserve contiguity
  - Unmapped VP's live on disk (swap) or nowhere (if not yet touched)



Address translation review

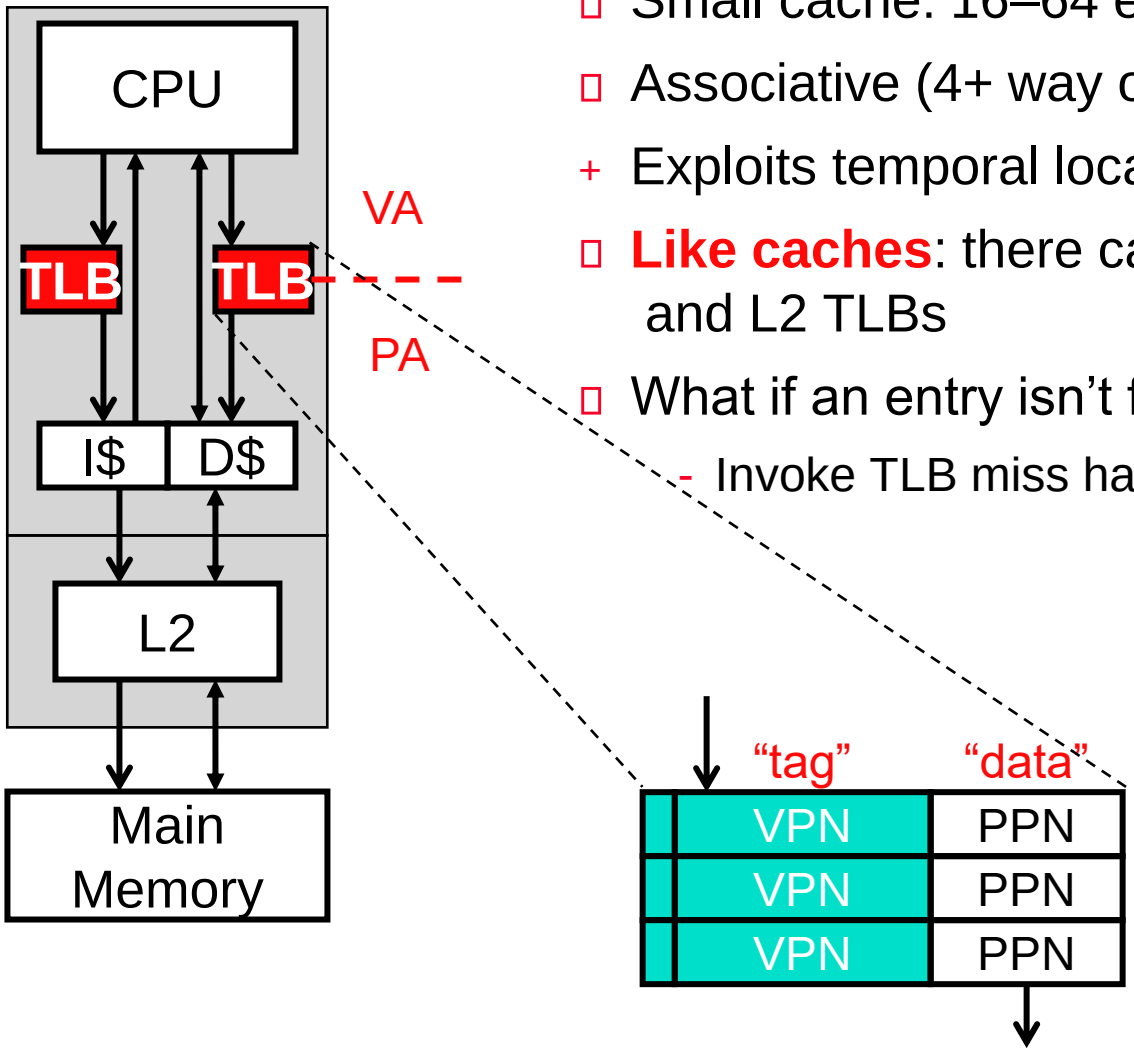


- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]

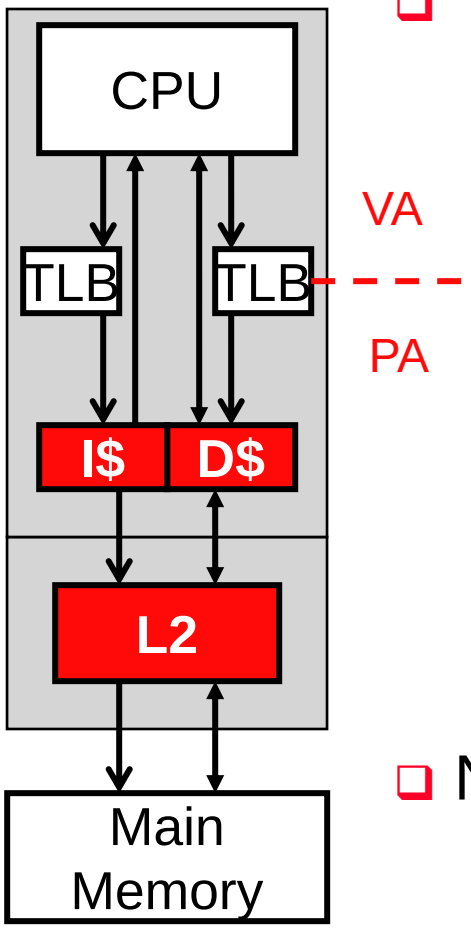
- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Address translation hardware support: TLBs

- **Translation Lookaside Buffer (TLB)**
  - Small cache: 16–64 entries
  - Associative (4+ way or fully associative)
  - \+ Exploits temporal locality in page table
  - **Like caches**: there can be split ITLB and DTLB and L2 TLBs
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler



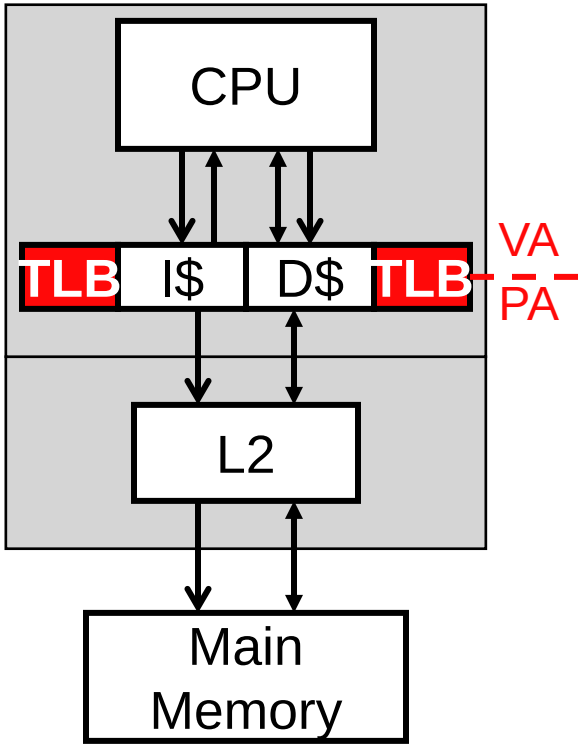
Serial TLB & cache access (PI/PT)

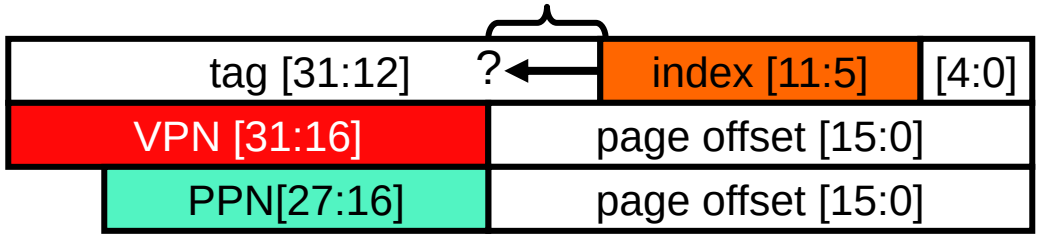


- **"Physical" caches**
  - Indexed and tagged by **physical addresses**
  - \+ Natural, "lazy" sharing of caches between apps/OS
    - VM ensures isolation (via **physical addresses**)
    - No need to do anything on context switches
    - Multi-threading works too
  - \+ Cached inter-process communication works
    - Single copy indexed by the physical address
  - – **Slow**: adds at least one cycle to t_{hit}

- Note: **TLBs are by definition "virtual"**
  - Indexed and tagged by **virtual addresses**
  - Flushed across context switches
    - Or extend with process identifier tags (x86)

Basic 6: Parallel TLB & cache access (VI/PT)

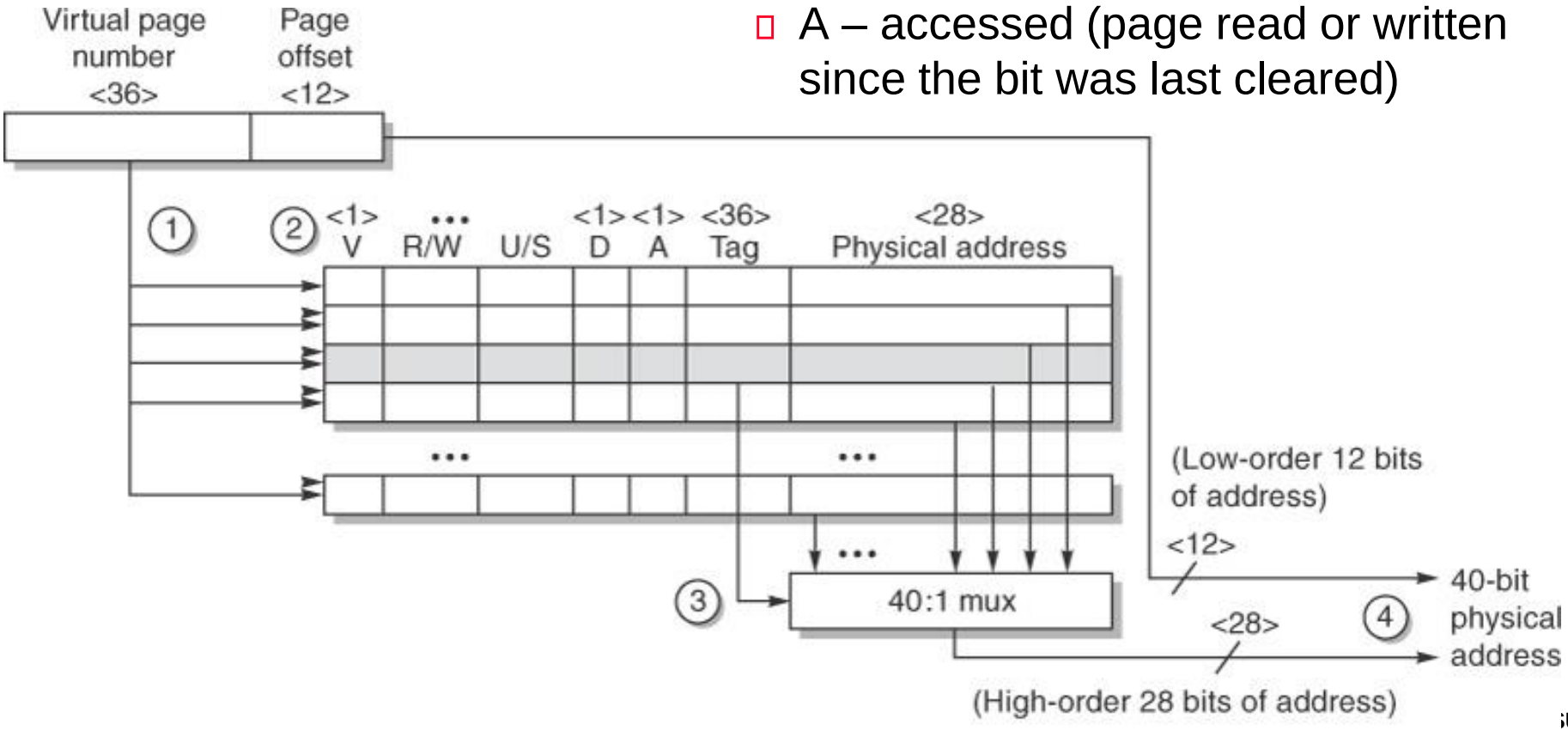




- What about parallel access?
  - Only if…
    (cache size) / (associativity) ≤ page size
    - Index bits same in virtual and physical addr's!
- Access TLB in parallel with cache
  - Cache access needs tag only at very end
  - \+ Fast: no additional t_{hit} cycles
  - \+ No context-switching/aliasing problems
  - Dominant organization used today
- Example: Core 2, 4KB pages, 32KB, 8-way SA L1 D$
  - Implication: associativity allows bigger caches

AMD Opteron L1 DTLB (48-bit VA, 4KB pages)

- Four steps of a fully associative TLB hit in circled numbers
  - R/W – read-only or read-write
  - U/S – user or privileged mode
  - D – dirty (page not TLB entry!)
  - A – accessed (page read or written since the bit was last cleared)



TLB misses

- **TLB miss:** translation not in TLB, but in page table
  - Two ways to "fill" it, both relatively fast

1. **Software-managed TLB**: e.g., Alpha, MIPS, ARM
   - Short (~10 instr) OS routine walks page table, updates TLB
   - \+ Keeps page table format flexible
   - – Latency: one or two memory accesses + OS switch (pipeline flush)

1. **Hardware-managed TLB**: e.g., x86
   - Page table root in hardware register, hardware "walks" table
   - \+ Latency: saves cost of OS call (avoids pipeline flush)
   - – Page table format is hard-coded

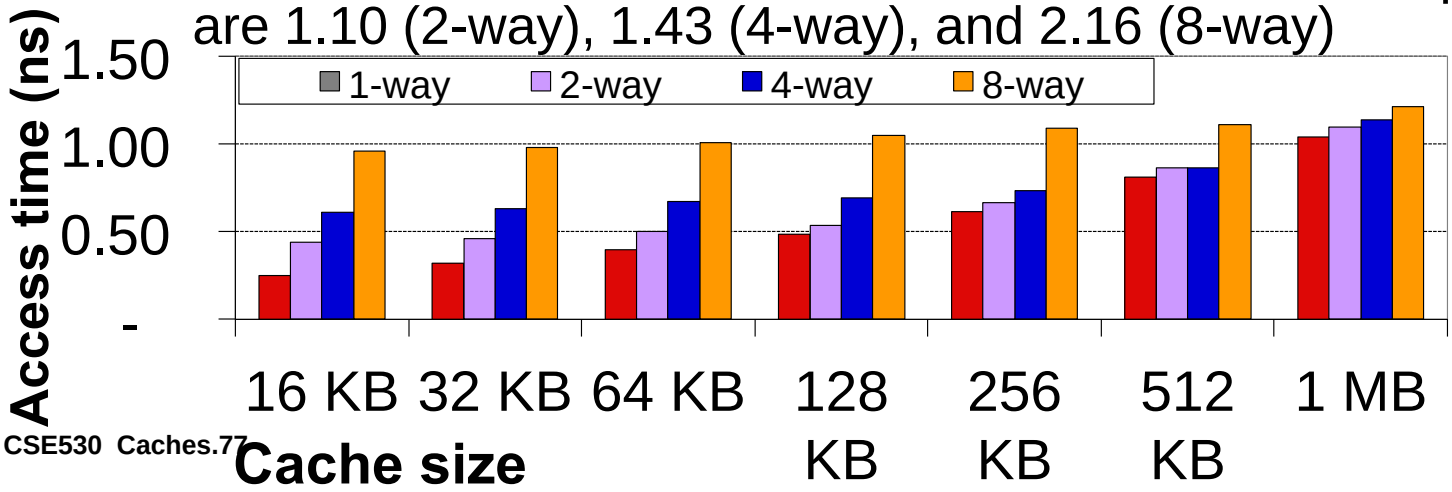
- Trend is towards hardware TLB miss handler

Advanced cache performance optimizations

- For reducing t_{hit}: small, simple caches, way prediction, trace caches
- For increasing cache bandwidth: pipelined caches, non-blocking caches, multibanked caches
  - Cache latency vs. Cache bandwidth
  - NUCA
- For reducing t_{miss}: critical word first, write merging buffers
- For reducing $\%_{miss}$: victim cache, compiler optimizations
- For reducing t_{miss} or $\%_{miss}$ via parallelism: hardware and software (compiler) prefetching

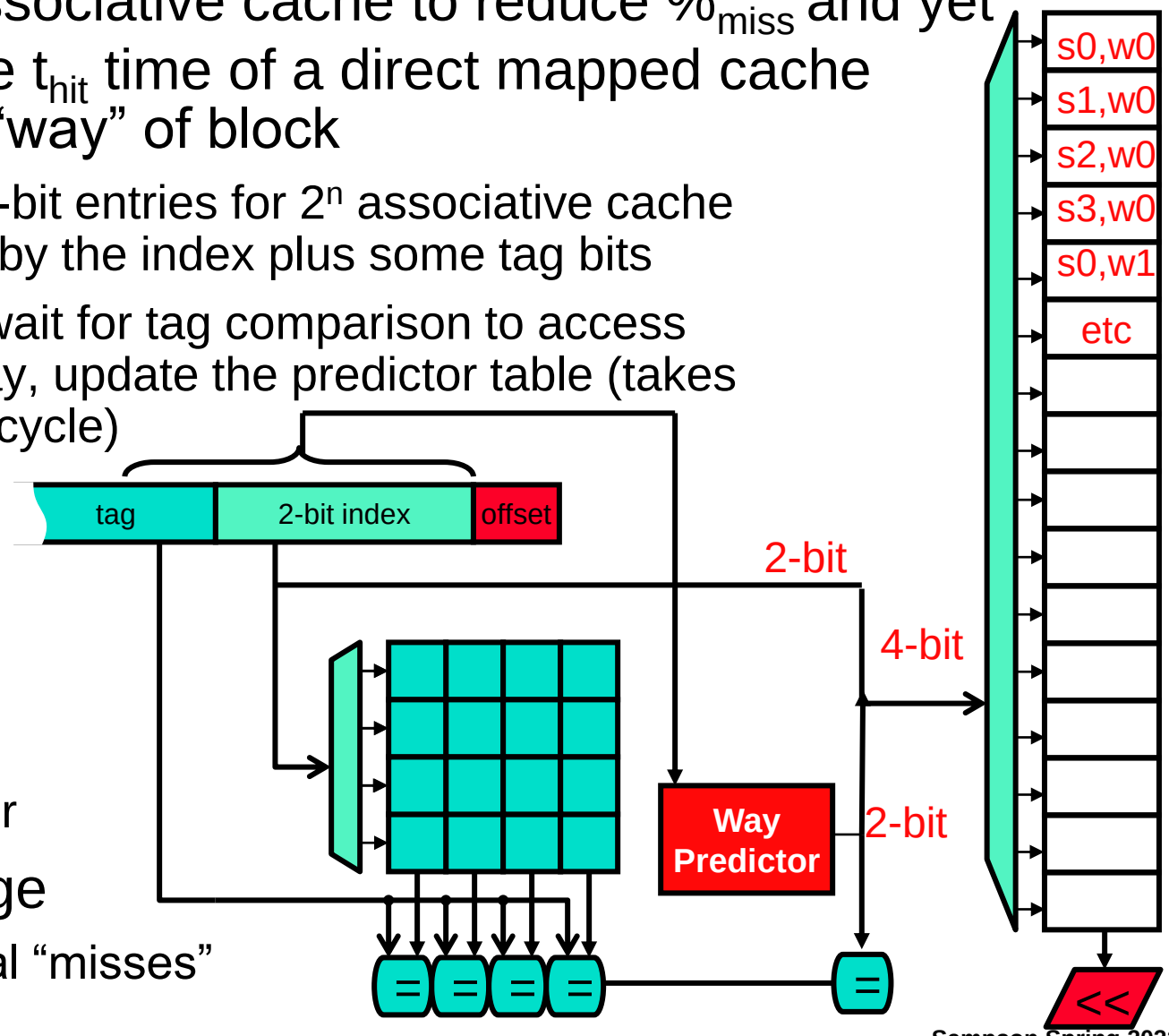
Adv-1: Small, simple caches to reduce t_{hit}

- Smaller is faster
  - Also critical to keep the L2 small enough to fit on chip or at least keep the tags on chip to allow fast tag checks and the larger L2 capacity offered by having it off chip
- Simpler is faster
  - Direct-mapped caches can overlap the tag check with the transmission of data to reduce t_{hit}
- Access times estimate for 32 nm, single bank, 64-byte blocks using CACTI 5.3 r 174 (web)
  - Median ratios of access time relative to the direct-mapped caches are 1.10 (2-way), 1.43 (4-way), and 2.16 (8-way)



Adv-2: Way prediction to reduce hit time

- Want set-associative cache to reduce $\%_{miss}$ and yet still have the t_{hit} time of a direct mapped cache
- Predict the "way" of block
  - Table of *n*-bit entries for 2^n associative cache accessed by the index plus some tag bits
  - If wrong, wait for tag comparison to access correct way, update the predictor table (takes one extra cycle)



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

Adv-3: Trace caches (instruction $)

- Blocks in a trace cache contain dynamic traces of the executed instr's vs. static sequences of instr's as determined by the memory layout
  - Built-in "branch predictor"

- Trace caches were used in the Pentium 4 (held the decoded uops, so also saved decode time)
  - \+ Better utilizes long blocks (don't exit in middle of block, don't enter in middle of block)
  - \- Complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size
  - \- Instructions may appear multiple times in multiple dynamic traces due to different branch outcomes
  - \- Can be expensive in area

Adv-4: Pipelined Cache Accesses

- Pipeline L1 cache access (so multiple cycles) – allows a faster processor clock and high cache bandwidth, but larger t_{hit} (cache latency measured in processor clocks)

- Instruction cache access pipeline stages:

  1: Pentium

  2: Pentium Pro through Pentium III

  4: Pentium 4

  - $\Rightarrow$ greater penalty on mispredicted branches
  - $\Rightarrow$ more clock cycles between the issue of the load and the use of the data

Pentium III and 4 pipelines

Pentium III Pipeline

Fetch 1	Fetch 2	De code 1	De code 2	De code 3	Re-name	ROB Rd	Rdy/ Sch	Disp	Exec

Pentium 4 Pipeline

Fetch 1	Fetch 2	Fetch 3	Fetch 4	Drive	Alloc	Renme 1	Renme 2	Queue	Sch 1	Sch 2	Sch 3	Disp 1	Disp 2	RF 1	RF 2	Exec	Flags	Br Ck	Drive

Adv-5: Lockup free cache

- **Lockup free (aka non-blocking)**: allows other accesses to be handled by the cache while a miss is pending
  - Consider: Load [r1] -> r2; Load [r3] -> r4; Add r2, r4 -> r5
    - Makes sense to let the processor go ahead and try to load r4 despite a D$ miss encountered when trying to load r2 (out-of-order)
- Implementation
  - **Miss Status Holding Registers (MSHR)**
    - To keep track of miss address, chosen frame, requesting instruction
      - So when miss returns know where to put block, who to inform
  - And (if serving multiple misses) multiple memory banks
- Common scenario: "hit under miss", "hit under misses"
  - Handle hits while miss(es) is/are pending; easy
- Less common, but common enough: "miss under miss"
  - A little trickier – must check MSHRs to avoid cache frame conflicts

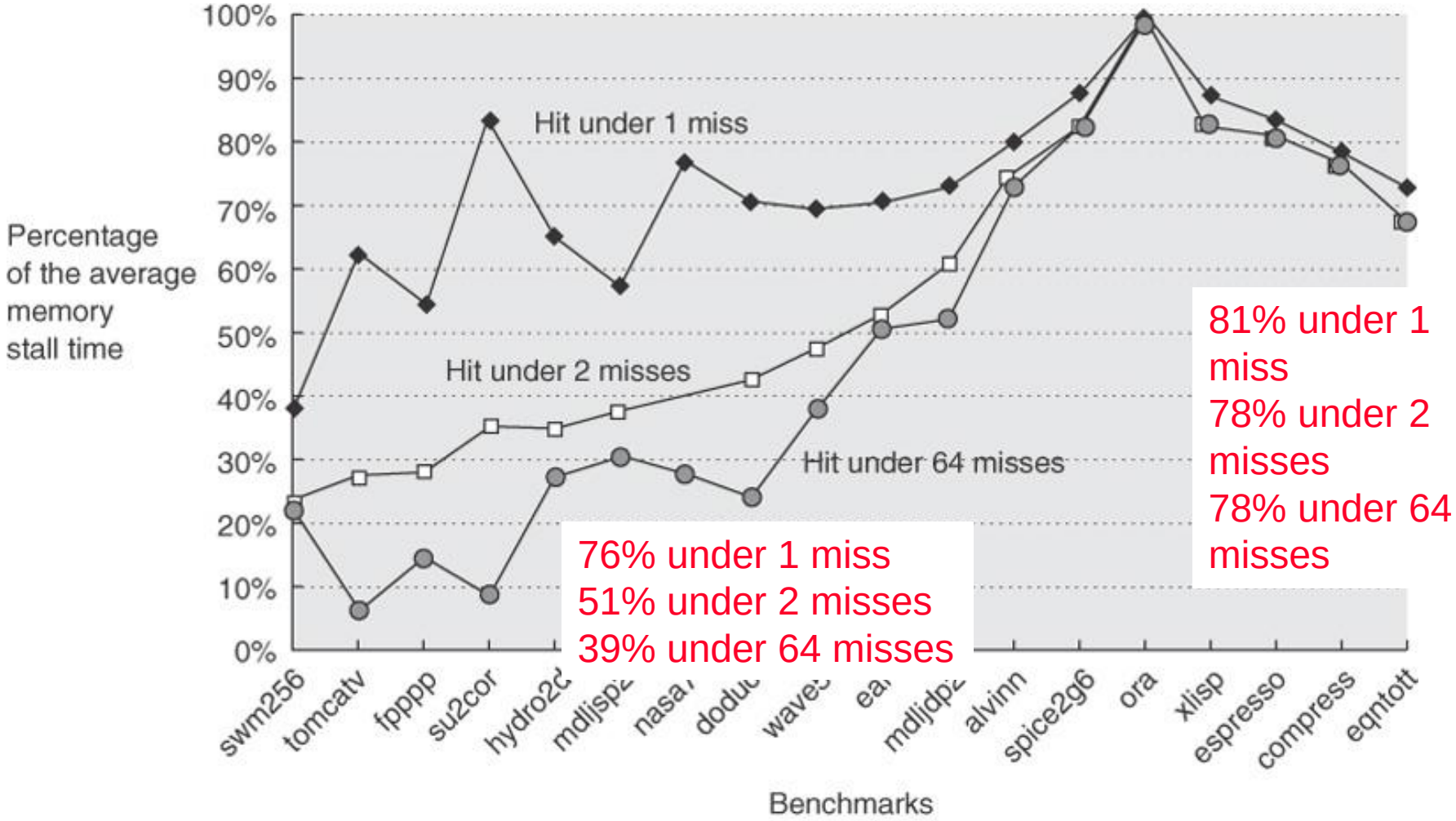
Lockup Free Caches (Cont.)

- MSHRs: miss status holding registers
  1. Is there already a miss to the same block?
  2. Route data back to CPU

  - Valid bit and tag: associatively compared on each miss
  - Status & pointer to block frame
  - What happens on a miss?
  - Tag L1 requests to allow out-of-order service from L2
  - Split-transaction/pipelined L1-L2 interface
  - Associative MSHRs could become bottleneck

Hit under miss gains

- Non-blocking vs blocking 8KB direct mapped D$, 32B blocks, 16 cycle L2



Cache Latency vs. Cache Bandwidth

- Latency can be handled by
  - hiding/tolerating techniques
    - e.g., parallelism
    - may increase bandwidth demand
  - reducing techniques

- Ultimately limited by physics

- Bandwidth can be handled by
  - banking/interleaving/multiporting
  - wider buses
  - hierarchies (multiple levels)

- What happens if average demand not supplied?
  - bursts are smoothed by queues
    - if burst is much larger than average => long queue
    - eventually increases delay to unacceptable levels

Increasing Issue → Increasing Bandwidth Needs

- Increasing issue width => wider caches

- Parallel cache access is harder than parallel FUs
  - fundamental difference: caches have state, FUs don't
  - one port affects future for other ports

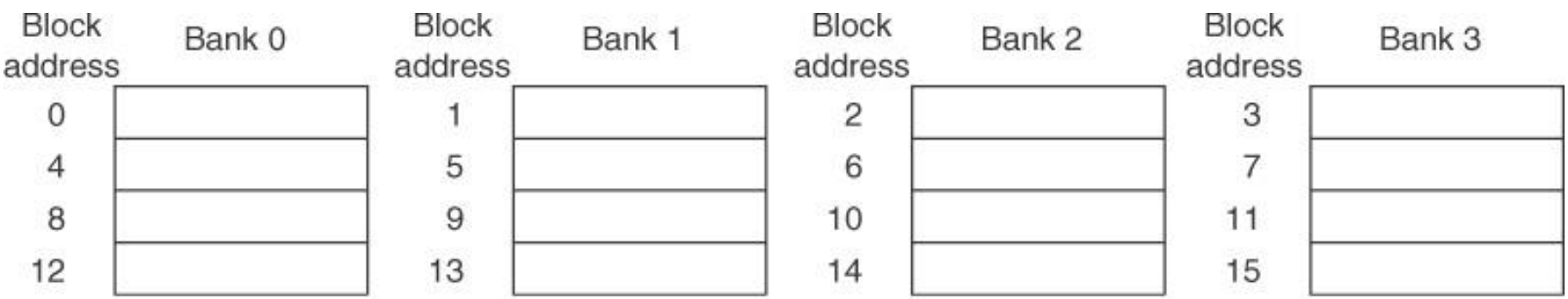
- Several approaches used
  - true multi-porting
  - multiple cache copies
  - virtual multi-porting
  - multi-banking (interleaving)
  - line buffers

Adv-6: Multibanked caches

- Divide the cache into multiple independent banks that can support simultaneous accesses
  - E.g.,T1 (“Niagara”) L2 has 4 banks
- Banking works best when accesses naturally spread themselves across banks ⇒ a simple mapping that works well is “sequential interleaving”
  - Spread block addresses sequentially across banks, e.g., for 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; …

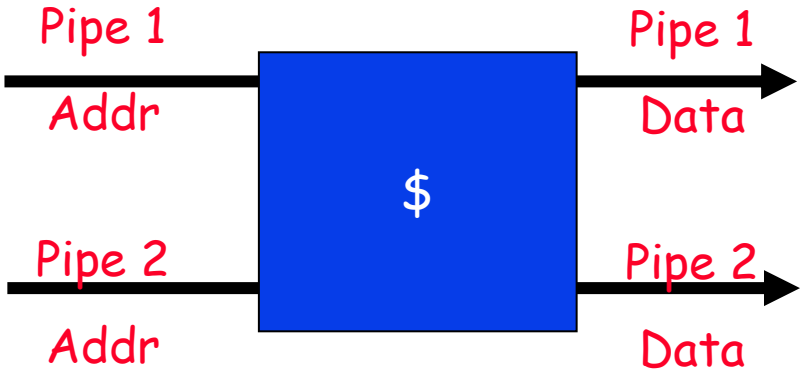
Block addr	Bank

2 bits



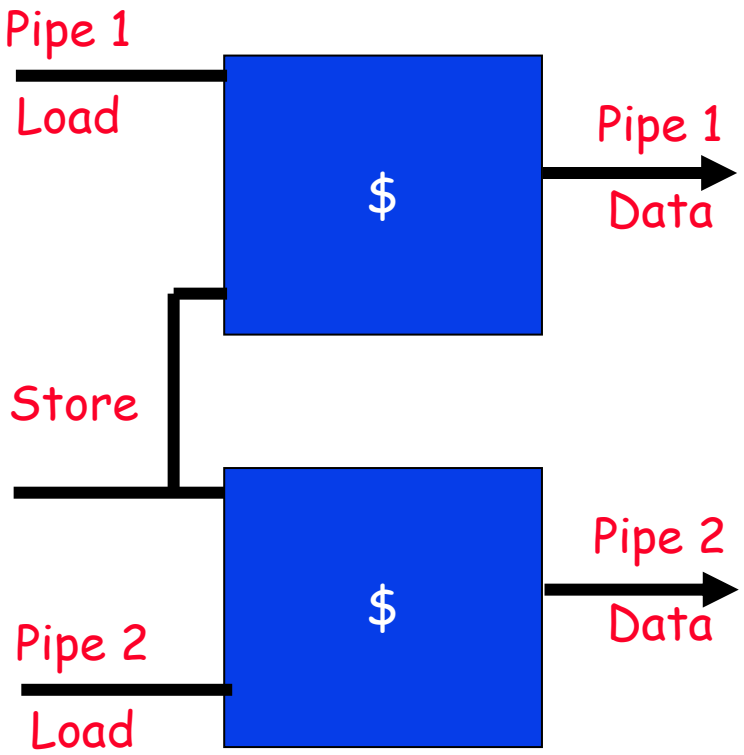
Multi-Port Caches: True Multiporting

- 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 multiported SRAMs are available, but
  - more chip area
  - slower access
    *(**very** undesirable for L1-D)*



Multiple Cache Copies

- Used in Alpha 21164
- Independent fast load paths
- Single shared store path

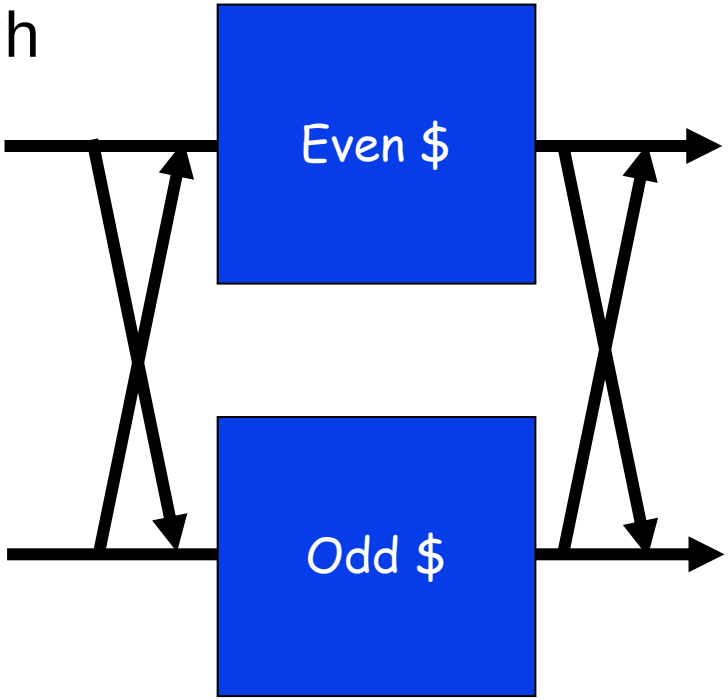


- Not a scalable solution
  - Store is a bottleneck
  - Doubles area

Multi-Banking (Interleaving) Caches

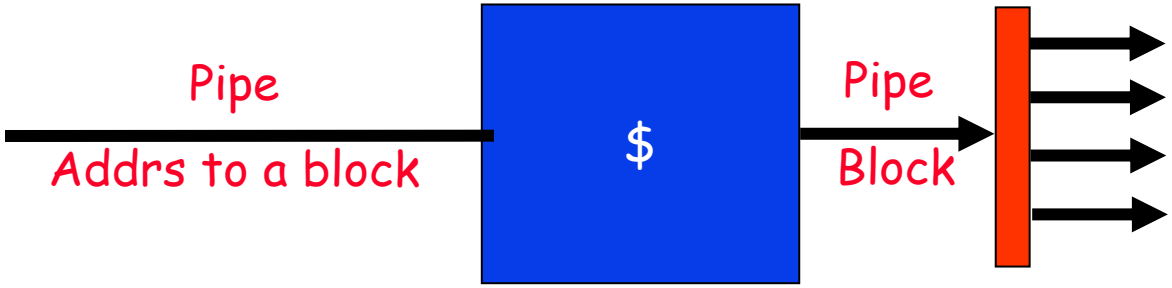
- Address space is statically partitioned and assigned to different caches
  Which addr bit to use for partitioning?

- 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 if compiler knows about the interleaving rules

Line Buffers (one form of "L0")



- Allows multiple ops to/from a block
- Increases bandwidth for wide issue

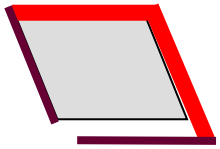
Multi-porting vs. Multibanking

- By itself, multi-banking underperforms because of *bank conflicts*
  - Many simultaneous requests to same bank
  - However:
    - Spatial locality – up to 75% of conflicts are to *same cache line*
    - Hence, add line buffer-like structure to allow all these to proceed
    - FIFOs buffers can smooth remaining bank access bursts

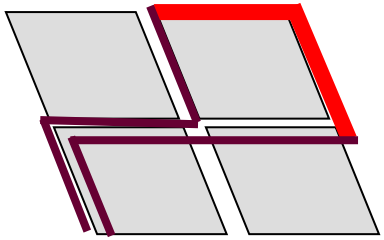
- Bottom line (Juan et al ICS 97):
  - 8 banks w/ FIFOs & same-line conflict optimization
    - ~5% slower than 8-port; about 1/6th the area
    - ~5% faster than 2-port; about 60% the area

Non-Uniform Cache Architecture

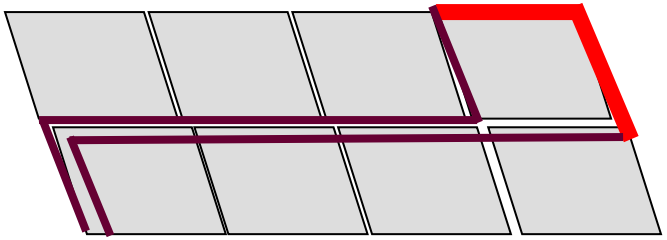
- ASPLOS 2002 proposed by UT-Austin (Kim, Burger, Keckler)
- Facts
  - Large shared on-die L2
  - Wire-delay dominating on-die cache



3 cycles
1MB
180nm, 1999

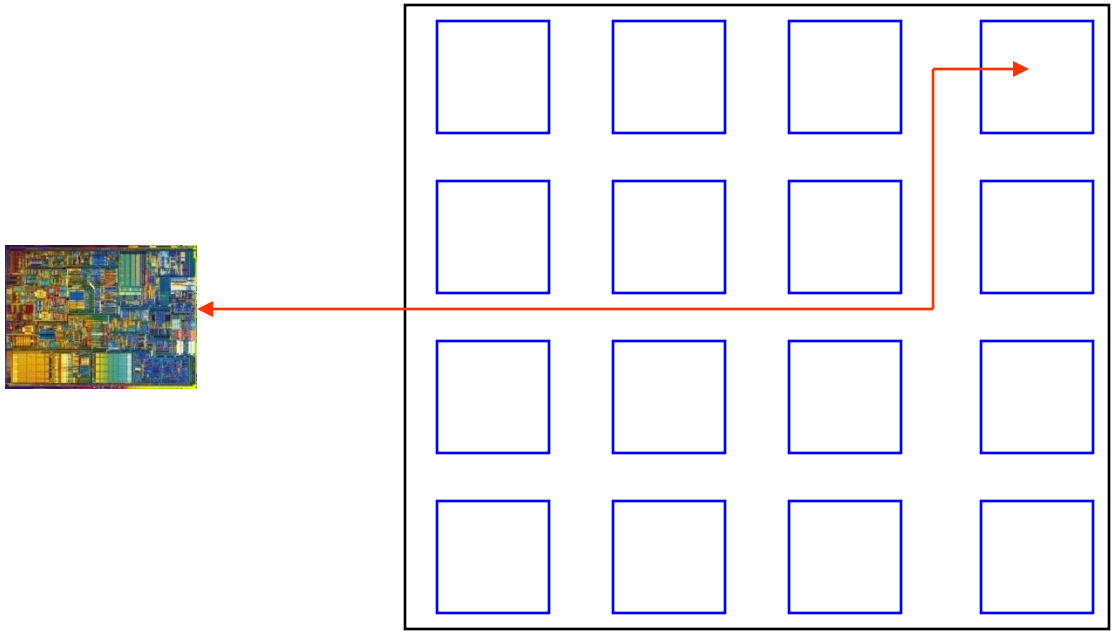


11 cycles
4MB
90nm, 2004



24 cycles
16MB
50nm, 2010

Multi-banked L2 cache

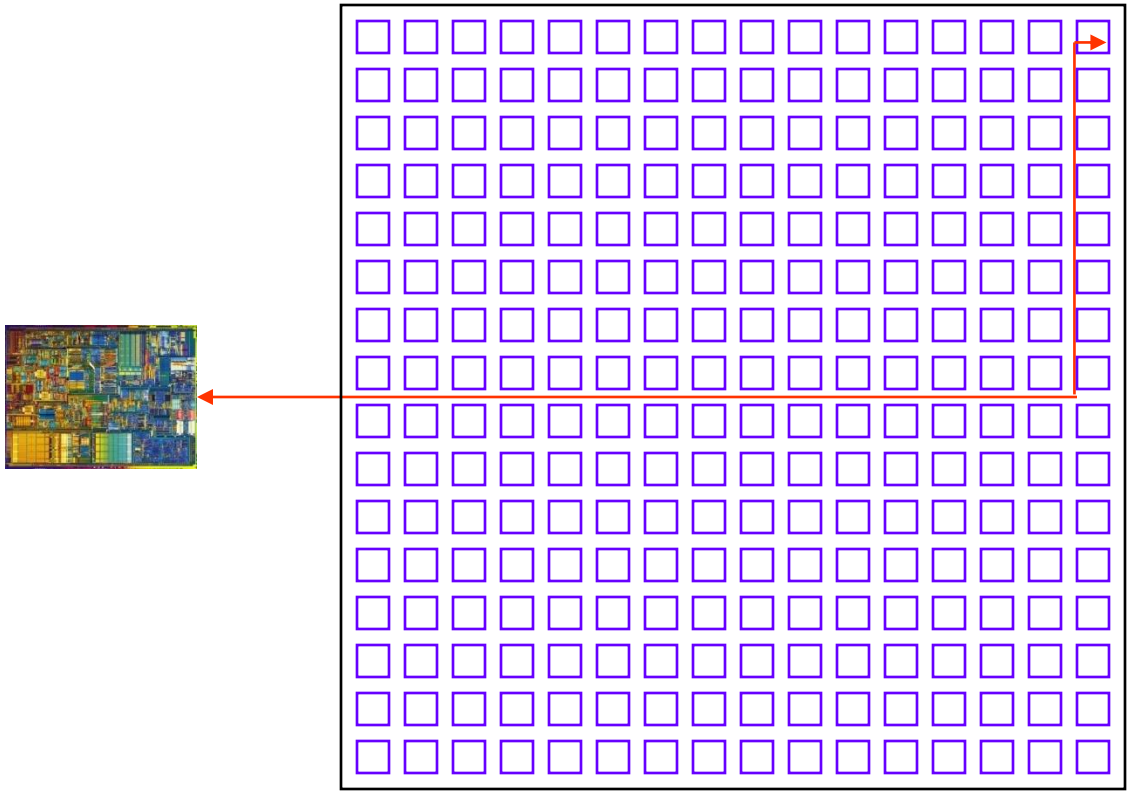


Bank=128KB
11 cycles

2MB @ 130nm

Bank Access time = 3 cycles
Interconnect delay = 8 cycles

Multi-banked L2 cache

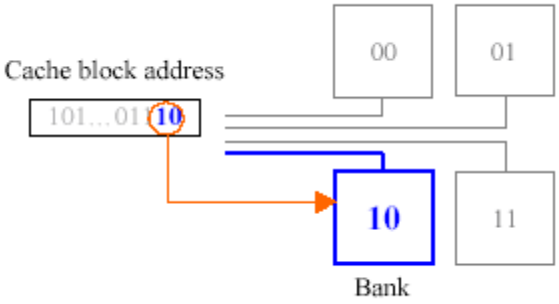


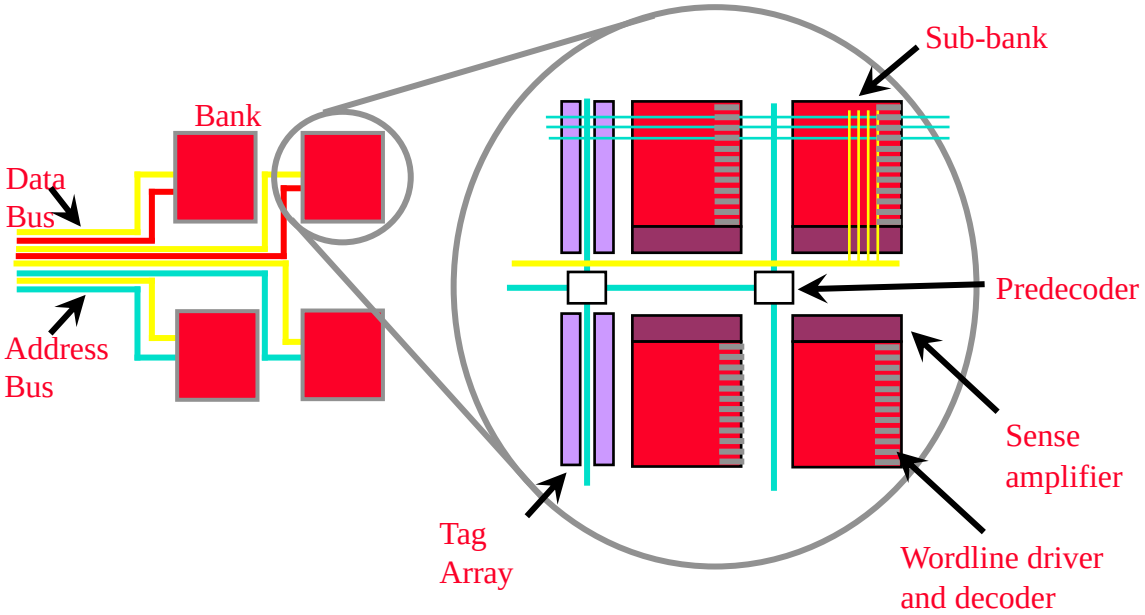
Bank=64KB
47 cycles

16MB @ 50nm

Bank Access time = 3 cycles
Interconnect delay = 44 cycles

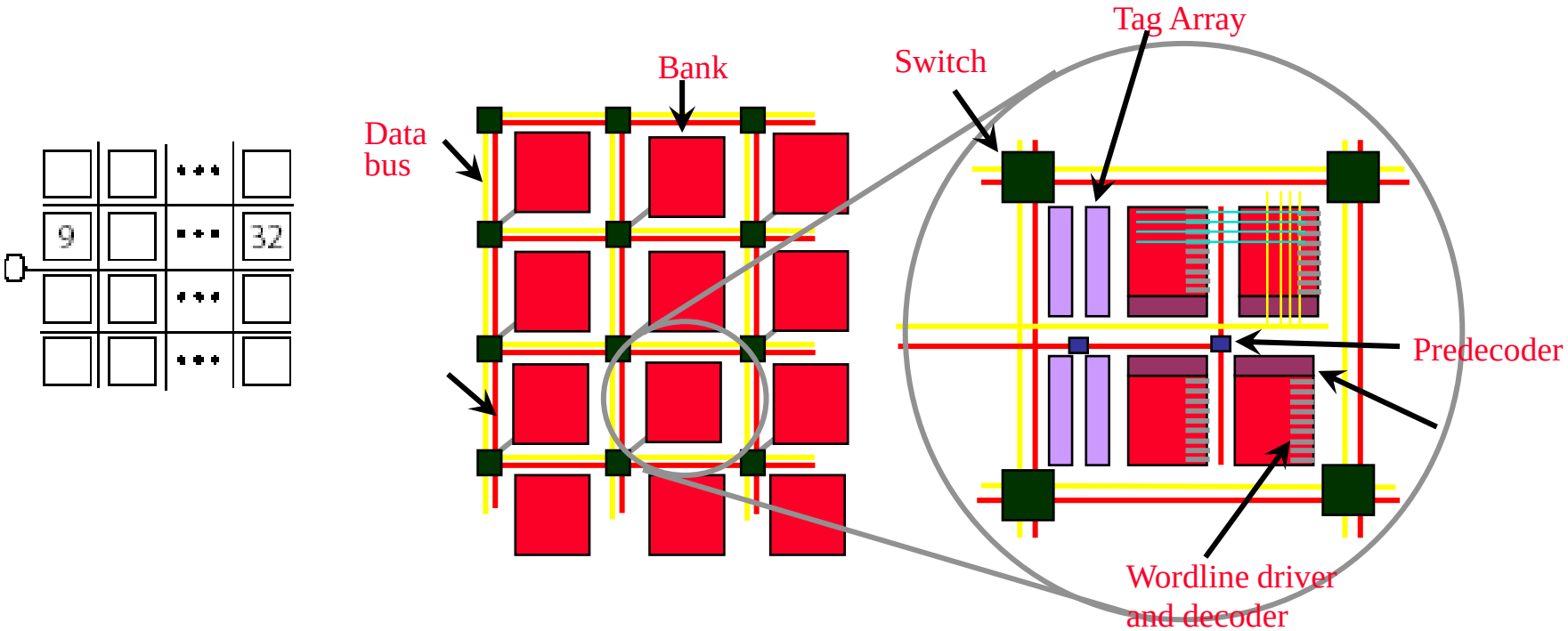
Static NUCA-1





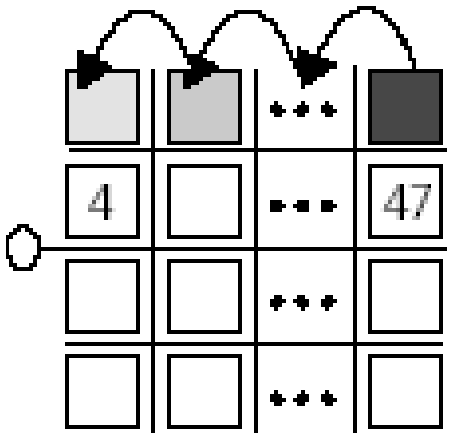
- Use private per-bank channel
- Each bank has its distinct access latency
- Statically decide data location for its given address
- Average access latency =34.2 cycles
- Wire overhead = 20.9% → an issue

Static NUCA-2



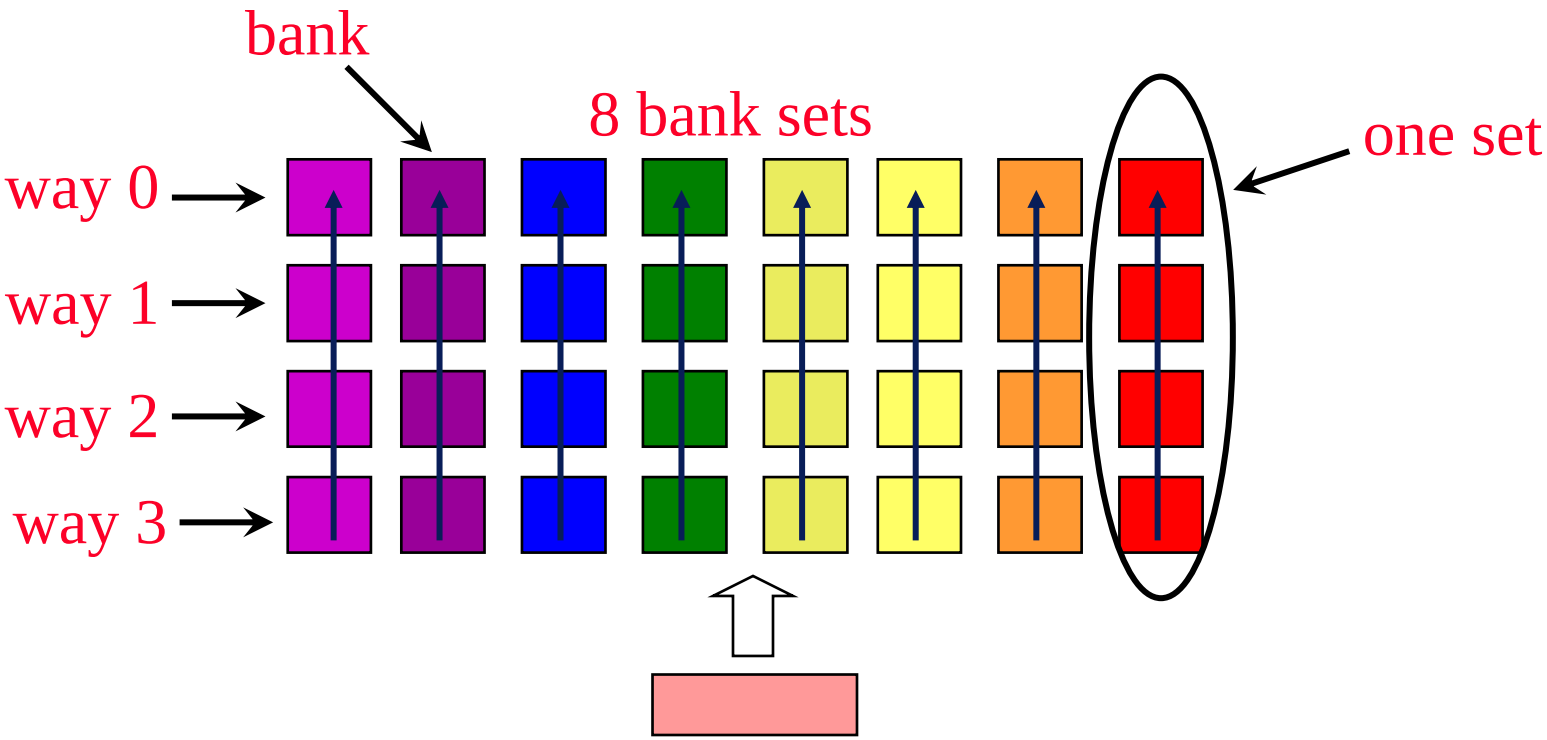
- Use a 2D switched network to alleviate wire area overhead
- Average access latency =24.2 cycles
- Wire overhead = 5.9%

Dynamic NUCA



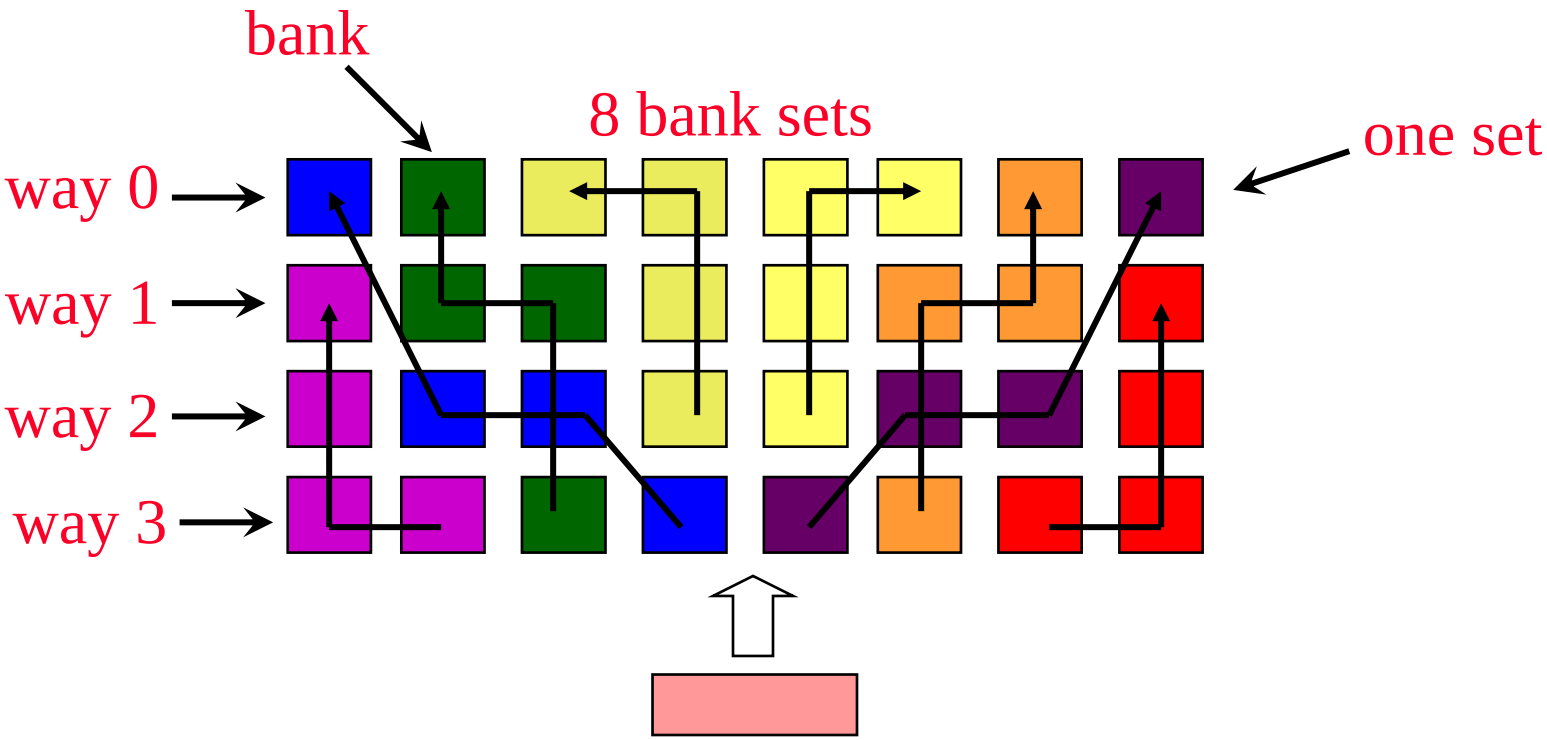
- Data can dynamically migrate
- Move frequently used cache lines closer to CPU

Dynamic NUCA



- **Simple Mapping**
- All 4 ways of each bank set needs to be searched
- Farther bank sets → longer access

Dynamic NUCA



- **Fair Mapping**
- Average access time across all bank sets are equal

Adv-7: Critical word first & early restart

- Don't wait for full block to load before sending the requested word to the processor

1. **Early restart** – Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the processor so it can continue
   - Spatial locality ⇒ tend to want next sequential word, so not clear size of benefit of just early restart
2. **Critical Word First** – Request the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block
   - Big blocks more popular today ⇒ so critical word 1st widely used

Large Blocks and Subblocking

- Can we go further than *critical word first?*
  - Can make bets as to order/need for different non-critical words within a block

- 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

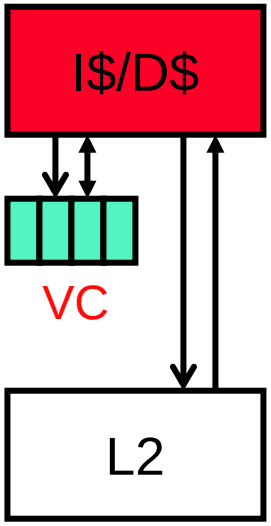
Adv-8: Write merging in the WBB

- WBB (which we have already seen) – allows the processor to continue while the write back (or write through) to memory is processed
  - **Write merging** – when new writes backs/throughs match an existing entry in the WBB, the two writes can be combined
    - Allows 4 sequential words to fit in one WBB entry (assuming a 4 word cache block and 4 word WBB buffer entry)
    - Multiword writes to memory are usually faster than writes performed one word at a time
- WBB is sometimes called a victim buffer (not to be confused with Jouppi's victim cache which holds block discarded from the cache on a miss whether dirty or not)
  - WBB or victim buffer is to allow the cache to proceed without waiting for dirty blocks to write to memory
  - Victim cache is to reduce the impact of conflict misses

Adv-9: Victim cache

- Conflict misses are due to not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (XYZ)+

- **Victim cache (VC)**: 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 VC
  - On miss, check VC: 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 for small/low-assoc $



Adv-9.5 Compiler optimizations

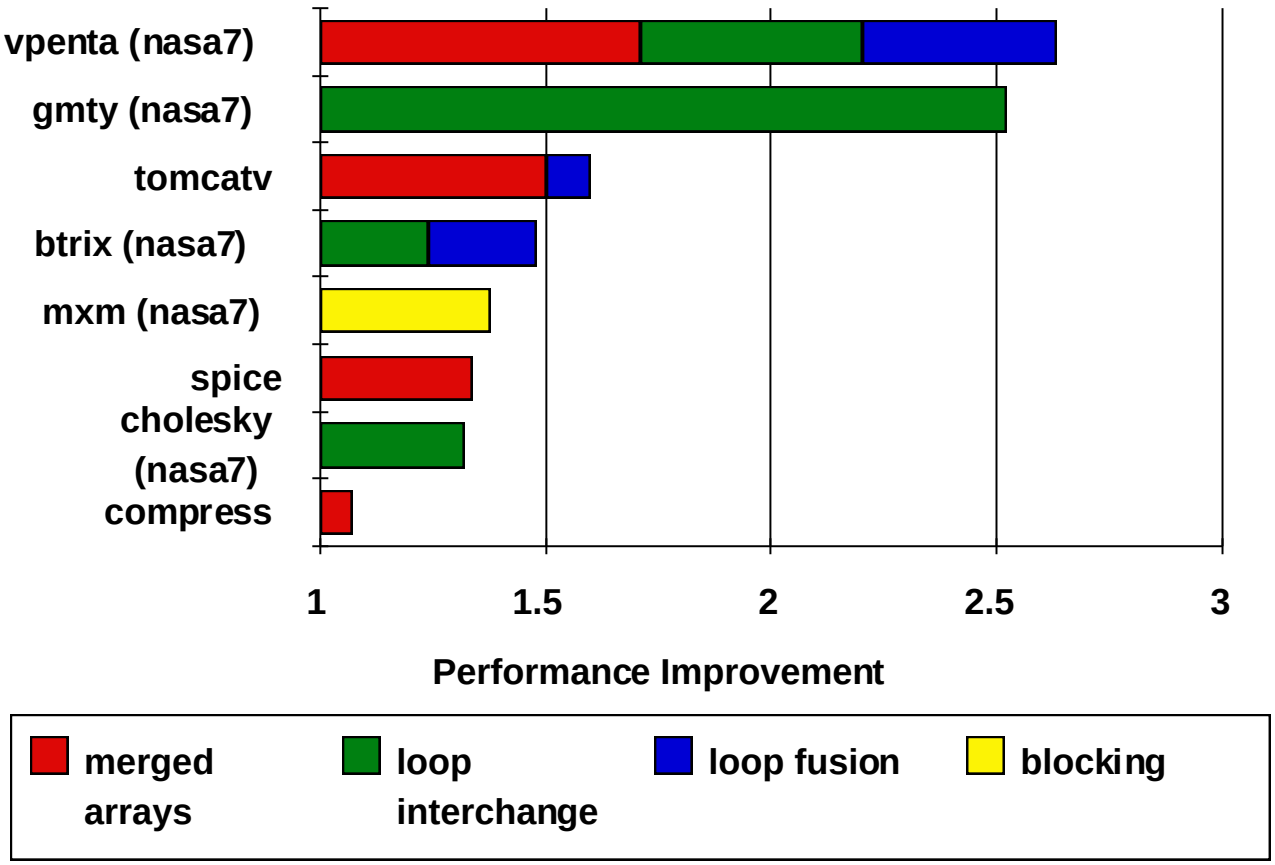
- Conflict (and capacity) misses often due to poor spatial or temporal locality
- Instructions
  - Compiler must know that restructuring preserves semantics
  - Reorder procedure to reduce conflict misses
  - Align basic blocks so that the entry point is the first word of a cache block
  - Branch straightening – if branch is likely to be taken, change the sense of the branch and swap the branch target basic block
- Data has even fewer restrictions on reordering than code
  - The next slides contains a few sample data reordering optimizations

Aside: Compiler optimizations (for data)

- **Loop interchange**: spatial locality
  - Exchanging the nesting of loops to maximize the use of data in a cache block before it is replaced
- **(Array) blocking**: temporal locality
  - Operate on submatrices (of large arrays) repeatedly vs going down a whole array column or row to maximize access of data in cache before it is replaced
- **Merged arrays**: spatial locality
  - Use a single array of compound elements vs two arrays
- **Loop fusion**: spatial locality
  - Combine two independent loops that have the same looping and some variable that overlap

- For more insights/details and more optimizations take CSE 521

Compiler optimizations performance impacts

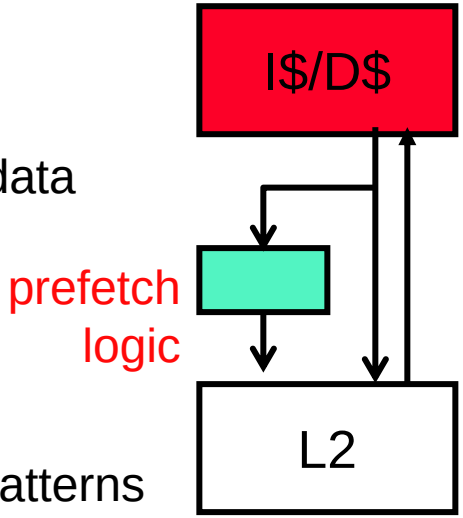


Prefetching

- **Prefetching**: put blocks in cache proactively/ speculatively
  - Key: anticipate upcoming miss addresses accurately
    - Can do in software or hardware
  - Relies on having unused memory bandwidth
- Simple example: **next block prefetching**
  - Miss on address **X** → anticipate miss on **X+block-size**
  - \+ Works for instr's: sequential execution
  - \+ Works for data: arrays
- **Timeliness**: initiate prefetches sufficiently in advance
- **Coverage**: prefetch for as many misses as possible
- **Accuracy**: don't pollute cache with unnecessary data
  - It evicts useful data

Adv-10: Hardware prefetching

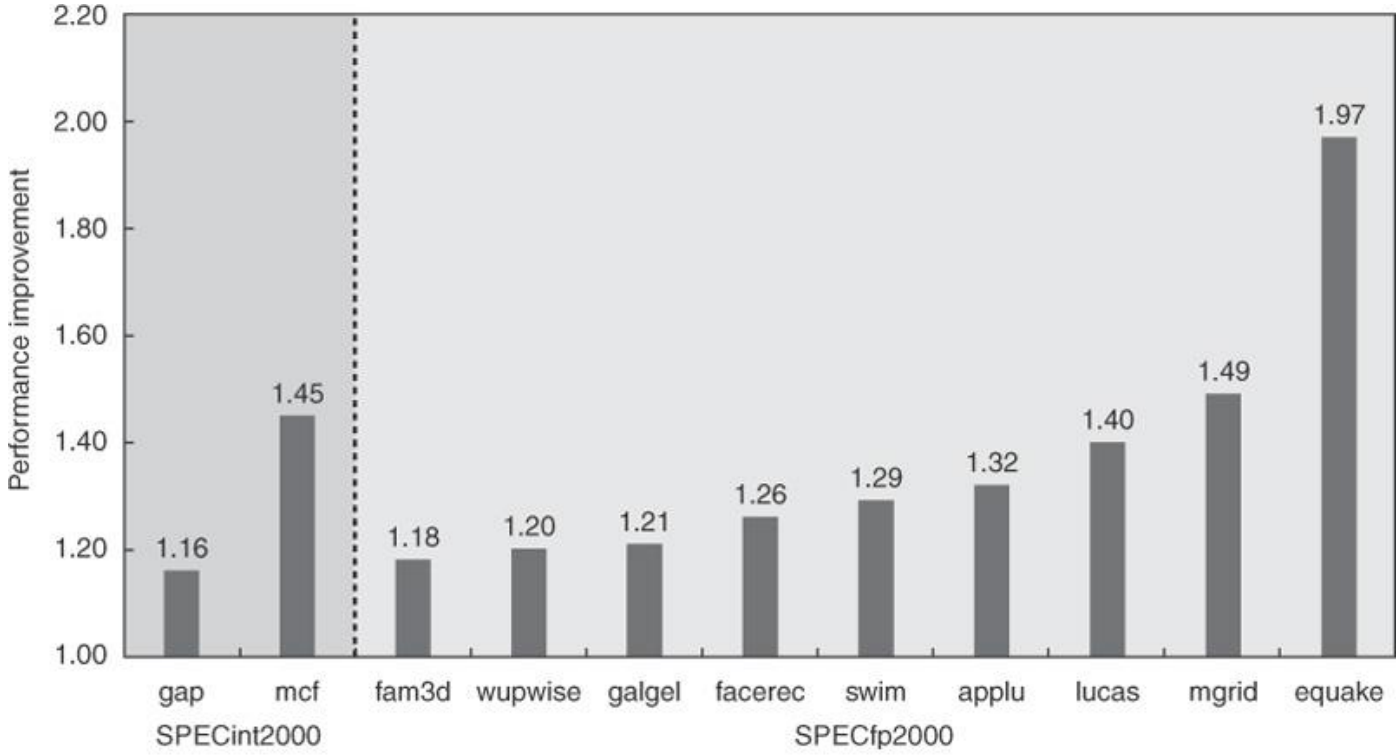
- What to prefetch?
  - **Stride-based sequential prefetching** – e.g., similar to Jouppi's stream buffers
    - Can also do N blocks ahead to hide more latency
    - \+ Simple, works for sequential things: instr's, array data
    - \+ Works better than doubling the block size
  - **Address-prediction**
    - Needed for non-sequential data: lists, trees, etc.
    - Use a hardware table to detect strides, common patterns



- When/where to prefetch?
  - On every reference?
  - On every miss?
  - Into L1 (or L2) or into a prefetch stream buffer feeding L1 (or L2), or stream buffers feeding both L1 and L2?

Hardware prefetching performance

- Pentium 4 prefetches into its L2 from up to 8 streams from 8 different 4KB pages
  - Invoked if there are two successive L2 cache misses to a page and if the distance between those blocks is < 256B
  - Speedup on omitted benchmarks less than 15%



Adv-11: Software prefetching

- Use a special "prefetch" instruction (so part of the ISA, but semantically invisible to the program)
  - Register prefetch (HP PA-RISC) or cache prefetch which loads data only into the cache
  - Usually nonfaulting (nonbinding) so prefetches turn into no-ops if they would normally result in an exception
- Inserted by programmer or compiler, e.g.,

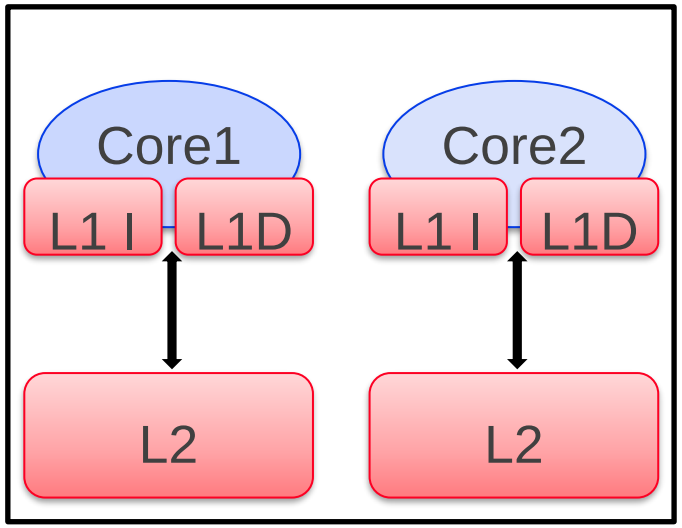
```
for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j+=BLOCK_SIZE) {
      __builtin_prefetch(&x[i][j]+BLOCK_SIZE);
      for (jj=j; jj<j+BLOCK_SIZE-1; jj++)
         sum += x[i][jj];
   }
```

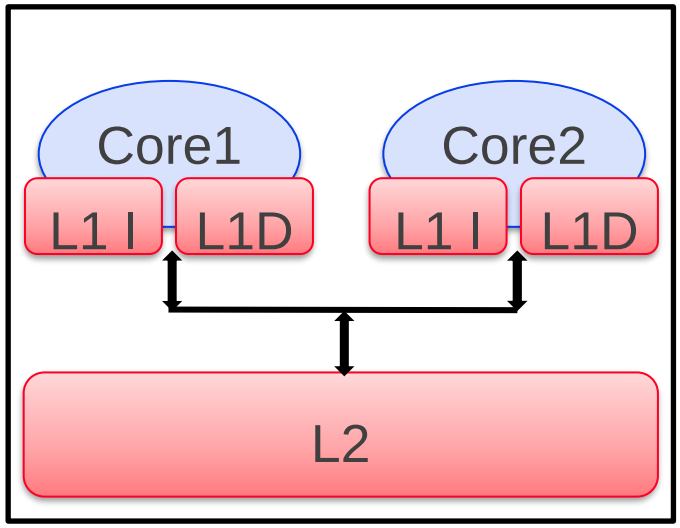
- Multiple prefetches bring in multiple blocks in parallel
  - Need lockup-free (nonblocking) caches
  - Need sufficient memory-level parallelism

Aside: 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

Multicore cache alternatives





- AMD's AthlonX2, IBM's POWER6
  - Good: Fast interconnect; No L2 app thread contention
    - Good for multiprogrammed workloads
  - Bad: App threads can't share L2 capacity … or data

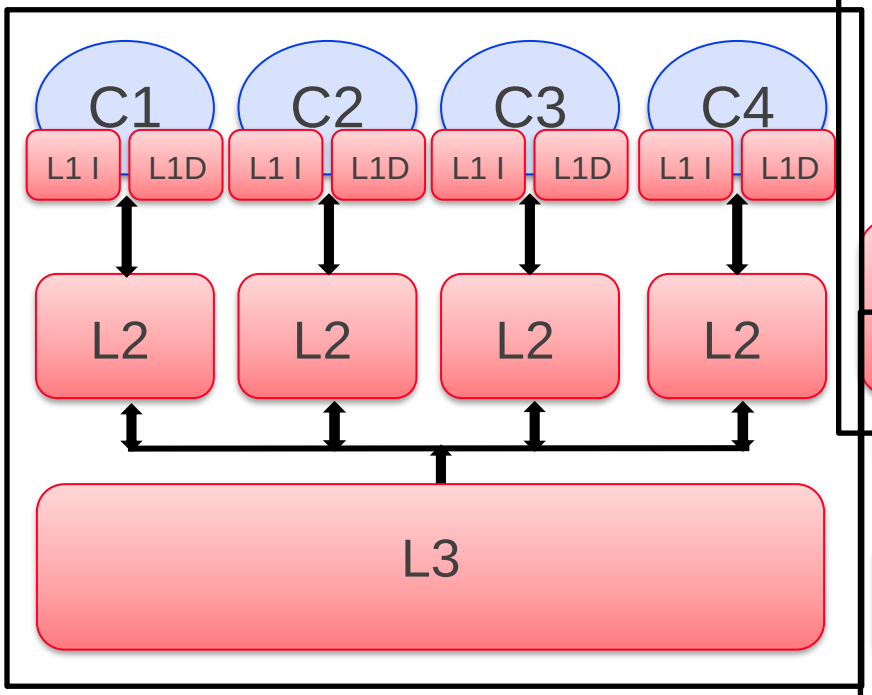
- Intel's CoreDuo
  - Good: App threads can share L2 data … and capacity
    - Good for multithreaded apps
  - Bad: Slower interconnect; L2 app thread contention

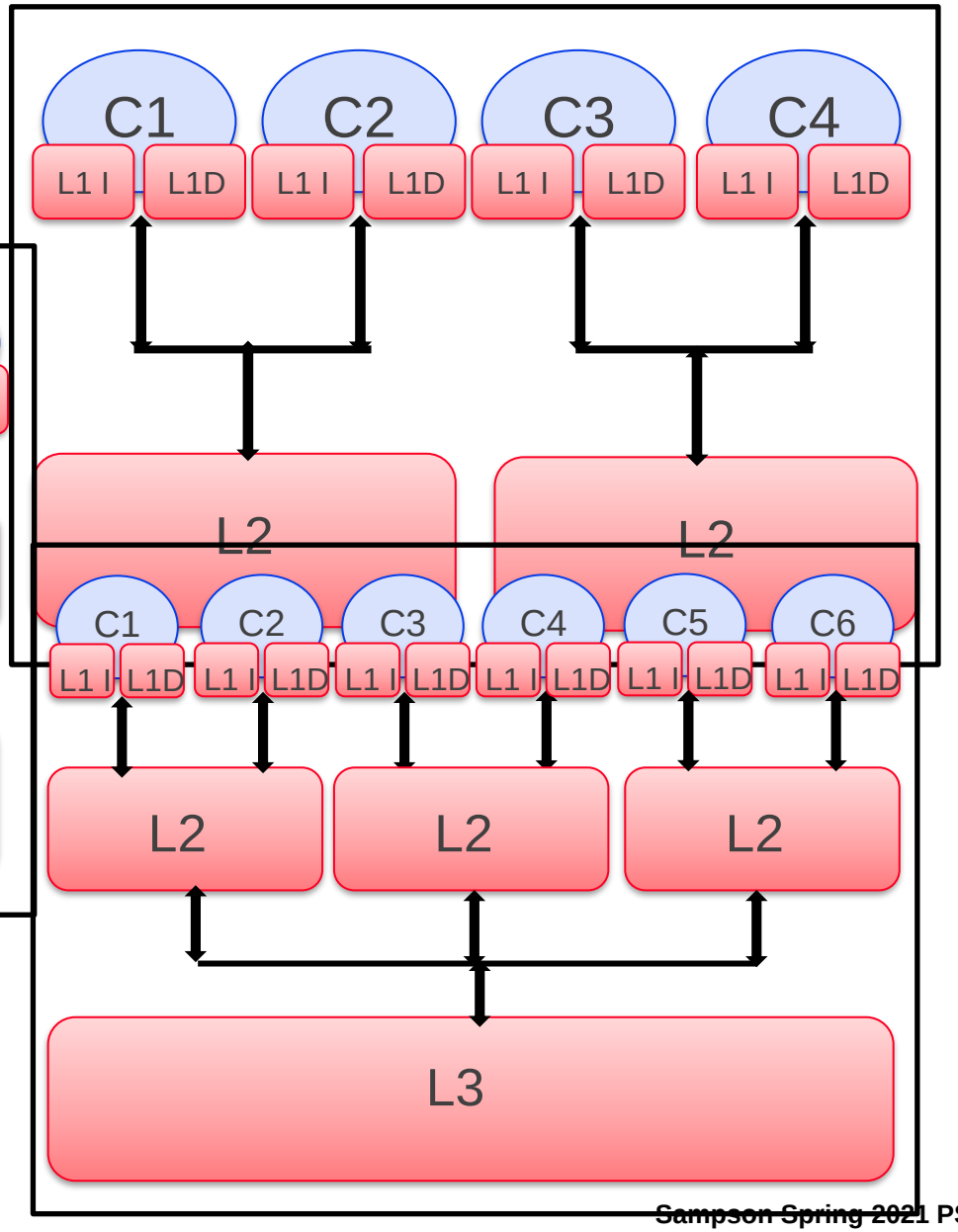
Classifying multicore misses : The CII Model

- Divide multicore cache misses into four categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite caches**
  - **Intracore**: miss caused because two application threads running on the same core replace each other's blocks
  - **Intercore**: miss caused because two application threads running on different cores replace each other's blocks in a shared cache level
  - **(Coherence)**: miss due to external invalidations
    - In multicores (more later)

Yet more multicore cache alternatives

- Intel's Harpertown
- Intel's Nehalem





- Intel's Dunnington