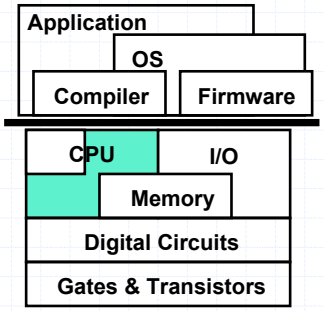
CIS 501
Introduction to Computer Architecture

Unit 3: Storage Hierarchy I: Caches

This Unit: Caches



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

Motivation

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

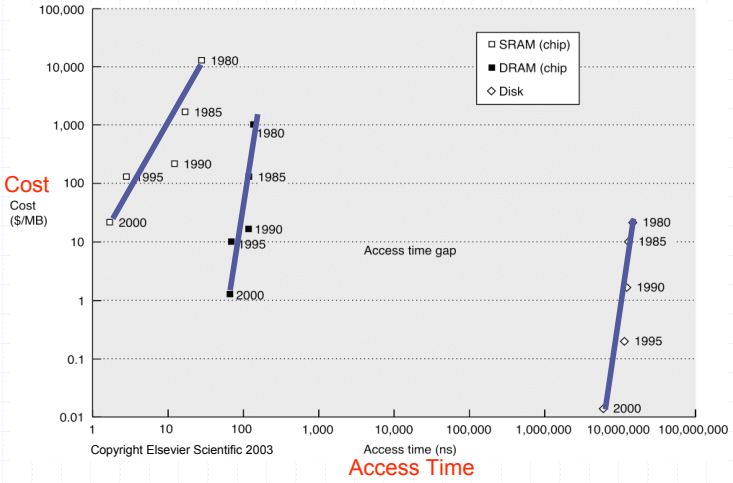
Types of Memory

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

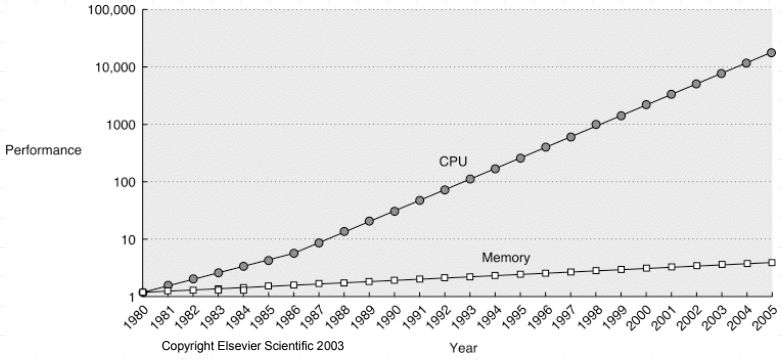
Storage Technology

- Cost - what can $300 buy today?
  - SRAM - 4MB
  - DRAM - 1,000MB (1GB) --- 250x cheaper than SRAM
  - Disk - 400,000MB (400GB) --- 400x cheaper than DRAM
- Latency
  - SRAM - <1 to 5ns (on chip)
  - DRAM - ~100ns --- 100x or more slower
  - Disk - 10,000,000ns or 10ms --- 100,000x slower (mechanical)
- Bandwidth
  - SRAM - 10-100GB/sec
  - DRAM - ~1GB/sec
  - Disk - 100MB/sec (0.1 GB/sec) - sequential access only
- Aside: Flash, a non-traditional (and nonvolatile) memory
  - 4,000MB (4GB) for $300, cheaper than DRAM!

Storage Technology Trends



The "Memory Wall"



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

Locality to the Rescue

- Locality of memory references
  - Property of real programs, few exceptions
  - Books and library analogy
- Temporal locality
  - Recently referenced data is likely to be referenced again soon
  - **Reactive**: cache recently used data in small, fast memory
- Spatial locality
  - More likely to reference data near recently referenced data
  - **Proactive**: fetch data in large chunks to include nearby data
- Holds for data and instructions

Known From the Beginning

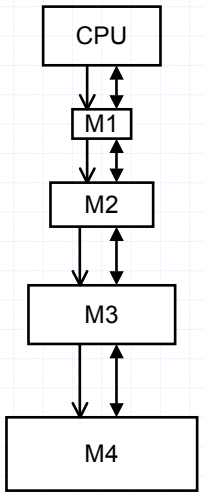
"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, VonNeumann

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

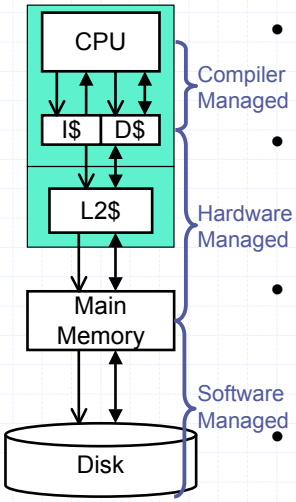
IAS memo 1946

Exploiting Locality: Memory Hierarchy



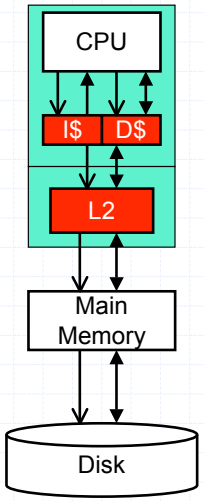
- Hierarchy of memory components
  - Upper components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by buses
  - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - Move data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

Concrete Memory Hierarchy



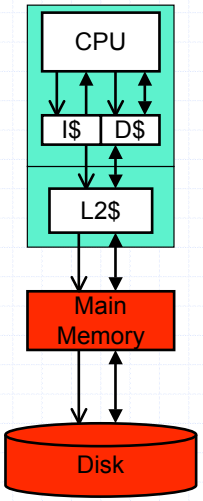
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8-64KB each
- 2nd level: **Second-level cache** (L2$)
  - On-chip, certainly on-package (with CPU)
  - Made of SRAM (same circuit type as CPU)
  - Typically 512KB to 16MB
- 3rd level: **main memory**
  - Made of DRAM
  - Typically 512MB to 2GB for PCs
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Made of magnetic iron oxide disks

This Unit: Caches



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

Looking forward: Memory and Disk



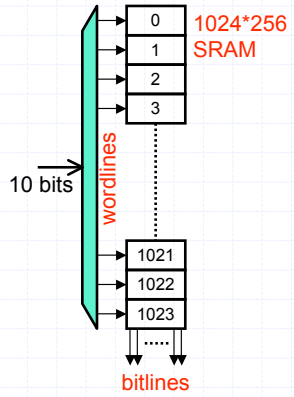
- Main memory
  - Virtual memory (guest lecture on Tuesday)
  - DRAM-based memory systems
- Disks and Storage
  - Properties of disks
  - Disk arrays (for performance and reliability)

Readings

- H+P
  - Chapter 5.1–5.7
- Paper: week from Thursday
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers"

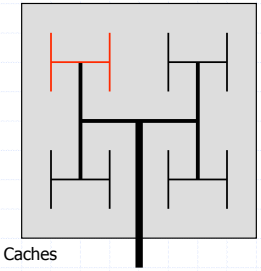
Basic Memory Array Structure

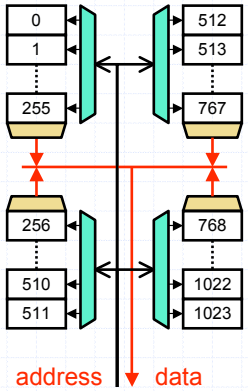
- Number of entries
  - 2^n, where n is number of address bits
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
  - One-bit address travels on "wordlines"
- Size of entries
  - Width of data accessed
  - Data travels on "bitlines"
  - 256 bits (32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



Physical Cache Layout

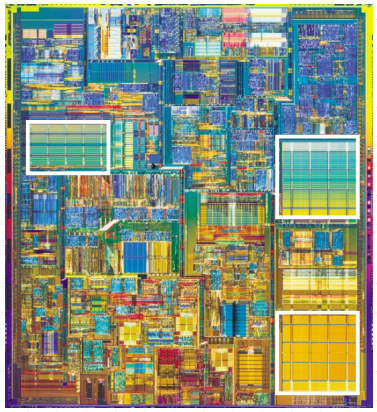
- Logical layout
  - Arrays are vertically contiguous
- Physical layout - roughly square
  - Vertical partitioning to minimize wire lengths
  - **H-tree**: horizontal/vertical partitioning layout
    - Applied recursively
    - Each node looks like an H





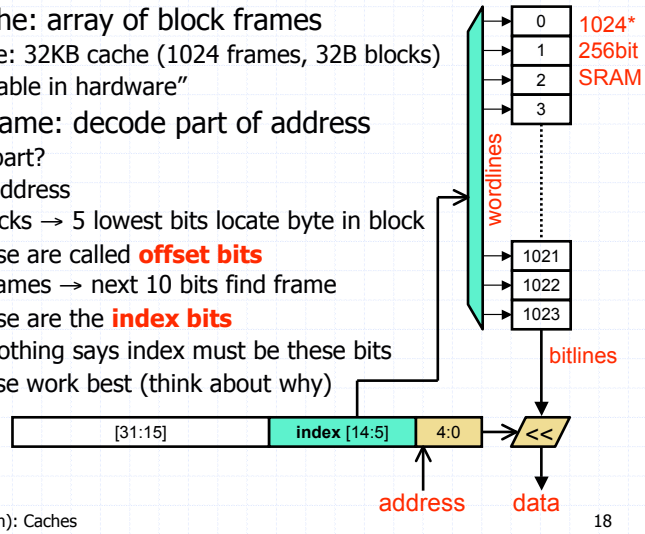
Physical Cache Layout

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



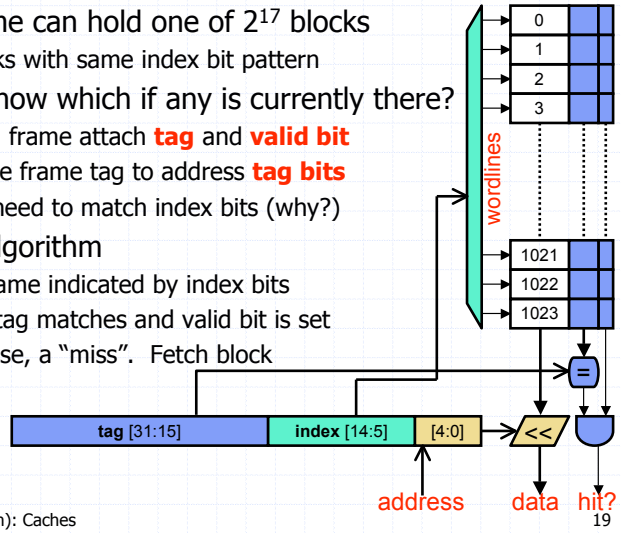
Basic Cache Structure

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



Basic Cache Structure

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



Calculating Tag Overhead

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

Cache Performance Simulation

- Parameters: 8-bit addresses, 32B cache, 4B blocks
  - Nibble notation (base 4) 

  - Initial contents: 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130

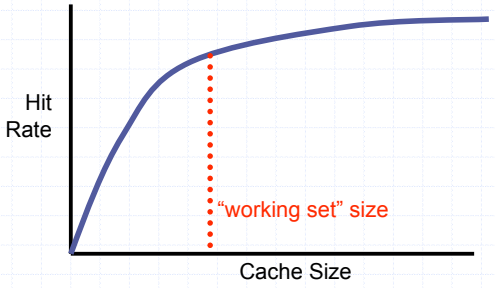
Cache contents (prior to access)	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	Miss
0000, 0010, **3020**, 0030, 0100, 0110, 0120, 0130	3030	Miss
0000, 0010, 3020, **3030**, 0100, 0110, 0120, 0130	2100	Miss
0000, 0010, 3020, 3030, **2100**, 0110, 0120, 0130	0012	Hit
0000, **0010**,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

Miss Rate: ABC

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

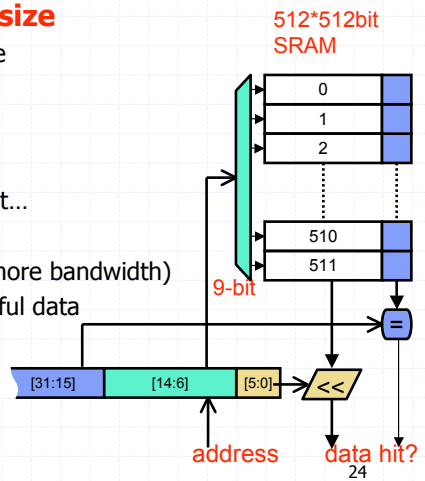
Increase Cache Size

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



Block Size

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



Block Size and Tag Overhead

- Tag overhead of 32KB cache with 1024 32B frames
  - 32B frames → 5-bit offset
  - 1024 frames → 10-bit index
  - 32-bit address – 5-bit offset – 10-bit index = 17-bit tag
  - (17-bit tag + 1-bit valid) * 1024 frames = 18Kb tags = 2.2KB tags
  - ~6% overhead
- Tag overhead of 32KB cache with 512 64B frames
  - 64B frames → 6-bit offset
  - 512 frames → 9-bit index
  - 32-bit address – 6-bit offset – 9-bit index = 17-bit tag
  - (17-bit tag + 1-bit valid) * 512 frames = 9Kb tags = 1.1KB tags
  - \+ ~3% overhead

Block Size and Performance

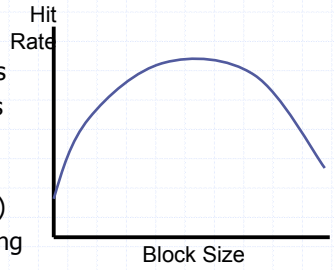
- Parameters: 8-bit addresses, 32B cache, **8B blocks**
  - Initial contents : 0000(0010), 0020(0030), 0100(0110), 0120(0130)

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

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

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

Conflicts

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

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

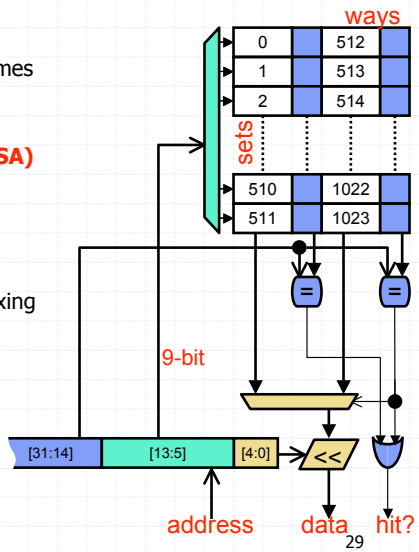
Cache contents (prior to access)	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	Miss
0000, 0010, 3020, 0030, 0100, 0110, 0120, 0130	**3030**	Miss
0000, 0010, 3020, **3030**, 0100, 0110, 0120, 0130	2100	Miss
0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130	0012	Hit
0000, 0010, 3020,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

Set-Associativity

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

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

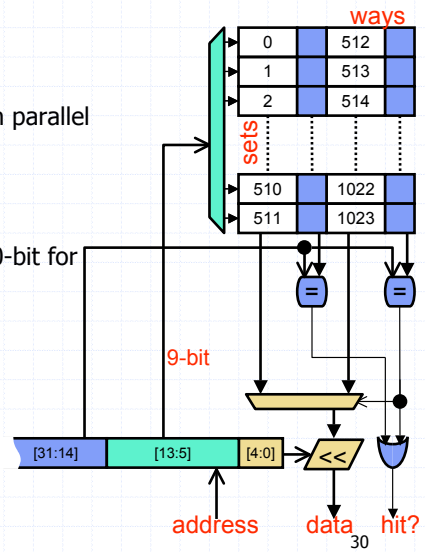
  - Note: valid bit not shown



Set-Associativity

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

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



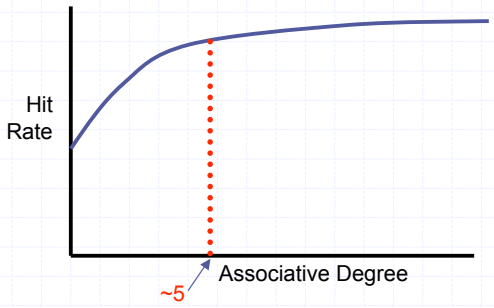
Associativity and Performance

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

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

Increase Associativity

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

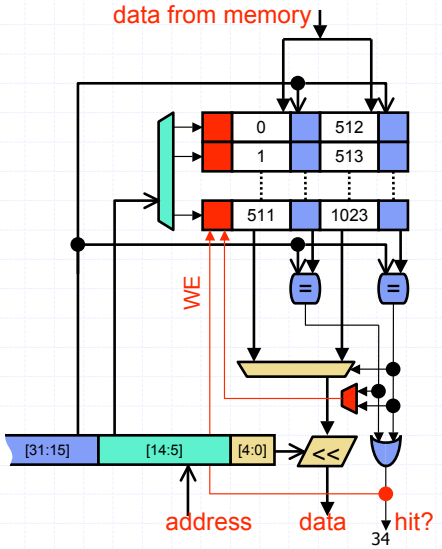


Replacement Policies

- 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?

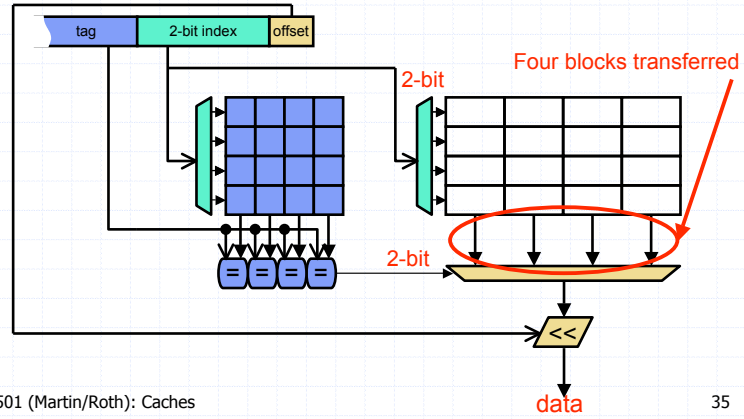
NMRU and Miss Handling

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



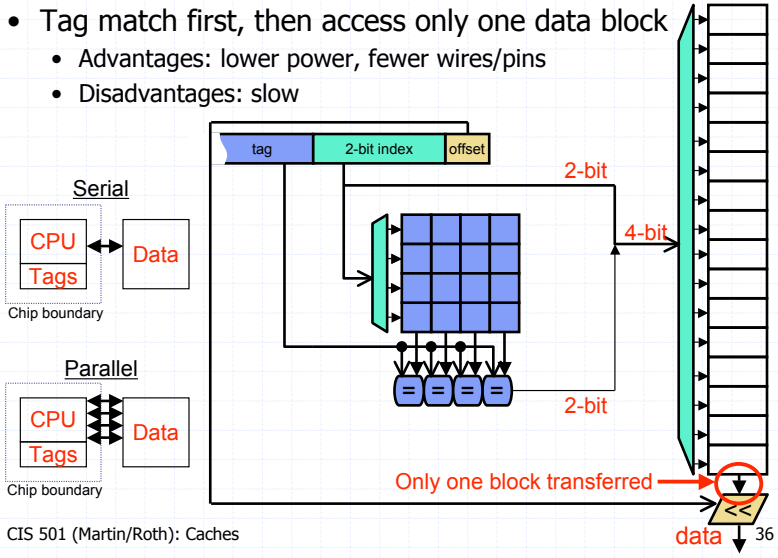
Parallel or Serial Tag Access?

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



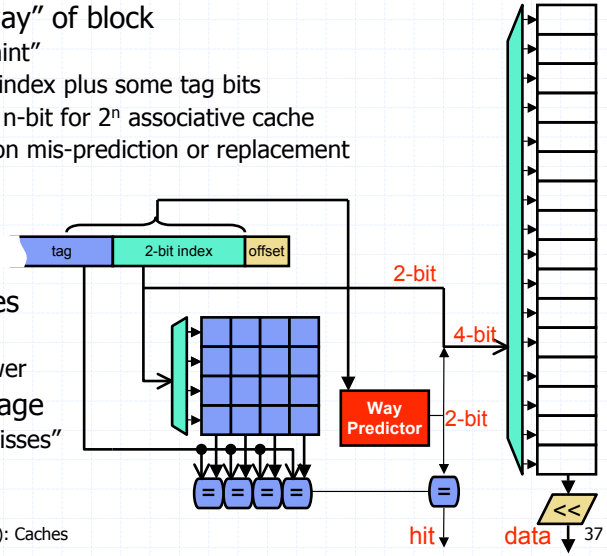
Serial Tag Access

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



Best of Both? Way Prediction

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



Classifying Misses: 3(4)C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
    - Identify? easy
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? 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
- Who cares? Different techniques for attacking different misses

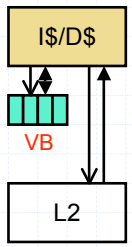
Cache Performance Simulation

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

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

Conflict Misses: Victim Buffer

- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (ABC)*
- **Victim buffer (VB)**: small fully-associative cache
  - Sits on I$/D$ fill 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 **$latency_{miss}$**?



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
```

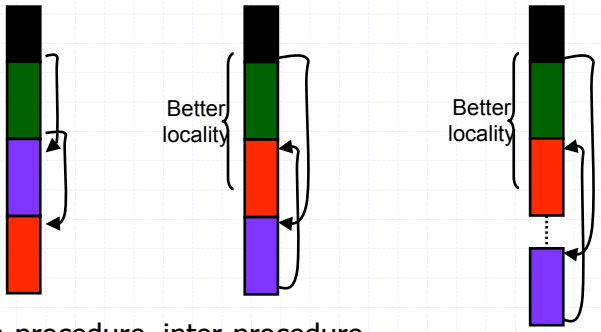
  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

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



- Intra-procedure, inter-procedure

Miss Cost: Critical Word First/Early Restart

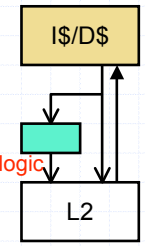
- Observation: $latency_{miss} = latency_{access} + latency_{transfer}$
  - $latency_{access}$: time to get first word
  - $latency_{transfer}$: time to get rest of block
  - Implies whole block is loaded before data returns to CPU
- Optimization
  - **Critical word first**: return requested word first
    - Must arrange for this to happen (bus, memory must cooperate)
  - **Early restart**: send requested word to CPU immediately
    - Get rest of block load into cache in parallel
  - $latency_{miss} = latency_{access}$

Miss Cost: Lockup Free Cache

- **Lockup free**: allows other accesses while miss is pending
  - Consider: Load [r1] -> r2; Load [r3] -> r4; Add r2, r4 -> r5
  - Only makes sense for…
    - Data cache
    - 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 split-transaction bus
    - Requires multiple MSHRs: search to avoid frame conflicts

Prefetching

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



Software Prefetching

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

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


Hardware Prefetching

- What to prefetch?
  - One block ahead
    - Can also do N blocks ahead to hide more latency
    + Simple, works for sequential things: insns, array data
  - **Address-prediction**
    - Needed for non-sequential data: lists, trees, etc.
- When to prefetch?
  - On every reference?
  - On every miss?
    + Works better than doubling the block size
  - Ideally: when resident block becomes dead (avoid useful evictions)
    – How to know when that is? ["Dead-Block Prediction", ISCA'01]

Address Prediction for Prefetching

- "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
  - Can do in hardware too
- An active area of research

Increasing Cache Bandwidth

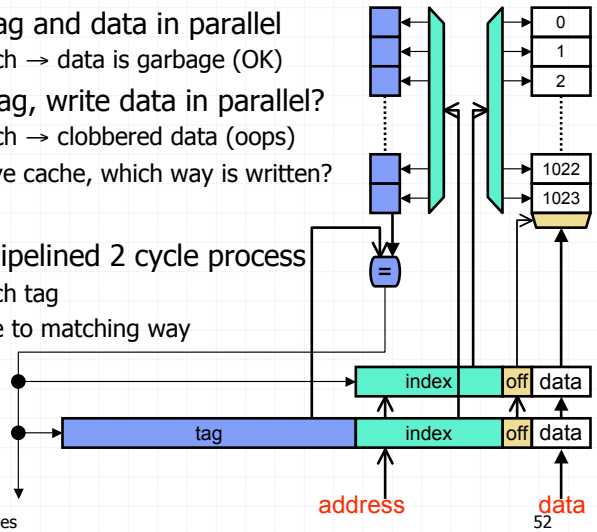
- What if we want to access the cache twice per cycle?
- Option #1: multi-ported SRAM
  - Same number of six-transistor cells
  - Double the decoder logic, bitlines, wordlines
  - Areas becomes "wire dominated" -> slow
- Option #2: banked cache
  - Split cache into two smaller "banks"
  - Can do two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

Write Issues

- So far we have looked at reading from cache (loads)
- What about writing into cache (stores)?
- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate

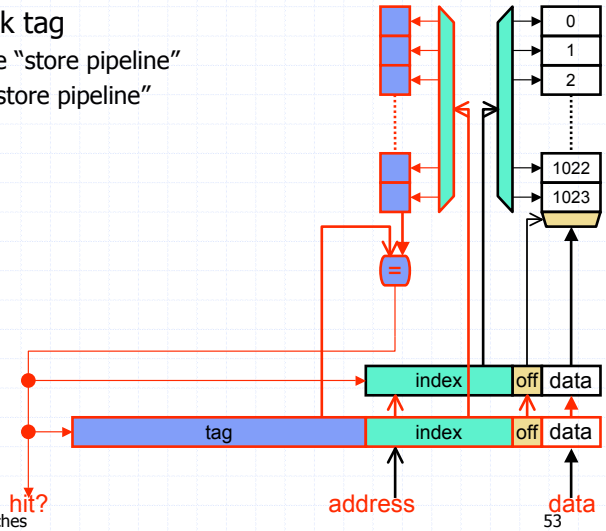
Tag/Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK)
- Writes: read tag, write data in parallel?
  - Tag mis-match → clobbered data (oops)
  - For associtaive cache, which way is written?
- Writes are a pipelined 2 cycle process
  - Cycle 1: match tag
  - Cycle 2: write to matching way



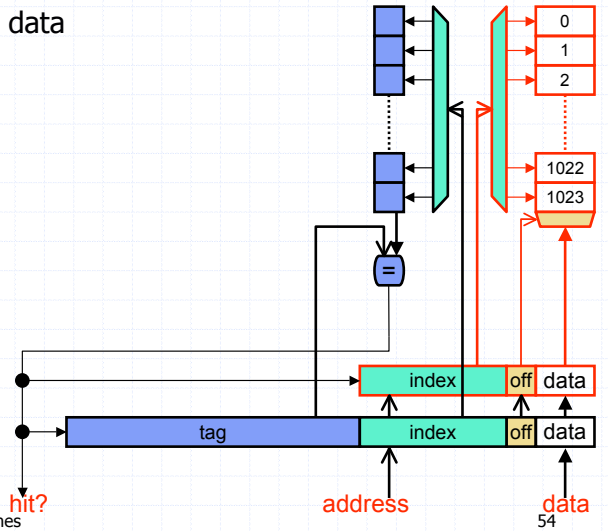
Tag/Data Access

- Cycle 1: check tag
  - Hit? Advance "store pipeline"
  - Miss? Stall "store pipeline"



Tag/Data Access

- Cycle 2: write data



Write-Through vs. Write-Back

- When to propagate new value to (lower level) memory?
  - **Write-through**: immediately
    - \+ Conceptually simpler
    - \+ Uniform latency on misses
    - – Requires additional bus bandwidth
  - **Write-back**: when block is replaced
    - Requires additional "**dirty**" bit per block
    - \+ Minimal bus bandwidth
      - Only writeback dirty blocks
    - – Non-uniform miss latency
      - Clean miss: one transaction with lower level (fill)
      - Dirty miss: two transactions (writeback + fill)
- Both are used, write-back is common

Write-allocate vs. Write-non-allocate

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

Low-Power Caches

- Caches consume significant power
  - 15% in Pentium4
  - 45% in StrongARM
- Two techniques
  - Way prediction (already talked about)
  - Dynamic resizing

Low-Power Access: Dynamic Resizing

- **Dynamic cache resizing**
  - Observation I: data, tag arrays implemented as many small arrays
  - Observation II: many programs don't fully utilize caches
  - Idea: dynamically turn off unused arrays
    - Turn off means disconnect power (V_{DD}) plane
    + Helps with both dynamic and static power
  - There are always tradeoffs
    – Flush dirty lines before powering down → costs power↑
    – Cache-size↓ → $\%_{miss}$↑ → power↑, execution time↑

Dynamic Resizing: When to Resize

- Use $\%_{miss}$ feedback
  - $\%_{miss}$ near zero? Make cache smaller (if possible)
  - $\%_{miss}$ above some threshold? Make cache bigger (if possible)
- Aside: how to track miss-rate in hardware?
  - Hard, easier to track miss-rate vs. some threshold
  - Example: is $\%_{miss}$ higher than 5%?
    - N-bit counter (N = 8, say)
    - Hit? counter –= 1
    - Miss? Counter += 19
    - Counter positive? More than 1 miss per 19 hits ($\%_{miss}$ > 5%)

Dynamic Resizing: How to Resize?

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

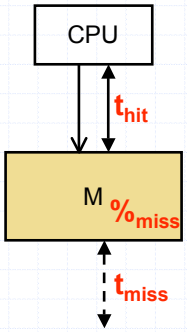
Memory Hierarchy Design

- Important: design hierarchy components together
- **I$**, **D$**: optimized for $latency_{hit}$ and parallel access
  - Insns/data in separate caches (**for bandwidth**)
  - Capacity: 8–64KB, block size: 16–64B, associativity: 1–4
  - Power: parallel tag/data access, way prediction?
  - Bandwidth: banking or multi-porting/replication
  - Other: write-through or write-back
- **L2**: optimized for $\%_{miss}$, power ($latency_{hit}$: 10–20)
  - Insns and data in one cache (for higher utilization, $\%_{miss}$)
  - Capacity: 128KB–2MB, block size: 64–256B, associativity: 4–16
  - Power: parallel or serial tag/data access, banking
  - Bandwidth: banking
  - Other: write-back
- **L3**: starting to appear ($latency_{hit}$ = 30)

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

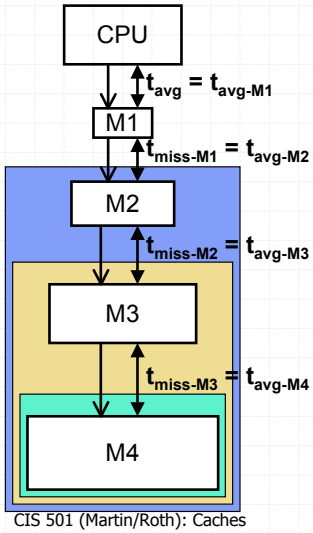
Memory Performance Equation



- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M
  - **$\%_{miss}$** (miss-rate): #misses / #accesses
  - **t_{hit}**: time to read data from (write data to) M
  - **t_{miss}**: time to read data into M
- Performance metric
  - **t_{avg}**: average access time

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

Hierarchy Performance



t_{avg}

t_{avg-M1}

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

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

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

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

...

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%

Performance Calculation I

- Parameters
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20%
  - Main memory: t_{hit} = 50ns
- What is $t_{avgD\$}$ without an L2?
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{hitM}$ = 1ns+(0.05*50ns) = 3.5ns
- What is $t_{avgD\$}$ 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

Performance Calculation II

- In a pipelined processor, I$/D$ t_{hit} is "built in" (effectively 0)
- Parameters
  - Base pipeline CPI = 1
  - Instruction mix: 30% loads/stores
  - I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles
- What is new CPI?
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02*10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{memory} * \%_{missD\$} * t_{missD\$}$ = 0.30*0.10*10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1+0.2+0.3= 1.5

An Energy Calculation

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

Current Cache Research

- "Drowsy Caches"
  - Data/tags allowed to leak away (power)
- "Frequent Value Cache"/"Compressed Cache"
  - Frequent values like 0, 1 compressed (performance, power)
- "Direct Address Cache" + "Cool Cache"
  - Support tag-unchecked loads in compiler and hardware (power)
- "Distance Associative Cache"
  - Moves frequently used data to closer banks/subarrays
  - Like an associative cache in which not all ways are equal

Summary

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