# CIS 501
# Computer Architecture

Unit 4: Caches

## This Unit: Caches

| Application | | |
|---|---|---|
| | OS | |
| Compiler | | Firmware |

| CPU | | I/O |
|---|---|---|
| | Memory | |
| Digital Circuits | | |
| Gates & Transistors | | |

- Basic memory hierarchy concepts
  - Speed vs capacity

- Caches

- Later
  - Organizing an entire memory hierarchy
  - Main memory
  - Virtual memory

- Note: much of this should be review, some new stuff (hopefully)

## Readings

- H+P
  - Appendix C.1–C.3
  - Chapter 5.1–5.2

- Paper: week from Tuesday
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", ISCA 1990
    - ISCA's "most influential paper award" awarded 15 years later

## 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
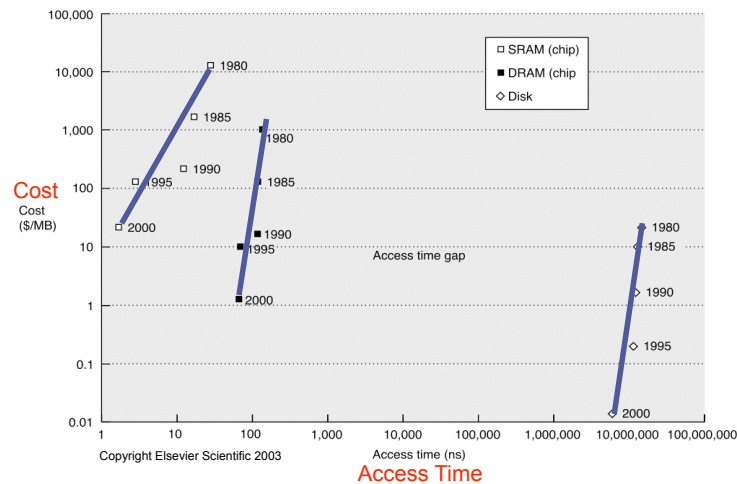
- Can't achieve all of these goals at once

# Review: Types of Memory

- **Static RAM (SRAM)**
  - 6 transistors per bit (two inverters, two other transistors for off/on)
  - 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

# Memory & Storage Technologies

- **Cost** - what can $200 buy today?
  - SRAM: 16MB
  - DRAM: 4,000MB (4GB) --- 250x cheaper than SRAM
  - Disk: 1,000,000MB (1TB) --- 250x 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 - 10-1000GB/sec
  - DRAM - ~10GB/sec
  - Disk - 100MB/sec (0.1 GB/sec) - sequential access only
- Aside: Flash, a non-traditional (and nonvolatile) memory
  - 32GB for $200, 8x cheaper than DRAM!  (But 32x more than disk)

# Storage Technology Trends



Copyright Elsevier Scientific 2003

# The "Memory Wall"



Copyright Elsevier Scientific 2003
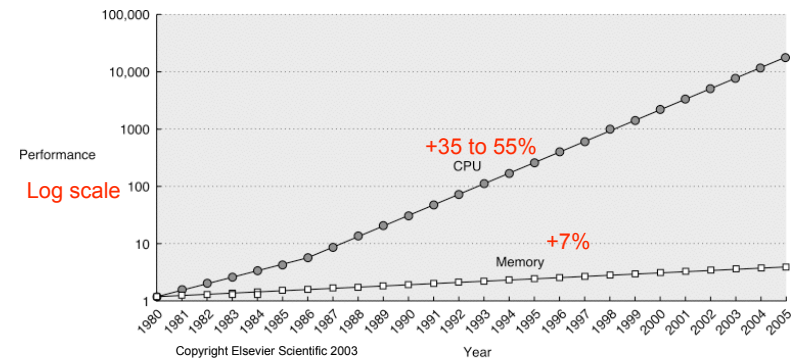
- Processors are get faster more quickly than memory (note log scale)
  - Processor speed improvement: 35% to 55%
  - Memory latency improvement: 7%

## 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

## Locality to the Rescue

- **Locality of memory references**
  - Property of real programs, few exceptions
  - Books and library analogy (next slide)

- **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
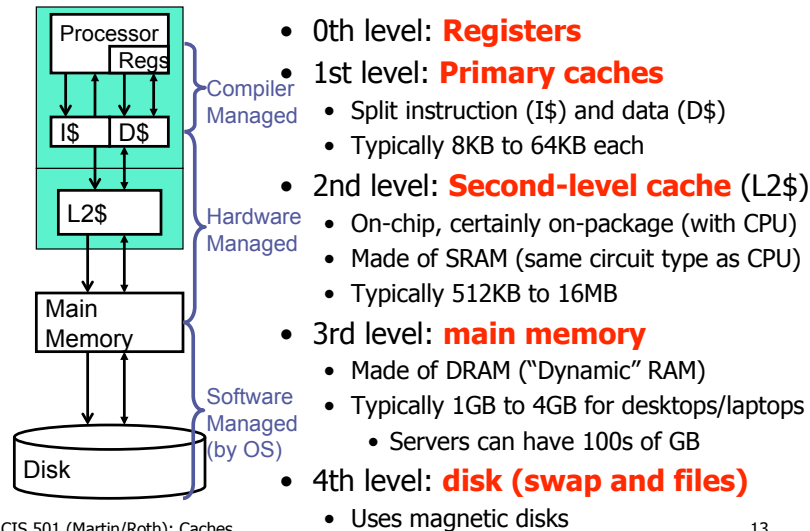
- Holds for data and instructions

## Library Analogy

- Consider books in a library

- Library has lots of books, but it is slow to access
  - Far away (time to walk to the library)
  - Big (time to walk within the library)

- How can you avoid these latencies?
  - Check out books, take them home with you
    - Put them on desk, on bookshelf, etc.
  - But desks & bookshelves have limited capacity
    - Keep recently used books around (**temporal locality**)
    - Grab books on related topic at the same time (**spatial locality**)
    - Guess what books you'll need in the future (prefetching)

## Exploiting Locality: Memory Hierarchy
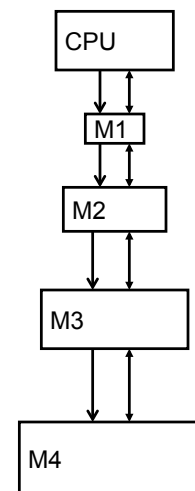


- Hierarchy of memory components
  - Upper components
    - Fast $\leftrightarrow$ Small $\leftrightarrow$ Expensive
  - Lower components
    - Slow $\leftrightarrow$ Big $\leftrightarrow$ Cheap
- Connected by "buses"
  - Which also have latency and bandwidth issues
- Most frequently accessed data in 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



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

## Library Analogy Revisited

- Registers ↔ books on your desk
  - Actively being used, small capacity

- Caches ↔ bookshelves
  - Moderate capacity, pretty fast to access

- Main memory ↔ library
  - Big, holds almost all data, but slow

- Disk (swap) ↔ inter-library loan
  - Very slow, but hopefully really uncommon

## Evolution of Cache Hierarchies
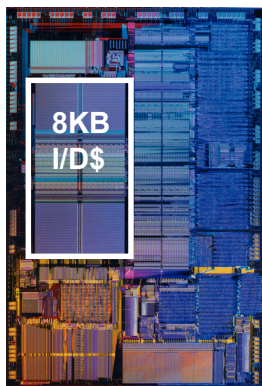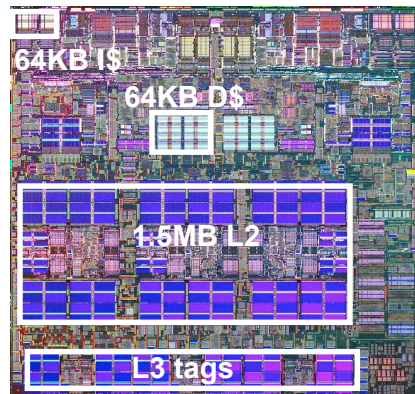


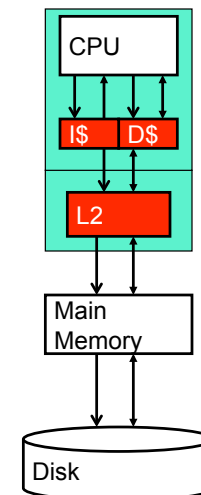Intel 486          IBM Power5 (dual core)

- Chips today are 30–70% cache by area
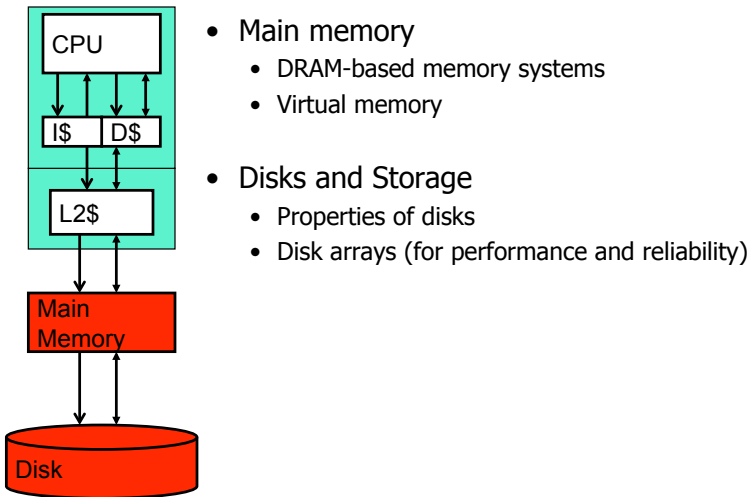
## This Unit: Caches



- "Cache": hardware managed
  - Hardware automatically retrieves missing data
  - Built from fast SRAM, usually on-chip today
  - In contrast to off-chip, DRAM "main memory"
- Cache organization
  - ABC
  - Miss classification
- High-performance techniques
  - Reducing misses
  - Improving miss penalty
  - Improving hit latency
- Some example performance calculations

# Looking forward: Memory and Disk



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

- Disks and Storage
  - Properties of disks
  - Disk arrays (for performance and reliability)

# 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



1024*256bit SRAM

10 bits

wordlines

bitlines
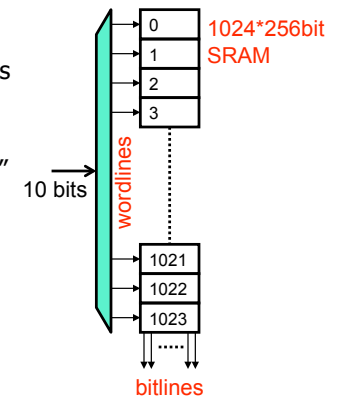
# FYI: Physical Memory Layout

- 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



address    data
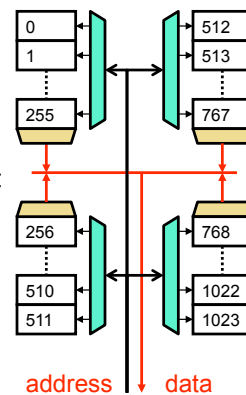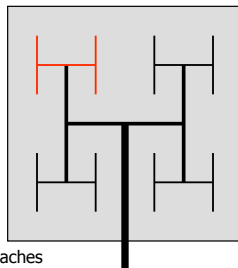
# Physical Cache Layout

- Arrays and h-trees make caches easy to spot in $\mu$graphs

# Caches: Finding Data via Indexing

- 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?
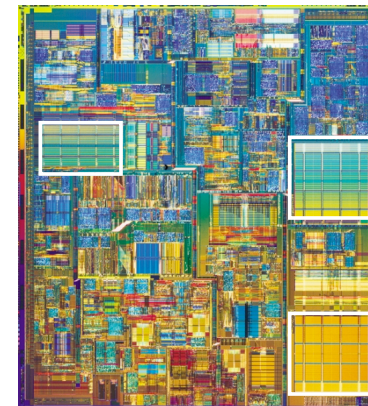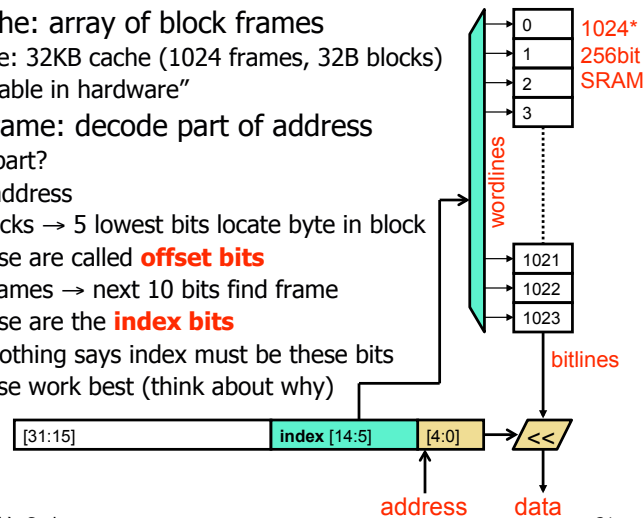  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block
    - These are called **offset bits**
  - 1024 frames → next 10 bits find frame
    - These are the **index bits**
  - Note: nothing says index must be these bits
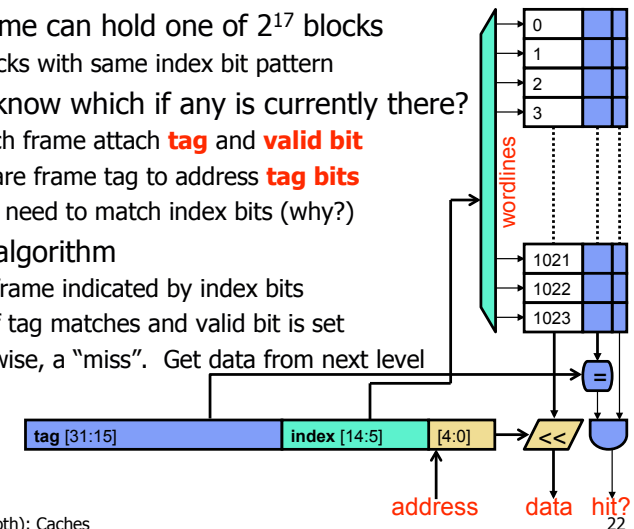  - But these work best (think about why)

1024*
256bit
SRAM

wordlines

| 0 |
| 1 |
| 2 |
| 3 |

| 1021 |
| 1022 |
| 1023 |

bitlines

[31:15] | **index** [14:5] | [4:0] | << |

address    data

# Knowing that You Found It: Tags

- Each frame can hold one of $2^{17}$ blocks
  - All blocks with same index bit pattern
- How to know which if any is currently there?
  - To each frame attach **tag** and **valid bit**
  - 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

wordlines

| 0 |
| 1 |
| 2 |
| 3 |

| 1021 |
| 1022 |
| 1023 |

=

**tag** [31:15] | **index** [14:5] | [4:0] | << |

address    data   hit?

# 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?
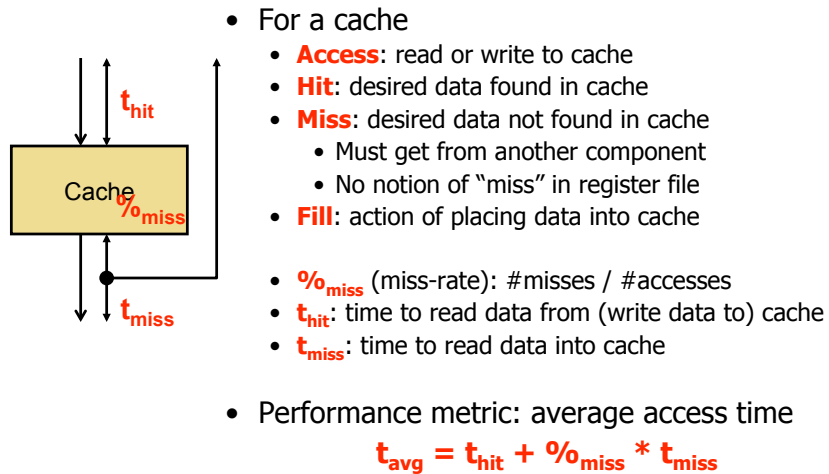  - Tag increases to 49bits, ~20% overhead (worst case)

# 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/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 component
    - No notion of "miss" in register file
  - **Fill**: action of placing data into cache

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

- Performance metric: average access time

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

# 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

# Measuring Cache Performance

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

# Cache Miss Paper Simulation

- 4-bit addresses → 16B memory
  - Simpler cache diagrams than 32-bits

- 8B cache, 2B blocks
  - Figure out number of sets: 4 (capacity / block-size)
  - Figure out how address splits into offset/index/tag 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, values don't matter

| Cache contents | | | | Address | Outcome |
|---|---|---|---|---|---|
| Set00 | Set01 | Set10 | Set11 | | |
| 0000\|0001 | 0010\|0011 | 0100\|0101 | 0110\|0111 | | |

# 4-bit Address, 8B Cache, 2B Blocks

| | | Main memory |
|---|---|---|
| 0000 | A | |
| 0001 | B | |
| 0010 | C | |
| 0011 | D | |
| 0100 | E | |
| 0101 | F | |
| 0110 | G | |
| 0111 | H | |
| 1000 | I | |
| 1001 | J | |
| 1010 | K | |
| 1011 | L | |
| 1100 | M | |
| 1101 | N | |
| 1110 | P | |
| 1111 | Q | |

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

**Data**

| Set | Tag | 0 | 1 |
|---|---|---|---|
| 00 | | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |

---

# 4-bit Address, 8B Cache, 2B Blocks

| | | Main memory |
|---|---|---|
| 0000 | A | |
| 0001 | B | |
| 0010 | C | |
| 0011 | D | |
| 0100 | E | |
| 0101 | F | |
| 0110 | G | |
| 0111 | H | |
| 1000 | I | |
| 1001 | J | |
| 1010 | K | |
| 1011 | L | |
| 1100 | M | |
| 1101 | N | |
| **1110** | **P** | |
| 1111 | Q | |

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

`Load: 1110  Miss`

**Data**

| Set | Tag | 0 | 1 |
|---|---|---|---|
| 00 | 0 | A | B |
| 01 | 0 | C | D |
| 10 | 0 | E | F |
| 11 | 0 | G | H |

---

# 4-bit Address, 8B Cache, 2B Blocks

| | | Main memory |
|---|---|---|
| 0000 | A | |
| 0001 | B | |
| 0010 | C | |
| 0011 | D | |
| 0100 | E | |
| 0101 | F | |
| 0110 | G | |
| 0111 | H | |
| 1000 | I | |
| 1001 | J | |
| 1010 | K | |
| 1011 | L | |
| 1100 | M | |
| 1101 | N | |
| **1110** | **P** | |
| 1111 | Q | |

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

`Load: 1110  Miss`

**Data**

| Set | Tag | 0 | 1 |
|---|---|---|---|
| 00 | 0 | A | B |
| 01 | 0 | C | D |
| 10 | 0 | E | F |
| 11 | **1** | **P** | **Q** |

---

# 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
    - **"Working set"**: insns/data program is actively using
    - Diminishing returns
  - – However $t_{hit}$ increases
    - Latency proportional to sqrt(capacity)
  - $t_{avg}$ ?

$\%_{miss}$

"working set" size

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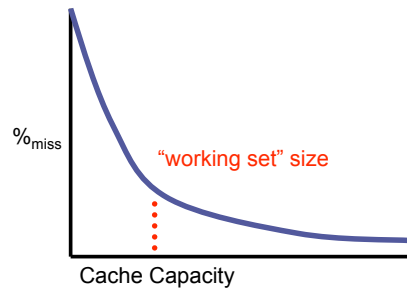
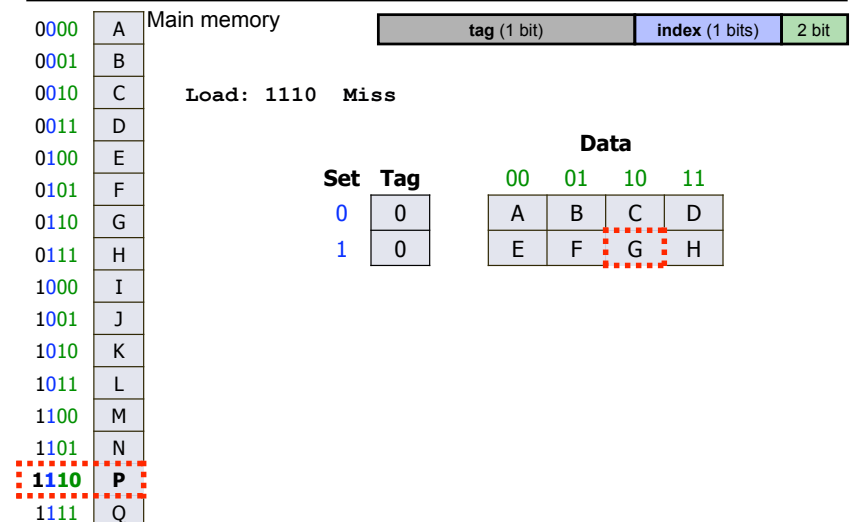- Given capacity, manipulate $\%_{miss}$ by changing **organization**

# Block Size

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

512*512bit SRAM

0
1
2

510
511

9-bit

block size↑

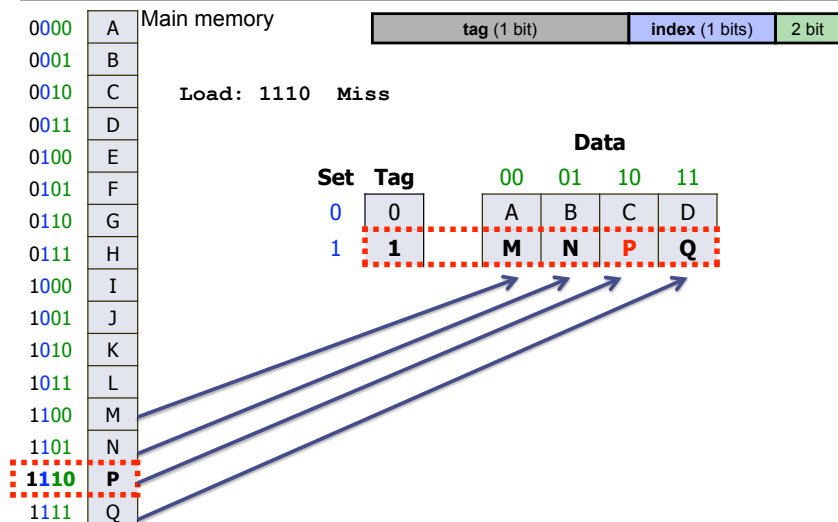[31:15]   [14:6]   [5:0]

address       data hit?
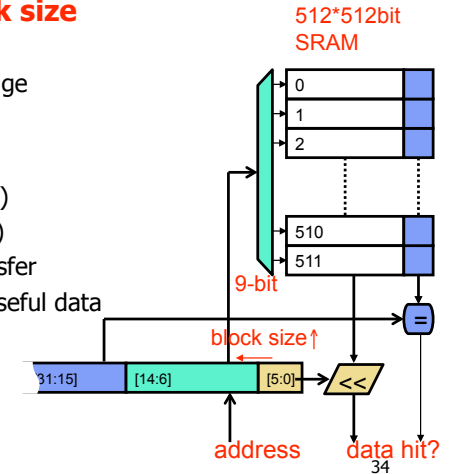
# Block Size and Tag Overhead

- 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

# 4-bit Address, 8B Cache, 4B Blocks

| | | Main memory |
|---|---|---|
| 0000 | A | |
| 0001 | B | |
| 0010 | C | |
| 0011 | D | |
| 0100 | E | |
| 0101 | F | |
| 0110 | G | |
| 0111 | H | |
| 1000 | I | |
| 1001 | J | |
| 1010 | K | |
| 1011 | L | |
| 1100 | M | |
| 1101 | N | |
| **1110** | **P** | |
| 1111 | Q | |

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

Load: 1110  Miss

**Data**

| Set | Tag | 00 | 01 | 10 | 11 |
|---|---|---|---|---|---|
| 0 | 0 | A | B | C | D |
| 1 | 0 | E | F | G | H |

# 4-bit Address, 8B Cache, 4B Blocks

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

Main memory

| | |
|---|---|
| 0000 | A |
| 0001 | B |
| 0010 | C |
| 0011 | D |
| 0100 | E |
| 0101 | F |
| 0110 | G |
| 0111 | H |
| 1000 | I |
| 1001 | J |
| 1010 | K |
| 1011 | L |
| 1100 | M |
| 1101 | N |
| 1110 | P |
| 1111 | Q |

Load: 1110  Miss

**Data**

| Set | Tag | 00 | 01 | 10 | 11 |
|---|---|---|---|---|---|
| 0 | 0 | A | B | C | D |
| 1 | 1 | M | N | P | Q |

---

# 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
    - Consider entire cache as one big block
- Both effects always present
  - Spatial prefetching dominates initially
    - Depends on size of the cache
  - Good block size is 16–128B
    - Program dependent

$\%_{miss}$ vs Block Size

---
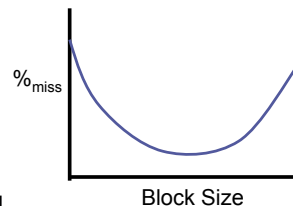
# Block Size and Miss Penalty

- 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)

# 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)**
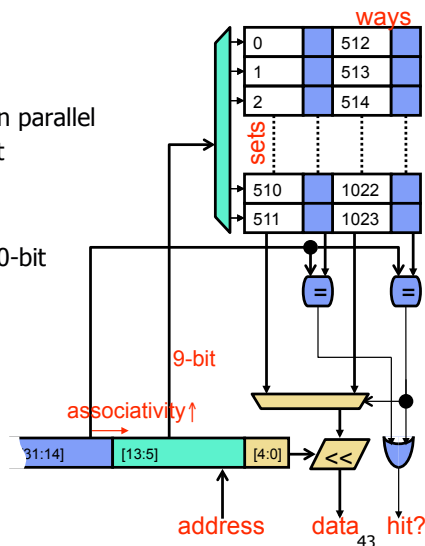
  - + Reduces conflicts
  - – Increases latency$_{hit:}$
    - additional tag match & muxing

- Note: valid bit not shown
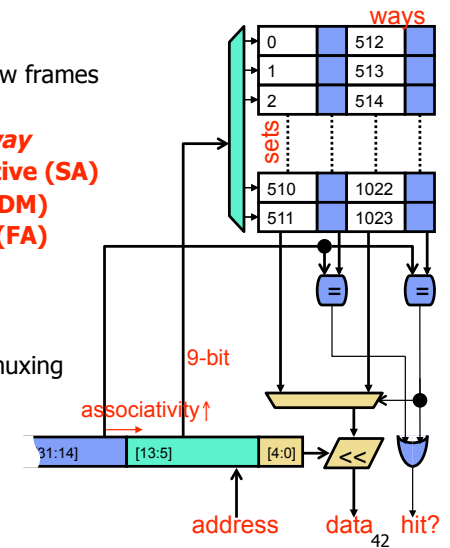
# Set-Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all 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 Miss Paper Simulation

- 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 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?
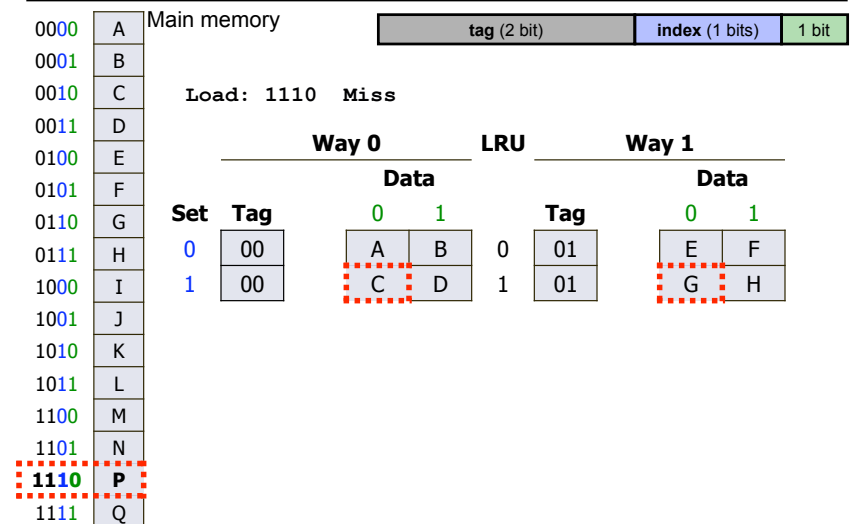
# NMRU and Miss Handling

- Add **LRU** field to each set
  - "Least recently used"
  - LRU data is encoded "way"
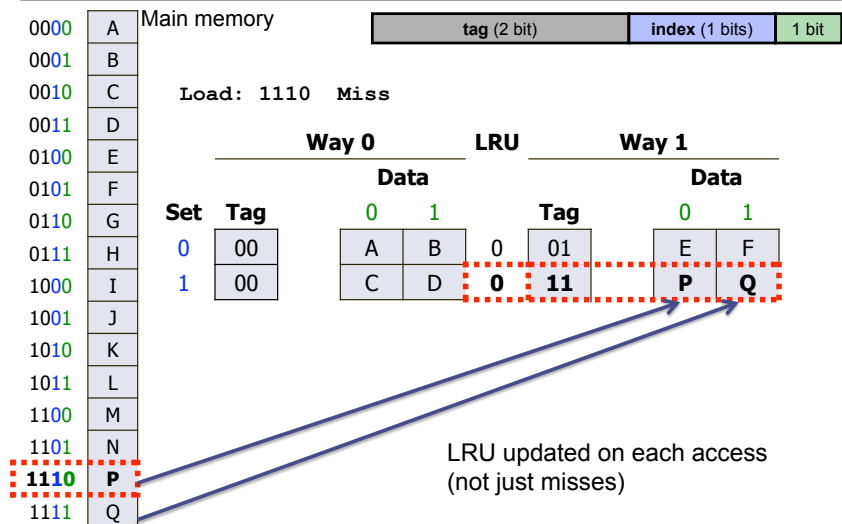  - Hit? update MRU

- LRU bits updated on each access
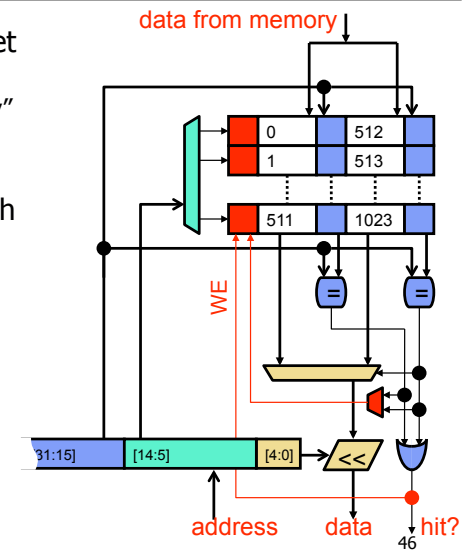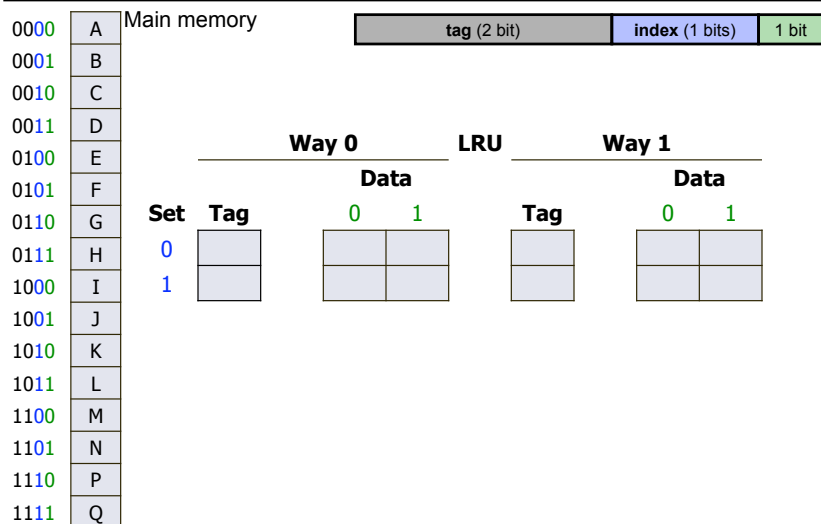
# 4-bit Address, 8B Cache, 2B Blocks, 2-way

# 4-bit Address, 8B Cache, 2B Blocks, 2-way

Load: 1110  Miss

## 4-bit Address, 8B Cache, 2B Blocks, 2-way

| | | Main memory |
|---|---|---|
| 0000 | A | |
| 0001 | B | |
| 0010 | C | |
| 0011 | D | |
| 0100 | E | |
| 0101 | F | |
| 0110 | G | |
| 0111 | H | |
| 1000 | I | |
| 1001 | J | |
| 1010 | K | |
| 1011 | L | |
| 1100 | M | |
| 1101 | N | |
| **1110** | **P** | |
| 1111 | Q | |

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

Load: 1110  Miss

| | | Way 0 | | | LRU | | Way 1 | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Data** | | | | | **Data** | | |
| Set | Tag | 0 | 1 | | | Tag | 0 | 1 | |
| 0 | 00 | A | B | | 0 | 01 | E | F | |
| 1 | 00 | C | D | | **0** | **11** | **P** | **Q** | |

LRU updated on each access
(not just misses)

CIS 501 (Martin/Roth): Caches                    49

## Parallel or Serial Tag Access?

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



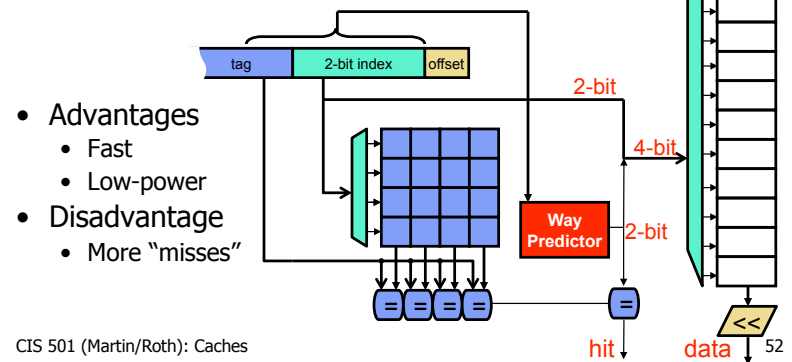Four blocks transferred

CIS 501 (Martin/Roth): Caches                    50
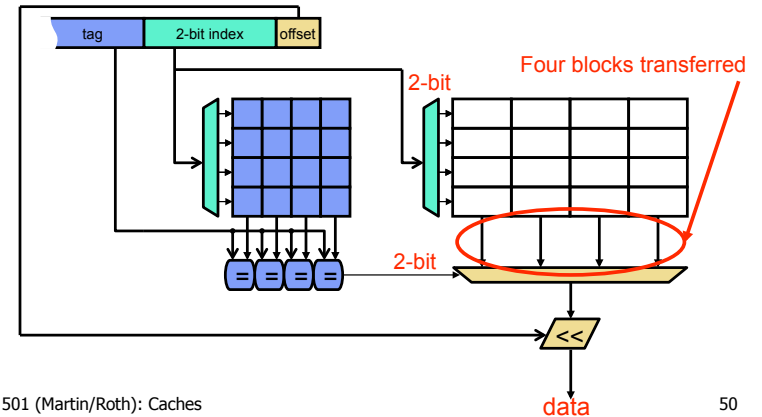
## Serial Tag Access

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



Only one block transferred

CIS 501 (Martin/Roth): Caches                    51
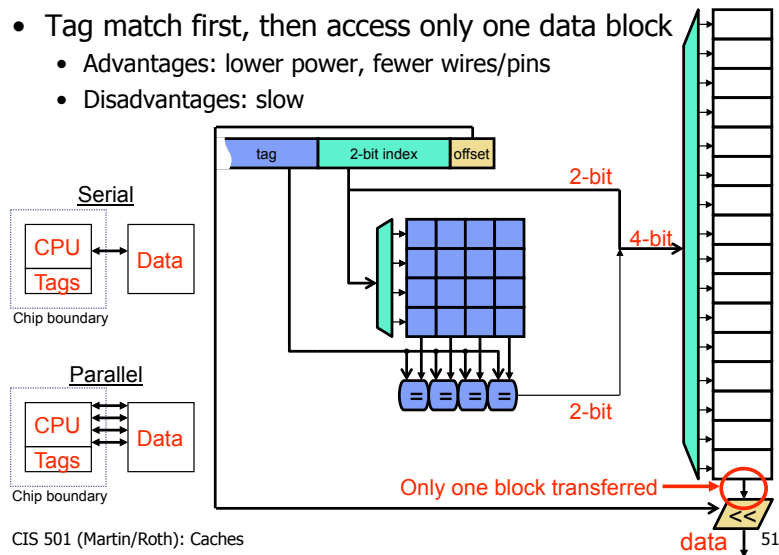
## Best of Both? Way Prediction

- Predict "way" of block
  - Just a "hint"
  - Use the index plus some tag bits
  - Table of $n$-bit entries for $2^n$ associative cache
  - Update on mis-prediction or replacement

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



CIS 501 (Martin/Roth): Caches                    52

# Associativity And Performance

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

$\%_{miss}$    ~5    Associativity
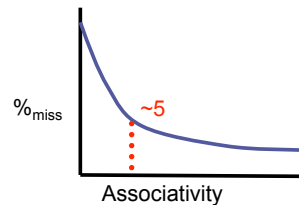
- Block-size and number of sets should be powers of two
  - Makes indexing easier (just rip bits out of the address)
- 3-way set-associativity? No problem

# Classifying Misses: 3C Model (Hill)

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of 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 (later)
- Calculated by multiple simulations
  - Simulate infinite cache, fully-associative cache, normal cache
  - Subtract to find each count

# Miss Rate: ABC

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

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

# Reducing Conflict Misses: Victim Buffer

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

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

I$/D$

VB

L2
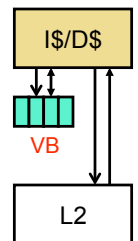
# Overlapping Misses: Lockup Free Cache

- **Lockup free**: allows other accesses while miss is pending
  - Consider: Load [r1] -> r2;  Load [r3] -> r4;   Add r2, r4 -> r5
  - **Handle misses in parallel**
    - "memory-level parallelism"
  - Makes sense for…
    - Processors that can go ahead despite D$ miss (out-of-order)
  - Implementation: **miss status holding register (MSHR)**
    - Remember: miss address, chosen frame, requesting instruction
    - When miss returns know where to put block, who to inform
  - Common scenario: "hit under miss"
    - Handle hits while miss is pending
    - Easy
  - Less common, but common enough: "miss under miss"
    - A little trickier, but common anyway
    - Requires multiple MSHRs: search to avoid frame conflicts

# 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)



Better locality

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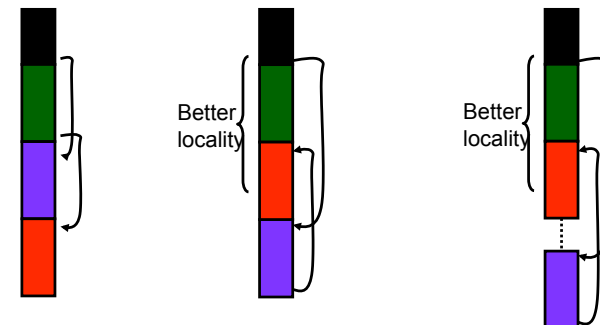
- Fewer taken branches, too
- Intra-procedure, inter-procedure

# Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - Key: anticipate upcoming miss addresses accurately
    - Can do in software or hardware

  - Simple example: **next block prefetching**
    - Miss on address **X** → anticipate miss on **X+block-size**
    + Works for insns: sequential execution
    + Works for data: arrays

  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Coverage**: prefetch for as many misses as possible
  - **Accuracy**: don't pollute with unnecessary data
    - It evicts useful data

---
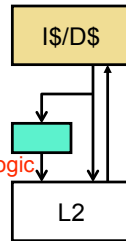
# Software Prefetching

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

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

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

---

# Hardware Prefetching

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

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

---

# More Advanced Address Prediction

- "Next-block" prefetching is easy, what about other options?
- **Correlating predictor**
  - Large table stores (miss-addr → 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

# Capacity Misses: Compression

- **Compression**: store insns/data in compressed form
    + Increase effective capacity
    − Increase latency$_{hit}$ (may have to decompress)
        - Decompression algorithm must be fast (i.e., not LZW)

- Pretty straightforward for I$ [Game+, IBM'99]
    - Compression performed statically
        + No layout "holes"
        + No need to modify cache itself (e.g., with additional tags)
    - Can use a "dictionary" approach (e.g., IBM CodePack for PowerPC)

- More difficult (but doable) for D$, L2 [Alameldeen+, ISCA'04]

# Write Issues

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

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

# Tag/Data Access

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

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

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

# Write Propagation

- When to propagate new value to (lower level) memory?

- **Option #1: Write-through**: immediately
    - On hit, update cache
    - Immediately send the write to the next level

- **Option #2: Write-back**: when block is replaced
    - Requires additional "dirty" bit per block
        - Replace **clean** block: **no extra traffic**
        - Replace **dirty** block: **extra "writeback" of block**
    + **Writeback-buffer**: keep it off critical path of miss
        - 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 contents to next-level
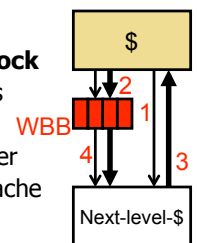
# Write Propagation Comparison

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

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

# Write Miss Handling

- How is a write miss actually handled?

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

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

# Write Misses and Write Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?

- **Write buffer**: a small buffer
  - Stores put address/value to write buffer, **keep going**
  - Write buffer writes stores to D$ in the background
  - Loads must search write buffer (in addition to D$)
  - + Eliminates stalls on write misses (mostly)
  - – Creates some problems (later)

- Write buffer vs. writeback-buffer
  - Write buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks
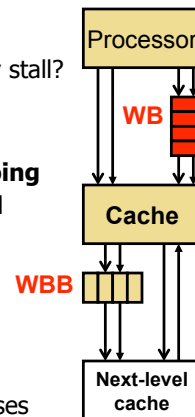
# Write Buffer Examples

- Example #1:
  - Store "1" into address A
    - Miss in cache, put in write buffer (initiate miss)
  - Load from address B
    - Hit in cache, read value from cache
  - Wait for miss to fill, write a "1" to A when done

- Example #2:
  - Store "1" into address A
    - Miss, put in write buffer (initiate miss)
  - Load from address A
    - Miss in cache, but do we stall? Don't need to stall
    - Just bypass load value from the write buffer

# Write Buffer Examples

- Example #3:
  - Store byte value "1" into address A
    - Miss in cache, put in write buffer (initiate miss)
  - Load word from address A, A+1, A+2, A+3
    - Hit in cache, read value from where?
    - Read both cache and write buffer (byte-by-byte merge)
- Store buffer holds address, data, and **per-byte** valid bits
- Example #4:
  - Store byte value "1" into address A (initiate miss)
  - Store byte value "2" into address B (initiate miss)
  - Store byte value "3" into Address A (???)
    - Can the first and last store share the same entry?
  - What if "B" fills first?
    - Can the second store leave before the first store?

# Memory Performance Equation
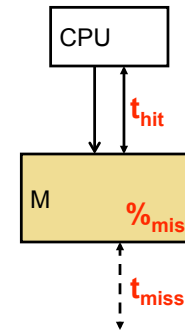


- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M

  - $\%_{miss}$ (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
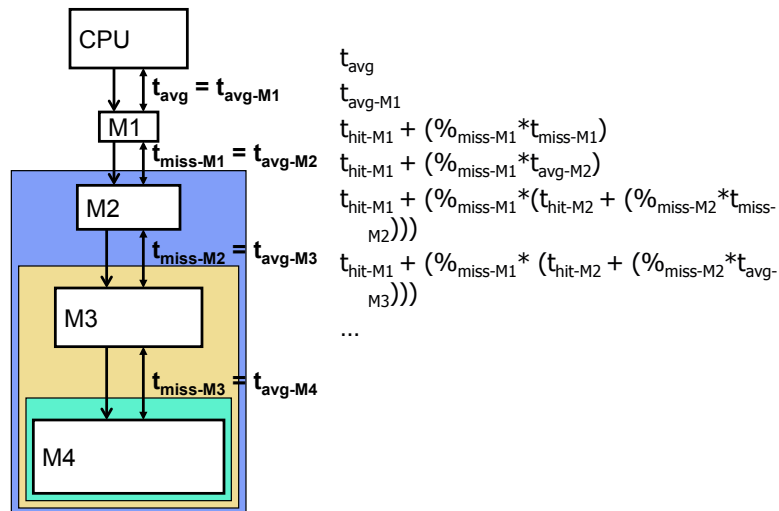


$t_{avg}$
$t_{avg-M1}$
$t_{hit-M1} + (\%_{miss-M1}*t_{miss-M1})$
$t_{hit-M1} + (\%_{miss-M1}*t_{avg-M2})$
$t_{hit-M1} + (\%_{miss-M1}*(t_{hit-M2} + (\%_{miss-M2}*t_{miss-M2})))$
$t_{hit-M1} + (\%_{miss-M1}* (t_{hit-M2} + (\%_{miss-M2}*t_{avg-M3})))$
...

# 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

- 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
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$} = 1+0.2+0.3 = 1.5$

# Performance Calculation II

- Parameters
  - Reference stream: all loads
  - D\$: $t_{hit}$ = 1ns, $\%_{miss}$ = 5%
  - L2: $t_{hit}$ = 10ns, $\%_{miss}$ = 20%   (local miss rate)
  - Main memory: $t_{hit}$ = 50ns

- What is $t_{avgD\$}$ without an L2?
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$}*t_{hitM} = 1ns+(0.05*50ns) = 3.5ns$

- 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 III

- Memory system parameters
  - D\$: $t_{hit}$ = 1ns, $\%_{miss}$ = 10%, 50% dirty, writeback-buffer, write-buffer
  - Main memory: $t_{hit}$ = 50ns
  - 32-byte block size

- Reference stream: 20% stores, 80% loads

- What is $t_{avgD\$}$?
  - Write-buffer → hides store misses latency
  - Writeback-buffer → hides dirty writeback latency
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{loads} * \%_{missD\$}*t_{hitM} = 1ns+(0.8*0.10*50ns) = 5ns$

# Bandwidth Calculation

- Memory system parameters
  - D\$: $t_{hit}$ = 1ns, $\%_{miss}$ = 10%, 50% dirty, writeback-buffer, write-buffer
  - Main memory: $t_{hit}$ = 50ns
  - 32-byte block size

- Reference stream: 20% stores, 80% loads

- What is the average bytes transferred per miss?
  - All misses: 32-byte blocks
  - Dirty evictions: 50% of the time * 32-byte block
  - 48B per miss

- Average bytes transfer per memory operation?
  - 10% of memory operations miss, so 4.8B per memory operation

# Designing a Cache Hierarchy

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

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

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

# Memory Hierarchy Parameters

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

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

# Split vs. Unified Caches

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

- **Unified L2, L3**: insns and data together
  - To minimize $\%_{miss}$
  - + Fewer capacity misses: unused insn capacity can be used for data
  - − More conflict misses: insn/data conflicts
    - A much smaller effect in large caches
  - Insn/data structural hazards are rare: simultaneous I\$/D\$ miss
  - Go even further: unify L2, L3 of multiple cores in a multi-core

# Hierarchy: Inclusion versus Exclusion

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

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

- **Non-inclusion**
  - No guarantees

# 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" + "NUCA"
  - Moves frequently used data to closer banks/subarrays
  - Like an associative cache in which not all ways are equal
- "Cache Compression"
  - Use simple compression to increase capacity of secondary caches
  - Saves bandwidth, too

# 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
- **Write issues**
  - Write-back vs. write-through/write-allocate vs. write-no-allocate