Admin

- Reading
  - H&P Chapter 5
  - "Cache Ways on Demand"
  - "An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches"
- Only two more homeworks...
- Work on your projects...
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 50-100ns
  - 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 (2010 Newegg.com)?
  - SRAM - 12MB (?? 2010)
  - DRAM - 12,000MB (12GB) – $0.025/MB
  - Disk – 2,000,000MB (2,000GB = 2TB) -- $0.00015/MB

- **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 - ~1-2GB/sec
  - Disk - 100MB/sec (0.1 GB/sec) - sequential access only

- **Aside: Flash, a non-traditional (and nonvolatile) memory**
  - 32,000MB (32GB) for $300, cheaper than DRAM!
Historical Storage Technology Trends

The “Memory Wall”

- Processors are getting 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
  - $\text{latency}_{\text{avg}} = \text{latency}_{\text{hit}} + \%_{\text{miss}} \times \text{latency}_{\text{miss}}$
  - 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
  - DRAM-based memory systems
- Disks and Storage
  - Properties of disks
  - Disk arrays (for performance and reliability)
Cache Performance

CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time

Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty + Writes x Write miss rate x Write miss penalty)

Memory stall clock cycles = Memory accesses x Miss rate x Miss penalty

CPU time = IC \times (\text{CPI}_{\text{execution}} + (\text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty})) \times \text{Clock cycle time}

Misses per instruction = Memory accesses per instruction \times Miss rate

CPU time = IC \times (\text{CPI}_{\text{execution}} + \text{Misses per instruction} \times \text{Miss penalty}) \times \text{Clock cycle time}
Example

- Miss penalty 50 clocks
- Miss rate 2%
- Base CPI 2.0
- 1.33 references per instruction
- Compute the CPUtime

\[
\text{CPUtime} = \text{IC} \times (2.0 + (1.33 \times 0.02 \times 50)) \times \text{Clock}
\]

- CPUtime = IC \times 3.33 \times \text{Clock}
- So CPI increased from 2.0 to 3.33 with a 2% miss rate

Improving Cache Performance

Ave Mem Acc Time =

\[
\text{Hit time} + (\text{miss rate} \times \text{miss penalty})
\]

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.
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 graphs.

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 $\rightarrow$ 5-bit offset
  - 1024 frames $\rightarrow$ 10-bit index
  - 32-bit address – 5-bit offset – 10-bit index = 17-bit tag
  - (17-bit tag + 1-bit valid) $\times$ 1024 frames = 18Kb tags = 2.2KB tags
  - $\approx$6% overhead
- What about 64-bit addresses?
  - Tag increases to 49bits, $\approx$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

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 3020, 0030, 0100, 0110, 0120, 0130</td>
<td>3030</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 3030, 2100, 0110, 0120, 0130</td>
<td>0030</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0110</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 1010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>1000, 1010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss</td>
</tr>
</tbody>
</table>

Hill's 3C Miss Rate Classification

- Compulsory
  - Miss caused by initial access

- Capacity
  - Miss caused by finite capacity
  - I.e., would not miss in infinite cache

- Conflict
  - Miss caused by finite associativity
  - I.e., would not miss in a fully-associative cache

- Coherence (4th C, added by Jouppi)
  - Miss caused by invalidation to enforce coherence
Miss Rate: ABC

- **Capacity**
  - + Decreases capacity misses
  - \(-\) Increases latency_{hit}
- **Associativity**
  - + Decreases conflict misses
  - \(-\) Increases latency_{hit}
- **Block size**
  - \(-\) Increases conflict/capacity misses (fewer frames)
  - + Decreases compulsory/capacity misses (spatial prefetching)
  - \(\blacktriangleright\) No effect on latency_{hit}
  - \(-\) May increase latency_{miss}

Increase Cache Size

- Biggest caches always have better miss rates
  - \(\blacktriangleright\) However latency_{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

### 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 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)

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000(0010), 0020(0030), 0100(0110), 0120(0130)</td>
<td>3020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), <strong>3020(3030)</strong>, 0100(0110), 0120(0130)</td>
<td>3030</td>
<td>Hit (spatial locality)</td>
</tr>
<tr>
<td>0000(0010), 3020(3030), 0100(0110), 0120(0130)</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), 3020(3030), <strong>2100(2110)</strong>, 0120(0130)</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>0000(0010), 3020(3030), 2100(2110), 0120(0130)</td>
<td>0020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), <strong>0020(0030)</strong>, 2100(2110), 0120(0130)</td>
<td>0030</td>
<td>Hit (spatial locality)</td>
</tr>
<tr>
<td>0000(0010), 0020(0030), 2100(2110), 0120(0130)</td>
<td>0110</td>
<td>Miss (conflict)</td>
</tr>
<tr>
<td>0000(0010), 0020(0030), <strong>0100(0110)</strong>, 0120(0130)</td>
<td>0100</td>
<td>Hit (spatial locality)</td>
</tr>
<tr>
<td>0000(0010), 0020(0030), 0100(0110), 0120(0130)</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), 0020(0030), <strong>2100(2110)</strong>, 0120(0130)</td>
<td>3020</td>
<td>Miss</td>
</tr>
</tbody>
</table>
Large Blocks and Subblocking

- Large cache blocks can take a long time to refill
  - refill cache line \textit{critical word first}
  - restart cache access before complete refill
- 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
- Sparse access patterns can use $1/S$ of the cache
  - $S$ is subblocks per block

\[ \begin{array}{ccccc}
  v & \text{subblock} & v & \text{subblock} & \cdots & v & \text{subblock} & \text{tag} \\
\end{array} \]

Conflicts

- What about pairs like 3030/0030, 0100/2100?
  - These will \textbf{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

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 3020, 0030, 0100, 0110, 0120, 0130</td>
<td>$3030$</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 3020, $3030$, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130</td>
<td>0020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, 3030, 2100, 0110, 0120, 0130</td>
<td>$0030$</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, $0030$, 2100, 0110, 0120, 0130</td>
<td>0110</td>
<td>Hit</td>
</tr>
</tbody>
</table>
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
Full-Associativity

- How to implement full (or at least high) associativity?
  - 1K tag matches? Unavoidable, but at least tags are small
  - 1K data reads? Terribly inefficient

Full-Associativity with CAMs

- **CAM**: content associative memory
  - Array of words with built-in comparators
  - Matchlines instead of bitlines
  - Output is "one-hot" encoding of match

- FA cache?
  - Tags as CAM
  - Data as RAM

- **Hardware is not software**
  - No such thing as software CAM (although "associative arrays" exist in some languages e.g., perl, python)
Associativity and Performance

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

<table>
<thead>
<tr>
<th>Cache contents</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0000,0100], [0010,0110], [0020,0120], [0030,0130]</td>
<td>3020</td>
<td>Miss</td>
</tr>
<tr>
<td>[0000,0100], [0010,0110], [0120,3020], [0030,0130]</td>
<td>3030</td>
<td>Miss</td>
</tr>
<tr>
<td>[0000,0100], [0010,0110], [0120,3020], [0130,3030]</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [0120,3020], [0130,3030]</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>[0100,2100], [0110,0010], [0120,3020], [0130,3030]</td>
<td>0020</td>
<td>Miss</td>
</tr>
<tr>
<td>[0100,2100], [0110,0010], [3020,0020], [0130,3030]</td>
<td>0030</td>
<td>Miss</td>
</tr>
<tr>
<td>[0100,2100], [0110,0010], [3020,0020], [3030,0030]</td>
<td>0110</td>
<td>Hit</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [3020,0020], [3030,0030]</td>
<td>0100</td>
<td>Hit (avoid conflict)</td>
</tr>
<tr>
<td>[2100,0100], [0010,0110], [3020,0020], [3030,0030]</td>
<td>2100</td>
<td>Hit (avoid conflict)</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [3020,0020], [3030,0030]</td>
<td>3020</td>
<td>Hit (avoid conflict)</td>
</tr>
</tbody>
</table>

Increase Associativity

- Higher associative caches have better miss rates
  - However latency_{hit} increases
- Diminishing returns (for a single thread)
Example Calculation #2

- Two caches: both 64KB, 32 byte blocks, miss penalty 70ns, 1.3 references per instruction, CPI 2.0 w/ perfect cache

- direct mapped
  - Cycle time 2ns
  - Miss rate 1.4%

- 2-way associative
  - Cycle time increases by 10%
  - Miss rate 1.0%

- Which is better?
  - Compute average memory access time
  - Compute CPU time

Example 2 Continued

- Ave Mem Acc Time =
  - Hit time + (miss rate x miss penalty)
    - 1-way: 2.0 + (0.014 x 70) = 2.98ns
    - 2-way: 2.2 + (0.010 x 70) = 2.90ns

- CPUtime = IC x CPIexec x Cycle
  - CPIexec = CPIbase + ((memacc/inst) x Miss rate x miss penalty)
  - Note: miss penalty x cycle time = 70ns
    - 1-way: IC x ((2.0 x 2.0) + (1.3x0.014x70)) = 5.27 x IC
    - 2-way: IC x ((2.0 x 2.2) + (1.3x0.010x70)) = 5.31 x IC
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

![Diagram of cache memory with MRU/LRU fields and data access paths]
Parallel or Serial Tag Access?

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

  ![Parallel Access Diagram]

  - Four blocks transferred

Serial Tag Access

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

  ![Serial Access Diagram]

  - Only one block transferred
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

<table>
<thead>
<tr>
<th>Cache contents</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, <strong>3020</strong>, 0030, 0100, 0110, 0120, 0130</td>
<td>3030</td>
<td>Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 0020, <strong>3030</strong>, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, <strong>2100</strong>, 0110, 0120, 0130</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0020</td>
<td>Miss (capacity)</td>
</tr>
<tr>
<td>0000, 0010, <strong>0020</strong>, 0030, 2100, 0110, 0120, 0130</td>
<td>0030</td>
<td>Miss (capacity)</td>
</tr>
<tr>
<td>0000, 0010, 0020, <strong>0030</strong>, 2100, 0110, 0120, 0130</td>
<td>0110</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0100</td>
<td>Miss (capacity)</td>
</tr>
<tr>
<td>0000, 1010, 0020, 0030, <strong>0100</strong>, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss (conflict)</td>
</tr>
<tr>
<td>1000, 1010, 0020, 0030, <strong>2100</strong>, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss (conflict)</td>
</tr>
</tbody>
</table>

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 for small caches
  - Does VB reduce $\%_{\text{miss}}$ or $\text{latency}_{\text{miss}}$?
Seznec’s Skewed-Associative Cache

Can get better utilization with less assoc?
average case? worst case?