Admin

- Work on Projects!
- Read the two papers “Cache Ways on Demand”, “NUCA”
This Unit: Caches

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

Review

- ABCs of caches
- 3C’s
- Hardware methods for reducing misses
- Know how to draw a block diagram of a cache

\[
\text{Ave Mem Acc Time} = \text{Hit time} + (\text{miss rate} \times \text{miss penalty})
\]

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

<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><strong>3030</strong></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, 0030, 2100, 0110, 0120, 0130</td>
<td><strong>0030</strong></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

  - 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, 0101, 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], [0120,0120], [0030,0130]</td>
<td>3030</td>
<td>Miss</td>
</tr>
<tr>
<td>[0000,0100], [0010,0110], [0120,020], [0130,030]</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>[0101, 2100], [0010,0110], [0120,020], [0130,030]</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>[0101, 2100], [0010,0110], [0120,020], [0130,030]</td>
<td>0020</td>
<td>Miss</td>
</tr>
<tr>
<td>[0101, 2100], [0010,0110], [0120,020], [0130,030]</td>
<td>0030</td>
<td>Miss</td>
</tr>
<tr>
<td>[0101, 2100], [0110,0110], [0120,020], [0130,030]</td>
<td>0110</td>
<td>Hit (avoid conflict)</td>
</tr>
<tr>
<td>[0101, 2100], [0010,0110], [0120,020], [0130,030]</td>
<td>0120</td>
<td>Hit (avoid conflict)</td>
</tr>
<tr>
<td>[0101, 2100], [0010,0110], [0120,020], [0130,030]</td>
<td>0120</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

**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 (longer hit time)

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, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>3030</td>
<td>Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss (compulsory)</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, 0030, 2100, 0110, 0120, 0130</td>
<td>0020</td>
<td>Miss (capacity)</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 3020, 3030, 0110, 0120, 0130</td>
<td>0030</td>
<td>Miss (capacity)</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 (capacity)</td>
</tr>
<tr>
<td>0000, 1010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss (conflict)</td>
</tr>
<tr>
<td>1000, 1010, 0020, 0030, 2100, 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

Bank 1

same index
redistribute to
different set

Bank 0

same index
same set

Can get better utilization with less assoc?

average case? worst case?
Can you as a programmer do anything to improve cache performance?

Compsci 220 / ECE 252 (Lebeck): Caches

Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
  - Several code restructuring techniques to improve both
    - Compiler must know that restructuring preserves semantics
- Change order of accesses to data
- 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];  // non-contiguous accesses
    ```
  - Better code
    ```
    for (i = 0; i<NROWS; i++)
        for (j = 0; j<NCOLS; j++)
            sum += X[i][j];  // contiguous accesses
    ```
Software Restructuring: Data

- **Loop blocking**: temporal locality
  - Poor code
    ```c
    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
    ```c
    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

Restructuring Loops

- **Loop Fusion**
  - Merge two independent loops
  - Increase reuse of data
  - Fusion Example:
    ```c
    for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i][j] = 1/b[i][j]*c[i][j];
    for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      d[i][j] = a[i][j]+c[i][j];
    ```
    **Fused Loop:**
    ```c
    for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      { a[i][j] = 1/b[i][j]*c[i][j];
        d[i][j] = a[i][j]+c[i][j];
      }
    ```

- **Loop Fission**
  - Split loop into independent loops
  - Reduce contention for cache resources
Layout and Cache Behavior

- Elements of a quadrant are contiguous
- Tile elements spread out in memory because of column-major mapping
- Fixed mapping into cache
- Self-interference in cache

Making Tiles Contiguous

- Change layout of data
- Elements of a quadrant are contiguous
- Recursive layout
- Elements of a tile are contiguous
- No self-interference in cache
**Non-linear Layout Functions**

- Different locality properties
- Different inclusion properties
- Different addressing costs

<table>
<thead>
<tr>
<th>4-D blocked</th>
<th>Morton order</th>
<th>Hilbert order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 4 5</td>
<td>0 3 4 5</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>2 3 6 7</td>
<td>1 2 7 6</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>8 9 12 13</td>
<td>14 13 8 9</td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>10 11 14 15</td>
<td>15 12 11 10</td>
</tr>
</tbody>
</table>

**Performance Improvement**

<table>
<thead>
<tr>
<th>CPU</th>
<th>UltraSPARC 2i</th>
<th>UltraSPARC 2</th>
<th>Alpha 21164</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock rate</td>
<td>300 MHz</td>
<td>300 MHz</td>
<td>500 MHz</td>
</tr>
<tr>
<td>L1 cache</td>
<td>16KB/32B/1</td>
<td>16KB/32B/1</td>
<td>8KB/32B/1</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512KB/64B/1</td>
<td>2MB/64B/1</td>
<td>96KB/64B/3</td>
</tr>
<tr>
<td>L3 cache</td>
<td>2MB/64B/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>320MB</td>
<td>512MB</td>
<td>512MB</td>
</tr>
<tr>
<td>TLB entries</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Page size</td>
<td>8KB</td>
<td>8KB</td>
<td>8KB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ultra 10</th>
<th>Ultra 60</th>
<th>Miata</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4D</td>
<td>MO</td>
<td>4D</td>
</tr>
<tr>
<td>BLKtxMxM</td>
<td>0.93</td>
<td>1.06</td>
<td>0.95</td>
</tr>
<tr>
<td>RECtxMxM</td>
<td>0.94</td>
<td></td>
<td>0.94</td>
</tr>
<tr>
<td>STRASSEN</td>
<td>0.87</td>
<td></td>
<td>0.79</td>
</tr>
<tr>
<td>CHOL</td>
<td>0.78</td>
<td></td>
<td>0.85</td>
</tr>
<tr>
<td>STDHaar</td>
<td>0.68</td>
<td>0.87</td>
<td>0.64</td>
</tr>
<tr>
<td>NONHaar</td>
<td>0.62</td>
<td>0.61</td>
<td>0.58</td>
</tr>
</tbody>
</table>
Software Restructuring: Code

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

- Intra-procedure, inter-procedure
- Related to trace scheduling

Miss Cost: Critical Word First/Early Restart

- Observation: \( \text{latency}_{\text{miss}} = \text{latency}_{\text{access}} + \text{latency}_{\text{transfer}} \)
  - \( \text{latency}_{\text{access}} \): time to get first word
  - \( \text{latency}_{\text{transfer}} \): 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
  - \( \text{latency}_{\text{miss}} = \text{latency}_{\text{access}} \)
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
  - Simplest scenario: “hit under miss”
    - Handle hits while miss is pending
    - Easy for OoO cores
  - More common: “miss under miss”
    - A little trickier, but common anyway
    - Requires split-transaction bus/interconnect
    - 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
    - How much latency do we need to hide (Little’s Law)?
    - 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

Summary

- ABCs of caches
- 3C’s
  
  Ave Mem Acc Time =
  
  Hit time + (miss rate x miss penalty)

  1. Reduce the miss rate,
  2. Reduce the miss penalty, or
  3. Reduce the time to hit in the cache.
- Hardware methods
- Software methods