This Unit: Caches

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

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

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, 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

---

Associativity and Performance

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

---

Increase Associativity

- Higher associative caches have better miss rates
  - However latency, 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 1.6%
  - 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) = 2.98 ns
  - 2-way: 2.2 + (0.010 x 70) = 2.90 ns
- CPU time = IC x CPIexec x Cycle
  - CPIexec = CPIbase + ((memacc/inst) x Miss rate x miss penalty)
  - 1-way: IC x ((2.0 x 2.0) + (1.3 x 0.014 x 70)) = 5.27 x IC
  - 2-way: IC x ((2.0 x 2.2) + (1.3 x 0.010 x 70)) = 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
      - NM RU (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)

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
  - Disadvantages
    - 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>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>3020 Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 3020, 0030, 0100, 0110, 0120, 0130</td>
<td>3030 Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 3020, 3030, 0100, 0110, 0120, 0130</td>
<td>2100 Miss (compulsory)</td>
</tr>
<tr>
<td>0000, 0010, 0030, 2100, 0110, 0120, 0130</td>
<td>0012 Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 3030, 2100, 0110, 0120, 0130</td>
<td>0030 Miss (capacity)</td>
</tr>
<tr>
<td>0000, 0010, 0020, 2100, 0110, 0120, 0130</td>
<td>0100 Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0130 Hit</td>
</tr>
<tr>
<td>0000, 1010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>2100 Miss (conflict)</td>
</tr>
<tr>
<td>1000, 1010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>3020 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 \(\%_{miss}\) or latency \(\%_{miss}\)?

Software Restructuring: Data $ $

- Can you as a programmer do anything to improve cache performance?
  - Several code restructuring techniques to improve both
    - Compiler must know that restructuring preserves semantics
  - **Change order of accesses to data**
    - 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 \(j = 0; j <\text{NCOLS}; j++\)
          - for \(i = 0; i <\text{NROWS}; i++\)
            - \(\text{sum} += X[i][j]; // \text{non-contiguous accesses}\)
        - Better code
          - for \(i = 0; i <\text{NROWS}; i++\)
            - for \(j = 0; j <\text{NCOLS}; j++\)
              - \(\text{sum} += X[i][j]; // \text{contiguous accesses}\)

Seznec's Skewed-Associative Cache

- Can get better utilization with less assoc?
  - average case? worst case?
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
    ```
    for (i=0; i<NELEMS; i+=CACHE_SIZE)
    for (k=0; k<NITERATIONS; k++)
    for (ii=0; ii<i+CACHE_SIZE; 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
  ```
  for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  a[i][j] = 1/b[i][j]*c[i][j];
  ```
  Fused Loop:
  ```
  for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  { 
  a[i][j] = 1/b[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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
</tbody>
</table>

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

Performance Improvement

<table>
<thead>
<tr>
<th>CPU</th>
<th>UltraSPARC 2</th>
<th>UltraSPARC 2</th>
<th>Alpha 21164</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock rate</td>
<td>500MHz</td>
<td>500MHz</td>
<td>500MHz</td>
</tr>
<tr>
<td>L1 cache</td>
<td>1MB/32B</td>
<td>1MB/32B</td>
<td>1MB/32B</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512MB/32B</td>
<td>512MB/32B</td>
<td>512MB/32B</td>
</tr>
<tr>
<td>L3 cache</td>
<td>2MB/32B</td>
<td>2MB/32B</td>
<td>2MB/32B</td>
</tr>
<tr>
<td>RAM</td>
<td>32MB</td>
<td>32MB</td>
<td>32MB</td>
</tr>
<tr>
<td>TLB entries</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Page size</td>
<td>1KB</td>
<td>1KB</td>
<td>1KB</td>
</tr>
<tr>
<td>Ultra 10</td>
<td>Ultra 60</td>
<td>Ultra 90</td>
<td></td>
</tr>
</tbody>
</table>

| BL-KM* | 0.95 | 0.95 | 0.95 |
| RSC-KM* | 0.96 | 0.96 | 0.96 |
| STRASSEN* | 0.97 | 0.97 | 0.97 |
| CNDL* | 0.75 | 0.75 | 0.75 |
| STDHALAR | 0.68 | 0.68 | 0.68 |
| NOHALAR | 0.67 | 0.67 | 0.67 |
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: latency_{miss} = latency_{access} + latency_{transfer}
- Goals:
  - latency_{access}: time to get first word
  - latency_{transfer}: time to get rest of block
- Optimizations:
  - 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

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 DS misses (out-of-order)
  - Implementation: miss status holding register (MSHR)
    - When miss returns know where to put block, who to inform
    - Simplest scenario: "hit under miss"
      - Handle hits while miss is pending
    - 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
  - Strategies:
    - 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(4[i][j]+BLOCK_SIZE);
          - 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
  - 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 useless 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**
  
  \[
  \text{Ave Mem Acc Time} = \text{Hit time} + (\text{miss rate} \times \text{miss penalty})
  \]
  
  1. Reduce the miss rate,
  2. Reduce the miss penalty, or
  3. Reduce the time to hit in the cache.
- Hardware methods
- Software methods