Memory Hierarchy—Improving Performance

Professor Alvin R. Lebeck
Computer Science 220
Fall 2006
Admin

- Homework #4 Due November 2\textsuperscript{nd}
- Work on Projects
- Midterm
  - Max 98
  - Min 50
  - Mean 80
- Read NUCA paper
Review: ABCs of caches

• Associativity
• Block size
• Capacity
• Number of sets $S = \frac{C}{BA}$
• 1-way (Direct-mapped)
  – $A = 1$, $S = \frac{C}{B}$
• N-way set-associative
• Fully associativity
  – $S = 1$, $C = BA$
• Know how a specific piece of data is found
  – Index, tag, block offset
Write Policies

• We know about write-through vs. write-back
• Assume: a 16-bit write to memory location 0x00 causes a cache miss.
• Do we change the cache tag and update data in the block?
  
  Yes: Write Allocate
  No: Write No-Allocate

• Do we fetch the other data in the block?
  
  Yes: Fetch-on-Write (usually do write-allocate)
  No: No-Fetch-on-Write

• Write-around cache
  
  – Write-through no-write-allocate
Sub-block Cache (Sectored)

• **Sub-block:**
  – Share one cache tag between all sub-blocks in a block
  – Each sub-block within a block has its own valid bit
  – Example: 1 KB Direct Mapped Cache, 32-B Block, 8-B Sub-block
    » Each cache entry will have: 32/8 = 4 valid bits

• **Miss:** only the bytes in that sub-block are brought in.
  – reduces cache fill bandwidth (penalty).

```
+----------------+----------------+----------------+----------------+----------------+
| Cache Tag      | SB3's V Bit    | SB2's V Bit    | SB1's V Bit    | SB0's V Bit    |
+----------------+----------------+----------------+----------------+----------------+
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
|                |                |                |                |                |
+----------------+----------------+----------------+----------------+----------------+

| Cache Data      | B31  ** B24   |                | B7  ** B0      |
|-----------------+----------------+----------------+----------------+
| Sub-block3      | Sub-block2     | Sub-block1     | Sub-block0     |
| Byte 1023       |                |                |                |
| Byte 992        |                |                |                |
```

© Alvin R. Lebeck 2006
Review: Four Questions for Memory Hierarchy Designers

• Q1: Where can a block be placed in the upper level? *(Block placement)*
  – Fully Associative, Set Associative, Direct Mapped

• Q2: How is a block found if it is in the upper level? *(Block identification)*
  – Tag/Block

• Q3: Which block should be replaced on a miss? *(Block replacement)*
  – Random, LRU

• Q4: What happens on a write? *(Write strategy)*
  – Write Back or Write Through (with Write Buffer)
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
Cache Performance

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

Misses per instruction = Memory accesses per instruction x Miss rate

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

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

- CPUtime = 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
**Example 2**

- Two caches: both 64KB, 32 byte blocks, miss penalty 70ns, 1.3 references per instruction, CPI 2.0 with 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: \(2.0 + (0.014 \times 70) = 2.98\text{ns}\)
  - 2-way: \(2.2 + (0.010 \times 70) = 2.90\text{ns}\)

- **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: \(IC \times ((2.0 \times 2.0) + (1.3 \times 0.014 \times 70)) = 5.27 \times IC\)
  - 2-way: \(IC \times ((2.0 \times 2.2) + (1.3 \times 0.010 \times 70)) = 5.31 \times IC\)
Review: Cache Performance

CPUtime = IC x (CPI_{execution} + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

hits are included in CPI_{execution}

Misses per instruction = Memory accesses per instruction x Miss rate

CPUtime = IC x (CPI_{execution} + Misses per instruction x Miss penalty) x Clock cycle time
Improving Cache Performance

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.
Reducing Misses

- **Classifying Misses: 3 Cs**
  - **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*. *(Misses in Infinite Cache)*
  - **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. *(Misses in Size X Cache)*
  - **Conflict**—If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses* or *interference misses*. *(Misses in N-way Associative, Size X Cache)*
3Cs Absolute Miss Rate

- 1-way
- 2-way
- 4-way
- 8-way

Conflict
Capacity

Cache Size (KB)

Compulsory

© Alvin R. Lebeck 2006
2:1 Cache Rule

Cache Size (KB)

Capacity

Conflict

1-way
2-way
4-way
8-way
3Cs Relative Miss Rate

Cache Size (KB)

- 1-way
- 2-way
- 4-way
- 8-way

Capacity

Conflict

Compulsory

© Alvin R. Lebeck 2006

CPS 220
How Can We Reduce Misses?

• Change Block Size? Which of 3Cs affected?

• Change Associativity? Which of 3Cs affected?

• Change Program/Compiler? Which of 3Cs affected?
1. Reduce Misses via Larger Block Size
2. Reduce Misses via Higher Associativity

- **2:1 Cache Rule:**
  - Miss Rate DM cache size $N \sim$ Miss Rate 2-way cache size $N/2$

- **Beware: Execution time is only final measure!**
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time external cache $+10\%$, internal $+2\%$ for 2-way vs. 1-way
Example: Avg. Memory Access Time vs. Miss Rate

- Example: assume Clock Cycle = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. Clock cycle for direct mapped

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.33</td>
<td>2.15</td>
<td>2.07</td>
<td>2.01</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.86</td>
<td>1.76</td>
<td>1.68</td>
</tr>
<tr>
<td>4</td>
<td>1.72</td>
<td>1.67</td>
<td>1.61</td>
<td>1.53</td>
</tr>
<tr>
<td>8</td>
<td>1.46</td>
<td>1.48</td>
<td>1.47</td>
<td>1.43</td>
</tr>
<tr>
<td>16</td>
<td>1.29</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
</tr>
<tr>
<td>32</td>
<td>1.20</td>
<td>1.24</td>
<td>1.25</td>
<td>1.27</td>
</tr>
<tr>
<td>64</td>
<td>1.14</td>
<td>1.20</td>
<td>1.21</td>
<td>1.23</td>
</tr>
<tr>
<td>128</td>
<td>1.10</td>
<td>1.17</td>
<td>1.18</td>
<td>1.20</td>
</tr>
</tbody>
</table>

(Red means A.M.A.T. not improved by more associativity)
3. Reducing Conflict Misses via Victim Cache

• How to combine fast hit time of Direct Mapped yet still avoid conflict misses?
• Add buffer to place data discarded from cache
• Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
4. Reducing Conflict Misses via Pseudo-Associativity

• How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?

• Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)

• Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  – Better for caches not tied directly to processor
5. Reducing Misses by HW Prefetching of Instruction & Data

• E.g., Instruction Prefetching
  – Alpha 21064 fetches 2 blocks on a miss
  – Extra block placed in stream buffer
  – On miss check stream buffer

• Works with data blocks too:
  – Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  – Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
  – Pointers vs. arrays
    – Kedem: Markov predictor (address correlation)

• Prefetching relies on extra memory bandwidth that can be used without penalty
6. Reducing Misses by SW Prefetching Data

• Data Prefetch
  – Load data into register (HP PA-RISC loads) binding
  – Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9) non-binding
  – Special prefetching instructions cannot cause faults; a form of speculative execution

• Issuing Prefetch Instructions takes time
  – Is cost of prefetch issues < savings in reduced misses?
Improving Cache Performance

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

• Classifying Misses: 3 Cs

  – **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*. *(Misses in Infinite Cache)*

  – **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. *(Misses in Size X Cache)*

  – **Conflict**—If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses* or *interference misses*. *(Misses in N-way Associative, Size X Cache)*
7. Reducing Misses by Program/Compiler Optimizations

• Instructions
  – Reorder procedures in memory so as to reduce misses
  – Profiling to look at conflicts
  – McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache with 4 byte blocks

• Data
  – *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  – *Loop Interchange*: change nesting of loops to access data in order stored in memory
  – *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  – *Blocking*: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows
Merging Arrays Example

/* Before */
int val[SIZE];
int key[SIZE];

/* After */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];

• Reducing conflicts between val & key
Loop Interchange Example

/* Before */
for (k = 0; k < 100; k = k+1)
    for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
    for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];

• Sequential accesses instead of striding through memory every 100 words
• What is miss rate before and after?
Loop Fusion Example

/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }

2 misses per access to a & c vs. one miss per access
Blocking Example

/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {r = 0;
     for (k = 0; k < N; k = k+1){
       r = r + y[i][k]*z[k][j];
     }
    x[i][j] = r;
  }

• Two Inner Loops:
  – Read all NxN elements of z[ ]
  – Read N elements of 1 row of y[ ] repeatedly
  – Write N elements of 1 row of x[ ]

• Capacity Misses a function of N & Cache Size:
  – 3 NxN => no capacity misses; otherwise ...

• Idea: compute on BxB submatrix that fits
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1)
        {r = 0;
            for (k = kk; k < min(kk+B-1,N); k = k+1) {
                r = r + y[i][k]*z[k][j];
            } 
            x[i][j] = x[i][j] + r;
        }

• Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
• B called *Blocking Factor*
• 6 loop variant exists
• Conflict Misses?
Reducing Conflict Misses by Blocking

- Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache
Summary of Program/Compiler Optimizations to Reduce Cache Misses

1.merge
2.loop interchange
3.loop fusion
4.blocking
Layout and Cache Behavior

• Tile elements spread out in memory because of column-major mapping
• Fixed mapping into cache
• Self-interference in cache
Making Tiles Contiguous

- 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>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<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>
</tr>
</tbody>
</table>

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

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

- Different locality properties
- Different inclusion properties
- Different addressing costs
## Performance Improvement

<table>
<thead>
<tr>
<th></th>
<th>UltraSPARC 2i</th>
<th>UltraSPARC 2</th>
<th>Alpha 21164</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>UltraSPARC 2i</td>
<td>UltraSPARC 2</td>
<td>Alpha 21164</td>
</tr>
<tr>
<td>Clock rate</td>
<td>300MHz</td>
<td>300 MHz</td>
<td>500MHz</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>2MB/64B/1</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>BLKMXM</td>
<td>0.93</td>
<td>1.06</td>
<td>0.95</td>
</tr>
<tr>
<td>RECMXM</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>0.67</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>
Comparison with TSS

BMXM, comparison with TSS

Execution time (seconds)

Problem size (elements)

- TSS/Ultra 10
- 4D/ Ultra 10, t=17
- 4D/ Ultra 10, t=30
Summary

\[ \text{CPUtime} = IC \times \left( \frac{\text{CPI}_{\text{Execution}}}{\text{Instruction}} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \right) \times \text{Clock cycle time} \]

- **3 Cs: Compulsory, Capacity, Conflict**
  - How to eliminate them
- **Program Transformations**
  - Change Algorithm
  - Change Data Layout
- **Implication:** Think about caches if you want high performance!