Lecture Outline

• How to improve cache performance?
  – Reducing Cache Miss Penalty
  – Reducing Miss Rate
  – Reducing Hit Time
  – Reducing Miss Penalty/Rate via parallelism
Where to misses come from?

- Classifying Misses: 3 Cs
  - Compulsory—First access to a block. Also called cold start misses or first reference misses. (Misses in even an 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 Fully Associative Cache)
  - Conflict—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. (Remaining Misses)

- 4th “C”: Coherence - Misses caused by cache coherence.

3Cs Absolute Miss Rate (SPEC92)
Cache Size

- Old rule of thumb: 2x size => 25% cut in miss rate
- What does it reduce?

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>Compulsory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.14</td>
</tr>
<tr>
<td>2</td>
<td>0.12</td>
</tr>
<tr>
<td>4</td>
<td>0.10</td>
</tr>
<tr>
<td>8</td>
<td>0.08</td>
</tr>
<tr>
<td>16</td>
<td>0.06</td>
</tr>
<tr>
<td>32</td>
<td>0.04</td>
</tr>
<tr>
<td>64</td>
<td>0.02</td>
</tr>
<tr>
<td>128</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Huge Caches => Working Sets

- First working set
- Capacity-generated traffic (including conflicts)
- Second working set
- Data traffic
- Other capacity-independent communication
- Cold-start (compulsory) traffic
- Replication capacity (cache size)

Example LU Decomposition from NAS Parallel Benchmarks
Cache Organization?

- Assume total cache size not changed:
- What happens if:

1) Change Block Size:

2) Change Associativity:

3) Change Compiler:

Which of 3Cs is obviously affected?

Larger Block Size (fixed size&assoc)

What else drives up block size?
### 3Cs Relative Miss Rate

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100%</td>
<td>80%</td>
<td>60%</td>
<td>40%</td>
<td>20%</td>
</tr>
<tr>
<td>1</td>
<td>90%</td>
<td>70%</td>
<td>50%</td>
<td>30%</td>
<td>10%</td>
</tr>
<tr>
<td>2</td>
<td>80%</td>
<td>60%</td>
<td>40%</td>
<td>20%</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>70%</td>
<td>50%</td>
<td>30%</td>
<td>10%</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>60%</td>
<td>40%</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>16</td>
<td>50%</td>
<td>30%</td>
<td>10%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>32</td>
<td>40%</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>64</td>
<td>30%</td>
<td>10%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>128</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

**Flaws:** for fixed block size  
**Good:** insight => invention

2:1 Rule of Thumb

---

Computer Science 146  
David Brooks
**Associativity vs Cycle Time**

- Beware: Execution time is only final measure!
- Why is cycle time tied to hit time?

- Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%
  - suggested big and dumb caches

---

**Example: Avg. Memory Access Time vs. Miss Rate**

- Example: assume Cache Cycle Time = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT 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)
Way Prediction

• Maintain hit-speed of a direct-mapped cache, get conflict miss reduction of associative cache
• Extra bits are used to predict the way of the next cache access
• Alpha 21264 uses 2-way set-associative I-Cache
  – If predictor is correct, 1-cycle I-cache latency
  – If incorrect, 3-cycle latency
  – SPEC95 => Prediction accuracy ~85%

Reducing Misses by Compiler Optimizations

• McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
• Instructions
  – Reorder procedures in memory so as to reduce conflict misses
  – Profiling to look at conflicts(using tools they developed)
• 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: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of structures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];

Reducing conflicts between val & key;
improve spatial locality

Padding and Offset Changes

- Cache size of 8KB

Before:
int a[2048];
int b[2048];
int c[2048];

for(i=0;i<2048;i++)
    C[i]=A[i]+B[i]

After:
int a[2050];
int b[2050];
int c[2050];

for(i=0;i<2048;i++)
    C[i]=A[i]+B[i]
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 in blocks instead of striding through memory every 100 words; improved spatial locality

C: Row-Major Order
X[0][0] X[0][1]
X[1][0] X[1][1]

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];
  }
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:
  - $2N^3 + N^2 \rightarrow$ (assuming no conflict; otherwise …)

- Idea: compute on BxB submatrix that fits in cache

Blocking: Before

Computer Science 146
David Brooks
Blocking Example

/* 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;  }

• B called Blocking Factor
• Capacity Misses from $2N^3 + N^2$ to $N^3/B+2N^2$
• Conflict Misses Too?

Blocking: After

Computer Science 146
David Brooks
Summary of Compiler Optimizations to Reduce Cache Misses (by hand)

Summary: Miss Rate Reduction

\[ \text{CPU time} = IC \times \left( \frac{CPI_{\text{nominal}}}{\text{Instruction}} \times \frac{\text{Memory accesses}}{\text{Miss rate}} \times \frac{\text{Miss rate}}{\text{Miss penalty}} \right) \times \text{Clock cycle time} \]

- 3 Cs: Compulsory, Capacity, Conflict
  0. Larger cache
  1. Reduce Misses via Larger Block Size
  2. Reduce Misses via Higher Associativity
  3. Reducing Misses via Victim Cache
  4. Reducing Misses via Way-Prediction
  5. Reducing Misses by Compiler Optimization
Review: Improving Cache Performance

• How to improve cache performance?
  – Reducing Cache Miss Penalty
  – Reducing Miss Rate
  – Reducing Miss Penalty/Rate via parallelism
  – Reducing Hit Time

Non-blocking Caches to reduce stalls on misses

• *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss
  – requires out-of-order execution
  – requires multi-bank memories
• “hit under miss” reduces the effective miss penalty by working during miss vs. ignoring CPU requests
• “hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses
  – Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  – Requires multiple memory banks (otherwise cannot support)
  – Penium Pro allows 4 outstanding memory misses
Value of Hit Under Miss for SPEC

- FP programs on average: AMAT = 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT = 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss

Reducing Misses by Hardware
Prefetching of Instructions & Data

- Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in "stream buffer" not the cache
  - On Access: check both cache and stream buffer
  - On SB Hit: move line into cache
  - On SB Miss: Clear and refill SB with successive lines

- 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

- Prefetching relies on having extra memory bandwidth that can be used without penalty
Hardware Prefetching

- What to prefetch?
  - One block ahead (spatially)
    - What will this work well for?
  - Address prediction for non-sequential data
    - Correlated predictors (store miss, next_miss pairs in table)
    - Jump-pointers (augment data structures with prefetch pointers)

- When to prefetch?
  - On every reference
  - On a miss (basically doubles block size!)
  - When resident data becomes “dead” -- how do we know?
    - No one will use it anymore, so it can be kicked out

Reducing Misses by Software Prefetching Data

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

- Prefetching comes in two flavors:
  - Binding prefetch: Requests load directly into register.
    - Must be correct address and register!
  - Non-Binding prefetch: Load into cache.
    - Can be incorrect. Faults?

- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth
Reducing Hit Times

• Some common techniques/trends
  – Small and simple caches
    • Pentium III – 16KB L1
    • Pentium 4 – 8KB L1
  – Pipelined Caches (actually bandwidth increase)
    • Pentium – 1 clock cycle I-Cache
    • Pentium III – 2 clock cycle I-Cache
    • Pentium 4 – 4 clock cycle I-Cache
  – Trace Caches
    • Beyond spatial locality
    • Dynamic sequences of instruction (including taken branches)

Hit Time Reduction

• Translation of Virtual (programmer) to Physical (memory) Addresses is a bottleneck on hit time
• Defer this topic to next time
**Cache Optimization Summary**

<table>
<thead>
<tr>
<th>Technique</th>
<th>MR</th>
<th>MP</th>
<th>HT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td>–</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Higher Associativity/Larger Cache</td>
<td>+</td>
<td>–</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Way-Predicting Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>+</td>
<td>+</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td>+</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Second Level Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Pipelined Cache</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Trace Cache</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

**Cache Bandwidth**

- Superscalars need multiple memory access per cycle
- Parallel cache access: more difficult than parallel ALUs
  - Caches have state so multiple accesses will affect each other
- “True Multiporting”
  - Multiple decoders, read/write wordlines per SRAM cell
  - Pipeline a single port by “double pumping” Alpha 21264
  - Multiple cache copies (like clustered register file) POWER4
- Interleaved Multiporting
  - Cache divides into banks – two accesses to same bank => conflict
Next Time

- Virtual Memory
- Main Memory