Eleven Advanced Optimization for Cache Performance

- Reducing hit time
- Reducing miss penalty
- Reducing miss rate
- Reducing miss penalty * miss rate


Reducing Hit Time

- Small and simple caches
- Pipelined cache access
- Trace caches
- Avoid time loss in address translation
  - Virtually indexed, physically tagged cache
    - simple and effective approach
    - possible only if cache is not too large
  - Virtually addressed cache
    - protection?, multiple processes?, aliasing?, I/O?

Small and Simple Caches

- Small size => faster access
- Small size => fit on the chip, lower delay
- Simple (direct mapped) => lower delay
- Second level – tags may be kept on chip

Cache access time estimates using CACTI

- .8 micron technology, 1 R/W port, 32 b address, 64 b o/p, 32 b block

© 2003 Elsevier Science (USA). All rights reserved.
Pipelined Cache Access

- Multi-cycle cache access but pipelined
- Reduces cycle time but hit time is more than one cycle
- Pentium 4 takes 4 cycles
- Greater penalty on branch misprediction
- More clock cycles between issue of load and use of data
  - IF IF IF in pipeline

Trace Caches: Pre-decoded

- What maps to a cache block?
  - Not statically determined
  - Decided by the dynamic sequence of instructions, including predicted branches
- Used in Pentium 4 (NetBurst architecture)
- Starting addresses not word size * powers of 2
- Better utilization of cache space
- Downside—same instruction may be stored multiple times

Reducing Cache Miss Penalty

Reducing Miss Penalty

- Multi-level caches
- Critical word first and early restart
- Giving priority to read misses over write
- Merging write buffer
- Victim caches

Multi Level Caches

Average memory access time =

\[ \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1} \]

\[ \text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2} \]

Critical Word First and Early Restart

- Read policy: Concurrent Read/Forward
- Load policy: Wrap around load

More effective when block size is large
**Read Miss Priority Over Write**
- Provide write buffers
- Processor writes into buffer and proceeds (for write through as well as write back)

On read miss
- wait for buffer to be empty, or
- check addresses in buffer for conflict

---

**Merging Write Buffer**
Merge writes belonging to same block in case of write through

---

**Victim Cache: Recycle bin/Dust bin**
- Evicted blocks are recycled
- Much faster than getting a block from the next level
- Size = 1 to 5 blocks
- A significant fraction of misses may be found in victim cache

---

**Reducing Cache Miss Rate**
- Large Block Size
- Larger Cache
- Higher Associativity
- Way prediction and pseudo-associative cache
- Compiler optimizations

---

**Large Block Size**
- Take benefit of spatial locality
- Reduces compulsory misses
- Too large block size - misses increase
- Miss Penalty increases
**Large Cache**
- Reduces capacity misses
- Hit time increases
- Keep small L1 cache and large L2 cache

**Higher Associativity**
- Reduces conflict misses
- 8-way is almost like fully associative
- Hit time increases: What to do?
  - Pseudo Associativity

**Way Prediction and Pseudo-associative Cache**
Way prediction: low miss rate of SA cache with hit time of DM cache
- Only one tag is compared initially
- Extra bits are kept for prediction
- Hit time in case of mis-prediction is high
Pseudo-assoc. or column assoc. cache: get advantage of SA cache in a DM cache
- Check sequentially in a pseudo-set
- Fast hit and slow hit

**Compiler optimizations**
- Loop interchange
  - Improve spatial locality by scanning arrays row-wise
- Blocking
  - Improve temporal and spatial locality

**Improving Locality**
Matrix Multiplication example
\[
[C] = [A] \times [B]
\]
\[L \times M \quad L \times N \quad N \times M\]

**Cache Organization for the example**
- Cache line (or block) = 4 matrix elements.
- Matrices are stored row wise.
- Cache can’t accommodate a full row/column.
  - L, M and N are so large w.r.t. the cache size
  - After an iteration along any of the three indices, when an element is accessed again, it results in a miss.
- Ignore misses due to conflict between matrices.
  - As if there was a separate cache for each matrix.
Matrix Multiplication: Code I

for (i = 0; i < L; i++)
for (j = 0; j < M; j++)
for (k = 0; k < N; k++)
C[i][j] += A[i][k] * B[k][j];

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>accesses</td>
<td>LM</td>
<td>LN</td>
</tr>
<tr>
<td>misses</td>
<td>LN/4</td>
<td>LNM/4</td>
</tr>
</tbody>
</table>

Total misses = LM(5N+1)/4

Matrix Multiplication: Code II

for (k = 0; k < N; k++)
for (i = 0; i < L; i++)
for (j = 0; j < M; j++)
C[i][j] += A[i][k] * B[k][j];

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>accesses</td>
<td>LMN</td>
<td>LN</td>
</tr>
<tr>
<td>misses</td>
<td>LNM/4</td>
<td>LN</td>
</tr>
</tbody>
</table>

Total misses = LN(2M+4)/4

Matrix Multiplication: Code III

for (i = 0; i < L; i++)
for (k = 0; k < N; k++)
for (j = 0; j < M; j++)
C[i][j] += A[i][k] * B[k][j];

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>accesses</td>
<td>L MN</td>
<td>LN</td>
</tr>
<tr>
<td>misses</td>
<td>L MN/4</td>
<td>LN</td>
</tr>
</tbody>
</table>

Total misses = LN(2M+1)/4

Reducing MissPenalty * MissRate

- Non-blocking cache
- Hardware prefetching
- Compiler controlled prefetching

Non-blocking Cache

In OOO processor

- Hit under a miss
  - complexity of cache controller increases
- Hit under multiple misses or miss under a miss
  - memory should be able to handle multiple misses
Hardware Prefetching

- Prefetch items before they are requested
  - both data and instructions
- What and when to prefetch?
  - fetch two blocks on a miss (requested+next)
- Where to keep prefetched information?
  - in cache
  - in a separate buffer (most common case)

Prefetch Buffer/Stream Buffer

MatMul: Code III

```
for (i = 0; i < L; i++)
  for (k = 0; k < N; k++)
    for (j = 0; j < M; j++)
      C[i][j] += A[i][k] * B[k][j];
```

Compiler Controlled Pre-fetching

- Semantically invisible (no change in registers or cache contents)
- Makes sense if processor doesn’t stall while prefetching (non-blocking cache)
- Overhead of prefetch instruction should not exceed the benefit

Course after Mid: Multicore Computing

Good luck for your Mid Semester Exam