Cache Optimization

Today's topics:
- Look at memory access times
  - improve hit time
  - reduce miss rate
  - reduce miss penalty

Full disclosure
- I'm winging this one
- more detail on white board than slides

Optimize What

- **Basic quantitative metric**
  \[
  \text{Average Memory Access Time} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}
  \]

- *but in a parallel world it's about exposed latency*
  \[
  \frac{\text{Memory Stall Cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total Miss Latency} - \text{Overlapped Miss Latency})
  \]

- *note that miss penalty*
  - \( f(\text{transfer_rate/bandwidth, latency\_next\_lower\_cache}) \)
  - so improve bandwidth helps

- **Optimize**
  - reduce hit time (Amdahl's law – it's the common case)
  - increase cache bandwidth
  - reduce miss penalty
  - reduce miss rate
  - increase overlap
Knee of Curve Problem

- Bigger problem for small caches – e.g. L1
  - latency vs. transfer time

Reduce Hit Time

- Small and simple caches
  - keep cache hierarchy on chip
    - off chip access is 10-100x penalty
  - small, direct-mapped
    - note L1$ size doesn't change much w/ technology
    - L2 is where the biggest change occurs
  - associativity is a double edged sword

- Next slide
  - models based on CACTI
    - common research tool
    - book model is CACTI-IV
      - note this doesn't accurately deal w/ wire delay
      - current version is 6.5
      - much better wire models – SPICE back annotated validation
    - undecided
      - might be used in HW4
Hit Time Effects

Hit-time Improvement #2

- **Way prediction**
  - basics covered last time
    - saves comparator power
    - increased associativity of set-assoc caches
      - reduces conflict misses
      - but way prediction give performance of direct mapped cache
  - tactics similar to branch prediction
    - PC of load or store
      - keep track of which way was hit
        - very similar to local Gselect predictor
      - keep track of progression
        - similar to stride prediction

- **Practice**
  - way prediction used in both MIPS and Pentium IV processors

- **Prediction accuracy**
  - ~85%
Increase Cache Bandwidth

- **Pipelined caches**
  - Just like processor
    - pipelining can increase latency
    - fill and spill penalty when things go wrong
  - throughput improves on average
    - note L1 latency increasing
      - 3'ish cycles today but launch a new access every cycle

- **Non-blocking caches**
  - for when L1+ misses occur
  - MSHR's and dynamic issue logic

---

**Non-blocking Cache Data**

![Graph showing non-blocking cache data with benchmarks on the x-axis and percentage of average memory stall time on the y-axis. The graph includes lines for hit under 1 miss, hit under 2 misses, and hit under 64 misses.]
Multi-Banked Caches

- **Interleave for improved bandwidth**
  - **practice**
    - Opteron – 2 banks
    - Sun Niagara – 4 banks
  - **Idea**
    - no conflict accesses issues faster than bank latency
    - alternative to hit-under-miss & MSHR solution
    - banks are smaller so latency is reduced

- **Bigger advantage w/ multi-core and shared L2**
  - **downside**
    - faster bus
    - OR multiple buses
  - **note:**
    - long wires on buses inherently induce problems
    - slower performance – C effect
    - high power – signal integrity
    - pre- and post-emphasis (e.g. QPI, HT, dynamic balance)

---

Reduce Miss Penalty

- **Critical word first**
  - overlap transfer with ability to use data
  - complicates next level access
    - not all that bad
    - at DRAM level
      - MEN_CTL in the way anyway
      - overhead to support is minor

- **Coalesce/Combining/Merge Write Buffer**
  - writes happen from a register value
  - cache lines bigger
    - so buffer writes by cache line
    - same unit of transfer
      - cache to cache
      - cache to DRAM
Combining Write Buffer Example

What are the cost effects?

Reduce Miss Rate

- Compiler optimizations
  - compiler knows cache organization
    » arrange code and data to minimize misses
  - loop interchange – improves spatial locality
    » walk cache line in inner loop
      • e.g. matrix multiply is the canonical example
  - blocking – improves spatial locality
    » put code into phases
      • do as much as you can on this data frame before moving to next frame
      • avoids register spill and refill as well as cache misses
      » matrix multiply again
Matmul Example ijk

/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

- Misses RATIO per Inner Loop Iteration:
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

1/m where m is number of elements in a cache line

Loop Interchange to jik

/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

- Misses per Inner Loop Iteration:
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
**Loop Interchange to kij**

```c
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

• **Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

**All Possibilities**

- Multiple columns of B fit in cache
- (misses/iter = 0.5)
- (misses/iter = 1.25)
- (misses/iter = 2.0)
**Blocked MatMul Example**

**Example:** \( N = 8; \) sub-block size = 4

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= 
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

**Key idea:** Sub-blocks (i.e., \( A_{xy} \)) can be treated just like scalars.

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]

---

**Blocked MatMul Performance**

[Graph showing performance of blocked matrices with sub-block size as a parameter.]

---

School of Computing
University of Utah

Page 9
Others

- **Prefetch**
  - reduces miss penalty and miss rate
    - if done right
    - added complexity, power, and screw up potential
      - discussed last lecture
  - can be done either by HW or SW

- **Next level cache**
  - reduces miss penalty
    - in best case
  - increases miss penalty
    - in worst case
    - “swing to miss” principle

Ancillary Caches

- **Victim cache (Jouppi)**
  - small cache to hold victimized lines
  - idea allows arbitrary associativity for small number of lines
    - total extra associativity = size of victim cache
  - downside
    - parallel check of regular and victim
    - fully associative

- **Trace cache (Weiser, Peleg)**
  - Intel P4
    - expensive – many instruction copies

- **Assist cache (HP and somebody you know)**
  - 1st touch goes to assist
  - 2nd touch goes to regular cache
    - makes prefetch less likely to contaminate cache
  - downside
    - similar to victim cache
## Summary I

<table>
<thead>
<tr>
<th>Technique</th>
<th>Miss Rate</th>
<th>Miss Penalty</th>
<th>Hit Time</th>
<th>HW Complexity</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Cache</td>
<td>win</td>
<td>lose</td>
<td>easy</td>
<td>easy</td>
<td>cost is approx. linear</td>
</tr>
<tr>
<td>Larger Block Size</td>
<td>win</td>
<td>lose</td>
<td>easy</td>
<td>trivial</td>
<td>engineering effort</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>win</td>
<td>lose</td>
<td>1</td>
<td>associative</td>
<td>match isn't free</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>bin</td>
<td></td>
<td>2</td>
<td>e.g. HP 7200</td>
<td></td>
</tr>
<tr>
<td>Pseudo-Associative</td>
<td>win</td>
<td></td>
<td>2</td>
<td>Used in L2 of</td>
<td>MIPS R10000</td>
</tr>
<tr>
<td>HW Prefetch of I and D</td>
<td>win</td>
<td></td>
<td>2</td>
<td>D fetch hard</td>
<td>to do correctly</td>
</tr>
<tr>
<td>Compiler controlled prefetch</td>
<td>win</td>
<td></td>
<td>3</td>
<td>Needs non-</td>
<td>blocking cache</td>
</tr>
<tr>
<td>Compiler cache scheduling (blocking, ...)</td>
<td>win</td>
<td></td>
<td>0</td>
<td>Too bad it's</td>
<td>hard to do for all applications - loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>focus for</td>
<td>now</td>
</tr>
</tbody>
</table>

## Summary II

<table>
<thead>
<tr>
<th>Technique</th>
<th>Miss Rate</th>
<th>Miss Penalty</th>
<th>Hit Time</th>
<th>HW Complexity</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prioritizing Read Misses</td>
<td>win</td>
<td></td>
<td>1</td>
<td></td>
<td>write buffer - simple</td>
</tr>
<tr>
<td>Subblock placement</td>
<td>win</td>
<td></td>
<td>1</td>
<td></td>
<td>good at reducing tag</td>
</tr>
<tr>
<td>Early restart + critical word</td>
<td>win</td>
<td></td>
<td>2</td>
<td></td>
<td>first MIPS R10K, IBM 620</td>
</tr>
<tr>
<td>Nonblocking Caches</td>
<td>win</td>
<td></td>
<td>3</td>
<td>MIPS R10K, AXP</td>
<td>21064</td>
</tr>
<tr>
<td>Second Level Caches</td>
<td>big win</td>
<td></td>
<td>2</td>
<td>big additional</td>
<td>cost for all that SRAM</td>
</tr>
<tr>
<td>Small Simple Caches</td>
<td>lose</td>
<td></td>
<td>0</td>
<td>trivial so</td>
<td>it's widely used</td>
</tr>
<tr>
<td>Avoid translation effects</td>
<td>win</td>
<td></td>
<td>2</td>
<td></td>
<td>AXP 21064 several PA-whatevers</td>
</tr>
<tr>
<td>Pipelining Writes</td>
<td>win</td>
<td></td>
<td>1</td>
<td>AXP 21064, PA-</td>
<td>8000, UltraSPARC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

• Cost
  • focus on HW cost
    » compiler viewed as free if you're a HW geek
  • low
    » small caches, way-prediction, pipelined cache access, banked caches, compiler tactics
  • medium
    » critical word first and early restart
    » instruction prefetch
      • access is more regular
    » victim and assist caches
  • expensive
    » trace caches
      • now that power is a big issue
    » data prefetch
      • access irregular → wasted speculation