JDEP 284H
Foundations of Computer Systems

Cache Memories

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/JDEP284

Giving credit where credit is due

Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
I have modified them and added new slides.

Topics

- Generic cache memory organization
- Direct mapped caches
- Set associative caches
- Impact of caches on performance

Cache Memories

Cache memories are small, fast SRAM-based memories managed automatically in hardware.

Hold frequently accessed blocks of main memory.

CPU looks first for data in L1, then in L2, then in main memory.

Typical bus structure:

![Bus Structure Diagram]

Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte block.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

The tiny, very fast CPU register file has room for four 4-byte words.

The small fast L1 cache has room for two 4-word blocks.

The big slow main memory has room for many 4-word blocks.

General Org of a Cache Memory

Cache is an array of sets.
Each set contains one or more lines.
Each line holds a block of data.

1 valid bit per line
1 tag bit per line
$B = 2^b$ bytes per cache block

$S = 2^s$ sets

$E = 2^e$ lines per set

Cache size: $C = B \times E \times S$ data bytes
Addressing Caches

Address A:

- The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.
- The word contents begin at offset <block offset> bytes from the beginning of the block.

Accessing Direct-Mapped Caches

Set selection:
- Use the set index bits to determine the set of interest.

Line matching and word selection:
- Line matching: Find a valid line in the selected set with a matching tag.
- Word selection: Then extract the word.

Direct-Mapped Cache Simulation

<table>
<thead>
<tr>
<th>Address trace (reads):</th>
<th>0 [0000], 1 [0001], 13 [1011], 8 [1000], 0 [0000]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) v [0000] (miss)</td>
<td>v [0000] data</td>
</tr>
<tr>
<td>(2) v [0001]</td>
<td>v [0001] data</td>
</tr>
<tr>
<td>(3) v [1011] (miss)</td>
<td>v [1011] data</td>
</tr>
<tr>
<td>(4) v [0000]</td>
<td>v [0000] data</td>
</tr>
<tr>
<td>(5) v [0001]</td>
<td>v [0001] data</td>
</tr>
</tbody>
</table>

Why Use Middle Bits as Index?

- High-Order Bit Indexing
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality

- Middle-Order Bit Indexing
  - Can hold C-byte region of address space in cache at one time
**Set Associative Caches**

Characterized by more than one line per set

- **Set 0:**
  - tag
  - cache block
  - valid

- **Set 1:**
  - tag
  - cache block
  - valid

- **Set S-1:**
  - tag
  - cache block
  - valid

\[ E = 2 \] lines per set

**Accessing Set Associative Caches**

**Set selection**

- identical to direct-mapped cache

**Line matching and word selection**

- must compare the tag in each valid line in the selected set.

1. The valid bit must be set.
2. The tag bits in one of the cache lines must match the tag bits in the address.
3. If (1) and (2), then cache hit, and block offset selects starting byte.

**Multi-Level Caches**

Options: separate data and instruction caches, or a unified cache

- Processor
- \( L_1 \) cache
- \( L_2 \) cache
- Unified \( L_2 \) Cache
- Memory
- disk

**Options:**

- size:
  - 200 B
  - 8-64 KB
  - 1-4MB SRAM
  - 128 MB DRAM
  - 30 GB
- speed:
  - 3 ns
  - 6 ns
  - 60 ns
  - 8 ms
- \$/Mbyte:
  - $100/MB
  - $1.50/MB
  - $0.05/MB
- line size:
  - 8 B
  - 32 B
  - 8 KB

**Intel Pentium Cache Hierarchy**

- Processor
- \( L_1 \) cache
- \( L_2 \) cache
- Main Memory

**Cache Performance Metrics**

**Miss Rate**

- Fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for \( L_1 \)
  - can be quite small (e.g., < 1%) for \( L_2 \), depending on size, etc.

**Hit Time**

- Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
  - 1 clock cycle for \( L_1 \)
  - 3-8 clock cycles for \( L_2 \)

**Miss Penalty**

- Additional time required because of a miss
- Typically 25-100 cycles for main memory
Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Examples:

- cold cache, 4-byte words, 4-word cache blocks

\[
\text{int sumarrayrows}(\text{int } a[M][N])
\{
\text{int } i, j, \text{sum} = 0;
\text{for (}i = 0; i < M; i++)
\text{for (}j = 0; j < N; j++)
\text{sum} += a[i][j];
\text{return sum;}
\}
\]

Miss rate = 1/4 = 25%

The Memory Mountain

Read throughput (read bandwidth)

- Number of bytes read from memory per second (MB/s)

Memory mountain

- Measured read throughput as a function of spatial and temporal locality.
- Compact way to characterize memory system performance.

Ridges of Temporal Locality

Slice through the memory mountain with stride=1

Illuminates read throughputs of different caches and memory
A Slope of Spatial Locality

Slice through memory mountain with size=256KB
- Shows cache block size.

Matrix Multiplication Example

Major Cache Effects to Consider
- Total cache size
- Exploit temporal locality and keep the working set small (e.g., by using blocking)
- Block size
- Exploit spatial locality

Description:
- Multiply N x N matrices
- O(N^3) total operations
- Accesses
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

Variable names:
- For (i=0; i<n; i++)
- For (j=0; j<n; j++)
- For (k=0; k<n; k++)

Description:
- Multiply N x N matrices
- O(N^3) total operations
- Accesses
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

Miss Rate Analysis for Matrix Multiply

Assume:
- Line size = 32B (big enough for 4 64-bit words)
- Matrix dimension (N) is very large
- Approximate 1/N as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:
- Look at access pattern of inner loop

Layout of C Arrays in Memory (review)

C arrays allocated in row-major order
- Each row in contiguous memory locations

Stepping through columns in one row:
- For (i = 0; i < N; i++)
- Accesses successive elements
- If block size (B) > 4 bytes, exploit spatial locality
- Compulsory miss rate = 4 bytes / B

Stepping through rows in one column:
- For (j = 0; j < N; j++)
- Accesses distant elements
- No spatial locality!
- Compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (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;
    }
}
```

```
/* 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;
    }
}
```

Matrix Multiplication (ijk)

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

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

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>
Matrix Multiplication (kij)

```
/* 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:

```
                          A  B  C
Fixed            0.0  0.25  0.25
Row-wise  Row-wise
```

---

Matrix Multiplication (ikj)

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

Misses per Inner Loop Iteration:

```
                          A  B  C
Fixed            0.0  0.25  0.25
Row-wise  Row-wise
```

---

Matrix Multiplication (jki)

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

Misses per Inner Loop Iteration:

```
                          A  B  C
Column-wise  Fixed  Column-wise
```

---

Matrix Multiplication (kji)

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

Misses per Inner Loop Iteration:

```
                          A  B  C
Column-wise  Fixed  Column-wise
```

---

Summary of Matrix Multiplication

<table>
<thead>
<tr>
<th>ijk (&amp; jik):</th>
<th>kij (&amp; ikj):</th>
<th>jki (&amp; kji):</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 loads, 0 stores</td>
<td>2 loads, 1 store</td>
<td>2 loads, 1 store</td>
</tr>
<tr>
<td>misses/iter = 1.25</td>
<td>misses/iter = 0.5</td>
<td>misses/iter = 2.0</td>
</tr>
</tbody>
</table>

---

Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

* Code scheduling matters, too.
Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- "block" (in this context) does not mean "cache block".
- Instead, it means a sub-block within the matrix.
- 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 Matrix Multiply Analysis

- Innermost loop pair multiplies a \( 1 \times \text{bsize} \) sliver of \( A \) by a \( \text{bsize} \times \text{bsize} \) block of \( B \) and accumulates into a \( 1 \times \text{bsize} \) sliver of \( C \)
- Loop over \( i \) steps through \( n \) row slivers of \( A \) & \( C \), using same \( B \)

```
for (i=0; i<n; i++)
for (j=jj; j<min(jj+bsize,n); j++)
    sum = a[i][k] * b[k][j];
    c[i][j] += sum;
```

Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- relatively insensitive to array size.

Concluding Observations

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
- Nested loop structure
- Blocking is a general technique

All systems favor "cache friendly code"

- Getting absolute optimum performance is very platform specific
- Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
- Keep working set reasonably small (temporal locality)
- Use small strides (spatial locality)