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.

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

Typical bus structure:

- CPU chip
- Cache bus
- System bus
- Memory bus
- I/O bridge
- Main memory
- L2 cache

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 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 flag bit per line
- \( B \times 2^t \) bytes per cache block
- \( S = 2^t \) sets
- \( C = B \times E \times S \) data bytes
Addressing Caches

Address A:  
1 bits  
$s$ bits  
b bits

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.

Direct-Mapped Cache Simulation

<table>
<thead>
<tr>
<th>Mem16 byte addresses, B=2 bytes/block, 512 sets, 16 entries/set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address trace (reads): 0 [1000], 1 [0001], 13 [1101], 8 [1000], 0 [0000]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0 [1000] (miss)</th>
<th>1 [0001] (hit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v tag data</td>
<td>v tag data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8 [1000] (miss)</th>
<th>0 [0000] (miss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v tag data</td>
<td>v tag data</td>
</tr>
</tbody>
</table>

Why Use Middle Bits as Index?

4-line Cache

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

High-Order Bit Indexing

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing

- Consecutive memory lines map to different cache lines
- 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 selection
- identical to direct-mapped cache

Accessing Set Associative Caches

Line matching and word selection
- must compare the tag in each valid line in the selected set.

Multi-Level Caches

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

Intel Pentium Cache Hierarchy

Processor

Main Memory

Cache Performance Metrics

Miss Rate
- Fraction of memory references not found in cache (misses/ references)
- Typical numbers:
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, 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 L1
  - 3-8 clock cycles for L2

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

```c
int sumarray(row(int a[M][N]) { int i, j, sum = 0; for (i = 0; i < M; i++) for (j = 0; j < N; j++) sum += a[i][j]; return sum; }
```

- repeated references to variables are good (temporal locality)

```c
void test(int elem, int stride) { double cycle, cycles = 0; int size = elem; int stride, double Mhz; double cycles; int elem = size / sizeof(int); cycle = 0; for (elem = elem; elem > 0; elem /= sizeof(int)); cycle = cycle / sizeof(int); cycles = cycles + cycle * (elem, stride) / (cycles / Mhz); return (size / stride) / cycles; } /* warm up the cache */
```

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.

```c
int main() { int size, double cycle = 0; /* Working set size ranges from 1 KB */ for (size = 1; size > 0; size /= sizeof(int)); /* Double Mhz */ /* Clock frequency */ init_data(size, cycle); /* Initialize each element in data to 1 */ for (size = cycle; size > 0; size /= sizeof(int)); /* Estimate the clock frequency */ for (size = cycle; size > 0; size /= sizeof(int)); /* Emit the data */ printf("time=%d", time); /* Output the results */ exit(0); }
```

Memory Mountain Test Function

```c
int sumarray(int a[M][N]) { int i, j, sum = 0; for (i = 0; i < M; i++) for (j = 0; j < N; j++) sum += a[i][j]; return sum; }
```

Memory Mountain Main Routine

```c
int main() { int size, double cycle, double Mhz = 550; /* Working set size ranges from 1 KB */ for (size = 1; size > 0; size /= sizeof(int)); /* Double Mhz */ /* Clock frequency */ init_data(size, cycle); /* Initialize each element in data to 1 */ for (size = cycle; size > 0; size /= sizeof(int)); /* Estimate the clock frequency */ for (size = cycle; size > 0; size /= sizeof(int)); /* Emit the data */ printf("time=%d", time); /* Output the results */ exit(0); }
```

Ridges of Temporal Locality

Slice through the memory mountain with stride=1

- Illuminates read throughputs of different caches and memory

```c
int main() { int size, double cycle, double Mhz = 550; /* Working set size ranges from 1 KB */ for (size = 1; size > 0; size /= sizeof(int)); /* Double Mhz */ /* Clock frequency */ init_data(size, cycle); /* Initialize each element in data to 1 */ for (size = cycle; size > 0; size /= sizeof(int)); /* Estimate the clock frequency */ for (size = cycle; size > 0; size /= sizeof(int)); /* Emit the data */ printf("time=%d", time); /* Output the results */ exit(0); }
```
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

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

Matrix Multiplication (ijk)

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

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++)
  - sum = a[i][j];
  - 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++)
  - sum = a[i][j];
  - accesses distant elements
  - no spatial locality!
  - compulsory miss rate = 1 (i.e., 100%)
Matrix Multiplication (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>ijk (kij)</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Matrix Multiplication (ikj)

```c
/* 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:**

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

Matrix Multiplication (jki)

```c
/* 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:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>jki (jki)</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Matrix Multiplication (kji)

```c
/* 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:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>kji (kji)</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Summary of Matrix Multiplication

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

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

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

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21}
C_{12} = A_{11}B_{12} + A_{12}B_{22}
C_{21} = A_{21}B_{11} + A_{22}B_{21}
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]

Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a \( b \times b \) size slice of \( A \) by a \( b \times b \) size block of \( B \) and accumulates into a \( b \times b \) size slice of \( C \)
- Loop over steps through \( n \) row slices of \( A \) & \( C \), using same \( B \)

\[
\text{for (ii = 0; ii < min(jj,bSize); ii++)}
\]
\[
\text{for (jj = 0; jj < min(ii,bSize); jj++)}
\]
\[
\text{for (kk = 0; kk < n; kk++)}
\]
\[
\text{for (ii = 0; ii < min(jj,bSize); ii++)}
\]

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)