15-213
“The course that gives CMU its Zip!”

Cache Memories
October 7, 2004

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 system structure:
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 Organization of a Cache

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

\[ S = 2^s \text{ sets} \]

\[ \uparrow \text{ tag bits per line} \]

\[ B = 2^b \text{ bytes per cache block} \]

\[ \text{E lines per set} \]

\[ \text{1 valid bit per line} \]

\[ \text{Cache size: } C = B \times E \times S \text{ data bytes} \]
Addressing Caches

Address A:

\[ \text{t bits} \quad \text{s bits} \quad \text{b bits} \]

\[ \text{m-1} \quad \text{0} \]

\[ \text{<tag>} \quad \text{<set index>} \quad \text{<block offset>} \]

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.
Addressing Caches

Address A:

```
\[ \begin{array}{ccc}
\text{t bits} & \text{s bits} & \text{b bits} \\
\hline
m-1 & & 0 \\
\hline
\langle \text{tag} \rangle & \langle \text{set index} \rangle & \langle \text{block offset} \rangle \\
\end{array} \]
```

1. Locate the set based on \[ \langle \text{set index} \rangle \]
2. Locate the line in the set based on \[ \langle \text{tag} \rangle \]
3. Check that the line is valid
4. Locate the data in the line based on \[ \langle \text{block offset} \rangle \]
Direct-Mapped Cache

Simplest kind of cache, easy to build
(only 1 tag compare required per access)

Characterized by exactly one line per set.

set 0: valid   tag   cache block
set 1: valid   tag   cache block
      ...       
set S-1: valid   tag   cache block

Cache size: \( C = B \times S \) data bytes
Accessing Direct-Mapped Caches

Set selection

- Use the set index bits to determine the set of interest.

selected set

<table>
<thead>
<tr>
<th>set 0:</th>
<th>valid</th>
<th>tag</th>
<th>cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>set 1:</td>
<td>valid</td>
<td>tag</td>
<td>cache block</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set S-1:</td>
<td>valid</td>
<td>tag</td>
<td>cache block</td>
</tr>
</tbody>
</table>

\[ \text{tag} \quad \text{set index} \quad \text{block offset} \]

\[ m^{-1} \quad 000001 \quad 0 \]
Accessing Direct-Mapped Caches

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

0110

= 1? (1) The valid bit must be set

= ?

If (1) and (2), then cache hit

(2) The tag bits in the cache line must match the tag bits in the address

0110  i  100

m-1 tag set index block offset
Accessing Direct-Mapped Caches

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

(selected set (i):)

(3) If cache hit, block offset selects starting byte.

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>i</td>
<td>100</td>
</tr>
</tbody>
</table>

m⁻¹ tag set index block offset
Direct-Mapped Cache Simulation

\( t=1 \quad s=2 \quad b=1 \)

\[ \begin{array}{ccc}
\times & \times & x
\end{array} \]

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 entry/set

Address trace (reads):

<table>
<thead>
<tr>
<th>( v )</th>
<th>( [0000_2] ), miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0001_2], hit</td>
</tr>
<tr>
<td>1</td>
<td>[0111_2], miss</td>
</tr>
<tr>
<td>7</td>
<td>[1000_2], miss</td>
</tr>
<tr>
<td>8</td>
<td>[0000_2]</td>
</tr>
</tbody>
</table>

\[\begin{array}{cccc}
& v & \text{tag} & \text{data} \\
\hline
& 1 & 0 & M[0-1] \\
& 1 & 0 & M[6-7] \\
\end{array}\]
Set Associative Caches

Characterized by more than one line per set

<table>
<thead>
<tr>
<th>set 0:</th>
<th>valid</th>
<th>tag</th>
<th>cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>valid</td>
<td>tag</td>
<td>cache block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>set 1:</th>
<th>valid</th>
<th>tag</th>
<th>cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>valid</td>
<td>tag</td>
<td>cache block</td>
</tr>
</tbody>
</table>

... E-way associative cache

E=2 lines per set
Accessing Set Associative Caches

Set selection

- identical to direct-mapped cache

```
<table>
<thead>
<tr>
<th>set 0:</th>
<th>set 1:</th>
<th>set S-1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid</td>
<td>tag</td>
<td>cache block</td>
</tr>
<tr>
<td>valid</td>
<td>tag</td>
<td>cache block</td>
</tr>
<tr>
<td>valid</td>
<td>tag</td>
<td>cache block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cache block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cache block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cache block</td>
</tr>
</tbody>
</table>

↑ bits s bits b bits

0 0 0 0 1
```

- m-1 tag set index block offset 0
Accessing Set Associative Caches

Line matching and word selection

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

=1? (1) The valid bit must be set

= ?

If (1) and (2), then cache hit

(2) The tag bits in one of the cache lines must match the tag bits in the address

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>i</td>
<td>100</td>
</tr>
</tbody>
</table>

m-1 tag | set index | block offset

- 14 -
Accessing Set Associative Caches

Line matching and word selection

- Word selection is the same as in a direct mapped cache

(3) If cache hit, block offset selects starting byte.

\[
\begin{array}{c|c|c|c}
\text{t bits} & \text{s bits} & \text{b bits} \\
0110 & i & 100 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{m-1 tag} & \text{set index} & \text{block offset}^0 \\
0110 & i & 100 \\
\end{array}
\]
2-Way Associative Cache Simulation

$t=2$, $s=1$, $b=1$

$M=16$ byte addresses, $B=2$ bytes/block,
$S=2$ sets, $E=2$ entry/set

Address trace (reads):

<table>
<thead>
<tr>
<th>$v$</th>
<th>$tag$</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>$M[0-1]$</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>$M[8-9]$</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>$M[6-7]$</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why Use Middle Bits as Index?

4-line Cache

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></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 S*B*E-byte region of address space in cache at one time
Maintaining a Set-Associate Cache

- **How to decide which cache line to use in a set?**
  - Least Recently Used (LRU), Requires $\lceil \lg_2(E) \rceil$ extra bits
  - Not recently Used (NRU)
  - Random

- **Virtual vs. Physical addresses:**
  - The memory system works with physical addresses, but it takes time to translate a virtual to a physical address. So most L1 caches are virtually indexed, but physically tagged.
Multi-Level Caches

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

- Processor
  - Regs
  - L1 d-cache
  - L1 i-cache

- Unified L2 Cache

- Memory

- disk

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Speed</th>
<th>Price/MB</th>
<th>Line Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regs</td>
<td>200 B</td>
<td>3 ns</td>
<td>$100/MB</td>
<td>8 B</td>
<td></td>
</tr>
<tr>
<td>L1 d-cache</td>
<td>8-64 KB</td>
<td>3 ns</td>
<td>$1.50/MB</td>
<td>32 B</td>
<td></td>
</tr>
<tr>
<td>L1 i-cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>1-4 MB SRAM</td>
<td>6 ns</td>
<td>$100/MB</td>
<td>32 B</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>128 MB DRAM</td>
<td>60 ns</td>
<td>$1.50/MB</td>
<td>8 KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30 GB</td>
<td>8 ms</td>
<td>$0.05/MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

larger, slower, cheaper
What about writes?

Multiple copies of data exist:
- L1
- L2
- Main Memory
- Disk

What to do when we write?
- Write-through
- Write-back
  - need a dirty bit
  - What to do on a write-miss?

What to do on a replacement?
- Depends on whether it is write through or write back
Intel Pentium III Cache Hierarchy

- **Reg.**
  - L1 Data
    - 1 cycle latency
    - 16 KB
    - 4-way assoc
    - Write-through
    - 32B lines

- **Processor Chip**
  - L1 Instruction
    - 16 KB, 4-way
    - 32B lines

- **L2 Unified**
  - 128KB--2 MB
  - 4-way assoc
  - Write-back
  - Write allocate
  - 32B lines

- **Main Memory**
  - Up to 4GB
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:
  - 1-2 clock cycle for L1
  - 5-20 clock cycles for L2

Miss Penalty
- Additional time required because of a miss
  - Typically 50-200 cycles for main memory (Trend: increasing!)

 Aside for architects:
- Increasing cache size?
- Increasing block size?
- Increasing associativity?
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 sum_array_rows(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;
}
```

**Miss rate = 1/4 = 25%**

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

**Miss rate = 100%**
Detecting the Cache Parameters

How can one determine the cache parameters?

- Size of cache?
- Size of cache block?
- Hit time?
- Miss penalty?
- Associatively?
- Number of levels in memory hierarchy?

Complicating factors

- Prefetch support (hardware and software)
- Non-blocking caches ("Hit-under-Miss" support)
- Superscalar processors with multiple, concurrent memory operations
- Victim caches, stream buffers, line-reservation
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.
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;
    
    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);
    
    test(elems, stride); /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23) /* ... up to 8 MB */
#define MAXSTRIDE 16 /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS]; /* The array we'll be traversing */

int main()
{
    int size; /* Working set size (in bytes) */
    int stride; /* Stride (in array elements) */
    double Mhz; /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0); /* Estimate the clock frequency */
    for (size = MAXBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
The Memory Mountain

Pentium III
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

Slopes of Spatial Locality

Ridges of Temporal Locality

Throughput (MB/sec)

Stride (words)

Working set size (bytes)

mem

L1

L2
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.

![Bar chart showing read throughput (MB/s) vs. stride (words)]

- one access per cache line
Matrix Multiplication Example

Major Cache Effects to Consider

- Total cache size
  - Exploit temporal locality and keep the working set small (e.g., use 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

```c
/* 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;
    }
}
```

Variable `sum` held in register
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for four 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++)
  sum += a[0][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 (i = 0; i < n; i++)
  sum += a[i][0];

- 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;
    }
}

Misses per Inner Loop Iteration:
\[\begin{array}{ccc}
A & B & C \\
0.25 & 1.0 & 0.0 \\
\end{array}\]
Matrix Multiplication (jik)

```c
/* 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></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>

**Inner loop:**

- **Row-wise**
- **Column-wise**
- **Fixed**
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></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
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];
    }
}

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

Inner loop:

Fixed  Row-wise  Row-wise

A       B       C
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></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>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
Summary of Matrix Multiplication

```c
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;
    }
}
```

**ijk (& jik):**
- 2 loads, 0 stores
- misses/iter = 1.25

```c
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];
    }
}
```

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = 0.5

```c
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;
    }
}
```

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = 2.0
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 mean 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.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
Blocked Matrix Multiply (bijk)

for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;

    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++) {
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
        }
    }
}
Blocked Matrix Multiply Analysis

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

```c
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
    }
}
```

Innermost Loop Pair

- Row sliver accessed $bsize$ times
- Block reused $n$ times in succession
- Update successive elements of sliver
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.

![Graph showing performance comparison between blocked and unblocked matrix multiplication](image)
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)