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 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.

$S = 2^S$ sets

$1$ valid bit per line

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

+ tag bits per line

$B = 2^b$ bytes per cache block

$E$ lines per set

set 0:

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>B-1</th>
</tr>
</thead>
</table>

set 1:

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>B-1</th>
</tr>
</thead>
</table>

...
Addressing Caches

Address A:

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

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.

1. Locate the set based on <set index>
2. Locate the line in the set based on <tag>
3. Check that the line is valid
4. Locate the data in the line based on <block offset>
**Direct-Mapped Cache**

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

Characterized by exactly one line per set.

\[
\begin{align*}
\text{set 0} & & \text{valid} & \text{tag} & \text{cache block} \\
\text{set 1} & & \text{valid} & \text{tag} & \text{cache block} \\
\text{set } S-1 & & \text{valid} & \text{tag} & \text{cache block} \\
\end{align*}
\]

\( E \) lines per set

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.
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

= ? (1) The valid bit must be set

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

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

---

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

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

---

Page 5
Direct-Mapped Cache Simulation

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

t=1 s=2 b=1

Address trace (reads):
0 [0000₂], miss
1 [0001₂], hit
7 [0111₂], miss
8 [1000₂], miss
0 [0000₂] miss

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>

Set Associative Caches

Characterized by more than one line per set

set 0:
valid | tag | cache block
valid | tag | cache block

set 1:
valid | tag | cache block
valid | tag | cache block

set S-1:
valid | tag | cache block
valid | tag | cache block

E-way associative cache

E=2 lines per set
Accessing Set Associative Caches

Set selection
- identical to direct-mapped cache

```
+----+----+----+
|set 0: | valid | tag | cache block |
|      | valid | tag | cache block |
|      | valid | tag | cache block |
|      |       |    |            |
```

selected set
- set 1:

```
+----+----+----+
|set 1: | valid | tag | cache block |
|       | valid | tag | cache block |
|       | valid | tag | cache block |
|       |       |    |            |
```

- set S-1:

```
+----+----+----+
|set S-1: | valid | tag | cache block |
|         | valid | tag | cache block |
|         | valid | tag | cache block |
|         |       |    |            |
```

```
t bits s bits b bits
0 0 0 0 1
```

Accessing Set Associative Caches

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

```
+----+----+----+----+----+----+----+
|selected set (i): | 1001 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|0110 | 1 |
```

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

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

```
t bits s bits b bits
0110 | 1 | 100
```

- 14 -
Accessing Set Associative Caches

Line matching and word selection
- Word selection is the same as in a direct mapped cache

<table>
<thead>
<tr>
<th>selected set (i):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

(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

2-Way Associative Cache Simulation

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

Address trace (reads):
0 [00002], miss
1 [00012], hit
7 [01112], miss
8 [10002], miss
0 [00002] hit

<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?

<table>
<thead>
<tr>
<th>4-line Cache</th>
<th>High-Order Bit Indexing</th>
<th>Middle-Order Bit Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>01</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>10</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>11</td>
<td>0011</td>
<td>0011</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 \( \lceil \log_2(E) \rceil \) 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

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Speed</th>
<th>$/MB</th>
<th>line size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regs</td>
<td>200 B</td>
<td>3 ns</td>
<td>$100/MB</td>
<td>8 B</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>
</tr>
<tr>
<td>L1 i-cache</td>
<td>1-4MB</td>
<td>6 ns</td>
<td>$0.05/MB</td>
<td>32 B</td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>128 MB</td>
<td>60 ns</td>
<td></td>
<td>8 KB</td>
</tr>
<tr>
<td>DRAM</td>
<td>30 GB</td>
<td>8 ms</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

- 21 -

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;
}
```

```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 = \(\frac{1}{4} = 25\%\)

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, linereservation
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.

Memory Mountain Test Function

/* 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 */
}
Memory Mountain Main Routine

```c
/* 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 > MINBYTES; 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
Working set size (bytes)
Stride (words)
Throughput (MB/sec)
L1
L2
mem
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., 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

```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;
    }
}
```

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)**

```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;
    }
}
```

**Misses per Inner Loop Iteration:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.25</td>
<td>B</td>
</tr>
</tbody>
</table>
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
    }
}
```

Inner loop:

Row-wise  Column-wise  Fixed

```
Misses per Inner Loop Iteration:
A  B  C
0.25 1.0 0.0
```

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];
    }
}
```

Inner loop:

```
Misses per Inner Loop Iteration:
A  B  C
0.0 0.25 0.25
```
Matrix Multiplication (ijk)

```c
/* ijk */
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];
    }
}
```

Inner loop:

- Fixed
- Row-wise
- Row-wise

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<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;
    }
}
```

Inner loop:

- Column-wise
- Fixed
- Column-wise

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<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>
</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></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Summary of Matrix Multiplication

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

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

- **jki** (i & ki):
  - 2 loads, 1 store
  - misses/iter = 2.0
**Pentium Matrix Multiply Performance**

Miss rates are helpful but not perfect predictors.
- Code scheduling matters, too.

![Graph showing performance comparison of different matrix multiplication methods](image)

**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

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

**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 (bijk)**

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

---

**Blocked Matrix Multiply Analysis**

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

![Diagram showing loop pairs and access pattern]

---

Page 22
**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)