Cache Memories
Oct. 2, 2003

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

15-213
“The course that gives CMU its Zip!”
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 \textit{L1 cache} has room for two 4-word blocks.

The big slow \textit{main memory} has room for many 4-word blocks.

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

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

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

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

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

\[ \begin{array}{c|c|c|c|c|c} \text{valid} & \text{tag} & 0 & 1 & \cdots & B-1 \\ \hline \end{array} \]

\[ \text{set 0:} \]

\[ \text{set 1:} \]

\[ \text{set S-1:} \]

\[ \begin{array}{c|c|c|c|c|c} \text{valid} & \text{tag} & 0 & 1 & \cdots & B-1 \\ \hline \end{array} \]

\[ \begin{array}{c|c|c|c|c|c} \text{valid} & \text{tag} & 0 & 1 & \cdots & B-1 \\ \hline \end{array} \]

\[ \begin{array}{c|c|c|c|c|c} \text{valid} & \text{tag} & 0 & 1 & \cdots & B-1 \\ \hline \end{array} \]

\[ \begin{array}{c|c|c|c|c|c} \text{valid} & \text{tag} & 0 & 1 & \cdots & B-1 \\ \hline \end{array} \]

\[ \begin{array}{c|c|c|c|c|c} \text{valid} & \text{tag} & 0 & 1 & \cdots & B-1 \\ \hline \end{array} \]

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

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

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

Address A:

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

- \( t \) bits
- \( s \) bits
- \( b \) bits

The word at address A is in the cache if the tag bits in one of the \(<\text{valid}>\) lines in set \(<\text{set index}>\) match \(<\text{tag}>\).

The word contents begin at offset \(<\text{block offset}>\) bytes from the beginning of the block.
### Addressing Caches

#### Address A:

- **t** bits
- **s** bits
- **b** bits

- **m-1**
- **<tag>**
- **<set index>**
- **<block offset>**

#### Locating the Cache Line:

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.

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

E=1 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.

![Diagram of Direct-Mapped Caches](image-url)
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):

1 0110

=1? (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

<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>
```

```plaintext
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

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

```
0110     i    100
```

- **t bits**
- **s bits**
- **b bits**

- **tag**
- **set index**
- **block offset**
Direct-Mapped Cache Simulation

\[ M = 16 \text{ byte addresses, } B = 2 \text{ bytes/block, } S = 4 \text{ sets, } E = 1 \text{ entry/set} \]

Address trace (reads):

\[
\begin{array}{ll}
0 & [0000_2], \text{ miss} \\
1 & [0001_2], \text{ hit} \\
7 & [0111_2], \text{ miss} \\
8 & [1000_2], \text{ miss} \\
0 & [0000_2], \text{ miss} \\
\end{array}
\]

\[
\begin{array}{ccc}
v & \text{tag} & \text{data} \\
1 & 0 & M[0-1] \\
\hline
\end{array}
\]

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

Characterized by more than one line per set

E-way associative cache
Accessing Set Associative Caches

Set selection

- identical to direct-mapped cache

```
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
```

\[ \begin{array}{c}
\text{\& bits} \\
\text{\# bits} \\
\text{\textbf{0 0 0 0 1}} \\
\end{array} \]
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

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

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

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

<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^0
2-Way Associative Cache Simulation

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

t=2  s=1  b=1

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?

### 4-line Cache

| 00 | 01 | 10 | 11 |

### 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

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

Unified L2 Cache

Memory

disk

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

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

- **Regs.**
  - 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;
}
```

```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 = 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, 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.
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

/* 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);
}
Ridges of Temporal Locality

Slice through the memory mountain with stride=1

- illuminates read throughputs of different caches and memory

![Bar chart showing read throughput (MB/s) against working set size (bytes) for main memory, L2 cache, and L1 cache regions. The x-axis represents working set sizes ranging from 8m to 1k, and the y-axis represents read throughput in MB/s. The chart highlights the performance differences across different cache and memory regions for varying working set sizes.](chart.png)
A Slope of Spatial Locality

Slice through memory mountain with size=256KB

- shows cache block size.

![Graph 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 $N \times N$ 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;
    }
}
```
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;
  }
}
```
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:**

- **A**: 0.25
- **B**: 1.0
- **C**: 0.0

![Diagram](image)
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:

<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>

Inner loop:

(i,k) (k,*) (i,*)

Fixed 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:

<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>

Inner loop:

Fixed Row-wise Row-wise
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)

/* 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>

Inner loop:
- (\*,k)
- (k,j)
- (\*,j)
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;
    }
}

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];
    }
}

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;
    }
}
```

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

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

**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)

```c
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 $1 \times \text{bsize}$ sliver of $A$ by a $\text{bsize} \times \text{bsize}$ block of $B$ and accumulates into $1 \times \text{bsize}$ sliver of $C$
- Loop over $i$ steps through $n$ row slivers of $A$ & $C$, using same $B$

```java
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 $\text{bsize}$ 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 versions]

- Cycles/iteration vs. Array size (n)
- Different line styles represent different blocked versions (bijk, bikj, ijk, ikj, ki, kj, ikj, jik)

---

Array size (n)  Cycles/iteration

- 25  5  
- 50  10  
- 75  15  
- 100  20  
- 125  25  
- 150  30  
- 175  35  
- 200  40  
- 225  45  
- 250  50  
- 275  55  
- 300  60  
- 325  65  
- 350  70  
- 375  75  
- 400  80  

- bikj (bsize = 25)
- bikj (bsize = 25)
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)