Cache Memories
Oct. 2, 2003

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

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 four 4-byte words.

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.

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.

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

Accessing Direct-Mapped Caches

Cache size: $C = B \times S$ data bytes
**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? (1) The valid bit must be set

---

**Direct-Mapped Cache Simulation**

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

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

<table>
<thead>
<tr>
<th>t</th>
<th>s</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Set Associative Caches**

Characterized by more than one line per set

---
Accessing Set Associative Caches

Set selection
- identical to direct-mapped cache

selected set
- set 0:
  - valid
tag
  - cache block

- set 1:
  - valid
tag
  - cache block

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

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

\[ \text{m-1} \quad \text{tag} \quad \text{set index} \quad \text{block offset} \]

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

\[ \text{selected set (i)}: \]

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

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

Word selection is the same as in a direct mapped cache

2-Way Associative Cache Simulation

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

Address trace (reads):

\[
\begin{array}{cccc}
0 & [0000_2] & \text{miss} \\
1 & [0001_2] & \text{hit} \\
7 & [0111_2] & \text{miss} \\
8 & [1000_2] & \text{miss} \\
0 & [0000_2] & \text{hit} \\
\end{array}
\]
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
- Consecutive memory lines map to different cache lines
- Can hold $S\times B\times E$-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

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**

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

- **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!)

---

**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, 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 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);
}

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);
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}

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)
Ridges of Temporal Locality

Slice through the memory mountain with stride=1

- illuminates read throughputs of different caches and memory

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

```
/* 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:**

<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 (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>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)**

```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>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 (ijk)

```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></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></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></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 (ikj):</th>
<th>kij (ikj):</th>
<th>jki (ikj):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 loads, 0 stores</td>
<td>2 loads, 1 store</td>
<td>2 loads, 1 store</td>
</tr>
<tr>
<td></td>
<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.

![Graph showing cycle/iteration vs. array size (n) for different matrix operations: kji & kji, kij & ikj, jik & ijk, and ijk & ijk.]

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

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

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

Innermost Loop Pair

- Row sliver accessed \(\text{bsize} \times \text{bsize}\) times.
- Block reused \(n\) times in succession.
**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 improvement](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)