15-213
"The course that gives CMU its Zip!"

Cache Memories
February 26, 2008

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

Cache Memories

Cache memories are small, fast SRAM-based memories.
- Hold frequently accessed blocks of main memory

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

Typical system structure:

- CPU chip
- Register file
- I/O bus
- Memory bus
- System bus
- Main memory
- Bus interface
- L2 cache
- L1 cache
- Main memory

Synchronization

First exam this evening
- If you have not received mail with a Subject line like “15-213 exam: conflict session C2” then we expect you at the main exam session
- Room split by Andrew username (not first/last/middle name!)
- a-c Wean 7500
- d-e McConomy Auditorium in University Center
- Bring with you
  - Your TA’s name and/or 15-213 section letter
  - Book and notes, if you wish
  - Suggested: know your powers of 2
  - No calculators

Synchronization - 2

Computer Club movie night
- "Colossus, The Forbin Project"
- Wednesday evening
- Wean 7500
- 19:00 Computer Club Intro, co-op pizza order
- 19:30 Movie

Determinant

Theorem 5. If A and B are square matrices of the same size, then det(AB) = det(A) * det(B).

The elegant simplicity of this result contrasted with the complex nature of both matrix multiplication and the determinant definition is both refreshing and surprising. We shall omit the proof.


Inserting an L1 Cache Between the CPU and Main Memory

- The tiny, very fast CPU 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.

$S = 2^n$ sets

$B = 2^m$ bytes per block

$E = 2^k$ bits per set

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

Addressing Caches

Addressing Caches

Address $A$:

$A = b_0 b_1 \ldots b_{E-1} a_0 a_1 \ldots a_{e-1}$

$A_0 \ldots A_2 = a_0 a_1 a_2$

$t$ bits

$s$ bits

$b$ bits

Set selection

1. The word at address $A$ is in the cache if the tag bits in one of the valid lines in set $s$ match the tag.
2. The word contents begin at offset $b$ from the beginning of the block.

Line matching and word selection

1. Look up the set based on $s$.
2. Look up the line in the set based on $t$.
3. Check that the line is valid.
4. Look up the data in the line based on $b$.

Direct-Mapped Cache

Simplest kind of cache, easy to build

Characterized by exactly one line per set.

$C = B \times S$ data bytes

Accessing Direct-Mapped Caches

Set selection

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

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.

$E = 2^k$ bits
Accessing Set Associative 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.

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.

Set selection:
- Word selection is the same as in a direct-mapped cache.
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 replacement?
- Depends on whether it is write through or write back
Cache Performance Metrics

**Miss Rate**
- Fraction of memory references not found in cache (re 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 cycles 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)

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

```c
#include <stdio.h>

#define N 1024
#define M 1024
#define MAXELEMS MAXBYTES/sizeof(int)
#define MAXSTRIDE 16
#define MAXBYTES (1 << 23)
define HRANGE 12
#define MINBYTES (1 << 10)

int main()
{
  volatile int sink;
  int i, j, sum = 0;
  for (j = 0; j < N; j++)
  {
    for (i = 0; i < M; i++)
    {
      result += data[i];
    }
    sum += a[i][j];
  }
  return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

Writing Cache Friendly Code

**Repeated references to variables are good**

(temporal locality)
- Stride-1 reference patterns are good (**temporal locality**)

**Examples:**
- Stride-1 reference patterns are good (temporal locality)

```c
int data[MAXELEMS];
#define MAXELEMS MAXBYTES/sizeof(int)
#define MAXSTRIDE 16
#define MAXBYTES (1 << 23)
define HRANGE 12
#define MINBYTES (1 << 10)
```

Memory Mountain Test Function

```c
int test(int elems, int stride)
{ int i, result = 0;
  volatile int sink;
  for (i = 0; i < elems; i += stride)
  { sum += a[i][j];
  }
  return sum;
}
```

The Memory Mountain

```c
#include <stdio.h>

#define N 1024
#define M 1024
#define MAXELEMS MAXBYTES/sizeof(int)
#define MAXSTRIDE 16
#define MAXBYTES (1 << 23)
define HRANGE 12
#define MINBYTES (1 << 10)

int main()
{
  volatile int sink;
  int i, j, sum = 0;
  for (j = 0; j < N; j++)
  {
    for (i = 0; i < M; i++)
    {
      result += data[i];
    }
    sum += a[i][j];
  }
  return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```
### X86-64 Memory Mountain

![X86-64 Memory Mountain](image1.png)

- **Pentium Nocona**: 2.4 GHz 12 MB on-chip L2 trace cache 16 KB on-chip L1 d-cache 64 KB on-chip unified L1 cache

### Opteron Memory Mountain

![Opteron Memory Mountain](image2.png)

- **AMD Opteron**: 2.2 GHz

### 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(P) total operations
  - Accesses
  - N reads per source element
  - N values summed per destination
  - But may be able to hold in registers

```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

![Miss Rate Analysis](image3.png)
**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++`

- 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++`

- 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:**

<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)**

```
/* 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)**

```
/* 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 (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>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)**

```
/* 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>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)

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

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>

Example: Blocked matrix multiplication

Example: N = 8; sub-block size = 4

Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    c[i][j] = sum;
    sum += a[i][k] * b[k][j];
  }
}
```

Code scheduling matters, too.

Summary of Matrix Multiplication

<table>
<thead>
<tr>
<th>ij (&amp; ik):</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 loads, 0 stores</td>
</tr>
<tr>
<td>misses/iter = 1.25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jk (&amp; ikj):</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 loads, 1 store</td>
</tr>
<tr>
<td>misses/iter = 0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jk (&amp; kji):</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 loads, 1 store</td>
</tr>
<tr>
<td>misses/iter = 2.0</td>
</tr>
</tbody>
</table>

Equation Rewriting

The math:

\[ C_{ij} = A_{ik} \times B_{kj} \]

For (i,j,k) = A[i][j][k]

Straightforward conversion to imperative code:

```plaintext
C = 0
C_{ij} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
```

Re-order the code to get more cache hits:

```plaintext
C = 0
C_{ij} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
C_{ijk} = A_{ik} \times B_{kj}
```

We use \( B_{kj} \) twice (with \( A_{ik} \)), then \( B_{kj} \) twice...

We can fit 1 \( B_{kj} \) in the cache no matter how big the matrices get

15-213, S’08

43-48
Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a 1 x bsize sliver of A by a bsize x bsize block of B and accumulates into 1 x 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;
    }
}
```

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)