15-213
Memory System Performance
March 21, 2000

Topics
• Impact of cache parameters
• Impact of memory reference patterns
  - memory mountain range
  - matrix multiply

Basic Cache Organization
Address space \( (N = 2^n \text{ bytes}) \)
Cache \( (C = S \times E \times B \text{ bytes}) \)

\[
\begin{align*}
\text{Address} & \quad (n = t + s + b \text{ bits}) \\
\text{Valid bit} & \quad \text{Tag} & \quad \text{Data} \\
1 \text{ bit} & \quad t \text{ bits} & \quad B = 2^b \text{ bytes (line size)} \\
\end{align*}
\]

Multi-Level Caches
Options: separate data and instruction caches, or a unified cache

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 clock cycle for L1
    3-8 clock cycles for L2

Miss Penalty
• additional time required because of a miss
  • Typically 25-100 cycles for main memory
Impact of Cache and Line Size

**Cache Size**
- impact on miss rate:
  - larger is better
- impact on hit time:
  - smaller is faster

**Line Size**
- impact on miss rate:
  - big lines can help exploit spatial locality (if it exists)
  - however, for a given cache size, bigger lines means that there are fewer of them (which can hurt the miss rate)
- impact on miss penalty:
  - given a fixed amount of bandwidth, larger lines means longer transfer times (and hence larger miss penalties)

Impact of Associativity

- Direct mapped, set associative, or fully associative?

**Total Cache Size (tags+data):**
- Higher associativity requires more tag bits, LRU state machine bits
- Additional read/write logic, multiplexers (MUXs)

**Miss Rate:**
- Higher associativity (generally) decreases miss rate

**Hit Time:**
- Higher associativity increases hit time
  - direct mapped is the fastest

**Miss Penalty:**
- Higher associativity may require additional delays to select victim
  - in practice, this decision is often overlapped with other parts of the miss

Impact of Write Strategy

- Write through or write back?

**Advantages of Write Through:**
- Read misses are cheaper.
  - Why?
- Simpler to implement.
  - uses a write buffer to pipeline writes

**Advantages of Write Back:**
- Reduced traffic to memory
  - especially if bus used to connect multiple processors or I/O devices
- Individual writes performed at the processor rate

Qualitative Cache Performance Model

**Compulsory (aka "Cold") Misses:**
- first access to a memory line (which is not in the cache already)
  - since lines are only brought into the cache on demand, this is guaranteed to be a cache miss
  - changing the cache size or configuration does not help

**Capacity Misses:**
- active portion of memory exceeds the cache size
  - the only thing that really helps is increasing the cache size

**Conflict Misses:**
- active portion of address space fits in cache, but too many lines map to the same cache entry
  - increased associativity and better replacement policies can potentially help
Measuring Memory Bandwidth

```c
int data[MAXSIZE];
int test(int size, int stride)
{
    int result = 0;
    int wsize = size/sizeof(int);
    for (i = 0; i < wsize; i+= stride)
        result += data[i];
    return result;
}
```

Measurement
- Time repeated calls to `test`
  - If size sufficiently small, then can hold array in cache

Characteristics of Computation
- Stresses read bandwidth of system
- Increasing stride yields decreased spatial locality
  - On average will get stride*4/B accesses / cache block
- Increasing size increases size of “working set”

Alpha Memory Mountain Range

Effects Seen in Mountain Range

Cache Capacity
- See sudden drops as increase working set size

Cache Block Effects
- Performance degrades as increase stride
  - Less spatial locality
- Levels off
  - When reach single access per line
**Alpha Cache Sizes**

- **MB/s for stride = 16**

**Ranges**
- 512k – 8k: Running in L1 (High overhead for small data set)
- 16k – 64k: Running in L2
- 128k: Indistinct cutoff (Since cache is 96KB)
- 256k – 2m: Running in L3
- 4m – 16m: Running in main memory

**Alpha Line Size Effects**

**Observed Phenomenon**
- As double stride, decrease accesses/block by 2
- Until reaches point where just 1 access / block
- Line size at transition from downward slope to horizontal line
- Sometimes indistinct

**Alpha Line Sizes**

**Measurements**
- 8k: Entire array L1 resident. Effectively flat (except for overhead)
- 32k: Shows that L1 line size = 32B
- 1024k: Shows that L2 line size = 32B
- 16m: L3 line size = 64B

**Xeon Memory Mountain Range**

- Pentium III Xeon 550 MHz
- 16 KB (L1)
- 512 KB (L2)
- L1 Resident
- L2 Resident
- Main Memory Resident
Xeon Cache Sizes

- MB/s for stride = 16

Ranges
- 5k - 16k: Running in L1. (Overhead at high end)
- 32k - 256k: Running in L2.
- 512k: Running in main memory (but L2 supposed to be 512K)
- 1m - 16m: Running in main memory

Xeon Line Sizes

Measurements
- 4k: Entire array L1 resident. Effectively flat (except for overhead)
- 256k: Shows that L1 line size = 32B
- 16m: Shows that L2 line size = 32B

Interactions Between Program & Cache

Major Cache Effects to Consider
- Total cache size
  - Try to keep heavily used data in cache closest to processor
- Line size
  - Exploit spatial locality
  - Variable sum held in register

Example Application
- 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

Matrix Mult. Performance: Sparc20

- As matrices grow in size, they eventually exceed cache capacity
- Different loop orderings give different performance
  - cache effects
  - whether or not we can accumulate partial sums in registers
Miss Rate Analysis for Matrix Multiply

Assume:
- Line size = 32B (big enough for 4 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

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

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>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>
### Summary of Matrix Multiplication

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

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

#### jki (i & kji):
- 2 loads, 1 store
- misses/iter = 2.0

#### Example Code:
```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;
    }
}
```

### Matrix Mult. Performance: DEC5000

- 2 loads, 1 store
- misses/iter = 0.5 (misses/iter = 1.25, misses/iter = 2.0)

### Matrix Mult. Performance: Sparc20

- 2 loads, 1 store
- misses/iter = 0.5 (misses/iter = 1.25, misses/iter = 2.0)

### Matrix Mult. Performance: Alpha 21164

- Too big for L1 Cache
- Too big for L2 Cache
- (misses/iter = 0.5, misses/iter = 1.25, misses/iter = 2.0)
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++) {
            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

- 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