Memory System Performance

October 17, 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 \( (C = S \times E \times B \text{ bytes}) \)

Address \( (n = t + s + b \text{ bits}) \)

Cache line

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>t bits</td>
<td>( B = 2^b \text{ bytes (line size)} )</td>
</tr>
</tbody>
</table>

Block

\( S = 2^s \text{ sets} \)

\( E \) lines/set

Cache line diagram:

- Address space
- Cache
- Address components: \( t \) tag, \( s \) sets, \( b \) bytes
- Cache line: Valid bit, tag, data
- Block with \( B \) bytes (line size)
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>$/Mbyte</th>
<th>line size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Dcache</td>
<td>200 B</td>
<td>3 ns</td>
<td>$100/MB</td>
<td>8 B</td>
<td>30 GB</td>
<td></td>
</tr>
<tr>
<td>L1 Icache</td>
<td>8-64 KB</td>
<td>3 ns</td>
<td>$1.50/MB</td>
<td>32 B</td>
<td>8 ms</td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1-4MB SRAM</td>
<td>4 ns</td>
<td>$0.05/MB</td>
<td>32 B</td>
<td>8 KB</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>128 MB DRAM</td>
<td>60 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>disk</td>
<td>30 GB</td>
<td>8 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

larger line size, higher associativity, more likely to write back
Key Features of Caches

Accessing Word Causes Adjacent Words to beCached

- B bytes having same bit pattern for upper \( n-b \) address bits

\[
\begin{array}{c|c|c}
\text{t} & \text{s} & \text{b} \\
00 & 101 & 1000 \\
\end{array}
\]

- In anticipation that will want to reference these words due to spatial locality in program

Loading Block into Cache Causes Existing Block to beEvicted

- One that maps to same set
- If \( E > 1 \), then generally choose least recently used
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 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
Categorizing Cache Misses

Compulsory (“Cold”) Misses:
- First ever access to a memory line
  - since lines are only brought into the cache on demand, this is guaranteed to be a cache miss
- Programmer/system cannot reduce these

Capacity Misses:
- Active portion of memory exceeds the cache size
- Programmer can reduce by rearranging data & access patterns

Conflict Misses:
- Active portion of address space fits in cache, but too many lines map to the same cache entry
- Programmer can reduce by changing data structure sizes
  - Avoid powers of 2
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;
}
```

Stride (words)

Size (bytes)
Measuring Memory Bandwidth (cont.)

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

DEC Alpha 21164
466 MHz
8 KB (L1)
96 KB (L2)
2 M (L3)

L1 Resident
L2 Resident
L3 Resident
Main Memory Resident
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

- .5k – 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 = 64?


Xeon Memory Mountain Range

Pentium III Xeon
550 MHz
16 KB (L1)
512 KB (L2)

MB/s

Stride

Data Set

Main Memory Resident

L1 Resident

L2 Resident

class15.ppt
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

Example Application

- **Multiply N x N matrices**
- **O(N³) 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;
    }
}
```

Variable `sum` held 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
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 line size (B) > 8 bytes, exploit spatial locality
    – compulsory miss rate = 8 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></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></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
    }
}
```

<p>| Misses per Inner Loop Iteration: |</p>
<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>

Inner loop:

- Row-wise
- Column-wise
- Fixed
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></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 (ijk)

/* 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];
    }
}

Inner loop:
(i,*), (i,k), (k,*)

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)

/* 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:

\[
\begin{array}{ccc}
A & B & C \\
1.0 & 0.0 & 1.0
\end{array}
\]
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>
## Summary of Matrix Multiplication

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Loads</th>
<th>Stores</th>
<th>Misses/Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ijk (i(j)k):</td>
<td>2</td>
<td>0</td>
<td>1.25</td>
</tr>
<tr>
<td>kij (i(k)j):</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>jki (k(j)i):</td>
<td>2</td>
<td>1</td>
<td>2.0</td>
</tr>
</tbody>
</table>

```
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;
    }
}
```
Matrix Mult. Performance: DEC5000

- <i>m</i> (misses/iter = 0.5)
- <i>kij</i> (misses/iter = 1.25)
- <i>jk</i> (misses/iter = 2.0)

mflops (d.p.) vs. matrix size (n)
Matrix Mult. Performance: Sparc20

Multiple columns of B fit in cache

(matrix size (n))

(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

(matrixes/iter = 0.5)
(matrixes/iter = 1.25)
(matrixes/iter = 2.0)
Matrix Mult.: Pentium III Xeon

Matrix Size (n)

MFlops (d.p.)

(ijk)

(ikk)

(jki)

(kjk)

(kji)

(misses/iter = 0.5 or 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

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

Key idea: Sub-blocks (i.e., $A_{xy}$) can be treated just like scalars.

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]
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.

```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;
    }
}
```

- Innermost Loop Pair

- Block reused $n$ times in succession
- Update successive elements of sliver
- 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.
Blocked Matrix Mult. Perf: DEC5000

\[ \text{matrix size (n)} \]

\[ \text{mflops (d.p.)} \]

- \( b_{ijk} \)
- \( b_{ikj} \)
- \( i_{kj} \)
- \( i_{jk} \)
- \( i_{jk} \)
Blocked Matrix Mult. Perf: Sparc20

![Graph showing performance of blocked matrix multiplication with Sparc20. The graph plots mflops (d.p.) against matrix size (n). The performance varies with different block sizes and matrix multiplication methods.]

Legend:
- `n` for \( bijk \)
- `m` for \( bikj \)
- `s` for \( ikj \)
- `u` for \( ijk \)
Blocked Matrix Mult. Perf: Alpha 21164

![Graph showing performance of different blocked matrix multiplication methods as a function of matrix size (n)].

- **bijk**
- **bikj**
- **ijk**
- **ikj**

**mflops (d.p.)** vs. **matrix size (n)**
Blocked Matrix Mult. : Xeon

MFlops (d.p.)

Matrix Size (n)

ijk
ikj
bijk
bikj
Observations

Programmer Can Optimize for Cache Performance

- How data structures are organized
- How data accessed
  - Nested loop structure
  - Blocking is general technique

All Machines Like “Cache Friendly Code”

- Getting absolute optimum performance very platform specific
  - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small