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

Valid bit | tag | data
1 bit | t bits | \(B = 2^b \text{ bytes (line size)}\)

Cache line (cache block)

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

\(E \text{ lines/set}\)
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>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Dcache</td>
<td>200 B</td>
<td>3 ns</td>
<td>$100/MB</td>
<td>8 B</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>
</tr>
<tr>
<td>L2 Cache</td>
<td>1-4MB SRAM</td>
<td>4 ns</td>
<td>$100/MB</td>
<td>32 B</td>
</tr>
<tr>
<td>Memory</td>
<td>128 MB DRAM</td>
<td>60 ns</td>
<td>$1.50/MB</td>
<td>8 KB</td>
</tr>
<tr>
<td>Memory disk</td>
<td>9 GB</td>
<td>8 ms</td>
<td>$0.05/MB</td>
<td></td>
</tr>
</tbody>
</table>

larger, slower, cheaper

larger line size, higher associativity, more likely to write back
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
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;
}
```
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)

L1 Resident

L2 Resident

Main Memory Resident

MB/s

Stride

DataSet

class19.ppt
• 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
- $O(N^3)$ total operations
- Accesses
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

```
/* 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
Layout of Arrays in Memory

C arrays allocated in row-major order

- each row in contiguous memory locations

Stepping through columns in one row:

```c
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:

```c
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)

/* 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></td>
<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></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></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></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<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></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)

/* 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:

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

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

```plaintext
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;
  }
}
```

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

```plaintext
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];
  }
}
```

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

```plaintext
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

![Graph showing performance of matrix multiplication on DEC5000 with varying matrix sizes (n) and different layout orders (i,k,j, s,i,j,k, u,j,k, q,j,k,i, m,k,j,i). The graph includes lines indicating performance at misses/iter = 0.5, 1.25, and 2.0.](class19.ppt)
Matrix Mult. Performance: Sparc20

Multiple columns of B fit in cache

(matrix size (n))

mflops (d.p.)

0 2 4 6 8 10 12 14 16 18 20

50 75 100 125 150 175 200

ikj (misses/iter = 0.5)
kij (misses/iter = 1.25)
ijk (misses/iter = 2.0)
jik
jki
kji

(matrix size (n))

class19.ppt

CS 213 F'99
Matrix Mult. Performance: Alpha 21164

Too big for L1 Cache
Too big for L2 Cache

(matrix size (n))

mflops (d.p.)

(ijk)

(ikj)

(jik)

(jki)

(kij)

(kji)

(misses/iter = 0.5)

(misses/iter = 1.25)

(misses/iter = 2.0)
Matrix Mult.: Pentium III Xeon

- MFlops (d.p.)
- Matrix Size (n)

(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)

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

```cpp
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

A row sliver accessed $bsize$ times

B block reused $n$ times in succession

C Update successive elements of sliver
Blocked Matrix Mult. Perf: DEC5000

![Graph showing performance of matrix multiplication for different sizes of matrices.](image)

- **Legend**:
  - `n`: bijk
  - `+`: bikj
  - `-`: ikj
  - `*`: ijk

- **Axes**:
  - **y-axis**: mflops (d.p.)
  - **x-axis**: matrix size (n)

- **Data Points**:
  - (50, 3.0)
  - (75, 2.5)
  - (100, 2.0)
  - (125, 1.5)
  - (150, 1.0)
  - (175, 0.5)
  - (200, 0.0)
Blocked Matrix Mult. Perf: Sparc20

![Graph showing mflops (d.p.) vs matrix size (n)]
Blocked Matrix Mult. Perf: Alpha 21164

matrix size (n)
mflops (d.p.)

bijk
bikj
ijk
ikj
Blocked Matrix Mult. : Xeon

MFlops (d.p.)

Matrix Size (n)

ijk

ikj

bijk

bikj