More about Caches

Randal E. Bryant
CS347 Lecture 8
Feb. 5, 1998

Topics

• Performance factors
• Access patterns & program performance
  – Use matrix multiply as case study
• Memory hierarchy bandwidths
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  |  \(B = 2^b \text{ bytes (block size)}\)

Cache block  
(cache line)

\(E \text{ blocks/set}\)  
\(S = 2^s \text{ sets}\)
Cache Performance Metrics

Miss Rate

- fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 5-10% for L1
  - 1-2% for L2

Hit Time

- time to deliver a block in the cache to the processor (includes time to determine whether the block 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 10-30 cycles for main memory
Impact of Cache and Block Size

Cache Size

• Effect on miss rate
  – Larger is better
• Effect on hit time
  – Smaller is faster

Block Size

• Effect on miss rate
  – Big blocks help exploit spatial locality
  – For given cache size, can hold fewer big blocks than little ones, though
• Effect on miss penalty
  – Longer transfer time
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, multiplexors

**Miss rate**
- Higher associativity decreases miss rate

**Hit time**
- Higher associativity increases hit time
  - Direct mapped allows test and data transfer at the same time for read hits.

**Miss Penalty**
- Higher associativity requires additional delays to select victim
Impact of Replacement Strategy

• RAND, FIFO, or LRU?

Total Cache Size (tags+data)
• LRU requires complex state machine for each set
• FIFO requires simpler state machine for each set
• RAND very simple

Miss Rate
• LRU has up to ~10% lower miss rate than FIFO
• RAND does much worse

Miss Penalty
• LRU takes more time to select victim
Impact of Write Strategy

• Write-through or write-back?

Advantages of Write Through
• Read misses are cheaper. Why?
• Simpler to implement.
• Requires 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
Allocation Strategies

• On a write miss, is the block loaded from memory into the cache?

Write Allocate:

• Block is loaded into cache on a write miss.
• Usually used with write back
• Otherwise, write-back requires read-modify-write to replace word within block

- But if you’ve gone to the trouble of reading the entire block, why not load it in cache?
Allocation Strategies (Cont.)

- On a write miss, is the block loaded from memory into the cache?

**No-Write Allocate (Write Around):**
- Block is not loaded into cache on a write miss
- Usually used with write through
  - Memory system directly handles word-level writes
Qualitative Cache Performance Model

Compulsory Misses
- First access to line not in cache
- Also called “Cold start” misses

Capacity Misses
- Active portion of memory exceeds cache size

Conflict Misses
- Active portion of address space fits in cache, but too many lines map to same cache entry
- Direct mapped and set associative placement only

Validation Misses
- Block invalidated by multiprocessor cache coherence mechanism
- Not our concern right now
Interactions Between Program & Cache

Major Cache Effects to Consider

• Total cache size
  – Try to keep heavily used data in highest level cache

• Block size (sometimes referred to “line size”)
  – Exploit spatial locality

Example Application

• Multiply $n \times n$ 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

```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
Matmult Performance (Sparc20)

- As matrices grow in size, exceed cache capacity
- Different loop orderings give different performance
  - Cache effects
  - Whether or not can accumulate in register
Layout of Arrays in Memory

C Arrays Allocated in Row-Major Order

- Each row in contiguous memory locations

Stepping Through Columns in Row

```c
for (i = 0; i < n; i++)
    sum += a[0][i];
```
- Accesses successive elements
- For block size > 8, get spatial locality
  - Cold Start Miss Rate = 8/B

Stepping Through Rows in Column

```c
for (i = 0; i < n; i++)
    sum += a[i][0];
```
- Accesses distant elements
- No spatial locality
  - Cold Start Miss rate = 1

Memory Layout

- Each row in contiguous memory locations
- For block size > 8, get spatial locality
  - Cold Start Miss Rate = 8/B
Miss Rate Analysis

Assume
- Block size = 32B (big enough for 4 double’s)
- n is very large
  - Approximate 1/n as 0.0
- Cache not even big enough to hold multiple rows

Analysis Method
- Look at access pattern by 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;
    }
}

Approx. Miss Rates

\[
\begin{array}{ccc}
a & b & c \\
0.25 & 1.0 & 0.0 \\
\end{array}
\]
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
    }
}

Approx. Miss Rates

\[
\begin{array}{ccc}
    a & b & c \\
    0.25 & 1.0 & 0.0 \\
\end{array}
\]
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];
    }
}

Inner loop:

Approx. Miss Rates

<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];
    }
}

Approx. Miss Rates

<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>

Inner loop:
- Fixed
- Row-wise
- Row-wise
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;
 }
}
```

Approx. Miss Rates

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Inner loop:

- Column-wise
- Fixed
- Column-wise

A B C
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;
    }
}
```

**Approx. Miss Rates**

<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

**ijk (L=2, S=0, MR=1.25)**

\[
\text{for } (i=0; i<n; i++) \{ \\
\quad \text{for } (j=0; j<n; j++) \{ \\
\quad\quad \text{sum} = 0.0; \\
\quad\quad \text{for } (k=0; k<n; k++) \\
\quad\quad\quad \text{sum} += a[i][k] \times b[k][j]; \\
\quad\quad \text{c[i][j]} = \text{sum}; \\
\quad \}\}
\]

**jik (L=2, S=0, MR=1.25)**

\[
\text{for } (j=0; j<n; j++) \{ \\
\quad \text{for } (i=0; i<n; i++) \{ \\
\quad\quad \text{sum} = 0.0; \\
\quad\quad \text{for } (k=0; k<n; k++) \\
\quad\quad\quad \text{sum} += a[i][k] \times b[k][j]; \\
\quad\quad \text{c[i][j]} = \text{sum}; \\
\quad \}\}
\]

**kij (L=2, S=1, MR=0.5)**

\[
\text{for } (k=0; k<n; k++) \{ \\
\quad \text{for } (i=0; i<n; i++) \{ \\
\quad\quad \text{r} = a[i][k]; \\
\quad\quad \text{for } (j=0; j<n; j++) \\
\quad\quad\quad \text{c[i][j]} += r \times b[k][j]; \\
\quad \}\}
\]

**ikj (L=2, S=1, MR=0.5)**

\[
\text{for } (i=0; i<n; i++) \{ \\
\quad \text{for } (k=0; k<n; k++) \{ \\
\quad\quad \text{r} = a[i][k]; \\
\quad\quad \text{for } (j=0; j<n; j++) \\
\quad\quad\quad \text{c[i][j]} += r \times b[k][j]; \\
\quad \}\}
\]

**jki (L=2, S=1, MR=2.0)**

\[
\text{for } (j=0; j<n; j++) \{ \\
\quad \text{for } (k=0; k<n; k++) \{ \\
\quad\quad \text{r} = b[k][j]; \\
\quad\quad \text{for } (i=0; i<n; i++) \\
\quad\quad\quad \text{c[i][j]} += a[i][k] \times r; \\
\quad \}\}
\]

**kji (L=2, S=1, MR=2.0)**

\[
\text{for } (k=0; k<n; k++) \{ \\
\quad \text{for } (j=0; j<n; j++) \{ \\
\quad\quad \text{r} = b[k][j]; \\
\quad\quad \text{for } (i=0; i<n; i++) \\
\quad\quad\quad \text{c[i][j]} += a[i][k] \times r; \\
\quad \}\}
\]
Matmult performance (DEC5000)

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

- **Legend:**
  - `ikj`
  - `kij`
  - `ijk`
  - `jik`
  - `jki`
  - `kji`

- **Axes:**
  - **Y-axis:** MFLOPS (D.P.)
  - **X-axis:** Matrix size (n)

- **Key Points:**
  - (L=2, S=1, MR=0.5)
  - (L=2, S=0, MR=1.25)
  - (L=2, S=1, MR=2.0)
Matmult Performance (Sparc20)

Multiple columns of B fit in cache?

- ikj
- kij
- ijk
- jik
- jki
- kji

(L=2, S=1, MR=0.5)
(L=2, S=0, MR=1.25)
(L=2, S=1, MR=2.0)
Matmult Performance (Alpha 21164)

Too big for L1 Cache

Too big for L2 Cache

(L=2, S=1, MR=0.5)

(L=2, S=0, MR=1.25)

(L=2, S=1, MR=2.0)
Block Matrix Multiplication

Example \( n=8, B = 4 \):

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

Key idea: Sub-blocks (i.e., \( A_{ij} \)) 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-1,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 1 X bsize sliver of A times 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

```
  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

- Update successive elements of sliver
- row sliver accessed bsize times
- block reused n times in succession

Update successive elements of sliver
Blocked matmult perf (DEC5000)
Blocked matmult perf (Sparc20)

![Graph showing mflops (d.p.) vs. matrix size (n) for different block orders.]

- bijk
- bikj
- ikj
- ijk
Blocked matmult perf (Alpha 21164)
The Memory Mountain Range

DEC Alpha 8400 (21164)
300 MHz
8 KB (L1) 96 KB (L2) 4 M (L3)
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 block
Bandwidth Matching

Challenge

• CPU works with short cycle times
• DRAM (relatively) long cycle times
• How can we provide enough bandwidth between processor & memory?

Effect of Caching

• Caching greatly reduces amount of traffic to main memory
• But, sometimes need to move large amounts of data from memory into cache

Trends

• Need for high bandwidth much greater for multimedia applications
  – Repeated operations on image data
• Recent generation machines (e.g., Pentium II) greatly improve on predecessors
High Bandwidth Memory Systems

Solution 1
High BW DRAM
Example: Page Mode DRAM
RAMbus

Solution 2
Wide path between memory & cache
Example: Alpha AXP 21064
256 bit wide bus, L2 cache, and memory.

Solution 3
Memory bank interleaving
Example: Alpha 8400
4 GByte 4 bank memory module