CSE 351

Caches
Agenda

• Upcoming Deadlines
  – Lab 3 due Friday 11/13 at 5:00pm
  – HW 3 will be released Friday, due Friday 11/20

• Section Survey

• Introduction to Caches

• Midterm and Lab 3 Questions?
Feedback for your TA

• In general, pace of this section is:
  1. too fast
  2. kind of fast
  3. just right
  4. kind of slow
  5. too slow

• Please **Keep** doing this:
• Please **Stop** doing this:
• Please **Start** doing this:
Performance Bottlenecks

• How do we define “computer performance”
  – Instruction throughput, i.e. how many instructions can we execute per second?

• What factors affect this?
  – CPU architecture gives an estimate for maximum instructions per second (IPS)
  – However, the CPU can only process data at the rate it receives the data
  – Reading from memory has become increasingly slow relative to the rate at which modern processors can execute instructions
  – Memory is a bottleneck!
What is a Cache?

• A CPU cache is a small, fast region of memory that is actually on the same piece of silicon as the processor
  – Modern processors have as many as 4 caches, each progressively larger and slower
  – They form a cache hierarchy
• Caches are orders of magnitude faster than the DRAM used for main memory
• However, they fill up quickly, so the CPU needs to be smart about how it uses caches
Caches Improve Performance

• Data caching is a common optimization in many systems

• Example: Web Browsers
  – Network latency is a bottleneck
  – Web browsers can’t display pages until they have received the data
  – Your web browser caches static files (images, .html, .css, .js, etc) that you have recently downloaded onto your hard drive
  – Relative to the network, your hard drive is speedy

• Processors use the same concept to improve performance, by caching data as well as instructions
Cache Mechanics

- Smaller, faster, more expensive memory.
- Caches a subset of the blocks (a.k.a. lines)

Data is copied in block-sized transfer units

- Larger, slower, cheaper memory.
- Viewed as partitioned into “blocks” or “lines”
General Cache Concepts: **Hit**

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: **Miss**

- **Data in block b is needed**
- **Block b is not in cache:** **Miss!**
- **Block b is fetched from memory**
- **Block b is stored in cache**
  - Placement policy: determines where b goes
  - Replacement policy: determines which block gets evicted (victim)
Why Caches Work

**Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses *tend* to be referenced close together in time

- How do caches take advantage of these access patterns?
Why Caches Work

**Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future
  - Recently referenced memory locations are stored in the cache

- **Spatial locality:**
  - Items with nearby addresses *tend* to be referenced close together in time
  - Nearby addresses are moved to the cache when one is needed
Locality Examples

- **Temporal**: \( a \) is accessed multiple times

\[
\begin{align*}
x &= a + 6; \\
y &= a + 5; \\
z &= 8 \times a;
\end{align*}
\]

- **Spatial**: increasing indices of \( b \) are accessed in sequence

\[
\begin{align*}
x &= b[0] + 6; \\
y &= b[1] + 5; \\
z &= 8 \times b[2];
\end{align*}
\]
Example: Any Locality?

sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;

• Data:
  – **Temporal**: `sum` referenced in each iteration
  – **Spatial**: array `a[ ]` accessed in stride-1 pattern

• Instructions:
  – **Temporal**: cycle through loop repeatedly
  – **Spatial**: reference instructions in sequence

• Being able to assess the locality of code is a crucial skill for a programmer
Where is the Locality?

```c
for (i = 1; i < 100; i++) {
    a = a * 7;
    b = b + x[i];
    c = y[5] + d;
}
```

- Remember: accessing arrays doesn’t automatically mean spatial or any locality since you could access the indices in random order
Locality Example #1

int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}

M = 3, N=4

<table>
<thead>
<tr>
<th>Row</th>
<th>Accessed Order</th>
<th>Layout in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a[0][0]</td>
<td>a[0][0]</td>
</tr>
<tr>
<td>1</td>
<td>a[0][1]</td>
<td>a[0][1]</td>
</tr>
<tr>
<td>2</td>
<td>a[0][2]</td>
<td>a[0][2]</td>
</tr>
<tr>
<td>3</td>
<td>a[0][3]</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>4</td>
<td>a[1][0]</td>
<td>a[1][0]</td>
</tr>
<tr>
<td>5</td>
<td>a[1][1]</td>
<td>a[1][1]</td>
</tr>
<tr>
<td>6</td>
<td>a[1][2]</td>
<td>a[1][2]</td>
</tr>
<tr>
<td>7</td>
<td>a[1][3]</td>
<td>a[1][3]</td>
</tr>
<tr>
<td>8</td>
<td>a[2][0]</td>
<td>a[2][0]</td>
</tr>
<tr>
<td>9</td>
<td>a[2][1]</td>
<td>a[2][1]</td>
</tr>
<tr>
<td>10</td>
<td>a[2][2]</td>
<td>a[2][2]</td>
</tr>
<tr>
<td>11</td>
<td>a[2][3]</td>
<td>a[2][3]</td>
</tr>
</tbody>
</table>

stride-1
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}

M = 3, N=4

Layout in Memory

Order Accessed

stride-N
What is wrong with this code?

How can it be fixed?

```c
int sum_array_3d(int a[M][N][L])
{
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < L; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

Layout in Memory (for M = ?, N = 3, L=4)
Locality and Data Structures

• Which has (at least the potential for) better spatial locality, arrays or linked lists?
Locality and Data Structures

• Which has (at least the potential for) better spatial locality, arrays or linked lists?
• Nodes in a linked list are not allocated contiguously
• On the other hand, array elements are allocated contiguously
• What about node fields? Data, next, and other fields could be allocated contiguously