Memory Hierarchy and Caches

CMPSCI 230 Computer Systems Principles
Objectives

• Memory Hierarchy
  – Learn about the memory hierarchy
  – Understand “nearness” and why it is important
  – Learn how “nearness” impacts the code that you write

• Caches
  – Learn what caches are
  – Understand why they are important
  – Learn how caches are accessed
  – Understand the performance advantage
The Memory Hierarchy

- Processor
- CPU
  - Processor Register
  - CPU Cache
    - Level 1 (L1) Cache
    - Level 2 (L2) Cache
    - Level 3 (L3) Cache
- Physical Memory
- Random Access Memory (RAM)
- Solid State Memory
  - Non-Volatile Flash-Based Memory
- Virtual Memory
  - File-Based Memory
- EDO, SD-RAM, DDR-SDRAM, RD-RAM and More...
- SSD, Flash Drive
- Mechanical Hard Drives

Super fast: Super expensive tiny capacity
Faster: Expensive small capacity
Fast: Priced reasonably average capacity
Average: Priced reasonably average capacity
Slow: Cheap large capacity

Illustration: Ryan J. Leng
Nearness

Where your data is located makes a huge difference in performance!
Nearness

Where your data is located makes a huge difference in performance!

The closer your data is to the execution core of the machine, the better performance!
Locality – sumvec

int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}
Locality – sumvec

```c
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}
```

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Locality – sumvec

```c
int sumvec(int v[N]) {
    int i, sum = 0;
    for (i = 0; i < N; i++) {
        sum += v[i];
    }
    return sum;
}
```

**v (N = 8)**

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Locality – sumvec

```c
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}
```

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Locality – sumvec

```c
int sumvec(int v[N]) {
    int i, sum = 0;
    for (i = 0; i < N; i++) {
        sum += v[i];
    }
    return sum;
}
```

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Locality – sumvec

```c
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        sum += v[i];
    }

    return sum;
}
```

v (N = 8)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v0</td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td>v7</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
int sumvec(int v[N]) {
    int i, sum = 0;
    for (i = 0; i < N; i++) {
        sum += v[i];
    }
    return sum;
}

• How are i and sum being accessed?
• Where should these values be stored?
• What about v?
Locality – sumrows

```c
int sumrows(int v[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }
    return sum;
}
```

\(v\) (\(M = 2, N = 3\))

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Locality – sumrows

```c
int sumrows(int v[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }
    return sum;
}
```

**v (M = 2, N = 3)**

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Locality – sumrows

```c
int sumrows(int v[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }
    return sum;
}
```

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Locality – sumrows

```c
int sumrows(int v[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }
    return sum;
}
```

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
int sumrows(int v[M][N]) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }

    return sum;
}

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Locality – sumrows

```c
int sumrows(int v[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }
    return sum;
}
```

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Locality – sumrows

```c
int sumrows(int v[M][N]) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }

    return sum;
}
```

$v (M = 2, N = 3)$

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
int sumrows(int v[M][N]) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }

    return sum;
}
Locality – sumrows

```c
int sumrows(int v[M][N]) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)
            sum += v[i][j];
    }

    return sum;
}
```

- How are i, j, and sum being accessed?
- Where should these values be stored?
- What about v?

v is accessed sequentially – it has a stride-1 access pattern
How can we improve efficiency?

• What is the access pattern of i, j, and sum?
  – We access them over and over again
  – This is in terms of time
  – This is called temporal locality
  – i, j, and sum exhibit good temporal locality

• How can we improve efficiency?
  – We can use registers!
  – Can be used to move data closer to the execution engine
  – What if we have too many local variables?
How can we improve efficiency?

• What is the access pattern \( v \)?
  – We access data that is near other data we access
  – This is in terms of \( space \)
  – This is called \textit{spatial locality}
  – \texttt{sumrows} exhibits \textit{good} spatial locality

• How can we improve efficiency?
  – We “can” use registers, but we will quickly run out

What can we do to improve efficiency?
Is this a software issue?
Can we extend the machine to improve this?
Locality – sumcols

```c
int sumcols(int v[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }

    return sum;
}
```

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Locality – sumcols

int sumcols(int v[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }

    return sum;
}

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
int sumcols(int v[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }

    return sum;
}

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
int sumcols(int v[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }
    return sum;
}

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
int sumcols(int v[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }
    return sum;
}

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
int sumcols(int v[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }

    return sum;
}

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
int sumcols(int v[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }

    return sum;
}

v (M = 2, N = 3)

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v00</td>
<td>v01</td>
<td>v02</td>
<td>v10</td>
<td>v11</td>
<td>v12</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Locality – sumcols

```c
int sumcols(int v[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }
    return sum;
}
```

- How are i, j, and sum being accessed?
- Where should these values be stored?
- What about v?
Locality – sumcols

```c
int sumcols(int v[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            sum += v[i][j];
    }

    return sum;
}
```

- How are i, j, and sum being accessed?
- Where should these values be stored?
- What about v?

v is NOT accessed sequentially – it has a stride-k access pattern, where k = 3
How can we improve efficiency?

• What is the access pattern of i, j, and sum?
  – We access them over and over again
  – This is in terms of time
  – This is called temporal locality
  – i, j, and sum exhibit good temporal locality

• How can we improve efficiency?
  – We can use registers!
  – Can be used to move data closer to the execution engine
  – What if we have too many local variables?
How can we improve efficiency?

• What is the access pattern \( v \)?
  – We access data that is near other data we access
  – This is in terms of \( space \)
  – This is called \textbf{spatial locality}
  – sumcols exhibits \textbf{poor} spatial locality

What can we do to improve efficiency?
Is this a software issue?
Can we extend the machine to improve this?
The Memory Hierarchy

Registers are the fastest memory locations on a machine.

Sometimes called L0.

It takes 1 machine cycle to access a register.

This is called access or memory latency.
Caches are larger banks of memory that are used to “remember” recently used data items.

There are typically 2 or 3 levels of cache:

L1 – smallest, 2-cycle latency
L2 – larger than L1, 7-cycle latency
L3 – larger than L2, < 100-cycle latency
The Memory Hierarchy

- Registers
- L1
- L2
- RAM

Execution Engine / CPU

Smaller, More expensive (\$\$)
LOWER LATENCY
The Memory Hierarchy

- Registers
- L1
- L2
- RAM

Execution Engine / CPU

Larger, Less Expensive ($$)
HIGHER LATENCY
The Memory Hierarchy

Caches are also used to remember instructions as well as data.

Typically:
L1 is split
I-Cache (Instructions)
D-Cache (Data)
L2 remembers both (shared)

What type of code can benefit from remembering recently used instructions?
The Memory Hierarchy

Registers

L1

L2

RAM

Most expensive memory!

At least 100-cycle latency!
The Memory Hierarchy: Access

Program requests data for the first time.

Initially, L1 and L2 are: COLD
Program requests data for the first time.

Initially, L1 and L2 are: COLD
The Memory Hierarchy: Access

Program requests data for the first time.

Initially, L1 and L2 are: COLD
The Memory Hierarchy: Access

Program requests data for the first time.

Initially, L1 and L2 are: COLD

Data Located in Main Memory!
Program requests data for the first time.

Initially, L1 and L2 are: COLD

Data Located in Main Memory!

Fetched, and returned to next higher level of memory.
The Memory Hierarchy: Access

Program requests data for the first time.

Initially, L1 and L2 are: COLD

Data Located in Main Memory!

_FETCHED, and returned to next higher level of memory.
The Memory Hierarchy: Access

Program requests data for the first time.

Initially, L1 and L2 are: COLD

100+ Latency

Data Located in Main Memory!

Fetched, and returned to next higher level of memory.
The Memory Hierarchy: Access

Program requests data for the next time.

Results in a **HIT**!

2-cycle latency!
The Memory Hierarchy: Access

Execution Engine / CPU

Registers

L1

L2

RAM

Program requests data for the next time.

Results in a **HIT**!

2-cycle latency!

Transferred in bulk. (Cache Line/Block)
After a number of requests to memory, the cache becomes warm or primed.

How does cache exploit spatial locality?

What about temporal locality?
Assume a cache hit takes 2 cycles and a cache miss goes to memory and takes 100 cycles. What is the maximum percent misses we can have for the average access time to be 4 cycles?

A) 50% (hit rate 50%)
B) 10% (hit rate 90%)
C) 2% (hit rate 98%)
D) 1% (hit rate 99%)
E) Something else
Cache Jargon

• Cold Cache
  – The cache is initially “empty”

• Warm Cache
  – After several fetches, cache becomes warm or primed

• Cache Miss
  – Fetch from lower level in hierarchy
  – Bring line into cache
  – Next access: **Cache Hit**

• Warmed Up
  – Cache holds most frequently/recently used data
• **Cache (pronounce “cash”)**
  – Small, fast storage device

• **Process of using/exploiting a cache**
  – “Caching”

• **Storage at cache k+1**
  – Partitioned into contiguous chunks of data objects called **blocks**
  – Each block has a *unique* address
  – Blocks can be fixed or variable size
Cache Basics

- **Storage at cache k**
  - Partitioned into a smaller set of blocks that are the same size as level k+1
  - At any point in time, the cache at level $k$ contains copies of a subset of the blocks from level k+1

- **Transfers**
  - Data is always copied back and forth between level $k$ and $k+1$ in block-sized transfer units.
Cache Basics

• Need a data object
  – If it is not in a register, we need to look it up in memory
  – We go through the cache hierarchy to do this

• Look Up
  – If the data item is there, we have a cache hit!
  – If the data item is not, we have a cache miss!
Cache Hit

Program needs data $d$ in register

$GPR$
Cache Hit

Program needs data $d$ in register

Must fetch data $d$ stored at level $k$ cache.
Cache Hit

Program needs data $d$ in register GPR.
Must fetch data $d$ stored at level $k$ cache.

If data $d$ is stored at level $k$ cache. We have a cache hit!
Cache Hit

Program needs data \( d \) in register Must fetch data \( d \) stored at level \( k \) cache.

If data \( d \) is stored at level \( k \) cache.
We have a cache hit!

Copy it to the register.
Program needs data \( d \) in register.

Must fetch data \( d \) stored at level \( k \) cache.

If data \( d \) is stored at level \( k \) cache.

We have a cache hit!

Copy it to the register.

Program reads \( d \) directly from level \( k \), which is faster than reading from level \( k+1 \).
Cache Miss

• Program needs data $d$ stored at level $k$ cache.

• If $d$ is not cached at level $k$, then we have a cache miss

• Program fetches $d$ from level $k+1$ and “adds it” to the cache at level $k$
Cache Miss

Program needs data $d$ in register

$\begin{align*}
  k & \quad L1 \\
  k+1 & \quad L2 \\
  k+2 & \quad L3 \\
  k+3 & \quad \text{Main Memory}
\end{align*}$
Cache Miss

GPR

Program needs data $d$ in register

$L1$

Must fetch data $d$ stored at level $k$ cache.

$L2$

$k+1$

$L3$

$k+2$

Main Memory

$k+3$
Cache Miss

Program needs data \( d \) in register GPR. Must fetch data \( d \) stored at level \( k \) cache.

If data \( d \) is not stored at level \( k \) cache. We have a cache miss!
Cache Miss

Program needs data $d$ in register

Must fetch data $d$ stored at level $k$ cache.

If data $d$ is not stored at level $k$ cache. We have a cache miss!

We must then fetch $d$ stored at level $k+1$ cache.

Main Memory
Cache Miss

Program needs data $d$ in register

Must fetch data $d$ stored at level $k$ cache.

If data $d$ is not stored at level $k$ cache. We have a cache miss!

We must then fetch $d$ stored at level $k+1$ cache.
Cache Miss

Program needs data $d$ in register GPR.

Must fetch data $d$ stored at level $k$ cache.

If data $d$ is not stored at level $k$ cache. We have a cache miss!

We must then fetch $d$ stored at level $k+1$ cache.

Data $d$ is then copied to level $k$ cache and then into GPR.
Cache Full Problem

What do we do if the cache is full?