15-213 Recitation
Caches and Blocking

Your TAs
Monday, February 24th, 2020 (15-213, 18-213)
Wednesday, February 26th, 2020 (18-613)
Agenda

- Logistics
- Cache Lab
- Cache Concepts
- Activity 1: Traces
- Activity 2: Blocking
- Practice Problems
- Appendix: Examples, Style, Git, fscanf
Learning Objectives

By the end of this recitation, we want you to know:

- Cache concepts
  - Basic cache organization
- Read and write trace files
- Blocking concepts
  - Matrix multiplication with blocking
Logistics

- Cache Lab is due **Thursday, Feb. 27th** at 11pm
- Midterm exam will be between **March 2nd - March 5th**
  - Review session **6-10pm Sunday, March 1st**
  - Practice problems on Exam Server
- Drop date **Monday, Feb. 24th**
Cache Lab: Overview

- Part 0: Write trace files for testing
  - Short and quick to familiarize yourself with the trace files
  - Extremely helpful for debugging later on!

- Part 1: Write a cache **simulator**
  - Substantial amount of C code!

- Part 2: Optimize some code to minimize cache misses
  - Substantial amount of thinking!

- Part 3: Style Grades
  - Worth about a letter grade on this assignment
  - Few examples in appendix
  - Full guide on course website
  - Git matters!
Cache Lab: Cache Simulator Hints

- **Goal:** Count hits, misses, evictions and # of dirty bytes

- **Procedure**
  - Least Recently Used (LRU) replacement policy
  - Structs are good for storing cache line parts (valid bit, tag, LRU counter, etc.)
  - A cache is like a 2D array of cache lines
    ```c
    struct cache_line cache[S][E];
    ```

- Your simulator needs to handle different values of S, E, and b (block size) given at run time
  - Dynamically allocate memory!

- Dirty bytes: any payload byte whose corresponding cache block’s dirty bit is set (i.e. the payload of that block has been modified, but not yet written back to main memory)
Cache Concepts
Cache Organization

E = $2^e$ lines/set

S = $2^s$ sets

V D Tag 0 1 2 3 .. B-1

B = $2^b$ bytes per block

“line” or “block”
Cache Read

- Address of word: | t bits | s bits | b bits |
  - Tag: t bits
  - Set index: s bits
  - Block offset: b bits

- Steps:
  - Use set index to get appropriate set
  - Loop through lines in set to find matching tag
  - If found and valid bit is set: hit
  - Locate data starting at block offset
Tying it all together: Bomblab

(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x00000000000400e80 <+0>:     sub    $0x8,%rsp
0x00000000000400e84 <+4>:     mov    $0x604420,%esi
0x00000000000400e89 <+9>:     callq  0x401326 <strings_not_equal>
0x00000000000400e8e <+14>:    test   %al,%al
0x00000000000400e90 <+16>:    je     0x400e97 <phase_1+23>
0x00000000000400e92 <+18>:    callq  0x401577 <explode_bomb>
0x00000000000400e97 <+23>:    add    $0x8,%rsp
0x00000000000400e9b <+27>:    retq

End of assembler dump.
Tying it all together: Bomblab

(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x000000000000400e80 <+0>:  sub $0x8,%rsp
0x000000000000400e84 <+4>:  mov $0x604420,%esi
0x000000000000400e89 <+9>:  callq 0x401326 <strings_not_equal>
0x000000000000400e8e <+14>:  test %al,%al
0x000000000000400e90 <+16>:  je 0x400e97 <phase_1+23>
0x000000000000400e92 <+18>:  callq 0x401577 <explode_bomb>
0x000000000000400e97 <+23>:  add $0x8,%rsp
0x000000000000400e9b <+27>:  retq
End of assembler dump.
Tying it all together: Bomblab

For the L1 dCache (data)

C = 32768 (32 mb)
E = 8
B = 64
S = 64

How did we get S?
Tying it all together: Bomblab

- 64 bit address space: $m = 64$
- $b = 6$
- $s = 6$
- $t = 52$
Tying it all together: Bomblab

\[ 0x00604420 \rightarrow 0b0000000110000000100010000100000 \]

- tag bits: \textcolor{blue}{0000000011000000100}
- set index bits: \textcolor{green}{010000}
- block offset bits: \textcolor{red}{100000}
Activity 1: Traces
Tracing a Cache

Example Cache: -s 1 -E 2 -b 2 (S=2 B=4)
Example Trace

L - Load
S - Store

Memory Location

Size

Jack.trace

L 0,4
S 0,4
L 0,4
L 4,1
L 5,1
L 6,1
L 7,1
Example Trace

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>15</td>
</tr>
<tr>
<td>0x01</td>
<td>21</td>
</tr>
<tr>
<td>0x02</td>
<td>13</td>
</tr>
<tr>
<td>0x03</td>
<td>18</td>
</tr>
<tr>
<td>0x04</td>
<td>51</td>
</tr>
<tr>
<td>0x05</td>
<td>30</td>
</tr>
<tr>
<td>0x06</td>
<td>ac</td>
</tr>
<tr>
<td>0x07</td>
<td>b3</td>
</tr>
</tbody>
</table>

Jack.trace
L 0,4
S 0,4
L 0,1
L 6,1
L 5,1
L 6,1
L 7,1
Example Trace

Why that line? Where are those values from?
Example Trace

What happens if values change?
Example Trace

Why is this still a hit?

What would happen if we had not previously loaded all four bytes?
Example Trace

Jack.trace
L 0,4  M
S 0,4  H
L 0,1  H
L 6,1  M
L 5,1
L 6,1
L 7,1

Memory

0x00 15
0x01 21
0x02 13
0x03 18
0x04 51
0x05 30
0x06 ac
0x07 b3

Just one Byte?
Example Trace

Jack.trace
L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1
L 6,1
L 7,1

Memory

Just one Byte?
NO!
Example Trace

Jack.trace
L 0,4  M
S 0,4  H
L 0,1  H
L 6,1  M
L 5,1
L 6,1
L 7,1

Why below and not above?
Why load all four bytes?
Example Trace

Jack.trace
L 0,4  M
S 0,4  H
L 0,1  H
L 6,1  M
L 5,1  H
L 6,1  
L 7,1  

Memory

0x00 15  0x01 21  0x02 13  0x03 18
0x04 51  0x05 30  0x06 ac  0x07 b3
Example Trace

Jack.trace

L 0,4  M
S 0,4  H
L 0,1  H
L 6,1  M
L 5,1  H
L 6,1  H
L 7,1  H

Memory

0x00 15
0x01 21
0x02 13
0x03 18
0x04 51
0x05 30
0x06 ac
0x07 b3
Example Trace

Jack.trace
L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1 H
L 6,1 H
L 7,1 H

Memory

\begin{verbatim}
0x00 15
0x01 21
0x02 13
0x03 18
0x04 51
0x05 30
0x06 ac
0x07 b3
\end{verbatim}
What would happen if we loaded from memory address 0x08?
Example Trace

What would happen if we loaded from memory address 0x08?
Activity 2: Blocking
Example: Matrix Multiplication

/* multiply 4x4 matrices */
void mm(int a[4][4], int b[4][4], int c[4][4]) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i][j] += a[i][k] * b[k][j];

Let's step through this to see what's actually happening
Example: Matrix Multiplication

- Assume a tiny cache with 4 lines of 8 bytes (2 ints)
  - S = 1, E = 4, B = 8
- Let’s see what happens if we don’t use blocking
\[
\begin{array}{c}
\text{iter} & i & j & k & \text{operation} \\
0 & 0 & 0 & 0 & c[0][0] += a[0][0] \times b[0][0]
\end{array}
\]

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
```
iter  i  j  k  operation
0  0  0  0  c[0][0] += a[0][0] * b[0][0]
1  0  0  1  c[0][0] += a[0][1] * b[1][0]
```

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
\[ c_{00} = a_{00}b_{00} \]
\[ c_{00} = a_{01}b_{10} \]
\[ c_{00} = a_{02}b_{20} \]

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( c_{00} += a_{00}b_{00} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( c_{00} += a_{01}b_{10} )</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>( c_{00} += a_{02}b_{20} )</td>
</tr>
</tbody>
</table>
\[
c_{0,0} = a_{0,0} \times b_{0,0} \\
= a_{0,1} \times b_{1,0} \\
= a_{0,2} \times b_{2,0} \\
= a_{0,3} \times b_{3,0}
\]

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Key:**
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
\begin{align*}
\text{iter} & \quad i & \quad j & \quad k & \quad \text{operation} \\
0 & 0 & 0 & 0 & c[0][0] += a[0][0] \times b[0][0] \\
1 & 0 & 0 & 1 & c[0][0] += a[0][1] \times b[1][0] \\
2 & 0 & 0 & 2 & c[0][0] += a[0][2] \times b[2][0] \\
3 & 0 & 0 & 3 & c[0][0] += a[0][3] \times b[3][0] \\
4 & 0 & 1 & 0 & c[0][1] += a[0][0] \times b[0][1]
\end{align*}

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\text{c} \\
\begin{array}{c}
\text{a} \\
\text{b}
\end{array}
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\text{c} \\
\begin{array}{c}
\text{a} \\
\text{b}
\end{array}
\end{array}
\end{array}
\end{array}
\]

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][0] * b[0][0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>c[0][0] += a[0][1] * b[1][0]</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>c[0][0] += a[0][2] * b[2][0]</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>c[0][0] += a[0][3] * b[3][0]</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>c[0][1] += a[0][0] * b[0][1]</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>c[0][1] += a[0][1] * b[1][1]</td>
</tr>
</tbody>
</table>
iter | i | j | k | operation
---|---|---|---|---
0 | 0 | 0 | 0 | c[0][0] += a[0][0] * b[0][0]
1 | 0 | 0 | 1 | c[0][0] += a[0][1] * b[1][0]
2 | 0 | 0 | 2 | c[0][0] += a[0][2] * b[2][0]
3 | 0 | 0 | 3 | c[0][0] += a[0][3] * b[3][0]
4 | 0 | 1 | 0 | c[0][1] += a[0][0] * b[0][1]
5 | 0 | 1 | 1 | c[0][1] += a[0][1] * b[1][1]
6 | 0 | 1 | 2 | c[0][1] += a[0][2] * b[2][1]

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache
\[
c_{0,0} = a_{0,0} \times b_{0,0} \\
\]

\[
c_{0,0} = a_{0,1} \times b_{1,0} \\
\]

\[
c_{0,0} = a_{0,2} \times b_{2,0} \\
\]

\[
c_{0,0} = a_{0,3} \times b_{3,0} \\
\]

\[
c_{0,1} = a_{0,0} \times b_{0,1} \\
\]

\[
c_{0,1} = a_{0,1} \times b_{1,1} \\
\]

\[
c_{0,1} = a_{0,2} \times b_{2,1} \\
\]

\[
c_{0,1} = a_{0,3} \times b_{3,1} \\
\]

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(c_{0,0} += a_{0,0} \times b_{0,0})</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(c_{0,0} += a_{0,1} \times b_{1,0})</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>(c_{0,0} += a_{0,2} \times b_{2,0})</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>(c_{0,0} += a_{0,3} \times b_{3,0})</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(c_{0,1} += a_{0,0} \times b_{0,1})</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(c_{0,1} += a_{0,1} \times b_{1,1})</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>(c_{0,1} += a_{0,2} \times b_{2,1})</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>(c_{0,1} += a_{0,3} \times b_{3,1})</td>
</tr>
</tbody>
</table>

**Key:**
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
What is the miss rate of a?

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache
c = a * b

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][0] * b[0][0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>c[0][0] += a[0][1] * b[1][0]</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>c[0][0] += a[0][2] * b[2][0]</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>c[0][0] += a[0][3] * b[3][0]</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>c[0][1] += a[0][0] * b[0][1]</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>c[0][1] += a[0][1] * b[1][1]</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>c[0][1] += a[0][2] * b[2][1]</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>c[0][1] += a[0][3] * b[3][1]</td>
</tr>
</tbody>
</table>

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache

What is the miss rate of a?
What is the miss rate of b?
Example: Matrix Multiplication (blocking)

```c
/* multiply 4x4 matrices using blocks of size 2 */
void mm_blocking(int a[4][4], int b[4][4], int c[4][4]) {
    int i, j, k;
    int i_c, j_c, k_c;
    int B = 2;
    // control loops
    for (i_c = 0; i_c < 4; i_c += B)
    for (j_c = 0; j_c < 4; j_c += B)
    for (k_c = 0; k_c < 4; k_c += B)
        // block multiplications
        for (i = i_c; i < i_c + B; i++)
            for (j = j_c; j < j_c + B; j++)
                for (k = k_c; k < k_c + B; k++)
                    c[i][j] += a[i][k] * b[k][j];
}
```

Let's step through this to see what's actually happening
Example: Matrix Multiplication (blocking)

- Assume a tiny cache with 4 lines of 8 bytes (2 ints)
  - $S = 1$, $E = 4$, $B = 8$
- Let’s see what happens if we now use blocking
iter | i | j | k | operation | Key:
c[0][0] += a[0][0] * b[0][0]

Grey = accessed
Dark grey = currently accessing
Red border = in cache
### Operation

- **iteration 0**
  - \( i = 0, j = 0, k = 0 \)
  - \( c[0][0] += a[0][0] \times b[0][0] \)

- **iteration 1**
  - \( i = 0, j = 0, k = 1 \)
  - \( c[0][0] += a[0][1] \times b[1][0] \)

### Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$c[0][0] += a[0][0] * b[0][0]$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$c[0][0] += a[0][1] * b[1][0]$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$c[0][1] += a[0][0] * b[0][1]$</td>
</tr>
</tbody>
</table>

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
\[
\begin{align*}
\text{iter} & \quad i & \quad j & \quad k & \quad \text{operation} \\
0 & 0 & 0 & 0 & c[0][0] += a[0][0] \times b[0][0] \\
1 & 0 & 0 & 1 & c[0][0] += a[0][1] \times b[1][0] \\
2 & 0 & 1 & 0 & c[0][1] += a[0][0] \times b[0][1] \\
3 & 0 & 1 & 1 & c[0][1] += a[0][1] \times b[1][1]
\end{align*}
\]

**Key:**
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
\[
\begin{align*}
\text{iter} & \quad i & \quad j & \quad k & \quad \text{operation} \\
0 & 0 & 0 & 0 & c[0][0] += a[0][0] \ast b[0][0] \\
1 & 0 & 0 & 1 & c[0][0] += a[0][1] \ast b[1][0] \\
2 & 0 & 1 & 0 & c[0][1] += a[0][0] \ast b[0][1] \\
3 & 0 & 1 & 1 & c[0][1] += a[0][1] \ast b[1][1] \\
4 & 1 & 0 & 0 & c[1][0] += a[1][0] \ast b[0][0]
\end{align*}
\]

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
\begin{align*}
\text{iter} & \quad i & \quad j & \quad k & \quad \text{operation} \\
0 & 0 & 0 & 0 & c[0][0] += a[0][0] * b[0][0] \\
1 & 0 & 0 & 1 & c[0][0] += a[0][1] * b[1][0] \\
2 & 0 & 1 & 0 & c[0][1] += a[0][0] * b[0][1] \\
3 & 0 & 1 & 1 & c[0][1] += a[0][1] * b[1][1] \\
4 & 1 & 0 & 0 & c[1][0] += a[1][0] * b[0][0] \\
5 & 1 & 0 & 1 & c[1][0] += a[1][1] * b[1][0] \\
\end{align*}

Key:
- Grey = accessed
- Dark grey = currently accessing
- Red border = in cache
iter | i | j | k | operation
---|---|---|---|---
0 | 0 | 0 | 0 | c[0][0] += a[0][0] * b[0][0]
1 | 0 | 0 | 1 | c[0][0] += a[0][1] * b[1][0]
2 | 0 | 1 | 0 | c[0][1] += a[0][0] * b[0][1]
3 | 0 | 1 | 1 | c[0][1] += a[0][1] * b[1][1]
4 | 1 | 0 | 0 | c[1][0] += a[1][0] * b[0][0]
5 | 1 | 0 | 1 | c[1][0] += a[1][1] * b[1][0]
6 | 1 | 1 | 0 | c[1][1] += a[1][0] * b[0][1]

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache
\begin{align*}
\text{c} & = \text{a} \times \text{b} \\
\text{Key:} & \\
& \begin{array}{l}
\text{Grey} = \text{accessed} \\
\text{Dark grey} = \text{currently accessing} \\
\text{Red border} = \text{in cache}
\end{array}
\end{align*}

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(c[0][0] += a[0][0] \times b[0][0])</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(c[0][0] += a[0][1] \times b[1][0])</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(c[0][1] += a[0][0] \times b[0][1])</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(c[0][1] += a[0][1] \times b[1][1])</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(c[1][0] += a[1][0] \times b[0][0])</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(c[1][0] += a[1][1] \times b[1][0])</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(c[1][1] += a[1][0] \times b[0][1])</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(c[1][1] += a[1][1] \times b[1][1])</td>
</tr>
</tbody>
</table>
\[
\begin{array}{cccc}
\text{iter} & i & j & k & \text{operation} \\
0 & 0 & 0 & 0 & c[0][0] += a[0][0] \times b[0][0] \\
1 & 0 & 0 & 1 & c[0][0] += a[0][1] \times b[1][0] \\
2 & 0 & 1 & 0 & c[0][1] += a[0][0] \times b[0][1] \\
3 & 0 & 1 & 1 & c[0][1] += a[0][1] \times b[1][1] \\
4 & 1 & 0 & 0 & c[1][0] += a[1][0] \times b[0][0] \\
5 & 1 & 0 & 1 & c[1][0] += a[1][1] \times b[1][0] \\
6 & 1 & 1 & 0 & c[1][1] += a[1][0] \times b[0][1] \\
7 & 1 & 1 & 1 & c[1][1] += a[1][1] \times b[1][1] \\
8 & 0 & 0 & 2 & c[0][0] += a[0][2] \times b[2][0]
\end{array}
\]
\[
\begin{array}{c}
c
\end{array} = \begin{array}{c}
a
\end{array} \times \begin{array}{c}
b
\end{array}
\]

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(c[0][0] += a[0][0] \times b[0][0])</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(c[0][0] += a[0][1] \times b[1][0])</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(c[0][1] += a[0][0] \times b[0][1])</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(c[0][1] += a[0][1] \times b[1][1])</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(c[1][0] += a[1][0] \times b[0][0])</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(c[1][0] += a[1][1] \times b[1][0])</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(c[1][1] += a[1][0] \times b[0][1])</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(c[1][1] += a[1][1] \times b[1][1])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>(c[0][0] += a[0][2] \times b[2][0])</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>(c[0][0] += a[0][3] \times b[3][0])</td>
</tr>
</tbody>
</table>
\[ \begin{array}{c c c c}
\text{iter} & i & j & k \\
0 & 0 & 0 & 0 & \text{c[0][0] += a[0][0] * b[0][0]} \\
1 & 0 & 0 & 1 & \text{c[0][0] += a[0][1] * b[1][0]} \\
2 & 0 & 1 & 0 & \text{c[0][1] += a[0][0] * b[0][1]} \\
3 & 0 & 1 & 1 & \text{c[0][1] += a[0][1] * b[1][1]} \\
4 & 1 & 0 & 0 & \text{c[1][0] += a[1][0] * b[0][0]} \\
5 & 1 & 0 & 1 & \text{c[1][0] += a[1][1] * b[1][0]} \\
6 & 1 & 1 & 0 & \text{c[1][1] += a[1][0] * b[0][1]} \\
7 & 1 & 1 & 1 & \text{c[1][1] += a[1][1] * b[1][1]} \\
\end{array} \]

\[ \begin{array}{c c c c}
\text{iter} & i & j & k \\
8 & 0 & 0 & 2 & \text{c[0][0] += a[0][2] * b[2][0]} \\
9 & 0 & 0 & 3 & \text{c[0][0] += a[0][3] * b[3][0]} \\
10 & 0 & 1 & 2 & \text{c[0][1] += a[0][2] * b[2][1]} \\
\end{array} \]
<table>
<thead>
<tr>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
<th>iter</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][0] * b[0][0]</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][2] * b[2][0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>c[0][0] += a[0][1] * b[1][0]</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][3] * b[3][0]</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>c[0][1] += a[0][0] * b[0][1]</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>c[0][1] += a[0][2] * b[2][1]</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>c[0][1] += a[0][1] * b[1][1]</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>c[0][1] += a[0][3] * b[3][1]</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>c[1][0] += a[1][0] * b[0][0]</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>c[1][0] += a[1][1] * b[1][0]</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>c[1][0] += a[1][1] * b[1][0]</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>c[1][1] += a[1][0] * b[0][1]</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>c[1][1] += a[1][0] * b[0][1]</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>c[1][1] += a[1][1] * b[1][1]</td>
</tr>
<tr>
<td>iter</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][0] * b[0][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>c[0][0] += a[0][1] * b[1][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>c[0][1] += a[0][0] * b[0][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>c[0][1] += a[0][1] * b[1][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>c[1][0] += a[1][0] * b[0][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>c[1][0] += a[1][1] * b[1][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>c[1][1] += a[1][0] * b[0][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>c[1][1] += a[1][1] * b[1][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>c[0][0] += a[0][2] * b[2][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>c[0][0] += a[0][3] * b[3][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>c[0][1] += a[0][2] * b[2][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>c[0][1] += a[0][3] * b[3][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>c[1][0] += a[1][2] * b[2][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iter</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>operation</td>
<td>iter</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>operation</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>------------------------------------</td>
<td>------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][0] * b[0][0]</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>c[0][0] += a[0][2] * b[2][0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>c[0][0] += a[0][1] * b[1][0]</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>c[0][0] += a[0][3] * b[3][0]</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>c[0][1] += a[0][0] * b[0][1]</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>c[0][1] += a[0][2] * b[2][1]</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>c[0][1] += a[0][1] * b[1][1]</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>c[0][1] += a[0][3] * b[3][1]</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>c[1][0] += a[1][0] * b[0][0]</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>c[1][0] += a[1][2] * b[2][0]</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>c[1][0] += a[1][1] * b[1][0]</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>c[1][0] += a[1][3] * b[3][0]</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>c[1][1] += a[1][0] * b[0][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>c[1][1] += a[1][1] * b[1][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iter</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>operation</td>
<td>iter</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>operation</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-----------------------------</td>
<td>------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c[0][0] += a[0][0] * b[0][0]</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>c[0][0] += a[0][2] * b[2][0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>c[0][0] += a[0][1] * b[1][0]</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>c[0][0] += a[0][3] * b[3][0]</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>c[0][1] += a[0][0] * b[0][1]</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>c[0][1] += a[0][2] * b[2][1]</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>c[0][1] += a[0][1] * b[1][1]</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>c[0][1] += a[0][3] * b[3][1]</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>c[1][0] += a[1][0] * b[0][0]</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>c[1][0] += a[1][2] * b[2][0]</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>c[1][0] += a[1][1] * b[1][0]</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>c[1][0] += a[1][3] * b[3][0]</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>c[1][1] += a[1][0] * b[0][1]</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>c[1][1] += a[1][2] * b[2][1]</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>c[1][1] += a[1][1] * b[1][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iter</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>operation</td>
<td>iter</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>operation</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------------------------------</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(c[0][0] += a[0][0] \times b[0][0])</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>(c[0][0] += a[0][2] \times b[2][0])</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(c[0][0] += a[0][1] \times b[1][0])</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>(c[0][0] += a[0][3] \times b[3][0])</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(c[0][1] += a[0][0] \times b[0][1])</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>(c[0][1] += a[0][2] \times b[2][1])</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(c[0][1] += a[0][1] \times b[1][1])</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>(c[0][1] += a[0][3] \times b[3][1])</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(c[1][0] += a[1][0] \times b[0][0])</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>(c[1][0] += a[1][2] \times b[2][0])</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(c[1][0] += a[1][1] \times b[1][0])</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>(c[1][0] += a[1][3] \times b[3][0])</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(c[1][1] += a[1][0] \times b[0][1])</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>(c[1][1] += a[1][2] \times b[2][1])</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(c[1][1] += a[1][1] \times b[1][1])</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>(c[1][1] += a[1][3] \times b[3][1])</td>
</tr>
</tbody>
</table>
What is the miss rate of a?

What is the miss rate of b?
Practice Problems
Class Question / Discussions

- We’ll work through a series of questions
- Write down your answer for each question
- You can discuss with your classmates
What Type of Locality?

- The following function exhibits which type of locality? Consider only array accesses.

```c
void who(int *arr, int size) {
    for (int i = 0; i < size-1; ++i)
        arr[i] = arr[i+1];
}
```

A. Spatial
B. Temporal
C. Both A and B
D. Neither A nor B
What Type of Locality?

• The following function exhibits which type of locality? Consider only array accesses.

```c
void who(int *arr, int size) {
    for (int i = 0; i < size-1; ++i)
        arr[i] = arr[i+1];
}
```

A. Spatial
B. Temporal
C. Both A and B
D. Neither A nor B
What Type of Locality?

• The following function exhibits which type of locality? Consider only array accesses.

```c
void coo(int *arr, int size) {
    for (int i = size-2; i >= 0; --i)
        arr[i] = arr[i+1];
}
```

A. Spatial
B. Temporal
C. Both A and B
D. Neither A nor B
What Type of Locality?

• The following function exhibits which type of locality? Consider *only* array accesses.

```c
void coo(int *arr, int size) {
    for (int i = size-2; i >= 0; --i)
        arr[i] = arr[i+1];
}
```

A. Spatial
B. Temporal
C. Both A and B
D. Neither A nor B
Calculating Cache Parameters

• Given the following address partition, how many int values will fit in a single data block?

Address: 18 bits | 10 bits | 4 bits

# of int in block

A. 0
B. 1
C. 2
D. 4
E. Unknown: We need more info
Calculating Cache Parameters

• Given the following address partition, how many int values will fit in a single data block?

Address:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set index</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

# of int in block

A. 0
B. 1
C. 2
D. 4
E. Unknown: We need more info
Assuming a 32-bit address (i.e. $m=32$), how many bits are used for tag ($t$), set index ($s$), and block offset ($b$).
Assuming a 32-bit address (i.e. m=32), how many bits are used for tag (t), set index (s), and block offset (b).
Which Set Is it?

- Which set is the address 0xFA1C located in?

Set 0:
- Valid
- Tag
- Cache block

Set 1:
- Valid
- Tag
- Cache block

Set 2:
- Valid
- Tag
- Cache block

Set 3:
- Valid
- Tag
- Cache block

8 bytes per data block

E = 1 lines per set

Set # for 0xFA1C

A. 0
B. 1
C. 2
D. 3
E. More than one of the above
Which Set Is it?

- Which set is the address 0xFA1C located in?

- Set # for 0xFA1C:
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. More than one of the above
Cache Block Range

• What range of addresses will be in the same block as address \texttt{0xFA1C}? 8 bytes per data block

Set 0:  
Set 1:  
Set 2:  
Set 3:  

A. \texttt{0xFA1C}
B. \texttt{0xFA1C – 0xFA23}
C. \texttt{0xFA1C – 0xFA1F}
D. \texttt{0xFA18 – 0xFA1F}
E. It depends on the access size (byte, word, etc)
Cache Block Range

- What range of addresses will be in the same block as address \(0x\text{FA1C}\)?

\[
\begin{array}{|c|c|c|}
\hline
\text{Set 0:} & \text{Valid} & \text{Tag} & \text{Cache block} \\ 
\text{Set 1:} & \text{Valid} & \text{Tag} & \text{Cache block} \\ 
\text{Set 2:} & \text{Valid} & \text{Tag} & \text{Cache block} \\ 
\text{Set 3:} & \text{Valid} & \text{Tag} & \text{Cache block} \\ 
\hline
\end{array}
\]

Addr. Range

A. \(0x\text{FA1C}\)
B. \(0x\text{FA1C} \text{ - } 0x\text{FA23}\)
C. \(0x\text{FA1C} \text{ - } 0x\text{FA1F}\)
D. \(0x\text{FA18} \text{ - } 0x\text{FA1F}\)
E. It depends on the access size (byte, word, etc)
If $N = 16$, how many bytes does the loop access of $a$?

```c
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```
Cache Misses

If N = 16, how many bytes does the loop access of a?

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

<table>
<thead>
<tr>
<th>Accessed Bytes</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
</tr>
</tbody>
</table>
Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on ‘pass 1’?

```c
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8
    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

<table>
<thead>
<tr>
<th>Miss Rate</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 %</td>
<td>25 %</td>
<td>33 %</td>
<td>50 %</td>
<td>66 %</td>
</tr>
</tbody>
</table>

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on ‘pass 1’?
Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on ‘pass 1’?

void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8
    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}

<table>
<thead>
<tr>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>
Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on ‘pass 2’?

```c
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8
    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

<table>
<thead>
<tr>
<th>Miss Rate</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 %</td>
<td>25 %</td>
<td>33 %</td>
<td>50 %</td>
<td>66 %</td>
</tr>
</tbody>
</table>

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on ‘pass 2’?
Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on ‘pass 2’?

void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}

Miss Rate

<table>
<thead>
<tr>
<th>Option</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0 %</td>
</tr>
<tr>
<td>B</td>
<td>25 %</td>
</tr>
<tr>
<td>C</td>
<td>33 %</td>
</tr>
<tr>
<td>D</td>
<td>50 %</td>
</tr>
<tr>
<td>E</td>
<td>66 %</td>
</tr>
</tbody>
</table>

Detailed explanation in Appendix!
Appendix: C Programming Style

- Properly document your code
  - Function + File header comments, overall operation of large blocks, any tricky bits
- Write robust code – check error and failure conditions
- Write modular code
  - Use interfaces for data structures, e.g. create/insert/remove/free functions for a linked list
  - No magic numbers – use `#define` or `static const`
- Formatting
  - 80 characters per line (use Autolab’s highlight feature to double-check)
  - Consistent braces and whitespace
- No memory or file descriptor leaks
Appendix: Git Usage

- Commit early and often!
  - At minimum at every major milestone
  - Commits don’t cost anything!

- Popular stylistic conventions
  - Branches: short, descriptive names
  - Commits: A single, logical change. Split large changes into multiple commits.
  - Messages:
    - Summary: Descriptive, yet succinct
    - Body: More detailed description on what you changed, why you changed it, and what side effects it may have
Appendix: Parsing Input with fscanf

• `fscanf(FILE *stream, const char *format, ...)`
  • “scanf” but for files

• Arguments
  1. A stream pointer, e.g. from `fopen()`
  2. Format string for parsing, e.g. “%c %d,%d”
  3+. **Pointers** to variables for parsed data
    • Can be pointers to stack variables

• Return Value
  • Success: # of parsed vars
  • Failure: EOF

• `man fscanf`
Appendix: fscanf() Example

```c
FILE *pFile;
pFile = fopen("trace.txt", "r"); // Open file for reading

// TODO: Error check sys call

char access_type;
unsigned long address;
int size;

// Line format is " S 2f,1" or " L 7d0,3"
//      - 1 character, 1 hex value, 1 decimal value
while (fscanf(pFile, " %c %lx, %d", &access_type, &address, &size) > 0)
{
    // TODO: Do stuff
}

fclose(pFile); // Clean up Resources
```
Appendix: Discussion Questions

• What did the optimal transversal orders have in common?

• How does the pattern generalize to int[8][8] A and a cache that holds 4 lines each of 4 int’s?
Appendix: Blocking Example

• We have a 2D array \( \text{int}[4][4] \ A; \)
• Cache is fully associative and can hold two lines
• Each line can hold two \text{int} \ values

Consider the following:

• What is the best miss rate for traversing \( A \) once?
• What order does of traversal did you use?

• What other traversal orders can achieve this miss rate?
Appendix: Cache Misses

If there is a 48KB cache with 8 bytes per block and 3 cache lines per set, how many misses if foo is called twice? $N$ still equals 16.

<table>
<thead>
<tr>
<th>Misses</th>
<th>A</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>16</td>
</tr>
</tbody>
</table>

NOTE: This is a contrived example since the number of cache lines must be a power of 2. However, it still demonstrates an important point.

```c
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```
Appendix: Cache Misses

If there is a 48KB cache with 8 bytes per block and 3 cache lines per set, how many misses if foo is called twice? N still equals 16.

NOTE: This is a contrived example since the number of cache lines must be a power of 2. However, it still demonstrates an important point.

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```
Appendix: Very Hard Cache Problem

• We will use a direct-mapped cache with 2 sets, which each can hold up to 4 int’s.
• How can we copy A into B, shifted over by 1 position?
  • The most efficient way? (Use temp!)

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Number of misses: ||||
Could’ve been 16 misses otherwise!
We would save even more if the block size were larger, or if temp were already cached
We access the int array in strides of 8 (note the comment and the i += 8). Each block is 64 bytes, which is enough to hold 16 ints, so in each block:

| 8 ints = 32B | 8 ints = 32B |
+----------------+----------------+
|m| | | | | | | | |h| | | | | | | | |
+----------------+----------------+
|       16 ints = 64B |

The "m" denotes a miss, and the "h" denotes a hit. This pattern will repeat for the entirety of the array.

We can be sure that the second access is always a hit. This is because the first access will load the entire 64-byte block into the cache (since the entire block is always loaded if any of its elements are accessed).

So, the big question is why the first access is always a miss. To answer this, we must understand many things about the cache.

First of all, we know that s, the number of set bits, is 6, which means there are 64 sets. Since each set maps to 64 bytes (as there are b = 6 block bits), we know that every 64 * 64 bytes = 4 kilobytes we run out of sets:

64B 64B 64B 64B
+----------------+----------------+----------------+----------------+
| set 0 | set 1 |       | set 63 | set 0 |       |
+----------------+----------------+----------------+----------------+
| 64 * 64B = 4KB |

Clearly, this pattern will repeat for the entirety of the array.
Appendix: 48KB Cache Explained (2)

However, note that we have $E = 8$ lines per set. That means that even though the next 4KB map to the same sets (0-63) as the first 4KB, they will just be put in another line in the cache, until we run out of lines (i.e., after we've gone through $8 \times 4KB = 32KB$ of memory). Splitting up the bigArr into 16KB chunks:

```
16KB    16KB    16KB
+-----------------------+
| section A | section B | section C |
+-----------------------+
|                       |           |           |
```

4KB each

We see that section A will take up 16KB = $4 \times 4KB$; like we said, each of those 4KB chunks will take up 1 line each, so section A uses 4 lines per set (and uses all 64 sets).

Similarly, section B also takes up 16KB = $4 \times 4KB$; again, each of those 4KB chunks will take up 1 line each, so section B also uses 4 lines per set (and uses all 64 sets).

Note that as all of this data is being loaded in, our cache is still cold (does not contain any data from those sections), so the previous assumption about the first of every other access missing (the "m" above) is still true.

After we read in sections A and B, the cache looks like:

```
line 0 1 2 3 4 5 6 7
+-----------------------+
 0 |       |       |
 1 |       |       |
 s . . .       .       .
 e . .   A   .   B   .
 t . .       .       .
 62|       |       |
 63|       |       |
+-----------------------+
```
However, once we reach section C, we’ve run out of lines! So what do we have to do? We have to start evicting lines. And of course, the least-recently used lines are the ones used to store the data from A (lines 0-3), since we just loaded in the stuff from B. So, first of all, these evictions are causing misses on the first of every other read, so that "m" assumption is still true. Second, after we read in the entirety of section C, the cache looks like:

```
line 0 1 2 3 4 5 6 7
  +---------------+
  0 |   |   |   |
  1 |   |   |   |
  s . .   .   .   .
  e . .   C   .   B
  t . .   .   .   .
 62|   |   |   |
 63|   |   |
  +---------------+
```

Thus, we know now that the miss rate for the first pass is 50%.
Appendix: 48KB Cache Explained (4)

If we now consider the second pass, we're starting over at the beginning of bigArr (i.e., now we're reading section A). However, there's a problem - section A isn't in the cache anymore! So we get a bunch of evictions (the "m" assumption is still true, of course, since these evictions must also be misses). What are we evicting? The least-recently used lines, which are now lines 4-7 (holding data from B). Thus, the cache after reading section A looks like:

```
line 0 1 2 3 4 5 6 7
+----------+
 0 |       |       |
 1 |       |       |
s . . . C . A .
et . .       .       .
62|       |       |
63|       |       |
+----------+
```

Then, we access B. But it isn't in the cache either! So we evict the least-recently-used lines (in this case, the lines that were holding section C, 0-3) (the "m" assumption still holds); afterwards, the cache looks like:

```
line 0 1 2 3 4 5 6 7
+----------+
 0 |       |       |
 1 |       |       |
s . . . B . A .
et . .       .       .
62|       |       |
63|       |       |
+----------+
```
And finally, we access section C. But of course, its data isn't in the cache at all, so we again evict the least-recently used lines (in this case, section A's lines, 4-7) (again, "m" assumption holds):

```
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
0  |   |   |   |   |   |   |   | 0
1  |   |   |   |   |   |   |   | 1
s  |   |   |   |   |   |   |   | s
e  |   |   |   |   |   |   |   | e
B  |   |   |   |   |   |   |   | B
t  |   |   |   |   |   |   |   | t
62 |   |   |   |   |   |   |   | 62
63 |   |   |   |   |   |   |   | 63
+---+---+---+---+---+---+---+
```

And so the miss rate is 50% for the second pass as well.

Thank you to Stan Zhang for coming up with such a detailed explanation!