Instructor: Porter Jones
Teaching Assistants: Amy Xu, Callum Walker, Sam Wolfson, Tim Mandzyuk

http://xkcd.com/908/
Administrivia

❖ Questions doc: https://tinyurl.com/CSE351-8-3

❖ hw16 due Wednesday (8/5) – 10:30am
❖ hw17 due Friday (8/7) – 10:30am

❖ Lab 4 due Wednesday (8/12) – 11:59pm
  ▪ All about caches!

❖ Unit Summary 2 Due Wednesday (8/5) – 11:59pm
What about writes?

- Multiple copies of data may exist:
  - multiple levels of cache and main memory

- What to do on a write-hit?
  - **Write-through**: write immediately to next level
  - **Write-back**: defer write to next level until line is evicted (replaced)
    - Must track which cache lines have been modified ("dirty bit")

- What to do on a write-miss?
  - **Write allocate**: ("fetch on write") load into cache, then execute the write-hit policy
    - Good if more writes or reads to the location follow
  - **No-write allocate**: ("write around") just write immediately to next level

- Typical caches:
  - Write-back + Write allocate, usually
  - Write-through + No-write allocate, occasionally
Write-back, Write Allocate Example

Note: While unrealistic, this example assumes that all requests have offset 0 and are for a block’s worth of data.

There is only one set in this tiny cache, so the tag is the entire block number!
Write-back, Write Allocate Example

1) \texttt{mov} \ $0xFACE, (F) \\
Write Miss!

Not valid x86, just using block num instead of full byte address to keep the example simple

Cache:

$$\begin{array}{cccc}
\text{Valid} & \text{Dirty} & \text{Tag} & \text{Block Contents} \\
1 & 0 & G & 0xBEEF \\
\end{array}$$

Step 1: Bring F into cache

Memory:

$$\begin{array}{c}
\text{Block Num} \\
F & 0xCAFE \\
G & 0xBEEF \\
\end{array}$$
Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`

Write Miss

Cache:

```
Valid  Dirty  Tag  Block Contents
  1      0      F      0xCAFE
```

Step 1: Bring F into cache

Step 2: Write 0xCAFE to cache only and set the dirty bit

Memory:

```
Block Num
  F       0xCAFE
  G       0xBEEF
  ... ...
```
Write-back, Write Allocate Example

1) mov $0xFACE, (F)
   Write Miss

Step 1: Bring F into cache
Step 2: Write 0xFACE to cache only and set the dirty bit
Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`  
   Write Miss

2) `mov $0xFEED, (F)`  
   Write Hit!

Cache:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Block Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>F</td>
<td>0xFACE</td>
</tr>
</tbody>
</table>

Memory:

Step: Write
0xFEED to cache only (and set the dirty bit)
Write-back, Write Allocate Example

1) mov $0xFACE, (F)  
Write Miss

2) mov $0xFEED, (F)  
Write Hit

Cache:

```
<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Block Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>F</td>
<td>0xFEED</td>
</tr>
</tbody>
</table>
```

Memory:

```
Block Num:
F
G

0xCAFE
0xBEF
```
Write-back, Write Allocate Example

1) `mov $0xFACE, (F)` Write Miss
2) `mov $0xFEED, (F)` Write Hit
3) `mov (G), %ax` Read Miss!

Cache:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Block Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>F</td>
<td>0xFEED</td>
</tr>
</tbody>
</table>

Memory:

<table>
<thead>
<tr>
<th>Block Num</th>
<th>Block Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0xCAFE</td>
</tr>
<tr>
<td>G</td>
<td>0xBEEF</td>
</tr>
</tbody>
</table>

Step 1: Write F back to memory since it is dirty
Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`  
   Write Miss

2) `mov $0xFEED, (F)`  
   Write Hit

3) `mov (G), %ax`  
   Read Miss

Cache:

```
<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Block Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>G</td>
<td>0xBEEF</td>
</tr>
</tbody>
</table>
```

Step 1: Write F back to memory since it is dirty

Step 2: Bring G into the cache so that we can copy it into %ax

Memory:

```
<table>
<thead>
<tr>
<th>Block Num</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xFEED</td>
<td>0xBEEF</td>
</tr>
</tbody>
</table>
```
Cache Simulator

❖ Want to play around with cache parameters and policies? Check out our cache simulator!
  ▪ [https://courses.cs.washington.edu/courses/cse351/cachesim/](https://courses.cs.washington.edu/courses/cse351/cachesim/)

❖ Way to use:
  ▪ Take advantage of “explain mode” and navigable history to test your own hypotheses and answer your own questions
  ▪ Self-guided Cache Sim Demo posted along with Section 6
  ▪ Will be used in hw17 – Lab 4 Preparation
Polling Question [Cache IV]

❖ Which of the following cache statements is FALSE?
  ▪ Vote at http://pollev.com/pbjones

A. We can reduce compulsory misses by decreasing our block size

B. We can reduce conflict misses by increasing associativity

C. A write-back cache will save time for code with good temporal locality on writes

D. A write-through cache will always match data with the memory hierarchy level below it

E. We’re lost...
Optimizations for the Memory Hierarchy

❖ Write code that has locality!
  ▪ **Spatial**: access data contiguously
  ▪ **Temporal**: make sure access to the same data is not too far apart in time

❖ How can you achieve locality?
  ▪ Adjust memory accesses in *code* (software) to improve miss rate (MR)
    • Requires knowledge of *both* how caches work as well as your system’s parameters
  ▪ Proper choice of algorithm
  ▪ Loop transformations
Example: Matrix Multiplication

\[ C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \]
Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row major matrix in memory:
    - COLUMN of matrix (blue) is spread among cache blocks shown in red
Naïve Matrix Multiply

# move along rows of A
for (i = 0; i < n; i++)
  # move along columns of B
  for (j = 0; j < n; j++)
    # EACH k loop reads row of A, col of B
    # Also read & write c(i,j) n times
    for (k = 0; k < n; k++)
      c[i*n+j] += a[i*n+k] * b[k*n+j];
Cache Miss Analysis (Naïve)

- **Scenario Parameters:**
  - Square matrix \( (n \times n) \), elements are \textit{doubles}
  - Cache block size \( K = 64 \text{ B} = 8 \text{ doubles} \)
  - Cache size \( C \ll n \) (much smaller than \( n \))

- **Each iteration:**
  \[
  \frac{n}{8} + n = \frac{9n}{8} \text{ misses}
  \]
Cache Miss Analysis (Naïve)

❖ Scenario Parameters:
   ▪ Square matrix \( n \times n \), elements are \textit{doubles}
   ▪ Cache block size \( K = 64 \) B = 8 \textit{doubles}
   ▪ Cache size \( C \ll n \) (much smaller than \( n \))

❖ Each iteration:
   ▪ \( \frac{n}{8} + n = \frac{9n}{8} \) misses
   ▪ Afterwards \textit{in cache}:
     (schematic)

Ignoring matrix \( C \)
Cache Miss Analysis (Naïve)

- Scenario Parameters:
  - Square matrix \((n \times n)\), elements are doubles
  - Cache block size \(K = 64\) B = 8 doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

- Each iteration:
  - \(\frac{n}{8} + n = \frac{9n}{8}\) misses

- Total misses: \(\frac{9n}{8} \times n^2 = \frac{9}{8}n^3\)

Ignoring matrix \(C\) once per product matrix element
Linear Algebra to the Rescue (1)

- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix “blocks”)

- For example, multiply two 4×4 matrices:

\[
A = \begin{bmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22} \\
    a_{31} & a_{32} \\
    a_{41} & a_{42}
\end{bmatrix}
\begin{bmatrix}
    a_{13} & a_{14} \\
    a_{23} & a_{24} \\
    a_{33} & a_{34} \\
    a_{43} & a_{44}
\end{bmatrix}
= \begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix}, \text{ with } B \text{ defined similarly.}
\]

\[
AB = \begin{bmatrix}
    (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\
    (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22})
\end{bmatrix}
\]
Linear Algebra to the Rescue (2)

Matrices of size $n \times n$, split into 4 blocks of size $r$ ($n=4r$)

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} \cdot B_{k2}$$

- Multiplication operates on small “block” matrices
  - Choose size so that they fit in the cache!
  - This technique called “cache blocking”
Blocked Matrix Multiply

- Blocked version of the naïve algorithm:

```c
#define move by r x r BLOCKS now
for (i = 0; i < n; i += r) {
    for (j = 0; j < n; j += r) {
        for (k = 0; k < n; k += r) {
            # block matrix multiplication
            for (ib = i; ib < i+r; ib++) {
                for (jb = j; jb < j+r; jb++) {
                    for (kb = k; kb < k+r; kb++) {
                        c[ib*n+jb] += a[ib*n+kb]*b[kb*n+jb];
                    }
                }
            }
        }
    }
}
```

- $r$ = block matrix size (assume $r$ divides $n$ evenly)
Cache Miss Analysis (Blocked)

❖ Scenario Parameters:
  - Cache block size $K = 64$ B = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $r \times r$ fit into cache: $3r^2 < C$

❖ Each block iteration:
  - $r^2 / 8$ misses per block
  - $2n/r \times r^2 / 8 = nr/4$

$n/r$ blocks in row and column

Ignoring matrix $C$
Cache Miss Analysis (Blocked)

❖ Scenario Parameters:
  ▪ Cache block size $K = 64$ B = 8 doubles
  ▪ Cache size $C \ll n$ (much smaller than $n$)
  ▪ Three blocks $(r \times r)$ fit into cache: $3r^2 < C$

$r^2$ elements per block, 8 per cache block

❖ Each block iteration:
  ▪ $r^2/8$ misses per block
  ▪ $2n/r \times r^2/8 = nr/4$

$n/r$ blocks in row and column

❖ Afterwards in cache (schematic)
Cache Miss Analysis (Blocked)

- **Scenario Parameters:**
  - Cache block size $K = 64 \text{ B} = 8$ doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $(r \times r)$ fit into cache: $3r^2 < C$

  - $r^2$ elements per block, 8 per cache block
  - $n/r$ blocks in row and column

- **Each block iteration:**
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$

- **Total misses:**
  - $nr/4 \times (n/r)2 = n^3/(4r)$

Ignoring matrix $C$
Matrix Multiply Visualization

- Here $n = 100$, $C = 32$ KiB, $r = 30$

**Naïve:**

- $\approx 1,020,000$ cache misses

**Blocked:**

- $\approx 90,000$ cache misses
Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache-friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache size, cache block size, associativity, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code
The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Aggressive prefetching

Slopes of spatial locality

Ridges of temporal locality
Learning About Your Machine

❖ **Linux:**
  - `lscpu`
  - `ls /sys/devices/system/cpu/cpu0/cache/index0/`
    - **Example:** `cat /sys/devices/system/cpu/cpu0/cache/index*/size`

❖ **Windows:**
  - `wmic memcache get <query>` *(all values in KB)*
  - **Example:** `wmic memcache get MaxCacheSize`

❖ Modern processor specs: [http://www.7-cpu.com/](http://www.7-cpu.com/)