Caches IV
CSE 351 Summer 2020

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](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!

Not dirty so copies are consistent
Write-back, Write Allocate Example

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

Write 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>
```

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: Bring F into cache

Not valid x86, just using block num instead of full byte address to keep the example simple
Write-back, Write Allocate Example

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

\begin{itemize}
  \item Step 1: Bring F into cache
  \item Step 2: Write 0\texttt{FACE} to cache only and set the dirty bit
\end{itemize}
Write-back, Write Allocate Example

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

<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>

Step 1: Bring F into cache

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

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

2) \texttt{mov} \ $0xFEED, (F) \\
Write Hit!

\begin{align*}
\text{Cache:} & \quad \begin{array}{c|c|c|c}
\text{Valid} & \text{Dirty} & \text{Tag} & \text{Block Contents} \\
\hline
1 & 1 & F & 0xFACE \\
\end{array} \\
\end{align*}

\begin{align*}
\text{Memory:} & \quad \begin{array}{c|c|c|c}
\text{Block} & \text{Num} & \text{Contents} \\
\hline
\text{F} & \text{0xCAFE} & \cdots & \cdots \\
\text{G} & \text{0xBEED} & \cdots & \cdots \\
\end{array} \\
\end{align*}

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

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

2) \texttt{mov} $0xFEED, (F) \quad \text{Write Hit}

\begin{center}
\begin{tabular}{cccc}
\hline
Valid & Dirty & Tag & Block Contents \\
\hline
1 & 1 & F & 0xFEED \\
\hline
\end{tabular}
\end{center}

\begin{center}
Memory:
\end{center}

\begin{center}
\begin{tabular}{c}
Block Num \\
\hline
F \quad 0xCAFE \\
G \quad 0xBEF \\
\hline
\end{tabular}
\end{center}
Write-back, Write Allocate Example

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

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

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

---

**Cache:**

- Valid: 1
- Dirty: 1
- Tag: F
- Block Contents: 0xFEED

**Memory:**

- Block Num: F
- Contents: 0xCAFE

- Block Num: G
- Contents: 0xBEFF

---

**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

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
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](http://pollev.com/pbjones)
  - **A.** We can reduce compulsory misses by decreasing our block size
  - **B.** We can reduce conflict misses by increasing associativity (True! More options to place blocks before eviction occurs)
  - **C.** A write-back cache will save time for code with good temporal locality on writes (frequently used blocks are rarely evicted, fewer write-backs)
  - **D.** A write-through cache will always match data with the memory hierarchy level below it (yes, main goal is data consistency)
  - **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 doubles
  - Cache block size \( K = 64 \) \( \text{B} = 8 \) doubles
  - Cache size \( C \ll n \) (much smaller than \( n \))

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

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
  - Afterwards in cache: (schematic)
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:

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

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

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

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

- 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
# 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 $K = 64 \text{ B} = 8 \text{ doubles}$
  - 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
  - $2n/r \times r^2/8 = nr/4$

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: $3r^2 < C$

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

- Afterwards in cache (schematic)
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
  - $2n/r \times r^2/8 = nr/4$

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

$n/r$ blocks

$n/r$ blocks in row and column

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

Ignoring matrix $C$
Matrix Multiply Visualization

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

**Naïve:**

≈ 1,020,000 cache misses

**Blocked:**

≈ 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
Fall off when hierarchy size is exceeded

Working set
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/)
Write-back, Write Allocate Example

1) `mov 0xFACE, F`  
2) `mov 0xFEED, F`  
3) `mov G, %ax`

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>

Memory:

```
F
| 0xCAFE |
G
| 0xBEEF |
```
Cache Miss Analysis Comparison

- Scenario Parameters:
  - Square matrix \((n \times n)\) of doubles
  - Cache block size \(K = 64\ B = 8\) doubles
  - Cache size \(C \ll n\) and three blocks \((r \times r)\) fit into cache: \(3r^2 < C\)

- Naïve:

- Blocked:
Exceptions - Handout

- An *exception* is a transfer of control to the operating system (OS) kernel in response to some *event* *(i.e. change in processor state)*
  - Kernel is the memory-resident part of the OS
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C

- *How does the system know where to jump to in the OS?*
User Code | OS Kernel Code

event

current_instr
next_instr

exception

text

exception processing by
exception handler, then:

- return to current_instr,
- return to next_instr, OR
- abort