Caches IV
CSE 351 Winter 2020

Instructor: Ruth Anderson
Teaching Assistants: Jonathan Chen, Josie Lee, Eddy (Tianyi) Zhou
Justin Johnson, Jeffery Tian
Porter Jones, Callum Walker

http://xkcd.com/908/
Administrivia

- Lab 3 due Monday (2/24)
- hw16 due Monday (2/24)

- hw17 due Wednesday (2/26)

- Lab 4 coming soon!
  - Cache parameter puzzles and code optimizations
What about writes?

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

- What to do on a **write-hit**? (block/data already in $S$)
  - **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**? (block/data not currently in $S$)
  - **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 these copies are consistent.
Write-back, Write Allocate Example

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

Write Miss!

Step 1: Bring F into cache
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

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:

<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
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!

\textbf{Step:} Write \texttt{0xFEED} to cache only (and set the dirty bit)

\begin{itemize}
  \item Cache:
    \begin{itemize}
      \item [Valid] 1
      \item [Dirty] 1
      \item [Tag] F
      \item [Block Contents] 0xFACE
    \end{itemize}

  \item Memory:
    \begin{itemize}
      \item [Block Num] F
      \item [Contents] 0xCAFE
      \item [Block Num] G
      \item [Contents] 0xBEFF
    \end{itemize}
\end{itemize}
Write-back, Write Allocate Example

1)  mov $0xFACE, (F)   2)  mov $0xFEED, (F)
Write Miss              Write Hit

Cache:

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

Memory:

<table>
<thead>
<tr>
<th>Block Num</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0xCAFE</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0xBEEF</td>
<td></td>
</tr>
</tbody>
</table>
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
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 7
  - Will be used in hw17 – Lab 4 Preparation
Polling Question

- Which of the following cache statements is FALSE?

  - Vote at [http://pollev.com/rea](http://pollev.com/rea)
  
  A. We can reduce compulsory misses by decreasing our block size. Smaller block size pulls fewer bytes into memory on a miss.
  
  B. We can reduce conflict misses by increasing associativity. More options to place blocks before evictions occur.
  
  C. A write-back cache will save time for code with good temporal locality on writes. Frequently-used blocks rarely get evicted, so fewer write-backs are needed.
  
  D. A write-through cache will always match data with the memory hierarchy level below it. Yes, its 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

```c
# 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];
```

\[
\begin{align*}
C(i,j) & = C(i,j) + A(i,:) \times B(:,j) \\
\end{align*}
\]
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

- Ignoring matrix \(C\)

To calculate a single value in \(C\):

- \(A\) has good spatial locality
- \(B\) has bad spatial locality

By the time we get to \(n+1\), block has been kicked out of memory.
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)
  - Red showing blocks remaining in the $$

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:

\[
A = \begin{bmatrix}
 a_{11} & a_{12} \\
 a_{21} & a_{22} \\
 a_{31} & a_{32} \\
 a_{41} & a_{42}
\end{bmatrix}
= \begin{bmatrix}
 A_{11} & A_{12} \\
 A_{21} & A_{22}
\end{bmatrix}, \text{ with } B \text{ defined similarly.}
\]

\[
AB = \begin{bmatrix}
 (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\
 (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22})
\end{bmatrix}
\]
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:

```plaintext
# move by \(rxr\) 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
  - $2n/r \times r^2 / 8 = nr/4$

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 $K = 64$ 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:**
  - $nr/4 \times (n/r)^2 = n^3/(4r)$ vs. $9n^3/8$
Matrix Multiply Visualization

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

**Naïve:**

![Naïve Visualization]

$\approx 1,020,000$

cache misses.

**Blocked:**

![Blocked Visualization]

$\approx 90,000$

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

Increasing data set size

Decreasing spatial 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/)