Caches III
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Office Hours Changes – check calendar!!
- Homework 3, due TONIGHT (5/5)
- Midterm, Monday (5/8)
- Lab 3, due Thursday (5/11)
- Mid-Quarter Feedback Survey!
Question

- We have a cache of size 2 KiB with block size of 128 B. If our cache has 2 sets, what is its associativity?
  
  A. 2
  B. 4
  C. 8
  D. 16

- If addresses are 16 bits wide, how wide is the Tag field? $k = \log_2(K) = 7$ bits, $s = \log_2(S) = 1$ bit, $t = m - s - k = 8$ bits
L18: Caches III

Cache Read

1) Locate set
2) Check if any line in set is valid and has matching tag: hit
3) Locate data starting at offset

Address of byte in memory:

- t bits
- s bits
- k bits

- tag
- set index
- block offset

data begins at this offset

\( E = \text{blocks/lines per set} \)

\( S = \# \text{sets} = 2^s \)

\( K = \text{bytes per block} \)

valid bit
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set
Block Size $K = 8 \text{ B}$

$S = 2^s$ sets

Address of `int`:

8B in block

find set
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set
Block Size $K = 8$ B

![Diagram of direct-mapped cache with address, tag, valid, and match bits.]
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set
Block Size $K = 8$ B

No match? Then old line gets evicted and replaced

This is why we want alignment!

no unnecessary extra cache accesses across block boundaries
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8 \text{ B}$

Address of short int:

$8 \text{ bits} | 0...01 | 100$

find set
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

Address of short int:

valid? + match: yes = hit

block offset
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

Valid? + Match: yes = hit

Address of `short int`:

```
bits 0...01 100
```

Short int (2 B) is here

No match?
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), ...
Types of Cache Misses: 3 C’s!

- **Compulsory** (cold) miss
  - Occurs on first access to a block

- **Conflict** miss
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - *e.g.* referencing blocks 0, 8, 0, 8, ... could miss every time
  - Direct-mapped caches have more conflict misses than $E$-way set-associative (where $E > 1$)

- **Capacity** miss
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (just won’t fit, even if cache was fully-associative)
  - **Note:** *Fully-associative* only has Compulsory and Capacity misses
Core i7: Associativity

Processor package

Core 0

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

... (three other cores)

Core 3

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

L3 unified cache (shared by all cores)

Main memory

Block/line size:
- 64 bytes for all

L1 i-cache and d-cache:
- 32 KiB, 8-way,
- Access: 4 cycles

L2 unified cache:
- 256 KiB, 8-way,
- Access: 11 cycles

L3 unified cache:
- 8 MiB, 16-way,
- Access: 30-40 cycles

slower, but more likely to hit
Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - Handling writes
- Program optimizations that consider caches
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, 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, update line in cache
    - Good if more writes or reads to the location follow
  - **No-write-allocate**: ("write around") just write immediately to memory

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

Contents of memory stored at address G

valid bit not shown, but assume 1 for all of example

Cache

Contents of memory stored at address G

dirty bit

tag (there is only one set in this tiny cache, so the tag is the entire block address!)

Memory

In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.
Write-back, write-allocate example

\[ \text{mov } 0x\text{FACE}, F \]

1. Check cache for addr F \(\rightarrow\) miss
2. Fetch F from Mem (because write-allocate)

Cache

\[ \begin{array}{c|c|c}
G & 0xBEEF & 0 \\
\end{array} \]

Memory

\[ \begin{array}{c|c|c}
F & 0xCAFE & \\
G & 0xBEEF & \\
\end{array} \]

dirty bit
Write-back, write-allocate example

mov 0xFACE, F

1. Fetch block
2. Write data into block

Step 1: Bring F into cache
Write-back, write-allocate example

\texttt{mov 0xFACE, F}

Step 2: Write 0xFACE to cache only \textit{and set dirty bit}
Write-back, write-allocate example

mov 0xFACE, F  \ (write-miss)

mov 0xFEED, F  \ (write-hit)

Write hit!
Write 0xFEED to cache only
Write-back, write-allocate example

mov 0xFACE, F  mov 0xFEED, F  mov G, %rax

read miss

dirty bit

Cache

<table>
<thead>
<tr>
<th></th>
<th>0xFEED</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th>0xCAFE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0xBEEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>
Write-back, write-allocate example

mov 0xFACE, F  
mov 0xFEED, F  
mov G, %rax

1. Write F back to memory since it is dirty
2. Bring G into the cache so we can copy it into %rax
Question

❖ Which of the following cache statements is FALSE?

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.
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];
```

Diagram:
1. **Read**
   - `A(i,:)`
2. **Read**
   - `B(:,j)`
3. **Read**
   - `C(i,j)`
4. **Write (guaranteed)**
   - `C(i,j)`

Check mem access pattern.
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
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
  - Afterwards in cache: (schematic)
  - 8 doubles wide
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:

\[
A = \begin{bmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22} \\
    a_{31} & a_{32} \\
    a_{41} & a_{42}
\end{bmatrix}
= \begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix},
\]

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

- Each block iteration:
  - \( \frac{r^2}{8} \) misses per block
  - \( 2n/r \times \frac{r^2}{8} = nr/4 \)

Ignoring matrix \( c \) 

\( n/r \) blocks in row and column
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)
Matrix Multiply Visualization

- Here $n = 100$, $C = 32$ KiB, $r = 30$
  
  Naïve:
  
  \[ \approx 1,020,000 \text{ cache misses} \]

  Blocked:
  
  \[ \approx 90,000 \text{ cache misses} \]
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

L1 $ size exceeded

L2 $ size exceeded

Working data set size

Increasing

Decreasing spatial locality

Stride (x8 bytes)

Size (bytes)

Read throughput (MB/s)

Memory performance

Memory Mountain

L1

L2

L3

Mem
Learning About Your Machine

- **Linux:**
  - `lscpu`
  - `ls /sys/devices/system/cpu/cpu0/cache/index0/`
    - Ex: `cat /sys/devices/system/cpu/cpu0/cache/index*/size`
  - `cat /proc/cpuinfo | grep cache | sort | uniq`

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

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