Caches III
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma    Hunter Zahn    John Kaltenbach    Kevin Bi
Sachin Mehta Suraj Bhat    Thomas Neuman    Waylon Huang
Xi Liu      Yufang Sun
Administrivia

- Lab 3 due **Friday**
  - 1 late day = Sunday @ 5pm
- No lecture on Friday – Veteran’s Day!
- Midterm Regrades due end of Thursday
Cache Read

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

N = blocks/lines per set

S = # sets = $2^I$

Address of byte in memory:

- Tag: $T$ bits
- Set index: $I$ bits
- Block offset: $0$ bits

- Data begins at this offset

$v$ = valid bit

$K =$ bytes per block
Example: Direct-Mapped Cache ($N = 1$)

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

$S = 2^1$ sets

Address of `int`:

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

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

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

Address of `int`:

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

Block offset

![Diagram](image)
Example: Direct-Mapped Cache (N = 1)

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

No match? Then old line gets evicted and replaced

This is why we want alignment!
Example: Set-Associative Cache ($N = 2$)

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

Address of `short int`:

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

```
<table>
<thead>
<tr>
<th></th>
<th></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>v</td>
<td>tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

Address of `short int`:

Valid? + Match: yes = hit

Block offset

Compare both
Example: Set-Associative Cache ($N = 2$)

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

valid? + match: yes = hit

Address of `short int`:

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 N-way set-associative (where $N > 1$)

- **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

...  

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
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 memory and all caches in-between
  - **Write-back**: defer write to memory 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

Cache

G 0xBEEF

dirty bit

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

Memory

F

0xCAFÉ

G

0xBEEF

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

mov 0xFA1E, F

Cache

Memory

dirty bit
Write-back, write-allocate example

```c
mov 0xFACE, F
```

**Cache**

```
F  0xCAFE  0
```

**Memory**

```
F  0xCAFE
G  0xBEEF
```

**Step 1: Bring F into cache**

Dirty bit
Write-back, write-allocate example

```plaintext
mov 0xFACE, F
```
Write-back, write-allocate example

\[
\text{mov 0xFACE, F} \quad \text{mov 0xFEED, F}
\]

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

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

![Cache Diagram]

![Memory Diagram]
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
Where else is caching used?

- Software caches are more flexible
  - File system buffer caches, browser caches, etc.
  - Content-delivery networks (CDN): cache for the Internet (e.g. Netflix)

- Some design differences
  - Almost always fully-associative
    - so, no placement restrictions
    - index structures like hash tables are common (for placement)
  - More complex replacement policies
    - misses are very expensive when disk or network involved
    - worth thousands of cycles to avoid them
  - Not necessarily constrained to single “block” transfers
    - may fetch or write-back in larger units, opportunistically
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 \(K = 64\) B = 8 doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

- **First 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 \(n\))

- **Other iterations:**
  - Again:
    \[
    \frac{n}{8} + n = \frac{9n}{8}
    \]
    \(n\) misses

- **Total misses:**
  \[
  \frac{9n}{8} \times n^2 = \frac{9}{8}n^3
  \]
  once per 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:

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

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

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

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

- 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 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: $3r^2 < C$

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

- **Other (block) iterations:**
  - Same as first iteration
  - $2n/r \times r^2/8 = nr/4$

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

Ignoring matrix $c$
Matrix Multiply Summary

- Naïve: \((9/8) \times n^3\)
- Blocked: \(1/(4r) \times n^3\)
  - If \(r = 8\), difference is \(4 \times 8 \times 9/8 = 36x\)
  - If \(r = 16\), difference is \(4 \times 16 \times 9/8 = 72x\)

- Blocking optimization only works if the blocks fit in the cache
  - Suggests largest possible block size up to limit \(3r^2 \leq C\)

- Matrix multiplication has inherent temporal locality:
  - Input data: \(3n^2\), computation \(2n^3\)
  - Every array element used \(O(n)\) times!
  - But program has to be written properly
Matrix Multiply Visualization

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

**Naïve:**

- Cache misses: 551,000
- $\approx 1,020,000$ cache misses

**Blocked:**

- Cache misses: 58,000
- $\approx 90,000$ 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

Slopes of spatial locality
Aggressive prefetching
Ridges of temporal locality
L1 $ size exceeded
L2 $ size exceeded
working data set size
Size (bytes)
0 2k 4k 6k 8k 10k 12k 14k 16k
Read throughput (MB/s)
0 2k 4k 6k 8k 10k 12k 14k 16k
Stride (x8 bytes)
S1 S3 S5 S7 S9 S11 Mem L1 L2 L3
Size (bytes)
decreasing spatial locality
increasing
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/)