Caches III  
CSE 351 Winter 2017

FASTER THAN LIGHT

Administrivia

❖ Luis is sad he can’t be here this morning 😔
Giving a lecture on DNA Storage!
Example Placement

Where would data from address 0x1833 be placed?

- Binary: 0b 0001 1000 0011 0011

A-bit address: 

\[ T = A - I - O \quad I = \log_2(C/K/N) \quad O = \log_2(K) \]

Block size: 16 B
Capacity: 8 blocks
Address: 16 bits

Set: Tag (T) | Index (I) | Offset (O)
---|---|---
0 | | |
1 | | |
2 | | |
3 | | |
4 | | |
5 | | |
6 | | |
7 | | |

\[ S \leq 8 \quad I = 73 \quad \text{Direct-mapped} \]

\[ S \leq 4 \quad I = ? \quad 2\text{-way set associative} \]

\[ S = 2 \quad I = ? \quad 4\text{-way set associative} \]
Block Replacement

- Any empty block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches
  - Caches typically use something close to least recently used (LRU)
    (hardware usually implements “not most recently used”)

\[ \text{LRU takes advantage of temporal locality} \]

Cache Puzzle #2

- What can you infer from the following behavior?
  - Cache starts empty, also known as a cold cache
  - Access (addr: hit/miss) stream:
    - (10: miss), (12: miss), (10: miss)
  - Associativity?
  - Number of sets?

\[ \text{Since the 10 block was kicked out, we know} \quad N = 1 \]

\[ \text{Since 12 was a miss, 10 and 12 are not in the same block!} \quad K = 1 \text{ or } 2 \quad \text{offset} \]

\[ \text{Number of sets?} \]

\[ K = 1 \text{ byte} \quad d = 10 \text{ bits} \]
\[ (2 = 11 \text{ bits}) \]
\[ S = 1 \text{ or } 2 \quad \text{same set} \]
\[ \text{offset} \]

\[ K = 2 \text{ bytes} \quad d = 10 \text{ bits} \]
\[ (2 = 11 \text{ bits}) \]
\[ S = 1 \text{ or } 2 \quad \text{same set} \]
\[ \phi \text{ or } 0 \]
General Cache Organization \((S, N, K)\)

\[
\begin{align*}
S &= \text{# sets} = 2^I \\
N &= \text{blocks/lines per set} \\
K &= \text{bytes per block}
\end{align*}
\]

Cache size:
\[
C = S \times N \times K \quad \text{data bytes} \\
\text{(doesn't include V or Tag)}
\]

Notation Changes

- We are using different variable names from previous quarters
  - Please be aware of this when you look at past exam questions or are watching videos

<table>
<thead>
<tr>
<th>Variable</th>
<th>This Quarter</th>
<th>Previous Quarters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>(K)</td>
<td>(B)</td>
</tr>
<tr>
<td>Cache size</td>
<td>(C)</td>
<td>(--)</td>
</tr>
<tr>
<td>Associativity</td>
<td>(N)</td>
<td>(E)</td>
</tr>
<tr>
<td>Address width</td>
<td>(A)</td>
<td>(m)</td>
</tr>
<tr>
<td>Tag field width</td>
<td>(T)</td>
<td>(t)</td>
</tr>
<tr>
<td>Index field width</td>
<td>(I)</td>
<td>(k, s)</td>
</tr>
<tr>
<td>Offset field width</td>
<td>(O)</td>
<td>(n, b)</td>
</tr>
<tr>
<td>Number of Sets</td>
<td>(S)</td>
<td>(S)</td>
</tr>
</tbody>
</table>
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

Address of byte in memory:

Tag bits | Set index | Block offset
---|---|---

Data begins at this offset

valid bit

K = bytes per block

\( S = \# \text{ sets} = 2^1 \)

N = blocks/lines per set

Example: Direct-Mapped Cache \((N = 1)\)

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

Address of int:

Tag bits | Offset
---|---

find set

set index = 1

\( 2^0 \cdot 0 = 0 \)

\( K = 8 \)
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

valid? + match?: yes = hit

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:

| bits | 0..01 | 100 |

Find set
Set index = 1

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

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

Address of short int:

| bits | 0..01 | 100 |

Compare both
valid? + match: yes = hit

Block offset
Example: Set-Associative Cache \((N = 2)\)

2-way: Two lines per set
Block Size \(K = 8\) B

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
- L3 unified cache (shared by all cores)

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

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

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 0xFACE, F
Write-back, write-allocate example

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

Cache:

\begin{align*}
\text{l} & \quad \text{0xCAFE} \\
\text{F} & \quad \text{0} \quad \text{dirty bit}
\end{align*}

Step 1: Bring F into cache

Memory:

\begin{align*}
\text{F} & \quad \text{0xCAFE} \\
\text{G} & \quad \text{0xBEEF}
\end{align*}

Write-back, write-allocate example

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

Cache:

\begin{align*}
\text{l} & \quad \text{0xFACE} \\
\text{F} & \quad \text{1} \quad \text{dirty bit}
\end{align*}

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

Memory:

\begin{align*}
\text{F} & \quad \text{0xCAFE} \\
\text{G} & \quad \text{0xBEEF}
\end{align*}
Write-back, write-allocate example

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

Cache

\[
\begin{array}{c|c}
\text{F} & 0xFACE \\
\end{array}
\]

Memory

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

Write hit! Write 0xFEED to cache only

Write-back, write-allocate example

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

Cache

\[
\begin{array}{c|c}
\text{F} & 0xFEED \\
\end{array}
\]

Memory

\[
\begin{array}{c|c}
\text{F} & 0xCAFE \\
\text{G} & 0xBEEF \\
\end{array}
\]
Write-back, write-allocate example

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

Cache

| G  | 0xBEEF | 0 |

Memory

| F  | 0xFEED |
| G  | 0xBEEF |

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

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}
\]

\[
\begin{align*}
C_{ij} & = \begin{array}{c}
\text{A} \\
\text{B}
\end{array} \\
& \times
\end{align*}
\]

- \( a_{i*} \)
- \( b_{*j} \)
Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row major matrix in memory:

  ![Diagram of 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 of Naïve Matrix Multiply]
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)

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

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

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

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 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
  - $2n/r^2/8 = nr/4$

- 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: \( 3r^2 < C \)

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

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

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:**

  - Naive: $\approx 1,020,000$ 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

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/)