Caches
Review: Cache Lines

If we want to read byte $99_{10}$ and it's not in the cache, we would bring in bytes $96_{10} - 127_{10}$ into the cache.
Review: Cache Lines

If we then wanted to read byte $126_{10}$, we would get a cache hit because we just brought in the line that it's in.
Review: Cache Lines

If we then wanted to read byte $96_{10}$, we would get a cache hit because we just brought in the line that it's in.

Main Memory

Cache line size = 32 bytes
Address size = 8 bits
Fully Associative Cache Example

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $96_{10} = 0b\ 0110\ 0000$
- $99_{10} = 0b\ 0110\ 0011$
- $126_{10} = 0b\ 0111\ 1110$

- cache line size = 32 bytes
- address size = 8 bits
Fully Associative Cache Example

Valid  Dirty  Tag  Data
---  ---  ---  ---
0    0    0x3  
1    0    0x3  
0    0    0x3  
0    0    0x3  

# byte offset bits = \( \log_2(\text{line size}) = \log_2(32) = 5 \)

# tag bits = # address bits - # offset bits = 3

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset Bits</th>
<th>Tag Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>96(_{10}) = 0b 0110 0000</td>
<td>0(_{10})</td>
<td>0x3</td>
</tr>
<tr>
<td>99(_{10}) = 0b 0110 0011</td>
<td>0x3</td>
<td>3(_{10})</td>
</tr>
<tr>
<td>126(_{10}) = 0b 0111 1110</td>
<td>0x3</td>
<td>30(_{10})</td>
</tr>
</tbody>
</table>
Fully Associative Cache Example

Valid | Dirty | Tag | Data
---|---|---|---
0 | | | 
1 | 0 | 0x3 | 
0 | | | 
0 | | | 

96_{10} = 0b 0110 0000  
96_{10} = 0b 0110 0000  
99_{10} = 0b 0110 0011  
99_{10} = 0b 0110 0011  
126_{10} = 0b 0111 1110  
126_{10} = 0b 0111 1110  

# byte offset bits = \log_2(\text{line size}) = \log_2(32) = 5

# tag bits = # address bits - # offset bits = 3
Direct Mapped Cache Example

- **Valid**: Determines if the cache line is valid.
- **Dirty**: Indicates if the data is dirty.
- **LRU**: Least Recently Used, a way to implement a replacement policy.
- **Tag**: The tag is used to match the address in the cache.

Cache line size = 32 bytes
Address size = 8 bits

- **96_{10} = 0b 0110 0000**
- **99_{10} = 0b 0110 0011**
- **126_{10} = 0b 0111 1110**
Direct Mapped Cache Example

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0x0</td>
<td></td>
</tr>
</tbody>
</table>

- Valid: 0 for invalid, 1 for valid
- Dirty: 0 for clean, 1 for dirty
- LRU: Least Recently Used

- Address size = 8 bits
- Cache line size = 32 bytes

# byte offset bits = \( \log_2(\text{line size}) = \log_2(32) = 5 \)

# index bits = \( \log_2(\# \text{ lines}) = \log_2(4) = 2 \)

# tag bits = \# address bits - \# offset bits = 3
Direct Mapped Cache Example

```
Valid  Dirty LRU  Tag  Data
0      0        0    
1      0        0    
2      0        0    
3      1        0x0  
```

Cache line size = 32 bytes
Address size = 8 bits

Addresses $0110 \ 0000$ through $0110 \ 1111$ are part of the same cache line and going to map to the same index in the cache.

Addresses $1110 \ 0000$ through $1110 \ 1111$ are part of the same cache line and going to map to the same index in the cache.

These two sets of addresses will conflict with each other.

96_{10} = 0b 0110 0000
99_{10} = 0b 0110 0011
126_{10} = 0b 0111 1110
2-way Set-Associative Cache Example

Set 0
- Valid: 0
- Dirty: 0
- LRU: 0
- Tag: 0

Set 1
- Valid: 0
- Dirty: 0
- LRU: 0
- Tag: 0

Valid Data
- 96_{10} = 0b 0110 0000
- 99_{10} = 0b 0110 0011
- 126_{10} = 0b 0111 1110

Cache line size = 32 bytes
Address size = 8 bits
2-way Set-Associative Cache Example

- Cache line size = 32 bytes
- Address size = 8 bits

Set 0
- Valid: 0
- Dirty: 0
- LRU: 0

Set 1
- Valid: 0
- Dirty: 1
- LRU: 0

Addresses:
- \(96_{10} = 0b\ 0110\ 0000\)
- \(99_{10} = 0b\ 0110\ 0011\)
- \(126_{10} = 0b\ 0111\ 1110\)

- \# byte offset bits = \(\log_2(\text{line size}) = \log_2(32) = 5\)
- \# index bits = \(\log_2(\# \text{ sets}) = \log_2(2) = 1\)
- \# tag bits = \# address bits - \# offset bits = 3
### Larger 2-way Set-Associative Cache

- **Cache line size**: 32 bytes
- **Cache size**: 256 bytes

<table>
<thead>
<tr>
<th>Set</th>
<th>Valid</th>
<th>Dirty</th>
<th>LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

A set includes all of the ways of that index.

The LRU bit tells us which way is the least recently used in the set.
4-way Set-Associative Cache

A set includes all of the ways of that index

The LRU bit tells us which way is the least recently used in the set
Comparing Layouts of an 8-Block Cache

With eight blocks, an 8-way set-associative cache is same as a fully associative cache.
Pros and Cons

• Fully Associative
  • Pro: No conflicts (but you can still run out of room)
  • Con: Requires a lot of hardware to check for tag matches

• Direct Mapped
  • Pro: Only need to check one entry in the cache
  • Con: Lots of conflicts

• Set Associative
  • Pro: Less hardware than fully associative
  • Con: Still prone to conflicts (but less than direct mapped)
Recall: Single-Cycle RISC-V RV32I Datapath
Caches

- Our datapath has two memories: IMEM and DMEM
- Each of these memories have their own separate caches
Improving Cache Performance Through Programming Techniques
Array Stride

```c
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```
Array Strides

```c
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```

`sizeof(int) = 4 bytes`
`array size = 32 elements`
`Fully associative cache`
`line size = 16 bytes`
`# lines = 16`

`stride = 1`

```
my_arr
```

- Miss
- Hit
Array Strides

```c
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```

sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
# lines = 16

stride = 1

my_arr

[Diagram of array access with misses and hits]

- Miss
- Hit
Array Strides

```c
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```

- `sizeof(int) = 4 bytes`
- `array size = 32 elements`
- `Fully associative cache`
- `line size = 16 bytes`
- `# lines = 16`
- `stride = 2`
- `my_arr`
Array Strides

```
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```

sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
# lines = 16

stride = 2

my_arr

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Miss
Hit
Brought in, but unused
Array Strides

```c
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```

```plaintext
size of (int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
# lines = 16
```

```
my_arr
```

```
Miss
Hit
Brought in, but unused
```
Array Strides

```c
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```

`sizeof(int) = 4 bytes`

Array size = 32 elements

Fully associative cache

Line size = 16 bytes

# Lines = 16

Stride = 4

### `my_arr`

- Miss
- Hit
- Brought in, but unused
Array Strides

```c
int sum_array(int *my_arr, int size, int stride) {
    int sum = 0;
    for (int i = 0; i < size; i += stride) {
        sum += my_arr[i];
    }
    return sum;
}
```

sizeof(int) = 4 bytes
array size = 32 elements

Fully associative cache
line size = 16 bytes
# lines = 16

stride = 4

If the stride >= block size, you don’t take advantage of bringing in an entire line

- Miss
- Hit
- Brought in, but unused
Matrix Multiply

\[
\begin{array}{ccc}
A & X & Y \\
X & Y & C
\end{array}
\]

\[
\begin{array}{ccc}
A & X & Y \\
X & Y & C
\end{array}
\]

= 

\[
\begin{array}{ccc}
A & X & Y \\
X & Y & C
\end{array}
\]
Matrix Multiply

A

X

Y

B

C

X

Y

X

Y

X

Y

=
Matrix Multiply

A

B

C

\[ \begin{align*}
X & \quad Y \\
X & \quad Y \\
X & \quad Y \\
\end{align*} \]

\[ \begin{align*}
X & \quad Y \\
X & \quad Y \\
X & \quad Y \\
\end{align*} \]

\[ \begin{align*}
X & \quad Y \\
X & \quad Y \\
X & \quad Y \\
\end{align*} \]
Matrix Multiply
Matrix Multiply

matrix of integers
sizeof(int) = 4 bytes

Fully associative cache
line size = 16 bytes
# lines = 16
Matrix Multiply

- Arrays are stored in row-major order
Matrix Multiply

| Miss | Hit | Brought in, but unused before eviction |

Fully associative cache

line size = 16 bytes

# lines = 16
Matrix Multiply

Miss

Hit

Brought in, but unused before eviction

Fully associative cache
line size = 16 bytes
# lines = 16
Matrix Multiply

A \times B = C

Miss
Hit
Brought in, but unused before eviction

Fully associative cache
line size = 16 bytes
# lines = 16
Making better use of the cache by transposing the matrix

Miss

Hit

Brought in, but unused for this step

Fully associative cache
line size = 16 bytes
# lines = 16
Making better use of the cache by transposing the matrix

Miss
Hit
Still in cache, but unused for this step

Fully associative cache
line size = 16 bytes
# lines = 16
Cache Blocking
Cache Blocking

- A technique where data accesses are rearranged to make better use of the data that is brought into the cache
- Helps prevent repeatedly evicting and fetching the same data from the main memory
Matrix Transpose

integer matrix
sizeof(int) = 4 bytes

Fully associative cache
line size = 16 bytes
# lines = 16
Matrix Transpose

\[
\begin{align*}
A & \quad \rightarrow \\ X & \quad Y \\
Y & \quad X
\end{align*}
\]

\[
\begin{align*}
\mathbf{A}^T & \\
X & \quad Y
\end{align*}
\]

Current Transpose
Matrix Transpose

Current Transpose
Matrix Transpose

Current Transpose
Matrix Transpose

- Miss
- Hit
- Brought in, but unused before eviction
Matrix Transpose

Current Transpose
Matrix Transpose

\begin{align*}
\text{Current Transpose} & \quad \text{Already Transposed}
\end{align*}
Matrix Transpose

Current Transpose

Already Transposed

$A^T$
Matrix Transpose

Current Transpose

Miss

Hit
Matrix Transpose

Current
Transpose

Already Transposed

Miss
Hit

\[
A \quad A^T
\]

\[
A \quad A^T
\]
Matrix Transpose

Current Transpose

Already Transposed

Miss

Hit
Analyzing Cache Performance
Terminology

• **Hit Rate**
  • number of hits / number of accesses

• **Miss Rate**
  • 1 - hit rate

• **Hit Time**
  • The time that it takes for you to access an item on a cache hit

• **Miss penalty**
  • On a miss, the time it takes to access the block after discovering that its not in the cache
Average Memory Access Time (AMAT)

- $\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$
AMAT Example

• What is the AMAT of a system where
  • Hit rate = 90%
  • Hit time = 4 cycles
  • Miss penalty = 20 cycles

\[
\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty}
\]
\[
\text{AMAT} = 4 \text{ cycles} + 0.1(20 \text{ cycles})
\]
\[
\text{AMAT} = 6 \text{ cycles}
\]
AMAT Example (Your turn)

• What is the AMAT of a system where
  • Hit rate = 75%
  • Hit time = 5 cycles
  • Miss penalty = 24 cycles
AMAT Example (Your turn)

- What is the AMAT of a system where
  - Hit rate = 75%
  - Hit time = 5 cycles
  - Miss penalty = 24 cycles

\[
\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty}
\]

\[
\text{AMAT} = 5 \text{ cycles} + 0.25(24 \text{ cycles})
\]

\[
\text{AMAT} = 11 \text{ cycles}
\]
How Does Associativity Affect AMAT?

- Hit time as associativity increases?
  - Increases
  - Direct Mapped -> 2-way
    - Introduce a multiplexor to choose correct way
  - 2-way -> 4-way
    - Smaller increase than direct mapped -> 2-way
    - The multiplexor is larger for 4-way

- Miss rate as associativity increases?
  - Decreases due to less conflict misses

- Miss penalty as associativity changes?
  - Mostly unchanged, replacement policy runs in parallel with fetching missing line from memory
How does \#entries affect AMAT?

- Hit time as \#entries increases?
  - Increases, since reading tags and data from larger memory structures

- Miss rate as \#entries increases?
  - Goes down due to increased capacity and fewer conflict misses

- Miss penalty as \#entries increases?
  - Unchanged

- At some point, the increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance
How does block size affect AMAT?

- **Hit time as block size increases?**
  - Hit time mostly unchanged, but might be slightly reduced as number of tags is reduced

- **Miss rate as block size increases?**
  - Goes down at first due to spatial locality, then increases due to increased conflict misses due to fewer blocks in the cache

- **Miss penalty as block size increases?**
  - Rises with larger block size
Another way to reduce miss penalty

• Include another cache!
Memory Hierarchy

Registers
Cache
Main memory
Disk

L1 Cache
L2 Cache
Main memory
Disk
L2 Cache

- L2 is bigger than L1
  - Leads to higher hit rate
- L2 is accessed only if the requested data in not found in L1
- L2 takes longer to access because it is larger and farther away from the processor
- All data in L1 can be found in L2 as well*
- If the line in L1 is dirty when it is evicted, you update the copy in L2*

* depends on policy
Additional Caches

- L1 cache
  - Embedded in the processor chip
  - Fast, but limited storage capacity

- L2 cache
  - Embedded on the processor chip OR on its own separate chip
  - Reduces L1 miss penalty

- L3 cache
  - On a separate chip
  - Reduces L1 and L2 miss penalty

- L4 cache (uncommon)
Core i7-6500U Cache Info

https://uops.info/cache.html#SKL

Core i7-6500U (Skylake)

- L1 data cache
  - Size: 32 kB
  - Associativity: 8
  - Number of sets: 64
  - Way size: 4 kB
  - Latency: 4 cycles [Link]
  - Replacement policy: Tree-PLRU (with linear insertion order if empty) [Link 1] [Link 2]

- L2 cache
  - Size: 256 kB
  - Associativity: 4
  - Number of sets: 1024
  - Way size: 64 kB
  - Latency: 12 cycles [Link]
  - Replacement policy: QLRU_H00_M1_R2_U1 [Link]
    - Similar to the Cannon Lake L2 policy, but:
      - If the cache is empty (after executing the WBINVD instruction), blocks are inserted from right to left
      - The initial ages of blocks inserted into an empty cache can depend on the previous state
      - See also [Vila et al.]

- L3 cache
  - Size: 4 MB
  - Associativity: 16
  - Number of CBoxes: 2
  - Number of slices: 4
  - Number of sets (per slice): 1024
  - Way size (per slice): 64 kB
  - Latency: 34 cycles [Link]
  - Replacement policy: Adaptive [Link]
Local vs Global Hit Rate

• Local Hit Rate
  • # hits at this level / # accesses to this level

• Global Hit rate
  • # hits at this level / # total number of accesses
AMAT with 2-level Cache

AMAT = hit time + miss rate * miss penalty

** all miss rates are local
AMAT with 2-level Cache

\[
AMAT = \text{hit time} + \text{miss rate} \times \text{miss penalty}
\]

\[
AMAT = L1 \text{ hit time} + L1 \text{ miss rate} \times L1 \text{ miss penalty}
\]

L1 miss penalty = L2 hit time + L2 miss rate \times L2 miss penalty

\[
AMAT = L1 \text{ hit time} + L1 \text{ miss rate} \times (L2 \text{ hit time} + L2 \text{ miss rate} \times L2 \text{ miss penalty})
\]

** all miss rates are local**
AMAT Example

• What is the AMAT of a system where
  • L1 hit rate = 75%
  • L1 hit time = 4 cycles
  • L2 hit rate = 90%
  • L2 hit time = 6 cycles
  • L2 miss penalty = 20 cycles

\[
\text{AMAT} = L1 \text{ hit time} + L1 \text{ miss rate} \times (L2 \text{ hit time} + L2 \text{ miss rate} \times L2 \text{ miss penalty})
\]

\[
\text{AMAT} = 4 + 0.25 \times (6 \text{ cycles} + 0.1 \times 20 \text{ cycles})
\]

\[
\text{AMAT} = 6 \text{ cycles}
\]
AMAT Example (Your turn)

- What is the AMAT of a system where
  - L1 hit rate = 60%
  - L1 hit time = 5 cycles
  - L2 hit rate = 95%
  - L2 hit time = 8 cycles
  - L2 miss penalty = 40 cycles

AMAT = L1 hit time + L1 miss rate * (L2 hit time + L2 miss rate * L2 miss penalty)
AMAT Example (Your turn)

- What is the AMAT of a system where
  - L1 hit rate = 60%
  - L1 hit time = 5 cycles
  - L2 hit rate = 95%
  - L2 hit time = 8 cycles
  - L2 miss penalty = 40 cycles

\[
\text{AMAT} = \text{L1 hit time} + \text{L1 miss rate} \times (\text{L2 hit time} + \text{L2 miss rate} \times \text{L2 miss penalty})
\]

\[
\text{AMAT} = 5 + 0.4 \times (8 \text{ cycles} + 0.05 \times 40 \text{ cycles})
\]

\[
\text{AMAT} = 9 \text{ cycles}
\]
Learn about your computer’s caches

- **MacOS:** `sysctl -a hw machdep.cpu`
- **Linux:** `lscpu`
- **Windows:** `wmic memcache list brief`
  - (I don’t know if this actually works...)