Systems I

Cache Organization

Topics
- Generic cache memory organization
- Direct mapped caches
- Set associative caches
- Impact of caches on programming
Cache Vocabulary

Capacity
Cache block (aka cache line)
Associativity
Cache set
Index
Tag
Hit rate
Miss rate
Replacement policy
General Org of a Cache Memory

Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.

$S = 2^s$ sets

$1$ valid bit per line $t$ tag bits per line $B = 2^b$ bytes per cache block

set 0:

valid | tag | 0 1 $\cdots$ B–1

set 1:

valid | tag | 0 1 $\cdots$ B–1

set S-1:

valid | tag | 0 1 $\cdots$ B–1

Cache size: $C = B \times E \times S$ data bytes
Addressing Caches

Address A:

- **t bits**
- **s bits**
- **b bits**

The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

---

set 0:

- v
- tag 0 1 ... B-1
- ...

set 1:

- v
- tag 0 1 ... B-1
- ...

set S-1:

- v
- tag 0 1 ... B-1
- ...

m-1

0
Direct-Mapped Cache

Simplest kind of cache
Characterized by exactly one line per set.

set 0: $\text{valid} \quad \text{tag} \quad \text{cache block}$

set 1: $\text{valid} \quad \text{tag} \quad \text{cache block}$

$\ldots$

set S-1: $\text{valid} \quad \text{tag} \quad \text{cache block}$

$E=1$ lines per set
Accessing Direct-Mapped Caches

Set selection

- Use the set index bits to determine the set of interest.

```plaintext
m-1 tag  set index  block offset

0 0 0 0 1

t bits s bits b bits
```

- Selected set

```
set 0: valid tag cache block
set 1: valid tag cache block
...
set S-1: valid tag cache block
```
Accessing Direct-Mapped Caches

Line matching and word selection

- **Line matching**: Find a valid line in the selected set with a matching tag
- **Word selection**: Then extract the word

1. The valid bit must be set
2. The tag bits in the cache line must match the tag bits in the address
3. If (1) and (2), then cache hit, and block offset selects starting byte.
# Direct-Mapped Cache Simulation

M=16 byte addresses, B=2 bytes/block, 
S=4 sets, E=1 entry/set

Address trace (reads):
0 \([0000_2]\), 1 \([0001_2]\), 13 \([1101_2]\), 8 \([1000_2]\), 0 \([0000_2]\)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
</tbody>
</table>

(1) 0 \([0000_2]\) \((\text{miss})\)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>M[8-9]</td>
</tr>
</tbody>
</table>

(4) 8 \([1000_2]\) \((\text{miss})\)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
</tbody>
</table>

(5) 0 \([0000_2]\) \((\text{miss})\)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>M[12-13]</td>
</tr>
</tbody>
</table>

(3) 13 \([1101_2]\) \((\text{miss})\)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>M[12-13]</td>
</tr>
</tbody>
</table>

(3) 13 \([1101_2]\) \((\text{miss})\)
Why Use Middle Bits as Index?

High-Order Bit Indexing
- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing
- Consecutive memory lines map to different cache lines
- Can hold C-byte region of address space in cache at one time
Set Associative Caches

Characterized by more than one line per set

```
set 0:
valid  tag  cache block
valid  tag  cache block
set 1:
valid  tag  cache block
valid  tag  cache block
...  
set S-1:
valid  tag  cache block
valid  tag  cache block
```

$E=2$ lines per set
Accessing Set Associative Caches

Set selection

- identical to direct-mapped cache

#### Diagram

- **Selected set**
  - `t` bits
  - `s` bits
  - `b` bits
- **Set 0**:
  - `valid`
  - `tag`
  - Cache block
- **Set 1**:
  - `valid`
  - `tag`
  - Cache block
- **Set S-1**:
  - `valid`
  - `tag`
  - Cache block

### Details
- `m-1` to `tag`
- `s` bits to `set index`
- `b` bits to `block offset`
Accessing Set Associative Caches

Line matching and word selection

- must compare the tag in each valid line in the selected set.

1. The valid bit must be set.
2. The tag bits in one of the cache lines must match the tag bits in the address.
3. If (1) and (2), then cache hit, and block offset selects starting byte.
Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time

- Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
  - 1-3 clock cycle for L1
  - 5-12 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
  - Typically 100-300 cycles for main memory
Memory System Performance

Average Memory Access Time (AMAT)

\[ T_{\text{access}} = (1 - p_{\text{miss}})t_{\text{hit}} + p_{\text{miss}}t_{\text{miss}} \]

\[ t_{\text{miss}} = t_{\text{hit}} + t_{\text{penalty}} \]

Assume 1-level cache, 90% hit rate, 1 cycle hit time, 200 cycle miss penalty

AMAT = 21 cycles!!! - even though 90% only take one cycle
Memory System Performance - II

How does AMAT affect overall performance?

Recall the CPI equation (pipeline efficiency)

\[ CPI = 1.0 + lp + mp + rp \]

- Load/use penalty \( (lp) \) assumed memory access of 1 cycle
- Further - we assumed that all load instructions were 1 cycle
- More realistic AMAT (20+ cycles), really hurts CPI and overall performance

<table>
<thead>
<tr>
<th>Cause</th>
<th>Name</th>
<th>Instruction Frequency</th>
<th>Condition Frequency</th>
<th>Stalls</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>( lp )</td>
<td>0.30</td>
<td>0.7</td>
<td>21</td>
<td>4.41</td>
</tr>
<tr>
<td>Load/Use</td>
<td>( lp )</td>
<td>0.30</td>
<td>0.3</td>
<td>21+1</td>
<td>1.98</td>
</tr>
<tr>
<td>Mispredict</td>
<td>( mp )</td>
<td>0.20</td>
<td>0.4</td>
<td>2</td>
<td>0.16</td>
</tr>
<tr>
<td>Return</td>
<td>( rp )</td>
<td>0.02</td>
<td>1.0</td>
<td>3</td>
<td>0.06</td>
</tr>
<tr>
<td>Total penalty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.61</td>
</tr>
</tbody>
</table>
Memory System Performance - III

\[ T_{\text{access}} = (1 - p_{\text{miss}}) t_{\text{hit}} + p_{\text{miss}} t_{\text{miss}} \]

\[ t_{\text{miss}} = t_{\text{hit}} + t_{\text{penalty}} \]

How to reduce AMAT?
- Reduce miss rate
- Reduce miss penalty
- Reduce hit time

There have been numerous inventions targeting each of these
Writing Cache Friendly Code

Can write code to improve miss rate

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Examples:
  ■ cold cache, 4-byte words, 4-word cache blocks

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%  Miss rate = 100%
Questions to think about

What happens when there is a miss and the cache has no free lines?
- What do we evict?

What happen on a store miss?

What if we have a multicore chip where the processing cores share the L2 cache but have private L1 caches?
- What are some bad things that could happen?
Concluding Observations

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 sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)