Memory & Cache

Zhaoguo Wang
Question

For instruction `movq (%rax), %rbx`, how many memory accesses need to be made?
Question

For instruction `movq (%rax), %rbx`, how many memory accesses need to be made?

Can we avoid all memory accesses?
Principle of locality

Temporal locality
- If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.

Spatial locality
- If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.
- Sequential locality
  - occurs when data elements are arranged and accessed linearly
Basic idea – caching

CPU

rax
rbx

Cache

Smaller,
Faster,
Closer to CPU

Memory
Basic idea – caching

CPU
rax
rbx

Cache

Memory

Buffer d in the cache
Send the data d to cpu
Access the data d

Smaller, Faster, Closer to CPU
Basic idea – caching

Cache

CPU

rax
rbx

Send the buffered d

Access the data d

Smaller, Faster, Closer to CPU

Memory
Basic idea – caching

Cache

CPU

rax d
rbx

Send the buffered d

Access the data d

~ 4 cycles

Smaller, Faster, Closer to CPU

100 ~ 200 cycles

Memory
Intuitive implementation

Caching at byte granularity:
- Search the cache for each byte access
  - `movq (%rax), %rbx` causes 8 times check
- High cost to maintain the address information

<table>
<thead>
<tr>
<th>PA</th>
<th>1 byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>…</td>
</tr>
<tr>
<td>0x101</td>
<td>…</td>
</tr>
<tr>
<td>0x102</td>
<td>…</td>
</tr>
<tr>
<td>0x103</td>
<td>…</td>
</tr>
</tbody>
</table>
Caching at block granularity

Observation
- Spatial locality

Solution
- Caching at block granularity
- Each block is called cache line, which has 64 bytes
Direct-mapped cache

Caching at block granularity
- Each cache line has 64 bytes
Direct-mapped cache

Caching at block granularity
- Each cache line has 64 bytes

1. In cache? No
2. Fetch
3. Return
4. Buffer the cacheline, send 0x1 to the CPU
Direct-mapped cache

Caching at block granularity
- Each cache line has 64 bytes
Check and identify the location

1. Use the index bits[6:7] to find the cache line which may buffer the data.
2. Compare the cache line tag bits[8:63]
3. On cache hit, use the offset bits[0:5] to find the data in the cache line.
4. On cache miss, fetch the data from memory.

CPU access data at (PA)

CPU Cache (64 bytes cache line)
Check and identify the location

1. Use the index bits [6:7] to find the cache line which may buffer the data.

CPU access data at (0x48 PA)

CPU Cache (64 bytes cache line)
Check and identify the location

CPU access data at (0x48 PA)

1. Use the index bits[6:7] to find the cache line which may buffer the data.
2. Compare the cache line tag bits[8:63] (Cache miss)

CPU Cache (64 bytes cache line)
Check and identify the location

1. Use the index bits [6:7] to find the cache line which may buffer the data.
2. Compare the cache line tag bits [8:63] (Cache miss)
3. Load 64 bytes from 0x40

CPU access data at (0x48 PA)

<table>
<thead>
<tr>
<th>Tag</th>
<th>64 (2^6) bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x0</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

CPU Cache (64 bytes cache line)
Check and identify the location

1. Use the index bits[6:7] to find the cache line which may buffer the data.
2. Compare the cache line tag bits[8:63] (Cache miss)
3. Load 64 bytes from 0x40
4. Send the data at the offset of 0x8 in buffered cache line.
Issue I

Access pattern:
- access 0x40  
- access 0x140
- access 0x40
- access 0x140

Cache line tag: 0
Cache line index: 1

Load 64 bytes start at 0x40 into 1st entry
### Issue I

**Access pattern:**
- access **0x40**  cache miss
- access **0x140**  cache miss
- access **0x40**
- access **0x140**

**Cache line tag:** 1  
**Cache line index:** 1

**Evict** old cache line (1st entry),
**Load** 64 bytes start at **0x140** into 1st entry
**Issue I**

Access pattern:

- access 0x40  cache miss
- access 0x140 cache miss
- access 0x40  cache miss
- access 0x140

Cache line tag: 0
Cache line index: 1

**Evict** old cache line (1\textsuperscript{st} entry),

**Load** 64 bytes start at 0x40 into 1\textsuperscript{st} entry
### Issue I

**Access pattern:**

- access **0x40** cache miss
- access **0x140** cache miss
- access **0x40** cache miss
- access **0x140** cache miss

**Cache line tag:** 1  
**Cache line index:** 1

<table>
<thead>
<tr>
<th>Tag</th>
<th>64 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**CPU Cache**

Evict old cache line (1\textsuperscript{st} entry),  
Load 64 bytes start at **0x140** into 1\textsuperscript{st} entry
Multi-way set associative cache

CPU Cache
Multi-way set associative cache

Access pattern:
- access 0x40  miss
- access 0x440
- access 0x40
- access 0x440

63:0

CPU Cache

Set Index  Cache line offset

Set 0:
- Tag    Cache line
- Tag    Cache line

Set 1:
- Tag    0x0    ...
- Tag    Cache line

Set 15:
- Tag    Cache line
- Tag    Cache line

0x0  0x1  0x0

Tag  Set Index  Cache line offset

0x40  0x440  0x40  0x440
Multi-way set associative cache

Access pattern:
- access 0x40  miss
- access 0x440  miss
- access 0x40
- access 0x440

Access pattern:
- access 0x40  miss
- access 0x440  miss
- access 0x40
- access 0x440

CPU Cache
Multi-way set associative cache

Access pattern:
- access 0x40 miss
- access 0x440 miss
- access 0x40 hit
- access 0x440 hit

CPU Cache

Set 0
- v Tag Cache line
- v Tag Cache line

Set 1
- v 0x0 ...
- v 0x1 ...

Set 15
- v Tag Cache line
- v Tag Cache line

Tag Set Index Cache line offset
Multi-way set associative cache

Access pattern:
access 0x40
access 0x440
access 0x840

Which cache line in set 1 should be evicted?
Cache line replacement policy

LFU (least-frequently-used)
- Replace the line that has been referenced the fewest times over some past time window

LRU (least-recently-used)
- Replace the line that has the furthest access in the past

All of these policies require additional time and hardware
Issue II

Can not access cache before the address translation.
Low latency cache

Virtual Index and Physical Tag
- Use VA to index set, calculate the tag from PA
- Cache set lookup can be parallel with address translation

1. Virtual Address
2. Physical Address → Set Index
3. Physical Address → Tag

<table>
<thead>
<tr>
<th>Set 0</th>
<th>v Tag</th>
<th>Cache line</th>
<th>v Tag</th>
<th>Cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>v Tag</td>
<td>Cache line</td>
<td>v Tag</td>
<td>Cache line</td>
</tr>
<tr>
<td>.......</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 15</td>
<td>v Tag</td>
<td>Cache line</td>
<td>v Tag</td>
<td>Cache line</td>
</tr>
</tbody>
</table>

CPU Cache

Memory

64 bytes data
Virtual Index and Physical Tag

2-way set associative cache

Set 0
- v Tag Cache line
- v Tag Cache line

Set 1
- v Tag Cache line
- v Tag Cache line

Set 15
- v Tag Cache line
- v Tag Cache line

CPU Cache
Virtual Index and Physical Tag

access va 0x1040  pa 0xffff0040

2-way set associative cache

VA: 

PA: 

0x3fff01

Cache line offset

CPU Cache
Memory hierarchy

CPU
- rax
- rbx

~ 4 cycles
L1 Cache
- Virtual Index
- Physical Tag

~ 12 cycles
L2 Cache
- Virtual/Physical Index
- Physical Tag

~ 35 cycles
L3 Cache
- Physical/Virtual Index
- Physical Tag

~ 150 cycles
Memory