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

Buffer d in the cache

Send the data d to cpu

Cache

rax
rbx

Smaller, Faster, Closer to CPU

Access the data d

Memory
Basic idea – caching

CPU

Send the buffered \( d \)

Access the data \( d \)

Cache

Smaller, Faster, Closer to CPU

Memory
Basic idea – caching

CPU

Cache

Memory

Send the buffered d

Access the data d

~ 4 cycles

Smaller, Faster, Closer to CPU

100 ~ 200 cycles
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

CPU access 0x1234

MMU

0x1234

CPU Cache

PA | 64 bytes
---|---
0x100 | …
0x40 | …
0x80 | …
0xc0 | …

Memory

0x1

0x0...088
0x0...080
0x0...078
0x0...070
0x0...068
0x0...060
0x0...058
0x0...050
0x0...048
0x0...040
...
Direct-mapped cache

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

1. Locate the data in the cache
   a. Calculate cache line tag
      \[ \text{PA} >> 6 \rightarrow 0x1 \ (0x48 >> 6) \]

CPU access data at 0x48 (PA)

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

1. Locate the data in the cache
   a. Calculate cache line tag
      - PA >> 6 \(\rightarrow\) 0x1 (0x48 >> 6)
   b. Calculate cache line index
      - tag \% number of entries
      \(\rightarrow\) 0x1 \% 4 \(\rightarrow\) 1

CPU access data at 0x48 (PA)

<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 (64 bytes cache line)
Check and identify the location

1. Locate the data in the cache
   a. Calculate cache line tag
      - PA >> 6 → 0x1 (0x48 >> 6)
   b. Calculate cache line index
      - tag % number of entries → 0x1 % 4 → 1
   c. Compare with the tags
      - Cache[1].tag == tag ???

CPU access data at 0x48 (PA)

<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 (64 bytes cache line)
Check and identify the location

1. Locate the data in the cache
   a. Calculate cache line tag
      - $\text{PA} >> 6 \rightarrow 0x1 (0x48 >> 6)$
   b. Calculate cache line index
      - tag $\%$ number of entries
      - $0x1 \% 4 \rightarrow 1$
   c. Compare with the tags
      - $\text{Cache[1].tag} == \text{tag}$
   d. On cache hit
      - Locate data with the offset
      - $0x48 \& 0x3f \rightarrow 0x8$
Load memory on cache miss

2. Load the data in the cache
   a. Calculate cache line tag
      - PA >> 6 → 0x1 (0x48 >> 6)

CPU access data at 0x48 (PA)

<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 (64 bytes cache line)
Load memory on cache miss

2. Load the data in the cache
   a. Calculate cache line tag
      - PA >> 6 → 0x1 (0x48 >> 6)
   b. Calculate cache line index
      - tag % number of entries
        → 0x1 % 4 → 1
Load memory on cache miss

2. Load the data in the cache
   a. Calculate cache line tag
      - PA >> 6 → 0x1 (0x48 >> 6)
   b. Calculate cache line index
      - tag % number of entries → 0x1 % 4 → 1
   c. Load 64 bytes from 0x40
      → 0x48 & ~0x2f
Issue I

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

Cache line tag: 0x40 >> 6 → 0x1  
Cache line index: 0x1 % 4 → 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: 0x140 >> 6 → 0x5
Cache line index: 0x5 % 4 → 1

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

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

Cache line tag: 0x40 >> 6 → 0x1
Cache line index: 0x1 % 4 → 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: \(0x140 \gg 6 \rightarrow 0x5\)
Cache line index: \(0x5 \% 4 \rightarrow 1\)

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

2-way set associative cache
Multi-way set associative cache

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

```
0x0 0x1 0x0
```

```
Set 0

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Tag</td>
<td>Cache line</td>
</tr>
</tbody>
</table>

Set 1

| v | 0x0 | ... |

Set 15

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Tag</td>
<td>Cache line</td>
</tr>
</tbody>
</table>
```

CPU Cache
Multi-way set associative cache

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

CPU Cache

63 9 6 5 0

Tag  Set Index  Cache line offset

Set 0

Set 1

Set 15

……

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
Multi-way set associative cache

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

Why 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 → Tag
3. Physical Address

CPU Cache

CPU

MMU

Set 0

Set 1

Set 15

0x1

0x0...088
0x0...080
0x0...078
0x0...070
0x0...068
0x0...060
0x0...058
0x0...050
0x0...048
0x0...040
0x0...032
0x0...024
0x0...016
0x0...008
0x0...000

Memory

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

64 bytes data
Virtual Index and Physical Tag

CPU Cache

Set Index

Cache line offset

Tag

Cache line offset

2-way set associative cache

Set 0

Set 1

Set 15

VA:

PA:
Virtual Index and Physical Tag

access va 0x1040 pa 0xffff0040

VA:   63
PA:  0x3fffc01

Cache line offset

2-way set associative cache

Set 0
Set 1
Set 15

CPU Cache
Memory hierarchy

CPU

- L1 Cache
  - Virtual Index
  - Physical Tag
  - ~ 4 cycles

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

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

- Memory
  - ~ 150 cycles
Writing Cache-Friendly Code

Principles

– Programs with better locality will tend to have lower miss rates

– Programs with lower miss rates will tend to run faster than programs with higher miss rates

  • Miss rate – fraction of memory references not found in cache (misses/references)
  • Typical numbers: 3-10% for L1, can be quite small (<1%) for L2, depending on size
Sum of 2 D arrays

```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;
}
```
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;
}