CSCI-UA.0201

Computer Systems Organization

Memory Management – Caching

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Principle of Locality

• Temporal locality
  – If memory location $x$ is referenced, then $x$ will likely be referenced again in the near future.

• Spatial locality
  – If memory location $x$ is referenced, then locations near $x$ will likely be referenced in the near future.

• Idea
  – Buffer recently accessed data in cache close to CPU
Basic Idea - Caching

Buffer data \( d \) in the cache

Send data \( d \) to CPU

Access data \( d \)

smaller faster closer to CPU

CPU

rax: \( d \)

rbx:

Cache

Memory
Basic Idea - Caching

- **CPU**
  - `rax: d`
  - `rbx:`

- **Cache**
  - Access data `d` smaller, faster, closer to CPU
  - Send cached `d` to CPU ~4 cycles

- **Memory**
  - 100-200 cycles
Intuitive implementation

• Caching at byte granularity:
  – Search the cache for each byte accessed
    • `movq (%rax), %rbx` → checking 8 times

• High bookkeeping overhead
  – each cache entry has 8 bytes of address and 1 byte of data
Caching at Block Granularity

Solution:
• Cache one block (cache line) at a time.
• A typical cache line size is 64 bytes

Advantage:
• Lower bookkeeping overhead
  – A cache line has 8 byte of address and 64 byte of data
• Exploits spatial locality
  – Accessing location x causes 64 bytes around x to be cached
Direct-mapped cache

Caching at block granularity
- Each cache line has 64 bytes

1. In cache? No.
2. Fetch
3. Return

CPU Cache

<table>
<thead>
<tr>
<th>PA</th>
<th>64 bytes</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>
Direct-mapped cache

Caching at block granularity
- Each cache line has 64 bytes

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

Caching at block granularity
• Each cache line has 64 bytes

1. In cache? Yes.

2. Send 0x2 to the CPU

<table>
<thead>
<tr>
<th>PA</th>
<th>64 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>...</td>
</tr>
<tr>
<td>0x40</td>
<td>...</td>
</tr>
<tr>
<td>0x102</td>
<td>...</td>
</tr>
<tr>
<td>0x103</td>
<td>...</td>
</tr>
</tbody>
</table>
Multi-way set associative cache

2-way set associative cache
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

These policies require additional time and hardware.
Further Optimizing Memory Access

Current design: address translation → cache access

• The two steps are performed sequentially

![Diagram showing the interaction between CPU, MMU, and cache lines with memory addresses and cache data.]
Further Optimizing Memory Access

Virtual Index and Physical Tag
• Use VA to index set, calculate the tag from PA
• Cache set lookup is done in parallel with address translation
Virtual Index and Physical Tag

- **Set 0**
  - Tag
  - Cache line

- **Set 1**
  - Tag
  - Cache line

- **Set 15**
  - Tag
  - Cache line

**CPU Cache**

2-way set associative cache
Memory Hierarchy

CPU

- rax:
- rbx:
- ...

L1 cache: ~4 cycles

L2 cache: ~12 cycles

L3 cache: ~35 cycles

Memory: ~150 cycles
Cache summary

• Caching can speed up memory access
  – L1/L2/L3 cache data
  – TLB caches address translation
• Cache design:
  – Direct mapping
  – Multi-way set associative
  – Virtual index + physical tag parallelize cache access and address translation
Cache-friendly Code

```c
int64 sumarrayrows(int64** a, int r, int c) {
    int i, j = 0;
    int64 sum = 0;

    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            sum += a[i][j];
    return sum;
}
```

Which implementation is more cache friendly?

(i,*)

(*,j)
int64 sumarrayrows(int64** a,
              int r, int c) {
    int i, j = 0;
    int64 sum = 0;

    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            sum += a[i][j];
    return sum;
}

int64 sumarrayrows(int64** a,
              int r, int c) {
    int i, j = 0;
    int64 sum = 0;

    for (int j = 0; j < c; j++)
        for (int i = 0; i < r; i++)
            sum += a[i][j];
    return sum;
}

How many cache misses?
Example:
• CPU Cache: 2-way associative, 2 sets, 64 bytes cache line
• Array: int64 a[6][8]; address of a[0][0] is 64-byte-aligned
• local variables: i, j, sum are stored in the registers
Simple Example

for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
**Simple Example**

```c
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
```

### Diagram

- **CPU Cache**
  - `a[5][7]`
  - `a[5][1]`
  - `a[5][0]`
  - `a[4][7]`
  - `a[4][1]`
  - `a[4][0]`
  - `a[3][7]`
  - `a[3][1]`
  - `a[3][0]`
  - `a[2][7]`
  - `a[2][1]`
  - `a[2][0]`
  - `a[1][7]`
  - `a[1][1]`
  - `a[1][0]`
  - `a[0][7]`
  - `a[0][1]`
  - `a[0][0]`

- **Memory**
  - `j:0, i:0`

- **Miss**
  - `sum += a[i][j];`
Simple Example

```cpp
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
```

CPU Cache

```
0  a[0][0], a[0][1], ..., a[0][7]
1  a[1][0], a[1][1], ..., a[1][7]
```

memory

miss
Simple Example

```java
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
```

j:0, i:2

CPU Cache

```java
[0]
[0][0], [0][1], ..., [0][7]
[0][0], [0][1], ..., [0][7]
[2][0], [2][1], ..., [2][7]
[1][0], [1][1], ..., [1][7]
[2][0], [2][1], ..., [2][7]
```

miss
Simple Example

for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        j:0, i:3
        sum += a[i][j];
Simple Example

for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
Simple Example

```c
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
```
Simple Example

for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
```
Simple Example

for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
Simple Example

```c
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
```
Simple Example

```java
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
        sum += a[i][j];
```

```
0
a[2][0],a[2][1]...a[2][7]  a[4][0],a[4][1]...a[4][7]

1
a[3][0],a[3][1]...a[3][7]  a[5][0],a[5][1]...a[5][7]
```
Simple Example

```c
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        sum += a[i][j];
```
Simple Example

```c
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        sum += a[i][j];
```

CPU Cache

```
0: a[0][0], a[0][1]...a[0][7]
1: 
```
Simple Example

for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        sum += a[i][j];
Simple Example

```c
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        sum += a[i][j];
```

CPU Cache:

```
a[0][0]
a[0][1]
a[0][2]
a[0][3]
a[0][4]
a[0][5]
a[0][6]
```

```
a[1][0]
a[1][1]
a[1][2]
a[1][3]
a[1][4]
a[1][5]
a[1][6]
```

```
a[2][0]
a[2][1]
a[2][2]
a[2][3]
a[2][4]
a[2][5]
a[2][6]
```

```
a[3][0]
a[3][1]
a[3][2]
a[3][3]
a[3][4]
a[3][5]
a[3][6]
```

```
a[4][0]
a[4][1]
a[4][2]
a[4][3]
a[4][4]
a[4][5]
a[4][6]
```

```
a[5][0]
a[5][1]
a[5][2]
a[5][3]
a[5][4]
a[5][5]
a[5][6]
```

```
...  
...  
...  
...  
...  
...  
```

```
memory
```

```
i:0, j:7
hit
```
Simple Example

for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        sum += a[i][j];
Simple Example

```c
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        sum += a[i][j];
```

CPU Cache

0: `a[0][0], a[0][1], ..., a[0][7]`
1: `a[1][0], a[1][1], ..., a[1][7]`

hit
Simple Example

```c
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        sum += a[i][j];
```

CPU Cache

memory
Matrix Multiplication

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned

```
for (int i=0; i < N; i++) {
    for (int k=0; k < N; k++) {
        for (int j=0; j < N; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
for (int i=0; i < N; i++) {
    for (int k=0; k < N; k++) {
        for (int j=0; j < N; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
for (int i=0; i < N; i++) {
    for (int j=0; j < N; j++) {
        for (int k=0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
for (int i=0; i < N; i++) {
    for (int k=0; k < N; k++) {
        for (int j=0; j < N; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Which one is cache friendly, which one is worst?
Matrix Multiplication (ijk)

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned

```java
for (int i=0; i < N; i++) {
    for (int j=0; j < N; j++) {
        for (int k=0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

```
1 miss
8 misses
```
Matrix Multiplication (ikj)

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned

```java
for (int i=0; i < N; i++) {
    for (int k=0; k < N; k++) {
        for (int j=0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```
for (int k=0; k < N; k++) {
    for (int j=0; j < N; j++) {
        for (int i=0; i < N; i++)
            C[i][j] += A[i][k] * B[k][j];
    }
}

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned