CENG311 Computer Architecture

Cache Performance

IZTECH, Fall 2023

04 January 2024

Cache Miss Rate

Cache size

Block size

Associativity

Replacement policy

Cache Size

Cache size: total data (not including tag) capacity

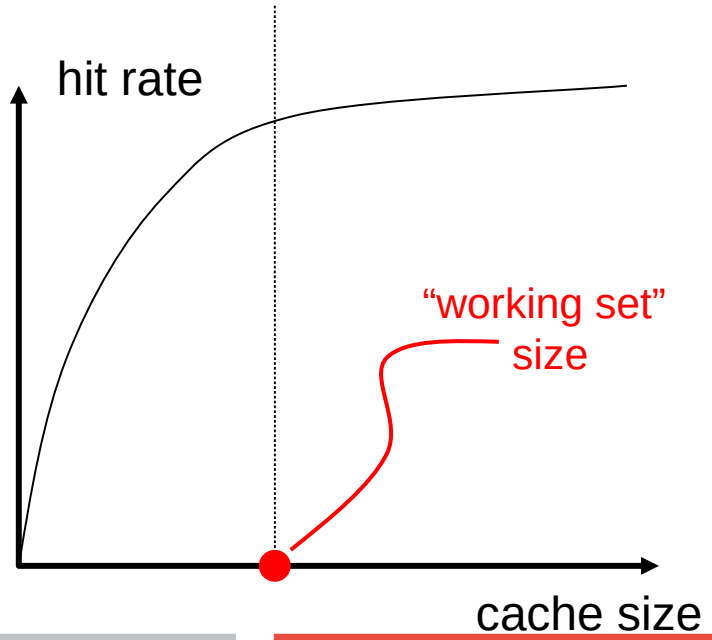
- Bigger can exploit temporal locality better
- Not ALWAYS better

Too large a cache adversely affects hit and miss latency

- Smaller is faster => bigger is slower
- Access time may degrade critical path

Too small a cache

- Doesn't exploit temporal locality well
- Useful data replaced often



Block Size

Larger blocks should reduce miss rate

- Due to spatial locality

But in a fixed-sized cache

- Larger blocks ⇒ fewer of them
  - More competition ⇒ increased miss rate
- Larger blocks ⇒ pollution

Larger miss penalty

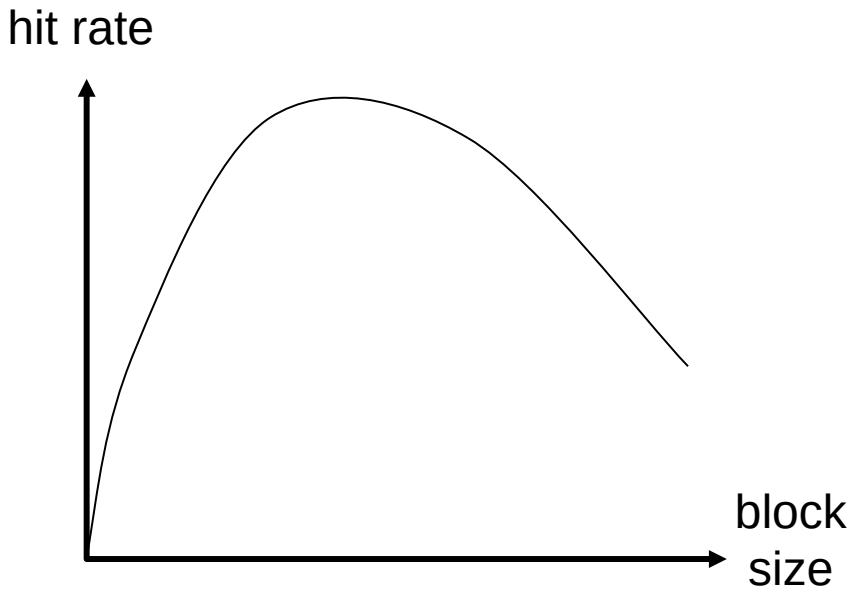
- Can override benefit of reduced miss rate

Block Size

Block size is the data that is associated with an address tag

Too small blocks

- Don't exploit spatial locality well
- Have larger tag overhead



Too large blocks

- Too few total # of blocks => less temporal locality exploitation
- Waste of cache space and bandwidth/energy if spatial locality is not high

Associativity

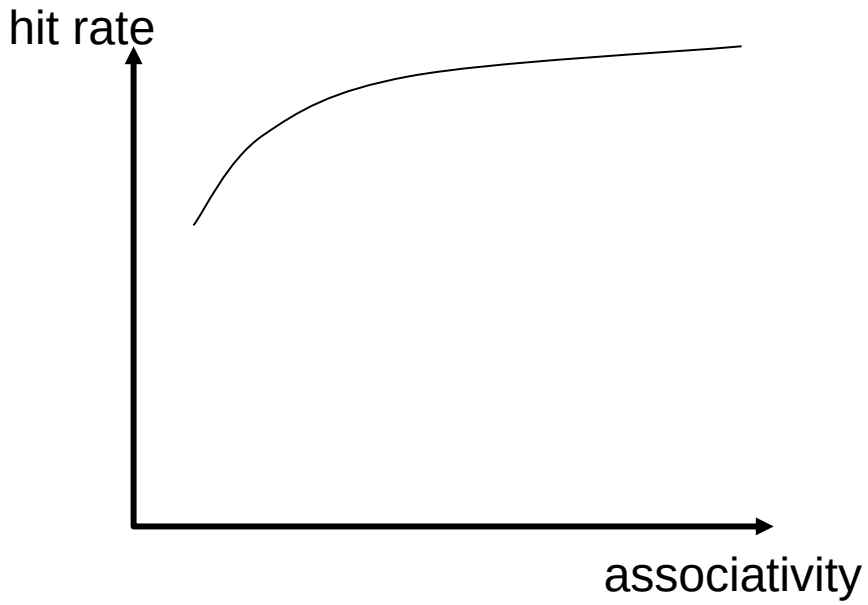
How many blocks can be present in the same index (i.e., set)?

Larger associativity

- Lower miss rate (reduced conflicts)
- Higher hit latency and area cost

Smaller associativity

- Lower cost
- Lower hit latency
  - Especially important for L1 caches



Eviction/Replacement Policy

Which block in the set to replace on a cache miss?

- Any invalid block first
- If all are valid: Random, FIFO, Least recently used, Not most recently used, Least frequently used, Least costly to re-fetch, Hybrid replacement policies
- Optimal replacement policy?

LRU vs Random

- Average hit rate of LRU and Random is similar

Cache Performance

Cache miss rate =

(#misses) / (#hits + #misses) =

(#misses) / (#accesses)

Average memory access time (AMAT)

= Time for hit + Miss rate * Miss penalty

Latency Analysis

For a given memory hierarchy level i it has an access time of t_i, the perceived access time T_i is longer than t_i

Except for the outer-most hierarchy, when looking for a given address there is

- a chance (hit-rate h_i) you "hit" and access time is t_i
- a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$

$$h_i + m_i = 1$$

Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + m_i \cdot T_{i+1}$$

h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Intel Pentium 4 Example

90nm P4, 3.6 GHz

L1 D-cache

C_1 = 16K

t_1 = **4 cyc**

L2 D-cache

C_2 =1024 KB

t_2 = **18 cyc**

Main memory

t_3 = **180 cyc**

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

How to Improve Cache Performance

Reducing miss rate

- More associativity
- Better replacement/insertion policies

Reducing miss latency or miss cost

- Multi-level caches
- Better replacement/insertion policies
- Multiple accesses per cycle

Reducing hit latency or hit cost

Software Approaches for Higher Hit Rate

Restructuring data access patterns

Restructuring data layout

Loop interchange

Data structure separation/merging

Blocking

Data Access Pattern

Example: If row-major

x[i,j+1] follows x[i,j] in memory

x[i+1,j]] is far away from x[i,j]

Poor code

```
for j = 1, rows
      for i = 1, columns
            sum = sum + x[i,j]
```

Better code

```
for i = 1, columns
      for j = 1, rows
            sum = sum + x[i,j]
```

Loop interchange

Blocking

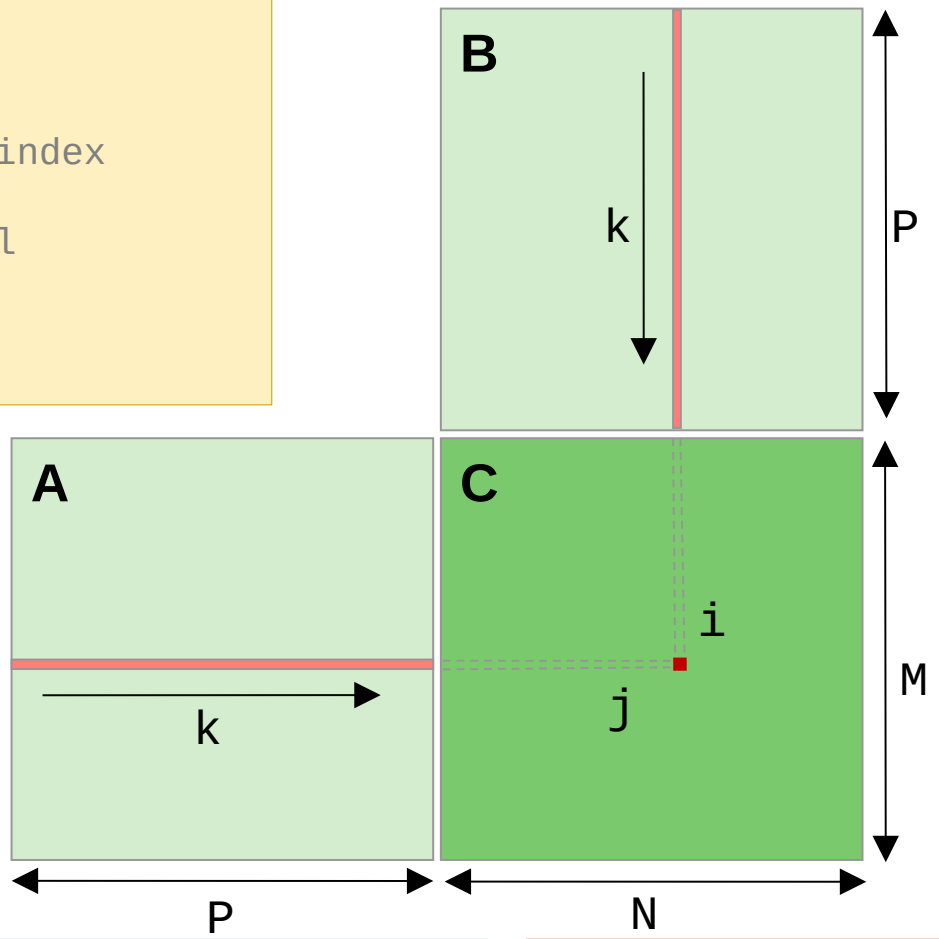
Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache

Avoids cache conflicts between different chunks of computation

Essentially: Divide the working set so that each piece fits in the cache

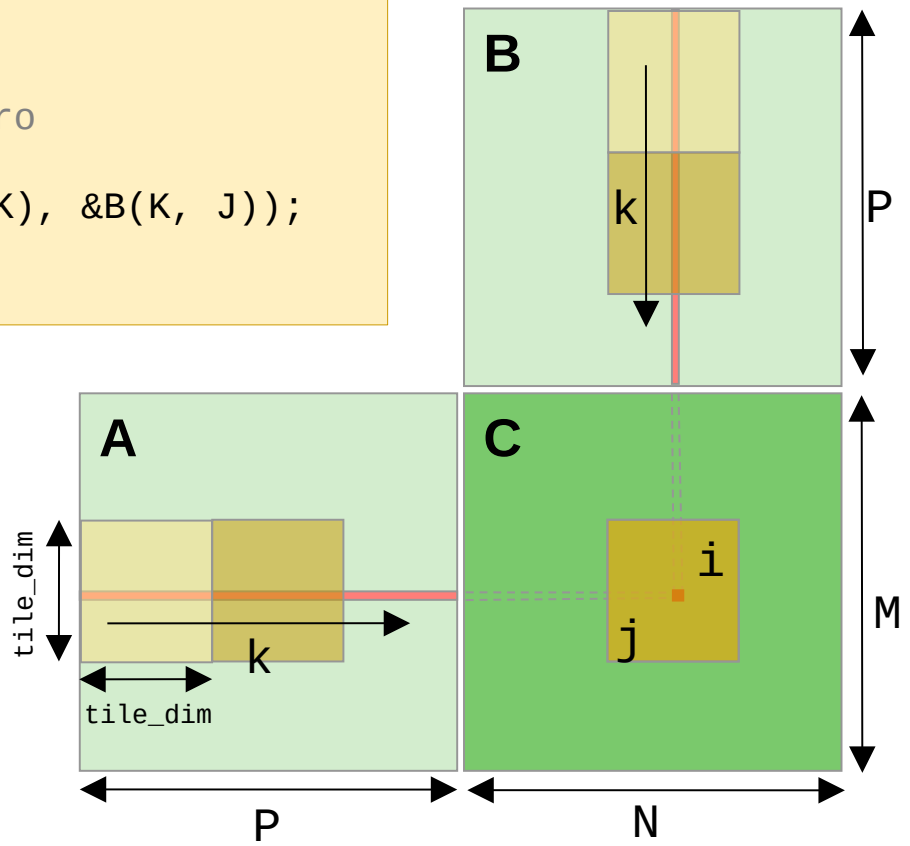
Naive Matrix Multiplication

```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (i = 0; i < M; i++){ // i = row index
    for (j = 0; j < N; j++){ // j = column index
        C(i, j) = 0; // Set to zero
        for (k = 0; k < P; k++) // Row x Col
            C(i, j) += A(i, k) * B(k, j);
    }
}
```



Tiled Matrix Multiplication

```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (I = 0; I < M; I += tile_dim){
    for (J = 0; J < N; J += tile_dim){
        Set_to_zero(&C(I, J)); // Set to zero
        for (K = 0; K < P; K += tile_dim)
            Multiply_tiles(&C(I, J), &A(I, K), &B(K, J));
    }
}
```



Data Layout

```
struct Node {
    struct Node* next;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
     if (node->key == input-key) {
     // access other fields of node
     }
     node = node->next;
}
```

```
struct Node {
    struct Node* next;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
     if (node->key == input-key) {
     // access node->node-data
     }
     node = node->next;
}
```

References

Related videos from Digital Design and Computer Architecture course (Spring 2022) in ETH Zurich, by Onur Mutlu

https://www.youtube.com/watch?v=jys92_j627A

https://www.youtube.com/watch?v=VxEGcEXiFj4