Lecture 28

Cache Performance

CPSC 275
Introduction to Computer Systems

Matrix Multiplication

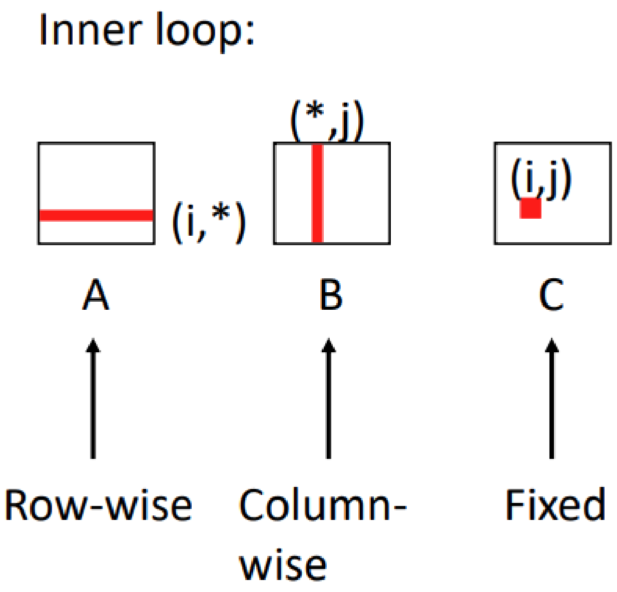
/* *ijk* version */

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

Complexity: $O(N^3)$

Goal: Improve temporal and spatial locality to reduce cache misses.

Access pattern:



B[k][j] fetched multiple times from main memory.

Block Matrix Multiplication

A and B can be written as:

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mm} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1m} \\ B_{21} & B_{22} & \cdots & B_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mm} \end{pmatrix}$$

where A_{ij} and B_{ij} are b-by-b submatrices of A and B, respectively. Then, $C = AB$ can be defined as:

$$C = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{m1} & C_{m2} & \cdots & C_{mm} \end{pmatrix}$$

where

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}, \quad 1 \leq i, j \leq m$$

Here, b is the block size and $n = bm$.

Block Matrix Multiplication

```
#define bsize 32   // block size

for (ii = 0; ii < N; ii += bsize)
    for (jj = 0; jj < N; jj += bsize)
        for (kk = 0; kk < N; kk += bsize) {
            // multiply submatrices
            sum = 0.;
            for (i = ii; i < ii + bsize; i++)
                for (j = jj; j < jj + bsize; j++)
                    for (k = kk; k < kk + bsize; k++)
                        sum += A[i][k] * B[k][j];
            C[i][j] = sum;
        }
```

How Blocking Improves Locality

- Spatial Locality
  - Each block of A and B is contiguous in memory.
  - Once a cache line is fetched, many nearby values are used before eviction.
- Temporal Locality
  - Each block of A and B is reused for multiple operations before being evicted.
  - C remains in cache until the block is completed.
- Instead of fetching entire rows/columns multiple times, each block is fetched once per block multiplication.

Choosing Block Size

- Each block small enough to fit in cache.

$$3 \times \text{bsize}^2 \times \text{sizeof}(\texttt{float}) < \text{cache size}$$

- Example: for a 640 KB L1 data cache (on our lab computers),

$$\text{bsize} \approx \sqrt{640\text{KB}/(3*4)} \approx 234 \text{ bytes}$$

What about writes?

- Multiple copies of data exist:
  - L1, L2, main memory, disk, etc.
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
  - No-write-allocate (writes immediately to memory)
- In practice,
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

Cache performance metrics

- *Miss rate*
  - fraction of memory references not found in cache (misses / accesses)
  - equivalent to 1 – hit rate
  - typical numbers (in percentage):
    - 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:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2
- *Miss penalty* - additional time required because of a miss
    - typically 50-200 cycles for main memory

Let’s think about those numbers

- Huge difference between a hit and a miss
- Would you believe 99% hits is twice as good as 97%?
  - e.g. consider:

    cache hit time of 1 cycle

    miss penalty of 100 cycles
  - average memory access time:

    97% hits: .97 * 1 cycle + 0.03 * 100 cycles = 3.97 cycles

    99% hits: .99 * 1 cycle + 0.01 * 100 cycles = 1.99 cycles

Cache performance

- Cache size
  - In theory, a larger cache will tend to increase the hit rate.
  - But, hard to make larger memories faster.
  - Larger caches tend to increase the hit time.
- Block size
  - In theory, larger blocks can help increase the hit rate.
  - But, for a given cache size, larger blocks imply a smaller number of cache lines. So what?
  - Larger blocks mean a higher miss penalty.
  - Compromise: 32-64 bytes

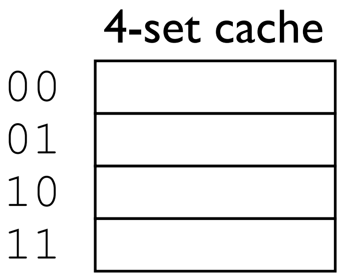
Cache performance, cont'd

- Level of associativity
  - Higher associativity means a lower risk of *thrashing*.
  - But, it's expensive to build, requiring:
    - more tag bits
    - additional logic
  - Also it's hard to make it fast because of:
    - Increased hardware complexity (hit time)
    - Time to determine a victim line (miss penalty)
  - In practice,
    - Lower associativity for L1 caches
    - Higher associativity for the lower caches
    - Example: Intel Core i7 (8-way for L1 and L2 and 16-way for L3)

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

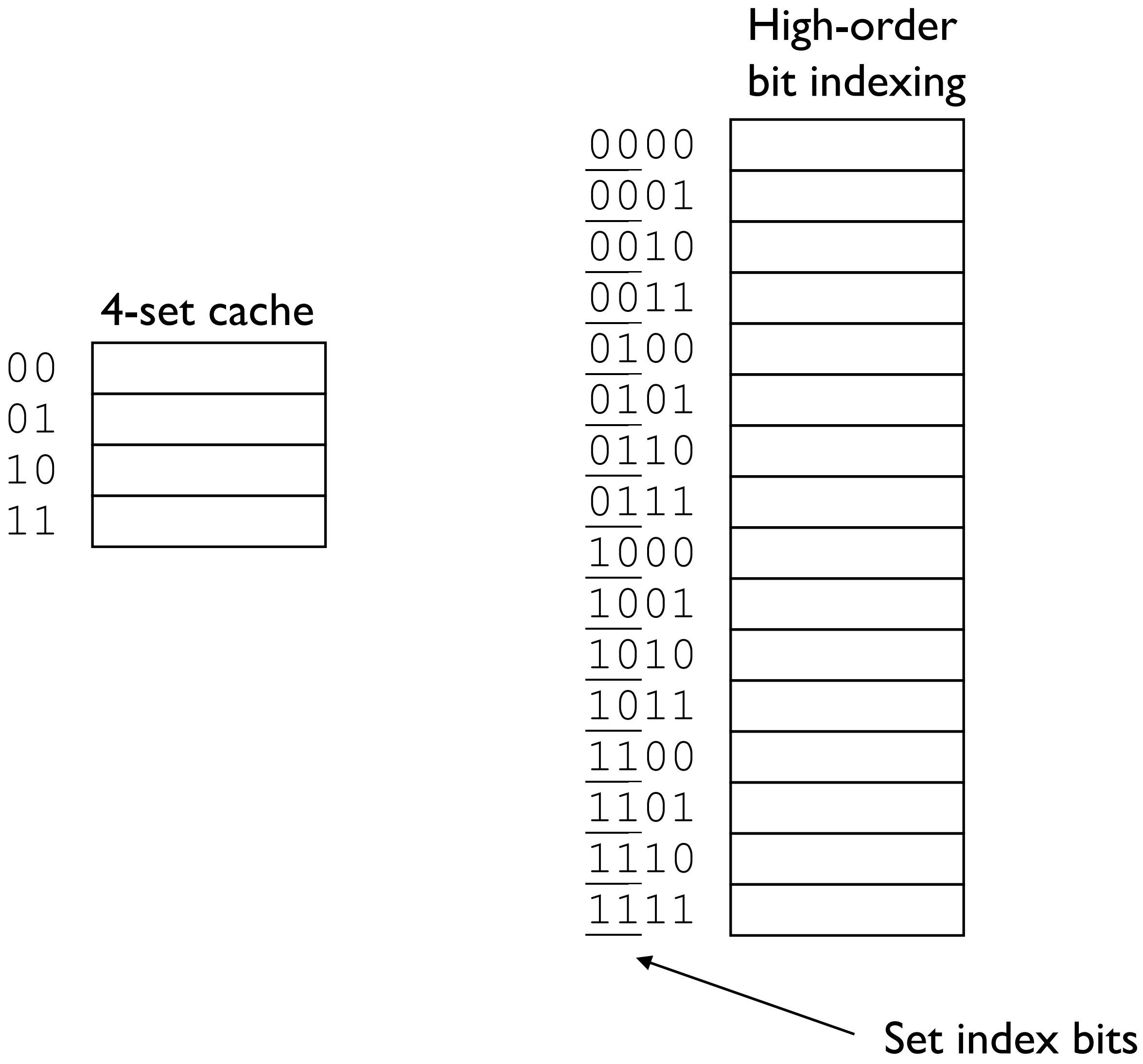
Why index with the middle bits?



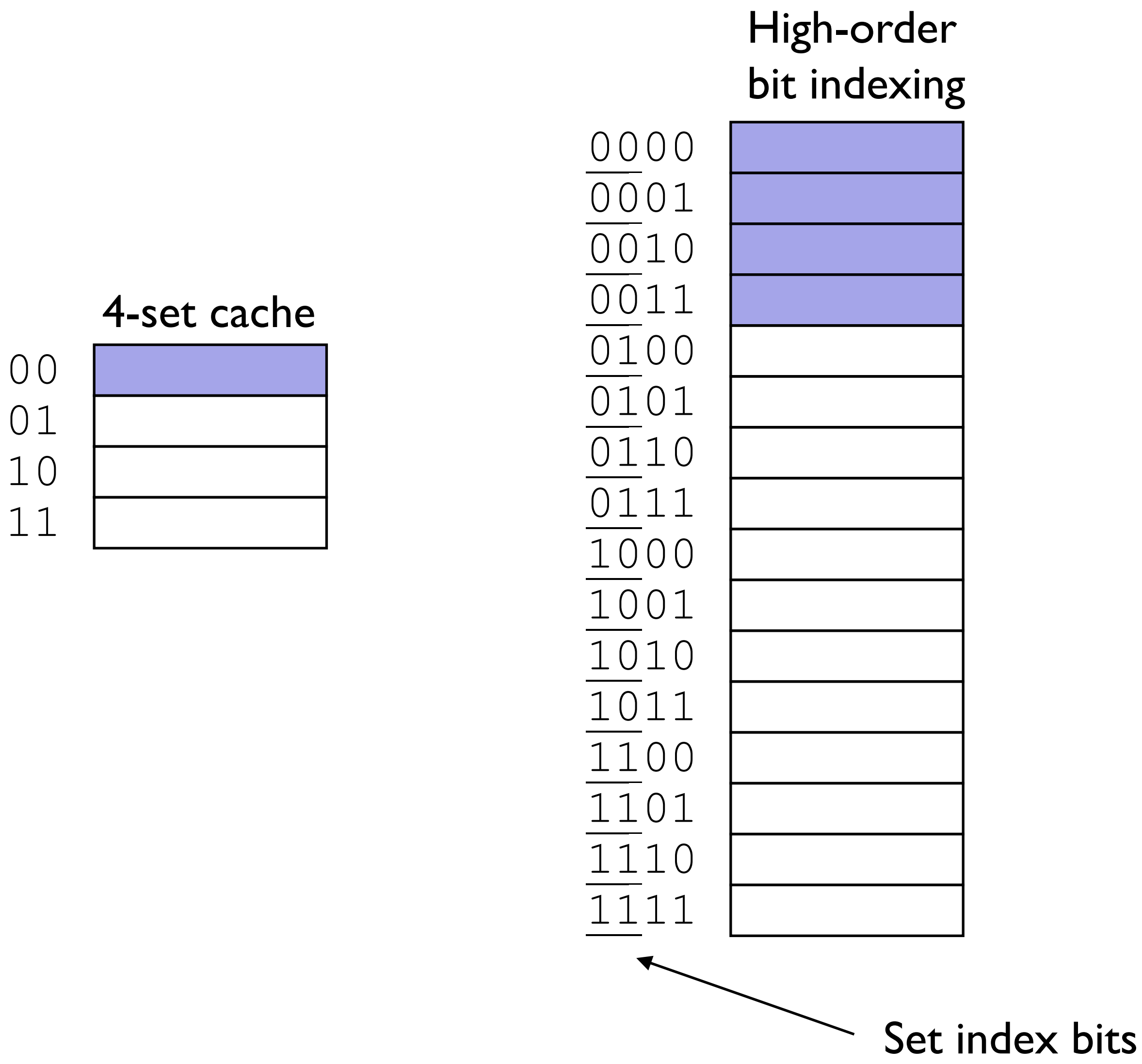
4-set cache

00
01
10
11

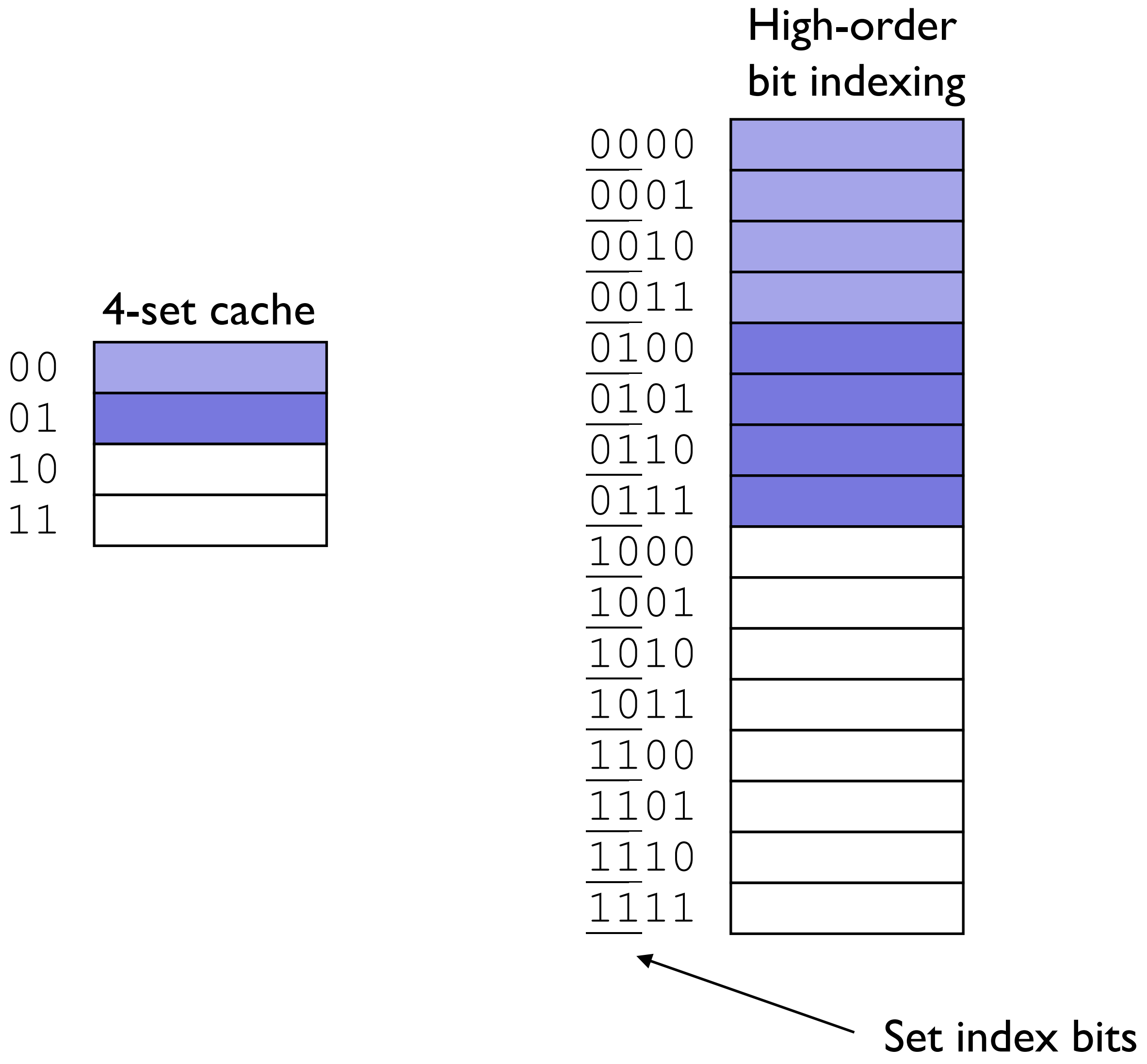
Why index with the middle bits?



Why index with the middle bits?



Why index with the middle bits?



4-set cache

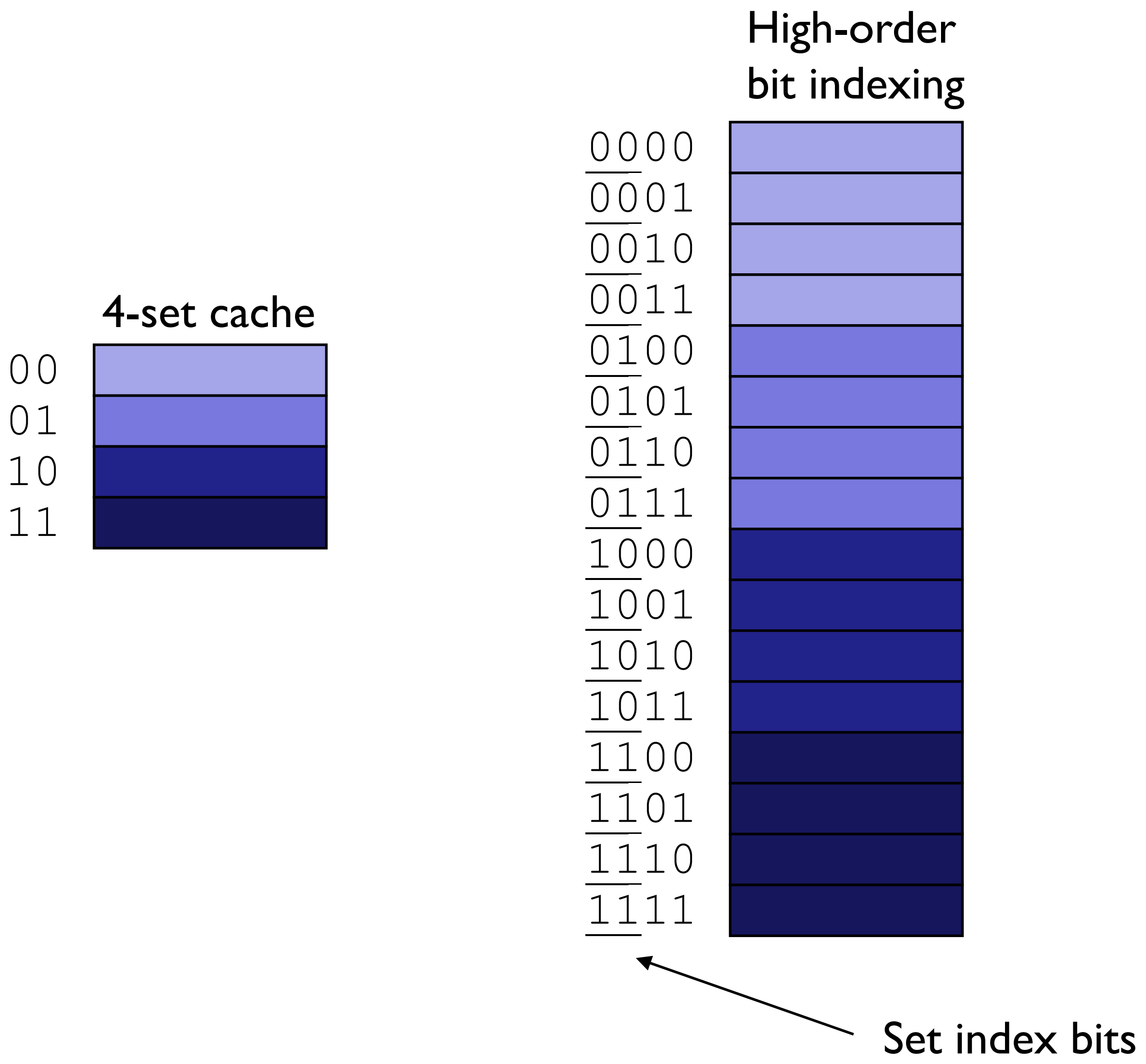
00
01
10
11

High-order bit indexing

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Set index bits

Why index with the middle bits?



4-set cache

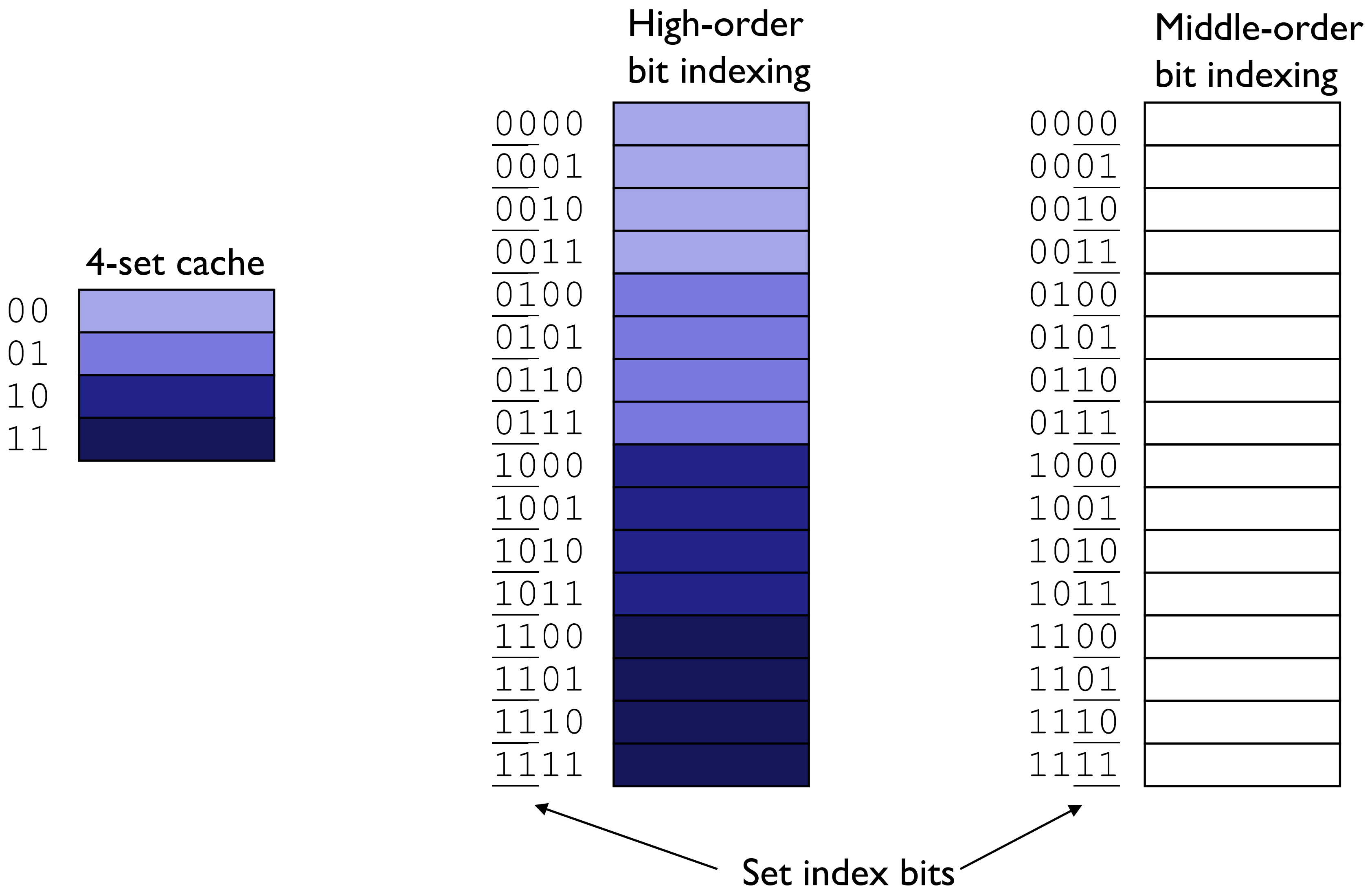
00
01
10
11

High-order bit indexing

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Set index bits

Why index with the middle bits?



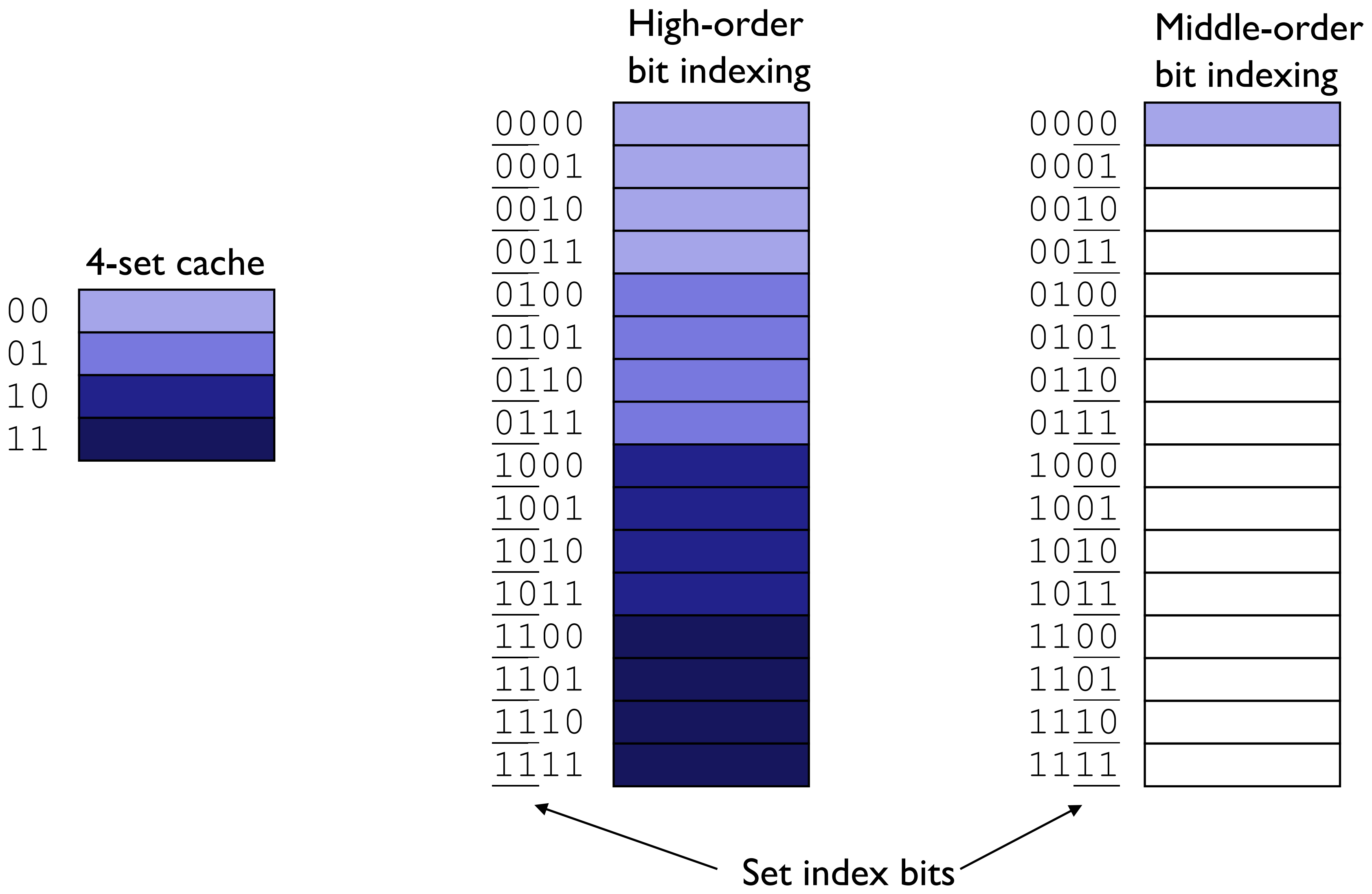
4-set cache

High-order bit indexing

Middle-order bit indexing

Set index bits

Why index with the middle bits?



4-set cache

00
01
10
11

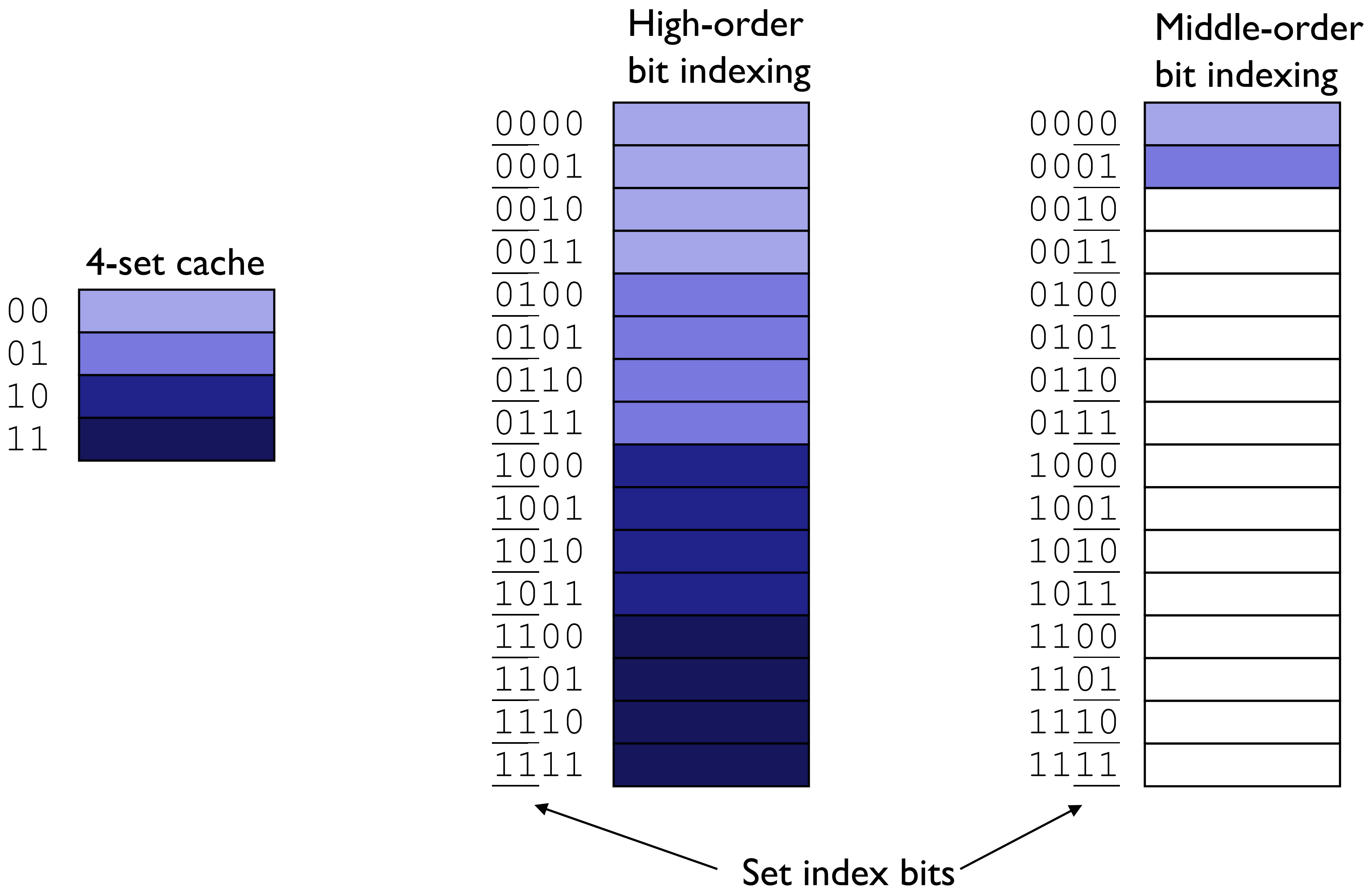
High-order bit indexing

Middle-order bit indexing

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Set index bits

Why index with the middle bits?



4-set cache

00
01
10
11

High-order bit indexing

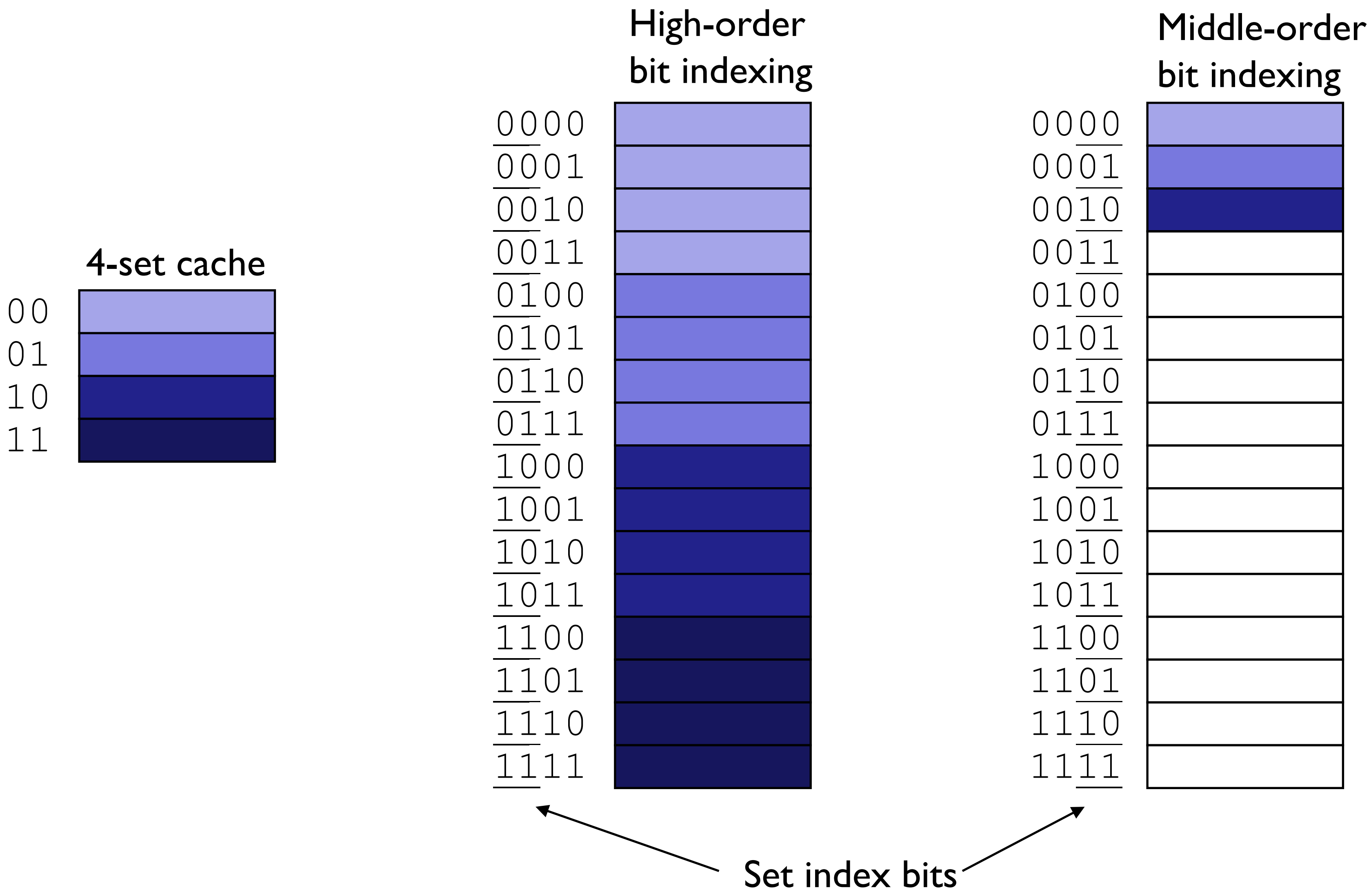
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Middle-order bit indexing

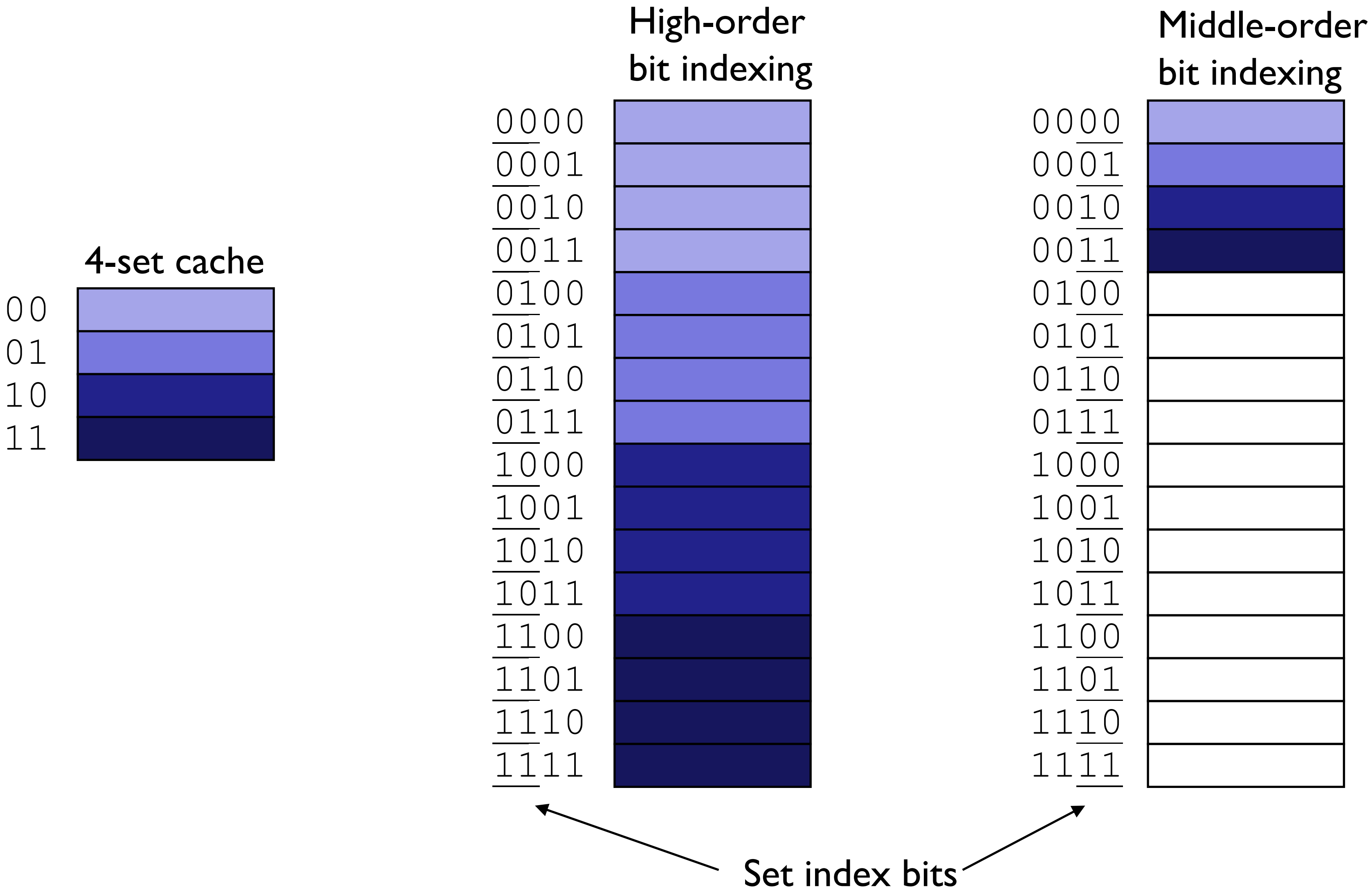
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Set index bits

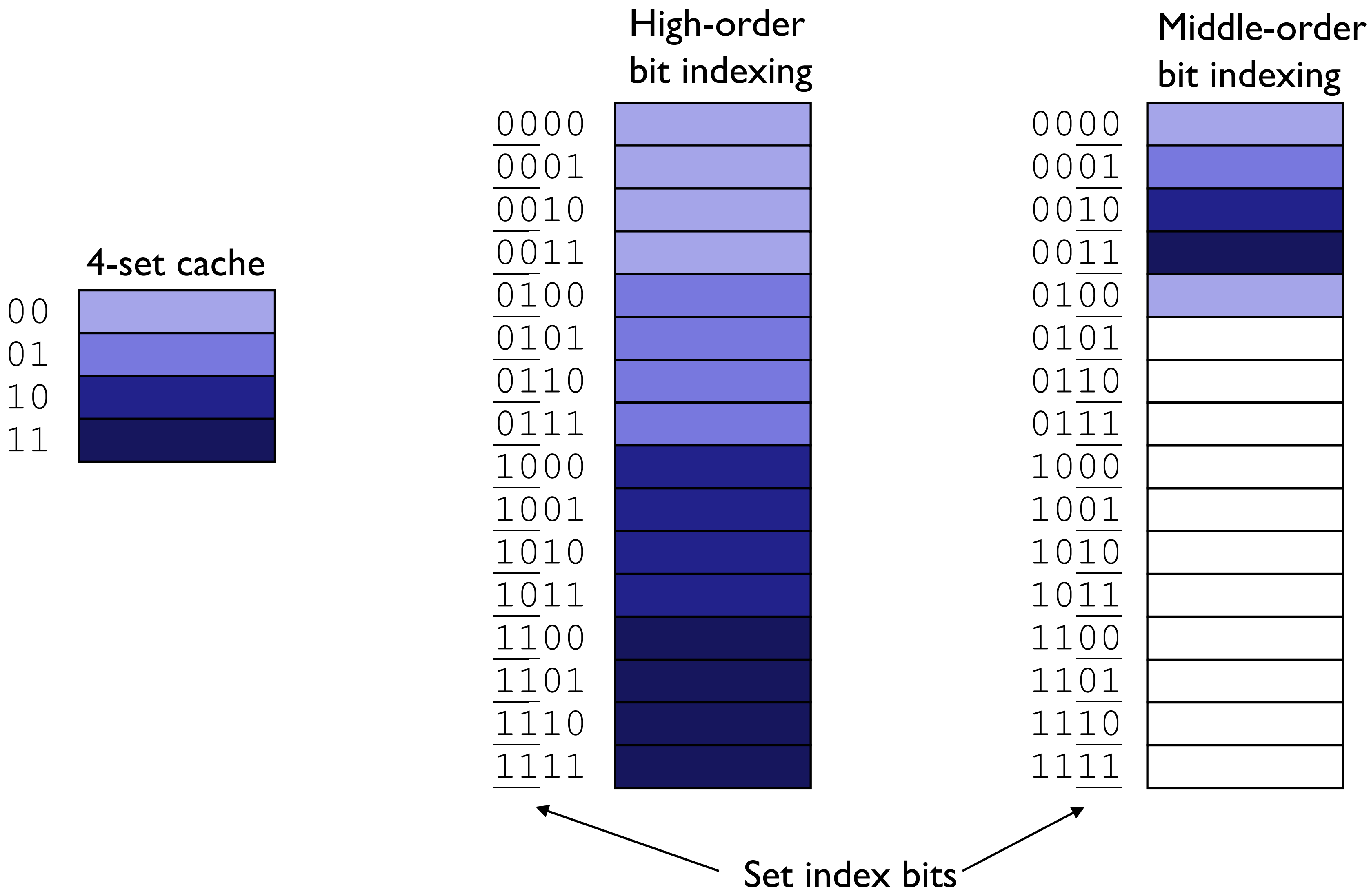
Why index with the middle bits?



Why index with the middle bits?



Why index with the middle bits?



4-set cache

00
01
10
11

High-order bit indexing

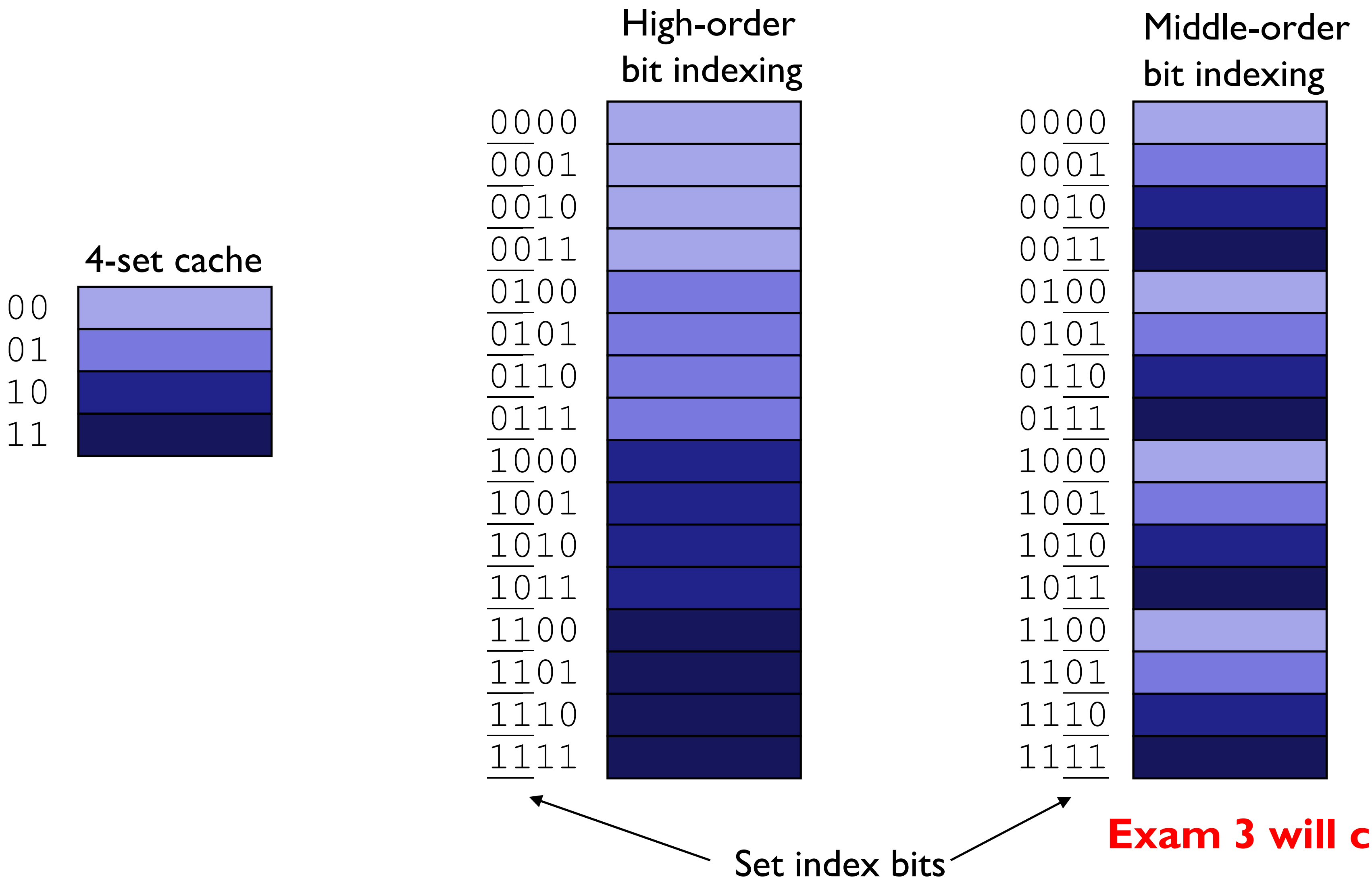
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Middle-order bit indexing

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Set index bits

Why index with the middle bits?



4-set cache

00
01
10
11

High-order bit indexing

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Middle-order bit indexing

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Set index bits

Exam 3 will cover up to here.