Types of Cache Misses: *The Three C's*

1 **Compulsory:** **On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.**

2 **Capacity:** **Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).**

3 **Conflict:** **In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.**

Miss Rate Reduction Techniques:

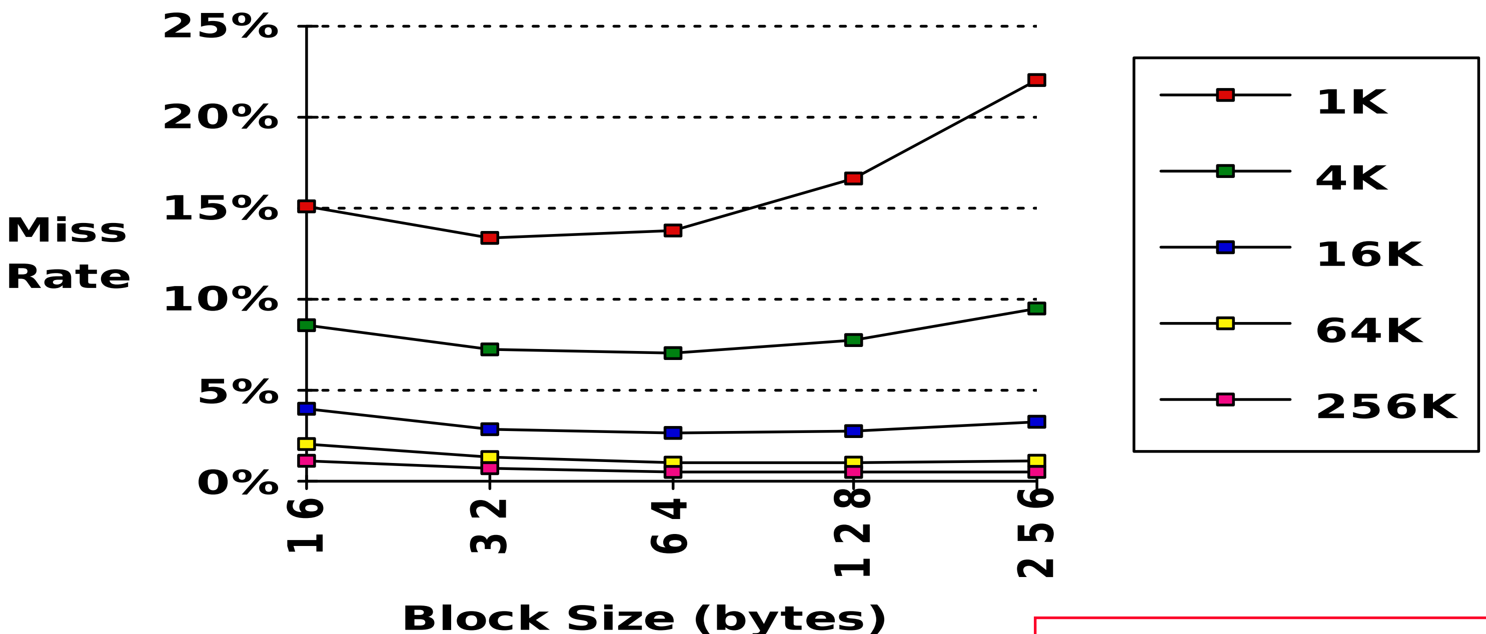
* Increased cache capacity
* Larger block size
* Higher associatively

* Compiler optimizations

Reduce Misses by Larger Block Size

- A larger block size improves cache performance by taking advantage of spatial locality
- For a fixed cache size, larger block sizes mean fewer cache block frames

Performance keeps improving to a limit when the fewer number of cache block frames increases conflict misses and thus overall cache miss rate



2. Reduce Misses by Increasing Cache Size

- **Increasing cache size reduces cache misses**
  - both capacity misses and conflict misses reduced

Miss Rate Reduction Techniques:

Higher Cache Associativity

- Reduces conflict misses

Cache Size (KB)	Associativity 1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

(Red means A.M.A.T. not improved by more associativity)

Compiler Optimizations

Compiler cache optimizations improve access locality characteristics of the generated code and include:

- ***Merging Arrays*: Improve spatial locality by single array of compound elements vs. 2 arrays.**
- ***Loop Interchange*: Change nesting of loops to access data in the order stored in memory.**
- ***Loop Fusion*: Combine 2 or more independent loops that have the same looping and some variables overlap.**
- ***Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows.**

Merging Arrays

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
   int val;
   int key;
};
struct merge merged_array[SIZE];
```

- combines two separate arrays (that might conflict for a single block in the cache) into a single interleaved array.
- This brings together corresponding elements in both arrays, which are likely to be referenced together.
- reduces misses by improving spatial locality.

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
    for (j = 0; j < 100; j = j+1)
         for (i = 0; i < 5000; i = i+1)
              x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
    for (i = 0; i < 5000; i = i+1)
         for (j = 0; j < 100; j = j+1)
              x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words in this case improves spatial locality.

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
   {    a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];}
```

- *Many programs have separate loops that operate on the same data.*
- *Combining these loops by grouping operations on the same (cached) data together.*