Improving Cache Performance

For improving the cache performance we need to consider the followings:

- Capacity misses can be damaging to the performance due to excessive main memory access.
- Increasing associativity, cache size and block width can reduces misses.
- Changing cache size affects both capacity and conflict misses since it spreads out references to more blocks.
- Some optimization techniques that reduces miss rate also increases hit access time.

For improving Cache performance we need to decrease average memory access time which is indirectly proportional to processor performance and directly proportional to CPU time.

Average access time =Hit time + Miss rate x Miss penalty.

So to decrease average memory access time we need to consider the followings

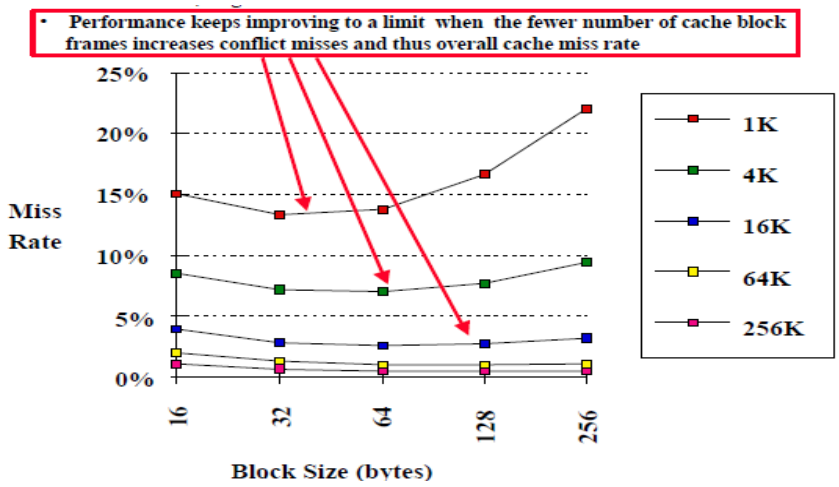
- **Reduce Miss Rate**
- **Reduce Cache Miss Penalty**
- **Reduce Cache Hit Time**

Reduce Miss Rate

Techniques for reducing cache misses are as follows

- **Reducing Misses via Larger Block Size**

  The simplest way to reduce miss rate is to increase the block size. The following figure shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce also compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

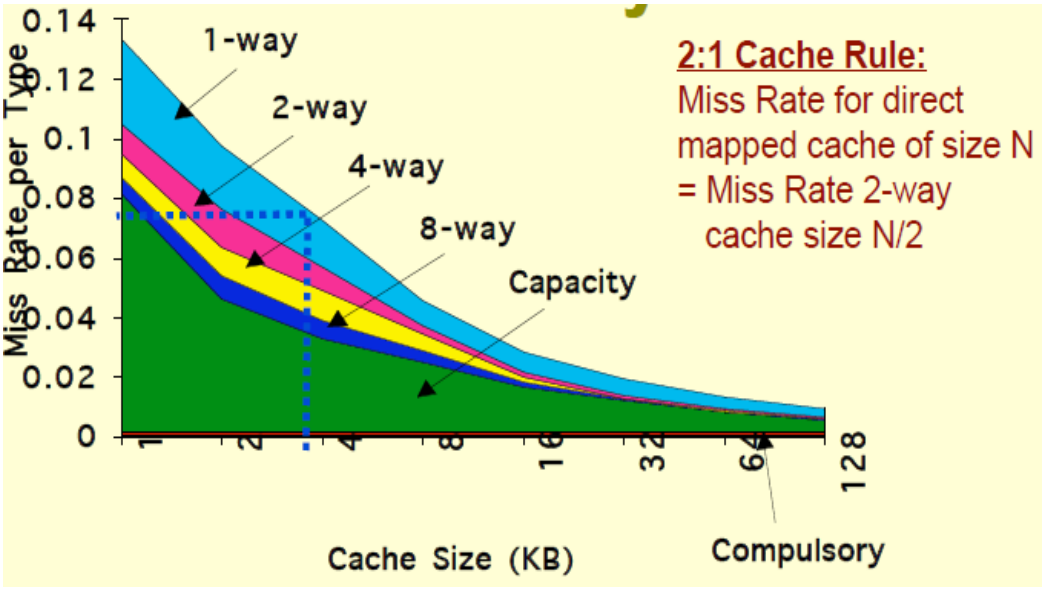


  Larger Block sizes reduces compulsory misses (principle of spatial locality)
  Conflict misses increases for target block sizes since cache has fewer blocks.

- **Reducing Misses via Larger Cache**

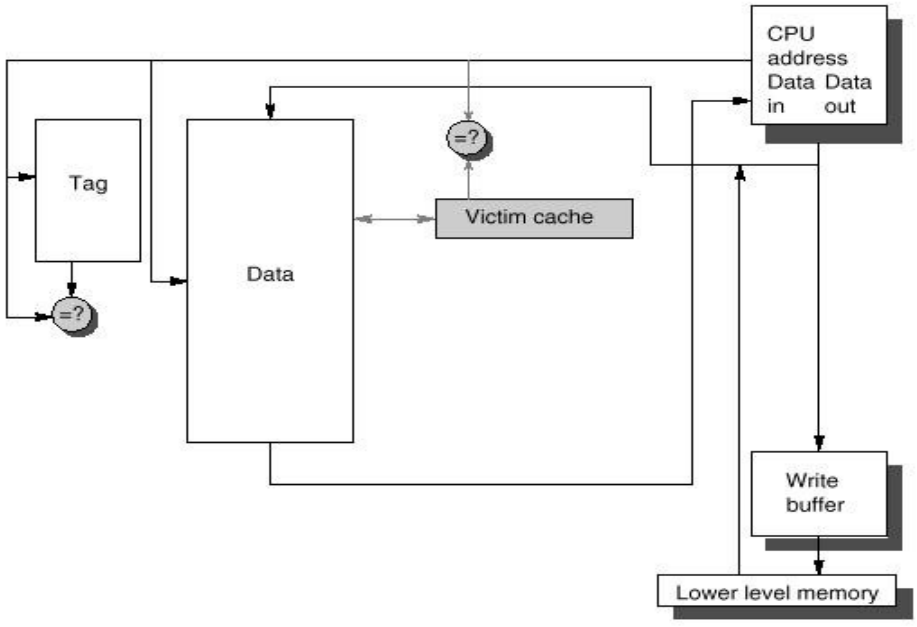
  The obvious way to reduce capacity misses is to increase capacity of the cache. The obvious drawback is potentially longer hit time and higher cost and power. This technique has been especially popular in off-chip caches.

- **Reducing Misses via Higher Associativity**
  Miss rates improvised with higher associativity. Let us consider the following figure:



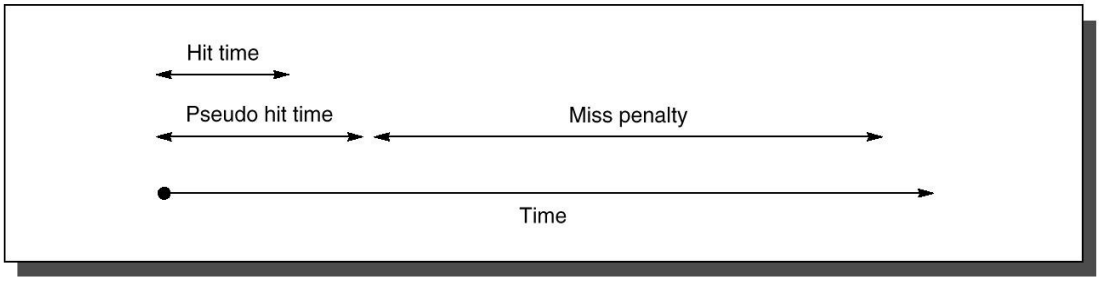
  Greater Associativity comes at the expense of target hit access time
  Hardware complexity grows for high associativity and clock cycle increases.

- **Reducing Misses via Victim Cache**
  A victim cache is a cache used to hold blocks evicted from a CPU cache upon replacement. The victim cache lies between the main cache and its refill path, and only holds blocks that were evicted from the main cache. The victim cache is usually fully associative, and is intended to reduce the number of conflict misses. On a cache miss victim cache is checked first for data before going to main memory. Many commonly used programs do not require an associative mapping for all the accesses. In fact, only a small fraction of the memory accesses of the program require high associativity. The victim cache exploits this property by providing high associativity to only these accesses.



- **Reducing Misses via Pseudo- Associativity**
  A pseudo-associative cache tests each possible way one at a time. A hash-rehash cache and a column-associative cache are examples of pseudo-associative cache. In the common case of finding a hit in the first way tested, a pseudo-associative cache is as fast as a direct-mapped cache. But it has a much lower conflict miss rate than a direct-mapped cache, closer to the miss rate of a fully associative cache.

  Two types of hits: regular (first access) and pseudo (second access) hits.



- **Reducing Misses via H/W Prefetching Instruction and Data**
  Another technique to reduce Cache misses using H/W prefetching instruction and data. The hardware pre-fetchs instructions and data while handling other cache misses, assuming that the pre-fetched items will be referenced shortly. Pre-fetching relies on having extra memory bandwidth that can be used without penalty.

$$\text{Average memory access time}_{\text{pre-fetch}} = \text{Hit time} + \text{Miss rate} \quad \text{Pre-fetch hit rate}$$
$$1 + \text{Miss rate} \quad (1 - \text{Pre-fetch hit rate}) \quad \text{Miss penalty}$$

- **Reducing Misses via S/W Prefetching Data**
  Another technique to reduce Cache misses using S/W prefetching data. It uses special instructions to pre-fetch data:
  – Load data into register
  – Cache Pre-fetch: load into cache

  Special pre-fetching instructions cannot cause faults since it is a form of speculative execution. This technique. Makes sense if the processor can proceeds without blocking for a cache access (lock-free cache). Loops are typical target for pre-fetching after unrolling (miss penalty is small) or after applying software pipelining (miss penalty is large).

- **Reducing Misses by Compiler Optimizations**
  Complier-based cache optimization reduces the miss rate without any hardware change or complexity.
  For Instructions
  – Reorder procedures in memory so as to reduce conflict misses
  – Profiling to determine likely conflicts among groups of instructions

  For Data
  – Merging Arrays: improve spatial locality by single array of compound elements vs. two arrays
  – Loop Interchange: change nesting of loops to access data in order stored in memory
  – Loop Fusion: Combine two independent loops that have 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 Example

```
/* 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];
```

Merging the two arrays:
– Reduces conflicts between val and key
– Improve 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];
```

Improvement
- 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];
                }
```

Improvement
• Two misses per access to a & c versus one miss per access
• Improves spatial locality

Data Access Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
for (j = 0; j < N; j = j+1)
{
        r = 0;
        for (k = 0; k < N; k = k+1)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;
}

 /* After */
for (jj = 0; jj < N; jj = jj+B)
        for (kk = 0; kk < N; kk = kk+B)
                for (i = 0; i < N; i = i+1)
                        for (j = jj; j < min(jj+B-1,N); j = j+1)
                        {
                                r = 0;
                                for (k = kk; k < min(kk+B-1,N); k = k+1)
                                {
                                        r = r + y[i][k] * z[k][j];};
                                        x[i][j] = x[i][j] + r;
                                }
                        }
```

==========