Reducing Miss Rate

- Miss rate reduction represents the classical approach to improving cache behavior, therefore has received a lot of (possibly misplaced) interest over last decade.
- Could you characterize the source of cache misses?
- Consider the "Tree C's" model,
    - Attempts to characterize misses into 3 classes,
        - Compulsory – cache at boot will always be empty.
            - First (cache) references are always misses (cold start).
        - Capacity – limited cache size
            - Entire program does not reside within cache.
            - Flushed blocks later require reloading.
        - Conflict – too many main memory (MM) blocks mapped to the same cache set (direct and set associative).
- Figure 1 qualifies miss rate on different cache sizes as a function of the three C's for 1- (i.e. direct), 2-, 4- and 8-way set associative caches.
    - Compulsory misses – effectively contribute no impact and are independent of cache size;
    - Conflict misses – decreases with increasing associativity;
    - Capacity misses – decreases with increasing cache size;
    - Miss rate – decreases with increasing associativity.

Limitations of the 3 C's model

- Only describes average cache miss behavior
    - Reason for individual misses is not available.
- No temporal information captured
    - Increasing associativity maximizes cache hits but may increases cache hit penalty!
- Following techniques for reducing miss rate need to be taken within the context of overall cache performance.

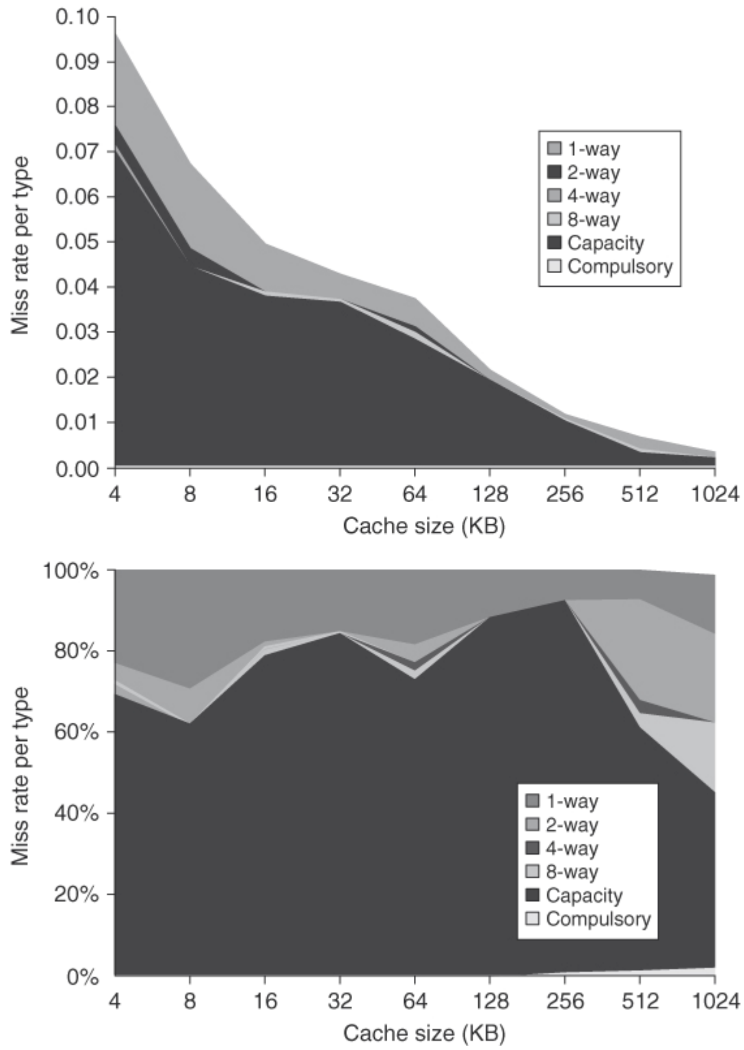

Figure 1: Total Miss Rate (top), Distribution of Miss Rates (bottom) as a function of the three C's

#1 – Larger Caches

- Motivation – let more data/ instructions reside in cache.
- Any disadvantage?
  - Longer hit time!!
- Table 1 lists average memory access time against block size for five cache sizes, with lowest time in bold.
- Pragmatics:
  - Use on off-chip caches, hit time already high and last defense against a call to disk

Table 1 – Average memory access time against block size for 5 cache sizes

Block Size	Miss Penalty	Cache Size			
		4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.558
64	88	7.160	**3.323**	**1.933**	**1.447**
128	96	8.469	3.659	1.979	1.470
256	112	11.961	4.685	2.288	1.549

#2 – Larger Block Sizes

- Motivation – maximize spatial locality.
- Figure 2 plots miss rate as a function of block size on SPEC 92 benchmark with DECstation 5000.
    - Optimal block size appears to be a ratio of cache size.
- Advantages
    - Emphasize spatial locality of data/ instructions;
    - Reduction in compulsory misses
        - (from Figure 1 this is actually of minimum significance);
- Disadvantages
    - Increasingly sensitive to miss penalty as block size increases;
    - Conflict misses increase,
        - Reduced block diversity, more MM blocks mapped to same cache set (set associative).

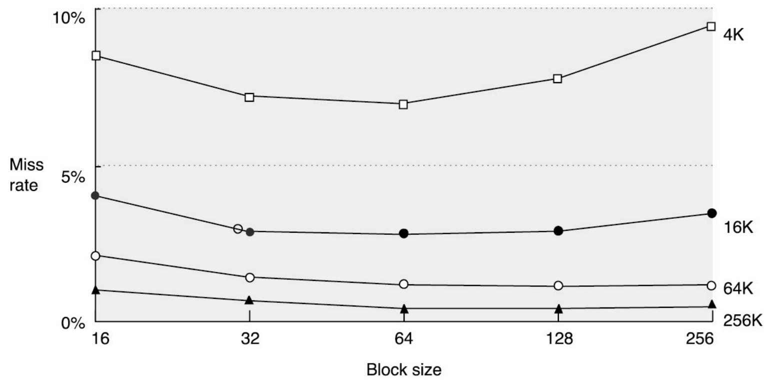

Figure 2: Miss rate against block size for 5 different cache sizes.

#3 – Higher Associativity

- With respect to Figure 1,
  - 8-way set associative approximates fully associative;
  - 2:1 cache rule of thumb (for cache size < 128 K bytes)
    - miss rate of directly mapped cache of size N = miss rate of 2-way set associative cache of size N / 2.
- Comment
  - Access time increases with associativity, Figure 3.
  - Current practice is to utilize 1- to 4-way set associative caches.
  - Short CPU clock cycle rewards simple (1-way) cache designs – e.g. L1 cache
  - High miss penalties rewards higher associativity – e.g. L2 cache

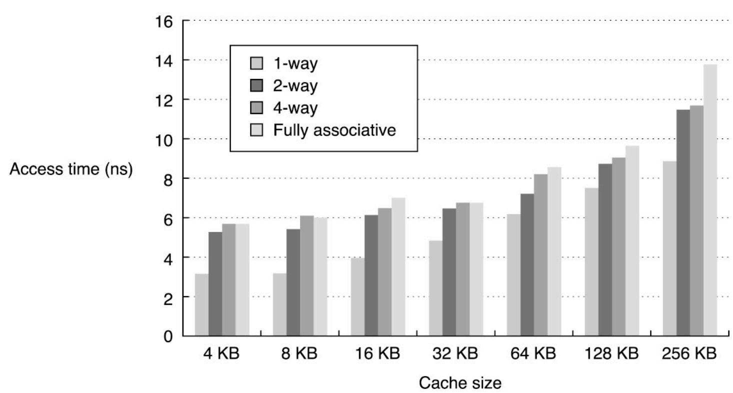

Figure 3: Assess time as a function of associativity and cache size.

#4 – Way Prediction

- As per dynamic branch prediction, each block in a set associative cache has a predictor.
    - Predictor selects which block of a set to try at the ***next*** cache access.
        - Tag comparison on predicted block conducted in parallel with the read.
    - If the prediction is incorrect,
        - CPU flushes the fetched instruction and the predictor is updated;
        - Continue to make tag comparisons for remaining blocks.

#5 – Compiler Optimizations

- Independent of specific hardware units.
- Concentrate on code re-organization for increasing the degree of temporal and spatial code locality.
- Consider two specific examples,
    - Loop interchange and Blocking.

(a) <u>Loop Interchange</u>

- Objective – Improve spatial locality
- Methodology
    - Cache does no have capacity for all matrix content.
    - Order of loop nesting provides good spatial locality if data accessed in the order in which it originally appears.
    - Cache thrashing results if cache capacity or conflicts stop all data appearing in cache "as one".
- Consider,

Unoptimized code	Optimized code

`for (j = 0; j < 100; j++)`
`{`
`    for (i = 0; i < 5000; i++)`
`    {`
`        x[i][j] = 2 * x[i][j];`
`    }`
`}`
	`for (i = 0; i < 5000; i++)`
	`{`
	`    for (j = 0; j < 100; j++)`
	`    {`
	`        x[i][j] = 2 * x[i][j];`
	`    }`
	`}`

(b) Blocking

- Objective – Improve Temporal locality of data.
- Problem
    - Interested in accessing same data in both row and column format.
    - Cache does not have the capacity for any more than a row or column.
        - i.e. reading entire row or column into cache is optimally wrong for one set of operations.
- Methodology
    - Complete all row and column operations on 'block' of data in one pass.
- Consider the following code,

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

- What is the linear math equivalent?
    - One value for 'r' requires all row of 'y' and entire column from 'z'.
- Process summarized by figure 4,
- Potential for domination by capacity misses.
    - If the cache may hold all 3 N by N matrices, then no problem.
    - If cache may hold one N by N matrix (z) and one row (y) then flush each row of 'y' N times.
    - If less than $N + N \times N$ then likelihood of cache thrashing increases.
    - Worst case capacity misses of $2N^3 + N^2$ accesses for N^3 operations.

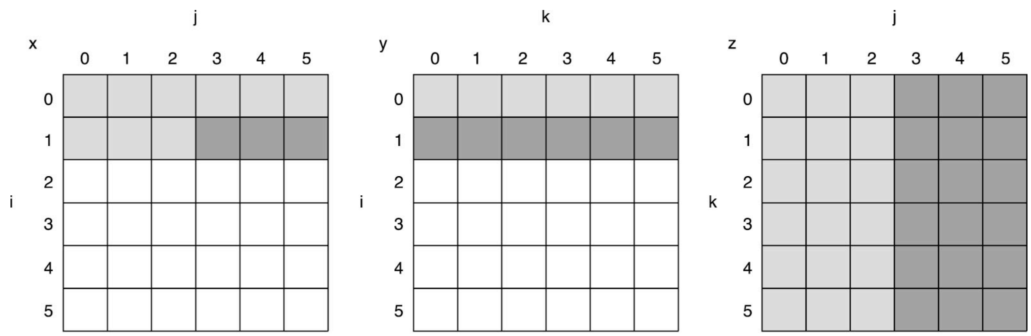

Figure 4: Naïve matrix multiplication (outer product) for index i = 1. (LHS is the result matrix)

Dark gray – recent access; Light gray – old access; White – no access.

- Solution
    - Divide the cache into sub-blocks, size B by B;
    - Value for result 'r' built up incrementally over all blocks.
    - Inner loops compute over the sub-blocks, figure 5;

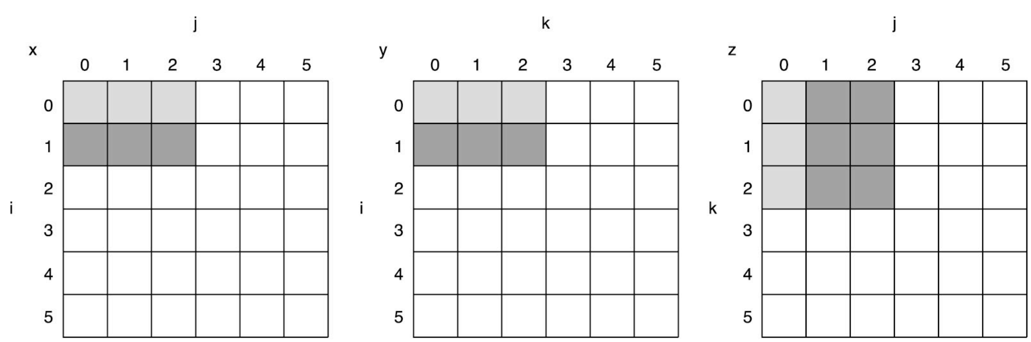

Figure 5: Block based outer product multiplication. Case of B = 3. (LHS is the result matrix)

- 'B' the blocking factor is selected to ensure that sub-blocks for all three variables (x, y, z) reside in cache (or registers).
- Number of capacity misses is now $2N^3 \div B + N^2$ accesses for N^3 operations.
    - I.e. speedup of 'B'
    - Following code illustrates a possible 'block based' implementation.

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

Summary – Reducing Cache Miss Rate

- Three C's model,
    - Large block size – reduce compulsory misses;
    - Large cache size – reduce capacity misses;
    - High associativity – reduce conflict misses.
    - Model ignores effects on hit time and miss penalties.
- Way Prediction
    - Reduces hit time with higher set associativity designs.
- Compiler optimizations
    - Improve organization of memory accesses.
    - Require compile time knowledge of parameters.