

CMSC 611: Advanced Computer Architecture

Cache (2)

Classifying Cache Misses

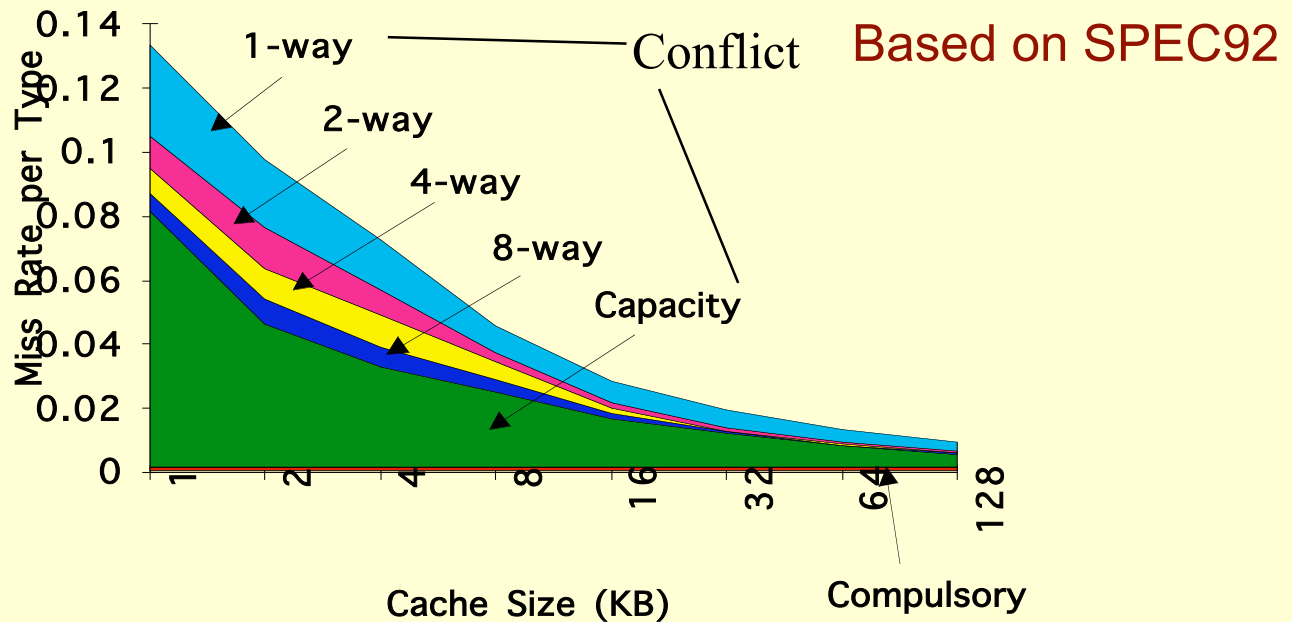
- Compulsory
 - First access to a block not in cache
 - Also called cold start or first reference misses
 - (Misses in even an Infinite Cache)
- Capacity
 - If the cache cannot contain all needed blocks
 - Due to blocks discarded and re-retrieved
 - (Misses in Fully Associative Cache)
- Conflict
 - Set associative or direct mapped: too many blocks in set
 - Also called collision or interference
 - (Misses in N-way Associative Cache)

Improving Cache Performance

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

Miss Rate Distribution

- Compulsory misses are small compared to other categories
- Capacity misses diminish with increased cache size
- Increasing associativity limits the placement conflicts

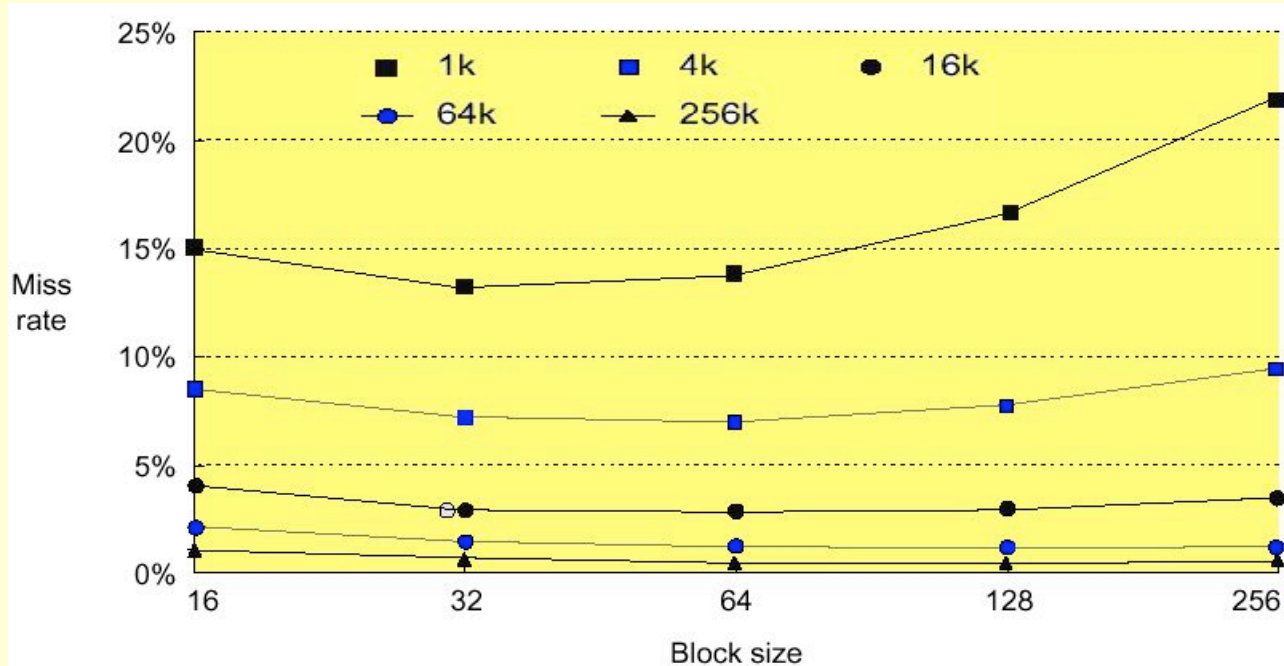


Techniques for Reducing Misses

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times Miss\ penalty \right) \times Clock\ cycle\ time$$

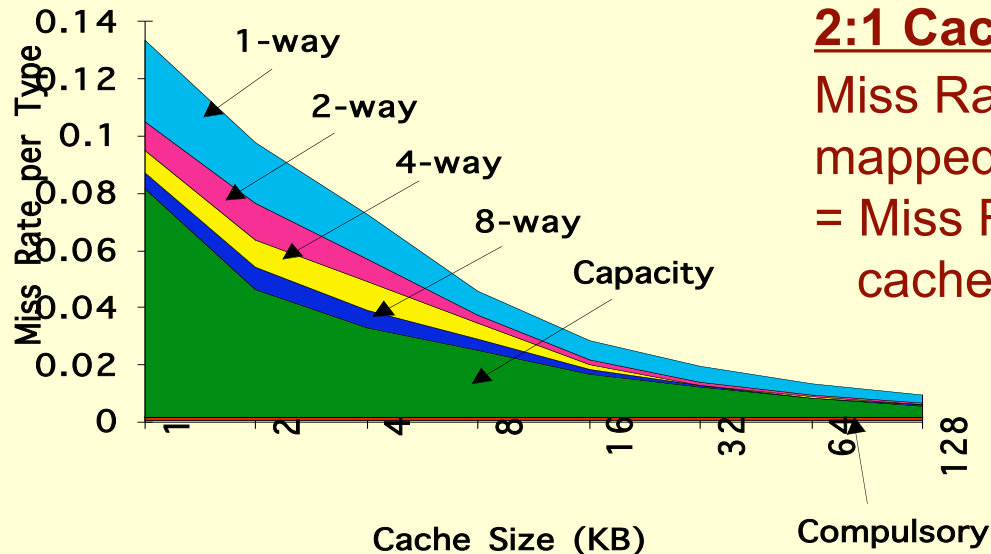
1. Reducing Misses via Larger Block Size
2. Reducing Misses via Higher Associativity
3. Reducing Misses via Victim Cache
4. Reducing Misses via Pseudo-Associativity
5. Reducing Misses by H/W Prefetching Instr. and Data
6. Reducing Misses by S/W Prefetching Data
7. Reducing Misses by Compiler Optimizations

Reduce Misses via Larger Block Size



- Larger block sizes reduces compulsory misses (principle of spatial locality)
- Conflict misses increase for larger block sizes since cache has fewer blocks
- The miss penalty usually outweighs the decrease of the miss rate making large block sizes less favored

Reduce Misses via Higher Associativity



- Greater associativity comes at the expense of larger hit access time
- Hardware complexity grows for high associativity and clock cycle increases

Example

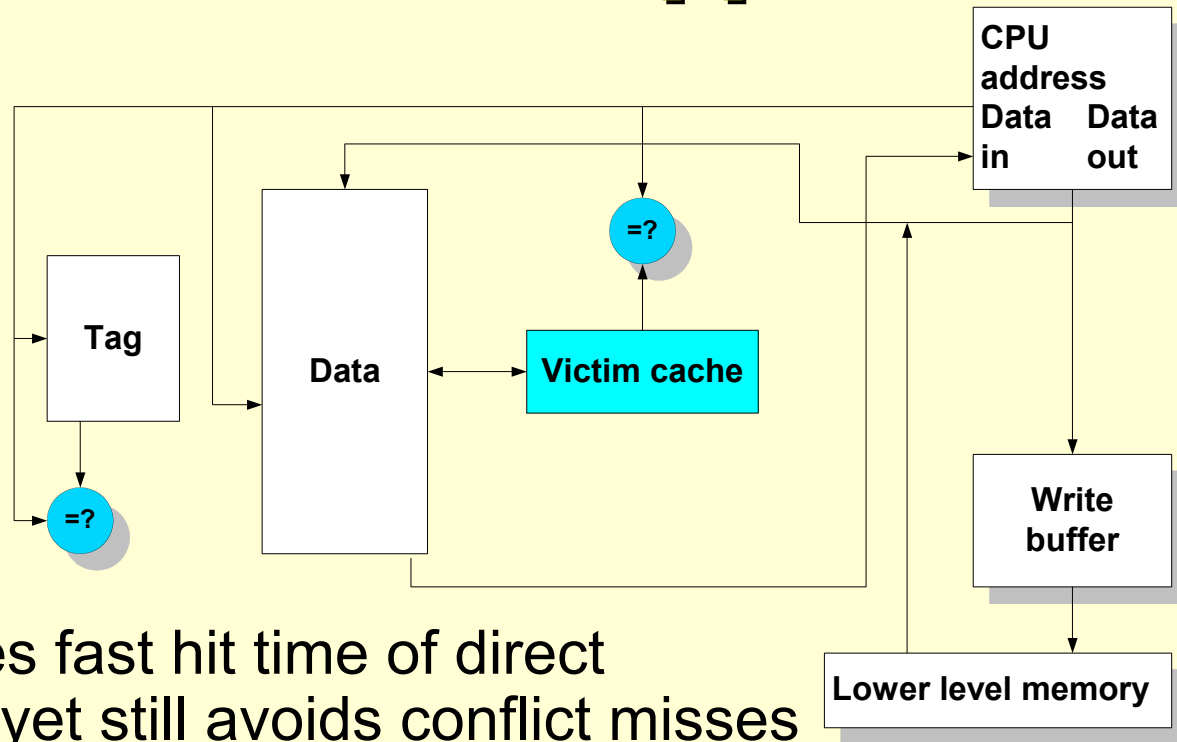
Assume hit time is 1 clock cycle and average miss penalty is 50 clock cycles for a direct mapped cache. The clock cycle increases by a factor of 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way associative cache. Compare the average memory access based on the previous figure miss rates

Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	1.79
64	1.70	1.60	1.57	1.59
128	1.50	1.45	1.42	1.44

A good size of direct mapped cache can be very efficient given its simplicity

High associativity becomes a negative aspect

Victim Cache Approach



- Combines fast hit time of direct mapped yet still avoids conflict misses
 - Adds small fully associative cache between the direct mapped cache and memory to place data discarded from cache
 - Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
 - Technique is used in Alpha, HP machines and does not impair the clock rate

Pseudo-Associativity Mechanism

- Combine fast hit time of Direct Mapped and lower conflict misses of 2-way set associative
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit
- Simplest implementation inverts the index field MSB to find the other pseudo set
- To limit the impact of hit time variability on performance, swap block contents
- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
 - Better for caches not tied directly to processor (L2)
 - Used in MIPS R1000 L2 cache, similar in UltraSPARC

H/W Pre-fetching of Instructions & Data

- Hardware pre-fetches instructions and data while handling other cache misses
 - Assume pre-fetched items will be referenced shortly
- Pre-fetching relies on having extra memory bandwidth that can be used without penalty

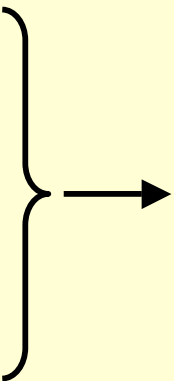
$$\text{Average memory access time} = \text{Hit time} + \text{Miss Rate} \times (\text{Prefetch hit rate} + (1 - \text{Prefetch hit rate}) \times \text{Miss penalty})$$

- Examples of Instruction Pre-fetching:
 - Alpha 21064 fetches 2 blocks on a miss
 - Extra block placed in “stream buffer”
 - On miss check stream buffer
- Works with data blocks too:
 - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
 - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches

Software Pre-fetching Data

- Uses special instructions to pre-fetch data:
 - Load data into register (HP PA-RISC loads)
 - Cache Pre-fetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
- Special pre-fetching instructions cannot cause faults (undesired exceptions) since it is a form of speculative execution
- Makes sense if the processor can proceed 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)
- Issuing Pre-fetch Instructions takes time
 - Is cost of pre-fetch issues < savings in reduced misses?
 - Higher superscalar reduces difficulty of issue bandwidth

```
for (i = 0; i < 3; i = i+1)
  for (j = 0; j < 100; j = j+1)
    a[i][j] = b[j][0] * b[j+1][0];
```



```
for (j = 0; j < 100; j = j+1)
  pre-fetch (b[i+7][0]);
  a[0][j] = b[j][0] * b[j+1][0];
  for (i = 1; i < 3; i = i+1)
    pre-fetch (a[i][j+7]);
    a[i-1][j] = b[j][0] * b[j+1][0];
```

Compiler-based Cache Optimizations

- Compiler-based cache optimization reduces the miss rate without any hardware change
- McFarling [1989] reduced caches misses by 75% (8KB direct mapped / 4 byte blocks)

For Instructions

- Reorder procedures in memory to reduce conflict
- Profiling to determine likely conflicts among groups of instructions

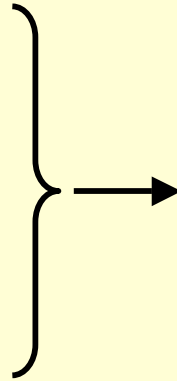
For Data

- **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

Examples

Merging Arrays:

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

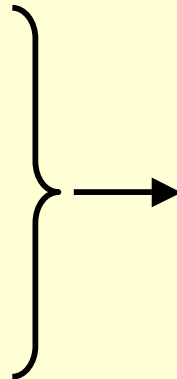


```
/* After: 1 array of structures */  
struct merge {  
    int val;  
    int key;  
};  
struct merge merged_array[SIZE];
```

- Reduces misses by improving spatial locality through combined arrays that are accessed simultaneously

Loop Interchange:

```
/* 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];
```

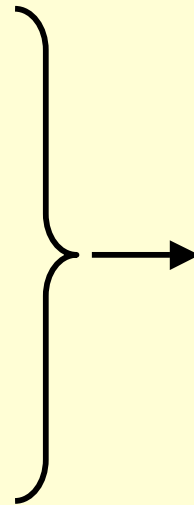
- Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Loop Fusion Example

- Some programs have separate sections of code that access the same arrays (performing different computation on common data)
- Fusing multiple loops into a single loop allows the data in cache to be used repeatedly before being swapped out
- Loop fusion reduces misses through improved temporal locality (rather than spatial locality in array merging and loop interchange)

/ 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];
  }
```

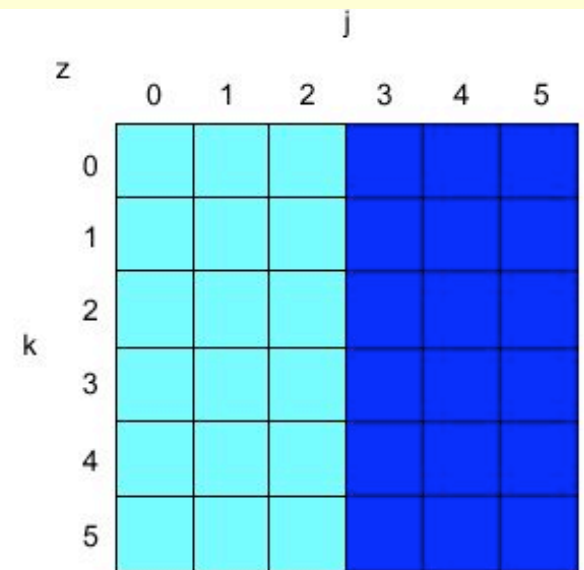
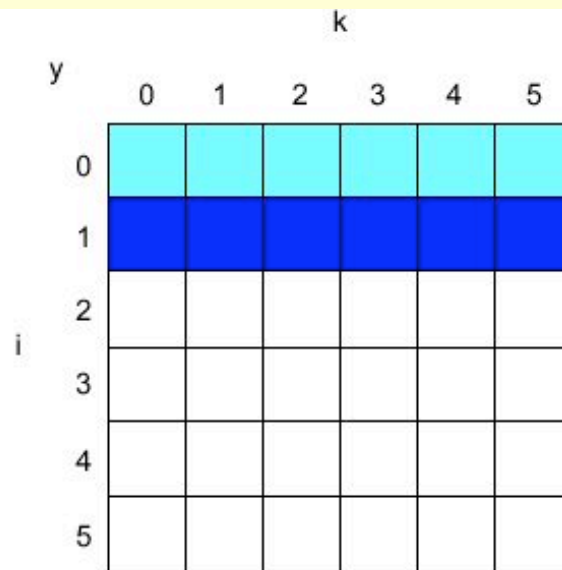
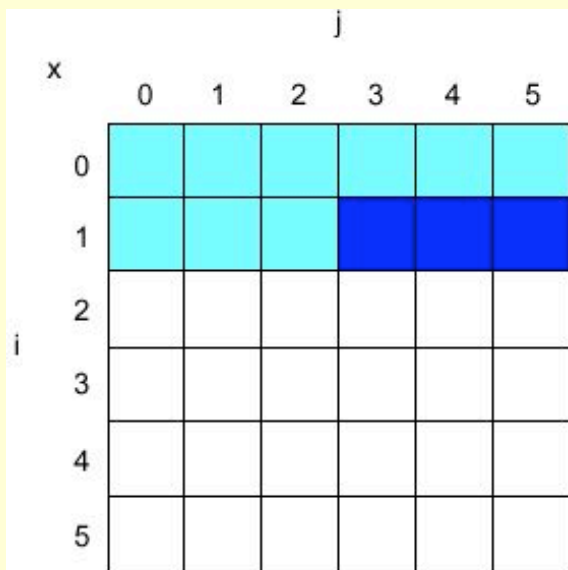
Accessing array “a” and “c” would have caused twice the number of misses without loop fusion

Blocking Example

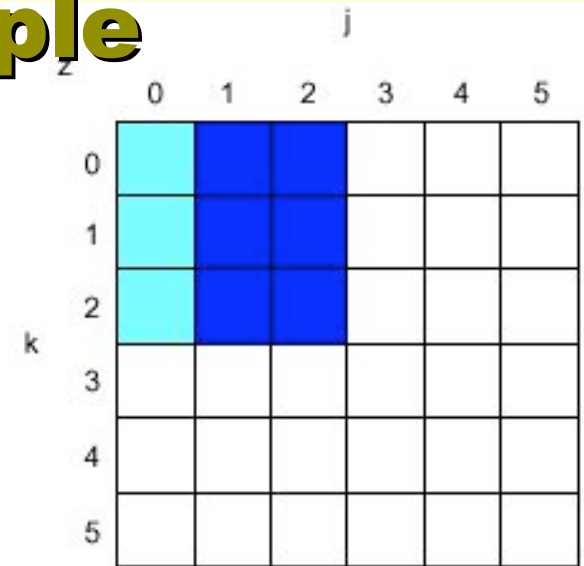
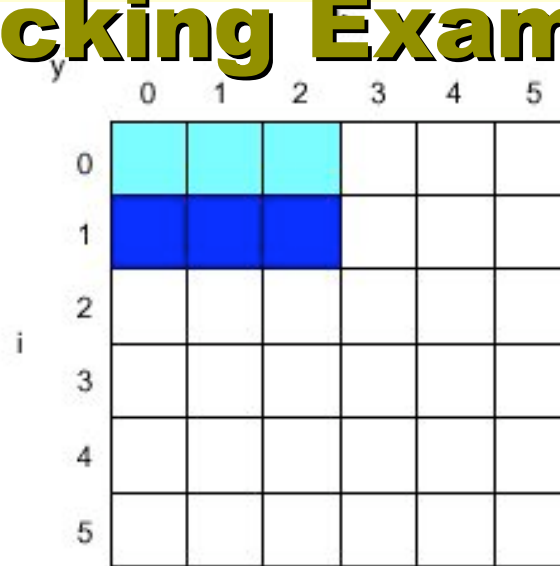
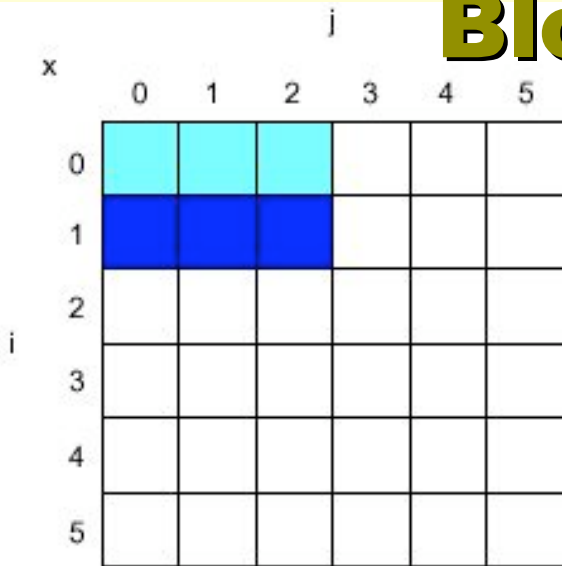
```
/* Before */
```

```
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;
  }
```

- Two Inner Loops:
 - Read all $N \times N$ elements of $z[]$
 - Read N elements of 1 row of $y[]$ repeatedly
 - Write N elements of 1 row of $x[]$
- Capacity Misses a function of N & Cache Size:
 - $3 \times N \times N \times 4$ bytes \Rightarrow no capacity misses;
- Idea: compute on $B \times B$ sub-matrix that fits



Blocking Example



/* After */

```

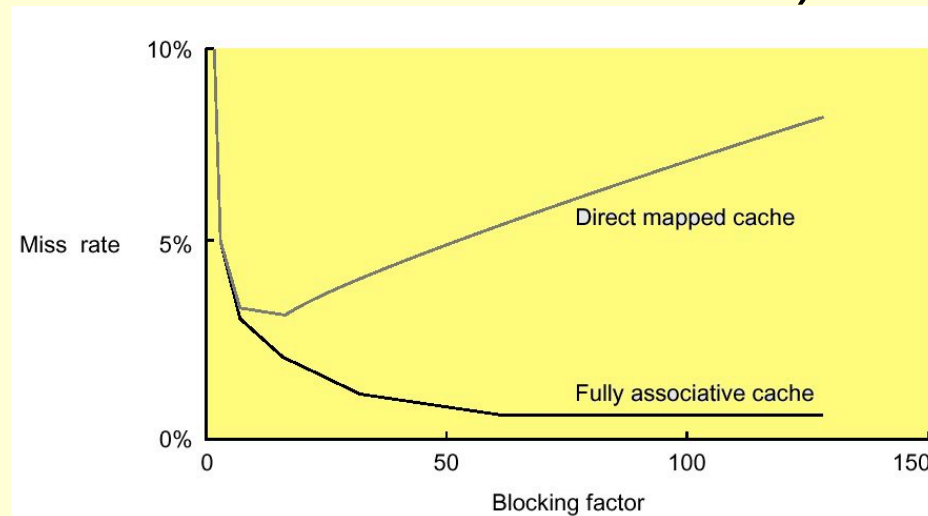
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
  for (j = jj; j < min(jj+B-1,N); j = j+1) {
    r = 0;
    for (k = kk; k < min(kk+B-1,N); k = k+1) {
      r = r + y[i][k] * z[k][j];
    }
    x[i][j] = x[i][j] + r;
  }

```

- B called *Blocking Factor*
- Memory words accessed $2N^3 + N^2 \rightarrow 2N^3/B + N^2$
- Conflict misses can go down too
- Blocking is also useful for register allocation

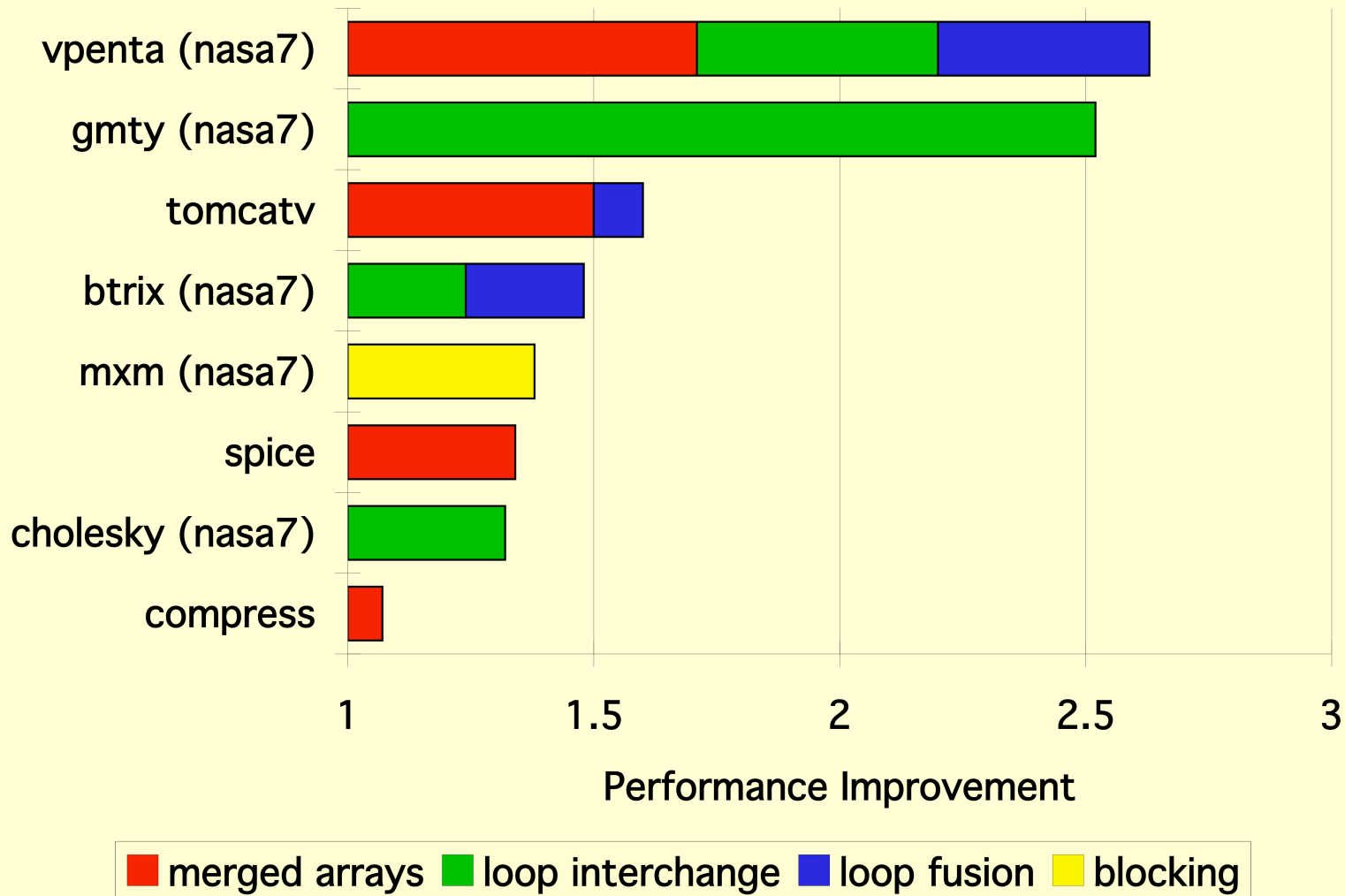
Blocking Factor

- Traditionally blocking is used to reduce capacity misses relying on high associativity to tackle conflict misses
- Choosing smaller blocking factor than the cache capacity can also reduce conflict misses (fewer words are active in cache)



Lam et al [1991] a blocking factor of 24 had a fifth the misses compared to a factor of 48 despite both fit in cache

Efficiency of Compiler-Based Cache Opt.



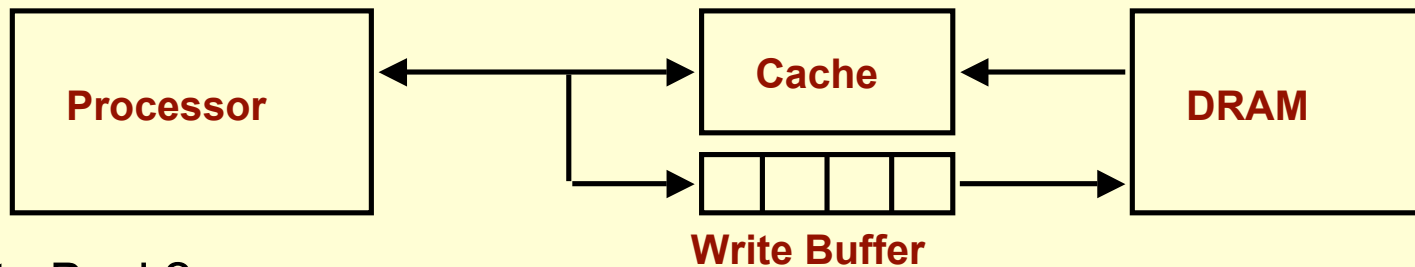
Reducing Miss Penalty

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times Miss\ rate \times Miss\ penalty \right) \times Clock\ cycle\ time$$

- Reducing miss penalty can be as effective as the reducing miss rate
- With the gap between the processor and DRAM widening, the relative cost of the miss penalties increases over time
- Seven techniques
 - Read priority over write on miss
 - Sub-block placement
 - Merging write buffer
 - Victim cache
 - Early Restart and Critical Word First on miss
 - Non-blocking Caches (Hit under Miss, Miss under Miss)
 - Second Level Cache
- Can be applied recursively to Multilevel Caches
 - Danger is that time to DRAM will grow with multiple levels in between
 - First attempts at L2 caches can make things worse, since increased worst case is worse

Read Priority over Write on Miss

- Write through with write buffers offer RAW conflicts with main memory reads on cache misses
- If simply wait for write buffer to empty, might increase read miss penalty (old MIPS 1000 by 50%)
- Check write buffer contents before read; if no conflicts, let the memory access continue

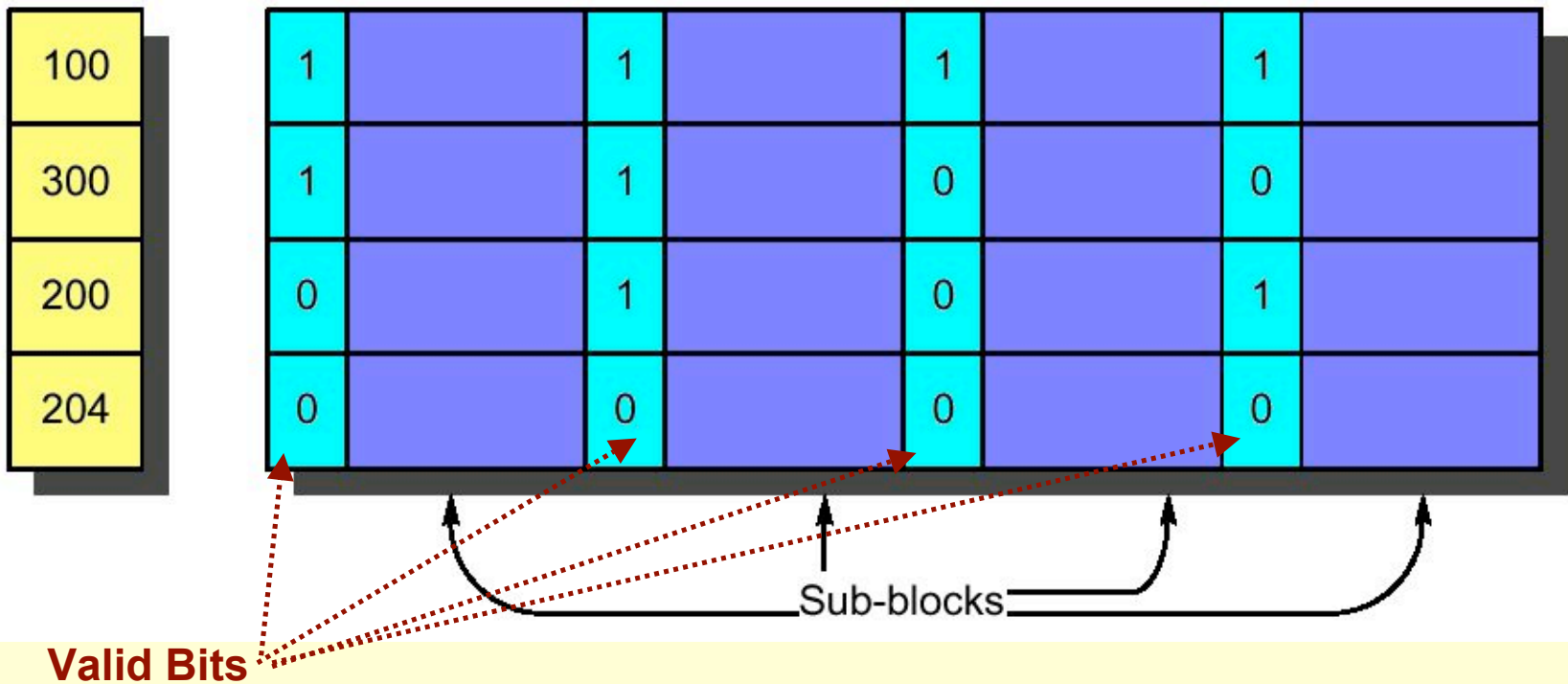


Write Back?

- ➔ Read miss replacing dirty block
- ➔ Normal: Write dirty block to memory, and then do the read
- ➔ Instead copy the dirty block to a write buffer, then do the read, and then do the write
- ➔ CPU stall less since restarts as soon as do read

Sub-block Placement

- Originally invented to reduce tag storage while avoiding the increased miss penalty caused by large block sizes
- Enlarge the block size while dividing each block into smaller units (sub-blocks) and thus does not have to load full block on a miss
- Include valid bits per sub-block to indicate the status of the sub-block (in cache or not)



Merging Write Buffer

Write address	V	V	V	V		
100	1	Mem[100]	0	0	0	0
108	1	Mem[108]	0	0	0	0
116	1	Mem[116]	0	0	0	0
124	1	Mem[124]	0	0	0	0

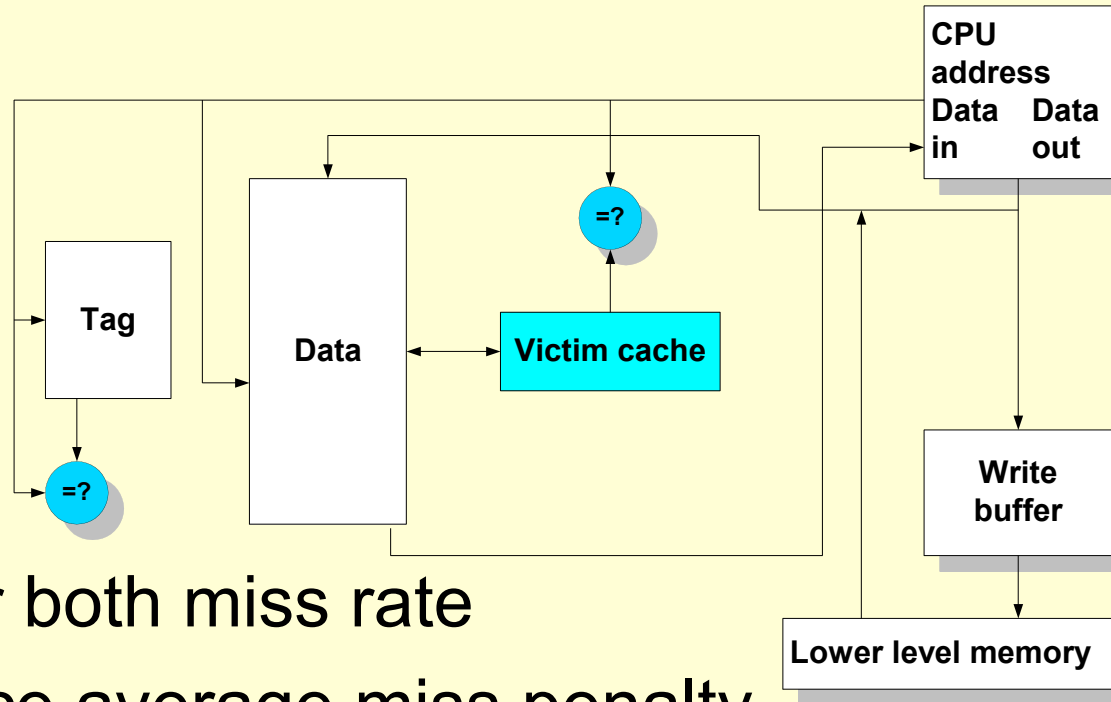
Buffer is full

Write address	V	V	V	V				
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Consolidation free up space

Extend the concept of sub-block by optimizing the write buffer handling

Victim Cache Approach



- Lower both miss rate
- Reduce average miss penalty
- Slightly extend the worst case miss penalty

Early Restart and Critical Word First

- Don't wait for full block to be loaded before restarting CPU
 - **Early restart**
 - As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
 - **Critical Word First**
 - Request the missed word first from memory
 - Also called wrapped fetch and requested word first



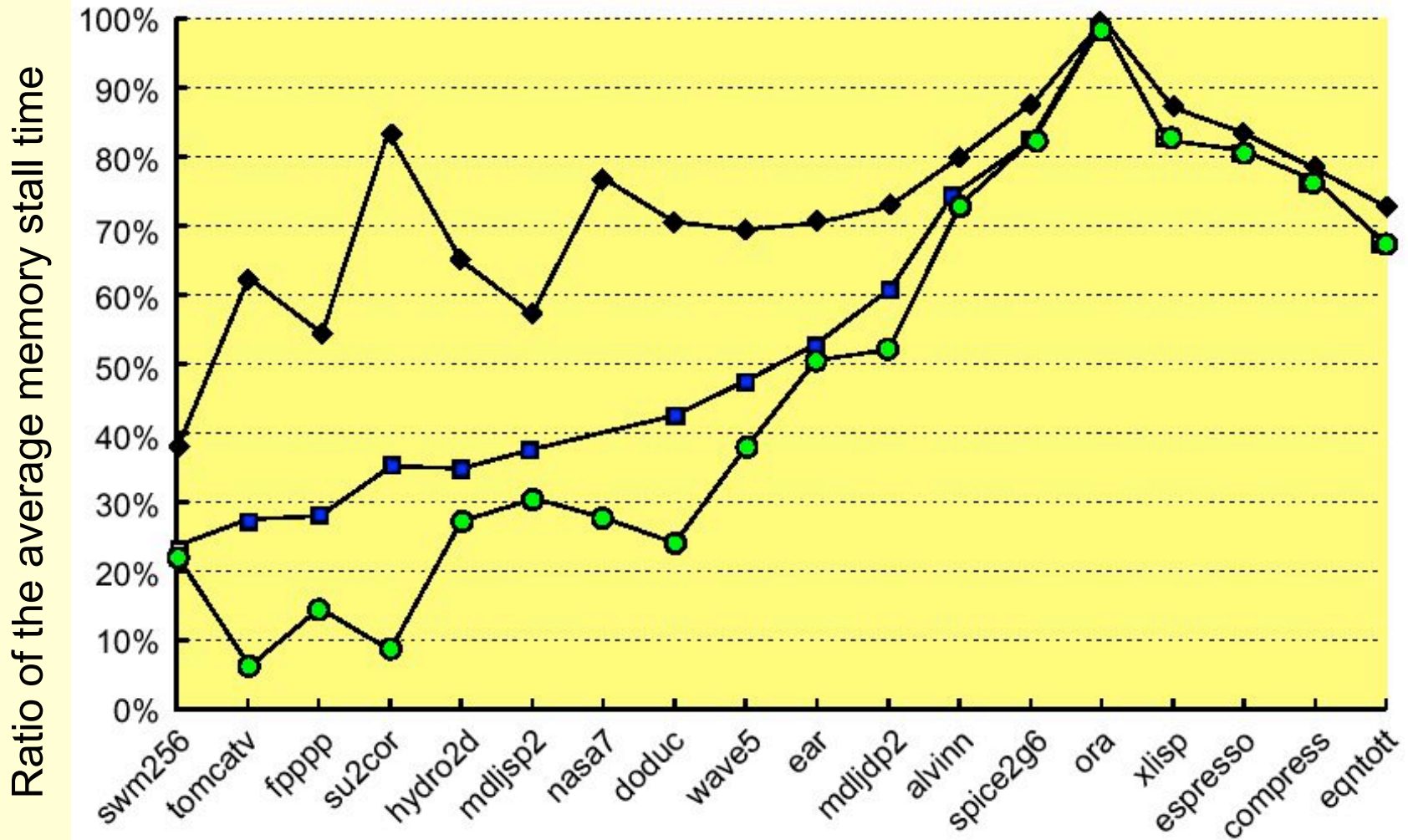
- Complicates cache controller design
- CWF generally useful only in large blocks
- Given spatial locality programs tend to want next sequential word, limits benefit

Non-blocking Caches

- Early restart still waits for the requested word to arrive before the CPU can continue execution
- For machines that allows out-of-order execution using a scoreboard or a Tomasulo-style control the CPU should not stall on cache misses
- “**Non-blocking cache**” or “**lock-free cache**” allows data cache to continue to supply cache hits during a miss
- “**hit under miss**” reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- “**hit under multiple miss**” or “**miss under miss**” may further lower the effective miss penalty by overlapping multiple misses
 - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
 - Requires multiple memory banks (otherwise cannot support)
 - Pentium Pro allows 4 outstanding memory misses

Performance of Non-blocking Caches

◆ Hit under 1 miss ■ Hit under 2 misses ● Hit under 64 misses



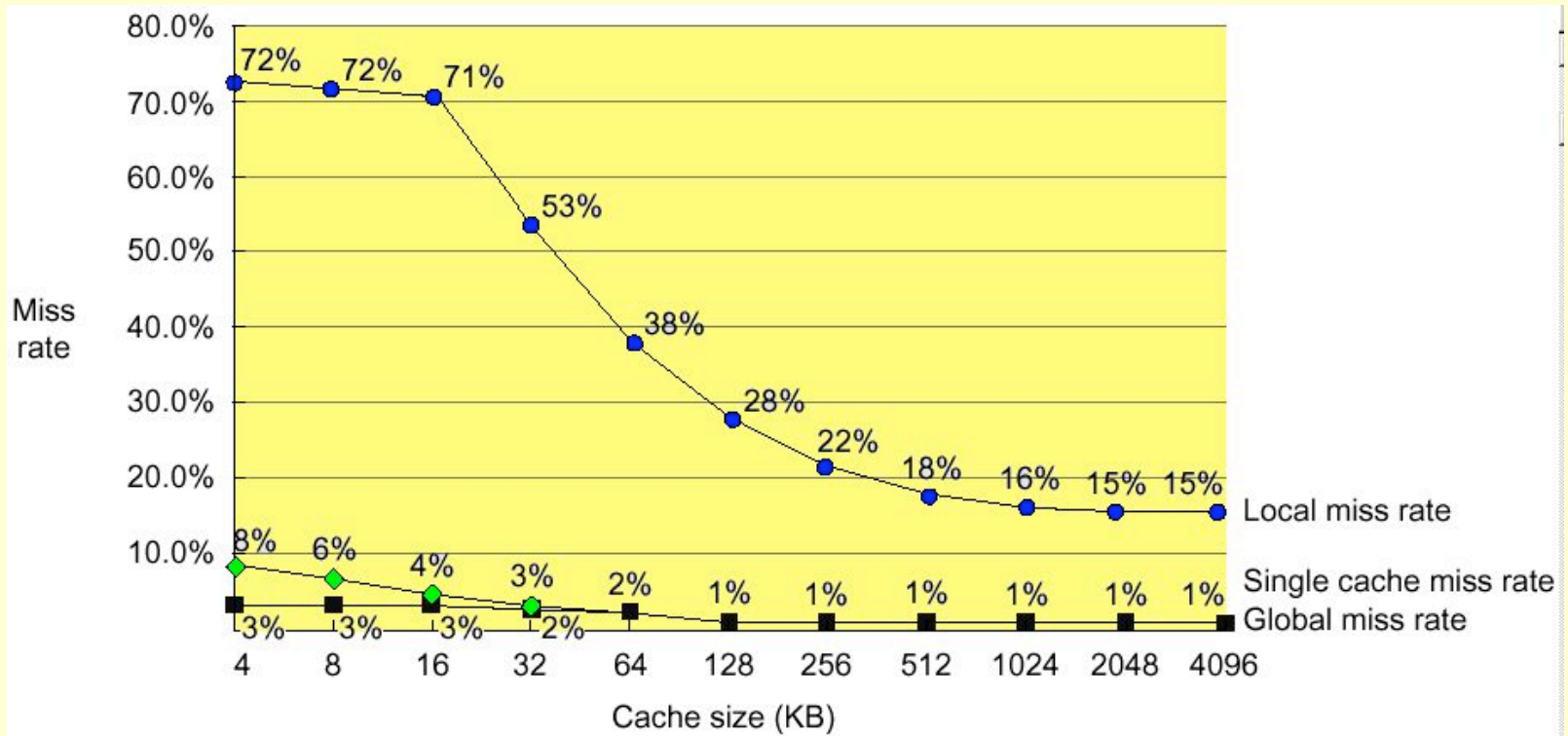
Second Level Cache

- The previous techniques reduce the impact of the miss penalty on the CPU
 - L2 cache handles the cache-memory interface
- Measuring cache performance

$$\begin{aligned} \text{AMAT} &= \text{HitTime}_{L1} + \text{MissRate}_{L1} \times \text{MissPenalty}_{L1} \\ &= \text{HitTime}_{L1} + \text{MissRate}_{L1} \times (\text{HitTime}_{L2} + \text{MissRate}_{L2} \times \text{MissPenalty}_{L2}) \end{aligned}$$

- Local miss rate
 - misses in this cache divided by the total number of memory accesses to this cache (MissRate_{L2})
- Global miss rate (& biggest penalty!)
 - misses in this cache divided by the total number of memory accesses generated by the CPU ($\text{MissRate}_{L1} \times \text{MissRate}_{L2}$)

Local vs Global Misses



(Global miss rate close to single level cache rate provided $L2 \gg L1$)