CMSC 611: Advanced Computer Architecture

Cache

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides

Techniques for Reducing Misses

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

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

Compiler-based Cache Optimizations

- Complier-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

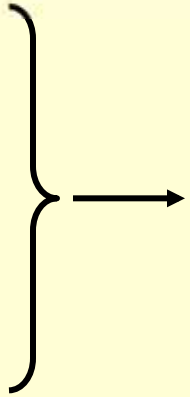
For Data

- 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];
```

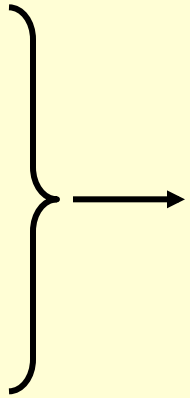


```
/* After: 1 array of stuctures */
struct merge {
        int val;
        int key;
};
struct merge merged_array[SIZE];
```

- 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];
```



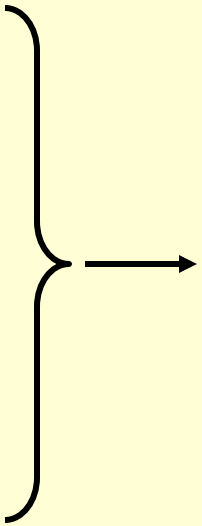
```
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];
```

- 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 missed 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];
```



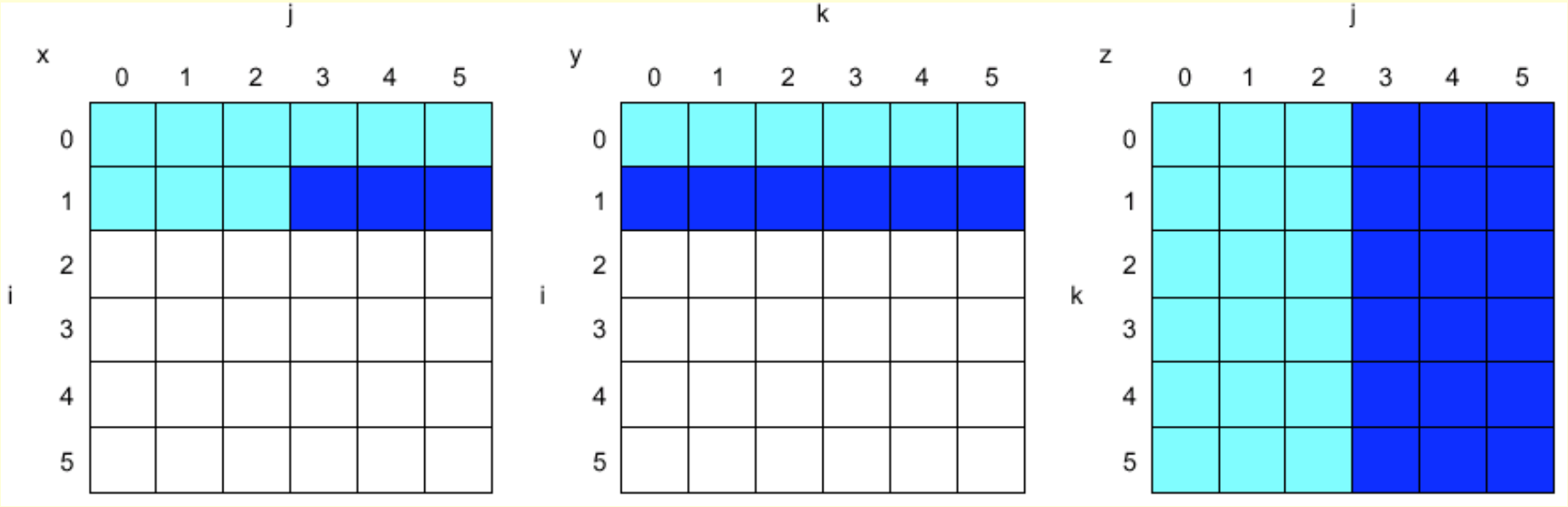
```
/* After */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1) {
a[i][j] = 1/b[i][j] * c[i][j];
          d[i][j] = a[i][j] + c[i][j];
}
```

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

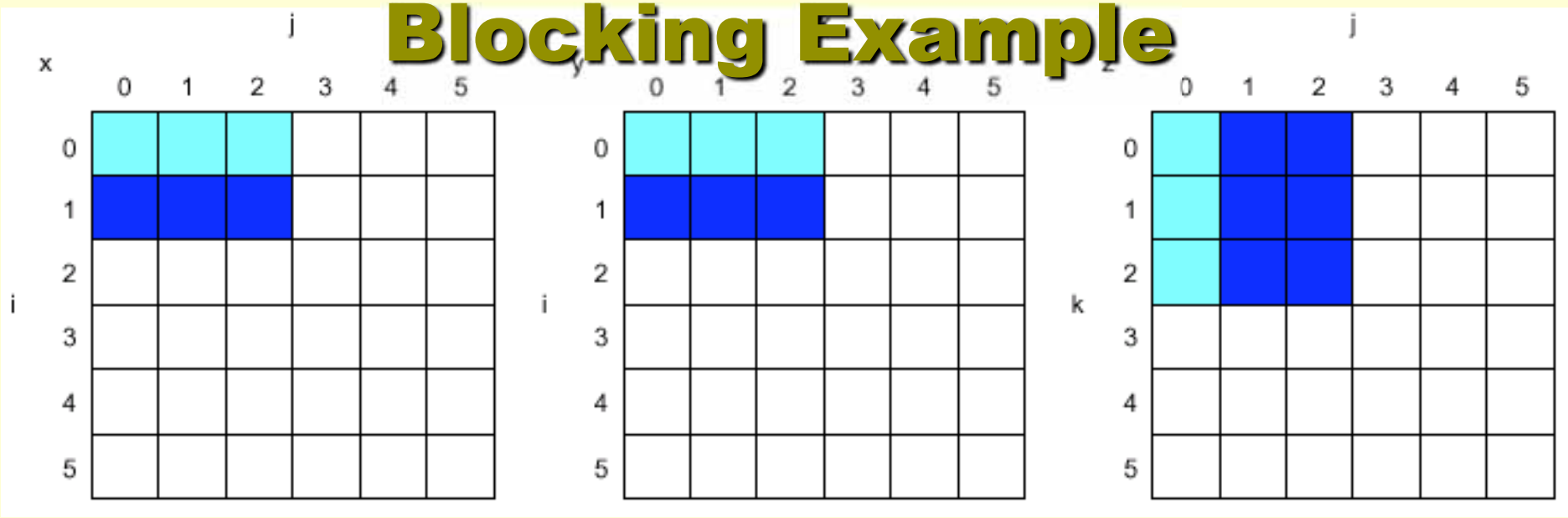
Blocking Example

```
/* 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 NxN 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 => no capacity misses;
- Idea: compute on $B \times B$ sub-matrix that fits



Blocking Example

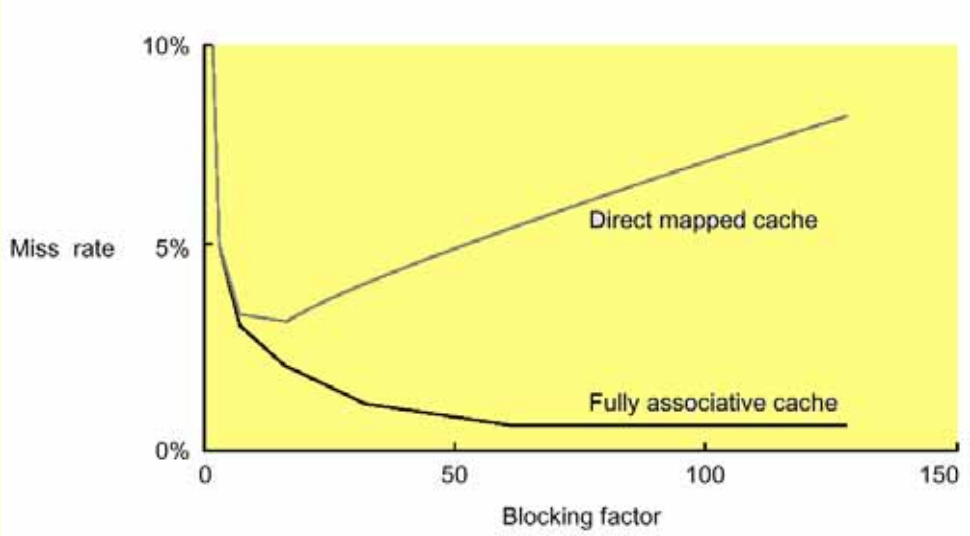



```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1) {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1) {
            r = r + y[i][k] * z[k][j];};
        x[i][j] = x[i][j] + r;
       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

- 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

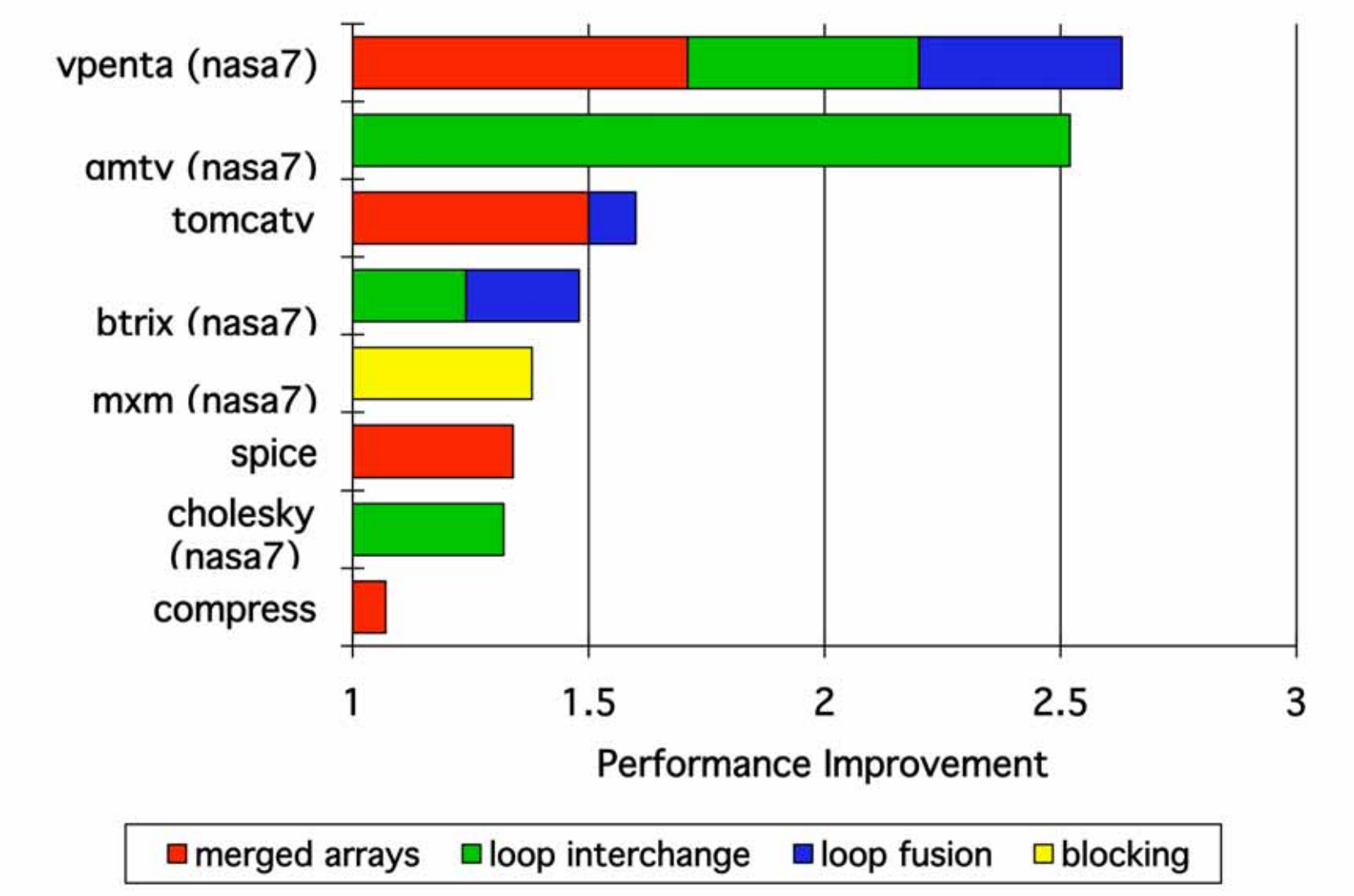
Blocking Factor

- 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.



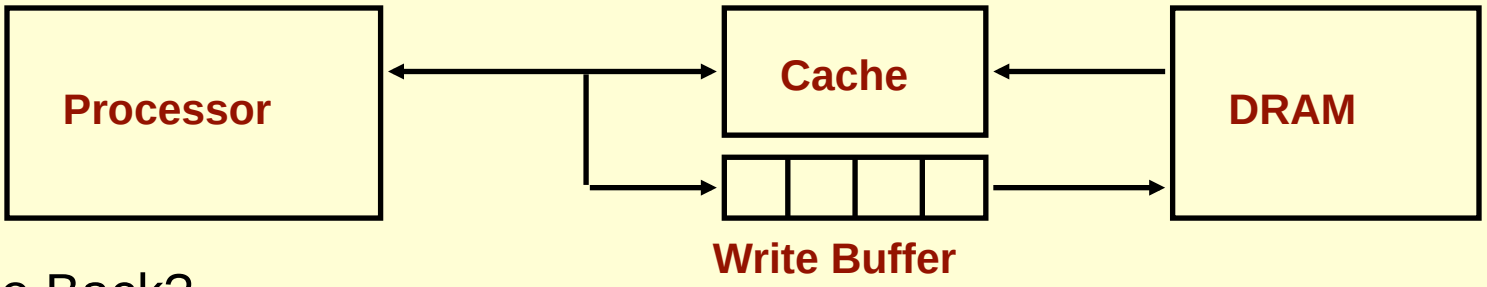
Reducing Miss Penalty

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

- 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

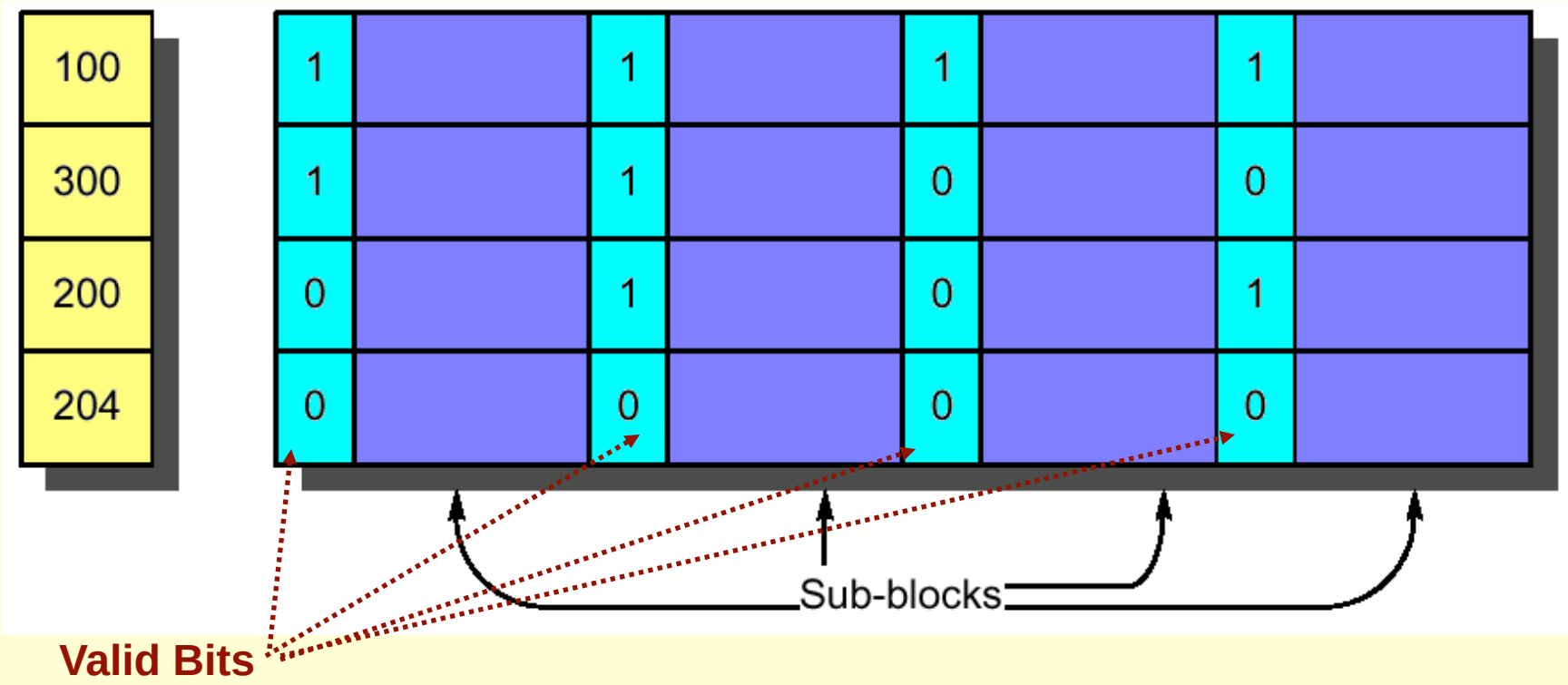


❑Write Back?

- ➨ 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

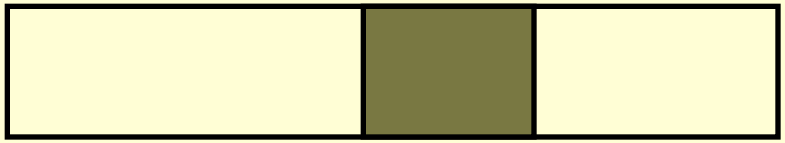
Sub-block Placement

- 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)



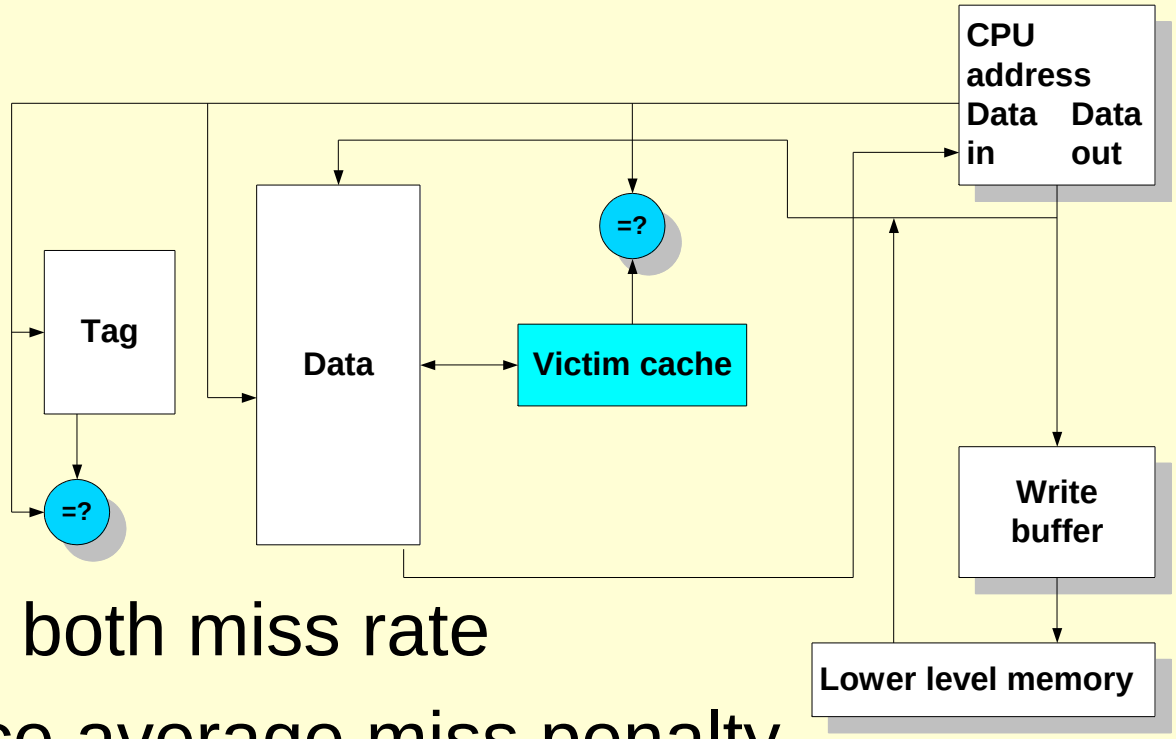
Early Restart and Critical Word First

- 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

Victim Cache Approach

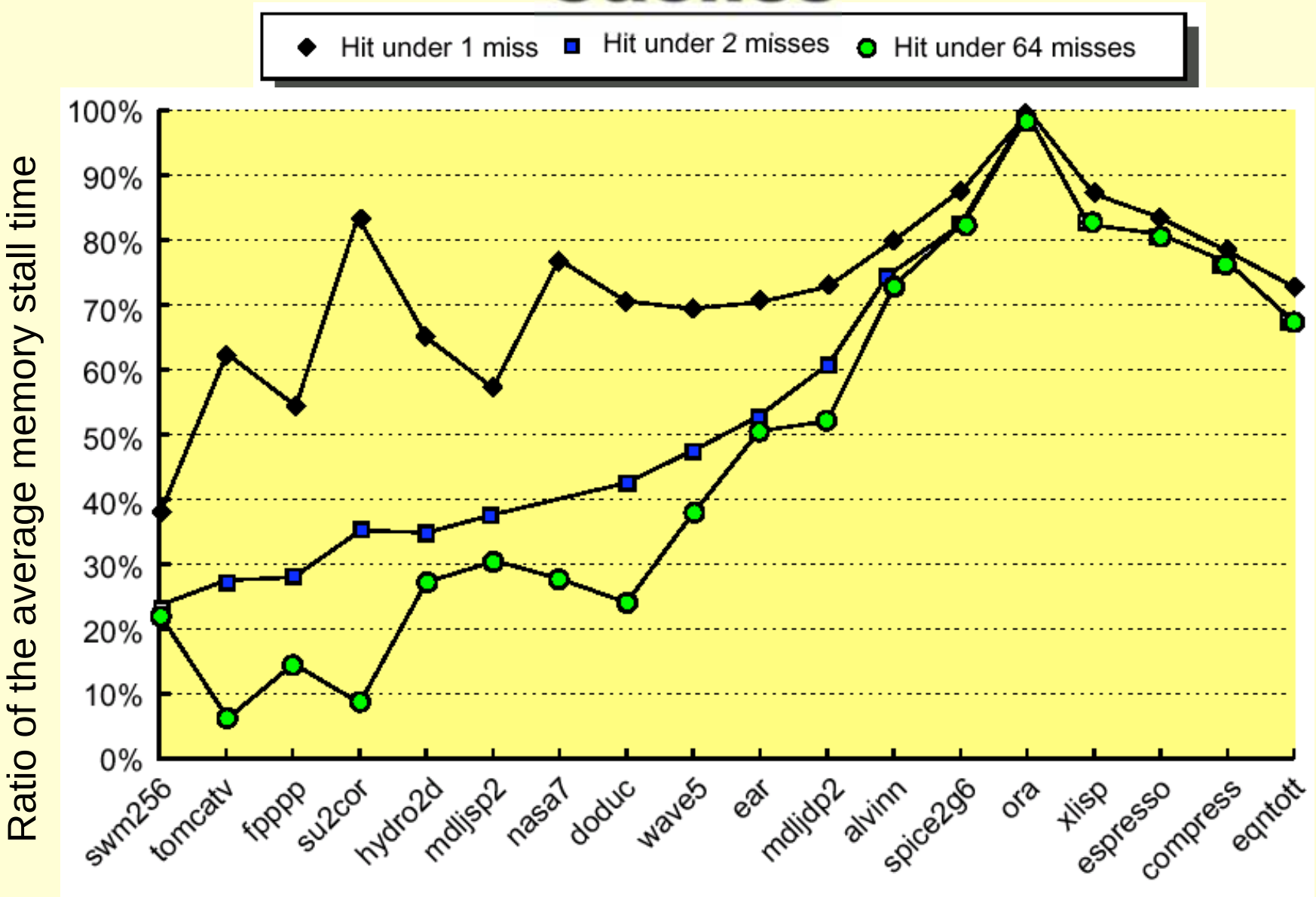


- Lower both miss rate
- Reduce average miss penalty
- Slightly extend the worst case miss penalty

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



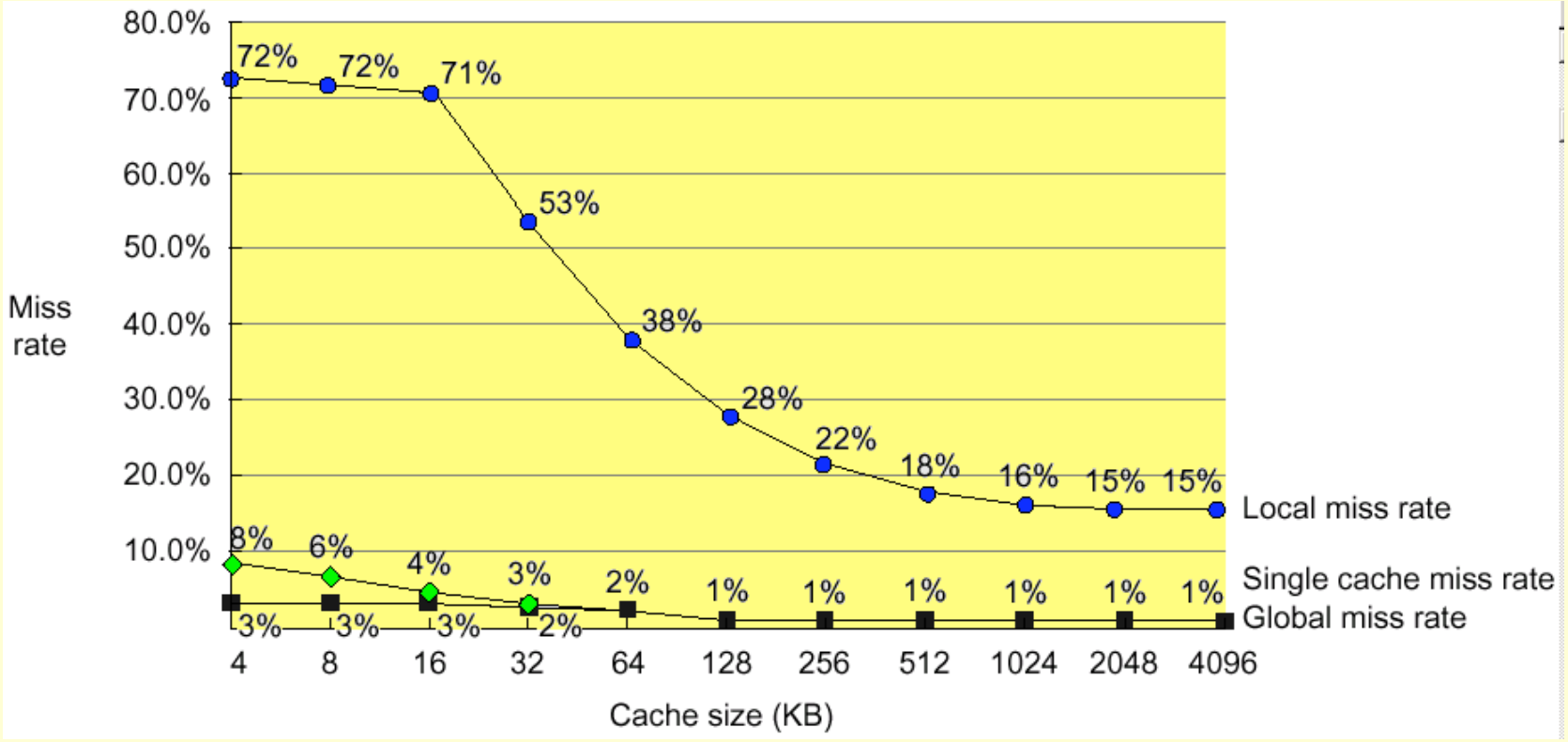
Second Level Cache

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

$$\begin{aligned} AMAT &= HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1} \\ &= HitTime_{L1} + MissRate_{L1} \times (HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}) \end{aligned}$$

- 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 ($MissRate_{L1} \times MissRate_{L2}$)

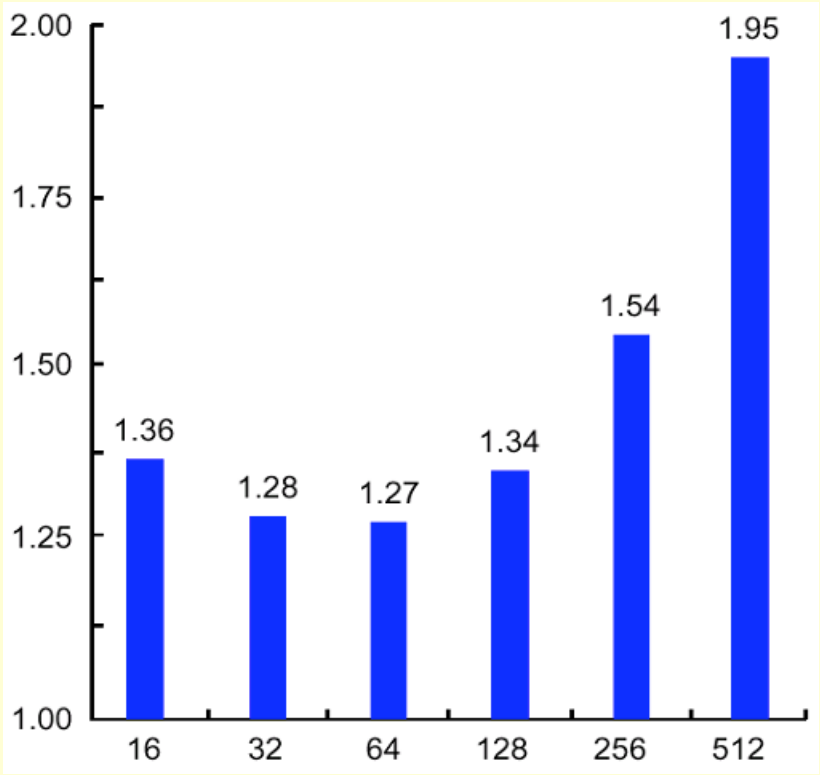
Local vs Global Misses



(Global miss rate close to single level cache rate provided L2 >> L1)

L2 Cache Parameters

- 32 bit bus
- 512KB cache



- L1 cache directly affects the processor design and clock cycle: should be simple and small
- Bulk of optimization techniques can go easily to L2 cache
- Miss-rate reduction more practical for L2
- Considering the L2 cache can improve the L1 cache design,
  - e.g. use write-through if L2 cache applies write-back

Reducing Hit Time

Average Access Time = Hit Time x (1 - Miss Rate) + Miss Time x Miss Rate

- Hit rate is typically very high compared to miss rate
  - any reduction in hit time is magnified
- Hit time critical: affects processor clock rate
- Three techniques to reduce hit time:
  - Simple and small caches
  - Avoid address translation during cache indexing
  - Pipelining writes for fast write hits

Simple and small caches

- Design simplicity limits control logic complexity and allows shorter clock cycles
- On-chip integration decreases signal propagation delay, thus reducing hit time
  - Alpha 21164 has 8KB Instruction and 8KB data cache and 96KB second level cache to reduce clock rate