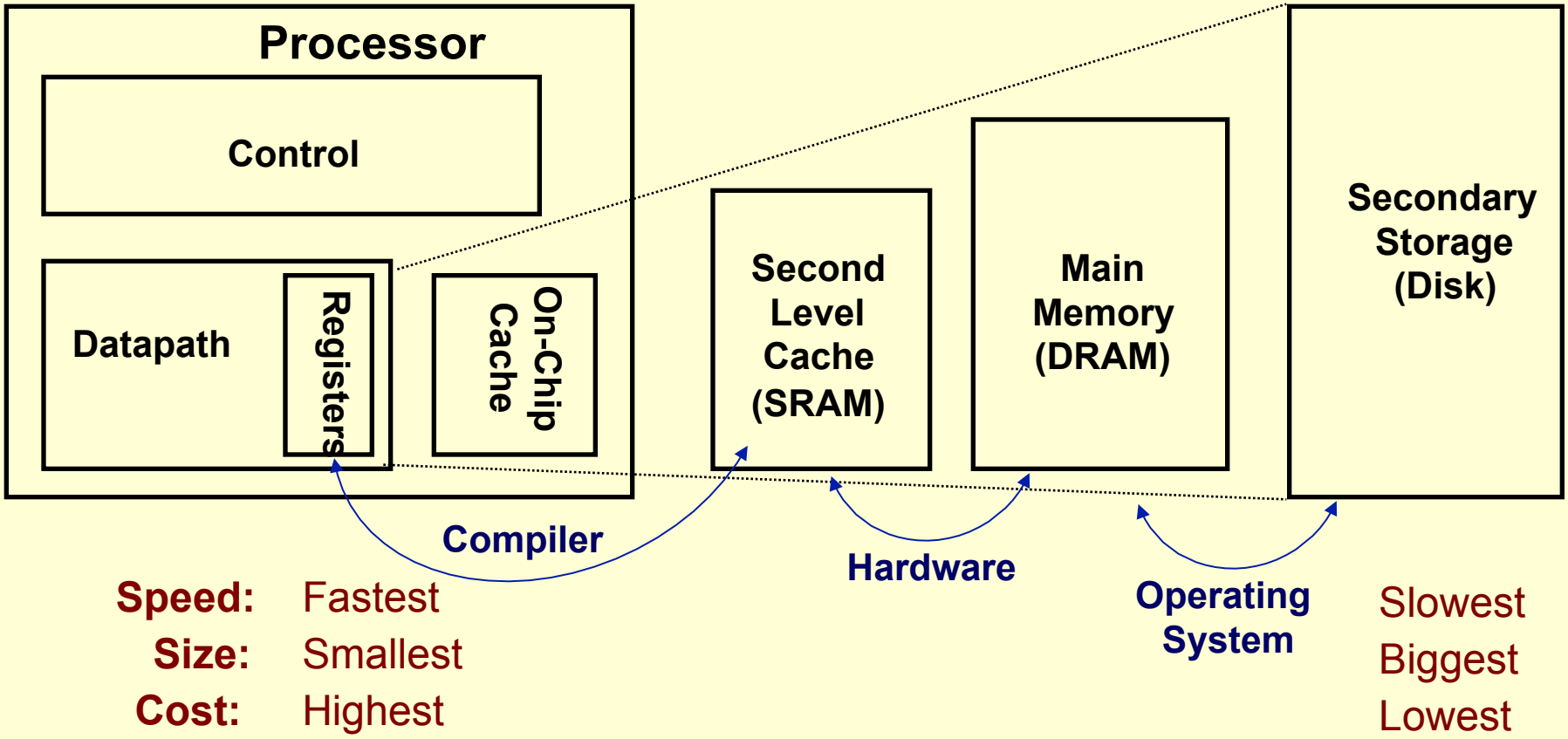
# CMSC 611: Advanced Computer Architecture

## Cache

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science

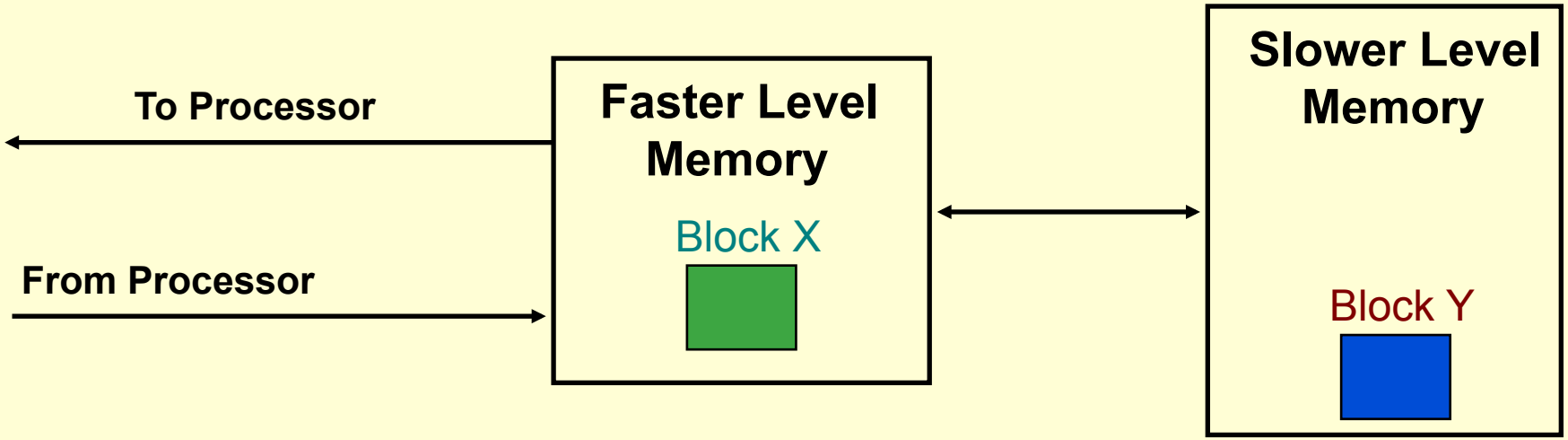
# Memory Hierarchy

- Temporal Locality (Locality in Time):
  ⇒ Keep most recently accessed data items closer to the processor
- Spatial Locality (Locality in Space):
  ⇒ Move blocks consists of contiguous words to the faster levels



# Memory Hierarchy Terminology

- Hit: data appears in some block in the faster level (example: Block X)
  - Hit Rate: the fraction of memory access found in the faster level
  - Hit Time: Time to access the faster level which consists of
    - Memory access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the slower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty



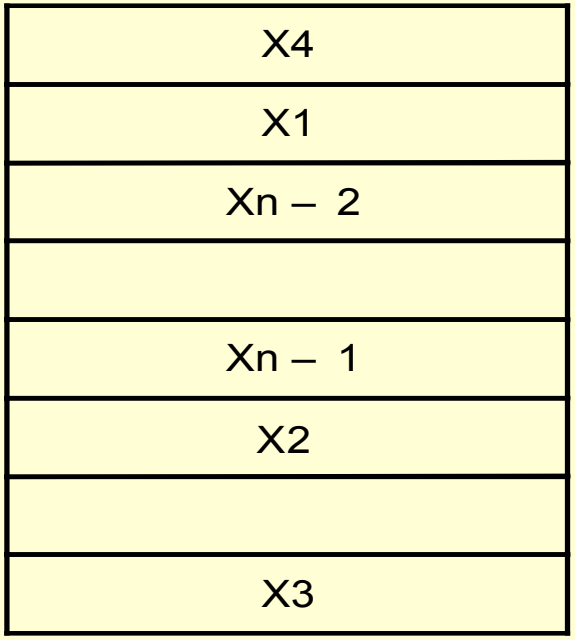
# Memory Hierarchy Design Issues

- Block identification
  - How is a block found if it is in the upper (faster) level?
    - Tag/Block
- Block placement
  - Where can a block be placed in the upper (faster) level?
    - Fully Associative, Set Associative, Direct Mapped
- Block replacement
  - Which block should be replaced on a miss?
    - Random, LRU
- Write strategy
  - What happens on a write?
    - Write Back or Write Through (with Write Buffer)

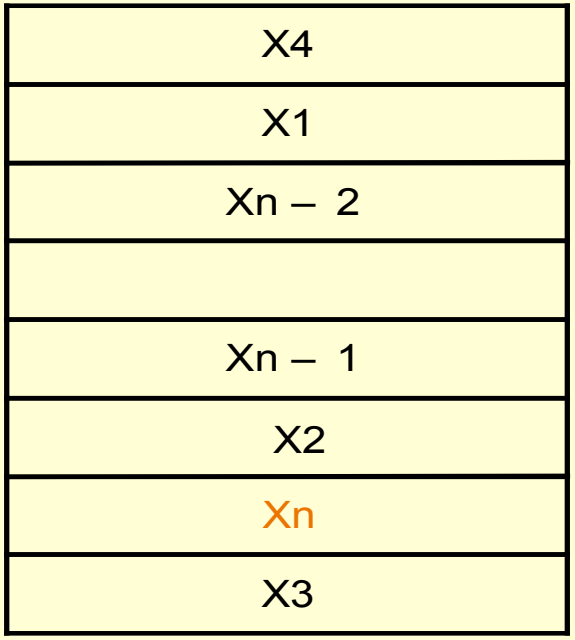
# The Basics of Cache

- Cache: level of hierarchy closest to processor
- Caches first appeared in research machines in early 1960s
- Virtually every general-purpose computer produced today includes cache

Requesting $X_n$ generates a miss and the word $X_n$ will be brought from main memory to cache




a. Before the reference to Xn

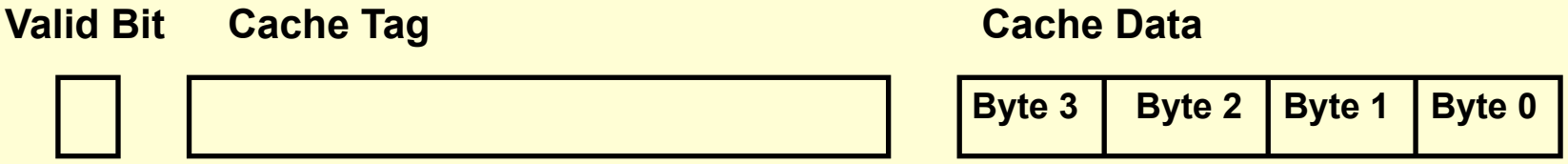



b. After the reference to Xn

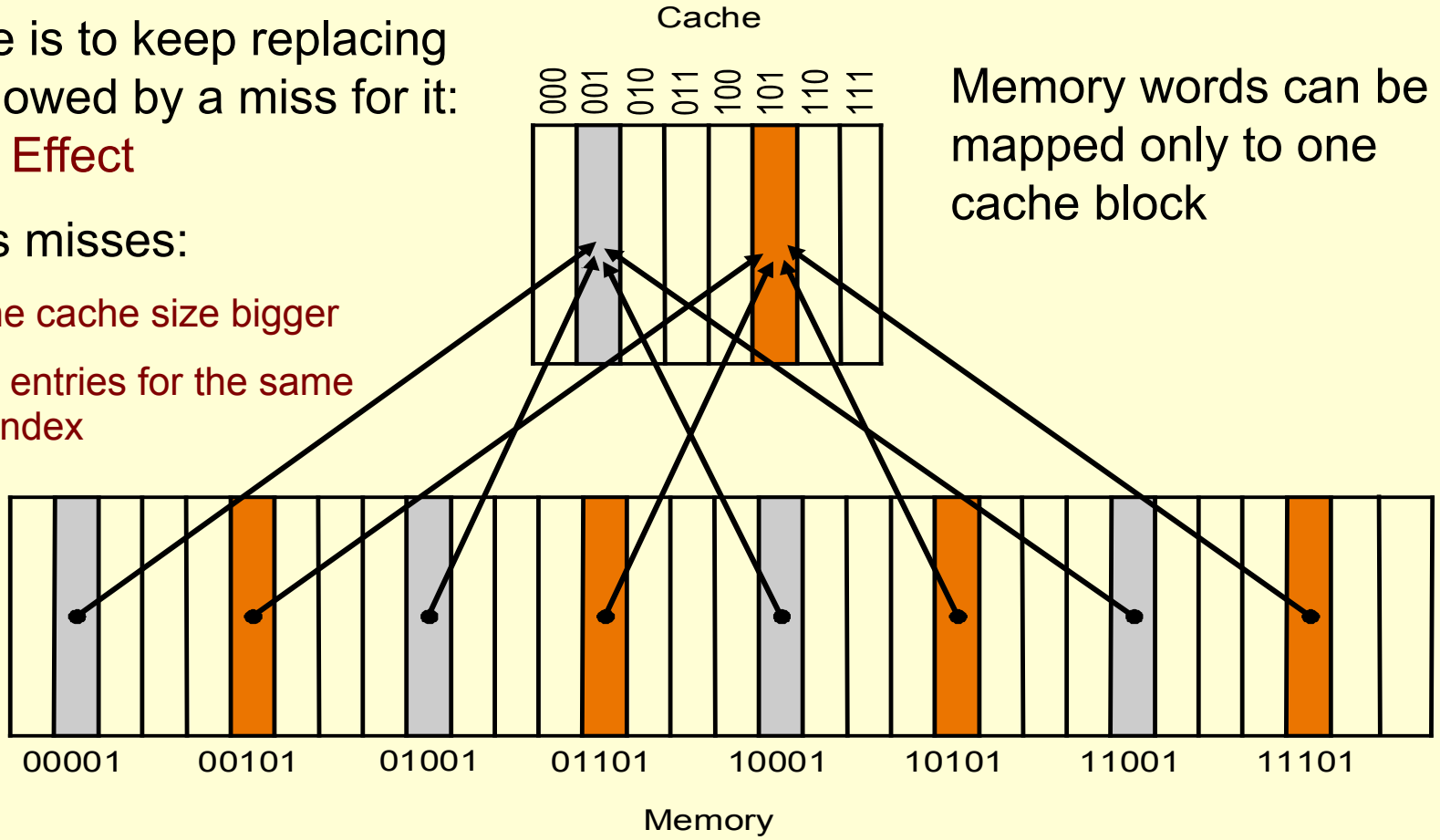
*Issues:*

- How do we know that a data item is in cache?
- If so, How to find it?

# Direct-Mapped Cache




- Worst case is to keep replacing a block followed by a miss for it: Ping Pong Effect
- To reduces misses:
  - make the cache size bigger
  - multiple entries for the same Cache Index




*Cache block address = (Block address) modulo (Number of cache blocks)*

# Accessing Cache

- Cache Size depends on:
  - # cache blocks
  - # address bits
  - Word size
- Example:
  - For n-bit address, 4-byte word & 1024 cache blocks:
  - cache size =

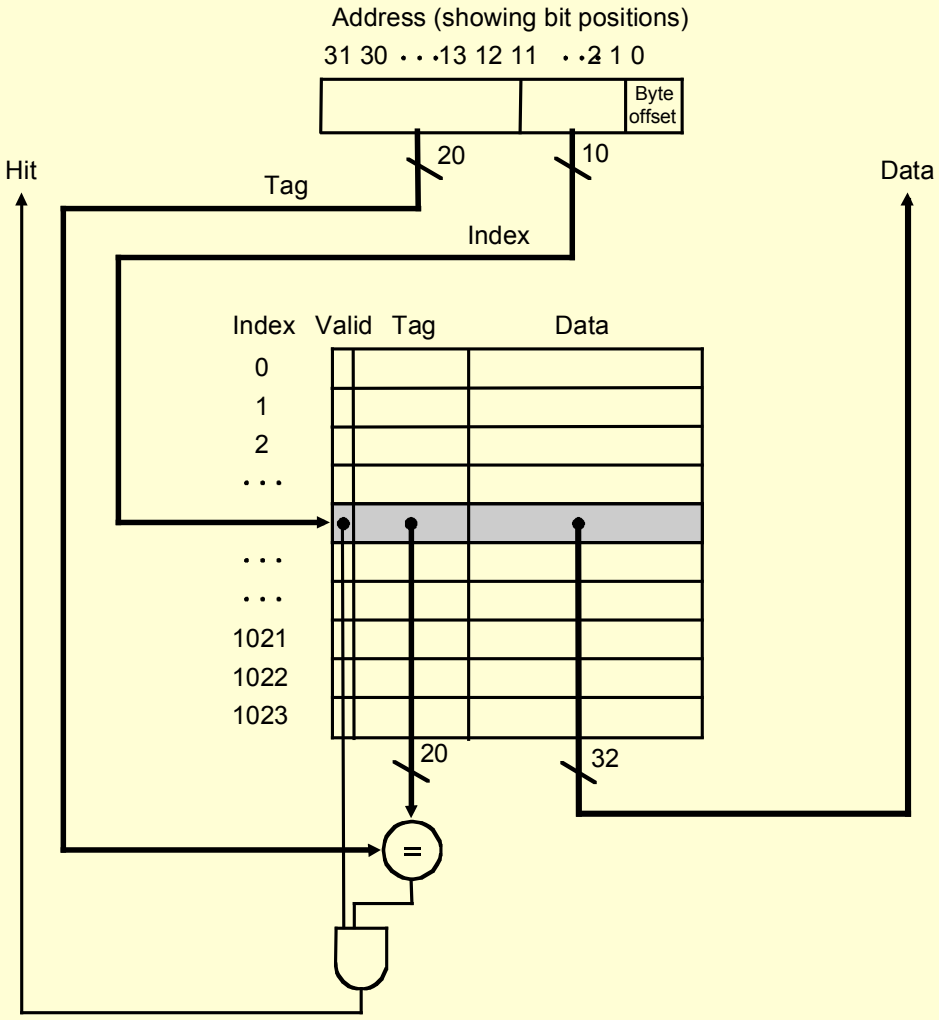
1024 [(n-10 -2) + 1 + 32] bit

**# cache blocks**

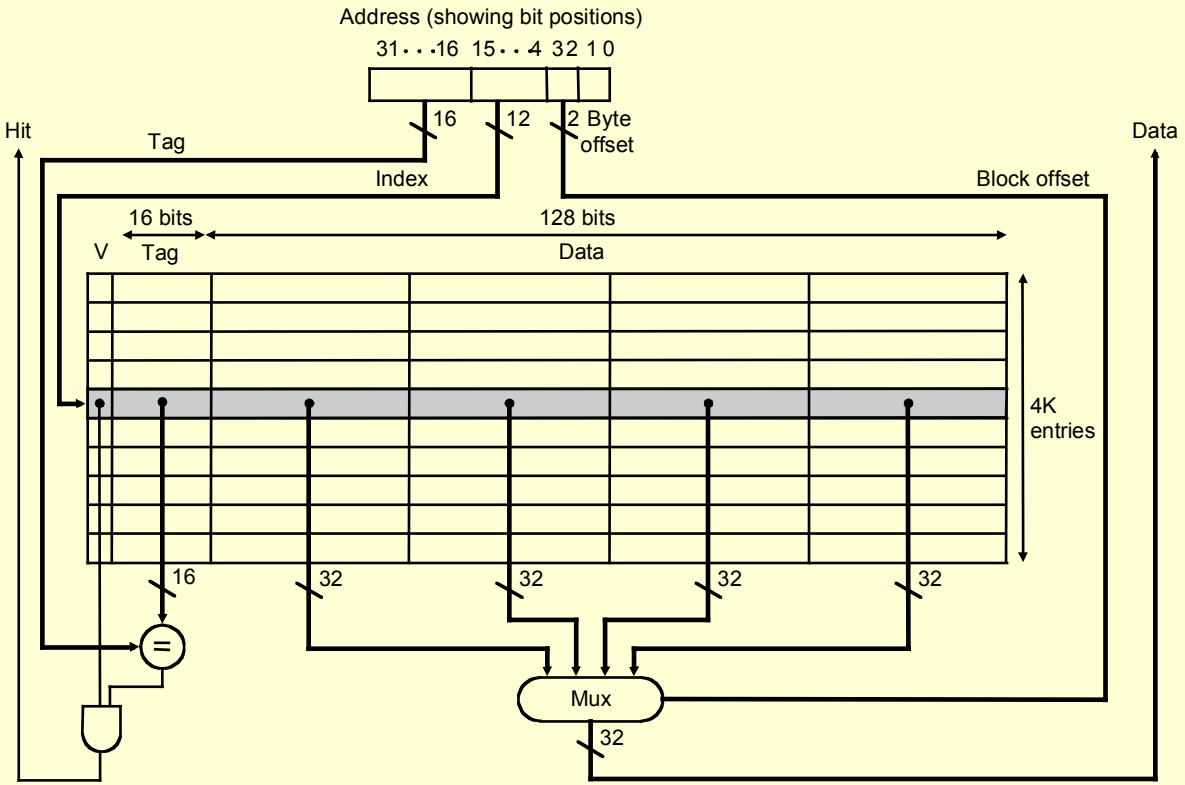
**Tag**

**Valid bit**

**Word size**



# Cache with Multi-Word/Block

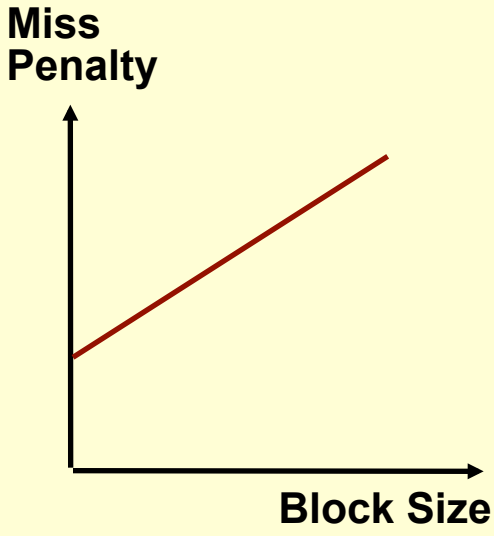


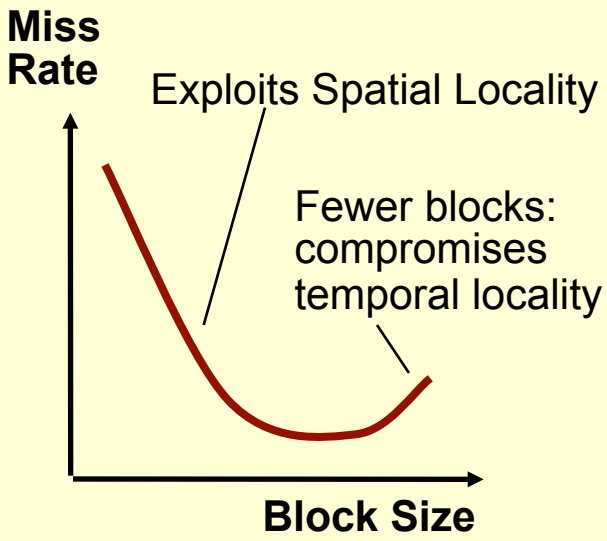

- Takes advantage of spatial locality to improve performance
- Cache block address = (Block address) modulo (Number of cache blocks)
- Block address = (byte address) / (bytes per block)

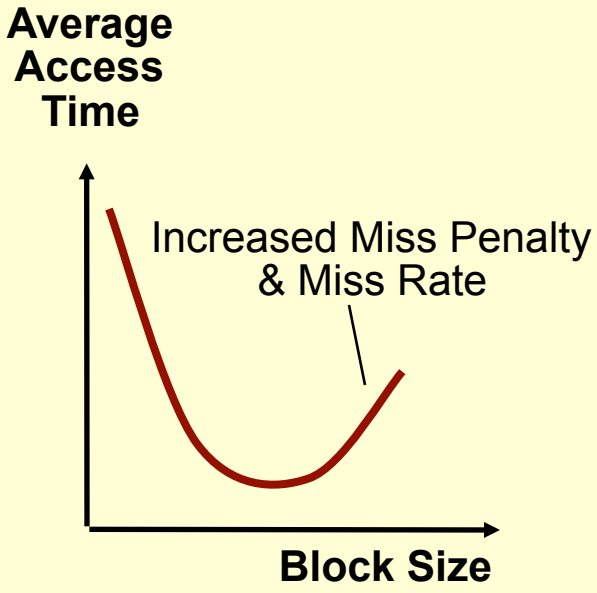
# Determining Block Size

- Larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- Average Access Time =

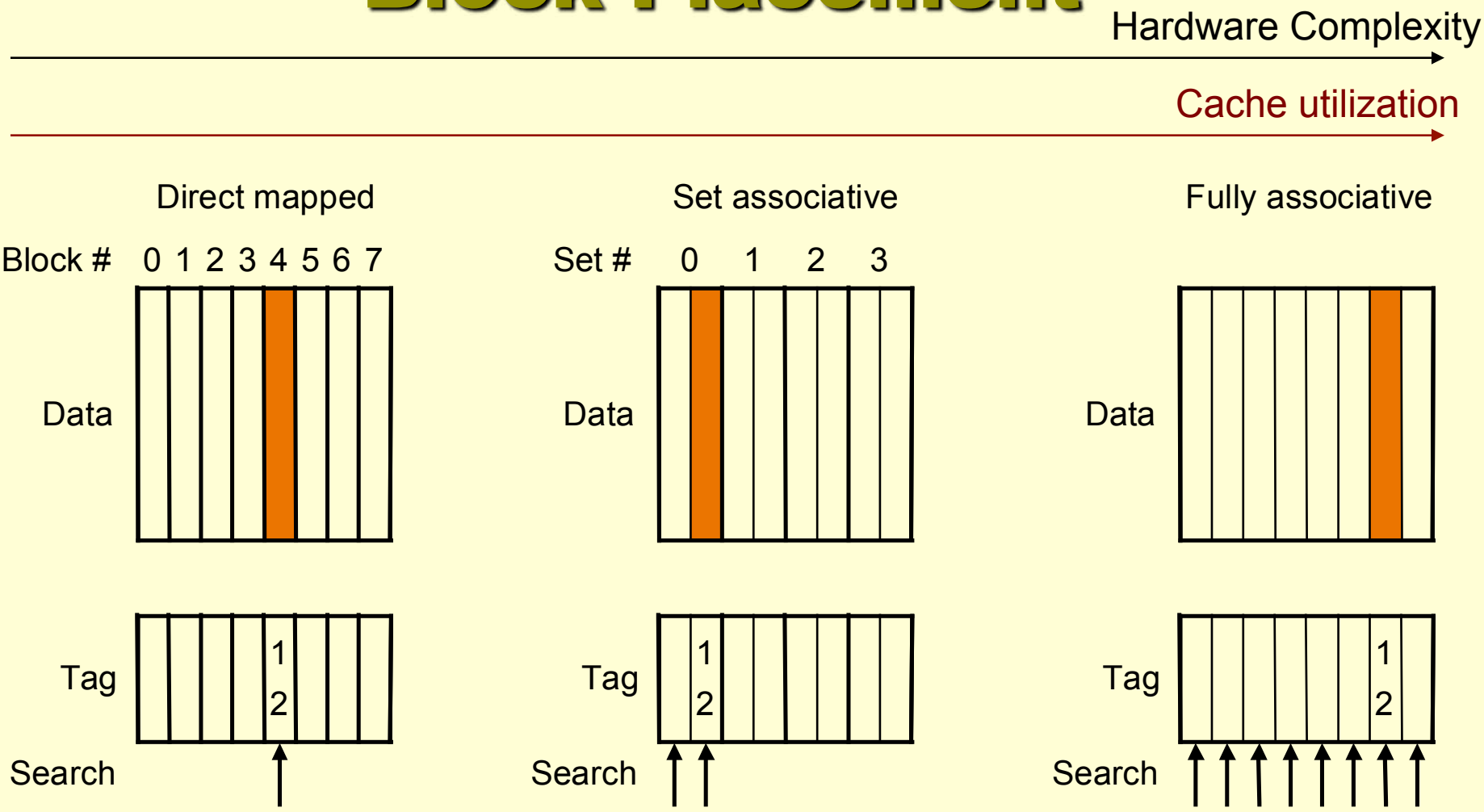
  Hit Time * (1 - Miss Rate) + Miss Penalty * Miss Rate



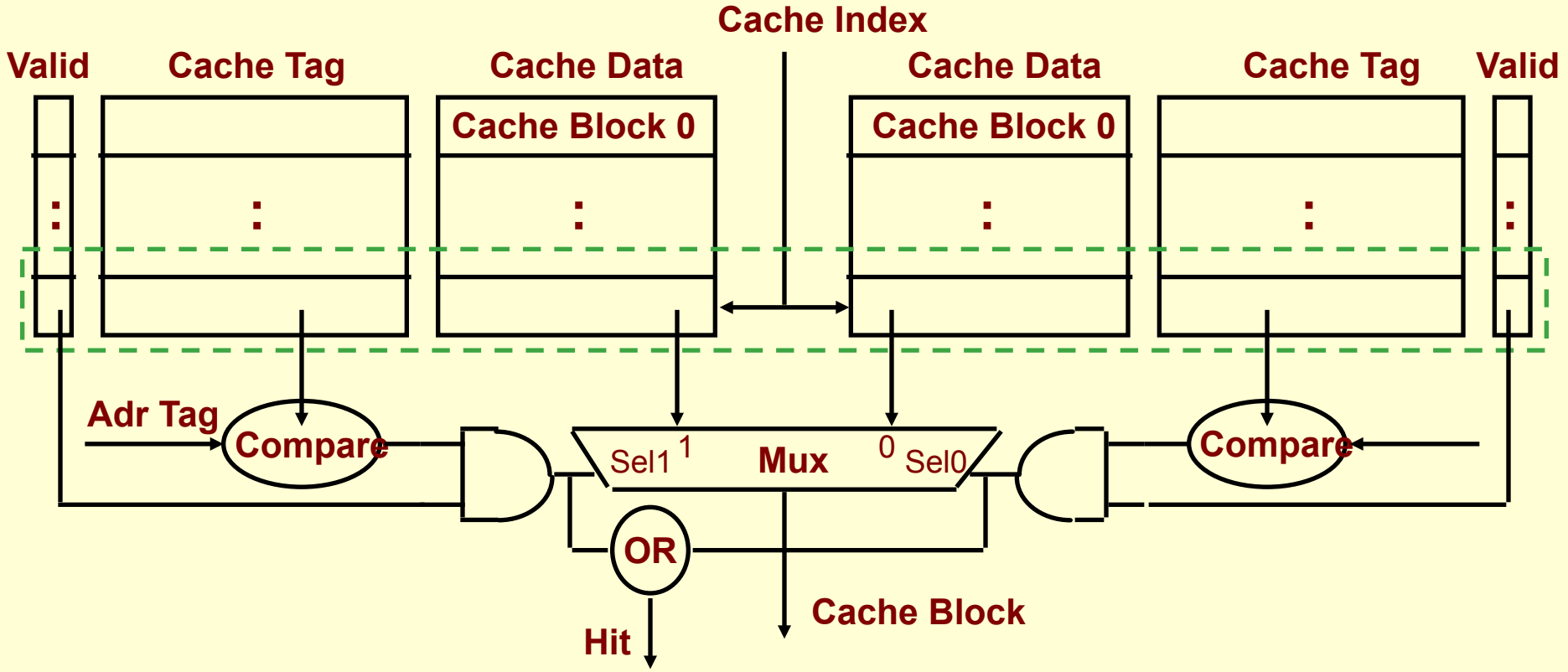
# Block Placement



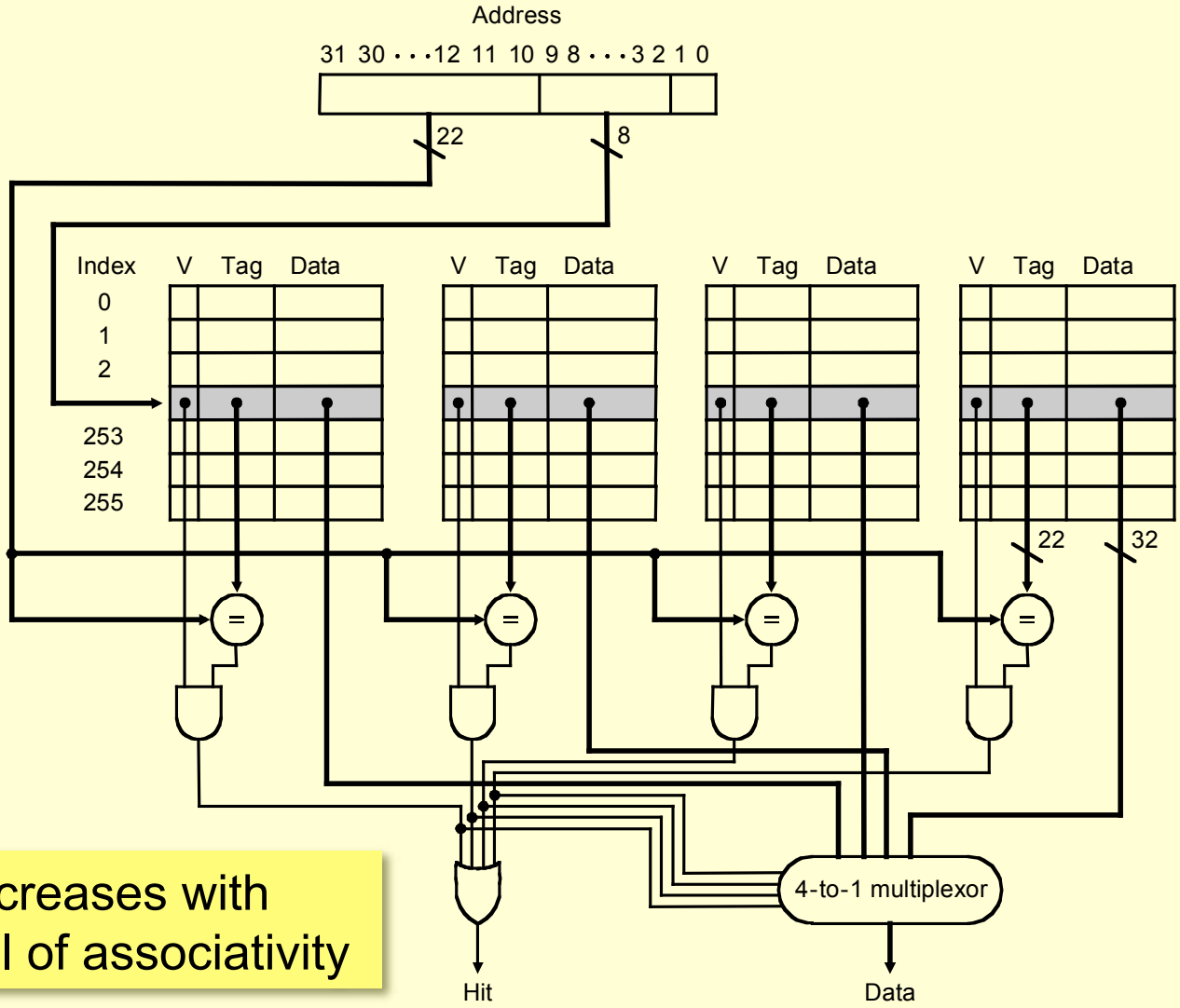

- Set number = (Block number) modulo (Number of sets in the cache)
- Increased flexibility of block placement reduces probability of cache misses

# N-way Set Associative Cache

- N entries for each Cache Index
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



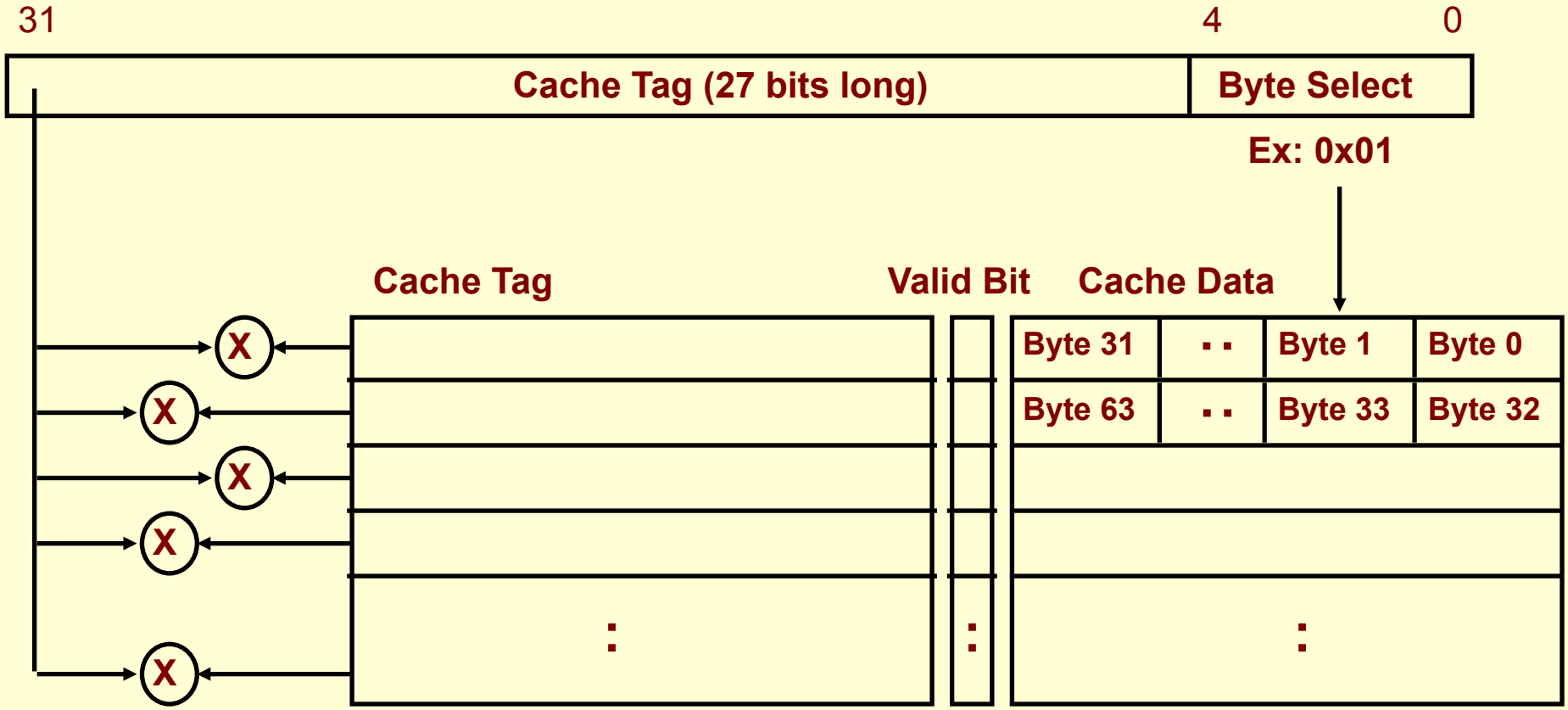
# Locating a Block in Associative Cache




Tag size increases with higher level of associativity

# Fully Associative Cache

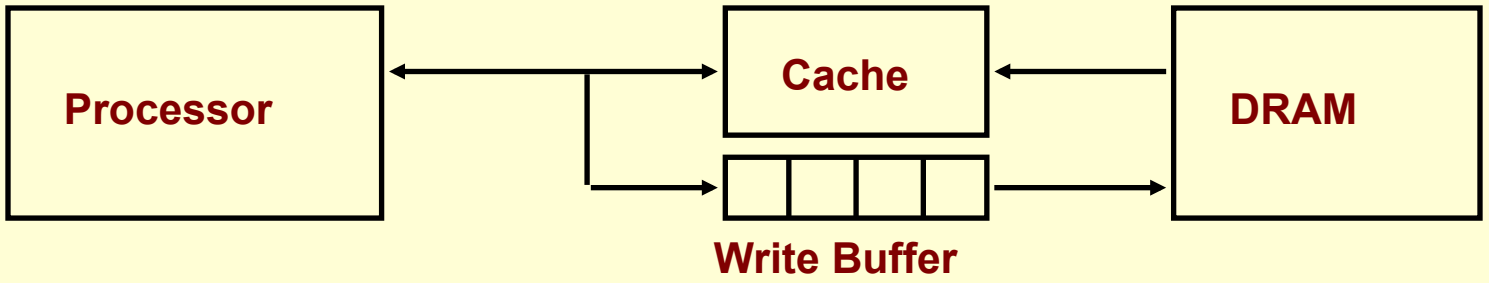
- Forget about the Cache Index
- Compare the Cache Tags of all cache entries in parallel
- Example: Block Size = 32 Byte blocks, we need N 27-bit comparators
- By definition: Conflict Miss = 0 for a fully associative cache



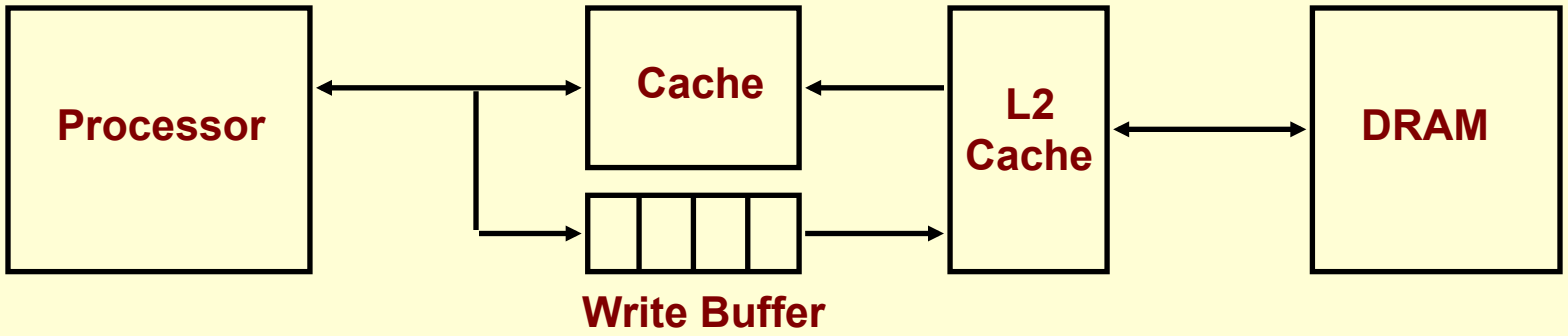
# Handling Cache Misses

- Read misses bring blocks from memory
- Write access requires careful maintenance of consistency between cache and main memory
- Two write strategies:
  - Write through: write to both cache and memory
    - Read misses cannot result in writes
    - No allocation of a cache block is needed
    - Always combined with write buffers so that don't wait for slow memory
  - Write back: write cache only; write to memory when cache block is replaced
    - Is block clean or dirty?
    - No writes to slow memory for repeated write accesses
    - Requires allocation of a cache block

# Write Through via Buffering




- Processor writes data into the cache and the write buffer
- Memory controller writes contents of the buffer to memory
- Increased write frequency can cause saturation of write buffer
- If CPU cycle time too fast and/or too many store instructions in a row:
  - Store buffer will overflow no matter how big you make it
  - The CPU Cycle Time get closer to DRAM Write Cycle Time
- Write buffer saturation can be handled by installing a second level (L2) cache



# Block Replacement Strategy

- Straight forward for Direct Mapped since every block has only one location
- Set Associative or Fully Associative:
  - Random: pick any block
  - LRU (Least Recently Used)
    - requires tracking block reference
    - for two-way set associative cache, reference bit attached to every block
    - more complex hardware is needed for higher level of cache associativity

| Associativity / Size | 2-way | | 4-way | | 8-way | |
|---|---|---|---|---|---|---|
| | LRU | Random | LRU | Random | LRU | Random |
| 16 KB | 5.2% | 5.7% | 4.7% | 5.3% | 4.4% | 5.0% |
| 64 KB | 1.9% | 2.0% | 1.5% | 1.7% | 1.4% | 1.5% |
| 256 KB | 1.15% | 1.17% | 1.13% | 1.13% | 1.12% | 1.12% |

- Empirical results indicates less significance of replacement strategy with increased cache sizes

# Measuring Cache Performance

- To enhance cache performance, one can:
  - Reduce the miss rate (e.g. diminishing blocks collisions)
  - Reduce the miss penalty (e.g. adding multi-level caching)
  - Enhance hit access time (e.g. simple and small cache)

$$\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{Cycle time}$$

$$\text{Memory stall cycles} = \text{Read stall cycles} + \text{Write stall cycles}$$

$$\text{Read stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

For write-through scheme:

Hard to control, assume enough buffer size

$$\text{Write stall cycles} = \left( \frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty} \right) + \text{Write buffer stalls}$$

# Example

Assume an instruction cache miss rate for gcc of 2% and a data cache miss rate of 4%. If a machine has a CPI of 2 without any memory stalls and the miss penalty is 40 cycles for all misses, determine how much faster a machine would run with a perfect cache that never missed. Assume 36% combined frequencies for load and store instructions

**Answer:**

Assume number of instructions = I

Instruction miss cycles = I × 2% × 40 = 0.8 × I

Data miss cycles = I × 36% × 4% × 40 = 0.56 × I

Total number of memory-stall cycles = 0.8 I + 0.56 I = 1.36 I

The CPI with memory stalls = 2 + 1.36 = 3.36

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{\text{I} \times CPI_{stall} \times \text{Clock cycle}}{\text{I} \times CPI_{perfect} \times \text{Clock cycle}} = \frac{CPI_{stall}}{CPI_{perfect}} = \frac{3.36}{2}$$

What happens if the CPU gets faster?

# Classifying Cache Misses

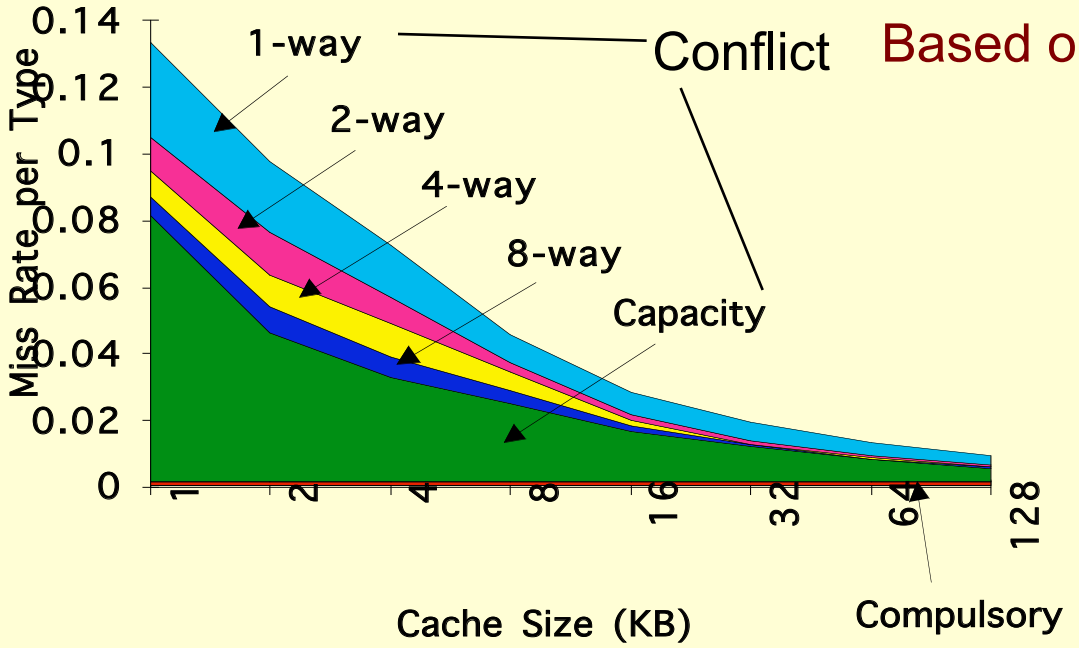
- Compulsory
  - First access to a block not in cache
  - Also called cold start or first reference misses
  - (Misses in even an Infinite Cache)
- Capacity
  - If the cache cannot contain all needed blocks
  - Due to blocks discarded and re-retrieved
  - (Misses in Fully Associative Cache)
- Conflict
  - 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 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

# Miss Rate Distribution

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

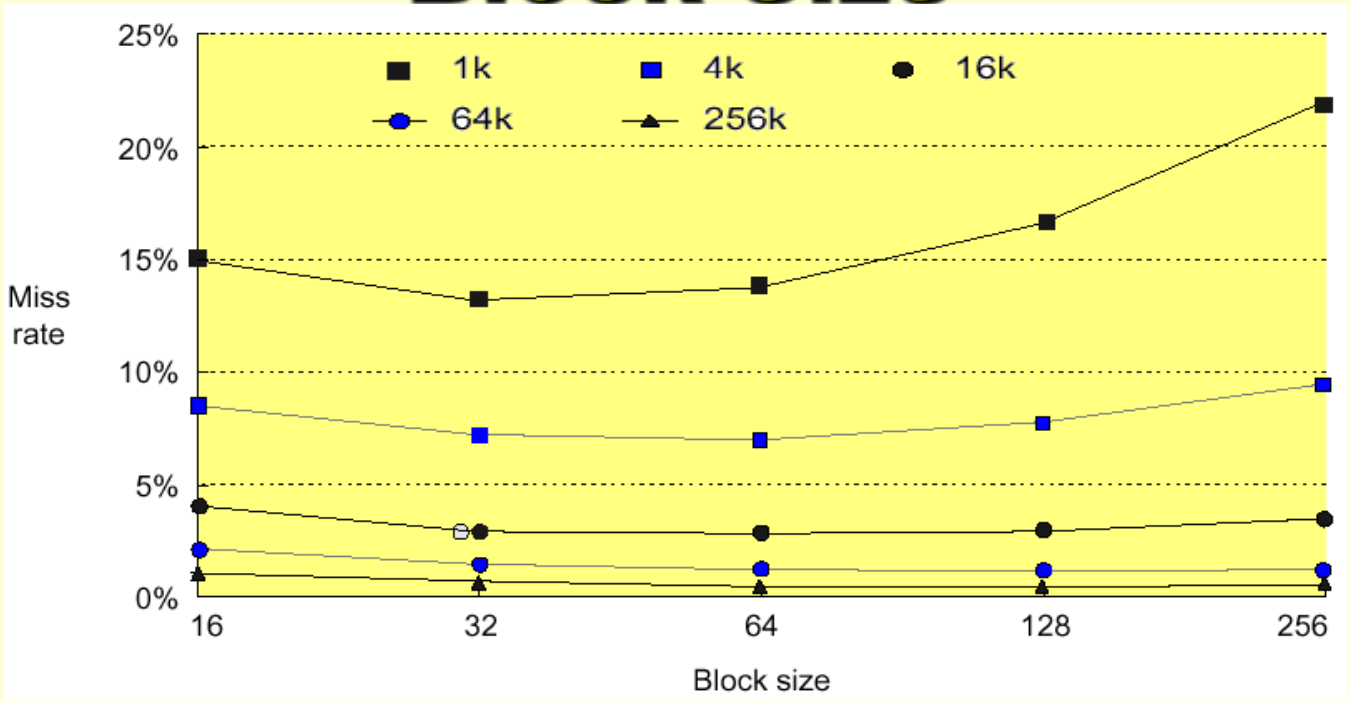


# Techniques for Reducing Misses

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

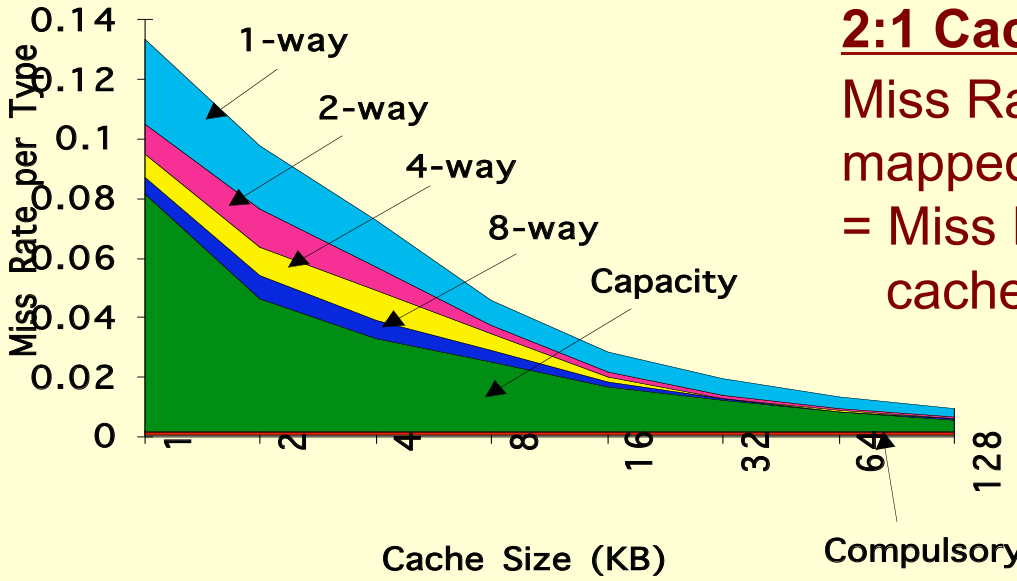
1. Reducing Misses via Larger Block Size
2. Reducing Misses via Higher Associativity
3. Reducing Misses via Victim Cache
4. Reducing Misses via Pseudo-Associativity
5. Reducing Misses by H/W Prefetching Instr. and Data
6. Reducing Misses by S/W Prefetching Data
7. Reducing Misses by Compiler Optimizations

# 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




**2:1 Cache Rule:**

Miss Rate for direct mapped cache of size N = Miss Rate 2-way cache size N/2

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

# Example

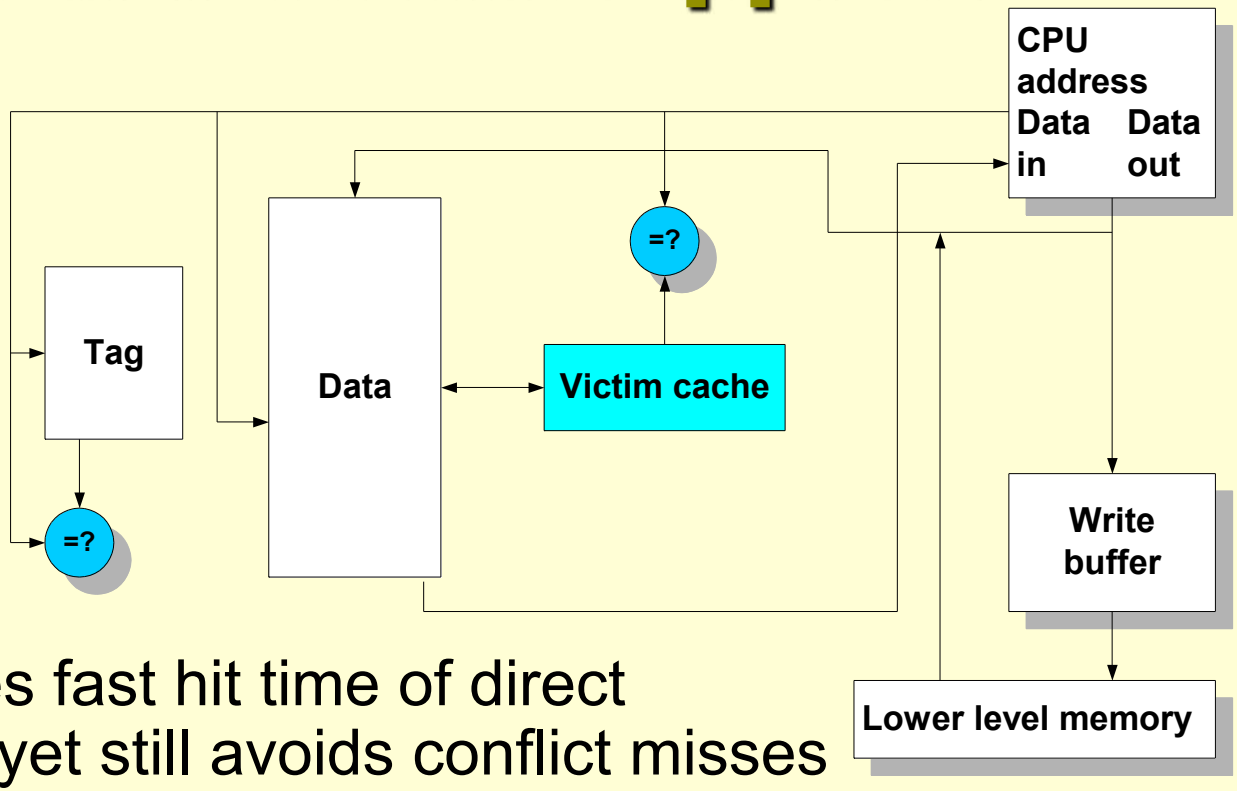
*Assume hit time is 1 clock cycle and average miss penalty is 50 clock cycles for a direct mapped cache. The clock cycle increases by a factor of 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way associative cache. Compare the average memory access based on the previous figure miss rates*

| 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 asssociative 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 handing 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

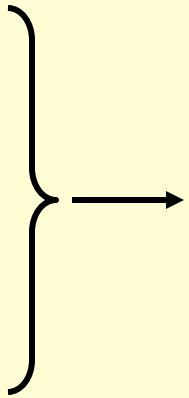
$$\text{Average memory access time} = \text{Hit time} + \text{Miss Rate} \times (\text{Prefetch hit rate} + (1 - \text{Prefetch hit rate}) \times \text{Miss penalty})$$

- 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 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)
- 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];
```

# 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

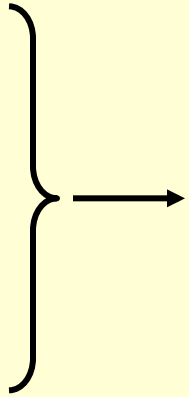
For Data

- 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];
```

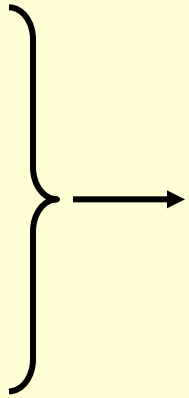


```
/* After: 1 array of stuctures */
struct merge {
        int val;
        int key;
};
struct merge merged_array[SIZE];
```

- 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];
```



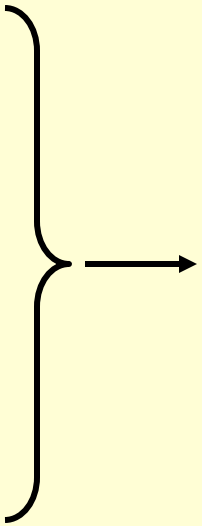
```
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];
```

- 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];
```



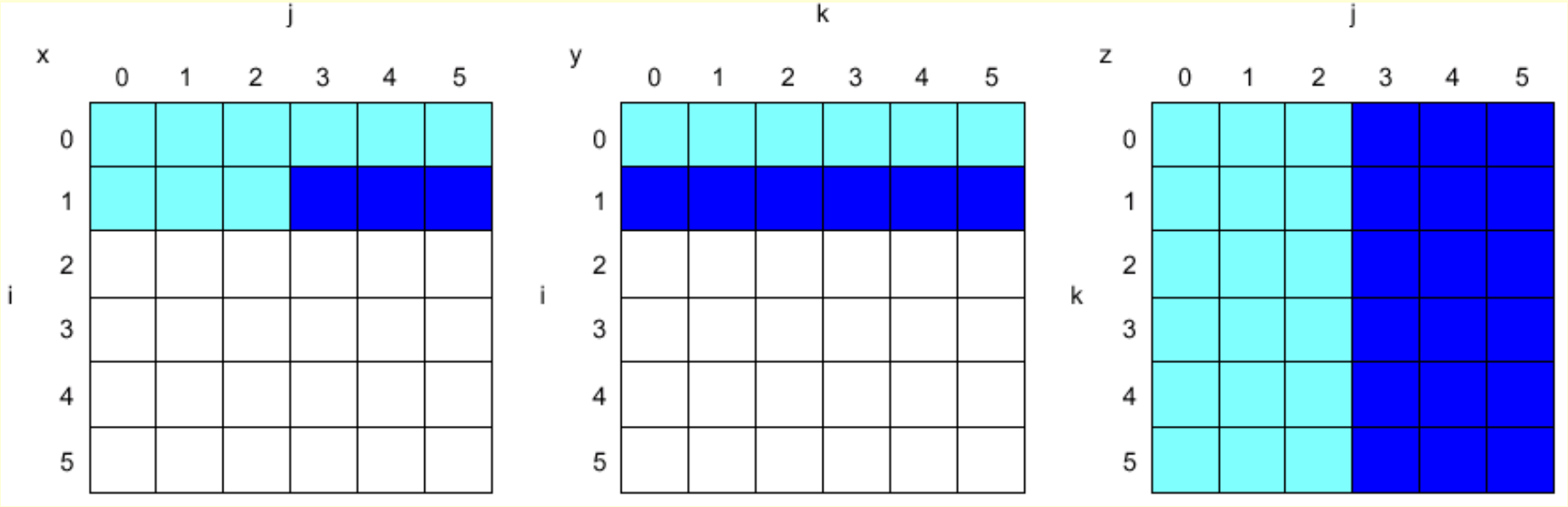
```
/* After */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1) {
          a[i][j] = 1/b[i][j] * c[i][j];
          d[i][j] = a[i][j] + c[i][j];
}
```

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

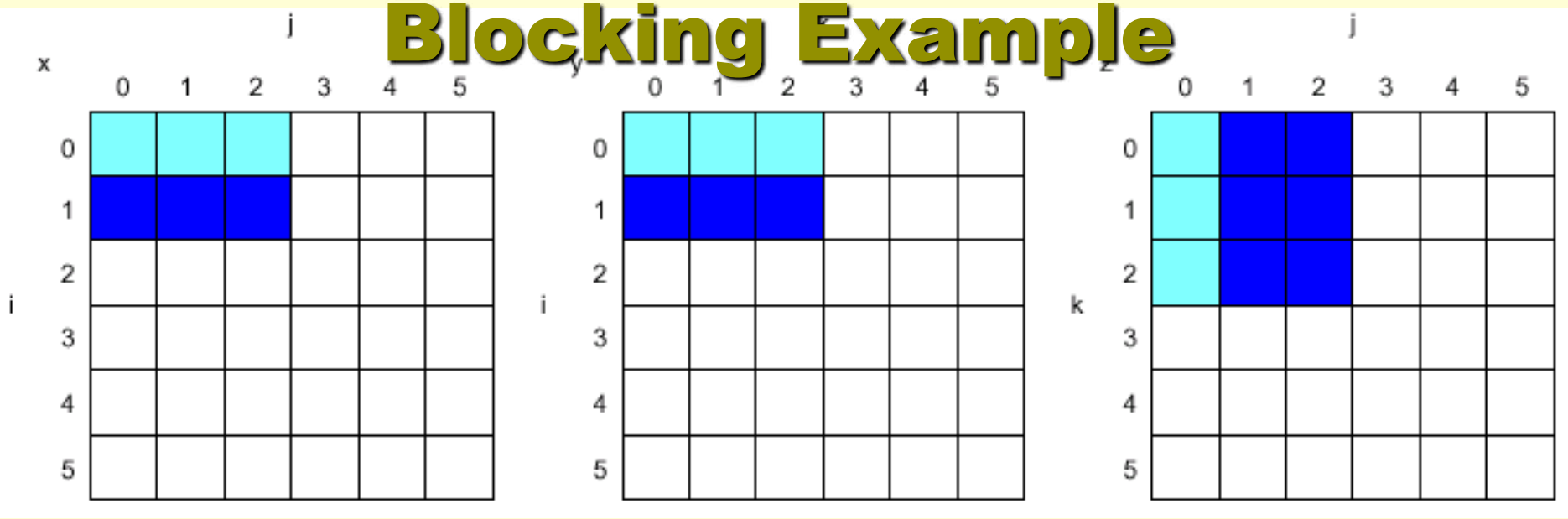
# Blocking Example

```
/* 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

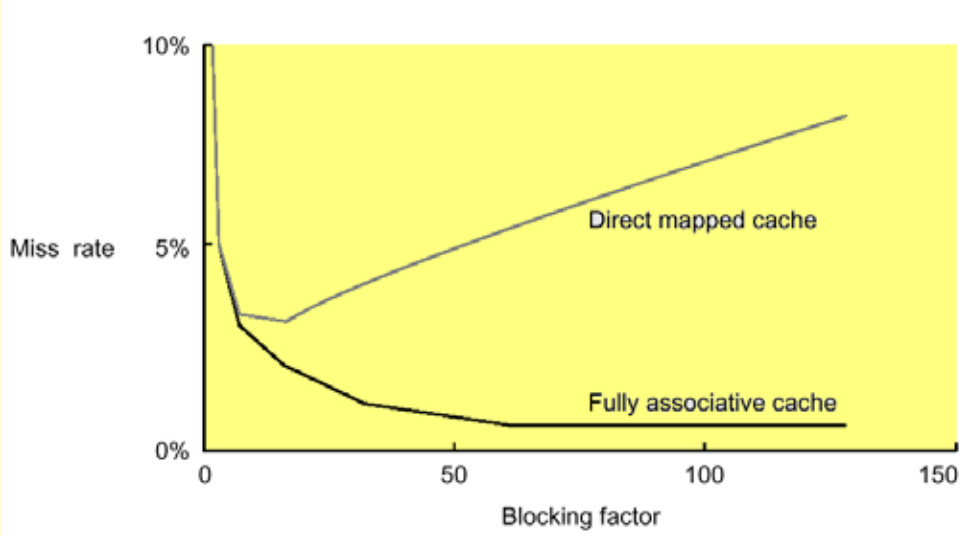



```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1) {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1) {
            r = r + y[i][k] * z[k][j];};
        x[i][j] = x[i][j] + r;
       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

- 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

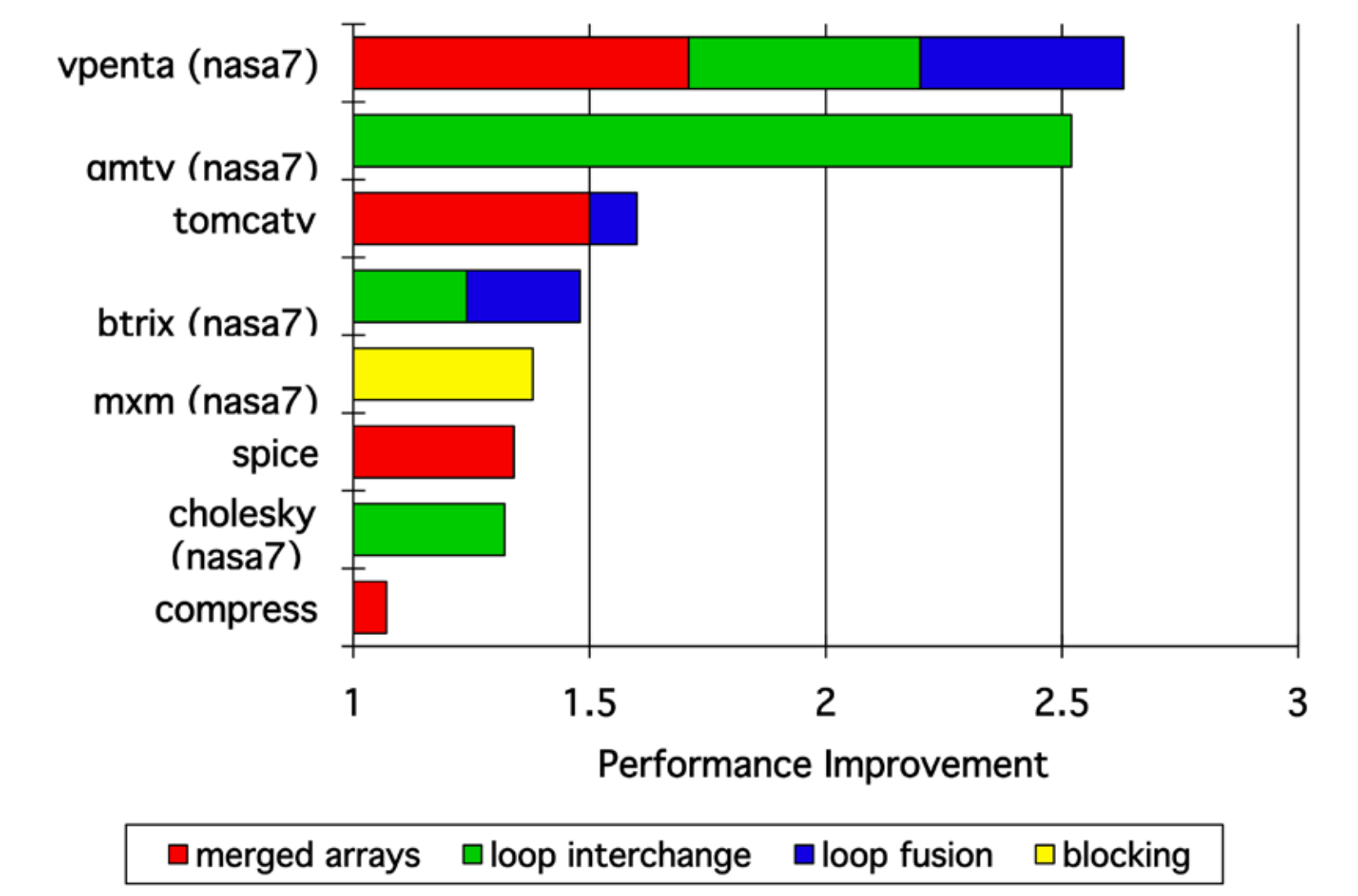
# Blocking Factor

- 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.



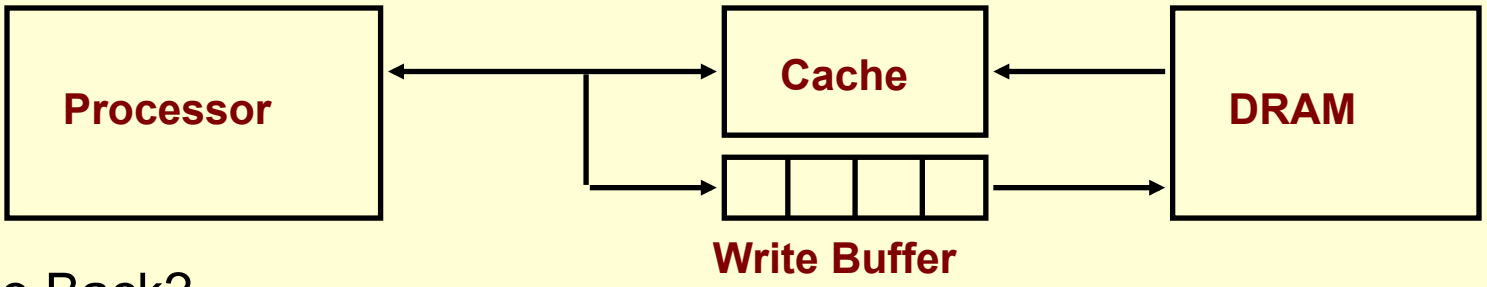
# Reducing Miss Penalty

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

- 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

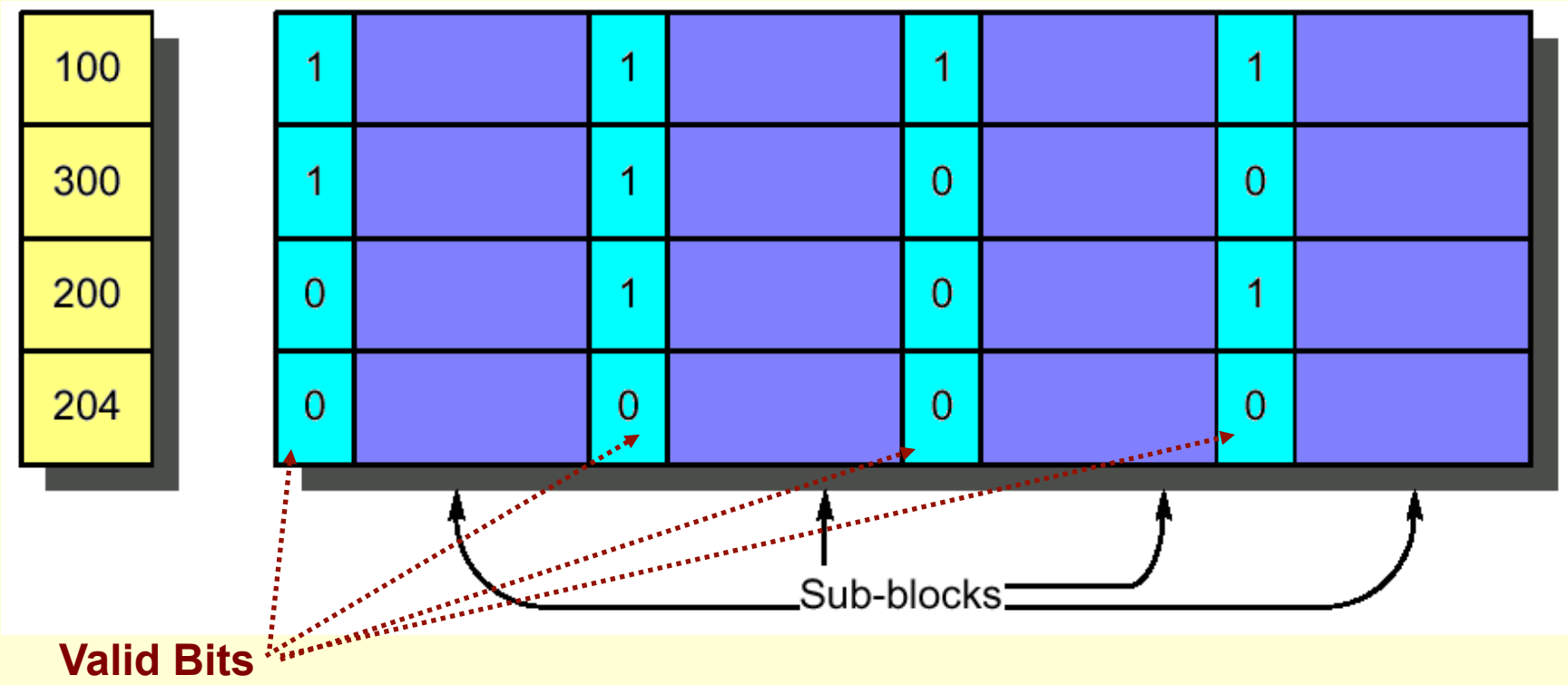



❑Write Back?

- ➡ 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

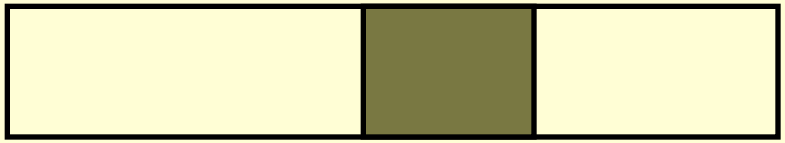
# Sub-block Placement

- 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)



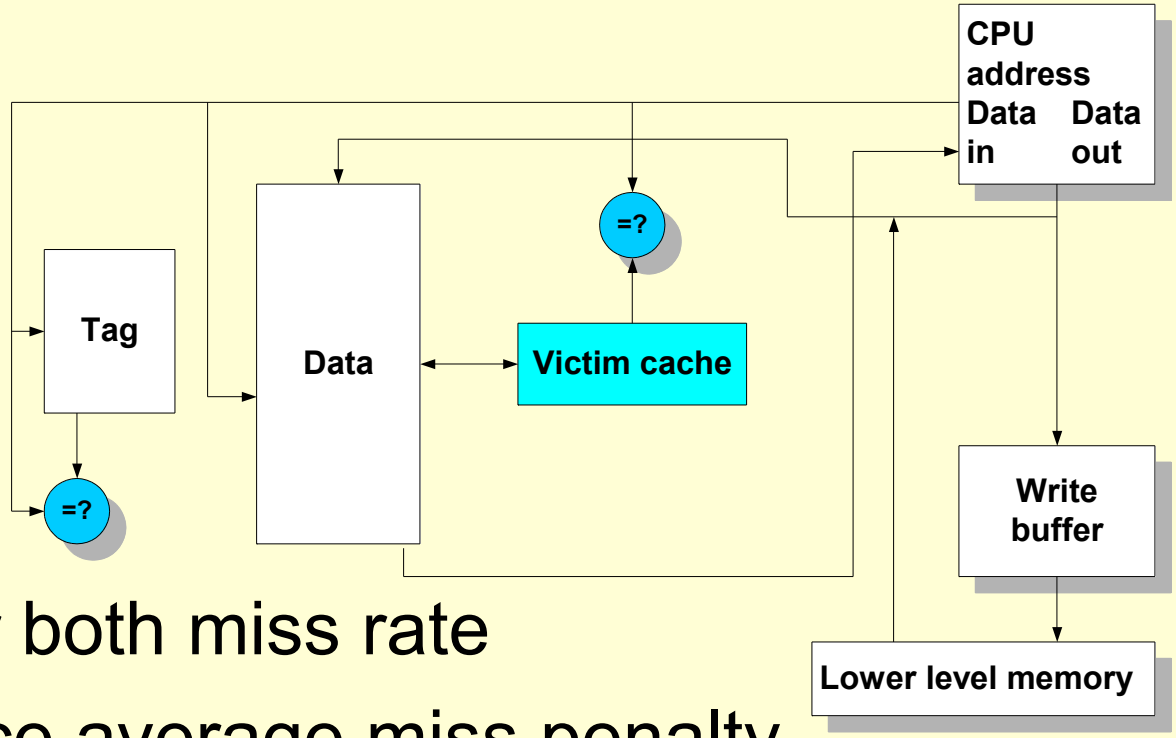
# Early Restart and Critical Word First

- 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

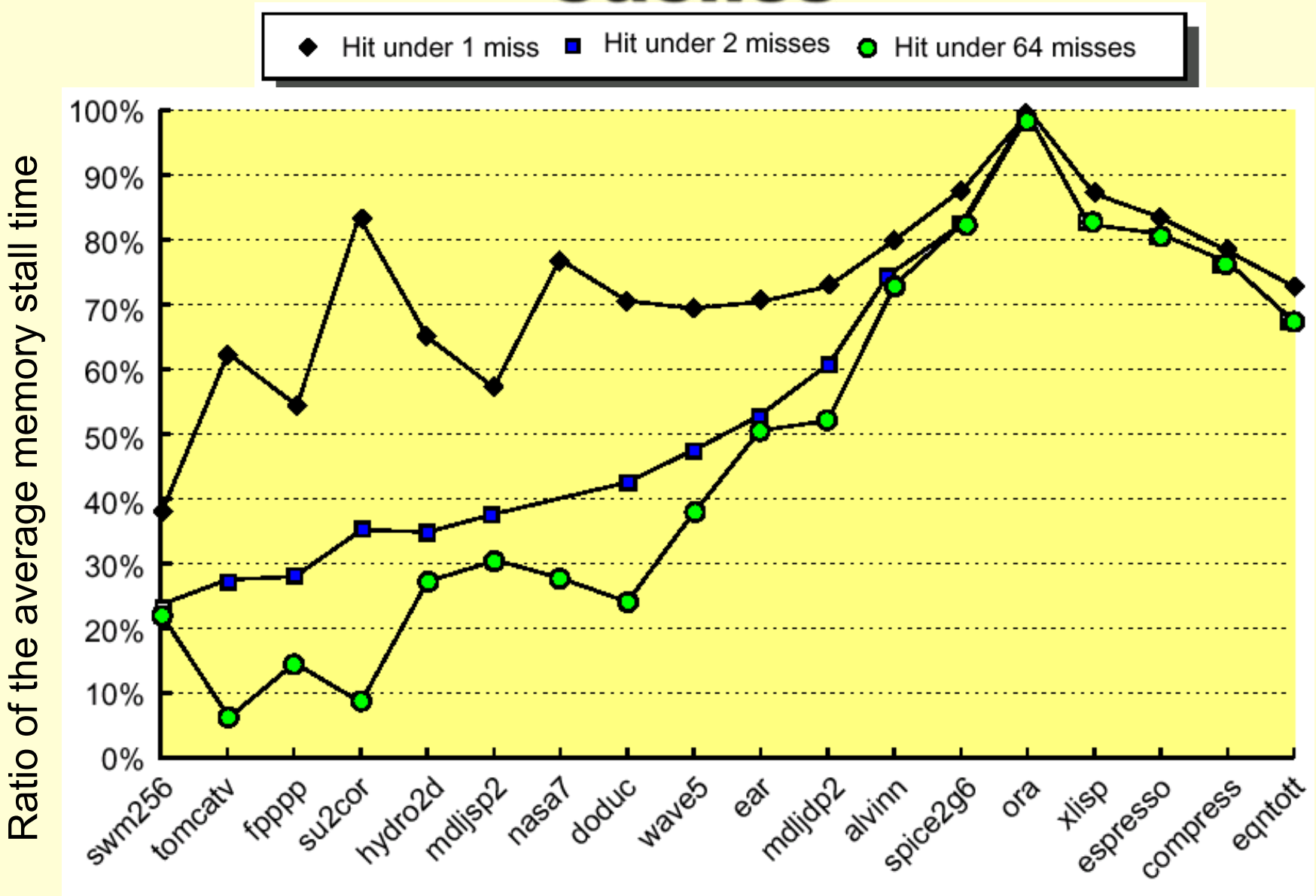



- Lower both miss rate
- Reduce average miss penalty
- Slightly extend the worst case miss penalty

# 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



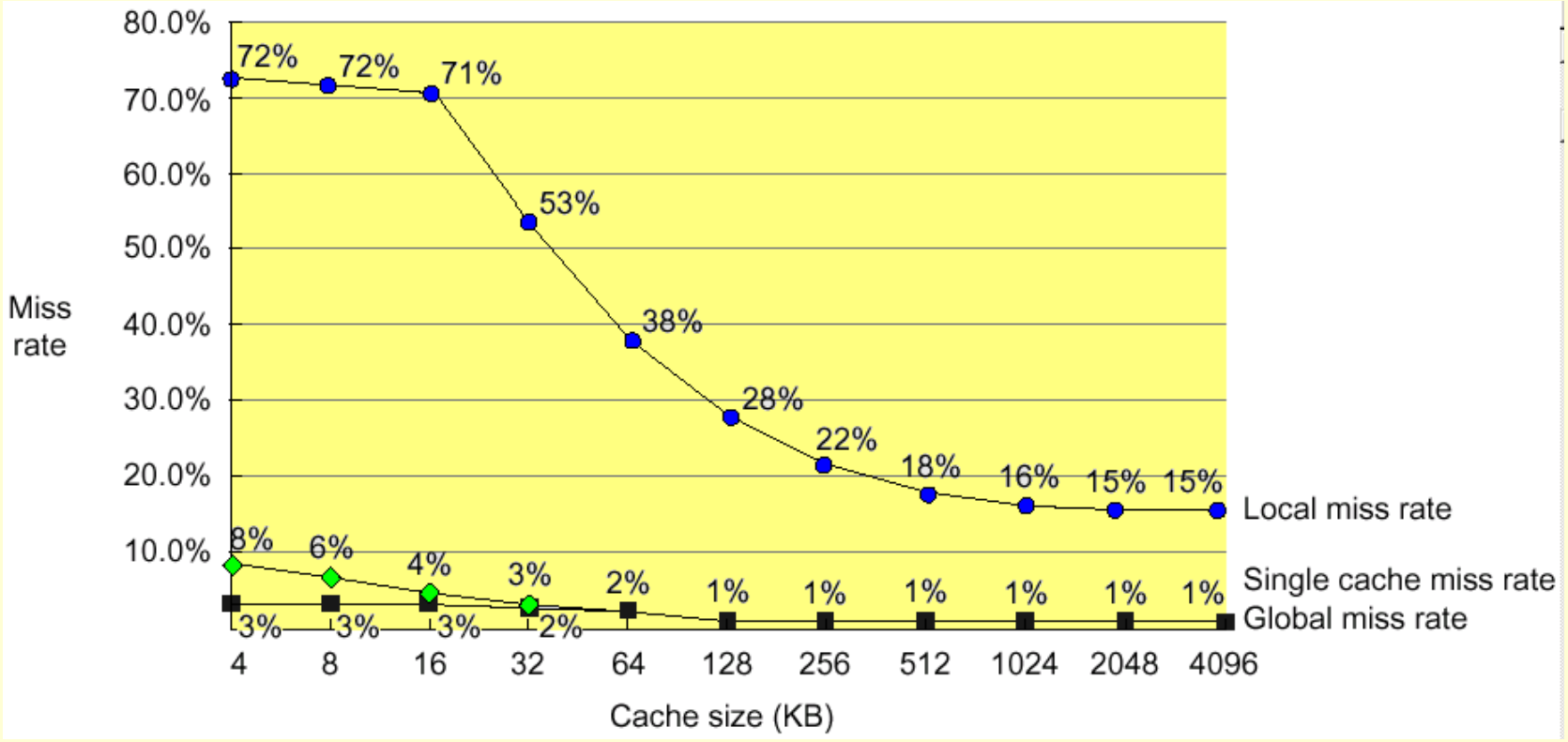
# Second Level Cache

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

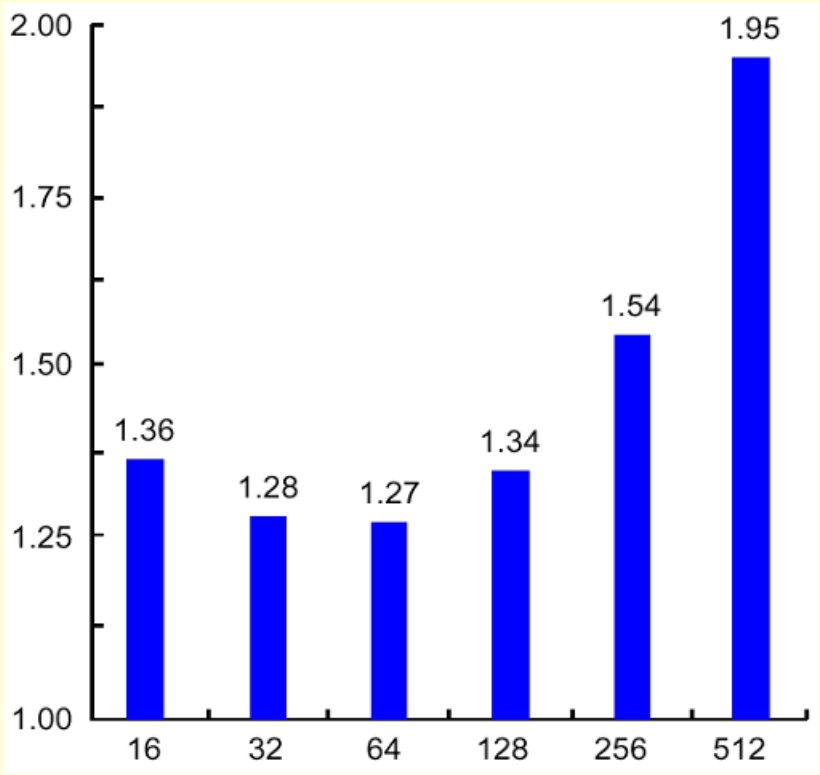
# Local vs Global Misses




(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