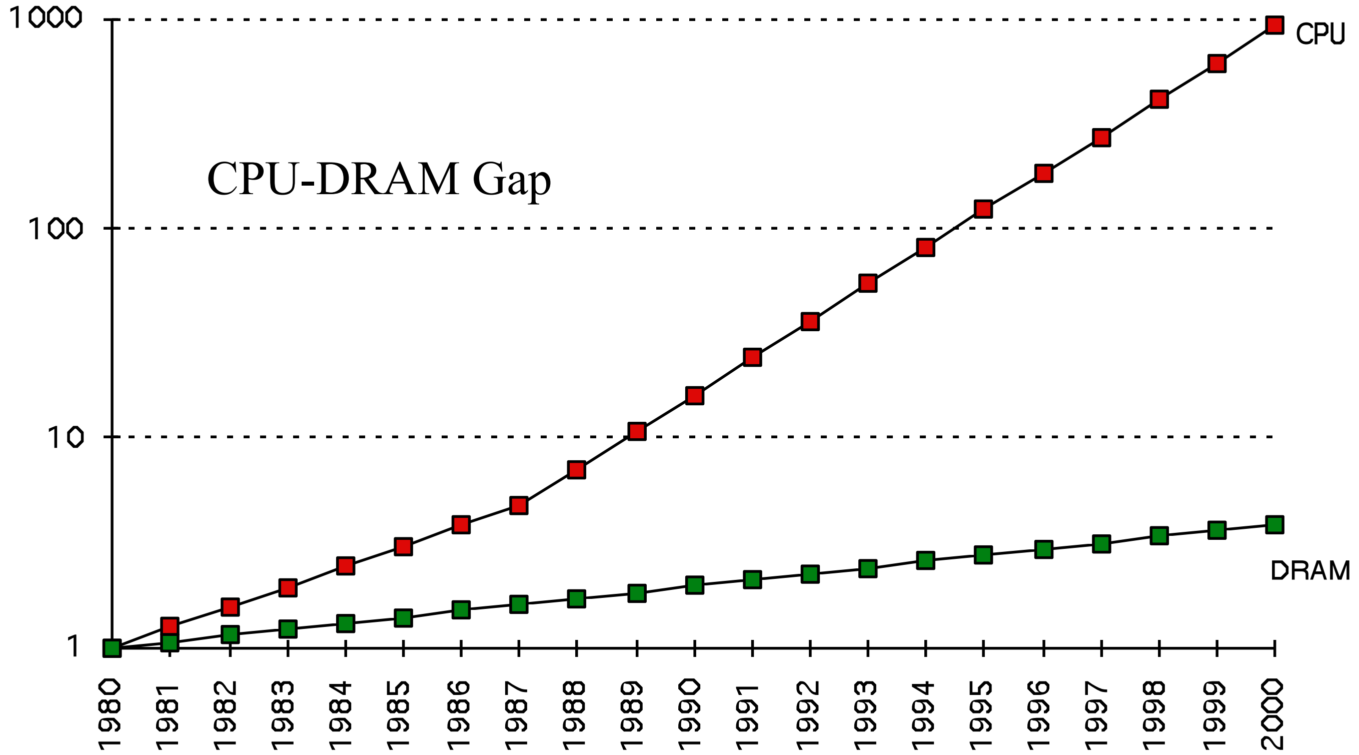
Cache Design

Who Cares about Memory Hierarchy?

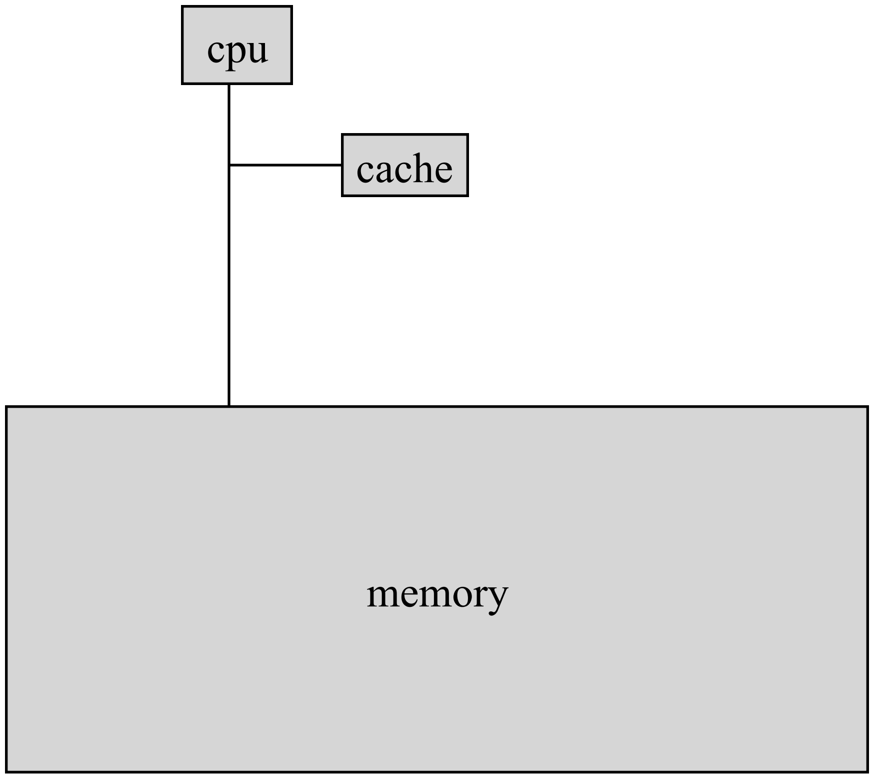
- Processor vs Memory Performance



1980: no cache in microprocessor;

1995 2-level cache

Memory Cache



Memory Locality

- Memory hierarchies take advantage of *memory locality*.
- *Memory locality* is the principle that future memory accesses are *near* past accesses.
- Memories take advantage of two types of locality
  - ***Temporal locality*** -- near in time
    - we will often access the same data again very soon
  - ***Spatial locality*** -- near in space/distance
    - our next access is often very close to our last access (or recent accesses).

1,2,3,4,5,6,7,8,8,47,8,9,8,10,8,8...

Locality and Caching

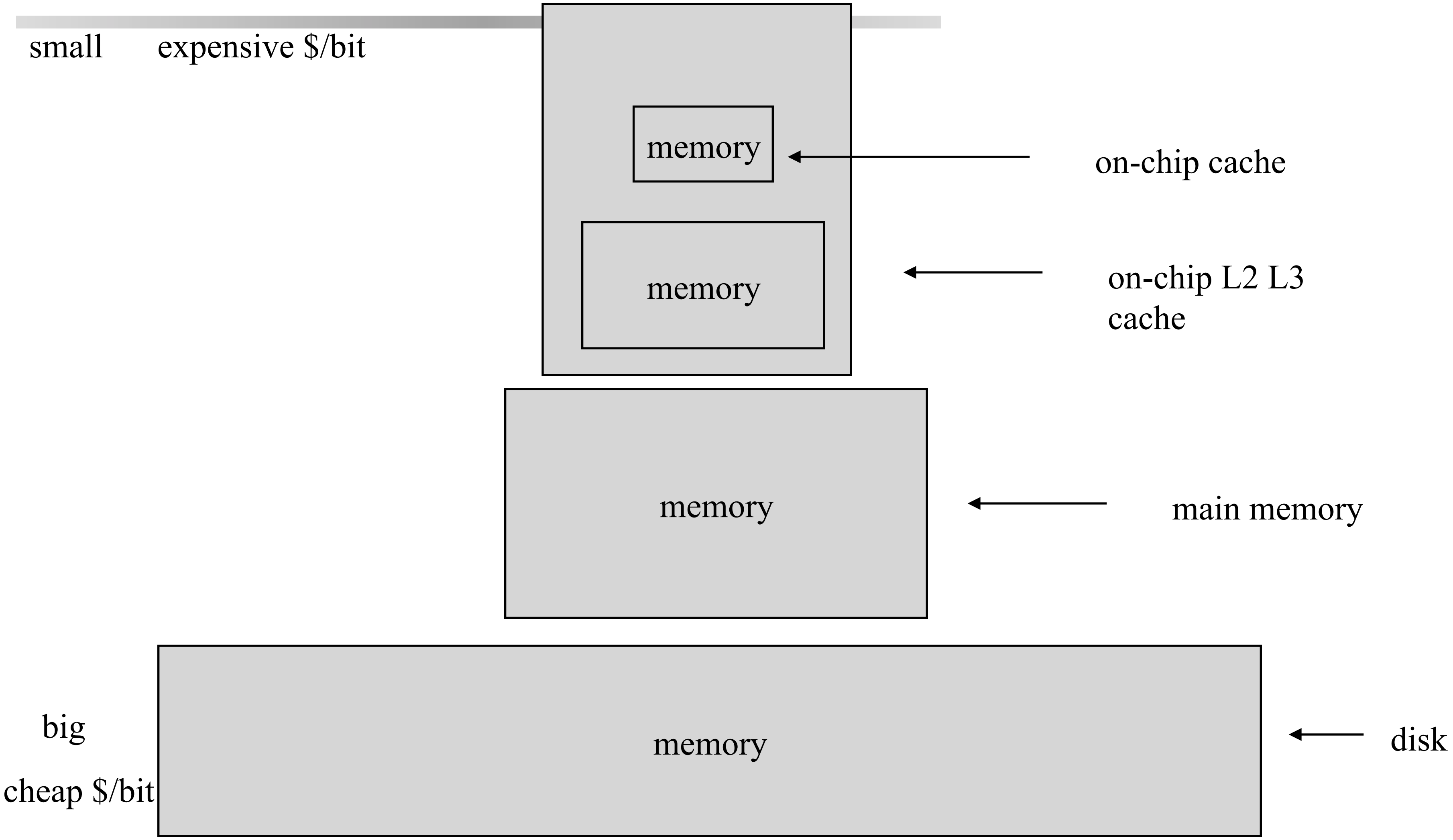
- Memory hierarchies exploit locality by *caching* (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can’t build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk capacity

SRAM (static RAM) – 1-5 ns access time

DRAM (dynamic RAM) – 40-60 ns

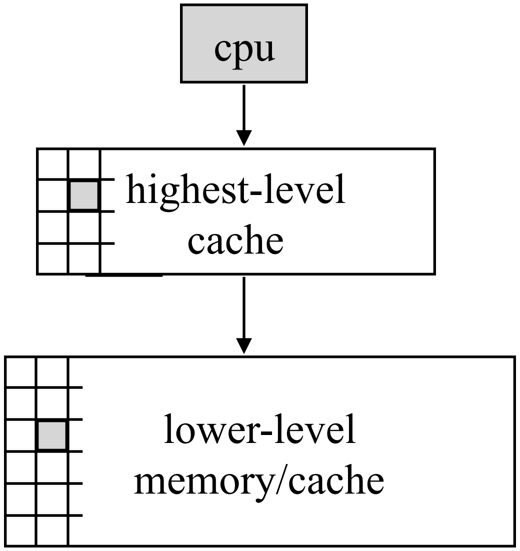
disk -- access time measured in milliseconds, very cheap

A typical memory hierarchy



- *so then where is my program and data??*

Cache Fundamentals

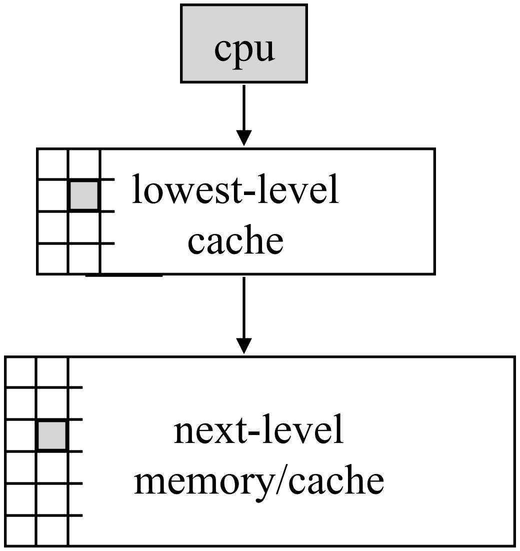


- *cache hit* -- an access where the data is found in the cache.
- *cache miss* -- an access which isn’t
- *hit time* -- time to access the higher cache
- *miss penalty* -- time to move data from lower level to upper, then to cpu
- *hit ratio* -- percentage of time the data is found in the higher cache
- *miss ratio* -- (1 - hit ratio)

Cache Fundamentals, cont.

- *cache block size* or *cache line size*-- the amount of data that gets transferred on a cache miss.
- *instruction cache* -- cache that only holds instructions.
- *data cache* -- cache that only caches data.
- *unified cache* -- cache that holds both.

  (L1 is unified → "princeton architecture")



Cache Characteristics

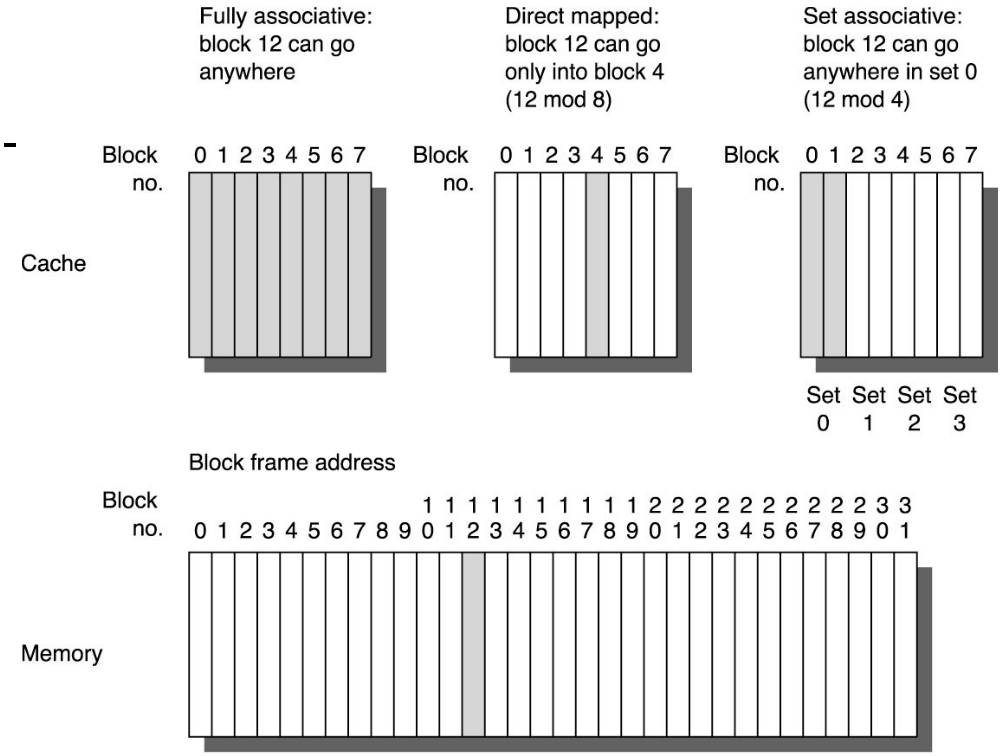
- Cache Organization
- Cache Access
- Cache Replacement
- Write Policy

Cache Organization: Where can a block be placed in the cache?

- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

(associativity = degree of freedom in placing a particular block of memory)

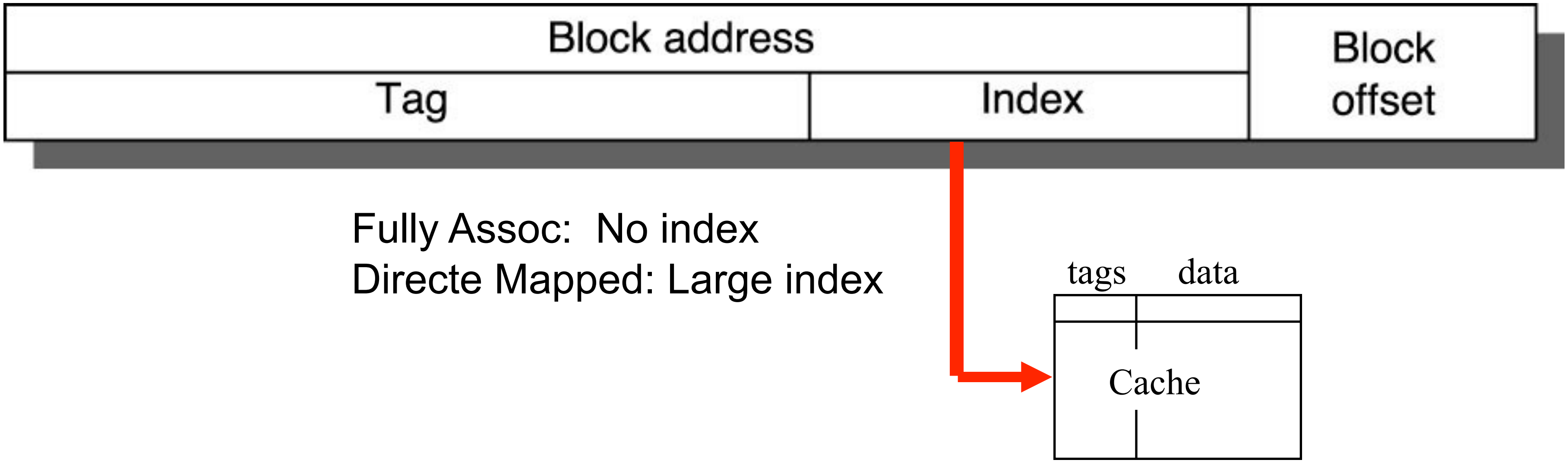
(set = a collection of blocks cache blocks with the same cache index)



Cache Access: How Is a Block Found In the Cache?

- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag

Block Address

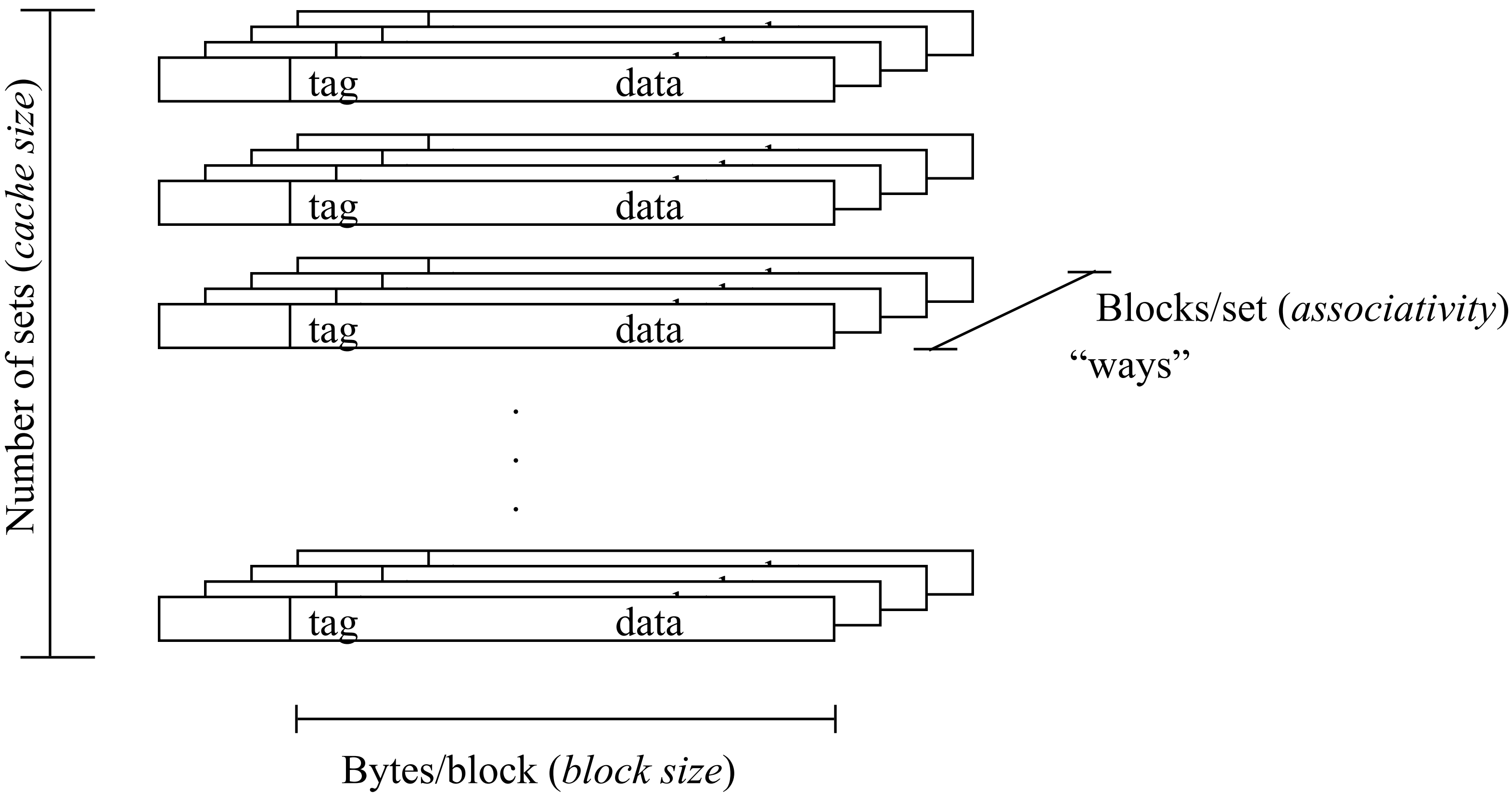


Fully Assoc: No index
Directe Mapped: Large index

Cache Organization

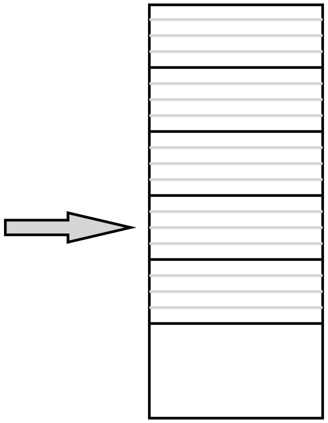
- A typical cache has three dimensions

tag	index	block offset


Number of sets (*cache size*)
tag
data
tag
data
tag
data
tag
data
Blocks/set (*associativity*)
“ways”
Bytes/block (*block size*)

Which Block Should be Replaced on a Miss?

- Direct Mapped is Easy
- Set associative or fully associative:
  - "Random" (large associativities)
  - LRU (smaller associativities)
  - Pseudo Associative

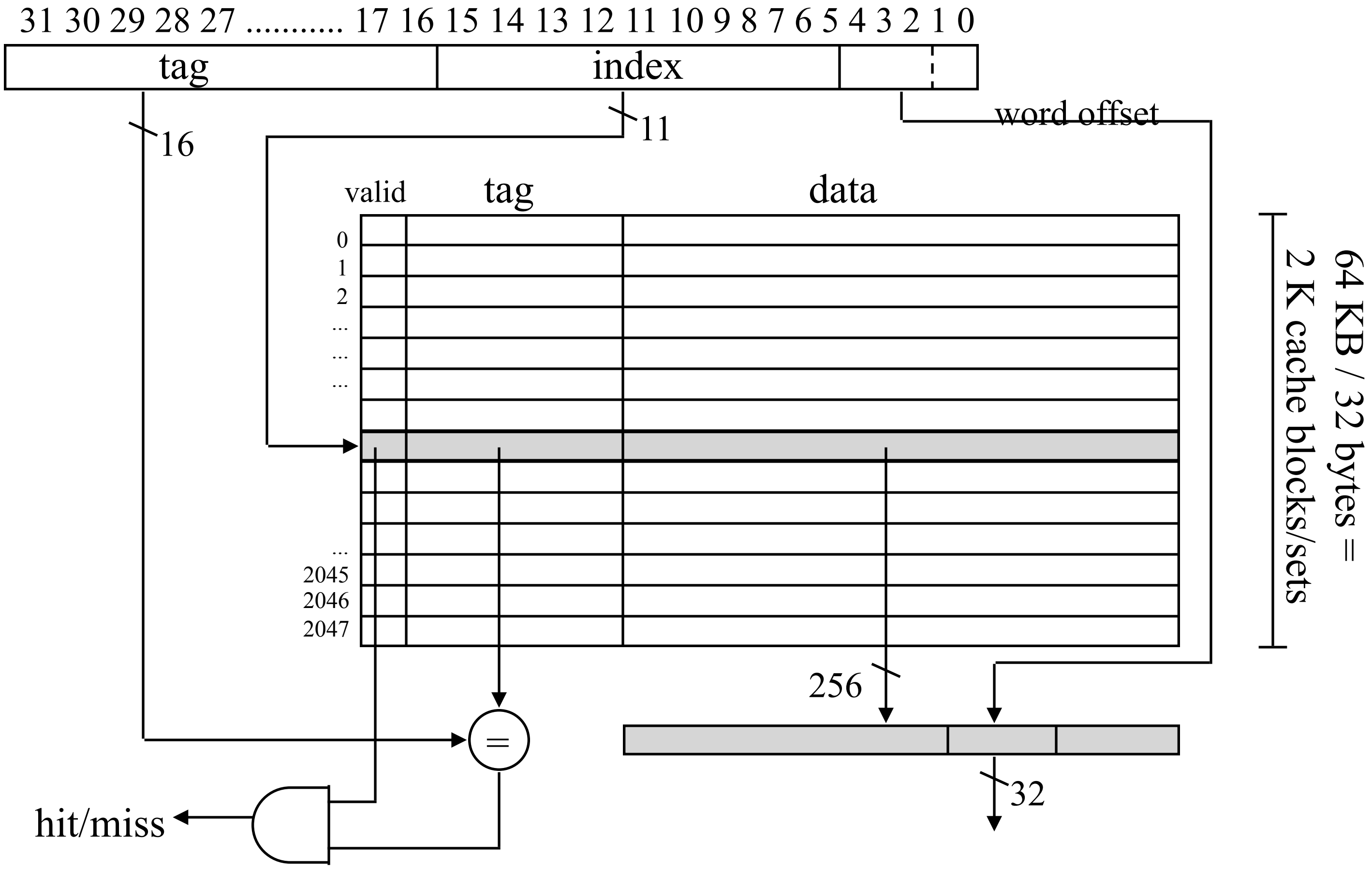


Associativity:		2-way		4-way		8-way
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Numbers are averages across a set of benchmarks. Performance improvements vary greatly by individual benchmarks.

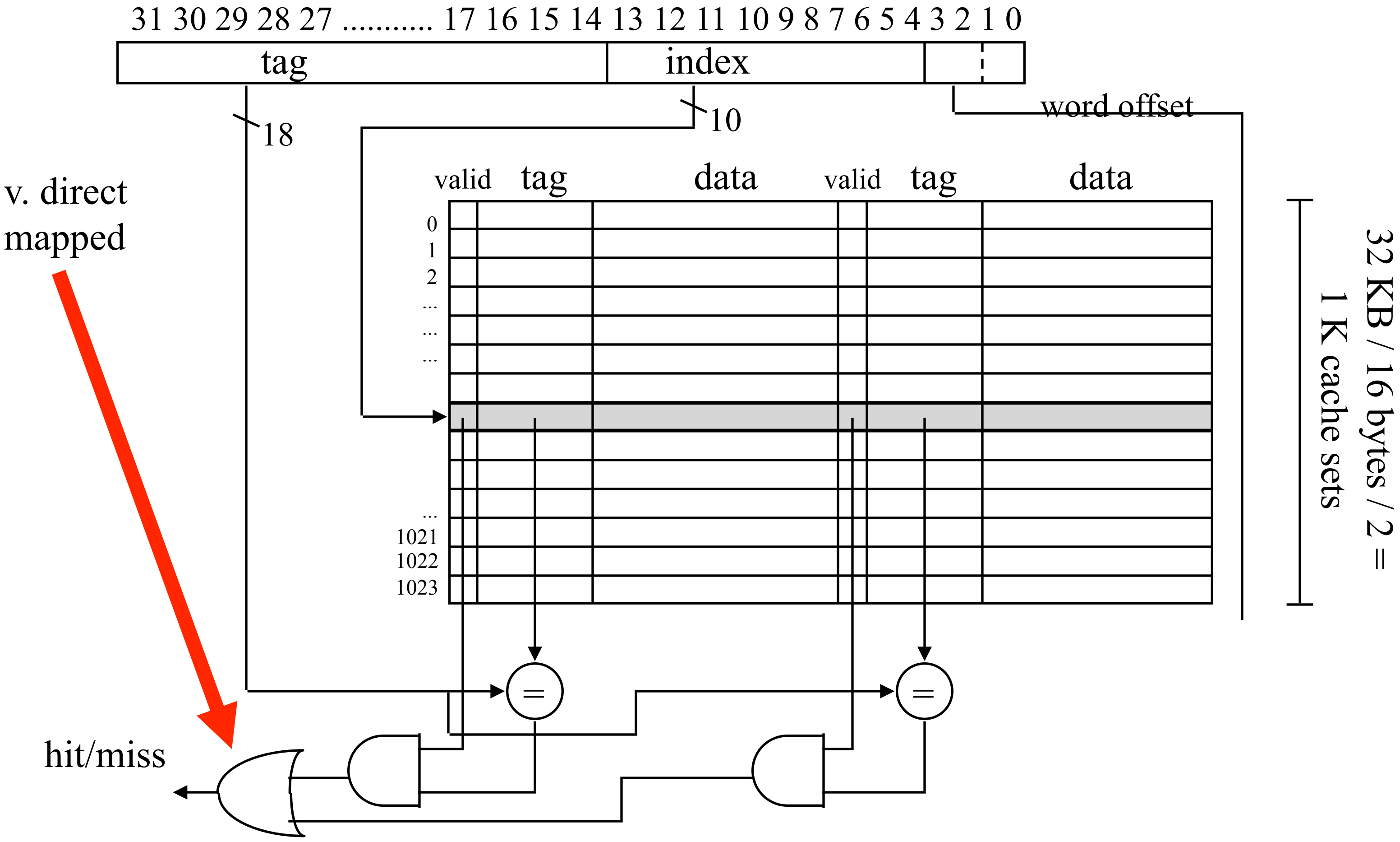
Accessing a Direct Mapped Cache

- 64 KB cache, direct-mapped, 32-byte cache block size



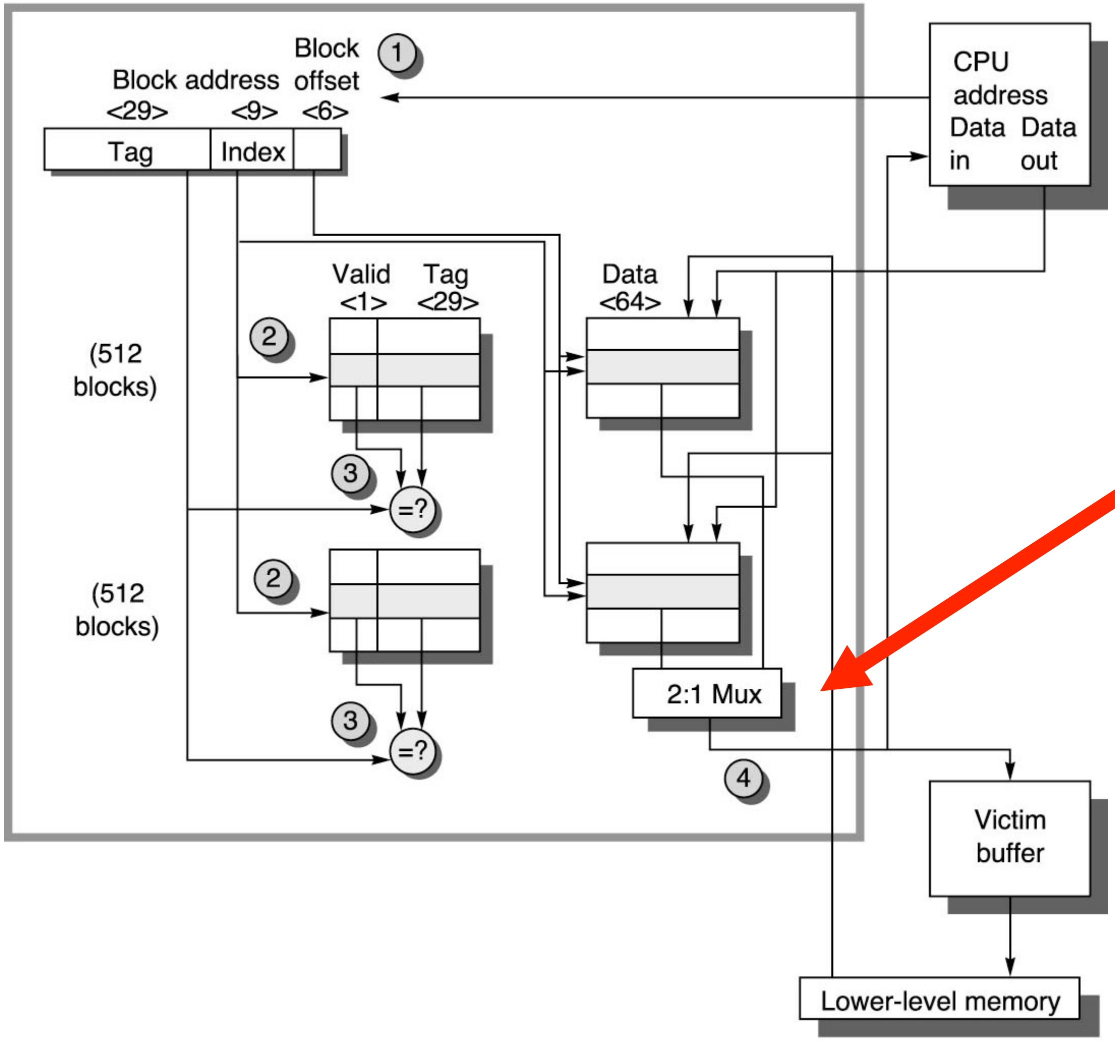
Accessing a 2-way Assoc Cache – Hit Logic

- 32 KB cache, 2-way set-associative, 16-byte block size



21264 L1 Cache

- 64 KB, 64-byte blocks, (SA (set assoc) = 2)

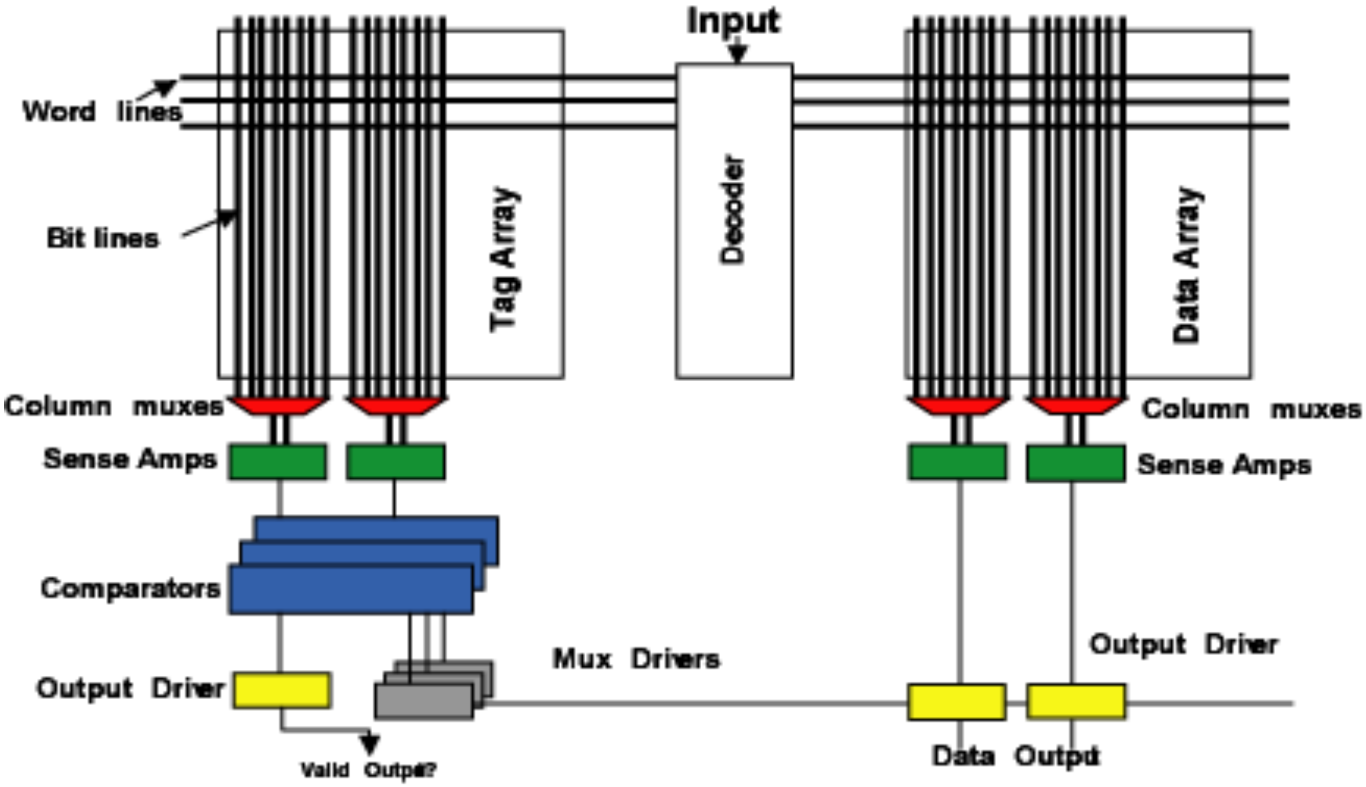


v. direct mapped – must know correct line that contains data to control mux –

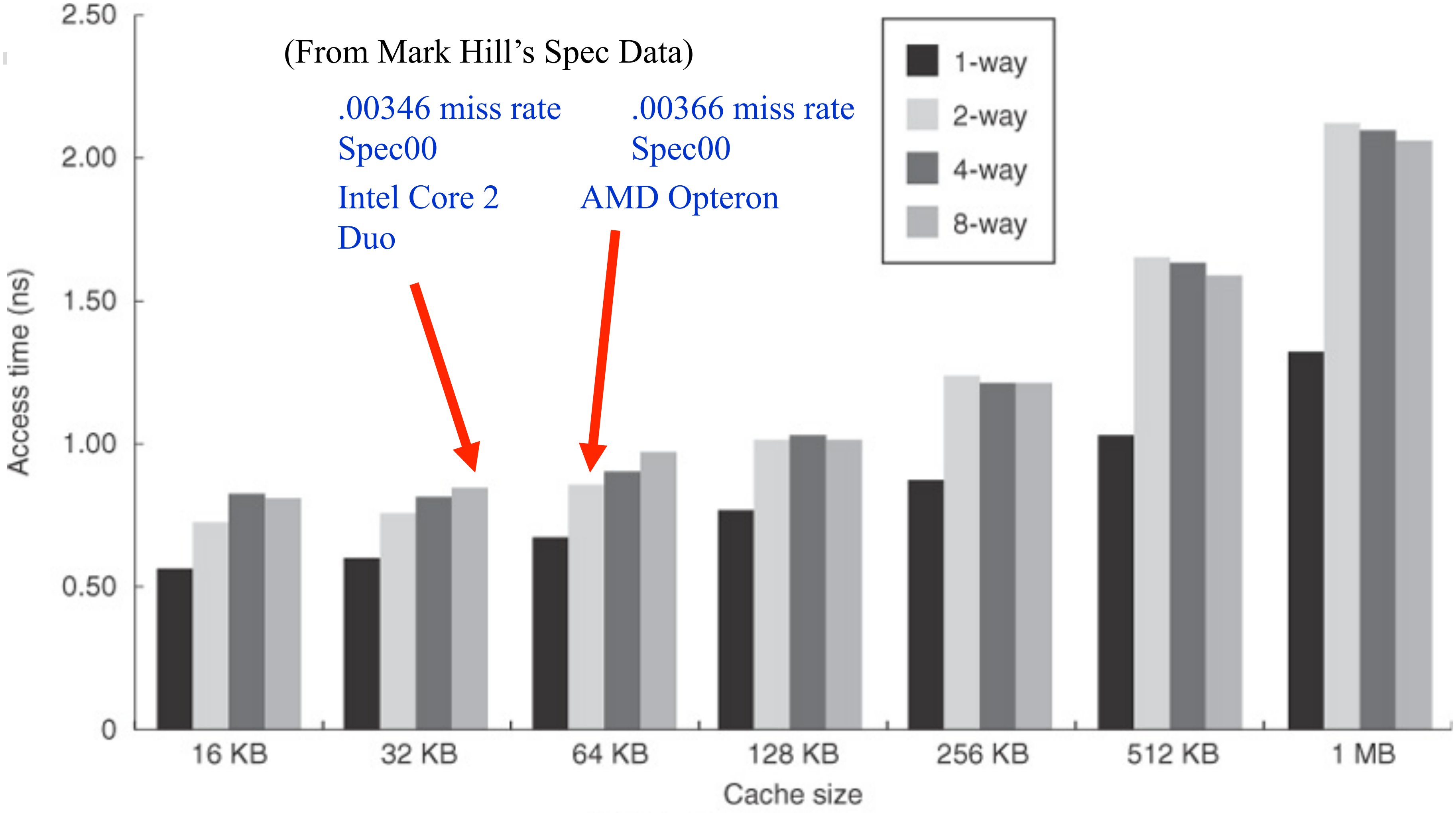
direct mapped cache can operate on data without waiting for tag

set assoc needs to know which set to operate on!
→ line predictor

Cache Organization



Evaluation of Cache Access Time via. Cacti + Simulation – Intel wins by a hair



(From Mark Hill's Spec Data)

.00346 miss rate
Spec00
Intel Core 2
Duo

.00366 miss rate
Spec00
AMD Opteron

90 nm, 64-byte clock, 1 bank

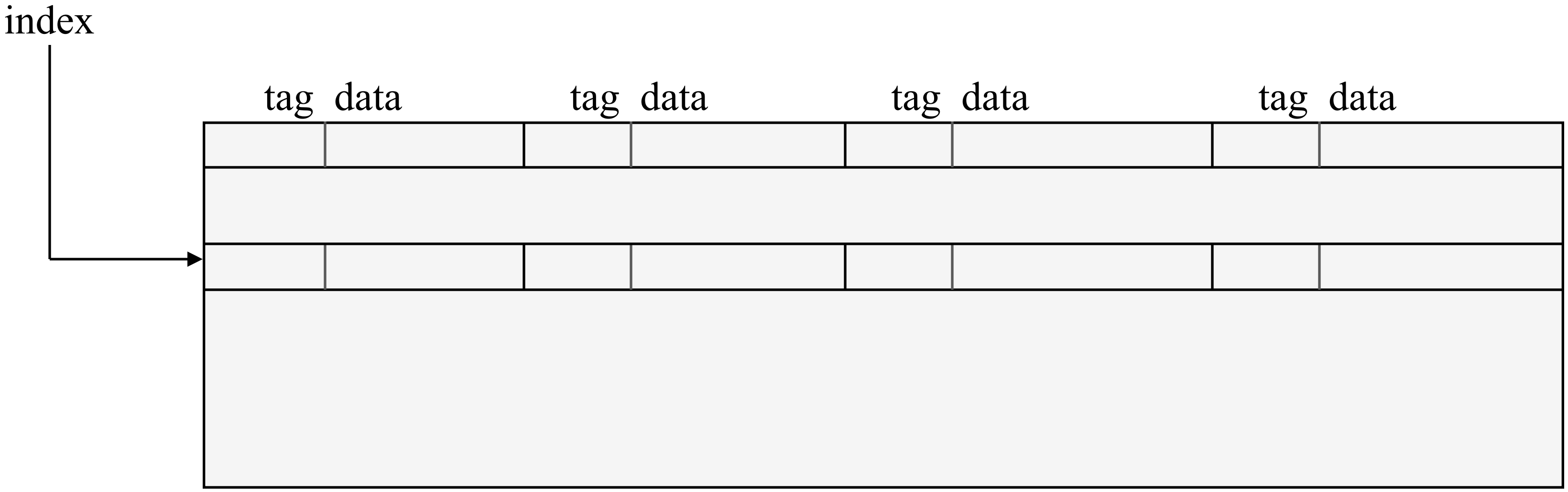
Cache Access

- Cache Size = #sets * block size * associativity
- What is the Cache Size, if we have direct mapped, 128 set cache with a 32-byte block?

- What is the Associativity, if we have 128 KByte cache, with 512 sets and a block size of 64-bytes?

Cache Access

- 16 KB, 4-way set-associative cache, 32-bit address, byte-addressable memory, 32-byte cache blocks/lines
- how many tag bits?
- Where would you find the word at address 0x200356A4?

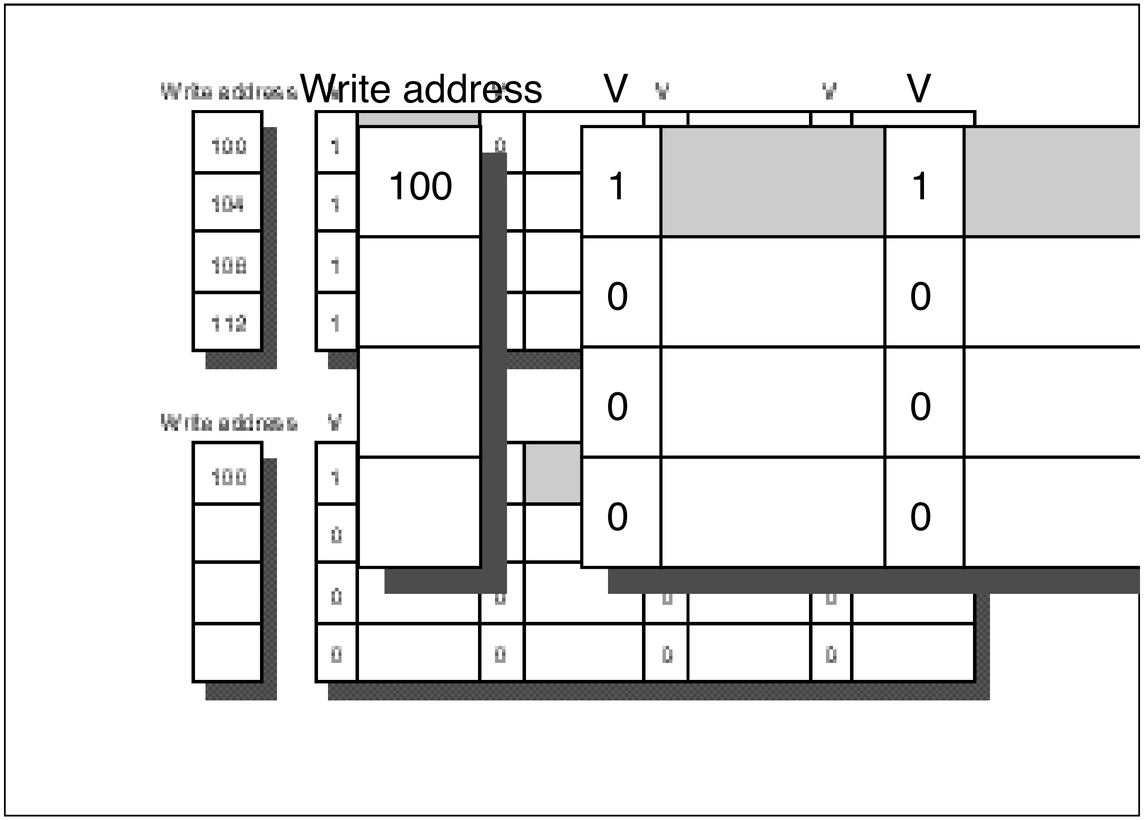


What Happens on a Write?

- *Write through*: The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each:
  - WT: read misses do not need to write back evicted line contents
  - WB: no writes of repeated writes
- WT always combined with *write buffers* so that don’t wait for lower level memory

What About Write Miss?

- *Write allocate*: The block is loaded into cache on a write miss
- *No-Write allocate*: The block is modified in the lower levels of memory but not in cache
- Write buffer allows merging of writes



Write address	V		V	
100	1	▨	1	▨
	0		0	
	0		0	
	0		0	

Unified versus Separate I+D L1 Cache (Princeton vs. Harvard Arch)

Separate Instruction Cache and Data Cache?

Size	Instruction Cache	Data Cache	Unified Cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

Area-limited designs may consider unified caches

Generally, the benefits of separating the caches are overwhelming… (what are the benefits?)

Cache Performance

- CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time

- Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty + Writes x Write miss rate x Write miss penalty)

- Memory stall clock cycles = Memory accesses x Miss rate x Miss penalty

Cache Performance

CPUtime = IC x (CPI$_{execution}$ + Memory stalls per instruction) x Clock cycle time

CPUtime = IC x (CPI$_{execution}$ + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

(includes hit time as part of CPI)

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

Improving Cache Performance

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

How are we going to improve cache performance??

1.

2.

3.

Cache Overview Key Points

- CPU-Memory gap is a major performance obstacle
- Caches take advantage of program behavior: locality
- Designer has lots of choices -> cache size, block size, associativity, replacement policy, write policy, ...
- Time of program still only reliable performance measure

Improving Cache Performance

1. Reduce the miss rate,

2. Reduce the miss penalty, or

3. Reduce the time to hit in the cache.

Reducing Misses

- Classifying Misses: 3 Cs
  - *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.
  - *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.
  - *Conflict*—If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses* or *interference misses*.

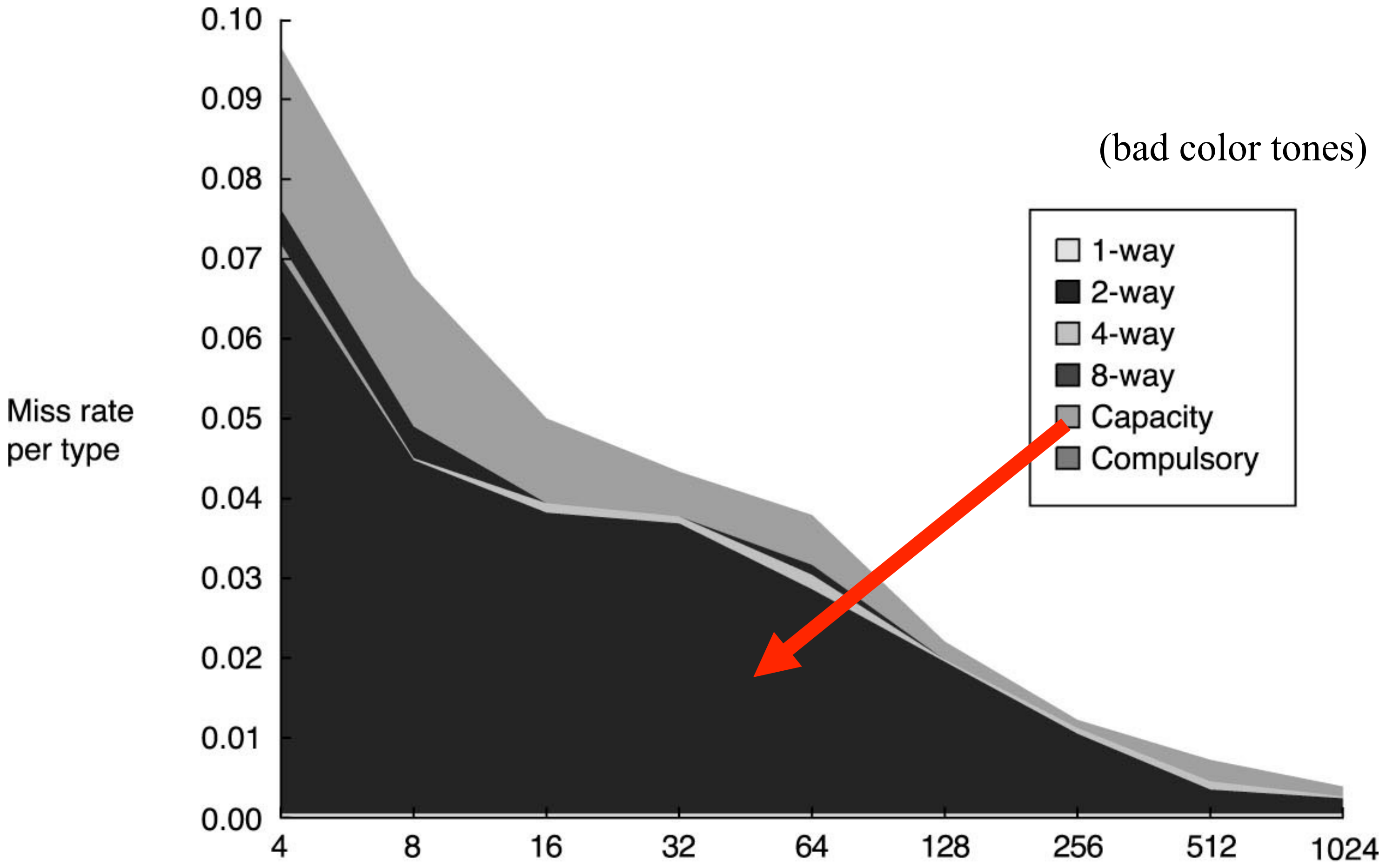
How To Measure

Misses in infinite cache

Non-compulsory misses in size X fully associative cache

Non-compulsory, non-capacity misses

3Cs Absolute Miss Rate

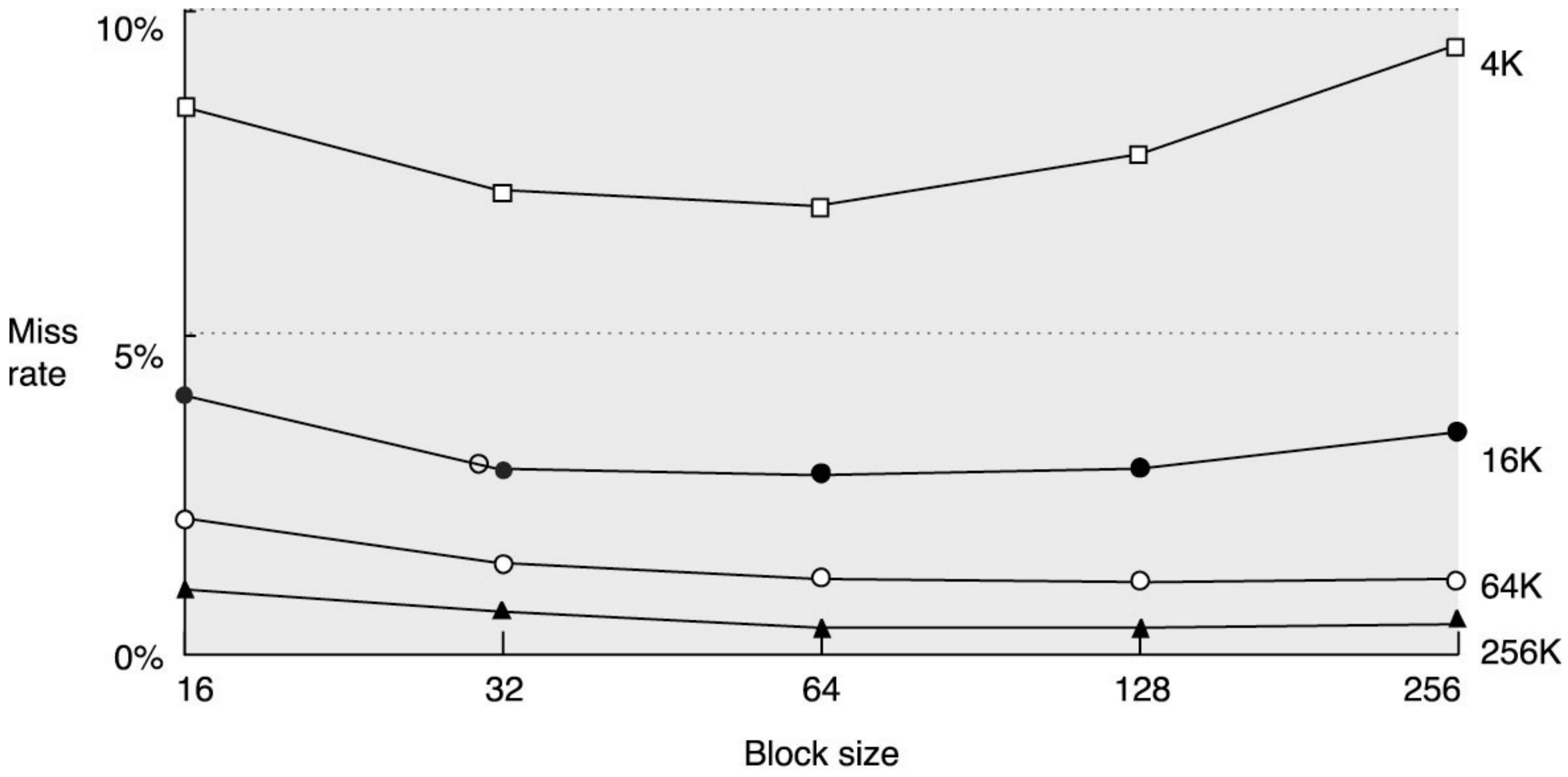


(bad color tones)

How To Reduce Misses?

- Compulsory Misses?
- Capacity Misses?
- Conflict Misses?
- What can the compiler do?

1. Reduce Misses via Larger Block Size



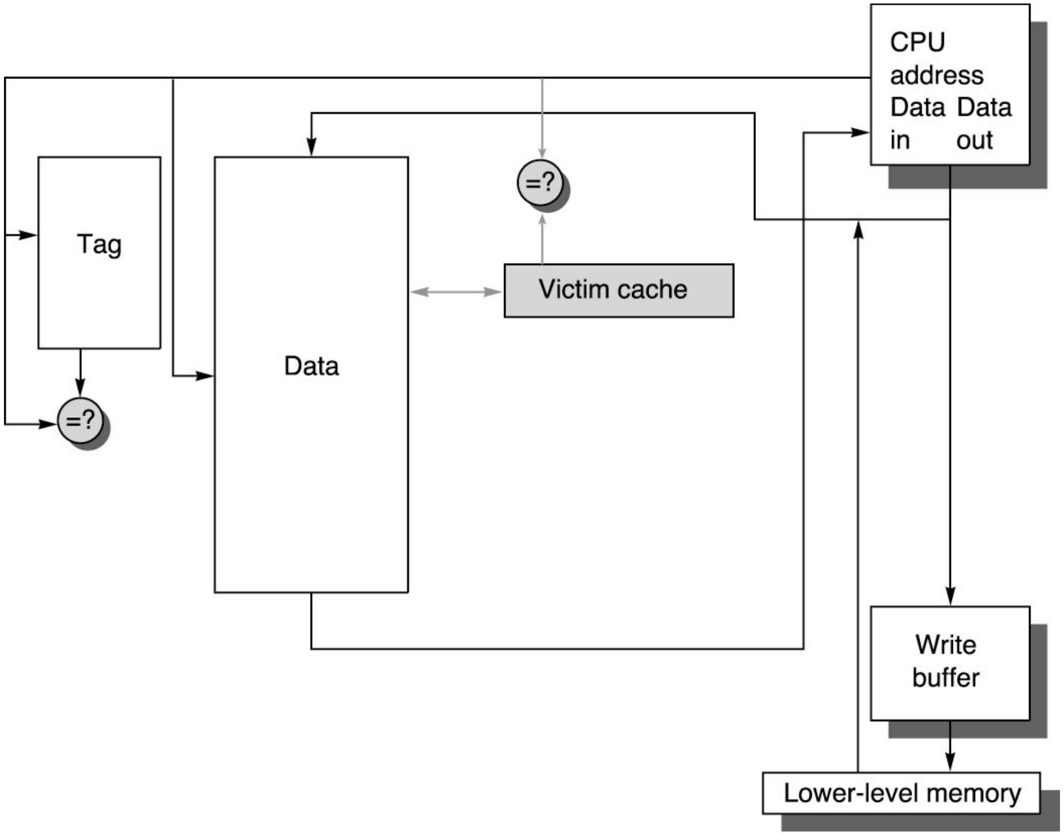
- 16K cache, miss penalty for 16-byte block = 42, 32-byte is 44, 64-byte is 48. Miss rates are 3.94, 2.87, and 2.64%?

2. Reduce Misses via Higher Associativity

- 2:1 Cache Rule:
  - Miss Rate DM cache size N ==
  - Miss Rate 2-way associative cache size N/2
- Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?

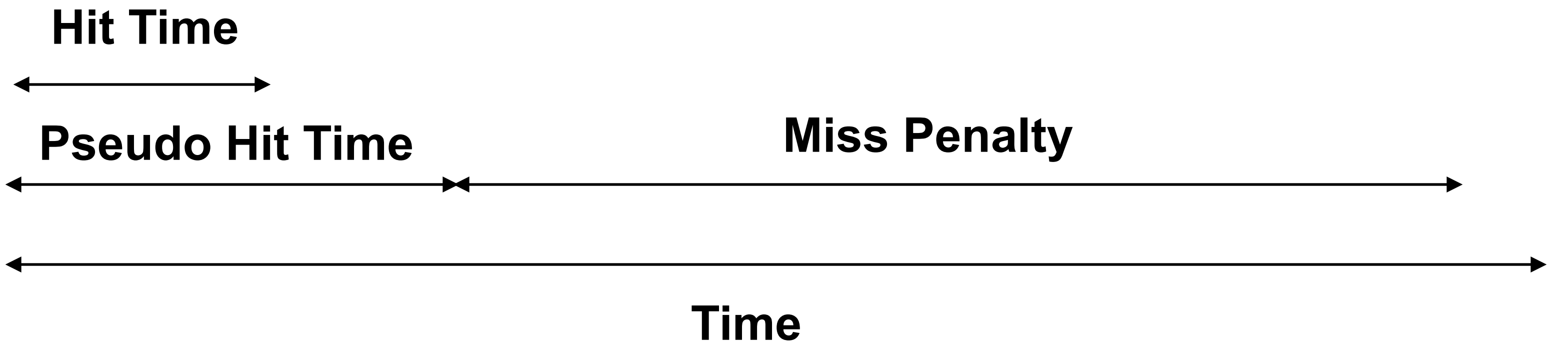
3. Reducing Misses via Victim Cache

- How to get fast hit time of Direct Mapped yet still avoid conflict misses?
- Add buffer 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



4. Reducing Misses via Pseudo-Associativity

- Combines fast hit time of Direct Mapped and the lower conflict misses of a 2-way SA cache.
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)



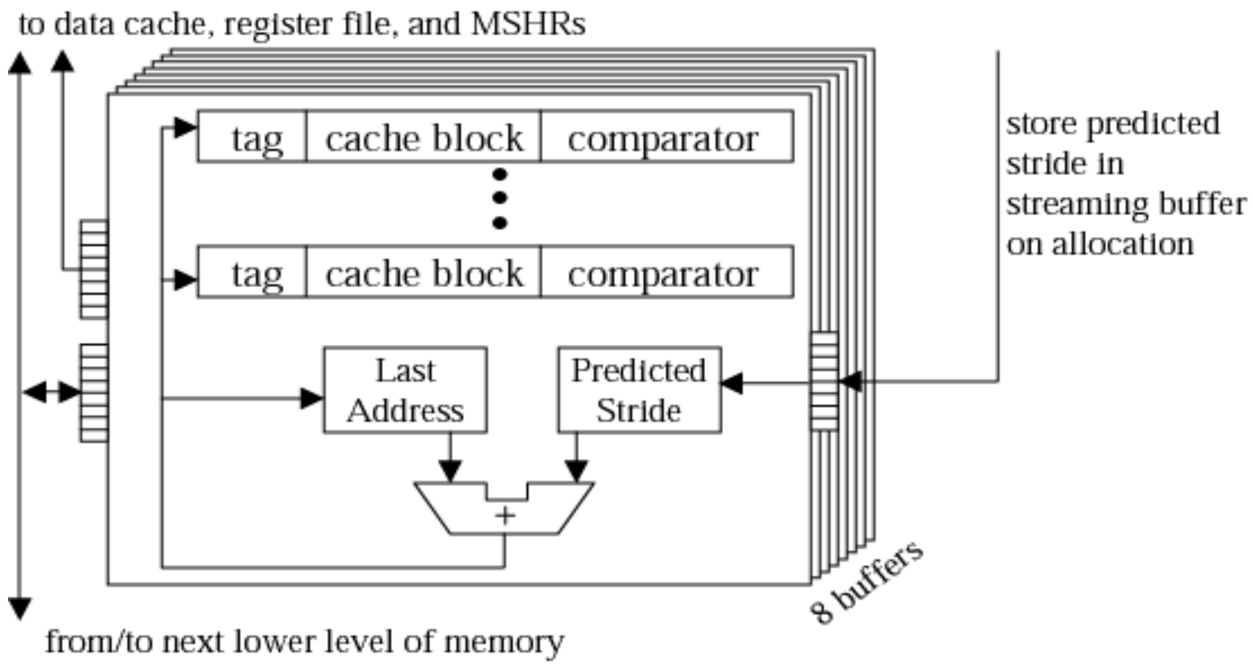
- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Better for caches not tied directly to processor

5. Reducing Misses by HW Prefetching of Instruction & Data

- E.g., Instruction Prefetching
  - 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
- Prefetching relies on extra memory bandwidth that can be used without penalty

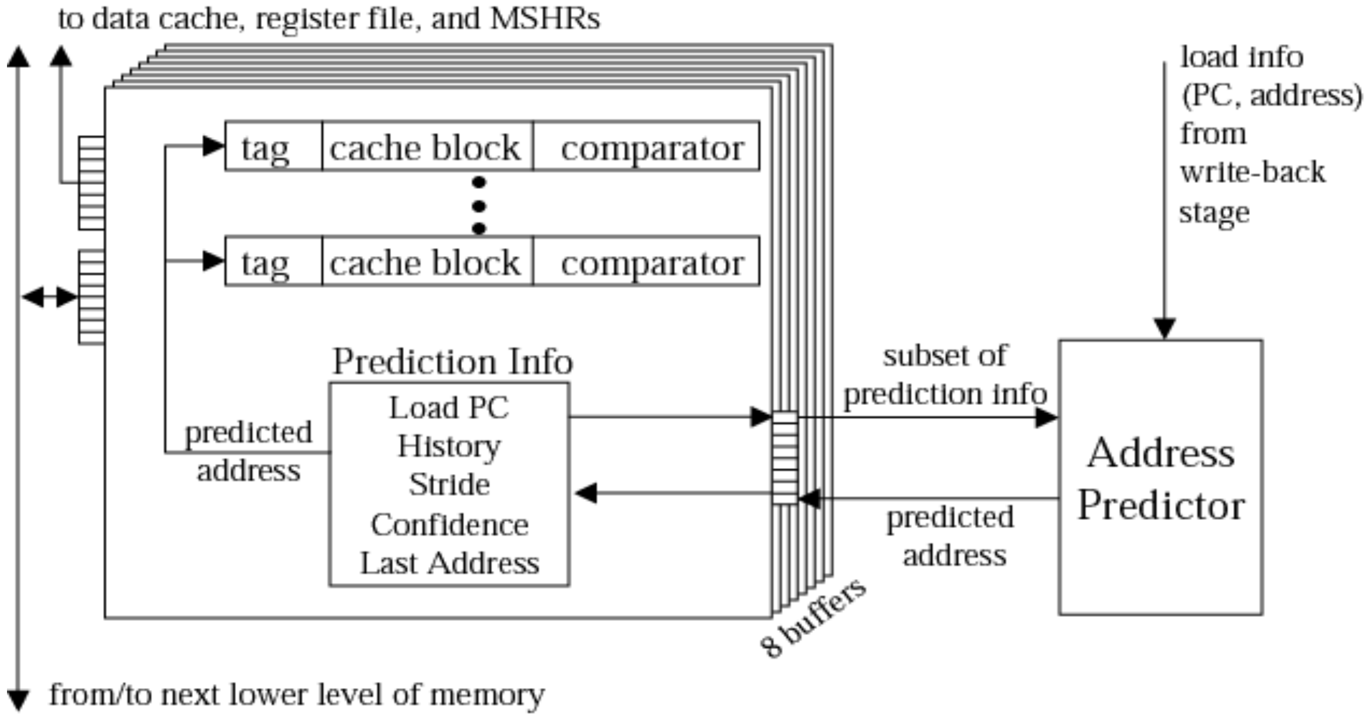
Stream Buffers

- Allocate a Stream Buffer on a cache miss
- Run ahead of the execution stream prefetching N blocks into stream buffer
- Search stream buffer in parallel with cache access
- If hit, then move block to cache, and prefetch another block

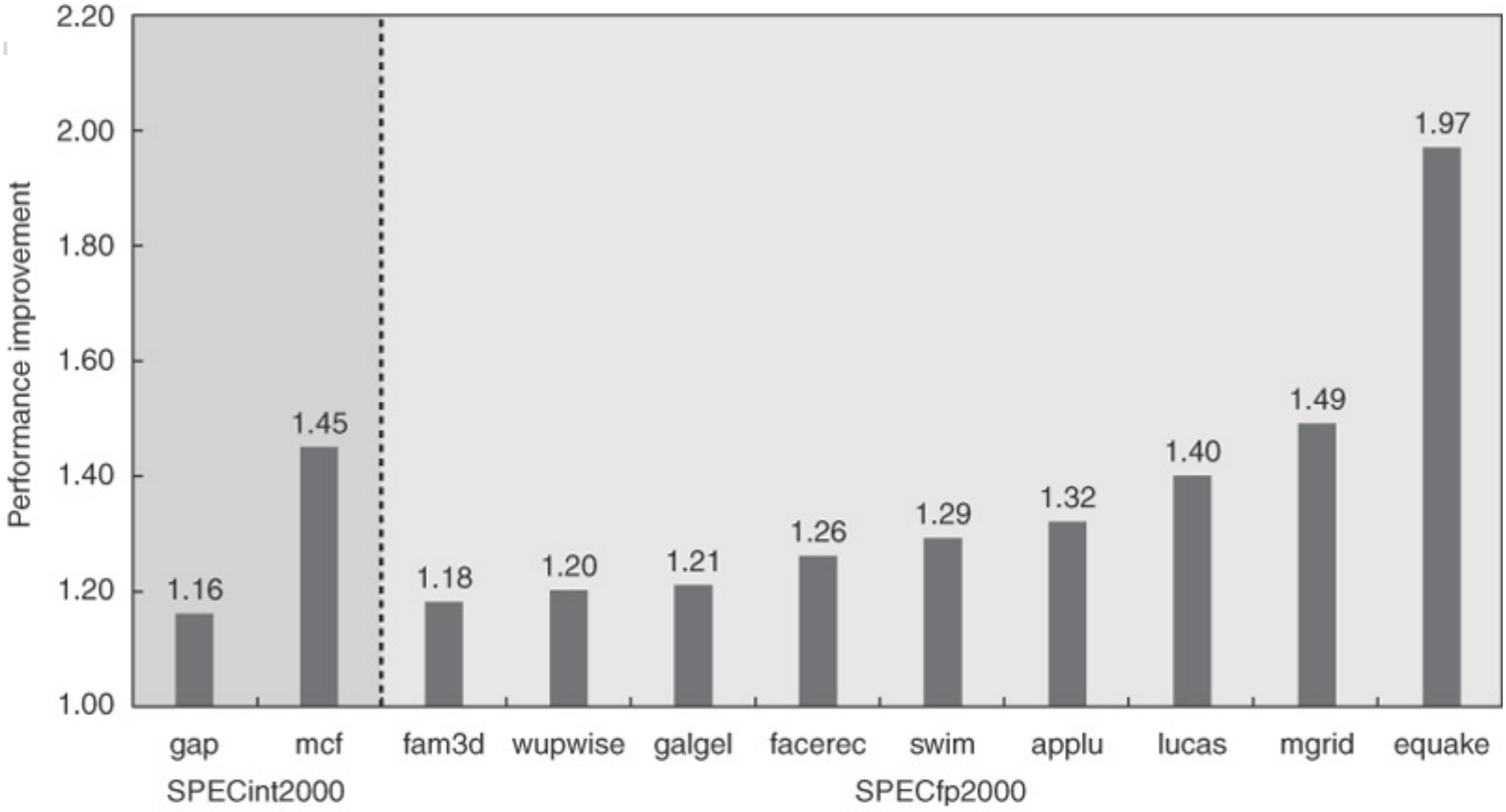

to data cache, register file, and MSHRs
tag
cache block
comparator
tag
cache block
comparator
Last Address
Predicted Stride
+
store predicted stride in streaming buffer on allocation
8 buffers
from/to next lower level of memory

Predictor-Directed Stream Buffers

- Effective for next-line and stride prefetching, what about pointers?
- Allow stream buffer to follow any prediction stream.
- Provide confidence techniques for stream allocation
  - Reduce stream thrashing



Impact of hardware prefetching on Pentium 4 – pretty large (15 missing benchmarks benefited less than 15%)



6. Reducing Misses by Software Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults
    - can be used speculatively
    - load into a r0
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?

7. Reducing Misses by Various Compiler Optimizations

- Instructions
  - Reorder procedures in memory so as to reduce misses
  - Profiling to look at conflicts
  - results in a 30% reduction in miss rate for an 8K I-cache
- Data
  - ***Reordering***
    - results in a 40% reduction in cache miss rate for an 8K D-cache
  - ***Merging Arrays***: improve spatial locality by single array of compound elements vs. 2 arrays
  - ***Loop Interchange***: change nesting of loops to access data in order stored in memory
  - ***Loop Fusion***: Combine 2 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 */
int val[SIZE];
int key[SIZE];

/* After */
struct merge {
   int val;
   int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key, create 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];
```

Sequential accesses instead of striding through memory every 100 words

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];}
```

2 misses per access to a & c vs. one miss per 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;
      };
```

- 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:
  - worst case => $2N^3 + N^2$.
- Idea: compute on BxB submatrix that fits in cache

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;
     };
```

- Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- B called *Blocking Factor*
- Conflict Misses Are Not As Easy...

Key Points

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

- 3 Cs: Compulsory, Capacity, Conflict
  - 1. Reduce Misses via Larger Block Size
  - 2. Reduce Misses via Higher Associativity
  - 3. Reducing Misses via Victim Cache
  - 4. Reducing Misses via Pseudo-Associativity
  - 5. Reducing Misses by HW Prefetching Instr, Data
  - 6. Reducing Misses by SW Prefetching Data
  - 7. Reducing Misses by Compiler Optimizations
- Remember danger of concentrating on just one parameter when evaluating performance
- Next: reducing Miss penalty

Improving Cache Performance

1. Reduce the miss rate,

2. Reduce the miss penalty, or

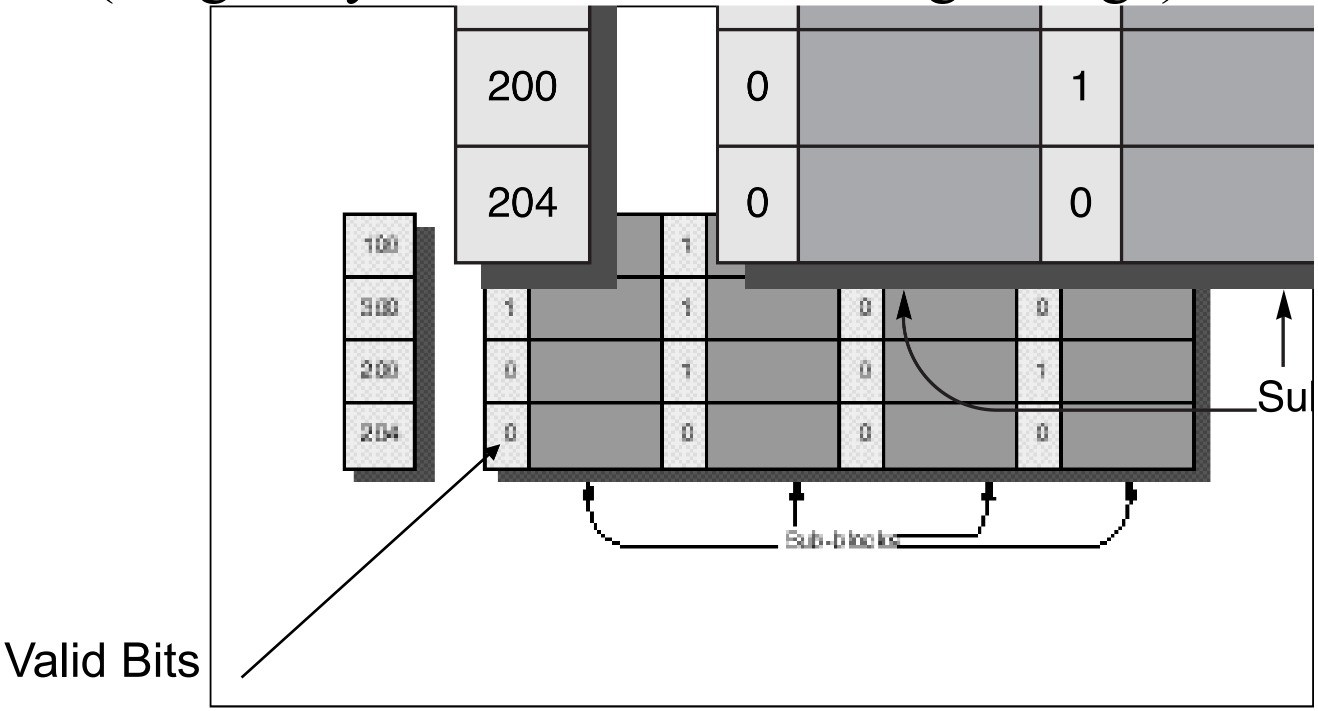
3. Reduce the time to hit in the cache.

1. Reducing Miss Penalty: Read Priority over Write on Miss

- 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 by 50%
- Check write buffer contents before read;
  if no conflicts, let the memory access continue
- Write Back?
  - Read miss may require write of 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 stalls less since it can restart as soon as read completes

2. Subblock Placement to Reduce Miss Penalty (also known as sectoring)

- Don’t have to load full block on a miss
- Have bits per subblock to indicate valid
- (Originally invented to reduce tag storage)



some PPC machines do this.

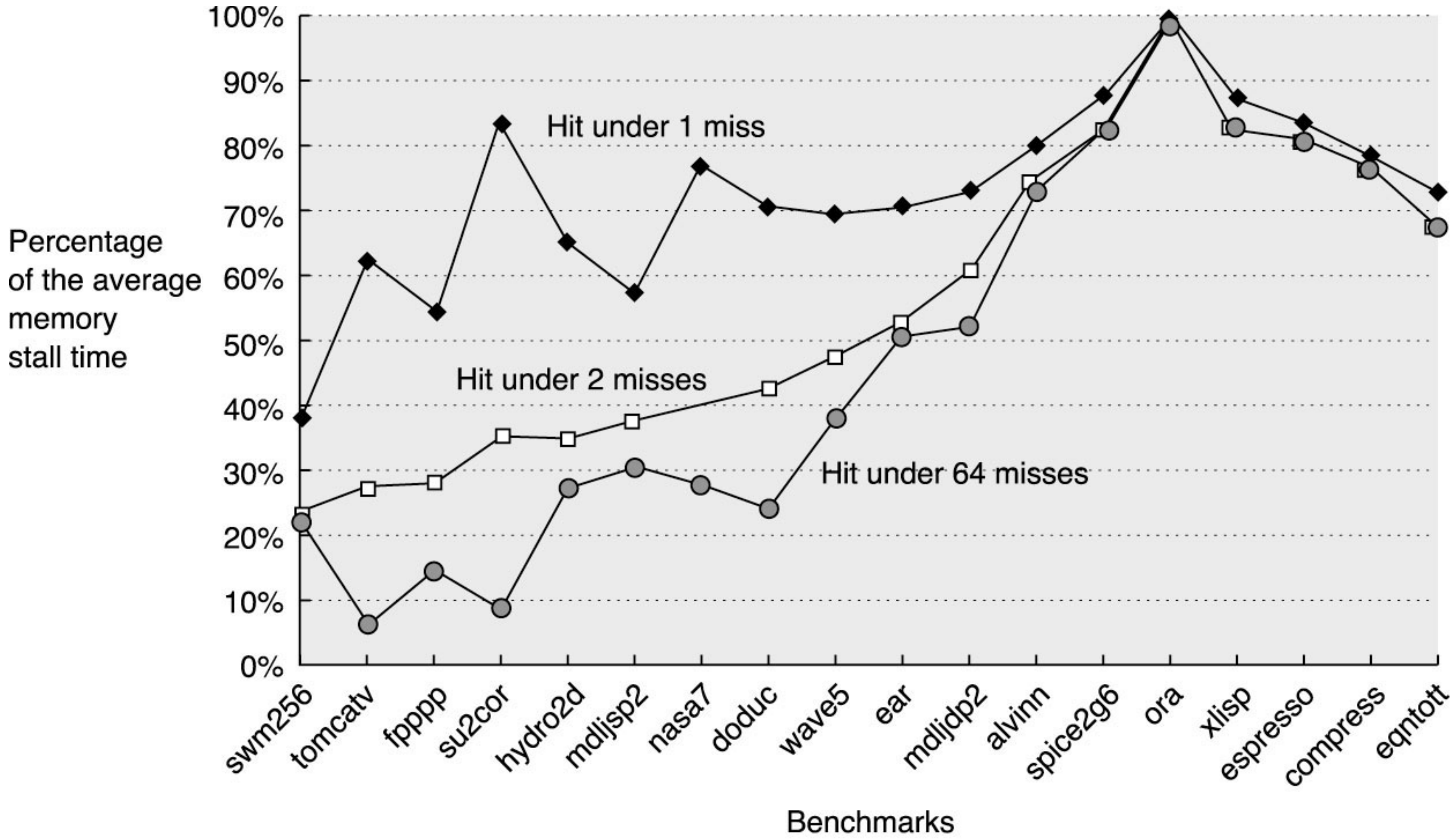
3. Early Restart and Critical Word First

- 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 and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called *wrapped fetch* and *requested word first*
- Most useful with large blocks,
- Spatial locality a problem; often we next want the next sequential word soon, so not always a benefit (early restart).

4. Non-blocking Caches to reduce stalls on misses

- *Non-blocking cache* or *lockup-free cache* allowing the data cache to continue to supply cache hits during a miss
- "*hit under miss*" reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the CPU
- "*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
- assumes "stall on use" not "stall on miss" which works naturally with dynamic scheduling, but can also work with static.

Value of Hit Under Miss for SPEC



- 8 KB cache, 16 cycle miss, 32-byte blocks
- old data; good model for misses to L2, not a good model for misses to main memory (~ 300 cycles)

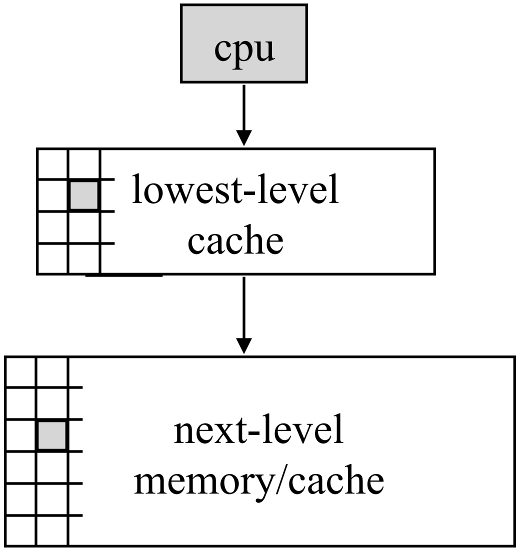
5. Use Multi-level caches

- L2 Equations

  $\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \text{ x Miss Penalty}_{L1}$

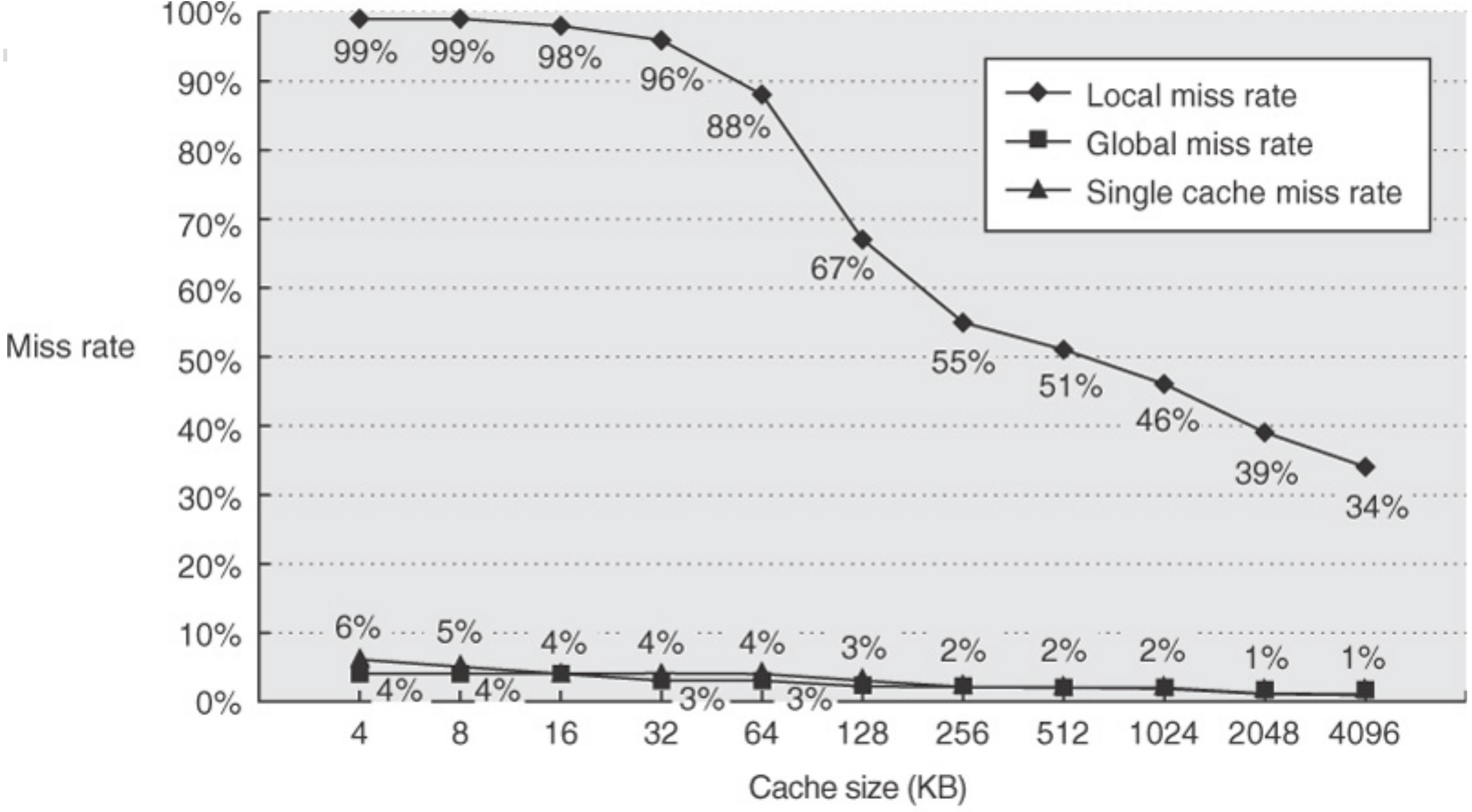
  $\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \text{ x Miss Penalty}_{L2}$

  $\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1}$
  $\text{x (Hit Time}_{L2} + \text{Miss Rate}_{L2} \text{ x Miss Penalty}_{L2})$



- Definitions:
  - *Local miss rate*— misses in this cache divided by the total number of memory accesses *to this cache* (Miss rate_{L2})
  - *Global miss rate*—misses in this cache divided by the total number of memory accesses *generated by the CPU* ($\text{Miss Rate}_{L1} \text{ x Miss Rate}_{L2}$)

L2 local miss rates pretty high

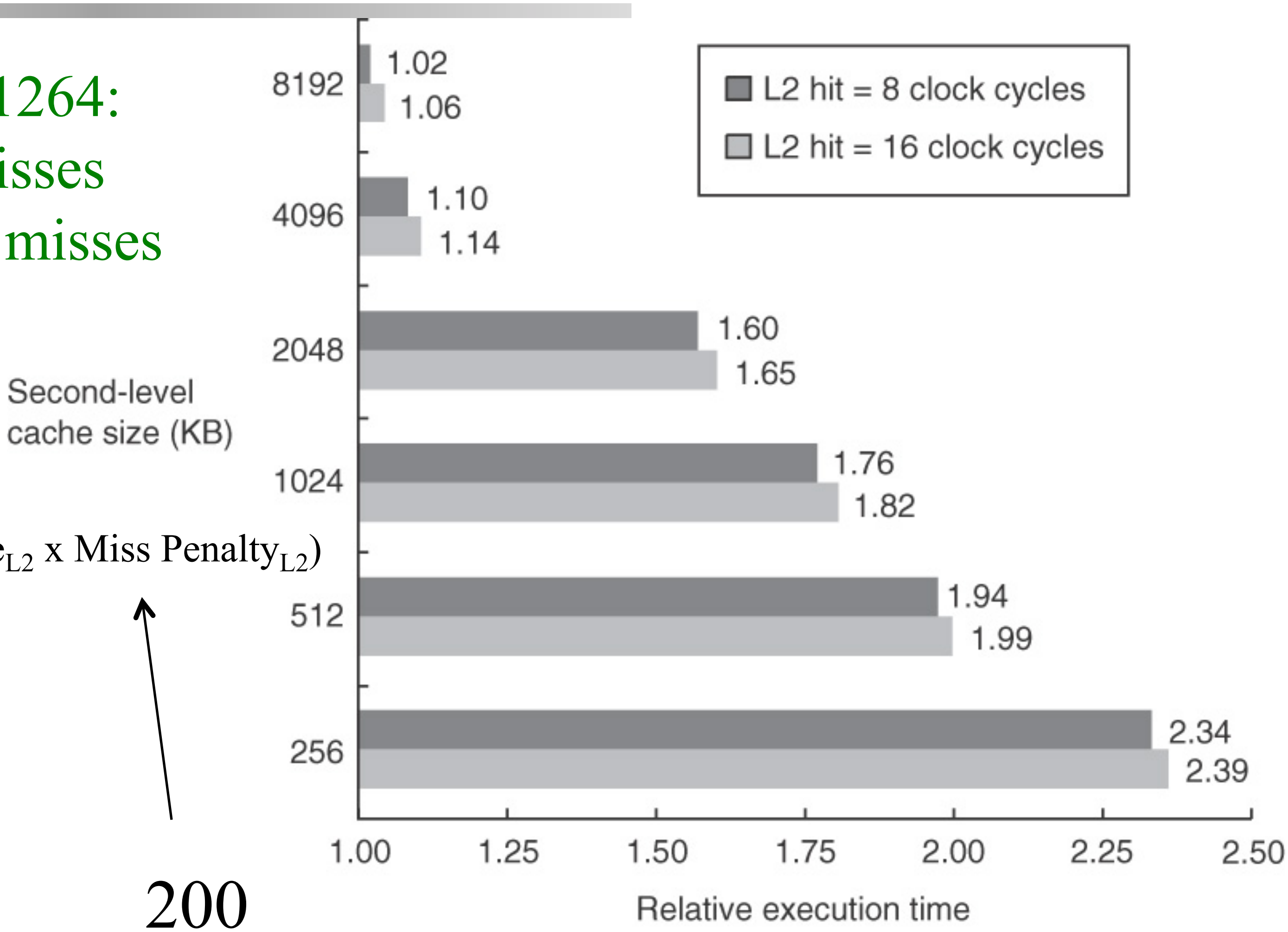


© 2007 Elsevier, Inc. All rights reserved.

L2 Size More Important Than Latency: Why?

Experiment is on 21264:
OoO hides L1 misses
well but not L2 misses

$$\text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \text{ x } (\underset{\substack{\uparrow \\ 8 \text{ or } 16}}{\text{Hit Time}_{L2}} + \underbrace{\underset{\substack{\uparrow \\ .5}}{\text{Miss Rate}_{L2}} \text{ x } \underset{\substack{\uparrow \\ 200}}{\text{Miss Penalty}_{L2}}}_{=100})$$



Multi-level Caches, cont.

- L1 cache local miss rate 10%, L2 local miss rate 40%. What are the global miss rates?
- L1 highest priority is fast hit time. L2 typically low miss rate.
- Design L1 and L2 caches in concert.
- Property of inclusion -- if it is in L2 cache, it is guaranteed to be in one of the L1 caches -- simplifies design of consistent caches.
- L2 cache can have different associativity (good idea?) or block size (good idea?) than L1 cache.

Reducing Miss Penalty Summary

- Five techniques
  - Read priority over write on miss
  - Subblock placement
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit Under Miss)
  - Multi-levels of Cache

Improving Cache Performance

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. *Reduce the time to hit in the cache.*