Caches Part II

slides adapted from Patterson's 61C

Review

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible
- So we create a memory hierarchy:
  - each successively lower level contains “most used” data from next lower level
  - exploits temporal locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea

Big Idea Review

- Mechanism for transparent movement of data among levels of a storage hierarchy
  - set of address/value bindings
  - address provides index to ***set*** of candidates
  - compare desired address with tag
  - service hit or miss
    - load new block and binding on miss

address: tag: 000000000000000000 index: 0000000001 offset: 1100

Index	Valid	Tag	0x0 - 0x3	0x4 - 0x7	0x8 - 0xb	0xc - 0xf
0						
1	1	0	a	b	c	d
2						
3						
⋮						

Outline

- Block Size Tradeoff
- Types of Cache Misses
- Fully Associative Cache
- N-Way Associative Cache
- Block Replacement Policy
- Multilevel Caches (if time)
- Cache write policy (if time)

Block Size Tradeoff (1/3)

- Benefits of Larger Block Size
  - Spatial Locality: if we access a given word, we're likely to access other nearby words soon (Another Big Idea)
  - Very applicable with Stored-Program Concept: if we execute a given instruction, it's likely that we'll execute the next few as well
  - Works nicely in sequential array accesses too

Block Size Tradeoff (2/3)

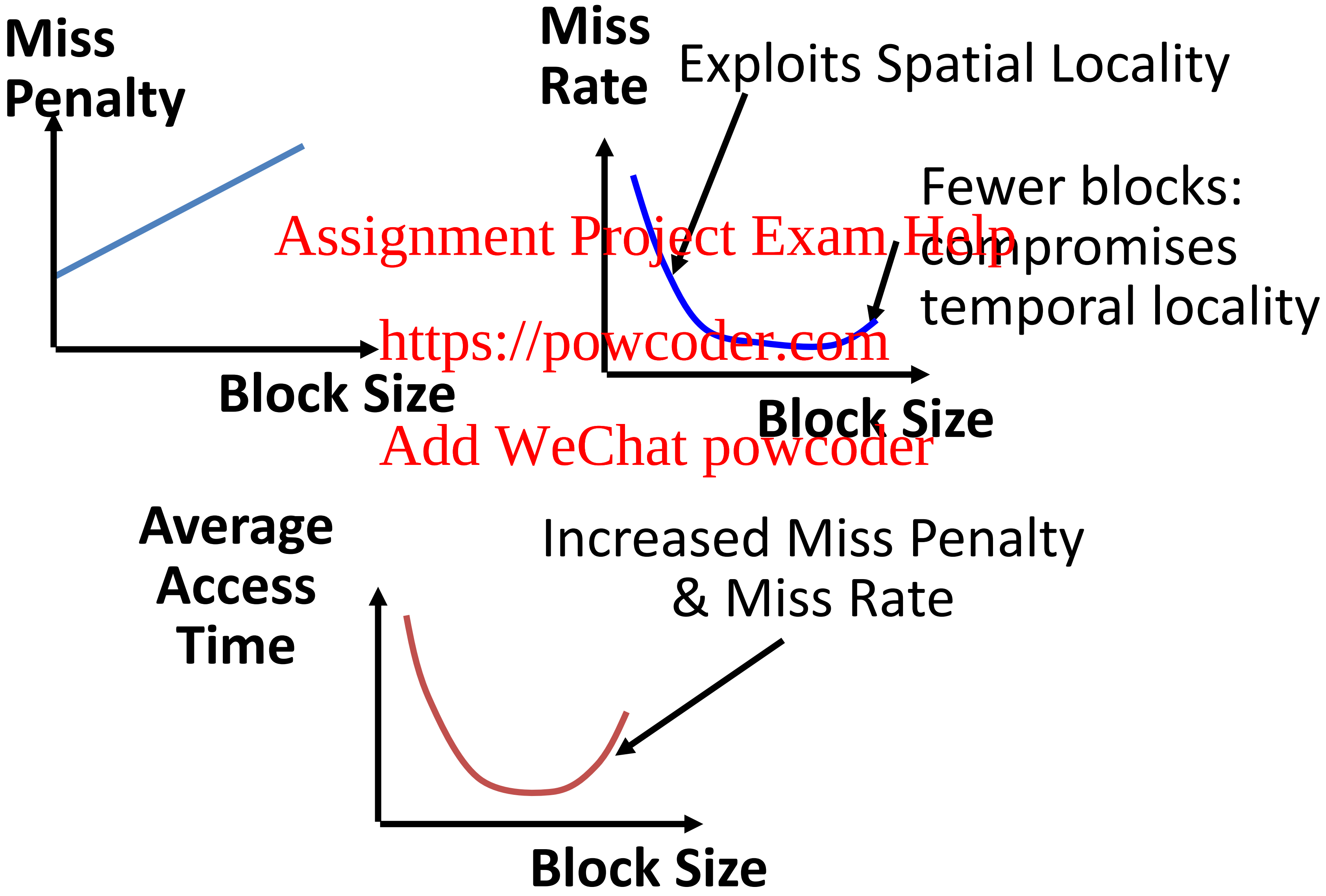
- Drawbacks of Larger Block Size
  - Larger block size means larger miss penalty
    - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
    - Result: miss rate goes up
- In general, minimize
  Average Access Time

  = Hit Time + Miss Penalty x Miss Rate

Block Size Tradeoff (3/3)

- Hit Time = time to find and retrieve data from current level cache
- Miss Penalty = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- Hit Rate = % of requests that are found in current level cache
- Miss Rate = 1 - Hit Rate

Block Size Tradeoff Conclusions



Types of Cache Misses (1/2)

- Compulsory Misses
  - occur when a program is first started
  - cache does not contain any of that program's data yet, so misses are bound to occur
  - can't be avoided easily, so won't focus on these in this course

Types of Cache Misses (2/2)

- Conflict Misses
  - miss that occurs because two distinct memory addresses map to the same cache location
  - two blocks (which happen to map to the same location) can keep overwriting each other
  - big problem in direct-mapped caches
  - how do we lessen the effect of these?

Dealing with Conflict Misses

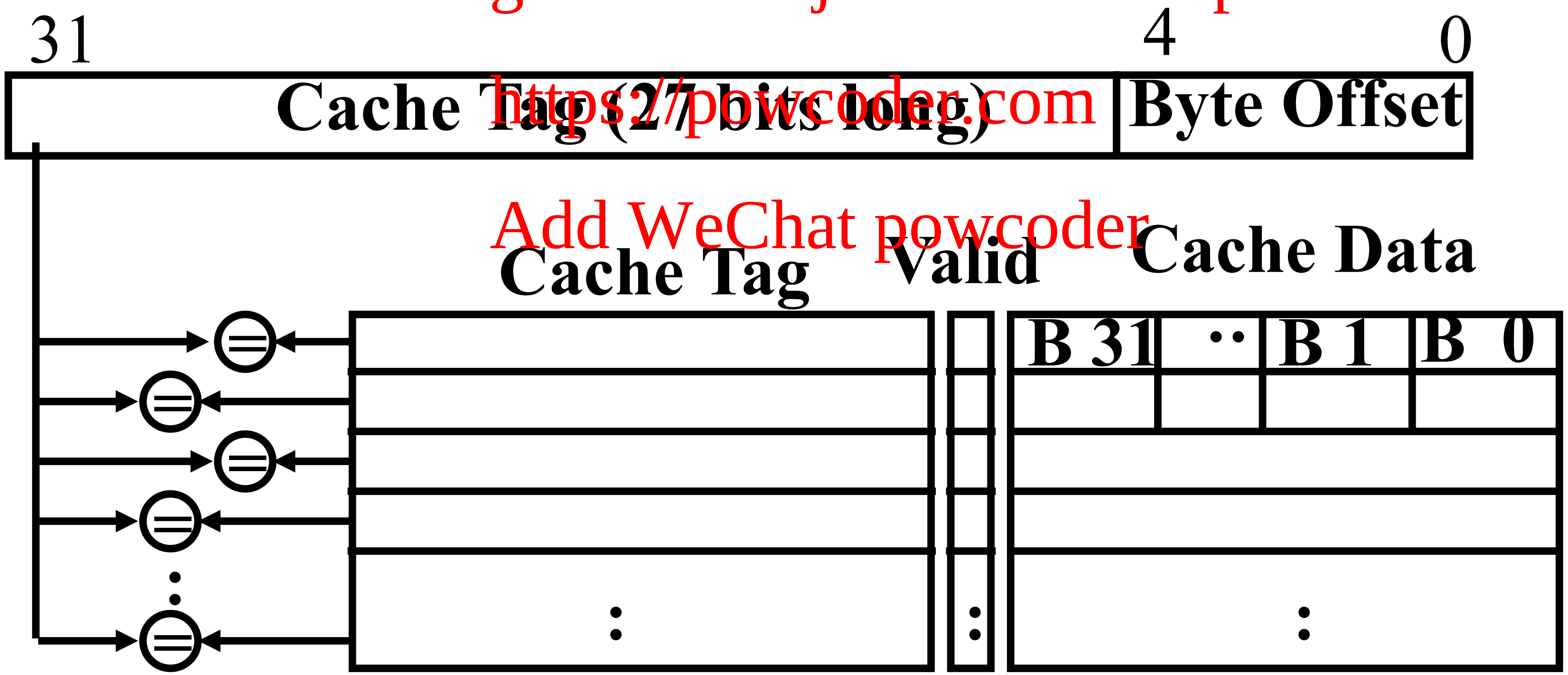
- Solution 1: Make the cache size bigger
  - relatively expensive
- Solution 2: Multiple distinct blocks can fit in the same Cache Index?

Fully Associative Cache (1/3)

- Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: non-existent
- What does this mean?
  - any block can go anywhere in the cache
  - must compare with all tags in entire cache to see if data is there

Fully Associative Cache (2/3)

- Fully Associative Cache (e.g., 32 B block)
  - compare tags in parallel



slides adapted from Patterson's 61C

Fully Associative Cache (3/3)

- Benefit of Fully Assoc Cache
  - No Conflict Misses (since data can go anywhere)

- Drawbacks of Fully Assoc Cache

  - Need hardware comparator for every single entry:

    - If we have a 64KB of data in cache with 4B entries, we need 16K comparators: very expensive
- Small fully associative cache may be feasible

Third Type of Cache Miss

- Capacity Misses
  - miss that occurs because the cache has a limited size
  - miss that would not occur if we increase the size of the cache
  - sketchy definition, so just get the general idea
- This is the primary type of miss for Fully Associate caches.

N-Way Set Associative Cache (1/4)

- Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: points us to the correct "row" (called a set in this case)
- So what's the difference?
  - each set contains multiple blocks
  - once we've found correct set, must compare with all tags in that set to find our data

N-Way Set Associative Cache (2/4)

- Summary:
  - cache is direct-mapped with respect to sets
  - each set is fully associative
  - If we have T blocks total, then we basically have an T/N direct-mapped cache, where at each index we find a fully associative N block cache. Each has its own valid bit and data.

N-Way Set Associative Cache (3/4)

- Given memory address:
  - Find correct set using Index value.
  - Compare Tag with all Tag values in the determined set.
  - If a match occurs, it's a hit, otherwise a miss.
  - Finally, use the offset field as usual to find the desired data within the desired block.

slides adapted from Patterson's 61C

N-Way Set Associative Cache (4/4)

- What's so great about this?
  - even a 2-way set associative cache avoids a lot of conflict misses
  - hardware cost isn't that bad: only need N comparators
- In fact, for a cache with M blocks,
  - it's Direct-Mapped if it's 1-way set associative (1 block per set)
  - it's Fully Associative if it's M-way set associative (M blocks per set)
  - so these two are just special cases of the more general set associative design

Block Replacement Policy (1/2)

- Direct-Mapped Cache: index completely specifies which position a block can go in on a miss

- N-Way Set Assoc (N > 1): index specifies a set, but block can occupy any position within the set on a miss

- Fully Associative: block can be written into any position (there is no index)
- Question: if we have the choice, where should we write an incoming block?

Block Replacement Policy (2/2)

- Solution!
- If there are any locations with valid bit off (empty), then usually write the new block into the first one.
- If all possible locations already have a valid block, we must use a replacement policy by which we determine which block gets “cached out” on a miss.

Block Replacement Policy: LRU

- LRU (Least Recently Used)
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality => recent past use implies likely future use: in fact, this is a very effective policy
  - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Block Replacement Example

- We have a 2-way set associative cache with a four word *total* capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):

  0, 2, 0, 1, 4, 0, 2, 3, 5, 4

  How many hits and how many misses will there for the LRU block replacement policy?

Block Replacement Example: LRU

- Addresses 0, 2, 0, 1, 4, 0, ...

0: miss, bring into set 0 (loc 0)

0 remainder 2 is 0, so set 0

2: miss, bring into set 0 (loc 1)

2 remainder 2 is 0, so set 0

0: hit

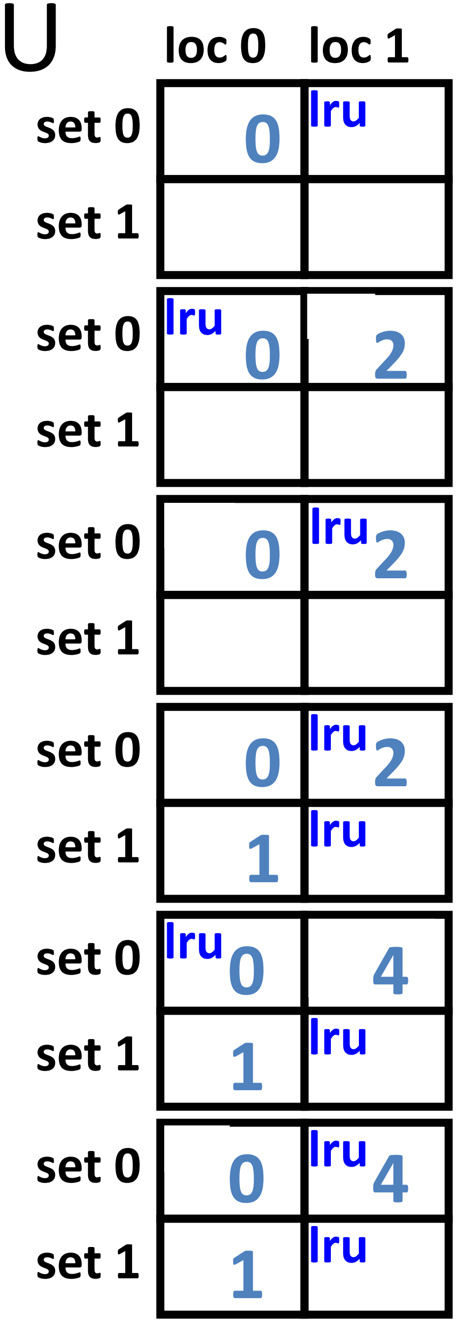
1: miss, bring into set 1 (loc 0)

1 remainder 2 is 1, so set 1

4: miss, bring into set 0 (loc 1, replace 2)

4 remainder 2 is 0, so set 0

0: hit



Ways to reduce miss rate

- Larger cache
  - – limited by cost and technology
  - – hit time of first level cache < cycle time
- More places in the cache to put each block of memory - associativity
  - – fully-associative
    - any block any line
  - – k-way set associated
    - k places for each block
    - direct map: k=1

Big Idea

- How do we chose between options of associativity, block size, replacement policy?
- Design against a performance model
  - Minimize: *Average Access Time*

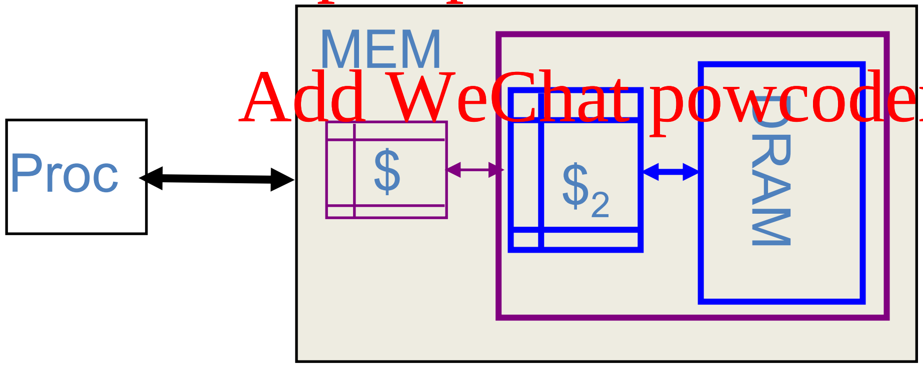
    = Hit Time + Miss Penalty x Miss Rate
  - influenced by technology and program behavior

Example

- Assume
  - Hit Time = 1 cycle
  - Miss rate = 5%
  - Miss penalty = 20 cycles
- Average memory access time = 1 + 0.05 x 20
  = 2 cycle

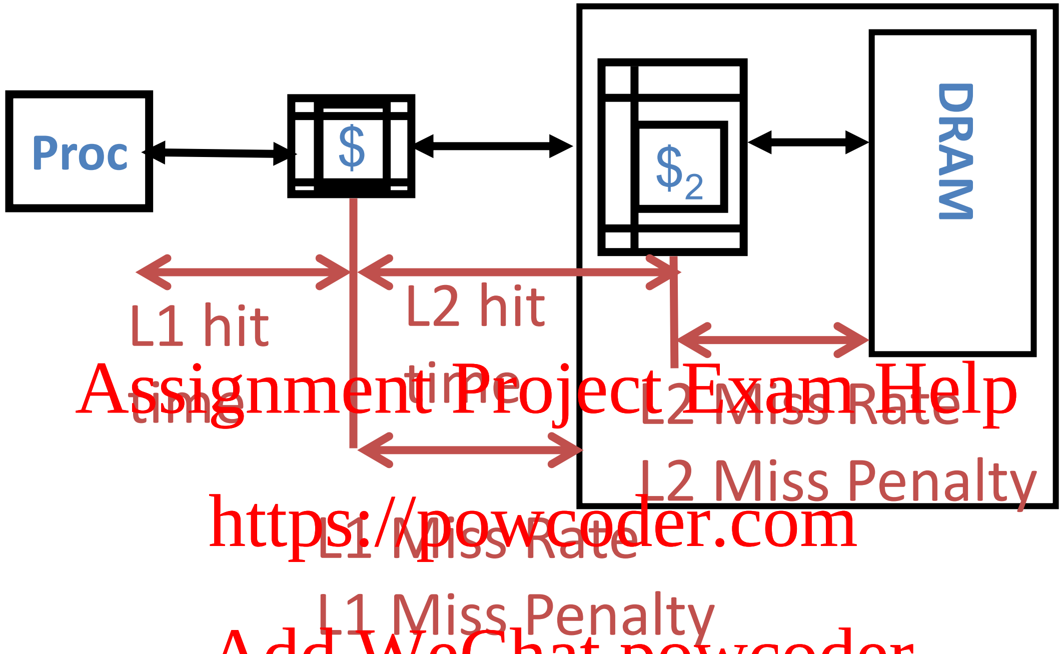
Improving Miss Penalty

- When caches first became popular, Miss Penalty ~ 10 processor clock cycles
- Today: 1000 MHz Processor (1 ns per clock cycle) and 100 ns to go to DRAM
  $\Rightarrow$ **100 processor clock cycles!**



Solution: another cache between memory and the processor cache: <u>Second Level (L2) Cache</u>

Analyzing Multi-level cache hierarchy



Avg Mem Access Time = L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty = L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time = L1 Hit Time +
L1 Miss Rate * (L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)

Typical Scale

- L1
  - size: tens of KB
  - hit time: complete in one clock cycle
  - miss rates: 1-5%
- L2
  - size: hundreds of KB
  - hit time: few clock cycles
  - miss rates: 10-20%
- L2 miss rate is fraction of L1 misses that also miss in L2
  - why so high?

Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 100 cycles
- Average memory access time = 1 + 0.05 x 100

  = 6 cycles

Example with L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = 100 cycles
- L1 miss penalty = 5 + 0.15 * 100 = 20
- Average memory access time = 1 + 0.05 x 20

  = 2 cycle

3x faster with L2 cache

What to do on a write hit?

- Write-through
  - update the word in cache block and corresponding word in memory

- Write-back
  - update word in cache block

  - allow memory word to be "stale"

  - *add 'dirty' bit to each line indicating that memory needs to be updated when block is replaced*
  - *OS flushes cache before I/O !!!*
- Performance trade-offs?

"And in conclusion…" (1/2)

- Caches are NOT mandatory:
  - Processor performs arithmetic
  - Memory stores data
  - Caches simply make data transfers go faster
- Each level of memory hierarchy is just a subset of next higher level
- Caches speed up due to temporal locality: store data used recently
- Block size > 1 word speeds up due to spatial locality: store words adjacent to the ones used recently

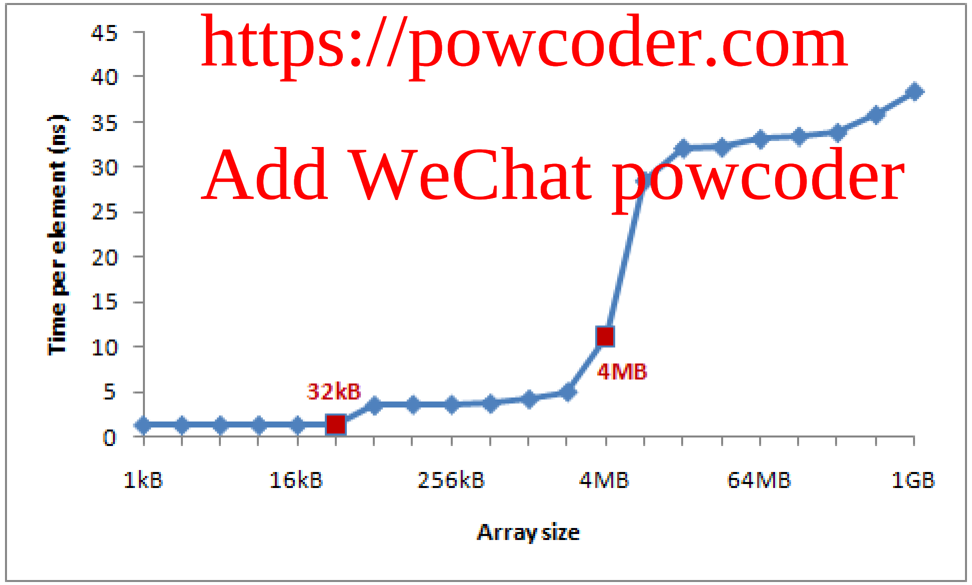
“And in conclusion…” (2/2)

- Cache design choices:
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - block replacement policy
  - 2nd level cache?
  - Write through v. write back?
- Use performance model to pick between choices, depending on programs, technology, budget, ...

A real example

- And additional reading (for fun):

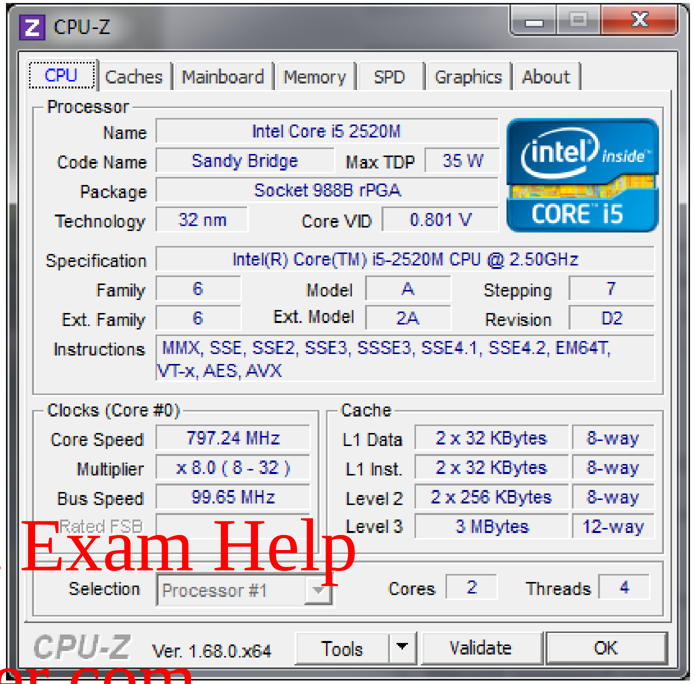
  http://igoro.com/archive/gallery-of-processor-cache-effects/

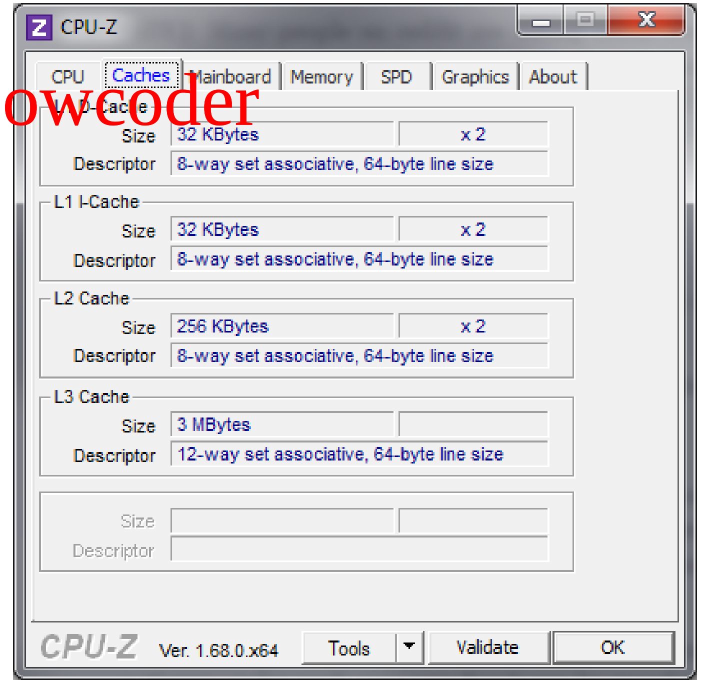


```java
package comp273;
/** Let us observe the effect of the cache */
public class CacheTest1 {
    public static void main( String[] args ) {
        int[] A = new int[128 * 1024 * 1024];
        double total;
        int N = 8;    // average together 8 samples

        // Loop 1
        total = 0;
        for ( int j = 0; j < N; j++ ) {
            double start = System.nanoTime();

            for (int i = 0; i < A.length; i++) A[i] *= 3;

            double stop = System.nanoTime();
            double elapsed = stop - start;
            total += elapsed;
        }
        double loop1Time = total / N;
        System.out.println( "average time loop 1 = " + loop1Time );

        // Loop 2
        total = 0;
        for ( int j = 0; j < N; j++ ) {
            double start = System.nanoTime();

            for (int i = 0; i < A.length; i+=32) A[i] *= 3;

            double stop = System.nanoTime();
            double elapsed = stop - start;
            total += elapsed;
        }
        double loop2Time = total / N;
        System.out.println( "average time loop 2 = " + loop2Time );
        System.out.println( "ratio is " + loop1Time / loop2Time +
                " but first loop does 32 times more work!!" );
    }
}
```

average time loop 1 = 126858657.625
average time loop 2 = 71715726.125
ratio is 1.7689098957721807
but first loop does 32 times more work!!

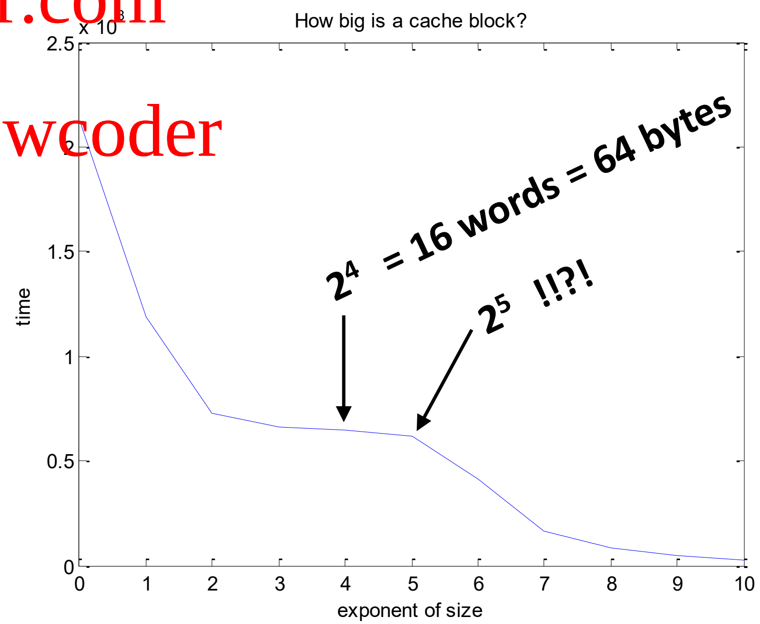
Loop 1 gets work done 18 times faster!





```java
package comp273;

/** Let us try to see how long a cache line is */
public class CacheTest2 {

    public static void main( String[] args ) {

        System.out.println("A=[");
        int[] A = new int[128 * 1024 * 1024];
        int K = 1;
        for ( int k = 0; k < 11; k++ ) {
            // try going through memory in increments K = 2^k
            long start = System.nanoTime();

            for (int i = 0; i < A.length; i += K) A[i] *= 3;

            long stop = System.nanoTime();
            System.out.println( k + " " + K + " " + (stop - start) );
            K = K*2;
        }
        System.out.println("];");
        System.out.println("plot(A(:,1),A(:,3));");
        System.out.println("title('How big is a cache block?');");
        System.out.println("ylabel('time');");
        System.out.println("xlabel('exponent of size');");
    }
}
```

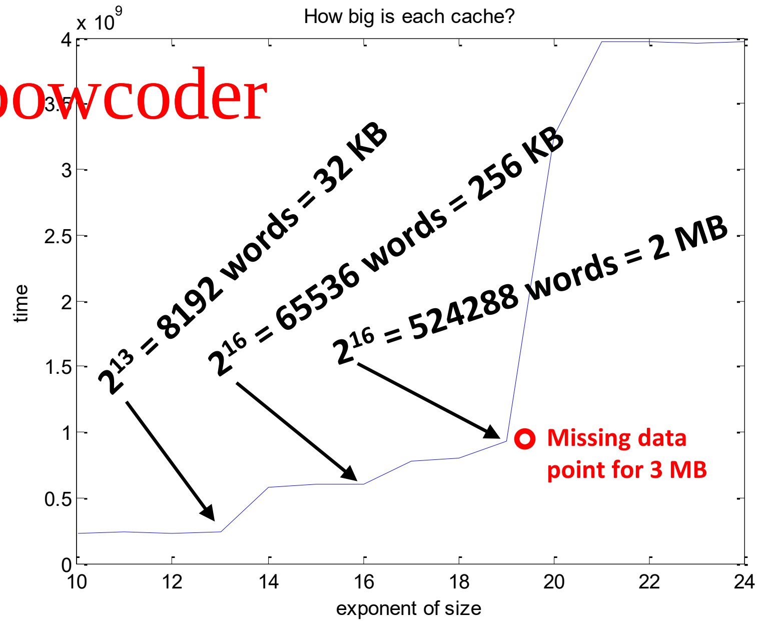
```
A=[
0 1 240591499
1 2 134307003
2 4 84736089
3 8 74437939
4 16 70215291
5 32 73695400
6 64 52077957
7 128 19758427
8 256 10488407
9 512 6369311
10 1024 3736937
];
plot(A(:,1),A(:,3));
title('How big is a cache block?');
ylabel('time');
xlabel('exponent of size');
```

slides adapted from Patterson's 61C

```java
package comp273;

/** Lets try to figure out the cache sizes... */
public class CacheTest3 {

    public static void main( String[] args ) {
        int steps = 4 * 64 * 1024 * 1024; // Arbitrary number of steps

        System.out.println("B=[");
        int size = 1024; // start with 1K array
        for ( int j = 0; j < 15; j++ ) {
            int[] A = new int[size];
            long start = System.nanoTime();
            int lengthMod = A.length - 1;

            for (int i = 0; i < steps; i++) A[(i * 32) & lengthMod]++;

            long stop = System.nanoTime();
            System.out.println( j + " " + size + " " + (stop - start) );

            System.gc(); // garbage collect now!
            size *= 2;
        }
        System.out.println("];");
        System.out.println("plot(B(:,1)+10,B(:,3))");
        System.out.println("title('How big is each cache?');");
        System.out.println("ylabel('time');");
        System.out.println("xlabel('exponent of size');");
    }
}
```

```
B=[
0 1024 226172611
1 2048 236937569
2 4096 227578791
3 8192 240087707
4 16384 581669367
5 32768 602241011
6 65536 607694375
7 131072 776806172
8 262144 799580776
9 524288 924929650
10 1048576 3261174238
11 2097152 3971720257
12 4194304 3972356366
13 8388608 3954041924
14 16777216 3971577666
];
plot(B(:,1)+10,B(:,3))
title('How big is each cache?');
ylabel('time');
xlabel('exponent of size');
```

Review and More Information

- Sections 5.3 - 5.4 of textbook