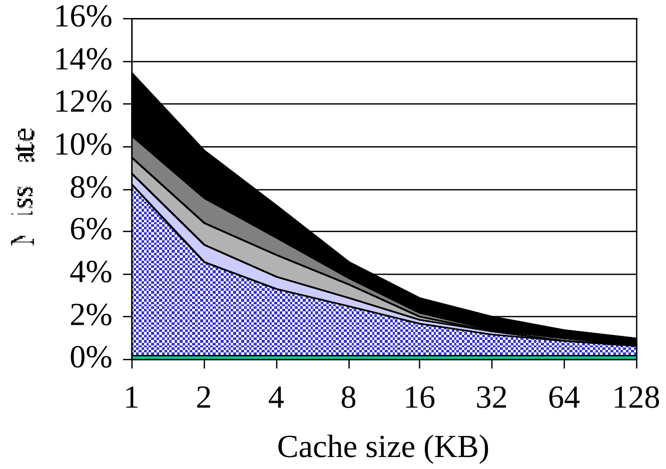
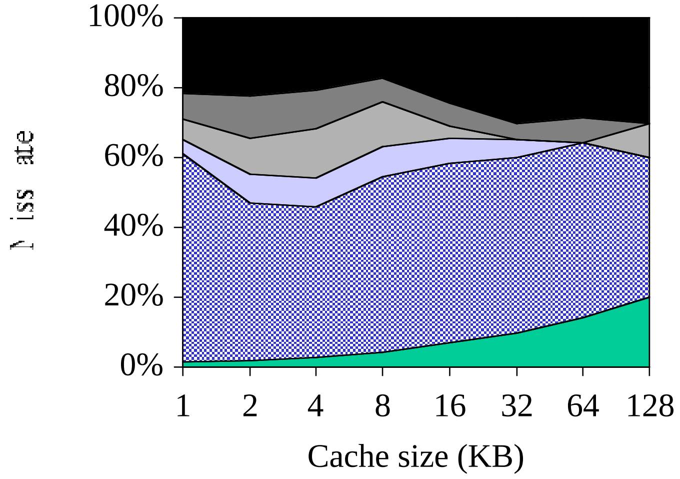
Improving cache performance

- The increasing speed gap between CPU and main memory has made the performance of the memory system increasingly important
- There are many distinct methods system architect use to reduce average memory access time
- These methods can be classified by whether they
  - Reduce the miss rate
  - Reduce the miss penalty
  - Reduce the time to hit in a cache
- Other methods may also increase capacity for a given cost...

Components of cache miss rate

- Three "C"s of cache misses
- Compulsory misses
  - First access to a block ***can't*** be in the cache
  - Occur regardless of cache size
- Capacity misses
  - Occur because cache isn't large enough to hold all blocks
  - Compulsory miss rate - the miss rate of a fully associative cache
- Conflict (collision) misses
  - The block can't be kept because the set is full
  - Difference between fully- and set- associative cache



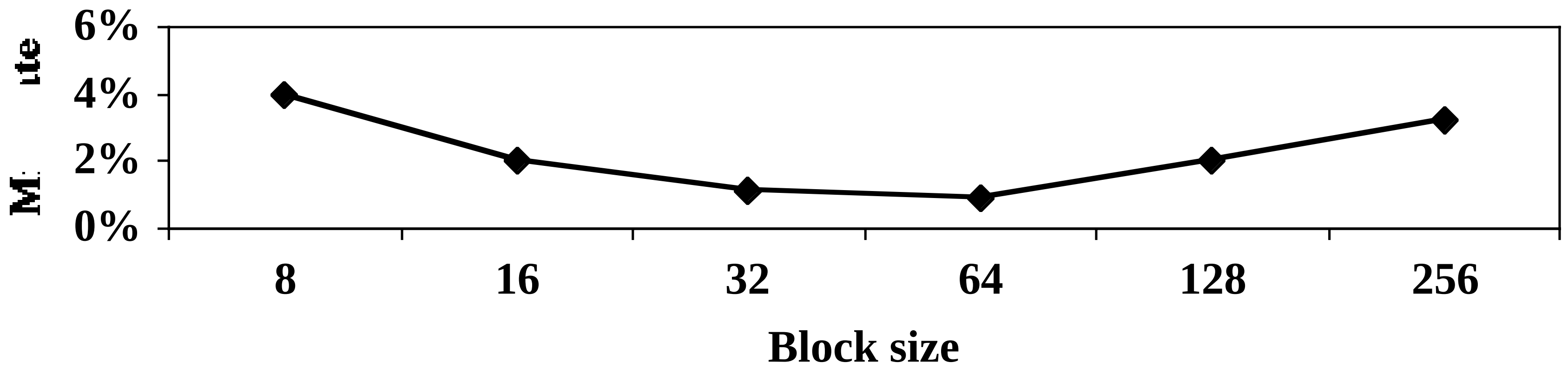


Reducing cache miss rate

- To reduce cache miss rate, we must eliminate some of the misses due to the three C's
  - ***Capacity*** misses can't be reduced much except by making the cache larger
  - ***Conflict*** misses and ***compulsory*** misses can be reduced in several ways
- Larger cache blocks
  - Decrease the compulsory miss rate by taking advantage of spatial locality
  - May increase the miss penalty by requiring more data to be fetched per miss
  - Likely to increase conflict misses since fewer blocks can be stored in the cache
  - May even increase capacity misses in small caches

Larger cache blocks

- The miss rate curve is U-shaped because
  - Small blocks have a higher miss rate
  - Large blocks have a higher miss penalty (even if miss rate is not too high)
- High latency, high bandwidth memory systems encourage large block sizes
  - The cache gets more bytes per miss for a small increase in miss penalty
  - 32-byte blocks are typical for 1-KB, 4-KB and 16-KB caches
  - 64-byte blocks are typical for larger caches
- Instruction caches tend to have larger blocks than data caches



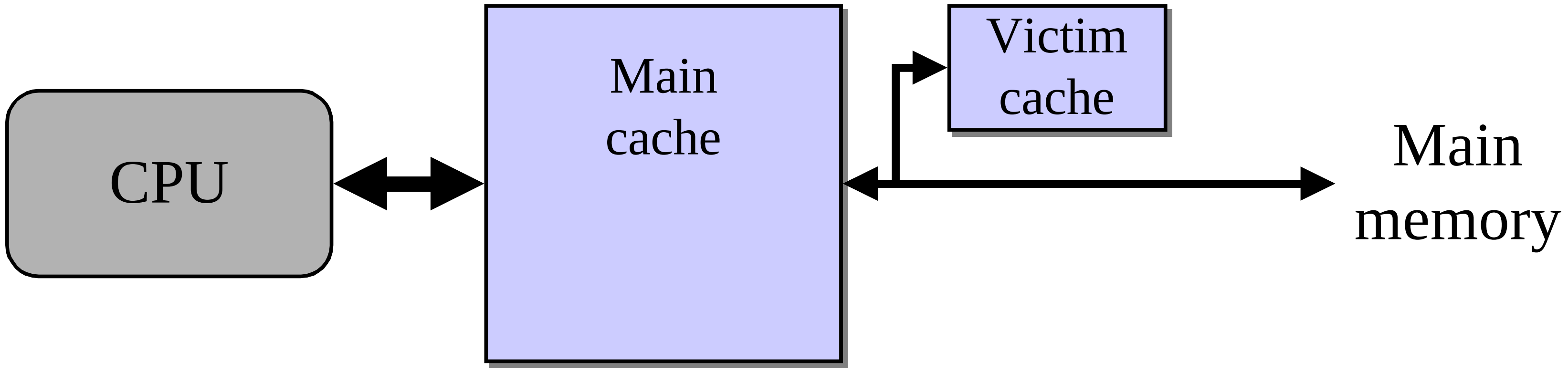
Higher associativity

- *Conflict* misses can be a problem for caches with low associativity (especially direct-mapped)
- Miss rates generally follow the 2:1 cache rule of thumb
  - A direct-mapped cache of size N has the same miss rate as a 2-way set-associative cache of size N/2
- However, higher associativity means
  - More hardware
  - Often, longer cycle times (increased hit time)
  - Possibly, more capacity misses
- 8-way set-associative caches are the maximum used today, and most systems use 4-way or less

  ⇒ Higher hit rate is offset by the slower clock cycle time

Victim caches

- Victim cache => small cache (often fully associative) that holds a few of the most recently replaced blocks or victims from the main cache
  - Reduces conflict misses and (secondarily) capacity misses
  - Particularly effective for small, direct-mapped data caches
    - A 4 entry victim cache handled from 20% to 95% of the conflict misses from a 4KB direct-mapped data cache
- Check victim cache before main memory on a miss
  - Swap victim block & cache block if hit in victim cache



Pseudo-associative caches

- These caches use a technique similar to double hashing
  - On a miss, the cache searches a different set for the desired block
  - The second (pseudo) set to probe is usually found by inverting one or more bits in the original set index
- Two separate searches are conducted on a miss
  - The first search proceeds as it would for a direct-mapped cache: since there's no associative h/w, hit time is fast if block found on first probe
  - The second probe takes some time (usually an extra cycle or two), but it's a lot faster than going to main memory
    - The secondary block can be swapped with the primary block on a "slow hit"
- This method reduces the effect of conflict misses
- Also improves miss rates without affecting the clock rate

Hardware prefetch

- Prefetching is the act of getting data from memory before it is actually needed by the CPU
  - Typically, the cache requests the next consecutive block to be fetched with a requested block, hopefully avoiding a subsequent miss
  - Compulsory misses reduced by retrieving the data before it is requested
  - Other misses may *increase* => useful blocks replaced in the cache
- Many caches hold prefetched blocks in a special buffer until they are actually needed
  - This buffer is faster than main memory but only has a limited capacity
- Prefetching also uses main memory bandwidth
  - Prefetching works well if the data is actually used
  - However, it can adversely affect performance if the data is rarely used and the accesses interfere with 'demand misses'

Compiler-controlled prefetch

- Some CPUs include prefetching instructions
  - Instructions request that data be moved into either a register or cache
  - These special instructions can either be faulting or non-faulting
    - Non-faulting instructions do nothing (no-op) if the memory access would cause an exception
- Prefetching shouldn't interfere with normal CPU operations
  - The cache must be *nonblocking* (also called *lockup-free*)
  - This allows the CPU to overlap execution with the prefetching of data
- Improves prefetch "hit" rates over hardware prefetch
  - However ,it does so at the expense of executing more instructions
  - Thus, the compiler tends to concentrate on prefetching data that are likely to be cache misses anyway
  - Loops are key targets since they operate over large data spaces and their data accesses can be inferred from the loop index in advance

Compiler optimizations

- This method requires ***no*** hardware modifications
  - However, it's often the most efficient way to reduce cache misses
  - The improvement results from better code and data organizations
- Examples
  - Code can be rearranged to avoid conflicts in a direct-mapped cache
  - Accesses to arrays can be reordered to operate on blocks of data rather than processing rows of the array
  - Arrays can be resized to avoid cache conflicts for related elements

Compiler optimization: merging arrays

- Combine two separate arrays (that might conflict for a single block in the cache) into a single interleaved array
- Bring together corresponding elements in both arrays, which are likely to be referenced together
- Reorganizing and fetching them at the same time can reduce misses
- This technique reduces misses by improving spatial locality

```
int value[SIZE];
int key[SIZE];
```

```
struct merged {
  int value;
  int key;
};
struct merge m[SIZE];
```

Compiler optimization: loop interchange

- Switch the order in which loops execute
  - Misses can be reduced due to improvements in spatial locality
- Example
  - These loops cause a miss on each memory access because of the long stride given by index *j* in the inner loop
  - Switching the order of the loops changes the stride to 1 => the elements are accessed in sequential order.

```
for (i=0; i<100; i++) {
  for (j=0; j<100; j++) {
    a[j][i] = a[j][i]*2;
  }
}
```

```
for (j=0; j<100; j++) {
  for (i=0; i<100; i++) {
    a[j][i] = a[j][i]*2;
  }
}
```

Compiler optimization: loop fusion

- Many programs have separate loops that operate on the same data
- Combining these loops allows a program to take advantage of **temporal locality** by grouping operations on the same (cached) data together
  - Caching may work even better because of sequential access between elements
  - Caching can hold results from previous iterations of the loop...

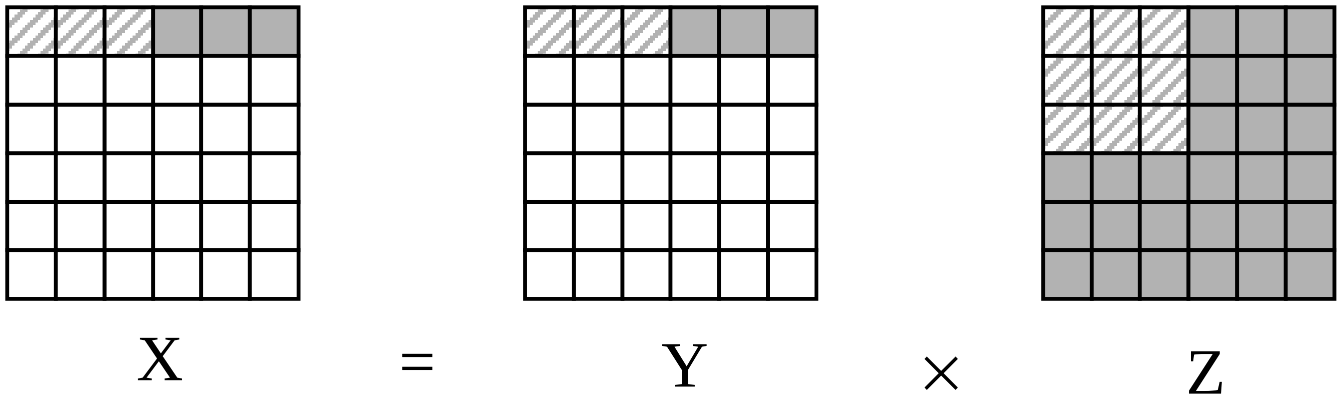
```
for (j=0; j<100; j++) {
  x[j] = x[j] + y[j];
}

for (j=0; j<100; j++) {
  y[j] = y[j] + x[j-1];
}
```

$\Rightarrow$

```
for (j=0; j<100; j++) {
  x[j] = x[j] + y[j];
  y[j] = y[j] + x[j-1];
}
```

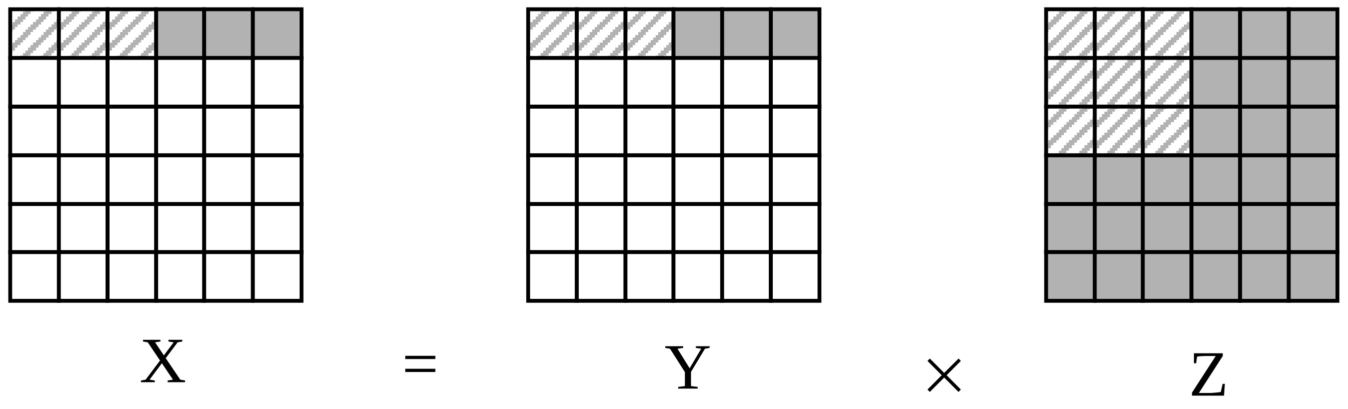
Compiler optimization: blocking

- Previous compiler optimizations work well on array accesses that occur along one dimension only
  - Loops that access both rows and columns can use other techniques
  - Unoptimized matrix multiplication => cache must hold the shaded areas
- Another technique: blocking
  - Capacity misses can occur for large matrices since it may not be possible to store all the elements of Z in the cache
  - Blocking operates on submatrices: reduces total memory words accessed by a factor of B (the blocking factor)



Compiler optimization: blocking

- Matrix multiplication is performed by multiplying the submatrices first
  - Matrix Y benefits from spatial locality
  - Matrix Z benefits from temporal locality
- This method is also used to reduce the number of blocks that must be transferred between disk and main memory
  - ⇒ The technique is effective for several levels of the hierarchy
- Given the increasing speed gap in processor speed and memory access times, these last two techniques will only increase in importance over time



Giving read misses priority

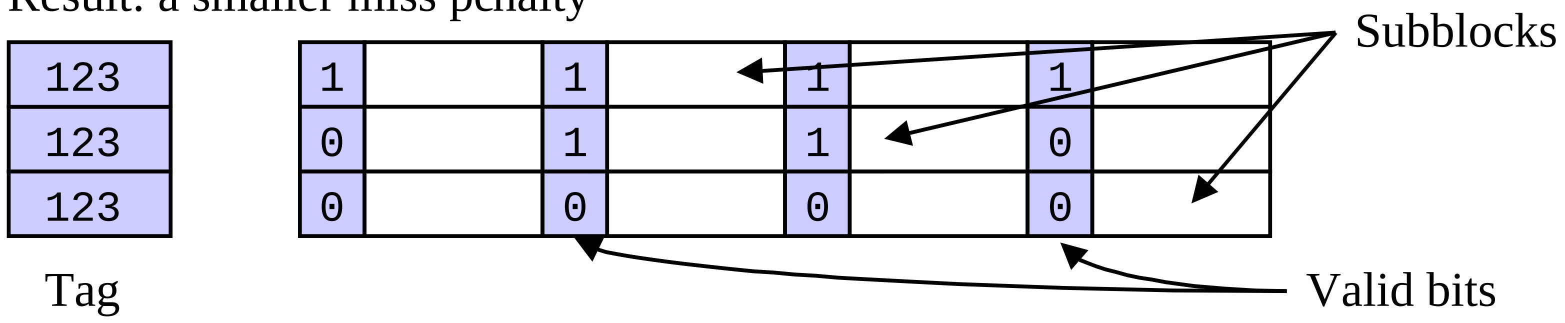
- If a system has a write buffer, delay writes to come after reads
- Problem: reads may request a value about to be written
- Solution 1: stall reads until the write buffer is empty
  - The write buffer in write-through is likely to have blocks queued up
  - Read miss penalty increases considerably
- Solution 2: check the write buffer for conflicts
  - In cases like this, the write buffer acts as a victim cache

```
SW 0(R3),R4
LW R11,4096(R3)
LW R12,0(R3)
```

If this is a direct-mapped 4KB cache, will R12 get the value from R4?

Using subblocks to reduce fetch time

- Tags can hurt performance by occupying too much space or by slowing down caches
  - Using large blocks reduces the amount of storage for tags (and makes them shorter), optimizing space on the chip
  - This may even reduce miss rate by reducing compulsory misses
  - However, the miss penalty for large blocks is high, since the entire block must be moved between the cache and memory
- Solution: divide each block into subblocks, each of which has a valid bit
  - Tag is valid for the entire block, but only a subblock needs to be read on a miss
  - A block is no longer the minimum unit transferred between cache and memory
  - Result: a smaller miss penalty



Early restart & critical word first

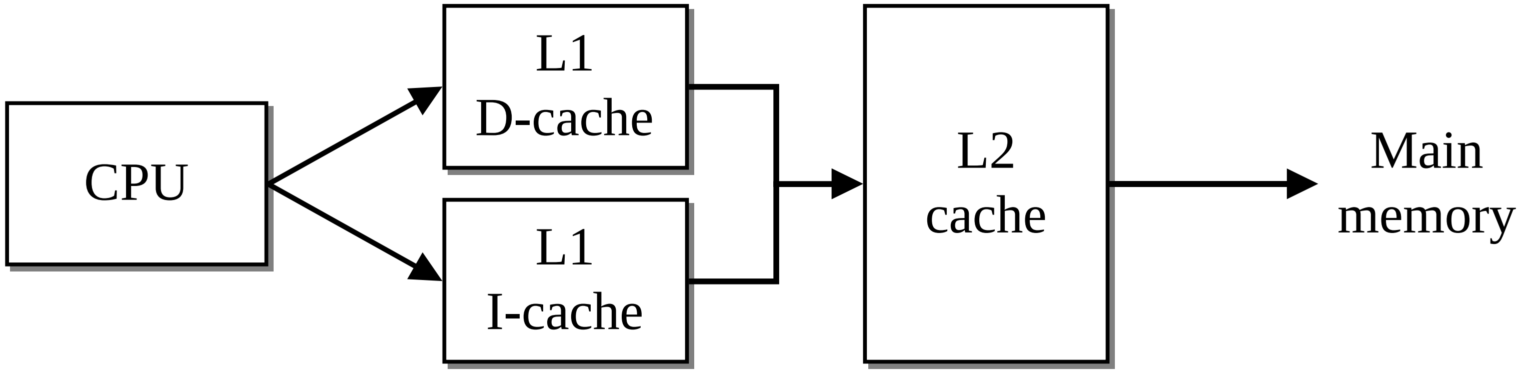
- Goal: optimize the order in which the words of a block are fetched and when the desired word is delivered to the CPU
- This strategy requires no extra hardware!
- Early restart
  - The CPU gets its data (and resumes execution) as soon as the desired word arrives in the cache
  - CPU doesn't wait for the rest of the block!
- Critical word first
  - Don't start the fetch of a block with the first word
  - Instead, fetch the requested word first and then fetch the rest afterwards
- Early restart & critical word first reduce the miss penalty
  - $\Rightarrow$ CPU can continue execution while most of the block is still being fetched

Non-blocking cache

- A nonblocking cache can allow the CPU to continue executing instructions after a data cache miss
  - Works well in conjunction with out-of-order execution
  - The cache continues to supply hits while processing read misses (hit under miss)
  - The instruction needing the missed data waits for the data to arrive
- Complex caches can even have multiple outstanding misses (miss under miss)
  - This greatly increases cache complexity
  - May be of relatively little benefit relative to the design complexity

Second level caches

- This method focuses on the interface between the cache and main memory
- Add a second-level cache between main memory and a small, fast first-level cache
  - This helps satisfy the desire to make the cache fast and large
  - The second-level cache allows
    - A small first-level cache that fits on the chip with the CPU
    - A first-level cache fast enough to handle hits in 1-2 CPU cycles
  - Hits for many memory accesses that would go to main memory are handled in the L2 cache, lessening the effective miss penalty



Performance of multi-level caches

- Calculating performance of a two-level cache is done similarly to that of a one-level cache
  - Miss penalty for level 1 is calculated using the hit time, miss rate, and miss penalty for the level 2 cache
- For two level caches, there are two miss rates
  - Global miss rate: the number of misses in the cache divided by the total memory accesses generated by the CPU (Miss rate_{L1}*Miss rate_{L2})
  - Local miss rate: the number of misses in the cache divided by the total memory accesses to *this* cache (Miss rate_{L2} for the 2nd-level cache)
    - The local miss rate for L2 is high because it's only getting the misses from the L1 cache (instead of all memory accesses)
- Global miss rate is often a more useful measure => fraction of the memory accesses that must go all the way to memory

$$Avg\ \ memory\ \ access\ \ time = Hit\ \ time_{L1} \times Miss\ \ penalty_{L1}$$

$$Miss\ \ penalty_{L1} = Hit\ \ time_{L2} + Miss\ \ rate_{L2} \times Miss\ \ penalty_{L2}$$

Desirable characteristics for an L2 cache

- Larger than the L1 cache
  - A miss in L1 is unlikely to be a hit in L2 unless L2 is much larger
  - The local hit rate for L2 depends on the size ratio between L1 and L2!
- Higher associativity
  - The main reason for low associativity was fast, small caches
  - The L2 cache need be neither, and will benefit from the higher hit rate that more blocks per set provides
- Larger block size
  - This reduces compulsory misses that are fetched from main memory
  - Since the L2 cache is large, the effect of increasing conflict misses (as is true for a smaller cache) is minimal

Multilevel inclusion

- If all of the data in the L1 cache is also in the L2 cache, the L2 cache has the multilevel ***inclusion*** property
  – Most caches enforce this property since it is easier to deal with cache consistency
  – Consistency between I/O and caches (and between caches in a multiprocessor) can be determined by checking second-level cache

Multilevel cache design

- Design of L1 and L2 caches: although they can be designed separately, it is helpful to know if there’s an L2 cache
  - Write-through in L1 is much more effective if there is an L2 writeback cache to buffer repeated writes
  - A direct-mapped L1 cache works well if the L2 cache satisfies most of the conflict misses
- Multilevel cache design summary
  - In general, cache design trades fast hits for few misses
  - For an L1 cache, fast hits are more important
  - For L2, there are many fewer hits, so fewer misses becomes more important
- Thus, larger caches with higher associativity and larger blocks are beneficial for L2 caches