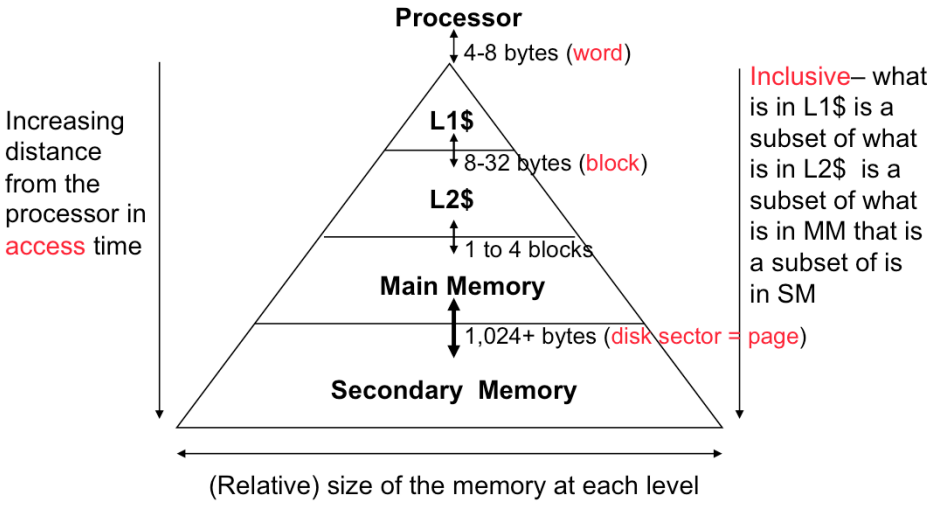
Cache Introduction Review

[Adapted from Mary Jane Irwin for
Computer Organization and Design,
Patterson & Hennessy, © 2005, UCB]

Other handouts
To handout next time

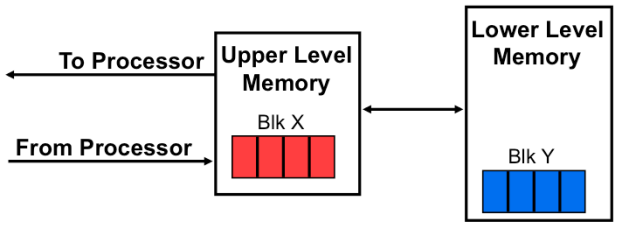
Review: The Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology



The Memory Hierarchy: Why Does it Work?

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



How does the memory hierarchy work? Well it is rather simple, at least in principle.

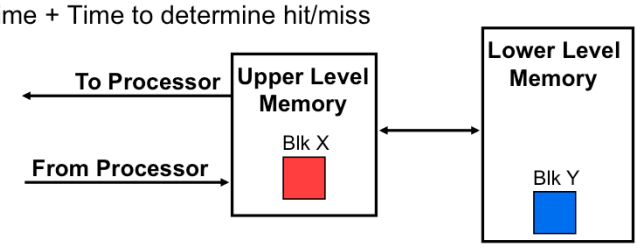
In order to take advantage of the temporal locality, that is the locality in time, the memory hierarchy will keep those more recently accessed data items closer to the processor because chances are (points to the principle), the processor will access them again soon.

In order to take advantage of the spatial locality, not ONLY do we move the item that has just been accessed to the upper level, but we ALSO move the data items that are adjacent to it.

+1 = 15 min. (X:55)

The Memory Hierarchy: Terminology

- Hit: data is in some block in the upper level (Blk X)
  - Hit Rate: the fraction of memory accesses found in the upper level
  - Hit Time: Time to access the upper level which consists of
    RAM access time + Time to determine hit/miss



- Miss: data is not in the upper level so needs to be retrieved from a block in the lower level (Blk Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level
    + Time to deliver the block to the processor
  - Hit Time << Miss Penalty

A HIT is when the data the processor wants to access is found in the upper level (Blk X).

The fraction of the memory access that are HIT is defined as HIT rate.

HIT Time is the time to access the Upper Level where the data is found (X). It consists of:

(a) Time to access this level.

(b) AND the time to determine if this is a Hit or Miss.

If the data the processor wants cannot be found in the Upper level. Then we have a miss and we need to retrieve the data (Blk Y) from the lower level.

By definition (definition of Hit: Fraction), the miss rate is just 1 minus the hit rate.

This miss penalty also consists of two parts:

(a) The time it takes to replace a block (Blk Y to BlkX) in the upper level.

(b) And then the time it takes to deliver this new block to the processor.

It is very important that your Hit Time to be much much smaller than your miss penalty. Otherwise, there will be no reason to build a memory hierarchy.

How is the Hierarchy Managed?

- registers ↔ memory
  - by compiler (programmer?)
- cache ↔ main memory
  - by the cache controller hardware
- main memory ↔ disks
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by the programmer (files)

TLB: http://en.wikipedia.org/wiki/Translation_Lookaside_Buffer

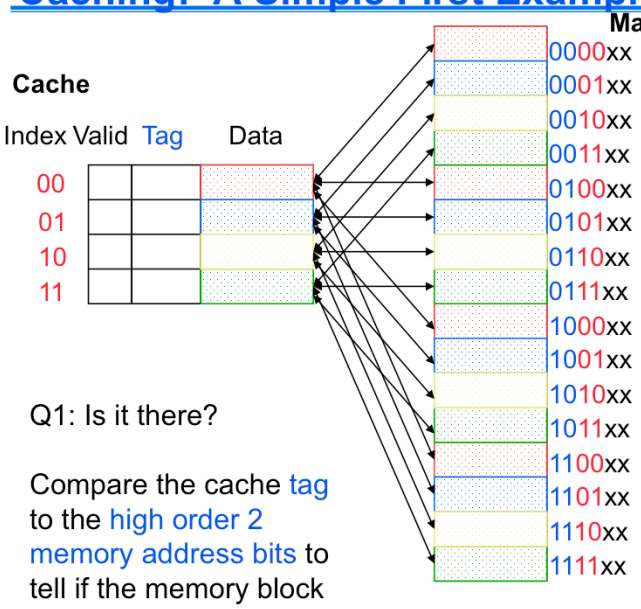
Cache

- Two questions to answer (in hardware):
  - Q1: How do we know if a data item is in the cache?
  - Q2: If it is, how do we find it?

- Direct mapped
  - For each item of data at the lower level, there is exactly one location in the cache where it might be - so lots of items at the lower level must share locations in the upper level
  - Address mapping:

    (block address) modulo (# of blocks in the cache)
  - First consider block sizes of one word

Caching: A Simple First Example



Two low order bits define the byte in the word (32b words)

Q1: Is it there?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 2 low order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

(block address) modulo (# of blocks in the cache)

For lecture

Valid bit indicates whether an entry contains valid information – if the bit is not set, there cannot be a match for this block

TAG is bits not used for cache address

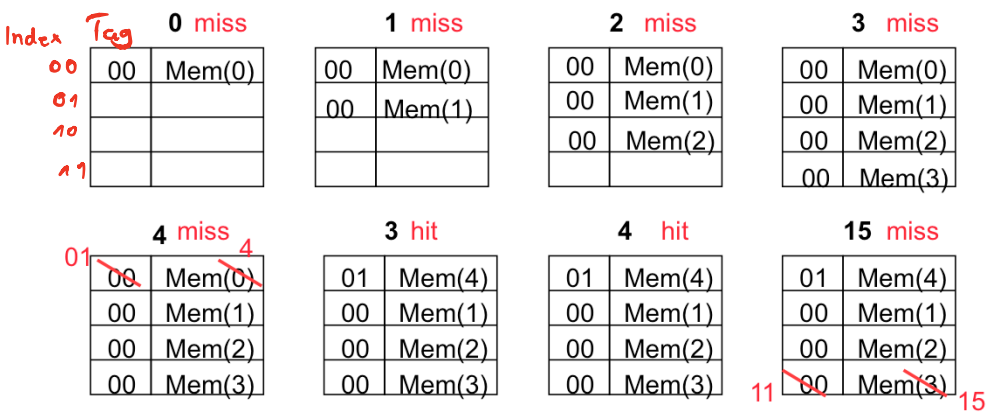
VALID BIT to signal cache data invalid

Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

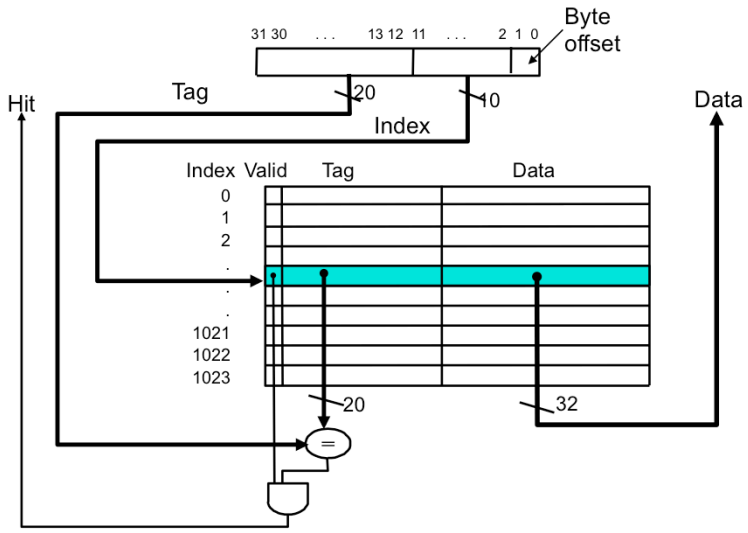


- 8 requests, 6 misses

For lecture

MIPS Direct Mapped Cache Example

- One word/block, cache size = 1K words



What kind of locality are we taking advantage of?

Let's use a specific example with realistic numbers: assume we have a 1 K word (4Kbyte) direct mapped cache with block size equals to 4 bytes (1 word).

In other words, each block associated with the cache tag will have 4 bytes in it (Row 1).

With Block Size equals to 4 bytes, the 2 least significant bits of the address will be used as byte select within the cache block.

Since the cache size is 1K word, the upper 32 minus 10+2 bits, or 20 bits of the address will be stored as cache tag.

The rest of the (10) address bits in the middle, that is bit 2 through 11, will be used as Cache Index to select the proper cache entry

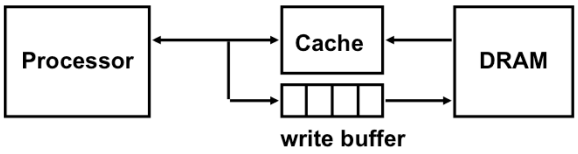
Temporal!

Handling Cache Hits

- Read hits (I$ and D$)
  - this is what we want!

- Write hits (D$ only)
  - allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache contents to the next level in the memory hierarchy when that cache block is "evicted")
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted
  - require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through) so don't need a dirty bit
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer, so only have to stall if the write buffer is full

Write Buffer for Write-Through Caching



- Write buffer between the cache and main memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: writes contents of the write buffer to memory
- The write buffer is just a FIFO
  - Typical number of entries: 4
  - Works fine if store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare
  - When the store frequency (w.r.t. time) → 1 / DRAM write cycle leading to write buffer saturation
    - E.g. use a write-back cache

We really didn't write to the memory directly. We are writing to a write buffer. Once the data is written into the write buffer and assuming a cache hit, the CPU is done with the write. The memory controller will then move the write buffer's contents to the real memory behind the scene.

The write buffer works as long as the frequency of store is not too high. Notice here, I am referring to the frequency with respect to time, not with respect to number of instructions.

Remember the DRAM cycle time we talked about last time. It sets the upper limit on how frequent you can write to the main memory.

If the store are too close together or the CPU time is so much faster than the DRAM cycle time, you can end up overflowing the write buffer and the CPU must stop and wait.

A Memory System designer's nightmare is when the Store frequency with respect to time approaches 1 over the DRAM Write Cycle Time. We called this Write Buffer Saturation. In that case, it does NOT matter how big you make the write buffer, the write buffer will still overflow because you simply feeding things in it faster than you can empty it. This is called Write Buffer Saturation and I have seen this happened before in simulation and when that happens your processor will be running at DRAM cycle time--very very slow.

The first solution for write buffer saturation is to get rid of this write buffer and replace this write through cache with a write back cache.

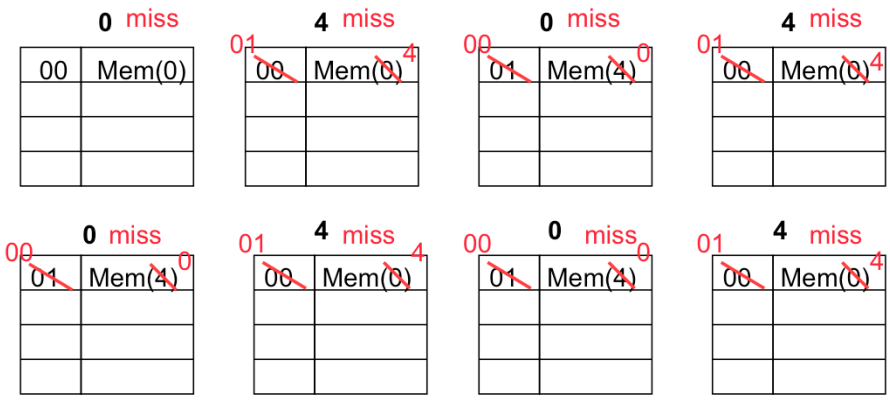
Another solution is to install the 2nd level cache between the write buffer and memory and makes the 2nd level write back.

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4



- 8 requests, 8 misses

- Ping pong effect due to conflict misses - two memory locations that map into the same cache block

For class handout

Sources of Cache Misses

- Compulsory (cold start or process migration, first reference):
  - First access to a block, “cold” fact of life, not a whole lot you can do about it
  - If you are going to run “millions” of instruction, compulsory misses are insignificant
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size

(Capacity miss) That is the cache misses are due to the fact that the cache is simply not large enough to contain all the blocks that are accessed by the program.

The solution to reduce the Capacity miss rate is simple: increase the cache size.

Here is a summary of other types of cache miss we talked about.

First is the Compulsory misses. These are the misses that we cannot avoid. They are caused when we first start the program.

Then we talked about the conflict misses. They are misses caused by multiple memory locations being mapped to the same cache location.

There are two solutions to reduce conflict misses. The first one is, once again, increase the cache size. The second one is to increase the associativity.

For example, say using a 2-way set associative cache instead of directed mapped cache.

But keep in mind that cache miss rate is only one part of the equation. You also have to worry about cache access time and miss penalty. Do NOT optimize miss rate alone.

Finally, there is another source of cache miss we will not cover today. Those are referred to as invalidation misses caused by another process, such as IO , update the main memory so you have to flush the cache to avoid inconsistency between memory and cache.

+2 = 43 min. (Y:23)

Handling Cache Misses

- Read misses (I$ and D$)
  - stall the entire pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- Write misses (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume

  or (normally used in write-through caches)

  2. Write allocate – just write the word into the cache updating both the tag and data

  or (normally used in write-through caches with a write buffer)

  3. No-write allocate – skip the cache write and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer is not full; must invalidate the cache block since it will be inconsistent (now holding stale data)

Let's look at our 1KB direct mapped cache again.

Assume we do a 16-bit write to memory location 0x000000 and causes a cache miss in our 1KB direct mapped cache that has 32-byte block select.

After we write the cache tag into the cache and write the 16-bit data into Byte 0 and Byte 1, do we have to read the rest of the block (Byte 2, 3, ... Byte 31) from memory?

If we do read the rest of the block in, it is called write allocate. But stop and think for a second. Is it really necessary to bring in the rest of the block on a write miss?

True, the principle of spatial locality implies that we are likely to access them soon.

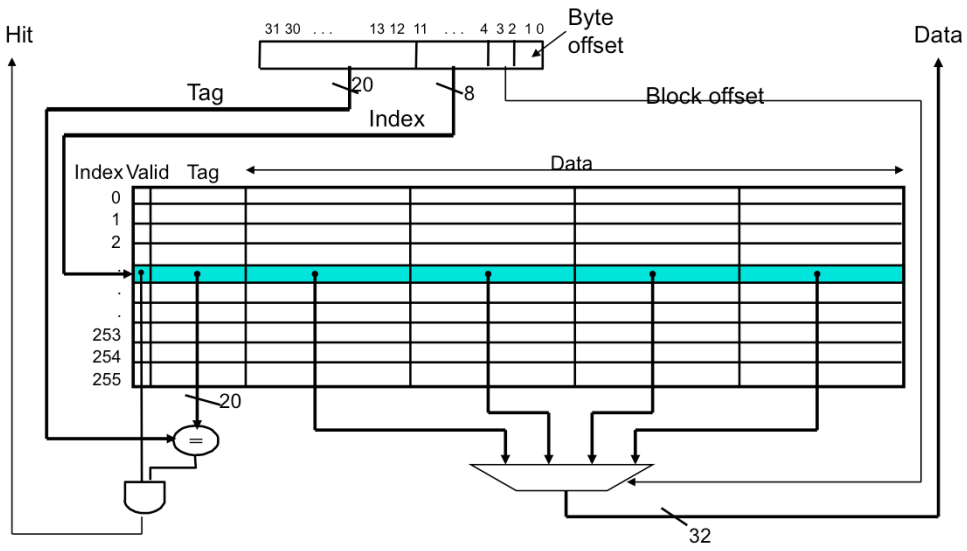
But the type of access we are going to do is likely to be another write.

So even if we do read in the data, we may end up overwriting them anyway so it is a common practice to NOT read in the rest of the block on a write miss.

If you don't bring in the rest of the block, or use the more technical term, Write Not Allocate, you better have some way to tell the processor the rest of the block is no longer valid.

Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



What kind of locality are we taking advantage of?

to take advantage for spatial locality want a cache block that is larger than word in size.

Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

~~00~~ 01	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

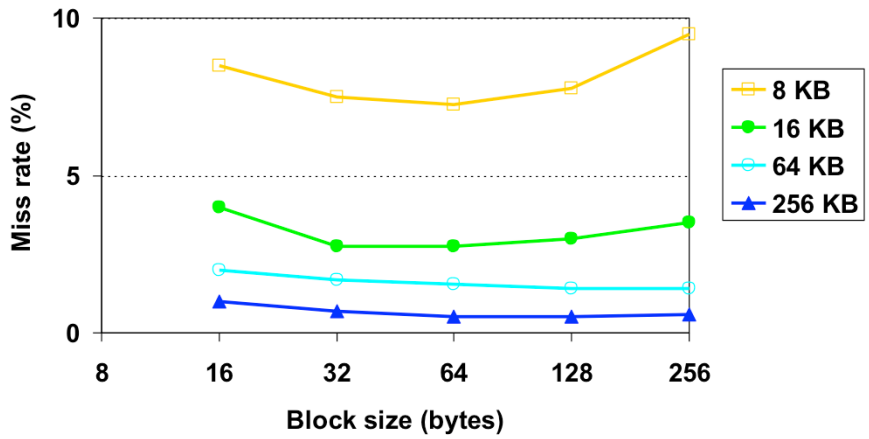
15 miss

11 01	Mem(5)	Mem(4)
~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

- 8 requests, 4 misses

For lecture

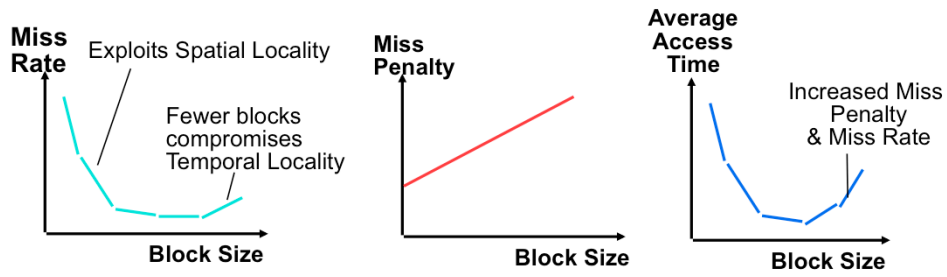
Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Block Size Tradeoff

- Larger block sizes take advantage of spatial locality but
  - If the block size is too big relative to the cache size, the miss rate will go up
  - Larger block size means larger miss penalty
    - Latency to first word in block + transfer time for remaining words



- In general, Average Memory Access Time
  = Hit Time + Miss Penalty x Miss Rate

As I said earlier, block size is a tradeoff. In general, larger block size will reduce the miss rate because it takes advantage of spatial locality.

But remember, miss rate is NOT the only cache performance metric. You also have to worry about miss penalty.

As you increase the block size, your miss penalty will go up because as the block gets larger, it will take you longer to fill up the block.

Even if you look at miss rate by itself, which you should NOT, bigger block size does not always win. As you increase the block size, assuming keeping cache size constant, your miss rate will drop off rapidly at the beginning due to spatial locality.

However, once you pass certain point, your miss rate actually goes up.

As a result of these two curves, the Average Access Time (point to equation), which is really the more important performance metric than the miss rate, will go down initially because the miss rate is dropping much faster than the increase in miss penalty.

But eventually, as you keep on increasing the block size, the average access time can go up rapidly because not only is the miss penalty is increasing, the miss rate is increasing as well.

Multiword Block Considerations

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows
    - Early restart – datapath resumes execution as soon as the requested word of the block is returned
    - Requested word first – requested word is transferred from the memory to the cache (and datapath) first
  - Nonblocking cache – allows the datapath to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$)
  - Can't use write allocate or will end up with a "garbled" block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block), so must fetch the block from memory first and pay the stall time

Early restart works best for instruction caches (since it works best for sequential accesses) – if the memory system can deliver a word every clock cycle, it can return words just in time. But if the processor needs another word from a different block before the previous transfer is complete, then the processor will have to stall until the memory is no longer busy.

Unless you have a nonblocking cache that come in two flavors

Hit under miss – allow additional cache hits during a miss with the goal of hiding some of the miss latency

Miss under miss – allow multiple outstanding cache misses (need a high bandwidth memory system to support it)

Cache Summary

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three major categories of cache misses:
  - Compulsory misses: sad facts of life. Example: cold start misses
  - Conflict misses: increase cache size and/or associativity
    Nightmare Scenario: ping pong effect!
  - Capacity misses: increase cache size
- Cache design space
  - total size, block size, associativity (replacement policy)
  - write-hit policy (write-through, write-back)
  - write-miss policy (write allocate, write buffers)

Let's summarize today's lecture. I know you have heard this many times and many ways but it is still worth repeating. Memory hierarchy works because of the Principle of Locality which says a program will access a relatively small portion of the address space at any instant of time. There are two types of locality: temporal locality, or locality in time and spatial locality, or locality in space.

So far, we have covered three major categories of cache misses.

Compulsory misses are cache misses due to cold start. You cannot avoid them but if you are going to run billions of instructions anyway, compulsory misses usually don't bother you.

Conflict misses are misses caused by multiple memory location being mapped to the same cache location. The nightmare scenario is the ping pong effect when a block is read into the cache but before we have a chance to use it, it was immediately forced out by another conflict miss. You can reduce Conflict misses by either increase the cache size or increase the associativity, or both.

Finally, Capacity misses occurs when the cache is not big enough to contains all the cache blocks required by the program. You can reduce this miss rate by making the cache larger.

There are two write policy as far as cache write is concerned. Write through requires a write buffer and a nightmare scenario is when the store occurs so frequent that you saturates your write buffer.

The second write polity is write back. In this case, you only write to the cache and only when the cache block is being replaced do you write the cache block back to memory.

No fancy replacement policy is needed for the direct mapped cache. That is what caused direct mapped cache trouble to begin with – only one place to go in the cache causing conflict misses.