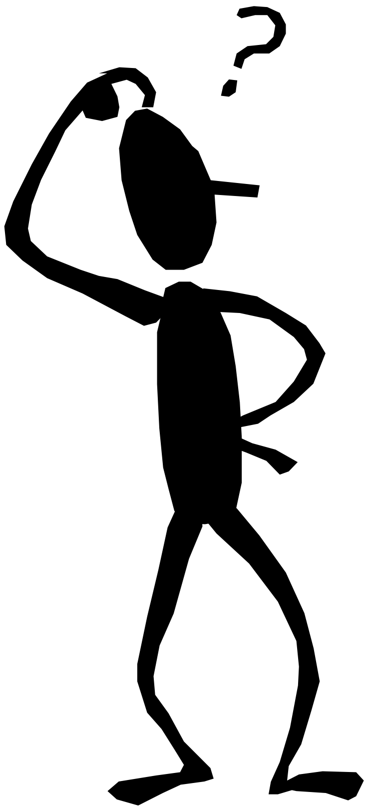
Four important questions



1. Where does the new block go?
2. Is data at address *addr* in the cache or not?
3. If the cache is full, which block to eliminate?
4. What about *writes*?

- Previous lectures answered the first 3. Today, we consider the 4th.

Writing to a cache

- Writing to a cache raises several additional issues.
- First, let’s assume that the address we want to write to is already loaded in the cache. We’ll assume a simple direct-mapped cache.

Index	V	Tag	Data
...			
110	1	11010	42803
...			

Address	Data
...	
1101 0110	42803
...	

- If we write a new value to that address, we can store the new data in the cache, and avoid an expensive main memory access.

Mem[1101 0110] = 21763

Index	V	Tag	Data
...			
110	1	11010	21763
...			

Address	Data
...	
1101 0110	42803
...	

Inconsistent memory

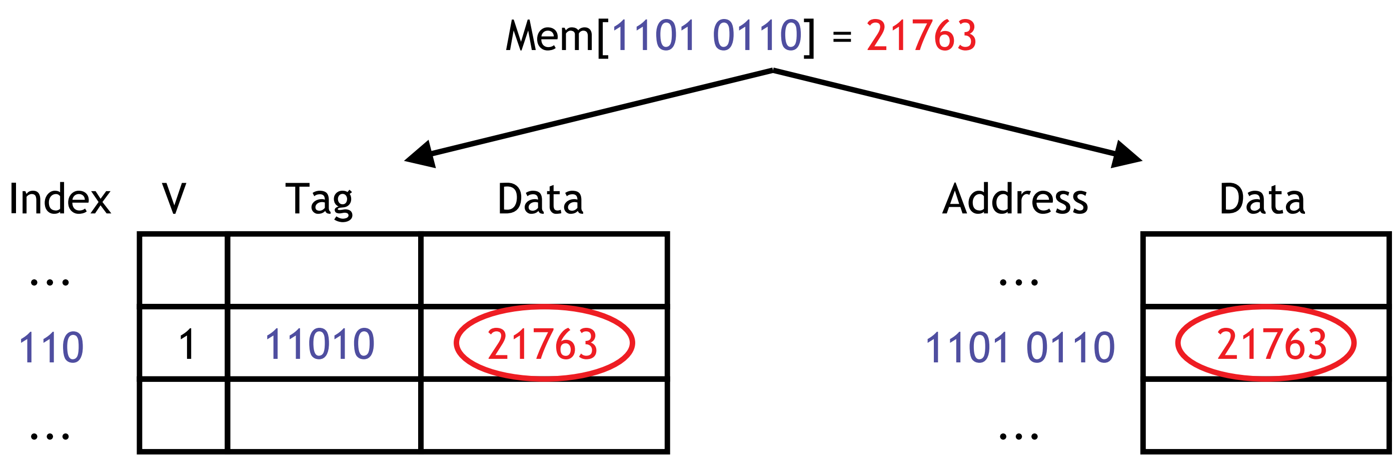
- But now the cache and memory contain different, inconsistent data!
- How can we ensure that subsequent loads will return the right value?
- This is also problematic if other devices are sharing the main memory, as in a multiprocessor system.

Index	V	Tag	Data
...			
110	1	11010	21763
...			

Address	Data
...	
1101 0110	42803
...	

Write-through caches

- A write-through cache solves the inconsistency problem by forcing all writes to update both the cache *and* the main memory.



- This is simple to implement and keeps the cache and memory consistent.
- Why is this not so good?

Write-back caches

- In a write-back cache, the memory is not updated until the cache block needs to be replaced (*e.g.*, when loading data into a full cache set).
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before.
  - The cache block is marked "dirty" to indicate this inconsistency

Mem[1101 0110] = 21763

Index	V	Dirty	Tag	Data
...				
110	1	1	11010	21763
...				

Address	Data
1000 1110	1225
1101 0110	42803
...	

- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data.

Finishing the write back

- We don’t need to store the new value back to main memory unless the cache block gets replaced.
- E.g. on a read from Mem[1000 1110], which maps to the same cache block, the modified cache contents will first be written to main memory.

Index	V	Dirty	Tag	Data
...				
110	1	1	11010	21763
...				

Address	Data
1000 1110	1225
1101 0110	21763
...	

- Only then can the cache block be replaced with data from address 142.

Index	V	Dirty	Tag	Data
...				
110	1	0	10001	1225
...				

Address	Data
1000 1110	1225
1101 0110	21763
...	

Write-back cache discussion

- Each block in a write-back cache needs a dirty bit to indicate whether or not it must be saved to main memory before being replaced—otherwise we might perform unnecessary write backs.
- Notice the penalty for the main memory access will not be applied until the execution of some *subsequent* instruction following the write.
  - In our example, the write to Mem[1101 0110] affected only the cache.
  - But the load from Mem[1000 1110] resulted in *two* memory accesses: one to save data to address 1101 0110, and one to load data from address 1000 1110.
- The advantage of write-back caches is that not all write operations need to access main memory, as with write-through caches.
  - If a single address is frequently written to, then it doesn't pay to keep writing that data through to main memory.
  - If several bytes within the same cache block are modified, they will only force one memory write operation at write-back time.

Write misses

- A second scenario is if we try to write to an address that is not already contained in the cache; this is called a write miss.
- Let’s say we want to store 21763 into Mem[1101 0110] but we find that address is not currently in the cache.

Index	V	Tag	Data
...			
110	1	00010	123456
...			

Address	Data
...	
1101 0110	6378
...	

- When we update Mem[1101 0110], should we *also* load it into the cache?

Write around caches (a.k.a. write-no-allocate)

- With a write around policy, the write operation goes directly to main memory *without* affecting the cache.

Mem[1101 0110] = 21763

Index	V	Tag	Data
...			
110	1	00010	123456
...			

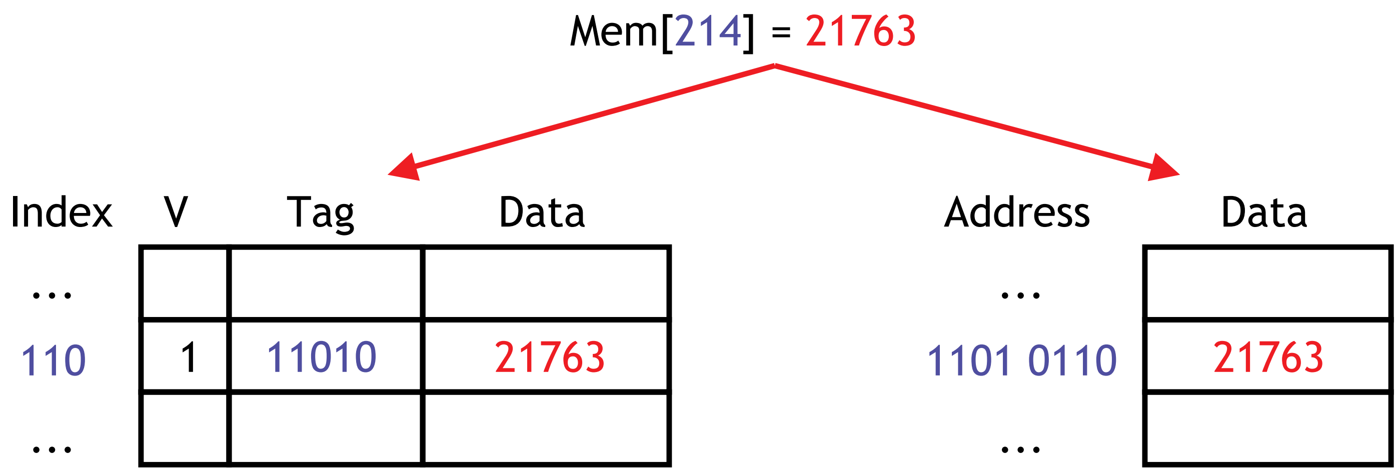
Address	Data
...	
1101 0110	21763
...	

- This is good when data is written but not immediately used again, in which case there's no point to load it into the cache yet.

```
for (int i = 0; i < SIZE; i++)
    a[i] = i;
```

Allocate on write

- An allocate on write strategy would instead load the newly written data into the cache.



- If that data is needed again soon, it will be available in the cache.

Which is it?

- Given the following trace of accesses, can you determine whether the cache is write-allocate or write-no-allocate?
  - Assume A and B are distinct, and can be in the cache simultaneously.

Miss	Load A
Miss	Store B
Hit	Store A
Hit	Load A
Miss	Load B
Hit	Load B
Hit	Load A

On a write-allocate cache this would be a hit

Answer: Write-no-allocate

Real Designs

Split Instruction/Data caches:

Pro: No structural hazard between IF & MEM stages

– A unified cache with one port stalls IF for lw/sw

Con: Static partitioning of cache – instructions vs. data

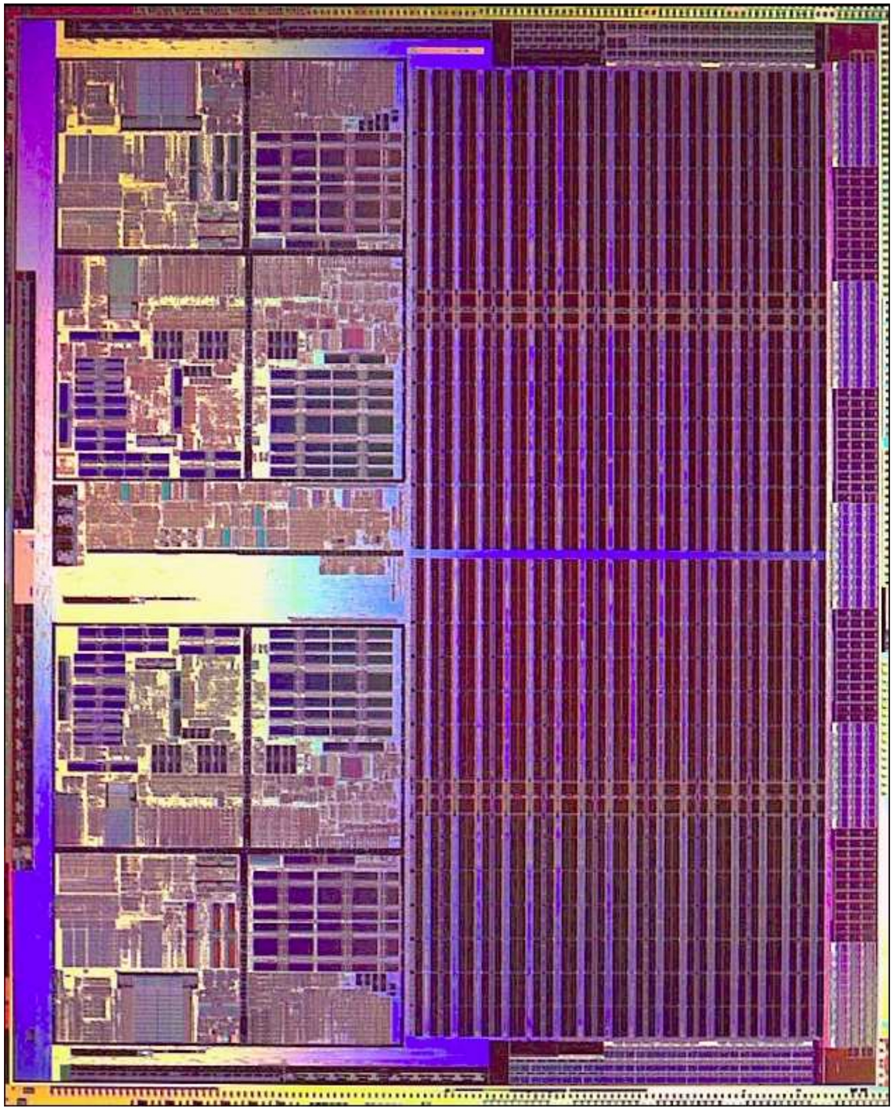
– Bad if unequal sizes: code/**DATA** or **CODE**/data

Cache Hierarchies:

Trade-off between access time & hit rate

– L1 cache has fast access time (okay hit rate)

– L2 cache has good hit rate (okay access time)



Dual-core Opteron

Average Memory Access Time

- Recall the formula:

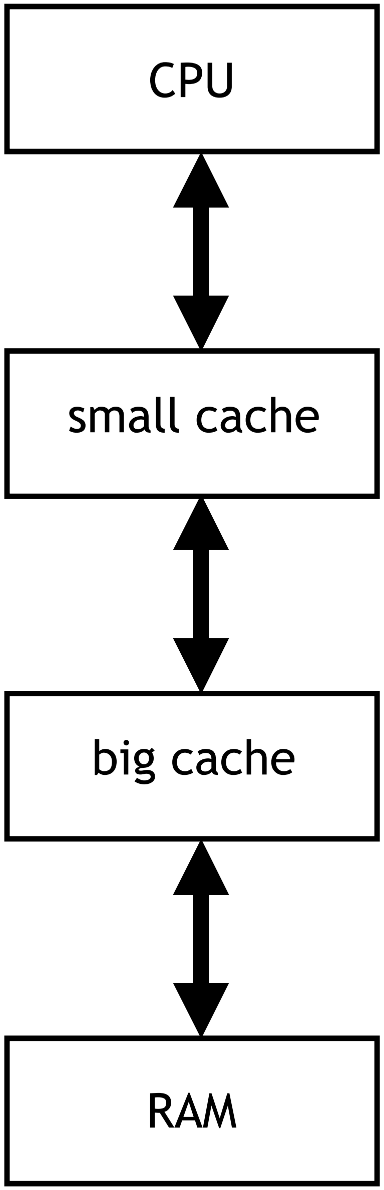
  AMAT = Hit time + (Miss rate x Miss penalty)

- To make AMAT smaller, we can decrease the miss rate
  – e.g. make the cache larger, add more associativity
  – but larger/more complex $\Rightarrow$ longer hit time!

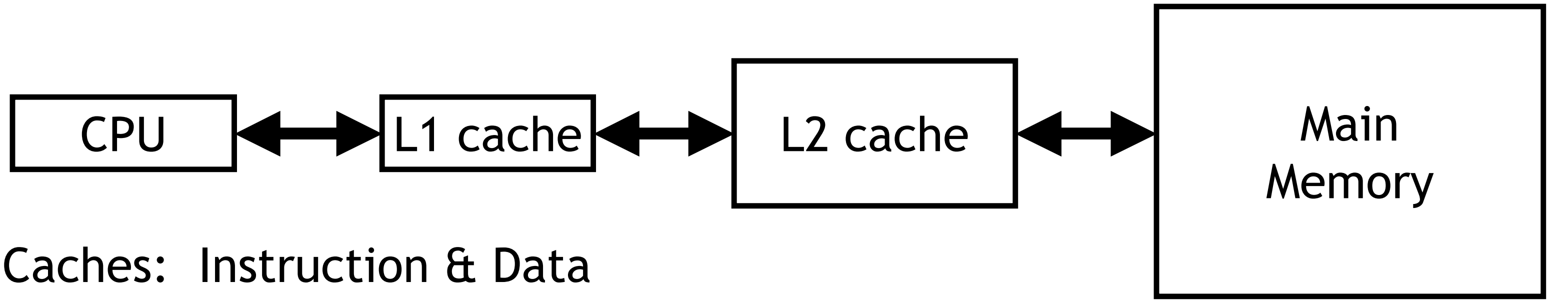
- Alternate approach: decrease the miss penalty
  – BIG idea: a big, slow cache is still faster than RAM!

- Modern processors have at least two cache levels

- Too many levels introduces other problems
  – keeping data consistent
  – communicating across levels



Opteron Vital Statistics



- L1 Caches: Instruction & Data
  - 64 kB
  - 64 byte blocks
  - 2-way set associative
  - 2 cycle access time

- L2 Cache:
  - 1 MB
  - 64 byte blocks
  - 4-way set associative
  - 16 cycle access time (total, not just miss penalty)

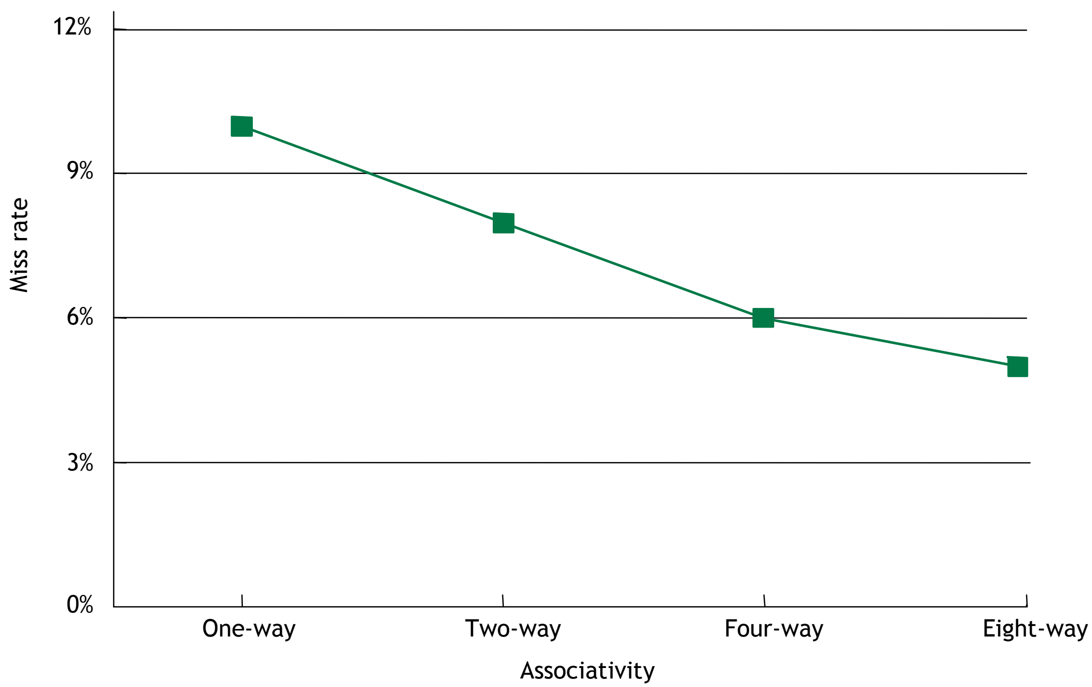
- Memory
  - 200+ cycle access time

Comparing cache organizations

- Like many architectural features, caches are evaluated experimentally.
  - As always, performance depends on the actual instruction mix, since different programs will have different memory access patterns.
  - Simulating or executing real applications is the most accurate way to measure performance characteristics.
- The graphs on the next few slides illustrate the simulated miss rates for several different cache designs.
  - Again lower miss rates are generally better, but remember that the miss rate is just one component of average memory access time and execution time.
  - You'll probably do some cache simulations if you take CS433.

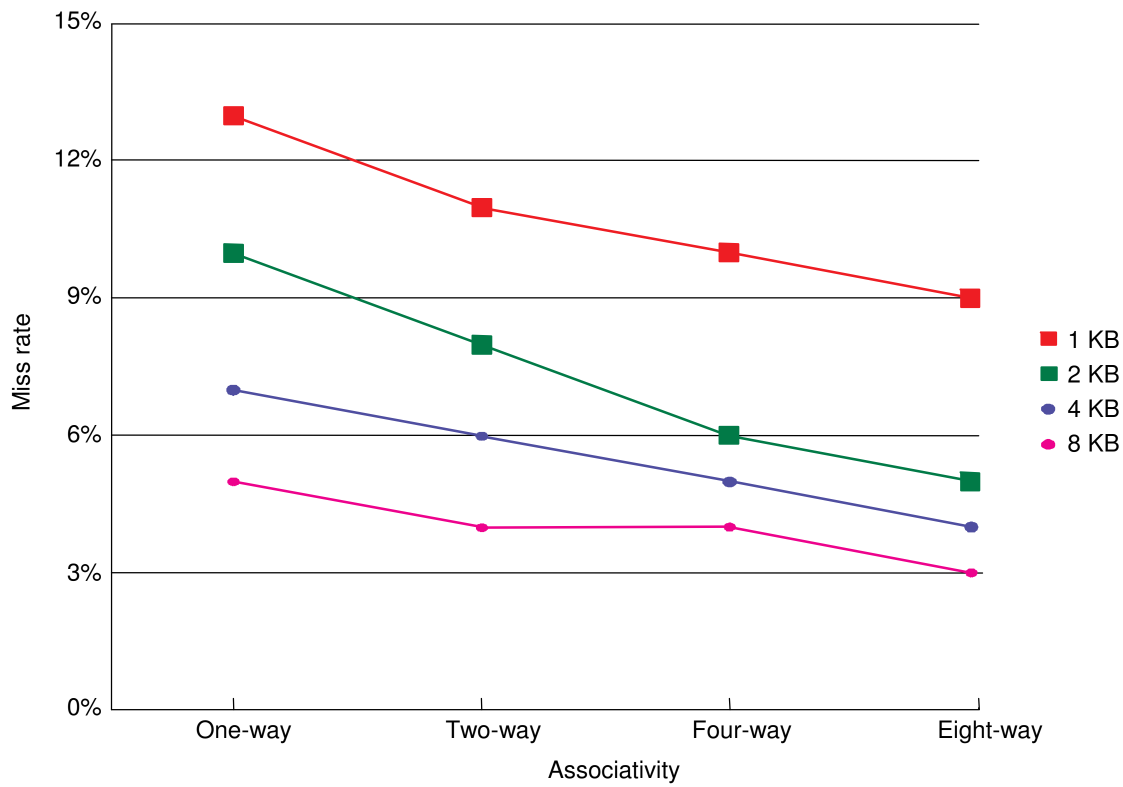
Associativity tradeoffs and miss rates

- As we saw last time, higher associativity means more complex hardware.
- But a highly-associative cache will also exhibit a lower miss rate.
  - Each set has more blocks, so there's less chance of a conflict between two addresses which both belong in the same set.
  - Overall, this will reduce AMAT and memory stall cycles.
- Figure 7.29 on p. 604 of the textbook shows the miss rates decreasing as the associativity increases.



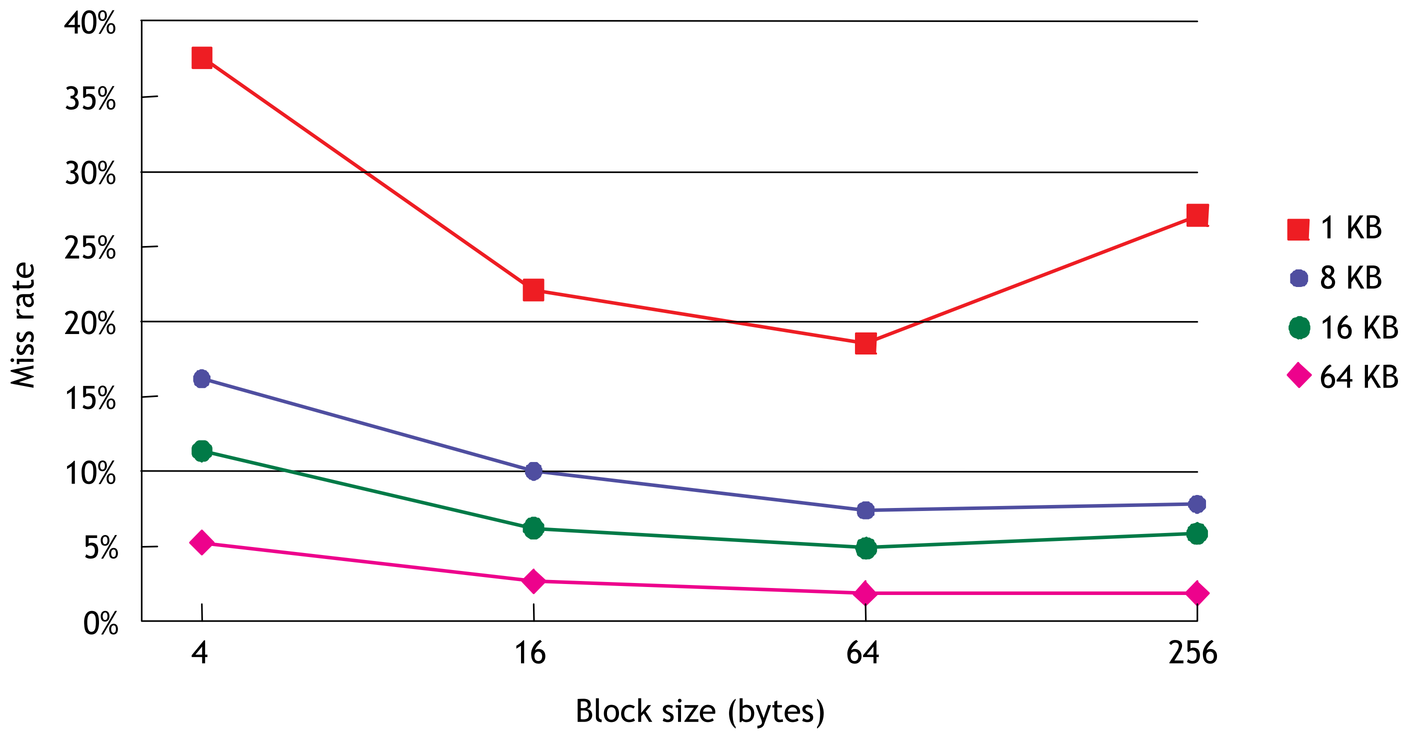
Cache size and miss rates

- The cache size also has a significant impact on performance.
  - The larger a cache is, the less chance there will be of a conflict.
  - Again this means the miss rate decreases, so the AMAT and number of memory stall cycles also decrease.
- The complete Figure 7.29 depicts the miss rate as a function of both the cache size and its associativity.



Block size and miss rates

- Finally, Figure 7.12 on p. 559 shows miss rates relative to the block size and overall cache size.
  - Smaller blocks do not take maximum advantage of spatial locality.
  - But if blocks are *too* large, there will be fewer blocks available, and more potential misses due to conflicts.



Memory and overall performance

- How do cache hits and misses affect overall system performance?
  - Assuming a hit time of one CPU clock cycle, program execution will continue normally on a cache hit. (Our earlier computations always assumed one clock cycle for an instruction fetch or data access.)
  - For cache misses, we'll assume the CPU must stall to wait for a load from main memory.
- The total number of stall cycles depends on the number of cache misses *and* the miss penalty.

Memory stall cycles = Memory accesses x miss rate x miss penalty

- To include stalls due to cache misses in CPU performance equations, we have to add them to the "base" number of execution cycles.

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

Performance example

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

$$\begin{aligned} \text{Memory stall cycles} &= \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty} \\ &= 0.33\ \mathtt{I} \times 0.03 \times 20 \text{ cycles} \\ &= 0.2\ \mathtt{I} \text{ cycles} \end{aligned}$$

- If $\mathtt{I}$ instructions are executed, then the number of wasted cycles will be $0.2 \times \mathtt{I}$.

  This code is 1.2 times slower than a program with a "perfect" CPI of 1!

Memory systems are a bottleneck

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

- Processor performance traditionally outpaces memory performance, so the memory system is often the system bottleneck.
- For example, with a base CPI of 1, the CPU time from the last page is:

$$\text{CPU time} = (\mathtt{I} + 0.2\ \mathtt{I}) \times \text{Cycle time}$$

- What if we could *double* the CPU performance so the CPI becomes 0.5, but memory performance remained the same?

$$\text{CPU time} = (0.5\ \mathtt{I} + 0.2\ \mathtt{I}) \times \text{Cycle time}$$

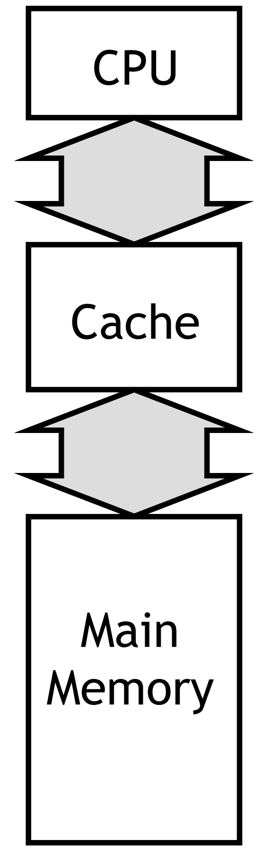
- The overall CPU time improves by just 1.2/0.7 = 1.7 times!
- Refer back to Amdahl's Law from textbook page 101.
  - Speeding up only part of a system has diminishing returns.

Basic main memory design

- There are some ways the main memory can be organized to reduce miss penalties and help with caching.
- For some concrete examples, let's assume the following three steps are taken when a cache needs to load data from the main memory.
  1. It takes 1 cycle to send an address to the RAM.
  2. There is a 15-cycle latency for each RAM access.
  3. It takes 1 cycle to return data from the RAM.
- In the setup shown here, the buses from the CPU to the cache and from the cache to RAM are all one word wide.
- If the cache has one-word blocks, then filling a block from RAM (*i.e.*, the miss penalty) would take 17 cycles.

$$1 + 15 + 1 = 17 \text{ clock cycles}$$

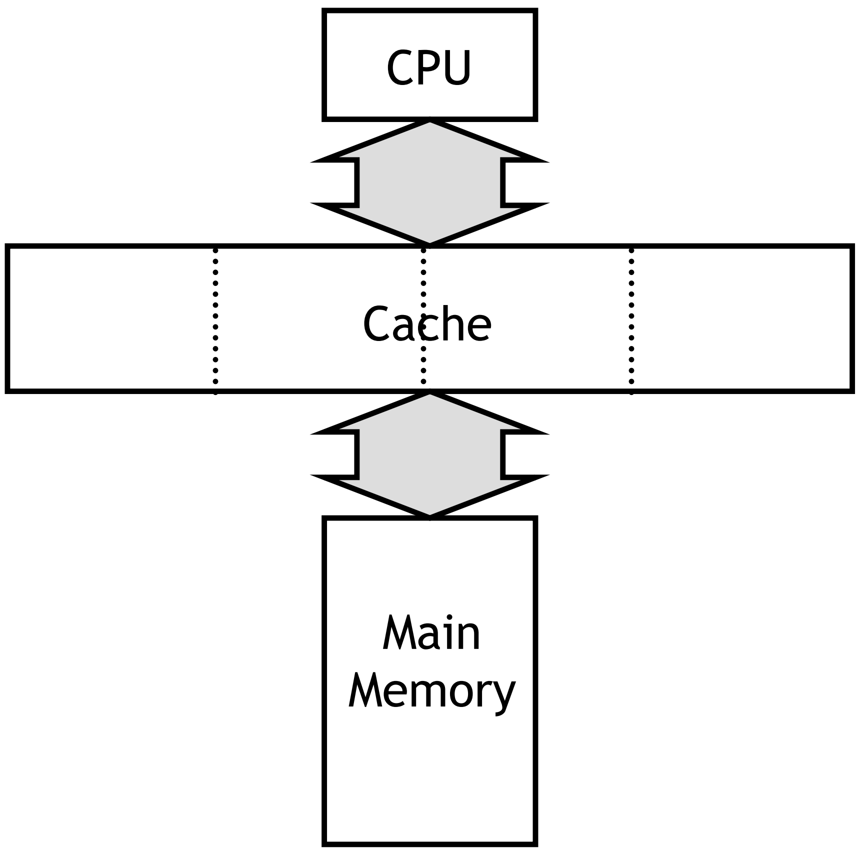
- The cache controller has to send the desired address to the RAM, wait and receive the data.



Miss penalties for larger cache blocks

- If the cache has four-word blocks, then loading a single block would need four individual main memory accesses, and a miss penalty of 68 cycles!

$$4 \times (1 + 15 + 1) = 68 \text{ clock cycles}$$

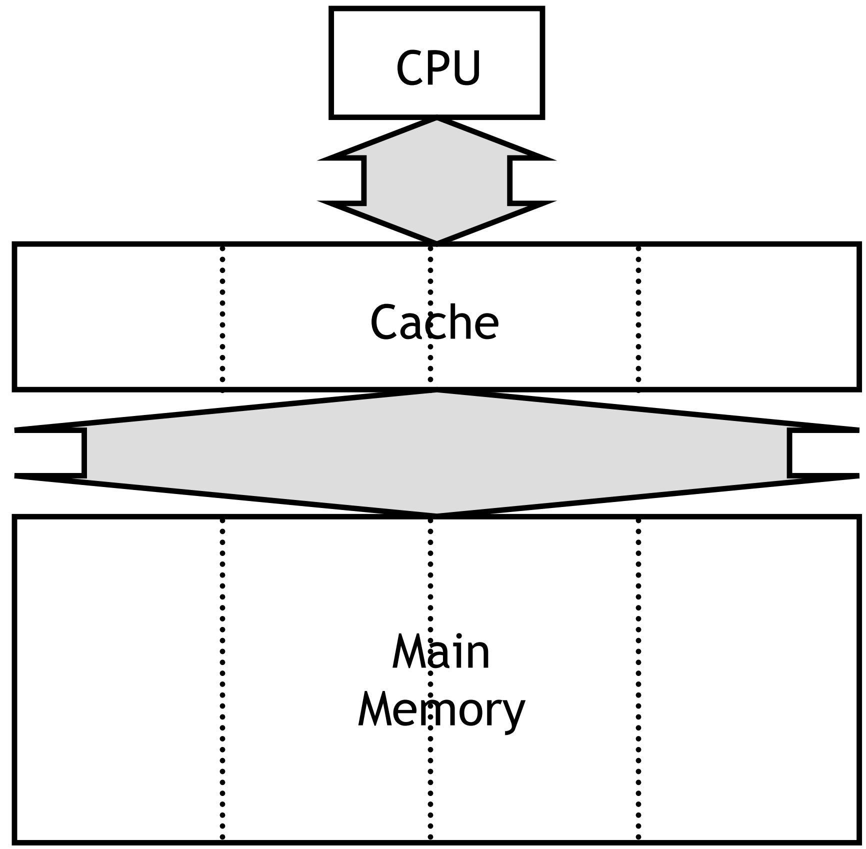


A wider memory

- A simple way to decrease the miss penalty is to widen the memory and its interface to the cache, so we can read multiple words from RAM in one shot.
- If we could read four words from the memory at once, a four-word cache load would need just 17 cycles.

  1 + 15 + 1 = 17 cycles

- The disadvantage is the cost of the wider buses—each additional bit of memory width requires another connection to the cache.

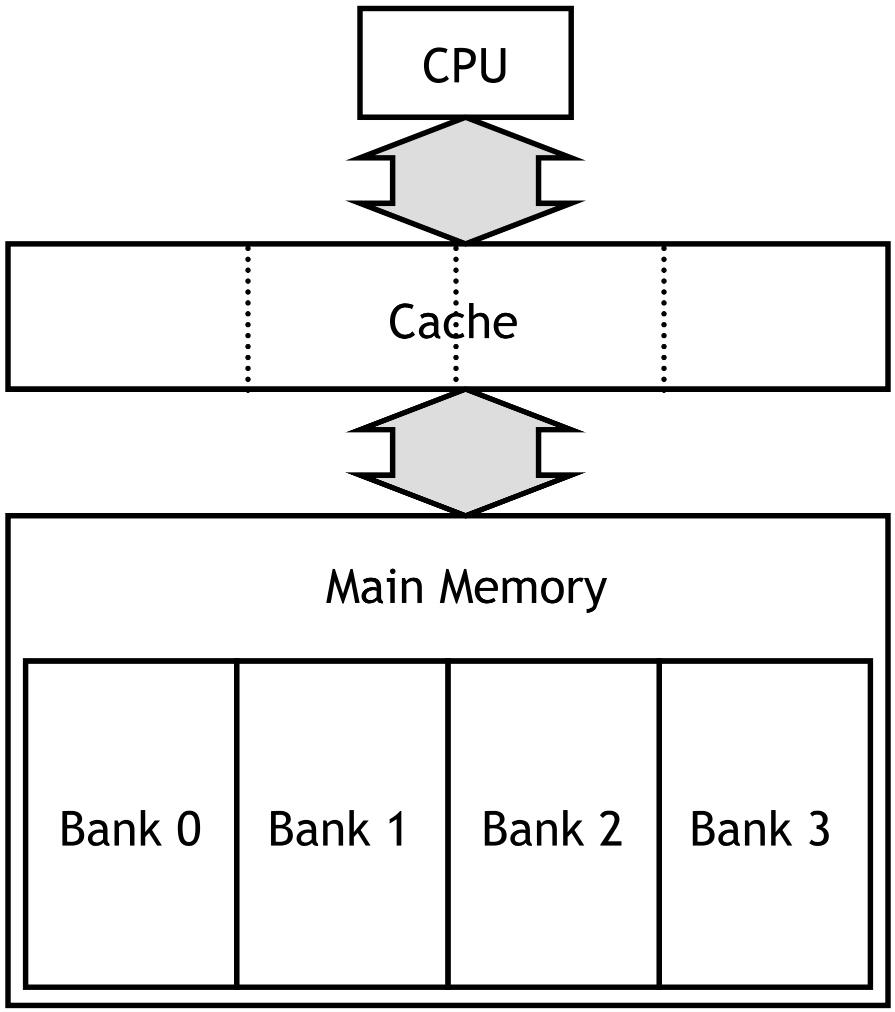


An interleaved memory

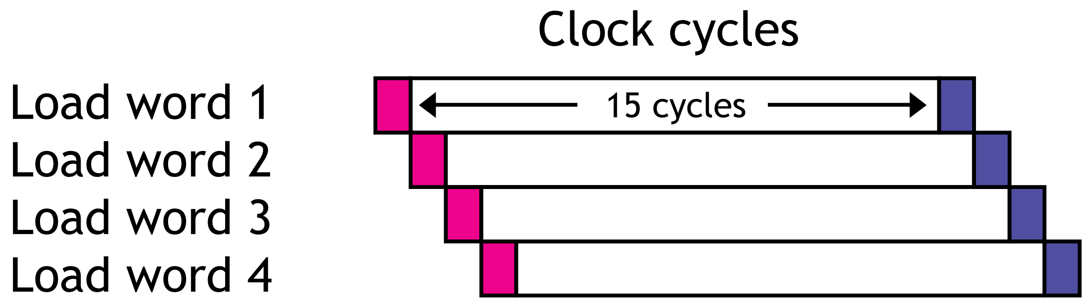
- Another approach is to interleave the memory, or split it into “banks” that can be accessed individually.
- The main benefit is overlapping the latencies of accessing each word.
- For example, if our main memory has four banks, each one byte wide, then we could load four bytes into a cache block in just 20 cycles.

$$1 + 15 + (4 \times 1) = 20 \text{ cycles}$$

- Our buses are still one byte wide here, so four cycles are needed to transfer data to the caches.
- This is cheaper than implementing a four-byte bus, but not too much slower.



Interleaved memory accesses



- Here is a diagram to show how the memory accesses can be interleaved.
  - The magenta cycles represent sending an address to a memory bank.
  - Each memory bank has a 15-cycle latency, and it takes another cycle (shown in blue) to return data from the memory.
- This is the same basic idea as pipelining!
  - As soon as we request data from one memory bank, we can go ahead and request data from another bank as well.
  - Each individual load takes 17 clock cycles, but four overlapped loads require just 20 cycles.

Which is better?

- Increasing block size can improve hit rate (due to spatial locality), but transfer time increases. Which cache configuration would be better?

	Cache #1	Cache #2
Block size	32-bytes	64-bytes
Miss rate	5%	4%

- Assume both caches have single cycle hit times. Memory accesses take 15 cycles, and the memory bus is 8-bytes wide:
  - i.e., an 16-byte memory access takes 18 cycles:

    1 (send address) + 15 (memory access) + 2 (two 8-byte transfers)

recall: AMAT = Hit time + (Miss rate x Miss penalty)

Summary

- Writing to a cache poses a couple of interesting issues.
  - Write-through and write-back policies keep the cache consistent with main memory in different ways for write hits.
  - Write-around and allocate-on-write are two strategies to handle write misses, differing in whether updated data is loaded into the cache.
- Memory system performance depends upon the cache hit time, miss rate and miss penalty, as well as the actual program being executed.
  - We can use these numbers to find the average memory access time.
  - We can also revise our CPU time formula to include stall cycles.

AMAT = Hit time + (Miss rate x Miss penalty)

Memory stall cycles = Memory accesses x miss rate x miss penalty

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

- The organization of a memory system affects its performance.
  - The cache size, block size, and associativity affect the miss rate.
  - We can organize the main memory to help reduce miss penalties. For example, interleaved memory supports pipelined data accesses.