

## Lecture 18

---

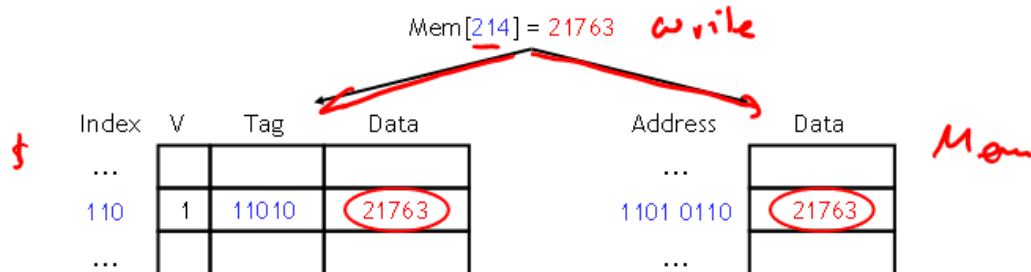
- Review:
  - Write-through?
  - Write-back?
- Block allocation policy on a write miss
- Cache performance

• HW3

• final review session

## Write-through caches

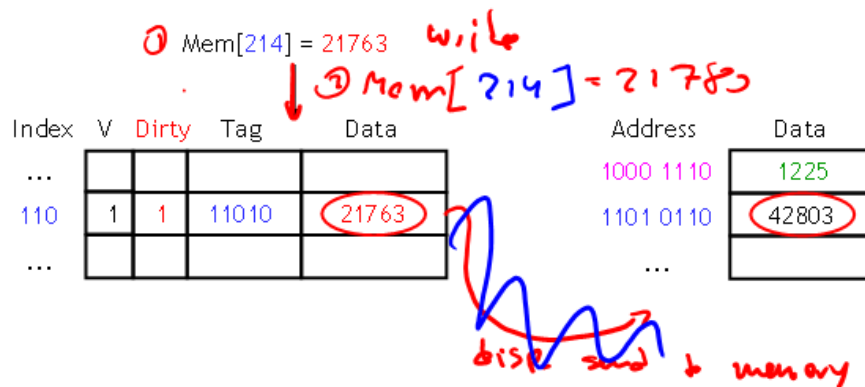
- A **write-through cache** forces all writes to update both the cache *and* the main memory.



- This is simple to implement and keeps the cache and memory consistent.
- The bad thing is that forcing every write to go to main memory, we use up bandwidth between the cache and the memory.

## 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

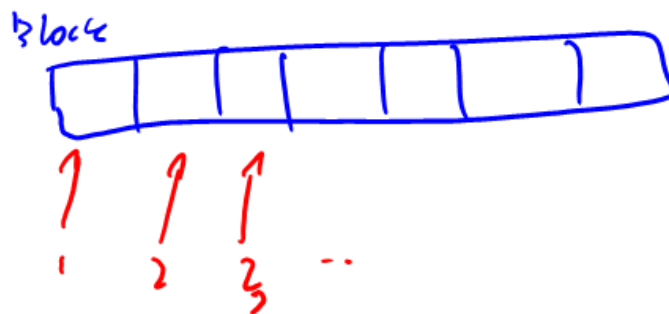


- Subsequent reads to the same memory address?
- Multiple writes to same block?

## Write-back cache discussion

---

- 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

---

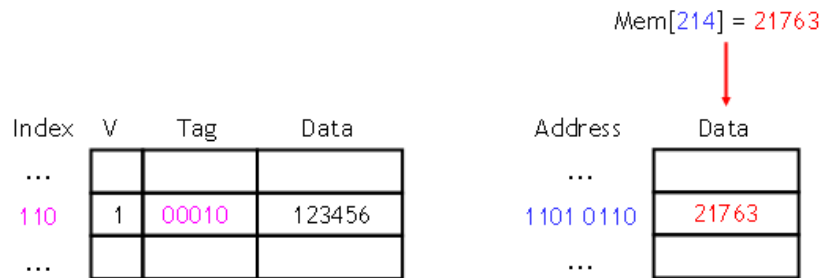
- A second scenario is if we try to write to an address that is not already contained in the cache; this is called a write miss.
- Let's say we want to store 21763 into Mem[1101 0110] but we find that address is not currently in the cache.

Index	V	Tag	Data	Address	Data
...				...	
110	1	00010	123456	1101 0110	6378
...				...	

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

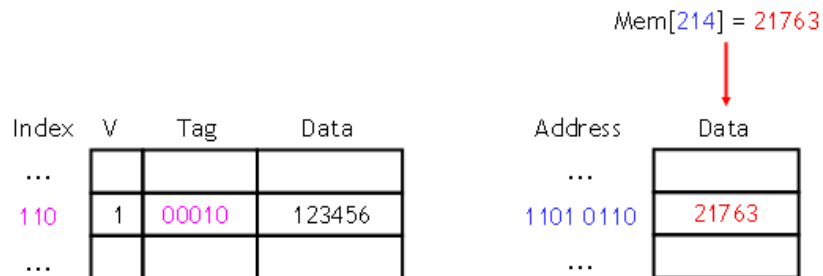
## Write around caches (a.k.a. write-no-allocate)

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



## 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.

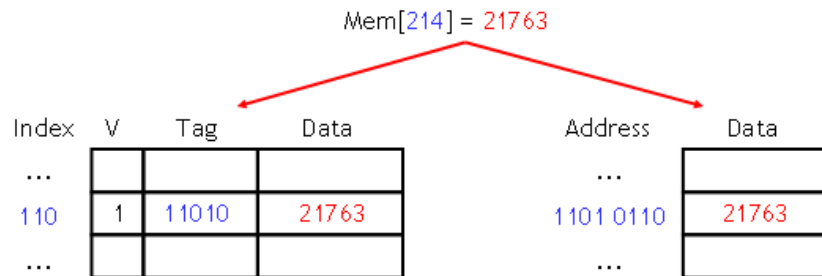


- This is good when data is written but not immediately used again, in which case there's no point to load it into the cache yet.

```
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

write no  
allocate

## 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

Answer: Write-no-allocate

On a write-allocate cache this would be a hit



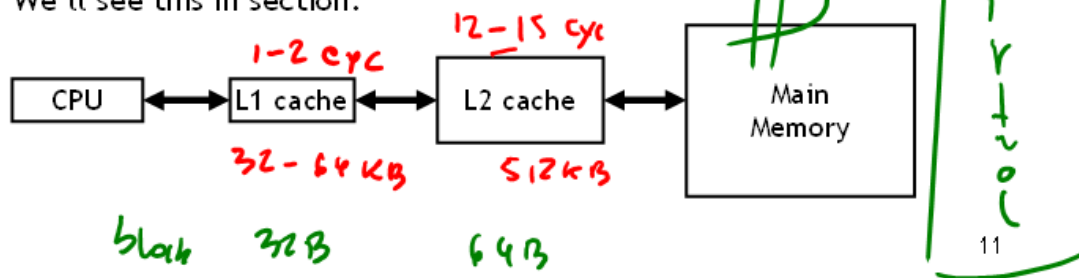
## First Observations

- Split Instruction/Data caches:

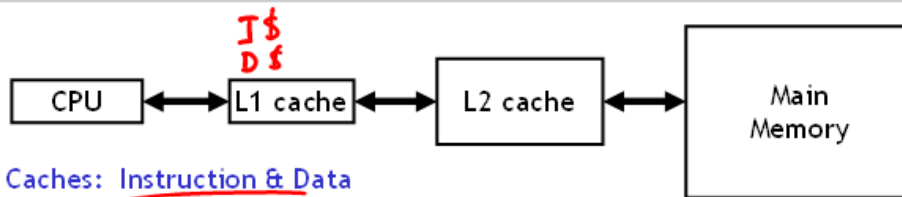
- Pro: No structural hazard between IF & MEM stages
  - A single-ported unified cache stalls fetch during load or store
- Con: Static partitioning of cache between instructions & data
  - Bad if working sets unequal: e.g., code/DATA or CODE/data

- Cache Hierarchies:

- Trade-off between access time & hit rate
  - L1 cache can focus on fast access time (okay hit rate)
  - L2 cache can focus on good hit rate (okay access time)
- Such hierarchical design is another “big idea”
- We’ll see this in section.



## ⇒ 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

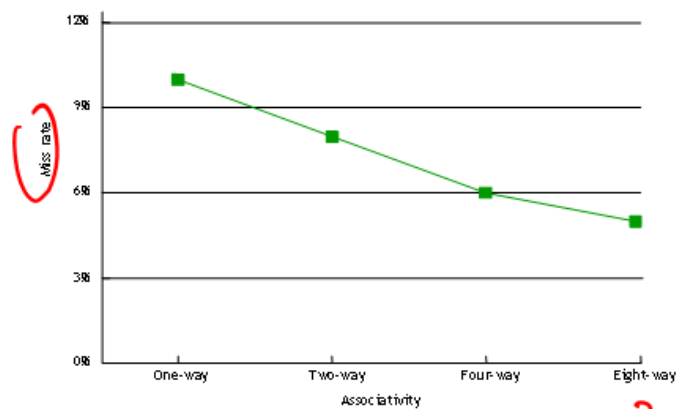
---

- 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

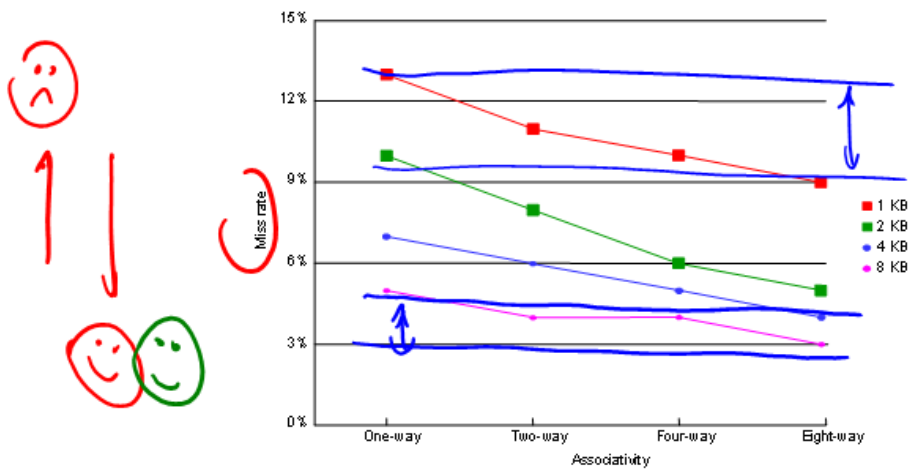
- As we saw last time, higher associativity means more complex hardware.
- But a highly-associative cache will also exhibit a lower miss rate.
  - Each set has more blocks, so there's less chance of a conflict between two addresses which both belong in the same set.
  - Overall, this will reduce AMAT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



↑ cost (power)

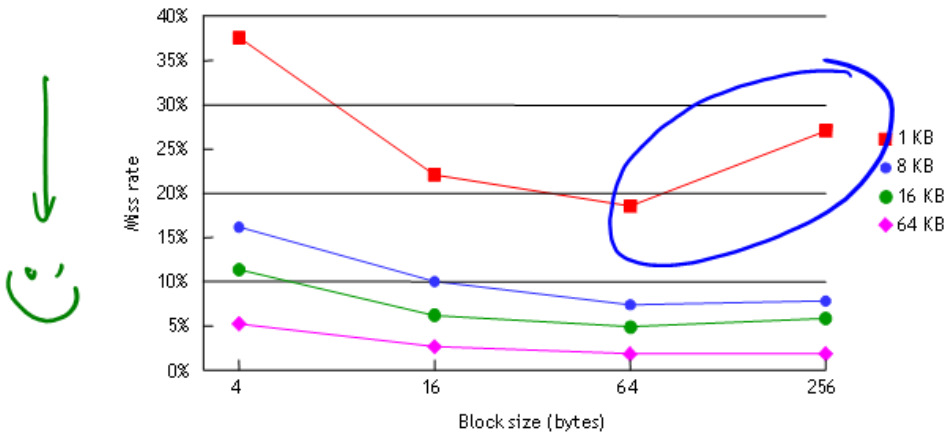
## Cache size and miss rates

- 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

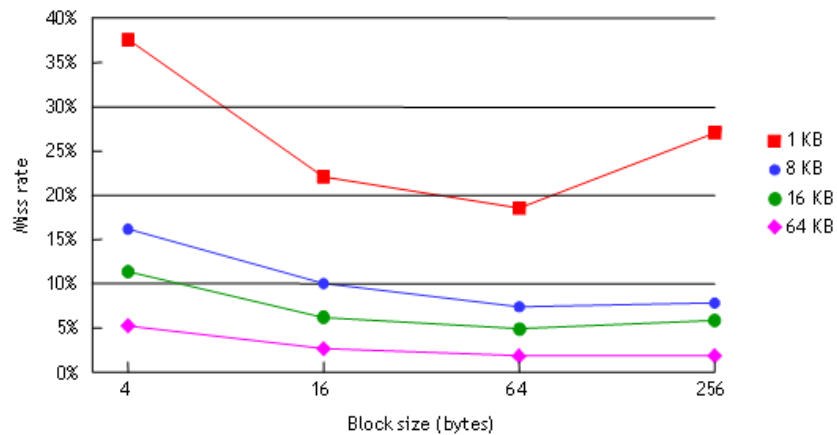
- Finally, Figure 7.12 on p. 559 shows miss rates relative to the block size and overall cache size.
  - Smaller blocks do not take maximum advantage of ~~space~~ ~~bandwidth~~.





## 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.

$$\text{Memory stall cycles} = \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty}$$

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

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



## 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.

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

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

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

## Memory systems are a bottleneck

---

CPU time = (CPU execution cycles + Memory stall cycles) × 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:

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

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

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

- 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.

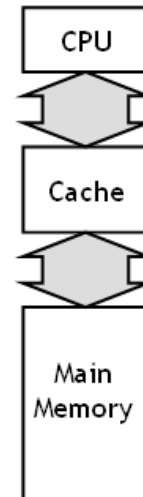
## Basic main memory design

---

- 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.

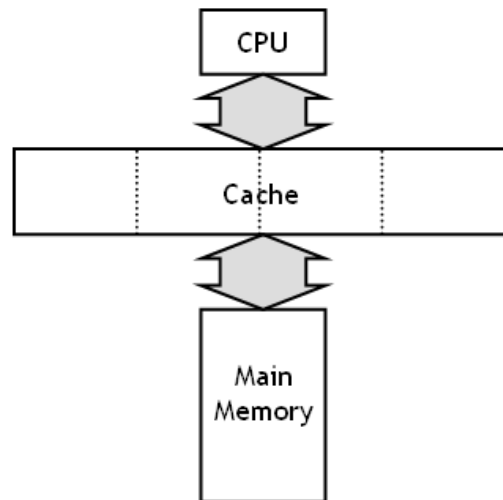


## Miss penalties for larger cache blocks

---

- 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!

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

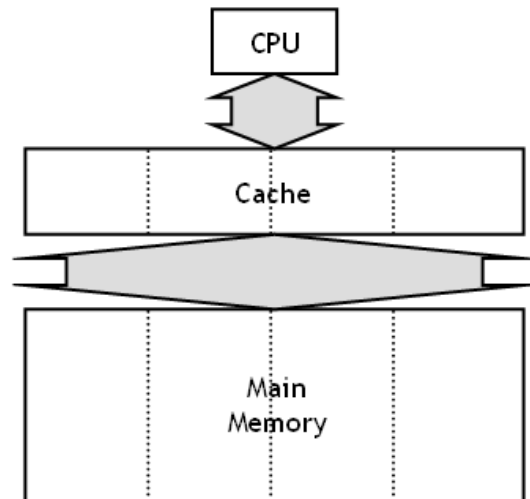


## A wider memory

- 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.

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

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

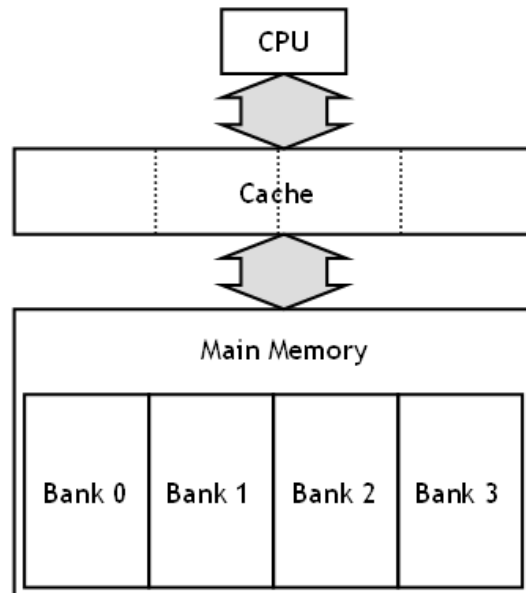


## An interleaved memory

- 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.

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

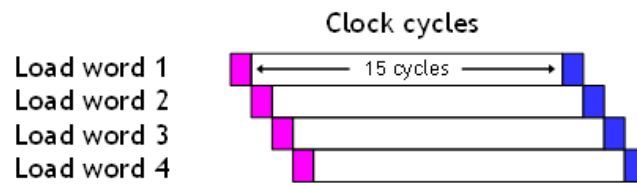
- Our buses are still one byte wide here, so four cycles are needed to transfer data to the caches.
- This is cheaper than implementing a four-byte bus, but not too much slower.





## 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:  $AMAT = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})$

## 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)

Cache #1:

$$\text{Miss Penalty} = 1 + 15 + 32\text{B}/8\text{B} = 20 \text{ cycles}$$

$$\text{AMAT} = 1 + (.05 * 20) = 2$$

Cache #2:

$$\text{Miss Penalty} = 1 + 15 + 64\text{B}/8\text{B} = 24 \text{ cycles}$$

$$\text{AMAT} = 1 + (.04 * 24) = \sim 1.96$$

recall:  $\text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})$

## 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**.

$$\text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})$$

$$\text{Memory stall cycles} = \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty}$$

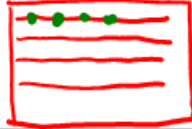
$$\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{Cycle time}$$

- 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.

## Writing Cache Friendly Code

- Two major rules:
- Repeated references to data are good (**temporal locality**)
- Stride-1 reference patterns are good (**spatial locality**)
- Example: cold cache, 4-byte words, 4-word cache blocks

V Tune



```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

Adapted from Randy Bryant