15213 Lecture 10: The Memory Hierarchy

Learning Objectives:

- Understand locality of reference and why useful programs tend to exhibit it.
- Be able to distinguish between the two types of locality.
- Given a simple program, be able to estimate its locality and suggest ways to improve it.
- Explain property of cache blocks and their advantage as a unit of cache organization.
- Be able to list the three major types of cache misses.

1 Locality of Reference

The two ends of the memory hierarchy exhibit wildly different performance: their latency (delay) and throughput (line rate) differ by many orders of magnitude. You might expect the performance of programs (and, ultimately, a computer system) to match that of the slowest storage technology it requires; fortunately, this is not the case. In fact, overall performance is usually closer to that of the *fastest* technology! Let's look at why.

1. Consider the following code:

   ```
   int foo = 0;
   ...
   ```

   Do you expect that the program is done accessing `foo`? Why or why not?
   Probably not. Why would a program initialize an int to no effect?! This would just get optimized out by the compiler anyway!

2. Now consider this code:

   ```
   int foo = 0;
   foo = prompt_int("Please enter foo");
   ...
   ```

   Do you expect that the program is done accessing `foo`? Why or why not?
   Still probably not! Why would you query the user if you didn't use the resuling input anywhere?!

The program property you just observed is known as **temporal locality**.

3. Consider the following code:

```
int bar[8];
bar[0] = prompt_int("Enter first score");
...
```

Which index of `bar` do you expect the program to access next?
Given that this program seems to be accessing each index in the array in turn, probably index '1'.

This property is known as **spatial locality**.

4. Given this code:

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

Indicate the type(s) of locality exhibited by each of the program's variables.

variable	temporal locality?	spatial locality?	no locality?
`a`	yes		
`a[i]`		yes	
`i`	yes		
`n`	yes		
`sum`	yes		

2 Improving Locality

Of course, although locality is a common program property, different programs exhibit it to varying extents. Take the following example:

```
1 int a[3][4];
2 // (init a)
3 int sum = 0;
4 for(int j = 0; j < 4; ++j)
5         for(int i = 0; i < 3; ++i)
6                 sum += a[i][j];
```

5. Using your knowledge of nested arrays' memory layout, number the entries of the following table to indicate the order in which the code accesses `a`'s memory locations.

`a[0][0]` `a[2][3]`

0				1							

Answer: 0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 11

6. What change could you make to the code to improve its spatial locality?
   You could flip the loop iteration order from 'j,i' to 'i,j.' This change would decrease the 'stride' of each memory access from 4 ints (16 bytes) to 1 int (4 bytes).

3 Caching Main Memory

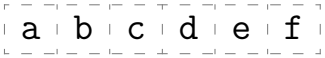
Locality means that programs tend to access the data at any given level of the memory hierarchy more often than that at the next level. Over the past several decades, increases in CPU speeds have greatly outpaced those in memory speeds, leaving a wide gap in the memory hierarchy. There is now a latency difference of at least two orders of magnitude between registers and main memory, meaning that the CPU can execute hundreds of instructions in the time it takes for a single memory access to complete. In order to hide the latency of main memory, modern memory hierarchies add one or more levels of **CPU cache** before it, whose contents the CPU manages automatically. We now examine this management mechanism.

One fundamental property of a CPU cache is that it splits memory into **blocks** of adjacent bytes: Whenever an access cannot be served from cache (a **cache miss**), the CPU transfers the entire block surrounding the requested address. While designs using a block size of a single byte are possible, larger sizes have become much more prevalent.

7. In terms of locality, what is the benefit of having multi-byte blocks?
   Since each 'miss' is loading the cache with multiple bytes at once, programs using bytes 'nearby' to those just loaded (programs exhibiting good spatial locality) execute faster for free.
8. Imagine *accessing* the fields of the following struct in the order `a`, `b`, `d`, `e` on a machine.

```
struct ure {
        int a;
        int b;
        int c;
        int d;
        int e;
        int f;
};
```

   Draw boxes around adjacent variables to construct blocks of two `int`s that would minimize the number of cache misses triggered by these accesses:



Draw one box around `a` and `b`, and another around `d` and `e`. This results in only 2 misses.

9. Is there any combination of boxes (no matter their size) that would avoid the miss on `a`?
   NO! The cache has no data in it at this point, so a's value has to be loaded from memory.

The cache misses you just observed are known as **compulsory misses** because they occur as a result of accessing data for the first time.

10. Imagine that the previous sequence of accesses were followed by one to the `c` member. With the blocks you drew before, what happens as you draw a box around `c`?
    You now need to cover one of the existing blocks. This is a problem because we now have two redundant copies of one of these memory locations in cache. Would we now have to keep them in sync as writes changed their values?
11. What additional requirement should we impose on blocks to avoid this situation?
    Blocks should be aligned with the cache, which is to say, every address should be mapped to only one block. This means that the box we drew around `d` and `e` before would now be invalid.

4 Reusing Cache Lines

While cached, each block occupies a location known as a **cache line**. When the cache runs out of lines, it must free space by **evicting** an existing block back to memory every time it needs to cache a new one. Where and how this happens is the subject of a later lecture.

```
1 int sum = 0, prod = 1;
2 // (init buffer, an array of N ints for some really large N)
3 for(int index = 0; index < N; ++index)
4         sum += buffer[index];
5 for(int index = 0; index < N; ++index)
6         prod *= buffer[index];
```

12. When running the above code, how many times does the *first* for loop access `buffer`?
    N.

13. Let's assume we have a cache with `M` blocks of 4 ints, where `M` is much larger than `N`. How many of those accesses miss? You can start by accessing `buffer[0]`, then `buffer[1]`, and so on until you find a pattern.
`buffer[0]` misses.
`buffer[1]` up to `buffer[3]` hit because `buffer[0]` loaded 4 ints into the cache.
`buffer[4]` misses, etc.
We can see that 25% of accesses miss.

14. How many `buffer` accesses and misses are there in the *second* for loop?
Since our cache is already loaded with the entirety of `buffer`, *none* of the second loop's `N` accesses miss

15. If we reduced the number of blocks from `M` to 1, this would greatly increase the number of misses in the second for loop. Why?
If the cache has only one block, then it will hold `buffer[N-4]` to `buffer[N-1]` after the first loop finishes. However, since the second loop starts over from `buffer[0]`, that data will immediately be replaced. 25% of the second loop's accesses will miss, in the same pattern as the first loop.

We call this new type of miss a **capacity miss** because it results from limited cache capacity. Notice that such misses occur even in programs exhibiting good locality.

16. Look back at the code sample at the beginning of section 2. How many bytes does the `a` array occupy?
3 rows * 4 cols * 4 bytes per int = 48 bytes total.

17. We run that code on a machine with 16-byte blocks. What is the fewest number of such blocks that `a` could fit in?
3 blocks.

18. For which indices of the array do you expect a cache miss on line 6 of the program?
Indices `[0][0]`, `[1][0]`, and `[2][0]`.

For performance reasons, many caches further limit which cache line(s) a given block of memory can map to (i.e. where a given address can be stored in cache). Depending on a program's memory access pattern, this can necessitate evictions before the cache is full.

19. Imagine the machine in the previous question mapped *all* the blocks corresponding to `a` to the same cache line. How would this change the pattern of misses?
Since the line can only fit one block, but is mapped to by multiple blocks, every single access to `a` misses, as each access to `a` lies outside the previously cached block.

You just encountered what are known as **conflict misses**. When talking a program's cache behavior, we often combine the the number of **capacity** and **conflict** misses.

20. *(Advanced)* Assume the cache is divided into 256 lines. The machine's cache line mapping in the previous question should seem overly naïve: when running this simple program, it would have wasted 255 of these lines! Using only byte-sized constant(s) and bitwise operations, propose a datalab-style mapping function from 64-bit memory address to cache line index. Your scheme should be kinder to the original program, but must not split blocks between separate cache lines. *(Hint: Think about which bits of the memory address you should use in order to best achieve these goals.)*
We know we need to map the space into exactly 256 lines, which means we need 8 bits to do so ($2^8 = 256$). We also know we'll need 4 bits to identify the byte offset within a given block, as blocks are $16 = 2^4$ bytes wide. Because we can't split blocks between separate lines, we'll need to use the lowest 4 bits of the address for our block offset. This means the next 8 bits give us our line index (the rest of the address will be used for the tag, see CS:APP for details). This means that the following mapping function suffices:

```
#include <inttypes.h>
uint8_t add2line(uint64_t address){
    uint64_t mask = 0x0FF0;
    return (uint8_t) ((address & mask) >> 4);
}
```