CS-473: System Programming for SOCs

Practical Work 5:
Structure Memory Layout and Caches

Report

Andrea Grillo

371099

Index

Theory Questions

Question 1

The *__packed* modifier adds the *packed* attribute to the definition of the struct. This tells the compiler not to insert the padding bits for alignment. This leads to a more compact memory footprint of the struct, but a more complex and slower code, as memory accesses can only be aligned on most architectures, so the compiler has to use workarounds to get the data using shifting and/or multiple accesses.

Question 2

By default, the C compiler does not reorder the fields of the structs, so the order will always be the same as the definition in the code, that is, the 4 bytes of the id integer and then the variable number of chars of the data array.

For the *item_t* without *__packed* attribute:

PARAM_DATALEN	ID size	Data size	Padding bytes	Total struct size
1	4	1	3	8
2	4	2	2	8
3	4	3	1	8
4	4	4	0	8

For the *item_t* with *__packed* attribute:

PARAM_DATALEN	ID size	Data size	Padding bytes	Total struct size
1	4	1	0	5
2	4	2	0	6
3	4	3	0	7
4	4	4	0	8

All the sizes are in bytes.

Question 3

Without *__packed* attribute:

- As the size of the *item_t* structure has the same total struct size for *PARAM_DATALEN* in 1..4 because of the padding bytes, the size and layout of the array of item_t structure will not change, and will be equal to 16 bytes.

With *__packed* attribute:

- The structures are stored next to each other without padding bytes, which leads to optimization of the total memory footprint.

PARAM_DATALEN	Item_t size	Array of 2 item_t size
1	5	10
2	6	12
3	7	14
4	8	16

Question 4

The *__packed* modifier reduces the memory footprint of the array of structures, so the total space usage is less. This means that more structs can fit in a cache line and the number of cache misses is lower. This does not happen for values of the *PARAM_DATALEN* for which there are no padding bits in the structure (*PARAM_DATALEN* multiple of *4*).

Question 5

Having a look at the tested program, we can see that it is always accessing just the *id* of every structure. When the size of the struct is smaller than a cache line, multiple *id*s can be read from a single line, and the number of ids that are stored in a single line is decreasing with the increase of data length. When we reach a certain size, the *item_t* structure is bigger than a cache line, but we just need the *id*, which is at the beginning of the structure. In this case, we need to evict the line for every different *id* access, and this number will not increase anymore with the increase of the data length.

In the graph, the number of cache misses is constant starting from data length *25*. I think that the probable size of the data cache line is *32 bytes*.

Question 6

As the unaligned memory accesses are not possible, the compiler has generated the code to recombine the unaligned data. To do so, the generated assembly is longer, and slower to execute. The compiler generates multiple *l.lbz* commands to access a single byte, and then multiple shifts and *or* between registers to recombine the data.

TASK 1

Question 7

The *node_t* structure is *64* bytes long, which is bigger than a cache line. The *node_count* function accesses the id and next member of the structs, which are respectively at the beginning and at the end of the structure. As the structure is big, the spatial locality is bad and the two variables will be stored in different data cache lines.

Question 8

My strategy is to improve the spatial locality of the memory accesses in the *node_count* function. As the function only accesses the id and next fields of the struct, I will reorder those fields to be next to each other in memory, at the beginning of the struct.

The number of cache misses has decreased from *43* to *27* in my tests.

TASK 2

Question 9

The *items_find* function has to cycle over the array of *item_t* structures. The *item_t* is a *36* bytes structure which includes an id integer and an array of *32* chars.

As the *item_t* structure is quite big, the whole array cannot fit in a single cache line, so accessing in series all the structures of the array leads to multiple cache misses, because a lot of lines will need to be evicted and loaded again to cache.

Question 10

My idea to decrease the number of caches misses makes use of the *mem* structure, using the *alloc* function.

I have modified the *item_t* structure definition to include just the id and a pointer to the data array, which will be stored in the *mem* structure. This means that now the size of the *item_t* structure is 8 bytes, so now more structures can fit in a cache line, and this improves the spatial locality of the *for* cycle in the *items_find* function and thus reduces the number of cache misses.

The number of cache misses has decreased from *16* to *4* in my tests.

TASK 3

Question 11

Row-major and column-major are methods for storing matrices in memory. Using the row-major method, the consecutive elements of a row are stored in memory next to each other, and vice versa for column-major.

In C the standard method is *row-major*.

Question 12

The problem about the multiply algorithm is that the spatial locality of the memory accesses is not optimized, as during the multiplication different rows are accessed sequentially, which are not contiguous in memory because of the row-major method, so this leads to multiple cache line misses and evictions.

Question 13

An easy way to decrease the number of cache misses is to change the cycling strategy in the multiply code. Just by inverting the inner cycle and the outer one makes the memory accesses much more optimized in terms of spatial locality, as the multiplication is done row by row, and all the elements in a single row are stored contiguously.

The number of cache misses has decreased from *65893* to *8257* in my tests.

TASK 4

Question 14

The *LEDS_NEW_BASE* address is within the cacheable area of the memory. As a result, the data of the leds will be cached. As the cache type is set to *write-back*, the data is not written to the led address when it's modified, it just stays in the cache until the cache line has to be evicted, which will not happen frequently as we always access the same memory region to access the leds.

Question 15

Possible solutions I have found:

1. Disable the cache, using the *'dcache_enable(0)'* instruction.
2. Change the cache type to *write-through* using the *CACHE_WRITE_THROUGH* constant, so all the modifications to data in the cache are written to memory as they are made.
3. Make the led addresses outside of cacheable memory. This can be done by disabling the call to the *init_leds* function and modifying the definition of the leds pointer to use the *LEDS_OLD_BASE* address.
4. Adding a dcache_flush instruction after every updating of the leds data.

All of these solutions either make cache useless or prevent the use of it. For memory accesses that are used to deal with hardware peripherals, the use of a cache is usually wrong.

Note: in the submitted code there is the possibility to select which solution to implement by setting a constant at the beginning of the file.