CS476 - PW4 Report

Julien de Castelnau (368509), Magnus Meyer (396302)

November 12, 2024

Exercises

1. As explained in the referenced article, alignment requirements can produce “unneccesary” padding, that is required to ensure that memory accesses are aligned. As an example, integers (4 bytes) must start on an address divisible by 4 (2 right-most bits 0), while long (8 bytes) needs to be on addresses divisible by 8, while chars can be on any address.

As described, this forces the compiler to introduce “padding”, in order to make sure that the next address is aligned with its type. The ”___packed” attribute specifies that fields within the struct should be as compact as possible. Source.

2.

- The ordering of the fields within the struct is fixed in all cases: 4 byte `id` followed directly by `data`, however long it is, plus padding at the end depending on `PARAM_DATALEN`
- `id` is always 4 bytes, while data is `PARAM_DATALEN` bytes long.
- If `__packed` is not present, padding bytes are `4 - PARAM_DATALEN` to satisfy requirement for `item_t` to be aligned to 4 byte boundary. Otherwise, it is 0.

3. If `__packed` is present, the `data` is immediately followed by the `id` of the next `item_t` in the array. Otherwise, there is the aforementioned number of bytes of padding separating the two elements.
4. The number of cache misses saturate at 256 when `PARAM_DATALEN` > 24. Since `PARAM_COUNT`=256, that means that there is a miss for every loop iteration. So every iteration accesses a different cache line, and since each iteration accesses the `ith` struct in the array, the struct must be at least as large as the cache line. The first size in which we saturate is when `PARAM_DATALEN = 25` or when the unpacked size is 32 bytes. So, the cache line size is 32 bytes.
5. `__packed` increases the generated code size, because the OpenRISC 32-bit load instruction doesn’t handle unaligned addresses. Extra code needs to be added to assemble the number from individual bytes/halfwords.

Task 1

7. The accesses to `node->id` and `node->next` cause cache misses. They are always on different lines due to the size of `data`. The struct is 64 bytes in total, which is 2 cache lines, and `id` is at the start, while `next` is at the end.

8. Move `next` and `prev` to the start of the struct. This yields 25 dcache misses down from 43. This avoids the cache miss on line 31 because both `id` and `next` are in the same line.

Task 2

9. The data cache misses come from accessing the `id` field of the struct at `items[i]`. Each iteration accesses a different line because the struct is 36 bytes long, while the cache line is 32 bytes.
10. If we use a data pointer, instead of storing the entire array, we can reduce the size of the struct from 36 bytes to just $4 + 8 = 12$ bytes. Doing this, we can fit 3 structs into a single cache line. To achieve this, we just allocate bytes when initializing the struct in *item_init*. This data is therefore somewhere else in memory.

Task 3

11. Row-major and column-major refers to the ordering of a 2D array in memory. In row-major, the row given by the first index (using C double brackets e.g. `A[][]` syntax) is contiguous in memory. In column-major the column given by the second index is contiguous. C uses row-major.
12. Primarily the accesses to out_vector and matrix. Because matrix is row-major each inner iteration loads a different cache line (size of row is 256 elements or 1024 bytes). `out_vector` may be similarly evicted frequently because the entire 1KiB of vector is stored to on every outer loop iteration.
13. Reorder the loops so that `i` is on the outside and `j` inside. This switches things so that `in_vector` is now evicted frequently instead of `out_vector`, but `matrix` is accessed in a cache-friendlier order, so the overall 65893 to 8257.

Task 4

14. It does not work because the writes to the LEDS are being cached. Normally, the I/O should be in an uncached region of memory but this is not the case here.
15.

- Disable the caches entirely by commenting out the body of `init_dcache()`. Alternatively, use `dcache_flush` after every iteration to force the dcache to flush to the bus.
- Disable write-back policy in `init_dcache()` and use write-through instead (remove `CACHE_WRITE_BACK` flag):

```
dcache_write_cfg (CACHE_FOUR_WAY | CACHE_SIZE_4K | CACHE_REPLACE_LRU);
```

This works because write-through means that the access is cached, but it is propagated to the system memory anyway.