Computer Architecture I

Spring, 2022

Homework 6

Reference Solution

ShanghaiTech University

Due: May 7, 2022, 23:59

## Instructions

This homework will help you review CPU caches. Remember to go through the textbook and all the lecture slides related.

## Submission Guideline

- Both hand-writing and typesetting are accepted. You may find a LaTeX template helpful on our course website.
- Only PDF submissions to Gradescope will be counted. If you choose to write by hand, make sure it is transformed into a PDF document. Both scanning and taking photos are accepted.
- Please assign your answers properly on Gradescope. Any submission without proper assignment will result in a 25% point reduction.

# 1 Shut Up and Take My Cache!

In this section, we will review some basics of cache. A program is run on a byte-addressed system with a single-level cache. After a while, the entire cache has the state in Figure 1.

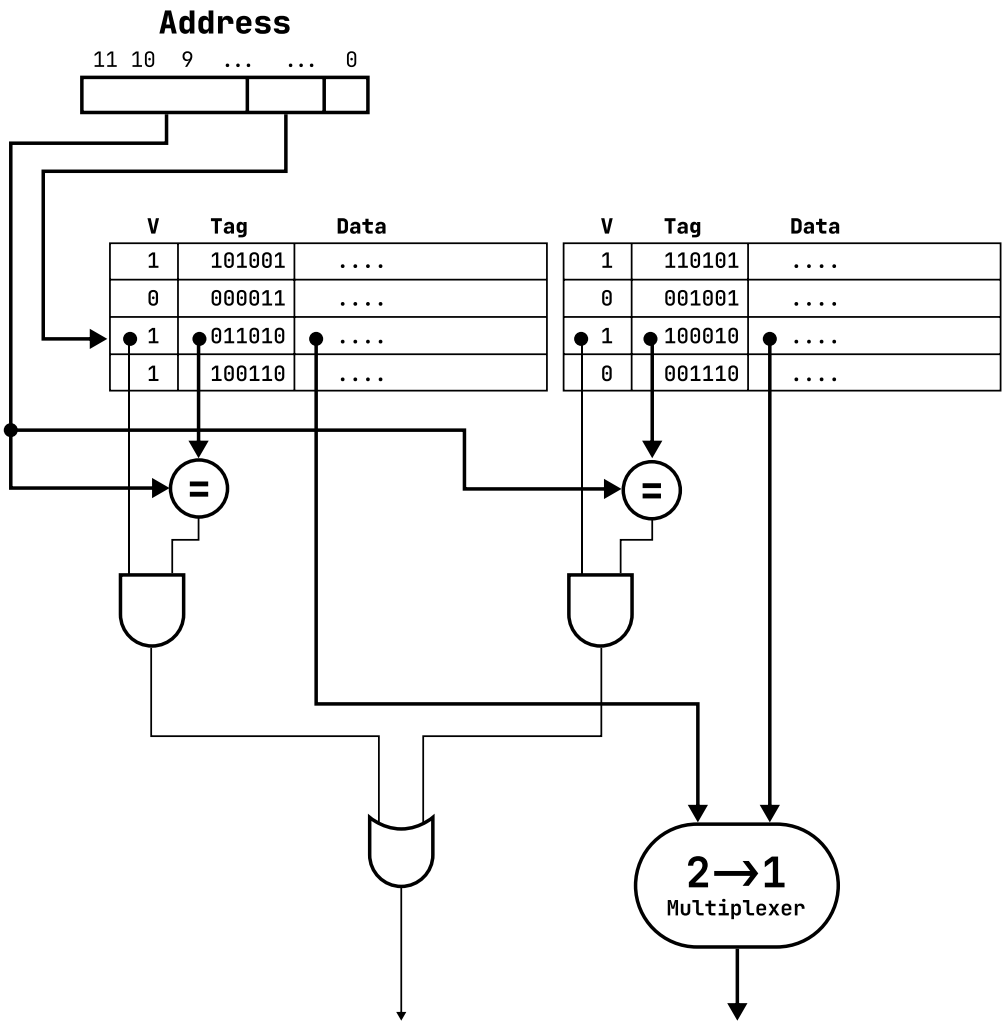



Figure 1: Layout for a cache implementation

1. (2 points) In Figure 1, what is a row stands for?
   A. One set. B. One cache block. C. One entire cache. D. Not listed in choices.

   **Solution:** √ **A** ◯ B ◯ C ◯ D

2. (2 points) In Figure 1, what is `V` stands for?
   A. Valid bit. B. Vacant bit. C. Visited bit.

   **Solution:** √ **A** ◯ B ◯ C

3. (2 points) What is the type of the cache described in Figure 1?
   A. Direct-Mapped cache.
   B. Set-Associative Cache (If you choose this, write down its associativity).
   C. Fully-Associative Cache (If you choose this, write down its associativity).

   **Solution:**
   ◯ A
   √ **B (Associativity: 2)**
   ◯ C (Associativity: _____)

4. (3 points) What is the ( Tag : Set Index : Byte Offset ) breakdown of memory addresses in Figure 1?

   **Solution:**
   1. Tag: **6**
   2. Index: **2**
   3. Byte Offset: **4**

5. (2 points) Is it TRUE that conflict misses cannot occur in fully-associated caches?
   A. Yes. B. No.

   **Solution:** √ **A** ◯ B

6. (3 points) Tell the difference(s) between Conflict Miss and Capacity Miss.

**Solution:** Conflict misses only occur with set-associative or direct-mapped caches, while capacity misses will occur even with full associativity.

Conflict misses are caused when multiple blocks compete for the same set, while capacity misses occur when blocks are replaced and then later retrieved. Conflict misses are caused by set competition, capacity misses occur because caches cannot contain all the blocks needed to satisfy the request. Although they sound very similar, but one is caused by collision and the other one is simply out of cache size.

7. (12 points) For each of the following accesses to the cache described in Figure 1, determine if each access would be a hit or miss based on the cache state shown above. If it is a miss, classify the miss type(s).

   If multiple miss types may exist depending on prior memory accesses, select *all possible* miss types. For each access, you will get

   - 2 points if you choose all correct choice(s),
   - 1 point if you choose partial correct choice(s), and,
   - 0 point if you give no choice(s) or wrong choice(s).

   Each memory access should be considered *dependently*. In particular, *Do update* the cache status after each memory access. If a replacement happens, data in the first slot will always be evicted.

| Order | Address | Access Outcome |
|---|---|---|
| 1 | `0b 101001 000100` | **1** |
| 2 | `0b 011010 110100` | **2** |
| 3 | `0b 111110 101000` | **3** |
| 4 | `0b 000011 111100` | **4** |
| 5 | `0b 000011 011001` | **5** |
| 6 | `0b 100110 101100` | **6** |

**Solution:**

Description: The crux of this question is that data is always set to some garbage (there's no such thing as blank in binary, and we often don't zero out data if we can avoid it; as such, the cache generally ends up containing whatever data used to be there the last time a program ran), even if the cache block is empty. As such, the valid bit is needed to tell if a block is actually there. If the valid bit is 0, then the block is considered empty, regardless of the corresponding tag.

Description: Plus, it should be pointed out that we consider the cache as a simple cache with neither cache maintenance operations nor coherence protocol invalidation request.

1. √ **Hit** ◯ Compulsory Miss ◯ Conflict Miss

   Description: Tag: `0b101001` Set: `0b00`. This is a hit because the tag matches for the given set and the valid bit is on.

2. ◯ Hit √ **Compulsory Miss** ◯ Conflict Miss

   Description: Tag: `0b011010` Set: `0b11`. This is a miss because the tag is not in the set. This is a compulsory miss because the address we load may be accessed for the first time and we still have an empty slot in this index. This is NOT a conflict miss because we only ever kick out a block when the cache is full, and only to replace that block with a new one; as such, we can't have kicked out a block while there is still empty space in the cache.

3. ◯ Hit √ **Compulsory Miss** √ **Conflict Miss**

   Description: Tag: `0b111110` Set: `0b10`. This is a miss because the tag is not in the set. This can either be compulsory or conflict, since we don't know if this block has been accessed before and ejected.

4. ◯ Hit √ **Compulsory Miss** ◯ Conflict Miss

   Description: Tag: `0b000011` Set: `0b11`. It's important to note this access is proceeded by 2, where the cache set still has room. As with 2.

5. ◯ Hit √ **Compulsory Miss** ◯ Conflict Miss

   Description: Tag: `0b000011` Set: `0b01`. As with 2.

6. ◯ Hit √ **Compulsory Miss** √ **Conflict Miss**

   Description: Tag: `0b100110` Set: `0b10`. As with 3.

8. (6 points) The specification sheet of the system is given below. What is the Average Memory Access Time (AMAT) of memory accesses in Question 7? What is the AMAT if we remove this cache? Please give your answer in nanosecond (ns). You shall have the formula, the unit of results and the deriving procedure presented in your answer. (Note: A 1 gigahertz (GHz) processor ticks a cycle for each 1 nanosecond)

| | |
|---|---|
| System Frequency | 2 GHz |
| Cache Access Latency | 2 Cycles |
| Main Memory Access Latency | 600 Cycles |

Table 1: Specification Sheet

**Solution:**

Miss rate: 5/6,

AMAT:

$$2 \text{ (Time for a hit)} + 5/6 \text{ (Miss rate)} \times 600 \text{ (Cycles)} = 502 \text{ Cycles} = 251 \text{ ns}.$$

AMAT If no cache presented:

$$600 \text{ Cycles} = 300 \text{ ns}$$

## 2 Oops . . . Too many bites (bytes)

In this sections, we will review the implementation of caches and replacement policies.

1. (15 points) Let's Draw It Out!

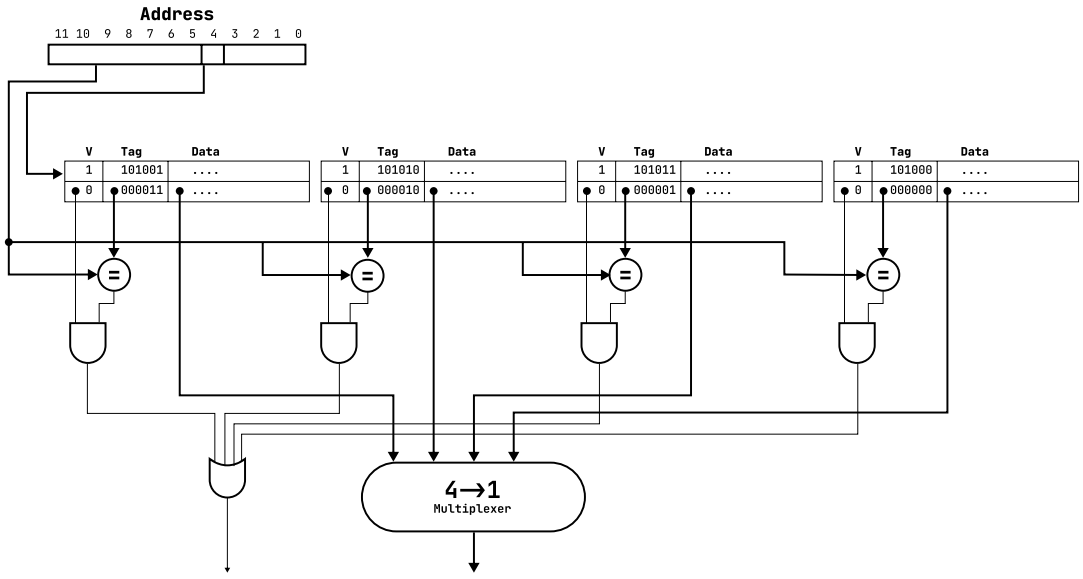
   Sketch the organization of a *four-way set associative* cache with a cache block size of 16 bytes and a total size of 128 bytes. Your sketch should have a style similar to Figure 1. The memory addresses are 12-bit long.

   In your sketch, the following components should be also presented.

   1. the width of set index, tag and data fields of memory addresses (3 points),
   2. logical components used for comparison and selection (2 points),
   3. the type of multiplexer (e.g. $2 \rightarrow 1$, $4 \rightarrow 1$, $8 \rightarrow 1$, $16 \rightarrow 1$, etc.) (1 point),
   4. the number of sets (1 point) , and,
   5. an implementation layout including wiring and placement of cache elements (8 points).

**Solution:**

1. the width of tag, set index, and data fields of memory addresses : 7, 1, 4
   (3 points, both explicit length and bit distribution are accepted),
2. logical components used for comparison and selection: The logical gate components shown below (2 points),
3. the type of multiplexer: $4 \rightarrow 1$ (1 point),
4. the number of sets: 2
   (1 point, both explicit number and implication in the sketch are accepted) , and,
5. an implementation layout including wiring and placement of cache elements
   (8 points. Byte selection inside a cache block is optional. An incorrect byte selection sketch does not result in point reduction but the correctness will be notified).



In the following questions, we will examine how replacement policies affect miss rate.

After the system is cold start, the address sequence of warm-up accesses is:
`0x2A, 0x3C, 0x7D, 0xCE, 0x5B, 0x01, 0x2C, 0x1D, 0x9B, 0x3E`.

After all warm-up accesses are done, the address sequence of follow-up accesses is:
`0x30, 0x40, 0x52, 0x44, 0x56, 0x48, 0x5A, 0x4C, 0x10, 0x3B, 0x5C, 0x30, 0x5E`.

2. (2 points) Which locality(s) can you observe in the follow-up accesses?

**Solution:** Note: There may exist(s) one or more correct choice(s).
√ **Spatial locality** √ **Temporal locality**

Description: For example, spatial locality is given by
`0x30, 0x40, (0x52), 0x44, (0x56), 0x48, (0x5A), 0x4C, 0x10, 0x3B, (0x5C), 0x30, (0x5E)`,
and temporal locality can be observed in
`(0x30), 0x40, 0x52, 0x44, 0x56, 0x48, 0x5A, 0x4C, 0x10, 0x3B, 0x5C, (0x30), 0x5E`.

3. (2 points) Assume *Least Recently Used* (LRU) replacement policy is applied. Circle out all access(es) with cache hit in the follow-up accesses.

**Solution:** Circle the access(es) that meets a cache hit! (like this: (0xFF))

(0x30), 0x40, (0x52), (0x44), (0x56), (0x48),
(0x5A), (0x4C), (0x10), (0x3B), (0x5C), (0x30), (0x5E).

Description: You may categorize all these access into two categories easily: the odd number of the left-most hexadecimal bit always goes to the second set, and the even one goes to the first set. And these two set accesses will not interrupt each other.
For set 0:
Warm-Up: `0x2A, 0xCE, 0x01, 0x2C`
Follow-Up: `0x40, 0x44, 0x48, 0x4C`.
Therefore, except the first `0x40`, the rest will be hit.
For set 1:
Warm-Up: `0x3C, 0x7D, 0x5B, 0x1D, 0x9B, 0x3E`
Follow-Up: `0x30, 0x52, 0x56, 0x5A, 0x10, 0x3B, 0x5C, 0x30, 0x5E`
Therefore, the accesses can be considered as below:
Way 1: `0x3C(0,M) -> 0x9B(4,M)`
Way 2: `0x7D(1,M) -> 0x3E(5,M) -> 0x30(6,H) -> 0x3B(11,H) -> 0x30(13,H)`
Way 3: `0x5B(2,M) -> 0x52(7,H) -> 0x56(8,H) -> 0x5A(9,H) -> 0x5C(12,H) -> 0x5E(14,H)`
Way 4: `0x1D(3,M) -> 0x10(10,H)`

4. (2 points) Assume *Most Recently Used* (MRU) replacement policy is applied. Circle out all access(es) with cache hit in the follow-up accesses.

**Solution:** Circle the access(es) that meets a cache hit! (like this: (0xFF))

(0x30), 0x40, (0x52), (0x44), (0x56), (0x48),
(0x5A), (0x4C), 0x10, (0x3B), 0x5C, 0x30, 0x5E.

Description: For set 0:
Warm-Up: 0x2A, 0xCE, 0x01, 0x2C
Follow-Up: 0x40, 0x44, 0x48, 0x4C.
Therefore, except the first 0x40, the rest will be hit.
For set 1:
Warm-Up: 0x3C, 0x7D, 0x5B, 0x1D, 0x9B, 0x3E
Follow-Up: 0x30, 0x52, 0x56, 0x5A, 0x10, 0x3B, 0x5C, 0x30, 0x5E
Therefore, the accesses can be considered as below:
Way 1: 0x3C(0,M) -> 0x3E(5,H) -> 0x30(6,H) -> 0x3B(11,H) -> 0x5C(12,M) -> 0x30(13,M) -> 0x5E(14,M)
Way 2: 0x7D(1,M)
Way 3: 0x5B(2,M) -> 0x52(7,H) -> 0x56(8,H) -> 0x5A(9,H) -> 0x10(10,M)
Way 4: 0x1D(3,M) -> 0x9B(4,M)

5. (6 points) The specification sheet of the system is given below. What is the Average Memory Access Time (AMAT) of memory accesses in the question 3 and 4? Please give your answer in nanosecond (ns). You shall have the formula, the unit of results and the deriving procedure presented in your answer. (Note: A 1 gigahertz (GHz) processor ticks a cycle for each 1 nanosecond)

| | |
|---|---|
| System Frequency | 2 GHz |
| Cache Access Latency | 2 Cycles |
| Main Memory Access Latency | 130 Cycles |

Table 2: Specification Sheet

**Solution:**

Miss rate: 1/13,

AMAT:

$$2 \text{ (Time for a hit)} + 1/13 \text{ (Miss rate)} \times 130 \text{ (Cycles)} = 12 \text{ Cycles} = 6 \text{ ns}.$$

Miss rate: 5/13,

AMAT:

$$2 \text{ (Time for a hit)} + 5/13 \text{ (Miss rate)} \times 130 \text{ (Cycles)} = 52 \text{ Cycles} = 26 \text{ ns}.$$

# 3 Let's See Some Real World Example

## 3.1 A Statement On Problem 3 Grading on May 7, 2022.

We found a mistake in Problem 3, Homework 6 related to memory alignment, thanks to Peng Cheng.

The structure `log_entry` given in this problem was originally designed with no memory alignment consideration. According to the ISO C standard, *each non-bit-field member of a structure or union object is aligned in an implementation defined manner appropriate to its type.*, where the alignment requires that *objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address*. Therefore, according to `gcc`'s doc, the alignment of any given `struct` or `union` type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union`.

As such, it is correct to have the struct aligned to its greatest alignment requirement, i.e. the 8-byte long. The struct will be like this in a real program:

```
struct log_entry {
    int src_ip;          //   4 Bytes: [  0 -   3]
    char URL[128];       // 128 Bytes: [  4 - 131]
    // Padding           //   4 Bytes: [132 - 135]
    long reference_time; //   8 Bytes: [136 - 143]
    char browser[64];    //  64 Bytes: [144 - 207]
    int status;          //   4 Bytes: [208 - 211]
    // Padding           //   4 Bytes: [212 - 215]
} log[NUM_ENTRIES];
```

However, considering the current situation that the memory alignment in type is beyond our course scope, we hereby announce,

1. the correct answer in Problem 3 done with either padded or unpadded structure format will be equally graded;
2. we will release two versions of answer later on (i.e. in this answer), and,
3. if you choose to write an answer with padded structure, please use the format given above.

## 3.2 Unpadded Version

Each time when you access the course website, your activity will be recorded into our web server logs! This is the definition of the web server log for our Computer Architecture course website. Assume our web server is a 32-bit machine. In this question, we will examine code optimizations to improve log processing speed. The data structure for the log is defined below.

```
struct log_entry {
    int src_ip;     /* Remote IP address */
    char URL[128]; /* Request URL. You can consider 128 characters are enough. */
    long reference_time; /* The time user referenced to our website. */
    char browser[64]; /* Client browser name */
    int status; /* HTTP response status code. (e.g. 404) */
} log[NUM_ENTRIES];
```

Assume the following processing function for the log. This function determines the most frequently observed source IPs during the given hour that succeed to connect our website.

```
topK_success_sourceIP (int hour);
```

1. (2 points) Which field(s) in a log entry will be accessed for the given log processing function?

**Solution:** Note: There may exist(s) one or more correct choice(s).
√ `src_ip` ◯ `URL` √ `reference_time` ◯ `browser` √ `status`

2. (1 point) Assuming 32-byte cache blocks and no prefetching, how many cache misses per entry does the given function incur on average?

**Solution:** **2.25**

Description: The cold start access will get 3 misses. For all accesses, `reference_time` and `status` will always be miss. But for `src_ip`, the situation is complicated. After the cold start, the access of `src_ip` will be hit as the loading of previous entry's `status`. But still, these two accesses may still located in different cache blocks. So the difficult part is to find out which accesses are miss. For $3n$ accesses, there will be $2n$ accesses missed at `reference_time` and `status`. The rest $n$ accesses for `src_ip` should be considered in three situations:

1. Cold start (n=1), this is a miss;
2. Hit with previous `status`;
3. Miss due to different blocks location with previous `status`.

Given the cache block size is 32 bytes, for the `log_entry` with 208 bytes, for $n+1$'s access that falls in the third situation:

$$(208n + 204) \mod 32 > 32 - 8.$$

which gives $n = 2z + 1, z = 0, 1, \ldots$. So the average miss is given by

$$\frac{1}{n}(2n \text{ (reference\_time \& status miss)} + (1 + \lfloor \frac{n}{4} \rfloor) \text{ (src\_ip)}) \to 2.25.$$

3. (3 points) How can you reorder the data structure to improve cache utilization and access locality? Justify your modification.

**Solution:**

```
struct log_entry {
    int src_ip;
    int status;
    long reference_time;
    char URL[128];
    char browser[64];
} log[NUM_ENTRIES];
```

As the fields of `status` and `reference_time` is reordered closer (1 point) to each other, we can observe spatial locality (1 point). And we can reduce the cache miss(es) per entry to 1 since a cache block is 32 bytes long (1 point).

4. (6 points) To mitigate the miss in the question 2, design a different data structure. How would you rewrite the program to improve the overall performance?

   Your answer shall include:

   - A new layout of data structure of our server logs.
   - A description of how your function would improve the overall performance.
   - How many cache misses per entries does your improved design incur on average?

**Solution:**

Grading Criteria:

- For layout, all information from the original structure should be correctly presented with no loss (1 point). And justify the structure is better for cache optimization (1 point)
- For function, state how the function looks like, or name techniques involved (e.g. loop unrolling) (1 point). Qualitatively analyze your function (1 point) and give the misses per entries (2 points).

Reference Solution:

- (Layout) `src_ip`, `reference_time` and `status` and other fields can be saved into separate arrays, thus serial accesses will benefit from previous accesses greatly since all data used will be consecutively loaded.
- (Function) To determine the most frequently observed IP's during the given hour with success state, a most naive implementation would take all logs and iterate on them. To make the function more effecient with the new layout, we can first use binary search to locate the eagered period involved in `reference_time[]` before iteration. For each iteration, due to array's locality, the misses per entry can be reduced to 0.75. (Description: For $n$ entries, $\frac{n}{8}$ misses for either `src_ip` or `status`, and $\frac{n}{4}$ misses for `reference_time`, giving the result approaches to 0.75).

### 3.3 Padded Version

Each time when you access the course website, your activity will be recorded into our web server logs! This is the definition of the web server log for our Computer Architecture course website. Assume our web server is a 32-bit machine. In this question, we will examine code optimizations to improve log processing speed. The data structure for the log is defined below.

```
struct log_entry {
    int src_ip;           //   4 Bytes: [  0 -   3]
    char URL[128];        // 128 Bytes: [  4 - 131]
    // Padding            //   4 Bytes: [132 - 135]
    long reference_time;  //   8 Bytes: [136 - 143]
    char browser[64];     //  64 Bytes: [144 - 207]
    int status;           //   4 Bytes: [208 - 211]
    // Padding            //   4 Bytes: [212 - 215]
} log[NUM_ENTRIES];
```

Assume the following processing function for the log. This function determines the most frequently observed source IPs during the given hour that succeed to connect our website.

```
topK_success_sourceIP (int hour);
```

1. (2 points) Which field(s) in a log entry will be accessed for the given log processing function?

**Solution:** Note: There may exist(s) one or more correct choice(s).
√ `src_ip` ◯ `URL` √ `reference_time` ◯ `browser` √ `status`

2. (1 point) Assuming 32-byte cache blocks and no prefetching, how many cache misses per entry does the given function incur on average?

**Solution:** **2.75**

Description: The cold start access will get 3 misses. For all accesses, `reference_time` and `status` will always be miss. But for `src_ip`, the situation is complicated. After the cold start, the access of `src_ip` will be hit as the loading of previous entry's `status`. But still, these two accesses may still located in different cache blocks. So the difficult part is to find out which accesses are miss. For $3n$ accesses, there will be $2n$ accesses missed at `reference_time` and `status`. The rest $n$ accesses for `src_ip` should be considered in three situations:

1. Cold start (n=1), this is a miss;
2. Hit with previous `status`;
3. Miss due to different blocks location with previous `status`.

Given the cache block size is 32 bytes, for the `log_entry` with 216 bytes, for $n+1$'s access that falls in the third situation:

$$(216n + 208) \mod 32 > 32 - 12.$$

which gives $n = 4z + 3, z = 0, 1, \ldots$. So the average miss is given by

$$\frac{1}{n}(2n \text{ (reference\_time \& status miss)} + (\lfloor \frac{3n}{4} \rfloor) \text{ (src\_ip)}) \to 2.75.$$

3. (3 points) How can you reorder the data structure to improve cache utilization and access locality? Justify your modification.

**Solution:**

```
struct log_entry {
    int src_ip;          //   4 Bytes: [  0 -   3]
    int status;          //   4 Bytes: [  4 -   7]
    long reference_time; //   8 Bytes: [  8 -  15]
    char URL[128];       // 128 Bytes: [ 16 - 143]
    char browser[64];    //  64 Bytes: [144 - 207]
} log[NUM_ENTRIES];
```

As the fields of `status` and `reference_time` is reordered closer (1 point) to each other, we can observe spatial locality (1 point). And we can reduce the cache miss(es) per entry to 1 since a cache block is 32 bytes long (1 point).

Description: As is tested here, there's no padding.

4. (6 points) To mitigate the miss in the question 2, design a different data structure. How would you rewrite the program to improve the overall performance?

   Your answer shall include:

   - A new layout of data structure of our server logs.
   - A description of how your function would improve the overall performance.
   - How many cache misses per entries does your improved design incur on average?

**Solution:** Grading Criteria:

- For layout, all information from the original structure should be correctly presented with no loss (1 point). And justify the structure is better for cache optimization (1 point)
- For function, state how the function looks like, or name techniques involved (e.g. loop unrolling) (1 point). Qualitatively analyze your function (1 point) and give the misses per entries (2 points).

Reference Solution:

- (Layout) `src_ip`, `reference_time` and `status` and other fields can be saved into separate arrays, thus serial accesses will benefit from previous accesses greatly since all data used will be consecutively loaded.
- (Function) To determine the most frequently observed IP's during the given hour with success state, a most naive implementation would take all logs and iterate on them. To make the function more effecient with the new layout, we can first use binary search to locate the eagered period involved in `reference_time[]` before iteration. For each iteration, due to array's locality, the misses per entry can be reduced to 0.75. (Description: For $n$ entries, $\frac{n}{8}$ misses for either `src_ip` or `status`, and $\frac{n}{4}$ misses for `reference_time`, giving the result approaches to 0.5).