CS610: Programming for Performance Assignment 1

Wadkar Srujan Nitin
221212

Solution 1

Elementary calculations

1. **Total capacity in words:**

$$\frac{2^{17}\text{ B}}{2^2\text{ B/word}} = 2^{15}\text{ words} = 32\text{K words}.$$

2. **Number of sets:**

$$\#\text{sets} = \frac{\text{cache size}}{\text{line size} \times \text{associativity}} = \frac{2^{17}}{2^7 \cdot 2^3} = 2^{17-7-3} = 2^7.$$

3. **Address Bit breakdown:**
   - Line size $= 2^7$ B $\Rightarrow$ block offset uses 7 LSBs.
   - #sets $= 2^7 \Rightarrow$ index uses the next 7 bits.
   - The remaining upper bits are the tag.
4. **Word alignment and within-block word index:**
   - Word size $= 2^2$ B $\Rightarrow$ the lowest 2 bits are always 0 for word addresses.
   - Within a block, the *word* index therefore uses $7 - 2 = 5$ bits.
5. **A and B compete in the same sets (same index bits):**
   - Last 16 bits of both addresses are `0x5678`.
   - Hence, for the same array index, $A[i]$ and $B[i]$ pick the same set.
6. **Starting block offset (for A and B):**
   - Byte `0x78` = `0b`0111 1000.
   - 7 LSBs (block offset) = `1111000`.
   - Since accesses are word-aligned, ignore the lowest 2 bits $\Rightarrow$ look at `11110` for the within-block *word* position.
7. **Alignment observation:**
   - The first two word accesses need to be analysed separately (word offset `11110`).
   - After those, we can analyze the rest of the cases in a high level manner.

Per-stride analysis

Stride $= 1$ (in words)

- Initially the cache is empty, so the very first access (block offset `11110`, word-aligned) will be a **miss** for both A and B.
- Since the cache is 8-way set associative, both the cache line for A and the cache line for B can reside in the same set simultaneously.
- The second access of A (word offset `11111`) will be a **hit**, because the entire 32-word line was already brought into the cache.

- From the third access onward (starting with word offset `00000` of the next block), we observe a **miss once per block** of 32 words.
- Therefore, within a single traversal of A, the total number of misses is:

  $$\left\lceil \frac{\text{SIZE}-2}{32} \right\rceil + 1,$$

  where SIZE is the number of words in array A.

Effect of outer loop iterations:

- The cache capacity is 32K words, but each array A and B individually has SIZE = 32K words.
- With LRU replacement, after one full traversal (inner loop), the cache will contain only the *last half* of each array, and the first half will be evicted.
- Consequently, the next outer loop iteration cannot reuse cached data; it suffers the same miss pattern as the first traversal.
- Thus, for N outer loop iterations, the total misses are:

  $$N \cdot \left(\left\lceil \frac{\text{SIZE}-2}{32} \right\rceil + 1 \right).$$

- As $N = 1000$ and SIZE = 32K words:

  $$1000 \cdot \left(\frac{32\text{K} - 2}{32} + 1 \right) = 1000 \cdot 1025.$$

Stride = 16

- The first access (block offset `11110`) will be a **miss**.
- With stride = 16 words ($= 2^4$), the block offset advances as:

  $$11110 \;+\; 10000 = 01110, \quad 01110 \;+\; 10000 = 11110,$$

  and so on. Thus, the accesses alternate between offsets `11110` and `01110`.
- In this alternating pattern, every *second access* is a **hit**, because the line fetched by the previous access is reused.
- Therefore, within a single traversal of A:

  $$\text{Total accesses} = \frac{\text{SIZE}}{\text{stride}} = \frac{32K}{16} = 2K.$$

  The total number of misses is:

  $$\left\lceil \frac{(2K-1)}{2} \right\rceil + 1 = 1025.$$

Effect of outer loop iterations:

- Even though only every 16th element is accessed, every cache line of the array is eventually brought into cache, because each line is touched once.
- As in the stride = 1 case, after one inner loop traversal the cache will contain only the *last half* of A and B (due to capacity and LRU replacement).
- Thus, the next outer loop iteration cannot reuse cached data; the same miss pattern repeats.
- For N outer loop iterations, the total number of misses is:

  $$N \cdot 1025.$$

- As $N = 1000$ the number of misses is:

  $$1000 \cdot 1025.$$

Stride $= 32$

- With stride $= 32$ words, each access touches only one word of a cache line but brings the entire cache line into the cache.
- Since subsequent accesses are to different cache lines, **every access is a miss**.
- Therefore, within a single traversal of A, the total number of misses is:

$$\frac{\text{SIZE}}{\text{stride}} = \frac{32K}{32} = 1024.$$

Effect of outer loop iterations:

- As in the stride $= 16$ case, even though only one word per line is accessed, every cache line of the array is brought into the cache once.
- With both A and B present and LRU replacement, at the end of one inner loop iteration the cache contains only the last half of both arrays.
- Thus, no useful reuse occurs across outer loop iterations, and the miss pattern repeats each time.
- For N outer loop iterations, the total number of misses is:

$$N \cdot 1024.$$

- As $N = 1000$ the number of misses is:

$$1000 \cdot 1024.$$

Stride $= 64$

- With stride $= 64$ words, consecutive accesses are 64 words ($=$ 2 cache lines) apart.
- This means each access skips one cache line and touches only every other cache line of the array.
- As a result, only **half of the cache lines** of the array are accessed.
- But because we also skip every other set, the number of sets effectively used is halved. Hence, the **effective cache capacity is reduced by half**.
- By arguments similar to the stride $= 32$ case, each accessed cache line causes a miss (no reuse across outer loop iterations).
- Therefore, the total number of misses is:

$$\frac{\text{SIZE}}{64} \cdot N = 512 \cdot 1000.$$

Stride $= 2\,\text{K}$

- With stride $= 2K$ words, each access skips

$$\frac{2K}{32} = 2^6$$

cache lines as well as sets.

- Hence, both the **effective array size** and the **effective cache capacity** are reduced by a factor of 2^6.
- By the same reasoning as the stride $= 64$ case, each accessed line causes a miss.
- Therefore, the total number of misses is:

$$\frac{\text{SIZE}}{2K} \cdot N = \frac{32K}{2K} \cdot N = 16 \cdot 1000.$$

Stride $= 8\,\mathrm{K}$

- With stride $= 8K$ words, each access skips
$$\frac{8K}{32} = 2^8$$
cache lines and sets.
- But the cache has only 2^7 sets. Therefore, all accesses map to the **same set**.
- The number of distinct cache lines accessed is:
$$\frac{\text{SIZE}}{8K} = \frac{32K}{8K} = 4.$$
- Thus, during the first traversal of A, we see 4 compulsory misses (one per distinct line).
- Since the cache is 8-way associative, both A and B can store their 4 lines each in the same set without evictions.
- Hence, in subsequent outer loop iterations, **all accesses hit**, and no further misses occur.
- Therefore, the total number of misses is simply:
$$4.$$

Solution 2

Direct Mapped Cache with $k-i-j$ Loop

Cache Misses Table

Loop	**A**	**B**	**C**
k	N	N	N
i	N	1	N
j	1	$N/16$	$N/16$
Total	N^2	$N^2/16$	$N^3/16$

Table 1: Cache misses for arrays A, B, and C in the $k-i-j$ loop order under a direct-mapped cache.

Explanation of Entries

- **Array A:**
  - k: In a direct-mapped cache, each cache line maps to exactly one set. During the i-loop, each consecutive access skips $1K/16 = 2^6$ cache sets. The cache has $64K/16 = 2^{12}$ sets in total, so we effectively use $2^{12}/2^6 = 2^6$ sets. The i-loop makes 2^{10} total accesses, so previously loaded lines are evicted every 2^6 accesses. Therefore, the k-loop sees the same miss pattern each time, and the table entry is N.
  - i: The size of cache line is 16 words. Each row has 1024 words. Hence moving to the next row would require bringing a new cache line causing a **cache miss** each time. Hence this entry is N
  - j: This loop does not affect A. Hence 1
- **Array B:**
  - k: The miss access pattern will repeat as we bring a new row of B per iteration of this loop and we already know that a row cannot fit in a cache line. Hence the entry is N.
  - i: Here no sets are skipped as the accesses are from a particular row and the entire row fits in cache (1024 words). Hence the miss pattern does not repeat and the entry is 1
  - j: Here, the words in the row are accessed consecutively. Hence we will encounter a **miss** per 16 words as the entire cache line is brought. Therefore, the entry is $N/16$
- **Array C:**
  - k: The cache size is 64K words but the matrix size is 1024K words. Hence the previous lines will be evicted totally from the cache resulting in a repeated pattern and hence this entry is also N
  - i: Here we bring a new row per iteration. Thus the miss pattern repeats and the entry is N.
  - j: Same explanations as j for Array B

Fully Associative Cache with $k-i-j$ Loop

Cache Misses Table

Loop	A	B	C
k	$N/16$	N	N
i	N	1	N
j	1	$N/16$	$N/16$
Total	N^2	$N^2/16$	$N^3/16$

Table 2: Cache misses for arrays A, B, and C in the $k-i-j$ loop order under a fully associative cache.

Explanation of Entries

- The explanations remain same for all entries except for the k loop of Array A as they are independent of the associativity of the cache. For the k loop of Array A we see that all the 1024 cache lines accessed by the inner loop stay in the cache as the associativity of the cache would be $64K/16 = 2^{12}$ now and $1024 < 2^{12}$. Thus when we go to the next iteration of the k loop we move to the next word of the cache line which is already present. This happens until the entire cache line is exhausted. Hence, we encounter one cache miss per 16 accesses and the entry is $N/16$.

Direct Mapped Cache with $j-k-i$ Loop

Cache Misses Table

Loop	**A**	**B**	**C**
k	N	N	N
i	N	N	N
j	N	1	N
Total	N^3	N^2	N^3

Table 3: Cache misses for arrays A, B, and C in the $j-k-i$ loop order under a direct-mapped cache.

Explanation of Entries

- **Array A:**
  - j: The cache size is 64K words but the matrix size is 1024K words. Hence the previous lines will be evicted totally from the cache resulting in a repeated pattern and hence this entry is N
  - k: In a direct-mapped cache, each cache line maps to exactly one set. During the i-loop, each consecutive access skips $1\text{K}/16 = 2^6$ cache sets. The cache has $64\text{K}/16 = 2^{12}$ sets in total, so we effectively use $2^{12}/2^6 = 2^6$ sets. The i-loop makes 2^{10} total accesses, so previously loaded lines are evicted every 2^6 accesses. Therefore, the k-loop sees the same miss pattern each time, and the table entry is N.
  - i: The row is made up of 1024 word but the cache line is made up of 16 words. Hence to go to the next row we would have to bring a new cache line resulting in a **cache miss** each time. Hence the entry is N
- **Array B:**
  - j: Same explanation as k loop for Array A
  - k: Same explanation as i loop for Array A
  - i: The array is not affected by this loop. Hence the entry is 1
- **Array C:**
  - j: This is also similar to the case of k loop for array A as the previously fetched cache lines get evicted and the end of the inner loop iterations we will see the newly fetched lines instead of the earlier ones. Hence this entry is also N
  - k: Same explanation as k loop for Array A
  - i: Same explanation as i loop for Array A

Fully Associative Cache with $j-k-i$ Loop

Cache Misses Table

Loop	**A**	**B**	**C**
k	N	$N/16$	$N/16$
i	$N/16$	N	1
j	N	1	N
Total	N^2	$N^2/16$	$N^3/16$

Table 4: Cache misses for arrays A, B, and C in the $k-i-j$ loop order under a fully associative cache.

Explanation of Entries

- The explanations for the entires will only be different for the ones which had value N due to the use of a particular subset of cache sets. We will argue about them separately over here.
- **Array A:**
  - k: Here the 1024 cache lines can stay in the cache simultaneously as the associativity of the cache is 2^12 which is greater than 1024. Thus the entire cache line will be used in subsequent iterations of the k loop once it is brought in the cache which would cause a single **cache miss**. Thus the entry is $N/16$
- **Array B:**
  - j: Same explanation as k loop for Array A
- **Array C:**
  - j: Here the k loop does not change the state of the cache and hence we can ignore it and give similar explanation to k loop for Array A
  - k: Here the 1024 cache lines brought in due to the inner loop stay in the cache due to the reason of associativity being larger. In this loop we do not change the column and keep iterating over the rows for same column index. This does not cause cache miss as the relevant cache lines are already present in the cache. Hence the entry is 1.

Solution 3

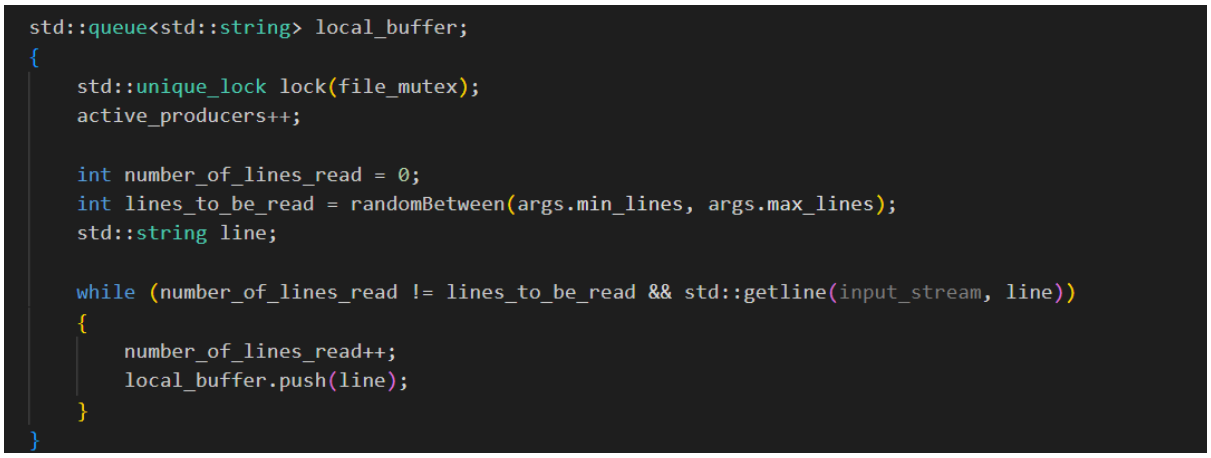
Compilation and Execution

The program is compiled using the `make` command. It is then executed by running the generated binary with the required arguments (input file, number of producers, minimum number of lines, maximum number of lines, buffer size in lines, and output file), as shown below:

```
./bin/problem3.out input.txt 50 3 6 5 out.txt
```

The implementation primarily uses two mutexes — `buffer_mutex` and `file_mutex` — along with one atomic variable `working_thread` for synchronization. The code can be divided into three major parts:

1. Producer Thread: Reading from Input File



```
std::queue<std::string> local_buffer;
{
    std::unique_lock lock(file_mutex);
    active_producers++;

    int number_of_lines_read = 0;
    int lines_to_be_read = randomBetween(args.min_lines, args.max_lines);
    std::string line;

    while (number_of_lines_read != lines_to_be_read && std::getline(input_stream, line))
    {
        number_of_lines_read++;
        local_buffer.push(line);
    }
}
```

A producer thread acquires the lock on `file_mutex` to read its share of L lines from the input file. Once the lines are read, it releases the lock and stores the data into a local buffer.

2. Producer Thread: Writing to Shared Buffer

For writing, a producer acquires the lock on `buffer_mutex`, assigns the `working_thread` variable to its own thread ID, and resets it only after completing its share of L lines.

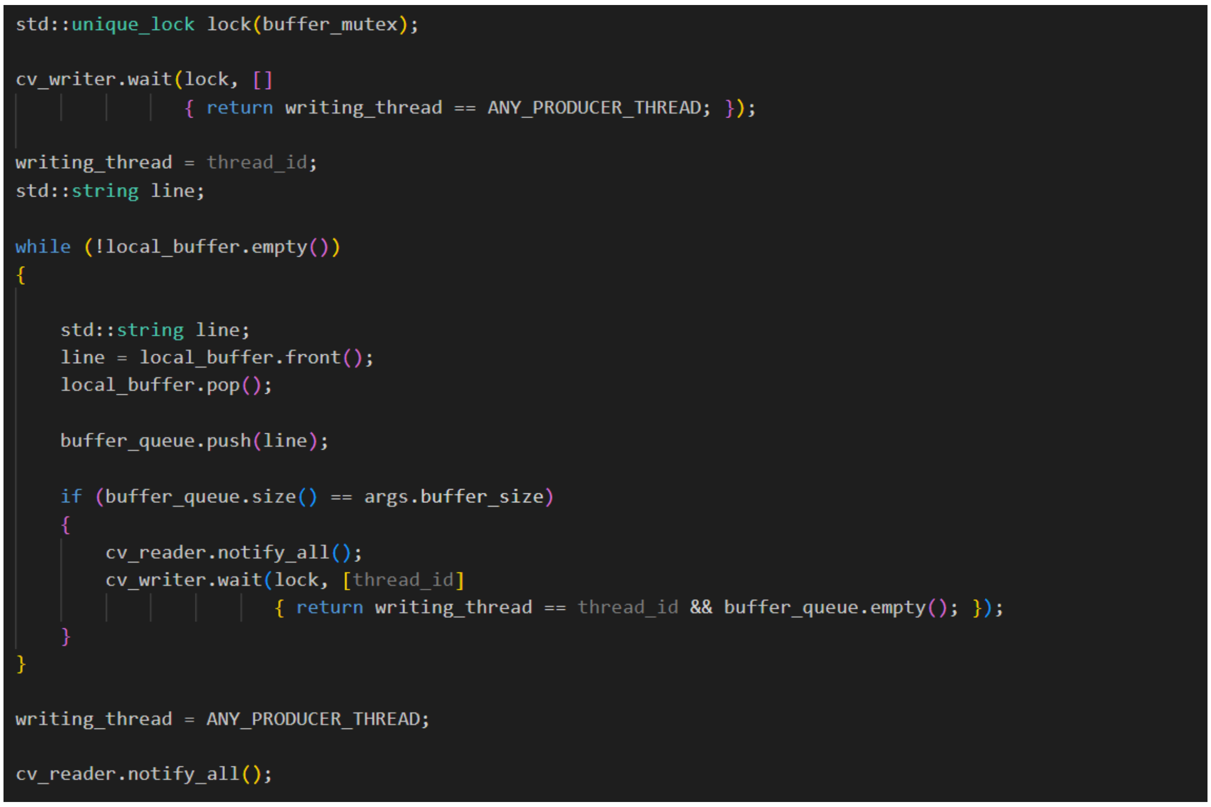
- The case $L > M$ is handled using the `working_thread` variable.
- Once a producer writes M lines to the shared buffer, it releases the lock and goes into a wait state through the condition variable `cv_writer`.
- Only consumer threads can then acquire the buffer lock. Other producer threads are forced to wait until the `working_thread` variable is reset. The active producer also waits until the buffer becomes empty, which indicates that a consumer has read the entire buffer.

3. Consumer Thread: Reading from Shared Buffer

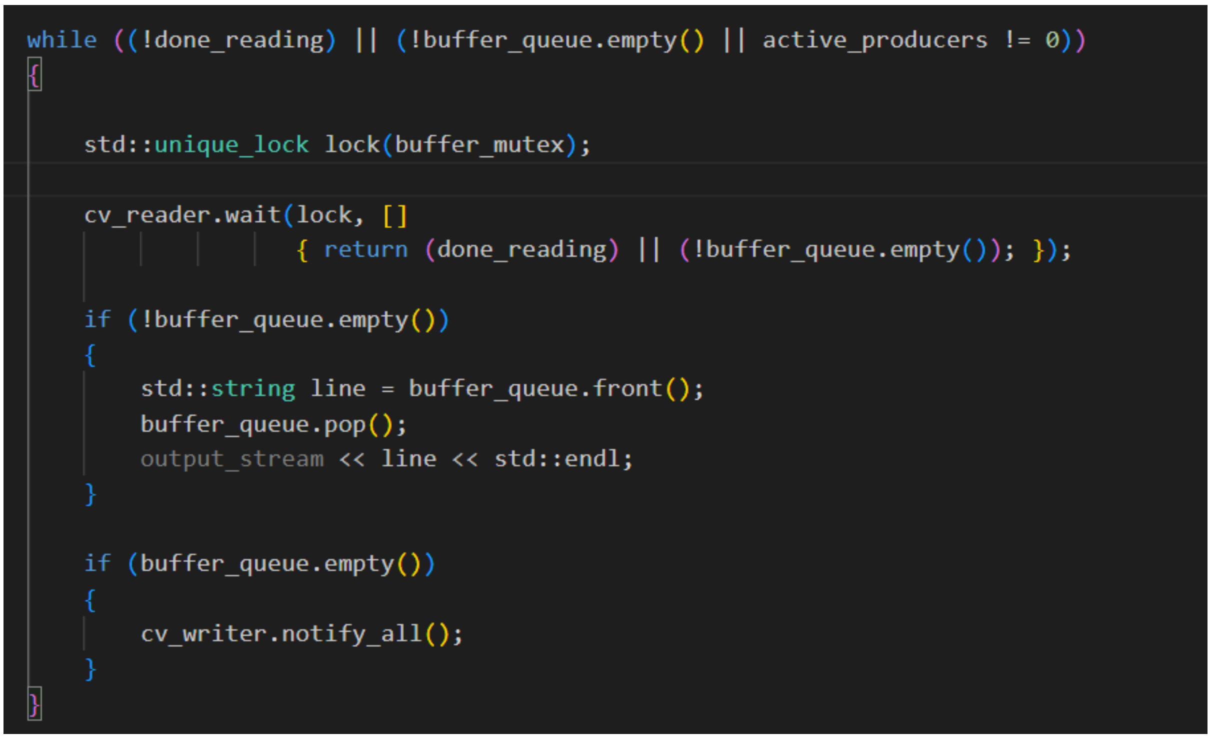
The consumer acquires the lock on `buffer_mutex` to read data from the shared buffer. This ensures that only one thread (producer or consumer) operates on the buffer at any given time. The consumer writes the data to the output file until the buffer becomes empty, after which it notifies the producers and waits until new data is available.

Concurrency Between File and Buffer Operations

The separation of locks ensures concurrency: reading from the input file (protected by `file_mutex`) can happen simultaneously with writing to the shared buffer (protected by `buffer_mutex`).



```cpp
std::unique_lock lock(buffer_mutex);

cv_writer.wait(lock, []
                { return writing_thread == ANY_PRODUCER_THREAD; });

writing_thread = thread_id;
std::string line;

while (!local_buffer.empty())
{

    std::string line;
    line = local_buffer.front();
    local_buffer.pop();

    buffer_queue.push(line);

    if (buffer_queue.size() == args.buffer_size)
    {
        cv_reader.notify_all();
        cv_writer.wait(lock, [thread_id]
                        { return writing_thread == thread_id && buffer_queue.empty(); });
    }
}

writing_thread = ANY_PRODUCER_THREAD;

cv_reader.notify_all();
```



```cpp
while ((!done_reading) || (!buffer_queue.empty() || active_producers != 0))
{

    std::unique_lock lock(buffer_mutex);

    cv_reader.wait(lock, []
                    { return (done_reading) || (!buffer_queue.empty()); });

    if (!buffer_queue.empty())
    {
        std::string line = buffer_queue.front();
        buffer_queue.pop();
        output_stream << line << std::endl;
    }

    if (buffer_queue.empty())
    {
        cv_writer.notify_all();
    }
}
```