High Performance Logging Library for Run-Time Efficiency with Multithreaded Support

BY

Low Chun Ee

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

JUNE 2024

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: **High Performance Logging Library for Run-Time Efficiency with Multithreaded Support**

Academic Session: **JUNE 2024**

I _____**LOW CHUN EE**_____

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in
Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

_____

(Author's signature)

_____

(Supervisor's signature)

Address:
_63, Lorong Jasa Intan 7, _____
_Taman Jasa Intan, 14000_____
_Bukit Mertajam, Pulau Pinang____

___Ts. Wong Chee Siang____
(Supervisor's name)

Date: _____12/9/2024_____

Date: ___12/9/2024_____

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

Date: 12/9/2024

SUBMISSION OF FINAL YEAR PROJECT /DISSERTATION/THESIS

It is hereby certified that Low Chun Ee (ID No: ***2106572***) has completed this final year project entitled " High Performance Logging Library for Run-Time Efficiency with Multithreaded Support" under the supervision of Ts. Wong Chee Siang (Supervisor) from the Department of Computer Science, Faculty of Information and Communication Technology , and Mr Tan Chiang Kang @ Thang Chiang Kang (Co-Supervisor)* from the Department of Computer Science, Faculty of Information and Communication Technology.

I understand that University will upload softcopy of my final year project in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

(*Student Name*)
Low Chun Ee

*Delete whichever not applicable

DECLARATION OF ORIGINALITY

I declare that this report entitled "**High Performance Logging Library for Run-Time Efficiency with Multithreaded Support**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : 

Name : ___Low Chun Ee_____

Date : _____12/9/2024_____

ACKNOWLEDGEMENTS

I would like to express thanks and appreciation to my supervisor, Ts. Wong Chee Siang and my moderator, Mr Tan Chiang Kang @ Thang Chiang Kang, who have given me a golden opportunity to involve in the software programming field study. Besides that, they have given me a lot of guidance in order to complete this project. When I was facing problems in this project, the advice from them always assists me in overcoming the problems. Again, a million thanks to my supervisor and moderator.

ABSTRACT

This project falls within the field of Software Engineering, specifically focusing on the development of a high-performance logging library optimized for run-time efficiency with multithreaded support. The primary issues addressed in this work are enhancing user-friendliness, maximizing performance, and making source code to be easier to understand. To tackle these challenges, the methodology involves utilizing the fmt library in C++, implementing a multiple-producer, multiple-consumer (MPMC) lock-free queue, leveraging advanced techniques such as futex for efficient synchronization, io_uring for asynchronous I/O operations, and C++ template metaprogramming for compile-time optimizations. The research process encompassed designing, implementing, and testing these components to ensure both usability and performance. The final product is a robust and efficient logging library written in C++, which demonstrates significant improvements in both usability, execution speed, and understandability compared to existing solutions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

1.1 Problem Statement and Motivation

In the world of software, logging is a bit like notebook. It's a way for programs to keep track of everything that happens while they run. This helps developers understand what their programs are doing, find problems, and make improvements. However, just like a well-organized notebook, developers need efficient tools for logging. Traditional methods of logging can be slow and complicated. My logging library aims to solve this by making the process faster, easier to use, and simpler to understand.

```cpp
#include <fstream>
#include <string>

int main() {
    std::ofstream logfile("log.txt");

    int userId = 42;
    std::string errorMessage = "File not found";

    logfile << "User ID: " << userId << ", Error: " << errorMessage << std::endl;

    // More code...

    logfile.close();
    return 0;
}
```

Figure 1.1.1 Example of traditional logging method in C++

Based on the image shown above which is a traditional method used to log a message into a file in C++, this approach actually works but can get cumbersome, especially as the complexity of the messages increases. For example, the users required to type << operators multiple times to insert their desired variables into the logging message.

```
User ID: 42, Error: File Not Found
```

Figure 1.1.2 Output of the code

In traditional logging methods, developers have to manually manage file I/O which can be cumbersome and slow. I/O operations particularly when performed repeatedly are expensive in terms of both time and system resources. This becomes a bottleneck especially when logging is done frequently during the execution of high-performance applications. The traditional approach of using streams such as fstream in C++ forces developers to manually format messages using << operators which not only increases code verbosity but also hampers readability and maintainability.

A well-designed logging library addresses these concerns by providing an easy-to-use interface that reduces the code needed to log messages. Instead of manually handling file operations or formatting, users can leverage a single function to log a message. Additionally, a logging library optimizes I/O operations often by batching writes or handling them asynchronously thereby improving performance significantly without compromising the accuracy of the logs.

The need for a logging library arises from the desire to simplify the process of tracking program behavior while maintaining high performance. Instead of manually managing log files and message formatting, developers can focus on writing efficient code knowing that the logging library will handle these aspects. By abstracting away the complexity of I/O and message formatting, the library reduces the cognitive load on developers allowing them to log crucial information with minimal effort.

```
int main() {
    logging::Log log;
    int user_id = 42;
    std::string error_message = "File not found";

    log.setOutputFile("output.txt");

    log.error("User ID: {}, Error message: {}", user_id, error_message);
    return 0;
}
```

Figure 1.1.3 Example of logging library that achieve the same output with Figure 1.1.2

Moreover, a logging library enhances the scalability of the application. As the system grows and becomes more complex, traditional logging methods may no longer be feasible due to the

sheer volume of logs, require more << operator use, and the performance impact of frequent file writes. A robust logging library mitigates this by optimizing logging at scale ensuring that performance remains high even as the volume of logs increases. In essence, a logging library is not just about convenience. It's about ensuring that logging remains efficient, high-performing, and user-friendly. However, there isn't a complete logging library for C++ programming that balances excellent performance with easier to learn and user-friendliness.

This project attempts to produce a logging library that offers excellent performance, easier to learn, and user-friendly APIs by taking on these difficulties head-on. Moreover, developers will be able to easily include logging into their applications with the help of clear documentation and user-friendly APIs which will make debugging, monitoring, and performance analysis jobs easier.

Furthermore, the motivation of this project is also hope that this libary can make modern C++ programming techniques more accessible to a wider audience. The goal is to equip enthusiasts and students with the necessary tools to better understand modern C++ by creating a logging library with an easy-to-use interface and organized source code. This project presents an opportunity to close the knowledge gap between theory and practice, giving young programmers a better understanding of modern C++ ideas.

1.2 Objectives

The primary aim of this project is to

- **enhance the user-friendliness of the logging framework by incorporating the fmt library.**

The fmt library provides a convenient and intuitive interface for formatting log messages and allow developers to express complex logging statements in a readable manner. By leveraging the fmt library, the logging process can be streamlined and improve developer productivity by simplifying the creation and customization of log messages.

- **implement logging framework using C++ template metaprogramming techniques to optimize code generation during compile time.**

By leveraging C++ template metaprogramming, efficient and type-safe logging code at compile time are generated and hence minimizing runtime overhead and maximizing

performance. This approach enables developers to seamlessly integrate logging functionalities into their applications without sacrificing runtime efficiency.

- **address concurrency challenges in multi-threaded environments by utilizing multi-producer multi-consumer queue in C++**

To manage logging operations in a multi-threaded environment, the project implements an MPMC queue, allowing multiple threads to produce and consume log messages concurrently. By employing this approach, the logging framework can efficiently handle high levels of concurrency without creating bottlenecks. Each thread writes to a shared queue which ensures that log messages are processed in a thread-safe manner. This approach significantly reduces contention between threads and ensures that the logging system can scale effectively across multiple cores.

- **enhance performance and efficiency by implementing an asynchronous I/O through io_uring**

The project leverages io_uring which is a modern Linux API that allows for efficient asynchronous I/O operations. This implementation minimizes the need for blocking I/O calls and enable the logging library to perform non-blocking writes to disk. By reducing the dependency on traditional locking mechanisms and using lock-free data structures where possible, the library ensures that I/O operations can be handled swiftly and with minimal overhead.

Furthermore, it's also critical to define what is and is not included in this project. First of all, it will only support Linux and not other operating systems. Furthermore, there will be no support for compatibility with other programming languages, and the library's capabilities will be limited to the C++ language.

In conclusion, the goal of this project is to provide a high-performance, user-friendly, and easier to learn C++ logging library. The development of a logging tool that is both user-friendly and performs better than its competitors is essential to this project. Additionally, the project aims to improve user accessibility to modern C++ programming methods by providing a more user-friendly and streamlined source code, which should make it easier for users to understand modern C++ concepts and the underlying computer architecture.

1.3 Project Scope and Direction

The primary scope of this project is to develop a C++ logging library that focuses on three key aspects which are user-friendliness, high performance, and ease of learning. The library is intended for developers who require a robust logging solution that is simple to integrate into their projects without the need for extensive configuration or prior knowledge of advanced logging techniques. The goal is to make the library accessible to developers at all skill levels and provide a streamlined interface that minimizes the amount of code needed to set up logging while still offering powerful performance features.

A major focus of the project will be the API's user-friendliness. The library will feature a minimalistic and intuitive API that allows developers to quickly integrate logging into their applications with minimal boilerplate. The API will be designed to be as simple as possible to enable developers to log messages, errors, and other relevant information with just a few lines of code.

Another critical aspect of the project is performance optimization. The library will leverage advanced C++ features such as io_uring to handle asynchronous and non-blocking I/O operations efficiently. By implementing multi-threaded logging with mechanisms such as MPMC queues and atomic operations, the library will be able to handle high-concurrency environments without creating performance bottlenecks.

The learning curve of the library will also be kept low with a focus on simplicity in both usage and understanding. The project will include thorough documentation and real-world examples to help developers quickly learn how to use the library even if they have limited experience with C++ logging systems. The library will be designed to be easy to understand and adopt

with clear and concise code that follows modern C++ practices. Integration with standard tools like CMake will ensure that developers can start using the library quickly and without friction.

The overall direction of this project is centered on creating a logging library that prioritizes developer experience, performance, and simplicity. By abstracting away the complexity of asynchronous I/O and multi-threaded logging, the library will provide users with a fast, reliable, and easy-to-learn solution that fits into a wide variety of projects. With its combination of high performance and ease of use, the library will serve developers who need a powerful yet simple logging solution that they can trust to handle the demands of modern software development.

1.4 Contributions

First of all, the logging library offers a useful tool to help different industries optimize their logging processes including system diagnostics and bug tracking. This efficiency results in improved software development productivity and reliability which benefits the technology and healthcare sectors. Additionally, the library is an invaluable teaching tool that provides a practical introduction to modern C++ programming techniques. The program encourages a more knowledgeable workforce by enabling students and aspiring developers to gain a deeper understanding of computer architecture and C++ by offering a simple yet effective tool for logging.

The project's underlying complexity and significance derive from the way it bridges the gap between basic computer understanding and real-world software engineering issues. The project is extremely satisfying and educational because it takes an extensive knowledge of computer architecture to tackle problems like concurrency control and performance optimization.

This project provides an attracting investigation of advanced approaches for readers who are interested in delving into the complexities of software optimization and system-level programming. Within the framework of practical software engineering problems, the project

offers an in-depth examination of optimization techniques for reducing mutex lock overhead, multi-producer multi-consumer queue, futex system call, and asynchronous I/O which further emphasizes its significance in the rapidly evolving software development industry. As a result, readers will be able to learn insightful things about innovative methods and tools that could influence software engineering in the future.

1.5 Report Organization

This report is organized into six chapters: Chapter 1 Introduction, Chapter 2 Literature Review, Chapter 3 System Design, Chapter 4 System Implementation and Testing, Chapter 5 System Outcome and Discussion, and Chapter 6 Conclusion. The first chapter provides an introduction to this project, which includes the problem statement, project background and motivation, project scope, project objectives, project contribution, and the organization of the report. The second chapter presents a literature review of existing logging libraries and concurrency handling techniques and assessing their strengths and weaknesses to establish a foundation for this project's approach. The third chapter discusses the overall system design, including the architecture of the high-performance logging library and the rationale behind key design decisions. The fourth chapter details the implementation process that explain the integration of multithreading, concurrency management, and asynchronous I/O operations within the system. Furthermore, the fifth chapter reports the results obtained from testing, evaluates the performance of the logging library, and discusses the implications of the findings. The final chapter concludes the report by summarizing the project's outcomes, discussing its contributions to the field, and suggesting potential future work.

Chapter 2 Literature Review

2.1 Review of the Technologies

2.1.1 Hardware Platform

This project is developed on an Intel Core i5 processor which is a mid-range CPU known for balancing performance and power efficiency. Intel's Core i5 processors are widely used in personal computers to offer sufficient computational resources for multi-threaded operations

and concurrent workloads. The i5 architecture includes multiple cores and supports hyper-threading in order to allow the system to handle several threads in parallel which is essential for a multi-threaded logging system. Additionally, the processor's integrated cache and memory bandwidth provide fast access to frequently used data which can further improving the performance of concurrent applications like a logging library. The project leverages these multi-core capabilities to ensure efficient execution of log writes particularly when using technologies like io_uring that benefit from parallelism and non-blocking I/O operations.

2.1.2 Firmware/OS

The development environment is set up using Windows Subsystem for Linux 2 (WSL2) which enables the use of a Linux kernel within a Windows operating system. Specifically, Ubuntu is used as the Linux distribution within WSL2. This configuration allows the project to harness the full power of the Linux kernel including system calls such as io_uring for high-performance asynchronous I/O. WSL2 provides near-native performance by running a real Linux kernel in a lightweight virtual machine making it suitable for tasks that require direct access to kernel-level resources. Ubuntu is one of the most popular and developer-friendly Linux distributions. It offers a stable and consistent environment for compiling and running C++ applications. Using WSL2 not only provides flexibility in development but also facilitates the testing and debugging of Linux-specific features without the need for dual-booting or using a dedicated Linux machine. This setup enables seamless integration of Linux's advanced I/O capabilities with the convenience of a Windows-based development workflow.

2.1.3 Programming Languages

The project is developed in C++, chosen for its combination of high performance, low-level memory management, and extensive concurrency support. C++ offers fine-grained control over hardware resources making it ideal for building a high-performance logging library where efficiency and speed are critical. C++20, the latest version used in this project introduces several new features that enhance both the performance and usability of the logging system.

One significant feature in C++20 is the concept mechanism which allows developers to define constraints on template parameters. Concepts improve code readability and maintainability by

ensuring that only valid types are passed to templates leading to better compile-time error checking. In the context of this project, concepts are used to ensure that only types capable of safe concurrency operations are passed into the logging system's templates particularly in areas such as the multi-producer, multi-consumer (MPMC) queue. This makes the code safer, more robust, and easier to debug.

The consteval keyword is another powerful feature of C++20 that enable compile-time execution of certain functions. By utilizing consteval, the project can optimize critical parts of the logging system such as constant expression evaluation for configurations which are computed at compile time rather than runtime. This reduces overhead in the final binary and speeds up the execution of the logging library. For example, compile-time calculations of buffer sizes or configuration settings ensure that the logging system is more efficient and tailored for high-performance scenarios.

Another key feature leveraged in the project is the source_location facility introduced in C++20 which allows the automatic capture of file names, line numbers, and function names where a log is generated. This feature is critical in a logging library as it provides context for log entries without requiring manual inputs from developers. By incorporating std::source_location, the logging system automatically appends relevant debugging information to log entries to enhance the system's debugging capabilities while maintaining a simple and user-friendly API.

In addition to these C++20 specific features, the project makes extensive use of atomic operations to implement a lock-free, thread-safe logging system. This approach avoids traditional locking mechanisms like mutexes which can lead to performance bottlenecks in multi-threaded environments. By using atomic operations, the system ensures that logging operations such as enqueueing and dequeueing log messages, are safely performed across multiple threads with minimal overhead.

The combination of C++20’s modern features such as concepts for safe template programming, consteval for compile-time optimization, and source_location for enhanced logging metadata along with C++'s traditional strengths in performance and concurrency, makes it an ideal choice for developing a high-performance, user-friendly logging library. These features contribute to a logging system that is not only efficient but also easy to use and maintain.

2.1.4 Algorithm

2.1.4.1 Multi-Producer Multi-Consumer Queue

The MPMC (Multiple Producer, Multiple Consumer) queue is a data structure designed to allow multiple threads to add (produce) and remove (consume) items concurrently without using locks. This makes it highly efficient for scenarios where many threads need to interact with the queue simultaneously. The MPMC queue doesn't rely on traditional locking mechanisms (like std::mutex). Instead, it uses atomic operations to manage access which allows threads to work on the queue without blocking each other. This reduces overhead and improves performance especially under heavy load.

In this MPMC queue, each thread (producer or consumer) is assigned a "turn" based on a sequence number. A thread can only proceed with its operation (enqueue or dequeue) when its turn arrives. This ensures that operations happen in a controlled and predictable order to avoid conflicts between threads. When a producer wants to add an item to the queue, it waits until its turn arrives. Once the turn matches, it places the item in the queue and moves on. This is done without locking, using atomic operations to ensure that no two producers try to add items to the same spot in the queue at the same time.

Consumers work similarly. They wait for their turn to remove an item from the queue. Once their turn arrives, they safely remove the item and ensure that each item is only consumed once. This is again managed through atomic operations to maintain efficiency and correctness. The queue is designed to be fair which means that no single thread is favored over others. It also handles wrap-around when the turn numbers get very large to ensure that the system remains stable and efficient over time.

The MPMC queue implementation also utilizes futex to manage thread coordination more efficiently. In scenarios where there is no contention (i.e., when a thread's turn arrives and it can proceed immediately), the MPMC queue avoids system calls. Threads can use atomic operations to check their turn and proceed without invoking the kernel and make the operation extremely fast. When there is contention (e.g., a thread's turn hasn't arrived yet because another thread is still working), the thread can go to sleep using futex. The futex system call allows a

thread to sleep efficiently until it is notified that it can proceed. This prevents busy-waiting where a thread would otherwise keep checking its turn in a loop and consume CPU resources.

The futex mechanism wakes up only the threads that need to proceed based on the current state of the queue and ensure that only the necessary threads are woken up and reduce unnecessary context switches and improving overall performance. When a thread completes its operation (such as enqueuing or dequeuing an item), it may need to wake up another thread whose turn has arrived. This is done using the futex_wake function which efficiently wakes up the next thread in line without involving the kernel unless absolutely necessary.

2.1.4.2 C++ Memory Model

std::memory_order_acquire and std::memory_order_release are memory orderings used in C++11 and later versions to specify the synchronization behavior of atomic operations in multi-threaded environments. These memory orderings are crucial for ensuring correct and efficient communication between threads, especially when dealing with shared data.

When a variable is modified by one thread and accessed by another concurrently executing thread, proper synchronization mechanisms must be used to ensure that the memory accesses occur in a predictable order, avoiding data races and undefined behavior.

Figure 2.1.4.2.1 Acquire memory order

std::memory_order_acquire is used to ensure that memory accesses performed by the current thread are not reordered before a particular atomic operation. When a thread performs a load operation with memory_order_acquire, it establishes a synchronization dependency that

prevents subsequent memory reads or writes (or both) from being reordered before the load operation. This ensures that any data read by the current thread is consistent with the memory state at the time of the acquire operation.

Figure 2.1.4.2.2 Release memory order

On the other hand, std::memory_order_release is used to ensure that memory accesses performed by the current thread are not reordered after a particular atomic operation. When a thread performs a store operation with memory_order_release, it establishes a synchronization dependency that prevents previous memory reads or writes (or both) from being reordered after the store operation. This ensures that any data modifications made by the current thread are visible to other threads that perform subsequent memory accesses. [5].

One of the reason that we need to use std::memory_order is the compiler and CPU optimizations can affect the behavior of code utilizing memory_order_acquire and memory_order_release. Compiler optimizations, such as instruction reordering and caching optimizations, may rearrange the order of memory accesses in the generated machine code to improve performance. However, these optimizations must respect the memory ordering specified by atomic operations to maintain the correctness of multi-threaded programs. Therefore, compilers need to generate appropriate memory barriers or fence instructions to enforce the specified memory orderings.

Similarly, CPUs may perform optimizations such as speculative execution and out-of-order execution to improve performance by executing instructions ahead of their program order. These optimizations can potentially violate the memory orderings specified by atomic

operations, leading to incorrect behavior in multi-threaded programs. To address this, modern CPUs provide memory ordering semantics that ensure the visibility and ordering of memory accesses as specified by atomic operations.

In summary, std::memory_order_acquire and std::memory_order_release are used to specify memory ordering semantics for atomic operations in multi-threaded C++ programs. Compiler and CPU optimizations must take into account these memory orderings to ensure correct and efficient execution of multi-threaded code. Compiler-generated memory barriers and CPU memory ordering semantics play a crucial role in enforcing the specified memory orderings and maintaining the correctness of multi-threaded programs.

2.1.4.3 Fmt Library

{fmt} is an open-source formatting library providing a fast and safe alternative to C stdio and C++ iostreams [3]. The fmt library which stands for "format," provides a modern and type-safe alternative to traditional methods like std::cout and printf for formatting and printing text. It offers a simpler and more expressive syntax for constructing formatted strings, reducing the likelihood of errors and improving code readability. Unlike printf, fmt library leverages C++ features like variadic templates and operator overloading to enable type-safe formatting without sacrificing performance. It provides a rich set of formatting options that supports both positional and named arguments as well as a wide range of data types such as integers, floating-point numbers, strings, and custom types.

```cpp
int main() {
    int age = 30;
    std::string name = "Alice";
    double height = 1.75;

    // Formatted string using fmt
    std::string formatted_str = fmt::format("Name: {}, Age: {}, Height: {:.2f} meters", name, age, height);
    fmt::print("{}\n", formatted_str); // Output: Name: Alice, Age: 30, Height: 1.75 meters

    // Using std::cout
    std::cout << "Name: " << name << ", Age: " << age << ", Height: " << std::fixed << std::setprecision(2) << height << " meters" << std::endl

    // Using printf
    printf("Name: %s, Age: %d, Height: %.2f meters\n", name.c_str(), age, height); // Less type-safe and more error-prone

    return 0;
}
```

Figure 2.1.4.3.1 Comparison between fmt, std::cout, and printf

Based on the figure above, fmt provides a modern and expressive syntax for constructing formatted strings making the code more readable and maintainable. With fmt, developers can use simple placeholders like {} to represent variables within the string and eliminate the need for cumbersome format specifiers like %s, %d, and %f used in printf. This reduces the likelihood of errors and makes the code easier to understand especially for complex formatting scenarios.

Secondly, fmt ensures type safety by leveraging C++ features like variadic templates and operator overloading. Unlike printf, which relies on variable arguments of type va_list and lacks compile-time type checking, fmt performs type checking at compile time to prevent common mistakes like mismatched format specifiers and arguments. Moreover, fmt provides advanced formatting options like specifying precision for floating-point numbers and controlling the alignment of text which are either cumbersome or limited in printf and std::cout.

On the other hand, traditional methods like printf and std::cout suffer from several shortcomings that make them less suitable for modern C++ development. Printf's reliance on C-style format specifiers and variable arguments makes the code prone to errors, especially when dealing with complex formatting or type mismatches.

Similarly, std::cout, while providing a more object-oriented approach to output, still suffers from verbosity and limited formatting capabilities. Constructing formatted output with std::cout often involves chaining multiple insertion (<<) operators which can be cumbersome and error-prone especially for complex formatting scenarios. Additionally, std::cout lacks support for advanced formatting options like controlling precision for floating-point numbers and alignment of text making it less flexible compared to fmt.

2.1.4.4 C++ Template Metaprogramming

C++ template metaprogramming is a technique used to perform computations and code generation at compile-time using templates. It allows developers to write code that operates on types rather than values and enable the compiler to generate different code paths based on the types provided as template arguments.

At its core, C++ templates provide a way to define generic classes and functions that can work with any data type. Template metaprogramming takes this concept further by allowing developers to manipulate types and perform computations using template parameters.

One of the key benefits of template metaprogramming is the ability to generate specialized code for different types at compile-time. When a template is instantiated with a specific type, the compiler generates code tailored to that type and optimizing performance and reducing runtime overhead. This is in contrast to runtime polymorphism where the behavior is determined at runtime through virtual function calls leading to potential performance overhead.

```
template <typename T>
T square(T x) {
    return x * x;
}

int main() {
        int int_num = 5;
        double double_num = 23.4;

        int result = square(int_num);
        double result_double = square(double_num);
}
```

Figure 2.1.4.4.1 Template function example

When the template function named square shown in figure above is called with an integer type (int), the compiler generates code to perform integer multiplication. Similarly, when called with a floating-point type (float or double), the compiler generates code for floating-point multiplication. This specialization happens at compile-time, ensuring optimal performance for each type.

Overall, C++ template metaprogramming is a powerful tool for writing efficient and flexible code that operates on types rather than values. By leveraging compile-time computations and code generation, developers can achieve better performance, maintainability, and flexibility in their C++ programs.

2.1.4.5 Io_uring

io_uring is a modern Linux API introduced in kernel version 5.1 that enables high-performance asynchronous I/O operations. It is designed to overcome the performance limitations of traditional I/O models such as blocking I/O and asynchronous I/O (aio) by minimizing the overhead associated with system calls and improving efficiency in scenarios that involve high volumes of I/O operations such as file logging, network services, or databases. io_uring achieves this by providing a ring-buffer-based interface where submission and completion queues allow for non-blocking, batched I/O operations without the need for frequent context switches between user and kernel space.

The core of io_uring revolves around two circular buffers, or "rings" which are the submission queue (SQ) and the completion queue (CQ). These queues allow user-space applications to submit I/O requests and later retrieve the results of these requests asynchronously.

Submission Queue is where the application submits I/O operations. Instead of making individual system calls for each operation (e.g., read(), write()), the application places requests into this queue. Once an I/O operation is finished, the kernel places a completion event in the CQ. The application can then read from this queue to check whether the I/O operation has completed successfully and handle the results accordingly. By batching I/O operations and allowing multiple submissions at once, io_uring reduces the number of system calls, resulting in significant performance improvements for high-volume I/O workloads.

One of the key features of io_uring is its ability to handle I/O operations in a completely non-blocking manner. When an application submits an I/O request, the function returns immediately without waiting for the operation to complete. The kernel processes the request in the background, and the application can continue performing other tasks in parallel. Once the I/O operation completes, the result is posted to the completion queue, allowing the application to retrieve it later.

This non-blocking behavior is especially useful for applications that need to handle many concurrent I/O operations, such as network servers or logging systems where waiting on a single I/O operation would create bottlenecks and reduce overall performance.

Another performance-enhancing feature of io_uring is its ability to handle zero-copy I/O which eliminates the need to copy data between user space and kernel space. In traditional I/O models, data must be copied from user-space buffers to kernel-space buffers, which introduces additional overhead. With zero-copy support, io_uring allows the kernel to directly access user-space memory regions, reducing memory bandwidth usage and improving overall performance in scenarios with large amounts of data.

io_uring allows applications to submit multiple I/O requests at once in a batch which greatly reduces the overhead associated with system calls. This batching mechanism is particularly beneficial in scenarios where an application needs to perform several I/O operations in quick succession such as logging to a file or handling multiple network connections. By submitting a batch of requests in one go, the application minimizes the number of system calls leading to faster execution and lower context-switching costs.

To further reduce the overhead of processing I/O operations, io_uring introduces a kernel polling mode. When kernel polling is enabled, the kernel constantly monitors the submission and completion queues and eliminate the need for the application to explicitly notify the kernel to start processing the queues. This is especially useful in high-throughput systems where minimizing latency is critical such as in real-time logging systems. With kernel polling, the completion of I/O operations can be detected more quickly and allowed the application to respond to them with minimal delay.

io_uring supports a wide range of I/O operations from basic file read/write operations to more advanced features like splice(), send()/recv() for sockets, and poll(). This flexibility makes io_uring suitable for a wide variety of use cases including file systems, network programming, and low-latency logging systems. The ability to handle different types of I/O operations within the same framework allows developers to build more efficient systems by consolidating various I/O mechanisms into a single, high-performance API.

In summary, io_uring represents a significant step forward in Linux asynchronous I/O because it provides developers with a powerful tool for building high-performance, non-blocking applications. Its efficient handling of I/O operations, minimal system call overhead, and ability to scale in multi-threaded environments make it an ideal choice for building logging libraries

that need to handle large volumes of data quickly and reliably. By leveraging io_uring, a logging system can achieve both high performance and low latency, ensuring that log messages are written efficiently even under heavy load.

2.1.5 Summary of the Technologies Review

In summary, the combination of the chosen hardware, operating system, programming language, and algorithms provides a robust foundation for building a high-performance and user-friendly logging library. The Intel Core i5 processor's multi-core architecture enables the efficient execution of concurrent operations which is essential for handling multiple threads in parallel. This hardware capability is crucial for optimizing the logging system and allowing it to scale well in environments with high logging throughput.

The use of WSL2 (Windows Subsystem for Linux 2) and Ubuntu as the development platform allows the project to take full advantage of Linux's advanced system calls such as io_uring for non-blocking I/O operations. By leveraging the Linux kernel within a Windows environment, the project benefits from the flexibility of a Windows-based development workflow while maintaining access to high-performance I/O mechanisms.

The programming language C++ provides low-level control over memory and system resources making it an excellent choice for implementing a high-performance logging system. C++'s concurrency features particularly in C++20 further enhance the project through the use of modern tools like concepts, consteval, and source_location. These features not only improve performance by reducing runtime overhead but also ensure safer, more readable code with improved compile-time guarantees.

Finally, the project's optimization techniques such as the use of MPMC queues, atomic operations, and non-blocking I/O through io_uring which enable the logging system to handle large volumes of log entries efficiently. The combination of these technologies ensures that the logging library achieves its goals of being both high-performing and easy to use with minimal impact on the overall system performance.

2.2 Review of the Existing System

2.2.1 Fmtlog

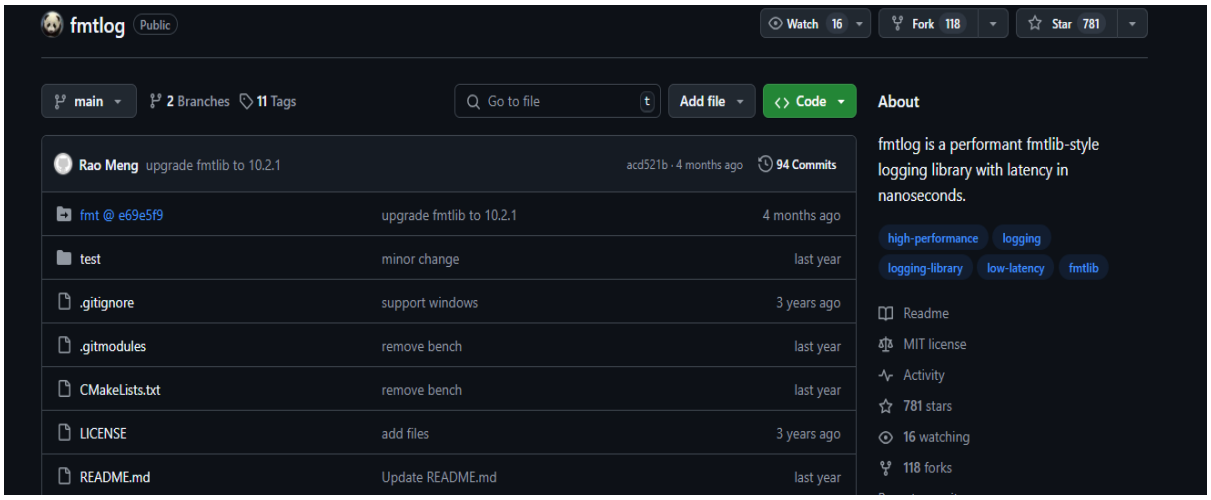


Figure 2.2.1.1 fmtlog github profile (781 stars)

fmtlog is a high-performance logging library designed for multithreaded environments, leveraging the fmt formatting library for efficient string manipulation. The library focuses on minimizing latency and overhead during logging operations, making it ideal for scenarios where logging speed and stability are critical. It achieves low-latency logging by employing two key optimization techniques inspired by the Nanolog logging library.

One of the primary optimizations is that fmtlog allocates a single-producer, single-consumer (SPSC) queue for each logging thread. This queue is used to avoid thread contention which typically happens when multiple threads attempt to log simultaneously. Instead of having each thread contend for access to a shared resource, each thread gets its own dedicated logging queue. These queues are automatically created when a thread logs a message for the first time and the background thread polls these queues to process the log messages. This design ensures that as the number of threads increases, the performance remains stable without degrading due to contention. The default size of each queue is 1 MB, though this can be customized using a compile-time macro (FMTLOG_QUEUE_SIZE). To further reduce latency, fmtlog provides a fmt::preallocate() function that users can call once a thread is created and ensure that the queue is allocated upfront.

Another important optimization is how fmtlog handles log message formatting. Instead of repeatedly formatting the same information (such as format strings, log levels, and source

locations), fmtlog stores this static information in a table on the first call to a log statement. Subsequent logs only push the index of the static info table entry along with the dynamic arguments to the queue, significantly reducing the size and time required to log each message. This minimizes memory usage and processing time as only the dynamic parts of the message (such as variables) are processed during the logging operation. When the background thread processes the queue, it uses the decoding function for each log statement to reconstruct the full log message.

The use of the fmt library known for its efficient and flexible string formatting capabilities, enhances fmtlog's performance. By offloading the string formatting to fmt, fmtlog benefits from the same optimizations that make fmt highly performant and ensure that log messages are formatted quickly without unnecessary overhead. Overall, fmtlog combines efficient queue management with minimal formatting overhead and provides a logging solution that is both fast and scalable in multithreaded environments.

2.2.2 Plog

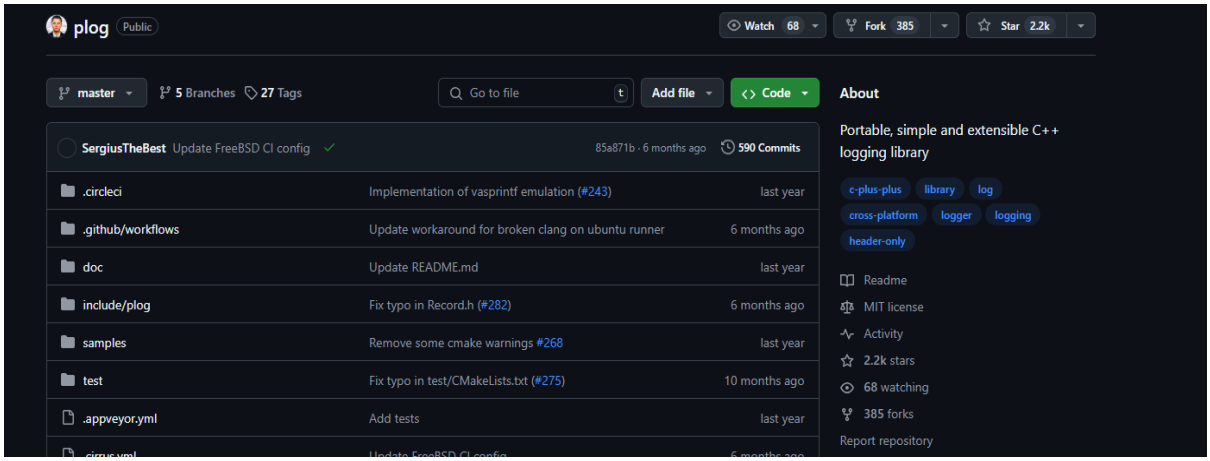


Figure 2.2.2.1 plog github profile (2.2k stars)

Plog is a lightweight, header-only logging library for C++ that has gained attention for its simplicity, flexibility, and performance optimizations. It is particularly designed to provide efficient logging for applications with stringent performance requirements.

A significant aspect of Plog is its focus on ease of use. Unlike more complex logging systems, Plog requires minimal configuration and can be integrated quickly into a project. This makes

it accessible to both novice and experienced developers who need a logging solution without diving deep into complex setups.

Being a header-only library, Plog eliminates the need for linking against external libraries which simplifies the build process. Developers can start using Plog by simply including the header file in their project. This avoids the overhead of managing external dependencies and makes it easier to integrate into existing codebases.

Plog offers a simple and intuitive API. Logging a message in Plog is as easy as calling:

Figure 2.2.2.2 API of plog

Plog distinguishes itself from other logging libraries with its focus on performance. It implements several optimization strategies to minimize logging overhead, making it suitable for high-performance applications. One of the key optimizations in Plog is its ability to configure log levels at compile time. Developers can define log levels such that logging calls below the specified level are removed entirely from the compiled binary. For instance, by setting the log level to WARNING at compile time, all DEBUG and INFO logs are omitted, reducing the performance impact of logging in production environments. This mechanism is particularly beneficial for performance-critical applications where log messages should not affect runtime performance but may still be useful during development or debugging.

Although Plog does not natively support asynchronous logging like some other logging libraries (such as NanoLog or Loguru), its simplicity and efficient design allow developers to easily integrate it with external asynchronous mechanisms. This is particularly useful for developers looking to decouple log generation from log writing to improve application performance under heavy load.

2.2.3 Loguru

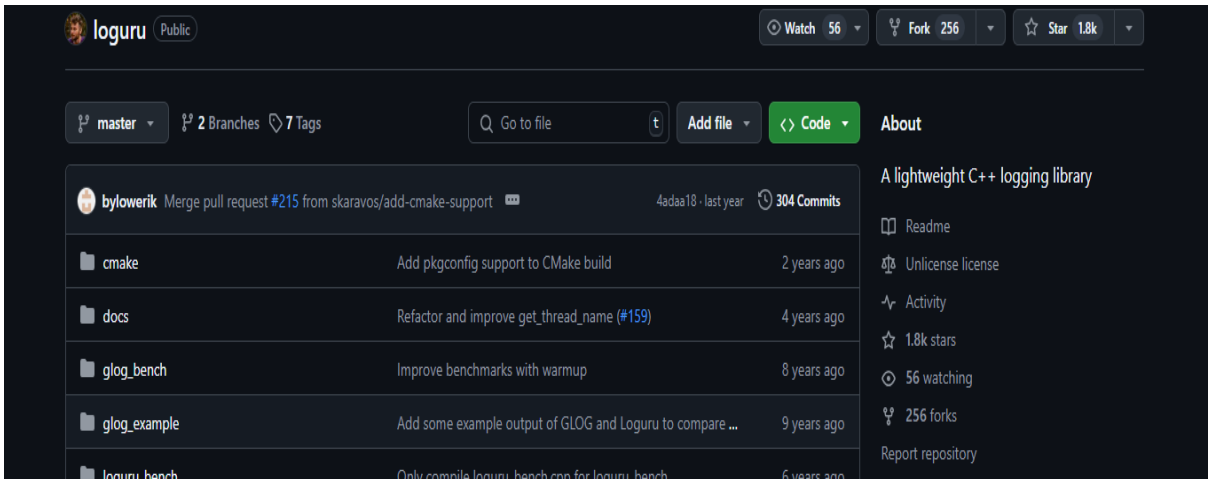


Figure 2.2.3.1 loguru github profile (1.8k stars)

Loguru is a modern C++ logging library aimed at simplifying the logging process with minimal configuration and high efficiency. It has gained popularity due to its intuitive API and performance optimizations. However, it also presents limitations that may impact its usability in certain contexts.

Loguru's design prioritizes simplicity and ease of use. It significantly reduces the typical boilerplate code found in other logging frameworks by offering macros such as LOG_F, CHECK_F, and LOG_IF_F. This enables developers to quickly add logging capabilities without configuring complex logging systems. The automatic logging of contextual information like function names, file paths, and line numbers also enhances its ease of use, providing essential debugging details without extra effort from the user.

However, one of the notable limitations is that Loguru heavily relies on the << operator for formatting log messages. This approach while familiar to C++ developers can become cumbersome when constructing complex log entries. Users must chain multiple << operations which can make the code less readable and verbose especially when compared to modern formatting libraries like fmt (used by libraries such as fmtlog). The absence of fmt-style formatting which allows for clearer and more concise string construction means that developers may spend additional time on formatting when using Loguru. This limitation affects its user-friendliness in scenarios where formatted output is complex or dynamically constructed.

Loguru excels in optimization particularly with its efficient handling of log writes. One of its key optimization techniques is buffered logging where logs are accumulated in memory before being written to disk in batches. This significantly reduces the frequency of I/O operations, improving performance in scenarios with high log volume. Developers can control the buffering to balance between performance and log integrity, making Loguru adaptable for real-time or production environments.

Concurrency is a critical concern for logging libraries particularly in multi-threaded environments. Loguru handles concurrency well by implementing thread-safe logging mechanisms. It uses mutexes to ensure that logging operations are synchronized, preventing data races and ensuring that log output remains consistent across threads.

In conclusion, Loguru is a highly user-friendly and efficient C++ logging library, well-suited for a variety of applications. Its simple API design, automatic inclusion of contextual information, and optimized performance through buffering and verbosity control make it an attractive option for developers seeking an easy-to-integrate logging solution. Its concurrency handling is robust, ensuring safe and efficient logging in multi-threaded environments.

Chapter 3 System Methodology/Approach OR System Model

3.1 System Design Diagram/Equation

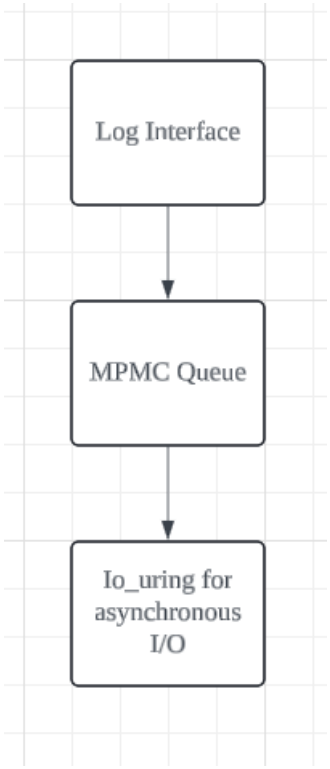


Figure 3.1.1 Architecture Diagram

This system design diagram outlines an enhanced logging system architecture that includes several key components such as a logging interface, a multiple-producer, multiple-consumer (MPMC) queue, and asynchronous file I/O functionality leveraging Linux's io_uring mechanism.

At the core of the system is the logging interface, which acts as the entry point for log messages generated throughout the application. This interface provides a unified API for logging messages of various severity levels, such as debug, info, error, and fatal. Developers can utilize this interface to log messages according to the criticality of events, ensuring that logs reflect the correct severity.

Once a log message is generated, it is placed in a multiple-producer, multiple-consumer (MPMC) queue. The MPMC queue is a high-performance data structure designed for

concurrent access from multiple threads. It utilizes atomic operations for thread-safe enqueueing and dequeueing of log messages, allowing multiple threads to produce log entries simultaneously without blocking or requiring traditional locking mechanisms. To ensure smooth coordination between multiple threads, the queue relies on atomic wait-and-notify operations and uses the futex system call for efficient synchronization in the scenarion of high contention.

The MPMC queue acts as a buffer, decoupling the log message generation from file I/O operations. This separation prevents delays caused by slower file write operations from impacting the performance of the application. Log messages are buffered in the queue until they can be written to disk.

The final step in the logging pipeline is the asynchronous file I/O functionality which handles writing log messages to files. This system takes advantage of the io_uring interface in the Linux kernel, which allows for highly efficient asynchronous I/O operations. Io_uring reduces the overhead of file system calls by providing a fast and efficient mechanism for queuing and completing I/O tasks in the kernel, further boosting the logging system's performance. By using io_uring, the system ensures that log messages are written to disk without blocking the application, even under heavy I/O load.

In summary, this logging system architecture efficiently manages log generation, categorization, buffering, and asynchronous storage through the use of key components such as a logging interface, log level abstraction, an MPMC queue with atomic operations and futex synchronization, and asynchronous file I/O using io_uring. This design ensures high performance and scalability, making it suitable for modern, multithreaded applications that require reliable logging even under heavy load conditions.

CHAPTER 4 SYSTEM DESIGN

4.1 System Component Specification

4.1.1 Logging Interface

```
template <typename... Args>
void debug(source_location<fmt::format_string<Args...>> fmt, Args&&... args) {
    addLogMessage(logging::LogLevel::debug, fmt, std::forward<Args>(args)...);
}

template <typename... Args>
void error(source_location<fmt::format_string<Args...>> fmt, Args&&... args) {
    addLogMessage(logging::LogLevel::error, fmt, std::forward<Args>(args)...);
}

template <typename... Args>
void info(source_location<fmt::format_string<Args...>> fmt, Args&&... args) {
    addLogMessage(logging::LogLevel::info, fmt, std::forward<Args>(args)...);
}

template <typename... Args>
void fatal(source_location<fmt::format_string<Args...>> fmt, Args&&... args) {
    addLogMessage(logging::LogLevel::fatal, fmt, std::forward<Args>(args)...);
}

    void setOutputFile(std::string_view);
```

Figure 4.1.1.1 API supported

The APIs shown in the provided image above are part of my logging library that allows developers to record messages with different levels of severity or importance. The levels of severity (such as debug, error, info, and fatal) reflect the nature of the message being logged. For example, debug is used for low-priority information useful during development while error and fatal are reserved for higher-priority issues.

Each API method like debug, error, info, and fatal accepts formatted strings (using the fmt library for formatting) and additional arguments that will be included in the log message. This enables users to log detailed messages with dynamic content making it easier to debug or monitor software behavior.

The setOutputFile API shown is designed to allow users to specify where the log messages should be stored. By calling setOutputFile with a file path, users can redirect the logs to a

specific file. This is useful in scenarios where logs need to be preserved for later analysis or when running programs in production environments where logs must be collected systematically.

```cpp
namespace logging {
    template<class T>
    struct source_location {
    private:
        T inner_;
        std::source_location loc_;

    public:
        template<class U> requires std::constructible_from<T, U>
        consteval source_location(U&& inner, std::source_location loc = std::source_location::current())
        : inner_(std::forward<U>(inner)), loc_(std::move(loc)) {}

        constexpr T const& format() const { return inner_; }

        constexpr std::source_location const& location() const { return loc_; }
    };

    class Log {
    public:
        Log() = default;
        ~Log();

        Log(const Log& other) = delete;
        Log& operator=(const Log& other) = delete;
        Log(Log&& other) noexcept  = delete;
        Log& operator=(Log&& other) noexcept = delete;

        #define _FUNCTION(name) \
        template<typename... Args> \
        void name(source_location<fmt::format_string<Args...>> fmt, Args&&... args) { \
            addLogMessage(logging::LogLevel::name, fmt, std::forward<Args>(args)...); \
        }
        LOGGING_FOR_EACH_LOG_LEVEL(_FUNCTION)
        #undef _FUNCTION

        void setOutputFile(std::string_view);
```

```cpp
    private:
        template <typename... Args>
        void addLogMessage(logging::LogLevel level, source_location<fmt::format_string<Args...>> fmt, Args&&... args) {
            std::string now = getPrefix();

            auto const& loc = fmt.location();

            auto log_msg = fmt::vformat(to_string_view(fmt.format()), fmt::make_format_args(args...));

            auto output_msg = fmt::format("{} {}:{} [{}] {}\n", now, loc.file_name(), loc.line(), logLevelToString(level), log_msg);

            auto deadline = std::chrono::steady_clock::now() + std::chrono::microseconds(250);

            mpmc_.write(std::move(output_msg));

            std::string pop_msg{};
            while (mpmc_.size() >= mpmc_.capacity() / 2) {
                mpmc_.read(pop_msg);

                io_context_.write(pop_msg.data(), pop_msg.size());
            }
        }

        template <typename T, typename... Args>
        fmt::basic_string_view<T> to_string_view(fmt::basic_format_string<T, Args...> fmt) {
            return fmt;
        }
```

```
        std::string getPrefix();
        std::string logLevelToString(logging::LogLevel);

    private:
        logging::IoContext io_context_;
        std::string_view file_path_;
        MPMCQueue<std::string> mpmc_{100};
    };
}
```

Figure 4.1.1.2 Log template class code

The Log class shown above is at the center of the system to provide functions to log messages at different severity levels (e.g., debug, info, error, and fatal). Each log message is generated using fmt::format which allows for efficient formatting.

The source_location struct is a template that wraps around the log message format string and captures the source location where the log is generated such as file name, file location and etc. This enables detailed log messages that include the origin of the log.

The use of X-macros is evident in how the different log levels (debug, info, error, fatal) are handled. In my Log class, the macro LOGGING_FOR_EACH_LOG_LEVEL defines the log levels and applies a function (_FUNCTION) to each one. This allows to generate similar code for each log level in a concise way and avoid the redundancy. By using X-macros, we can easily add or modify log levels by updating the macro definition which keeps the code cleaner and more maintainable. This pattern is useful for defining repetitive functionality across multiple components or log levels without rewriting the same code.

Implicit conversion is another concept used in the source_location struct. The source_location constructor accepts any type U that can be converted to T using the std::constructible_from concept. This allows the source_location to handle different types of format strings making it more flexible. When pass the format string to the addLogMessage function, implicit conversion happens automatically to convert the format string into the source_location type, simplifying how the API is used. This is an example of how implicit conversion can streamline interactions between different types in C++ while preserving type safety.

After that, the log messages are enqueued into a multiple-producer, multiple-consumer queue (MPMCQueue) with a default capacity of 100. The queue serves as a buffer, allowing multiple threads to concurrently enqueue log messages without contention, thus improving performance in multithreaded environments. The tryWriteUntil function attempts to write to the queue with a timeout, ensuring that the system can handle load gracefully.

When the queue reaches half of its capacity, log messages are dequeued and written to a file asynchronously using io_uring. The IoContext class handles the asynchronous I/O operations, allowing for non-blocking file writes. This decouples the logging process from the file I/O, ensuring that the application is not slowed down by file operations.

The setOutputFile function sets the path to the log file where messages will be written. The log level and the source location of the message are included in each log entry, providing detailed and structured log output. This design leverages modern C++ techniques like concepts and concurrency primitives to build an efficient, scalable logging system.

4.1.2 Logging Level

```
namespace logging {

#define LOGGING_FOR_EACH_LOG_LEVEL(f) \
    f(debug) \
    f(info) \
    f(error) \
    f(fatal)

enum class LogLevel : std::uint8_t {
#define _FUNCTION(name) name,
    LOGGING_FOR_EACH_LOG_LEVEL(_FUNCTION)
#undef _FUNCTION
};

}
```

Figure 4.1.2.1 LogLevel enum class code

The code above defines an enum class in the logging namespace where it categorizes different logging levels. The macro LOGGING_FOR_EACH_LOG_LEVEL is used to list the log levels such as debug, info, error, and fatal. This is an example of an X-macro, a technique that allows to define repetitive items in one place and reuse them in different contexts.

The LogLevel enum class is created using this macro. The X-macro defines each log level name by invoking the _FUNCTION macro for each item in the LOGGING_FOR_EACH_LOG_LEVEL list. This reduces redundancy and makes it easy to add new log levels in the future by simply modifying the macro.

Once the macro is expanded, the LogLevel enum class will contain entries like debug, info, error, and fatal, each represented by a std::uint8_t value. This structure helps categorize log messages by severity, making the logging system more organized and flexible.

4.1.3 Multi-Producer Multi-Consumer Queue

```
template <template <typename T, template <typename> class Atom, bool Dynamic> class Derived,
          typename T, template <typename> class Atom, bool Dynamic>
class MPMCQueueBase<Derived<T, Atom, Dynamic>> {
    static_assert(std::is_nothrow_constructible<T, T&&>::value);

public:
    typedef T value_type;

    using Slot = SingleElementQueue<T, Atom>;

    explicit MPMCQueueBase(size_t queueCapacity)
        : capacity_(queueCapacity),
          pushTicket_(0),
          popTicket_(0),
          pushSpinCutoff_(0),
          popSpinCutoff_(0)
    {
        if (queueCapacity == 0) {
            throw std::invalid_argument("MPMCQueue with explicit capacity 0 is impossible");
        }

        assert(alignof(MPMCQueue<T, Atom>) >= hardware_destructive_interference_size);
        assert(static_cast<uint8_t*>(static_cast<void*>(&popTicket_)) -
               static_cast<uint8_t*>(static_cast<void*>(&pushTicket_)) >=
               static_cast<ptrdiff_t>(hardware_destructive_interference_size));
    }
```

```
template <typename T, template <typename> class Atom = std::atomic, bool Dynamic = false>
class MPMCQueue : public MPMCQueueBase<MPMCQueue<T, Atom, Dynamic>> {
    using Slot = SingleElementQueue<T, Atom>;

public:
    explicit MPMCQueue(size_t queueCapacity) : MPMCQueueBase<MPMCQueue<T, Atom, Dynamic>>(queueCapacity) {
        this->stride_ = this->computeStride(queueCapacity);
        this->slots_ = new Slot[queueCapacity + 2 * this->kSlotPadding];
    }

    MPMCQueue() noexcept {}
};
```

Figure 4.1.3.1 MPMC Queue class code

The MPMCQueue class is a template class that allows the queue to hold different types (T), and it supports customization through atomic operations (Atom) and an optional dynamic mode (Dynamic). It inherits from the MPMCQueueBase class which provides most of the core functionality. The constructor of MPMCQueue initializes the queue with a capacity, computes a stride value for optimized slot access, and allocates memory for the queue slots. These slots are instances of the SingleElementQueue class, which represents individual storage locations in the queue.

```
protected:
    static constexpr std::size_t hardware_destructive_interference_size = 64;

    enum {
        kAdaptationFreq = 128,

        kSlotPadding = (hardware_destructive_interference_size - 1) / sizeof(Slot) + 1
    };

    alignas(hardware_destructive_interference_size) size_t capacity_;

    Slot* slots_;

    int stride_;

    alignas(hardware_destructive_interference_size) Atom<uint64_t> pushTicket_;

    alignas(hardware_destructive_interference_size) Atom<uint64_t> popTicket_;

    alignas(hardware_destructive_interference_size) Atom<uint32_t> pushSpinCutoff_;

    alignas(hardware_destructive_interference_size) Atom<uint32_t> popSpinCutoff_;

    char pad_[hardware_destructive_interference_size - sizeof(Atom<uint32_t>)];
```

Figure 4.1.3.2 MPMC Queue essential variables

In the base class MPMCQueueBase, the pushTicket_ and popTicket_ atomic variables track the number of enqueued and dequeued elements, respectively. The queue uses these tickets to ensure that each enqueue and dequeue operation happens in the correct order. The stride_ variable is used to calculate the index of the slot that a thread should access, ensuring that the slots are spaced out in a way that minimizes cache line contention especially when multiple threads are writing to or reading from the queue simultaneously.

```
void enqueueImpl(const uint32_t turn, Atom<uint32_t>& spinCutoff, const bool updateSpinCutoff, T&& goner, ImplByMove) noexcept {
    sequencer_.waitForTurn(turn * 2, spinCutoff, updateSpinCutoff);
    new (&contents_) T(std::move(goner));
    sequencer_.completeTurn(turn * 2);
}
```

```
void dequeueImpl(uint32_t turn, Atom<uint32_t>& spinCutoff, const bool updateSpinCutoff, T& elem, ImplByMove) noexcept {
    sequencer_.waitForTurn(turn * 2 + 1, spinCutoff, updateSpinCutoff);
    elem = std::move(*ptr());
    destroyContents();
    sequencer_.completeTurn(turn * 2 + 1);
}
```

Figure 4.1.3.3 Enqueue and dequeue function code

The enqueue and dequeue operations in the queue are managed using a ticket system. When a producer thread calls the write function, it attempts to obtain a push ticket by using atomic compare-and-swap (CAS) operations on pushTicket_. If the ticket is obtained, the producer can proceed to enqueue its element into the slot. Similarly, the read function uses a ticket-based system for dequeuing. The tickets ensure that the correct element is enqueued or dequeued at the correct time, maintaining consistency across multiple threads.

```
template <class Clock>
bool tryObtainPromisedPushTicketUntil(uint64_t& ticket, Slot*& slots, size_t& cap, int& stride, const std::chrono::time_point<Cloc
    bool deadlineReached = false;
    while (!deadlineReached) {
        if (tryObtainPromisedPushTicket(ticket, slots, cap, stride)) {
            return true;
        }

        deadlineReached = !slots[idx(ticket, cap, stride)].tryWaitForEnqueueTurnUntil(turn(ticket, cap), pushSpinCutoff_,
        (ticket % kAdaptationFreq) == 0, when);
    }
    return false;
}
```

```
template <class Clock>
bool tryObtainPromisedPopTicketUntil(uint64_t& ticket, Slot*& slots, size_t& cap, int& stride, const std::chrono::time_point<Clo
    bool deadlineReached = false;
    while (!deadlineReached) {
        if (static_cast<Derived<T, Atom, Dynamic>>(this)->tryObtainPromisedPopTicket(ticket, slots, cap, stride)) {
            return true;
        }

        deadlineReached = !slots[idx(ticket, cap, stride)].tryWaitForDequeueTurnUntil(turn(ticket, cap), pushSpinCutoff_,
        (ticket % kAdaptationFreq) == 0, when);
    }
    return false;
}
```

Figure 4.1.3.4 Obtain ticket function code

A significant part of the code is dedicated to turn-based sequencing using the TurnSequencer. This mechanism ensures that threads enqueue and dequeue elements in the correct order, even in a highly concurrent environment. Each slot in the queue can only be accessed by one thread at a time, and the TurnSequencer ensures that the thread holds the "turn" before accessing a slot. Once a thread completes its operation (either enqueuing or dequeuing), it "completes" its turn, allowing other threads to proceed.

```
template <typename T, template <typename> class Atom>
struct SingleElementQueue {
    ~SingleElementQueue() noexcept {
        if ((sequencer_.uncompletedTurnLSB() & 1) == 1) {
            destroyContents();
        }
    }

    template <typename = typename std::enable_if<std::is_nothrow_constructible<T>::value || std::is_nothrow_constructible<T, T
    void enqueue(const uint32_t turn, Atom<uint32_t>& spinCutoff, const bool updateSpinCutoff, T&& goner) noexcept {
        enqueueImpl(turn, spinCutoff, updateSpinCutoff, std::move(goner), typename std::conditional<std::is_nothrow_constructi
                    ImplByMove, ImplByRelocation>::type());
    }

    template <class Clock>
    bool tryWaitForEnqueueTurnUntil(const uint32_t turn, Atom<uint32_t>& spinCutoff, const bool updateSpinCutoff,
                                    const std::chrono::time_point<Clock>& when) noexcept {
        return sequencer_.tryWaitForTurn(turn * 2, spinCutoff, updateSpinCutoff, &when) != TurnSequencer<Atom>::TryWaitResult:
    }

    bool mayEnqueue(const uint32_t turn) const noexcept {
        return sequencer_.isTurn(turn * 2);
    }

    void dequeue(uint32_t turn, Atom<uint32_t>& spinCutoff, const bool updateSpinCutoff, T& elem) noexcept {
        dequeueImpl(turn, spinCutoff, updateSpinCutoff, elem, typename std::conditional<std::is_trivially_copyable<T>::value,
    }
```

Figure 4.1.3.5 SingleElementQueue template class code

The SingleElementQueue class represents the individual slots in the queue. It is responsible for holding a single element (T) and ensuring that it is safely constructed, moved, or destroyed during enqueue and dequeue operations. Depending on the type of element (T), the queue uses either a move-based or relocation-based strategy for transferring data into and out of the slot. This flexibility is managed through template specialization and conditional compilation based on the properties of T.

The class also handles spinning and waiting through spin-waiting mechanisms (such as pushSpinCutoff_ and popSpinCutoff_) which control how long a thread will spin-wait before yielding if it cannot immediately obtain a ticket or access a slot. This helps reduce contention and improves performance in high-concurrency scenarios.

In summary, this code implements an efficient, lock-free MPMC queue that supports multiple producers and consumers with minimal contention. The use of atomic variables, ticket-based queuing, and turn-based sequencing ensures safe and consistent operations, while avoiding locks and maximizing throughput in concurrent environments.

4.1.4 Turn Sequencer

TurnSequencer class designed to manage the sequential execution of threads based on a "turn" system. At its core, the TurnSequencer uses atomic operations to keep track of the current turn and the state of the waiters. The concept of "turns" allows threads to know when it is their turn to proceed, while others wait until their turn arrives. The TurnSequencer ensures that threads waiting for a particular turn will not proceed until their turn is reached.

This class contains a state_ variable is a 32-bit atomic integer that encodes both the current turn and the number of waiters waiting for upcoming turns. The kTurnShift constant is used to split the 32-bit state into two parts: the upper bits store the current turn, and the lower bits store the waiter count which represents how many waiters are ahead in line for future turns.

```
explicit TurnSequencer(const uint32_t firstTurn = 0) noexcept : state_(encode(firstTurn << kTurnShift, 0)) {}
```

Figure 4.1.4.1 TurnSequencer constructor code

The constructor initializes the state_ variable by encoding the initial turn (firstTurn) and setting the number of waiters to 0. By default, the firstTurn is set to 0, meaning that the first turn will be "turn 0" when the sequencer starts.

```
bool isTurn(const uint32_t turn) const noexcept {
    auto state = state_.load(std::memory_order_acquire);
    return decodeCurrentSturn(state) == (turn << kTurnShift);
}
```

Figure 4.1.4.2 TurnSequencer isTurn function code

The method above checks whether the given turn is the current turn. It compares the decoded current turn from state_ with the requested turn. This is a simple check, often used to decide whether a thread should proceed or wait.

```
void waitForTurn(const uint32_t turn, Atom<uint32_t>& spinCutoff, const bool updateSpinCutoff) noexcept {
    const auto ret = tryWaitForTurn(turn, spinCutoff, updateSpinCutoff);
}
```

Figure 4.1.4.3 TurnSequencer waitForTurn function code

The method above is called by threads to wait for their turn. Internally, it calls tryWaitForTurn, which contains the logic to either spin (busy-wait) or put the thread to sleep using futex calls if the wait extends beyond a certain threshold.

```
template <class Clock = std::chrono::steady_clock, class Duration = typename Clock::duration>
TryWaitResult tryWaitForTurn(const uint32_t turn, Atom<uint32_t>& spinCutoff,
                             const bool updateSpinCutoff, const std::chrono::time_point<Clock, Duration>* absTime = nullptr) noexcept {
```

Figure 4.1.4.4 TurnSequencer tryWaitForTurn function code

This is where the actual waiting logic occurs. The method attempts to wait for a specific turn (turn). It uses a combination of busy-waiting and futex-based sleeping to optimize thread performance.

At first, the thread spins (repeatedly checks the state) for a few iterations to see if the turn has advanced. This is done in the fast path to avoid the overhead of system calls like futex in cases where the thread's turn will arrive quickly. If the turn does not arrive within the busy-waiting threshold, the thread is put to sleep using the futexWait system call. This prevents excessive CPU usage while waiting for a longer time. The method also accounts for thread contention and "wrap-around" of turn values by ensuring that even if the current turn is numerically lower than the requested turn due to wrap-around, the comparison remains correct.

```
void completeTurn(const uint32_t turn) noexcept {
    uint32_t state = state_.load(std::memory_order_acquire);
    while (true) {
        assert(state == encode(turn << kTurnShift, decodeMaxWaitersDelta(state)));
        uint32_t max_waiter_delta = decodeMaxWaitersDelta(state);
        uint32_t new_state = encode((turn + 1) << kTurnShift, max_waiter_delta == 0 ? 0 : max_waiter_delta - 1);

        if (state_.compare_exchange_strong(state, new_state)) {
            if (max_waiter_delta != 0) {
                detail::futexWake(&state_, std::numeric_limits<int>::max(), futexChannel(turn + 1));
            }
            break;
        }
    }
}
```

Figure 4.1.4.5 TurnSequencer completeTurn function code

The method above is called when a thread has finished its turn. It advances the sequencer to the next turn by incrementing the current turn in state_. If there are other threads waiting for the next turn, it wakes them up using the futexWake system call, signaling that their turn is ready.

The method ensures that the thread that has finished its turn updates the state correctly and handles any waiting threads. If there are no waiters, the system skips waking up other threads, optimizing performance by avoiding unnecessary futex calls.

```
uint32_t decodeCurrentSturn(uint32_t state) const noexcept {
    return state & ~kWaitersMask;
}

uint32_t decodeMaxWaitersDelta(uint32_t state) const noexcept {
    return state & kWaitersMask;
}

uint32_t encode(uint32_t currentSturn, uint32_t maxWaiterD) const noexcept {
    return currentSturn | std::min(uint32_t{kWaitersMask}, maxWaiterD);
}
```

Figure 4.1.4.6 TurnSequencer decode helper functions code

The state_ variable encodes two pieces of information which are the current turn (most significant bits) and the maximum waiter delta (least significant bits) which indicates how many threads are waiting for future turns. Two helper methods, encode and decode, are used to encode and decode these values. encode combines the current turn and waiter delta into a single integer. decodeCurrentSturn extracts the current turn from state_. decodeMaxWaitersDelta extracts the number of waiters from state_. This encoding ensures that the sequencer can track both the current turn and how many threads are waiting for future turns efficiently using atomic operations.

```
private:
    static constexpr bool kSpinUsingHardwareClock = 1;
    static constexpr uint32_t kCyclesPerSpinLimit = kSpinUsingHardwareClock ? 1 : 10;

    static constexpr uint32_t kTurnShift = 6;
    static constexpr uint32_t kWaitersMask = (1 << kTurnShift) - 1;

    static constexpr uint32_t kMinSpinLimit = 200 / kCyclesPerSpinLimit;

    static constexpr uint32_t kMaxSpinLimit = 20000 / kCyclesPerSpinLimit;

    std::atomic<std::uint32_t> state_;
```

Figure 4.1.4.7 TurnSequencer essential variables

Several constants are defined to manage the behavior of the sequencer.

kTurnShift: Determines how much the turn value is shifted to make space for the waiter delta.

kMinSpinLimit and kMaxSpinLimit: Define the range for busy-waiting before switching to futex-based waiting.

kCyclesPerSpinLimit: Defines how many cycles to spin before considering switching to the futex-based wait.

The TurnSequencer class implements an efficient mechanism for coordinating thread execution in a turn-based manner. It ensures that threads take turns in the correct order and uses busy-waiting combined with futex-based sleeping to optimize for performance and minimize contention. The atomic operations and futex system calls allow it to handle multiple threads efficiently, even in highly concurrent environments.

The TurnSequencer is ideal for scenarios where threads must take turns in accessing a shared resource or performing a sequence of operations, and it ensures fairness and ordering between threads, all while optimizing performance with minimal contention.

4.1.5 Futex

This futex (Fast Userspace Mutex) implementation focuses on providing low-level synchronization mechanisms which utilizing Linux's futex system call. The futex system call allows threads to wait on and wake each other in a more efficient way compared to traditional mutexes, especially when there is no contention. This implementation uses FUTEX_WAIT_BITSET and FUTEX_WAKE_BITSET, which are extensions to the standard futex system calls.

At its core, this implementation provides a wrapper around the futex system call for both waiting (futexWait) and waking (futexWake) threads based on a specific memory address, commonly a 32-bit integer. The futex itself is an atomic variable that threads can wait on when the variable's value is not what they expect, and they can wake up once the value changes.

```
FutexResult nativeFutexWaitImpl(const void* addr, uint32_t expected, std::chrono::system_clock::time_point const* absSystemTime,
                                std::chrono::steady_clock::time_point const* absSteadyTime, uint32_t waitMask) {
    assert(absSystemTime == nullptr || absSteadyTime == nullptr);
```

Figure 4.1.5.1 Futex wait function code

This function is responsible for putting the calling thread to sleep when a specific condition is met, using the FUTEX_WAIT_BITSET operation. Before putting the thread to sleep, it ensures that the value of the atomic variable (addr) is what the thread expects (i.e., expected). If the value has changed, the thread will not sleep and the function will return FutexResult::VALUE_CHANGED.

The core of the function revolves around using FUTEX_WAIT_BITSET. This variant of the futex wait allows for the use of a bitmask (waitMask). Only if the thread's bit is set in both the wait mask and wake mask will the thread be woken up later. This enables more control over which threads are woken up, particularly useful in scenarios where multiple threads are waiting on the same futex but some should be ignored for specific wake-up conditions.

If a timeout is provided (via absSystemTime or absSteadyTime), the thread will sleep for a limited time, after which it will return FutexResult::TIMEDOUT. The function also handles interrupts (EINTR) and cases where the futex value has already changed (EWOULDBLOCK), returning FutexResult::INTERRUPTED or FutexResult::VALUE_CHANGED respectively.

The bitmask (waitMask) allows for up to 32 separate conditions (each represented by a bit in a 32-bit mask) that can be checked atomically during the wake-up process, providing a finer granularity of control over which threads should be woken up.

```
int nativeFutexWakeImpl(const void* addr, int count, uint32_t wakeMask) {
    int rv = syscall(
        __NR_futex,
        addr,
        FUTEX_WAIT_BITSET | FUTEX_PRIVATE_FLAG,
        count,
        nullptr,
        nullptr,
        wakeMask);

    if (rv < 0) {
        return 0;
    }
    return rv;
}
```

Figure 4.1.5.2 Futex wake function code

This function is responsible for waking up threads that are waiting on the futex, using the FUTEX_WAKE_BITSET operation. It calls syscall(__NR_futex) with the

FUTEX_WAKE_BITSET flag, which means it will wake up threads waiting on the futex at the provided memory address (addr), but only if their wait mask intersects with the wake mask (wakeMask). This is critical for performance, as it avoids waking up unnecessary threads.

The count parameter controls how many threads should be woken up. If this is set to std::numeric_limits<int>::max(), it will attempt to wake all threads that match the wake condition. Otherwise, it will wake up a limited number of threads, improving efficiency in scenarios where waking all waiting threads is not required. If the system call fails (returns a negative value), it returns 0, indicating that no threads were woken up. Otherwise, it returns the number of threads that were successfully woken up.

The FUTEX_WAIT_BITSET and FUTEX_WAKE_BITSET operations are specialized versions of the regular futex system calls that allow for a bitmask to be applied when waiting and waking threads. These operations are useful when you need more control over which threads are allowed to wake up, as they provide the ability to set conditions on wakeups based on bitmasks.

FUTEX_WAIT_BITSET allows a thread to wait until it is woken up based on a specific bitmask. Only if the wait condition and the mask are satisfied will the thread be put to sleep, ensuring that only threads with the right conditions will wait.

FUTEX_WAKE_BITSET is used to wake threads that are waiting on the futex. The wake mask is compared with the wait mask of each sleeping thread, and only those threads that have a matching bit set will be woken up. This allows for efficient wake-up of only the relevant threads.

This futex implementation provides an efficient way for threads to wait and be woken up based on atomic conditions and bitmask operations. The use of FUTEX_WAIT_BITSET and FUTEX_WAKE_BITSET adds more control over which threads are woken up, preventing unnecessary wake-ups and improving performance in highly concurrent environments. The code also provides flexibility with timeouts, handling interruptions, and ensuring that the futex state is checked atomically to avoid race conditions.

4.1.6 Io_uring

The IoContext class in the code is designed to handle asynchronous file writing using the Linux io_uring interface, which offers high-performance I/O operations by minimizing system call overhead. This class is part of my logging system that can handle concurrent log writes across multiple threads efficiently. It leverages io_uring to submit batched I/O requests, reducing the need for blocking or repeated system calls. Additionally, a turn-based synchronization mechanism (TurnSequencer) ensures that log entries are processed in the correct order when multiple threads are involved.

```
namespace logging {
    class IoContext {
    public:
        IoContext();

        IoContext(const IoContext&) = delete;

        IoContext(IoContext&&) = delete;

        IoContext& operator=(const IoContext&) = delete;

        IoContext& operator=(IoContext&&) = delete;

        ~IoContext();

        int register_file(std::string_view);
        void write(const char*, size_t);

    private:
        struct io_uring io_uring_;
        int fds[2];
        std::atomic<uint32_t> count_{0};
        std::atomic<uint32_t> turn_{0};
        TurnSequencer<std::atomic> turn_sequencer_;
        alignas(64) std::atomic<uint32_t> spinCutoff_;
    };
}
```

Figure 4.1.6.1 io_uring class code

At the core of the IoContext class is the io_uring structure, represented by the io_uring_ variable. This structure manages the submission and completion queues for asynchronous operations. All file writing operations are handled through this interface, allowing the system

to submit write requests without waiting for their completion, which is crucial for performance in high-concurrency environments.

The file descriptor for the target file is stored in the fds array, specifically in fds[0]. The file is opened when the user calls the register_file function, which opens a log file in append mode. This function prepares the system to start logging to the file, handling any potential errors if the file cannot be opened.

```
void logging::IoContext::write(const char* message, size_t len) {
    turn_sequencer_.waitForTurn(turn_, spinCutoff_, true);
    struct io_uring_sqe* sqe = io_uring_get_sqe(&io_uring_);
    if (!sqe) {
        fprintf(stderr, "Failed to get submission queue entry\n");
        return;
    }

    // Allocate a new buffer for each message to avoid overwriting
    char* new_buffer = new char[len];
    memcpy(new_buffer, message, len);

    // Prepare the write operation using the unique buffer
    io_uring_prep_write(sqe, fds[0], new_buffer, len, 0);
    sqe->user_data = reinterpret_cast<uint64_t>(new_buffer);  // Store the buffer address in user_data

    // int count = count_.fetch_add(1, std::memory_order_acq_rel) + 1;  // Increment and get the new value
    int turn = turn_.fetch_add(1, std::memory_order_acq_rel);
    turn_sequencer_.completeTurn(turn);

    // Atomically increment count before the condition check
    int count = count_.fetch_add(1, std::memory_order_acq_rel) + 1;  // Increment and get the new value

    if (count == QUEUE_DEPTH / 2) {
        io_uring_submit(&io_uring_);
```

```
        // Reset the counter to 0 atomically
        count_.store(0, std::memory_order_release);

        struct io_uring_cqe* cqe;
        for (int i = 0; i < QUEUE_DEPTH / 2; ++i) {
            int ret_wait = io_uring_wait_cqe(&io_uring_, &cqe);  // Block until a completion is available

            // Free the buffer once the write is completed
            char* completed_buffer = reinterpret_cast<char*>(cqe->user_data);
            delete[] completed_buffer;

            io_uring_cqe_seen(&io_uring_, cqe);  // Mark CQE as seen
        }
    }
}
```

Figure 4.1.6.2 io_uring write function code

The write function is the most important part of the class, responsible for handling the actual log writes asynchronously. Each time a log message is passed to this function, the TurnSequencer ensures that the thread attempting to write waits for its "turn" in order to avoid

race conditions or conflicts. Once the thread's turn is up, a submission queue entry (SQE) is retrieved from io_uring, and the log message is copied into a new buffer. This buffer is then submitted for a non-blocking write operation using io_uring_prep_write. The buffer's address is stored in user_data to allow for clean-up once the write operation is complete. After enough writes have been queued (half of the defined queue depth), the system submits these requests for execution and processes any completed writes by freeing the associated buffers.

One of the critical variables in this class is count_, an atomic counter that tracks the number of log messages submitted for writing. Once the counter reaches half the defined queue depth, the function submits the batched writes to io_uring. This batching reduces the frequency of submissions, improving overall performance. Another important atomic variable is turn_, which works alongside TurnSequencer to ensure that threads perform their writes in an orderly fashion. By incrementing turn_ atomically and ensuring that each thread waits for its turn to submit, the class avoids the contention and disorder that can occur in multi-threaded environments.

```
logging::IoContext::~IoContext() {
    if (count_ != 0) {
        io_uring_submit(&io_uring_);

        struct io_uring_cqe* cqe;
        for (int i = 0; i < count_; ++i) {
            int ret_wait = io_uring_wait_cqe(&io_uring_, &cqe);  // Block until a completion is available

            // Free the buffer once the write is completed
            char* completed_buffer = reinterpret_cast<char*>(cqe->user_data);
            delete[] completed_buffer;

            io_uring_cqe_seen(&io_uring_, cqe);  // Mark CQE as seen
        }
    }
    io_uring_queue_exit(&io_uring_);
}
```

Figure 4.1.6.3 io_uring destructor code

The destructor of IoContext (~IoContext) ensures that any remaining writes are flushed and completed before the object is destroyed. If there are pending write operations, the destructor submits them and waits for their completion, cleaning up the buffers used for writing. This guarantees that no data is lost when the object is deallocated. Finally, the io_uring_queue_exit function is called to free the resources used by io_uring, ensuring that all associated resources are properly released.

This implementation offers several advantages over traditional I/O mechanisms. By using io_uring, the system minimizes the number of context switches between user and kernel space, reducing latency and improving throughput for I/O operations. It also allows for batching of operations, meaning that multiple I/O requests can be submitted at once and completed asynchronously. This is especially beneficial for high-throughput systems like logging libraries, where many I/O operations can occur concurrently without slowing down the rest of the application.

Overall, the IoContext class is designed to make efficient use of io_uring for asynchronous, non-blocking file writes, which is particularly well-suited for high-performance logging in a multithreaded environment.

4.2 System Build Process

I followed a systematic approach using CMake to streamline the configuration and build process. The entire project is designed to be modular allowing for flexibility in integrating different components like fmt and liburing libraries. The primary goal was to ensure that the logging library is built efficiently while allowing users to customize specific parameters like optimization flags and debugging options.

Before beginning the build, users need to install the required dependencies such as fmt for formatting output and liburing for efficient asynchronous I/O. These libraries are not bundled within the project so they must be installed on the user's system especially if they are running WSL2 Ubuntu. The installation steps for these dependencies are straightforward.

To install fmt on Ubuntu, you can use the following commands:

```
sudo apt update
sudo apt install libfmt-dev
```

Figure 4.2.1 Installation command for fmt library

For liburing, you will need to clone the repository and build it manually:

```
sudo apt update
sudo apt install liburing-dev
```

Figure 4.2.2 Installation command for liburing library

Once both dependencies are installed, you can proceed with building the logging library. After cloning the repository containing the library, users can generate the necessary build files and compile the project with a few commands. To do so, run:

```
cmake -B build
make
```

Figure 4.2.3 Build command for my library

The cmake -B build command configures the project by generating build files in the build directory which keeps the source tree clean. After the build files are prepared, running the make command compiles the project and produce the desired output including the library and executable.

If any issues arise during the installation or configuration process, or if you'd like to see how the libraries are linked and managed in the build process, you can refer to the CMakeLists.txt file which is available in my GitHub repository https://github.com/Chunee/Log-Library/tree/main. This file contains all the necessary details for linking external libraries and setting up the project's build and compilation flags.

```cpp
int main() {
    logging::Log log;
    int user_id = 42;
    std::string error_message = "File not found";

    log.setOutputFile("output.txt");

    log.error("User ID: {}, Error message: {}", user_id, error_message);
    return 0;
}
```

Figure 4.2.4 Example of using my library

To begin using the library, users simply instantiate a logging::Log object which provides the core logging functionality. Once this object is created, users can easily specify an output file for their log entries with the setOutputFile() function.

The API further simplifies logging by allowing users to write log messages with varying levels of importance such as error messages. By using formatted logging methods like log.error(), users can seamlessly include dynamic content such as variables within their log entries.

output.txt

```
2024-08-03 11:35:38.259 140048957253504  /home/chunee/log/test.cpp:20 [error] User ID: 42, Error message: File not found
```

Figure 4.2.5 Output example shown

CHAPTER 5 SYSTEM IMPLEMENTATION

5.1 Hardware Setup

Description	Specifications
Model	ROG Strix G531GT_G531GT
Processor	Intel Core i5-9300H (2.4 GHz)
Operating System	Windows 10 (64-bits)
Graphic	NVIDIA GeForce GTX 1650
Memory	8GB RAM (DDR4)
Storage	512GB SSD

Table 5.1.1 Hardware Setup

5.2 Software Setup

Before starting to develop this high performance logging library, there are four software needed to be installed and downloaded in my laptop:

1. Ubuntu 22.04
2. Git 2.34.0
3. CMake 3.27.0-rc4
4. G++ 11.4.0

5.3 Settings and Configuration

This logging library is configured using CMake and compiled with the G++ compiler and ensuring compatibility with C++20 standards. First, the CMake version 3.27.0-rc4 is used as the build system generator. The CMakeLists.txt file is created to define the necessary project configuration including the required version of CMake and C++20 as the language standard.

The project uses multiple external libraries such as fmt for formatting and liburing for asynchronous I/O, which are linked using the find_package command in CMake. The G++ 11.4.0 compiler is specified for compiling the project, with necessary flags like -std=c++20 to ensure compliance with modern C++ standards. All project files are organized into respective source and header directories which are then included in the CMake configuration to automate the build process.

Once configured, the cmake command like cmake -B build is executed to generate the makefile followed by the make command to compile the logging library. The CMake configuration ensures that all dependencies and necessary compiler flags are handled, allowing the logging library to be built efficiently on the specified system setup.

CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION

6.1 System Testing and Performance Metrics

6.1.1 Performance Comparison

```
for (int j = 0; j < 30; ++j) {
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 100; ++i) {
        log.error("User ID: {}, Error message: {}", i, error_message);
    }

    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
```

Figure 6.1.1.1 Test case for single-threaded scenario

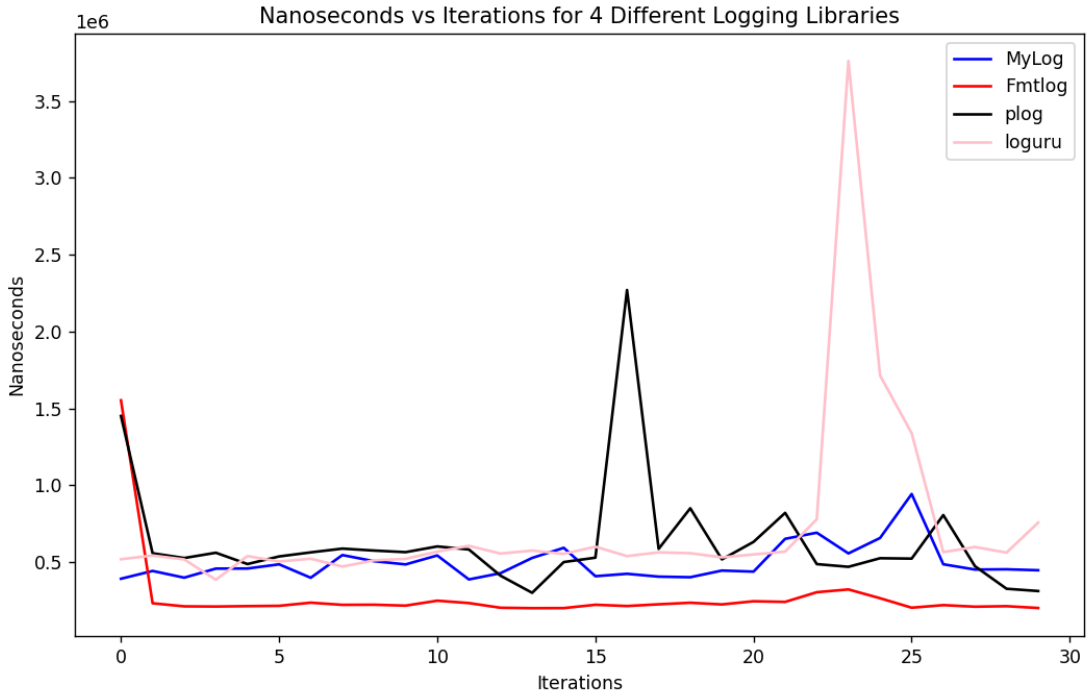


Figure 6.1.1.2 Result of nanoseconds vs iterations

The graph above compares the performance of my logging library (blue line), with three other logging libraries such as fmtlog (red line), plog (black line), and loguru (pink line) over 30 test executions. The x-axis represents the number of times I tested (from 1 to 30 iterations), while the y-axis shows the time (in nanoseconds) required to log a message 100 times within a loop.

Based on the result, my logging library generally performs well, sitting between fmtlog and plog in terms of consistency and speed. It maintains a relatively low and stable nanosecond time, indicating that it handles logging operations efficiently within the tested framework. My logging library doesn't experience large spikes like plog or loguru, which suggests it is more robust in terms of avoiding performance degradation across different test iterations. However, fmtlog slightly outperforms my logging library in terms of overall nanoseconds.

My logging library demonstrates stable and consistent performance across all 30 iterations, only slightly trailing fmtlog. The performance is superior to both plog and loguru which show occasional performance spikes. The consistent performance of my logging library along with its minimal fluctuation suggests that my design using fmt and MPMC queues is effective for handling high-frequency logging tasks without significant overhead. However, fmtlog still outperforms the others which may be due to optimizations that I could explore to further improve my logging library's efficiency.

```
for (int j = 0; j < 30; ++j) {
    std::vector<std::thread> threads;
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 16; ++i) {
        threads.push_back(std::thread([i, user_id, error_message, &log] {
            log.error("User ID: {}, Error message: {}", i, error_message);
        }));
    }

    for (auto&t : threads) {
        t.join();
    }

    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
```

Figure 6.1.1.3 Test case for multi-threaded scenario

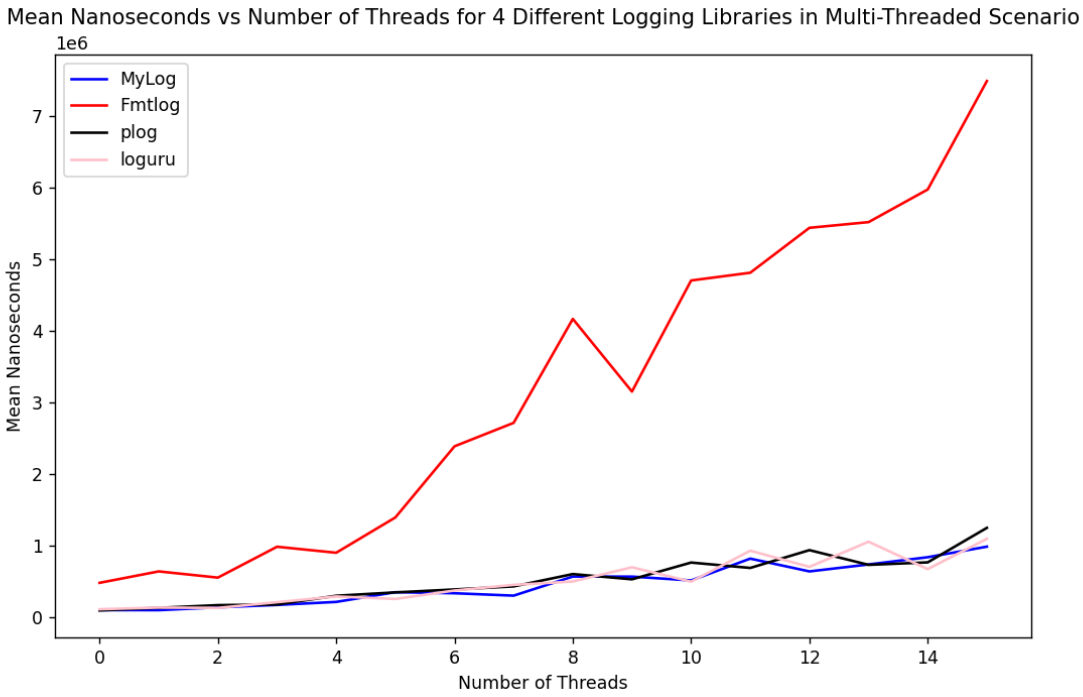


Figure 6.1.1.4 Result of mean nanoseconds vs number of threads

The graph above illustrates the performance of four different logging libraries which are my library, fmtlog, plog, and loguru measured in terms of mean nanoseconds per iteration. The x-axis represents the thread spawned count while the y-axis shows the mean time (in nanoseconds) required for logging operations in each iteration. Each data point is the mean of 30 iterations for logging operations performed with varying numbers of threads, starting from 1 thread and gradually increasing up to 16 threads.

The purpose of the experiment is to compare how these libraries handle multi-threaded logging in a high-concurrency scenario assessing both their performance and scalability as the number of threads increases.

Based on the image given above, fmtlog (Red Line) shows a steep increase in time with each iteration which indicates that it faces significant overhead as the iterations increase. The potential reason for this is fmtlog's design by using thread-local storage (TLS). Each thread maintains its own log buffer which can lead to contention or inefficiencies when the data from these threads must be merged or ordered.

Fmtlog's approach of adjusting ordering using a heap may also introduce overhead because merging logs from different threads requires additional computation to ensure that the log entries are ordered correctly. As the number of iterations grows, this ordering process could be more computationally expensive, resulting in the observed increase in time.

Loguru (Pink Line) and Plog (Black Line) perform relatively well in the graph compared to fmtlog but they still experience a slight increase in time as iterations progress. The reason behind this could be the << operator they use for formatting logs. While this operator is familiar in C++ and relatively fast, it is not as optimized as the fmt library. This could cause minor inefficiencies when constructing log messages especially under high-volume, multi-threaded workloads.

Despite using <<, these libraries seem to handle concurrency well and don't suffer the same performance degradation as fmtlog. This suggests that the concurrency mechanism they employ is effective for managing log writes but may not be as scalable under extreme conditions.

My library (blue line) shows a stable performance with minimal growth in time across iterations. It leverages the fmt library for string formatting and utilizes an MPMC (multi-producer, multi-consumer) queue. The use of fmt provides efficient formatting compared to the << operator. Additionally, the use of the MPMC queue with a turn sequencer and futex-based synchronization offers a highly optimized, lock-free or low-contention mechanism for managing concurrent log writes across threads.
In conclusion, the graph reflects the performance trade-offs of each logging library in a multi-threaded environment. Fmtlog experiences the most significant overhead due to thread-local storage and heap-based ordering, while loguru and plog though faster, are limited by their user-friendliness of the << operator for formatting. My library is leveraging the fmt library and a highly optimized concurrency mechanism that outperforms the others demonstrating the effectiveness of combining modern formatting tools with efficient MPMC queues.

6.1.2 User-Friendly API Comparison

```cpp
for (int j = 0; j < 30; ++j) {
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 100; ++i) {
        log.error("User ID: {}, Error message: {}", i, error_message);
    }

    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
```

Figure 6.1.2.1 My logging library API example

```cpp
for (int j = 0; j < 30; ++j) {
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 100; ++i) {
        PLOG_DEBUG << "User ID: " << i << ", Error message: " << error_message;
    }

    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
```

Figure 6.1.2.2 plog API example

```cpp
for (int j = 0; j < 30; ++j) {
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 100; ++i) {
        LOG_F(ERROR, "User ID: %d, Error message: %s", i, error_message.c_str());
    }

    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
```

Figure 6.1.2.3 loguru API example

```cpp
for (int j = 0; j < 30; ++j) {
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < 100; ++i) {
        FMTLOG(fmtlog::ERR, "User ID: {}, Error Message: {}", i, error_message);
    }
    fmtlog::poll();

    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
```

Figure 6.1.2.4 fmtlog API example

When developing a logging library, one of the key goals is to create a system that is both powerful and user-friendly. Comparing the use of the fmt library in my logging library and fmtlog to other popular logging libraries such as plog and loguru clearly demonstrates how different logging frameworks can affect the user experience especially in terms of readability and ease of use.

In the case of plog as illustrated in the image, the library relies heavily on the use of the << operator for message formatting. While this can work for simple messages, it quickly becomes cumbersome as the complexity of the log message increases. Each component of the message such as a user ID or error message, must be separately chained using <<. This can clutter the code making it less readable and more prone to errors especially if the order or type of arguments changes.

On the other hand, loguru adopts a printf-style format string approach which require users to specify format specifiers like %d for integers or %s for strings. While this method is efficient and familiar to many, it introduces a level of complexity that can be frustrating. Users must be aware of the data type of each argument and match it with the correct format specifier. This not only increases the chance of errors such as using %d for a floating-point number but also requires additional steps like converting std::string objects to const char* using c_str().

In contrast, a logging library built using the fmt library offers a much more streamlined and intuitive experience. With fmt, users can simply include placeholders ({}) in the log message, and the library automatically handles type resolution at runtime. There is no need for users to manually specify the type of each argument nor are there any additional steps to convert data types. This reduces the mental overhead for developers and allows them to focus on the content of the message not the formatting. As a result, the code remains clean, concise, and easy to maintain, even when logging complex messages with multiple variables.

```
std::vector<int> vec{1, 2, 3};
log.error("Elements of vector: {}", vec);
```

Figure 6.1.2.5 my API example for specific data structure

Figure 6.1.2.6 output

The fmt library also offers an intuitive and powerful approach to logging complex data types such as std::vector. As shown in the image, the fmt library allows the user to log the contents of a vector directly by passing the vector as an argument in the placeholder {}. This level of flexibility is one of the key advantages of using fmt as it automatically formats the elements of the vector and produce a clear and readable log message without requiring any additional manipulation of the data.

In contrast, libraries like plog and loguru do not provide such out-of-the-box support for logging complex data types like std::vector. In the case of plog, the operator << is primarily designed for handling simple data types mean users would have to manually iterate over the vector and use multiple << operators to log each element. This process can be cumbersome and leads to less readable code.

Similarly, loguru which relies on a printf-style formatting system does not directly support containers like std::vector. Users would need to implement custom formatting logic or convert the vector into a more suitable form for logging which complicates the logging process and requires additional steps such as writing loops or helper functions to convert the vector elements into a string format.

In conclusion, my logging library still offers nearly the same speed as plog and loguru while maintaining superior user-friendliness, thanks to its seamless integration with the fmt library. While libraries like fmtlog may outperform in raw speed especially in single-threaded scenarios due to its highly optimized design, it falls short in multi-threaded environments.

In contrast, my logging library is designed to handle concurrency effectively while still utilizing the powerful formatting capabilities of fmt. This allows for a balanced trade-off between speed and multi-threading support, ensuring that performance is not compromised

even when handling complex data types like std::vector or custom objects,which neither plog nor loguru support as easily. Additionally, the simplicity of using {} placeholders for formatting without the need for chaining multiple operators as in plog or specifying format specifiers like in loguru makes the logging process both faster to write and cleaner to read. Thus, my library combines high performance, multi-threading capability, and ease of use hence providing a robust solution for modern logging needs.

6.2 Project Challenges

In this project, several challenges emerged and each presenting a unique obstacle in the development process. One of the initial difficulties involved configuring CMake to integrate external libraries like fmt and liburing. Ensuring that these libraries are correctly linked and compatible with the rest of the project often requires troubleshooting complex build configurations which can be particularly time-consuming and error-prone.

Using tools like GDB to trace and identify issues in concurrent threads presents challenges as traditional debugging techniques often fall short when tracking interactions between threads. Identifying race conditions, deadlocks, and other multithreading issues requires patience and a deep understanding of both the program's behavior and the debugger itself.

Memory management also poses a persistent challenge. Detecting memory leaks, invalid access, and other subtle issues can often go unnoticed without specialized tools or testing environments. These issues can lead to unpredictable behaviors that are difficult to replicate and debug and also adding further complexity to the development process if left unchecked.

In addition to these difficulties, understanding and implementing advanced optimization techniques such as atomic operations and MPMC (Multiple Producer Multiple Consumer) queues is essential for ensuring that the logging library performs well in high-concurrency scenarios. These techniques require a solid understanding of low-level performance concepts and their practical implications on the system.

Furthermore, learning new concepts like liburing which handles efficient asynchronous I/O operations presents its own challenge, as mastering unfamiliar libraries while developing an application requires both time and dedicated effort. Finally, becoming proficient with the Vim editor is necessary for efficient code navigation and editing, yet it comes with a learning curve

that may slow initial productivity. Overcoming these hurdles is essential to the success of the project.

6.3 Objective Evaulation

The project successfully met its objective of enhancing user-friendliness by incorporating the fmt library into the logging framework. The fmt library's ability to format log messages with a clean, intuitive syntax significantly improved the developer experience. As a result, complex logging statements became more readable and easier to maintain.

In addition, the objective of implementing C++ template metaprogramming was achieved, optimizing code generation at compile time. This approach minimized runtime overhead by generating type-safe and efficient logging code. The successful integration of template metaprogramming techniques allowed developers to use the logging library without performance penalties and ensure that it delivered on the promise of high performance and seamless integration within applications.

Concurrency challenges in multi-threaded environments were effectively addressed by employing a multi-producer multi-consumer (MPMC) queue. This implementation ensured that multiple threads could concurrently produce and consume log messages without significant bottlenecks. By using the MPMC queue, the logging framework handled high levels of concurrency and minimizing contention between threads. This solution not only increased the scalability of the system across multiple cores but also ensured thread safety throughout the logging process.

Furthermore, the project succeeded in improving performance and efficiency through the use of io_uring for asynchronous I/O operations. The io_uring API allowed for non-blocking writes and this reduce the reliance on traditional blocking I/O calls and enhancing overall system responsiveness. As a result, the logging library exhibited strong performance in high-load scenarios.

Overall, this project achieved its goal of creating a high-performance, user-friendly logging, and simpler to learn logging library exclusively for Linux and C++. Each of the stated

objectives was successfully implemented and provided a modern C++ solution that is both accessible and efficient.

CHAPTER 7 Conclusion and Recommendation

7.1 Conclusion

In conclusion, this project provided a valuable and enriching experience particularly in managing concurrency. Although working with concurrency presented significant challenges, it was also an engaging process that allowed me to deepen my understanding of multi-threaded programming. Successfully implementing the MPMC queue and ensuring thread safety while minimizing contention was a rewarding accomplishment which highlighted the complexity and importance of concurrency in modern software development.

Additionally, io_uring proved to be an excellent framework for handling asynchronous I/O operations. Its non-blocking nature and efficient I/O management enhanced the performance of the logging library particularly in high-demand environments. This project reinforced how io_uring can be a powerful tool for building scalable and responsive applications and I found its integration to be one of the highlights of the development process.

The project also introduced me to key concepts in C++ metaprogramming, concurrent lock-free data structures, and atomic operations. Leveraging template metaprogramming for optimizing code at compile-time, understanding how to implement lock-free data structures, and working with atomic primitives greatly expanded my knowledge and skills in these advanced C++ topics. This exposure has been both challenging and rewarding, offering insights into building efficient and scalable systems.

Overall, this project was a valuable learning experience, and I look forward to continuing my work in this area. I hope to further contribute to the development and refinement of this logging library, applying the lessons I've learned and exploring even more sophisticated techniques to optimize performance and concurrency management.

7.2 Recommendation

In the future, several enhancements could be implemented to further improve the functionality and versatility of the logging library. First and foremost, expanding the API offerings is

essential to provide developers with more flexibility in how they interact with the library. While the current set of features addresses the core requirements of logging, additional APIs could be added to accommodate a wider range of logging scenarios such as set the thread name, and etc. This would give developers more control over the logging process and allow for finer-grained management of log output.

Additionally, expanding the output destinations for log messages would greatly enhance the library's utility in modern applications. Currently, the library primarily logs to files, but introducing support for logging to web services, consoles, or even remote systems would make it more adaptable to various use cases. For example, applications running in cloud environments often require logs to be sent to centralized services, making web or network-based logging essential. The ability to output logs to multiple locations simultaneously such as a file and a console would also provide more flexibility for developers especially during debugging and development phases.

Another key recommendation is the implementation of a termination mechanism when logging at the "fatal" level. In many critical systems, a fatal log entry often signifies an unrecoverable error and immediate termination of the application is necessary. Adding this feature would make the logging library more robust and ensure that fatal errors are handled appropriately and preventing further damage or data corruption in the event of critical failures.

Lastly, expanding the logging library to support additional operating systems could increase its applicability across different environments. While the library currently focuses on Linux, adding support for platforms like Windows and macOS would make it more versatile and accessible to a wider range of developers.

REFERENCES

[1] Fmtlib. “A modern formatting library” GitHub
https://github.com/fmtlib/fmt (accessed September. 12, 2024)

[2] “{fmt} A modern formatting library”. fmt.dev.
https://fmt.dev/latest/index.html (accessed September. 12, 2024)

[3] CppCon, Charles Frasch, USA. *Single Producer Single Consumer Lock-Free FIFO From the Ground-Up*.(2023). Accessed: September. 12, 2024. [Online Video].
Available: https://www.youtube.com/watch?v=K3P_Lmq6pw0

[4] CppCon, Alex Dathskovsky, USA. *C++ Memory Model: from C++11 to C++23*. (2023). Access: September. 12, 2024. [Online Video].
Available: https://www.youtube.com/watch?v=SVEYNEWZLo4

[5] fmtlog. "fmtlog is a performant fmtlib-style logging library with latency in nanoseconds." GitHub. https://github.com/MengRao/fmtlog (accessed September. 12, 2024)

[6] plog. "Portable, simple and extensible C++ logging library" GitHub
https://github.com/SergiusTheBest/plog (accessed September. 12, 2024)

[7] loguru. "A lightweight C++ logging library" GitHub
https://github.com/emilk/loguru (accessed September. 12, 2024)

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: 3, 3	**Study week no.: 2**
Student Name & ID: Low Chun Ee 2106572	
Supervisor: Ts. Wong Chee Siang	
Project Title: High Performance Logging Library for Run-Time Efficiency with Multithreaded Support	

1. WORK DONE [Please write the details of the work done in the last fortnight.] - implemented thread local storage - able to log out the message - fixed some bugs
2. WORK TO BE DONE - solve ordering of message problem
3. PROBLEMS ENCOUNTERED - solutions to solve ordering of message problem
4. SELF EVALUATION OF THE PROGRESS - not bad

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: 3, 3	**Study week no.: 6**
Student Name & ID: Low Chun Ee 2106572	
Supervisor: Ts. Wong Chee Siang	

Project Title: High Performance Logging Library for Run-Time Efficiency with Multithreaded Support

1. WORK DONE [Please write the details of the work done in the last fortnight.] - implemented multi-producer multi-consumer queue
2. WORK TO BE DONE - benchmark the result
3. PROBLEMS ENCOUNTERED - Build multiple open source libraries
4. SELF EVALUATION OF THE PROGRESS - Not bad

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: 3,3	**Study week no.: 10**
Student Name & ID: Low Chun Ee 2106572	
Supervisor: Ts. Wong Chee Siang	
Project Title: High Performance Logging Library for Run-Time Efficiency with Multithreaded Support	

1. WORK DONE [Please write the details of the work done in the last fortnight.] - Benchmark the performance
2. WORK TO BE DONE - Write the report - Discuss the performance result
3. PROBLEMS ENCOUNTERED - None
4. SELF EVALUATION OF THE PROGRESS - Not bad

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: 3,3	**Study week no.: 11**
Student Name & ID: Low Chun Ee 2106572	
Supervisor: Ts. Wong Chee Siang	
Project Title: High Performance Logging Library for Run-Time Efficiency with Multithreaded Support	

1. WORK DONE [Please write the details of the work done in the last fortnight.] - Discussed the performance result
2. WORK TO BE DONE - Write the report and poster
3. PROBLEMS ENCOUNTERED - None
4. SELF EVALUATION OF THE PROGRESS - Not bad

Supervisor's signature

Student's signature

POSTER

HIGH PERFORMANCE LOGGING LIBRARY FOR RUNTIME EFFICIENCY WITH MULTI-THREADED SUPPORT

A C++ logging library that is user-friendly, simpler to learn, and optimized using template metaprogramming, lock free MPMC queue, and io_uring for asynchronous I/O

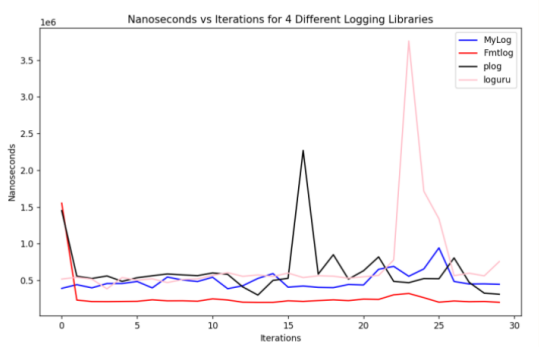
INTRODUCTION

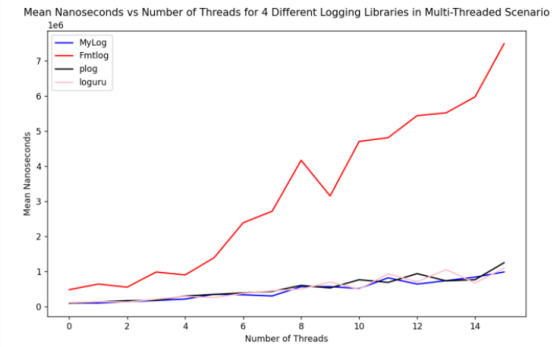
- Logging libraries play a crucial role in capturing program output, helping developers track the behavior of applications.
- Unlike simple print statements (e.g., std::cout, fstream << operator), logging libraries provide powerful features like log levels, formatting such as timestamp, location of line, and thread id, and high performance.
- Efficient logging is essential in applications with high performance demands, such as web servers or real-time systems, where logging operations must not slow down the core functionality.

OBJECTIVE

- enhance the user-friendliness of the logging framework by incorporating the fmt library.
- implement logging framework using C++ template metaprogramming techniques to optimize code generation during compile time.
- address concurrency challenges in multi-threaded environments by utilizing multi-producer multi-consumer queue in C++.
- enhance I/O operation by using io_uring library in Linux.

BENCHMARK RESULT



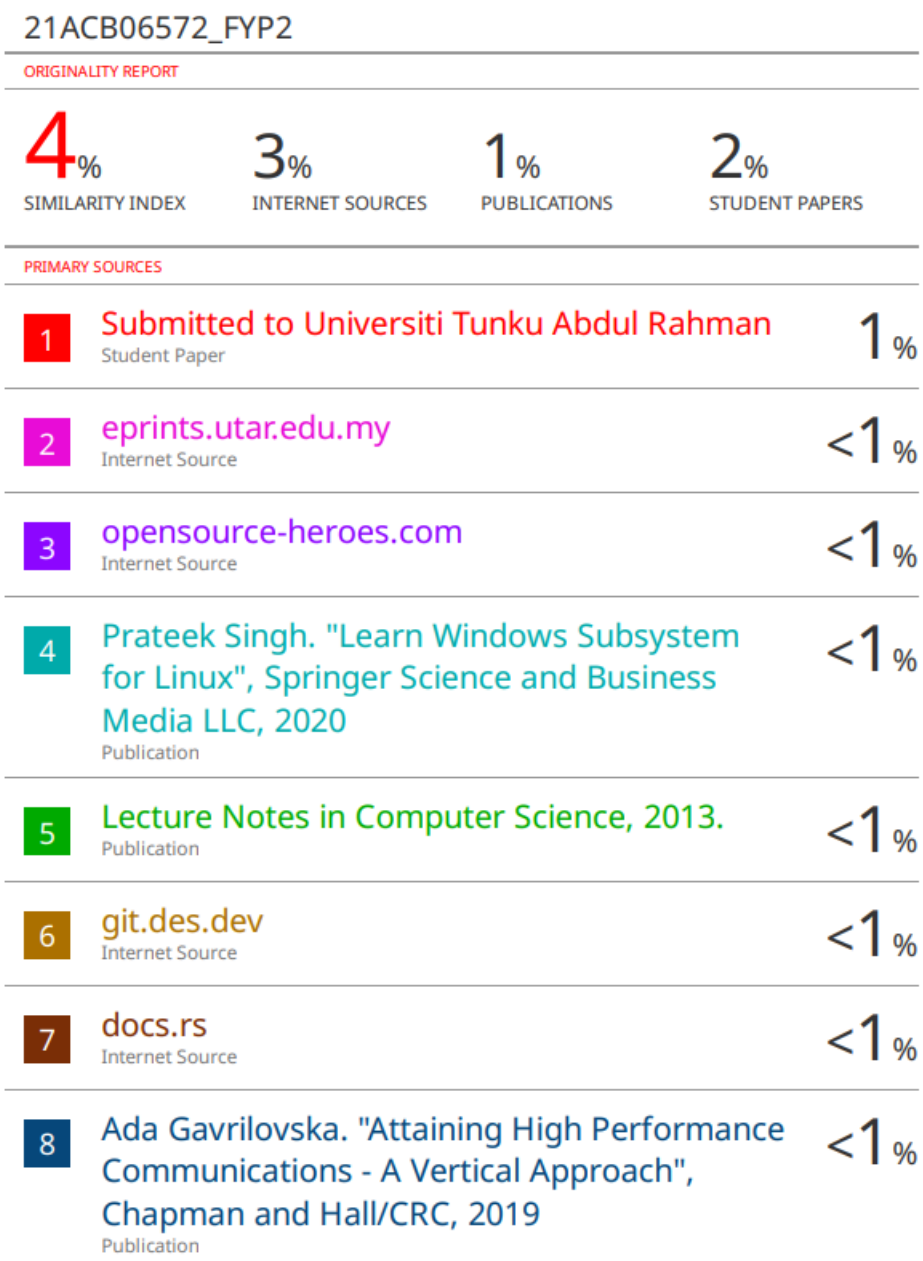


ARCHITECTURE DIAGRAM

CONCLUSION

- My logging library demonstrates consistent and low-latency performance across multi-threaded scenarios, outperforming competitors like Fmtlog, which struggle with increased thread counts and logging loads.
- Planned enhancements include expanding the API, supporting logging to web services, and implementing fatal error handling for immediate application termination in critical situations.

PLAGIARISM CHECK RESULT



21ACB06572_FYP2

ORIGINALITY REPORT

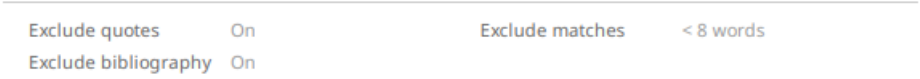
4%	3%	1%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Universiti Tunku Abdul Rahman Student Paper	1%
2	eprints.utar.edu.my Internet Source	<1%
3	opensource-heroes.com Internet Source	<1%
4	Prateek Singh. "Learn Windows Subsystem for Linux", Springer Science and Business Media LLC, 2020 Publication	<1%
5	Lecture Notes in Computer Science, 2013. Publication	<1%
6	git.des.dev Internet Source	<1%
7	docs.rs Internet Source	<1%
8	Ada Gavrilovska. "Attaining High Performance Communications - A Vertical Approach", Chapman and Hall/CRC, 2019 Publication	<1%

9	fict.utar.edu.my Internet Source	<1 %
10	Submitted to Indiana Wesleyan University Student Paper	<1 %
11	Submitted to Midlands State University Student Paper	<1 %
12	www.javaskool.com Internet Source	<1 %
13	Submitted to Purdue University Student Paper	<1 %
14	Submitted to Queen's College Student Paper	<1 %
15	Submitted to Swinburne University of Technology Student Paper	<1 %
16	Peizhao Ou, Brian Demsky. "Checking Concurrent Data Structures Under the C/C++11 Memory Model", Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '17, 2017 Publication	<1 %
17	Tao Ma, Xin-Yu Liu, Shuang-Long Cai, Jin Zhang. "Development and validation of a nomogram for predicting rapid relapse in	<1 %

	triple-negative breast cancer patients treated with neoadjuvant chemotherapy", Frontiers in Cell and Developmental Biology, 2024 Publication	
18	www.power.org Internet Source	<1 %
19	cms.faa.gov Internet Source	<1 %
20	export.arxiv.org Internet Source	<1 %
21	ryonaldteofilo.medium.com Internet Source	<1 %
22	Lorenzo Quirós Díaz. "Layout Analysis for Handwritten Documents. A Probabilistic Machine Learning Approach", Universitat Politecnica de Valencia, 2022 Publication	<1 %
23	Shams Al Ajrawi, Charity Jennings, Paul Menefee, Wathiq Mansoor, Mansoor Ahmed Alaali. "chapter 9 Advanced C++ Programming Techniques", IGI Global, 2024 Publication	<1 %
24	api.mountainscholar.org Internet Source	<1 %


Exclude quotes On
Exclude bibliography On
Exclude matches < 8 words

Universiti Tunku Abdul Rahman			
Form Title : Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 01/10/2013	Page No.: 1of 1

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Low Chun Ee
ID Number(s)	2106572
Programme / Course	Bachelor of Computer Science
Title of Final Year Project	High Performance Logging Library for Runtime Efficiency with Multi-threaded Support

Similarity	**Supervisor's Comments (Compulsory if parameters of originality exceeds the limits approved by UTAR)**
Overall similarity index: 4 % **Similarity by source** Internet Sources: 3 % Publications: 1 % Student Papers: 2 %	
Number of individual sources listed of more than 3% similarity: 0	
Parameters of originality required and limits approved by UTAR are as Follows: **(i) Overall similarity index is 20% and below, and** **(ii) Matching of individual sources listed must be less than 3% each, and** **(iii) Matching texts in continuous block must not exceed 8 words** ***Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.***	

Note Supervisor/Candidate(s) is/are required to provide softcopy of full set of the originality report to Faculty/Institute

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.

Signature of Supervisor

Name: Ts. Wong Chee Siang

Signature of Co-Supervisor

Name: Mr Tan Chiang Kang @ Thang Chiang Kang

Date: 12/9/2024 Date: 12/9/2024

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
(KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	2106572
Student Name	Low Chun Ee
Supervisor Name	Ts. Wong Chee Siang

TICK (√)	DOCUMENT ITEMS Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
√	Title Page
√	Signed Report Status Declaration Form
√	Signed FYP Thesis Submission Form
√	Signed form of the Declaration of Originality
√	Acknowledgement
√	Abstract
√	Table of Contents
√	List of Figures (if applicable)
√	List of Tables (if applicable)
	List of Symbols (if applicable)
	List of Abbreviations (if applicable)
√	Chapters / Content
√	Bibliography (or References)
√	All references in bibliography are cited in the thesis, especially in the chapter of literature review
	Appendices (if applicable)
√	Weekly Log
√	Poster
√	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)
√	I agree 5 marks will be deducted due to incorrect format, declare wrongly the ticked of these items, and/or any dispute happening for these items in this report.

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report.

_____
(Signature of Student)

Date: 12/9/2024