The Design and Architecture of a Multithreaded Web Server

Design Document

Qasim Ijaz

CS3821 — BSc Final Year Project

Supervised by: Dr DongGyun Han
Department of Computer Science
Royal Holloway, University of London

Contents

1 Abstract

1.1 Introduction

This comprehensive design document presents the architecture and workflow for the multi-threaded web server developed for my BSc final year project. This report encompasses various aspects of the project, from the technology stack, component design, concurrency model, server core functionalities, performance optimisation, and scalability and reliability measures. The technology stack section outlines the programming languages, development tools, compilers, and testing, profiling, and sanitisation tools employed in the project. The component design section delves into the initial single-threaded web server, thread pool architecture, LRU cache implementation, and memory mapping strategy.

The concurrency model section details threading and synchronisation mechanisms, addressing the handling of challenges like race conditions and deadlocks. It also discusses thread safety and the use of atomic operations to ensure a robust multi-threaded environment. Server core functionalities cover the request processing pipeline, HTTP protocol handling, and connection management. Throughout the report I will discuss and mention the important of optimisation and how various aspects of how the server is designed will affect the operation, efficiency and scalability of the multi-threaded web server.

This document provides a comprehensive overview of the multi-threaded web server project, addressing design choices and strategies to handle challenges, optimise performance, ensure scalability and reliability in a real-world context.

2 Technology Stack

2.1 Programming Languages

The first part of the technology stack section is the choice of programming language. The choice of programming language is an extremely important one, this decision is not a matter of preference; rather, it is a strategic selection with serious implications for the overall performance, scalability, and maintainability of our multi-threaded web server. In this section, we delve into the rationale behind this choice and why the languages chosen are the best suited for the task of developing a multithreaded web server.

2.2 The C++ Programming Language

The main programming language selected for the development of the multithreaded web server is C++, a language known for its versatility. C++ is a feature rich language which provides users with a balance between high-level abstractions and low-level control [3], providing a flexible environment for efficient development. The intricate demands of our web server project requires nuanced control over multiple low-level functionalities, and C++ empowered us to navigate complex memory management intricacies, optimise performance, and engage closely with the underlying hardware.

One of the key features of C++ in our context is its optimisation capabilities. The language provides an extensive set of tools to fine-tune code [14], allowing us to tailor the software for maximum efficiency and optimal resource utilisation. This level of control is vital in crafting a high-performance multi-threaded server. In summary, the choice of C++ as the primary programming language stems from its versatility, control over low-level operations and optimisation capabilities. This decision lays a robust foundation, enabling the development of a resilient and high-performance multi-threaded web server tailored to meet the challenges of concurrent processing.

3 Development Tools and Compilers

3.1 Integrated development environment (IDE)

The efficiency and effectiveness of developing a multithreaded web server is closely tied to the development tools employed. The selection of development tools is vital and will affect the project's overall performance, efficiency and scalability. The first major development tool I will use whilst developing the multithreaded web server is VSCode . The selection of Visual Studio Code as our primary integrated development environment (IDE) lies in the fact that VSCode provides users with a feature-rich development experience. VSCode 's intuitive interface, enriched by a variety of extensions/plugins, and robust debugging capabilities elevate our development process. By leveraging VSCode , our development commitments extend beyond a choice of tools; it showcases a dedication to ensuring code readability and ensuring an efficient development life-cycle for our multi-threaded web server project.

3.2 GCC and Clang Compilers

The next tool that will be vital throughout development is the choice of compiler. Our project leverages two powerful compilers: GCC (g++) and Clang. GCC provides robust optimisation capabilities and broad platform support [13]. Clang is known for its fast and expressive diagnostics and error messages, these unique attributes make each compiler useful for different things and both contribute to enhanced code readability and maintainability [9]. Compiler flags used with GCC and Clang are another vital factor which will impact performance and how the final binary executable functions, for example flags such as -O3 and -O2 play a pivotal role in optimisation [7], influencing the balance between speed and size. Additionally, flags like -march=native and -flto enable compiler-level optimisations [4] tailored to the underlying hardware and support link-time optimisation, respectively. These flags underscore the significance of compilers in determining how optimised and tailored to the target platform the code ultimately becomes.

In essence, the compilers selected and the use of compiler flags are integral to the project's success. They embody the commitment to performance and cross-platform compatibility. By understanding and leveraging the capabilities of GCC, Clang, and associated flags, we ensure that our multi-threaded web server is not only well-crafted in its code but also finely tuned for optimal performance across various environments and platforms.

4 Testing, Profiling and Sanitisation Tools

During the development of our multi-threaded web server, the selection of testing, profiling, and sanitisation tools stands as an important part in ensuring the robustness, performance optimisation, and security of our codebase. In this section I will outline the various tools I will use while developing the multithreaded web server. The tools I will cover include testing tools and frameworks, code profiling tools and code sanitisation tools.

4.1 Linux Perf

One of the key profiling tools I will utilise for performance testing throughout development of the multithreaded web server is Linux Perf , this tool offers a powerful suite of tools to delve deep into the performance metrics of a program [2] which is essential for optimising our web server. Through Perf , we gain insights into key performance indicators, including cache misses, CPU cycles, and other vital statistics. This data is not just informative but instrumental in guiding our optimisation efforts, allowing us to fine-tune our multi-threaded architecture for optimal performance. The integration of Perf ensures that our performance enhancements are targeted at specific parts of the web server based on the statistics from Perf , enabling targeted improvements and performance gains.

4.2 C++ std::chrono Library

An essential tool for performance measurement in the development of our web server is the C++ chrono library. This library enables precise time estimation within specific functions [8], offering valuable insights into the efficiency of different components. As we optimise our multi-threaded web server, C++ chrono becomes vital in identifying bottlenecks and areas for improvement. Its accuracy supports decision-making during the optimisation process, contributing to the overall efficiency of our codebase.

4.3 Catch2 Unit Testing Framework

Unit testing is a vital part of our development process. I will be using the Catch2 unit testing framework to conduct testing with the multithreaded web server. By leveraging Catch2 , we are able to systematically validate individual units of our code, ensuring that each component functions [1] as expected in isolation. The user-friendly syntax of Catch2 allows us, as developers, to effectively create and maintain unit tests, this improves the overall code reliability and maintainability. The usage of Catch2 allows us to improve the overall security of the web server. By allowing us to catch and rectify issues early in the development cycle, we are able to contribute to the overall stability of the multi-threaded web server.

4.4 Code Sanitisation (ASan, UBSan, TSan)

The last type of tool that will be vital in the development of the multithreaded web server is program sanitisers. Tools such as AddressSanitizer , UBSan , and TSan collectively form an advanced defence against bugs, undefined behaviour, and security vulnerabilities [15] within our codebase. AddressSanitizer detects memory-related issues, UBSan identifies instances of undefined behaviour and TSan ensures thread safety in our web server. The proactive integration of these sanitisation tools ensures the development of a secure and reliable multi-threaded web server. By identifying critical problems with the codebase at an early stage, these tools significantly contribute to the overall robustness and security of our codebase. Their inclusion aligns with the best practices in software development, where proactive measures against potential pitfalls could lead to a more secure and efficient multithreaded web server.

5 Component Design

In crafting the architecture of the multi-threaded web server, component design is vital for achieving efficiency, scalability, and reliability. This section delves into the foundational elements that make up the core of the server. From the initial single-threaded web server to the intricacies of thread pool component and LRU cache implementation, our focus is on creating a robust and well-organised structure. The following exploration highlights the rationale, considerations, and strategies underlying the design choices that define the core functionalities of the multi-threaded web server.

5.1 Initial Single-Threaded Web Server

The first core component of the multi-threaded web server is a single-threaded web server. This server contains the fundamental starting features of a web server which I will build on top of when developing the final multi-threaded web server. This initial server incorporates key features crucial for its role and subsequent evolution such as:

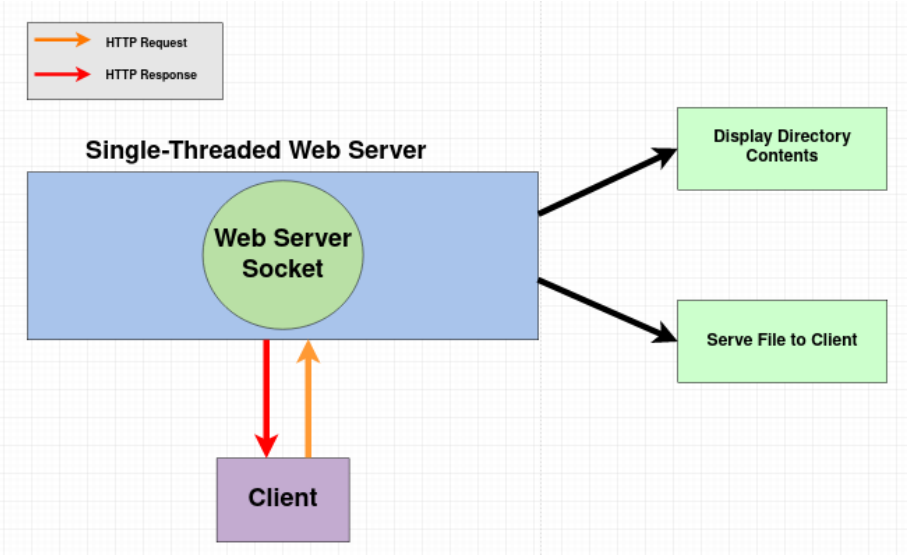

Figure 1: High Level Overview of Single-Threaded Web Server Design

- **Listening for Client Connection:** The server enters a waiting state, where it will wait for incoming requests on a specified port. This phase establishes a vital communication channel for potential interactions with clients.
- **Read HTTP Request:** Upon connection, the server retrieves the complete HTTP request from the client. This step is vital for understanding the client's intent and initiating the necessary actions.
- **Parse and Decode URL:** The server engages in parsing and decoding the URL embedded in the HTTP request. This process involves extracting the resource path and decoding any URL-encoded characters—a crucial step for accurate identification of the requested resource.

– **Log Request Details:** As a measure of transparency and accountability, the server logs essential details, including the source IP of the request, the specific resource sought and the date/time of the request being made. This logging mechanism serves operational needs and acts as a reliability measure through comprehensive record-keeping of requests and server errors.

– **Check for Directory Traversal Attack:** The server scrutinises the path that the HTTP request specifies to prevent unauthorised access to the file system, fortifying the server's integrity against potential threats.

– **Resolve Path and Send HTTP Response:** With security checks in place, the server translates the requested path into the corresponding file or directory on the server. This process is foundational for streamlined data retrieval and ensures accurate resource mapping.

  With the resolved path, the server assembles and transmits a HTTP response. This response encapsulates the requested data or an error code, reflecting the server's dynamic nature in responding to client queries.

In essence, this single-threaded server not only serves immediate operational needs but also sets the stage for the integration of advanced components in the evolution towards a more scalable, and high-performance multi-threaded web server.

5.2 Thread Pool Architecture

The next key component of the multi-threaded web server is the thread pool. The thread pool encompasses the core threading and concurrency operations of the web server and is one of the most vital components for the final web server. The main use of the thread pool is for producers in the web server to push tasks into a task pool, once tasks have been pushed into the task pool, consumers will then consume the tasks and execute them. Due to the multi-threaded nature of a thread pool and the focus on low latency and high performance, the thread pool must be efficient and scalable.

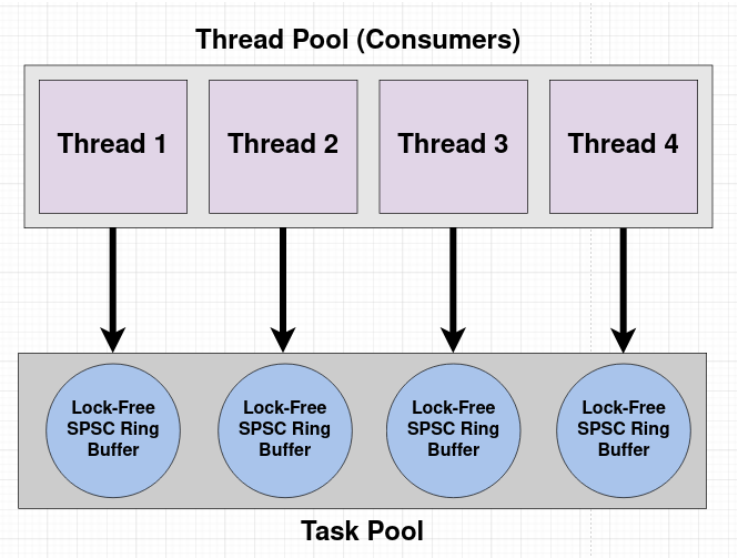

Figure 2: High Level Overview of Thread Pool

The diagram above shows a high level overview of the thread pool, this component is designed to manage concurrent tasks efficiently. It is structured around two primary elements: a ring buffer for task management and a fixed number of worker threads for task execution. Each worker thread has an associated ring buffer, enabling it to handle tasks independently. This design aims to optimise for low latency and high performance, which are critical in a web server environment. Some of the key features of the thread pool are:

- **Use of Atomic Operations:** Atomic operations are integral to the thread pool's architecture. They ensure thread safety and consistency across multiple threads without the overhead of traditional locking mechanisms. This choice significantly enhances the pool's efficiency, especially under high concurrency scenarios.

- **Dual Task Submission Methods:** The thread pool provides two methods for task submission. The first method distributes tasks using a round-robin task assignment approach, ensuring an even workload distribution across threads. The second method allows tasks to be directly assigned to a specific thread, offering flexibility in task management.

- **Lock-Free Ring Buffer for Task Queuing:** Inspired by the Linux Kernel's AF_XDP socket queue, the ring buffer for task queuing is implemented in a lock-free manner. This design choice minimises contention and maximises throughput, essential for handling high volumes of web server requests.

- **Cache Optimisation:** The thread pool and its underlying data structures such as the ring buffer, are aligned to cache line boundaries (64 bytes). This alignment reduces the likelihood of false sharing and cache line bouncing, leading to better CPU cache utilisation and overall performance.

- **Thread Pool Memory Pooling:** The thread pool and its ring buffer use memory pooling with compile-time determined sizes, optimising memory allocation and deallocation. This static approach streamlines memory management, particularly beneficial for environments where memory is reused throughout the program's runtime. Such a strategy enhances performance by reducing overhead associated with dynamic memory operations, making it ideal for high-throughput scenarios.

5.3 LRU Cache Implementation

The next core component of the multithreaded web server is the LRU (Least Recently Used) Cache. This component will be used to efficiently manage file access, significantly enhancing the server's performance. The LRU Cache is designed to swiftly retrieve and store files, utilising memory mapping to directly access file contents in memory. With a fixed capacity set at compile-time, it effectively balances memory usage while ensuring quick data access. The cache's LRU policy prioritises recently accessed items, keeping them readily available, and evicts the less frequently used items to manage its capacity. This approach is particularly beneficial in a high-load server environment where efficient and rapid access to files is paramount.

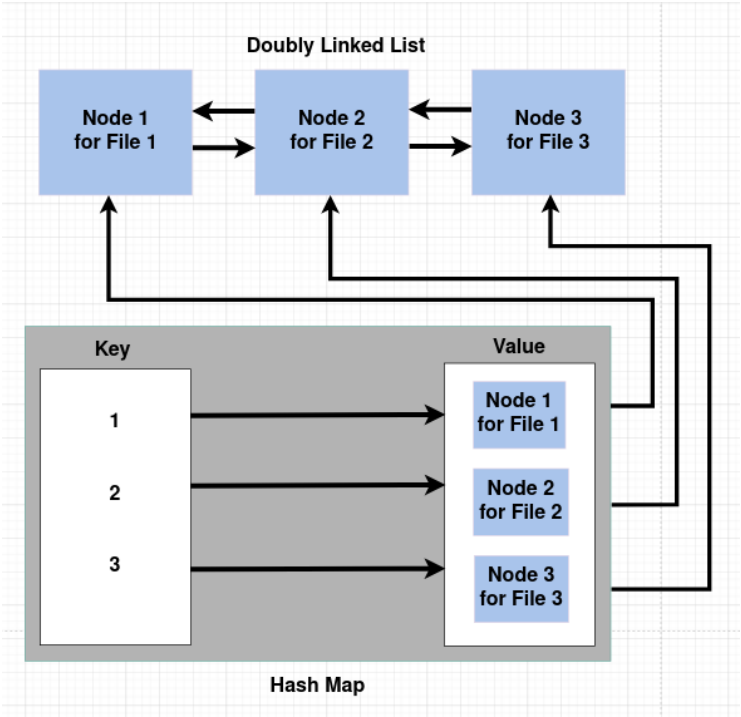

Figure 3: High Level Overview of LRU (Least Recently Used) Cache

– **File Memory Mapping:** The LRU cache uses memory mapping for file handling, enabling direct access to file contents from memory. This method proves to be highly efficient, as it circumvents the overheads associated with traditional I/O operations. These overheads typically include extensive system calls and data copying. By adopting memory mapping, the cache allows for faster data access, thereby enhancing overall system performance. The following diagram highlights the key memory mapping functionality behind the LRU Cache and how files on disk are mapped to virtual memory locations.

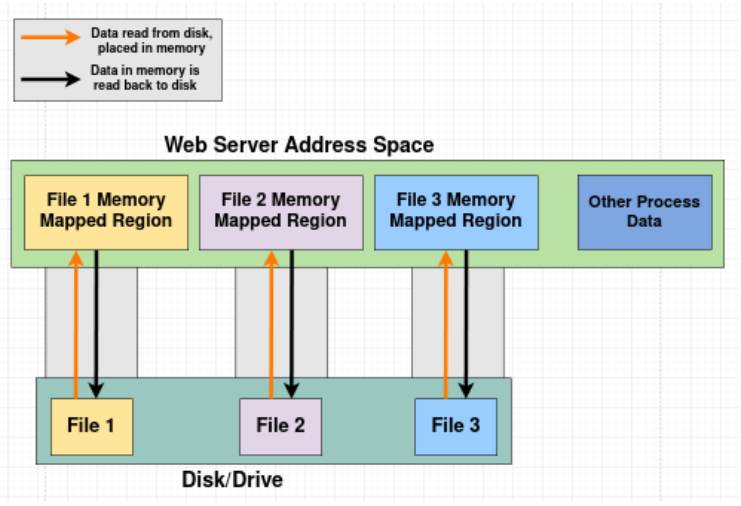

Figure 4: Overview of LRU Cache Memory Mapping Functionality

– **O(1) Complexity for Key Operations:** The LRU Cache achieves constant time complexity (O(1)) for critical operations like access, insertion, eviction, and addition. This is made possible through the combination of a doubly-linked list for maintaining the LRU order and a hash map for quick key-based access.

– **Thread Safety with Mutex Locks:** Thread safety is ensured by using mutex locks. This feature is crucial in a multi-threaded environment to prevent concurrent access issues such as race conditions between threads that are trying to access the same critical sections. By utilising a mutex lock the LRU Cache can preserve data integrity and prevent potential conflicts or inconsistencies in data access and manipulation.

– **Eviction Policy:** The LRU eviction policy is implemented to manage the cache size. When the cache reaches its capacity, the least recently used item is evicted to make room for new items. This strategy is effective in maintaining the cache size while ensuring that frequently accessed data remains readily available.

In conclusion, the LRU cache implementation is well-designed for efficient file memory management. The combination of file memory mapping, O(1) operations, thread safety, and a clear eviction policy makes it a valuable component for the multi-threaded web server.

6 Concurrency Model

6.1 Threading and Synchronisation Mechanisms

In the multi-threaded web server, the threading and synchronisation mechanisms are vital for ensuring efficient operation across multiple concurrent threads. Firstly, the multi-threaded web server leverages the C++ standard library's std::thread for efficient thread management [12]. This choice allows for a high degree of control and flexibility in thread creation and management, crucial for handling the workloads of a web server. The std::thread library will facilitate the creation of a thread pool at the server's startup, enabling quick and efficient handling of incoming requests without the overhead associated with dynamic thread creation and destruction.

In terms of synchronisation mechanisms, the server predominantly relies on atomics through the C++ standard library's std::atomic [10] and lock-free data structures. The use of these mechanisms provides significant performance benefits, particularly in terms of speed, as they eliminate the need for expensive locking and unlocking operations. This approach is beneficial in high-concurrency environments, where minimising the overhead from synchronisation primitives is essential for maintaining high throughput.

While mutex locks might be initially employed in certain critical sections of the server, there will be an effort to optimise them out wherever possible. Mutexes, while useful for ensuring exclusive access to shared resources, can introduce overhead due to context switching, as well as the costs associated with putting threads to sleep and subsequently waking them up. Additionally, mutexes can be memory-intensive. Therefore, transitioning towards atomic operations and lock-free structures not only reduces these overheads but also enhances the overall efficiency of the server.

6.2 Potential Issues

In building a robust multi-threaded web server, it's imperative to anticipate potential challenges related to threading, synchronisation, and concurrency. These aspects are foundational to the server's stability and efficiency but can also be sources of complex issues if not managed correctly.

6.2.1 Race Conditions

A challenge in any multi-threaded environment is the risk of race conditions, this is where multiple threads access and modify shared data concurrently, but if the synchronisation between the accesses is erroneous it could lead to unpredictable results and undefined behaviour [11]. To mitigate this, the web server will implement robust synchronisation mechanisms using std::mutex and std::atomic. These tools ensure that critical sections of the code are accessed sequentially rather than concurrently. The server development process will incorporate the use of TSan (ThreadSanitizer), a tool for detecting synchronisation issues in C++ applications. TSan will enable early identification and resolution of race conditions.

6.2.2 Deadlocks

Another potential issue is the problem of deadlocks. Deadlocks occur when multiple threads are waiting indefinitely for resources locked by each other, this can bring the server to a standstill. To prevent this, the server's design will prioritise careful consideration of lock acquisition and release order [5]. This approach is aimed at avoiding common deadlock patterns, such as circular wait conditions [6]. By proactively identifying and addressing potential deadlocks, we can ensure the server remains responsive and efficient under various operational conditions

6.2.3 Thread Contention and Overhead

Thread contention and overhead represent another issue that could impact the server's performance and efficiency. This problem often stems from excessive thread creation or the intensive use of synchronisation primitives, such as mutexes, which can lead to significant performance issues due to problems such as increased memory usage, increased context switching and the need to wakeup and block threads. To tackle this the server will focus on using lock-free data structures wherever feasible, coupled with a preference for atomic operations over mutex locks. This shift towards lock-free and atomic mechanisms is aimed at minimising synchronisation overhead and contention and maximising throughput, thus enhancing the overall efficiency of the server.

7 Server Core Functionalities

7.1 HTTP Protocol Handling:

The server's design for HTTP protocol handling is centered around efficiently processing GET requests, a fundamental feature for web servers. GET requests are critical for retrieving specified resources, such as web pages or images. The design includes comprehensive mechanisms for parsing these requests, ensuring accurate interpretation of URIs and headers. This parsing is vital for correctly identifying and serving requested resources. Additionally, the server might extend its capabilities to support HEAD requests. If implemented, this will allow clients to retrieve header information without the associated body, which can be beneficial for bandwidth optimisation in scenarios where only resource metadata is required. The design also contemplates robust error handling and response generation, ensuring that the server responds appropriately to various HTTP scenarios and adheres to protocol standards.

7.2 Connection Management

The design for connection management in the server is a critical component for enhancing user experience and optimising server performance. It involves maintaining persistent connections as prescribed by the HTTP/1.1 standard, which significantly reduces the overhead of repeatedly establishing connections for each request. This persistent connection approach is beneficial for clients that make multiple requests, as it allows for faster subsequent requests and responses. The server will also incorporate smart mechanisms for managing these connections, including efficient handling of idle connections and timeouts. This ensures that the server resources are used and are available for new incoming connections, thereby maintaining an optimal balance between resource utilisation and performance.

7.3 Non-Blocking Sockets

The integration of non-blocking sockets into the server's architecture is a decision aimed at enhancing the server's ability to handle multiple simultaneous connections efficiently. Non-blocking sockets allow the server to perform I/O operations, such as reading requests and sending responses, without pausing the execution of other essential tasks. This is crucial in a multi-client environment where the server needs to maintain responsiveness across numerous concurrent connections. The use of non-blocking sockets aids in preventing bottlenecks that could arise from waiting on I/O operations, thus contributing significantly to the server's scalability and overall throughput. This design choice is particularly effective in handling high traffic loads, ensuring that the server remains responsive and efficient under varying network conditions.

8 Bibliography

References

[1] Bonfanti, S., Gargantini, A., and Mashkoor, A. Design and validation of a c++ code generator from abstract state machines specifications. *Journal of Software: Evolution and Process 32*, 2 (2020), e2205. [Accessed: 25th Nov 2023] https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2205.

[2] De Melo, A. C. The new linux'perf'tools. In *Slides from Linux Kongress* (2010), vol. 18, pp. 1–42. [Accessed: 25th Nov 2023] http://vger.kernel.org/~acme/perf/lk2010-perf-acme.pdf.

[3] Fog, A. Optimizing software in c++, 2006. [Published: 2004-2023], [Accessed: 24th Nov 2023] https://www.agner.org/optimize/optimizing_cpp.pdf.

[4] Glek, T., and Hubicka, J. Optimizing real world applications with gcc link time optimization. [Accessed: 25th Nov 2023] https://www.researchgate.net/publication/47374866_Optimizing_real_world_applications_with_GCC_Link_Time_Optimization.

[5] Habermann, A. N. Prevention of system deadlocks. *Communications of the ACM 12*, 7 (1969), 373–ff. [Accessed: 27th Nov 2023] https://dl.acm.org/doi/pdf/10.1145/363156.363160.

[6] Jain, S., Kumar, N., and Chauhan, K. An overview on deadlock resolution techniques. *Volume 7* (2019), 1–3. [Accessed: 28th Nov 2023] https://d1wqtxts1xzle7.cloudfront.net/64185803/an-overview-on-deadlock-resolution-techniques-IJERTCONV7IS12016-libre.pdf?1597479777=&response-content-disposition=inline%3B+filename%3DIJERT_An_Overview_on_Deadlock_Resolution.pdf&Expires=1701544222&Signature=GxW96~psy0yBN2iDXXKxS~s9Fxf9NCCpXgGNQzPpLZzcvTsef3v5N0guhBBaOXW8X7tUHgvT9Hu3uo7_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.

[7] Jones, M. T. Optimization in gcc. *Linux journal 2005*, 131 (2005), 11. [Accessed: 25th Nov 2023] https://www.ecb.torontomu.ca/~courses/coe518/LinuxJournal/elj2005-131-optimizationGCC.pdf.

[8] Josuttis, N. M. The c++ standard library: a tutorial and reference. [Accessed: 25th Nov 2023] https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11176/111761P/The-new-C-serialization-library-supporting-backward-and-forward-compatibility/10.1117/12.2536387.short.

[9] Lattner, C. Llvm and clang: Next generation compiler technology. In *The BSD conference* (2008), vol. 5, p. 33. Accessed: 1st Oct 2023, https://reup.dmcs.pl/wiki/images/0/09/53_BSDCan2008ChrisLattnerBSDCompiler.pdf.

[10] Leis, V., Haubenschild, M., and Neumann, T. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull. 42*, 1 (2019), 73–84. [Accessed: 27th Nov 2023] https://web.archive.org/web/20220306194839id_/http://sites.computer.org/debull/A19mar/p73.pdf.

[11] Netzer, R. H., and Miller, B. P. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS) 1*, 1 (1992), 74–88. [Accessed: 27th Nov 2023] https://dl.acm.org/doi/abs/10.1145/130616.130623.

[12] Shoshany, B. A c++ 17 thread pool for high-performance scientific computing. *arXiv preprint arXiv:2105.00613* (2021). [Accessed: 27th Nov 2023] https://arxiv.org/abs/2105.00613.

[13] Von Hagen, W. *The definitive guide to GCC*. Apress, 2011. [Accessed: 24th Nov 2023] https://books.google.co.uk/books?id=wQ6r3UTivJgC&lpg=PR3&ots=EOSkSm5wFv&dq=gcc%20compiler&lr&pg=PR4#v=onepage&q=gcc%20compiler.

[14] Wu, P.-C., and Wang, F.-J. On efficiency and optimization of c++ programs, April 1996. [Accessed: 24th Nov 2023], https://www.researchgate.net/publication/220282377_On_Efficiency_and_Optimization_of_C_Programs.

[15] Zakaria, F. Applying c/c++ sanitizers on the nix toolchain. [Accessed: 26th Nov 2023] https://sorensenucsc.github.io/CSE211-fa2021/projects/nix_memory_sanitizer/CSE211_Final_Report.pdf.