Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Programming Strategies for Low-Latency Applications

Author:
Gordon Lee

Supervisor:
Dr. Paul A. Bilokon

Second Marker:
Prof. William J. Knottenbelt

June 20, 2022

Abstract

Low-latency programming is poorly documented partly due to its value within industry; high-frequency trading is one of many professional fields where firms are striving to achieve latency reductions in the magnitude of nanoseconds. There are fragments of information available about what programmers can do to make their programs faster, but the strategies that give trading firms the competitive edge in the market are closely guarded secrets.

The purpose of this project is to provide an educational resource for software engineers that are looking to design and write low-latency applications. This work presents two closely-related products that achieve this.

The first product is the Low-Latency Programming Repository, which contains the theories behind various programming strategies that can achieve higher performance within an application. These theories are backed up by empirical evidence in the form of reproducible microbenchmarks that demonstrate these techniques in action. The common goal of each technique is to reduce the execution time of programs, whether it be through the careful management of the memory cache, or the compiler transformations performed by hand.

The second product is the Java Modular Packet Processor, a multi-threaded application and library that processes packets using a network of components that perform individual operations. Packets are sent between components using the LMAX Disruptor, an esteemed data structure that serves as an alternative to queues. This product is used in conjunction with the Low-Latency Programming Repository to demonstrate a quantitative performance improvement of using lock-free alternatives to general-purpose data structures.

Acknowledgements

I would like to thank my supervisor, Dr. Paul Bilokon, for his continuous support throughout this piece of work. His expertise in trading and financial systems has been quintessential in the development of my project.

I am also really grateful for the support from my personal tutor, Chiraag Lala, who provided me with a lot of direction and assurance in the latter stages of my project.

For my parents who have done everything they can to support me throughout my life: thank you for giving me a clear direction in my education and career, and pushing me to achieve my long-term goals.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In a world where computers are now an expectation rather than a luxury, the demand for high-performance systems grows exponentially each year. Latency is one of the key metrics used to measure the performance of an application, and lower latencies are being demanded in all fields of computing: from trading firms where a delay of a few nanoseconds for an order can result in a loss in the magnitude of millions of pounds, to self-driving cars that need to make decisions within milliseconds to determine whether an accident can be narrowly avoided.

1.1 Motivation

Gordon Moore famously stated in 1965 that the number of transistors in a dense integrated chip doubles every two years. However, this is no longer applicable to the present day; in 2019, Nvidia CEO Jensen Huang claimed that Moore's Law is "dead", and the expense and difficulty to double the number of transistors driving processing power is increasing. This sentiment was supported by Charles Leiserson the next year, a leading computer scientist at MIT specialising in parallel programming. He stated that "It's over. This year that has become really clear." [1].

The reliance for creating faster software has gradually shifted from the performance of the hardware towards the abilities of the software engineer. In order to maximise the performance that modern hardware can offer, the engineer must understand how to design and tune systems in such a way that every instruction generated by the compiler is deliberate and essential.

1.2 Objectives

The purpose of this project is to centralise the fundamentals that a software engineer must be familiar with to begin writing and designing low-latency applications. This comes in the form of a repository of programming techniques, with their performance backed up by empirical evidence through reproducible microbenchmarks. The unified goal of each technique is to reduce the execution time of programs, whether it be through the careful management of the cache to prevent pipeline stalls, or the code transformations performed by hand that shift the responsibility of optimisation from the compiler to the programmer.

These are the contributions of this project:

1. **A repository of optimisations and design choices that contribute towards faster program execution times** - The Low-Latency Programming Repository provides an entry-level overview of different programming strategies that help to achieve performance with low-latency applications. To the best of our knowledge, there is no academic resource like this available within the public domain.
2. **A novel application demonstrating the performance impacts of using certain optimisations** - The Java Modular Packet Processor is an application that performs operations on packets and their headers. These operations will be interchangeable and processors can be

built to a user's specifications. There does not seem to be a Java-based application publicly available that can process packets with this level of flexibility.

3. **Reproducible benchmarks demonstrating the effectiveness of the strategies mentioned in both the repository and the application** - These benchmarks demonstrate how the theory behind each programming strategy can produce tangible results by using real-world applications. This will also give the strategies in the Low-Latency Programming Repository further credibility.

1.3 Report Structure

This report is structured as follows:

- **Chapter 2** contains technical overviews of relevant concepts within compilers, computer architecture, concurrency, and benchmarking which are mentioned throughout the report.
- **Chapter 3** is a broad review of relevant topics within the low-latency domain: programming languages, existing applications, and the industries that makes use of these.
- **Chapter 4** comprehensively details the strategies that were investigated and written to the Low-Latency Programming Repository under three categories: hand-written compiler optimisations, branch-free computing, and general application design.
- **Chapter 5** describes the high-level design and architecture of the Java Modular Packet Processor, its implementation, and its performance.
- **Chapter 6** evaluates the educational impact of the project and the quality of its contributions.
- **Chapter 7** summarises the project and discusses the ethical considerations that must be taken into account. The chapter also discusses future work that could take the project further.

Chapter 2

Preliminaries

This chapter presents an overview of key information related to computer architecture needed to understand the terminology used in the project and its corresponding background information. A brief overview of compilers and their role in translating programs is provided due to its significant role in producing high-performing applications, as well as the fundamentals of concurrency and synchronisation primitives. Finally, an introduction to benchmarking methods in C++ and Java are presented, with a summary of how benchmarks are used within this project.

2.1 CPU Pipelining

Pipelining is a technique used by CPUs where multiple instructions are overlapped in execution, taking advantage of the parallelism that exists during the execution of an instruction. Pipelining is widespread in modern computing, and can be found in the processors that cost only a few pounds to the processors that are used for even the most demanding workloads within industry.

Each step (formally known as a *pipe stage* or *pipe segment*) of the pipeline is responsible for one operation in the processing of an instruction. The steps operate in parallel with each other and are arranged sequentially. Instructions enter at the front, progress through the pipeline, and exit at the back.

2.1.1 Pipeline Example: Five-Stage RISC Pipeline

RISC V is a load-store architecture that is useful for illustrating basic concepts. An instruction set (ISA) dictates the machine code that a computer can understand.

The RISC instruction set can be implemented with a pipeline that uses the following stages:

1. IF: Instruction fetch
2. ID: Instruction decode
3. EX: Execution
4. MEM: Memory access
5. WB: Write-back

Starting a new instruction on every clock cycle means that each instruction takes 5 clock cycles to complete, but with each clock cycle another instruction can complete its processing in the pipeline. Table 2.1 demonstrates the instructions carried out over a number of clock cycles. For each cycle, a new instruction is processed by the first step of the pipeline. Consequentially, the instructions "shift" across the pipeline over time. This simple example does not represent the issues of pipelining and represents a best-case scenario of a CPU pipeline. The next section goes into detail about the issues that can occur.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

Table 2.1: A simple RISC pipeline processing instructions over 9 clock cycles.

2.1.2 Pipeline Hazards

Pipelining does have flaws, with the most significant scenario occurring when instructions from the instruction stream are prevented from executing during its designated clock cycle. This situation is known as a *hazard*, which reduces the ideal performance of a pipeline. There are three categories of hazards:

1. **Control hazards**: caused by the pipelining of branches and other instructions that are data-dependent, resulting in the following set of instructions being unknown.
2. **Data hazards**: caused by operands (data) not being available on time, perhaps due to a complex operation.
3. **Structural hazards**: caused by a lack of execution resources, which may occur due to hardware not supporting combinations of instructions simultaneously.

These hazards can *stall* the pipeline, creating a *pipeline bubble*; a gap in the pipeline where a step is not processing an instruction. Figure 2.1 demonstrates a five-stage pipeline experiencing an ALU stall. The *Mem* stage is missing an instruction, possibly due to a lack of compute resources available to evaluate a complex operation.

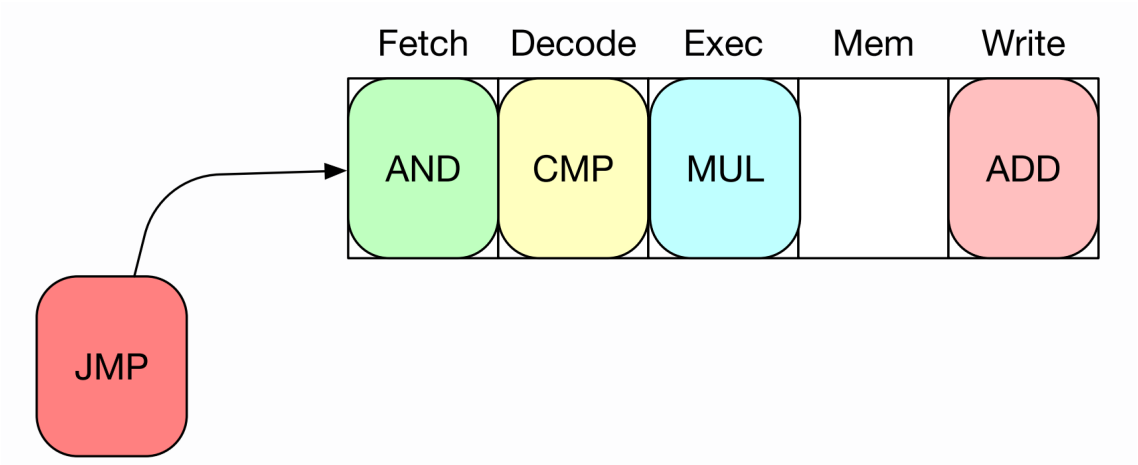


Figure 2.1: An ALU stall, where no instruction is being processed in the *Mem* stage.

2.2 Compilers

A computer program must be translated into a form in which it can be executed by a computer. Software systems that carry out these translations are known as *compilers*. The field of compiler development brings together several disciplines within computer science: programming languages, computer architecture, language theory, algorithms, and software engineering.

This section briefly outlines the responsibilities of *optimising compilers*, compilers that not only perform the functional requirements of translating programs into low-level code, but searches for potential optimisations that can increase the performance of the application [2].

2.2.1 Compiler Optimisations

Using constructs from high-level languages can introduce substantial runtime overhead if a compiler naively translates each independent component into assembly/machine code. Eliminating unnecessary instructions, or replacing a sequence of instructions with a better performing sequence of instructions, is known as *code optimisation*.

Most optimisations are based on *data-flow analyses*, which are algorithms that gather information about a program. The analyses determine the properties that each instruction holds within the program, which must hold every time the instructions are executed. Different properties are identified depending on the type of analysis carried out.

A common example is the *constant-propagation analysis*, which identifies variables that have a unique constant value. An optimising compiler will often replace the variable references with constant values to reduce the overhead of obtaining the value of the variable, when its value will be known regardless of the flow of the program.

A compiler optimisation must preserve the semantics of the original program. A compiler only knows how to apply low-level semantic transformations using:

- **Algebraic identities:** e.g. i + 0 = i
- **Program semantics:** e.g. performing the same operation on the same value has the same result

It is important to recognise that an optimisation does not necessarily produce an "optimal" output. There are cases when an optimisation may degrade performance. The effectiveness of an optimisation depends on the definition of "performance", since desirable attributes of an application can vary based on its purpose. For example, some programs may be optimised for speed (i.e. execution time), such as those that are written for low-latency systems. Other programs may be optimised for object code size, perhaps in embedded systems where space is limited. [3]

2.2.2 Compiler Intrinsics

Compilers may have their own set of functions within a higher-level programming language that direct the compiler to perform specific operations on a program's code, avoiding the need to manually carry out transformations in a lower-level language. Many of these operations will be optimisations which may not be performed automatically by the compiler.

These are known as *intrinsics*, and the use of these can affect the portability of code. For example, a C++ program containing GCC/G++-specific intrinsics may not compile if the program is compiled on a different system using the Clang compiler. Documentation of computer applications will usually dictate the compiler that should be used, and in some cases its version as well [4].

2.3 Concurrency

Until recently, advancements in computer architecture resulted in increases of clock speeds, allowing software to naturally improve in performance over time. Now, clock speeds can no longer be increased without overheating, so the "free ride" in achieving performance is effectively over.

Manufacturers are now turning to multicore architectures, with multiple processors communicating with each other through shared caches. These processors exploit parallelism, where multiple processors are used to work on a single task. Modern advancements in hardware usually relate to increased parallelism and not increased clock speed. Exploiting parallelism is now one of the main challenges in the field of computer science [5].

2.3.1 Race Conditions

A *race condition* occurs when two threads access a shared resource (eg. variable, object, memory location) simultaneously. This means that the two operations performed by the threads may yield different results depending on the order in which the operations were performed.

The following scenario illustrates this:

- Thread A and Thread B want to read and modify a shared variable, X.
- Both threads reads the value from the variable.
- Both threads perform their operations on the value.
- The threads now "race", with the variable containing the value of the thread that writes to it last.

The problem with this scenario is that the result is non-deterministic; the variable can either hold the value of Thread A or Thread B, depending on which thread wrote to the variable last. This can be the root cause for undefined behaviour, i.e. the function with this race condition cannot guarantee a consistent result [6].

2.3.2 Mutual Exclusion

Mutual exclusion is one of the most widespread (and important) forms of coordination in concurrency, which aims to prevent race conditions and undefined behaviour. Mutual exclusion is built on the principle that a thread will never enter a *critical section* whilst another thread is accessing that section. The critical section will usually refer to an interval within a program where a shared resource is being accessed.

Mutual exclusion can be achieved through synchronisation primitives, which are devices that vary between languages but have the common goal of achieving concurrency without race conditions and satisfies the requirement of mutual exclusion.

2.3.3 Synchronisation Primitives

Synchronisation primitives exist as the basic facilities of a language for achieving concurrency.

For example, the following classes are part of the concurrency support library in C++ [7]:

- **std::thread**: represents a single thread of execution.
- **std::atomic**: a template for creating objects with store/load functions which will set/get items atomically.
- **std::mutex**: a primitive that achieves mutual exclusion by having an "owner" thread of the mutex, blocking threads threads that try to claim ownership of the mutex whilst it is already owned.
- **std::condition_variable**: a more complex primitive that can block multiple threads at the same time until another thread modifies a shared variable and notifies the condition variable.

The exact synchronisation primitives available will depend on the language being used.

2.4 Benchmarking

Benchmarking is used extensively within this project to demonstrate the effectiveness of implementing particular programming techniques through quantitative data.

Meaningful performance results come from conducting fair experiments, which is a challenge because of the noise present in computer systems. Microbenchmarking tools are designed to isolate the section of code that is of interest, ensuring that the overhead of setting up and tearing down the application is minimised. This can be abstracted into a three-step process:

1. Set the system up into a predefined state.
2. Perform operations while measuring the desired performance metrics.
3. Clean up the benchmark by gracefully closing/detaching any resources used.

A single datapoint is not enough to draw reasonable conclusions about the program being measured. Benchmarking tools will usually run several iterations of the test, aggregating them into a single value (possibly also with some indicator of the variance).

Listing 2.1 Benchmarking skeleton using Google Benchmark

```
#include <benchmark/benchmark.h>

static void BM_SomeFunction(benchmark::State& state) {
  // Perform setup here
  for (auto _ : state) {
    // This code gets timed
    SomeFunction();
  }
}
// Register the function as a benchmark
BENCHMARK(BM_SomeFunction);
// Run the benchmark
BENCHMARK_MAIN();
```

Listing 2.2 Explicit arguments for Google Benchmark

```
BENCHMARK(BM_SomeFunction)->Arg(8)->Arg(1<<10)->Arg(8<<10);
```

This section outlines two popular benchmarking tools used throughout the project, and how benchmarking is present throughout this project.

2.4.1 Google Benchmark

Google Benchmark is an open-source library used to benchmark code snippets written in C++. Listing 2.1 outlines the basic skeleton of running a benchmark. The benchmarking function takes in a `state`, which is used within a for loop that times the code of interest. The state contains the relevant objects of the benchmark, including any additional parameters that the programmer may want to change on each iteration.

The benchmarking suite has the flexibility to generate benchmarks with a varying set of arguments. These are declared during the registration of the benchmark, with an example shown in Listing 2.2 with manually defined values. The `BENCHMARK` macro takes in the benchmarking function, and passes arguments of 8, 1«10, and 8«10. These values can be generated with `Ranges` and other utility functions.

Additionally, configurations of multiple different parameters can be generated if the user wishes to have multi-dimensional tests. For example, a simple function can have two arguments: the `size` of a given array and the `stride` that the array is accessed with. Google Benchmark can generate a set of tests covering every permutation of these two variables using `ArgsProduct`, as shown in Listing 2.3.

These arguments can then be accessed within the benchmark, allowing users to easily identify trends as the parameters of the benchmark change. Google Benchmark has a dense set of functionalities, but this report only contains the surface-level features of the tool. More information about the project can be found in its GitHub documentation [20].

Listing 2.3 Generated arguments for Google Benchmark

```
BENCHMARK(BM_SomeFunction)
    ->ArgsProduct({
      {8, 16, 32, 64, 128}, // Array Size
      {1, 2, 4}        // Stride
    });
```

Listing 2.4 Benchmarking code using the Java Microbenchmark Harness

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Mode;

public class MyBenchmark {

    @Benchmark @BenchmarkMode(Mode.Throughput)
    public void testMethod() {
        // Benchmarked code
    }

}
```

2.4.2 Java Microbenchmark Harness

The Java Microbenchmark Harness (JMH) is another benchmarking library that can be used to build, run, and analyse both micro and macro benchmarks written not only in Java but other languages targeting the Java Virtual Machine.

Scientifically benchmarking applications using JVM-based languages is a challenge because of the numerous optimisations that are implicitly performed by the JVM and underlying hardware. Poor implementations of benchmarks may give false-positive results when testing isolated sections of code due to optimisations that would not normally be applied in production. JMH prevents the optimisations that the JVM and underlying hardware perform to help the user obtain results that accurately reflect real-world performance.

Unlike C++, Java has *annotations*, a mechanism that provides metadata to code. JMH uses annotations to identify the functions that should be benchmarked along with corresponding parameters. Listing 2.4 shows a simple class that is set up for benchmarking with JMH. The benchmark will measure "throughput", i.e. the number of operations per second, or the number of times the benchmark method could be executed. The `@BenchmarkMode` dictates this, and the `@Benchmark` annotation indicates that this method contains the code to be measured.

2.4.3 Benchmarking in this Project

Chapter 4 contains a variety of programming strategies, many of which stem from compiler optimisations and small code transformations. These are demonstrated in C++, which is why Google Benchmark is used for measuring the performance gained from these transformations. Each set of benchmarks attempts to use parameters which may uncover interesting characteristics among the optimisations implemented.

The application/library outlined in Chapter 5 uses the Disruptor as its processing framework, which is written in Java. For this reason, JMH is used to measure the performance of the application when using different data structures for its producer/consumer pattern.

Chapter 3

Background

This chapter provides a broad review of relevant subjects within low-latency programming. This includes the programming languages that are often used within the field, and a couple of example industries where low-latency applications are critical for success. An insight into the Disruptor library is also given, which is a crucial component of the contributions of this project.

3.1 Programming Languages

Certain languages are better suited for low-latency applications than others. Low-latency programming is a domain where improving speeds by microseconds is immensely valuable, mainstream interpreted languages such as Python or PHP struggle to hold their ground against compiled languages such as C++ or Java because of the overhead that comes at runtime. A language with a strong memory model is also beneficial to allow lock-free programming. Generally speaking, low-latency programming prefers compile-time overhead over runtime overhead.

3.1.1 C++

C++ is an extension of the C programming language and is often described as "C with object orientation capabilities". C++ has often been described as "closer to the metal" than most languages, as it does not have a lot of abstraction and requires a strong understanding of how the machine works. Pointers, memory addresses, and the stack/heap are all examples of concepts that must be understood in order to utilise C++ to its full potential.

Most individuals consider C++ as an example of a high performance language. This is likely due to its fine-grained memory management, which leads to predictable performance and opportunities for tuning and optimisations. The language's smaller memory footprint contributes to better cache performance.

C++ is often used in conjunction with its Standard Template Library (STL), which provides common data structures and functions to the programmer [8]. The STL is not optimised for low-latency, which is why specialised data structures are used in domains where performance is critical.

3.1.2 Java

Java is another high-level, object-oriented programming language that is popular within industry. Unlike C++, Java is compiled to bytecode and run on the *Java Virtual Machine* (JVM), regardless of the underlying architecture. Unfortunately, the use of this language in low-latency systems is secretive and poorly documented. However, low-latency applications written in Java do exist, with one of the most prominent examples being the Disruptor, developed by the LMAX Group.

A major issue with Java is the presence of nonlinear latency spikes caused by its garbage collection [9], which is detrimental for applications where consistency and determinism is highly valued

(particularly in the HFT space). The Disruptor accommodates for this by preallocating memory to avoid the collection of its objects, proclaiming this as *mechanical sympathy* for the JVM.

An interesting perspective from Chandler Carruth suggests that Java is faster than C++ in very specific scenarios. If the virtual machine is well-tuned, and the garbage collector is tuned to precisely match the garbage collection interrupt frequency of the application, Java will often beat the performance of a C++-based application [10]. However, the primary setback is to have an entire system optimised to accommodate these requirements.

3.1.3 Rust

Rust was introduced in 2010 as a safer alternative to C++ in the domain of memory management, ownership, and concurrency.

The two common approaches to memory management have already been mentioned; memory is allocated manually in C/C++, and garbage collectors are used in Java. Rust has a mechanism that sits somewhat between these, with an *ownership model* that has the convenience of a garbage collector whilst preserving the speed and efficiency of manual management. The ownership system follows a set of rules checked by the compiler, without slowing down the program whilst it is running.

The ownership model has an additional advantage of preventing data races, turning many concurrency errors into compile-time errors. This is a key component of Rust, and has been dubbed as *fearless concurrency* [11]. *Mutable references* prevent read-write contention and provides a safer environment for developing concurrent programs.

Whilst this makes software development safer than in traditional languages like C/C++, it does not equate to better performance. The speed of programs written in these languages depends on the optimisations performed, which are mostly dependent on the compilers used. However, Rust has demonstrated itself as a worthwhile competitor to the fast, mainstream languages over the past decade.

3.1.4 Additional Languages

It is important to recognise that systems are not always written in one language. C++ may be used for the latency-critical components of an application, whilst the framework may be using something entirely different.

However, low-latency programming is not restricted to just these languages. The primary discussion has revolved around object-oriented languages but there are alternatives used in industry, especially with the increased popularity in functional programming languages.

A good example is Jane Street, a global quantitative trading firm that adopted OCaml as their primary development platform. OCaml is a statically typed functional language that was initially chosen for its expressiveness, helping traders to verify that high-performance code reflected their intentions. Though this does not directly indicate that OCaml is ideal for low-latency programming, it demonstrates that it is capable of delivering applications that meet the performance requirements of high frequency trading [12].

Scala is a strong, statically-typed general-purpose programming languages that combines object-oriented programming and functional programming into a concise alternative to Java [13]. Scala also runs on the JVM (therefore requiring the same considerations as Java does when building low-latency applications). Though there are not many examples of low-latency applications written in Scala (as it is rare for commercial projects to be published online), it exists as a strong contender for Java-based applications.

3.2 Low-Latency within Professional Fields

3.2.1 High-Frequency Trading

High frequency trading is one of the most well-known applications of low-latency programming. *Colocation* refers to a dedicated space within a data center belonging to stock exchanges, where

systems owned by high frequency traders and firms can be placed in close proximity to the stock exchange servers. This gives the traders' systems an advantage in the order of nanoseconds, which the traders are often willing to pay millions for.

There are numerous regulations put into place to ensure that colocation is fair and that its participants are not at a disadvantage. Trading venues must not discriminate between users; all subscribers must have access to the same services and platforms whilst operating under the same conditions. This includes:

- Space
- Power
- Cooling
- Market Connectivity
- Technical Support

The venues must also monitor each connection and their latencies within the colocation services that are offered. All of this must be conducted with complete transparency, with the details of their arrangements published and available [14].

It is also important to acknowledge the existence of specialised hardware built for ultra low-latency applications. Field Programmable Gate Array (FPGA) chips have specific characteristics that allow them to execute a subset of trading algorithms up to 1000 times faster than on traditional computers [15]. This report will be focused primarily on the software/programming techniques used to achieve low-latency, so FPGAs and similar hardware solutions will not be discussed.

3.2.2 5G Networking

The goal of 5G networking is to enable a fully mobile and interconnected society. 4G wireless technology has struggled to meet its performance goals, particularly with poor coverage, interconnectivity, and quality of service. 5G technology is built on modified 4G, promising higher throughput and lower latencies [16]. The goals of 5G (as of 2015) can be summarised as follows:

- 1000x increase in capacity
- Up to 10Gb/s speeds
- Below 1ms latency

The significant reduction in latency will help the development of new technology in several area, including Internet of Things, virtual reality, and vehicle to everything.

The networks and communications sector has traditionally been the domain of hardware developers, with hardware-based innovations leading to the development of the next generation. Now, 5G is powered by advancements in cloud technology and edge computing, with infrastructure built on cloud-native networks [17].

3.3 Existing Applications: The Disruptor

Many financial applications depend on queues to exchange data between processing stages. Using multiple queues in an end-to-end operation will add hundreds of microseconds to the overall latency, indicating that there is a huge opportunity for optimisation with this data structure.

The Disruptor is an open-source, high performance data structure written in Java and developed by the LMAX Group. It aims to reduce latency and high levels of jitter that are discovered when using bounded queues by offering an alternative for exchanging data between concurrent threads. From their own benchmarks, LMAX claims that the mean latency when using the Disruptor for a three-stage pipeline is 3 orders of magnitude lower than an equivalent queue-based approach, and handles approximately 8 times more throughput.

LMAX have built an order matching engine, real-time risk management, and a highly available in-memory transaction processing system using this design pattern. However, it does not have a use

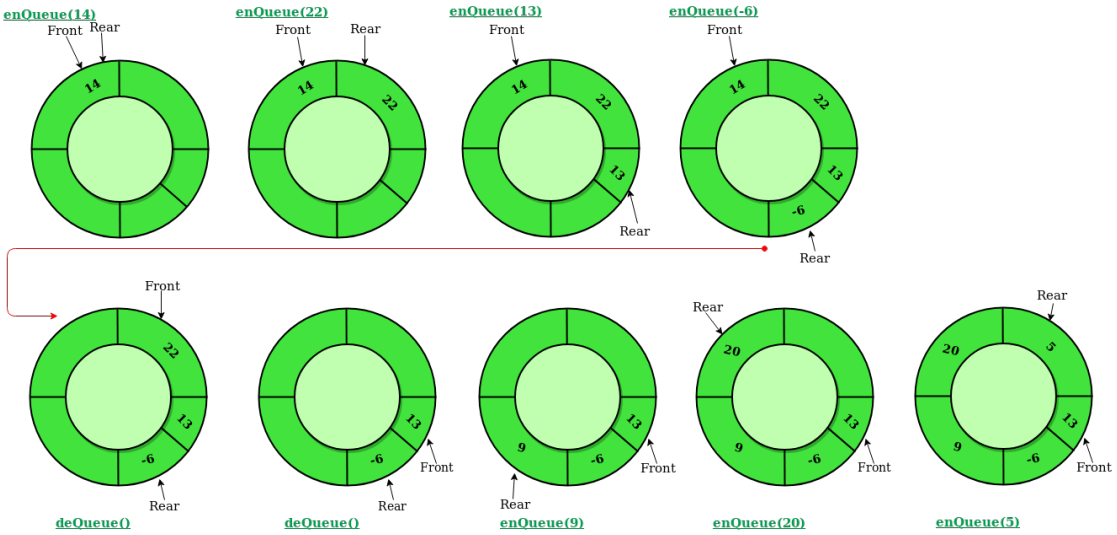


Figure 3.1: Diagram of a simple Ring Buffer [18]

only within the finance industry; it is a general-purpose pattern that provides a high performance solution to a complex concurrency problem.

3.3.1 Data Structures

A bounded ring buffer is the data structure that sits at the core of the Disruptor, storing an array of pointers to entries, or alternatively an array of structures representing the entries. The entries are usually containers storing the data, rather than the data itself.

All memory required for this data structure is pre-allocated on start up to prevent entries from being cleaned up by garbage collection. Instead, they will live for the lifetime of the Disruptor. There is a critical issue that can arise with garbage collection in low-latency systems:

- Increasing the amount of memory allocated introduces a greater burden on the garbage collector. Garbage collectors perform optimally when objects have a short lifetime, or live throughout the entire program.

- Heavy loads in a queue-based system will lead to a reduced rate of processing, increasing the lifetime of the objects waiting to be processed, potentially causing them to be promoted to an older generation when using generational garbage collectors. Copying objects between generations causes latency jitter, and they will have to be collected from the older generations which will be costly operations. This can result in pauses lasting seconds/GB when using large memory heaps, whenever fragmented memory space needs compaction.

Sequencing is used to manage concurrency in the Disruptor between the producers/consumers and the ring buffers that they interact with. Producers claim the next slot in sequence, corresponding to claiming an entry in the ring buffer. A simple counter can be used if only one producer exists, or an atomic counter which is incremented using compare-and-swap operations for multiple producers. A claimed value's corresponding entry in the ring buffer can then be written to by the producer. The update is then committed by updating another counter representing the cursor that indicates where the latest entry is available on the ring buffer.

Consumers wait for a sequence to become available by reading the cursor using memory barriers. The memory barrier ensures that changes made to the ring buffer are visible to the waiting consumers. Consumers also contain their own sequence, which is updated when they process entries. These allow producers to track the state of consumers, preventing the ring from wrapping (overwriting existing entries).

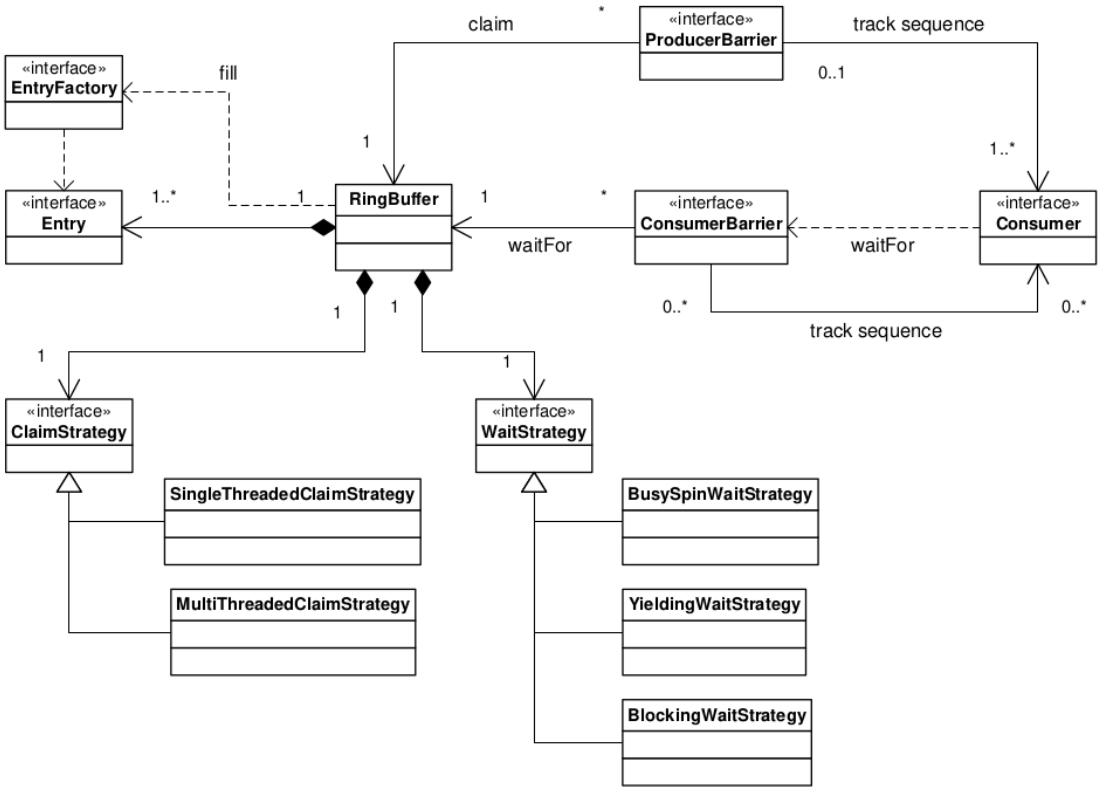


Figure 3.2: Class Diagram of the Disruptor Framework.

3.3.2 Design

The section outlines the relationships in the Disruptor framework. The *RingBuffer* stores *Entries*, which are usually implemented as wrappers around the objects that are of interest. *Producers* claim entries using the *ProducerBarrier*, write changes into the entry, then commit the entry back through the ProducerBarrier. A *Consumer* just needs to provide an object that receives callbacks when a new entry is available, resulting in an event based design pattern.

The concurrency concerns found in concurrent queues are separated for the producers and consumers when they interact with the RingBuffer. The ProducerBarrier manages the problem of concurrently claiming slots in the ring buffer, simultaneously tracking the dependants (consumers) to prevent the ring from wrapping. Consumers are alerted about new entries by the ConsumerBarrier, and they can then be constructed into a processing pipeline [19].

Chapter 4

Low-Latency Programming Repository

This chapter presents the Low-Latency Programming Repository, a collection of programming optimisations and techniques made available in a public repository to be used as an educational resource for software engineers. The repository is available on GitHub at:

https://github.com/gordonl0811/LowLatencyProgramming

This chapter details the research conducted on existing strategies that improve performance by reducing the execution time of programs. A key contribution of this project was not only documenting these findings, but supplying reproducible microbenchmarks that demonstrate the effectiveness of each method. The repository contains instructions for reproducing the benchmarks reported in its documentation, and full results of the benchmarks in this chapter can be found in the Appendix.

The different strategies used to improve the performance of low-latency applications have been aggregated into three categories:

- **Hand-Written Compiler Optimisations:** optimisations that can be performed manually by the programmer when the compiler fails to do so.
- **Branch-Free Programming:** a programming technique that eliminates the use of branches to minimise control hazards.
- **Application Design:** a broader category that covers a variety of program design choices.

4.1 Hand-Written Compiler Optimisations

For most fields that involve programming, software developers and software engineers can rely on the compiler to perform optimisations on their code to achieve the functional requirements of the application whilst reducing common sources of performance degradation. These are often introduced as a result of the abstraction offered by the programming language being used.

However, the technical knowledge required by programmers increase ten-fold when writing applications where every nanosecond counts. Compilers cannot be trusted to perform every ideal optimisation, especially for more complex segments of code where the opportunity for optimisation is non-trivial. Sometimes the programmer must perform these optimisations by hand, so it is important for them to understand not only what the compiler does automatically, but how to implement it manually themselves.

4.1.1 Inlining

Inlining is a simple compiler optimisation that replaces a function call with the body of the function in the compiled assembly code. This is a simple space-time tradeoff which may increase the speed of the executing program at the cost of larger binaries produced. Compilers will analyse the costs

Listing 4.1 Code for Cube() and Main()

```
1 static int Cube(int x) {
2     return x * x * x;
3 };
4
5 static int Main() {
6     int num = 8;
7     int result = Cube(num);
8     return result;
9 }
```

and benefits that occur from inlining a function, given that it meets the requirements for being inlined.

Motivation

Functions are used to organise code and break down programs into individual components. This makes code reusable via function calls, and function calls are usually made multiple times within a program. However, these calls can be expensive. This is a brief overview of the operations that are executed:

1. The stack frame is pushed onto the stack
2. The subroutine/functions instructions are executed
3. The stack frame is popped from the stack
4. The function returns to its return address

The call for the destination function can also cause the instruction pipeline to be flushed, causing a pipeline stall. Repeatedly calling small functions means that a lot of overhead is generated for little work done. This could be avoided by copying the code into the function call site, but this contradicts the software engineering philosophy of creating modular, well-organised code.

Description

Function inlining aims to eliminate the overhead of subroutine/function calls by embedding the body of the functions in the place of the call at compile time, i.e. when translating source code into assembly.

Consider the `Cube()` function in Listing 4.1, which takes an integer and multiples it by itself to return its cubed value. A simple `Main()` function is also present, returning the result of passing a value of 8 through the `Cube()` function.

This C++ code produces the assembly in Listing 4.2 when compiled with `x86-64 gcc 11.2` without optimisations (and therefore without inlining). The `Main()` function sets up its parameters and calls the `Cube()` function on line 17. The `Cube()` function multiples the value in the `eax` register by itself and returns the value.

No level of optimisation was used in this example because GCC (and other modern optimising compilers) is able to recognise that the function can be inlined. In C++, the programmer can add the `inline` keyword to the function definition to hint to the compiler that the function should be inlined.

GCC uses the `always_inline` attribute to force a function to be inlined (as long as it does not breach any requirements). Other compilers will have their own set of compiler intrinsics. Listing 4.3 shows the `Cube()` function as before but with the `always_inline` attribute included.

When compiled again with the `Main()` function, the generated assembly looks rather different (as shown in Listing 4.4). The key difference is that the assembly instructions of `Cube()` are now embedded within the `Main()` function, replacing the `call` instruction present in the original

Listing 4.2 Assembly for Cube() and Main()

```
Cube(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        imul    eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
Main():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 8
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    Cube(int)
        mov     DWORD PTR [rbp-8], eax
        mov     eax, DWORD PTR [rbp-8]
        leave
        ret
```

Listing 4.3 Cube() inlined with GCC

```
static int __attribute__ ((always_inline)) Cube(int x) {
    return x * x * x;
};
```

compiled assembly. The cubing instructions are still present, and the size of the function has only increased by one instruction.

There are tradeoffs that need to be considered when inlining functions. The absence of the call stack makes debugging more challenging. This optimisation also results in *code bloat*, where the program becomes longer than it would be without the inlined function. The more locations where a function is inlined will cause the binaries of the compiled program to grow. In the worst case scenario, this causes *thrashing*, where sections of code will constantly be swapped between higher and lower levels memory.

The size of the compiled binary files are not much of a concern when optimising for low-latency and speed. However, there are certain cases where a function cannot be inlined. For example, in C++ a compiler cannot inline a function if it contains one of the following properties:

- The function is recursive
- It has a variable length argument list
- It is a virtual function that is called virtually (direct calls can be inlined)

These are a few of the common properties that prevent inlining. Comprehensive documentation of the more subtle restrictions can be found online.

Benchmarks

The `Cube` function was tested against the inlined version, scaling the number of times the function was performed in a for loop by 10 each time. The graph in Figure 4.1 shows a noticeable improvement using the inlined version of the function, with approximately 15% increased performance throughout each test.

Listing 4.4 Assembly for Main() with Cube() inlined

```
1  Main():
2          push   rbp
3          mov    rbp, rsp
4          mov    DWORD PTR [rbp-4], 8
5          mov    eax, DWORD PTR [rbp-4]
6          mov    DWORD PTR [rbp-12], eax
7          mov    eax, DWORD PTR [rbp-12]
8          imul   eax, eax
9          imul   eax, DWORD PTR [rbp-12]
10         mov    DWORD PTR [rbp-8], eax
11         mov    eax, DWORD PTR [rbp-8]
12         pop    rbp
13         ret
```

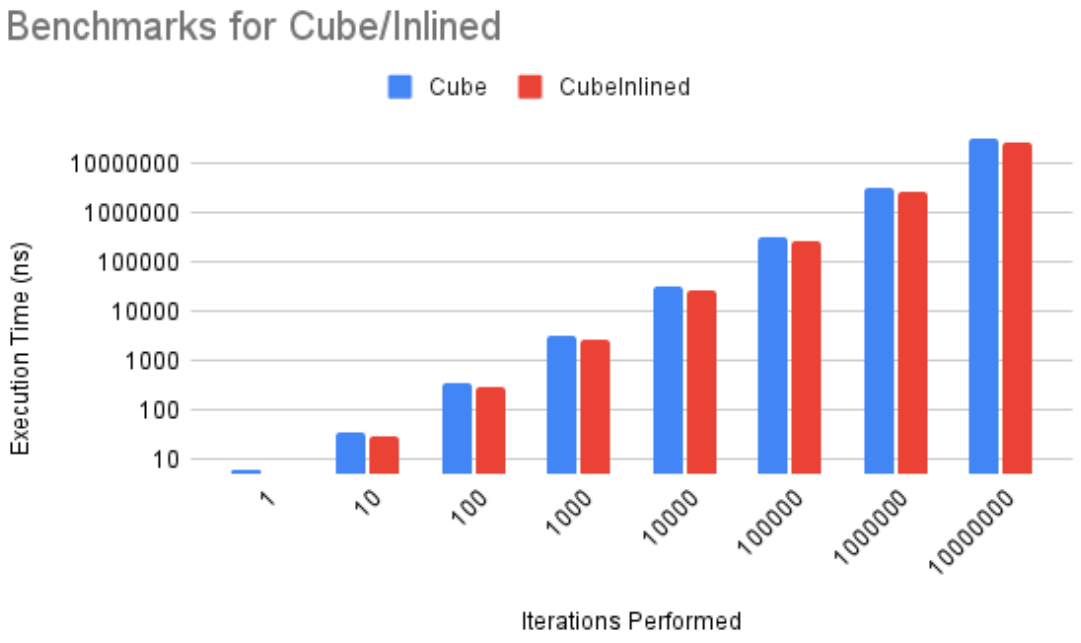


Figure 4.1: Execution times running the `Cube()` functions with an increasing number of repeated function calls.

Listing 4.5 Code for SumVectors()

```
static vector<int> SumVectors(const vector<int>& x,
    const vector<int>& y, int size) {

    vector<int> z(size);

    for (auto i = 0; i < size; i++) {
        z[i] = x[i] + y[i];
    }

    return z;

}
```

4.1.2 Loop Unrolling

Loop unrolling is a loop transformation technique that minimises the overhead induced by loops (e.g. branches) by repeating independent statements sequentially within the loop. The reduced number of iterations increases the potential for improved performance, and may result in better utilisation of the instruction pipeline.

Motivation

The execution of loops can bottleneck a section of an application, with the loops having substnatial overhead associated with them because of the branch penalty incurred on each iteration (also known as *loop overhead*).

Listing 4.5 contains a function that takes two `int` vector arguments, `x` and `y`, of (presumably) equal `size`. It adds the elements together per-index and stores the result in a third array, `z`, which is returned.

After each summation, the index of the loop `i` is compared to the `size` of the vectors, resulting in a branch and incurring loop overhead. In this particular example, the code branches `size` times. This generates a significant amount of overhead which may bottleneck an application using the function.

Description

Manual loop unrolling is performed by refactoring a loop's iterations into a sequence of instructions. A sequence of statements can be repeated `n` times, with `n` being known as the *loop unrolling factor*.

The example used earlier can be rewritten as shown in Listing 4.6, with a loop unrolling factor of 2. The structure of the loop has been changed so that `i` is incremented by 2 instead of 1, and two summations are performed on each iteration instead. Therefore, only `size`/2 branches are incurred, reducing the overall loop overhead.

This example is simple because of the assumed precondition that `size` is divisible by 2. Without this guarantee, another check may need to be performed (i.e. `size` is odd) to perform a final iteration of the loop after the main loop shown above.

As with many other optimisations, loop unrolling is a space-time tradeoff, but excessively using the technique may actually degrade performance. Listing A.1 contains the generated assembly of the body of the for-loop in `SumVectors()`, and Listing A.2 shows the corresponding unrolled function's assembly (these have been placed in the appendix due to their large size). The size of the block of instructions almost doubles in size. Naturally, larger loop unrolling factors can be used to reduce the number of branches, but the larger blocks of code may cause thrashing if the program code does not fit into the instruction cache.

Listing 4.6 Code for SumVectors() unrolled

```
static vector<int> SumVectorsUnrolled(const vector<int>& x,
    const vector<int>& y, int size) {

    vector<int> z(size);

    for (auto i = 0; i < size / 2; i += 2) {
        z[i] = x[i] + y[i];
        z[i+1] = x[i+1] + y[i+1];
    }

    return z;

};
```

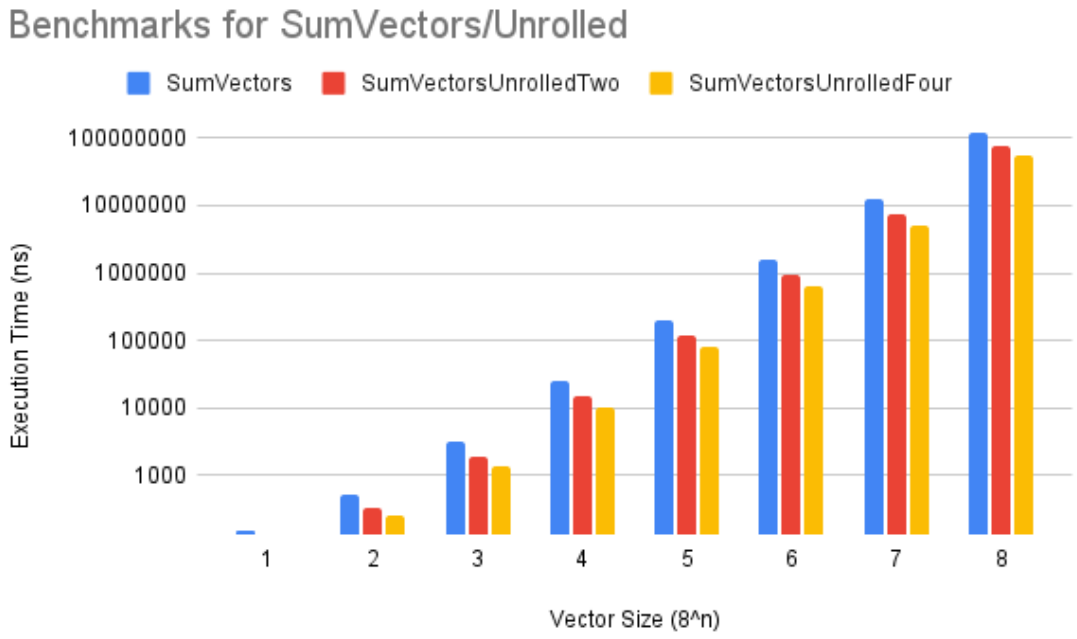


Figure 4.2: Execution times running the `SumVectors()` functions with an increasing vector size and varying loop unrolling factors.

Benchmarks

Figure 4.2 shows how an unrolled implementation has a steady performance increase of about 40% when using a loop unrolling factor of 2, and 60% when using an unrolling factor of 4. This shows a clear performance boost for medium/large-sized streams of data, though the benefits are not as pronounced for smaller vectors (and in the case of using a factor of 4, we see slightly worse performance than the factor of 2).

It is worth outlining that these benchmarks and functions exploited the fact that the size of the vectors were divisible by 2 and 4. Without this precondition, extra guards could be set up after the core loop to catch the extra cases. Omitting these guards saves the functions a few branches, which saves microscopic amounts of time. The programmer should include these based on the preconditions of the data/vector being processed - functional correctness is usually more important than performance.

4.1.3 Prefetching

Prefetching is a technique where processors are instructed to speculatively fetch instructions or data from their storage into faster memory. There are two main classifications, the first being hardware prefetching, usually involving a dedicated mechanism in the processor that recognises patterns of instructions or data being requested by the program. The alternative approach is software prefetching, which is where prefetch instructions are inserted into code. This is either performed by the compiler, or manually by the programmer.

Motivation

Computers often have a two-level cache hierarchy consisting of primary on-chip cache and secondary off-chip cache. A *cache miss* occurs when a CPU attempts to retrieve data in a cache, but the data is not there (contrary to a cache hit). The data must then be retrieved from a lower level of cache/memory (i.e. the secondary cache) which has a longer access time. This time penalty, added to the time penalty from attempting to retrieve data from the first level of cache, is detrimental to low-latency applications and must be mitigated where possible.

Description

The CPU can be instructed to fetch instructions or data into faster (and more local) memory, reducing latency when the time to actually use it arrives. This technique is known as prefetching, where modern hardware can automatically speculate the instructions/data that will be used by recognising patterns in the execution of a program using a *hardware prefetcher*. There are two main approaches: *sequential* prefetching and *stride* prefetching. The prefetcher works well for regular memory accesses, but is not as effective for irregular accesses (such as those that are data-dependent).

Software prefetching is another approach to instructing the CPU when to prefetch instructions. The compiler is able to do this automatically by embedding prefetch instructions into the compiled code. For example, one method involves first prefetching data into L2 cache with a `vprefetch1` instruction, followed by a `vprefetch0` instruction to move the data into L1 cache. GCC uses `__builtin_prefetch` on memory addresses to explicitly move data into a cache, as long as the target supports prefetching.

The timing of the prefetch is also important. If a prefetch instruction is performed too late the data will not yet be in cache and a cache miss will occur as before. If a prefetch instruction is too early, the data could be evicted from the cache before it is accessed, essentially causing another cache miss.

Listing 4.7 contains an implementation of a binary search. The two `mid` locations are prefetched for the next iteration of the `while` loop using the `__builtin_prefetch` intrinsic, with the second and third arguments specifying read-write access and temporal locality respectively. The `0` indicates that the code is preparing for a read-only access, and the `1` indicates a low degree of temporal locality (it will not be reused often).

Listing 4.7 BinarySearch with prefetching

```
static int BinarySearchPrefetched(int target, int nums[],
    int size) {

    int left = 0;
    int mid;
    int right = size - 1;

    while (left <= right) {

        mid = (left + right) / 2;

        __builtin_prefetch (&nums[(mid + 1 + right) / 2], 0, 1);
        __builtin_prefetch (&nums[(left + mid - 1) / 2], 0, 1);

        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else {
            return mid;
        }
    }

    return -1;

}
```

There are plenty of cases where software prefetching is advantageous to relying on automatic hardware prefetching. The following list contains examples of when explicitly requesting data is beneficial to an application:

- **Irregular memory accesses:** The binary search is a good example of prefetching memory irregularly. The hardware prefetcher is unlikely to predict the data that should be fetched as the access pattern of the array seems random, so by using GCC's `__builtin_prefetch` intrinsic we can see an improvement in the benchmarks.
- **Handling a large number of data streams:** Prefetching from a stream of data (e.g. from loops) can be limited by the hardware available. For example, a *Loop Stream Detector* detects when the CPU is executing a loop in an application. When the number of streams exceeds the capacity of the hardware, hardware prefetchers struggle to keep track of the number of streams. This can be mitigated by utilising software prefetching and inserting prefetch requests for each stream independently.
- **Handling short data streams:** Hardware prefetchers do not automatically know the direction and distance of a stream. Training time is needed, and even aggressive prefetchers need at least two cache misses to detect the direction of a stream. This means that short streams of data will result in latency penalties from training, whilst the hardware prefetchers may not even be trained at all. In the context of low-latency applications, the programmer may want the training time (and cache misses) to be eliminated altogether. For this reason, explicit software prefetching will be required to prevent the extra time penalty that occurs. It is worth noting that modern hardware has been developing the ability to handle short streams with hardware-based prefetching, but explicitly declaring prefetch instructions within software eliminates the risk of hardware failing to handle the stream without a cost.
- **Fine-grained control over cache:** With software prefetching, the programmer has more control of where the prefetched data is placed in the cache hierarchy. In most cases, the data should be placed in L1 cache (the highest and fastest level of memory), but hardware prefetchers either have their own cache insertion policy which may end up with data being

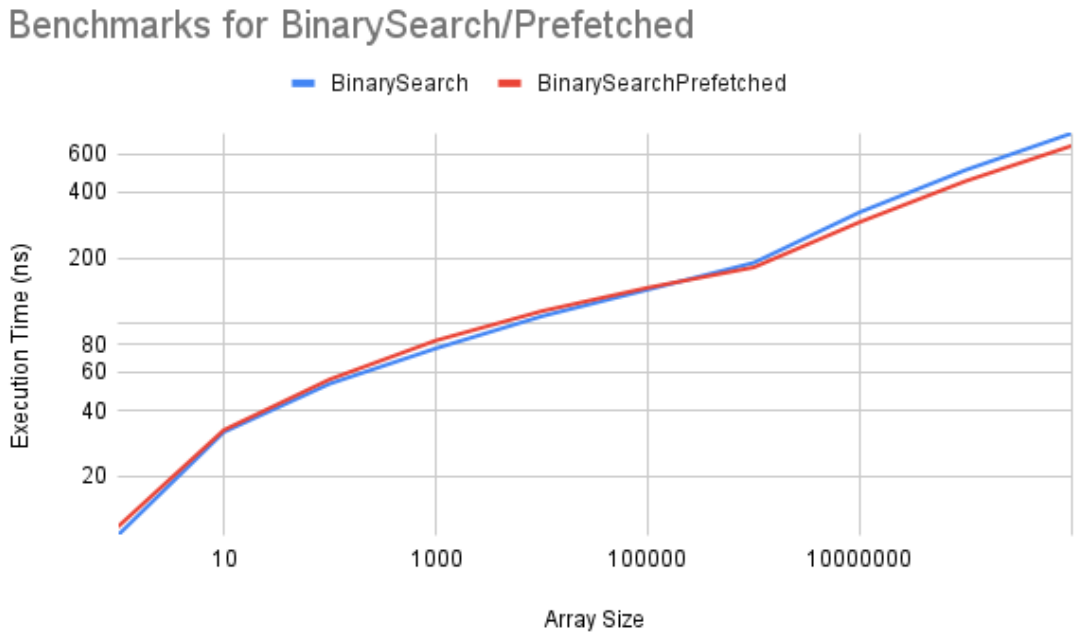


Figure 4.3: Execution times running the `BinarySearch()` functions with varying array sizes.

placed unintentionally in L2 or L3 cache. For applications where minimising latency is critical, the programmer will want as much control of the cache as possible.

Benchmarks

Figure 4.3 shows the results of benchmarking the `BinarySearch` function, increasing the size of the array each time by a factor of 10. The results show that there was not any benefit to prefetching for arrays with up to 100,000 elements. A few nanoseconds were lost, possibly due to mistimed prefetch instructions. However, arrays with more than 1,000,000 elements demonstrate a performance improvement of around 10%.

4.2 Branch-Free Computing

Pipeline hazards are defined under certain classes; one class of hazards are *control hazards*, caused by both conditional and unconditional branches (jumps). Branch instructions are implementations of control flow in loops (for-loops, while-loops, etc.) and conditionals (if statements, switch statements) that cause computers to execute a different sequence of instructions if it is *taken*. If it is *not taken* then the instruction following it is executed.

Conditional branches are a source of latency in applications because its condition must be evaluated and the conditional branch instruction must have completed the execution stage in the instruction pipeline. It is unknown whether the branch is taken until this is completed, so the correct instruction that should be executed after it is unknown. This results in a *pipeline bubble* - empty slots in the instruction pipeline because the following instruction cannot be determined.

Branch-free programming is becoming a popular approach to designing applications, both at the hardware and software level. This section examines *predication*, a hand-written programming technique where the programmer transforms their code to make it deterministic, and *SIMD instructions*, an instruction set that takes advantage of hardware capable of processing data in parallel.

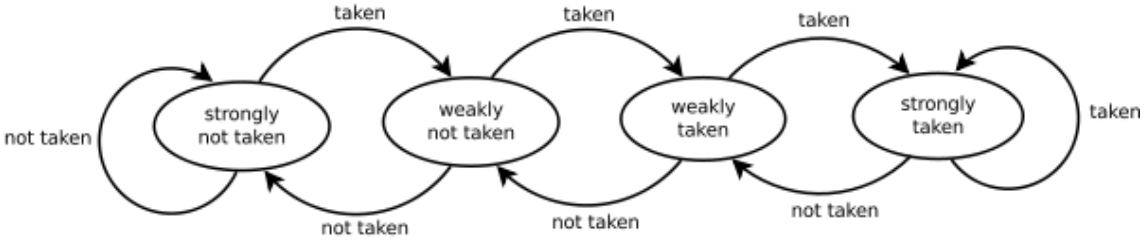


Figure 4.4: The four states used in a simple state machine for branch prediction.

Listing 4.8 Code for CountEvens()

```
1  static int CountEvens(std::vector<int> &nums) {
2
3      int count = 0;
4
5      for (int i = 0; i < nums.size(); i++) {
6          if (nums[i] % 2 == 0) {
7              count++;
8          }
9      }
10
11     return count;
12 }
```

4.2.1 Branch Prediction

There are different schemes that solve the performance issues with branches at the hardware level. *Branch predictors* are one such solution which keep track of whether branches from conditional jumps are taken or not.

Description

Branch Predictors are digital circuits that attempt to improve the performance of the instruction pipeline and minimise control hazards. By *predicting* whether a branch is taken or not, instructions can be speculatively processed within the pipeline, keeping it full and minimising delays. Making an incorrect decision (otherwise known as a branch misprediction) results in the following instructions in the pipeline to be flushed, as they were processed prematurely.

One method of branch prediction is to use a state machine with four states, as shown in Figure 4.4. Evaluated branches update their corresponding state machine by moving its state closer towards the *strongly not taken* and *strongly taken* states based on the result of the conditional branch.

Another implementation of a branch predictor uses a *pattern history table*, with its entries being two-bit counters. The advantage is that it can spot recurring patterns, which would is not taken into account by the 4-state state machine. The branch predictor can then preemptively process the instructions that should follow the branch.

With this understanding of branch predictors and how they operate, the penalties that can come with branches can be mitigated in a number of ways. One way is to use *predictable* data, i.e. data that will cause branches to mostly be either taken or not, which assists the branch predictor in making the correct decision for which instructions should be fetched.

Benchmarks

Consider the function in Listing 4.8, which returns the number of even numbers in a vector of integers. Benchmarking this function with vectors that have different proportions of even numbers yields an interesting graph of results, shown in Figure 4.5. A smooth curve is present, with the execution time reaching its maximum when the vector has an even distribution of even and odd numbers.

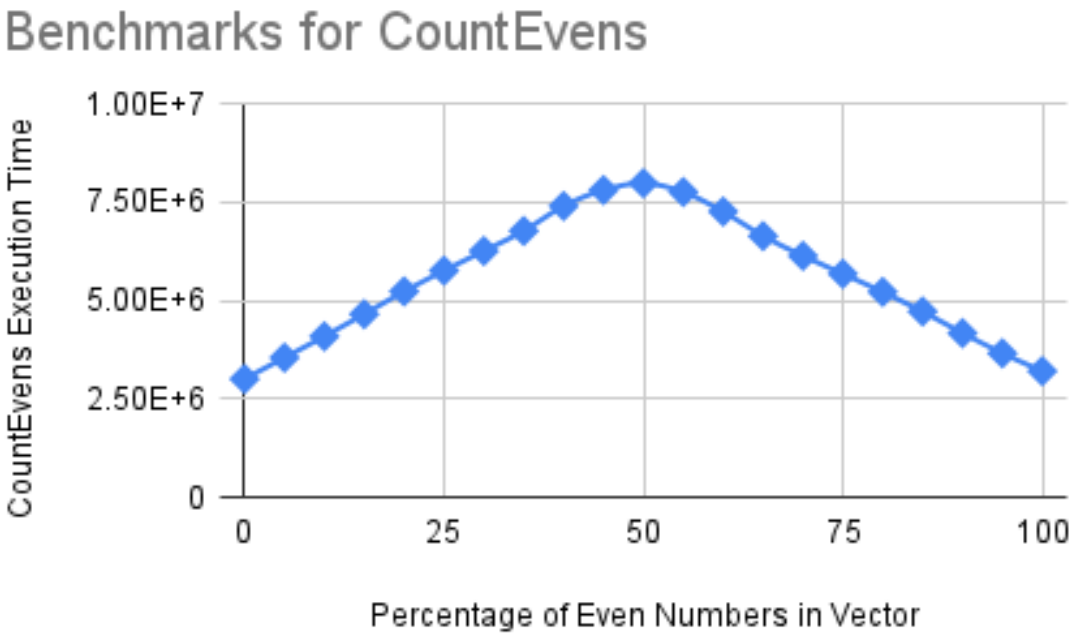


Figure 4.5: Execution times running the `CountEvens()` function, varying the percentage of even numbers within the vector.

These results demonstrate how a vector with an equal number of odd and even numbers suffers the most from branch misprediction penalties, with the execution time of the function being 2.5x the duration of the function of entirely odd numbers. The conclusion that can be taken from this is that branches do not necessarily result in longer execution times, as predictable input data can mitigate the number of branch mispredictions caused.

4.2.2 Predication

Predication is a branch-free programming technique that avoids conditional branch instructions. Predication has non-branch instructions associated with a predicate, and a boolean value controls whether the state of the program is changed (with the state being modified if `true` is evaluated).

Motivation

Branches are sources of pipeline hazards that introduce latencies during runtime. The cost of a branch misprediction can be high on architectures that are deeply pipelined. This may not seem like a problem for powerful systems, but low-latency applications can require every nanosecond available to gain an advantage.

Producing code that avoids branches can often be performed by modern compilers, but detecting every pattern available is not realistic nor is it something that an engineer would want to rely on the compiler to do.

Description

Listing 4.9 contains a function which returns a copy of a `vector<int>` excluding values above a defined threshold. A branch is performed on each iteration of the loop, checking if the current value exceeds the threshold. This means that each iteration has the risk of a branch misprediction, which could result in a total latency penalty of a considerable size.

Listing 4.10 is an improved implementation of the `TrimVector` function. This exploits the evaluation of `input[i] < max` into a boolean, where true and false is equivalent to 1 and 0 respectively. This value is then added to the `outputI` variable, (used to indicate the position in the array) which eliminates the branch altogether. This technique is known as *predication*.

Listing 4.9 Code for TrimVector()

```
static std::vector<int> TrimVector(int max,
    const std::vector<int>& input, int size) {

    std::vector<int> output(size);
    int outputI = 0;

    for (auto i = 0; i < size; i++) {
        if (input[i] < max) {
            output[outputI++] = input[i];
        }
    }

    return output;
}
```

Listing 4.10 Code for TrimVector()

```
static std::vector<int> TrimVectorPredicated(int max,
    const std::vector<int>& input, int size) {

    std::vector<int> output(size);
    int outputI = 0;

    for (auto i = 0; i < size; i++) {
        output[outputI] = input[i];
        outputI += (input[i] < max);
    }

    return output;
}
```

Listing 4.11 Additional check at the end of TrimVectorPredicated()

```
if (output[outputI] < max) {
    output.pop_back();
}
```

Listing 4.12 Generic condition statement

```
if cond {
    foo();
} else {
    bar();
}
```

This predicated function, however, is functionally different to the original. `output[outputI]` will hold the last element of `input` as the elements of `input` are always assigned somewhere to the back of `output`. Therefore, an edge case exists where the last element of `output` will be incorrect if the last value of `input` is not greater than `max`. This is fixed by adding a final check at the end of the code, as shown in Listing 4.11.

Consider a more general example outlined in Listing 4.12. If a condition in the `if` branch evaluates to true, the `foo()` function is called. Otherwise, `bar()` is called in the `else` branch. Listing 4.13 shows what the instructions generated from compiling this pseudocode could look like.

Predication will instead look to eliminate the branches by having instructions associated with predicates, only being executed when the predicate is true. Listing 4.14 suggests what the instructions would ideally look like. The instructions contain both function calls, but one will execute whilst the other will not.

Benchmarks

The benchmarks for `TrimVector` are showin by the graph in Figure 4.6 with a considerable performance improvement when predicated once the given vector exceeds a certain size, just before the vector contains 1000 elements. It is worth noting that the original function takes 14 times longer between a vector size of 1000 and 10000, whilst its counterpart has the expected 10x increase.

4.2.3 SIMD Instructions

SIMD (Single Instruction, Multiple Data) describes computers with the capability of performing a single operation on multiple data points in parallel. *SIMD Instructions* and their corresponding intrinsics allow the programmer to utilise the vector registers that are present in these computers/processors to obtain a substantial gain in performance and execution speed.

Listing 4.13 Generic instructions generated from the condition statement

```
branch-if-cond-to l1
call bar
branch-to l2
l1:
call foo
l2:
.
.
.
```

Listing 4.14 Generic condition statement predicated

```
1 (cond) call foo
2 (not cond) call bar
```

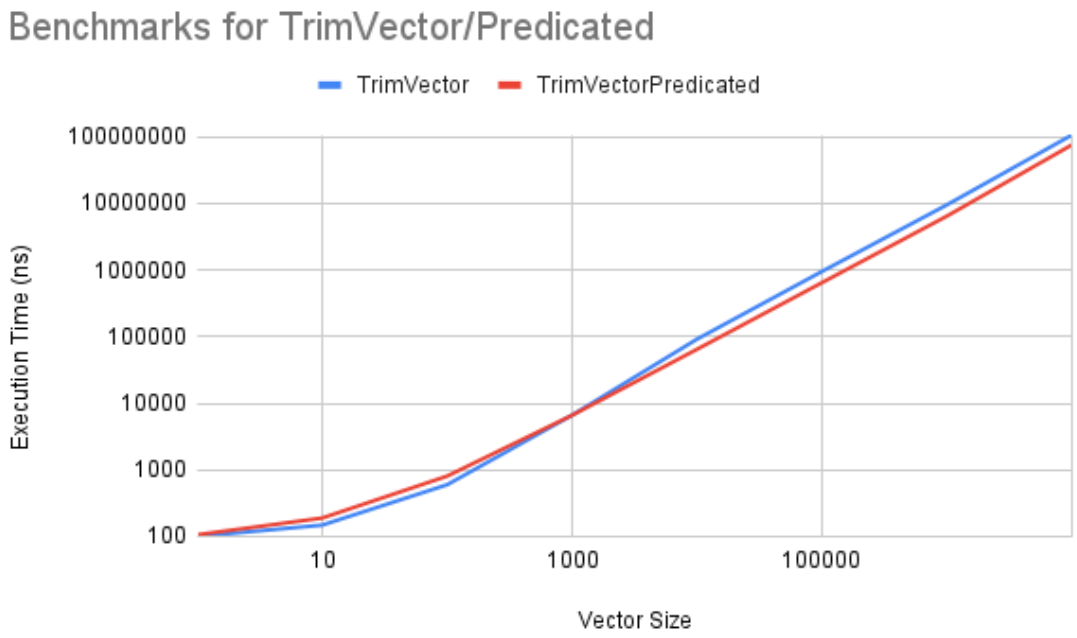


Figure 4.6: Execution times running the `TrimVector()` functions with an increasing vector size.

Listing 4.15 Code for MultiplyAddScalar()

```
static void MultiplyAddScalar(const float* a, const float* b,
    const float* c, float* d) {

    for (int i = 0; i < 8; i++) {
        d[i] = a[i] * b[i] + c[i];
    }

}
```

Listing 4.16 Code for MultiplyAddVectorized()

```
static __m256 MultiplyAddVectorized(__m256 a, __m256 b, __m256 c) {
    return _mm256_fmadd_ps(a, b, c);
}
```

Motivation

Flynn's taxonomy categorises the forms of parallel computer architectures, proposed in 1966 and extended in 1972. This was extended further by Ralph Duncan in 1990 with *Duncan's taxonomy*, which proposed the addition of vector processing.

Scalar processors are the simplest type of processors, usually processing one item at a time, handling each instruction sequentially. These are otherwise known as *SISD* (single instruction single data) CPUs.

Recent developments in computing have standardised vector processors, which instead operate on arrays of data. A vector processor can perform an instruction on several data points using SIMD (single instruction multiple data) instructions, with modern SIMD instructions being introduced to Pentium processors in 1999.

Description

Vectors are instruction operands represented as a one-dimensional array of data elements, being either integer or floating-point values. Vector registers hold these data elements as a single vector and can vary in size. Intel's AVX-512 instruction set performs vector operations on 512-bit vectors, equivalent to 16 `ints`.

Compilers attempt to auto-vectorize code, which they are successful for in simple cases like the element-wise multiplication of two vectors. However, there are cases where auto-vectorisation is unsuccessful. In most cases, the programmer must manually vectorize their code, as shown in Listing 4.15, which multiplies the elements of two arrays containing 8 floats together and adds the values of a third array, storing it in a fourth array (d).

SIMD instructions can perform the same calculation by loading the values into vector registers and performing the same calculation on them. In the following example, the arrays of floats have been casted to the `__m256` data type, allowing it to be loaded into an AVX register. The `mm256_fmadd_ps` then performs the same calculation as before (a * b + c) and returns the result in Listing 4.16.

Use Cases

The benefits of SIMD instructions/operations are almost universal where they can be applied, and are widely utilised for applications for 3D graphics or audio/video processing. The following list contains applications that benefit from vectorisation significantly:

- **3-Dimensional Processing**: There is significant potential for parallelism when working with 3D vertices and their transformations, which applications can exploit by using SIMD instructions to increase performance. This allows more objects to be rendered in scenes, and opens up the possibility of shadows and reflections to be rendered in real-time.

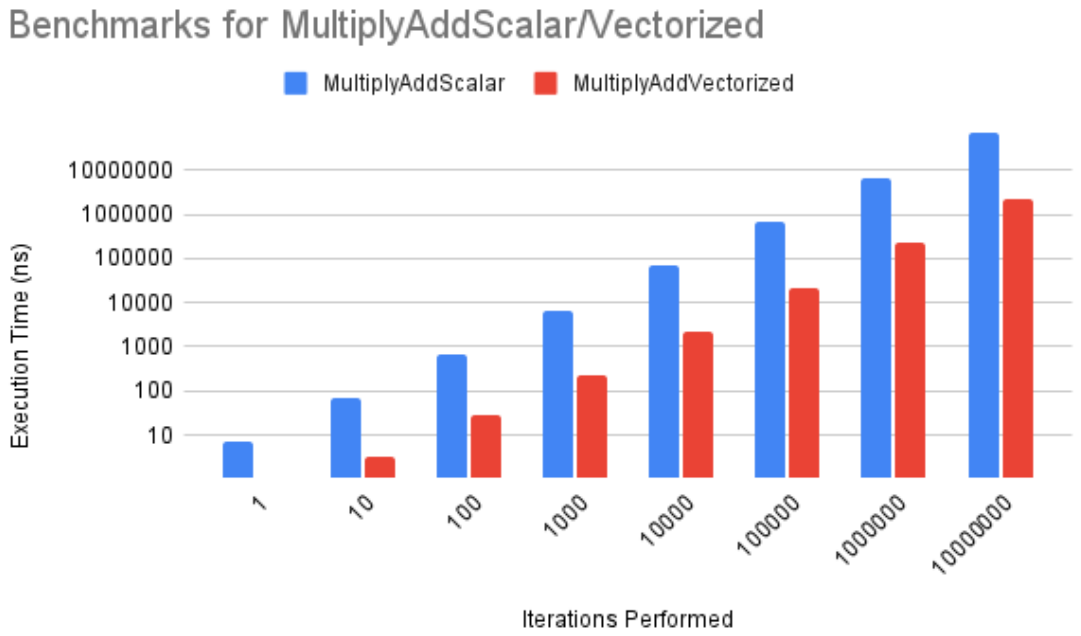


Figure 4.7: Execution times running the `MultiplyAdd()` functions with an increasing number of repeated function calls.

- **Image Processing**: The parallelism in both the code and data structures of imaging software allows SIMD instructions to improve the performance of the application. The increased performance enables users to maintain interactivity even as the image size increases greatly, and should result in gains of the speed of image transformations and manipulations.
- **Video Processing**: Considering that videos are streams of images shown to an end user, the benefits of image processing described in the previous section also apply to video processing, which typically are even more demanding performance-wise. SIMDs provide the potential for video transformations to be performed in real-time, which requires a huge amount of optimisation considering the performance challenges that are already present in image processing.

Benchmarks

There was a clear disparity between the speed of the calculation performed when using classical operations against vectorised operations with the `MultiplyAdd` function, with almost a 70% performance increase when the operations were repeated a significant number of times, as shown in Figure 4.7.

The movement of the data in the array to the vector registers were not accounted for, so the performance impact suggested by the numbers may be slightly overstated. However, real-world applications will often perform multiple operations on the vector registers before unloading the values, so the overhead from loading/unloading will be less significant.

4.3 Application Design

Low-latency applications do not achieve performance just from micro-optimisations throughout the codebase. Hand-written compiler optimisations will only squeeze extra micro/nanoseconds out of an application. Whilst this may be the difference between success and failure, the overall system must first be designed appropriately to have a competitive level of performance. This section outlines examples of broader design choices that contribute to greater performance, from considerations of the design structures used in the application, to the modern hardware that can be utilised that completely change the way that data is processed.

Listing 4.17 Code for a single-threaded Increment()

```
static void Increment(int &counter, int iterations) {
    for (int i = 0; i < iterations; i++) {
        counter++;
    }
}
```

Listing 4.18 Code for a multi-threaded, atomic Increment()

```
static void IncrementAtomic(std::atomic<int> &counter, int iterations) {
    for (int i = 0; i < iterations; i++) {
        counter.fetch_add(1);
    }
}
```

4.3.1 Lock-Free Programming

Many modern applications are multi-threaded to improve performance and utilise the computational power available. However, a lot of programs work on shared resources (e.g. files, data structures) that must have safeguards to prevent data races. The contention between threads to access these resources can hamper the benefits of parallel programming, even degrading the performance of an application in certain scenarios.

Motivation

Intuition will often suggest that having more threads working on a task will decrease the execution time of it, as tasks can be distributed to several parallel workers. Examine the following three implementations of a function that increments a `counter` a given number of times: Listing 4.17 shows a single thread and does not use any synchronisation primitives. Listing 4.18 uses *atomic variables* to prevent multiple threads reading/writing to the variable simultaneously, using *memory barriers* to enforce a certain ordering when accessing the counter to prevent data races. Listing 4.19 is an alternative multi-threaded implementation useing a mutex/lock, which only allows the thread that "holds" the mutex to operate on the counter.

Benchmarking these functions shows considerable performance degradation when having more than one thread working on the task. Table 4.1 shows the results when using these functions to increment a counter 10 million times. Contrary to the idea that more threads corresponds to increased performance, the results show that having two threads compete for a single resource caused the execution time to triple for both `IncrementAtomic()` and `IncrementMutex()`. Not only that, significant overhead is induced from having additional concurrency primitives protecting the counter variable from creating a data race.

Listing 4.19 Code for a multi-threaded, lock-based Increment()

```
static void IncrementMutex(int &counter, int iterations, std::mutex &mtx) {
    for (int i = 0; i < iterations; i++) {
        mtx.lock();
        counter++;
        mtx.unlock();
    }
}
```

Function	Execution Time (ns)
Increment (one thread)	12,713,615
IncrementAtomic (one thread)	54,969,640
IncrementAtomic (two threads)	151,883,027
IncrementMutex (one thread)	162,426,559
IncrementMutex (two threads)	472,225,654

Table 4.1: Execution times of the Increment() functions using different synchronisation primitives.

Description

The broader issue that was observed in the `Increment` implementations was *write-contention* - two threads were attempting to modify a variable, but to preserve the correctness of the data synchronisation primitives must be used to maintain mutual exclusion. In other examples, *read-contention* may also be an issue though the overheads associated with it are less pronounced. Read/write locks are designed to give threads access to variables whenever possible, restricting access to variables unless absolutely necessary.

This level of overhead is significant for low-latency applications, and there have been designs to minimise contention where possible. A common design pattern is to have a single threaded *producer* which writes to a data structure. As demonstrated in the previous section, having a single thread writing to data storage performs better than multiple threads that need to manage mutual exclusion between themselves. The *consumers* only need read-access, so they are able to concurrently read objects without a concern for exclusivity.

If an application must use mutexes/locks and atomic operations, it is important to use the synchronisation primitives that are most suitable for the task. For example, *spinlocks* are an implementation of a lock that are useful for short critical sections (a section of code executed by multiple threads where different sequences of execution affects the result) and/or when the lock will not often be contested... However, the worst case behaviour of spinlocks are extremely expensive and can cause massive overheads when used incorrectly.

An alternative to spinlocks (among many others) are *sleeping locks*, which ask the OS to put the calling thread to sleep if it does not initially obtain the lock. When the lock is freed, the thread is woken back up using a system call. The problem is that the system calls to make the thread sleep are expensive, making this solution suboptimal for locks that are not held for long periods of time.

There are lock implementations that combine the advantages of both spinlocks and sleeping locks (e.g. futexes), otherwise known as *hybrid locks*. It is important for the programmer to understand the advantages and disadvantages with each lock to choose the most appropriate one for each use case.

4.3.2 Kernel Bypass

Kernels provide a layer of abstraction between the applications in a computer and the hardware that they are running. This abstraction results in limitations among applications, which cannot fully utilise the capabilities of the underlying hardware. Bypassing the kernel is a concept where processing is moved from the kernel to the user space.

This section focuses specifically on networking, where the difference in capabilities between the kernel and the hardware is at its greatest.

Motivation

The basic Linux kernel is limited in its performance, and prevents applications from properly utilising the hardware available in the computer. Figure 4.8 shows how the kernel sits between the applications in the userspace and the hardware. Within the kernel is the network stack, which implements protocols up to the transport layer (application layer protocols such as HTTP or FTP are typically implemented in the userspace).

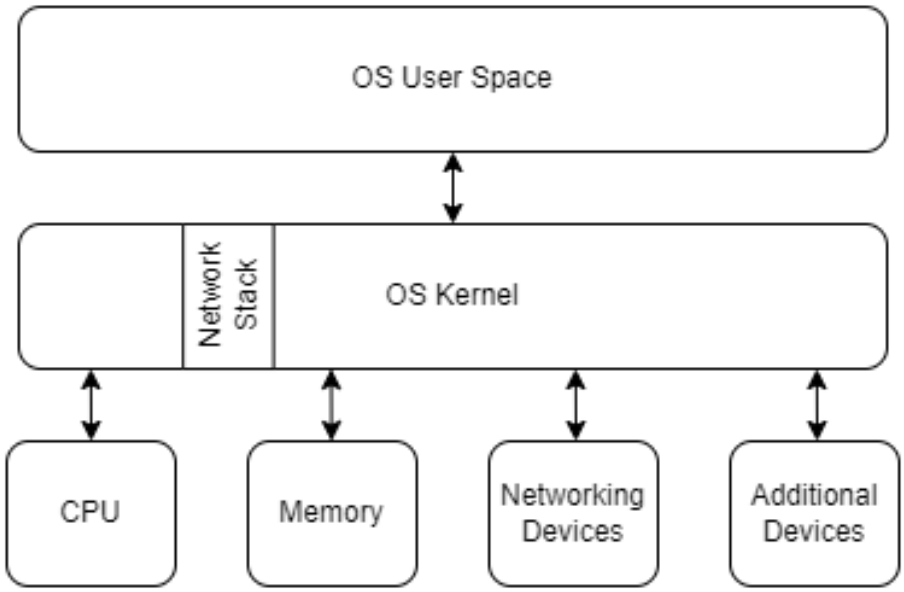


Figure 4.8: A high-level diagram of the kernel within a computer.

This level of abstraction has performance consequences, with overheads that are widely known and documented [21]. POSIX sockets are the current standard interface for programming within networking, and most in-kernel network stacks implement socket operations as systems calls. These have substantial overheads (e.g. context switches), and they do not take advantage of multicore CPU architectures. Furthermore, the socket API forces the operating system to use dynamic memory allocation when handling packets by wrapping it in a buffer object, which puts pressure on the OS.

The kernel can only process about 1 million packets per second, whilst modern 10Gbps Network Interface Cards (NICs) can often process 10 million per second, and higher-end NICs hitting over 200 Gigabit Ethernet (GbE) cards are being produced. A 200GbE card can deliver packets with as low as 61 nanoseconds between them. This means there is much less time available to process packets without a significant backlog. Clearly, the current design of the kernel cannot keep up with the capabilities of networking hardware.

Description

One approach to overcoming the setbacks of the kernel is to bypass it altogether. Moving the processing of protocols to the user space allows programmers to avoid the overheads of system calls and OS abstractions. Ethernet connections have a couple of solutions:

1. Assign NICs to applications, allowing them to be programmed. The *Data Plane Development Kit* is an example of this.
2. Map NIC queues to the OS's address space. The *netmap* project is a framework that does this, achieving high speed packet I/O.

Figure 4.9 illustrates these solutions within the computer architecture, which span from the user space directly to the hardware.

Both of these approaches allow the packets to be processed within the user space with minimal overhead induced by the operating system. This allows high-end NICs to be utilised properly with-

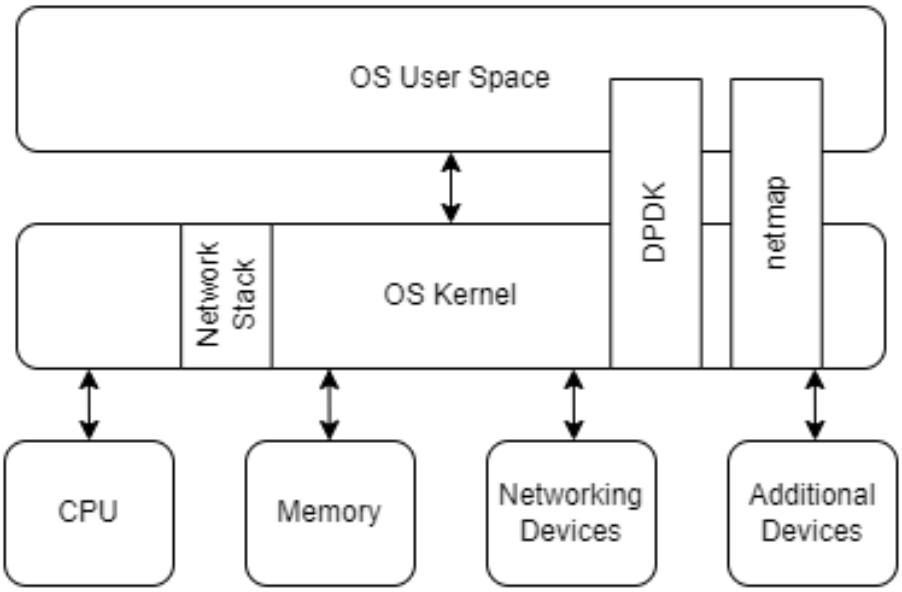


Figure 4.9: A high-level diagram of the kernel within a computer.

out being bottlenecked by the kernel, shifting the responsibility of high performance applications to the abilities of the programmer/engineer. Additional responsibilities are now present within the user space in its method of handling incoming packets. These must be decoded and interfaces will need to be provided to successfully manage their messages.

4.3.3 Cache Warming

Cache warming occurs when an application artificially moves data into the cache to ensure that it is available when needed at critical moments. This means the cache is kept "warm", and eliminates the penalties incurred from cache misses.

Motivation

In some scenarios the low-latency sections of code might be the least frequently executed part of the application, which leads to potential issues with how it is stored in memory. After a certain amount of time, function blocks and their corresponding data will be removed from the instruction and data cache. This will result in cache miss penalties which may significantly harm the performance of the application.

Description

Cache Warming is the technique of executing the critical sections of code periodically to ensure that it is stored in the cache (as well as any related data). However, there is a possibility that the application does not want the function to carry out all its actions, so it is important to set this up in a way that prevents any unwanted actions from executing [22].

A relevant example is a high-frequency trading platform that continuously reads market data and sends out buy orders at a relatively lower rate. The buy orders must be executed quickly to reach the exchange as fast as possible, and cache misses will severely hamper the application's capability to do so.

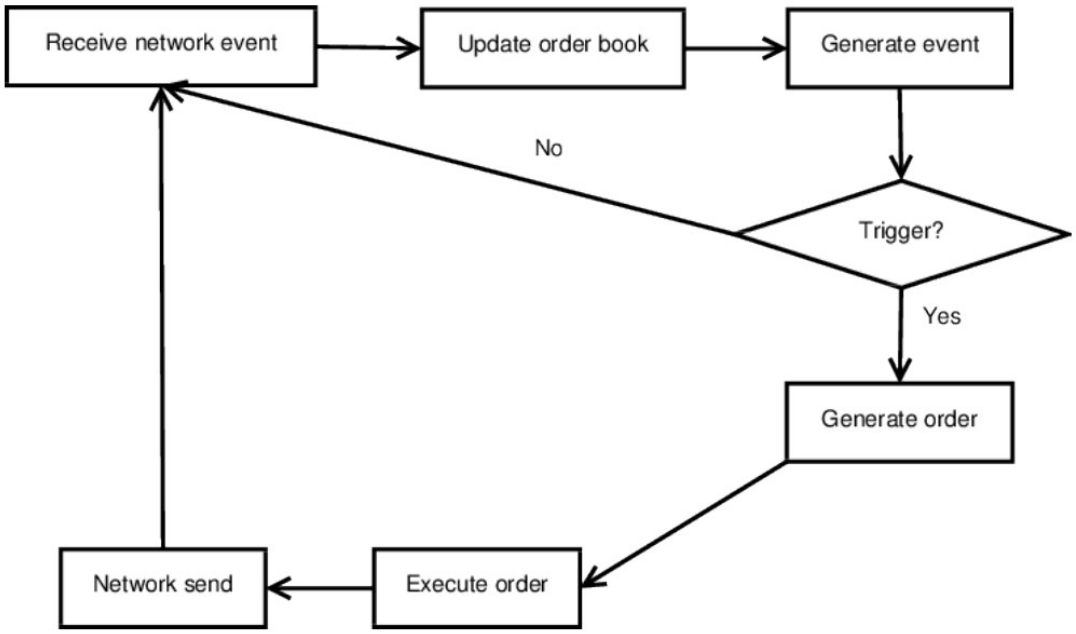


Figure 4.10: A financial application without Cache Warming

Figure 4.10 shows the decision tree of such a scenario. A network event is received, causing the application to update its order book and generate an event. Most of the time, nothing will be triggered, and it will continue polling/waiting for a network event to occur again.

The problem occurs when the event is triggered. An order is generated, executed, and sent via the network. However, the instructions are unlikely to be stored within the instruction cache because of the lengthy interval without calling that code.

To solve this issue, a solution can be implemented as illustrated in Figure 4.11. After an event is generated, choosing not to trigger it will lead to a second decision, determining whether warming is necessary. If so (perhaps after a certain amount of time has elapsed), a dummy order is generated and executed, keeping the performance-critical segments of code within the cache and circumventing avoidable cache miss penalties.

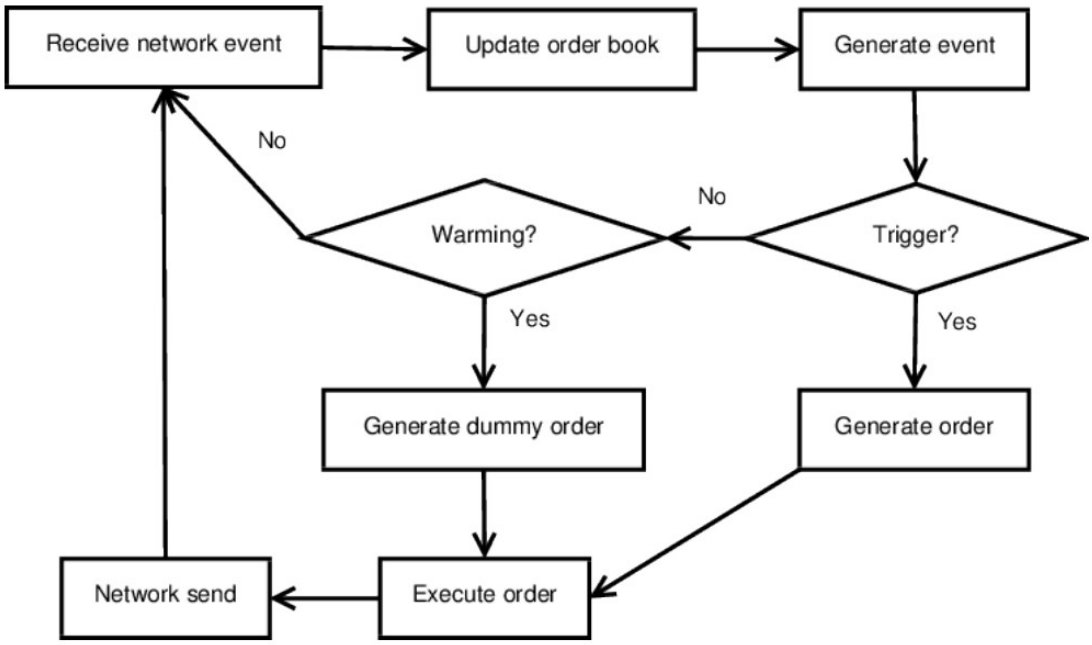


Figure 4.11: The same financial application but with Cache Warming

Chapter 5

Java Modular Packet Processor

This project presents the Java Modular Packet Processor (JMPP), a lightweight Java application and library for processing network packets with a user-defined graph of individual processing operations. The JMPP library allows users to create variations of *Processors*, consisting of components that perform different operations on packets. These components are interchangeable, allowing users to specify a network of operations to route packets through with a graph-like structure.

This library was not created with the intention of producing production-level applications, but to serve as a supplementary educational resource for the Low-Latency Programming Repository to demonstrate the quantitative advantages of using lock-free data structures for multi-threaded programs. The most important contribution of this chapter are the benchmarks produced when using the LMAX Disruptor to pass packets between components instead of queues.

This section presents an overview of the different components of JMPP, an example implementation of a Processor using the library, and its performance when using the Disruptor. The repository is available on GitHub at:

https://github.com/gordonl0811/JMPP

5.1 High-Level Architecture

Processors consist of *Readers* and *Components*. *Disruptors* are used to pass *Packets* between these, which serve as an alternative to classic queue-based implementations. This section describes these core concepts and their roles within JMPP, and how to tie they work together to develop a Processor.

5.1.1 Disruptors

The Disruptor is an open-source, high performance data structure written in Java and developed by the LMAX Group. It aims to reduce latency and high levels of jitter that are discovered when using bounded queues by offering an alternative for exchanging data between

This library uses Disruptors to pass `Packets` between components, simulating a processing pipeline that can fork and rejoin streams of Packets based on user-defined logic.

5.1.2 Readers

`Readers` are responsible for providing packets to the processor. Readers feed packets into a Disruptor, allowing Components to process each item. In its current state, JMPP is limited to obtaining packets from PCAP files; processors cannot take in streams of packets from sockets, which is a major feature that would be implemented if the library was intended for real workloads.

Component	Description
IPv4Filter	Outputs packets with an IPv4 (Layer 3) header
IPv6Filter	Outputs packets with an IPv6 (Layer 3) header
TCPFilter	Outputs packets with a TCP (Layer 4) header
UDPFilter	Outputs packets with a UDP (Layer 4) header
IPv4DestinationFilter	Outputs packets that have an IPv4 address within the given range
IPv6DestinationFilter	Output packets that have an IPv4 address within the given range
MACAddressRewriter	Rewrites the source/destination addresses of the Layer 2 header
IPAddressRewriter	Rewrites the source/destination addresses of the Layer 3 header
PortRewriter	Rewrites the source/destination ports of the Layer 4 header
Writer	Writes the packets to a specified file
Dropper	Discards packets i.e. no-op

Table 5.1: Available JMPP components and their respective subcategories.

5.1.3 Components

`Components` are the heart of JMPP and are responsible for analysing and modifying the packets that are passed through them. These are classified into 3 distinct subcategories:

- `Filters` identify packets with specific traits, such as common headers within the TCP/IP model, or specific address ranges. They will publish the packets of interest to its output Disruptor.
- `Rewriters` modify attributes of packets at different layers, including addresses, ports, etc.
- `Outputters` publish packets externally (except for `Droppers`), e.g. into a PCAP. In the graph of Components for each processor, these can be considered the last stage of processing.

Every component records the number of packets that it processes, which is accessible through the `getPacketCount()` method. Future work could involve making metric collection an option, reducing the performance impact that may be associated with this. Table 5.1 outlines the components available and their functionalities.

5.1.4 The Processor Lifecycle

Processors have three distinct stages, reflected in the `PacketProcessor` interface:

1. `initialize()`: All Disruptors are started, ready to receive packets.
2. `start()`: Readers start producing packets until a certain condition is met (or can be left running indefinitely).
3. `shutdown()`: Gracefully shuts down the Disruptors associated with the Processor.

These need to be called in turn; omitting the `shutdown()` call will result in undefined behaviour. The separation of these stages allows for more accurate benchmarking, which is demonstrated in Section 5.3.

5.2 Implementation

JMPP is written in Java to maintain compatibility with the Disruptor. This section outlines the specific details of the classes within JMPP and their relationships with each other. Figure 5.1 contains a visual representation of the JMPP library, a class diagram generated by the IntelliJ IDE.

JMPP uses the lightweight "pkts.io" library to decode PCAP files, translating them into `Packet` objects with a number of utility functions. The library limits the functionality of JMPP, which will require a different packet decoding infrastructure to increase the features available.

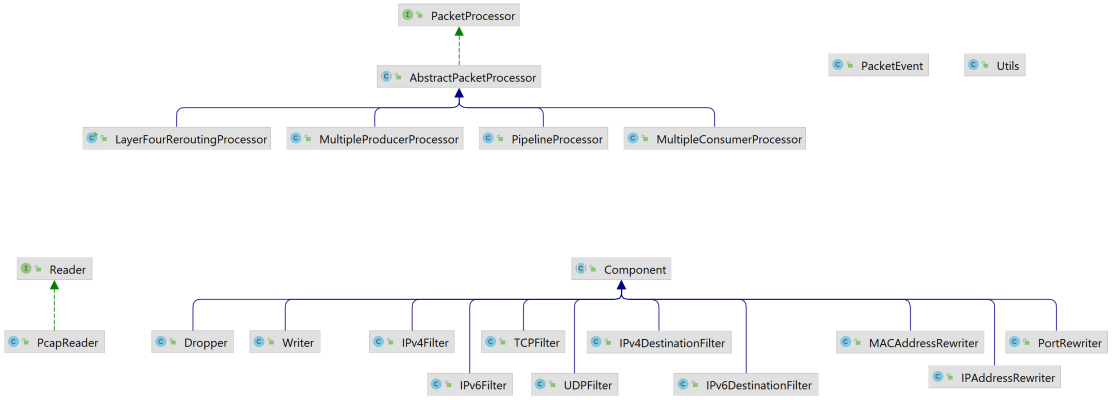


Figure 5.1: Class Diagram for JMPP.

5.2.1 Readers

Readers have a base `Reader` interface which defines three methods:

- `void initialize()`: Start up the output Disruptor associated with the Reader.
- `void start()`: Begin reading packets from the Reader's source and loading them into the Disruptor.
- `void shutdown()`: Shut down any Disruptors and close any sources being used.

For the purpose of this project, JMPP only has the `PcapReader` implementation of the `Reader` interface.

5.2.2 Components

Every components inherits a `Component` class, which mostly defines abstract methods for each concrete implementation:

- `void initialize()`: Start up the Disruptors associated with the component.
- `void shutdown()`: Shut down Disruptors associated with the component.
- `void process(Packet packet)`: Perform the operation on the packet, optionally publishing it to an output Disruptor.

Additionally, the abstract `Component` class implements the `void onEvent()` method required by Disruptors to handle the objects taken from their ring buffers. This class implements this by calling the `process()` method (implemented by the concrete classes) and increasing a counter that records the number of packets handled by the component. This can be accessed with a public `long getPacketCount()` method, and is used for terminating Processors (outlined in the next section).

5.2.3 Processors

Processors are responsible for orchestrating the interactions between the components and tying them together with Disruptors. A user could set up a network of components and manage each one manually, but the library provides an `AbstractPacketProcessor` class with a user guide to accelerate the development of Processors.

To supplement the methods described in Section 5.1.4, Processors are expected to implement the following methods:

1. `setReaders()`: Return a list of the `Readers` being used. This can be as simple as using the `List.of(component1, component2, ...)` method.
2. `setComponents()`: Return a list of the `Components` being used in a similar fashion to `setReaders()`.

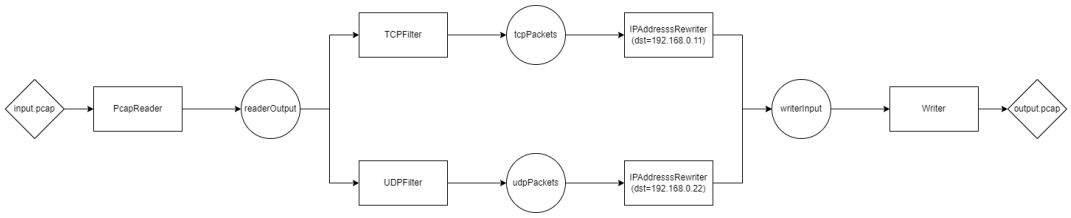


Figure 5.2: Architecture diagram for a Processor that redirects TCP/UDP packets.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
2	0.001000	127.0.0.1	127.0.0.1	TCP	54	20 → 80 [SYN] Seq=0 Win=8192 Len=0
3	0.008000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
4	0.013000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
5	0.018001	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0
6	0.021502	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
7	0.026000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
8	0.031000	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0
9	0.035000	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0
10	0.039000	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0

Figure 5.3: Wireshark output for the TCP/UDP packets before processing.

3. `shouldTerminate()`: The condition for a processor to finish its `start()` call. A common implementation is to check whether a certain number of packets have been processed using `getPacketCount()` on `Outputter` components. The user can set this to return `false` to keep the processor running indefinitely.

The constructor of the Processor is then responsible for creating Disruptors for each component and connecting them together. The methods in the `AbstractPacketProcessor` will initialise all the defined readers and components with `initialize()`, release packets into the processor with `start()`, and gracefully stop the components with the final `shutdown()` call.

JMPP states that it is the user's responsibility to ensure that the components are correctly wired together and that the Processor is designed appropriately; for example, it would not make sense for an `IPv4Filter` to output to an `IPv6AddressFilter`.

5.2.4 Example: Redirecting TCP and UDP Packets

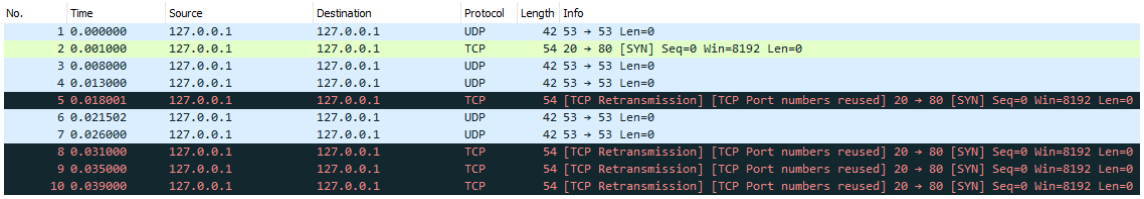
A slightly more complex example can be used to showcase the flexibility of the library. Considering the following requirements of a processor:

- A stream of packets containing a mixture of TCP and UDP packets are received
- Packets have a destination of `127.0.0.1`
- TCP packets are to be redirected to `192.168.0.11`
- UDP packets are to be redirected to `192.168.0.22`

Figure 5.2 shows how this flow of operations can be represented with a Processor using JMPP. A `PcapReader` sends packets to a `readerOutput` Disruptor, which is read by a `TCPFilter` and `UDPFilter`. These send the relevant packets to their own Disruptors, which are read by `IPAddressRewriters` which change IPs to `192.168.0.11` and `192.168.0.22` respectively. These components write the processed packets to a shared `writerInput` Disruptor, which a `Writer` will send to a PCAP.

Writers are used to check the functional correctness of the processor; in a more realistic scenario, these packets will be sent to their corresponding destinations by wiring up the Processor to a socket.

The processor can be tested by providing it with a PCAP of 10 packets, 5 of which are TCP packets and another 5 are UDP packets. Figure 5.3 shows the contents of the PCAP decoded by Wireshark. Figure 5.4 shows the packets after they have been processed. The TCP and UDP packets are not in the exact order as they were before due to being processed in parallel, but ordering is retained within the TCP and UDP streams respectively.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
2	0.001000	127.0.0.1	127.0.0.1	TCP	54	20 → 80 [SYN] Seq=0 Win=8192 Len=0
3	0.008000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
4	0.013000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
5	0.018001	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0
6	0.021502	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
7	0.026000	127.0.0.1	127.0.0.1	UDP	42	53 → 53 Len=0
8	0.031000	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0
9	0.035000	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0
10	0.039000	127.0.0.1	127.0.0.1	TCP	54	[TCP Retransmission] [TCP Port numbers reused] 20 → 80 [SYN] Seq=0 Win=8192 Len=0

Figure 5.4: Wireshark output for the TCP/UDP packets after processing.

5.3 Performance: Disruptor vs Queues

The main contribution this library has to this project is the demonstration of the performance with a lock-free data structure like the Disruptor over queues. The objective is to show the latency reductions that can be unlocked through carefully choosing data structures that are built for higher performance. In this case, the Disruptor is designed for producer-consumer workloads following the single-writer principle. Avoiding the use of locks gives the data structure an edge over alternatives like the standard queues available within Java.

5.3.1 Benchmarking Setup

The Java Microbenchmark Harness (JMH) was used to measure the performance of the application, with the focus being the time taken between a packet entering the processor to the moment it is outputted by the Processor.

The key to obtaining trustworthy benchmarks is to isolate this hot path of the application and measure its execution multiple times. JMPP was designed with this in mind, with the compartmentalisation of its life-cycle allowing users to measure the execution time of a batch of packets without accounting for the setup and tear-down time penalties.

The high-performing `java.util.concurrent.ArrayBlockingQueue` was used to compare the performance of the application with an alternative data structure to the Disruptor. This required the application to be completely re-implemented with the queue; the unorthodox API of the Disruptor meant that a slightly different approach had to be taken with sending/receiving packets. The Disruptor also had extra functionality, particularly the ability to multicast events to its consumers.

The main changes included:

- Individual components implemented the `Runnable` interface and were initialised as threads.
- Each component would poll the queue to receive packets.
- Shutdown involved cleaning `Threads` up by interrupting them.
- Multicasting was replicated with a dedicated component that iterated through queues and pushed packets to all of them.

JMH has two distinct stages for running benchmarks in addition to the measurement of each execution, conducted via Java annotations. The `@Setup` stage allowed for Processors to be created and initialised, and `@Teardown` allowed Processors to be cleaned up gracefully. This meant that only the `start()` function was measured; the moment the first packet entered the Processor, until the last packet was fully processed.

5.3.2 Results

Three different setups were benchmarked as illustrated by Figure 5.5. Droppers were used as the final component instead of writers because of the significant overhead introduced from interacting with files. This section reports the results of benchmarking each Processor design. Comprehensive results tables can be found in the Appendix.

Pipeline

The results for the Processor with a simple pipeline indicated a significant performance improvement using the Disruptor-based approach. The Processor observed a 70% decrease in execution

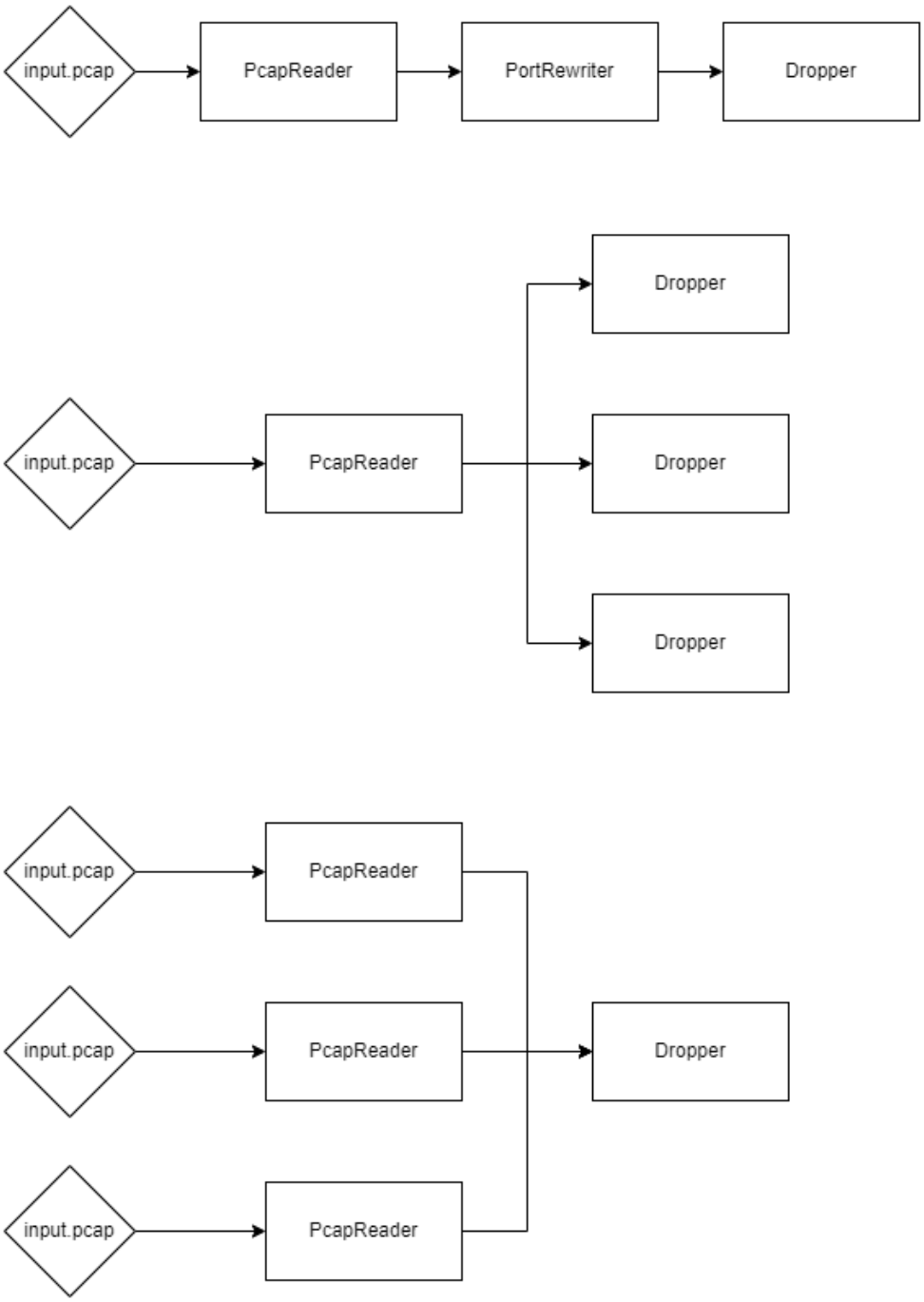


Figure 5.5: From top to bottom: a Processor with a pipeline design, multiple consumers, and multiple producers.

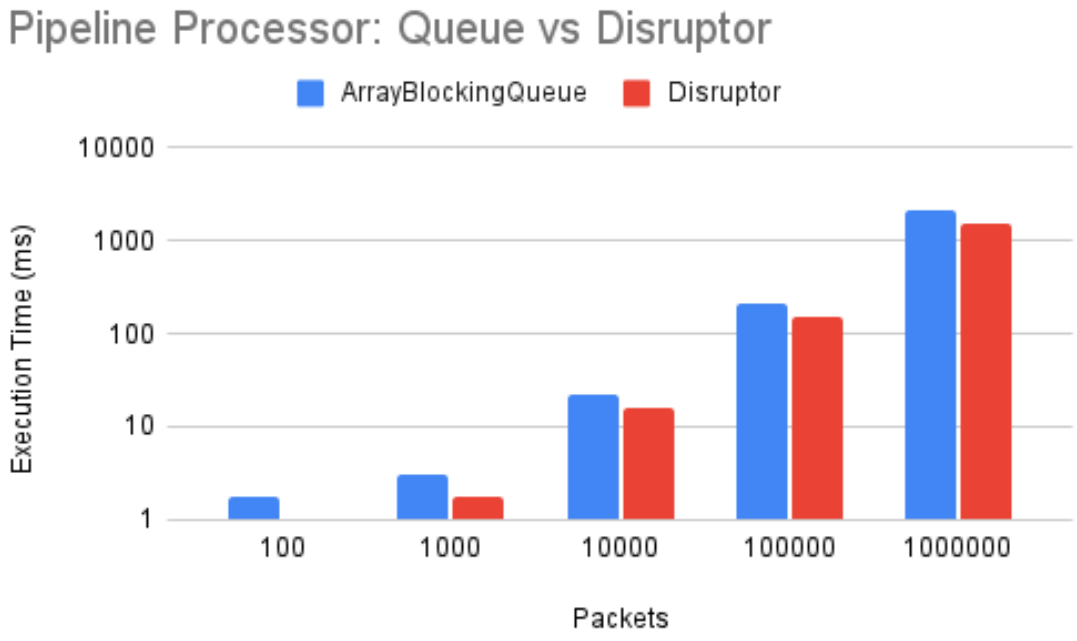


Figure 5.6: Results for the Processor with a pipelined design.

time with a batch of 100 packets, which stabilised from 10000 packets onwards with a performance improvement of just over 30%.

Multiple Consumers

Similar to the previous section, the Pipeline with multiple consumers demonstrated a notable benefit from using the Disruptor. Figure 5.7 shows the results of the benchmarks. Execution times were consistently reduced by around 70%, indicating a significant performance improvement. However, this may be overstated due to the extra `Multicaster` component used by the queue implementation, which would induce extra overhead.

Multiple Producers

The processor with multiple producers generates rather counter-intuitive numbers (shown in Figure 5.8), with the implementation with queues outperforming the Disruptor-based implementation as the batch of packets increases in size.

There are a few possible explanations for this result. The Disruptor's technical guide explicitly states that the greatest performance benefits come from using a single-writer principle. This principle suggests that a single producer will be more effective than multiple producers writing to a shared data location. This is because of the overhead associated with using locks and other concurrency mechanisms, which multiple producers will need to use to maintain functional correctness by eliminating data races. Disruptors with multiple producers will suffer a performance hit compared to their single-writer alternatives.

Another possible reason could be a failure to carefully benchmark the Disruptor-based implementation, with setup/teardown overheads being bundled into the timed execution. In contrast, the thread-based approach of the queue implementation may be better isolated than its counterpart.

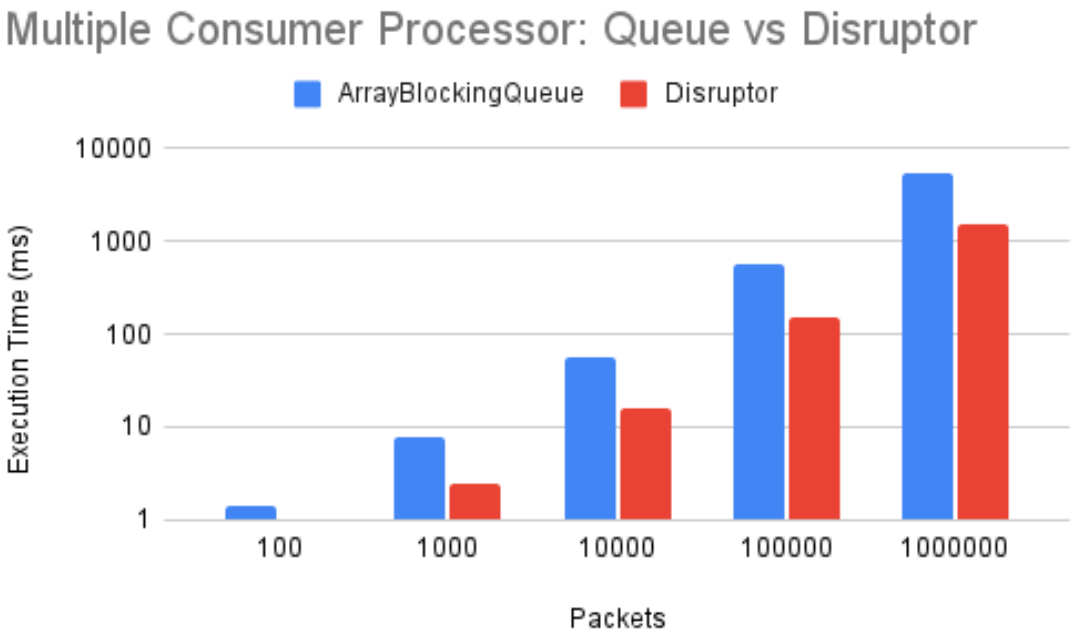


Figure 5.7: Results for the Processor with multiple consumers and a single producer.

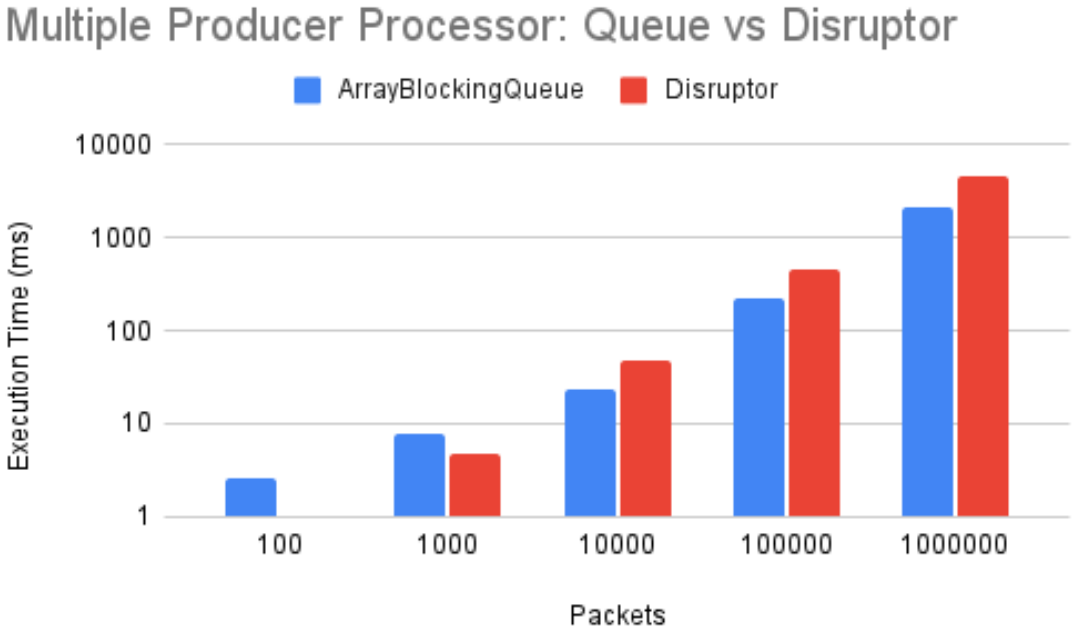


Figure 5.8: Results for the Processor with a single producer and multiple producers.

Chapter 6

Evaluation

This chapter evaluates the two products of this project with defined sets of criteria. The evaluations are heavily oriented around user feedback, both with quantitative and qualitative data collected from a sample of the desired target audience.

The most important statistic to be collected was the overall impact of the repositories on the participants' understandings of low-latency programming. Individuals were asked the question, "Rate your knowledge of low-latency programming", between 1 (no knowledge at all) and 10 (industry-level expertise).

Being an entirely subjective question, the value collected alone is useless for comparing levels of understanding between participants. However, a useful result that can be measured is the increase in understanding when the question is asked again after thoroughly reading through both repositories and their documentation.

This project had a target of having readers report back a 2.5-point average increase in their understanding of the low-latency programming domain. Meeting this value would suggest that readers felt they had a wider appreciation of programming strategies that help contribute to performance, without necessarily expecting readers to assume significant expertise on specific topics.

6.1 Feedback Collection

Obtaining useful metrics for teaching resources is a challenge. There are a few approaches that are taken within educational institutions:

- **Formative assessments:** continuous tests to measure an individual's progress in their understanding of a topic, usually with low-stakes involved.
- **Summative assessments:** a final evaluation of the understanding of a topic by comparing the results to a standard or benchmark, usually with high-stakes involved.
- **Feedback:** subjective responses that demonstrate how an individual feels about their understanding of a topic.

A combination of these approaches would be the ideal solution for evaluating the usefulness of this project and its products from an educational perspective. However, doing so is unfeasible as there are key differences that prevent this from being effectively implemented.

Formative assessments assume that the recipient is learning information over an extended period of time. Given the nature of this work, which does not expect individuals to spend more than a few hours learning content, there are not enough opportunities to measure this appropriately. Summative assessments are slightly more feasible. Short quizzes at the end of each section could be useful, but there is a danger of tailoring the questions to the content in the source material, producing biased results which do not accurately reflect whether the resource has credibility. Additionally, there is no real pressure for individuals to do well, as the resources are independent of any accreditation.

The most relevant approach, which this evaluation mostly focuses on, is user feedback from a sample of individuals with a strong background in computer science and computing. This is because of their foundations in the subject that will help to understand terms and scenarios that may be addressed, whilst not necessarily having the expertise in the low-latency domain already.

This evaluation collected feedback from 10 participants, most being students at various UK-based universities, whilst the rest were recent graduates who have experience as software engineers. This particular demographic was chosen because of its close alignment with the target audience of the project; software engineers with the fundamentals of computer science and software engineering, but with little to no experience with low-latency programming.

6.2 Low-Latency Programming Repository

6.2.1 User Feedback

Participants were asked to thoroughly analyse the Low-Latency Programming Repository by reading through the base documentation on the landing page, then reading through each strategy outlined in the repository's `docs/` folder. The reader then evaluated the repository based on the following metrics:

- Depth: the level of detail of the topics covered
- Quality: the standard of the technical writing

Depth

Average Score: 7.9

Lowest Score: 6.0

This measurement was based on the participants' opinions about the level of detail that the repository provided in each topic. This could be based on whether there were outstanding questions about the topic which were not addressed.

This average score given indicated that the participants were mostly happy with the detail available. However, some participants suggested that it would have been useful to have an explanation of specific sections of the assembly code. This is less intuitive to read than high-level programming languages, so readers who are not familiar with the syntax will struggle to understand its context.

Quality

Average Score: 9.1

Lowest Score: 7.0

This metric assesses the level of technical writing that is displayed in the repository. This includes whether the writing was aptly supported by figures and tables, and if concepts were described succinctly without losing clarity.

The high average score indicated that readers thought the repository was well-written and comprehensible. One evaluator mentioned that one or two examples were usually given with each topic, and that this was "sufficient to help give an introduction to how each strategy works in practice". Another comment mentioned that the graphs with log-scales could be difficult to read, so the provision of the raw figures for each benchmark were a useful addition.

One participant did mention that a few terms were glossed over, and could have done with further explanations. For example, the concept of *thrashing* could have been described with some assisting pictures.

6.3 Java Modular Packet Processor

The primary purpose of JMPP was to exhibit the performance benefits of using a lock-free data structure, and this section reports on the feedback that participants had on this library as a sup-

plementary resource However, there is still scope to evaluate the library based on its performance and functionality, and this section will also evaluate it based on these considerations.

6.3.1 User Feedback

Participants were asked to evaluate the JMPP library by reading through the base documentation on the landing page, analysing the source code for JMPP in the repository, then examine the partial re-implementation and benchmarks of JMPP in the Low-Latency Programming Repository. After completing these steps, participants were asked to consider the following three metrics, once again rating them out of 10:

- Code Quality: the standard and organisation of the code that was written
- Design: the overarching purpose of the library
- Relevancy: the relation between JMPP and the Low-Latency Programming Repository

Code Quality

Average Score: 8.6

Lowest Score: 8.0

This metric is used to assess the quality of the library was the standard of the code itself. This includes the structure of the packages, the organisation of the class hierarchy, and the commenting within the source code.

The generally high score and low deviation of results suggests that readers were comfortable navigating around the repository and could understand the purpose of each section of code.

Design

Average Score: 7.4

Lowest Score: 4.0

This metric assesses the overall design of the repository and how clear it was to understand the purpose of it. This also included the benchmarks that were carried out and what was being measured.

The slightly lower score suggests that there could be work done to make the motivations of the library clearer, which was to demonstrate the numerical advantages of using lock-free data structures in applications.

Relevancy

Average Score: 7.6

Lowest Score: 5.0

This measurement assesses whether the library successfully justifies the effectiveness of lock-free designs, and whether the participants felt that the relationship between JMPP and the Low-Latency Repository were clear.

Similarly to the scores given for the Design metric, this suggests that the library could do more to support the findings of the Low-Latency Repository. This could be carried out by implementing more low-latency strategies on the codebase, and benchmarking them appropriately. One reader in particular mentioned that having a formal hypothesis about the benchmarks and comparing it to the actual results could have strengthened the relationship between JMPP and the repository.

6.3.2 Performance

Whilst JMPP is unique in its functionality, it does not have the performance capable of handling real-time workloads with sophisticated hardware. There are several reasons for this, the core reason being that the code was not written with low-latency optimisations in mind (beyond the implementation of the Disruptor).

The library uses a lot of inheritance, which has a slight overhead associated with them. To mitigate this, the library would need to be re-implemented with concrete classes. This would result in a lot of code duplication and would require more detailed documentation on how to develop processors with the library. Naturally, the application should be benchmarked and profiled to assess which optimisations are good for producing lower latencies.

Based on the findings of the Low-Latency Programming Repository, it is also clear that the application cannot make use of the higher throughput that modern NICs are capable of handling. The application is limited by the Linux kernel, and may need to use a kernel bypassing library to raise the ceiling of the workloads it can process.

6.4 Overall Feedback

On average, participants stated that their level of understanding was at 4.3. All participants reported an improvement in their knowledge within the low-latency domain, with the average increase being 2.7 points. Whilst an increase was expected, the magnitude of this was significant, and meets the initial goal of this project. This indicates that this project was successful in its purpose, and serves as a useful learning resource for software engineers early in their career.

Chapter 7

Conclusion

7.1 Summary

This project presented two closely-related products, the first being the Low-Latency Programming repository, containing insights into low-latency programming strategies intended for newcomers to the field of high performance computing. The secretive nature of low-latency programming has presented an opportunity to provide an educational resource that does not exist in academic literature (to the best of our knowledge). This repository covers three key areas:

- **Hand-written compiler optimisations:** code transformations that often need manual implementation for latency-critical applications.
- **Branch-free computing:** increasing the predictability of programs and paving the way for vectorising instructions.
- **Application design:** broader design choices that enable applications to be competitive and improve performance at a larger scale.

The second product is the Java Modular Packet Processing library. The relative simplicity of the library allows it to successfully demonstrate the advantages of lock-free programming, a design strategy that is explored in the Low-Latency Programming repository. The library used the Disruptor data structure, in place of ArrayBlockingQueues which a re-implemented version of the library contains. Benchmarking the original and re-implemented library illustrates the significant gains in performance that can be achieved by selecting appropriate data structures.

Both resources were evaluated by a sample group of students with a strong background in programming, through a degree in Computing, Computer Science, or a closely related subject. The results of this evaluation were positive, with the average participant claiming that their self-assessed understanding of low-latency increased by 2.7 points (on a scale from 1 to 10).

7.2 Future Work

7.2.1 Low-Latency Programming Repository

The Low-Latency Programming Repository contains just the surface of high performance computing, and could benefit from more depth and breadth in its content. Whilst it still serves as a good focal point for newcomers to the field, there is a lot more work that can be done to expand it further.

Additional performance analysis

The repository does a good job of demonstrating the raw performance gains through careful benchmarking of most programming strategies. However, additional metrics are available and would have been nice to include. This would be achieved through application profiling, from simple built-in

tools such as Linux's `perf`, to Intel's more sophisticated VTune Profiler. This means the following measurements could be collected:

- Pipeline stalls
- Cache hits
- Cache misses
- Effective CPU Usage

Specialised hardware

Hardware such as Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) were avoided in this project, partly because it shifts the focus from software improvements to hardware, which is not the core subject. However, these play a significant role in the advancement of low-latency programming, and would be worthwhile exploring in detail. They are also extremely relevant within industry:

- FPGAs are used by high-frequency trading firms due to its parallel architecture and determinism, making it ideal for reducing latencies.
- ASICs are being explored for their use within Real-Time Machine Learning, which will be built on low-latency systems.

7.2.2 Java Modular Packet Processor

As the primary purpose of JMPP was to serve as a supplementary resource for the Low-Latency Programming Repository, future work would involve implementing and benchmarking additional low-latency techniques on the library. There is a huge scope of what could be carried out, most of which comes directly from the strategies discussed within Chapter 4.

The utility libraries used in JMPP are the bottlenecks for both functionality and performance. JMPP is built on the lightweight pkts.io Java library developed by aboutsip, which is severely limited in its capabilities beyond decoding PCAP files. Replacing this would be the first step; this section discusses what the alternatives could be and the optimisation opportunities that would arise.

Kernel bypassing libraries as infrastructure

To achieve the performance that can keep up with the capabilities of modern hardware, the library could use kernel bypassing libraries/tools such as DPDK. This will significantly increase the rate of packets that can be processed, opening up the potential for the application to be used with real workloads.

Hand-written optimisations in practice

Using kernel bypass libraries to replace the in-kernel network stack requires requires the user space to handle and decode raw packets. "pkts.io" does this to some extent, but a low-latency library would most likely need to be designed and implemented. There does not seem to be an open-source, Java-based library that does this, so it is likely that this would be developed alongside JMPP. This presents an opportunity to demonstrate the hand-written compiler optimisations outlined in the Low-Latency Programming Repository, which could then be benchmarked in a similar fashion to the original Disruptor benchmarks.

7.3 Ethical Considerations

In this section we consider implications for stored packet data, misuse of packet processors/decoders, and software licensing.

No sensitive data is used within the library. Packets processed for testing/benchmarking purposes were generated using Scapy, a python-based tool used for forging network packets. The PCAP files stored within the repository are prototypes and therefore do not contain real-world data.

As this project investigates low-latency programming and creates resources and code specifically for educational purposes, there are fewer concerns about the project being used in conjunction with real-life applications.

However, packet decoders such as Wireshark do have a risk of misuse, and can enable unlawful actions such as collecting data from networks without the owner of the network's informed consent. There is also potential for military applications to be developed and used for controversial motivations. For these reasons, the possibility of misusing JMPP must also be considered.

Whilst this library is very unlikely to be used for such purposes in its current state (due to its limited nature), the project may need safeguards set up to prevent misuse if it is developed further. Clear warnings would be outlined within the repository and the application about the act of monitoring/damaging networks.

A final, minor ethical consideration that has been identified is the publication of open-source work. JMPP uses other open-source libraries, so careful steps have been carried out to ensure that JMPP has complied with them.

Appendix A

Low-Latency Programming Repository

A.1 Benchmarking Results

Iterations Performed	Execution Time (ns)	Inlined Execution Time (ns)
1	6	5
10	35	30
100	339	287
1,000	3,237	2,756
10,000	32,711	27,364
100,000	322,723	274,317
1,000,000	3,266,517	2,755,897
10,000,000	32,828,489	27,445,521

Table A.1: Execution times for the `Cube()` functions

Vector Size (8^n)	Execution Time (ns) Unrolling Factor = 1	Execution Time (ns) Unrolling Factor = 2	Execution Time (ns) Unrolling Factor = 4
1	148	128	129
2	501	333	259
3	3,204	1,905	1,357
4	24,682	14,468	9,843
5	197,597	115,575	78,331
6	1,584,229	954,879	626,115
7	12,884,701	7,495,871	5,089,184
8	118,495,092	76,134,393	56,429,343

Table A.2: Execution times collected for the `SumVectors()` function.

Array Size	Execution Time (ns)	Prefetched Execution Time (ns)
1	11	12
10	32	33
100	53	56
1,000	77	84
10,000	108	114
100,000	143	146
1,000,000	190	181
10,000,000	324	292
100,000,000	506	450
1,000,000,000	745	653

Table A.3: Execution times collected for the `BinarySearch()` functions.

% of Even Numbers	Execution Time (ns)
0	3,003,347
5	3,543,721
10	4,084,344
15	4,648,943
20	5,227,015
25	5,754,339
30	6,253,471
35	6,762,285
40	7,395,470
45	7,799,465
50	7,974,268
55	7,750,755
60	7,254,713
65	6,632,476
70	6,128,936
75	5,674,009
80	5,212,384
85	4,717,636
90	4,165,762
95	3,654,579
100	3,202,696

Table A.4: Execution times collected for the `CountEvens()` function.

Vector Size	Execution Time (ns)	Predicated Execution Time (ns)
1	100	106
10	148	189
100	598	806
1,000	6,638	6,609
10,000	91,995	64,771
100,000	955,303	648,566
1,000,000	9,568,340	6,481,235
10,000,000	106,974,999	76089237

Table A.5: Execution times running the `TrimVector()` functions with an increasing vector size.

Iterations Performed	Scalar Execution Time (ns)	Vectorised Execution Time (ns)
1	7	1
10	69	3
100	679	30
1,000	6,793	227
10,000	67,856	2,273
100,000	678,666	22,019
1,000,000	6,845,885	218,869
10,000,000	69,030,639	2,186,023

Table A.6: Execution times collected for the `MultiplyAdd()` functions.

A.2 Compiled Assembly Instructions

Listing A.1 Assembly for SumVectors())

```
.L8:
        mov     eax, DWORD PTR [rbp-20]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-48]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     ebx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-20]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-56]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     eax, DWORD PTR [rax]
        add     ebx, eax
        mov     eax, DWORD PTR [rbp-20]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-40]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     DWORD PTR [rax], ebx
        add     DWORD PTR [rbp-20], 1
```

Listing A.2 Assembly for SumVectors() unrolled

```
.L8:
        mov     eax, DWORD PTR [rbp-20]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-48]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     ebx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-20]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-56]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     eax, DWORD PTR [rax]
        add     ebx, eax
        mov     eax, DWORD PTR [rbp-20]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-40]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     DWORD PTR [rax], ebx
        mov     eax, DWORD PTR [rbp-20]
        add     eax, 1
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-48]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     ebx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-20]
        add     eax, 1
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-56]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     eax, DWORD PTR [rax]
        add     ebx, eax
        mov     eax, DWORD PTR [rbp-20]
        add     eax, 1
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-40]
        mov     rsi, rdx
        mov     rdi, rax
        call    std::vector<int, ...(cnt.)
        mov     DWORD PTR [rax], ebx
        add     DWORD PTR [rbp-20], 2
```

Appendix B

Java Modular Packet Processor

B.1 Benchmarking Results

Packets	Execution Time (ns)	
	ArrayBlockingQueue	Disruptor
100	1.718	0.555
1,000	3.074	1.762
10,000	22.405	15.695
100,000	216.334	154.171
1,000,000	2,140.646	1,546.638

Table B.1: JMPP Results: Pipeline Processor

Packets	Execution Time (ns)	
	ArrayBlockingQueue	Disruptor
100	1.408	0.474
1,000	7.739	2.400
10,000	56.853	15.663
100,000	555.341	155.146
1,000,000	5,513.875	1,565.199

Table B.2: JMPP Results: Multiple Consumer Processor

Packets	Execution Time (ns)	
	ArrayBlockingQueue	Disruptor
100	2.620	0.937
1,000	7.623	4.636
10,000	22.780	46.544
100,000	220.938	454.166
1,000,000	2,191.004	4,593.804

Table B.3: JMPP Results: Multiple Producer Processor

Bibliography

[1] Rotman D. We're not prepared for the end of Moore's Law. MIT Technology Review. February 2020. Available from:
https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/

[2] Aho A, Lam M, Sethi R, Ullman J. Compilers: Principles, Techniques, & Tools, Second Edition. Chapter 1. Addison-Wesley. 2006.

[3] Aho A, Lam M, Sethi R, Ullman J. Compilers: Principles, Techniques, & Tools, Second Edition. Chapter 9. Addison-Wesley. 2006.

[4] Multiple contributors. Compiler Intrinsics, Microsoft Docs. Available from:
https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics

[5] Herlihy M, Shavit N. The Art of Multiprocessor Programming, Revised 1st Edition, Chapter 1. Morgan Kaufmann. 2008.

[6] Multiple contributors. Race Conditions and Deadlocks, Microsoft Docs. Available from:
https://docs.microsoft.com/en-us/troubleshoot/developer/visualstudio/visual-basic/language-compilers/race-conditions-deadlocks

[7] Available from:
https://en.cppreference.com/w/cpp/thread

[8] GeeksForGeeks. The C++ Standard Template Library. Available from:
https://www.geeksforgeeks.org/the-c-standard-template-library-stl/

[9] Thompson M, Farley D, Barker M, Gee P, Stewart A. LMAX Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. 2011. Available from:
https://lmax-exchange.github.io/disruptor/disruptor.html#_memory_allocation

[10] Carruth C. Efficiency with Algorithms, Performance with Data Structures. CppCon 2014. Available from:
https://www.youtube.com/watch?v=fHNmRkzxHWs

[11] Fearless Concurrency. The Rust Programming Language. Available from:
https://doc.rust-lang.org/book/ch16-00-concurrency.html

[12] Jane Street Company Website. Available from:
https://www.janestreet.com/programming/

[13] Scala Website. Available from:
https://www.scala-lang.org/

[14] Latency equlisation: The need for fair and non-discriminatory colocation services. Available from:
https://www.interxion.com/uk/blogs/2018/082/latency-equalisation-the-need-for-fair-and-non-discriminatory-colocation-services

[15] In Pursuit of Ultra-low-latency: FPGA in High-Frequency Trading. Velvetech. Available from:
https://www.velvetech.com/blog/fpga-in-high-frequency-trading/

[16] Gopal B G, Kuppusamy P G. A Comparative Study on 4G and 5G Technology for Wireless Applications. 2015. IOSR Journal of Electronics and Communication Engineering.

[17] software.org. 5G Is Software. 2020. Available from: https://software.org/reports/5g-is-software/

[18] GeeksForGeeks. Circular Queue | Set 1 (Introduction and Array Implementation). Available from: https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/

[19] Thompson M, Farley D, Barker M, Gee P, Stewart A. LMAX Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. 2011. Available from: https://lmax-exchange.github.io/disruptor/disruptor.html

[20] Google. google/benchmark on GitHub. Available from: https://github.com/google/benchmark/blob/main/docs/user_guide.md

[21] Høiland-Jørgensen: Fast programmable packet processing in the operating system kernel. 2018. Available from: https://dl.acm.org/doi/10.1145/3281411.3281443

[22] Bakhvalov D. Performance Analysis and Tuning on Modern CPUs, Chapter 10.3, Cache Warming. Independently Published. 2020.