

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

---

# C++ Design Patterns for Low-Latency Software Applications

---

*Author:*

Alexander Thomas Arzt

*Supervisor:*

Dr. Robert Chatley

*Second Supervisor:*

Prof. William Knottenbelt

Submitted in partial fulfillment of the requirements for the MSc degree in MSc  
Computing of Imperial College London

September 2023

## Abstract

In today's fast-paced global economy, many businesses across multiple industries require their software applications to execute as fast as possible to maintain a competitive edge. However, traditional software engineering design patterns prioritise code robustness and maintainability over latency minimisation. This thesis aims to bridge this gap through a fourfold approach: (1) creating a formal catalogue of low-latency design patterns in C++, the primary language for low-latency development, based on input from domain experts; (2) rigorously benchmarking each pattern to ascertain its impact on latency; (3) developing an open-source application capable of scanning C++ code to identify implementation opportunities for these patterns; and (4) extending the application to autonomously implement the correct design pattern where appropriate, thereby enabling automated code refactoring. Six concrete design patterns were identified, of which five demonstrated a statistically significant latency-reducing effect. The developed application (found [here](#)) accurately identifies four of the patterns and successfully implements them, leveraging OpenAI's Chat-GPT. The contributions of this thesis provide both a robust theoretical framework and a practical toolset for low-latency software development in C++. As such, they should be of particular interest to any business striving for optimised software performance.

---

---

## Acknowledgments

I want to thank Dr. Robert Chatley, Prof. William Knottenbelt, and Prof. Marios Kogias for their invaluable contributions to this project. Their combined expertise, guidance, and support have been instrumental in shaping this work into what it is today. In addition, I am deeply grateful to Dr. Dagmar Arzt, Giulia Gopsill, Enes Sarituc, Önder Sarituc, Dr. Milan Kuzmanovic, and Ayca Weber for their unwavering support throughout the past. Lastly, this work is dedicated to the individual I owe the most, the late Dr. Thomas Arzt (1955-2020).



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Designing Software for Speed: The Importance of Low Latency . . .	1
1.2	Objectives . . . . .	2
1.2.1	Legal, Social, Ethical and Professional Requirements . . . . .	2
1.3	HFT as an Illustrative Example . . . . .	3
1.3.1	A Short Introduction to HFT . . . . .	3
1.4	Outline . . . . .	4
<b>2</b>	<b>Preliminaries</b>	<b>5</b>
2.1	Design Patterns in Software Engineering . . . . .	5
2.2	Why C++ . . . . .	6
2.3	The Fast Path . . . . .	9
2.4	CPU Caches . . . . .	10
2.4.1	The Basics Of Caching . . . . .	10
2.4.2	Cache Structure . . . . .	12
2.4.3	Critical Stride . . . . .	14
2.4.4	Optimising Memory Access . . . . .	15
<b>3</b>	<b>Design Patterns Catalogue</b>	<b>17</b>
3.1	Cold Code Isolation Pattern . . . . .	18
3.2	DOD Pattern . . . . .	21
3.3	Denormalised Data Pattern . . . . .	23
3.4	Cache Priming Pattern . . . . .	25
3.5	TB Branch Elimination Pattern . . . . .	27
3.6	CRTP . . . . .	29
3.7	Miscellaneous . . . . .	31
<b>4</b>	<b>Benchmarking</b>	<b>34</b>
4.1	Foreword . . . . .	34
4.1.1	General Aspects . . . . .	34
4.1.2	Statistical Foundations . . . . .	36
4.2	Cold Code Isolation Pattern . . . . .	37
4.2.1	Setup . . . . .	37
4.2.2	Results . . . . .	38
4.3	DOD Pattern . . . . .	40
4.3.1	Setup . . . . .	40

4.3.2	Results . . . . .	41
4.4	Denormalised Data Pattern . . . . .	43
4.4.1	Setup . . . . .	43
4.4.2	Results . . . . .	43
4.5	Cache Priming Pattern . . . . .	45
4.5.1	Setup . . . . .	45
4.5.2	OCE - Results . . . . .	46
4.5.3	Results . . . . .	48
4.5.4	Interim Conclusion . . . . .	49
4.6	TB Branch Elimination Pattern . . . . .	50
4.6.1	Setup . . . . .	50
4.6.2	Results . . . . .	51
4.7	CRTP . . . . .	53
4.7.1	Setup . . . . .	53
4.7.2	Results . . . . .	54
4.7.3	Excursion . . . . .	54
4.7.4	Interim Conclusion . . . . .	57
<b>5</b>	<b>Software Application: The LLDPA</b>	<b>58</b>
5.1	Functionality . . . . .	58
5.1.1	C++ Code Analysis . . . . .	58
5.1.2	Automated Code Refactoring . . . . .	59
5.2	Demonstration . . . . .	60
5.2.1	The Code . . . . .	60
5.2.2	Running The Application . . . . .	63
5.3	Evaluation . . . . .	68
5.3.1	CppTest1 . . . . .	68
5.3.2	CppTest2 . . . . .	69
5.3.3	Summary . . . . .	70
<b>6</b>	<b>Epilogue</b>	<b>71</b>
6.1	Contribution . . . . .	71
6.2	Future work . . . . .	72

# Chapter 1

## Introduction

### 1.1 Designing Software for Speed: The Importance of Low Latency

In today's digitalised world, numerous businesses across many domains require their software applications to execute as fast as possible in order to stay competitive and thus remain profitable. Ghosh [1] highlights several industries where low-latency requirements for software are particularly prominent and explains the underlying reasons:

- **Telecommunications:** Applications of telecommunications providers must handle many connections simultaneously, and thus, speed is paramount for efficiency and optimal user experience.
- **Operating Systems:** Despite modern operating systems' substantial size and tremendous complexity, they must be performant to remain competitive.
- **Flight Software and Traffic Control:** In commercial and military aircraft, software must react as fast as possible to external events. Because human lives are at stake, an application might become entirely useless if it runs with suboptimal latency.
- **Gaming:** Since video games are interactive graphical applications, fast rendering speed is vital to ensure good responsiveness to the user.
- **Augmented and Virtual Reality:** AR and VR applications have similar needs as video games. They must manage vast volumes of data from diverse sources instantly and ensure user interactions are as fluid as possible.
- **Search Engines:** Search engines like Google demand low latency and highly optimised data structures and algorithms in order to deliver fast results to the user.

The above industries represent only a brief selection from a much more extensive array of applicable fields.



From a software engineering perspective, optimising an application for latency implies that the code in which the application is written has to be designed so that run-time overhead is minimised. Unfortunately, the conventional software engineering design patterns found in the literature (e.g. [2]) were created to maximise code maintainability and robustness instead of minimising execution latency. Even though some public knowledge about latency-optimising techniques exists, sourced primarily from presentations by domain experts (e.g. [3, 4, 5]), there are no formally catalogued design patterns that have a proven, statistically significant latency-lowering effect.

Furthermore, in software development, while there is a suite of tools for implementing conventional design patterns, none exist for low-latency design patterns due to their absence from the established literature.

## 1.2 Objectives

The objectives of this thesis are comprehensive:

1. Consolidate the aforementioned scattered pieces of information provided by the domain experts into a formal catalogue of low-latency software engineering design patterns.
2. Benchmark each pattern by comparing its performance to the original code snippet without the pattern, and then determine if there is a statistically significant difference in latency.
3. Develop an open-source application that analyses source code and detects opportunities in the code to implement the low-latency design patterns.
4. Extend the application so that after analysing the provided source code and generating refactoring suggestions, it can automatically refactor it by correctly implementing the appropriate design patterns.

It is paramount to emphasise that the design patterns catalogued in this report are strictly confined to C++. Consequently, the developed application is restricted to the analysis and the refactoring of C++ code. The rationale behind this decision is that C++ is the primary choice for developing latency-critical applications, as further elaborated on in Section 2.2.

### 1.2.1 Legal, Social, Ethical and Professional Requirements

There are several aspects to consider from ethical, social, legal, and professional standpoints. Ethically, cataloguing low-latency design patterns should consider proprietary rights, potentially seeking permissions where applicable. The application,

which automates code analysis and refactoring, might inadvertently introduce security vulnerabilities or biases in the refactored code, thereby bearing social implications. From a legal perspective, it is vital to ensure that the use of external libraries complies with licensing agreements. Professionally, adherence to software engineering best practices - such as thorough documentation and unit testing - is essential for ensuring the tool's reliability and usability.

## 1.3 HFT as an Illustrative Example

One of the most prominent industries associated with rapidly fast program execution times is High-Frequency Trading (HFT). Due to this prominence, most readers of this report will likely come from this domain and try to apply the dissertation's contributions within an HFT context. Further, most of the domain experts mentioned in Section 1.1 work in this industry, and thus their material and presented C++ snippets are tailored towards HFT applications. Because of those two reasons, HFT will serve as an illustrative example throughout the rest of this thesis.

### 1.3.1 A Short Introduction to HFT

HFT represents a highly competitive finance domain characterised by its zero-sum nature, where one participant's gain is inevitably another's loss. HFT is an umbrella term encompassing many trading strategies [6]. According to Aldridge's [6] exploration of the different definitions, the main characteristics of HFT are: high-speed trade execution, where trades are often executed in milliseconds or microseconds; high daily trading volume, which implies that positions are frequently entered and exited within seconds; sophisticated algorithms for decision-making, without human intervention; a firm reliance on cutting-edge technologies for data analysis and order execution; and the pursuit of profit opportunities arising from short-term market conditions. For the scope of this thesis, it is sufficient to define HFT as a form of automated trading whose ability to generate profits largely depends on swiftly responding to market changes and executing trades with minimal delay, that is, to operate the financial markets with extremely low latency.

From the characteristics outlined in the previous paragraph, it is easy to infer the paramount importance of speed in HFT. Given the zero-sum nature of trading, the quickest player often seizes the advantage, exploiting short-lived market opportunities before others can respond [7]. This underlines the adage for HFT: speed is not just beneficial; it is absolutely vital for success.

## 1.4 Outline

The rest of this report is structured as follows:

- **Preliminaries** (Chapter 2): The purpose of the next chapter is to equip the reader with the technical foundations for the ensuing chapters.
- **Design Patterns Catalogue** (Chapter 3): The subsequent chapter presents a detailed catalogue containing the low-latency software engineering design patterns. Each pattern, along with its applicability and implementation, is extensively discussed.
- **Benchmarking** (Chapter 4): This chapter presents the benchmarking results for each catalogued design pattern.
- **Software Application** (Chapter 5): Chapter 5 showcases the developed software application engineered for C++ code analysis and refactoring.
- **Epilogue** (Chapter 6): The final chapter assesses the fulfilment of the research objectives and identifies avenues for potential future research.

# Chapter 2

## Preliminaries

### 2.1 Design Patterns in Software Engineering

Establishing a concrete definition of what constitutes a software design pattern is important to provide a solid foundation for the upcoming chapters.

The period since 1985 has witnessed extraordinary advancements in hardware power, which have fueled a remarkable surge in the complexity of creating and maintaining software systems [8]. As a result of this growing complexity and the challenges that came with it, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides recognised the need to define and formalise coding patterns that address recurring problems in the continuously evolving domain of software development. Their book, "Design Patterns: Elements of Reusable Object-Oriented Software" (1994), marked a significant milestone in software engineering [9]. It provides a comprehensive catalogue of patterns to effectively manage complexity, enhance code quality, and foster collaboration among developers.

Gamma et al. [2] define a design pattern by stating, "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. [...] Each design pattern focuses on a particular object-oriented design problem or issue" (p. 3). Concretely, a design pattern consists of four essential elements: a pattern name, a context-specific design problem to be solved, an abstract solution given by the pattern, and trade-offs that arise using the pattern [2].

An example that illustrates the definition above is the so-called Singleton pattern. This design pattern offers a solution to enforce the constraint that only a single instance of a particular class can exist at any given time [2]. Assume we have a class called `BankAccount` and must enforce the mentioned constraint for security reasons. A concrete implementation based on the Singleton pattern could look as follows:

```
1 class BankAccount {
```

```
2 private:
3     BankAccount() {};
4     static BankAccount* instance;
5     static mutex mtx; // mutex for thread safety
6
7 public:
8     static BankAccount* getInstance() {
9         mtx.lock();
10        if (!instance) instance = new BankAccount();
11        mtx.unlock();
12        return instance;
13    }
14
15 };
16
17 //define static attribute
18 BankAccount* BankAccount::instance = nullptr;
19 mutex BankAccount::mtx;
```

**Code Snippet 2.1:** Singleton pattern in C++

By having a private constructor and a public `getInstance` method, the `BankAccount` class prevents any outside entities from creating more than one instance of it. A notable drawback of the Singleton pattern is that it complicates unit testing. Its global state can introduce dependencies between tests, making them less predictable and more challenging to manage.

## 2.2 Why C++

Given the utmost importance of speed in HFT, it is unsurprising that it serves as the sole criterion when selecting the programming language for the software side of the operation. This essential need for speed has led to the adoption of C++ as the de-facto industry standard, primarily due to some of the language's unique design choices, which offer *control over performance* unlike any other language [3]. Bjarne Stroustrup, the inventor of C++, states that "C++ was designed so that every language feature is usable in code under severe time and space constraints" [10, p. 30]. Specifically, the guiding maxim within the C++ language is the so-called zero-overhead principle ("what you don't use, you don't pay for") [11]. This principle has substantially contributed to the language's remarkable performance potential and manifests itself throughout most of its features [11].

One of the most prominent features of C++ related to the zero-overhead principle is the absence of a built-in garbage collector (GC) for managing dynamically allocated memory on the so-called heap [11]. The heap is a particular memory region used for dynamic allocation. Once the heap becomes excessively fragmented, a heap manager in languages with a GC might invoke it [12]. This action temporarily

interrupts the main program flow, allowing the GC to scan the heap and reclaim memory occupied by objects that are no longer in use [12]. Unsurprisingly, this non-deterministic interruption [12] can pose a significant risk for latency-sensitive applications. In C++, the absence of a GC implies that dynamic memory has to be managed manually, which introduces the risk of memory leaks, i.e. a situation where dynamically allocated memory is not deallocated correctly, preventing memory resources from being available for reuse [13]. Notably, despite the absence of a built-in garbage collector in C++, Stroustrup designed the language to allow for optional garbage collection if the programmer insists on it [11].

Another essential feature contributing to C++'s performance is that it is a compiled language, i.e., a compiler converts the source code into machine code before execution. This process allows for extensive optimisations during compilation, resulting in efficient executable code that can be run directly on the CPU [13]. C++ offers the most powerful compiler optimisation techniques available [14]. Related to compile-time optimisation, C++ provides the ability to perform metaprogramming within the language itself<sup>1</sup> using templates and constant expressions [13]. "Metaprogramming, [...] allows [...] to write code that transforms itself into regular C++ code" [13, p. 238], meaning the C++ compiler executes the metaprogram to generate C++ code, which it then translates into binary [13]. The following example [15] illustrates the idea:

```
1 #include <iostream>
2 using namespace std;
3 typedef unsigned int uint;
4
5 template<uint N>
6 class Factorial {
7 public:
8 //compiler evaluates constexpr and generates constant value
9     static constexpr uint val = N * Factorial<N - 1>::val;
10 };
11
12 template<>
13 class Factorial<0> {
14 public:
15     static constexpr uint val = 1;
16 };
17
18 int main() {
19     static_assert(Factorial<6>::val == 720,
20                 "Factorial is not correct.");
21     cout << Factorial<6>::val << endl;
22     return 0;
```

<sup>1</sup>Instead of using another language to generate C++ code [13].

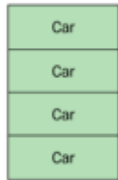
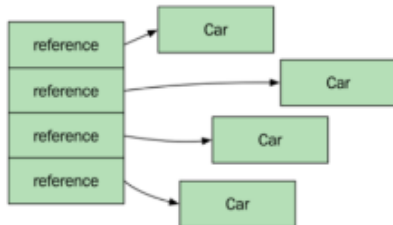
23 }

**Code Snippet 2.2: Metaprogramming Example**

The crucial point is that `Factorial<6>::val` is computed at compile time. Hence metaprogramming allows certain expressions to be evaluated during the compile phase and thus decreases run-time overhead [10], which is particularly advantageous for performance-critical applications. The concept exploits the fact that C++ templates are Turing-complete at compile time [10].

A third performance-enhancing aspect of C++ is its flexibility in object memory allocation. Unlike - for example - Java, which exclusively allocates objects on the heap, C++ grants the programmer the freedom to store objects either on the stack or the heap. Consequently, objects can be created without constantly allocating dynamic memory, which reduces computational overhead by avoiding the relatively expensive dynamic allocation. Moreover, C++ permits the creation of multiple objects within a singular heap allocation, improving efficiency further and allowing for related objects to be stored adjacently in memory. This spatial locality can significantly boost performance as related data is usually accessed simultaneously. [13]<sup>2</sup>

Andrist [13] provides an example to illustrate the second aspect of a single heap allocation in C++ compared to multiple heap allocations in Java:

C++	Java
<pre> auto n = 4; auto cars = std::vector&lt;Car&gt;{}; cars.reserve(n); for (auto i=0; i&lt;n;++i) {     cars.push_back(Car{2}); } </pre>	<pre> int n = 4; ArrayList&lt;Car&gt; cars =     new ArrayList&lt;Car&gt;(); for (int i=0; i&lt;n; i++) {     cars.addElement(new Car(2)); } </pre>
<p>The following diagram shows how the Car objects are laid out in memory in C++:</p> 	<p>The following diagram shows how the Car objects are laid out in memory in Java:</p> 

**Figure 2.1:** Heap Allocation in C++ vs. Java [13, p. 8].

As the code snippet depicted in Figure 2.1 shows, Java has to perform five heap allocations, whereas C++ only needs one. Moreover, when iterating over the Car-typed

<sup>2</sup>In the scope of this thesis, quoting one specific author *after* the period of the paragraph's final sentence indicates that the entire paragraph is a paraphrase based on this author.

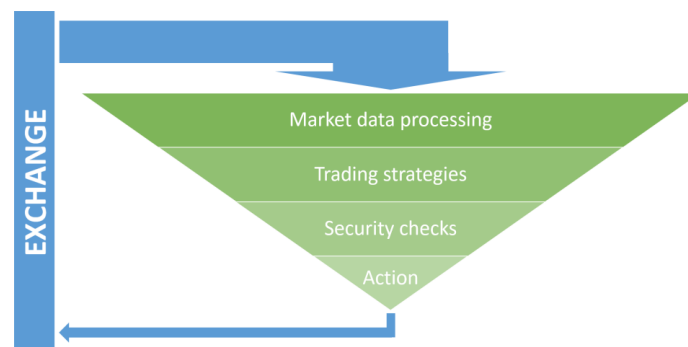
container, C++ is faster due to the contiguous storage of the Car objects on the heap<sup>3</sup>.

While C++'s unparalleled performance potential has cemented its position as the primary choice for latency-critical components in HFT systems, other languages also find their use cases in the industry. Python, offering superior user-friendliness and a broad suite of statistical and machine learning libraries like `pandas` and `scikit-learn`, is extensively used for data analysis and modelling tasks [16]. However, given the specific focus of this thesis on low latency design patterns, C++ will be the exclusive language utilised throughout.

## 2.3 The Fast Path

The 'Fast Path' (also called 'Hot Path' or 'Critical Path') refers to the code within an HFT codebase that facilitates the actual trading activity. Consequently, minimising latency is the primary concern for the fast path, and all optimisation efforts are made here. [5]

In particular, this code is responsible for (1) processing market data received from some financial exchange and (2) applying specific trading logic to the processed data to identify potential trading opportunities. If no opportunities are detected, the system swiftly reverts to stage (1) to listen again for new opportunities. Conversely, if a trading opportunity arises, the system proceeds to (3) execute some risk checks before (4) creating a trading order and sending it to the exchange. [17]



**Figure 2.2:** The Fast Path [17].

Figure 2.2 illustrates that the frequency of code execution diminishes with each subsequent stage of the fast path, with code related to stage (1) being executed the most frequently. According to Cook [18], the fast path represents a small fraction, typically between 2-5%, of the entire codebase. He further emphasises that stage (4) of the fast path is reached a mere 0.01% of the time.

<sup>3</sup>A C++ `std::vector` stores the objects in an array on the heap [12].

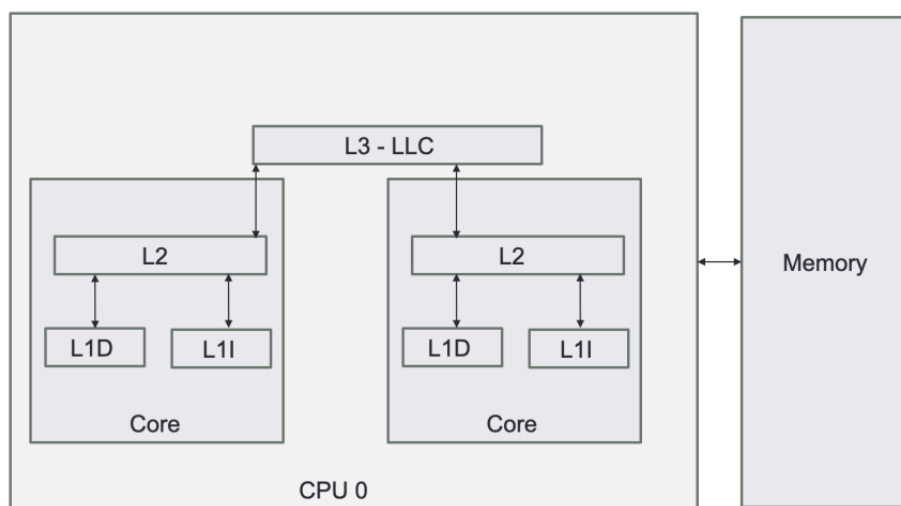


## 2.4 CPU Caches

### 2.4.1 The Basics Of Caching

One of the core principles of computing is that the further data (including processor instructions) are stored from the Central Processing Unit (CPU), the more time it requires to fetch them [19].

Fortunately, programs tend to work with data that resides within a specific memory region during a certain period of time. Modern CPUs exploit this spatial and temporal locality by utilising various levels of cache memory, allowing them to avoid the relatively slow data retrieval from RAM<sup>4</sup>. However, it is essential to note that cache memory is expensive, prompting a strategic cache hierarchy within the CPU consisting of Level 1 (L1), Level 2 (L2), and Level 3 (L3). Each successive level is placed further away from the CPU, translating into slower access but compensated by larger storage capacities. [16]



**Figure 2.3:** Cache System [16, p. 63].

Figure 2.3 depicts a typical cache structure for a modern-multicore CPU. The fastest memory tier, the L1 cache, is split into a data cache (d-cache) and an instruction cache (i-cache). This division allows for enhanced optimisation due to the distinct processing requirements for data and instructions. In contrast, the L2 and L3 caches, along with RAM, do not separate instructions from data. Moreover, L1 and L2 caches are core specific to each core in the CPU, whereas the L3 cache is shared among multiple cores. [16]

<sup>4</sup>Moreover, the rate of CPU speed advancements significantly outpaces that of data retrieval from RAM, thereby emphasising the critical importance of incorporating caches within the CPU [12].

The interactions of a typical modern CPU with its caches can be summarised as follows:

- The CPU leverages temporal locality by keeping recently used data - likely needed to be again soon - in the cache [20].
- Spatial locality implies that data stored close to recently used data will likely be accessed soon. Therefore, the spatial locality is exploited by loading not just a specific data item into the cache but also its adjacent data. [20]
- The two previous points are enhanced by the fact that the CPU speculatively preloads data into the cache, a process known as pre-fetching [20]. The cache level at which this data is stored is determined by the probability of the data being accessed next. For instance, the L1 cache holds data most likely to be needed next by the CPU [21].
- When a set of multiple cache storage units (a cache set) becomes fully occupied, cache management schemes such as the Least Recently Used (LRU) algorithm determine which specific unit in the set should be evicted to accommodate a new one [20].
- When a specific data item or instruction is needed for computation, the CPU first looks for it in the L1 cache. If the data is not found there - an event known as a cache miss, which is computationally expensive - the CPU then searches the L2 cache. Should the data remain missing, the CPU checks the L3 cache. If even the L3 cache does not contain the needed data, a significant slowdown occurs due to the time required to fetch the data from RAM. [21]

Figure 2.4 illustrates the size and access speed differences for L1, L2, and L3 caches in a typical computer system and compares them with other types of memory. It should be noted that these numbers can vary substantially depending on the system; nonetheless, their relative differences should be approximately of the same magnitude.

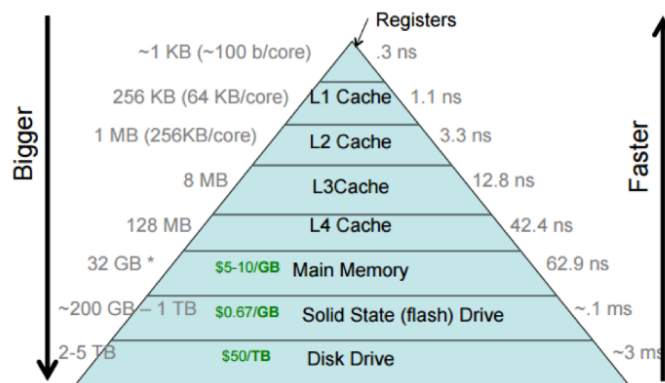


Figure 2.4: Different Cache Levels in Comparison [22].

### 2.4.2 Cache Structure

Given the disparity in size between a computer's RAM and the internal caches of a CPU, a mapping scheme is required to map the memory addresses from the larger RAM space to a specific cache location [23].

A straightforward solution to this issue is 'direct mapping': each RAM address is mapped to a single cache location using some mapping function  $h$  [23]. For instance, consider a scenario where the two least significant bits of an address are used as  $h$  [23]. Figure 2.5 depicts the issue that arises with direct-mapped caches:

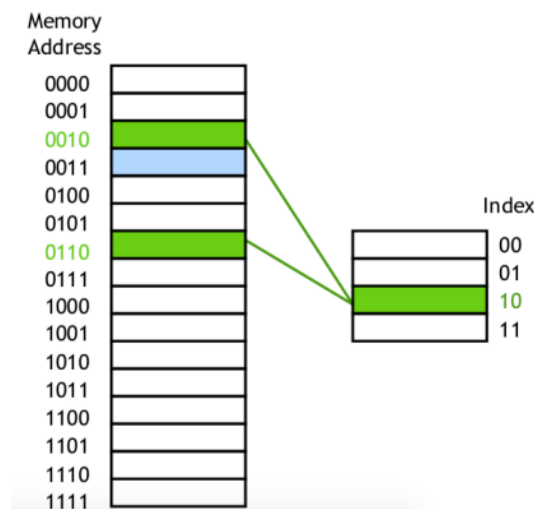


Figure 2.5: Direct Mapped Caches [23].

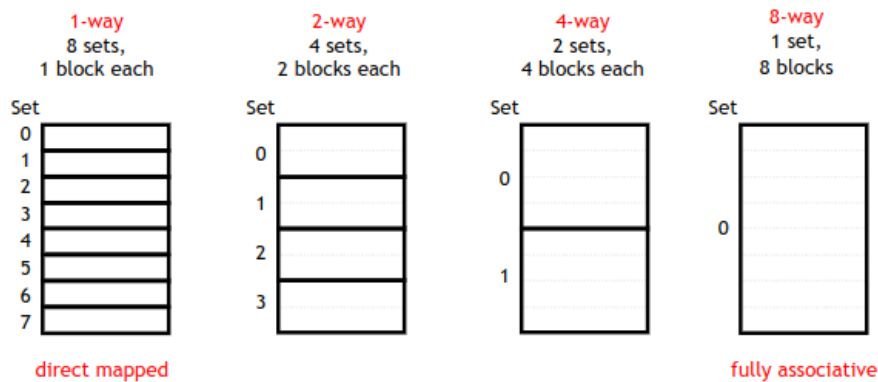
If a program persistently alternates between requesting data from RAM addresses  $0010_2$  and  $0110_2$ , each access attempt will feature a cache miss. This is because these two addresses are mapped to the same cache location. Consequently, they continuously evict each other's data from the cache, negating the benefits of caching. This situation parallels the phenomenon of collisions found in hash map operations. [23]

An alternative mapping scheme which avoids these 'cache conflicts' is fully associative mapping, where a given RAM address can be mapped to any location within the cache [23]. This method, however, implies searching the entire cache, which could substantially increase latency [20]. In order to avoid this slow-down, the cache is searched in parallel using a dedicated hardware component known as a comparator [20]. While effective, this solution significantly increases hardware costs [20].

Therefore, a hybrid scheme between direct and fully associative mapping, known as set associative mapping, is commonly used in practice. This approach strikes a compromise between flexible mapping and the need for manageable hardware costs. In set associative mapping, caches have a so-called  $n$ -way set-associative structure. [20]

Figure 2.6 depicts the differences between cache structures arising from various

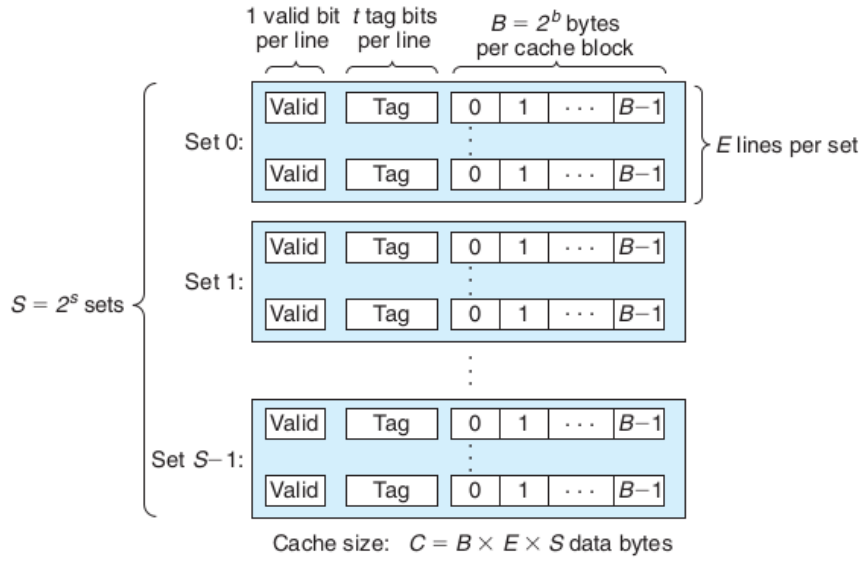
mapping schemes, assuming a cache with eight possible locations. The extreme cases - direct mapping and fully associative - are represented on the far left and far right, respectively. The second and third caches from the left showcase the structure arising from the hybrid approach. In a 2-way set-associative scheme, specific RAM addresses are deterministically mapped to one of four cache sets. However, within the designated set, the address can be mapped to either of two possible locations, subject to available space in the set. Meanwhile, the 4-way set-associative structure divides the cache into two sets for deterministic mapping, each providing four potential locations for data storage. [23]



**Figure 2.6:** Different Cache Structures in Comparison [23].

Previously, for the sake of simplicity, storage units in a CPU cache were referred to as 'locations'. However, more accurately, cache memory is divided into 'lines', with each line typically containing 64 bytes [12]. This design choice is rooted in the principle of spatial locality: Whenever the CPU accesses data from a specific RAM address, an entire block - slightly less<sup>5</sup> than the line size - of contiguous data around the targeted memory address is transferred from RAM to a specific cache line [23].

<sup>5</sup>because of valid bit and tag bits



**Figure 2.7:** General Cache Organization in an N-Way Set-Associative Cache [24, p. 597].

Bryant and O'Hallaron [24] define a cache's structure formally using a four-element tuple:  $(S, E, B, m)$ , as displayed in Figure 2.7. In this tuple:

- $S$  represents the number of sets in the cache,
- $E$  is the number of lines within each set,
- $B$  denotes the number of bytes in each line's data block, and
- $m$  corresponds to the length of the RAM address in bits.

Furthermore, each cache line includes one valid bit and  $t$  'tag' bits. The valid bit serves as an indicator of whether the line contains valid data. On the other hand, the tag bits identify the specific block of RAM data stored in the cache line. [24]

### 2.4.3 Critical Stride

The 'critical stride' is a phenomenon related to the n-way set-associative cache structure and can affect latency through cache misses [12]. For a more comprehensive understanding of this concept, it is helpful first to examine a realistic example of how data is written from RAM to the cache:

Suppose a cache size of 8KB, translating into 32 sets, each containing four 64-byte cache lines<sup>6</sup>. Further, assume a word-addressable RAM of  $16 \text{ bits} \times 2^{32}$  rows, i.e. 8GB RAM size. The formula<sup>7</sup> to map a memory address to a cache set is

$$(\text{set no.}) = (\text{memory address}) / (\text{line size}) \% (\text{no. of sets}).$$

<sup>6</sup>valid bit and tag bits are not included [24]

<sup>7</sup>/ refers to integer division; % means modulo; both operators have the same precedence

If the CPU reads or writes data from or to memory address  $a = 10000_{10}$ , it stores it to one of the four cache lines in set  $10000_{10}/64_{10} \% 32_{10} = 28_{10}$ . In adherence to the principle of spatial locality, the entire 64-byte block spanning from address  $9984_{10}$  to  $10015_{10}$  is loaded into the designated cache line within set 28, optimising future accesses to nearby memory locations. [12]

Now assume a scenario where a program operates on data stored at addresses  $12032_{10}$ ,  $14080_{10}$ ,  $16128_{10}$ , and  $18176_{10}$  after accessing RAM address  $10000_{10}$ . Given the mapping scheme from the previous example, all these addresses are mapped to one of the cache lines in set  $28_{10}$ . Under the premise of an LRU eviction policy, the data from address  $10000_{10}$  would be evicted from its cache line upon the program's demand for data from RAM address  $18176_{10}$  due to the limitation of having only four cache lines available. Consequently, a cache miss would occur if the data from address  $10000_{10}$  were to be reaccessed. The problem is only caused by the fact that all the addresses are  $2048_{10}$  (or a multiple of it) apart from one another, a phenomenon referred to as the 'critical stride'. The critical stride can be calculated as  $(critical\ stride) = (number\ of\ sets) \times (line\ size)$ . [12]

The critical stride can substantially impact program performance. Consider the L1 cache for the following: If a program has numerous variables dispersed throughout RAM, there is a risk that some of them are stored apart by a multiple of the critical stride, leading to unnecessary evictions in the d-cache. Similarly, if various functions used in the same program segment have their RAM addresses set apart by a multiple of the critical stride, a disproportionately high number of i-cache evictions will occur. Consequently, these excessive cache misses, caused by storing elements at multiples of the critical stride in memory, can severely degrade program execution speed. [12]

#### 2.4.4 Optimising Memory Access

Building on the discussion about caches outlined in the preceding three sections, Fog [12] provides several optimisation strategies to increase program execution speed:

- **Optimising D-Cache Usage:** Data used together should be stored close together in RAM. Therefore, variables and objects are ideally declared within the function they are used in, as this leads to their storage on the stack, which typically resides in the L1 cache. Consequently, minimising the use of global or static variables, along with dynamic memory allocation, is recommended whenever feasible.
- **Optimising I-Cache Usage:** Following the same logic as the previous point, functions that are called together should be stored close together in RAM. Despite the modular convenience of keeping functions in different source files, storing these modules contiguously in RAM can be advantageous if their functions are called from the same program part. This can be achieved by controlling the link order of the modules, typically managed by their appearance order in the project window or makefile. Furthermore, frequently used functions

should be separated from infrequently used ones and seldom used branches should be placed at the end of a function or in a separate function. The rationale is to fill the limited cache storage only with 'high-demand' code, i.e., the most frequently executed code.

- **Optimising Variable Access:** For efficient access, variables should be stored at memory addresses divisible by their size. This is because modern CPUs read memory in blocks rather than single bytes [25]. Hence, reading data from unaligned RAM addresses results in multiple read operations and extra overhead for data assembly [25]. For instance, when a CPU that reads 8-byte blocks encounters a 'double' data type (represented using 8 bytes) stored at address  $22_{10}$ , it must execute two read operations (one for data at addresses  $16_{10}$ - $23_{10}$  and another for data at  $24_{10}$ - $31_{10}$ ). If, however, the variable is stored at address  $24_{10}$ , a single operation (fetching data at addresses  $24_{10}$ - $31_{10}$ ) would suffice.
- **Access Contiguous Data Sequentially:** Data stored in a contiguous data structure, such as a C++ array, should ideally be accessed sequentially rather than randomly. This strategy takes advantage of the CPU's spatial locality principle, whereby adjacent data are also loaded into the cache when a specific data point is fetched, thus substantially decreasing the number of cache misses.

# Chapter 3

## Design Patterns Catalogue

*"Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*

Christopher Alexander [26, p. 10]

The subsequent sections of this chapter present the low-latency design patterns, adopting a structure motivated by the framework used by Gamma et al. [2]. Each pattern is examined under the following dimensions:

- **Pattern Name:** A name offers a unique identifier, giving each pattern its distinct abstraction.
- **Motivation (Problem):** This part outlines the specific problem the pattern aims to address within HFT environments.
- **Intent:** In the context of HFT, all actions aim to reduce latency. However, this point outlines the specific action undertaken to reduce latency.
- **Applicability:** In which situations and circumstances is this pattern applicable?
- **Implementation:** A concrete example of the pattern's implementation is provided in this segment.
- **Consequences:** This part delves into the advantages and drawbacks of implementing the pattern.

The rationale behind presenting the design patterns using this formal framework is to convert the informal presentation of techniques provided by the domain experts into a structured and rigorous catalogue. Consequently, this catalogue can serve as a proper literature reference for future practitioners, empowering them to apply these design patterns within their specific low-latency domains.



## 3.1 Cold Code Isolation Pattern

The pattern discussed in this section is derived from the presentations held by Mariusz Pusz and Carl Cook, as well as Jonathan Müller [17, 5, 27]. The objectives of this pattern align with the i-cache-optimisation strategy outlined in Section 2.4.4, affirming the insights provided by the experts.

- **Pattern Name:** Cold Code Isolation Pattern
- **Motivation (Problem):** One critical insight from Section 2.4.4 is that frequently executed code<sup>1</sup> should be isolated from infrequently executed code<sup>2</sup> to improve i-cache utilisation. Consequently, avoiding the 'pollution' of a cache line with cold code (a situation known as cache line fragmentation [28]) results in a reduction in the number of necessary cache reads to access the same amount of hot code. More importantly, the prudent use of cache space reduces the risk of future cache misses, as a larger portion of the hot code can be retained in the i-cache. Therefore, in HFT, the challenge is to design fast path code in such a manner that cold code is effectively separated from hot code.
- **Intent:** The intent behind this pattern is to isolate cold code to avoid cache line fragmentation. This ensures that the i-cache predominantly retains hot code, lowering latency by decreasing the number of expected i-cache misses.
- **Applicability:** This pattern can be applied to any isolatable cold code within the fast path.
- **Implementation:** Cook provides an example illustrating a scenario where some error checks are conducted before an order is sent to the exchange:

```
1 // hot code before ...
2 if (checkForErrorA()) // cold code starts
3     //medium-sized code block
4 else if (checkForErrorB())
5     //medium-sized code block
6 else if (checkForErrorC())
7     //medium-sized code block
8 else
9     sendOrderToExchange(); // hot code again
```

**Code Snippet 3.1:** Cold Code Isolation Pattern (Problem 1)

Since exceptions occur infrequently, any code associated with exception handling can be classified as cold code. The issue with the order of the code blocks is the resultant line fragmentation in the i-cache, which should be avoided for the reasons explained earlier. Encapsulating the entire cold code within

<sup>1</sup>referred to as hot code

<sup>2</sup>defined as cold code

a single function and thus removing it refines the code to be more i-cache friendly<sup>3</sup>:

```

1 int errorFlags;
2 //...
3 if (!errorFlags) //hot code continuous until here
4     sendOrderToExchange();
5 else
6     HandleError(errorFlags);

```

**Code Snippet 3.2:** Cold Code Isolation Pattern (Solution 1)

Pusz provides another example which illustrates how the pattern can be implemented in a class:

```

1 class vector_downward {
2     uint8_t *make_space(size_t len) {
3         //The entire then-block is cold code
4         if (len > static_cast<size_t>(cur_ - buf_)) {
5             auto old_size = size();
6             //...
7             //lots of code
8             //...
9             allocator_.deallocate(buf_);
10            buf_ = new_buf;
11        }
12        cur_ -= len; //hot code begins
13        assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
14        return cur_;
15    }
16    // ...
17 };

```

**Code Snippet 3.3:** Cold Code Isolation Pattern (Problem 2)

By removing the cold code after the if-statement from the `make_space()`-method (which is frequently executed according to Pusz), the entire method becomes i-cache-optimised:

```

1 class vector_downward {
2     uint8_t *make_space(size_t len) {
3         if (len > static_cast<size_t>(cur_ - buf_))
4             reallocate(len);
5         cur_ -= len;
6         assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
7         return cur_;
8     }
9 };

```

<sup>3</sup>Cook states that this method typically reduces latency by 100-200 nanoseconds.

```
8     }
9
10    void reallocate(size_t len) {
11        auto old_size = size();
12        //....
13        buf_ = new_buf;
14    }
15    // ...
16};
```

**Code Snippet 3.4:** Cold Code Isolation Pattern (Solution 2)

Additionally, Pusz explains that the `make_space()`-method's notable size reduction led to a 20% performance improvement in other areas of his code. This enhancement was directly attributed to the compiler's improved ability to inline the method when generating the executable file.

The third and final example from Müller illustrates that cold code isolation does not always involve its removal. In the following function, a loop iterates through a vector of random numbers, accumulating their sum. However, there is one edge case (the cold code): if a vector element equals 0, then some large code block is executed:

```
1 int foo(const std::vector<unsigned>& data)
2 {
3     unsigned count = 0;
4     for (const auto& d: data)
5     {
6         count += d;
7         if (d == 0)
8         {
9             // block of 64 lines - not outsourceable
10            // into a separate function
11        }
12    }
13    return count;
14 }
```

**Code Snippet 3.5:** Cold Code Isolation Pattern (Problem 3)

The presence of a 64-line block of cold code between the update of the `count` variable (hot code) and the end of the loop (also hot code) introduces cache line fragmentation. A simple reordering of the code proved to be a solution for this issue, resulting in a significant speed-up in Müller's benchmarking:

```
1 int foo(const std::vector<unsigned>& data)
2 {
3     unsigned count = 0;
4     for (const auto& d: data)
5     {
6         if (d == 0)
7         {
8             // block of 64 lines - not outsourceable
9             // into a separate function
10        }
11        count += d;
12    }
13    return count;
14 }
```

**Code Snippet 3.6:** Cold Code Isolation Pattern (Solution 3)

- **Consequences:** This pattern provides a seemingly easy solution to reduce latency. However, one potential complication that may arise during its implementation is the challenge of accurately identifying and effectively isolating the cold code, a task which may prove to be less straightforward than in the depicted examples.

## 3.2 DOD Pattern

The pattern in this section is inspired by Mike Acton's insights, who popularised the data-oriented design philosophy [29]. The Data-Oriented Design (DOD) Pattern is closely related to the Cold Code Isolation Pattern discussed earlier and aligns with the d-cache optimisation principles in Section 2.4.4.

- **Pattern Name:** The Data-Oriented Design Pattern
- **Motivation (Problem):** One of the fundamental object-oriented (OO) design principles is encapsulation: the data members of an object are encapsulated together with the functions that operate on the data into one entity. However, this implies that more frequently accessed attributes (hot data) are mixed with less frequently used ones (cold data). The motivation for avoiding the resulting cache line fragmentation, in this case within the d-cache, was thoroughly explained in Section 3.1. Consequently, a unique challenge that low-latency programmers face is to design classes such that hot data is successfully isolated from cold data.
- **Intent:** This pattern isolates frequently accessed data members of a class to reduce d-cache misses.
- **Applicability:** The DOD pattern is suitable for classes where data items of a particular hot attribute are accessed sequentially. Specifically, let A denote a hot

attribute of class Foo. If  $\{\text{foo.i}\}_{i=1}^n$  represents  $n$  instances of this class, then iterative access implies accessing the A attribute for each of the  $n$  instances sequentially.

- **Implementation:** The following example is based on Pusz's work [17], illustrating a class responsible for analysing the risk of open trading positions:

```

1 class RiskAnaylzer {
2     struct position {
3         float pnl; // hot data
4         float percExposure; // hot data
5         string errortxt_1; // cold data
6         string errortxt_2; // cold data
7         array<char, 100> otherData; // cold data
8     };
9
10    vector<position> openPositions;
11
12 public:
13     RiskAnaylzer(int size):
14         openPositions{size} {}
15
16     int openPositions() {return openPositions.size()};
17
18     float totalPnl(){ // not d-cache-friendly
19         float total_pnl = 0;
20         for(auto p: openPositions) total_pnl += p.pnl;
21         return total_pnl;
22     }
23
24     float totalPercExposure(){ //not d-cache-friendly
25         float total_pe = 0;
26         for(auto p: openPositions)
27             total_pe += p.percExposure;
28         return total_pe;
29     }
30
31     position getPositionData(int idx) {
32         return openPositions[idx]};
33
34     //more methods to add and remove open posiions...
35 };

```

**Code Snippet 3.7:** DOD Pattern (Problem)

Due to frequent and iterative access of the pnl attribute and the percExposure attribute of all open positions, encapsulating those attributes together with other member variables in a single struct leads to poor d-cache utilisation. By

extracting the hot data from the struct and putting the remaining cold data into a vector which can be accessed using an index for each position, the code becomes much more d-cache-friendly:

```

1 class RiskAnaylzer {
2     vector<float> pnls;
3     vector<float> riskExposures;
4     struct otherPositionData{
5         string errortxt_1;
6         //rest of the cold data
7     };
8
9     vector<otherPositionData> coldData;
10
11 public:
12     RiskAnaylzer(int size): coldData{size} {}
13
14     int openPositions() {return coldData.size()};
15
16     float totalPnl() //d-cache-friendly
17     { /* sum over pnls-vector */ }
18
19     float totalPercExposure() //d-cache-friendly
20     { /* sum over riskExposures-vector */ }
21
22     otherPositionData getColdData(int idx) {
23         return coldData[idx];
24     }
25     //more methods to add and remove open posiions...
26 };

```

Code Snippet 3.8: DOD Pattern (Solution)

- **Consequences:** Implementing a DOD pattern may offer only limited benefits as most advantages of OO design get sacrificed. In particular, the intentional omission of encapsulation can lead to code that is harder to maintain, understand, and modify.

### 3.3 Denormalised Data Pattern

The following pattern, along with the content in this section, is based on the insights and work of Carl Cook [18]:

- **Pattern Name:** Denormalised Data Pattern
- **Motivation (Problem):** In software engineering, data normalisation is recognised as a best practice to prevent redundancy. However, this approach in-

herently necessitates lookup operations whenever particular data is to be retrieved. In an HFT context, such operations can be costly since lookups typically involve fetching data from RAM rather than cache memory.

- **Intent:** The denormalised data pattern aims to eliminate expensive data lookups by intentionally incorporating redundant information as attributes within classes, assuring that the needed data is readily available in the cache whenever an object requires it.
- **Applicability:** This pattern can be implemented across all fast path classes executing data lookup operations.
- **Implementation:** The subsequent code snippet depicts a typical set of classes (structs are used for brevity purposes) involved in creating and sending a trading order to some financial exchange:

```
1 struct Market {
2     int ID;
3     char shortName[4];
4     int quantityMultiplier;
5     //...
6 };
7
8 struct Asset {
9     float price;
10    //...
11    int marketID;
12 };
13
14 // ----- Order Creation -----
15 //...
16 Message orderMessage;
17 orderMessage.price = asset.price;
18 // lookup for quantityMultiplier
19 Market& market = Markets.FindMarket(asset.marketID);
20 orderMessage.qty = market.quantityMultiplier * qty;
21 //...
```

**Code Snippet 3.9: Denormalised Data Pattern (Problem)**

Cook argues that if a trader consistently needs to retrieve the quantity multiplier for the asset from the market - a piece of information that infrequently changes - then this data should be embedded within the instrument itself, i.e.

```
1 struct Market {
2     int ID;
3     char shortName[4];
4     int quantityMultiplier;
```

```
5 //...
6 };
7
8 struct Asset {
9     float price;
10    int quantityMultiplier;
11    //...
12    int marketID;
13};
```

**Code Snippet 3.10:** Denormalised Data Pattern (Solution)

which would avoid the lookup operation in line 9 of Code Snippet 3.9.

- **Consequences:** Although this pattern avoids costly lookup operations and thus reduces the number of d-cache misses, one potential downside is that denormalised data could result in larger, more complex code.

## 3.4 Cache Priming Pattern

This section's pattern is based on a method introduced by Carl Cook and Jonathan Keinan [18, 30]. Cook attests to the significant efficacy of this technique, citing it as one of the most effective strategies for reducing latency.

- **Pattern Name:** Cache Priming Pattern
- **Motivation (Problem):** As described in Section 2.3, stage (4) of the hot path - the creation of a trading order and its transmission to the exchange - is executed only 0.01% of the time. Consequently, the infrequent execution of this stage results in the relevant data and instructions being absent from the cache when they are actually needed, i.e. a cache miss occurs.
- **Intent:** The goal of this pattern is to artificially maintain crucial trading data and code within the cache. By doing so, a cache miss is circumvented when a trade is actually executed.
- **Applicability:** This pattern ought to be implemented when designing the execution-related part of the hot path.
- **Implementation:** Keinan provides the following example, illustrating the problem before the pattern is implemented:

```
1
2 void buy_stocks(){
3     const int available = fetch_available_amount();
4     if (available < 10) return;
5     const int buy_amount = min(100, available);
6     const int bought_amount = send_order(buy_amount);
```



```
7     logger(bought_amount); // assume global variable
8 }
9
10 void run(){
11     while (true)
12     {
13         float stock_price = get_stock_price();
14         //frequently executed
15         if (!should_buy(stock_price))
16             update_stock_price_history(stock_price);
17         else //rarely executed
18             buy_stocks();
19     }
20 }
```

**Code Snippet 3.11:** Cache Priminig Pattern (Problem)

In order to keep the cache primed for the execution of a genuine trading order, the `buy_stocks()`-function must be routinely called, but without facilitating an actual trade unless explicitly intended. This can be accomplished by incorporating a boolean argument into the function and adjusting the code as follows:

```
1
2 void buy_stocks(bool actual_trade){
3     const int available = fetch_available_amount();
4     if (available < 10) return;
5     const int buy_amount = min(100, available);
6     const int bought_amount = send_order(buy_amount,
7                                         actual_trade);
8     logger(bought_amount, actual_trade);
9 }
10
11 void run(){
12     while (true)
13     {
14         float stock_price = get_stock_price();
15         if (!should_buy(stock_price)){
16             update_stock_price_history(stock_price);
17             buy_stocks(false); // cache priming
18         }
19         else
20             buy_stocks(true); //actual trade execution
21     }
22 }
```

**Code Snippet 3.12:** Cache Priminig Pattern (Solution)

In order to safeguard against unintended modifications to the program state, it is crucial to forward the boolean variable `actual_trade` to other functions within the `buy_stocks` routine that may alter it.

- **Consequences:** While there are notable advantages, implementing this pattern may entail certain drawbacks. These include increased code complexity and substantial efforts to redesign the code such that the program state remains unaffected during cache priming.

## 3.5 TB Branch Elimination Pattern

The design pattern presented in this section is inspired by the contributions of Carl Cook, Mateusz Pusz, and Nimrod Sapir [5, 17, 4].

- **Pattern Name:** Template-Based (TB) Branch Elimination Pattern
- **Motivation (Problem):** Often, a function's specific behaviour is determined by the value of one or more of its input parameters. A straightforward approach involves writing the function such that its specific behaviour is determined at run-time using if-statements. However, this implies computational costs at run-time<sup>4</sup>. Moreover, even though modern compilers can optimise away unused branches in these situations (i.e. cold code removal), there is no guarantee they will do so in every instance<sup>5</sup>.
- **Intent:** The purpose of this pattern is to utilise templates to eliminate compile-time-determinable branches. Shifting the decision-making process to compile-time eliminates unnecessary computational overhead at run-time and the dependency on compiler optimisations.
- **Applicability:** This pattern is appropriate for performance-critical functions or methods where behaviour is determined by an input argument's value using if-statements, assuming the argument's value is known at compile time.
- **Implementation:** The following example is based on Cook's materials. Assume the task is to write code that facilitates the execution of trading orders. A straightforward design approach would be to combine buy-order and sell-order logic into one function and then determine the function's behaviour based on the type of the order using an if-statement:

```
1
2 enum class Side {Buy, Sell};
3
4 void ExecutionEngine::execute(Side side){
5     float price = calcPrice(side, fairValue, credit);
6     checkRiskLimits(side, price);
}
```

<sup>4</sup>Even if the branches are easy to predict for the CPU's branch predictor

<sup>5</sup>Additionally, the programmer would need to consistently review the generated assembly to verify.

```

7     sendOrder(side, price);
8 }
9
10 float ExecutionEngine::calcPrice(Side side, float value,
11     float credit){
12     //if-statement evaluated at run time!
13     if (side == Side::Buy) return value - credit;
14     else return value + credit;
15 }
16 void ExecutionEngine::checkRiskLimits(Side side, float
17     price){
18     // some logic + if-statement
19 }
20 void ExecutionEngine::sendOrder(Side side, float price){
21     // some logic + if-statement
22 }

```

**Code Snippet 3.13:** Template-Based Branch Elimination (Problem)

However, since the type of order is known at compile time, it is possible to shift the decision-making process regarding a function's specific operation from run-time to compile time. Thus, latency-hostile if-statements can be avoided:

```

1 template<Side T>
2 void ExecutionEngine<T>::execute(){
3     float orderPrice = calcPrice(fairValue, credit);
4     checkRiskLimits(orderPrice);
5     sendOrder(orderPrice);
6 }
7
8 template<>
9 float ExecutionEngine<Side::Buy>::calcPrice(float value,
10     float credit){
11     return value - credit;
12 }
13
14 template<>
15 float ExecutionEngine<Side::Sell>::calcPrice(float value
16     , float credit){
17     return value + credit;
18 }
19
20 //analogous approach for the other methods...

```

**Code Snippet 3.14:** Template-Based Branch Elimination (Solution)

- **Consequences:** The main disadvantages of employing this method include the

potential for code bloat, where the code generated by the compiler may substantially increase in size, resulting in prolonged compilation times and high memory usage. Furthermore, it often yields source code that is exceedingly complex and difficult to read. [16]

## 3.6 CRTP

The Curiously Recurring Template Pattern (CRTP) is a well-established design pattern. While already known, it is presented in this catalogue due to its prevalence in HFT and the significance of its benchmarking results. Cook, Pusz, and Sapir advocate using this pattern for substituting run-time (dynamic) with compile-time (static) polymorphism [5, 17, 4].

- **Pattern Name:** Curiously Recurring Template Pattern
- **Motivation (Problem):** Method overriding<sup>6</sup> - a form of polymorphism<sup>7</sup> - is typically achieved in C++ using inheritance and virtual functions [31]. Virtual functions are particularly attractive since their binding to the correct function implementation is done dynamically when called through a pointer (or reference), allowing for the flexibility of using parent-class-typed pointers to reference child-class-typed objects [31]:

```
1 class Parent {
2     public:
3         virtual void foo() {cout << "Parent\n";}
4 };
5
6 class Child: public Parent {
7     public:
8         virtual void foo() override {cout << "Child\n";}
9 };
10
11 // main()
12 Parent* ParPtr = new Child();
13 ParPtr->foo(); // dynamically dispatched
```

**Code Snippet 3.15:** Dynamic Function Dispatch

Concretely, `ParPtr->foo()` is bound correctly to `Child::foo()` using the type of the object pointed to instead of the pointer's type [32]. However, this process can only be done at run-time since the object's type is unknown to the compiler at compile time [32]. Consequently, this so-called dynamic polymorphism comes with a run-time overhead in the form of vtable lookups [31] and hence, should be avoided in latency-critical environments.

<sup>6</sup>A child class re-implements an inherited method without changing the method's signature [31].

<sup>7</sup>Functions with different behaviour and/or different input arguments share the same name [31].

- **Intent:** The CRTP intends to replace dynamic with static polymorphism and thus avoids run-time overhead caused by dynamic function dispatch.
- **Applicability:** The CRTP is applicable to class hierarchies on the fast path that utilise dynamic polymorphism.
- **Implementation:** The following example is inspired by Sapir's materials [4]. Imagine a class hierarchy for trading orders where some methods are shared while others require a class-specific implementation. This can be modelled using the usual approach, i.e. relying on inheritance and virtual functions:

```
1 class Order {
2 private:
3     virtual void create_order()
4         { /*Special Implementation*/ }
5     virtual void check_order()
6         { /*Special Implementation*/ }
7     void send_order()
8         { /*Shared Implementation*/ }
9 public:
10    void execute_order() {
11        create_order();
12        check_order();
13        send_order();
14    }
15 };
16
17 class SpecialOrder: public Order {
18 private:
19     void create_order() override
20         { /*Special Implementation*/ }
21     void check_order() override
22         { /*Special Implementation*/ }
23 };
```

**Code Snippet 3.16:** CRTP (Problem)

However, the calls to the private virtual functions within `execute_order()` are always dynamically dispatched<sup>8</sup> since they are implicitly made using the `this` pointer<sup>9</sup>. Because the type of the object that `this` points to is unknown at compile time, its resolution occurs at run-time. The CRTP allows for static polymorphism, i.e. different behaviour for `execute_order()` based on the object type it is invoked on while avoiding the run-time overhead of dynamic function dispatch:

<sup>8</sup>even if `execute_order()` is not called through pointer or reference

<sup>9</sup>i.e., `this->create_order()` and `this->check_order()`

```

1 template<class derived>
2 class OrderTemplate {
3 private:
4     void send_order() { /*Shared Implementation*/ }
5 public:
6     void execute_order() {
7         //correct implementation known at compile time
8         static_cast<derived*>(this)->create_order();
9         static_cast<derived*>(this)->check_order();
10        send_order();
11    }
12 };
13
14 class Order: public OrderTemplate<Order>{
15 private:
16     friend class OrderTemplate<Order>;
17     void create_order() { /*Special Implementation*/ }
18     void check_order() { /*Special Implementation*/ }
19 };
20
21 class SpecialOrder: public OrderTemplate<SpecialOrder>{
22 private:
23     friend class OrderTemplate<SpecialOrder>;
24     void create_order() { /*Special Implementation*/ }
25     void check_order() { /*Special Implementation*/ }
26 };

```

Code Snippet 3.17: CRTP (Solution)

As shown in the previous code snippet, `Order` and `SpecialOrder` are passed as template parameters to `OrderTemplate`, which serves as the blueprint class containing the methods with shared implementation. ‘Injecting’ the class-specific methods of `Order` and `SpecialOrder` into `OrderTemplate` at compile time using templates allows for their static dispatch.

- **Consequences:** A crucial drawback of the CRTP is that it does not offer the same flexibility as virtual methods, i.e. using parent-class-typed pointers to reference child-class-typed objects. Additionally, the CRTP tends to increase code complexity, making the code more challenging to understand and maintain.

## 3.7 Miscellaneous

This section compiles a variety of latency-lowering techniques, each insufficient to form a distinct design pattern but nonetheless of relevance for any software development in the low-latency realm.

- **Expression Short Circuiting:** Cook [33] recommends to short-circuit expressions whenever possible:

```

1 //rewrite
2 if (expensiveCheck() && inexpensiveCheck())
3 //as
4 if (inexpensiveCheck() && expensiveCheck())

```

Code Snippet 3.18: Expression Short Circuiting

- **Memory Preallocation:** Preallocate memory if possible to avoid frequent, costly reallocation operations and memory fragmentation<sup>10</sup> [3]:

```

1
2 std::vector<X> f(int n){
3     std::vector<X> result; //suboptimal
4     for (int i = 0; i < n; ++i)
5         result.push_back(X(_));
6     return result;
7 }
8
9 //preallocate memory instead before looping
10 std::vector<X> f(int n){
11     std::vector<X> result;
12     //optimal (requires a priori knowledge)
13     result.reserve(n);
14     for (int i = 0; i < n; ++i)
15         result.push_back(X(_));
16     return result;
17 }

```

Code Snippet 3.19: Memory Preallocation

- **Avoiding Redundant Micro-Computations:** Carruth [3] asserts that while redundant micro-computations may not surface in a profile, they can still induce slowdowns if scattered across the entire code base:

```

1
2 //suboptimal: the hash of the key (cache[key]) is
3 //computed over and over again
4 X* getX(string key, unordered_map<string,
5         unique_ptr<X>>& cache)
6 {
7     if (cache[key]) return cache[key].get()
8     cache[key] = make_unique<X>(...);
9     return cache[key].get();

```

<sup>10</sup>`std::vector` is a container that holds a pointer to an array dynamically allocated on the heap [12].

```
9 }  
10  
11 //optimal  
12 X* getX(string key, unordered_map<string,  
13         unique_ptr<X>>& cache)  
14 {  
15     unique_ptr<X>& entry = cache[key];  
16     if (entry) return entry.get()  
17     entry = make_unique<X>(...);  
18     return entry.get();  
19 }
```

**Code Snippet 3.20:** Avoiding Redundant Micro-Computations

- **Contiguous Data Structures:** In light of the spatial locality principle, it is highly advisable to favour contiguous data structures such as `std::vector` over non-contiguous ones like `std::list` [3, 33].
- **Avoid Mixing float and double:** Fog recommends to avoid mixing float and double data types to circumvent the run-time overhead caused by type conversions [12, p. 153]:

```
1 //suboptimal: requires b to be casted to double and  
2 //converting the result back to float  
3 float b = 3.4f;  
4 float a = b * 1.5;  
5  
6 //optimal  
7 float c = b * 1.5f;
```

**Code Snippet 3.21:** Avoid Mixing float and double

- **Cache-Friendly Matrix Iteration:** In C++, the row-major layout is used for 2D arrays, i.e. the elements of the same row are stored contiguously in memory. Therefore, iterating over a 2D matrix should be done row-wise<sup>11</sup> to capitalize on data locality. [34]

<sup>11</sup>iterating over the elements in one row before proceeding to the next row



# Chapter 4

## Benchmarking

### 4.1 Foreword

#### 4.1.1 General Aspects

Deriving reliable and reproducible performance metrics for the low-latency design patterns presented in the previous chapter is a nontrivial task that requires careful consideration of numerous factors.

Firstly, one must account for a computer's inherent non-determinism that can impact the benchmarking results. This randomness, stemming from things like Operating System behaviour or hardware-related aspects, can result in inconsistent benchmarking outcomes across multiple trials. Consequently, in order to derive statistically sound inference from the benchmarking, it is necessary to (1) run a single benchmarking experiment multiple times, (2) derive an empirical distribution<sup>1</sup> for a pattern-variant as well as a non-pattern variant of the same code and then (3) utilise appropriate statistical tests, such as a permutation test, to compare these distributions.

Secondly, choosing a suitable benchmarking library is crucial. Gathering performance metrics for the design patterns equates to so-called micro-benchmarking: only small code parts of a program are relevant for evaluation. Therefore, any benchmarking library must have a low computational overhead to avoid distorting the micro-benchmarking results. In terms of performance metrics, the employed framework should be able to gather information from the CPU's performance counters, which allows for a precise analysis of a pattern's hypothesised effect; e.g. does the Cold Code Isolation Pattern indeed improve latency<sup>2</sup> by lowering the number of i-cache misses?

Several benchmarking frameworks for C++ exist. However, not all of them satisfy the aforementioned prerequisites. For example, after attempting to reproduce

<sup>1</sup>of the metric of interest

<sup>2</sup>Gross recommends that code performance should always be measured in CPU clock cycles instead of time as it is more accurate and avoids the overhead associated with a clock [35].

Jonathan Müller’s [27] micro-benchmarking results and engaging in a discussion with him when unable to do so, it was determined that Google’s renowned benchmark library carries significant overhead. This overhead is substantial enough that just the re-ordering of two tested functions can result in contradictory benchmarking outcomes. Additionally, benchmark does not support retrieving data from the CPU’s performance counters. Conversely, Linux’ perf, another popular benchmarking tool, does provide data from the CPU’s performance registers, but Gross [35] argues that it also introduces excessive overhead. According to Gross, the papi framework<sup>3</sup> is the most suited library, meeting all necessary conditions.

Thirdly, before any benchmarking experiments and attempts at drawing conclusions, it is crucial to note that the exact benchmarking results are largely contingent upon the system architecture used to execute the tests. Nevertheless, results acquired with appropriate statistical rigour should still hold significant insights. They should provide a reliable basis for determining the relative performance between a pattern-variant and a non-pattern-variant of the same code. Even if the absolute numbers differ from system to system, the relative differences should be reasonably consistent, granting a solid foundation for making decisions about code optimisation. Bearing this in mind, the hardware features of the machine used for the tests in the upcoming sections are:

Parameter	Value
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	39 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	8 (logical)
On-line CPU(s) list	0-7
Vendor ID	GenuineIntel
Model name	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
CPU family	6
Model	142
Thread(s) per core	2 (4 phys. process. units $\times$ 2 threads $\rightarrow$ 8 logical CPUs)
Core(s) per socket	4 (i.e. 4 phys. process. units within 1 multi-core CPU)
Socket(s)	1
Stepping	12
CPU max MHz	4600.0000
CPU min MHz	400.0000
L1d cache	128 KiB (4 inst.) $\rightarrow$ 32KB = 512 cache lines $\times$ 64 byte (1 inst.)
L1i cache	128 KiB (4 inst.) $\rightarrow$ 32KB = 512 cache lines $\times$ 64 byte (1 inst.)
L2 cache	1 MiB (4 inst.) $\rightarrow$ 256KB = 4096 cache lines $\times$ 64 byte (1 inst.)
L3 cache	8 MiB (1 inst.)

**Table 4.1:** System Information

A fourth and final aspect to consider is the importance of compiling all programs

<sup>3</sup>Performance Application Programming Interface ([link](#))

containing benchmarking tests without any compiler-generated optimisations to ensure credible results. Hence, all of the following tests were compiled using the g++ compiler (version 11.3.0-1ubuntu1 22.04.1) with the `-O0` flag set, disabling all optimisations. While this approach is appropriate for evaluating the design patterns in isolation, it is worth noting that compiler optimisations play a significant role in improving latency of real-world software applications. Thus, the findings and conclusions drawn from these isolated tests should be interpreted within the context of their specific benchmarking conditions and may not directly translate to real-world software engineering environments where compiler optimisations are typically employed.

### 4.1.2 Statistical Foundations

The notation used throughout the subsequent sections is defined as follows:

- $H_0$  denotes the null hypothesis, whereas  $H_A$  denotes the alternative hypothesis.
- $\mu_X$  represents the expected value of random variable  $X$ .
- $\alpha$  denotes the significance level of the statistical test, which can be interpreted as the probability of rejecting  $H_0$  even though it is true (called a ‘Type I error’ or a ‘false positive’). A p-value, on the other hand, can be interpreted as the probability of obtaining the observed outcome (or a more extreme value) under  $H_0$ .

A permutation test is a non-parametric statistical test that allows for testing the difference between the same statistic (e.g. mean) from two different data groups (e.g. trial vs. control group). Permutation tests are appealing as they are simple and do not rely on any parametric assumptions about the underlying data-generating mechanism. Instead, the only required assumption is that the data points are exchangeable between the groups under  $H_0$ . [36]

For instance, let  $X$  and  $Y$  be random variables representing the observations from a trial and control group, respectively. Consider the hypotheses:

$$H_0 : \mu_X = \mu_Y \quad vs. \quad H_A : \mu_X > \mu_Y.$$

Then a permutation test would be conducted as follows:

```

 $\theta \leftarrow$  Estimate  $\mu_X - \mu_Y$  using the empirical means of the unpermuted data;
 $V \leftarrow$  initialise empty array of size NumPerm;
for  $i \leftarrow 1$  to NumPerm do
    | DataPermuted  $\leftarrow$  randomly exchange observations between groups;
    |  $\tilde{\theta}_i \leftarrow$  recompute the statistic using DataPermuted;
    |  $V[i] \leftarrow \tilde{\theta}_i$ ;
end
p-value  $\leftarrow$  Proportion of  $\{\tilde{\theta}_i\}_{i=1}^{NumPerm}$  in  $V$  where  $\tilde{\theta}_i \geq \theta$ ;
return p-value

```

**Algorithm 1:** Permutation Test

If the obtained p-value is smaller than  $\alpha$ ,  $H_0$  will be rejected in favour of  $H_A$ .

## 4.2 Cold Code Isolation Pattern

### 4.2.1 Setup

The benchmarking procedure for the Cold Code Isolation pattern, detailed in Section 3.1, is as follows: Two versions of a function are utilised - `foo_slow()` and `foo_fast()`. Both functions loop through a vector of 400,000 unsigned integers sampled uniformly from a range between 0 and 200,000. Within this loop, each element in the vector is incrementally added to some variable alongside the execution of some other additional operations. In the uncommon instances where an element has a value of either 0, 1000, 10000, 100000, or 200000, supplementary cold code is executed.

```

1 using uvec = std::vector<unsigned>;
2 #define REPEAT8(x) x;x;x;x;x;x;x;x;x;
3 #define REPEAT16(x) //...
4
5 int foo_slow(uvec &data) {
6     int count = 0;
7     for (auto &e: data) {
8         count += e; // hot code
9         if (e == 0) REPEAT64(count++); // cold code
10        count++; // hot code
11
12        if (e == 1000) REPEAT64(count++); // cold code
13        count++; // hot code
14
15        if (e == 10000) REPEAT64(count++); // cold code
16        count++; // hot code
17
18        if (e == 100000) REPEAT64(count++); // cold code
19        count++; // hot code
20
21        if (e == 200000) REPEAT64(count++); // cold code
22    }
23    return count;
24 }
25
26 void edgeCaseHandler(int errorCode, int &count) {
27     REPEAT64(count++);
28     return;
29 }
30
31

```

```

32 int foo_fast(uvec &data) {
33     int count = 0;
34     for (auto &e: data) {
35         if (e == 0 || e == 1000 || e == 10000 ||
36             e == 100000 || e == 200000)
37             edgeCaseHandler(e, count); // cold code removed
38
39         //hot code
40         count += e;
41         count++;
42         count++;
43         count++;
44         count++;
45     }
46     return count;
47 }

```

**Code Snippet 4.1:** Cold Code Isolation Pattern (Benchmarking)

In `foo_slow()`, the hot code is deliberately intermingled with the cold code, leading to i-cache line fragmentation. Conversely, `foo_fast()` is purposely structured to segregate the cold code from the hot code by placing it within a separate function (`edgeCaseHandler()`). An inspection of the compiled program using `objdump` reveals that although both functions perform identical operations, `foo_slow()` occupies  $5ff_{16} = 1535_{10}$  bytes while `foo_fast()` requires merely  $129_{16} = 297_{10}$  bytes.

One essential detail worth mentioning is that both functions process identical input data within each specific benchmarking iteration. However, throughout the 500 iterations, new data is generated in each round using a different random seed.

## 4.2.2 Results

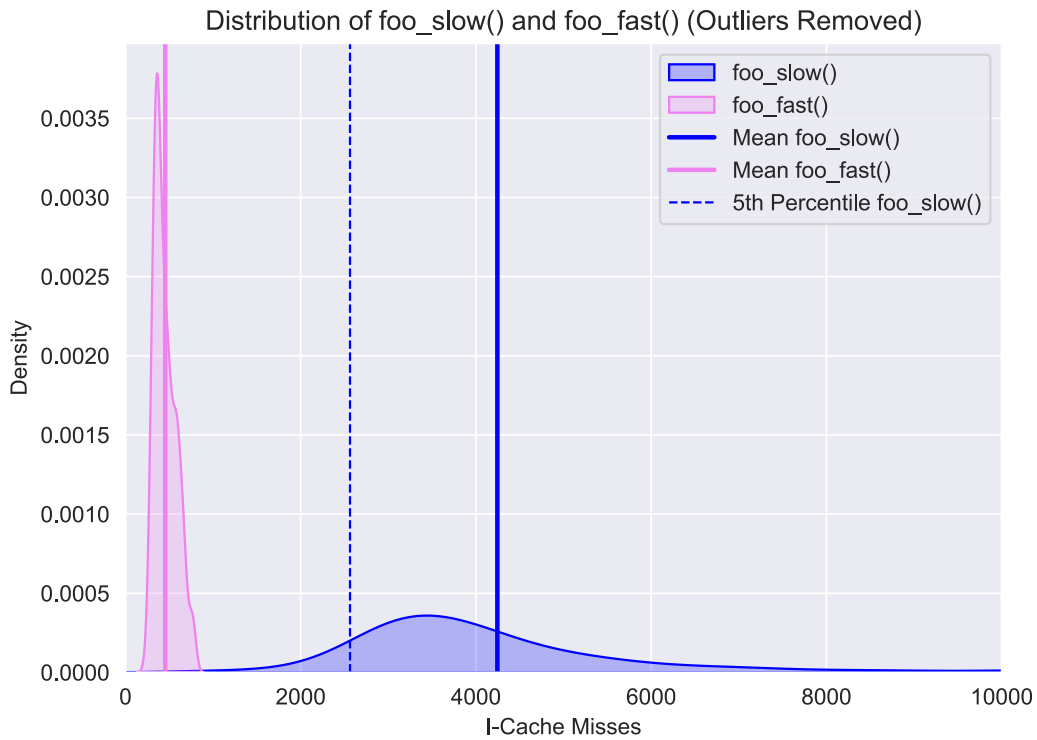
Let the random variable  $X$  denote the number of i-cache misses for `foo_slow()`, and let  $Y$  denote the same for `foo_fast()`. In this experimental setup, the hypotheses are formulated with a significance level of  $\alpha = 0.05^4$  as follows:

$$H_0 : \mu_X = \mu_Y \quad vs. \quad H_A : \mu_X > \mu_Y$$

Table 4.2 presents all the benchmarking metrics obtained after outlier removal - specifically, all data points exceeding the 99th percentile were omitted - for both `foo_slow()` and `foo_fast()`. These measurements are aggregated from a total of 500 iterations. In Figure 4.1, the distribution of i-cache misses for each function is visualised through kernel density estimates, offering a smoother and more precise representation of the data distributions.

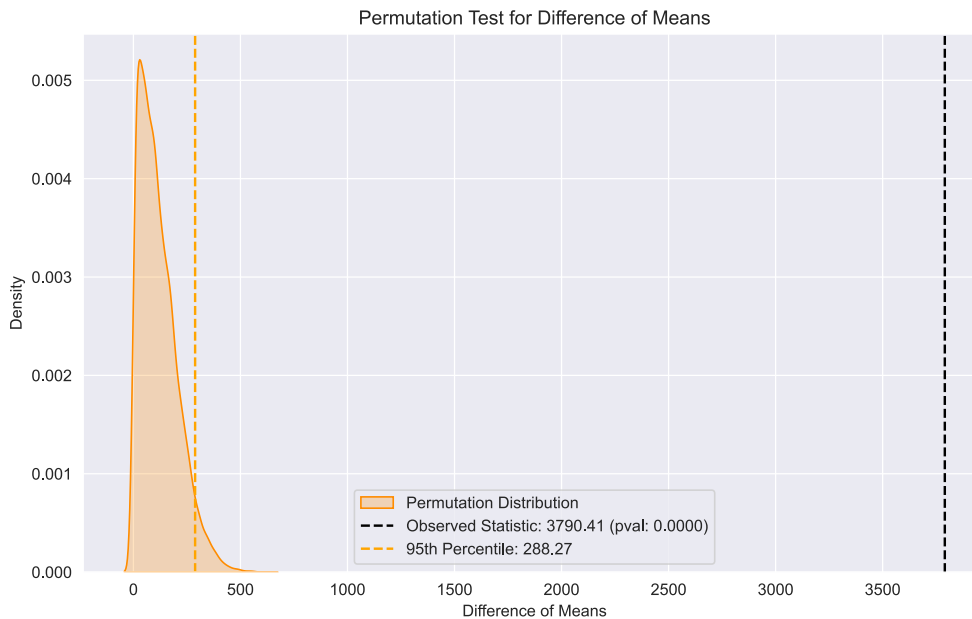
<sup>4</sup>This level of significance remains the same throughout the rest of this chapter.

	foo_slow()		foo_fast()	
	Average	StdDev	Average	StdDev
Instruction cache misses	4244.32	1901.59	453.91	123.97
Data cache misses	28298.7	2434.95	25167.5	103.14
Branch mispredictions	191.37	66.32	37.18	8.77
Total cycles	$6.92 \times 10^8$	$5.42 \times 10^6$	$2.02 \times 10^7$	$6.99 \times 10^5$

**Table 4.2:** Metrics for `foo_slow()` and `foo_fast()` (500 iterations).**Figure 4.1:** Kernel Density Estimates for I-Cache Misses (500 iterations).

The permutation test for the mean i-cache misses produced a p-value of  $0.000001 < \alpha = 0.05$ , leading to the rejection of  $H_0$ . Thus, the evidence strongly supports the argument in Section 3.1, namely that the Cold Code Isolation pattern significantly reduces i-cache misses, thereby positively impacting latency<sup>5</sup>. Figure 4.2 displays the test results.

<sup>5</sup>The difference in mean CPU cycles is also strongly influenced by branch mispredictions (stemming from the functions' design). Though this complicates quantifying the i-cache misses' direct impact on latency, a basic understanding of caches allows us to infer that fewer i-cache misses alone will lead to reduced latency.



**Figure 4.2:** Permutation Test Results (10000 iterations).

It is imperative to highlight that the results in this and subsequent sections are influenced by the design specifics of the toy examples, whose primary purpose is to be illustrative. For instance, increasing the cold code volume in `foo_slow()` will proportionally increase the disparity in mean i-cache misses.

## 4.3 DOD Pattern

### 4.3.1 Setup

The benchmarking setup for the DOD pattern builds on the example provided in Section 3.2<sup>6</sup>: The `RiskAnalyzer` class performs the critical task of monitoring all open trading positions, collecting vital information such as total profit and loss (PNL) and total risk exposure by iterating over all open positions.

Two distinct implementation approaches are contrasted in this scenario. One utilises the OOP methodology, encapsulating data and functions within objects. On the other hand, the alternative strategy implements the DOD pattern for both `RiskAnalyzer` and `Position` classes. In DOD, the hot attributes, namely `pnl` and `percExp`, of all `Position` objects are stored contiguously in a vector.

As explained in Section 3.2, the hypothesis is that storing all hot attributes contiguously will enable the `RiskAnalyzer` to operate more efficiently due to better d-cache utilisation. Specifically, when the `RiskAnalyzer` iterates over these hot attributes to

<sup>6</sup>Therefore, to avoid repetition and maintain brevity, specific code snippets are not included in this section.

compute the total risk, the DOD approach should, in theory, result in fewer d-cache misses and hence, better performance.

In order to assess this hypothesis, a comparative experiment was performed. Both RiskAnalyzer implementations computed the total PNL and risk exposure for 50 open trading positions over 10,000 iterations, each generating an empirical distribution of the d-cache misses.

A crucial observation is that in the OOP implementation, a Position object occupies 112 bytes. Hence, 50 such objects, when stored contiguously in a C++ vector, amass a memory footprint of 5600 bytes, equating to around 87.5 d-cache lines that must be iterated over. In contrast, the DOD implementation utilises a vector of 50 floats for both frequently accessed attributes within the RiskAnalyzer object, leading to a substantially smaller total of 200 bytes - about the size of 3 d-cache lines.

### 4.3.2 Results

Let the random variable  $X$  denote the number of d-cache misses for the OOP variant, and  $Y$  represent the same for the DOD variant. The hypotheses are:

$$H_0 : \mu_X = \mu_Y \quad vs. \quad H_A : \mu_X > \mu_Y$$

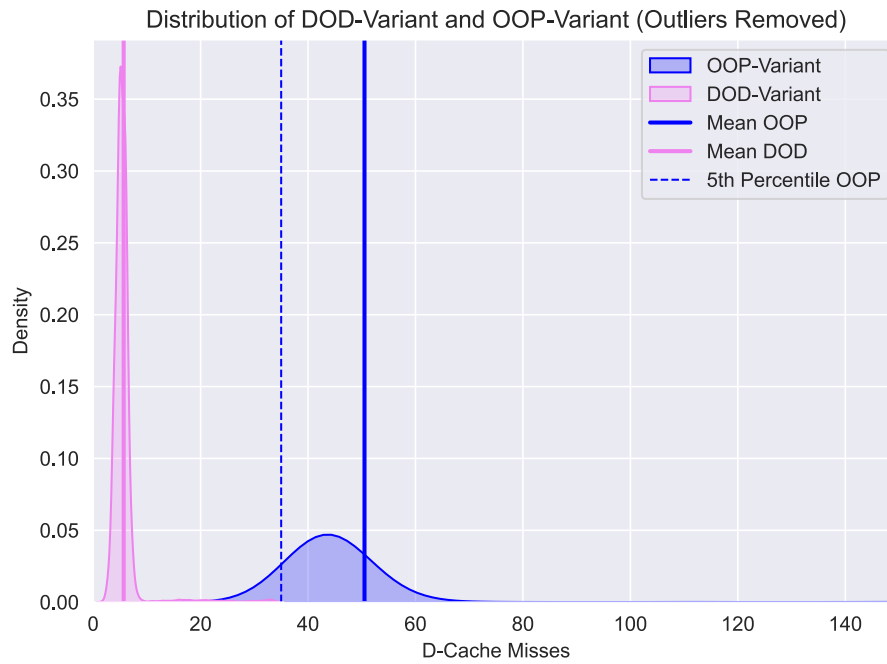
Table 4.3 shows all the metrics obtained over 10000 iterations:

	OOP Variant		DOD Variant	
	Average	StdDev	Average	StdDev
Instruction cache misses	74.11	12.18	27.64	5.32
Data cache misses	50.51	33.7613	5.68	2.99
Branch mispredictions	3.06	0.251172	2.99	0.01
Total cycles	19642.6	1935.67	4265.07	297.63

**Table 4.3:** Metrics for OOP and DOD Variant (10000 iterations)

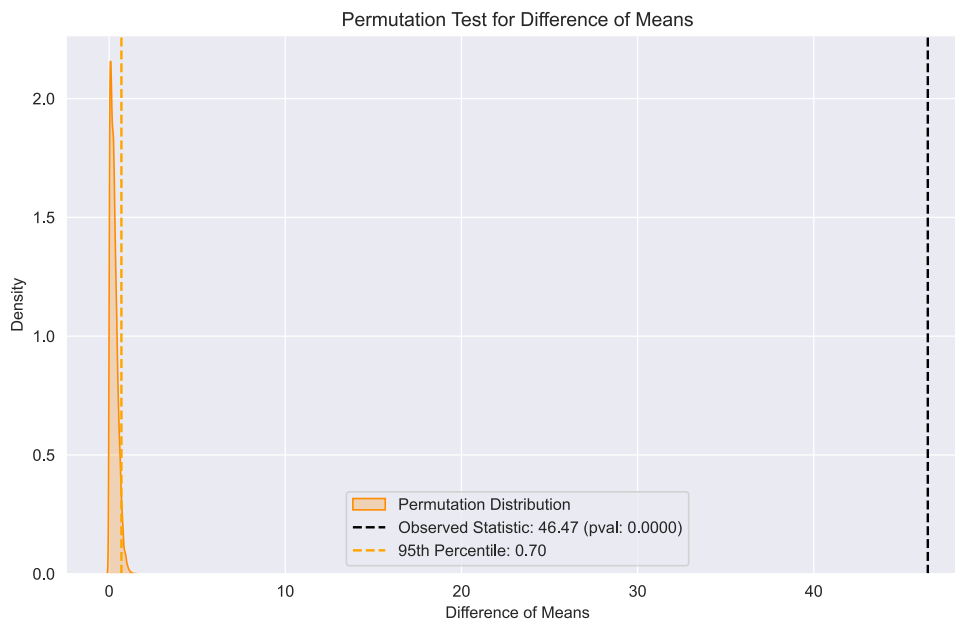
The empirical distributions obtained for the specific metric of interest - the number of d-cache misses - under the two distinct paradigms, OOP and DOD, are as follows:





**Figure 4.3:** Kernel Density Estimates for D-Cache Misses (10000 iterations).

The results of the permutation test are:



**Figure 4.4:** Permutation Test Results (10000 iterations).

The p-value of 0.00001 indicates a clear rejection of  $H_0$ , supporting the conclusion that the DOD variant is significantly more d-cache-friendly than its OOP counterpart.

## 4.4 Denormalised Data Pattern

### 4.4.1 Setup

The benchmarking procedure for the Denormalised Data Pattern draws inspiration from Cook's example, detailed in Section 3.3. An instance of the `TradingEngine` class monitors 200 distinct `Instrument` objects spread across ten different trading exchanges. This monitoring process involves 1000 iterations over each instrument, totalling 200,000<sup>7</sup> iterations. In each iteration, the respective instrument generates a trading signal with a probability of 0.1%<sup>8</sup>.

There are two variations of the `Instrument` class. The normalised version lacks two exchange-related attributes, necessitating their retrieval from a dictionary when creating the order for the respective instrument. In contrast, the denormalised version incorporates these two attributes, eliminating the need for lookup operations.

As hypothesised in Section 3.3, the denormalised version is presumed to be more d-cache friendly, as the necessary data is immediately available and does not require an additional lookup operation.

```

1
2 //denormalised variant
3 struct InstrumentDenorm {
4     InstrumentDenorm(int id, int exId, float multiplier,
5                     int ordertype);
6     int ID;
7     int exchangeID;
8     bool checkSignal();
9     //two extra attributes to avoid lookup
10    float multiplier;
11    int orderType;
12
13 };

```

**Code Snippet 4.2:** Denormalised Data Pattern (Benchmarking)

### 4.4.2 Results

Let the random variable  $X$  represent the number of d-cache misses for the Normalised variant, and let  $Y$  denote the same for the Denormalised variant. The hypotheses are:

$$H_0 : \mu_X = \mu_Y \quad \text{vs.} \quad H_A : \mu_X > \mu_Y$$

<sup>7</sup>This is for one benchmarking run. The entire benchmarking experiment consists of 500 runs.

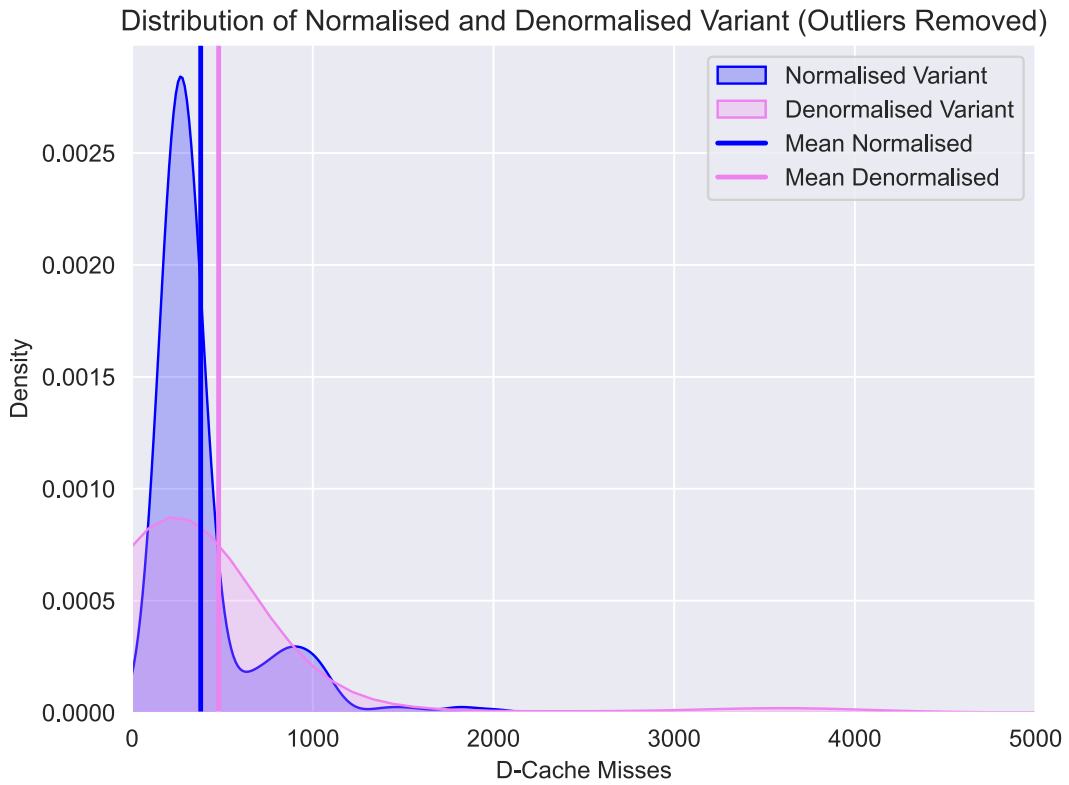
<sup>8</sup>This value is derived from Cook's observation that an order is created only in 0.1% of cases, as referenced in Section 2.3.

The results obtained are:

	Normalised Implementation		Denormalised Implementation	
	Average	StdDev	Average	StdDev
Instruction cache misses	1333.94	518.67	650.24	305.38
Data cache misses	379.78	281.85	479.13	1404.74
Branch mispredictions	1840.56	79.36	1216.41	30.15
Total cycles	3.23104e+07	119588	3.30258e+07	642982

**Table 4.4:** Metrics for Normalised and Denormalised Implementations (500 iterations).

Contrary to expectations, the average number of d-cache misses is higher for the Denormalised variant, as shown in Table 4.4. The empirical distributions for the d-cache misses can be visualised as follows:



**Figure 4.5:** Kernel Density Estimates for D-Cache Misses (500 iterations).

Given the unexpected results, it is not necessary to conduct a permutation test and  $H_0$  can already be accepted.

The experiment's outcome might be linked to the DOD pattern detailed in the previous section: By encapsulating (more) cold data within the Instrument objects in the Denormalised version, the fragmentation of cache lines within the d-cache could increase, leading to a rise in the number of cache misses. Another interesting observation is that the average number of i-cache misses in the Denormalised version is

approximately half that of the Normalised version. A plausible explanation could be that the dictionary lookup code in the Normalised version is executed infrequently (only 0.1% of the time), resulting in an i-cache miss when it does run.

## 4.5 Cache Priming Pattern

### 4.5.1 Setup

The experimental setup for the Cache Priming Pattern, outlined in Section 3.4, closely aligns with the example detailed in the same section. A loop persistently monitors for a trading signal, and upon receipt of such a signal, a trading order is created and sent off to the exchange:

```
1 //Variant without Cache Priming
2 for(int i = 0; i < INF, ++i){
3     if (shouldBuy()) { // true 0.01% of the time
4         //OCE code
5         auto p = getStockPrice();
6         auto stopLoss = getRiskLimitPrice(riskParams);
7         auto order = Order();
8         order.price = p;
9         order.stopLoss = stopLoss;
10        order.orderType = 1;
11        order.send(100); //buy 100 stocks
12        logging();
13    }
14 }
15
16 //Cache Priming Variant
17 for(int i = 0; i < INF, ++i) {
18     if (i % 200 == 0) {
19         auto dummyPrice = getStockPrice();
20         auto dummyStopLoss = getRiskLimitPrice(riskParams);
21         auto dummyOrder = Order();
22         dummyOrder.price = dummyPrice;
23         dummyOrder.stopLoss = dummyStopLoss;
24         dummyOrder.orderType = 1;
25         dummyOrder.send(0); // dont buy any stocks
26     }
27
28     if (shouldBuy()) {
29         //same OCE code ...
30     }
31 }
```

**Code Snippet 4.3:** Cache Priming Pattern (Benchmarking)

As explained in Section 3.4, the hypothesis is that the occasional generation of dummy orders decreases the likelihood of a d- and i-cache miss when the OCE code<sup>9</sup> is actually executed - which happens only 0.1% of the time [5].

However, during the benchmarking of this particular pattern, an important question surfaced that neither Cook [5] nor Keinan [30] touched upon when introducing this technique: Does reducing the amount of cache misses for the OCE code through cache-priming affect the latency of the overall loop, i.e. is there a trade-off? Consequently, this section has two *Results* subsections. *OCE - Results* features metrics exclusively for the OCE code, while *Results* exhibits the benchmarking outcomes for the entire loop. In both variants of the code, the infinite loop was replaced with a finite loop, and the entire experiment consisted of 1000 trials.

### 4.5.2 OCE - Results

Consider the random variable  $X$  to represent the number of d-cache (i-cache) misses in the Non-Cache-Priming implementation, and let  $Y$  symbolise the equivalent for the Cache-Priming implementation. Therefore:

$$H_0 : \mu_X = \mu_Y \quad vs. \quad H_A : \mu_X > \mu_Y$$

Table 4.5 depicts the obtained metrics:

	Normal Variant		Cache Priming Variant	
	Average	StdDev	Average	StdDev
Instruction cache misses	109.08	13.39	82.62	15.34
Data cache misses	8.12	9.81	5.51	1.03
Branch mispredictions	2.33	0.78	2.45	1.25
Total cycles	2501.04	473.41	2379.36	382.05

**Table 4.5:** OCE-Results for both Variants (1000 iterations)

The kernel density estimates and the respective permutation test results for both types of cache misses are shown below:

<sup>9</sup>Order-Creation-And-Execution code

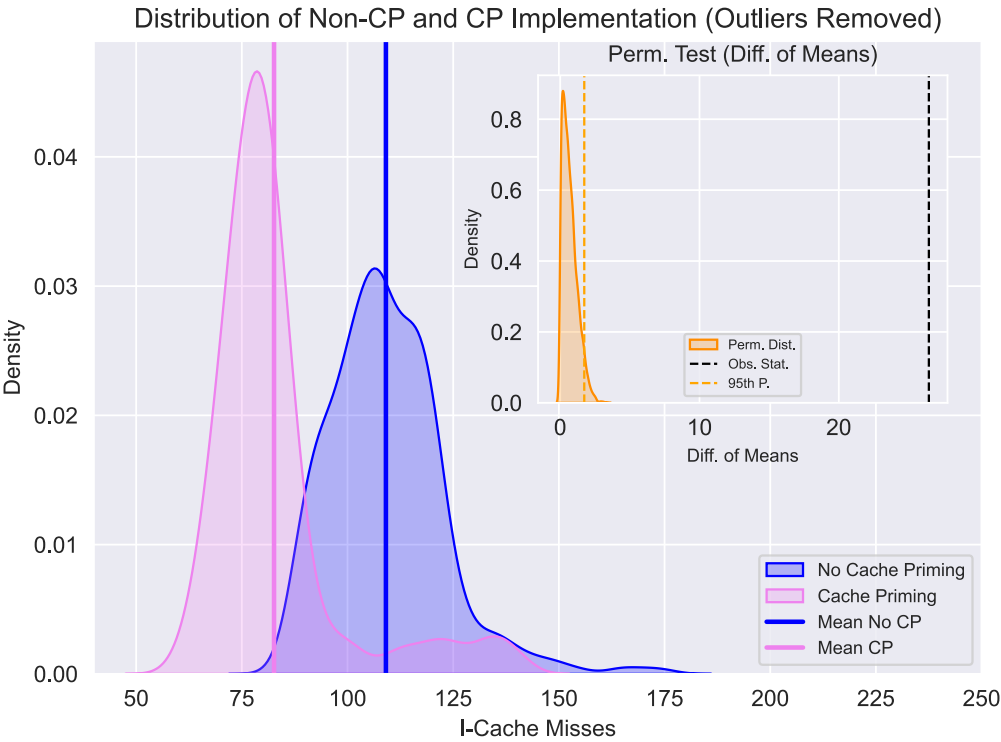


Figure 4.6: I-Cache Misses (1000 iterations).

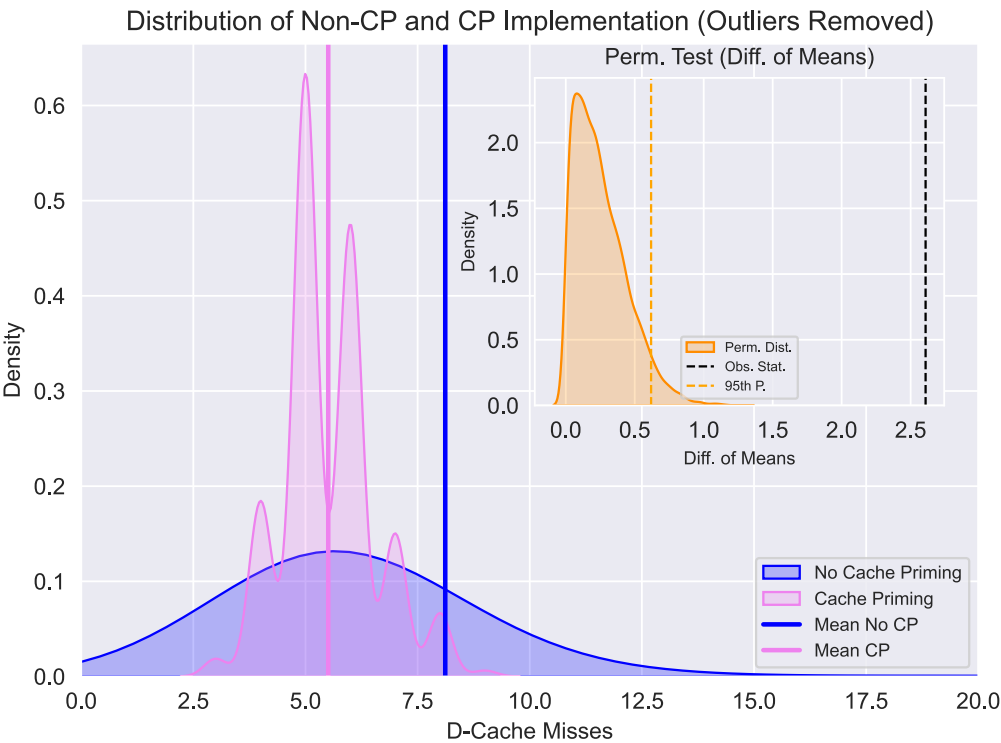


Figure 4.7: D-Cache Misses (1000 iterations).

Given that the p-values of both statistical tests are below  $\alpha$ ,  $H_0$  is rejected in favor of  $H_A$  for both types of cache misses.

### 4.5.3 Results

The hypotheses are the same as in the previous subsection:

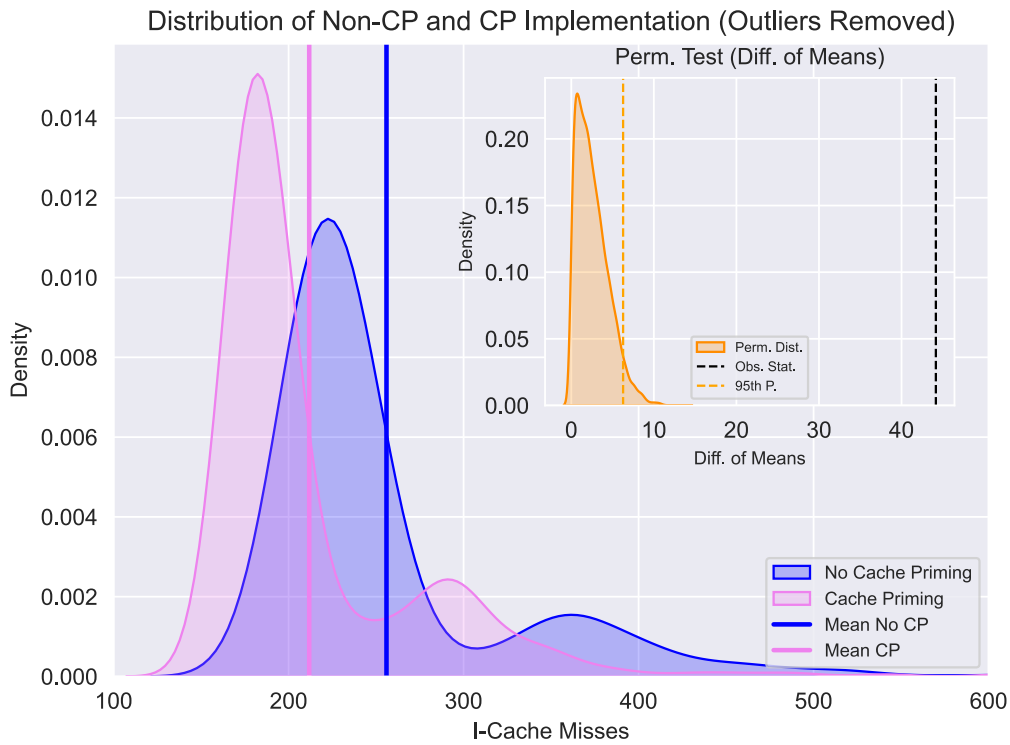
$$H_0 : \mu_X = \mu_Y \quad vs. \quad H_A : \mu_X > \mu_Y$$

The results obtained are:

	Normal Variant		Cache Priming Variant	
	Average	StdDev	Average	StdDev
Instruction cache misses	255.55	61.23	211.78	47.52
Data cache misses	10.63	7.47	10.34	6.31
Branch mispredictions	82.85	19.28	415.04	31.79
Total cycles	2.32e+06	21657.20	2.62e+06	21921.70

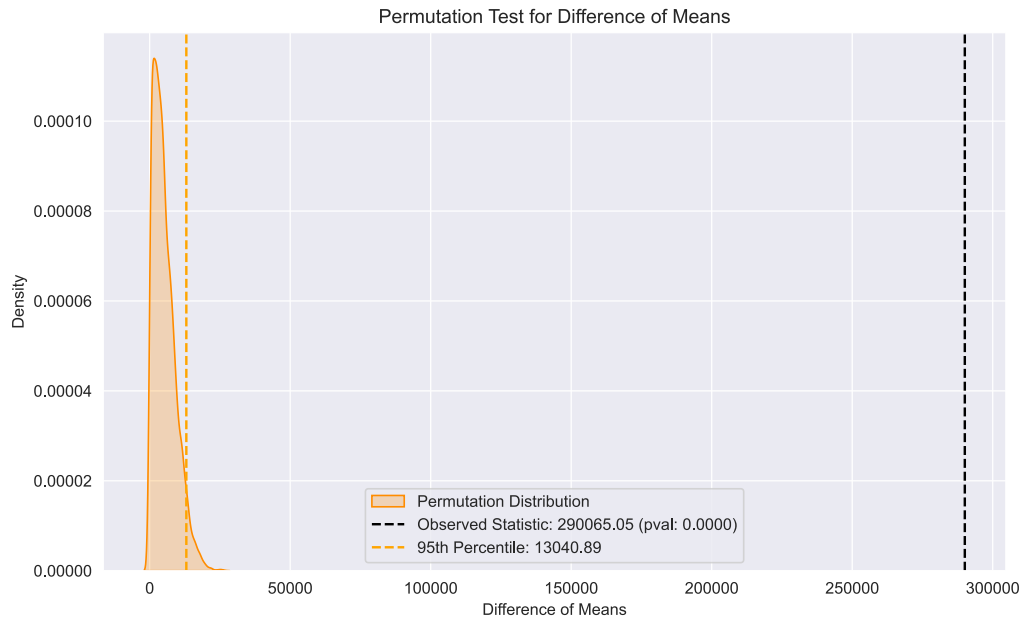
**Table 4.6:** Results for both Variants (1000 iterations).

Given the negligible difference in average d-cache misses,  $H_0$  can already be accepted for this metric. Figure 4.8 shows the empirical distributions and the permutation test result for the i-cache misses:



**Figure 4.8:** I-Cache Misses (1000 iterations).

The p-value obtained from the permutation test is 0.00001, leading to a rejection of  $H_0$  for the i-cache misses. Interestingly, despite a significant reduction in i-cache misses, the total CPU cycles appear to increase in the Cache Priming variant. Table 4.6 offers a potential explanation: Cache Priming leads to a significant increase in branch mispredictions, likely due to the additional if-statement in the loop. Hence the question arises whether the difference in mean total CPU cycles is significant, which appears to be the case:



**Figure 4.9:** CPU Clock Cycles (PT - 10000 iterations).

#### 4.5.4 Interim Conclusion

Considering the results from the two previous subsections, it is evident that implementing Cache Priming using an additional branch (if-statement) introduces a trade-off. On the one hand, the pattern effectively reduces the number of cache misses in the OCE-part of the trading engine, where latency matters the most. On the other hand, it increases the latency in the rest of the code.

However, if this trade-off is unacceptable, the pattern must be implemented without the additional branch and in such a way that the cache-priming-code is executed only *occasionally*<sup>10</sup>. Unfortunately, this implementation may not be trivial and could require a complete redesign of the entire execution engine.

<sup>10</sup>Executing it in every iteration of the loop turned out to be severely counterproductive.



## 4.6 TB Branch Elimination Pattern

### 4.6.1 Setup

The functions utilised for benchmarking the TB Branch Elimination Pattern are taken from the example provided by Bogosavljević [37]:

```
1  //non-templated version
2  float foo_slow(std::vector<int>& vec, bool incl_negatives) {
3      int len = vec.size();
4      int average = 0;
5      int count = 0;
6      for (int i = 0; i < len; i++){
7          if (incl_negatives) {
8              average += vec[i];
9          }
10         else {
11             if (vec[i] >= 0){
12                 average += vec[i];
13                 count++;
14             }
15         }
16     }
17 }
18
19 if (incl_negatives) return average / len;
20 else return average / count;
21 }
22
23 //templated version
24 template <bool incl_negatives>
25 float foo_fast(std::vector<int>& vec) {
26     int len = vec.size();
27     int average = 0;
28     int count = 0;
29     for (int i = 0; i < len; i++) {
30         if (incl_negatives){
31             average += vec[i];
32         }
33         else {
34             if (vec[i] >= 0){
35                 average += vec[i];
36                 count++;
37             }
38         }
39     }
40 }
```

```

41     if (incl_negatives) return average / len;
42     else return average / count;
43 }

```

**Code Snippet 4.4:** TB Branch Elimination (Benchmarking)

Despite the good predictability of the branches in `foo_slow()`, evaluating the if-statement incurs a certain run-time cost. By using templates<sup>11</sup>, the compiler will instantiate two versions of `foo_fast()`, which eliminates the if-statement and the code of the unused branch [37]. Consequently, the assumption is that `foo_fast()` would require fewer overall CPU clock cycles<sup>12</sup>. Both functions are benchmarked 1000 times using a vector filled with randomly generated data.

## 4.6.2 Results

Define the random variable  $X$  as the number of CPU clock cycles required for `foo_slow()`, and similarly, let  $Y$  denote the corresponding number for `foo_fast()`. Consequently:

$$H_0 : \mu_X = \mu_Y \quad vs. \quad H_A : \mu_X > \mu_Y$$

Table 4.7 displays the obtained results:

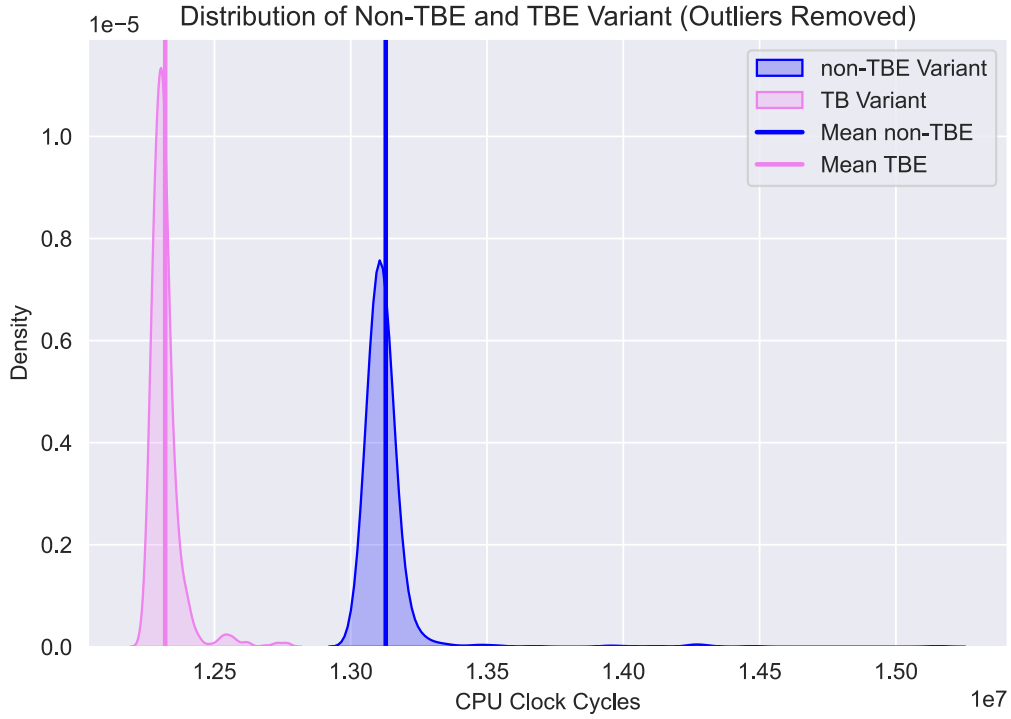
	foo_slow()		foo_fast()	
	Average	StdDev	Average	StdDev
Instruction cache misses	168.80	31.14	161.60	36.39
Data cache misses	62625.80	37.84	62625.40	35.63
Branch mispredictions	6.80	2.21	7.15	2.51
Total cycles	13,128,400	137,777	12,319,300	61,800.20

**Table 4.7:** Metrics for Both Function Versions (1000 iterations).

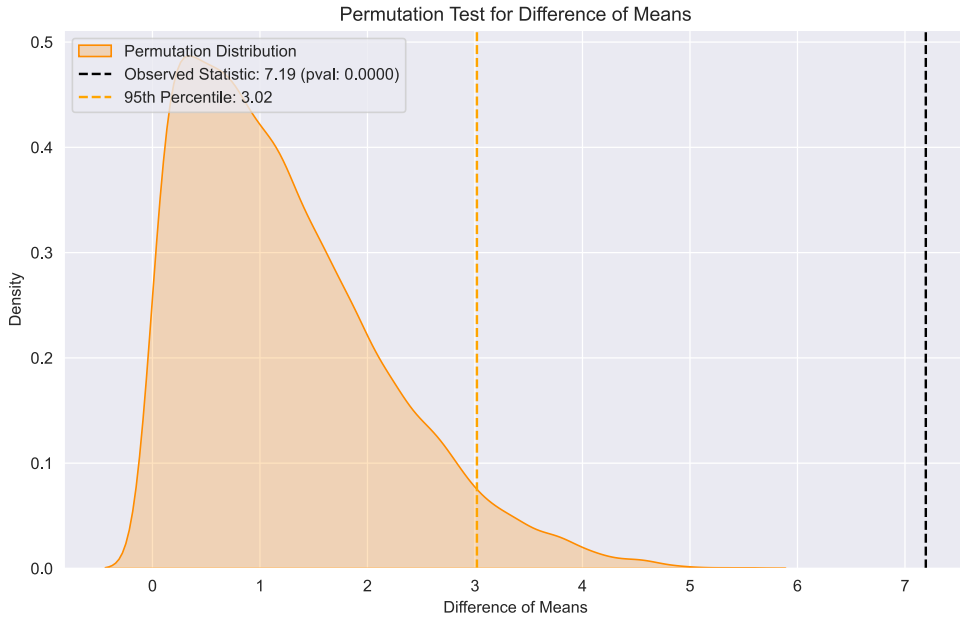
The following two graphics depict the respective density estimates for the total CPU clock cycles required and the corresponding permutation test:

<sup>11</sup>As outlined in Section 3.5, the pattern relies on the fact that `incl_negatives`'s value is already known at compile time.

<sup>12</sup>I-cache misses should also be reduced if the unused branches are large enough - which is not the case in this example.



**Figure 4.10:** Kernel Density Estimates for CPU Clock Cycles (1000 iterations).



**Figure 4.11:** Permutation Test Results (10000 iterations).

Given the p-value of 0.00001,  $H_0$  can be rejected. Therefore, it is reasonable to infer

that this pattern does indeed improve latency by directly reducing the number of CPU clock cycles required.

## 4.7 CRTP

### 4.7.1 Setup

The setup for benchmarking dynamic polymorphism (virtual methods) against static polymorphism (CRTP) is as follows:

```

1 //dynamic polymorphism -----
2 class Base {
3 protected:
4     int counter;
5 public:
6     Base(): counter(0) {};
7     virtual void inc(){counter += 1;}
8     int getCounter(){return counter;}
9 };
10
11 class Special: public Base {
12 public:
13     void inc() override {counter += 2;}
14 };
15
16 //static polymorphism -----
17 template <class Derived>
18 class CRTPTemplate {
19 protected:
20     int counter;
21 public:
22     CRTPTemplate() : counter(0) {}
23     void inc(){static_cast<Derived*>(this)->incImpl();}
24     int getCounter(){return counter;}
25 };
26
27 class BaseCRTP: public CRTPTemplate<BaseCRTP> {
28 private:
29     friend class CRTPTemplate<BaseCRTP>;
30     void incImpl() {counter += 1;}
31 };
32
33 // analogous approach for SpecialCRTP ..
34
35 //main() -----
36 Special* spec = new Special();

```

```

37 spec->inc() //dynamic dispatch
38 SpecialCRTP* specCRTP = new SpecialCRTP();
39 specCRTP->inc() // static dispatch

```

**Code Snippet 4.5:** CRTP (Benchmarking)

As explained in Section 3.6, the hypothesis is that dynamic function dispatch comes with a run-time cost in the form of vtable lookups. Hence, the static dispatch variant should result in more efficient code. In order to gather benchmarking results, both `inc` methods were called 100'000 times for 200 experiment iterations.

## 4.7.2 Results

Let the random variable  $X$  represent the number of CPU clock cycles required for the dynamic variant, and let  $Y$  denote the same for the CRTP. Thus:

$$H_0 : \mu_X = \mu_Y \quad vs. \quad H_A : \mu_X > \mu_Y$$

The obtained results are as follows:

	Dynamic Variant		CRTP	
	Average	StdDev	Average	StdDev
Instruction cache misses	39.3568	15.0312	59.3317	26.2481
Data cache misses	5.41	4.88068	9.9397	9.35746
Branch mispredictions	4.17085	0.620325	3.28643	1.13412
Total cycles	852183	4551.47	2542250	241930

**Table 4.8:** Results for Dynamic Variant vs CRTP (200 iterations)

Considering these quite unexpected results, it is reasonable to accept  $H_0$  and to omit any further permutation test. One plausible reason the CRTP version of the code demanded more CPU clock cycles is the additional function call: `incImpl()` is invoked within `inc()`.

Given the surprising nature of the results, and in light of the dominant view in the existing literature that vtable lookups associated with virtual methods lead to a significant performance drop, it becomes imperative to delve deeper. This led to further investigations, which are elaborated upon in the subsequent subsection.

## 4.7.3 Excursion

### Compile Time Optimisations

As stated at the beginning of this chapter, all experiments are conducted without any compiler optimisations (i.e. `-O0`). However, after enabling the first level of compiler optimisations (`-O1`), the obtained assembly code revealed some valuable insights:

```

1 ;dynamic variant: spec->inc() x 100k
2 .L3:
3   ;load vtable pointer
4   mov     rax, QWORD PTR [rbp+0]
5   mov     rdi, rbp
6   ;use vtable ptr to call correct function
7   call    [QWORD PTR [rax]]
8   ;check loop condition
9   sub     ebx, 1
10  jne     .L3
11
12 ;static variant: specCRTP->inc() x 100k
13 .L4:   ;loop runs 100k times and does nothing
14   add     eax, 2
15   cmp     eax, 200000
16   jne     .L4
17
18 ; compiler computes result a-priori and uses it as rvalue
19 ; e.g. for cout << specCRTP->getCounter();
20   mov     esi, 200000

```

**Code Snippet 4.6:** Obtained Assembly for Both Variants

In the dynamic version, the virtual method `inc()` is invoked via a vtable pointer, necessitating run-time dispatch, preventing any compile-time optimisations such as function inlining. The contrary happened in the CRTP version: Because of static dispatch, the compiler was able to pre-compute the result of 100'000 `specCRTP->inc()` calls, allowing their removal from the loop. This led to a substantial speed-up:

	Regular OOP		CRTP	
	Average	StdDev	Average	StdDev
Instruction cache misses	30.598	17.2287	29.0503	12.698
Data cache misses	4.35176	5.57292	5.23116	4.27547
Branch mispredictions	2.46231	0.708774	3.835	1.20625
Total cycles	502050	4547.78	101075	4081.57

**Table 4.9:** Performance Metrics with Compiler Optimisations (200 iterations)

The significant difference in mean total CPU cycles can be deduced from the observed numbers and their corresponding standard deviations. Therefore, a statistical test is omitted at this point.

Bogosavljevic's [38] insights align with the findings obtained in this subsection and the previous one: The run-time overhead caused by dynamic dispatch<sup>13</sup> is usually negligible. However, a more notable disadvantage of dynamic binding is that it prevents any compile-time optimisations.

<sup>13</sup>Depending on the virtual function's size and the size of the vtable.

### Virtual Function Prediction

Another interesting aspect, emerging from further investigations incited by the unexpected findings in subsection 4.7.2, is as follows: Similar to branch prediction for program control flow, modern CPUs attempt to predict the correct virtual method in the vtable and carry out speculative execution [38].

The results of the following experiment underline the aforementioned point. Three different child classes are derived from a base class, each overriding the inherited `inc()` method:

```

1
2 class Base {
3     protected:
4         int counter = 0;
5     public:
6         virtual void inc(){counter += 0;}
7 };
8
9 class Child1: public Base {
10 //overrides inc() ...
11 };
12
13 //...
```

**Code Snippet 4.7:** VTable Prediction (Setup)

Subsequently, two vectors are populated; each with a total of three million pointers of type `Base`<sup>14</sup>. Of these, one million pointers reference objects of class `Child1`, another million point to objects of class `Child2`, and the remaining million to objects of class `Child3`. The pointers in the first vector follow a deterministic ordering, adopting an A-B-C-cyclic pattern that corresponds to `Child1`, `Child2`, and `Child3`, respectively. In contrast, the pointers in the second vector are randomly shuffled, thereby eliminating any systematic sequence of the pointed-to types. Iterating over each vector, while invoking `inc()` in each iteration, yields the following results:

	Not Shuffled		Shuffled	
	Total	% of all	Total	% of all
CPU Cycles	676'145'506	-	2'457'079'119	-
Branches	252'444'371	-	422'020'461	-
Branch Mispredictions	108'466	0.04%	3'647'254	0.86%

**Table 4.10:** Performance Metrics for Vtable Predictions

As illustrated in Table 4.10, the iteration over the shuffled vector incurs significantly

<sup>14</sup>Again, this is one of the key advantages that virtual methods hold over the CRTP, namely the flexibility to reference a child object using a parent-typed pointer.

higher costs. This increase is largely due to frequent mispredictions as the CPU attempts to predict the correct virtual method within the vtable.

#### 4.7.4 Interim Conclusion

The benchmarking experiment conducted for the CRTP yields the following conclusions: Contrary to commonly held beliefs in the literature, dynamic dispatch is not computationally more expensive than static dispatch *if* compiler optimisations are not enabled *and* the correct method in the vtable can be easily predicted. However, when compiler optimisations are employed, dynamically dispatched functions face a severe disadvantage since they do not allow for any optimisations at compile time, unlike the statically dispatched methods of the CRTP.



# Chapter 5

## Software Application: The LLDPA

### 5.1 Functionality

#### 5.1.1 C++ Code Analysis

In order to allow the Low-Latency Design Patterns Analyser (LLDPA) to effectively identify implementation opportunities for the design patterns presented in Chapter 3, two key challenges needed to be addressed:

1. **Parsing C++ Code:** The application must be able to ‘understand’ the supplied C++ code correctly.
2. **Mapping the Design Patterns:** Each catalogued design pattern must be mapped to a particular deterministic characteristic, which the LLDPA subsequently searches for.

The most robust and precise way to address the first point is to leverage the parsing capabilities of a compiler and then generate an abstract syntax tree (AST). The application can then use the AST to search for the characteristics mentioned in point two. The LLDPA, which is written in Python, uses the `clang.cindex` library - the Python interface of the clang compiler - to achieve this objective.

Concerning the second point, the specific characteristics the LLDPA searches for to generate a refactoring suggestion for a particular low-latency design pattern are:

- **TBBE:** The branches inside a method or function are determined by one or more of its input arguments.
- **CRTP:** If a class uses a virtual method, the LLDPA will suggest refactoring the class using the CRTP. Unfortunately, `clang.cindex` cannot determine if a virtual function is called through a pointer or a reference (since only then is the function call dispatched dynamically). Hence, the LLDPA must resort to merely identifying virtual functions.
- **Cold Code Isolation:** The application runs a mock program – supplied by the programmer – and then uses `gcov` to generate coverage reports for each C++

class. Subsequently, the LLDPA finds all the if-else statements in the code and compares how often the if-condition was evaluated versus the execution counts of the associated then- and else-blocks. If a block is executed less than 15% of the time, it is marked as cold code.

- **DOD:** To evaluate which class attributes are hot attributes, the LLDPA uses the data from the gcov reports and determines how many times a specific getter or setter method was invoked in the mock program. An attribute of a particular class is regarded as 'hot' if it is accessed more than three times as much as the least accessed attribute of the same class.

The application does not provide refactoring suggestions for the Denormalised Data Pattern due to its subpar benchmarking results (cf. Section 4.4.2), and the Cache Priming Pattern, as it is too HFT-specific and its mapping is quite difficult.

### 5.1.2 Automated Code Refactoring

Despite generating suggestions (in the form of comments) for design pattern implementations, the LLDPA leverages the power of large language models (LLMs) to allow for automated refactoring, alleviating the programmer from any manual labour. In particular, the application uses the OpenAI API to interact with OpenAI's Chat-GPT.

The specific process in the back-end can be summarised as follows: A prompt is created, consisting of three distinct parts. The first part presents sample C++ code to Chat-GPT that illustrates the low-latency design patterns. The second part formulates a request to the model, asking it to refactor any subsequent C++ code accordingly. The final part incorporates a copy of the original C++ code that the programmer supplied, along with the comments from the source code analysis to assist the model and improve its precision.

What if the modified code returned by Chat-GPT lacks some necessary refactorings, or the existing refactorings are unsatisfactory? To address this issue, the LLDPA allows the user to select the file containing the refactored code and re-submit it to the model. For this re-submission, the user can include custom requirements by entering them into a designated text field within the application's graphical user interface (GUI). In the back-end, a new prompt is generated that merges the original code, the refactored code from the initial submission, and a re-refactoring command. The command emphasises explicitly the requirements the user has supplied through the text field. This additional feature enables a more customised interaction with the model, allowing users to guide the refactoring process according to their specific needs.

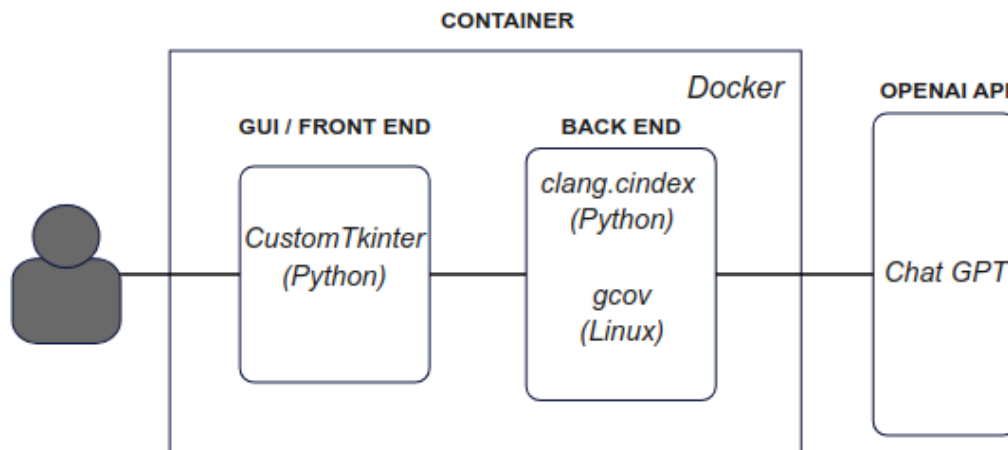


Figure 5.1: LLDPA Architecture

## 5.2 Demonstration

### 5.2.1 The Code

#### The Mock Program

The mock program, written by the user, is supposed to be a realistic simulation of the main program they plan to execute in their low-latency application. A mock program must be run because the Cold Code Isolation and DOD patterns require run-time coverage reports and thus do not allow for static analysis. By enabling users to supply their own mock program, they can simulate any API calls themselves and utilise any data<sup>1</sup> they desire. This flexibility allows the LLDPA to be used as a developer tool across different low-latency applications while maintaining a fairly lightweight profile.

The mock program used in the upcoming demonstration looks as follows:

```

1 #include <iostream>
2 #include <random>
3 #include "Foo.h"
4
5 //sample data from uniform dist. for demonstration
6 std::vector<int> generate_random_data(int n, int low, int
    high, int seed) { /*implementation*/ }
7
8 int main() {
9     Foo* obj = new Foo("Alex");
10    obj->set_status(23);
  
```

<sup>1</sup>To obtain meaningful results, the data provided should come from the same distribution as future application data.

```

11     for (int i = 0; i < 100; i++) obj->set_status(i);
12     for (int i = 0; i < 10000; i++) obj->get_name();
13
14     for (int i = 0; i < 100; i++){
15         auto data = generate_random_data(10000, -10,
16                                         1000, i+1);
17         obj->calculate_result(data, false);
18         obj->calculate_result(data, true);
19     }
20     auto data = generate_random_data(10000, -10,
21                                     1000, 234);
22     for (int i = 0; i < 10000; i++) obj->foo(data);
23     for (int i = 0; i < 10000; i++) obj->incResult();
24     std::cout << "Mock Program Finished. Result: " <<
25               obj->get_result() << std::endl;
26     return 0;
27 }

```

Code Snippet 5.1: mock.cpp

### The Implementation Files

The targets of the code analysis are the C++ files that contain the implementations for the various classes and functions used in the mock program. The LLDPA creates a copy of each C++ implementation file (in HTML format) and includes the refactoring suggestions articulated as comments. These copies are stored in a separate folder named HTML\_OUT.

The class implementation for the Foo class used in the upcoming demonstration looks as follows:

```

1  #include "Foo.h"
2
3  Foo::Foo(std::string name) : name(name), status(1) {}
4
5  int Foo::get_status() {
6      return status;
7  }
8
9  float Foo::get_result(){
10     return result;
11 }
12
13 float Foo::calculate_result(std::vector<int> &data,
14                             bool inc_neg) {
15     float res = 0.0;
16     int count = 0;

```

```
17     for (auto e: data){
18         if (inc_neg) res += e;
19         else{
20             if (e >= 0){
21                 res += e;
22                 count++;
23             }
24         }
25     }
26     if (inc_neg) result = res/data.size();
27     else result = res/count;
28     return res;
29 }
30
31 void Foo::foo(std::vector<int> &data) {
32     int res = 0;
33     for (auto& e: data){
34         if (e != 325) {
35             res += e;
36         }
37         else {
38             res--;
39             int p = 0;
40             res--;
41             p++;
42             //more cold code
43         }
44     }
45 }
46
47 std::string Foo::get_name() {
48     return name;
49 }
50 //virtual method
51 void Foo::incResult() {
52     result++;
53 }
54
55 void Foo::set_status(int newStatus) {
56     status = newStatus;
57 }
```

Code Snippet 5.2: Foo.cpp

The methods of the Foo class are deliberately designed to prompt the following refactoring suggestions from the LLDPA:

Method Name	LLDPA Suggestion
Foo::calculate_result	TBBE
Foo::foo	Cold Code Isolation
Foo::incResult	CRTP
getters and setters	used for DOD

**Table 5.1:** Expected LLDPA suggestions for Foo

### Assumptions

In order for the LLDPA to work, certain requirements must be fulfilled:

- The mock program and other source code files must reside in the same directory to ensure accessibility.
- The classes are divided into header and implementation files, reflecting common practice in most applications. The header files must also be located in the same directory as the rest of the files.
- The implementation files for the classes should be concise enough so that they can be submitted to Chat-GPT with a single prompt.
- The naming convention for getter and setter methods must follow the pattern `get_` or `set_`, followed by the attribute name in lowercase letters.

### 5.2.2 Running The Application

The design of the GUI allows for the code analysis and the refactoring to be conducted independently of one another. However, it is imperative that the analysis is completed first before proceeding with any refactoring tasks.

The following screenshot illustrates how `mock.cpp` is executed and `Foo.cpp` is analysed for opportunities to implement all possible low-latency design patterns:

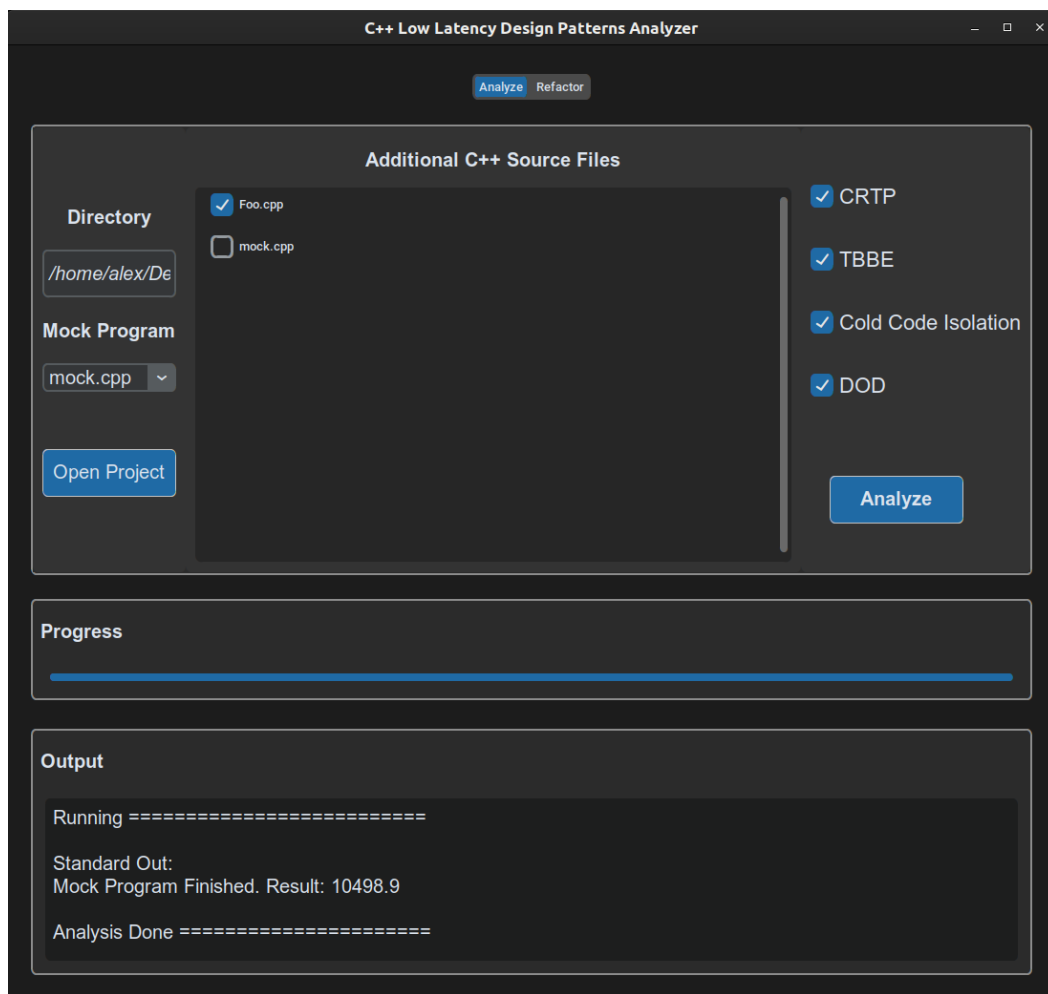


Figure 5.2: GUI (Analysis Tab)

Once the analysis is complete, the folder `HTML_OUT` is created within the project directory. This folder contains the file `Foo.cpp.html`, which contains the results of the analysis:

```

1 //Setters, getters & some code omitted for brevity purposes
2
3 float Foo::calculate_result(std::vector<int> &data,
4                             bool inc_neg) {
5 // Refactor this method by applying the TBBE Pattern.
6 float res = 0.0;
7 int count = 0;
8 for (auto e: data){
9     if (inc_neg){
10         res += e;
11     }
12     else{
13         if (e >= 0){

```

```

14         res += e;
15         count++;
16     }
17 }
18 }
19 if (inc_neg){
20     result = res/data.size();
21 }
22 else {
23     result = res/count;
24 }
25 return res;
26 }
27
28 void Foo::foo(std::vector<int> &data) {
29     int res = 0;
30
31     for (auto& e: data){
32         if (e != 325) {
33             res += e;
34         }
35         else {
36 // Refactor this ELSE block by moving it into seperate
37 method (0.1%).
38             res--;
39             int p = 0;
40             //...
41         }
42     }
43 }
44 void Foo::incResult() {
45 // Refactor this class using the CRTP to get rid of virtual
46 method.
47     result++;
48 }

```

Code Snippet 5.3: Foo.cpp.html

With respect to the DOD pattern, the file `hot_attributes.html` can also be found within the `HTML_OUT` directory, displaying the hot attributes for each class:

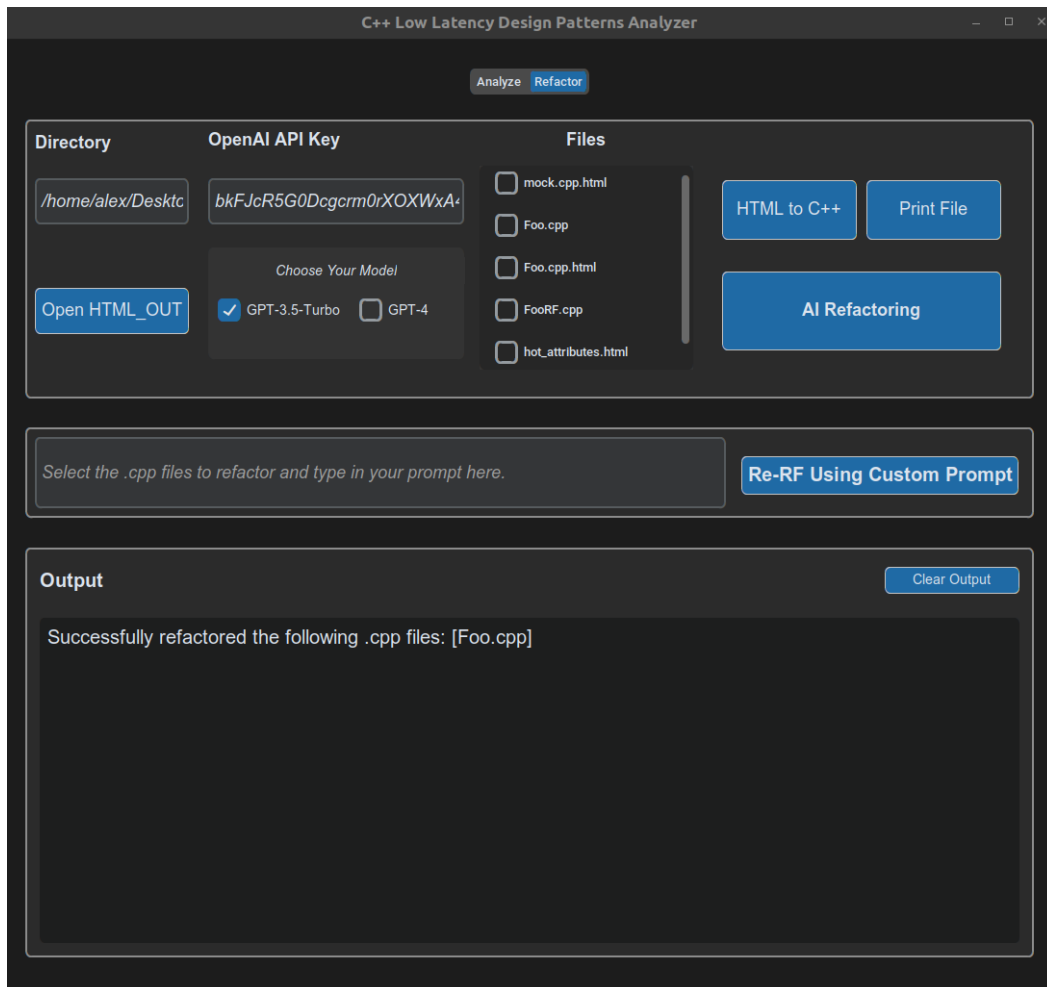
File	Class	Hot Attribute	Access Count
/home/alex/Desktop/Test3/Foo.cpp	Foo	status	101
/home/alex/Desktop/Test3/Foo.cpp	Foo	name	10000

Table 5.2: Table displayed in `hot_attributes.html`

Upon completion of the analysis, and the subsequent creation of the `HTML_OUT` folder,



the user may proceed to utilise the LLDPA's refactoring functionality:



**Figure 5.3:** GUI (Refactoring Tab)

After navigating to the correct HTML\_OUT folder, the `Foo.cpp.html` file from the analysis part needs to be converted into a `.cpp` file. This conversion can be accomplished by clicking the `HTML to C++` button, which causes the LLDPA to merge the code from the HTML file with the code from the `.h` file located in the parent directory. The resulting `HTML_OUT/Foo.cpp` file is an exact copy of `../Foo.cpp` but with the addition of the comments from the analysis.

Subsequently, clicking the `AI Refactoring` button sends the file `HTML_OUT/Foo.cpp` to Chat-GPT for refactoring. The refactored code received is saved in a new file named `HTML_OUT/FooRF.cpp`, which can be inspected using the `Print File` button. If the results are not satisfactory, the user can select `HTML_OUT/FooRF.cpp`, enter a custom prompt, and then click on `Re-RF Using Custom Prompt`. This action will prompt the model to refactor the code again, but this time taking into account the user's specific requirements.

The refactored code contained in `HTML_OUT/FooRF.cpp` turned out to be satisfactory

and does not require another refactoring using a custom prompt:

```
1 #include <string>
2 #include <iostream>
3 #include <vector>
4
5 template <typename Derived>
6 class CRTPTemplate {
7 public:
8     void incResult()
9         {static_cast<Derived*>(this)->incImpl();}
10 };
11
12 class Foo: public CRTPTemplate<Foo>{
13 public:
14     //constructor, getters and setters omitted for brevity
15
16     void incImpl() {
17         result++;
18     }
19
20     template <bool include_negatives>
21     float calculate_result(std::vector<int> &data) {
22         float res = 0.0;
23         int count = 0;
24         for (auto e: data){
25             if (include_negatives){
26                 res += e;
27             }
28             else{
29                 if (e >= 0){
30                     res += e;
31                     count++;
32                 }
33             }
34         }
35         if (include_negatives){
36             result = res/data.size();
37         }
38         else {
39             result = res/count;
40         }
41         return res;
42     }
43
44     void foo(std::vector<int> &data) {
45         int res = 0;
```

```

46         for (auto& e: data){
47             if (e != 325) {
48                 res += e;
49             }
50             else {
51                 handleSpecialCase(res);
52             }
53         }
54     }
55
56 private:
57     std::string name;
58     int status;
59 protected:
60     float result;
61     void handleSpecialCase(int res){
62         int p = 0;
63         res--;
64         //.. more cold code
65     }
66 };

```

Code Snippet 5.4: FooRF.cpp

## 5.3 Evaluation

In order to evaluate the LLDPA's capabilities and the resulting latency improvements, two different C++ test projects are analysed, refactored and then benchmarked. The original and the refactored code can be found [here](#). Both test projects were compiled with optimisations (-O1) as this is the standard approach in real-world projects, ensuring a closer reflection of reality.

### 5.3.1 CppTest1

The test project CppTest1 refers to the code depicted in the preceding section. As shown earlier, the LLDPA accurately identified (Precision: 100%, Recall: 100%) and correctly addressed all refactoring opportunities at the first attempt. Benchmarking both the original and the refactored versions yields the following results:

Metric	Pre-Refactoring	Post-Refactoring
Cycles	240'994'124	141'156'842
Cache Misses	56'834	53'840
Branch Misses	132'251	139'639

Table 5.3: Benchmarking Metrics for CppTest1

### 5.3.2 CppTest2

This test project requires a more sophisticated refactoring compared to CppTest1 because certain methods are intentionally engineered to contain refactoring opportunities for more than one design pattern. For instance, the following method not only suggests the CRTP (given its virtual nature) but also incorporates cold code blocks, which vary between classes:

```

1 void Calculator::calculate_result(std::vector<int> &data) {
2     int sum = 0;
3     for (auto& e : data){
4         if (e % 543 != 0){
5             sum += e;
6         }
7         else { // cold code specific to parent class
8             int k = 0;
9             k++;
10            k++;
11            k++;
12            k++;
13            sum += k;
14        }
15    }
16    this->result = sum;
17 }
18
19 void Child::calculate_result(std::vector<int> &data) {
20     int sum = 0;
21     for (auto& e : data){
22         if (e % 229 != 0){
23             sum += e;
24         }
25         else { // cold code specific to child class
26             int k = 5;
27             k++;
28             k++;
29             k++;
30             k++;
31             sum += k;
32         }
33     }
34    this->result = sum;
35 }

```

**Code Snippet 5.5:** CppTest2 - Snippet

The rest of the code is omitted for brevity purposes, and the reader is encouraged to refer to it in the aforementioned code repository. Refactoring the entire code correctly required two attempts with a custom prompt: While the application accurately

identified all (Precision: 100%, Recall: 100%) and correctly refactored most opportunities (7 out of 8), it mistakenly used a `constexpr`<sup>2</sup> for the implementation of the TBBE pattern, rather than employing templates. The benchmarking results for both versions of the mock program are as follows:

Metric	Pre-Refactoring	Post-Refactoring
Cycles	389'643'764	47'156'842
Cache Misses	69'834	48'840
Branch Misses	136'251	58'639

**Table 5.4:** Benchmarking Metrics for CppTest2

### 5.3.3 Summary

Based on the benchmarking results for CppTest1 and CppTest2, it is evident that the LLDPA's ability to identify refactoring opportunities and then correctly implement the respective design pattern leads to a latency improvement when executing the mock program with the refactored code. However, a few key points need emphasis:

1. The difference in latency between the original and the refactored code largely depends on the mock program and the compiler optimisations employed.
2. While the benchmarking in the preceding two sections lacks the statistical rigour of the benchmarking procedures in Chapter 4, it still serves its purpose: to showcase that under identical conditions (i.e., using the same mock program), utilising the LLDPA results in latency improvements.

<sup>2</sup>While this is not technically incorrect, it relies on the compiler to eliminate the unused branch.

# Chapter 6

## Epilogue

### 6.1 Contribution

In evaluating the achievement of the four research objectives, outlined in Section 1.2, the following critical self-assessments can be made:

1. Compiling numerous latency-reducing techniques presented by various domain experts into a formal catalogue of low-latency design patterns:
  - The patterns were catalogued in a highly structured manner, including illustrative implementation examples and comprehensive technical explanations.
  - However, due to time constraints, not all techniques proposed by the domain experts were formalised into design patterns. Only those deemed most relevant were selected for inclusion.
2. Benchmarking the catalogued design patterns:
  - The benchmarking process adhered to strict statistical standards, exceeding those commonly found in other code benchmarking experiments. Furthermore, the chapter features comprehensive analyses of unexpected anomalies observed during the experiments, offering in-depth insights to the reader.
  - Regarding the results of the experiments, five out of six<sup>1</sup> patterns have statistical evidence supporting their hypothesised latency-lowering effect.
3. Developing an application that scans C++ code for implementation opportunities of the design patterns:
  - The developed application effectively identifies refactoring opportunities, primarily due to its utilisation of the `clang.cindex` library for code parsing and a good mapping of the design patterns to identifiable characteristics within the code.

<sup>1</sup>Every pattern except the Denormalised Data Pattern.

- One limitation is that the LLDPA operates under certain restrictive assumptions, such as requiring all files to reside within the same directory.
4. Extending the application for automated code refactoring:
- Automated refactoring using Chat-GPT produced highly satisfactory outcomes during testing. However, the application's refactoring capabilities were only evaluated on two small toy projects.
  - Currently, the LLDPA exclusively supports Chat-GPT for refactoring, which might limit future usage of other, more advanced LLMs.

## 6.2 Future work

In future research endeavours, the applicability of the low-latency design patterns to other high-performance programming languages like Go and Rust should be investigated. This presents a notable challenge given that specific C++ features, upon which some of the patterns in this thesis rely, are implemented differently in other languages. For instance, Rust generics are slightly different to C++ templates.

Regarding the LLDPA, several things can or should be done in future development iterations:

- The application should undergo modifications to eliminate its current dependency on specific constraints for analysing C++ code.
- Further, the tool's capabilities should be extended to manage larger C++ implementation files during refactoring. This may require sending multiple prompts to Chat-GPT.
- Upon achieving robustness and scalability, the application should be empirically tested on real-world C++ projects, which typically exhibit greater complexity and larger source files.
- Lastly, if new design patterns emerge - focused on other quality attributes such as security - the core architecture of the LLDPA can be adapted to accommodate these patterns.

# Bibliography

- [1] S. Ghosh. *Building Low Latency Applications with C++: Develop a complete low latency trading ecosystem from scratch using modern C++*. Packt Publishing, 2023. ISBN 9781837634477. pages 1
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612. pages 2, 5, 17
- [3] Chandler Carruth. CppCon 2014: Chandler Carruth 'Efficiency with Algorithms, Performance with Data Structures'. YouTube Video, 2014. URL <https://www.youtube.com/watch?v=fHNmRkzxHws&list=PLPaC2bZS78qjvnOPGzu6s64fW2ZV0opph&index=1&t=2525s>. pages 2, 6, 32, 33
- [4] Nimrod Sapir. Core C++ 2019 :: Nimrod Sapir :: High Frequency Trading and Ultra Low Latency development techniques. YouTube Video, 2019. URL [https://www.youtube.com/watch?v=\\_0aU8S-hFQI](https://www.youtube.com/watch?v=_0aU8S-hFQI). pages 2, 27, 29, 30
- [5] Carl Cook. Pacific++ 2017: Carl Cook 'Low Latency C++ for Fun and Profit'. YouTube Video, 2017. <https://www.youtube.com/watch?v=BxfT9fiUsZ4&list=PLPaC2bZS78qjvnOPGzu6s64fW2ZV0opph&index=4&t=3990s>. pages 2, 9, 18, 27, 29, 46
- [6] Irene Aldridge. *High-frequency trading: a practical guide to algorithmic strategies and trading systems*, volume 604. John Wiley & Sons, 2013. pages 3
- [7] Christian Leber, Benjamin Geib, and Heiner Litz. High frequency trading acceleration using fpgas. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 317–322. IEEE, 2011. pages 3
- [8] Niklaus Wirth. A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3):32–39, 2008. doi: 10.1109/MAHC.2008.33. pages 5
- [9] Cheng Zhang. *An empirical assessment of the software design pattern concept*. PhD thesis, Durham University, 2011. URL [http://etheses.dur.ac.uk/859/1/Cheng\\_Zhang's\\_Thesis.pdf](http://etheses.dur.ac.uk/859/1/Cheng_Zhang's_Thesis.pdf). pages 5
- [10] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, fourth edition, 2013. pages 6, 8



- 
- [11] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994. pages 6, 7
- [12] Agner Fog. Optimizing software in c++: An optimization guide for windows, linux, and mac platforms, 2004–2022. URL [https://www.agner.org/optimize/optimizing\\_cpp.pdf](https://www.agner.org/optimize/optimizing_cpp.pdf). pages 6, 7, 9, 10, 13, 14, 15, 32, 33
- [13] Björn Andrist and Viktor Sehr. *C++ High Performance: Master the Art of Optimizing the Functioning of Your C++ Code*. Packt Publishing, Birmingham and Mumbai, second edition, 2020. pages 7, 8
- [14] Chandler Carruth. Understanding compiler optimization - Opening Keynote Meeting C++ 2015. YouTube, 2015. URL <https://www.youtube.com/watch?v=FnGCDLhaxKU&t=139s>. Accessed: 2023-05-31. pages 7
- [15] Matt Might. "c++ template metaprogramming with lambda calculus". Accessed: 2023-05-30, 2023. URL <https://matt.might.net/articles/c++-template-meta-programming-with-lambda-calculus/>. Factorial Example. pages 7
- [16] S. Donadio, S. Ghosh, and R. Rossier. *Developing High-Frequency Trading Systems: Learn how to implement high-frequency trading from scratch with C++ or Java basics*. Packt Publishing, 2022. ISBN 9781803243429. pages 9, 10, 29
- [17] Mateusz Pusz. code::dive 2017 – Mateusz Pusz – Striving for ultimate low latency. <https://www.youtube.com/watch?v=vzDl0Q91MrM>, 2017. Accessed: 2023-06-10. pages 9, 18, 22, 27, 29
- [18] Carl Cook. CppCon 2017: Carl Cook “When a Microsecond Is an Eternity: High Performance Trading Systems in C++”. YouTube video, 2017. URL <https://www.youtube.com/watch?v=NH1Tta7purM>. pages 9, 23, 25
- [19] Kevin Goldstein. Building low latency trading systems. Online, 2018. URL <https://www.youtube.com/watch?v=yBNpSq00oRk>. YouTube. pages 10
- [20] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016. pages 11, 12
- [21] Steve Scargall. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020. pages 11
- [22] Brown University. Lab4: Caching. <https://cs.brown.edu/courses/csci1310/2020/assign/labs/lab4.html?spm=a2c65.11461447.0.0.11147f65MAg7LI>, 2020. Accessed: 2023-06-05. pages 11
- [23] Luis Ceze. Caches, video 4: Cache organization. The Hardware/Software Interface Class by Luis Ceze and Gaetano Borriello. University of Washington. Available at: <https://www.youtube.com/watch?v=gD2pldZ3n-c>, 2016. Accessed: 2023-06-06. pages 12, 13
-

- 
- [24] R. Bryant and D. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 2 edition, 2010. ISBN 978-0-13-610804-7. ISBN-10: 0-13-610804-0. pages 14
- [25] Noel Llopis. Data alignment, part 1. <https://www.gamedeveloper.com/programming/data-alignment-part-1>, Mar 2009. Accessed: 2023-06-09. pages 16
- [26] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. pages 17
- [27] Jonathan Müller. Writing cache friendly c++. Meeting C++, 2018. URL <https://www.youtube.com/watch?v=Nz9SiF0QVKY&t=2604s>. YouTube video. pages 18, 35
- [28] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers, 2019. URL [https://liberty.cs.princeton.edu/Publications/isca19\\_frontend.pdf](https://liberty.cs.princeton.edu/Publications/isca19_frontend.pdf). Accessed: 2023-06-14. pages 18
- [29] Mike Acton. CppCon 2014: Mike Acton "Data-Oriented Design and C++". Online video, 2014. URL <https://www.youtube.com/watch?v=rX0ItVEVjHc>. Accessed on 2023-06-16. pages 21
- [30] Jonathan Keinan. CppCon 2018: "Cache Warming: Warm Up The Code". Presentation, 2018. URL <https://www.youtube.com/watch?v=XzRxikGgaHI&t=5s>. pages 25, 46
- [31] William Knottenbelt and Marios Kogias. Virtual functions and abstract classes, 2022. Lecture notes from the course "Object Oriented Design and Programming". Imperial College London. pages 29
- [32] IBM. Virtual functions (c++ only), March 2021. URL <https://www.ibm.com/docs/en/zos/2.3.0?topic=only-virtual-functions-c>. Accessed: 2023-08-21. pages 29
- [33] Carl Cook. The speed game: Automated trading systems in c++ - meeting c++ 2016. YouTube, 2016. URL <https://www.youtube.com/watch?v=u1OLGX3HNcI>. Accessed: June 17, 2023. pages 32, 33
- [34] Eli Bendersky. Memory layout of multi-dimensional arrays. <https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays>, 2015. Accessed: Jul 10, 2023. pages 33
- [35] David Gross. Counting Nanoseconds: Microbenchmarking C++ Code - Meeting C++ 2019, 2019. URL <https://www.youtube.com/watch?v=Czr5dBfs72U&t=1167s>. [Online; accessed 21-June-2023]. pages 34, 35
-

- 
- [36] Marloes Maathuis. Permutation tests. [https://stat.ethz.ch/lectures/ss19/comp-stats.phpcourse\\_materials](https://stat.ethz.ch/lectures/ss19/comp-stats.phpcourse_materials), 2019. Lecture slides in the course "Computational Statistics". pages 36
- [37] Ivica Bogosavljević. How branches influence the performance of your code and what can you do about it?, July 5 2020. URL <https://johnnysswlab.com/how-branches-influence-the-performance-of-your-code-and-what-can-you-do-about-> pages 50, 51
- [38] Ivica Bogosavljevic. The hidden performance price of c++ virtual functions - ivica bogosavljevic - cppcon 2022. YouTube video, 2022. URL [https://www.youtube.com/watch?v=n6PvvE\\_tEPk](https://www.youtube.com/watch?v=n6PvvE_tEPk). Accessed on 2023-06-28. pages 55, 56