Aalto University
School of Science

Master's Programme in Security and Cloud Computing

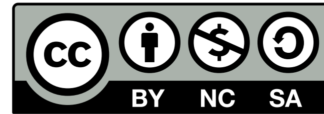
Measuring and improving the performance of a latency-sensitive Java real-time system

Andrea Amadei

Master's Thesis
2024

© 2024

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 4.0 International" license.



Aalto University
School of Science

Author Andrea Amadei

Title Measuring and improving the performance of a latency-sensitive Java real-time system

Degree programme Security and Cloud Computing

Major Security and Cloud Computing

Supervisor Senior University Lecturer **Arto Hellas**, Aalto University
Professor **Raja Appuswamy**, EURECOM

Advisor **Markus Aalto** (MSc), Supercell

Date 30 September 2024 **Number of pages** 84 **Language** English

Abstract

This thesis investigates the performance challenges of Java-based real-time systems, with a specific focus on latency-sensitive environments such as game servers. Java, widely recognized for its portability and safety, presents unique challenges when used in real-time applications due to the inherent unpredictability introduced by its platform. Key features of the JVM, such as just-in-time compilation and garbage collection, can cause significant variability in execution times, making it difficult to meet the strict timing requirements essential for real-time performance.

The study examines the impact of Java features on system latency and predictability through a combination of profiling tools, real-time monitoring, and controlled load testing. It also evaluates several optimization techniques, including garbage collection tuning, platform configuration adjustments, and programming techniques to ease just-in-time compilation.

The findings reveal that while these optimizations can reduce latency and improve predictability to some extent, they do not fully eliminate the unpredictability associated with Java. The research highlights that particular care should be taken when implementing features that might conflict with the inner workings of the Java platform. The broader implications of this study indicate that for applications where real-time performance is critical, further refinement of platform configurations is almost a necessity, although the performance advantages are worth the effort.

Keywords real-time, Java, JVM, GC, JIT, game server, performance, tickrate, predictability

Acknowledgements

First and foremost, I would like to acknowledge SECCLO, the Erasmus+ Programme, and Aalto University for their financial support, which made this thesis possible.

A great thank you to both my academic supervisors, Senior University Lecturer Arto Hellas and Professor Raja Appuswamy, for their precious advice and extensive knowledge, which ultimately helped me structure, develop, and write the thesis.

An immense thank you to Markus Aalto, Leo Linnamaa, Mikko Tiihonen, Robert Kamphuis, and the whole Server Tech team. Your help was priceless, and so was the knowledge I gained from this project. I am extremely grateful to have worked with so many talented individuals.

Finally, I would like to thank my family and friends, who accompanied me every step of the way.

Helsinki, 30 September 2024

Andrea Amadei

With the support of the
Erasmus+ Programme
of the European Union

Contents

Abstract 3

Acknowledgements 4

Contents 5

1 Introduction 7
- 1.1 Context and motivation 7
- 1.2 Thesis objective and scope 7
- 1.3 Structure of the thesis 8

2 Introduction to real-time computing 9
- 2.1 Definition of real-time computing 9
- 2.2 Characteristics of real-time computing 10
- 2.3 Classification of real-time systems 12

3 Game servers as real-time systems 14
- 3.1 The role of game servers 14
- 3.2 Inner workings of real-time game server 15
  - 3.2.1 Definition of tick 15
  - 3.2.2 The importance of tickrate 16
  - 3.2.3 Composition of a tick 17
  - 3.2.4 Advanced tickrate techniques 21
- 3.3 Classification of real-time game server 22
- 3.4 Meaningful metrics of game server performance 23
  - 3.4.1 General assumption when measuring performance 23
  - 3.4.2 Tick-related metrics 24
  - 3.4.3 Load-related metrics 26
  - 3.4.4 Ideal metrics in a perfect scenario 27

4 Real-time Java 28
- 4.1 Introduction to the Java platform 28
- 4.2 Features and challenges of real-time Java 29
- 4.3 Inner workings of the Java Virtual Machine 32
  - 4.3.1 Just-In-Time compilation 32
    - 4.3.1.1 Types of JIT compilers 33
    - 4.3.1.2 Tiered compilation 34
    - 4.3.1.3 Speculation and Deoptimization 36
  - 4.3.2 Memory management and garbage collection 37
    - 4.3.2.1 Garbage collection 40
    - 4.3.2.2 Types of garbage collectors 42

5 Methods 45
5.1 Iterative exploratory data analysis and problem solving 45
5.2 Metric collection 45
5.2.1 Prometheus real-time monitoring 45
5.2.2 Java Flight Recorder profiling 47
5.2.3 JVM diagnostic logging 48
5.3 Load-testing setup 48
5.3.1 High-level description of the system 48
5.3.2 Comparative testing 49
5.3.3 Automatic test case generation 50
5.3.4 Repeatability and test validation 51

6 Evaluation and Results 52
6.1 Game engine problems 52
6.1.1 Introduction to the analysis method 52
6.1.2 Problem analysis 52
6.1.3 Solutions 58
6.1.3.1 Speeding up the JIT compiler 58
6.1.3.2 Disabling speculations with Zing JVM 58
6.1.3.3 Making the application easier to speculate 60
6.1.4 Evaluation 60
6.2 Runtime and processor evaluation 62
6.2.1 Introduction to the analysis method 62
6.2.2 Evaluation of Java version 63
6.2.3 Evaluation of faster processors 65
6.2.4 Evaluation of disabling kernel-level mitigations 67
6.3 JVM fine-tuning 68
6.3.1 Introduction to the analysis method 68
6.3.2 Evaluation of garbage collection tuning 68
6.3.3 Evaluation of better tick-scheduling strategy 70

7 Discussion 73
7.1 Research objectives 73
7.2 Comparison with existing literature 74
7.2.1 Java warming performance issues 74
7.2.2 Java version evaluation 74
7.2.3 Cloud instances evaluation 75
7.2.4 Linux kernel mitigations evaluation 75
7.2.5 Garbage collection evaluation 75
7.3 Limitations of the work 76

8 Conclusions 77

References 79

1 Introduction

1.1 Context and motivation

In today's digital landscape, real-time systems are increasingly integral to the performance of numerous applications, particularly in areas where latency and response time are critical [12]. Real-time systems are software applications subject to stringent time constraints. Unlike traditional computing systems, real-time programs must guarantee a response within specified time constraints, often referred to as "deadlines" [15].

One such domain is online multiplayer gaming [16], where servers must process and synchronize data for hundreds or even thousands of players simultaneously, ensuring a seamless and engaging experience. However, achieving low latency and high performance in such environments is challenging, especially when using Java, a language traditionally favored for its portability and safety rather than its real-time capabilities [35]. Despite its widespread use, Java's inherent features, such as just-in-time compilation and garbage collection, can introduce unpredictability, making it challenging to meet the stringent timing requirements of real-time systems.

1.2 Thesis objective and scope

The overarching aim of this thesis is to enhance the performance of Java-based real-time systems, particularly those that are highly sensitive to latency, such as game servers. All testing and improvements have been performed on a currently active multiplayer online game for mobile phones with millions of players. In this game, players simultaneously engage in online battles with up to ten players. Players can also choose from various game modes, some of which are periodically available. This goal is pursued by thoroughly examining the performance bottlenecks inherent in these systems, with a specific focus on the challenges posed by the Java platform.

The research is guided by several key objectives. First, it seeks to identify the primary performance bottlenecks in a Java-based real-time system, particularly those operating in latency-sensitive environments. It also explores the impact of JVM-specific features, such as garbage collection and Just-In-Time (JIT) compilation, on the predictability and latency of these systems. Furthermore, the thesis investigates what optimization strategies can be implemented to mitigate the adverse effects of these JVM features on real-time performance and assesses the effectiveness of these strategies in enhancing the overall performance and predictability of a real-time Java system.

The scope of this research is concentrated on a detailed case study of a real-time game server architecture, which serves as a practical and relevant context for the investigation. This study involves a thorough examination of the server's performance across various conditions, utilizing profiling tools and real-time monitoring systems to gather data. While the focus is on game servers, the methodologies employed and the conclusions drawn are intended to be applicable to other Java-based real-time systems that share similar challenges with latency and performance constraints.

1.3 Structure of the thesis

This thesis is structured as follows: Chapter 2 provides an overview of real-time computing, including the unique challenges posed by real-time applications. Chapter 3 explores the specific context of game servers as real-time systems, detailing their architecture and performance metrics. Chapter 4 discusses the Java platform's suitability for real-time applications, with a focus on the role of the JVM. Chapter 5 outlines the methodologies used for performance evaluation, including monitoring and testing approaches. Chapter 6 presents the evaluation result, highlighting the effectiveness of various optimization strategies. Finally, Chapters 7 and 8 offer a discussion of the findings, conclusions drawn from the research, and recommendations for future work.

2 Introduction to real-time computing

2.1 Definition of real-time computing

A real-time computer system can be defined as a system that responds to external events within a predictable amount of time [15, 46]. In a real-time system, the time window the system should respect when performing a computation over external inputs is known as a "deadline". In real-time systems, deadlines must be respected at all costs, regardless of system load or other unpredictable external factors. Depending on the nature of the system, missing a deadline can degrade the system's quality of service and, in some more extreme cases, compromise the system. In essence, real-time systems attempt to guarantee response times to specific inputs within a predefined and often strict time frame.

Real-time computing defines the concept of "task" [12, 46]. A task is a unit of workload that must be executed within a predetermined deadline. Most real-time systems are designed to manage multiple tasks simultaneously, each with different deadlines, constraints, execution times, and precedence [12]. Each task must respect its deadline despite coexisting with other tasks with similar requirements; therefore, execution time must be carefully planned to ensure that each task respects its individual time frame. The system component in charge of planning each task's execution window is called the "scheduler" [45].

The scheduler, or sometimes "orchestrator", is a critical component of real-time computing systems responsible for managing the execution of tasks based on their timing requirements. Given a set of tasks and the resources in the system, scheduling is the process of determining when and for how long each task will be allowed execution [46]. The scheduler's role is crucial since it can determine the success or failure of tasks and the overall system. For example, suppose the scheduler wrongly estimates the actual time of execution of a task or else wrongfully allocates a task, causing it to miss its deadline. In that case, consequences can be severe, even catastrophic, since many real-time systems are used in extreme safety-critical applications, such as the control system of fly-by-wire aircraft [12, 46]. In some real-time computing systems, tasks can sometimes be performed under tight temporal requirements, from milliseconds down to microseconds of precision. This makes the scheduler's job extremely important and challenging, given how difficult it might be to correctly schedule thousands of different tasks and how a single mistake can lead to disastrous consequences. Unfortunately, most real-life scheduling problems are NP-hard; therefore, no perfect solution exists [9].

Because different systems and tasks can have different planning requirements, real-time systems can pick from a variety of different scheduling techniques [12], each with its advantages and disadvantages. Schedulers are an intricate part of the system that needs to be gotten right, primarily because, in many cases, extensive testing is the only feasible option to fully validate a scheduler. Hardware can also impact scheduler effectiveness, making it more difficult since different architectures can be suited differently. For these reasons, scheduling techniques must be picked carefully,

as they can seriously impact the stability of the whole system.

2.2 Characteristics of real-time computing

Real-time computing systems differ from more traditional computing architectures in several ways [46]: time is limited, tasks can have precedence, latency must be minimal, and therefore, compute resources cannot be fully utilized, predictability is a key requirement. Here is a detailed analysis of these aspects of real-time computing.

Time is a precious resource

In a real-time system, tasks must be allocated and scheduled to meet deadlines. Generally, "the correctness of a computation depends not only on the logical correctness but also on the time at which the results are produced" [46]. Because time is limited and sometimes insufficient to satisfy all deadlines, scheduling must be performed carefully to avoid missing deadlines. In some cases, running a constantly under-loaded system can be preferable rather than risking missing a single deadline since wasting system resources can be better than the consequences of a missed deadline.

Some tasks take precedence

In real-time computing, not all tasks are equal. Because of the scarcity of time, some tasks might be unable to meet their deadlines due to unforeseen external events, such as a sudden decrease in computing performance. If this is possible, schedulers can take advantage of a priority-based scheduling system [29]. In such systems, each task is assigned a priority level, with the understanding that higher-priority tasks will take precedence over lower-priority ones if it's anticipated that there might not be enough time to meet every task's deadline.

Priorities are specifically designed to mitigate damage as much as possible in case of task mis-scheduling. However, priorities will not prevent damage completely since even missing the deadline of a low-priority task can have significant consequences. This approach differs from non-real-time applications since, in those cases, a late execution is often preferable to a non-execution.

Latency must be minimized

In a real-time system, latency is the time elapsed between the reception of an external event and its consequent response to that same input. The goal of a real-time system is to minimize the latency of specific tasks while also ensuring all deadlines are respected [35].

This is not always easy because tasks must be executed closer to their deadlines to reduce latency. This approach effectively decreases the staleness of data used in the computation, reducing latency. However, it also increases the possibility of a missed deadline since the possible execution window is significantly reduced, making scheduling mistakes harder to recover from.

Computing resources cannot be fully utilized

Many traditional computing systems try to maximize the usage of computing resources to save on hardware costs. The strategy is simple: assign as many tasks as possible to a single computing node and later eliminate those remaining empty nodes.

Unfortunately, this is not possible when dealing with real-time systems for several reasons. Primarily, scheduling too many tasks on a single computing node might overload it, causing a higher probability of missed deadlines. Scheduling too many tasks on the same node could also mean that not all tasks requiring a low latency could get executed with fresh enough data since the scheduler would have no way to accommodate every task. Another reason resource optimization is avoided in real-time computing is fault tolerance [46]. Because a node failure can cause the complete disruption of all tasks running on it, squeezing as many tasks as possible onto a single node increases the damage in case of a fault (commonly known as *blast radius*). For these reasons, hardware resources cannot be easily optimized in real-time systems, and doing so can have serious consequences in case of adversities.

Predictability is key

Most real-time systems work best when dealing with predictable operations [29]. Depending on the strictness of the system, predictability can be defined in four major ways, in order of strictness [46]:

- **Strict certainty.** A real-time system is defined as *predictable* if it is possible to demonstrate at design time that all its timing constraints can be met 100% of the time. If the timings of each task can be defined as strictly certain, then scheduling is made easier even in adverse conditions since respecting each deadline is mathematically guaranteed. Unfortunately, this strict approach is unrealistic in complex systems. In order to mathematically prove strict certainty, each task and its properties must be known before execution. Furthermore, guaranteeing certainty even in worst-case scenarios can be even more challenging since some systems might require extensive re-work to comply with an unrealistic requirement. For these reasons, proving strict certainty is unfeasible in most systems.

- **Probabilistic guarantee.** A real-time system is defined as *probabilistically predictable* if at least a certain fraction of tasks are likely to meet their deadline according to a predetermined threshold. Unless when dealing with particularly strict systems, having the probabilistic guarantee that each task will respect its deadline is often enough while also being far easier to prove. In some other cases, it could still be plausible to demonstrate that some critical tasks can be completed with a 100% guarantee while only proving others to a probabilistic extent. However, proving a probabilistic guarantee also requires the knowledge of all tasks at design time, therefore this approach might still be too strict for some complex systems.

- **Run-time deterministic guarantee.** A real-time system is defined as *run-time deterministically predictable* if the scheduler is able to determine at run-time if the constraints of each task can be satisfied without sacrificing the constraints of any other tasks in the process. Unlike the previous ones, this approach is easily feasible since it does not require prior knowledge of each running task, just its execution properties. However, it does not guarantee any predictability at design time, which might be problematic in some stricter systems.

- **Run-time estimate.** When a system cannot prove either certain or probabilistic predictability, it could still perform a run-time estimate, given tasks are periodic and share some similarities with each other. To achieve this, each task is classified based on its properties, and the system measures its execution time. After a long run time, it might be possible to identify some patterns to define both a common and worst-case timing constraint. While estimating task constraints at run-time can be useful for scheduling purposes, it does not guarantee any property whatsoever, therefore it is by far the least effective predictability measure.

In a real-time system, deadlines are crucial, and it is easier for the scheduler to plan predictable tasks correctly since predictable tasks have a well-known execution time. Predictable tasks also have the advantage of allowing for a better-defined execution plan, especially if most tasks are executed at predictable periodic intervals. On the contrary, if a system is not easily predictable, scheduling becomes harder, and consequently, deadline violations can become more common and harder to tolerate. For these reasons, predictability is a crucial property of a real-time system, although not always possible.

2.3 Classification of real-time systems

Real-time computing systems can be classified into three categories based on the consequences that might occur when a deadline is missed [28, 46].

- **Hard real-time systems.** In a *hard real-time system*, missing a single deadline will cause a total system failure. Hard real-time systems usually must provide strict guarantees and fault tolerance in all cases and conditions; therefore, no deadline can ever be missed. Designing and building a hard real-time system is often a tedious task since all aspects of the system, both hardware and software, must be meticulously designed and tested. Total fault tolerance against both hardware and software failure must be guaranteed; therefore, the real-time components of the system must be able to operate correctly in every circumstance, no matter how extreme or unlikely those conditions are. Because of these requirements, hard real-time systems are usually expensive and challenging to engineer and, therefore, only used when strictly necessary. Some examples of these systems are medical devices such as pacemakers, control systems of fly-by-wire aircraft, industrial process controllers used in assembly lines, and many others [15].

- **Strict real-time systems.** Missing a deadline in a *strict real-time systems* (sometimes also referred to as *firm real-time system*) can have some serious consequences, but unlike in hard real-time systems, it doesn't cause a total system failure. In a strict real-time system, the result of a task becomes irrelevant if the deadline is exceeded; therefore, late tasks can seriously affect the system's accuracy, but they do not impede its correct functionality [46]. In some strict real-time systems, missing a large number of deadlines in a short period of time could also have serious consequences, although no official definition is given due to the nebulous nature of the term "strict". In general, strict deadlines can tolerate a certain degree of failure, unlike hard deadlines, which can have catastrophic consequences when missed. An example of strict real-time systems is high-frequency trading systems, whose algorithms produce useful data only for short periods of time, after which the rapid and unpredictable nature of the stock market nullifies its validity. Although obtaining stale data is not desirable, missing a deadline doesn't necessarily mean failure. For this reason, the system can be classified as strict. Non-periodic tasks are also often classified as strict since estimating an execution deadline can be non-trivial given the uniqueness of the task, which means that missing a deadline is to be expected, although not desirable or acceptable for the system.

- **Soft real-time systems.** All systems that can tolerate missed deadlines without any serious consequences are considered *soft real-time systems*. Unlike in hard and strict real-time systems, missing a task deadline doesn't cause a system failure, nor does it render the result of the task useless. However, missing a deadline is still undesirable since it might impact the system's reliability or its user perception, although without any serious consequences. Most complex and non-critical systems tend to be soft real-time systems simply because estimating and handling deadline failures can be extremely difficult; therefore, they are avoided if not needed. Examples of soft real-time systems are live video streaming or video games. In both cases, missing a deadline can disrupt the user experience by displaying a slightly delayed video feed, but no serious consequences arise.

3 Game servers as real-time systems

3.1 The role of game servers

Multiplayer games often require a centralized server to process and synchronize state between game clients [8, 57]. In this architecture, each player installs and executes the game engine on their own device, defined as the game client, which runs all basic game features. Each client will then send and receive information to the centralized system, known as the game server, which will process them and keep all game players synchronized. The role of the game server is fundamental to the correct working of the game experience for the following reasons.

- **Client synchronization.** Game clients need to stay synchronized with each other for the game to be correctly playable [18]. Without synchronization, players cannot possibly know the state of other clients; therefore, the game cannot proceed. In some cases, game clients must be tightly synchronized with each other in order to provide a seamless simulation that seems continuous in time to the eyes of the player. This game experience, although possibly more realistic, also requires a more complex infrastructure to shorten delays as much as possible [30]. In some cases where synchronization delays are near zero, the game server can become a true real-time system that must process game updates and keep the clients synchronized within short, well-defined deadlines.

- **Game logic processing.** Multiplayer games must define and implement a set of rules for the game to be playable correctly. Game rules are the fundamental principles that define how the game is played, including objectives, win/lose conditions, and allowable actions, such as moving, attacking, and collecting items. The series of instructions that implement the game rules are often referred to as *game logic code* [27] or *game script* [3]. To ensure the correct unfolding of the game, the server must be able to simulate the game logic entirely and then notify all game clients about the simulation results [27]. Because the game server is the only component of the infrastructure that can see the state of all clients simultaneously, only the server is capable of fully determining the correct outcome of a certain action performed by any of the game clients, especially if said action can influence more than one player at the same time.

- **Game authority.** The game server has an authoritative role on every client for similar reasons of centrality [8], as previously mentioned. Because the game server is the only component of the infrastructure that can see the state of all clients simultaneously, only the game server can resolve conflicts between clients if discordant actions are submitted concurrently. When this happens, the server must be able to fully determine the correct outcome according to the rules and keep track of the state of the game, independently from what each party demands. Furthermore, in response to any authoritative request from the server, each game client must oblige accordingly, even in the scenario when the game client and server disagree on the result of a logic computation. As a

consequence of the absolute authority of the server over the clients, the game client's own incomplete computations are irrelevant to the server since it alone has the power to fully determine the outcome of a certain action, despite the clients' claims. This authoritative approach is fundamental because game clients cannot be trusted [18, 27]. In fact, since the server only has control over the sent data but has no control over the actual data residing on the client, the server has no possible way of knowing exactly what is happening inside a client at any given time. Consequently, the server would also have no way of knowing if the client logic code has been tampered with, meaning that a player could cheat by modifying the rules of the game and then sending incorrect computation results to the server, which would simply trust it without being able to verify the correctness. For these reasons, maintaining authority over the clients allows the server not only to guarantee the correctness of the game but also to prevent the clients from cheating.

- **Fog of war.** *Fog of War* (also referred to as *FoW*) is a game mechanic adopted by some strategic games that prevent players from being fully aware of the enemy's units [21]. Despite its peculiar game implications, FoW also requires some expedient from the technical point of view since clients must not be aware of the full game state but only of the information that concerns them. In this scenario, the game server must treat each client as an individual entity that can only receive a subset of the total information. Receiving information that should not be seen by the client, just like in a system where the server is not fully authoritative over the clients, can allow fraudulent players to cheat by utilizing the information that FoW would have prevented them from obtaining. Without a central game server, implementing FoW or a similar obscurity mechanism would be nearly impossible since only a centralized and authoritative entity can have the power to decide which player should receive which information. Furthermore, as similarly described above, trusting the clients to filter out non-obtainable pieces of information could be subject to cheating, hence the requirement of a game server.

As previously mentioned, the server can become a real-time system in some cases where the game needs constant and fast updates between clients. This scenario will be the main focus of this thesis.

3.2 Inner workings of real-time game server

3.2.1 Definition of tick

Most game servers are effectively complex real-time systems. A real-time game server, to operate correctly, should handle the following operations in order [8, 60]:

1. **Input processing**: acquire and process client inputs to be aware of actions performed by the players.

2. **Game logic**: run the game logic against the previously acquired inputs and obtain the results of the actions as a new game state.

3. **Output processing**: send the computation results to clients that will receive, process, and react accordingly.

The combination of these three tasks executed in order is called a "tick", which is the server's discrete steps to simulate the game [48]. Contrary to popular belief, most game servers are tick-based, which means the game progresses in finite and measurable steps instead of one continuous loop that lasts the whole game. However, by updating the game state at a fast pace, players are given the illusion of continuity, just like movies trick the viewer into believing they are watching a moving image by displaying consequential versions of the same subject in a very short amount of time. The amount of ticks processed in a period of time is commonly known as "tickrate" [47] and can be measured in *tick/s* (ticks per second) or *Hz*.

3.2.2 The importance of tickrate

In most scenarios, each of these tasks must be performed in rapid succession, even at sub-second intervals. Popular games adopt a tickrate that ranges from *20 tick/s* (tick processing time of *50 ms*), such as *Call of Duty: Warzone*, to *128 tick/s* (tick processing time of *7.8125 ms*), such as *Valorant* [13]. By increasing the tickrate, the precision of the simulation likely increases because the server will update the state of the game according to the client inputs at a faster pace. On the contrary, decreasing the tickrate usually decreases the precision since the state of the game will be updated less often.

However, increasing or decreasing the tickrate of a game server does not necessarily mean that the game experience will improve or degrade. Online games are complex systems that exchange data packets between clients and servers over the internet. Unfortunately, this method of communication is imperfect, as communication delays, packet loss, and other problems can occur [57]. For these reasons, a higher tickrate might not guarantee a better player experience if data packets are frequently lost since part of the communication between client and server will never reach its destination anyway. On the other hand, a lower tickrate does not imply a worse player experience either. For example, a person playing a mobile game over a mobile network might not notice any difference in performance if the tickrate increases, given the network's high latency (time elapsed between the sending and receiving of a packet). Therefore, a high tickrate might provide the same gaming experience as a lower one.

It is also important to note that to reach higher tickrates, the server must be able to process ticks at least as fast as the tickrate implies. For example, by adopting a tickrate of *50 tick/s*, the server must be able to perform at least one state update every *20 ms*, which, depending on the time required to run the game logic code, might be simply not possible on slower hardware. A higher tickrate also implies more packets being sent and received by the server, which also means increased network traffic.

For these reasons, the tickrate of a game server must be carefully considered. Although a better tickrate might provide a better player experience, this is not guaranteed

and can vary deeply between games. Furthermore, higher tickrate servers can have greater associated costs to run and maintain, which might not be justifiable from a business perspective if the player experience is hardly impacted by it. Finally, for some games that are particularly insensitive to latency or fast-paced actions, the tickrate might not matter, such as turn-based games.

3.2.3 Composition of a tick

As previously mentioned, a tick comprises three distinct phases: input processing, game logic, and output processing, as shown in Figure 1.

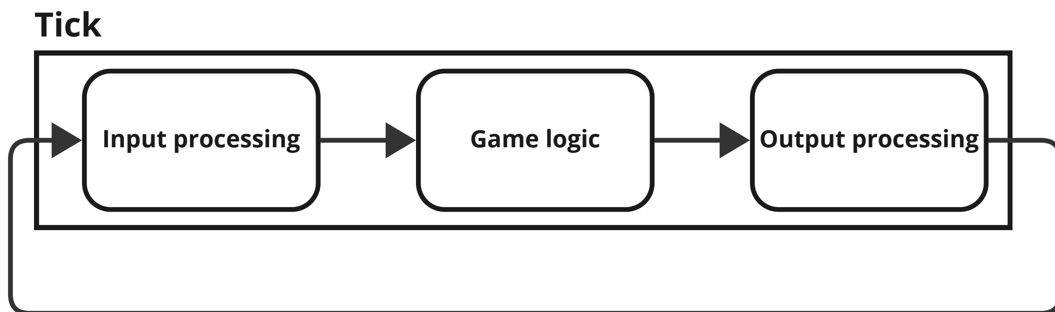


Figure 1: A tick is composed of three distinct phases (input processing, game logic, output processing) computed cyclically.

Input processing

Input processing is the first phase of the game tick, where the game captures and handles user inputs from various devices, such as keyboards, mice, game controllers, and touchscreens, sent to the server by the clients. During this phase, the game collects data on the player's actions, such as key presses, mouse movements, and button clicks. Input processing has three main goals:

- **Input decrypting and decoding.** During the game execution, the server will receive several client inputs through the internet. For the inputs to be meaningful, the server must first parse their content into a usable data structure and, if inputs are encrypted, decrypt them [8].

- **Input filtering.** After inputs are collected, decrypted, and decoded, the server must ensure their validity [17]. The server can do this by verifying the received data's correctness, ensuring that no packets were lost in the process and that inputs refer to the correct game tick. If an input is deemed incorrect by the server, it is dropped. From a client's point of view, submitting an incorrect or irrelevant tick equates to not submitting any input at all. In some more severe cases, servers can also enforce penalties for submitting wrong commands on purpose, such as kicking the player out of the game. In other cases, the server

can also enforce a rate-limiting policy, which only allows the clients to send a certain amount of packets per time unit. In either case, these actions are performed to ensure clients don't submit fraudulent commands to disrupt the system, such as during Distributed Denial of Service (*DDoS*) attacks, in which a malicious party can send thousands of fake user inputs to a game server to cause a slow down, or worse, cause the system to go offline. According to the authors of [58], the gaming industry is currently the most targeted sector of DDoS attacks.

- **Input buffering.** Because of the intermittent nature of ticks, player inputs cannot be processed immediately by the game server. Instead, inputs will be acknowledged immediately as they reach the server, but at first, only buffered into temporary storage until the next tick is ready to be processed [8, 48].

  In most use cases, servers tend to process inputs as they come in, with an architecture called *event-driven*, especially in the case of web servers that provide the user with web pages. However, game servers are usually an exception because they process a certain type of workload, which is hardly compatible with this architecture. Furthermore, an event-driven architecture works best when the server inputs are idempotent [14, Chapter 9.2.2], which means that the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. Unfortunately, most games are not idempotent because repeating the same inputs more than once has different effects than a single button press. For these reasons, game servers might react poorly to the order or timing of inputs without input buffering, especially since, in most video games, the state is updated as a finite process instead of an event loop where requests are processed as they come in, and the game is updated accordingly. Buffering inputs, however, partially solves this problem since inputs can be used when needed and in the order of arrival without requiring a more complicated logic code that can handle the event-driven paradigm.

Game logic

Game Logic is the second phase of the game tick, where the core mechanics and rules of the game are processed based on the inputs received and the game's current state [48]. This phase involves updating the game state and handling various in-game systems. The primary update needed is the game rules. However, other computations could also be required depending on the game, such as physics, collision detection, non-playable characters (NPCs) simulation, bot AI, and many more [18, 56]. Key tasks in this phase can include the following operations:

- **Game rules.** The server advances the game by applying the rules dictated by the logic code [48]. This is the main step in every game server since this actually allows the game to progress. Different types of games can implement this step differently, although the core functionality remains the same. In many games,

where the players control characters moving and interacting in space, the game recomputes the position, velocity, and other movement attributes of the game entities, such as characters, objects, enemies, and projectiles [8].

- **Physics and Collision Detection.** Many modern games, especially action titles, involve physics simulation features [32, 56]. The term game physics can be quite misleading since only classical mechanics are typically referenced when discussing physics in a game, which are the laws governing the movement of large objects under the influence of gravity and other forces, such as mass, inertia, bounce, and buoyancy. Although these laws have largely been superseded by newer theories, such as relativity and quantum mechanics in academic physics, they are still often employed in games to give objects the properties of solid objects. In these cases, every type of physical law implemented in a game must be simulated on the server, often with a great computational effort. Over time, the challenge of achieving realistic physics and the recurring need for similar effects across various games has led developers to seek general solutions that can be reused. Reusable technology must be quite general; for example, a ballistics simulator designed only for arrows can have arrow behavior hard-coded into it. If the same code also handles bullets, the software must abstract from specific projectiles and simulate the general physics they all share. This is referred to as a *physics engine*, a common piece of code that generally understands physics but is not programmed with the specifics of each game's scenario. Many complex games prefer the physics engine approach, which must also be simulated on the server, often with even greater computational complexity and an opaque system, since many physics engines are closed-source commercial products acting as middleware on the server. Together with game physics, many games also tend to process collision detection, which is the effort to understand when two or more objects collide. While game physics often influences objects in the game scenery or that the player can interact with, collision detection can also impact the player's movement since many games implement special features to push entities away from each other in case of a collision. For example, if two players decide to move to the same position (while the game rules don't allow such behavior), the game server must tell the clients that each player was moved away from that position. While collision detection might look like a simple task, it can become quite sophisticated in the case of complex entity models, such as a realistic human body, or complex player movement, such as simulated walking instead of simply moving across space. For these reasons, collision detection can sometimes also happen inside the physics engine.

- **NPC simulation and bot AI.** In video games, a *non-playable character* (NPC) is a character not controlled by a real player [56]. On the other hand, a *bot* (short for robot) is a simulated entity that plays the game in the place of another human player [21]. Both NPCs and bots utilize various types of artificial intelligence (AI) systems that play video games like a human would. Simulating human-like behavior can be quite complex, especially for bots, which often need to act

indistinguishably from a real human player. The algorithms that control both bots and NPCs must be implemented onto the game server, especially when game clients don't have to know the difference between a player-controlled and AI-controlled entity.

Output processing

Output processing is the third and final phase of the tick, where the game server prepares and presents the updated game state to the players [8]. Once all inputs from all players have been collected and the new game state is ready, each client must be notified of the changes by receiving a state update from the server. Different games can use different techniques to inform the clients about the new state, in particular:

- **Full state update.** The new state is sent to the clients in full [27]. This is usually the easiest method to implement since it doesn't require any particular technique to track changes. Once the client receives and decodes the new state, the previous state is discarded, and the new one is visualized instead. Despite its simplicity, this method can be problematic if the game state is big in size since sending big packets multiple times per second can easily overwhelm a network. On the other hand, this approach is completely reliable around packet loss. Even if packets are never received or received in the wrong order, newer state updates already contain all the information necessary to compensate for the fault. For these reasons, the full state update approach is simple and reliable, although problematic if the game state is bigger.
- **Differential state update.** The server computes all the differences between the previous and current state and sends the differences alone to the clients [54]. Despite being more complicated than the previous one, mostly because the server needs to understand the differences between consecutive states, this approach provides many advantages. The main difference is that by only encoding differences instead of the full state, both the packet size and the computational complexity drastically decrease if the state doesn't vary much between ticks. On the contrary, if the game state varies too much between consequent ticks, any possible benefit is canceled or even reverted. Furthermore, this method also requires better handling of packet loss since all updates must be delivered and strictly ordered. In fact, if a single packet is lost or out of order, the client will remain in an out-of-sync state since the client cannot possibly recover any of the lost information from newer updates. For these reasons, the differential state update is more performant than the previous method despite being more complex and requiring a better system for packet delivery.
- **Hybrid state update.** Some games can implement a hybrid method between the two, where the server can send full and differential updates to clients, depending on the scenario. In general, differential updates will be used in most situations unless the difference between two ticks is too big for any benefits deriving from this approach, in which case the server will send the full game state instead.

In some cases, the clients might also request a full state update specifically, for example, if the client understands that some information was lost. If used correctly, the hybrid method can perform best in both situations.

Regardless of the update technique implemented, the steps required for output processing are:

- **Output collecting.** Before being sent, each update must be collected from the results of the game logic part of the tick. Despite looking trivial, update collection must be performed carefully, especially if different players can receive different content [21]. If done incorrectly, some game clients might be able to obtain the information they should not be able to access otherwise, such as when the *Fog of War* mechanism is used. If done correctly, advanced output-collecting techniques can help reduce the computational complexity of parts of the logic code. For example, if the game supports it, some common parts of the state update can only be processed once and collected multiple times for all targeted clients, increasing the game server's efficiency.

- **Output encoding and encrypting.** Just like during the first step of input processing, outputs need to be encoded into a meaningful structure to later be decoded by the clients. If encryption is used, outputs must also be encrypted after being encoded [30].

- **Output sending.** Outputs of the computation are finally sent to the clients over the Internet. Depending on the transport protocol, the packets might have no guarantee of reaching the destination or arriving in the correct order [30]. If this is the case, some game servers might implement a retry mechanism to store and re-send some packets if lost or if the client specifically requests it. Some other games might also store the packets before sending them to be analyzed later.

3.2.4 Advanced tickrate techniques

Modern game servers can sometimes use more advanced tickrate techniques to overcome some limitations of the original system and make the server more performant. Here are a few of them.

- **Variable tickrate.** Maintaining a high tick rate can be expensive, as explained in Section 3.2.2. However, a high tick rate could also benefit player experience in some scenarios but not in others. To address this issue, some game servers can adopt a variable tickrate technique, where the tickrate can change dynamically depending on multiple factors. For example, some games might use a lower tickrate for non-competitive game modes to save cost while using a higher tickrate for competitive ones to provide a better player experience when it matters. For example, the game *Call of Duty: Modern Warfare* defaults to a tickrate of *20 tick/s*, but drops it to *12 tick/s* when playing a private match between friends [13].

- **Multi-tick system.** In some games, different steps of the game logic can be more or less computationally expensive. For example, performing a physics simulation for game objects can be more demanding than updating the player's position and movement. However, providing such big precision in a physics simulation might not be needed for a good player experience, although providing a precise player movement might be. The solution to this problem could be a multi-tick system, where the server processes different actions of the game logic at different tickrates [48]. For example, updating the physical simulation of a game at half the tickrate of the rest of the game logic could reduce cost while maintaining a high-quality experience where it is noticeable and a lower-quality one where it is less impactful.

- **Sub-tick system.** To provide for a higher input precision while keeping the tickrate low, some game servers might opt for a sub-tick system, where inputs can be measured more precisely than the precision of a tick [8]. One way to achieve this is to label inputs on the client with a timestamp that represents a more precise timing than what the tick would provide. All game logic is then processed as normal, except that the extra input labeling could help the server understand when the input happened despite being part of a less precise tick time. For example, if a client labels a player's movement as "*x* after tick *n*", the game server might be able to treat the input as it happened before tick *n+1*, even though the actual player input is received along all other inputs of tick *n+1*.

3.3 Classification of real-time game server

A game server can be categorized as a real-time system based on its need to process and respond to events within strict timing constraints to ensure players' seamless and interactive experience. According to the definitions in Section 2.2 and 2.3, game servers usually fall into these categories.

- **Soft real-time system.** A game server should be classified as a *soft* real-time system due to the nature of its operational requirements and the acceptable levels of performance variability. First, most game servers must process and respond to player inputs, update the game state, and communicate changes to all connected players within milliseconds to ensure a smooth and interactive experience. While it is crucial to minimize delays to maintain game responsiveness, the system can tolerate occasional missed deadlines or slight delays without causing catastrophic failure. This tolerance for occasional delays aligns with the characteristics of a soft real-time system. Furthermore, the primary goal of a game server is to maintain a high-quality player experience, which includes providing responsive controls and real-time feedback. While consistency is important, the occasional minor delay does not render the game unplayable. Instead, the system is designed to handle such delays in a way that minimizes impact on the player, fitting the definition of a soft real-time system. Game servers must also handle variability in network conditions, player actions, and computational load. Soft real-time

systems are designed to manage such fluctuations gracefully, ensuring that performance remains within acceptable bounds. This flexibility is crucial for game servers, adapting to changing conditions without significantly disrupting gameplay. Finally, in a soft real-time system, the focus is on maintaining an acceptable quality of experience rather than meeting strict, non-negotiable deadlines, therefore game servers could not be classified as either *hard* or *strict* real-time systems.

- **Run-time deadline estimate.** The vast majority of game servers can only provide a run-time estimate of deadlines due to the inherent variability and unpredictability of the factors influencing their operation. The first key reason limiting game servers is the unpredictability of the network. Game servers rely on network communication to exchange data with clients. Network latency can fluctuate due to various factors, such as internet traffic, bandwidth limitations, and routing issues. These fluctuations make it difficult to predict precise deadlines, therefore providing any strict deadline seems almost impossible. Furthermore, game servers are constrained by the game logic they need to run. In some cases, it can involve complex computations, including physics simulations, AI processing, and collision detection. The time required for these computations can vary based on the in-game scenarios and the number of entities involved. In some cases, game servers can also handle very computationally different events triggered by players or game states. The complexity and frequency of these events can change dynamically, impacting the server's processing time. Because most of these events depend entirely on player inputs, it is incredibly difficult to estimate their impact before run-time. For these reasons, a game server can only provide a run-time estimate of deadlines due to its highly dynamic and unpredictable nature.

3.4 Meaningful metrics of game server performance

3.4.1 General assumption when measuring performance

The analyzed game server simulates real-time *battles* between players. When a player enters the game, they will prompted to look for a battle, which will put them in a queue of people looking to play. Once enough people with similar latency, skill level, game type preferences, and other hidden factors are in the queue, a battle will start and all involved players will start to compete for the victory. Battles are generally short lived, usually lasting less than five minutes from start to finish. Once the battle is over, results are acknowledged by the system and players are brought back to the initial state, where they can decide to queue for a battle again.

Measuring a game server's performance is crucial to understanding its correct functioning. Furthermore, with detailed enough metrics, it might be possible to gain insight into how the players are actually experiencing the game, even from a server point of view. Unfortunately, obtaining meaningful metrics from a game server is not an easy task, especially in a complex system like the one most games use to provide a

high-quality experience to millions of players.

The following metrics mostly apply to a whole cluster of game servers rather than a singular instance. In particular, the following properties of the system are given for granted.

- The whole system comprises multiple instances to compose a single cluster.
- Each cluster node runs a single game server and nothing more.
- Each game server runs multiple battles simultaneously, each played by a certain number of players, unique for each battle.
- Battles don't share any data with each other except the common logic code.

3.4.2 Tick-related metrics

One class of helpful metrics obtainable from a game server is those related to tick processing time. By understanding how much time the server spends processing and waiting for each tick, the quality of the battle can be determined. Depending on the number of battles and players, the timing taken to process ticks might drastically change, especially if the server is overloaded and cannot process ticks in time. Figure 2 visually shows the metrics that will be explained more in-depth in the following paragraphs.

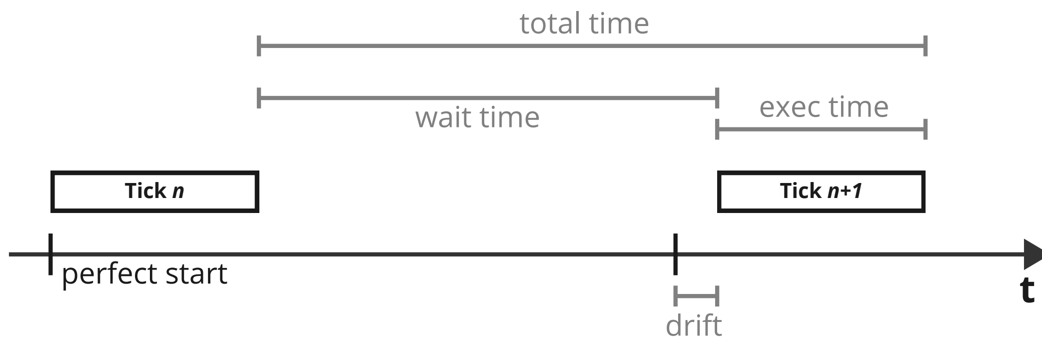


Figure 2: Visual representation of time spans of a tick.

Tick execution time

Tick execution time measures the exact amount of time it took the game server to process and execute the game logic to advance the game simulation by one tick, from start to finish [43]. Tick execution doesn't include the time taken to acquire the inputs from the clients and send the outputs to the players, mostly because game servers treat input and output processing as concurrent actions, which are not included in the main game logic loop. Furthermore, including input and output measurements could skew the results, which would be influenced by external factors of the system,

such as network performance. Tick execution time can help understand the quality of the gameplay. For example, if the execution time increases, it could mean the server is processing more battles simultaneously, or the game logic has become more computationally complex.

Tick wait time

Tick wait time measures the amount of time elapsed between the end of execution of the previous tick and the start of execution of the current tick [1]. Under normal conditions, a game server should only spend a fraction of its computation time processing ticks. The rest of the time is either spent idling or executing other tasks extraneous to the game server, such as operating system functions. All the time not spent processing ticks is counted towards the tick wait time, which consequently increases if the tick execution time decreases, and vice versa. If the wait time lowers under a certain threshold, the system might start to become overloaded since ticks are taking too much time to be executed, causing possible delays to other tasks and making deadline misses harder to recover from since the server and the task scheduler have less time to react and fix the scheduling.

Tick total time

Tick total time measures the amount of time elapsed between the end of execution of the previous tick and the end of execution of the current tick, which corresponds to the total time taken to process a tick in full. Alternatively, it could also be measured as the amount of time elapsed between the start of execution of the previous tick and the start of execution of the current tick. Because ticks are either getting executed or waiting for execution, the sum of execution time and wait time equals the total time, therefore:

$$total = execution + wait$$

The total execution time measures the actual perceived tick time since it considers both the execution and waiting phases, which are not visible to an external observer, such as a game client. For this reason, total tick time is a metric that closely represents the player experience.

Tick drift

Tick drift measures the amount of time elapsed between the *perfect scheduling time* of a tick and the actual scheduling time of the same tick. In a real-time game server, ticks must be scheduled in advance. Because perfectly respecting the planned tick schedule is nearly impossible in a soft real-time system, it is likely that the execution phase of a tick might start later than planned. Depending on the scheduler, it is also possible that some ticks might get executed before the scheduled execution time, which means that the tick drift will be negative. Tick drift can be a helpful metric when game servers are complex real-time systems that schedule and execute multiple tasks simultaneously. For example, if a non-game-related task overloads the server, tick processing might be

delayed, although the actual processing time might still be in the acceptable range. Although total tick time might be under control, tick drift might show that clients were updated by the server later than expected, causing a worse player experience.

This metric is completely original and only applicable in some specific scenarios. Since all real-time tasks should have the same recurring frequency, all deadlines should be distributed evenly in time with a frequency equal to the tickrate. In order real-time systems, however, deadlines can be more irregular; therefore, defining a perfect scheduling time might simply not be possible.

3.4.3 Load-related metrics

Load metrics help understand how much load a certain game server handles at any given time. Load metrics for a server are essential indicators that help monitor, assess, and optimize the performance and scalability of the server, especially if combined with metrics that measure the quality of the player experience, such as tick metrics.

All load-related metrics presented in this section are original and possibly exclusive to the current use case since different systems can handle battles and players in completely different ways.

Number of battles and players

Measures the instant number of ongoing battles and the total number of players playing in those battles. In a system where each game server runs multiple battles simultaneously, tracking the number of battles and players can be vital to understanding the server's load level. For example, it might be possible to observe some correlations between the number of battles and the tick drift since more tasks to execute might cause more frequent scheduling delays. If the number of players per battle is variable, then it might also matter in terms of performance since battles with more players most likely perform worse, which might be reflected in an increase in tick execution time. Tracking the number of battles and players can also help to understand the breaking point of the system, which corresponds to the point at which the real-time systems start showing signs of overload, such as unacceptably high total tick times.

System CPU load

Measures the load on the system processor as a percentage number. Measuring CPU load as a single percentage number in a multi-core system can be tricky, especially if different cores can process vastly different amounts of instructions. In general, the assumption is that game servers and real-time systems will try to maximize efficiency by distributing load equally across cores. Therefore, to calculate the load percentage number, it might be possible to compute the average percentage of time spent in a not-idle state, given c represents the number of cores of the system and $idle_n$ represents the percentage of time spent idling for core n:

$$load = 1 - \frac{\sum_{n=1}^{c} idle_n}{c}$$

By measuring the system CPU load, it might also be possible to determine correlations between the number of battles and players, the current system load, and the number of ticks that took unacceptably long to process. For example, an overloaded system might show a higher average CPU usage, which means that the scheduler will need to schedule tasks more carefully to avoid delays. It is also important to notice that game servers are unlikely to be bare-metal systems, which means they usually run on an underlying operating system (OS). The game server and the operating system will share the same processor at any given time; therefore, the effect of OS-level activities might be reflected in the measurement despite being completely out of the game server's control.

3.4.4 Ideal metrics in a perfect scenario

In a perfect scenario, tick and load metrics will look as follows.

- **Tick execution time.** As low as possible. A low execution time means the game server is executing ticks efficiently and with ample margins of expansion in case of heavier loads or operations.
- **Tick wait time.** As close as possible to the defined tick time, especially if the tick execution time is close to zero.
- **Tick total time.** As close as possible to the defined tick time. Ideally, the total tick time should be exactly the expected tick time, which means ticks will be processed and players updated at a constant and predictable rate.
- **Tick drift.** Exactly zero. A low tick drift means the scheduler is working correctly, and tick execution can start when expected, which helps avoid mis-schedules and delays.
- **Number of players and battles.** As high as possible, without influencing the previous metrics. By hosting many battles on a single server, infrastructure costs can be greatly reduced, but only if the quality of the battles stays high.
- **System CPU load.** As low as possible. Although not only dependent on the game server, a low CPU load level can help the scheduler by providing more idle time that can be used to simplify the schedule, especially with a high number of concurrent battles.

4 Real-time Java

4.1 Introduction to the Java platform

The Java language and platform, developed by Sun Microsystems and now maintained by Oracle Corporation, are a robust, versatile, and widely used computing environment designed to support the development and execution of software applications [33]. Unlike many of the programming languages previously released, Java aims to offer portability, speed, and safety, making it a viable choice for many applications, including real-time systems.

Portability

Java tries to achieve platform independence using the *Java Virtual Machine* (JVM). Unlike traditional languages, which are compiled into binary instructions that can be executed by the processor directly, Java compiles its programs into *Java bytecode*, which is a custom binary format that includes simple custom instructions [33]. The main difference between binary instructions and bytecode is that the processor cannot execute Java bytecode directly. Instead, it must be interpreted by the JVM, which acts like an intermediate runtime system. The runtime system operates in a secure virtual environment while carrying out all the typical tasks of a physical processor [33]. Like an operating system, it controls memory and performs a stack-based instruction set. Most crucially, it accomplishes all of this while adhering to a well-specified open standard that anyone wishing to create a virtual machine that complies with Java can implement. Thanks to this high-level abstraction, Java can achieve portability across a wide range of platforms since a compatible JVM version is all that is needed to operate correctly in a certain environment. On the other hand, traditional languages cannot achieve such portability because each platform can require different instructions and, in some cases, even completely different architectural patterns. For these reasons, Java has become a popular choice for developers who want to deploy their products across different platforms without having to guarantee compatibility with each one separately.

Speed

Contrary to popular belief, Java is designed to be fast [34]. Although introducing the Java Virtual Machine as an intermediate layer between the program and the machine has a significant performance impact, the Java platform utilizes advanced techniques to ensure a high-speed application execution. The main strategy used by Java to guarantee superior performance is *Just-In-Time* compilation (JIT) [33]. Under normal circumstances, the JVM will interpret bytecode instructions and execute them in a sandbox environment. Although interpretation is safe and works in every condition, the speed of the interpreted code is significantly slower than any other traditionally compiled language. However, the JVM has the capabilities to measure and detect those parts of the application that would benefit from compilation and proceeds to compile them during runtime using the JIT compiler. Over time, large parts of the code base

will be executed directly instead of interpreted, thus granting high performance while maintaining portability. This feature makes Java viable for most high-performance applications, including real-time systems [35].

Safety

Java provides a *safe-by-design* platform to run commercial applications that require stability and security [33]. Unlike traditional languages, Java prevents memory corruption completely by providing automatic memory management mechanisms. In traditional languages like C++, memory management is completely manual and sole responsibility of the programmer. Although theoretically more powerful, the system requires the final user to fully understand the system and plan memory allocations and deallocations carefully. If done incorrectly, a phenomenon known as *memory corruption* can happen, where memory is allocated without ever being released, or the program can try to access forbidden areas of the memory layout. In either case, memory corruption usually leads to security vulnerabilities and crashes [42]. These issues can be even more impactful in long-lasting applications, such as servers, since repeating the same usage patterns for thousands of hours will eventually exacerbate problematic behaviors. Java completely fixes the problem of memory corruption by introducing automatic memory management, where the ability of the programmer to manage memory manually is completely removed. Instead, the system will automatically allocate memory when needed and then provide a tool to deallocate it when necessary, called the *Garbage Collector* (GC) [11]. Although this system introduces runtime costs, it also prevents many problems that affect most traditional languages entirely. Apart from automatic memory management, the Java platform also provides numerous other security mechanisms, such as sandboxing, which effectively isolates the JVM from the rest of the system [33]. By doing so, sandboxing can help protect systems from malicious parties and ensure the safe execution of Java applications.

4.2 Features and challenges of real-time Java

Given the Java platform's portability, performance, and safety aspects, it is natural that Java can be considered a language well-suited for developing real-time systems. Unfortunately, in reality, Java presents some challenges when dealing with real-time systems. Unlike most languages designed for real-time programming, Java was created primarily to simplify programming rather than help programmers create software that consistently complies with real-time restrictions [35]. For example, the runtime costs added by the JIT compiler and the GC can make estimating deadlines and scheduling tasks harder than on traditional platforms, especially on hard real-time systems. Therefore, the general consensus has been that Java, at least in its current state, is not completely appropriate for developing real-time software. On the other hand, most commercial real-time systems, especially those not employed in safety-critical scenarios, do not require overly stringent deadlines and near-perfect schedulers. Therefore, Java could still be a viable option in those cases. Furthermore, many soft real-time systems consider predictability only a secondary requirement compared to

stability and safety, which is where Java truly excels. For these reasons, despite many complications that might arise from its simple nature, Java is still considered by many a great choice for real-time programming [6].

Java's strengths in a real-time environment

Here are presented some of Java's biggest strengths when utilized in a real-time system [33, 35].

- **Reliability.** Java is extremely reliable in most circumstances, including in real-time systems. Thanks to the automatic memory management system, sandboxing, and various abstractions that separate the JVM from the underlying operating system, the Java platform is extremely stable and can operate without any issues for years to come. Robustness is an essential advantage in real-time systems, especially those that require faultless operations for thousands of hours without interruptions. On the other hand, traditional systems that utilize manual memory management can struggle significantly to operate for years without issues, especially in complex systems. In particular, memory corruption is so problematic that some safety-critical systems might forbid memory allocation completely [22]. For these reasons, Java particularly appeals to real-time system developers looking for a platform that provides safety without sacrificing features.

- **Portability.** Java's portability makes developing and distributing software for multiple platforms easier. Because some real-time systems are required to run on embedded systems, which might feature uncommon hardware limitations, Java significantly simplifies development on those platforms, given that a compatible JVM version exists. Even in those real-time systems where portability is not an issue, Java's simplified deployment process can significantly improve the efficiency of the development process, making the Java platform attractive to many developers.

- **Efficiency.** Despite being a hybrid interpreted language, Java is significantly faster than the competition, mostly thanks to JIT compilation. In fact, after a necessary warm-up period, Java can become as fast as traditionally compiled languages like C++ while retaining most of the advantages of interpreted languages. Because speed is often necessary in many real-time systems that run thousands of tasks per second, Java can become a viable alternative to traditional languages, especially given the fact that speed is achieved while providing a reliable and portable environment.

Java's weaknesses in a real-time environment

Despite the many advantages, Java also presents several weaknesses that can complicate the development of real-time systems [33, 35].

- **JVM overhead.** Despite being very efficient, the Java Virtual Machine running along every Java application presents a meaningful overhead. Because the JVM is constantly performing background operations to ensure the correct functioning of the application, such as JIT compilation or garbage collection, the actual efficiency of operations is lower than expected. Furthermore, most of the operations performed by the platform are opaque to the external viewer, meaning that it is nearly impossible to understand what is happening inside the JVM at any time. This mechanism is often called *black box*, and its purpose is to provide an abstraction layer to separate the user from the platform. For example, by disallowing the program from viewing the amount of memory in use, Java tries to force developers to adopt programming patterns that embrace multi-platform paradigms. Although abstraction is a powerful tool that can ease development and increase productivity, it comes at the expense of the obscurity of the underlying platform. For this reason, the sunken overhead of the Java platform can pose a real challenge to fine-grained environments like real-time systems since it explicitly disallows developers from understanding part of the platform they are working with.

- **Poor predictability.** Predictability is an absolute necessity for *hard* and *strict* real-time systems. Unfortunately, as previously mentioned, the Java platform introduces opaque overheads that are nearly impossible to predict. Because the JVM utilizes background processes to execute tasks that are necessary for the inner workings of the platform, and those tasks are a completely separate system that is not accessible to the application itself, predictability is seriously hindered. For example, tasks that access a certain part of the heap might be delayed significantly when the garbage collector needs to compress that memory region, and the whole process can be unpredictable from the tasks' perspective. For this reason, strong predictability is too difficult to achieve consistently on the Java platform, making Java an unpopular choice for hard and strict real-time systems.

- **Inconsistent execution speed.** The JVM utilizes JIT compilation to progressively optimize some parts of the application during runtime. Although this approach significantly improves efficiency over time, it also introduces inconsistencies in execution speed. Unfortunately, the JIT compiler is another opaque and hard-to-predict mechanism of the JVM, which means that newly compiled routines will experience an abrupt speed change. Even if efficiency usually improves when the JIT compilation process is performed, recently compiled methods will also be harder to predict, especially if self-tuning schedulers are used to predict execution times of tasks based on previous runs. To make matters worse, in some rare cases, the JIT compiler can make mistakes, making predictability even worse. Therefore, the JIT compiler can cause inconsistencies that will worsen predictability, a key aspect in real-time systems.

4.3 Inner workings of the Java Virtual Machine

The Java Virtual Machine is a powerful but complex system. To provide portability, speed, and safety, many mechanisms that traditional languages would consider the developer's responsibility are executed internally by the Java platform. In particular, Java tries to guarantee those features by abstracting many layers of responsibility that would normally allow the user to interface with the underlying system. The two most prominent examples of this design choice are automatic memory management through the Garbage Collector and Just-In-Time compilation. All the information contained in the following sections is based on the book "Java Performance" [36] and on the official documentation [52].

4.3.1 Just-In-Time compilation

Computers and their CPUs can execute only a few specific instructions called machine code. Traditional compiled languages, such as C++, generate their programs thanks to a tool called *compiler*, which processes the written program and converts it to machine code automatically. Every part of the compilation is performed before the program is ever executed, often referred to as *build time*. Compiled languages are usually extremely fast since all necessary operations are executed before the program even starts. However, they usually lack flexibility and portability since once the program is compiled, no further operations shall ever modify it. More modern languages, like Python, prefer to utilize an interpreter by contrast. Instead of compiling the application, the language interpreter reads through the user-written code and executes it directly. Unlike compilation, this approach is extremely flexible since everything is executed at runtime, including operations that a compiler could not possibly perform at build time. Furthermore, interpretation also provides better portability since the same version of the application could be interpreted differently depending on the platform on which it is being executed, something that would be nearly impossible for compiled languages. Unfortunately, these kinds of languages require an additional layer that separates the application from the platform they are running on, which is the language interpreter itself. For this reason, traditionally interpreted languages are slower than their compiled counterpart, which might be a deal-breaker in many scenarios. Java, however, uses a hybrid approach that tries to obtain the best from both worlds.

Java applications are both compiled and interpreted. First, user-written code is converted by a build-time compiler into an intermediate low-level representation called *Java bytecode*. Unlike machine code, bytecode cannot be executed directly by the CPU, but it requires an interpreter to execute it. However, interpreting bytecode is much more efficient than interpreting anything human-written, therefore Java code will run faster than the competition. Furthermore, unlike machine code, which is often platform-specific, bytecode is universal, making it executable on every platform supported by the JVM. Unfortunately, machine code is still more performant even if bytecode is more efficient than human-written code. To compete with traditionally compiled languages, the JVM provides an additional optimization step called the *Just-In-Time* (JIT) compiler. Unlike traditional compilers, the JIT compiler will

perform necessary compilations at runtime, slowly compiling routines in order to offer better performances over time. Although Java applications will perform worse than their build-time compiled counterparts, over time, JIT compilation will improve efficiency and provide comparable performance minus the Java platform overhead.

4.3.1.1 Types of JIT compilers

To deliver the best performance with a relatively small footprint, Java features two distinct Just-In-Time compilers, unlike many other JIT-compiled languages that might use only one. By using two different compilers, the JVM tries to balance the compilation overhead with its performance benefits. For example, rarely used parts of the application might only require minimal compilation to achieve an acceptable level of performance. Meanwhile, some frequently used parts might require more in-depth tuning for better efficiency, even at the expense of a slow and complex compilation process. Because JIT compilation is a black-box process invisible to the final user, different implementations of the JVM can vastly differ. By far, the most common Java platform implementation is *HotSpot* by Sun Microsystems [52], which features two JIT compilers named *C1* and *C2*.

- **C1 - Profiling compiler.** The C1 compiler is the first and simpler JIT compiler. Unlike traditional compilers, it only performs basic compilation operations and produces partially optimized machine code. Because of this, C1 is considered a fairly fast compiler, which makes it convenient in some scenarios where achieving a small compilation time is more important than a highly optimized method. The C1 compiler is also known as the *profiling compiler* since it can perform some runtime performance measurements on freshly compiled code. Because profiling methods at runtime have a significant performance impact, the compiler can toggle between three different measurement modes: fully enabled, basic profiling only, and fully disabled. Profiling data can be used by other compilers in later stages to provide even better optimizations.

- **C2 - Speculative compiler.** The C2 compiler is the second and most complex compiler. It can perform advanced and in-depth optimizations to the compiled code, although it is computationally expensive compared to C1. In some cases, the C2 compiler can completely modify its behavior depending on the targeted platform to fully utilize the provided instruction set. In order to provide highly efficient code, C2 uses the profiling data obtained by C1 to optimize more than normally possible. Under normal circumstances, only some parts of the application are frequently executed, such as key routines for the correct working of the service, while others are not, such as error-correcting methods. Thanks to the data obtained by the C1 compiler with profiling enabled, C2 is able to identify those parts and further optimize them by speculating based on previous executions, hence the name *speculative compiler*. By removing and reorganizing parts of the codebase to match the profiling data, C2 can often squeeze more performance than C1 at the cost of added complexity and increased compiling times.

It is important to note that C1 and C2 are not separate systems; instead, they both require each other. In particular, C2 requires the profiling data obtained by C1 to provide a higher-level optimization strategy. On the other hand, by activating profiling, C1 is actively sacrificing performance to provide C2 data that might be useful in creating a more optimized version of the code in the future. Therefore, both JIT compilers have been engineered as different parts of the same system that should always operate together.

4.3.1.2 Tiered compilation

Because Java features two JIT compilers, it is crucial to understand when the JVM switches between one or the other. In its bytecode representation, Java only distinguishes between two structural elements of the codebase, which are *classes* and *methods*, which always belong to a class. During JIT compilation, only methods are ever taken into account and are always treated as individual entities. The main consequence of this classification is that, even if belonging to the same class, different methods can undergo different compilation and optimization strategies depending on their characteristics and importance in the application. Instead, classes are almost invisible to the JIT compilers and are rarely taken into consideration.

In older Java versions, the final user had to manually specify which strategy to use at startup, by specifying a command line flag that could either be *-client* or *-server*. When choosing the client strategy, Java would assume immediate mediocre performance was preferable to a slower warm-up time followed by eventually high performance, mainly because clients are short-lived applications that, therefore, might not be executed long enough to see any long-term benefits. For this reason, the client strategy would only utilize the C1 compiler, excluding C2 completely from the process. On the contrary, when choosing the server strategy, Java would assume that long-term benefits were preferable to short-term ones, especially given the usually long life of servers compared to clients. Therefore, in those scenarios, C2 would be the preferred JIT compiler, although C1 was initially used to obtain the profiling data necessary for C2 to generate highly-optimized code.

Starting from the release of Java 8 in 2014, *tiered compilation* was introduced, which replaces the previous system with a general-purpose one that also provides self-tuning capabilities. The new system offered the final user plenty of choices for fine-tuning but did not provide any choice for the strategy used. Instead, the system would automatically understand which parts of the application required which compilation strategy during runtime, theoretically combining the best of both strategies. In general, tiered compilation works by measuring the number of invocations of each method and then reacting accordingly. Methods that are invoked the most are considered *hot* (hence the name *HotSpot*), which means high optimization for those methods should always be a top priority to provide better overall performance. For this reason, hotter methods will be compiled into a more optimized tier sooner than others to provide immediate performance benefits. However, eventually, all methods will be invoked enough times to be considered worth compiling with C2. Unlike the strategy approach, tiered compilation aims for both short-term and long-term performance benefits by

Level	Compiler	Efficiency	Activation reasons
0	Interpreter	- - -	
1	C1 without profiling	++	Level 3, C2 provides no benefits
2	C1 with basic profiling	+	Level 3, C2 is temporarily busy
3	C1 with full profiling	-	200 invocations
4	C2	+++	Level 3, 5000 invocations

Table 1: Tiered compilation levels

trying to understand which methods need which strategy the most, which is far more effective than treating the entire application as a single entity.

Tiered compilation features a total of five levels, ranging from 0 to 4. In general, methods will start from the lowest level and then proceed towards the highest one; however, in some cases, levels can be skipped. In some even rarer cases, methods can also proceed backward to a lower level. The most common path taken by methods is starting at level 0, proceeding temporarily at level 3, and finally ending at level 4. Despite a common misconception, methods compiled to a higher level are not necessarily more efficient. All levels are presented in Table 1.

- **Level 0 - Interpreter.** Level 0 is the basic level where all methods start. Methods at level 0 are not compiled; instead, they are interpreted directly by the JVM. Although interpretation is always the slowest execution method possible, starting at level 0 requires no compilation time whatsoever, so unused or rarely used methods (usually thousands in larger codebases) are allocated no compilation time whatsoever.

- **Level 3 - C1 with full profiling.** Level 3 is the first compiled level that each method will eventually reach. Under normal circumstances, at least 200 invocations are required for a method to reach this level. However, a dynamic scaling factor could be used to balance the number of invocations at runtime accurately. Although C1 can provide some performance benefits, methods are still quite inefficient when at level 3 since full profiling can be an expensive process. Unfortunately, despite the inefficiency, level 3 is strictly necessary to proceed to level 4, which is the final goal of tiered compilation.

- **Level 4 - C2.** Level 4 is the final compilation level and by far the most efficient since the C2 JIT compiler heavily optimizes methods. Unlike C1, the C2 compiler requires profiling data to work correctly, therefore, each method must reach level 3 before eventually going to level 4. Under normal circumstances, 5000 invocations are required to trigger the C2 compilation step, although the number may vary because of the dynamic scaling factor. Although C1 and C2 work similarly to each other, C2 is far slower than C1, which means that some particularly hot methods that have reached the necessary threshold to be compiled to level 4 might need to wait in a queue for their turn if C2 is particularly busy.

- **Level 2 - C1 with basic profiling.** Level 2 is an uncommon level that the JVM rarely uses. It becomes useful when a certain method has already reached level 3 and is ready to be compiled to level 4; however, C2 is temporarily too busy with other compilations to process it immediately. If that is the case, the JVM might prefer to port the method to level 2 instead of keeping it to level 3 because level 2 features only basic profiling compared to level 3. By lowering the effectiveness of profiling, the JVM tries to improve efficiency when enough profiling data is already collected, but the compilation that would use that data is postponed. After C2's workloads finally decrease, methods stuck at level 2 can finally resume their process and be compiled at level 4.
- **Level 1 - C1 with no profiling.** Level 1 is another uncommon level that the JVM rarely uses. It becomes only viable when, after profiling at level 3 has ended, the JVM realizes that compiling the method with C2 would hardly bring any performance benefit. This is usually the case for methods lacking branching, mostly because C2 optimizes based on speculations obtained by observing common paths during branches. Because C2 is significantly slower than C1, unless C2 is strictly needed to provide a more efficient implementation, it is otherwise avoided. For this reason, some methods can remain compiled with C1, although all profiling is disabled since no further information is needed in the process. Unlike other levels, once a method reaches level 1, it will stay there forever.

Tiered compilation can be customized in depth by many parameters that can configure its behavior at startup. For example, all thresholds can modified at will. For this reason, all values used to illustrate the tiered compilation process are only valid if the default parameters are used.

4.3.1.3 Speculation and Deoptimization

The C2 JIT compiler is often referred to as *speculative compiler* because it speculates some properties of the methods it compiles based on data obtained by previous executions. Java code can be described as a series of nested branches forming a tree. A branch is a conditional code path where a certain action only occurs if the associated condition is true. In many cases, certain branches are executed more often than others. Depending on the context, in some other cases, some branches might never be executed, which renders them virtually useless. The C2 compiler can perform speculative optimizations by guessing which branches might never get executed and removing them. Not compiling branches has a double effect on efficiency. First, it saves compilation time since all application parts that are considered useless are simply ignored. Second, the generated machine code can be compiled in a way that favors the fast execution of the common path while disadvantaging the branches that are speculated away. To speculate as accurately as possible, C2 requires data to understand the effective usage of all branches; for this reason, the C1 compiler includes a profiler.

Unfortunately, speculations cannot guarantee absolute correctness; therefore, C2 can make mistakes. Whenever a certain speculation happens, and a branch is

removed, Java cannot simply ignore the existing code; otherwise, recovering from a mistake could be nearly impossible. Instead, it will replace it with a *trap*, a special instruction that freezes the execution of the current method and gives control back to the JVM. By default, if a certain branch is not executed at least 5 times during the profiling phase, the C2 compiler speculates it will never be executed again; therefore, compilation can be avoided, and further optimization can be achieved. In many real-life scenarios, this can be true. Java is a language that emphasizes error handling, which means those who use it to develop business applications will likely develop many routines to identify and handle errors. In most scenarios, errors are very uncommon, therefore the error-handling code stays unused. By identifying these instances, the C2 compiler can further optimize the applications. In most cases, users will appreciate the speed of the compiled version without ever noticing the speculations that live underneath. Unfortunately, because speculations are never guaranteed, the branch that was previously considered safe to remove can become relevant again by being executed. When this happens, the CPU will reach an uncompiled part of the application and will encounter a trap, which will transfer control back to the JVM. The JVM must resume execution without using the level 4 compiled code because it contains a wrong speculation. Because the C2 compiler is particularly slow and execution must resume immediately, compilation cannot be performed directly. Therefore, the JVM will fall back to using the previously compiled method with the highest level (apart from the invalid one at level 4). Unlike level 4, level 3 methods are less efficient, so execution will suddenly and abruptly slow down. Furthermore, because C2 can be busy with other compilations, it is not guaranteed that a new and correct version of the method will be available anytime soon. This process is known as *deoptimization*.

In many cases, deoptimization is an intended feature with only minor consequences. For example, if an error occurs and triggers some error-handling routines that were previously optimized away, the impact on performance could be temporarily big, but because errors are usually rare, the effects will not be everlasting. Therefore, the JVM assumes that deoptimizations can happen, but they are usually rare and insignificant for a long-running application. However, this is not always true, and in some rarer cases, deoptimizations can have serious consequences.

4.3.2 Memory management and garbage collection

Unlike many traditional languages, Java uses an automatic memory management system. Java's memory management model, which includes automatic memory management and garbage collection, is a key feature that simplifies development and enhances application stability and security. This model abstracts away the complexities of manual memory management, which are prevalent in traditional languages like C++, and replaces them with an automated system that guarantees almost complete safety at the expense of some computational overhead. Many developers consider automatic memory management to be an exceptionally useful feature that is well worth the overhead it introduces, hence Java's popularity.

Automatic memory management in Java means that the Java Virtual Machine (JVM) handles the allocation (reserving memory) and deallocation (freeing memory

previously reserved) of objects, relieving developers from the burden of manual memory management. In traditional languages like C++, memory management is completely manual. Allocations and deallocations are performed manually on a per-object basis, and the developer has complete control over the entire system. Manual memory management is extremely powerful but also has various weaknesses that are driving developers away from it.

Strengths of manual memory management

Although not perfect, manual memory management is powerful and flexible, making it the primary choice for many applications that require performance and complex data structures.

- **Fine-grained memory control.** Manual memory management allows developers to control memory freely, which means they can have precise control over memory allocation and deallocation. This allows them to build complex data structures that allow efficient and effective operations without any overhead. Manual memory control also allows the usage of pointer arithmetic, which allows memory addresses to be addressed directly, and mathematical operations can be performed on them, which can help further optimize data structures.

- **Memory usage optimization.** By allocating memory manually, the user can fully optimize allocated memory and possibly save significant amounts of it. For example, developers can implement custom memory allocators and strategies tailored to the specific needs of their applications, which can lead to more efficient memory usage. In other cases, allocated memory can be recycled for other purposes without needing re-allocation, which can increase efficiency significantly.

- **Deterministic resource management.** With manual memory management, memory and other resources can be released immediately when they are no longer needed. This vastly differs from automatic systems, which usually employ a garbage collector that deallocates memory in a hard-to-predict manner. In general, manual memory management guarantees no overheads and high efficiency while also avoiding hidden computation expenses.

Weaknesses of manual memory management

Despite the high degree of freedom it offers, manual memory management presents various problems that are hard to ignore.

- **Error-prone.** Manual memory management is difficult, even for more experienced programmers. Unfortunately, ensuring correctness is impossible because the problem is NP-hard. To prevent errors, many innovative techniques exist that can still grant the user control over memory management, but none of them can grant freedom and safety at the same time.

The C++ standard library proposes the concept of *smart pointers* [26]. Smart pointers work just like regular pointers by containing a reference to an object, but they usually wrap the reference with additional functionalities that can help avoid common programming mistakes. For example, a smart pointer can determine when the object it points to becomes unused and automatically deallocates its memory without needing any particular programmer's attention. Unfortunately, the *smartness* of smart pointers introduces a computational overhead necessary to allow seamless memory deallocation; therefore, safety comes at the expense of performance. Furthermore, smart pointers do not guarantee correctness because developers can still misuse them; their only goal is to provide a less error-prone method of manually managing memory.

The Rust programming language also introduces a system to provide manual memory safety, which utilizes an innovative type system to ensure complete memory safety [31]. Rust introduces the concept of *ownership* into its type system, which forces all references to have a unique owner and later *borrow* to other parties. Similarly to how smart pointers work, the compiler can infer from ownership when to allocate and deallocate memory, with a strict guarantee of safeness. For this reason, Rust can achieve complete memory safety without any runtime overhead. Unfortunately, ownership limits the usage of references. Therefore, some data structures that are commonly used in languages like C++, such as linked lists, cannot be implemented anymore. To prevent this problem, Rust can still occasionally allow developers to use *unsafe* memory management, where all previous rules and guarantees are ignored. Once again, it is not possible to guarantee safety, freedom, and performance at the same time.

- **Security risks.** Manual memory management, if done incorrectly, can introduce several security vulnerabilities to the application. Because manual memory management is so error-prone, security vulnerabilities are common in applications developed in those languages. One of the biggest security problems related to manual memory management is memory corruption. Memory corruption can occur when the data saved in the memory of an application or any of its references becomes corrupted with unintended content, mostly due to programming mistakes. Thanks to memory corruption, an attacker could potentially exploit the vulnerable application with techniques such as buffer overflows (accessing areas of memory that are not supposed to be accessed), use after free (accessing a part of memory that we previously deallocated), and many others [42]. Because manual memory management places all responsibility for memory handling on the programmer, nobody can prevent these vulnerabilities apart from the programmer themself. Unfortunately, in 2021, memory corruption accounted for 67% of all 0-day vulnerabilities analyzed [49], highlighting the relevance of memory corruption even today.

- **Increased development effort.** To avoid memory-related issues, developers must manually track memory allocations and deallocations, which increases the complexity of the code and the potential for errors. Furthermore, managing

memory manually adds to the maintenance burden, as developers must ensure that all allocated memory is properly freed, especially in large and complex applications. Because correctness is never guaranteed in traditional languages, a big development effort is used to ensure memory safety since no tool could ever ensure complete safety. For these reasons, ensuring memory safety can be an expensive and inefficient process, which is absolutely necessary in software development with manual memory management.

4.3.2.1 Garbage collection

Java achieves a fully automatic memory management system thanks to garbage collection. The GC allows developers to effortlessly handle object life cycles. Objects are allocated only when required, and the JVM automatically releases them when they are no longer in use. Garbage collection is an extremely complex process, and no perfect solution exists. From a basic point of view, the GC performs three distinct operations: *mark*, *sweep*, and *compact* [36].

- **Marking.** During this phase, the GC must recognize all objects that are no longer in use while keeping all live objects untouched. This operation is usually inefficient and complicated due to the nature of Java references. One of the traditional approaches used is *reference counting*, which involves counting the number of references pointing to every object; once an object is no longer referenced, it should be safe to deallocate it. Unfortunately, reference counting is not effective when dealing with circular references, although it is very efficient. For example, if two objects *A* and *B* exist and are instantiated so that object A is referencing B and object B is referencing A, reference counting will never be able to recognize those objects as unused because there will always be at least one reference to them. More effective algorithms start scanning for references from the root objects (those objects that will be permanently referenced as part of the JVM) and progress through the entire memory, identifying all objects that are being referenced. Once the whole memory tree is traversed, all the objects that were not scanned are considered no longer live and thus marked for removal. Unfortunately, this approach is much more inefficient than the previous one, although allows clearing the most memory. Newer GC algorithms can adopt even more advanced strategies to further optimize performance, such as executing the marking execution concurrently.

- **Sweeping.** After marking is done, objects will need to be removed and their respective memory deallocated. The sweeping operation is fairly straightforward since it involves little logic, which is instead being used in the marking phase. More advanced algorithms also perform sweeping concurrently to increase the efficiency of the application and minimize lag.

- **Compacting.** Freeing memory from unused objects is not enough. When memory is deallocated, existing live objects will stay in place, while all other objects will be removed. In some cases, the newly free memory region will

be located within other parts of memory still in use. Because objects need to occupy contiguous memory locations to operate correctly, some newly freed memory regions can be unsuitable for objects too large for that region. This issue is known as *memory fragmentation* and can lead to high memory consumption despite the existence of many usable regions. To avoid this problem, garbage collectors must compact all memory by re-distributing live objects in contiguous memory regions. Compacting is usually the heaviest operation performed by a GC because it requires pausing the execution of all parts of the application that are using the live objects being moved. For this reason, some GCs only perform compacting operations when strictly necessary, while others only perform compacting in smaller memory regions at a time. In either case, compacting is a necessary but inefficient memory operation to minimize wasted memory.

Generational Garbage Collection

Many modern garbage collectors are defined as *generational* and are based on the observation that most objects are short-lived. In Java, it is common practice to use immutable objects. They are a particular type of instances that, after being initialized, can no longer be modified. In many cases, these objects are discarded shortly after being allocated and are also fairly simple to track down. Therefore, it could be beneficial to treat short-lived objects differently than others, hence

In generation garbage collection, memory is divided into generations, usually called *old generation* and *new generation*. The young generation is also further divided into sections called *Eden*. When a new object is created, it is assumed by default as short-lived, and therefore, it is placed in the Eden section of memory. Because the young generation is supposed to be bigger than the old one, the GC will perform two different collection cycles: a more frequent one for the young generation (also called *young GC*) and a more complex one for the old generation (also called old GC). When a young GC cycle happens, all objects in Eden are either removed if unused or moved in the rest of the young generation (also called the *survivor* space). Young objects that have also survived several cycles become part of the old generation. When the young GC is done, all Eden zones are free, and memory in the young generation is compacted. When the old generation is also nearly full, the old GC will intervene and perform similarly only to the oldest objects.

Generational garbage collection has the big advantage of offering quick pauses to clean most objects while also performing a complete cleaning more rarely. It also compacts objects frequently, leaving little memory wasted due to fragmentation. On the other hand, most commercially available generational GCs are not continuous, therefore, they are required to stop the application entirely from time to time. This approach could be potentially problematic, especially for applications that require strict deadlines, such as real-time systems.

GC	Status	Target
Serial GC	Supported	Small footprint
Parallel GC	Supported	High throughput
CMS	Deprecated	Low latency
G1GC	Supported	Balanced, big heaps
ZGC	Experimental	Low latency, big heaps
Shenandoah	Experimental	Low latency, consistent
Epsilon GC	Experimental	No collection

Table 2: Available GCs in Java 21

4.3.2.2 Types of garbage collectors

Modern JVMs feature multiple strategies and their respective implementations, each providing various benefits to different areas. Over time, some GCs have been completely discontinued and are no longer available on modern JVMs, while new ones have emerged and thrived. As of Java 21, here is a list of all the supported garbage collectors [36].

Serial GC

The Serial Garbage Collector is Java's simplest and oldest garbage collector. It uses a single thread to perform all garbage collection work, making it suitable for single-threaded environments and small applications with low memory footprints. This collector is the default choice when the application is running on a client-class machine (such as 32-bit JVMs on Windows) or on a single-processor machine. Although the serial collector once appeared to be on the verge of becoming obsolete, the rise of containerization has brought back its importance. Virtual machines and Docker containers that use a single CPU core have made this algorithm relevant again. For all other scenarios, the serial GC is hardly ever considered a viable option.

Parallel GC

The Parallel Garbage Collector, also known as the *Throughput Collector*, works similarly to the Serial GC but uses multiple threads for garbage collection. It aims to maximize application throughput by reducing the total time spent on garbage collection. Just like its serial counterpart, it stops the entire application when performing both young and old collections, fully compacting the heap in the latter. Just like the name suggests, it becomes the most beneficial for applications where high throughput is more important than low pause times.

Concurrent Mark and Sweep (CMS)

The Concurrent Mark and Sweep Garbage Collector is designed for applications that require low pause times and can tolerate a slight reduction in throughput. It performs most of its work concurrently with the application threads, reducing the frequency

and duration of full application pauses but adding a small but noticeable performance overhead at all times. In general, CMS tries to perform all mark and sweep operations concurrently and only resorts to a full application stop when strictly needed. This process is known as *concurrent mode failure*. Unfortunately, CMS can perform very poorly in some situations where little memory can be reclaimed concurrently, to the point where out-of-memory JVM crashes can be a common occurrence if the GC is not properly configured, which can be a tedious process. Because of this tuning issue, as well as newer and more viable GCs becoming available, the usage of CMS is now officially discouraged, and its use has been deprecated.

Garbage-first GC (G1GC)

The G1 Garbage Collector is designed to provide predictable pause times and high throughput. It divides the heap into regions and performs garbage collection on selected regions based on the amount of garbage they contain. G1 is also known as a *mostly concurrent collector* because it performs some expensive work concurrently with the application, minimizing pauses. Its regional approach also allows parts of the application to keep running even when a collection becomes necessary in some other region. Unlike CMS, G1 is able to self-tune itself at runtime, granting optimal performance in the long run, although it might be penalized in short-lived applications. One of the main issues with G1 is that it performs best with large heap sizes to help with the regional approach, in particular more than 6 GB, of which live objects should occupy at least half. Because G1 is viable for a wider range of applications, it requires almost no manual tuning, and presents a good compromise between latency and throughput, it is set to replace CMS entirely and is enabled by default for server applications on compatible hardware.

ZGC

The Z Garbage Collector is designed for applications requiring low-latency garbage collection and can handle very large heaps (up to terabytes). ZGC performs most of its work concurrently with the application threads, ensuring minimal and predictable pause times. Under normal circumstances, ZGC will not stop the execution of application threads for more than 10 ms, up to 10 times less than what is achievable with G1GC, at the price of a moderate performance overhead. Unlike most modern GCs, ZGC is not generation, which means it can show poor performances in scenarios where young garbage is allocated at fast speeds, although a highly experimental generational operation mode is available for use. Unlike other GCs, ZGC does not support pointer compression, a feature of the Java machine that allows the utilization of 32-bit pointers on 64-bit machines with up to 32 GB of memory, further increasing CPU usage. In general, selecting ZGC over G1GC means preferring latency over throughput.

Shenandoah

Shenandoah is another low-pause-time garbage collector, developed by Red Hat and therefore not available in some commercial JVMs. Like ZGC, it aims to reduce pause times by performing most of its work concurrently with the application threads. Although their inner working is significantly different, both ZGC and Shenandoah perform almost identically. One of the interesting characteristics of Shenandoah is the fact that collection pauses should take roughly the same time, no matter the heap size.

Epsilon GC

The Epsilon Garbage Collector is a no-op garbage collector. It is designed for environments where memory management is handled externally or is not a concern, either because the program is extremely short-lived or because no memory allocation is ever performed. Epsilon GC does not perform any garbage collection tasks, but it simply allocates memory until the heap is exhausted, at which point the application will crash or halt. Enabling Epsilon GC will effectively disable the GC completely, yielding significant performance improvements but also providing no possible way of reusing memory. For this reason, using Epsilon GC is an extremely unpopular choice among developers, but it might be worth the performance benefits in very minor cases.

5 Methods

5.1 Iterative exploratory data analysis and problem solving

The empirical part of the thesis focuses on optimizing a real-time game server developed in Java. Because of the complexity of the Java platform in general and the Java Virtual Machine in particular, the main goal of the thesis is to analyze and possibly correct the application's behavior by better understanding Java's inner workings. By limiting the number of changes to the application, this thesis's secondary goal is also the reusability of the findings. In fact, by understanding the JVM at a deeper level, it could be theoretically possible to apply most of the findings to other applications too, without the need for an in-depth analysis of the code base.

Most of this thesis will plan to optimize a real-time Java application through an exploratory problem analysis. The approach used was systematic, exploratory, and iterative, focusing on precisely identifying and understanding core issues. This process, commonly known as *active analysis* [20], involved an initial phase of extensive problem identification, formulating hypotheses based on preliminary data. Essential to this phase was the gathering of accurate measurements, which required the deployment of sophisticated data acquisition systems to ensure extensive coverage and precise results. A load-testing setup was employed to validate the findings and ensure the robustness of the solutions across various scenarios. This setup was designed to simulate real-world conditions as closely as possible, allowing for broader evaluations that helped refine the problem-solving strategies and ensure the scalability and effectiveness of the solutions. This approach can be defined as iterative because analysis, measurement, and testing phases were conducted cyclically until satisfactory results.

The following sections will explain in detail how the analysis was performed, how data was gathered, and how the solutions were tested on the system.

5.2 Metric collection

5.2.1 Prometheus real-time monitoring

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability [38]. It is primarily used for real-time metric collection. It was originally developed by SoundCloud and has since become a project of the Cloud Native Computing Foundation (CNCF). Due to its robust features and ability to integrate with a wide range of systems, Prometheus is well-suited for monitoring dynamic cloud environments.

Prometheus collects data from a list of targets, which can be viewed as the different services of a bigger cluster. Each target exposes its relevant metrics to a component of the Prometheus system called *collector*, which scrapes and ingests data. Scraping is usually done at constant rates to be able to monitor the selected metrics at any point in time. Prometheus stores all scraped metrics in a time-series database using a highly efficient and compressed storage format built specifically for this purpose. Prometheus stores data as metrics, each identified by a unique name for referencing

and querying purposes. Metrics can be further detailed with any number of key-value pairs, known as labels. These labels can provide information about the data source, such as the originating server, and application-specific details like status codes, source of origin, and more. This flexibility in specifying and querying labels in real-time is what makes Prometheus' data model multi-dimensional., which allows for rich querying and aggregation.

Prometheus allows to export metrics in four different formats [39]:

- **Gauge.** A Gauge is a metric that represents a value at a specific point in time, which can fluctuate. Gauge values are instant and do not represent their trend in time, although they could be reconstructed thanks to the time series they are stored as. In a car, an example of a gauge could be the speedometer, which is used to measure the instant speed of the vehicle.

- **Counter.** A Counter is a cumulative monotonic metric, meaning that it only increases over time. It represents values that naturally accumulate, such as the number of requests served, errors encountered, or tasks completed. Thanks to Prometheus' query capabilities, it is possible to derive numerous other properties from counters, such as their rate over a time interval. In a car, an example of a counter could be the odometer, which measures the total distance traveled by a vehicle. Because the total distance driven so far can only ever increase, it is considered a monotonic measurement and, therefore, fits all criteria to be exposed as a counter.

- **Histogram.** A Histogram samples observations and counts them. It provides a count of observations and a sum of all observed values, usually divided into distinct spans called *buckets*. By diving observations in buckets, Prometheus aims to reduce the size of ingested data and keep cardinality low while keeping track of sparse data. However, because buckets can only be as precise as the number of them, some precision is necessarily lost when dealing with non-discrete quantities. In a car, an example of a histogram could be a system that tracks the engine temperature by measuring a value several times per second. If temperatures are placed in buckets precisely enough, the histogram will show the number of observations that occurred over the car's lifecycle for each temperature bucket.

- **Summary.** Summaries are similar to histograms but are more focused on calculating quantiles. Unlike histograms that distribute data points over buckets based on the boundaries of the observed values, summaries do the same but with percentiles. Summaries can be useful when a metric like a histogram is needed, but quantiles can be aggregated, therefore simplifying the querying process. In general, histograms are always preferable to summaries unless reducing querying times is strictly needed. In modern Prometheus versions, summaries are to be considered obsolete.

Prometheus features its own query language, PromQL (Prometheus Query Language), enabling users to select and aggregate data [40]. Designed to integrate

seamlessly with a time-series database, PromQL offers specialized functionalities for time-based queries and can be very efficient compared to its non-specialized counterparts. Thanks to PromQL, it is possible to retrieve, aggregate, and compute complex operations over any previously ingested time series.

In this thesis, Prometheus was used to collect usage metrics from the operating system and the JVM, as well as some custom-built metrics that allowed the observation of tick times and other game-server-related metrics, such as tick times and load-related metrics explained in detail in Section 3.4.

Other third-party tools, such as Grafana [19], can also be used to query and visualize data, although their use goes beyond the goal of this thesis.

5.2.2 Java Flight Recorder profiling

Java Flight Recorder (JFR) is a performance profiling and diagnostics tool originally developed by Oracle and directly built into the Java Virtual Machine [36]. It is designed to gather detailed runtime information about a Java application with minimal overhead, making it suitable for both development and production environments. JFR records a wide range of data, including method execution times, CPU and memory usage, thread activity, JIT compilations, and garbage collection events. This data is captured as events, which can be analyzed to identify performance bottlenecks, understand application behavior, and troubleshoot issues. JFR's low impact on system performance allows it to be run continuously, providing a comprehensive view of application performance over time. The collected data can be analyzed using various tools, such as Mission Control, also developed by Oracle, to visualize and interpret the metrics, helping developers optimize their applications and ensure efficient operation. One of the most used features of Mission Control is its ability to generate flame graphs, which allow the user to visualize in detail the amount of resources spent inside individual methods of an application. In general, JFR and Mission Control have the capability to view inside the JVM and diagnose possible problems.

Although their purpose is similar, Prometheus and JFR are vastly different tools. Prometheus is a general-purpose monitoring and alerting toolkit designed for collecting and processing real-time metrics from various sources. It excels in monitoring the performance and health of distributed systems, cloud applications, and infrastructure components. Prometheus scrapes metrics from targets, stores this data in a time-series database, and provides powerful querying capabilities through PromQL. It is particularly effective for monitoring large-scale systems, providing insights into metrics such as CPU usage, memory consumption, request rates, and error rates. However, Prometheus does not provide low-level insights into the underlying system, nor does it provide detailed information about performance. On the other hand, JFR is a profiling and diagnostics tool specifically built into the JVM. JFR is designed to gather detailed runtime information about Java applications. JFR is highly suited for in-depth analysis of Java applications, enabling developers to identify performance bottlenecks, understand application behavior, and troubleshoot issues. Unlike Prometheus, JFR data is not designed to be viewed in real-time, cannot be aggregated, and is not designed to work on distributed systems. In general, Prometheus provides a simple but efficient

system to view large amounts of metric data in real time, while JFR provides a tool to analyze performance issues in-depth, but only for isolated systems.

In this thesis, JFR and Mission Control were used to collect and analyze the performance data of the game server. They were particularly useful in detecting issues and bottlenecks in the logic code and allowed to investigate in-depth the correct working of the JIT compilers and the GC. Other tools were also used to analyze further compilation-related issues based on the data extracted by JFR. The particular merit of JFR is its ease of use, which makes understanding the extremely complex processes of a big application easier. Both Prometheus and JFR completed each other, reaching over where the other tool could not provide.

5.2.3 JVM diagnostic logging

Some JVMs, especially newer ones, allow enabling diagnostic logs to better understand the processes happening underneath the regular application. These logs can prove quite useful when trying to understand hidden processes, such as JIT compilation or garbage collection [36]. Because of the JVM's black box architecture, most background processes are completely invisible to both the developer and the application itself. However, these processes can become visible thanks to diagnostic logging, and understanding them is finally possible. Unfortunately, these logs are not a standardized feature. Therefore, different JVM versions might differ quite drastically from others, both in the quality of the logs and in the observable modules. Furthermore, logs are not always human-friendly for analysis, although numerous tools have been developed over the years to aid developers in understanding them.

5.3 Load-testing setup

5.3.1 High-level description of the system

Load testing is performed in an environment that resembles the one used in the live game as closely as possible. The only differences are that traffic is supplied artificially instead of real players, and the infrastructure is only contained within a single cloud region instead of multiple regions worldwide.

The cluster is formed by multiple groups of components that must work together to provide the service. Because each component depends on the features provided by the others, the whole cluster is required to function properly, and no single component can be tested separately. This also applies to the servers that run the game battles themself since, for them to work correctly, all other support components of the cluster must be operational. These are the groups of components that take part in the cluster's operations:

- **Battle servers.** The core of the cluster. It executes all battles and is structured like a previously described real-time game server system in Java.
- **Support components.** All other base components of the clusters. Although they cannot be considered true game servers, they still perform all related support

operations, such as handling and distributing traffic from players, processing logins, matching players together and distributing them to the game servers, and many others.

- **Load generators.** Only present in the load-testing setup. They generate fake traffic to be sent to the rest of the cluster, imitating the behavior of real players. The amount of traffic can be controlled to provide a more or less intensive load to the system. When playing battles, load generators are smart enough to understand the game's rules and submit instances that emulate what a real player would do, although without the reasoning behind it.

- **Monitoring and observability components.** Other support components that are used to monitor the cluster, such as the Prometheus system for metric collection and analysis.

The cluster is supposed to run fully on the cloud. Thanks to Infrastructure-as-a-Service (IaaS) providers, infrastructure can be provisioned automatically, allowing users to plan and configure all the necessary resources. The server application is also deployed automatically on the previously created infrastructure and can be observed remotely to gather data. In general, the detailed work of the IaaS and deployment process is beyond this thesis's scope.

5.3.2 Comparative testing

The load-testing environment allows multiple versions of certain components to run if they are compatible with each other. Compatibility is usually achieved by ensuring the same messages are exchanged between all system components and that, given a generic input, the component's output is exactly the same. If compatibility between components can be ensured, then it is also possible to deploy multiple versions of the same component for research purposes and then observe the behavior of the two.

This comparative testing approach is widely used in various related disciplines. In UX development, some web applications try to optimize their content offering with a technique called *A/B testing* [25]; it involves providing unsuspecting users with two different versions of the same resource and classifying which works best based on the observed behavior. DevOps engineers also use a similar technique called *blue-green deployment* [44], allowing them to quickly deploy new versions of the same internet-based application without causing disruption. It consists of deploying the older and the newer versions of the same software, namely the green and the blue version, at the same time, hoping to spot any behavioral difference that might indicate the newer version is problematic. The approach used to perform all tests presented in this thesis is similar. It leverages the multi-versioning capabilities of the cluster to deploy multiple versions of the same game server. Because all game traffic is automatically balanced across all game instances, each game server will receive a comparable amount of input traffic to the others. Therefore, this approach allows for the simultaneous testing and comparison of multiple strategies.

This approach provides several benefits compared to more traditional approaches, where each test is performed separately. First, it provides perfectly comparable results since all tests are performed in almost identical conditions and at the same time. When testing a complex system, it is common for some variables to be hidden. If that is a concern, then this approach ensures that all different testing groups are subject to the same hidden variables and will keep testing more rigorous. This comparative testing approach also presents shorter test periods since all possible configurations are tested simultaneously rather than separately, further optimizing the time taken to test. Finally, this approach is more reliable since it consists of a single trial rather than multiple versions of the same one executed at different times. For these reasons, comparative testing was chosen as the main testing method.

5.3.3 Automatic test case generation

Creating new testing setups to test new variables can become time-consuming. Each trial requires a different internal server configuration and possibly a different infrastructure. Furthermore, in some cases, multiple variables need to be tested simultaneously, which requires the creation of multiple test groups, one for each combination of every value of the selected variables. To shorten and simplify this process, this thesis involved the creation of a fully automated test case generation script. The script is able to generate internal configuration and IaaS configuration files, which are given the properties of the test to perform. The automation consists of a script that parses through a single configuration file that collects all information about variables and their respective properties and generates all lower-level configurations needed. If multiple variables are specified, the automation will generate a number of test groups equal to the Cartesian product of all possible values of every variable, each with its unique properties.

For example, to test for different amounts of memory and different Java versions simultaneously, the following *YAML* configuration file can be used:

```
groups:
  - name: java_version
    values: ['java11', 'java21']
  - name: memory
    values: ['6GB', '8GB']

properties:
  - targets: [ 'java_version=java11' ]
    properties:
      ami: jdk11_image
  - targets: [ 'java_version=java21' ]
    properties:
      ami: jdk21_image
  - targets: [ 'memory=6GB' ]
    properties:
      jvm_arguments:
        memory: -Xmx6G -Xms6G
  - targets: [ 'memory=8GB' ]
    properties:
```

```
    jvm_arguments:
      memory: -Xmx8G -Xms8G
```

Which, in order, would automatically generate the following testing groups, each with their corresponding properties set:

- Java 11, 6 GB of memory
- Java 11, 8 GB of memory
- Java 21, 6 GB of memory
- Java 21, 8 GB of memory

This approach optimized the time taken to create new test cases, and human error was reduced to a minimum since all relevant information was automatically generated. The script also presents more advanced features, such as the possibility to target multiple variables at the same time, the ability to give priority to those values that would be repeated and therefore overwritten, the ability to set default values for all instances, and templating capabilities to generate low-level configuration files fast and efficiently.

5.3.4 Repeatability and test validation

Particular effort was put into ensuring all tests conducted were valid and repeatable. In general, to ensure validity and repeatability, the following measures were adopted:

- **Multiple instances per group.** To avoid issues related to hard-to-predict behaviors of instances and outliers, at least five instances were deployed for each test group. Measurements are then compared and averaged between the same group, resulting in a hybrid measurement that should better represent the group. In case some instances behaved completely differently than the rest of their group, results were nullified, and the test repeated, for example, when suspecting the occurrence of hardware issues.

- **Load unbalance monitoring.** Testing multiple instances in parallel will yield correct data only if the load level of each is comparable. To ensure acceptable load levels, each instance load is compared to the average of all instances. If the difference exceeds 5%, it is likely that the load balancing algorithm is not working as intended, and the test must be repeated.

- **Deterministic test generation.** The algorithm to generate test cases is deterministic, which means consequent runs of the same configuration will result in the same output. By leveraging this mechanism, it is possible to repeat the same test multiple times, and the results will likely be the same. In the case of unforeseen external factors, because testing is done in parallel, results might vary but still present the same macroscopic properties, unless in the rare case of a hidden variable that might favor a group compared to the others.

6 Evaluation and Results

This chapter focuses on three critical areas: game engine problems, runtime and processor evaluation, and JVM fine-tuning. Each section comprehensively analyzes the respective aspects, offering insights into the performance, efficiency, and optimization strategies employed. After the first one, all sections only consider the optimized version of the game server as the baseline.

6.1 Game engine problems

6.1.1 Introduction to the analysis method

This first section addresses the various problems encountered within the game engine, the part of the game server that effectively runs battles. Game engines are complex systems that require meticulous coordination of numerous components to deliver seamless performance. This section identifies the common issues that arise, such as performance issues during game mode changes and player spikes. Through systematic problem identification and exploratory analysis, the aim is to uncover the root causes of these issues and propose viable solutions to enhance the overall performance and stability of the game engine.

6.1.2 Problem analysis

Upon initial analysis, the system seems to act poorly during the warm-up phase immediately after startup. In particular, the game servers show an unexpectedly high average CPU usage, which is more or less common across all tested instances, which can indicate poor related performance.

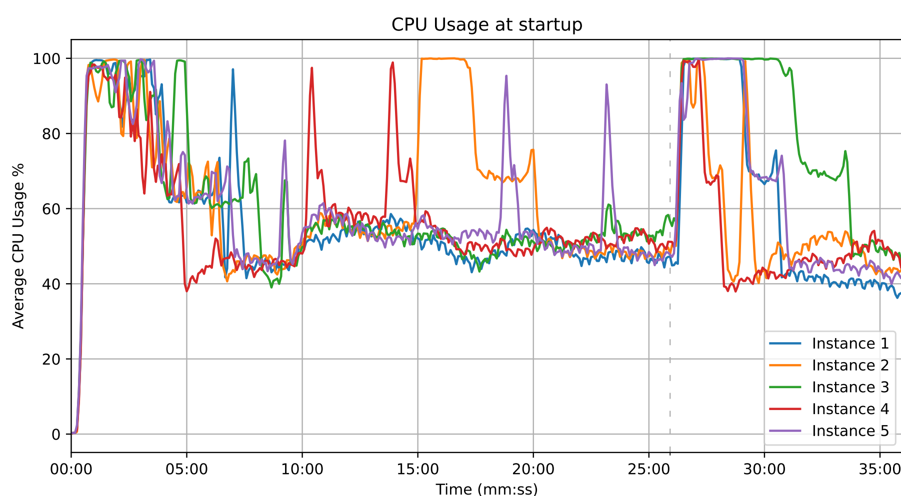


Figure 3: CPU load during startup and game mode change.

Figure 3 shows extremely high CPU usage, often reaching the maximum value of 100%. After an initial warm-up period where all instances reach the maximum processor usage for roughly 5 minutes, the average value decreases. However, certain instances can show seemingly random peaks even after the initial spikes. In some cases, these CPU peaks can last for minutes. This poorly predictable behavior continues for over 20 minutes, after which a programmed game mode change occurs simultaneously for all instances. During a game mode change, the game servers make a new and different kind of game variant available, replacing one that was playable before. Seconds after this game mode change, all instances reach the maximum CPU usage again and maintain this state for up to 5 minutes, returning to normal levels.

This behavior can be defined as problematic for many different reasons:

- **Extremely high CPU load.** Game servers are real-time systems and, as such, are very sensitive to CPU timings. As explained in Chapter 2, real-time systems can hardly utilize all compute resources efficiently, or else the risk of missing deadlines is too high to be acceptable. A high CPU load, in this case up to 100%, means the processors have little to no idle time available, which means missing deadlines can be fatal to the system due to its inability to compensate given the absence of resources to do so.

- **Long CPU spikes.** During normal computing conditions, CPU spikes can occur. During a CPU peak, the system fully utilizes all compute resources. However, in regular circumstances, CPU peaks should last for seconds, or else the system, especially in real-time, can suffer serious faults. In this case, CPU spikes could last up to 5 minutes in certain instances, meaning that the game server processes were most likely compromised.

- **Heavy game mode change.** The system reacted extremely poorly to a game mode change, which should not present an issue in normal circumstances. In general, after an initial warm-up time where high resource usage is considered normal, Java applications should start being optimized further and further by the JIT compiler. Unfortunately, this is not the case in this test since the application reacted poorly to change even after more than 25 minutes of warm-up.

- **Nonuniform behavior of identical instances.** Despite all instances being identical and all generated loads being roughly the same, all instances performed differently from each other. This behavior is even more worrying if the fact that most of the game logic used in the game servers is deterministic and inputs and load balancing vary very little between instances.

Effects on player experience

As hypothesized, excessive CPU loads negatively affect the player experience, as shown by the graphs in Figure 4. In general, the following aspects are well visible from the graphs. All measured metrics were explained in more detail in Section 3.4.

- **Tick execution time.** Although the 50th percentile of tick execution time was stable and close to 300 μs, the 90th, 95th, and 99th percentile started spiraling out of control as soon as the server started. Execution time returned to regular levels after roughly 7 minutes from the start, but it started showing a similar pattern when the game mode change happened. From a player's point of view, execution times of up to 2.5 s in the worst case are totally unacceptable, especially given the theoretical upper limit of 1 s [55]. Furthermore, such response times would most likely cause the client to disconnect after not receiving an update for too long.

- **Tick wait time.** Tick wait time was more or less stable during the entire testing period, with the clear exception of the 50th percentile, which reached values close to zero during tick execution spikes. An in-depth explanation of the phenomenon will be provided later.

- **Tick total time.** Because tick execution times are extraordinarily high during critical phases, tick total times almost perfectly reflect the trend, as shown in the graph.

- **Tick drift.** Tick drift is the most concerning metric measured in this test. Despite being required to be as low as possible, drift started increasing slowly but steadily during the critical phases. In some cases, tick drift reached unacceptable values of more than a minute and accumulated over time, which would probably cause battles to be shut off due to the extreme lateness shown in the graphs. When tick drift reaches these levels, the game server is most likely out of control and completely overloaded.

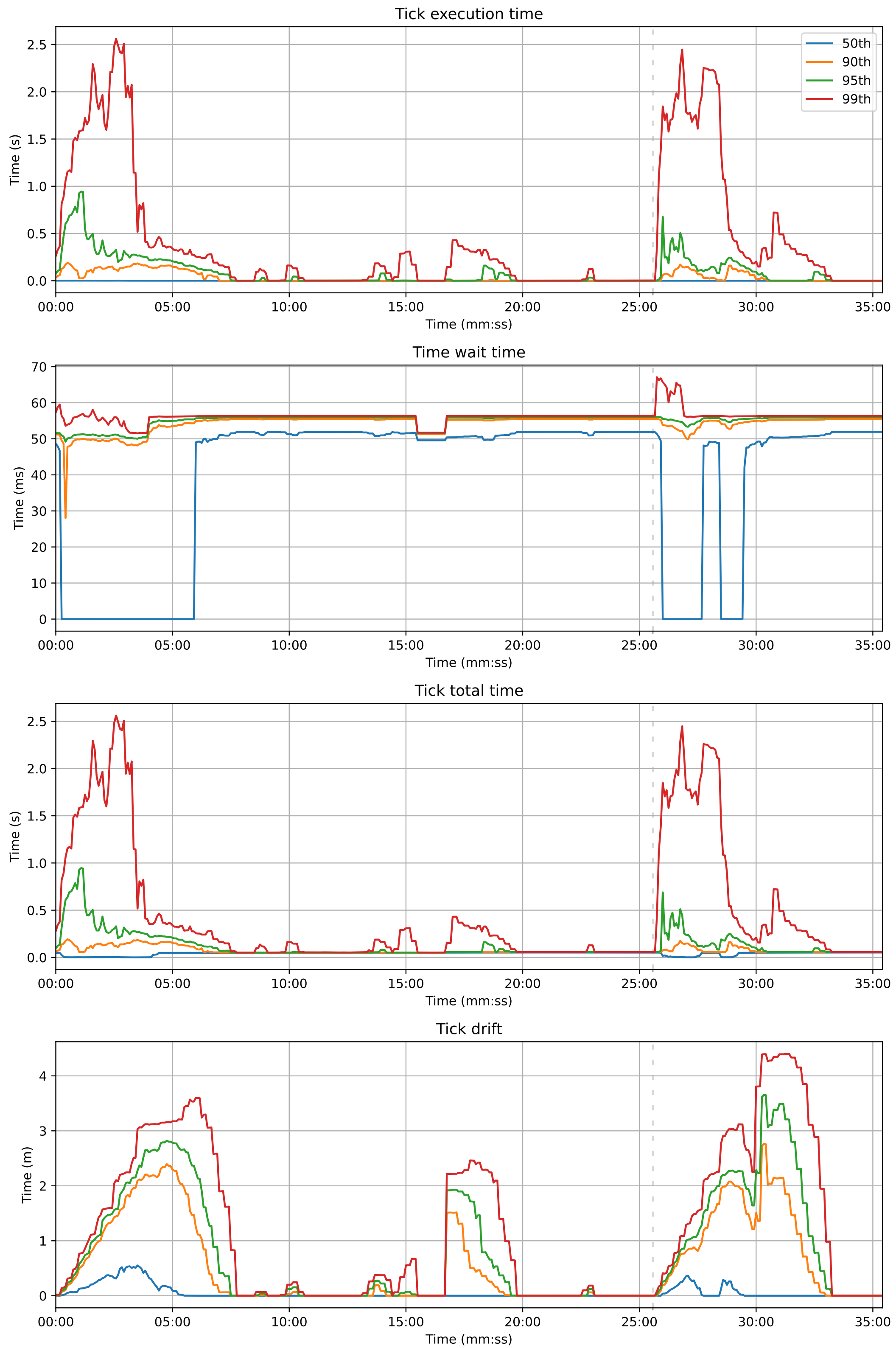


Figure 4: Various tick measurements during startup show the server's problematic behavior and possibly poor player experience.

JIT compilation issues

The JIT compilation time spent by each instance, like the ones shown in Figure 5, highlights a clear correlation between high CPU loads and time spent compiling, although not perfectly. When the CPU load drastically increases in one instance, the time the JVM spends on compilation is also measurable in seconds. This data might indicate an issue in JIT compilation that requires the Java platform to spend more time than normally necessary in compiling methods. This theory is confirmed by the JIT compilation logs produced by the JVM, which are omitted for confidentiality reasons. Observing the logs shows that the C2 compiler spends most of its time re-compiling the same method multiple times due to deoptimizations. This phenomenon is peculiar since multiple de-optimization phenomenons happening simultaneously on multiple instances and in a short time interval is not common, and perfectly explains the CPU spikes and the bad timings.

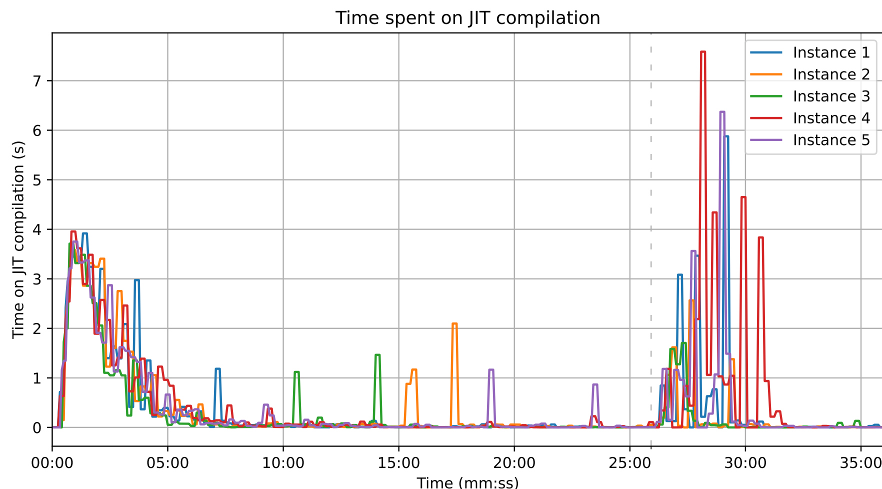


Figure 5: JIT compile times almost perfectly correspond to the CPU and latency spikes.

Upon further investigation, it becomes clear that only a handful of methods are causing these re-compilation problems. When analyzed, all methods are long and complicated, with several nested branches and many conditions that could trigger a deoptimization and consequent recompilation. What is even more concerning is the presence of several different branches that use the currently played game mode as a condition. By introducing these branches, the C2 speculative compiler struggles to predict the optimized path of execution correctly, and therefore, the same method can be recompiled multiple times. In particular, as mentioned in Section 4.3.1.3, the C2 compiler will try to predict what branches can be removed from the application based on their previous usage during the C1 profiling phase. Because C1 profiling happens while a certain game mode is being played, a sudden change of game mode will render many of the previously implemented speculations incorrect, triggering a deoptimization.

Even if only a certain branch was incorrectly speculated, the JVM tiered compilation strategy only works on a method level; therefore, the entire method will need to be brought back to the previous level available, causing a performance drop. Because these problematic methods are so long and complex, several deoptimizations can and will happen during startup, when the collected profiling data is insufficient to predict the routine's behavior, and during a game mode change, when the previous behavior changes completely. This hypothesis partially explains why many decompilations happen but fails to explain why CPU load increases drastically, and player experience degrades to unacceptable levels.

Deoptimization loop

In Table 1, it is possible to notice how, in tiered compilation, going from Level 4 (C2) to Level 3 (C1 with full profiling) can cause a significant performance drop. The same path is taken by methods that are experiencing deoptimization, which means that the performance of those methods will decrease substantially and without any forewarning. To make matters worse, a deoptimized method will stay at its less-performant state for as long as it takes the C2 compiler to operate on that routine. Given how slow C2 is and how it could operate with a substantial queue of other methods to recompile, the unoptimized method might stay in that state for a substantial period of time, causing an incredible drop in the whole application performance.

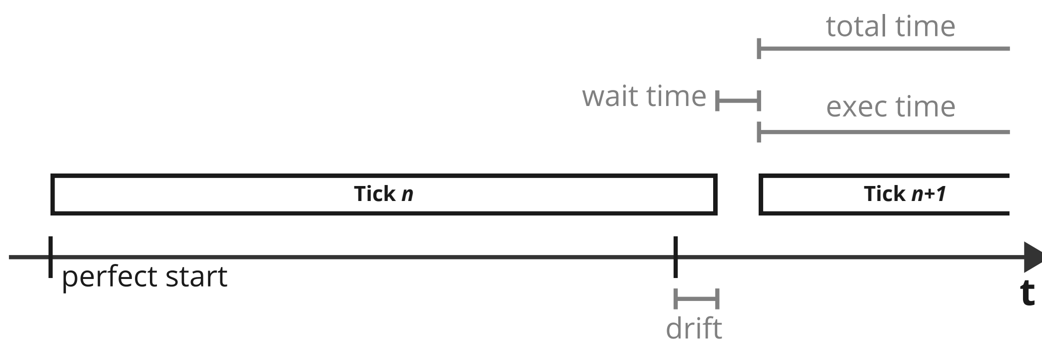


Figure 6: Visual representation of the time spans of a tick during a deoptimization phase.

In the meantime, the scheduler and the rest of the game server are unaware of this performance drop; in this very case, they are unaware that performance drops like this could be possible. Figure 2 shows what the main tick metrics explained in Section 3.4.2 represent in a normal scenario. However, the new scenario will probably look like Figure 6 because of deoptimization. Because the execution time is now slower than ever, some ticks will exceed their deadlines and occupy part of the time windows that the upcoming tick should have occupied. This phenomenon will decrease the wait time to nearly zero since the upcoming tick is already late. Furthermore, the tick drift will increase steadily and theoretically could continue to increase to infinity since all

deadlines will be missed until the deoptimization phase is over. For these reasons, the scheduler cannot react to this sudden change in the execution speed of real-time tasks, even while observing a drastic increase in missed deadlines.

The data completely confirms the hypothesis. In particular, it explains why tick drift slowly spirals out of control, tick execution and total time increase drastically and wait times drop to zero.

6.1.3 Solutions

6.1.3.1 Speeding up the JIT compiler

Although counterintuitive, a possible solution would be to increase the amount of threads dedicated to the JIT compiler. Even if, upon initial analysis, the JIT compiler seems to be spending too much time on recompilations due to deoptimizations, the actual performance problems are caused by those methods that stay in an unoptimized state for long periods of time. This can usually happen when a large number of methods are awaiting to be compiled in a queue, but the C2 compiler is not fast enough to process them at an acceptable speed. By default, the JVM will allocate the JIT compiler one or two threads, depending on the amount of physical cores available on the system. Thanks to the Java startup argument *-XX:CICompilerCount=<n>*, it is possible to manually set the number of dedicated threads, where <*n*> is the number. By increasing the number of threads to 8 on an 8-core virtual machine, it is possible to observe a reduction of ticks that took more than 55ms to process of up to 5% during critical phases. Although a simple reduction in late ticks is not enough to fix the problem, this approach can help mitigate it at virtually no cost, especially when dealing with a compiler-intensive application.

6.1.3.2 Disabling speculations with Zing JVM

Because the main problem analyzed in this section is mainly caused by mistakes that occur when the C2 JIT compiler tries to speculate some uncommon routines, one may wonder a possible solution would be to disable speculations entirely. Unfortunately, the vast majority of JVMs do not allow such an option, and probably for a good reason. Speculations are an important mechanism that allows the Java platform to produce far more optimized code at very little performance cost. Without speculations, the C2 compiler is deprived of one of the best optimization tools available. Therefore, the generated machine code will greatly suffer from their absence, so it might be worth completely disabling tiered compilation.

Some JVMs, however, use a different compilation strategy, allowing their users to disable speculations at will. One such Java Virtual Machine is Zing (also known as Azure Platform Prime), provided by Azul [5]. Unlike most JVMs, which are free and open-source, Zing is a commercial product that provides many features in addition to those already present in most JVMs. The two most important ones are C4GC (Continuously Concurrent Compacting Collector) [51], a generational pauseless GC, and Falcon, a complete replacement of the C2 JIT compiler. Falcon promises up to

20% better application performance by generating more optimized compared to C2 at the expense of longer compilation times. Unlike the C2 compiler, Falcon allows the user to fine-tune its inner workings thanks to a set of startup arguments that can toggle various features, including its speculative behavior.

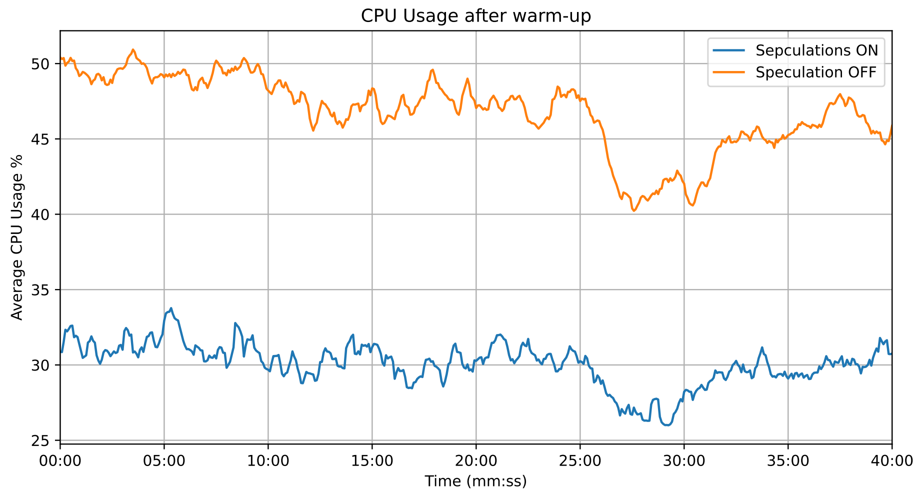


Figure 7: CPU load with speculations deactivated, after warmup.

By disabling most of Falcon's speculative behavior thanks to the startup flags *-XX:-FalconUseBranchElimination -XX:-FalconSpeculateUnreachedJumps -XX:-FalconSpeculateUnreachedSwitchCases -XX:-FalconSpeculateUnreachedCalls -XX:-FalconSpeculateNoThrowCalls* (notice how the minus symbol before the argument name is used to disable it), the game server effectively stops showing any CPU or tick spikes, as expected. Unfortunately, disabling speculations has the clear consequence of generating less performant code, as shown in Figure 7, which shows the behavior of Zing after the warm-up, with and without speculative behavior. Although this approach fixes the problem, the CPU usage increase that results from it makes it hard to justify, especially considering that it requires purchasing a commercial JVM license without actually using any of the available optimization features.

Modern versions of the HotSpot JVM also introduced some features that would allow fine-tuning compiler settings per method. This feature is known as *Compiler Directives* [23] and could allow the user to dictate the JIT compiler should behave differently when handling certain routines, just like more granular startup arguments. Unfortunately, no parameter can fully disable speculations, and those that should avoid performing aggressive optimizations showed no real benefits when tested.

6.1.3.3 Making the application easier to speculate

By far, the most effective method to avoid deoptimization problems is refactoring the application to make it easier for the C2 compiler to speculate. Unlike previous improvements, this requires the knowledge of the codebase and several changes to it, although many of them could be considered trivial.

The key strategy to perform these changes is divided into two parts:

1. **Identify hard-to-predict methods.** This can be performed by analyzing flame graphs obtained by JFR or the JVM compilation logs. Hard-to-predict methods are those parts of the code that are subject to many deoptimizations over time and are executed more frequently than others. These routines are usually the cause of most JIT compilation issues.

2. **Split those methods in easier-to-predict chuks.** Because compilation is performed per method, splitting those problematic functions into smaller and easier-to-predict methods should solve any issues. This can be performed by analyzing all branches of the original methods and identifying those that could cause speculative problems, such as game mode-dependent branches. Once the problematic code paths are divided into separate functions, deoptimization will still occur to them, but they will also involve a smaller part of the application. If problematic chunks become small enough in size, their problematic behavior will be almost invisible to an external observer.

6.1.4 Evaluation

By increasing the number of threads dedicated to the JIT compiler and making some parts of the application easier to speculate, the performance issue was mostly mitigated. Figure 8 shows the same tick times as Figure 4 after the issue has been resolved. Note, however, that the y-axis is now on a scale of milliseconds, while in Figure 4, it was in seconds. This graph clearly shows how spikes are almost completely gone and, in any case, well within the acceptable ranges. The graphs also show that a small peak occurs during a game mode change, although exceptionally small compared to the previous one. This testifies to how deoptimizations still happen during game mode changes; only their behavior is quickly resolved, and execution resumes normally.

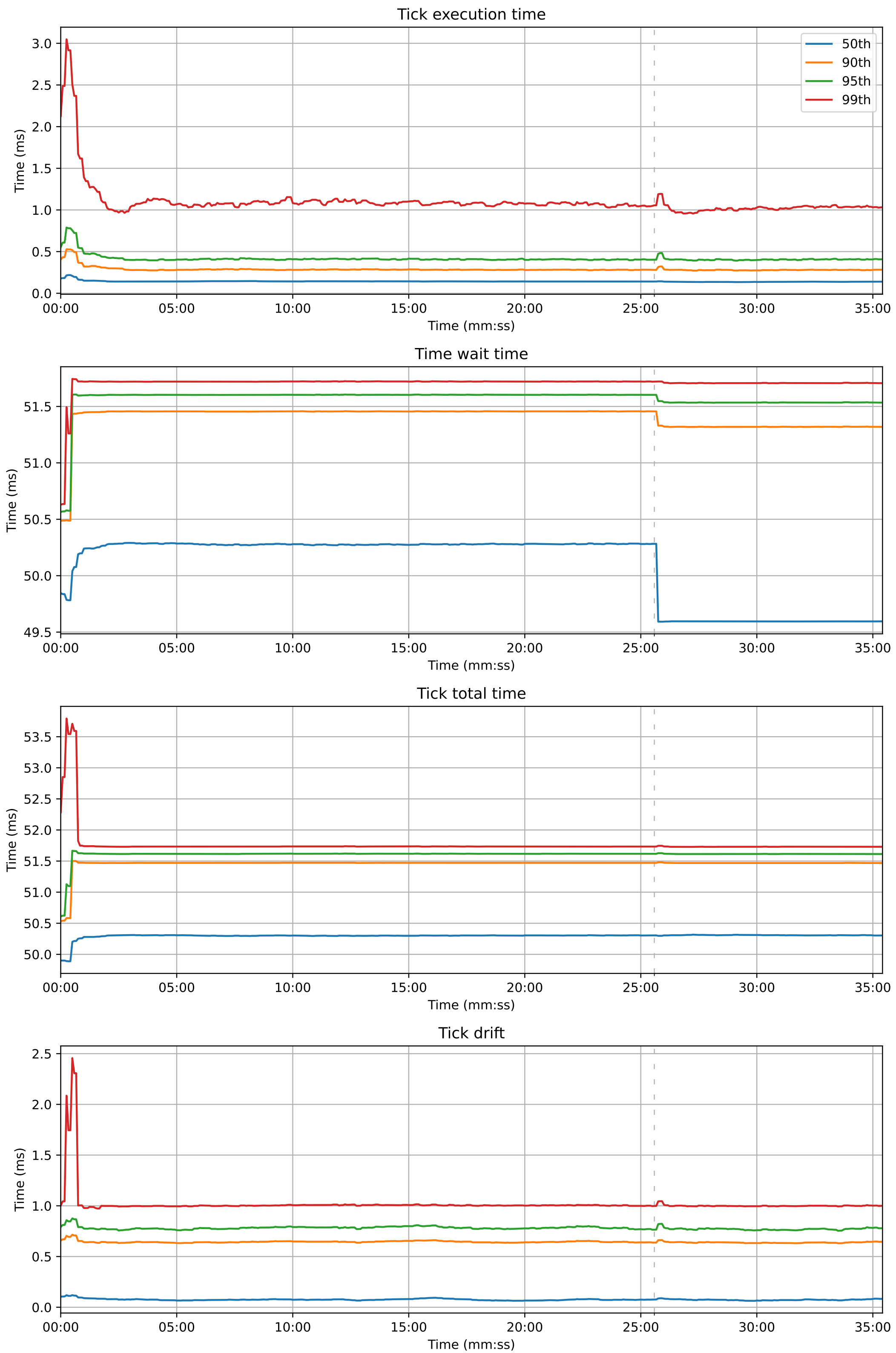


Figure 8: Same tick measurements during startup, after resolving the issue shown in the previous graph in Figure 4. The y-axis is in milliseconds, while in Figure 4, the y-axis is in seconds.

In general, the following changes are noticeable:

- Execution time is always under control, and 99% of the time, tick execution happens within *1 ms*.
- Wait time is almost perfectly constant, except for the game mode change, which causes more battles with fewer players in them. The result is a slightly decreased wait time due to an increase in the number of real-time tasks.
- The median total tick time is extremely close to the targeted value of *50 ms*, and even the 99th percentile is less than *2 ms* late.
- Tick drift is evenly distributed between *0* and *1 ms*. This is to be expected since the current scheduling system has a precision of *1 ms*. Therefore, under normal circumstances, all ticks will be up to 1ms late from their perfect execution starting point.

Another interesting figure is the median execution time after warm-up, which dropped from roughly 230 μs to roughly 140 μs, a −39% reduction. Thanks to simple refactoring, the C2 compiler could speculate more effectively and optimize the generated code.

Thanks to these optimizations, a player and battle load five times greater than previously possible on identical hardware was achieved without degrading the user experience, which could allow reducing the number of compute instances by up to five times, leading to significant monetary and energy savings.

In conclusion, all performance issues related to the warm-up phase of Java and consequent game mode changes have been resolved completely.

6.2 Runtime and processor evaluation

6.2.1 Introduction to the analysis method

This second section is focused on the runtime and processor evaluation. The evaluation aims to optimize cost and performance by switching to a more cost-effective platform or one that provides the best player experience in those cases where cost becomes irrelevant.

The underlying platform consists of three distinct parts: the Java Virtual Machine (JVM), the Operating System (OS), and the cloud Virtual Machine (VM).

- **VM.** For the VM testing, only cloud instances provided by Amazon Web Services (AWS) will be considered due to compatibility issues and simplicity. In general, AWS instances that use the Nitro hypervisor [4] should show performance results close to their native counterparts, thanks to the virtualization platform's small footprint. Therefore, all VMs will be considered equivalent to physical machines, and AWS's naming convention will be used.

- **OS.** The OS will not be included in the tests for several reasons, such as compatibility issues and too many options to test.

- **JVM.** For the JVM testing, all different Java distributions will be considered identical. Generally, most vendors should comply with the Java specifications and submit their JVM to the Java Technology Compatibility Kit (TCK). Doing so ensures full compatibility between vendors and compliance with the standard, thanks to a suite of hundreds of thousands of tests. Furthermore, the most widely available Java distributions are almost completely derived from the HotSpot JVM; therefore, performance is expected to be nearly identical between distributions. Those distributions that do introduce major changes to the HotSpot architecture or do not derive from it, such as GraalVM or Zing, will not be considered in this tests. For simplicity reasons, Amazon Corretto was the chosen JVM for these tests.

6.2.2 Evaluation of Java version

Java provides new and improved features over time, thanks to the release of major versions. A new major version is expected to be published bi-yearly and must maintain full compatibility with all previous versions. In general, an application compiled for a certain target version will be compatible with every major release greater than the target one but not compatible with all inferior versions.

The aim of the test is to verify if newer Java versions could potentially improve the performance of older applications without requiring any substantial changes whatsoever. The target version of the game server is Java 11, therefore, its performance will be compared with Java 21, the most recent Long-Term Support (LTS) version.

Java 21 metrics show surprisingly large performance improvements over Java 11, especially in worst-case latency.

- CPU usage is lower, as shown in Figure 9.

- Java 21 shows almost 5 times fewer late ticks (which took more than 55 ms to process) than Java 11.

- Tick execution times have dropped from 148 μs to 139 μs, a 6% decrease.

Justifying most of these results is not easy. Unfortunately, many changes that occur to the inner workings of some of Java's components are almost completely untracked, if not internally only. For this reason, it is nearly impossible to understand which practical code change corresponds to an actual performance improvement, especially between major versions that are so far apart.

Because Java 21 has proven to be the better option, and without any drawbacks, all further testing will be conducted with Java 21.

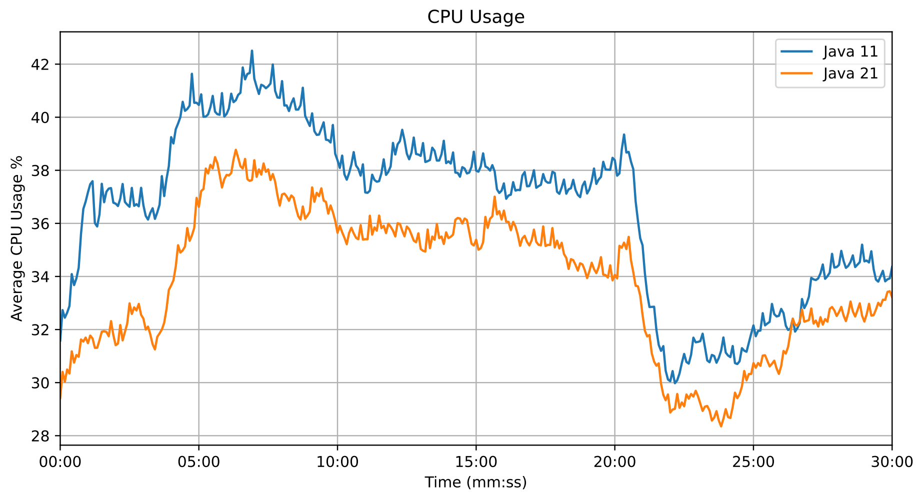


Figure 9: CPU load difference between major Java versions.

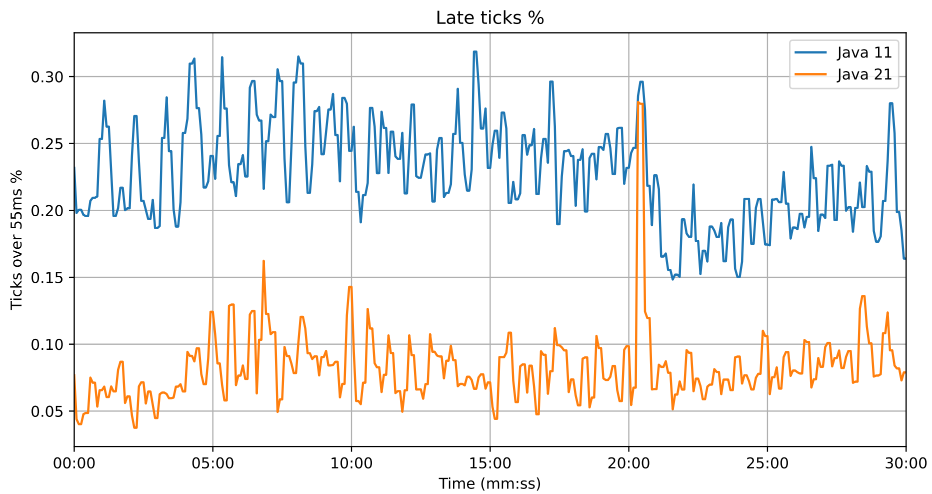


Figure 10: Late ticks difference between major Java versions.

6.2.3 Evaluation of faster processors

Newer processors have the potential to provide better performance for a lower price. Because of the scalable nature of the analyzed game server, a more performant server has the capabilities of not only providing a better player experience but also hosting more battles than its competitors, significantly lowering cost.

Under normal circumstances, the system runs on AWS *c5.2xlarge* instances. AWS naming scheme for instances follows these rules [2]:

- First letter: workload type, in this case *c* stands for *compute optimized*.
- Number: generation, in this case, AWS's fifth generation.
- Optional second letter: CPU vendor, absent in this case. If present, the following letters are used:
  - *i*: Intel processor, x86 64bit architecture
  - *a*: AMD processor, x86 64bit architecture
  - *g*: Graviton processor, ARM 64bit architecture, AWS's self-produced processors
- Instance size: instance size, in this case *2xlarge*.

At the time of writing this thesis, the 7th generation of instances is the most recent available. Therefore, all 6th and 7th generation instances will be compared to the currently used *c5*. For consistency reasons, all tested instances will have the same size (number of cores and amount of RAM).

Graviton instances are available starting from the 6th generation and, unlike the others, are built upon the ARM architecture. CPU Architecture changes are often problematic due to compatibility issues between processors. However, thanks to Java's portability, executing the same application on both architectures simultaneously required minimal code changes.

Each instance time was tested in both CPU usage and 99th percentile of tick execution time. In both metrics, lower values indicate better performance and are often correlated. Each test was executed three separate times with low, medium, and high player load. All JVMs were already warm during every test, therefore, game mode changes and other impactful operations should have minimal overhead. All reference prices have been obtained from AWS's *us-east-1* region in North Virginia. A final "Bang-per-Dollar" (*BP$*) value was also computed to express the performance improvement over the hourly cost compared to the baseline, which is *c5.2xlarge*. For example, a *BP$* value of *1.5x* means that a certain instance is offering a 50% improvement of its performance, compared to a *c5.2xlarge*, for the same price.

The data in Table 3, Table 4, and Table 5 show upgrading to newer instances might make sense in every analyzed case. Of particular notice are c6a, c7a, and c7g instances, which perform very similarly. Despite being less powerful, Graviton processors are still performing well thanks to their low price, mostly because of the low energy

Instance type	**Price** ($/h)	**CPU Usage**	**BP$**	**Tick p99** (μs)	**BP$**
c5.2xlarge	0.34	27.9%	1.00x	980	1.00x
c6a.2xlarge	0.306	21.9%	1.42x	657	1.66x
c6g.2xlarge	0.272	31.4%	1.11x	1039	1.18x
c6i.2xlarge	0.34	22.3%	1.25x	767	1.28x
c7a.2xlarge	0.41056	17.0%	1.36x	520	1.56x
c7g.2xlarge	0.29	25.1%	1.30x	723	1.59x
c7i.2xlarge	0.357	22.8%	1.17x	698	1.34x

Table 3: Low load level

Instance type	**Price** ($/h)	**CPU Usage**	**BP$**	**Tick p99** (μs)	**BP$**
c5.2xlarge	0.34	46.6%	1.00x	1051	1.00x
c6a.2xlarge	0.306	35.7%	1.45x	680	1.72x
c6g.2xlarge	0.272	48.5%	1.20x	1005	1.31x
c6i.2xlarge	0.34	36%	1.29x	798	1.32x
c7a.2xlarge	0.41056	26.2%	1.47x	512	1.70x
c7g.2xlarge	0.29	39.1%	1.40x	726	1.70x
c7i.2xlarge	0.357	35.8%	1.24x	716	1.40x

Table 4: Medium load level

Instance type	**Price** ($/h)	**CPU Usage**	**BP$**	**Tick p99** (μs)	**BP$**
c5.2xlarge	0.34	66.8%	1.00x	1244	1.00x
c6a.2xlarge	0.306	50.5%	1.47x	773	1.79x
c6g.2xlarge	0.272	65.2%	1.28x	1083	1.44x
c6i.2xlarge	0.34	51.3%	1.30x	911	1.37x
c7a.2xlarge	0.41056	35.1%	1.58x	536	1.92x
c7g.2xlarge	0.29	52.4%	1.49x	760	1.92x
c7i.2xlarge	0.357	50.2%	1.27x	803	1.48x

Table 5: High load level

consumption offered by the ARM architecture. For this reason, c7a and c7g reach the same price-performance ratio in two opposite ways: the AMD processor offers by far the best performance available but at a high price, while the Graviton processors offer good performance for an extremely competitive price. Intel processors, on the other hand, have performed the worst, both in terms of pure performance and when accounting for the price.

In conclusion, according to the data, switching to a c7a or c7g instance might allow for the same number of battles and players at almost half the price. Thankfully, Java's portability simplifies the conversion process.

6.2.4 Evaluation of disabling kernel-level mitigations

Linux distributions, by default, apply some kernel-level patches to mitigate hardware vulnerabilities. Modern Linux kernel versions can apply tens of patches to ensure security to their users, especially against side-channel attacks. In this class of security vulnerabilities, an attacker aims to gather information or influence the program execution of a system by exploiting the indirect effects of the system or its hardware. These attacks become particularly relevant in multi-tenant systems where multiple parties, one of which could be malicious, have access to the same set of resources, such as a shared processor on the cloud. By mitigating these vulnerabilities, the Linux kernel uses some of the CPU time on extra operations to ensure an attacker does not have access to resources that should be hidden. For this reason, a performance overhead is to be expected, especially when executing operations with elevated rights, which are the ones actively mitigated by the kernel.

Thankfully, these attacks do not directly affect the game server use case. First, although all computing is done on shared processors, the AWS Nitro system prevents all sorts of side-channel attacks by ensuring each tenant has access to a restricted

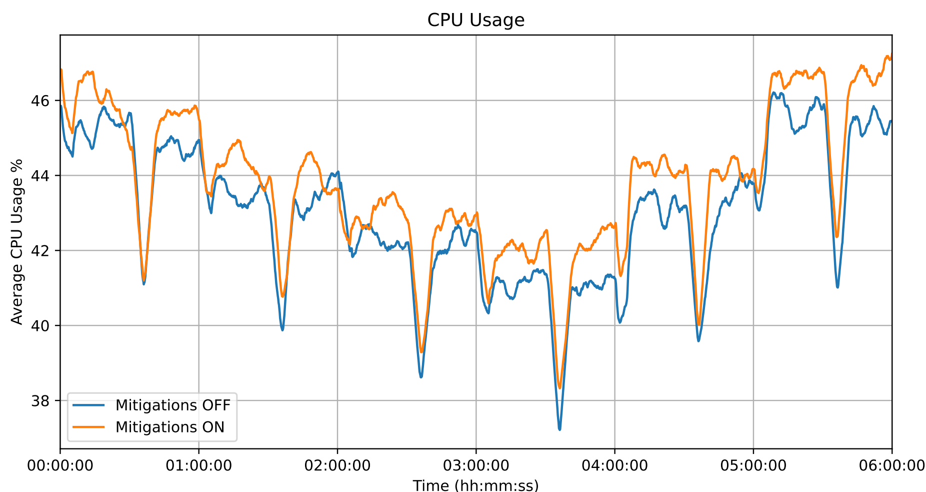


Figure 11: CPU load comparison between Linux kernel mitigations.

set of resources. From a security point of view, kernel mitigations become partially irrelevant because Nitro will already provide patches for them on a lower level than the kernel. Furthermore, mitigations become less relevant if resources are hardly a possible target, like in this scenario. As a game server, the system does not treat sensitive data, therefore, an attacker would hardly view it as a feasible target. On the other hand, leaving kernel mitigations on provides an extra layer of security in case that provided by the hypervisor is not effective enough.

For these reasons, this test aims to understand the performance implications of kernel-level mitigations. Given the target's small attack surface, if the performance gains are big enough, it might be worth turning mitigations off.

Turning mitigations off shows a small but consistent performance increase, especially in CPU usage. Figure 11 shows a subtle CPU load decrease when mitigations are turned off, especially visible when the processor usage is more stable. This difference can be explained by the fact that the game server relies heavily on context switches, which are used by the kernel when transitioning between two different processes. Because context switching is inherently sensitive to many vulnerabilities, their mitigations have a big impact on applications that rely heavily on them, like in this case. However, given how small their impact is, especially compared to the benefits of moving towards newer CPU generations, disabling mitigations might simply not be worth it.

In conclusion, kernel-level mitigations do have a measurable impact on the game server performance, probably because of the heavy use of context switching. However, the performance benefits are not big enough to justify the possible security risk.

6.3 JVM fine-tuning

6.3.1 Introduction to the analysis method

The last section is dedicated to JVM fine-tuning, which is the process of choosing the right parameters or execution strategy to get the most performance out of the Java platform. Although often overlooked, JVM tuning can help increase the performance of any application with minimal code changes and almost no cost. However, extensive knowledge of the inner workings of the JVM might also be required. It is also noteworthy that badly performed tuning or invalid options can lead to serious performance and stability issues. Therefore, extensive testing is required.

6.3.2 Evaluation of garbage collection tuning

Selecting the right Garbage Collector (GC) can have a great impact on performance. Section 4.3.2.2 explains in depth the differences between GCs and which one should be selected for the right use case. This test will compare the performance of the three modern GCs available on Java 21: G1GC, ZGC (with and without the generational option), and Shenandoah. Serial and Parallel GCs have been purposefully excluded from this testing because of their legacy nature. Likewise, Epsilon GC was excluded

from this testing because it is inherently unsuitable for applications with live memory allocation, and CMS was excluded due to its deprecation.

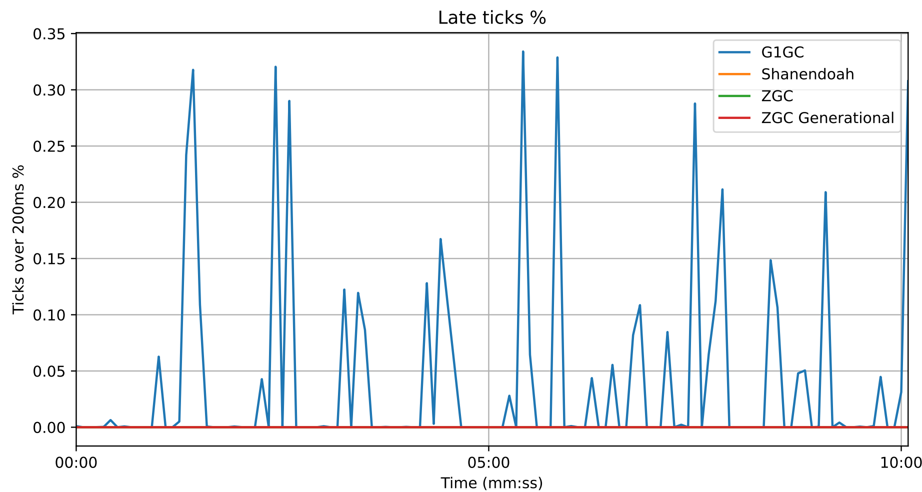


Figure 12: Late ticks (over 200 *ms*) comparison of different GCs. Because of the high spikes, it is clear why G1GC is unsuitable for a game server workload.

As shown in Figure 12, it is immediately clear that G1GC is not fit for real-time systems. Unlike all the others, G1GC is the only garbage collector whose goal is to achieve high throughput compared to low pauses. The consequence is low CPU usage compared to the other methods, at the expense of occasionally high latency and missed deadlines. Because in real-time systems deadlines are the most important target metric, G1GC is not a feasible option and, therefore, will be excluded in all future comparisons. It is also noteworthy that G1GC would have been the default garbage collection strategy of the JVM, which shows how important tuning some parameters of the Java platform is.

Figure 13 shows the percentage of ticks that took more than 55 *ms* to complete. Despite their differences, all GCs behave almost the same, with higher but acceptable spikes during collection times. This is also to be expected since the main goal of pauseless GCs is to minimize lag by reducing pauses to a minimum.

Analyzing tick execution times, the 99th percentile in particular, also shows an interesting pattern. While ZGC reports an execution time of 1.22 *ms* and ZGC *Generational* of 1.21 *ms*, Shenandoah's execution time reaches 1.42 *ms*, 16% more than the other two. This data highlights how Shenandoah, despite operating on similar data, has a higher overhead than ZGC but without any tangible performance benefit. On the other hand, ZGC performs the best with or without the *Generational* operation mode.

For these reasons, ZGC was chosen as the best-suited garbage collector. Because the *Generational* mode did not seem to introduce any difference and because it's still highly experimental, the final choice for garbage collectors was ZGC non-generational.

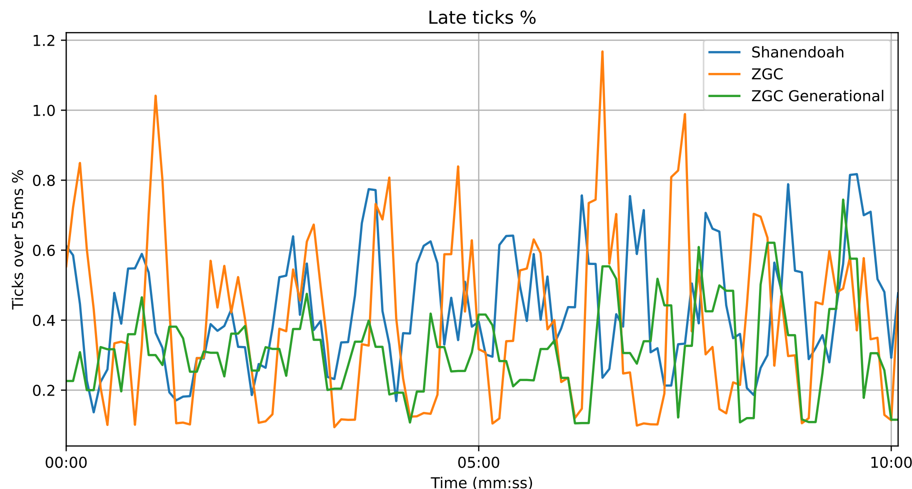


Figure 13: Late ticks (over 55 ms) comparison of different GCs, G1GC excluded.

6.3.3 Evaluation of better tick-scheduling strategy

The scheduler of a real-time system is the heart of operations. Its strategy is solely responsible for the system's success or failure since task execution depends on it. In most cases, such as the one being analyzed in this thesis, minimizing missed deadlines, or in this case late ticks, should be the key discriminator.

The current system uses a one-thread-per-battle system where each battle is encapsulated in a thread and executed independently. The thread will stay in an idle state until its execution time, after which it will run the tick logic code and go back to the idle state. Because the JVM has no control over threads, all scheduling is delegated to the system scheduler, which also handles other processes. Despite its apparently primitive architecture, this system works fairly well, especially if the Java application is one of the only processes in the whole VM. This approach also eliminates an extra scheduling layer, which might introduce a slight overhead, but loses all control over the threads once they are started.

A new and possibly improved system could use Java's thread pool implementation, which works by assigning a fixed number of threads and then executing tasks on those threads as soon as they become available. This system would greatly reduce the number of threads while introducing a slight overhead when scheduling tasks, although probably acceptable. This system would also allow a more fair scheduling strategy because all tasks will eventually get executed more or less around their designation mark unless the whole system is heavily overloaded. There is no such guarantee in the original system since the scheduling is completely out of the application's control.

The new scheduling strategy featuring a thread pool shows a noticeably lower percentage of ticks over 70 ms and over 100 ms, as shown in Figure 14. However, the percentage of late ticks is still extremely small in both cases, therefore, although noticeable from the graphs, the difference is probably irrelevant in itself.

Another major improvement is the distribution of tick drift. While the older scheduling strategy showed a tick drift of 650 μs for the 90th percentile and 1 ms for the 99th percentile, the newer scheduling strategy vastly improves them by recording a tick drift of 43 μs for the 90th percentile and 65 μs for the 99th percentile. This improvement was partially expected due to the nature of the scheduling system itself. While the older method used the Java *sleep* function, which has a default precision of 1ms and therefore can delay ticks by up to 1 ms as shown in the 99th percentile, the newer method is much more precise.

Unfortunately, this major improvement in scheduling comes at a cost of CPU usage, as shown in Figure 15. As previously hypothesized, introducing a second scheduler on

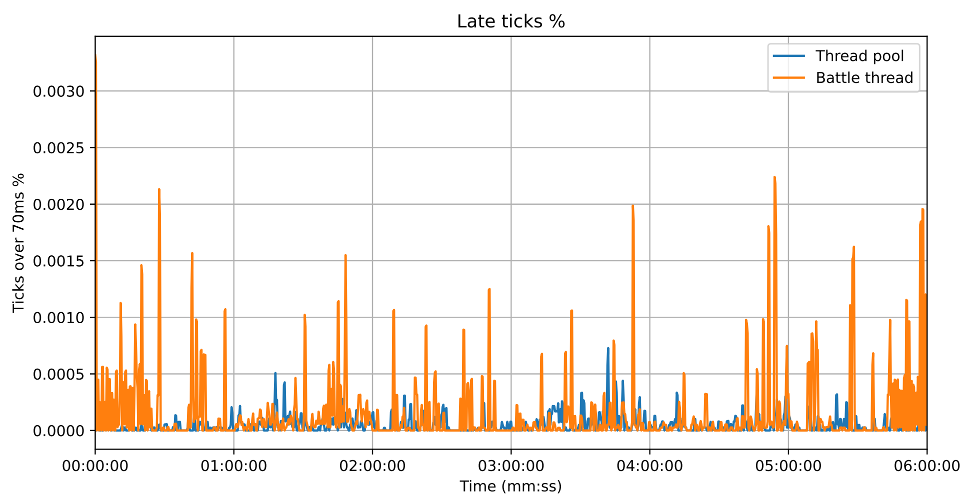


Figure 14: Late ticks comparison of different scheduling strategies

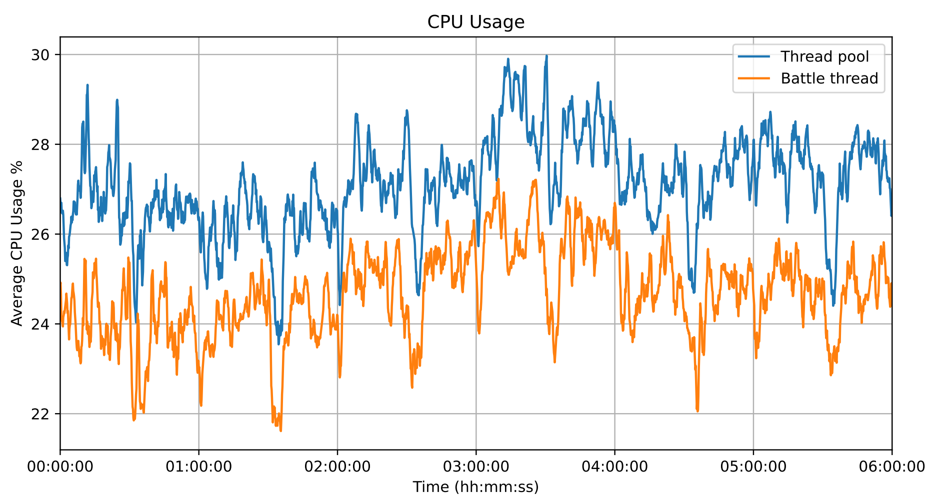


Figure 15: Late ticks comparison of different scheduling strategies

top of the one offered by the operating system introduced a non-negligible overhead, although probably acceptable. Because in a real-time system, achieving low latency is more important than obtaining a low CPU usage, this new scheduling technique is probably acceptable.

In conclusion, this new scheduling strategy, although heavier on the CPU, does decrease missed deadlines and schedules tasks more precisely, therefore it should be considered an improvement.

7 Discussion

7.1 Research objectives

The findings of this thesis highlight the significant performance bottlenecks that arise in Java-based real-time systems, particularly in latency-sensitive environments such as game servers. One of the most critical challenges identified is the Java Virtual Machine's unpredictable nature due to its reliance on garbage collection and just-in-time compilation. These features, while beneficial for general-purpose applications, introduce variability in execution times, making it difficult to maintain the stringent timing requirements necessary for real-time systems. The research shows that under high load conditions, these JVM activities can lead to substantial delays, particularly during garbage collection cycles and compilation-intensive workloads, where the system may experience unpredictable pauses that negatively impact latency and overall system responsiveness.

Evaluating the impact of JVM features on predictability

The analysis conducted in this thesis underlines the complex relationship between JVM-specific features and the predictability and latency of Java-based real-time systems. The study reveals that while JIT compilation can improve execution speed over time, it also introduces inconsistencies in performance, especially when some code paths show difficult-to-predict behaviors. These inconsistencies are problematic in real-time systems where predictable timing is crucial. Similarly, the garbage collector's operation, though essential for memory management, often results in unpredictable pauses that are detrimental to maintaining consistent low-latency performance. The evaluation demonstrates that these JVM features, while enhancing general application performance, are significant obstacles to achieving the level of predictability required for real-time systems.

Exploring optimization strategies for mitigating Java-induced issues

In response to the challenges posed by the JVM, this thesis explores several optimization strategies to mitigate the negative impacts on real-time performance. Techniques such as garbage collection tuning, altering JVM configurations, and employing alternative JVMs designed for low-latency applications, such as Zing, were tested to determine their effectiveness in reducing latency and improving predictability. The results indicate that, while these optimizations do not entirely eliminate the unpredictability inherent in the JVM, they can lead to some serious improvements. For example, fine-tuning garbage collection can reduce the frequency and duration of pauses, although it cannot fully guarantee eliminating those. Similarly, JIT compilation settings can be adjusted to prioritize early and rapid compilation, although this might come at the cost of higher processor utilization. Therefore, while the optimizations provide some relief, they highlight the trade-offs involved in balancing performance and predictability in Java-based real-time systems.

Assessing the effectiveness of applied solutions

The effectiveness of the applied optimization strategies is an overall success, with some methods showing more promise than others in enhancing the overall system performance. Although some changes proved extremely effective in the analyzed scenarios, others showed mediocre results. Most importantly, the effectiveness of these optimizations varied depending on the specific workload and system conditions, suggesting that there is no one-size-fits-all solution.

The research concludes that while achieving the ideal balance of performance and predictability remains challenging in Java-based real-time systems and can require a huge testing effort, significant progress can be made through targeted optimizations.

7.2 Comparison with existing literature

7.2.1 Java warming performance issues

Section 6.1 analyzes the performance problems that can arise during the warm-up phase of the JVM and offers possible solutions on how to fix them. Despite speeding up the JIT compiler by allocating more compute resources to it or switching the JVM entirely to produce non-speculative machine code showed some benefits, the most successful strategy was by far refactoring methods to be easier to speculate and optimize. The problematic warm-up phase of Java is well-known in the literature, especially in environments where high efficiency is required, such as large-scale commercial deployments. The authors of [59] describe similar issues during warm-up, such as poor performance when processing hundreds of thousands of real-time operations per second. Similarly to this thesis, data was collected by profiling and byte analysis. In this case, the issue was solved by submitting the newly created JVMs to an artificial load that could stress the hot parts of the application enough to be fully optimized by the JIT compiler. The authors of [37] performed a similar study, although not necessarily Java-oriented. Their solution was the creation of profiles with previously obtained data that the virtual machine could leverage to perform better optimizations during runtime. Despite the vastly different solutions, all approaches aim to gather large amounts of data to predict the application's behavior.

7.2.2 Java version evaluation

Section 6.2.2 evaluates the performance differences between distinct Java versions, particularly Java 11 and Java 21, on HotSpot-based JVMs. From the results, it is clear that the newer version of Java performs significantly better than the older one, especially in tasks such as JIT compilation and garbage collection. Although various improvements in these JVM components can explain this behavior, the gap between versions is too big to fully justify it. Another possible theory is the fact that Java 21 might have a smaller footprint and use resources more conservatively than its older counterpart. Comparing Java versions is not a common practice, especially because upgrades are usually performed to grant compatibility and utilize new features, therefore any performance benefit or regression is usually ignored. Despite this, it

is also possible to see major differences between Java versions in the literature in some cases. For instance, the authors of [41] compare the behavior of different GC algorithms across Java versions 8, 11, and 12. The results are perfectly compatible with the ones observed in this thesis, where newer versions tend to perform better than older ones. This behavior is not surprising, given the fact that highly-utilized algorithms are constantly refined and optimized by the Java community, therefore it is only natural that newer versions are more performant.

7.2.3 Cloud instances evaluation

Section 6.2.3 evaluates newer and faster instances available on the public cloud to determine which performs better per price. The test data showed that ARM-based instances performed significantly better than their x86 counterparts, especially considering the lower hourly price. In general, ARM instances are cheaper due to their lower energy consumption while executing similar workloads, therefore the operating costs of cloud providers are greatly reduced as a consequence [53]. Several studies have been published on the performance of ARM CPUs over the years, including [24]. In this paper, the power of ARM processors is greatly emphasized, especially compared to their cost. The authors of the paper estimated a 37% cost reduction when utilizing an ARM platform for a web application compared to the more traditional x86.

7.2.4 Linux kernel mitigations evaluation

Section 6.2.4 evaluates the effects of software mitigations on performance. In newer Linux kernels, several software mitigations prevent attackers from gaining access to protected resources because of unresolved hardware vulnerabilities of the host machine. Because of their pure software nature, mitigations might have a noticeable performance impact on applications, although not all workloads are affected the same way. In this thesis, a noticeable performance impact on CPU usage and late ticks was clearly visible from the data, although not significant enough to renounce the security benefits provided by these mitigations. However, the performance drop can be much more impactful in some other cases. This is described by the authors of [7] observed up to 20% worse performance on some workloads, while the authors of [10] declared performance drops that can range from 2% up to 100% in certain scenarios. Although the tests performed in this thesis did not necessarily justify disabling all kernel mitigations, different workloads might find this much more desirable given the cost penalty derived from them.

7.2.5 Garbage collection evaluation

Section 6.3.2 compares and evaluates different GC algorithms and their performance on the game server. The testing concluded that ZGC without *Generational* mode performed the best, mostly thanks to its short pauses and smaller overhead compared to other GCs. The authors of [50] completely confirm the collected data in many ways. In their study, it is confirmed that ZGC has the lowest pauses, with Shenandoah closely

behind. ZGC also showed the best performance on latency-sensitive workloads, and while Shenandoah was again closely behind, its significantly higher overhead made it less desirable for this kind of application. The paper finally states that ZGC is by far the best GC for low-latency applications, and the same conclusion was reached in this thesis.

7.3 Limitations of the work

While this research provides insights into improving the performance of Java-based real-time systems, it is important to acknowledge its limitations.

First, the study focuses primarily on a specific use case, which, while representative, may not capture the full range of challenges faced by other types of real-time systems. The findings and optimizations discussed may not directly apply to systems with different workloads or real-time requirements.

Second, the optimization strategies were evaluated under controlled conditions that may not fully reflect the variability of real-world environments. This controlled setting limits the generalizability of the results, particularly in highly dynamic or resource-constrained environments. For instance, testing with real players might be needed to fully assess the validity of the proposed changes.

These limitations suggest that further research is necessary to explore additional strategies, particularly those that involve deeper modifications to the JVM or alternative approaches outside the traditional Java ecosystem.

8 Conclusions

This thesis explored the challenges of optimizing performance in Java-based real-time systems, focusing on latency-sensitive environments such as game servers. The primary aim was to identify key performance bottlenecks, particularly those related to the Java Virtual Machine, and propose and evaluate strategies to mitigate these issues. By doing so, the research improved system predictability and reduced latency, ensuring smoother and more reliable real-time performance.

The methodology employed in this study was an iterative exploratory data analysis and problem solving, which involved an in-depth analysis of the JVM's impact on real-time performance using a combination of profiling tools and controlled load-testing scenarios. This allowed for the identification of significant sources of latency, such as garbage collection and just-in-time compilation. Following this analysis, a series of optimization strategies were implemented and tested, including garbage collection tuning, JVM configuration adjustments, and the use of alternative JVMs designed for low-latency applications. These strategies were evaluated for their effectiveness in improving system performance under real-time conditions.

The research found that while Java offers many benefits for real-time systems, such as portability and safety, its inherent features can introduce significant unpredictability. The main performance bottlenecks were identified as the unpredictability introduced by JVM activities, particularly during JIT compilation. The optimization strategies tested provided some improvements, particularly in reducing latency and enhancing predictability, but they also revealed trade-offs, such as increased resource usage and complexity. Overall, although not perfect, the proposed solutions drastically improved some of the key metrics that could enhance the player experience.

To summarize, the findings were as follows:

- Newer versions of Java feature significant performance improvements compared to older ones.
- By understanding the inner working of the JIT compiler and making the codebase easier to speculate, it is possible to drastically improve performance and player experience.
- Moving to newer and more efficient server instances, such as from AWS *c5.2xlarge* to a *c7g.2xlarge*, can lead to improved performance in terms of the number of battles and players at a lower cost.
- Selecting the right GC can lead to significant performance improvements, such as a consistently reduced latency in real-time applications like in this case.

The findings of this thesis also highlight the responsibility of developers when engineering real-time applications. While Java remains a strong candidate for many applications due to its robust ecosystem and cross-platform capabilities, developers must be aware of the potential performance issues introduced by the JVM. This

research suggests that specific wariness shall be applied in scenarios where real-time performance is critical, followed by in-depth testing of the proposed solutions.

Looking forward, future work could focus on further refining JVM configurations or testing new JVM variants that are specifically optimized for real-time applications. Additionally, expanding the scope of research to include different types of real-time systems beyond game servers could provide more generalized insights. Another promising area for future research would be exploring hybrid approaches that combine the Java language with different runtime environments that might offer better predictability, therefore leveraging the strengths of Java while addressing its limitations in real-time contexts.

References

[1] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6: 355–366, 2003.

[2] Amazon EC2 instance type naming conventions, 2024. URL https://docs.aws.amazon.com/ec2/latest/instancetypes/instance-type-names.html.

[3] Eike Falk Anderson. A classification of scripting systems for entertainment and serious computer games. In *2011 Third International Conference on Games and Virtual Worlds for Serious Applications*, pages 47–54, 2011. doi: 10.1109/VS-GAMES.2011.13.

[4] AWS Nitro, 2024. URL https://aws.amazon.com/ec2/nitro/. Lightweight Hypervisor - AWS Nitro System.

[5] Azul Zing, 2024. URL https://www.azul.com/products/prime/. High Performance JVM for Superior Java | Azul Platform Prime.

[6] David F. Bacon, Perry Cheng, David Grove, Michael Hind, V. T. Rajan, Eran Yahav, Matthias Hauswirth, Christoph M. Kirsch, Daniel Spoonhower, and Martin T. Vechev. High-level real-time programming in java. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, page 68–78, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930914. doi: 10.1145/1086228.1086242. URL https://doi.org/10.1145/1086228.1086242.

[7] Jonathan Behrens, Adam Belay, and M. Frans Kaashoek. Performance evolution of mitigating transient execution attacks. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 251–265, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519559. URL https://doi.org/10.1145/3492321.3519559.

[8] Yahn W Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, volume 98033, 2001.

[9] Shahid H Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE transactions on Software Engineering*, pages 583–589, 1981.

[10] Lucy Bowen and Chris Lupo. The performance cost of software-based security mitigations. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 210–217, New York, NY, USA, 2020.

Association for Computing Machinery. ISBN 9781450369916. doi: 10.1145/3358960.3379139. URL https://doi.org/10.1145/3358960.3379139.

[11] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. A performance study of java garbage collectors on multicore architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, page 20–29, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334044. doi: 10.1145/2712386.2712404. URL https://doi.org/10.1145/2712386.2712404.

[12] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), oct 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978814. URL https://doi.org/10.1145/1978802.1978814.

[13] Ethan Davison. Valorant's super-fast servers are attracting streamers and pros in droves. here's why. *The Washington Post*, 2020-04-14. URL https://www.washingtonpost.com/video-games/esports/2020/04/14/valorant-tick-rate-servers-pros-streamers/.

[14] R. Fielding, M. Nottingham, and J. Reschke. Http semantics. Standard, International Organization for Standardization, June 2022. URL https://www.rfc-editor.org/rfc/rfc9110.html.

[15] Borko Furht, Dan Grostick, David Gluch, Guy Rabbat, John Parker, and Meg McRoberts. *Introduction to Real-Time Computing*, pages 1–35. Springer US, Boston, MA, 1991. ISBN 978-1-4615-3978-0. doi: 10.1007/978-1-4615-3978-0_1. URL https://doi.org/10.1007/978-1-4615-3978-0_1.

[16] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726–740, 2014. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2014.07.004. URL https://www.sciencedirect.com/science/article/pii/S1383762114001015.

[17] Julien Gascon-Samson, Jörg Kienzle, and Bettina Kemme. Dynfilter: Limiting bandwidth of online games using adaptive pub/sub message filtering. In *2015 International Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6, 2015. doi: 10.1109/NetGames.2015.7382988.

[18] Josh Glazer and Sanjay Madhav. *Multiplayer game programming: Architecting networked games*. Addison-Wesley Professional, 2015.

[19] Grafana, 2024. URL https://grafana.com/. Grafana: The open observability platform.

[20] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[21] Johan Hagelback and Stefan J. Johansson. Dealing with fog of war in a real time strategy game environment. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 55–62, 2008. doi: 10.1109/CIG.2008.5035621.

[22] Gerard J Holzmann. The power of ten–rules for developing safety critical code. *Software Technology*, 10, 2018.

[23] Java Compiler Directives, 2024. URL https://docs.oracle.com/en/java/javase/22/vm/writing-directives.html. Java Virtual Machine Guide - Writing Directives.

[24] Qingye Jiang, Young Choon Lee, and Albert Y. Zomaya. The power of arm64 in public clouds. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 459–468, 2020. doi: 10.1109/CCGrid49817.2020.00-47.

[25] Ramesh Johari, Pete Koomen, Leonid Pekelis, and David Walsh. Peeking at a/b tests: Why it matters, and what to do about it. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1517–1525, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348874. doi: 10.1145/3097983.3097992. URL https://doi.org/10.1145/3097983.3097992.

[26] Nicolai M Josuttis. *The C++ standard library: a tutorial and reference*, chapter 5.2. Addison-Wesley Professional, 2012.

[27] Abdul Malik Khan, Ivica Arsov, Marius Preda, Sophie Chabridon, and Antoine Beugnard. Adaptable client-server architecture for mobile multiplayer games. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, Brussels, BEL, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 9789639799875. doi: 10.4108/ICST.SIMUTOOLS2010.8704. URL https://doi.org/10.4108/ICST.SIMUTOOLS2010.8704.

[28] Hermann Kopetz and Wilfried Steiner. *Real-time systems: design principles for distributed embedded applications*. Springer Nature, 2022.

[29] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, jan 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL https://doi.org/10.1145/321738.321743.

[30] Y Liu, Jing Wang, Michael Kwok, Jeff Diamond, and Michel Toulouse. Fps game performance in wi-fi networks. In *4th International Game Design and Technology Workshop and Conference (GDTW2006)*, pages 41–49. Citeseer, 2006.

[31] Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, page 103–104, dec 2014. doi: 10.1145/2692956.2663188. URL https://doi-org.libproxy.aalto.fi/10.1145/2692956.2663188.

[32] Ian Millington. *Game Physics Engine Development*. Taylor & Francis, 2007.

[33] Patrick Niemeyer and Jonathan Knudsen. *Learning java*. " O'Reilly Media, Inc.", 2005.

[34] G.P Nikishkov, Yu.G Nikishkov, and V.V Savchenko. Comparison of c and java performance in finite element computations. *Computers & Structures*, 81(24): 2401–2408, 2003. ISSN 0045-7949. doi: https://doi.org/10.1016/S0045-7949(03)00301-8. URL https://www.sciencedirect.com/science/article/pii/S0045794903003018.

[35] Kelvin D. Nilsen. Issues in the design and implementation of real-time java, 1996. URL https://www.cs.cornell.edu/courses/cs614/1999sp/papers/rtji.pdf.

[36] Scott Oaks. *Java Performance*. O'Reilly Media, Inc., 2020. ISBN 9781492056119.

[37] Guilherme Ottoni and Bin Liu. Hhvm jump-start: Boosting both warmup and steady-state performance at scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 340–350, 2021. doi: 10.1109/CGO51591.2021.9370314.

[38] Prometheus, 2024. URL https://prometheus.io/. Prometheus - Monitoring system and time series database.

[39] Prometheus, 2024. URL https://prometheus.io/docs/concepts/metric_types/. Prometheus - Metric types.

[40] Prometheus, 2024. URL https://prometheus.io/docs/prometheus/latest/querying/basics/. Prometheus - Querying basics.

[41] P. Pufek, H. Grgić, and B. Mihaljević. Analysis of garbage collection algorithms and memory management in java. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1677–1682, 2019. doi: 10.23919/MIPRO.2019.8756844.

[42] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture*, pages 291–302, 2005. doi: 10.1109/HPCA.2005.29.

[43] Shinu M. Rajagopal, Supriya M., and Rajkumar Buyya. Fedsdm: Federated learning based smart decision making module for ecg data in iot integrated edge–fog–cloud computing environments. *Internet of Things*, 22:100784, 2023.

ISSN 2542-6605. doi: https://doi.org/10.1016/j.iot.2023.100784. URL https://www.sciencedirect.com/science/article/pii/S2542660523001075.

[44] Chaitanya K. Rudrabhatla. Comparison of zero downtime based deployment techniques in public cloud infrastructure. In *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 1082–1086, 2020. doi: 10.1109/I-SMAC49090.2020.9243605.

[45] Lui Sha and John B Goodenough. Real-time scheduling theory and ada. *IEEE Computer*, 23(4):53–62, 1990.

[46] Kang G Shin and Parameswaran Ramanathan. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82 (1):6–24, 1994.

[47] Source Multiplayer Networking, 2005. URL https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. Source Multiplayer Networking.

[48] Benjamin Sowell, Alan Demers, Johannes Gehrke, Nitin Gupta, Haoyuan Li, and Walker White. From declarative languages to declarative processing in computer games. *arXiv preprint arXiv:0909.1770*, 2009.

[49] Maddie Stone. The more you know, the more you know you don't know - a year in review of 0-days used in-the-wild in 2021. *Project Zero*, 04 2021. URL https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.

[50] Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira. Selecting a gc for java applications. *Norsk Informatikkonferanse (NIK)*, 1:1–14, 2021.

[51] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, page 79–88, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450302630. doi: 10.1145/1993478.1993491. URL https://dl.acm.org/doi/10.1145/2076022.199349.

[52] The Java HotSpot Performance Engine Architecture, 2024. URL https://www.oracle.com/java/technologies/whitepaper.html. The Java HotSpot Performance Engine Architecture.

[53] Bogdan Marius Tudor and Yong Meng Teo. On understanding the energy consumption of arm-based multicore servers. *SIGMETRICS Perform. Eval. Rev.*, 41(1):267–278, jun 2013. ISSN 0163-5999. doi: 10.1145/2494232.2465553. URL https://doi.org/10.1145/2494232.2465553.

[54] Unreal Engine - Introduction to Iris, 2024. URL https://dev.epicgames.com/documentation/en-us/unreal-engine/introduction-to-iris-in-unreal-engine#netobject.

[55] Usability Engineering, 1993. URL https://www.nngroup.com/articles/response-times-3-important-limits/. Response Times: The 3 Important Limits.

[56] Chaitya Vohera, Heet Chheda, Dhruveel Chouhan, Ayush Desai, and Vijal Jain. Game engine architecture and comparative study of different game engines. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6, 2021. doi: 10.1109/ICCCNT51525.2021.9579618.

[57] Shaoxuan Wang and Sujit Dey. Modeling and characterizing user experience in a cloud server based mobile gaming approach. In *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*, pages 1–7, 2009. doi: 10.1109/GLOCOM.2009.5425784.

[58] Candid Wueest. The continued rise of ddos attacks. *White Paper: Security Response, Symantec Corporation*, 2014.

[59] Fangxi Yin, Denghui Dong, Sanhong Li, Jianmei Guo, and Kingsum Chow. Java performance troubleshooting and optimization at alibaba. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 11–12, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356596. doi: 10.1145/3183519.3183536. URL https://doi.org/10.1145/3183519.3183536.

[60] Marcelo P. M. Zamith, Esteban W. G. Clua, Aura Conci, Anselmo Montenegro, Regina C. P. Leal-Toledo, Paulo A. Pagliosa, Luis Valente, and Bruno Feij. A game loop architecture for the gpu used as a math coprocessor in real-time applications. *Comput. Entertain.*, 6(3), nov 2008. doi: 10.1145/1394021.1394035. URL https://doi.org/10.1145/1394021.1394035.