Vrije Universiteit Amsterdam

Bachelor Thesis

Benchmarking the performance impact of mods on Minecraft-like games

Author: Guivari Dzar Amri (2723271)

1st supervisor: Jesse Donkervliet
2nd reader: Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 23, 2024

Abstract

Community created mods are important in extending the usability of an MVE, and the Minecraft modding community shows the huge size of people using and creating mods (60.9 Billion mod downloads for Minecraft in curseforge.com). The lack of research into Minecraft modding, and the disorganization of the modding community makes it a challenge for people to evaluate the performance impact of mods. By finding performance impact, people may use the data to improve their mods, and better conduct empirical research into mods. In this work, we create a benchmark for Minecraft mods by designing, implementing and conducting experiment using the created benchmark. We found that mods that introduce new entities affect performance the most. We found that mods produce lower performance impact in parallel as compared to individually, with a decrease of 0.28%. Lastly, we find that mods that introduce a new dimension have a noticable higher performance impact than mods that introduce single-dimension features.

Contents

1 Introduction **1**

1.1 Problem statement 1

1.2 Research Questions 2

1.3 Research Methodology 3

1.4 Thesis structure 3

2 Background **5**

2.1 Minecraft as an MVE 5

2.2 Minecraft Modding and Mod Loaders 7

2.3 Curse Forge Platform 9

3 Design **11**

3.1 Requirements 11

3.2 Design Overview 13

3.3 Benchmarking Process 13

3.3.1 Pre-configuration 13

3.3.2 Benchmarking 15

3.3.3 Post-processing 15

3.4 Performance Metrics 15

3.4.1 Tick Duration 16

3.4.2 Lag Spikes 16

3.5 System Under Test 17

3.5.1 Modded Minecraft Version 17

3.5.2 Mods for Benchmarking 17

3.6 Workload 18

3.6.1 Player Simulation 18

3.6.2 Server and World configuration 18

4 Implementation **21**
4.1 Overview 21
4.2 Forge Mod Loader 22
4.3 Implementing Player Emulation 22
4.4 Collecting Performance Metrics 23

5 Experiment Results **25**
5.1 Experiment Setup 25
5.2 Main Findings 28
5.2.1 MF1: Entity-related processes have more performance impact than managing chunks 29
5.2.2 MF2: Performance impact higher in mod-implemented dimension compared to base Vanilla dimension when considering average performance 30
5.2.3 MF2: Increase in MSPT higher in sum of mods run individually compared to mods run in parallel 31
5.2.4 Spark profiler overhead 32
5.3 Limitations and/or threat to validity 33

6 Related Work **35**

7 Conclusion **37**

References **39**

A Appendix **41**

1

Introduction

Online Modifiable Virtual Environments are a popular modern way for individuals to connect with each other. They provide a simulated space people may use to interact with each other, regardless of distance. These spaces can be used for socialization, education, and general recreational use. The ability to modify the virtual space can give users a sense of belonging to the MVE, creating a bond and possibly fostering a community. This is especially useful in situations where people cannot meet each other in-person, such as the case during the Covid-19 pandemic where a popular MVE, Minecraft, assisted people to socialize.(1). In 2023, Minecraft reached 300 million sales (2). It is one of the most popular MVEs, and has also facilitated education (3).

Modding is an important activity in MVEs. Minecraft allows the use of modification (mods), which are extensions to the system that can add, optimize, and modify features. Modding in Minecraft is popular, and there is a massive community that uses Minecraft mods. Mods have been used for education (4), health (5) One of the most popular Minecraft mod distribution websites, Curse Forge, lists over 179.4 thousand mods, and 60.8 billion mod downloads (6). This does not include mods that have been de-listed, nor private user-created mods.

1.1 Problem statement

Although popular, the impact of mods on system performance has not yet been thoroughly investigated. Though regular mod creators and mod users might have an intuitive understanding on how mods affect performance, there is little empirical research on how mods affect MVE performance. Moreover, without official recognition from Mojang, there is no definitive framework for modding. There are numerous different standards and conventions, and a large number of created mods are one-off passion projects.

Research into the performance impact of mods in Minecraft may provide insight into the common performance effects user-created mods have, and possibly assist in finding common areas of weakness in user created mods that cause a significant performance impact. A benchmark would provide the tools to facilitate this type of research.

This thesis proposes the design and implementation of a benchmark for mods in Minecraft-like applications. The designed benchmark should evaluate performance impact of user-created mods within Minecraft.

1.2 Research Questions

RQ1. **How to design a benchmark for comparing Modded MVE performance between different mods?**

Mods can vary very differently in regards to features, implementation and intended use. With this concern in mind, how can we fairly compare mods? Similar mods may provide identical functionalities, but require the player to do different actions. This concern impacts the design of the benchmark workload. The designed benchmark should eliminate as much overhead as possible, outside of use of the mod. This applies to the design of player simulation, experiment procedure, and the system used to run the mods.

RQ2. **How to implement such a benchmark?** To fairly compare the performance impact of mods in an MVE, the implementation following the design addressing **RQ1** should be implemented. However, with little research into Minecraft mods, there is not a readily available set of tools to use to create a fair benchmark. There exists tools to facilitate player simulation, and tools to collect Minecraft system metrics, but they are rarely used with mods in mind. Another concern is the implementation of a fair player simulation workload. At which point does the implemented workload have enough features that covers a "sufficient" number of mod features? The design will have to take into consideration a tradeoff between the ability to use all mod features, and the fairness of the workload given to each mod being benchmarked.

RQ3. **How to use the benchmark to compare the performance of mods on MVEs?** Comparisons between performance impact would need collected metrics from experiments using the benchmark implented to address **RQ3**. Such metrics should give insight onto the severity of the performance impact. Additionally a profiler should be used on the MVE to obtain information about system processes, giving further insight onto the areas of interest in the MVE which impacts performance. For studying the performance impact of a single mod, it would be best to compare it with a base version of the MVE with the same workload. The design should allow such comparisons with an unmodded version of Minecraft.

1.3 Research Methodology

To answer the research questions, we will design and implement a suitable benchmark, which will then be used to compare the performance impact of mods in the Minecraft MVE.

Addressing **RQ1**, the benchmark will be designed for use on a modded version of Minecraft, supporting a popular mod loader, the Forge Modloader. Modloaders are explained in detail in Section 2.2. The benchmark will support automated testing environments, increasing consistency while conducting experiments.

Addressing **RQ2**, we will make use of automation and workload emulation tools available for Minecraft. These tools will be modified to support a modded Minecraft system.

Addressing **RQ3**, experiments using the implemented benchmark will be conducted to compare the performance impact between Minecraft mods within a mod category. A metric collector and profiler will be used to obtain relevant metrics.

1.4 Thesis structure

The background section will provide information on how the Minecraft system operates, the common modding methods and problems regarding Minecraft modding, and an explanation of Mod Loaders which is an essential part of using and creating mods. It also provides a brief background of the Curse Forge platform which will dictate how this thesis categorizes mods and chooses mods for experimentation.

The design section will explain the thought process behind the decisions made in creating the benchmark, sections of benchmarking relevant to Minecraft modding, the reasoning behind choosing relevant metrics, and the intended way to use the benchmark.

The implementation section will explain the tools used to implement the benchmark, what configurations were conducted to facilitate mods, and the profilers concerned with collecting metrics.

The evaluation section will show the benchmark in use. Experiments regarding evaluating the performance impact of world-generation related mods are conducted. And the main findings from said experiments are explained.

Lastly in the conclusion section, the lessons learned from creating the benchmark are explained. It also discusses the shortcomings of the benchmark, and possible future works related to the thesis.

2

Background

This section describes Minecraft as an MVE, Mod Loaders, the Forge Mod Loader, and the Curse Forge Platform.

2.1 Minecraft as an MVE

Minecraft allows users to interact with its environment consisting of blocks. Similar to most MVEs, Minecraft uses a client-server architecture. The client handles presentation of the game to the user such as rendering, particles, and GUIs. The server handles the main game system: game events, calculations, registering, and file handling. In the case of a singleplayer session, a local server is used. Mods are implemented using Mod Loaders (MLs), which injects mod jar files into a Minecraft class launcer, explained in Section 2.2.

The Minecraft game loop is referred to as a **tick**. Figure 2.1 depicts how the Minecraft tick operates. Under optimal conditions, the maximum length of a game tick is 50ms, which results in a constant 20 ticks per second, resulting in a smoother gameplay experience. This period of 50ms is referred to as the **tick interval**.

Within a tick, all server tasks such as game computations, file handling, game events, and communication with players, is handled. The time taken to process these tasks are referred to as the **tick duration**. As shown in 2.1, when the tick duration is shorter than the tick interval, Minecraft makes the server wait for the full tick interval. This interval is referred to as the **tick wait duration**. In some cases, scheduled tasks are done specifically under this wait time.

Sometimes, the server is overloaded with tasks and takes more than 50ms to complete its tasks. This is when the tick duration is longer than the tick interval. In this case, the optimal 20 ticks per second is not met. This term is referred to as a **server lag**. Under

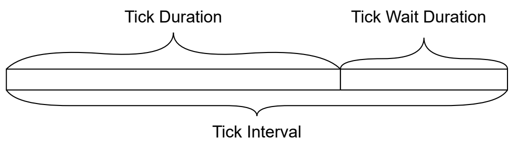


Figure 2.1: The Minecraft Game Loop.

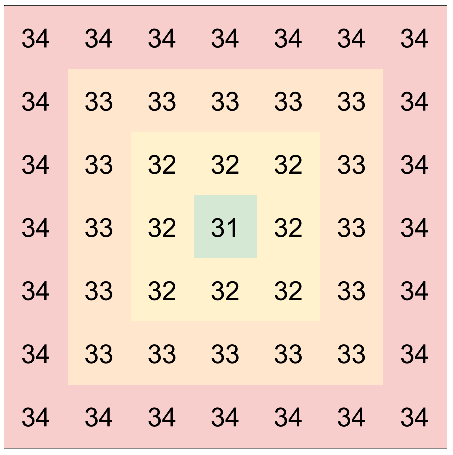


Figure 2.2: Example of load levels respective to a player in the center.

server lag, the game freezes and waits for the tick to process. No matter how long the tick takes, Minecraft makes no attempt to abort and re-process the tick. Therefore bugs in the system may fully cause the game to freeze for an infinite amount of time and cause the server to time out.

In a less extreme case, the server is simply taking a longer time than usual to process a tick. Very small server lag may not be noticeable to the player. But under cases where the server struggles to keep up with the workload, a **lag spike** may occur where for a period of time the server takes a significantly longer time to process a tick. Lag spikes are an important application level metric as players do not enjoy playing a game which does not run at the optimal rate. Lag spikes will be one of the metrics considered during benchmark design in section 3.4.

Chunks are how Minecraft handles processing. Minecraft sections its processes per-chunk. A chunk is a 16-by-16 area of the game. Within each chunk events may trigger which sends "tickets" to the server, which it has to process. The amount of tickets a chunk can send to a server is restricted by its load level. Figure 2.2 depicts an example of chunk load levels respective to a player in the center. Understanding how Minecraft prioritizes chunks are important in this thesis, as it affects the workload a server has to process.

Each chunk has a **load level**, which is inverse to the priority a chunk is given. The more important a chunk is, the lower the load level. Chunks with a lower load level, levels 31 and lower, have all their game aspects available. When a chunk has a lower priority, such as when there is no player nearby, it is assigned a higher load level up to a point where no game aspects are available. This is the case for chunks with a load level of 34 and higher.

As depicted in 2.2, load levels typically increase the further away a chunk is from a player. The lower the load level, the more priority a chunk has, and the more ability for it to give large workloads on the server. Therefore, performance impact is affected by how the system processes chunks, and chunk levels.

2.2 Minecraft Modding and Mod Loaders

Minecraft has 2 editions - Java and Bedrock, written in Java and C++ respectively. While the Bedrock edition supports more platforms (7), there is a lack of modding APIs and support for Bedrock. Most mods are developed in Java, where even the developer of Minecraft has given notes on how the Java edition works (8)].

Mods are usually in the forms of Java jars. These jars contain classes to be injected into a Minecraft instance. Mods may introduce, add and modify new Java classes. Running mods in parallel may have compatibility issues. Mods may overwrite each other's classes or just fully delete a class that another mod may rely on. If Mod developers do not communicate upon a set of systems and rules, mods will not run in parallel. This problem is addressed with Mod Loaders. Mod Loaders are further explained in the next section.

Mod Loaders (MLs) are modding frameworks that assists with the managing and loading of mods. They are usually similarly structured to the MVE they're based on, but with the addition of a standardized system that assists with mod compatibility and dependencies - introducing standardization previously mentioned missing. When loading mods for Minecraft, they ensure mod dependencies are met and that the mod is compatible with the Minecraft version. From a non-developer standpoint MLs also allow easy mod management mainly: updating mod versions, turning mods on and off, solving dependency errors.

Essentially Mod Loaders follow the following steps:

1. Refers to a base Minecraft .jar for base classes.
2. Discovers .jar files in the mod folder.
3. Manages dependency injection of mod .jars, in reference to the base Minecraft .jar.
4. Managed jars are loaded in the ML's custom class loader for implementation during loading of the modded Minecraft system.

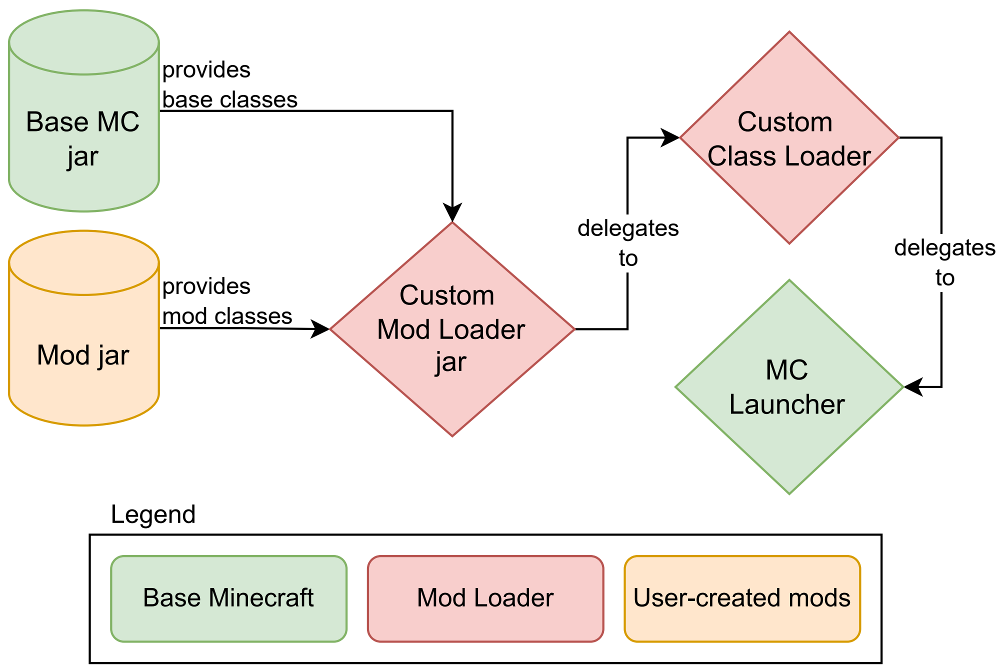


Figure 2.3: Simplified system model of a modded MVE.

Once this is done the new class objects are available to the Minecraft game environment.

There are multiple Mod Loaders available for Minecraft. That number may increase, it is common for developers to attempt to develop a mod in a way that existing Mod Loader systems do not support, and they end up making their own Mod Loader to facilitate their needs. As mods were never officially recognized, the community surrounging mods lack a standard or framework. Developers often clash when implementing Mod Loaders, to the point where there is an oversaturation of frameworks available for Minecraft modding (9). The ever increasing amount of Mod Loaders means that casual players may have to constantly switch between systems, and that some mods may never be run simultaneously as they were developed in different Mod Loaders.

Modding is further simplified with community-created mod libraries. They are mods that act as an API to assist other mods, and do not add functionality by themselves. If a mod developer used a mod library, they would have to rely on the library to support the Minecraft version they want to develop their mod in.

Mod	**Downloads**	**Mod Category**
Just Enough Items	340.3M	Map & Information, API & Library
Mouse Tweaks	253.6M	Utility & QoL, Storage, Miscellaneous
Journey Map	233.8M	Map & Information
Bookshelf	214.3M	Server Utility, API & Library, Miscellaneous
Appleskin	214.3M	Food, Map & Information
Controlling	213.0M	Utility & QoL, Cosmetic
AutoRegLib	186.1M	API & Library
Clumps	184.8M	Utility & QoL, Storage, Server Utility
Mantle	179.4M	API & Library
Storage Drawers	170.5M	Storage, Cosmetic

Table 2.1: A list of the top Minecraft Versions listed in Curse Forge.

2.3 Curse Forge Platform

Curse Forge is a mod distribution platform providing an extensive collection of mods for games. As of writing, Curse Forge hosts over 500,000 mods for 89 games. The biggest Mod collection in Curse Forge is Minecraft with 177.3 thousand mods, and 59.8 billion Mod downloads. Curse Forge is the most popular Minecraft mod distribution platform.

Regarding Minecraft, Curse Forge also lists other MVE extensions aside from Mods. These include shaders, data packs, resource packs, worlds, and Modpacks. This paper will solely focus on mods.

For each mod a developer lists on Curse Forge, they may also give their mod a category tag, upload several versions of the mods which can be labelled with the Minecraft and Mod Loader version they support, list the mod dependencies, and provide important links to external sites such as the mod's source repository or documentation. Curse Forge is practical for both developers and players.

In this paper, most of the decisions regarding mods are made with Curse Forge in Mind. This affects selection of Mods to benchmark, the selection of a Mod Loader and analyzing user mod categories.

3

Design

This section answers **RQ1**. It details the main components of the modding performance benchmark. This benchmark allows for automation of experiments to produce semi-replicable results. The benchmark uses a client-to-server system, with configuration features that allow the user to modify the benchmark environment to imitate an actual use case.

3.1 Requirements

The common requirements for a benchmark are considered, following requirements commonly found in the design of any benchmark. Extended requirements are added in relevant to this paper, chosen in accordance to the Benchmark's intended use case and aim.

The requirements listed below are applicable to the benchmark being designed. The descriptors of these requirements explain how they are relevant to the benchmark developed in this thesis. It also explains why it is difficult to address these requirements.

R1. **Fairness** The benchmark should limit bias from factors outside of the testing environment. Any overhead present, from profilers or other processes running from the machine conducting the benchmark, should not introduce unaccounted variance in performance impact measured by the benchmark. If there is unaccounted overhead, the measured performance impact may not be representative of the mod being benchmarked. If the overhead varies depending on the mod used, it may give affected mods unintended advantages in the performance impact it produces. This is difficult as low-overhead profilers have to be used, a consistent and replicable experimentation process will have to be designed, and a machine capable of automating processes with little variance will have to be used.

R2. **Representative Workload** The benchmark should be able to replicate an average user. In a normal use case, a real player will be able to understand the mod functionalities and the different actions they have to take to use those functionalities. The player simulation used will have to be able to simulate actions of that player and provide a workload representative to a real use case.

R3. **Relevant metrics** The benchmark should be able to collect metrics that give information about performance impact relevant to the user. This is difficult as simple metrics, such as CPU and memory usage, may not be of concern to players. Players would be more concerned with whether Minecraft is running smoothly, or if it has a lot of lagspikes. Formulating these application-level metrics is important in determining performance impact relevant to the user.

R4. **Parallelizable** The benchmark should support the testing of multiple mods within one environment. In a typical use case, players may use multiple mods at once. The benchmark should be able to support the use of multiple mods, even with the same workload. This is important as using multiple mods at once is common with players, and is one of the common areas where negative performance impact is a concern. This requirement can be a challenge as the player simulation used has to take into account the features all the mods being used provides.

R5. **Replicable** Repeating tests done by the benchmark should show a similar output. Allowing replicability of benchmark experiments increases the credibility of the benchmark. If a benchmark would produce a very different metric with the same experiment, then the benchmark would be unreliable. Some Minecraft features are randomized, such as the world's procedural generation, and the generation of game entities. This randomness will have to be reduced as much as possible by the benchmark.

R6. **Configurability** The benchmark should provide the user with configurations to change the desired experiment environment relevant to the mod tested. With the large variance in mod features, a configurable workload is necessary to sufficiently test the features of mods under experiment. This may be difficult as the benchmark will have to allow user configurations to player simulation and world data, while keeping the experiment fair and consistent.

3.2 Design Overview

The mod benchmark is designed around giving the user as much flexibility as possible in creating a benchmarking environment. This can mean creating their own player script, choosing the system environment, choosing a specific server configuration, and even choosing a pre-loaded world the server loads in.

It's unproductive to create one experiment that tests every single functionality a mod can affect - it would be a very long lists of experiments, all of which would have to be accompanied by a player simulation script that takes into account the intended player interaction with the mod. Instead, the freedom for the user to create a testing environment allows for a more personalized benchmarking process.

Specific mod features can be tested for performance impact, refining the benchmarking process. Users can choose to implement a player script and world scenarios that test every single feature in a mod. Users may also simply choose a more general set of player scripts applicable to several mods at once, allowing for a replicate benchmarking environment and a fair comparison between performance impact of mods.

To answer the requirements listed, the next sections will detail the design of the benchmark along with a reference to the requirement they fulfill.

Figure 3.1 depicts the design for the benchmark. The user provides configuration for two main sections, the system configuration and the workload. The server includes the Minecraft game loop, the Mod Loader used to load the server with mods, and the World Data of the game. The Player Simulation includes the simulation script, which creates and controls simulated clients. Each of these clients communicates with the server, and send packets an actual player-controlled client may send, apprearing to the server as regular, player-controlled client. Both these sections are profiled and metrics collected clients, which are processed and given to the player.

3.3 Benchmarking Process

There are three stages of the benchmarking process. The pre-configuration phase, the experiment phase, and the post-processing phase.

3.3.1 Pre-configuration

Here users may configure the desired workload for benchmarking. The configurable workload is listed below, and is explained in further detail in section 3.6. Each of these work-

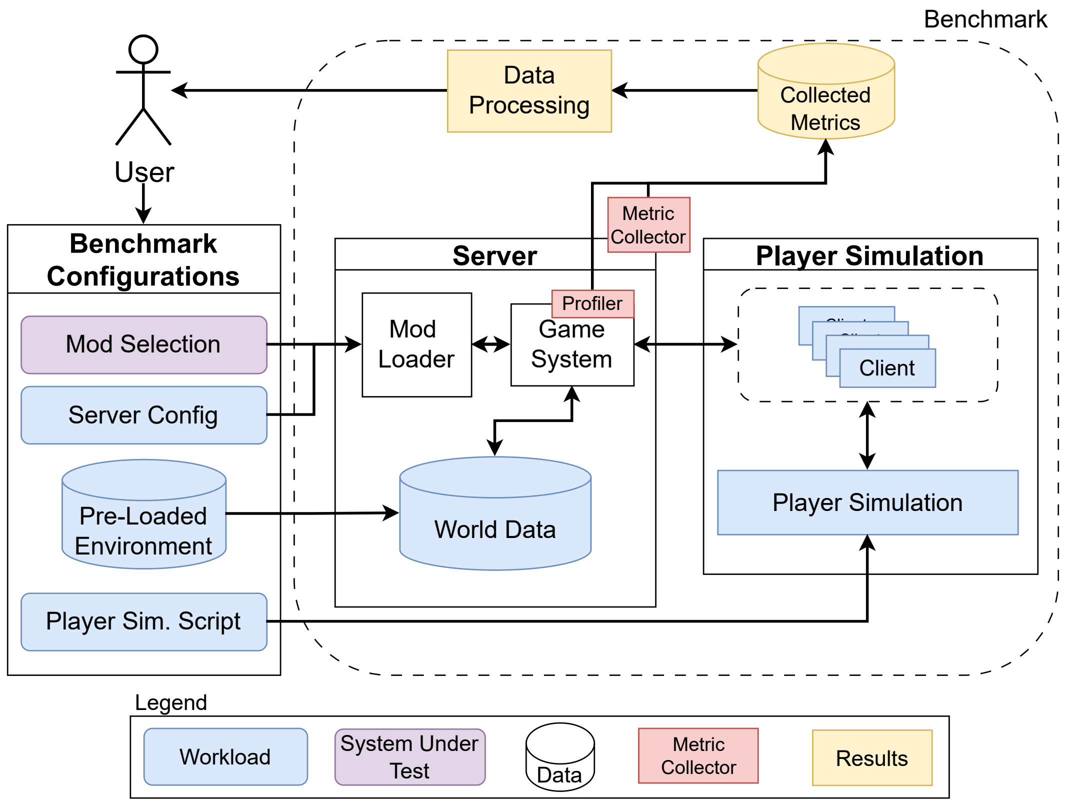


Figure 3.1: Design overview of our Benchmark for modded MVEs.

load configurations can be tweaked to the user's intended benchmark leading to an infinite amount of available workload to the user.

- **Loaded Mods**: Multiple mods may be chosen at the same time for benchmarking. User will have to make sure mods are previously compatible, and that Mod dependencies are met - the benchmark will not check this
- **Player Simulation**: Simulated players, multiple may run in parallel, each with different behaviours. User may use the provided player simulation library to create their own player workload.
- **Server Configs**: Basic server configurations any Minecraft server uses. World seed, gamemode, port number, and more. This may be a necessary tweak in case any Mods require a specific server configuration to work.
- **World Data**: Players may instruct the benchmark to start an experiment with a

new world - forcing the server to perform world procedural generation immediately. Alternatively, users may load their own world in. This may include a personalized testing world, a popular community-created world, or any mod that is compatible with the mods in use.

3.3.2 Benchmarking

Once the user has decided their configurations, the Benchmark fully handles the Mod Benchmarking. The user does not need to do anything except start the benchmark and wait for it to complete.

An automation software application will be used for this phase, for **RE5**. Automation allows for as fair a benchmark as possible by making sure actions taken to run the server, run player simulation and data collection are consistent. This reduces unaccounted variability, addressing **RE1**.

During this phase profilers are used to obtain metrics from the server. These profilers will collect as much metrics as possible, including Ticks Per Second, CPU usage, Memory usage and packets sent. These are collected in several formats, but will all be compiled and presented in the post-processing phase.

3.3.3 Post-processing

In this phase the Benchmark uses the metrics collected to present the data to the user. Relevant metrics are picked and put into graphs as a presentable format. The data is presented in graph format, which makes it easier for users to analyze the data and interpret performance impact. Each graph addresses a relevant Benchmark metric. This addresses **RE3**. During the tests, performance is monitored and collected. Output of all the server and client readings are available after the test.

3.4 Performance Metrics

System level metrics refers to measurements derived from the system, such as CPU usage, Memory usage, Packets sent and received. Application level metrics are more specific, and refers to measurements relevant to the success of the system's goals, such as server ticks per second, number of lag spikes, duration of lag spikes (both explained in section 3.5.2), and number of supportable players.

Metric Name	Description
Tick length	Time taken for server to process a tick
Tick per second	Number of Ticks per second
Player count	Number of players connected to server
Lag Spike count	Number of times a tick takes longer than 50ms
Lag Spike duration	Duration of time lag spikes occur
RAM usage	RAM usage of the server
CPU load	CPU load total
Network usage	Rate of incoming and outgoing bytes

Table 3.1: Chosen metrics for the Benchmark

3.4.1 Tick Duration

Ticks refer to the Minecraft game loop. Within a tick the server performs processes within the environment, updates the environment, and sends updates to the client.

An optimal Minecraft environment has 20 ticks per second, each tick lasting 50ms. Servers deliver processing every tick. If a server has processed a tick before the tick interval, the server waits until it is time to process the next tick.

3.4.2 Lag Spikes

As explained in Section 2.1, lag spikes are when the game runs well above the acceptable tick rate of 20 ticks per second.

There are two main types of lag spikes. The first is when a single tick takes abnormally much longer to calculate than usual, making the game freeze for a few seconds. This is typically caused by resource intensive events such as player disconnects and reconnects.

The second type of lag spikes are when the game takes a slight but noticeable increase in time to process several ticks. This is typically caused by a consistent amount of large workload being demanded of the server. This causes the game to play slower than usual until the source of the large workload is no longer needed to be processed. An example event that causes this is a large amount of game entities within a game region, causing the server to process a large amount of workload until the respective entities do not need to be processed anymore.

Mod	Num. of Downloads	Mod Category
Mouse Tweaks	253.6M	Utility & QoL, Storage, Miscellaneous
Appleskin	214.3M	Food, Map & Information
Controlling	213.0M	Utility & QoL, Cosmetic
Clumps	184.8M	Utility & QoL, Storage, Server Utility
Storage Drawers	170.5M	Storage, Cosmetic
Iron Chests	163.4M	Storage
Quark	159.6M	Ores & Resources, Cosmetic
Tinker's Construct	157.9M	Mobs, Technology, Processing, Armor, Tools & Weapons
Nature's Compass	147.5M	Map & Information, Biomes, Armor, Tools & Weapons, Technology
Just Enough Resources	147.3M	Ores & Resources, Mobs, Biomes, Map & Information, Addons

Table 3.2: Most downloaded Minecraft mods, excluding API & Library mods.

3.5 System Under Test

3.5.1 Modded Minecraft Version

The player may choose from one available version of Modded Minecraft to be modded, and if using a Mod Loader, then the used Mod Loader also has to be consistent. This means users can benchmark mods from any Minecraft versions, but it is strongly recommended that comparisons of benchmarked mods be done between mods in the same Modded Minecraft version.

Choosing a consistent modded Minecraft version makes sure the system mods are compared in is similar, minimizing external factors such as varying overhead between systems. This addresses **RE1**. The benchmark is not responsible for addressing performance inconsistencies between Minecraft versions.

3.5.2 Mods for Benchmarking

The user may refer to the Curse Forge website for a selection of Mods. The benchmark will at the very least support mods using the most popular Mod Loader, to satisfy **RE3**. Additionally, the benchmark will allow the user to benchmark multiple Mods at once in accordance to **RE2**.

However the benchmark will assume that the chosen mods will already be compatible for use with each other and will not fix compatibility issues already present. This is acceptable according to **RE2**, as in a real use case users will not use Mods in conjunction that are not compatible with each other. **RE2** does suffer as player simulation features may be missing to replicate an accurate use case of the Modded feature. This negative impact to Accuracy can be minimized by choosing Mods which features can be replicated using player actions available in Vanilla Minecraft. Table 3.2 shows the most popular, non-API mods. The most popular mod categories are Utility & QoL, Storage, and Cosmetic.

3.6 Workload

For workload, the user selects the system environment to run the mods in, mods for benchmarking, the intended player simulation, and optionally server and world configurations. These features are designed to adhere to **RE6** to give Benchmark users as much options as possible to trigger mod features.

3.6.1 Player Simulation

Next the user configures the player simulation. The player simulation should at the least be able to imitate player behaviour in Vanilla Minecraft. As stated when addressing RQ1, a lot of mod features are close to if not the same as Vanilla features, so bot simulation that can be done in Vanilla should sufficiently trigger most mod features.

The Benchmark will provide users the option to manually create player simulation. It will allow most Vanilla player functionality. The user may program their player configuration, but common player simulation scripts will be provided for the player to use.

3.6.2 Server and World configuration

Server configurations are available to the user. This may be necessary when mods require the user to further modify the server configurations to enable features. It also provides the user to configure the server to imitate a typical use case, such as memory allocation, world generation seed, spawning mechanics. Otherwise, the only necessary world configuration would be to enable JMX monitoring for the profilers to work, and to enable server offline mode, allowing player simulation to work.

The Benchmark further allows the user to choose the game world the server uses. This may include pre-made user worlds specific to mods, or even user-created worlds designed to be paired with the player simulation script to trigger and test mod features. A suitable world map can be manually created to create particular stress situations for mod testing. It is also possible to download player-created worlds to emulate a map an average user would use. Lastly a blank world may be used, making the server immediately start generating a procedurally generated world.

4

Implementation

This section answers **RQ2**, it describes the tools used to implement this benchmark following the design described in Section 3. Subsections will each describe the tool used, their role in the benchmark, and the justification of using that tool.

4.1 Overview

This benchmark implementation implements the design's focus towards replicability and fairness. the implementation is intended for use on the Distributed ASCI Supercomputer 6 (DAS-6) machine, and uses Ansible for automation. We decided to use the DAS-6 for its computation power, useful when using the machine as a distributed sytem to run multiple experiments at once. The machine's ability to use several distributed systems, referred to as nodes, allows for the execution of the benchmark server and client in separate systems. This reduces the impact server and client processes may have on each other, and allows us to measure more accurate metrics in the server.

The most complex part of the implementation is implementing player emulation in Section 4.3. The player emulation tool we used, Mineflayer, is well documented and updated, but does not inherently support mods. A module was found that allowed the player emulation program to work with the Forge Mod Loader version of modded Minecraft. However troubleshooting of the module had to be done as it was not as well updated as the main player emulation program.

To implement metric collectors and profilers, a JMX metric collector software called Jolokia was used to collect game metrics from launching the server and throughout the experiment. To collect process metrics, a Minecraft profiler called the Spark profiler was used. It allows us to examine which processes in the server took up the most rick duration.

4.2 Forge Mod Loader

This implementation uses the Forge Mod Loader (FML). FML is the most popular Mod Loader available in Minecraft. It was one of the earliest Mod Loaders created for Minecraft, supports Minecraft versions as early as version 1.0.0, (10) and has 10,000+ available Mods for Minecraft version 1.19.2 alone (6). As such it has an older and larger community and mod selection wider than other Mod Loaders such as Fabric or Quilt.

Acting as an API FML provides developers with hooks and tools to simplify Mod development. Developers can easily register new features such as new game mobs and blocks. Developers can use FML's event bus to create mod features, and intercept Vanilla events. It allows for developing side-specific mods, either client-side or server-side mods, or both. It provides mod versioning which enables developers to simplify mod updating processes for players (11).

While FML has a much more extensive list of Mods, its development team has slowed down with most members migrating to a new Mod Loader project called NeoForge , and updates are slow. Additionally FML has a higher overhead than more modern Minecraft Mod Loaders, and is overall less efficient (12). Its core functionalities were developed during Minecraft versions, comparably inefficient compared to Mod Loaders developed recently. As such, using the Forge Mod Loader is a trade-off, gaining between a wider selection of mods, but sacrificing efficiency.

4.3 Implementing Player Emulation

Mineflayer is a high-level JavaScript API to create Minecraft bots (13). By itself, Mineflayer does not run an actual instance of a Minecraft client. It instead sends a server packets a typical client would send to communicate, thereby simulating players. The server does not know that these are not real players and such treats them like normal. Mineflayer sends acknowledgements, player inputs, data requests, and anything that a player using a Vanilla client would send. Mineflayer does requires servers to work in offline mode, as it does not request account authentication from clients. Mineflayer cannot simulate Minecraft account authentications, which are usually kept private with purchased Minecraft accounts.

Mineflayer works with multiple Minecraft versions. It has authentication handling that recognizes the Minecraft handshake protocol, and can handle requests such as ping and status requests. With the Forge Mod Loader, Mineflayer requires an addon called the Minecraft-Protocol-Forge (MPF). FML has a different authentication protocol that ensures

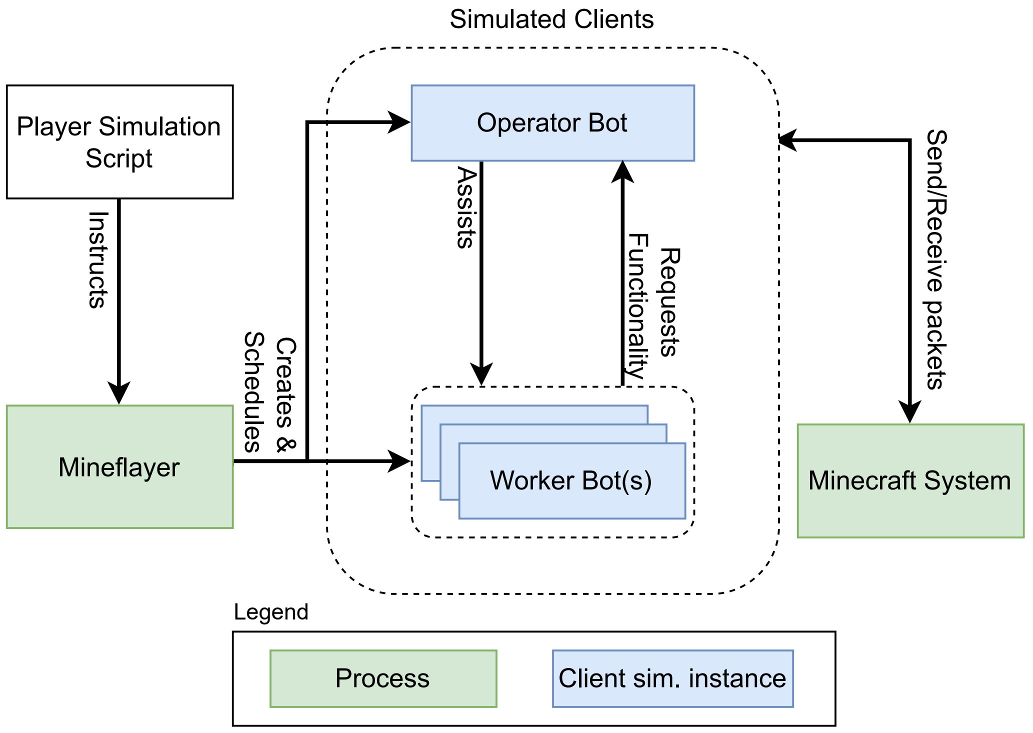


Figure 4.1: Implemented system of the player simulation script using Mineflayer.

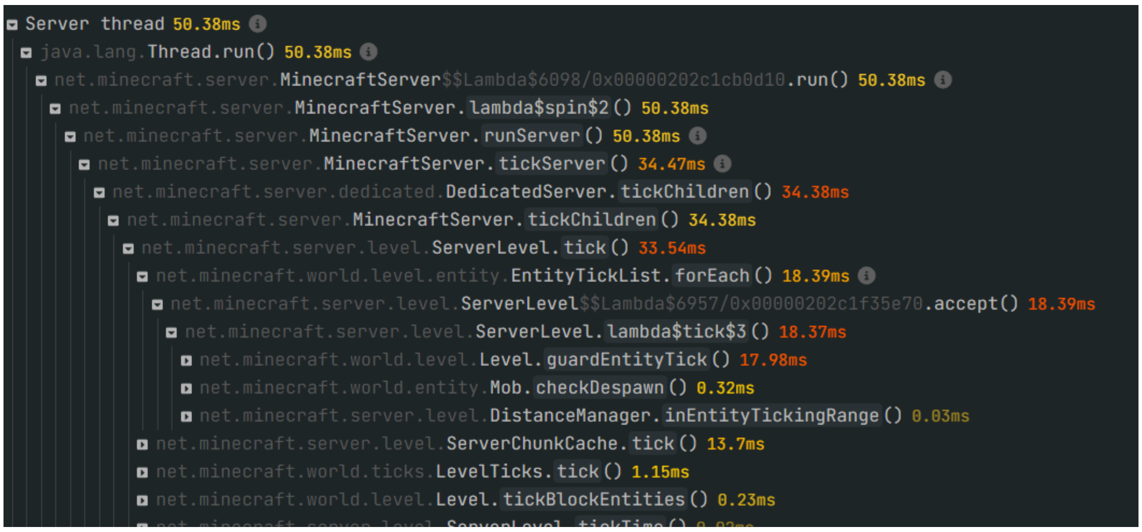
clients have any required client-side mods. Using auto-versioning, MPF sends back a response that lists the same mods the server requests from the client, even though no client or modded client is actually present (11).

Therefore as an API that provides player simulation and compatability with FML, this implementation will use Mineflayer as its main Player Simulation tool. As of writing, Mineflayer supports Modded Minecraf up to versions that use the FML3 protocol. That is, Mineflayer works up to FML version 1.19.4 - 45.3.0.

4.4 Collecting Performance Metrics

Several profilers are used in this benchmark. These profilers each are chosen to obtain specific system-level and application-level metrics from the machine, server, and the Minecraft process. This section will describe each profiler and what metrics they collect.

The Spark Minecraft profiler (Spark) is a mod for Minecraft that profiles metrics from the server, including the process' CPU usage, Millisecond Per Tick (MSPT), Ticks Per



```
Server thread 50.38ms
 java.lang.Thread.run() 50.38ms
  net.minecraft.server.MinecraftServer$$Lambda$6098/0x00000202c1cb0d10.run() 50.38ms
   net.minecraft.server.MinecraftServer.lambda$spin$2() 50.38ms
    net.minecraft.server.MinecraftServer.runServer() 50.38ms
     net.minecraft.server.MinecraftServer.tickServer() 34.47ms
      net.minecraft.server.dedicated.DedicatedServer.tickChildren() 34.38ms
       net.minecraft.server.MinecraftServer.tickChildren() 34.38ms
        net.minecraft.server.level.ServerLevel.tick() 33.54ms
         net.minecraft.world.level.entity.EntityTickList.forEach() 18.39ms
          net.minecraft.server.level.ServerLevel$$Lambda$6957/0x00000202c1f35e70.accept() 18.39ms
           net.minecraft.server.level.ServerLevel.lambda$tick$3() 18.37ms
            net.minecraft.world.level.Level.guardEntityTick() 17.98ms
            net.minecraft.world.entity.Mob.checkDespawn() 0.32ms
            net.minecraft.server.level.DistanceManager.inEntityTickingRange() 0.03ms
         net.minecraft.server.level.ServerChunkCache.tick() 13.7ms
         net.minecraft.world.ticks.LevelTicks.tick() 1.15ms
         net.minecraft.world.level.Level.tickBlockEntities() 0.23ms
```

Figure 4.2: Example usage of the Spark profiler to diagnose cause of lag.

Second (TPS), Player count, Game Entity count, and active chunks count. Spark profiler also provides a viewer of the profile as an extendable tree, letting users see which part of the Minecraft process is taking the most computing time per tick on average. Figure 4.2 shows an example of using Spark profiler to diagnose lag of a tick taking longer than 50ms. It shows that the EntityTickList method is taking up the most ms and could be causing the negative performance impact.

This profiler provides a lot of important application-level metrics relevant to the benchmark. It covers most of the metrics needed in the benchmark. However, Spark processes its data in average metric per minute, and does not provide per-second or per-tick metrics. Therefore it is difficult to justify the use of Spark profiler when the user is concerned about a short lag spike lasting 2-3 seconds, barely affecting overall metrics.

Spark Profiler is used to mainly acquire application-level metrics from the Forge Server. Metrics most notable are Ticks Per Second (TPS) and Milliseconds Per Tick (MSPT).

Jolokia is a JMX-HTTP bridge which assists JMX monitoring and management tools. Here, Jolokia is used to access application-level metrics related to the Minecraft JVM (14). It is launched alongside the Minecraft launcher, and provides access to the JVM operation during execution.

From Jolokia, each individual tick can be monitored, and MSPT can be collected. Jolokia means to cover application-level metrics that Spark profiler cannot accurately obtain.

5 Experiment Results

This section addresses **RQ3**. We explain the use of the Benchmark in an experiment. The experiment centers around finding the performance impact of Mods categorized as World Generation, and runs in the modded Minecraft version **1.19.2 - 43.4.0**. The first section explains the experiment setup. The second section explains the experiment results.

5.1 Experiment Setup

This section describes the chosen Minecraft version, mod choice, player simulation, and world choice for the experiment. This follows the intended implementation, and gives justification on the choices made to create the benchmarking environment.

Selecting a Minecraft Forge Version

The Modded Minecraft version 1.19.2 is chosen for expriments. Referring to Table 2.1, the Minecraft version 1.20.1 has the most mods available for use. But since Mineflayer does not support this Modded Minecraft version, this experiment will use the next best option 1.19.2. This version has a sufficiently large amount of mods available for testing, and is the most updated version compatible with Mineflayer. Versions higher than 1.19.4 are not considered as at the time of writing, the Mineflayer-protocol-forge handshake tool detailed in section 4.3, does not yet support them.

Choosing a mod category

As specified in Section 3, for a fair comparison between mods, it is recommended to compare mods within a specified category. For the experiments done in this thesis, the **World Generation** mod category is chosen for evaluation. This mod category do not often require the user to perform special interactions, other than traversing the newly generated areas. This can be emulated using Mineflayer.

MC Version	Mods listed
1.20.1	>10,000
1.19.2	>10,000
1.18.2	>10,000
1.16.5	>10,000
1.12.1	5,927
1.19.4	5,709
1.17.1	3,991
1.15.2	3,870
1.16.4	3,789
1.18.1	3,612

Table 5.1: A list of the top Minecraft Versions listed in Curse Forge

World Generation mods affect how Minecraft generates and populates the game world. Methods to do this might include modifying world generation mechanics, adding new areas to the existing available areas to generate, adding new blocks and structures for generation, adding new mobs to populate areas, and even adding a whole new dimension complete with its own unique areas and mobs.

Choosing mods with varying implementations

For this experiment three mods will be tested, each with an identical workload regarding player simulation, game world, and server configurations. These three mods are chosen for their popularity, and their intended features. Table 2.1 shows the three mods chosen. All these mods are within the World Generation category listed in CurseForge.

These three specific mods were chosen for the different features they provide to modify Minecraft's world generation. We chose to pick mods with different functionalities to specifically compare the performance impact of these new functionalities that all implement a new world generation feature.

Biomes O' Plenty introduces new blocks and 50+ new areas for the game to generate. This includes ares within different dimensions in the game. Biomes O' Plenty does not introduce new game mobs, and focuses on enhancing Minecraft world generation. Areas from the base game are still allowed to be procedurally generated.

Naturalist introduces new game mobs intended to replicate real-life animals. These mobs are given complex behaviour, such as having different interactions with the player depending on the current in-game time. The workload Naturalist produces is related directly to mob or entity behaviour and events. These events are related to world generation,

Mod name	**Downloads**	**Features**
⓪ Vanilla	n/a	Base game system
① Biomes O' Plenty	139.3M	New Areas & Blocks
② Naturalist	24.5M	New Mobs
③ The Twilight Forest	130.5 M	New Dimension, Mobs, Areas & Blocks
④ All mods	n/a	All of the above

Table 5.2: Chosen systems to be tested for experiment.

Experiment	Walker	CL	Walker	CL	Mods
Overhead Test	4	2	0	0	⓪ ⓪*
PG Emulation	0	4	0	2	⓪ ① ② ③
Full Emulation	2	2	1	1	⓪ ① ② ③ ④

Table 5.3: Chosen Mod workload for World Generation benchmarking.

as the mobs exist to enrich the areas they appear in, and are spawned in relation to the procedurally generated area.

The Twilight Forest introduces a new dimension into the game called The Twilight Forest, where only new areas occur, is filled with new mobs, and is totally separate from the three existing vanilla dimensions. This new dimension is accompanied with new blocks and game mobs. It is surmised that Twilight Forest most affects performance when players are within the new dimension.

Game world

To keep procedural generation as fair as possible, a constant world seed will be used. Some mods will require a knowledge of the world, such as places of interest such as mod-introduced areas. A minimally pre-loaded world will be used. This world would have previously been examined by the user, to find areas of interest. For example, for The Naturalist mod, knowledge of the generated areas is needed to teleport Walker bots to appropriate areas they cannot get stuck in. Additionally, since the chosen Twilight Forest mod implements a new dimension, bots will be deployed within the mod's introduced dimension. This is illustrated in Table 5.3 where the green column illustrates the base Overworld dimension, and the red column illustrates a separate dimension relevant to the mod being tested.

To keep fairness in workload, during experiments using mods that do not introduce a new dimension, identical bots will be sent to an existing base Minecraft dimension to even out the workload. For example, in the case of the full emulation experiment: When testing

Twilight Forest a Walker and CL bot are deployed in the Twilight dimension. When testing Naturalist, Biomes O' Plenty and Vanilla, a walker and CL bot will be sent to the Nether dimension.

Player Simulation Within World Generation mods, what is most likely gives the most stress is during world generation, and players roaming or interacting with the generated areas. The player connected amount will be kept consistent in this experiment, as testing for performance impact is intended to test mods with a consistent workload. Table 5.3 details the experiments that will be run, showing a constant 7 players being connected per experiment, with an Operator bot not being shown in the table. The bots used will be detailed below:

Operator bot: Only one bot of this type is used per experiment. It is not included in table, for its purpose is to facilitate experiments and not to emulate an actual bot. This bot is given admin permission. It is responsible for starting the Spark profiler and giving permissions and commands to other bots. The operator bot would communicate with the other bots through in-game chat. In the chat, worker bots would as the operator bot for permissions, items, or to teleport them to a certain area.

Chunk Generator bot: A bot that flies at constant pace around the world randomly to constantly generate new chunks. This workload is sufficient to evaluate procedural generation.

The Chunk Generator bot would require a special gamemode "spectator" that allows it to fly around the world and phase through blocks uninterrupted, while still generating chunks and spawning mobs in the viscinity.

Roamer bot: A bot that roams around generated areas in a fixed radius. This bot would have to be given the "survival" game mode, so that mobs within generated areas can interact with the player. This workload is sufficient to evaluate game mob and block interactions.

The Roamer bot would require tools to interact with the game environment, such as a Sword or a Pickaxe item. It would also have to be given an infinite amount of health, as having the bot die in-game would stop it from generating a constant roaming workload.

5.2 Main Findings

The detailed experiments were performed with the system implemented in 4. Each experiment followed the setup explained, and spanned a total of 6 minutes. The first minute of each experiment was reserved to allow time for bot simulations to connect to the server,

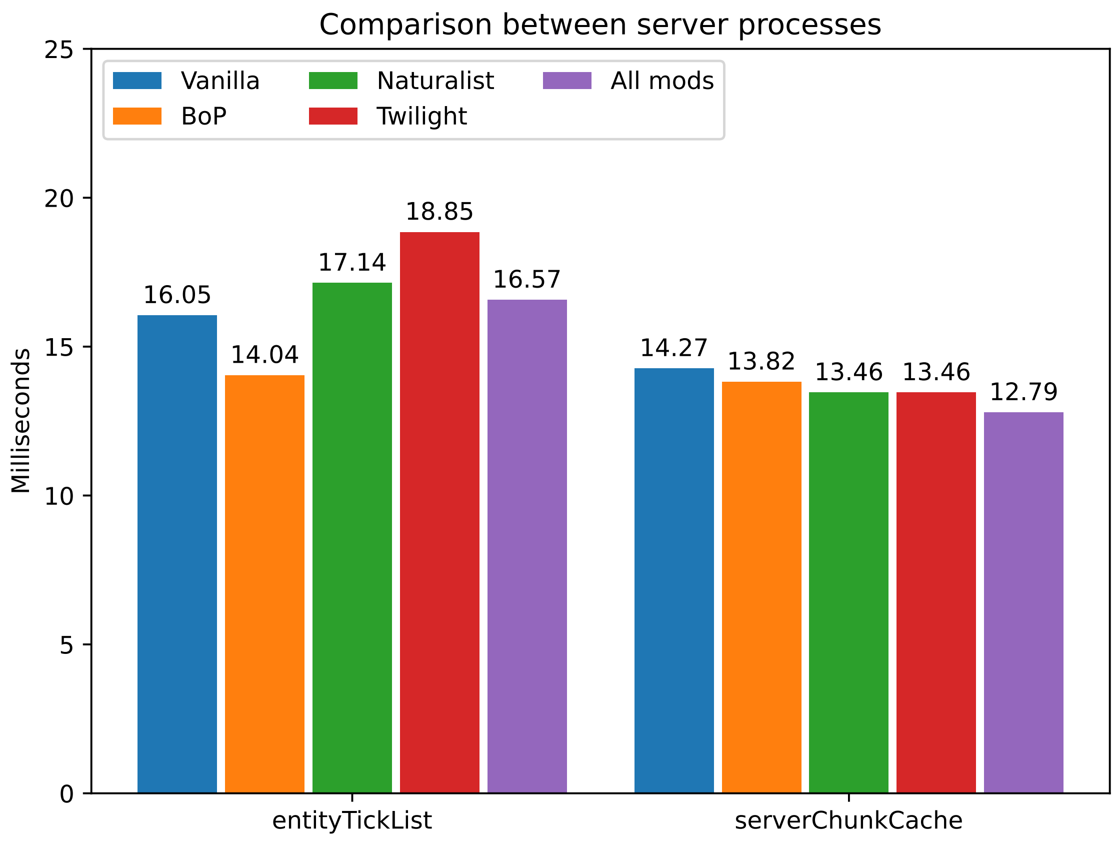


Figure 5.1: Processing entities show a higher performance impact than managing environment chunks.

and the bots to be set up. For each experiment, two separate DAS-6 nodes were used: one to run the server, and one to run player simulation. After each experiment, process logs were analyzed to make sure the experiment ran without problems. In the case where a client abruptly disconnected due to timeout issues, the experiment was reconducted.

The following sections describe the main findings from each experiment. A section is reserved to show the overhead the Spark profiler imposes on the system.

5.2.1 MF1: Entity-related processes have more performance impact than managing chunks

During server execution, Spark profiled the processes conducted, providing the amount of time each process took per tick on average. Two important processes, the entityTickList and serverChunkCache were collected. The entityTickList process iterates over entities in the game environment, updating their status and behaviours. The serverChunkCache iterates over chunks in the game environment, performing tasks to managing and updating the tasks. Some tasks are as listed: block updates, entity spawning, chunk generation. In short, entityTickList processes entities and serverChunkCache processes chunk behaviour and rules.

When comparing the two processes, entityTickList shows a higher millisecond required compared to serverChunkCache. It is most apparent in the Twilight mod, where entityTickList takes up 5 more milliseconds to process than serverChunkCache. Looking at Figure 5.2.1, the mod with the closest amount between the two processes is Biomes O' Plenty.

This might be due to the new areas Biomes O' Plenty introduces. These new biomes may not have registered themselves to allow game entities to generate on them. This can reduce the amount of entities generated in the game world, since these new areas do not allow the normal amount of entities to generate there. As a result, there is less time needed to process entityTickList. This may also explain the reason why the experiment with All Mods have lower process times than the experiment with just Twilight Forest, despite them both having Twilight Forest, the mod with the most performance impact.

Despite the reduction in entities, the entityTickList still shows a higher time required to process.

5.2.2 MF2: Performance impact higher in mod-implemented dimension compared to base Vanilla dimension when considering average performance

The figure 5.2.2 shows the boxplot of the distribution of tick length taken from the experiment with only Chunk Loader bots. Twilight Forest has the highest MSPT during experiments, consistently going over the 50 MSPT threshold after players are connected and sent to the Mod's dimension. The boxplot shows that when compared individually, the Twilight Forest mod shows the highest distribution of tick lengths when compared to the other mods.

Looking further into the collected metrics, when looking at Figure 5.2.2, the Twilight Forest mod shows a less number of lag spikes as compared to the Naturalist mod. This means that although the Twilight Forest mod has a higher distribution of tick length, it still performs more consistently than the Naturalist mod, which only introduces new game entities.

Despite introducing its own entities, dimensions and blocks, Twilight Forest performs at a more stable rate than the Naturalist mod. From this finding we can assume that mods that add new dimensions do increase performance impact the most, but it does not necessarily have a negative impact on tick consistency and the number of lag spikes.

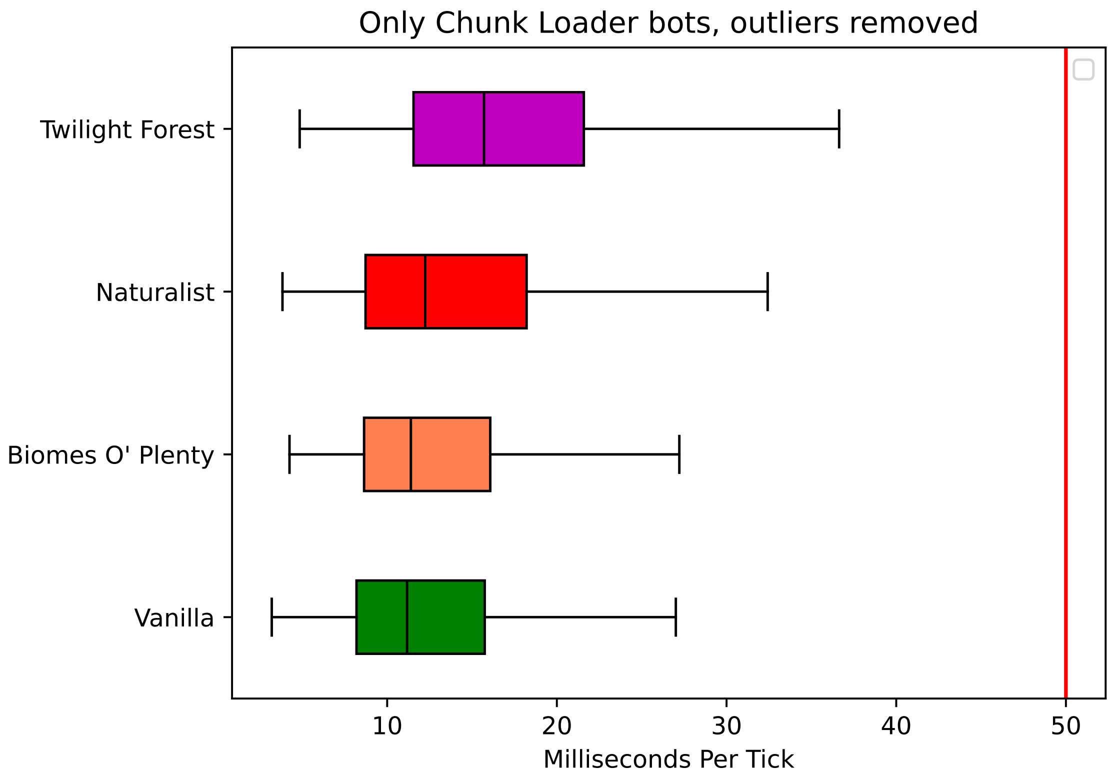


Figure 5.2: Twilight Forest shows highest tick length distribution individually.

5.2.3 MF2: Increase in MSPT higher in sum of mods run individually compared to mods run in parallel

Figure 5.2.3 shows the difference in mean MSPT accross mods run individually compared to mods run in parallel, in reference to a Vanilla Minecraft experiment. When added together, the mean milliseconds per tick across all three mods add up to an increase of 13.34ms compared to the Vanilla instance. However when all mods are run in parallel, the mean mspt only shows an increase of 9.63ms, which is 72% of the sum of the increase in MSPT in mods run individually.

To explain this, it could be that the mods run in parallel have overlapping features. For example Biomes O' Plenty introduces new areas to the game that are not recognized by Naturalist, and no entities from Naturalist will be generated in the areas generated by Biomes O' Plenty, thereby reducing the server process needed to process game entities. Another reason for this performance impact difference is the maximum amount of game entites a Minecraft instance allows. Within a chunk, Minecraft generates entities in respect to the amount of already existing entities within the chunk. As such, when each mod is run

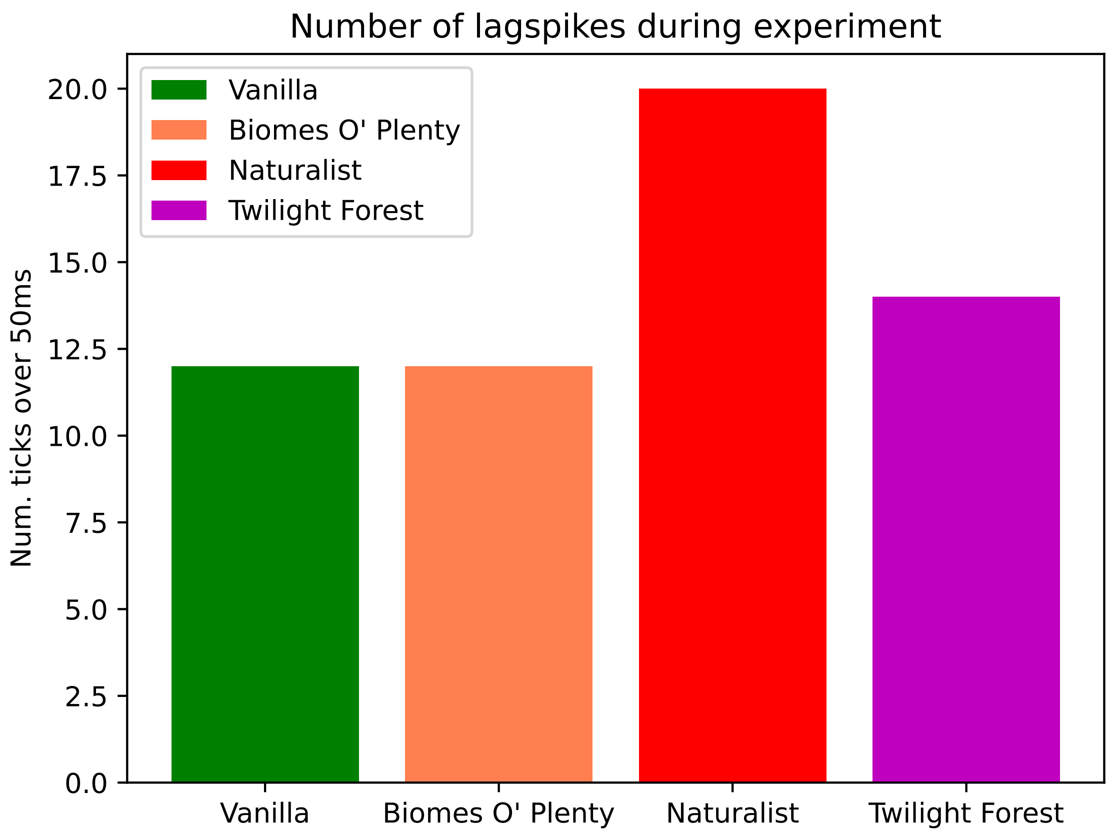


Figure 5.3: Twilight Forest shows highest tick length distribution individually.

individually, there could be a higher number of entities present summing all experiments. This is in comparison to the one experiment with all mods. And as seen in main finding 5.2.1, entities induce a higher performance impact than other chunk processes.

5.2.4 Spark profiler overhead

A simple experiment was conducted with identical bots, world and server configurations. As shown in figure 5.2.4, little difference is seen between the tick length distribution. Jolokia produced metrics that showed systems with and without the Spark profiler had an average of 21.27ms and 21.30ms respectively. Therefore, the Spark profiler is within the acceptable amount of overhead for use in the experiment.

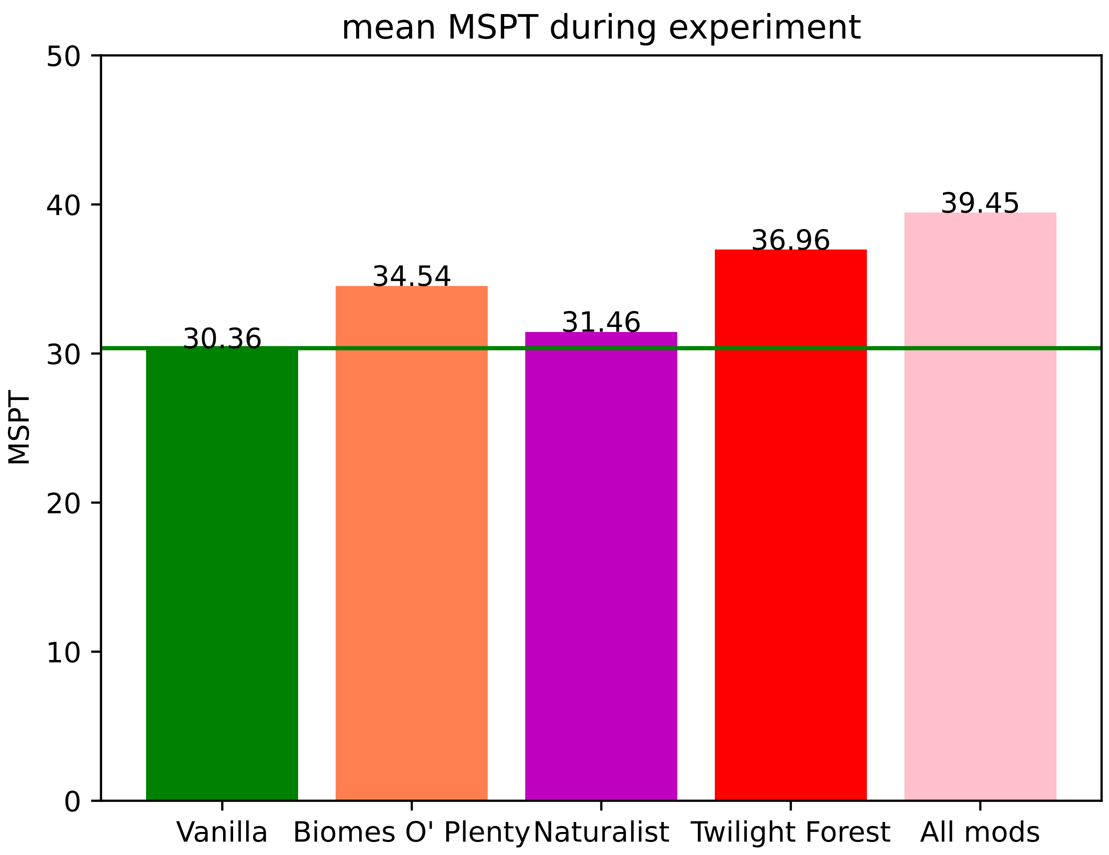


Figure 5.4: Sum of individual mod performance impact smaller than mods run in parallel.

5.3 Limitations and/or threat to validity

A limitation of this experiment is the inconsistency with the connection stability of Mineflayer. As mentioned at the start of experiment setup, simulated clients would randomly time out from the server. This would cause lagspikes irrelevant to the mod being tested. This could be the result of the mineflayer-forge-protocol not accounting for a mod request from the client. As a result the client would not send the correct response packet, and the server would disconnect them. This could potentially be solved in the future as the mineflayer-forge-protocol gets improved.

Another limitation is inconsistencies during the Minecraft game process. Though this benchmark has limited variability as much as possible, Minecraft still uses randomness to generate entities and their behaviour, along with game events. As such, experiments may not produce the exact same results when replicated. This is a concern especially when comparing mods that have a similar performance impact. The inconsistency in workload

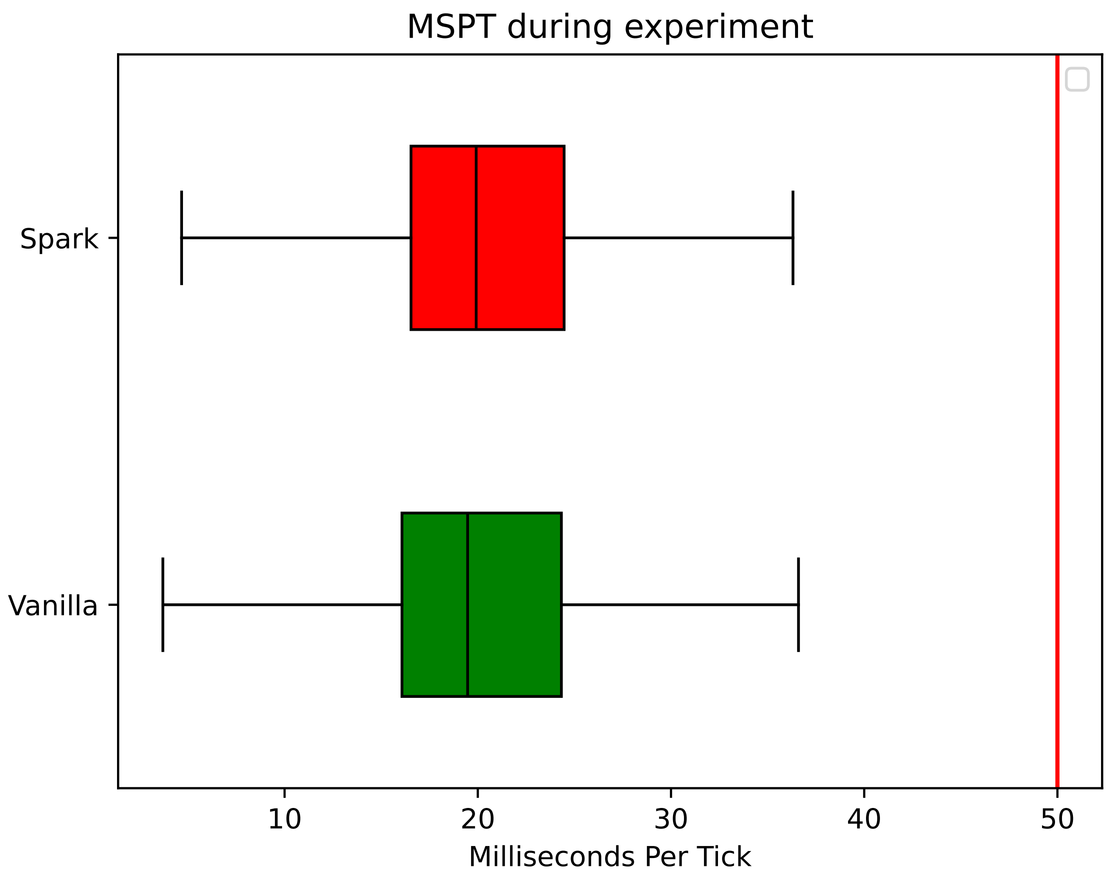


Figure 5.5: Tick length comparison between experiments with and without the spark profiler.

might cause a mod to produce higher or lower performance impact randomly. A possible fix is to implement, or modify an existing modded Minecraft version, such that all randomness aspect of the game is based on the server seed, so that a predictable set of events happens during the experiment. Another more extreme fix is to completely remove randomness from the game except world generation. This would also concern the random generation of entities.

Another limitation is that the benchmark is only compatible for Minecraft Forge versions up to 1.19.4. This prevents us from experimenting on mods from other Mod Loaders. This is especially a concern as the Forge mod loader is decreasing support, with most of its development team moving to a new, more modern Mod Loader project called NeoForge (15). The main problem to this is that as more modern Mod Loaders develop, this benchmark will become increasingly irrelevant. This is unless the mineflayer-forge-protocol team develops support for newer versions of Forge.

6

Related Work

As far as we are aware, this thesis is the first attempt to benchmark the performance of user-created mods for Minecraft, in an academic work. There is a lack of research concerning the technical aspect of Minecraft modding. There exists works that conducts benchmarking on MVE systems similar to Minecraft.

An empirical study of the characteristics of popular Minecraft mods (16). This study conducts empirical research into the characteristics of Minecraft mods accross 1,114 popular and 1,114 unpopular Minecraft mods from the CurseForge mod distribution platform. It analyzes mod relations regarding mod characteristics and popularity. Our paper uses this study when choosing a mod category for experiments, and to gain better understanding of why some mods are popular and others are not.

Yardstick: A Benchmark for Minecraft-like Services (17). This paper developed a benchmark for a Minecraft-like MVE. It investigates the performance and scalability of a Minecraft server in relation to player count. It finds that Minecraft-like MVEs are poorly parallelized. The Yardstick benchmark implemented influenced the design and implementation of the benchmark used in this thesis.

Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games (18). This paper also uses a benchmark for a Minecraft-like MVE similar to Yardstick. It expands upon the Yardstick paper, investigating if performance variability is the cause of poor paralellization of Minecraft-like services. It finds that environment-based workloads such as chunk loading contribute greatly to performance variability.

7 Conclusion

The creation and use of mods in MVEs are only getting more popular, and it is thanks to the dedication of users and developers who enjoy the MVE. Keeping a consistent standard of mods is challenging as these developers and users are mostly independent, with a lack of knowledge useful when creating and optimizing mods to run in an affordable machine. The mod benchmark serves as a tool to assist in investigating the performance impact of user-created mods, serving as a tool for research. With this, we answer the research questions listed at the start of the thesis.

RQ1. **How to design a benchmark for comparing Modded MVE performance between different mods?** In Chapter 3, we designed a benchmarking model that provides users with the ability to configure a benchmark workload suitable to the mod being tested. The design focuses on replicability and the collection of metrics relevant to the user. We designed a model that allows the user to configure a workload applicable to mods with similar functionalities.

RQ2. **How to implement such a benchmark?** In Chapter 4, we implemented the designed benchmark model for use with the Forge Mod Loader and Minecraft versions 1.19.4 and older. We implemented a player simulation program that works with modded Minecraft servers. We used profilers and metric collectors that collected system and application metrics that are relevant to the user. We designed a workload that was applicable to a Vanilla Minecraft instance, and modded Minecraft instances relevant to World Generation.

RQ3. **How to use the benchmark to compare the performance of mods on MVEs?**, In Chapter 5, the implemented benchmark was used to compare the performance of mods with the World Generation functionality. We found mods affect entity processing negatively, with all tested mods increasing the time taken to process entities. We found that mods that implement a new dimension average a higher tick length, but does not affect significantly the amount of lag spikes. We found that the increase in performance impact is smaller when mods are run in parallel, as compared to the sum of performance impact of the same mods run individually, showing a 3.71ms difference.

References

[1] Richardson I. Hjorth L. *Playing During COVID-19.* Palgrave Macmillan, Cham, Swizerland, 1st edition, 2020. 1

[2] Tom Gerken. **Minecraft becomes first video game to hit 300m sales**. *BBC*, 10 2023. 1

[3] Jonathan Delafield-Butt Omar Alawajee. **Minecraft in Education Benefits Learning and Social Engagement**. In *International Journal of Game-Based Learning (IJGBL) 11(4)*, 2021. 1

[4] Ian Knight Reem Al Washmi, J Bana. **Design of a Math Learning Game using a Minecraft Mod**. *European Conference on Games Based Learning: ECGBL2014*, pages 10–17, 2014. 1

[5] Ruck Thawonmas Satoko Ito, Marcel Wira. **User Friendly Minecraft Mod for Early Detection of Alzheimer's Disease in Young Adults**. In *2022 IEEE Games, Entertainment, Media Conference (GEM)*. IEEE, 2023. 1

[6] curseforge. **CurseForge website**. 1, 22

[7] mojang. **Java or Bedrock edition**. 7

[8] Notch. **Notch's statement on modding**. 7

[9] Peter Christiansen. **Players, Modders and Hackers**. In *Understanding Minecraft : Essays on Play, Community and Possibilities.* McFarland, 2014. 8

[10] Aaron Mills. **A Brief History of Minecraft Modding**. 22

[11] Forge. **Forge Mod Loader documentation**. 22, 23

[12] Maddy Miller. **Forge, NeoForge, and Fabric. Which should you use?** 22

[13] PrismarineJS. **Mineflayer Github**. 22

[14] Roland Huss. **Jolokia**. 24

[15] NeoForged team. **The NeoForged Project**. 34

[16] et al. Lee D., Rajbahadur. **An empirical study of the characteristics of popular Minecraft mods**. *Empir Software Eng*, 2020. 35

[17] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 243–253, New York, NY, USA, 2019. Association for Computing Machinery. 35

[18] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games**. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23, page 173–185, New York, NY, USA, 2023. Association for Computing Machinery. 35

Appendix A

Appendix

The code used for the implementation of the benchmark can be found at **https://github.com/atlarge-research/Minecraft-Mod-Benchmark**