

Vrije Universiteit Amsterdam



Bachelor Thesis

---

# Solarstick: A Evaluation Framework For JVM Configuration Impact on MVE Performance

---

**Author:** Dilano Emanuel Jermaine Doelwijt      2740234

*1st supervisor:*      Jesse Donkervliet  
*2nd reader:*         Daniele Bonetta

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

August 20, 2024

---

## Abstract

Modifiable virtual environments (MVEs) entertain millions of players each year. They allow players to explore their creativity through sculpting terrain, constructing buildings, and adventuring through the endless world provided by the MVE. Along with their unique gameplay comes a unique workload that has proven challenging to scale to more than a few hundred players per instance. This number provides a stark contrast with the hundreds of millions of players that play these games globally. Given their popularity, the scalability of these games must increase by several orders of magnitude to allow their large communities to play together. The amount we know about the performance of MVEs is steadily increasing. However, the impact of the JVM's configuration on the performance of the MVE has remained unstudied. Even though JVM configuration can significantly impact the performance of applications. In this thesis, we design an evaluation framework to study the impact of JVM configuration on MVE performance. We identify configurations that have an impact on the performance of the JVM. Furthermore, quantify the impact that these configurations have. Specifically, we observe that the choice of garbage collector can increase the maximum tick time by 390%. Other JVM parameters also have a significant but smaller impact on MVE performance. For example, the chosen maximum heap size can decrease the minimum CPU usage by 22%. Lastly, using MVE-specific JVM configurations can decrease minimum CPU usage by 22% and maximum heap memory usage by 37%. Most importantly, MVE-specific JVM configurations can lead to an increase in the maximum player count, showing an increase of 10 during experiments.

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement . . . . .	2
1.2	Research Questions . . . . .	2
1.3	Research Methodology . . . . .	3
1.4	Thesis Contributions . . . . .	3
1.5	Plagiarism Declaration . . . . .	4
<b>2</b>	<b>MVE and Java Architecture</b>	<b>5</b>
2.1	MVE model . . . . .	5
2.2	Java Architecture . . . . .	7
2.3	JVM Configuration . . . . .	7
2.3.1	JVM implementation . . . . .	8
2.3.2	Heap . . . . .	8
2.3.3	Garbage Collector . . . . .	9
<b>3</b>	<b>Design of Solarstick</b>	<b>11</b>
3.1	Solarstick Requirements . . . . .	11
3.2	Solarstick Design Overview . . . . .	12
3.3	Workloads . . . . .	13
3.4	Metrics . . . . .	14
<b>4</b>	<b>Implementation of Solarstick</b>	<b>17</b>
4.1	Solarstick Implementation Overview . . . . .	17
4.1.1	Directing Node . . . . .	17
4.1.2	Client Node(s) . . . . .	19
4.1.3	Server Node . . . . .	19
4.2	Workloads . . . . .	20

## CONTENTS

---

<b>5</b>	<b>What Is the Performance Impact of JVM Configuration on MVEs?</b>	<b>23</b>
5.1	Experimental Setup . . . . .	23
5.2	MF1: OpenJDK and GraalVM CE Do Not Show Significant Differences . .	25
5.3	MF2: GC Implementation Can Have Significant Impact on Performance . .	27
5.4	MF3: A Higher Maximum Heap Size Can Positively Impact CPU Load . . .	31
5.5	MF4: MVE Specific JVM Configurations Can Positively Impact Performance	33
5.6	Limitations and/or Threat to Validity . . . . .	38
5.7	Evaluation Summary . . . . .	39
<b>6</b>	<b>Related Work</b>	<b>41</b>
<b>7</b>	<b>Conclusion</b>	<b>43</b>
7.1	Answering Research Questions . . . . .	43
7.2	Limitations and Future Work . . . . .	44
	<b>References</b>	<b>49</b>
<b>A</b>	<b>Reproducibility</b>	<b>55</b>
A.1	Abstract . . . . .	55
A.2	Artifact check-list (meta-information) . . . . .	55
A.3	Description . . . . .	56
A.3.1	How to access . . . . .	56
A.3.2	Hardware dependencies . . . . .	56
A.3.3	Software dependencies . . . . .	56
A.4	Installation . . . . .	56
A.5	Experiment workflow . . . . .	56
A.6	Evaluation and expected results . . . . .	57
A.7	Experiment customization . . . . .	57
A.8	Methodology . . . . .	58
<b>B</b>	<b>JVM configurations</b>	<b>59</b>

# 1

## Introduction

Modifiable virtual environments (MVEs) make up a large part of the active gaming community. Minecraft, for example, has sold 300 million copies since its release in 2011 (1), and has achieved over 140 million active users in the past (2) and still bolsters a large active community counting millions of players.

MVEs are a highly popular type of game in which players have fine-grained control over the virtual environment. For example, players can modify and even program the environment at will. Additionally, MVEs are successfully used in educational settings across the world (3, 4), Minecraft even has an education edition (5).

Due to the modifiable nature of the world and the number of actors the server has to keep track of, scaling MVEs has proven difficult (6). This has led to MVEs dividing their players across many isolated instances. Microsoft's cloud-based Minecraft service, Minecraft Realms (7), is limited to 10 players per instance.

The environment containing all files and programs necessary to execute a particular program is called the Runtime Environment (RE). MVEs rely on REs to operate. The RE is responsible for the facilitation of the execution and the actual execution of the MVE. The implementation and configuration of the RE can impact the performance of an application running on the RE (8). However, the impact of RE configuration on MVE performance is unknown.

In this thesis, we investigate the impact of RE configurations on MVE performance. We design a performance evaluation framework for evaluating this performance impact. Specifically, we construct a set of requirements for such a framework and outline a design to address the requirements. We then implement a prototype performance evaluation framework using the created design. We outline the challenges for such a framework and the structure of our implementation. Subsequently, we develop a set of experiments

## 1. INTRODUCTION

---

that use our created prototype. The experiments are designed to be representative of a real-world scenario. The experiments make use of the prototype we implemented. The results are evaluated to quantify the performance impact of JVM configurations on MVE performance.

### 1.1 Problem Statement

We know that JVM configurations can significantly impact the performance of applications (8, 9, 10). However, we do not know the impact that JVM configurations have on MVEs specifically. MVEs are currently able to support at most a few hundred players per instance, which is small compared to their player base of millions of players, making research into the scalability of MVEs important. However, investigating the performance impact of JVM configurations on MVEs is challenging due to the need for a framework that allows for evaluation and the wide range of available configurations for the JVM.

### 1.2 Research Questions

**RQ1:** How to design a performance evaluation framework to evaluate JVM configuration impact on MVE performance? (Section 3)

We suspect that the JVM can have a significant impact on the performance of MVEs. Experiments are needed to confirm these suspicions. In order to conduct experiments, we require a framework. Creating a framework to evaluate and understand the impact of JVM configurations is challenging. When designing such a framework, many challenges arise; for example how to design workloads or what metrics to collect. Workloads need to be representative of real-world scenarios and meaningful. For MVEs, the workload consists of the amount of players, the player behavior, and the virtual environment. These workload components need to be considered when designing the workload, as well as how to simulate the designed workload. The selected metrics need to provide an overview of the performance impact. Furthermore, how to collect the metrics and how to automate the process.

**RQ2:** How to implement such a framework in practice? (Section 4)

Implementing the framework in practice is important to prove that the design works. Furthermore, the implemented framework can be used to evaluate the performance impact.



However, implementing such a framework is challenging; it requires combining a set of systems not designed to work together. Automating the combination of such a set of systems is non-trivial; the framework must form the glue between these systems, allowing them to work together. In this process, a range of challenges arise; for example, Installation of dependencies needs to be automated. Incompatibilities between systems must be remedied or worked around. The MVE server must automatically be set up and executed. Subsequently, bots must automatically connect to the server and simulate player behavior. During this process, the relevant metrics must be collected and stored.

**RQ3:** What is the performance impact of JVM configuration on MVEs? (Section 5)  
To the best of our knowledge, no standard benchmark exists for exploring JVM configurations and their impact on application performance. Designing a benchmark for this exploration is challenging. It poses challenges such as, but not limited to, which parameters to change, how to combine the changed parameters, which workloads to use, and which metrics to collect in the process.

## 1.3 Research Methodology

This thesis utilizes the following research methodologies:

- M1. Design and prototyping (11, 12, 13);
- M2. Designing appropriate workload-level benchmarks and quantifying a running system prototype (14, 15, 16);

## 1.4 Thesis Contributions

In line with the research question and methodology, the contributions of this thesis are as follows:

1. The design of Solarstick: a framework for evaluating the performance impact of JVM configurations on MVE performance.
2. A prototype of Solarstick, a framework for evaluating the performance impact of JVM configurations on MVE performance.
3. A set of experiments using Solarstick with workload-level benchmarks leading to novel insights.


## 1. INTRODUCTION

---

### 1.5 Plagiarism Declaration

I, Dilano Emanuel Jermaine Doelwijt, declare that this thesis is my original work, has not been copied from any other source, and has not been submitted elsewhere for assessment. All sources used in the writing process have been properly cited and acknowledged. I understand that plagiarism is a serious offense and the importance of upholding academic integrity.

Dilano Emanuel Jermaine Doelwijt

A handwritten signature in black ink, consisting of several overlapping loops and a vertical stroke on the left side.

August 20, 2024

## 2

# MVE and Java Architecture

In this section, we describe the following: the operational model of MVEs running on the Java platform, the operational model of the Java platform, and JVM configurations. Important to note is that this section only covers the components and processes relevant to this thesis.

## 2.1 MVE model

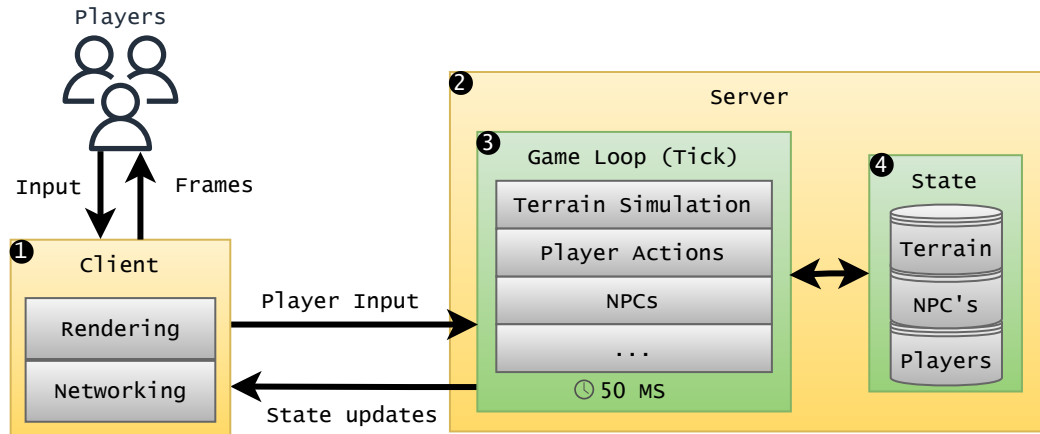
Modifiable Virtual Environments (MVEs) are games that feature a virtual environment that players can modify. The most popular example is Minecraft, attracting millions of players monthly (2). In this thesis, we focus on Minecraft.

The operation of MVEs is modeled in Figure 2.1. This model has been created based on the model proposed in Meterstick (17). The architecture is divided into parts: server (②) and client (①). Players run the client on their device; they can connect to a server using the client. The server is hosted centrally by some party and accepts connections from clients.

The client (①) has two responsibilities: Visualizing the virtual environment and translating player input into in-game actions. The client applies these actions to the local state and sends the performed actions to the server. If the server state differs from the local state, the client changes its state to match that of the server.

The server (②) manages the state (④) of the MVE and runs the game loop (③). The server is responsible for updating, simulating, and distributing the state of the virtual environment. The server receives the performed player actions from the connected clients and updates the state based on the received actions. The server distributes the state to the connected clients.

## 2. MVE AND JAVA ARCHITECTURE



**Figure 2.1:** MVE Model based on the model proposed in Meterstick (17).

The server simulates the state in a loop; the game loop (3), each iteration of this loop is called a tick. During the game loop, the terrain, player actions, and NPC actions are simulated, and the state is updated accordingly. The game loop is run at a fixed frequency. If a tick finishes early, the server waits until the time for the next tick. This frequency is typically set to 20 Hz or 50 ms per tick (17).

Terrain Simulation simulates all features of the terrain that change without direct action from the player (e.g., flowing water, falling blocks); these features are called dynamic terrain features. The server is responsible for simulating and tracking the dynamic terrain features. The load that dynamic terrain features impose on the server depends on the state of the virtual environment (4).

Clients locally track the state of the MVE, and this state is validated against the state received from the server. The server falls behind schedule when the tick frequency drops below the 20 Hz threshold. This can cause the state tracked by the server to be behind the local state of clients. However, clients adapt the state they receive from the server. This can lead to players seeing the environment revert to a previous state. The negative effects are not limited to just the reverts; input latency and unresponsiveness are other possible consequences.

The virtual environment in Minecraft is procedurally generated on the fly. The generation is based on a seed assigned to the world when it is created. Terrain is generated when it is deemed necessary based on player location and the configured render distance. When objects in the game world are not in proximity to players, the MVE may decide that object is not relevant to the execution and choose to store the object on disk instead of in memory.

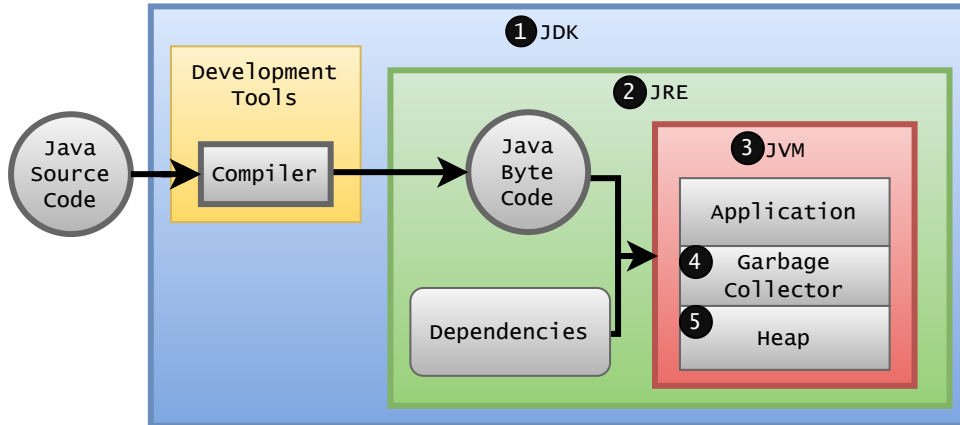


Figure 2.2: Java program architecture.

## 2.2 Java Architecture

In Figure 2.2, we model Java architecture it features 3 main components: the Java Development Kit (①), the Java Runtime Environment (②), and the Java Virtual Machine (③). The Java Development Kit (JDK) (①) provides tools for the development of Java programs (e.g., compilers and debuggers). The JDK includes a JRE (②) and a JVM (③), which are used for running programs. Java source code is compiled into Java Byte Code using a compiler. Java byte code is not platform-specific; it does not contain platform-specific optimizations. The JVM optimizes for the specific platform during execution. Because the byte code is not platform-specific, it can be executed on any platform with a working JRE implementation. Optimizing during execution allows for adaptive optimization by the Just in Time (JIT) compiler; more about the JIT can be found in Section 2.3. The Java Runtime Environment (JRE) (②) runs on the host platform and provides all files and programs necessary to execute Java byte code. The JVM is instantiated by the JRE and uses the provided dependencies. The Java Virtual Machine (JVM) (③) executes the Java byte code using the dependencies provided by the JRE. Further explanation of the JVM can be found in Section 2.3.

## 2.3 JVM Configuration

The JVM can be configured based on a range of options. The configuration of the JVM changes the working of the different components and affects the performance of the application executed using the JVM (8, 9, 18). We cover the following options for JVM

## 2. MVE AND JAVA ARCHITECTURE

---

configuration: GC implementation, Heap size, JVM implementation, and more. This section gives a brief overview of the JVM and how these configurations work.

The Java Virtual Machine (JVM) executes the Java byte code using the dependencies provided by the JRE. The JVM keeps track of the program's state, manages memory, optimizes the code for the host platform, and more. In this section, we outline the features relevant to this thesis.

During execution, objects are created, and these created objects live in a section of memory called the heap. The heap is only used for the created objects. The size of the memory section reserved for the heap can be increased or decreased during a program's execution.

When an object on the heap is no longer reachable from the current state of the JVM, the object becomes garbage. The garbage created during a program's execution is cleaned up by a process within the JVM called the garbage collector. The garbage collector is responsible for finding and eliminating garbage on the heap.

During execution, most code is interpreted into machine code on the fly. However, the Just In Time (JIT) compiler identifies sections of Java byte code that are frequently used and optimizes them. Optimization is done by translating Java byte code directly to the machine code for the host platform and storing this in the JVM for future use; the next time this section of Java byte code is called, the stored machine code will be used directly.

### 2.3.1 JVM implementation

The JVM does not have a default implementation. A wide range of implementations are available, and each implementation implements the components of the JVM differently. However, OpenJDK (19) serves as an open-source reference implementation. There is no strict set of rules; however, any implementation must support the execution of Java byte code.

In this thesis, we take a look at two implementations of the JVM: OpenJDK (19) and GraalVM Community edition (GraalVM CE) (20). GraalVM CE is based upon OpenJDK and uses mainly the same components but introduces its own JIT compiler with new optimizations.

### 2.3.2 Heap

The heap is a memory section allocated by the JVM for dynamically created objects. The heap offers a range of options; we will briefly describe relevant options and their effects.

The heap resizes during executions as required. The upper and lower bounds can be configured; if the lower and upper bounds are set to the same value, resizing does not happen. The heap size starts at the lower bound.

The size of the heap impacts the performance of the Garbage Collector. When the heap is smaller, there are fewer bytes for the garbage collector to keep track of. This leads to a lighter workload for the garbage collector. However, a small heap can cause an increase in the demand for garbage collection, which can stack up and outweigh the benefits. The opposite holds; a larger heap leads to a higher workload for the garbage collector but can decrease the demand for garbage collection. When deciding what settings to use for the heap size, a trade-off must be made between frequency and duration of garbage collection. Additionally, the chosen heap size must meet the application's memory requirements. Otherwise, the execution may be stopped due to a lack of memory. This creates a difficult problem with considerable trade-offs.

When the heap is resized, the JVM tries to allocate more than is needed in order to have memory in reserve. This is done to avoid stalls due to a lack of memory where the program has to wait for memory to be allocated. The amount of reserve memory can be configured. A larger amount of reserve memory increases the size of the heap, increasing the number of bytes the garbage collector needs to keep track of; a smaller amount of reserve memory can increase the number of stalls and the demand for garbage collection.

### 2.3.3 Garbage Collector

The Garbage collector is the part of the JVM responsible for cleaning up garbage on the heap. There are multiple implementations of the garbage collector. The default garbage collector for OpenJDK 22 is G1GC, another popular implementation is ZGC. The used garbage collector is configurable, the available options depend on the JVM implementation. The approaches of G1GC and ZGC are outlined in the following paragraphs by summarizing differences and then exploring the approach of the garbage collectors.

G1GC and ZGC utilize the same method to clean up garbage; non-garbage objects are copied to an empty memory section, and the old section is freed. Key differences between G1GC and ZGC lie in concurrency and the heap's structure. When G1GC cleans up garbage, all noncritical processes on the JVM are paused. ZGC cleans up garbage while the processes are running. G1GC divides the heap into regions based on how long an object has existed, and ZGC divides the heap into sections without any distinction in age.

G1GC is concurrent and runs in parallel, meaning it does work during the execution of the application and on multiple threads. G1 divides the heap into multiple virtual sections:

## 2. MVE AND JAVA ARCHITECTURE

---

the young generations and the old generations. The young generations contain newly created objects. When an object has existed for a set amount of time, it is first transferred internally in the young generations and eventually, after more time, is transferred to the old generations. The size of these generations, at what age objects should be moved, and how many objects should be moved each step are configurable.

G1GC focuses on the young generation, specifically sections with a lot of garbage since this is the most efficient. When needed, G1GC incrementally cleans up the old generations. The timing and conditions for garbage collection are configurable. G1GC finds garbage mostly concurrently, but the garbage collection happens during a pause. This pause temporarily halts the non-critical processes running on the JVM. G1GC attempts to minimize the duration of the pause. The (desired) maximum pause time can be configured. G1 will try to achieve the set maximum pause time, but it is not guaranteed.

ZGC divides the heap into many regions according to different size classes, the size of which can be configured. It runs mostly concurrently, only pausing to synchronize. ZGC marks and cleans up garbage concurrently. ZGC uses metadata outside the memory regions occupied by objects to ensure a synchronized state of the heap between application threads and the GC threads. The specifics of this system are outside the scope of this thesis. Knowing that the system exists is sufficient. ZGC relocates objects concurrently by updating the location stored in the metadata. Because relocating is done concurrently, pauses can be short.



## 3

# Design of Solarstick: A Performance Evaluation Framework for JVM Configuration Impact on MVEs

In this chapter, we propose Solarstick a design for: A performance evaluation framework for JVM configuration impact on MVEs. Through this design, we address research question 1: How to design a performance evaluation framework to evaluate JVM configuration impact on MVE performance?

### 3.1 Solarstick Requirements

To address the research questions declared in Section 1.2, Solarstick must meet several criteria. These criteria have been formulated into the following requirements:

- RE1. **Representative workloads:** The framework must support workloads representative of real-world scenarios in terms of the virtual environment and bot behavior.
- RE2. **Representative results:** The framework has to ensure that results are representative of real-world deployments.
- RE3. **Relevant measurement:** The framework must allow for measuring of metrics relevant to the performance of the JVM and MVE.
- RE4. **Adaptable:** Framework must be adaptable to different experiments, e.g., an experiment focusing on the GC or an experiment focusing on the JIT by allowing the JVM configuration and measured metrics to be adapted to experiment requirements.

### 3. DESIGN OF SOLARSTICK

---

Requirements RE1, RE3, and RE4 are covered in Section 3.2. RE2 is covered in the implementation, more specifically Section 4.1.

## 3.2 Solarstick Design Overview

We define a process to conduct an experiment using Solarstick, the workload used as input to the SUT, and how and what to monitor about the SUT (How in this section, what in Section 3.4).

The workflow of Solarstick addresses the following requirements RE1, RE3, and RE4. The Performance of the MVE is evaluated using a configuration determined by the user, forming an experiment.

Solarstick configures the JVM, the used configuration is configured by the user before the experiment. The user is able to select exactly which JVM options to use. Solarstick applies a workload to the SUT; this workload is configured before the experiment. The configuration of the MVE (i.e., render distance or virtual environment) and the bots. The virtual environment can be chosen from provided worlds, or the user can provide their own world (RE4). The bot amount can be configured, and the behavior of the bots can be chosen from provided bot scripts, or users can provide their own bots script (RE1 and RE4). Discussion surrounding the workloads included in Solarstick and in what way they are representative is found in Section 4.2.

Solarstick collects metrics from the entire SUT, meaning JVM (i.e., Heap size), MVE (Only tick time), and system metrics (i.e., RAM usage) are available to the user (RE3). Which metrics are tracked can be configured by the user before the experiment (RE4). Metrics can be selected from the set of metrics supported by our metric scraper, which is covered in Section 4.1.3. However, tick time is always tracked and processed into a plot. After an experiment, the raw data and plots are made available to the user.

In Figure 3.1, we give an overview of the architecture of Solarstick. Solarstick features 3 core components: The director (❶), the bot simulator (❷), and the System Under Test (SUT) (❸). This represents a real-world deployment by separating the client (bot simulator), server (SUT), and our director from each other (RE2).

The director (❶) is the main part of our framework; it directs the processes during the experiments. It downloads, installs, configures, and runs the other components. When an experiment has finished, metrics stored on the SUT are processed by the director using the metric processor. The produced results are made available to the user.

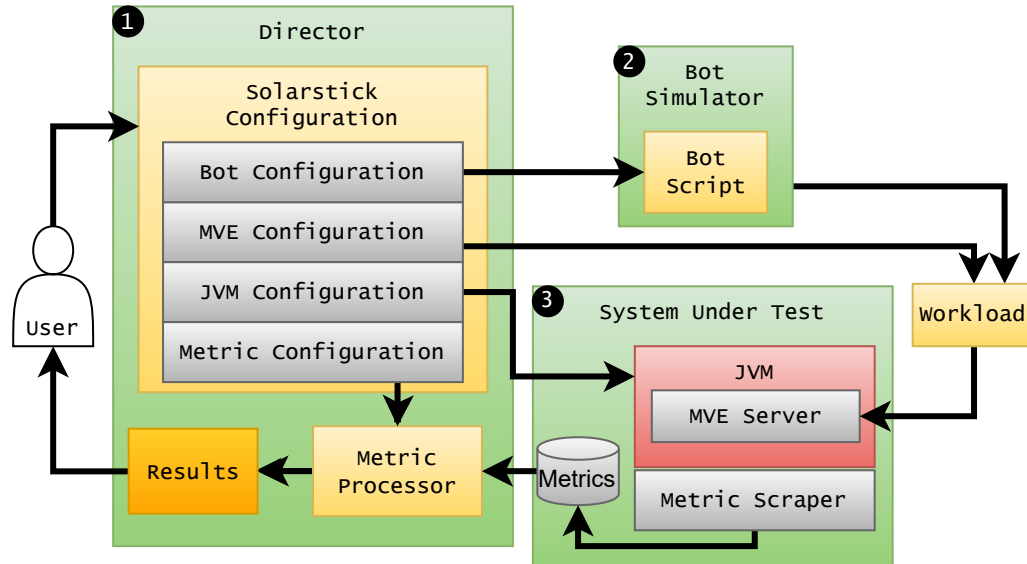


Figure 3.1: Solarstick design architecture.

The bot simulator (②) represents the client in the MVE operational model. It runs the bot script that is configured by the user. The script will simulate bots that connect to the MVE server on the SUT (③).

The SUT (③) represents the server in the MVE operational model. It runs the JVM which in turn executes the MVE server that the simulated bots connect to. The metrics scraper scrapes metrics from the JVM, MVE server, and the system; these metrics are stored on the SUT.

### 3.3 Workloads

To address RE2 Representative results, we design 2 bot behaviors that can be used during experiments; one workload that is meant to stress the SUT and a workload to test maximum player count or facilitate environmental workloads. The implementation details of these workloads are outlined in Section 4.2.

The first workload features bots flying through the world of the MVE. This stresses most parts of the MVE, forcing it to simulate all players constantly and load and unload the virtual environment and NPCs in the environment. Exploring is a common part of MVE gameplay and is very representative of a real-world scenario with this behavior we aim to simulate this behavior. Furthermore, the workload stresses the SUT since it combines multiple elements, such as player simulation and terrain simulation.

### 3. DESIGN OF SOLARSTICK

---

The second workload features bots that are mostly idle, and are not applying any significant load to the SUT. This workload is meant to allow the workload to come from elements other than bot behavior, for example, from the player count or dynamic terrain features. We do not use a fully idle behavior since we want the players to still be part of the workload just not the focus. When a player is fully idle the MVE may be able to optimize the simulation of the player, which is unwanted. We want the findings to not only be applicable to idle behaviors but to other low-impact behaviors as well. Hence, we opt for a mostly idle behavior with some repeated action; this aims to be representative of a real-world situation where players are carrying out simple tasks or are socializing.

#### 3.4 Metrics

In this section, we propose and describe metrics that give us a performance baseline, contributing to our main research question RQ3: What is the performance impact of JVM configurations on MVEs? We select components relevant to the main research question RQ3, and propose a metric for each (RE3):

Tick time helps assess the performance of the MVE server. The tick time shows whether the server completes the game loop on time. Additionally, tick time gives us insights into the player experience. Tick time needs to be retrieved from the MVE server directly; this requires the use of a protocol that is able to communicate with the MVE server to retrieve this data. Each second should contain 20 ticks; taking the average of the 20 ticks gives us a good overview of how the MVE server performed during that second. High granularity is required; tick time allows us to assess how the server is performing, and patterns in the tick time can be coupled back to JVM or workload details. This can give more insights into the performance of the MVE. However, the time of each tick is not necessary; the average tick time per second is sufficient to see such patterns and assess the performance of the MVE. When no pattern is observed tick time can still be used to evaluate the performance of the MVE and whether the MVE is completing ticks on time.

CPU time per second helps assess the load the SUT is experiencing. This can help in discovering configurations that increase the scalability of the MVE by lowering the burden on the SUT. Additionally, we can investigate garbage collection by investigating the effect garbage collection has on CPU usage. CPU usage is retrieved directly from the host platform, and there are many ways to retrieve it. High granularity is required to observe patterns in the CPU usage that are created by the garbage collection or workload

details. These patterns are interesting because they give us insight into the impact the GC configuration has on the JVM.

Heap Memory usage gives us insight into the load the SUT is experiencing, as well as the behavior of the garbage collector; when garbage is collected, memory usage will go down. The patterns formed in the memory usage by garbage collection can give us insight into the garbage collection's frequency and the size of each collection. Additionally, configurations that increase the scalability of the MVE due to lower memory usage can be identified. The heap memory usage must be retrieved from the JVM; this can be achieved with the Java Management Extensions (JMX) (21). High granularity is required to observe the patterns the garbage collector creates in the heap memory usage.

Garbage collector time per second gives us insight into how much time is used for garbage collection each second. This can be combined with the other metrics to evaluate the garbage collector's performance; for example, CPU usage and Heap memory usage can be compared to the garbage collector time per second to identify the pattern of garbage collection and evaluate the impact of garbage collection on those metrics. Garbage collector time per second must be retrieved from the JVM: This can be achieved with the Java Management Extensions (JMX) (21). High granularity is required to evaluate the behavior of the garbage collector from the patterns in this metric.

In this chapter we have addressed research question 1: How to design a performance evaluation framework to evaluate JVM configuration impact on MVE performance? By going through the design process and documenting the requirements and specifics of such a design. Furthermore, our creating the design helps address research question 2: How to implement such a framework in practice? and research question 3: What is the performance impact of JVM configuration on MVEs? By providing a design that can be used in the process of addressing these research questions.

### 3. DESIGN OF SOLARSTICK

---

## 4

# Implementation of Solarstick

In this chapter, we propose a prototype of Solarstick as designed in 3, our prototype builds upon components of Yardstick (6) and the Opencraft tutorial (22). This chapter addresses research question 2: How to implement such a framework in practice? by actually implementing a prototype of such a framework and reporting on our process and findings. Additionally, this section helps us address research question 3: What is the performance impact of JVM configuration on MVEs? By providing a framework that can be used to evaluate the impact of JVM configurations on MVEs.

We outline the architecture and workflow of this prototype. The prototype runs on dedicated compute nodes that can be categorized into three categories: the Directing Node (❶), the Client Node(s) (❷), and the Server Node (❸). We discuss each and describe the processes running on them.

## 4.1 Solarstick Implementation Overview

The experiment is started by running a shell script on the directing node. This script sets up a Python virtual environment (❹) using Miniconda; Miniconda is installed if not present. Miniconda is a minimum installation of Anaconda (23, 24). Anaconda is open-source software that can manage your Python- packages and environments. Miniconda creates a Python virtual environment containing the dependencies for the directing node and client node(s).

### 4.1.1 Directing Node

The experiment is started on the Directing Node (❶), which contains all the information about the experiment, and a Python virtual environment (❹). The node has 3 core parts:

#### 4. IMPLEMENTATION OF SOLARSTICK

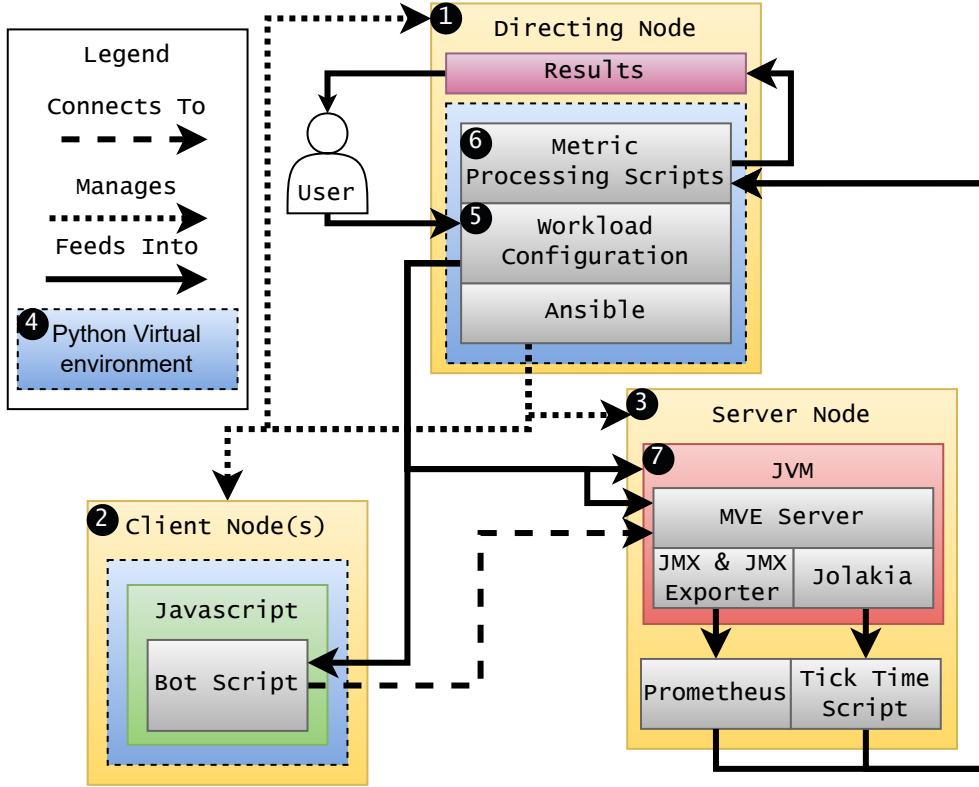


Figure 4.1: Solarstick implementation.

Ansible, Workload Configuration (5), The Metric Processing Script (6). In this section, we describe the setup and the involvement of these parts.

Ansible is an open-source automation engine; We heavily modify the Ansible script from the Opencraft tutorial (22) and use it to configure and direct the processes running on the nodes used during experiments. The main Ansible process runs in the Python virtual environment on the directing node (25). The Ansible process reserves nodes using the reservation system on the DAS-5 (26). The reserved nodes are configured and become the server node and client node(s).

The Workload Configuration (5) encompasses all the configuration options for the experiment: The JVM configuration, bot behavior, bot amount, the MVE settings, and the environment used for the MVE. It features multiple files, but the file that brings it all together is the experiment configuration YAML file.

The experiment configuration file allows configurations of the following: iteration amount and duration of each iteration. The Java command used to execute the server JAR; The command line options for the used JVM implementation can be used. The Bot behavior can



## 4.1 Solarstick Implementation Overview

---

be chosen; available bot behaviors are outlined in section 4.2. Two settings can configure the number of bots: the number of client nodes and the number of bots per client node. The MVE world used during the experiment can be configured; we outline included worlds in section 4.2.

However, two more files change the configuration of the experiment. Firstly, the "server.properties" file specifies the settings used by the MVE according to the standard for Minecraft servers (27). This allows for changing MVE settings(e.g., render distance or maximum allowed tick time). Lastly, the metric names JSON file that specifies what metrics to capture and how to label the graphs produced by the metric processing script. This supports any metrics exported by the Node exporter (28) and the JMX exporter (29). The range of metrics covered by the Node exporter and JMX exporter allows for relevant measurement; these 2 exporters export application/JVM metrics (JMX exporter) and system-level metrics (Node exporter) together, and these fulfill RE3 Relevant measurement. The Metric processing script (6) is written in Python and executes in the Python virtual environment. It is executed when the experiment has concluded. It calls to the HTTP API exposed by Prometheus. It calls the API for each metric specified in the metric names file. The received metrics are stored in a JSON file, which is later used by another script to generate plots using the information specified in the metric names file.

### 4.1.2 Client Node(s)

The Client Node(s) (2) run the bot simulation script in a Python virtual environment (4). The workload configuration on the directing node configures the bot simulation. The bots connect to the MVE running on the server node. In this section, we give an overview of the client node.

We use Mineflayer (30), an open-source Javascript library for bot simulation. The library exposes a JS API that allows for creating bots and configuring complex bot behaviors. The Python virtual environment present on the client node includes NodeJS. We use the package manager included in NodeJS to install Mineflayer. The workload configuration (5) determines the executed bot script. Details and implementation of the behaviors are discussed in section 4.2.

### 4.1.3 Server Node

The Server node (3) runs the JVM (7), which runs the MVE server along with monitoring and metric collection tools. In this section, we outline how the server node works.

## 4. IMPLEMENTATION OF SOLARSTICK

---

The JVM (7) executes the JAR for the MVE server with the command-line options specified in the workload configuration (5), along with command-line options to run the metric scraping services. The metric scraping services are Java management Extensions(JMX) (21), JMX Exporter (29), and Jolokia (31).

Java Management Extensions exposes metrics regarding the execution of the JVM. The application running on the JVM can supply metrics to JMX. The MVE exposes its tick time in the form of an array. JMX exporter and Jolokia listen on the designated port for the metrics. The JMX exporter transforms the data into a format accepted by Prometheus. We use Jolokia to extract the tick time exposed by JMX. Jolokia allows us to use HTTP requests to retrieve the data. We use Jolokia because JMX exporter does not support the format in which the tick time is exposed.

We use a modified version of the Python script from the yardstick project (6) to calculate and pull tick time from Jolokia. The script uses the HTTP API offered by Jolokia to retrieve the tick times. The script attempts to retrieve this data once every 2.5 seconds. Since the array exposed by the MVE contains the ticks from the last 5 seconds. This way, the script can track which ticks are new and which are old. We modify the script to be more reliable and to write results to a file.

We run Prometheus on the server node; Prometheus is an open-source monitoring toolkit (32); it exposes an HTTP API that allows for structured querying using their language for queries called PromQL. Once an experiment has concluded, we run a Python script that queries the HTTP API for each metric specified in the workload configuration (5)

The MVE server uses the environment and settings specified in the workload configuration. The server node runs in a temporary directory; we do this so that the state is not kept between the experiments' runs. The MVE server and the JVM are given 2 minutes to start up and stabilize. Following this, we schedule a 90-second period for the bots to join and the server to stabilize.

### 4.2 Workloads

In this framework, we include two bot behavior scripts, but more can be implemented and used by the framework. The jump behavior features bots that jump up and down constantly. The explore behavior features bots that fly up to get clear of any terrain and fly in a set direction. Additionally, we supply a world from the MVE community available

for download on the internet (33). In this section, we outline the included world and the behaviors.

We propose a core structure for our bot behavior scripts. We implement a 'Bot' class to represent each of the bots; this class is a wrapper for the Mineflayer Bot class. We create instances of this class using a loop. The command line arguments determine the number of bots and the name of each bot. The script expects the following command line arguments:

1. Internal IP: represents the MVE server.
2. Node identifier: Specifies the node the script is executed on. Used in the naming of the bots for clarity.
3. Bot amount: Specifies the amount of bots to be simulated.

The values for 1 and 2 are determined during the execution of the script and are based on the used nodes. Number 3 is specified in the workload configuration (5).

The bots join over a span of 60 seconds. The interval between each bot joining is calculated by dividing 60 by the total number of bots. We utilize this spread-out approach to reduce the load on the server and avoid crashes. Once a bot joins, it executes a function within the 'Bot' class that outlines its behavior.

We implement the following two behaviors:

- Jump: Bots repeatedly jump in place. We achieve this behavior by turning on the *jump* control state in the Mineflayer API.
- Explore: Bots fly upwards and then fly in a straight line toward a single direction. We achieve this behavior by turning on one of the following control states in the Mineflayer API: *forward*, *back*, *left*, *right*. We decide which control state to turn on for each bot based on their sequence number. We create an array with these four options and select the direction using the sequence number modulo 4.

The jump behavior is relatively simple and has a light load on the server. The bots repeat the same simple action. This behavior is designed for experiments with environmental workloads or for testing the maximum amount of bots the server can handle concurrently. We choose this behavior over idle behavior to ensure the server has to consider these bots in the game loop. This approach aims to provide a more representative behavior. More on why we opted for these behaviors is described in Section 4.2

The explore behavior is relatively heavy. The bots are spread out because they are evenly divided over the four directions. Additionally, since the bots join at intervals, the

## 4. IMPLEMENTATION OF SOLARSTICK

---



**Figure 4.2:** Hillside Manor the world included with Solarstick

bots flying in each direction are spread out. This forces the server to constantly generate, load, and send parts of the virtual environment. These sections of the virtual environment then quickly become non-relevant because the bots keep moving. This is a Representative scenario, which is common since traveling and exploration are key parts of MVEs. This behavior can be used in experiments that require or desire representative bot behavior.

Figure 4.2 features in-game screenshots of the world we include. Hillside Manor: A community-made map, with buildings surrounding the start area (33). In this section, we discuss the choice to include this world.

The Hillside Manor world features a collection of buildings surrounding the start area. To the best of our knowledge, there is no public data on what an average Minecraft world is. Therefore, we opt for a popular community map, which we perceive to be medium-scale, to replicate an average Minecraft world. Hillside Manor is available for download on the website Planet Minecraft (33), Where the world has garnered more than 700,000 downloads.

In this chapter, we have addressed research question 2: How to implement such a framework in practice? and provided a framework for addressing research question 3: What is the performance impact of JVM configuration on MVEs? We have achieved this by implementing and documenting the implementation of a framework for evaluating the performance impact of JVM configurations on MVE performance.

# 5

## What Is the Performance Impact of JVM Configuration on MVEs?

### 5.1 Experimental Setup

In this chapter, we provide an outline of the experiments we conducted and report on their results. The experiments and their results aim to address research question 3: What is the performance impact of JVM configuration on MVEs? By showing the impact that JVM configurations have on MVEs using experiments. Important to note is that the explanation we give of the observed differences in this chapter is speculative based on the observed measurements. Additional experiments are required to confirm the speculations.

The conducted experiments have led to 4 main findings:

- MF1. Tested JVM implementations do not have a significant impact on MVE performance (Section 5.2). We find that for the tested implementations, the choice of JVM implementation does not significantly impact the performance of the MVE.
- MF2. GC implementation can have significant impact on performance (Section 5.3). We find that the choice of GC implementation can lead to a 390% increase in the max-

**Table 5.1:** Experiments.

Focus	Section	Parameter(s) Changed
JVM implementation	Section 5.2	JVM
GC implementation	Section 5.3	GC, Max Heap Size
Heap Size	Section 5.4	Max Heap Size
Community Configurations	Section 5.5	Various

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---

**Table 5.2:** Experiment configuration. Values in bold are the default for our experiments. Acronyms: W-Workload, C-Configuration.

Parameter	Type	Value
World	W	<b>Hillside Manor</b>
Bots	W	<b>24</b>
Bot Behavior	W	Jump, <b>Explore</b>
JVM implementation	C	<b>OpenJDK</b> , GraalVM CE
GC implementation	C	<b>G1</b> , ZGC
Min Heap Size	C	<b>2 GiB</b>
Max Heap Size	C	<b>Small: 4 GiB</b> , High: 32 GiB
Duration	C	<b>120 seconds</b>
Startup time	C	<b>210 seconds</b>
Iterations	C	1, <b>3</b>

**Table 5.3:** Brief overview of the hardware running the nodes used in the experiments.

Component	Specification
OS	CentOS Linux version 7.9
Memory	64 GB
CPU	x86_64 Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz 32 Core

imum tick time, when comparing G1GC to ZGC using a low maximum heap size. Additionally, when using a high maximum heap size, ZGC can lead to a 269% increase in maximum heap memory usage.

MF3. A higher maximum heap size can positively impact CPU load (Section 5.4). We observe a decrease of 8% in maximum CPU usage and a 22% decrease in minimum CPU usage when comparing a low maximum heap size to a high maximum heap size.

MF4. JVM configurations developed for MVEs can positively impact MVE performance (Section 5.5). We find that such a configuration can lead to an 8% decrease in maximum CPU usage and a 37% decrease in maximum heap usage when compared to an out-of-the-box configuration.

We performed these tests on the DAS-5, a medium-scale distributed supercomputer for academic and educational use (26). It offers dedicated compute nodes; an overview of the hardware used to run these nodes is given in Table 5.3. The Java process running the MVE is pinned to the first four cores during the experiments.

## 5.2 MF1: OpenJDK and GraalVM CE Do Not Show Significant Differences

During experiments, we use the official Minecraft distribution called "Vanilla" Minecraft (34) version 1.20. We select this distribution due to its popularity and its official status. We use version 1.20 because it is the most recent major release supported by the used bot simulation tool; Mineflayer (30, 35).

As was discussed in section 2.1, the load on the server is a combination of player amount, player behavior, and the virtual environment. During the experiments, we use a fixed amount of bots; this number is set at twenty-four. This was decided based on the twenty-five used in Meterstick (17). Twenty-four bots were chosen instead of twenty-five to allow the experiment to be conducted with two bot simulation nodes, each simulating 12 bots.

The bot behavior in use during experiments is the explore behavior. The simulated bots fly upwards for a set amount of blocks and decide the direction to fly toward based on their bot ID, such that in each iteration, the same ID will choose the same direction. The bots indefinitely fly in the selected direction. After connection, bots start their behavior immediately. However, bots do not join at the same time, causing the bots to be spread out rather than cluttered together. This behavior was chosen because of its reproducibility and representative server workload. More about this behavior can be read in chapter 4.

The experiments we conduct are featured in table 5.1. For most of our experiments, we focus on changing one -in some cases, 2- parameters per experiment to give a broad overview of the possibilities and isolate each parameter's impact. Additionally, we conduct experiments comparing out-of-the-box performance against JVM configurations recommended by the MVE community. We do this to evaluate the potential impact of a JVM configuration developed for this application. We outline the experiment parameters in table 5.2.

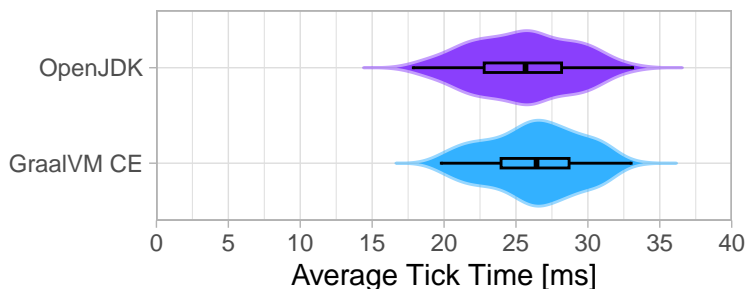
We choose a world for each experiment to improve repeatability and achieve a representative workload. If we do not do this, each run will get a different procedurally generated world based on a random seed. Descriptions and images of used worlds can be found in section 4.2

## **5.2 MF1: OpenJDK and GraalVM CE Do Not show Significant Differences in MVE Performance Under Tested Workload**

This experiment aims to find the impact that the chosen JVM implementation can have on the performance of the MVE. Since there are no strict guidelines for implementing the JVM, and JVM's have shown to have differences (36), it is interesting for us to see how these

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---



**Figure 5.1:** Violin plot of 2.5-second sliding window average tick time in ms for the graalVM CE and Open JDK over three iterations of 120 seconds.

different implementations impact the performance of an MVE. The difference between JVM implementations which will be most relevant for us here is the JIT compiler. GraalVM CE is based on the OpenJDK implementation but replaces the JIT c2 compiler with its own Graal compiler. Oracle has developed the Graal Compiler in Java and introduces optimizations such as advanced inlining, partial escape analysis, code duplication, and speculative optimizations (37, 38). Specifics regarding the optimizations are not within the scope of this thesis and will not be covered.

In Figure 5.1, we compare the 2.5-second sliding window average tick time between the GraalVM CE and Open JDK Hotspot JVM across three iterations of 120 seconds. In the figure, it is visible that JVM implementation does not significantly impact the performance of the MVE. The median of the tick time for the OpenJDK is 25.66 ms and for Graal, 26.43 ms. The spread is also similar; OpenJDK has a minimum of 17.86 ms and a maximum of 33.11 ms, whereas GraalVM has a minimum of 19.81 ms and a maximum of 33.01 ms. Additionally, there are no notable differences in garbage collection, CPU usage, or heap usage.

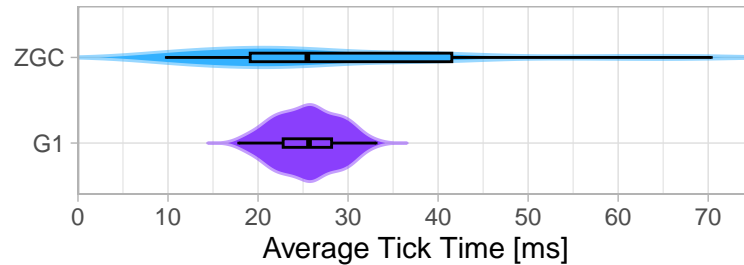
We speculate that the lack of difference is due to the main improvements of Graal and the nature of an MVE. The GraalVM CE replaces the last optimization step in the JIT -C2 in OpenJDK- with its own Graal Compiler. We speculate the lack of improvement is due to the improvements the Graal compiler introduces not being relevant or effective in this use case.

We can conclude that the choice between GraalVM CE and Open JDK does not have a significant impact on the performance of Minecraft using workload, which we argue to be representative.

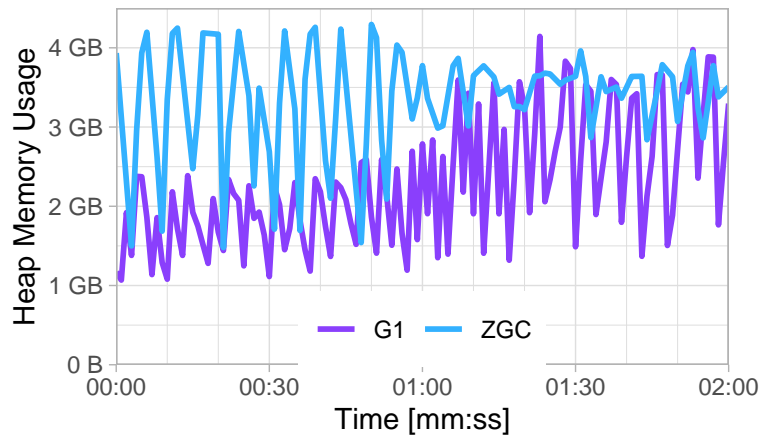


### 5.3 MF2: GC Implementation Can Have Significant Impact on Performance

---



**Figure 5.2:** Violin plot of 2.5-second sliding window average tick time in ms for G1GC and ZGC over three iterations of 120 seconds



**Figure 5.3:** Line plot of heap memory usage in Gigabytes for G1GC and ZGC over one iteration of 120 seconds

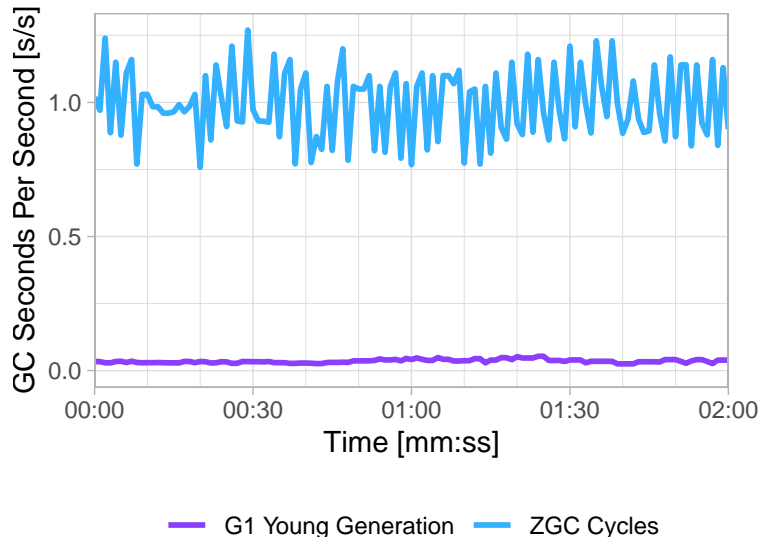
### 5.3 MF2: GC Implementation Can Have Significant Impact on Performance

This experiment aims to find the potential impact of the chosen GC implementation on the performance of the MVE. Memory management is an important part of JVM performance. The importance of memory management makes it interesting to investigate how different garbage collector implementations affect the MVE performance. These different implementations may use totally different approaches to the collection of garbage; how these affect the MVE is unknown and can lead to interesting insights.

In figure 5.2, we compare the average tick time over a 2.5-second sliding window for the G1GC and ZGC over three iterations of 120 seconds. In the graph, the medians are close 25.66 ms for G1GC and 25.48 ms for ZGC. However, the 25th percentile and 75th percentile are very different. Running G1GC resulted in an interquartile range of 5.38 ms, while runs

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---



**Figure 5.4:** Line plot of GC collection seconds per second for G1GC and ZGC, over one iteration of 120 seconds

using ZGC resulted in an interquartile range of 22.40 ms. This leads us to conclude that there is a significant difference in performance. The whiskers on the ZGC plot indicate that the 50ms threshold has been passed; passing this threshold impacts player experience negatively, as discussed in Section 2.1.

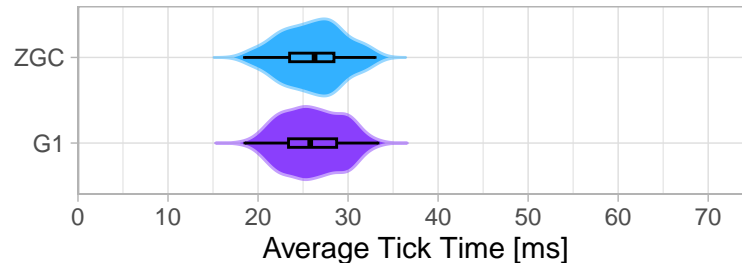
Notable is the lower minimum achieved by ZGC during our experiment. The lowest value observed for ZGC is 9.82 ms, whereas G1GC's lowest value is 17.86 ms. Furthermore, the 25th percentile for ZGC is lower than that of G1: 19ms and 23ms, respectively. However, lower minimums do not matter for player experience as long as the tick time is below 50ms.

Additionally, in figure 5.3, we can see that ZGC frequently runs into the maximum heap size of 4 GiB. Contrary to ZGC, we see G1GC staying inside the allocated heap size comfortably, with a few peaks that hit the maximum. This points to ZGC struggling with managing memory. Furthermore, the literature tells us that ZGC's approach introduces a large number of allocation stalls - especially when operated on low heap sizes - which degrade the performance of the GC. The degraded performance is most likely due to the non-generation model that ZGC runs on (39). So our low heap size may be one of the main reasons for this reduction in performance.

In figure 5.4, we can see that ZGC is spending considerably more time on garbage collection than G1GC. The G1GC old generation has been left out since no old generation garbage collection was done. This indicates that the problem is the approach used for garbage collection. The generational approach used by G1GC seems effective for our

### 5.3 MF2: GC Implementation Can Have Significant Impact on Performance

---



**Figure 5.5:** Violin plot of 2.5-second sliding window average tick time in ms for G1GC and ZGC with maximum heap size 32 GiB over three iterations of 120 seconds

experiment. This can be attributed to the chosen bot behavior. The bots are constantly moving, leading to a young generation full of garbage ready to be collected. Because G1GC does not consider the old generations unless necessary, work is being saved by not considering the old generation. ZGC concurrently monitors the entire heap and may not be sectioning managing the objects on the heap effectively for this workload.

Due to the results and the speculated reason for the differences, we ran an additional experiment using the same workload and setup but with a 32 GiB maximum heap size instead of 4 GiB.

In figure 5.5 shows a similar spread of tick times between G1GC and ZGC. This experiment is run with a maximum heap size of 32 GiB over three iterations of 120 seconds. This supports the earlier speculation regarding the reason for the performance difference between G1GC and ZGC. Both garbage collectors show a relatively stable tick time and never cross the 50 ms threshold.

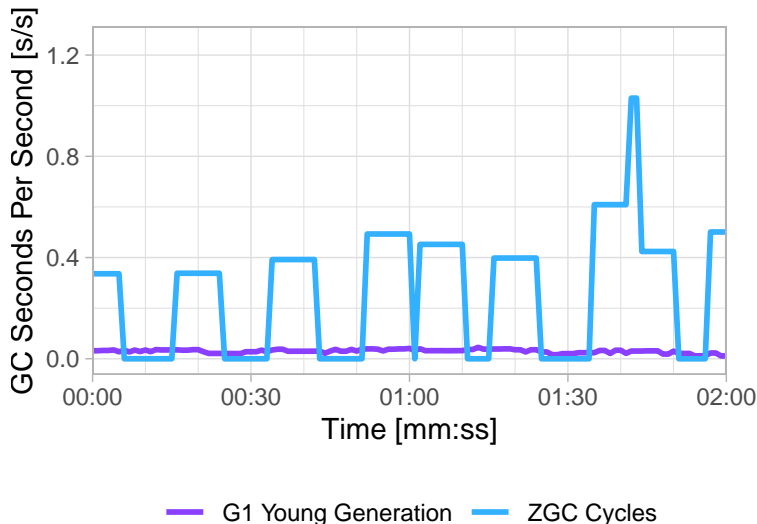
In figure 5.6, we compare the time spent in seconds on garbage collection each second between G1GC and ZGC running with 32 GiB as the maximum heap size over one iteration of 120 seconds. We see that ZGC is doing much more work each second. However, the seconds spent on GC work per second is noticeably lower than in figure 5.4. Again, leaning into our speculation regarding the heap size.

In figure 5.7, we compare the used heap memory in Gigabytes between G1GC and ZGC running with 32 GiB as the maximum heap size over one iteration of 120 seconds. We can see that while ZGC hits the limit much less than in the experiment using a low heap size, ZGC still uses much more memory than G1GC.

Table 5.4 compares the heap usage in Gigabytes for G1GC and ZGC using a maximum heap size of 32 GiB over three iterations of 120 seconds. Here, we can see the increased memory usage discussed in the previous paragraph. The heap memory usage significantly

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---



**Figure 5.6:** Line plot of GC collection seconds per second for G1GC and ZGC with a maximum heap size of 32 GiB, over one iteration of 120 seconds

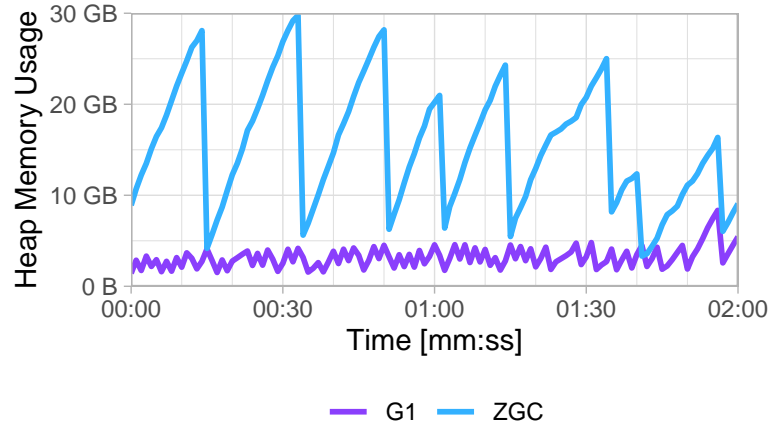
**Table 5.4:** Statistical summary of the heap usage in Gigabytes for G1GC and ZGC using a maximum heap size of 32 GiB, over three iterations of 120 seconds.

Configuration	Min	25th percentile	Median	75th percentile	Max
G1GC	1.42 GB	2.36 GB	3.70 GB	4.00 GB	8.34 GB
ZGC	3.32 GB	10.42 GB	15.18 GB	20.92 GB	20.96 GB

increases across all statistics when using ZGC. Notably, the minimum, median, and maximum increase by 134%, 310%, and 269%, respectively. However, an increase in memory usage is not directly negative, as memory only adds value when it is in use. Regardless, we argue that higher memory usage is a negative indicator in this case since the tick time is similar and the memory usage is drastically higher when comparing G1GC to ZGC. We argue this indicates a better ability to scale for G1GC. However, to confirm this, more experiments would be needed. To explain this difference, we again reference the generational model that G1GC employs and its affinity for the MVE and the workload used in this experiment.

We can conclude that the choice of GC can negatively impact the performance of the MVE. When executed with a low maximum heap size, the observed performance difference is an increase in maximum observed tick time of 390% from 33.11 ms using G1GC to 162.09 ms using ZGC. When executed with a high maximum heap size, the tick time is comparable

## 5.4 MF3: A Higher Maximum Heap Size Can Positively Impact CPU Load



**Figure 5.7:** Line plot of Heap memory usage in Gigabytes for G1GC and ZGC with maximum heap size 32 GiB, over one iteration of 120 seconds

between G1GC and ZGC; however, the median heap memory usage is 310% higher, and the maximum heap memory usage is 269% higher when executed using ZGC compared to G1GC. Leading us to conclude that ZGC should not be used in low-memory situations.

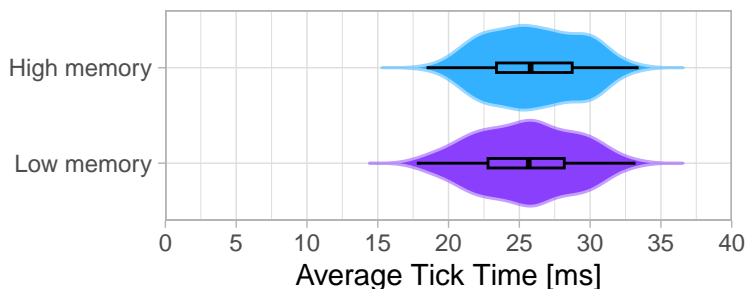
## 5.4 MF3: A Higher Maximum Heap Size Can Positively Impact CPU Load

The aim of this experiment is to evaluate the impact of the heap size on the performance of the MVE. The heap and management of the heap through garbage collection is an important part of the performance of the JVM. The size of the heap represents a trade-off between the frequency of garbage collection and the duration of garbage collection. Additionally, the maximum heap size must meet the memory requirement of the application running on the JVM. With the purpose of scaling this trade-off is interesting to look at and understand, we aim to find out how the trade-off affects the performance of the MVE. During this experiment, we vary the used maximum heap size; we use a low maximum heap size of 4 GiB and a high maximum heap size of 32 GiB. The initial heap size is fixed to 2 GiB.

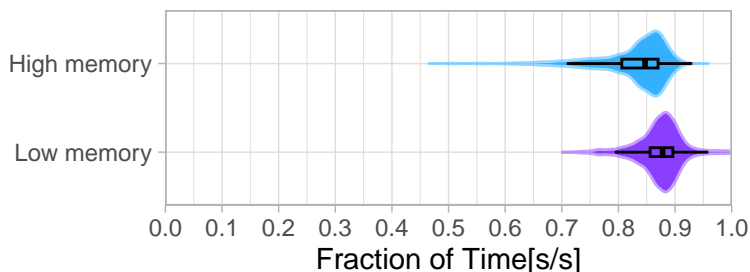
In figure 5.8, we compare the 2.5-second sliding window average tick time between the experiment using low maximum heap size and high maximum heap size across three iterations of 120 seconds. The figure shows that the used maximum heap size does not directly impact the ticking time of the MVE; we see similar median and interquartile ranges. Both have a rounded median of 26 and a rounded interquartile range of 5.

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---



**Figure 5.8:** Violin plot of 2.5-second sliding window average tick time in ms, for MVE executed with low maximum heap size and high maximum heap size over three iterations of 120 seconds.



**Figure 5.9:** Violin plot of the average CPU usage over the cores in use in seconds per second, for MVE executed with low and high maximum heap size, over three iterations of 120 seconds.

In figure 5.9, it is visible that the experiment using high maximum heap size reaches lower CPU usages than the low counterpart. The discussed CPU usage is in seconds per second. In this section, we express this usage as percentages for easier reading. The medians are similar: 85% for both. However, the minimums are quite different: 50% for the high maximum heap size and 72%, respectively, for the low maximum heap size. Additionally, the 25th percentile shows a difference: 80% and 85%, respectively. Furthermore, the maximum is also lower for the high maximum heap size run, namely: 92% against the 100% of the low maximum heap size run.

We argue that the difference can be attributed to a reduced need for garbage collection. This reduced need allows the garbage collector to avoid collecting garbage in sections of the heap with low garbage density and instead focus on collecting garbage in high garbage density sections that are cheap to collect, which in turn eases the load on the CPU.

This leads us to conclude that a higher maximum heap size can positively impact the load on the CPU, potentially allowing the MVE to be scaled further when the CPU is the limiting factor. A higher maximum heap size can lead to an 8% decrease in maximum CPU usage, a 5% decrease in the 25th percentile, and a 22% decrease in minimum CPU

## 5.5 MF4: MVE Specific JVM Configurations Can Positively Impact Performance

---

usage.

### 5.5 MF4: JVM Configurations Developed for MVEs Can Positively Impact MVE Performance

This experiment aims to find the potential impact of configurations developed for the MVE, on the performance of the MVE. These full configurations alter the working of the JVM; examples of components affected are the garbage collector, the heap, memory allocation, and multithreading. Each of these components is important to the performance of the JVM. Furthermore, configurations tuned for a specific application have shown their ability to positively impact the performance of the application in the past (8, 40, 41), making this an interesting experiment. We use configurations mentioned on the Minecraft fandom wiki (42), along with the default JVM configuration. The wiki mentions 3 configurations:

- **Micro.soft**: The default setup used by the Microsoft client to run Minecraft (42).
- **brucethemoose**: A configuration developed by community members and published under the GitHub username brucethemoose (43).
- **Aikar**: A configuration developed by a community member and published in a blog post on the creator’s personal blog under the moniker Aikar (44).

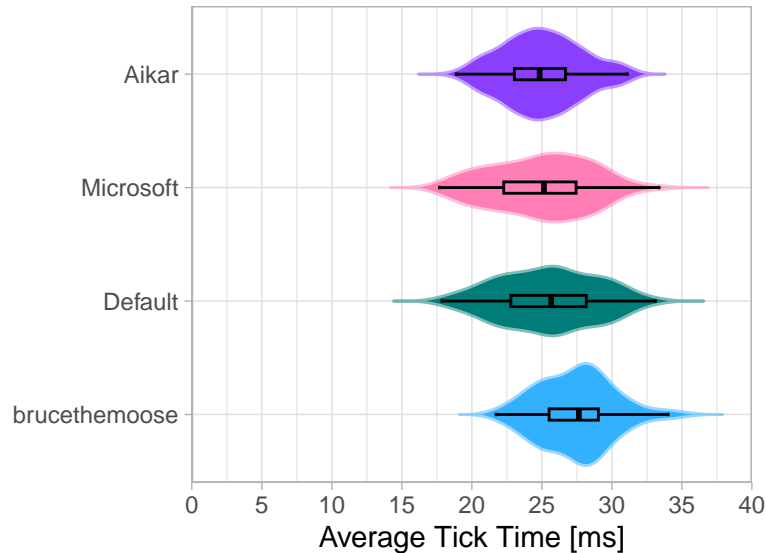
Each of these configurations in full can be found in the appendix 7.2. We will refer to the configurations as the name in bold, and the default setup will be referred to as **default**. The configuration selected by the OpenJDK on DAS-5 as default can be found in the appendix 7.2. We utilize the G1GC for each configuration along with the minimum heap size of 2GiB and a maximum of 4GiB, according to the defaults outlined in table 5.2. These configurations do give their own recommendations regarding heap sizes. However, we choose our 2 GiB minimum and 4 GiB maximum to keep the experiment results comparable.

We break this finding into three sections: tick time, CPU usage, and heap memory usage. We report our findings for each in the following paragraphs. Subsequently, we will discuss potential underlying causes for the findings.

In figure 5.10, we compare the average tick time over a 2.5-second sliding window for the Microsoft, Default, brucethemoose, and Aikar JVM configurations over three iterations of 120 seconds. We see in the figure that the utilized configuration impacts the tick time and the spread of the tick time. We observe differences between the spread of the tick

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---



**Figure 5.10:** Violin plot 2.5-second sliding window average of the tick time in ms, for the Microsoft, Default, brucethemoose and Aikar JVM configurations, over three iterations of 120 seconds.

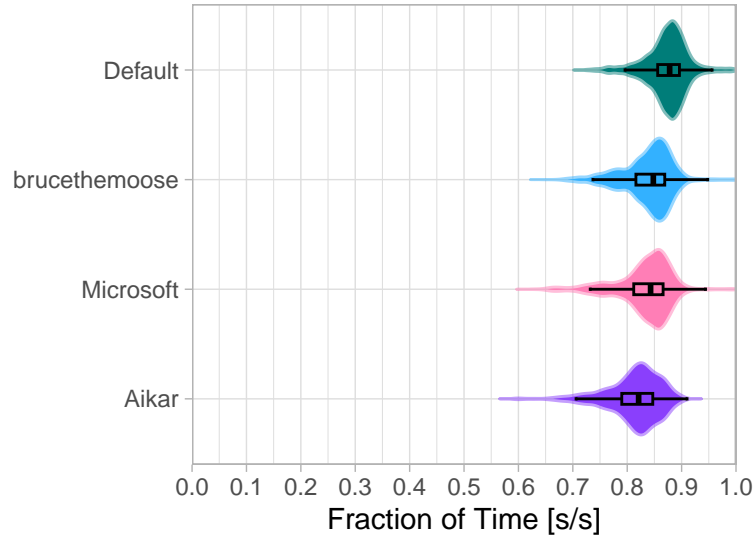
times. Notably, the interquartile range of brucethemoose and Aikar are smaller than those of the Default and Microsoft configurations. Additionally, we can observe that the tick time spread of the Default configuration is very similar to the Microsoft configuration and that the spread of the Aikar and brucethemoose configurations are similar despite the Aikar showing lower ticks. However, all tick times visible are below the 50 MS tick time threshold, although lower tick times and more stability can be argued to be positive indicators.

In figure 5.11, we compare the average CPU usage in the utilized cores in seconds per second for the Microsoft, Default, brucethemoose, and Aikar JVM configurations over three iterations of 120 seconds. In the figure, it is visible that the Default configuration averages the highest CPU usage; each of the other configurations shows an improvement over the Default, while Aikar shows the biggest improvement. The fraction of the second used by the CPU each second will be referred to in percentages in this section. The default configuration has a median of 88%, and the Aikar configuration has a median of 82%; a 6% decrease in median CPU usage. The Default configuration has a maximum usage of 100%, whereas Aikar has a maximum of 91%, showing a 9% decrease in maximum CPU usage.

In figure 5.12, we compare line plots of the heap memory usage in Gigabytes for the Microsoft, Default, brucethemoose, and Aikar JVM configurations over one iteration of



## 5.5 MF4: MVE Specific JVM Configurations Can Positively Impact Performance



**Figure 5.11:** Violin plot of the average CPU usage in the used cores in fractions of a second for the Microsoft, Default, brucethemoose, and Aikar JVM configurations over three iterations of 120 seconds.

**Table 5.5:** Statistical summary of the heap usage in Gigabytes for the Microsoft, Default, brucethemoose, and Aikar JVM configurations over three iterations of 120 seconds.

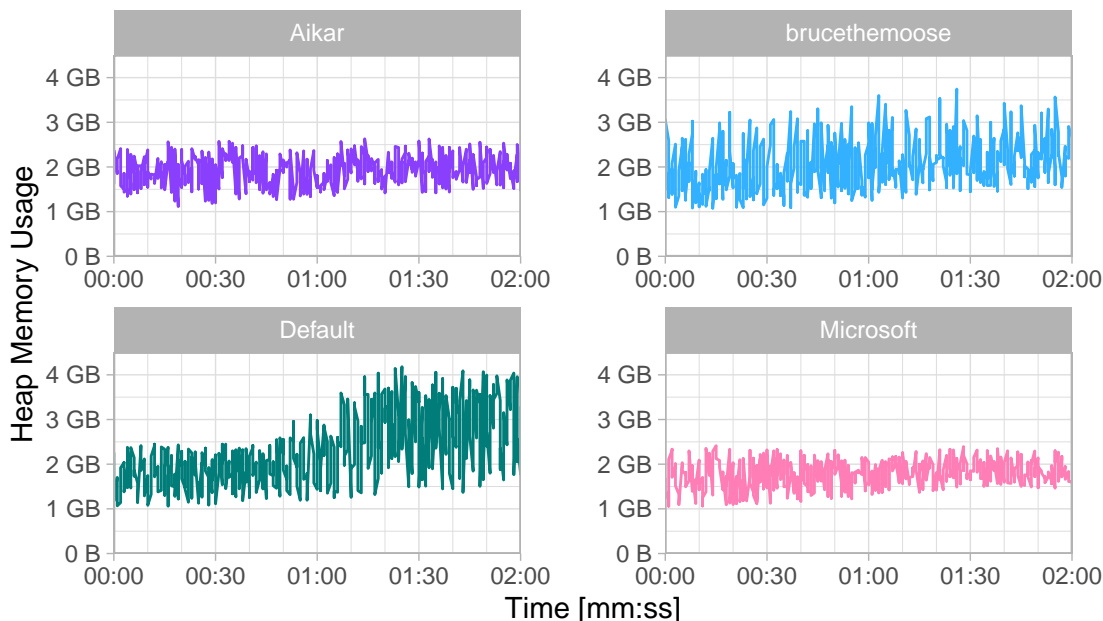
Configuration	Min	25th percentile	Median	75th percentile	Max
Microsoft	1.06 GB	1.56 GB	1.79 GB	2.05 GB	2.41 GB
Default	1.06 GB	1.68 GB	2.18 GB	2.85 GB	4.18 GB
brucethemoose	1.08 GB	1.58 GB	1.97 GB	2.45 GB	3.74 GB
Aikar	1.11 GB	1.66 GB	1.92 GB	2.20 GB	2.63 GB

120 seconds. In the figure, we can interpret each dip as a garbage collection. We can see that brucethemoose and the default configuration have similar patterns featuring high peaks and low lows, but after each collection, the memory usage shoots up very quickly. Microsoft and Aikar have a pattern with less extreme peaks and without the memory peaking right after a collection.

In table 5.5, we see a statistical summary of the heap usage in Gigabytes for the Microsoft Default, brucethemoose, and Aikar JVM configurations over three iterations of 120 seconds. Here, we can see that the maximum usage of Microsoft and Aikar are drastically lower than that of the Default and brucethemoose, with Default clearly being the worst, given it has the highest value for the median: 2.18 GB, 75th percentile 2.85 GB, and maximum: 4.18 GB. We see Microsoft performing the best, given it features the lowest value seen in each

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---



**Figure 5.12:** Line plots of heap memory usage in Gigabytes for the Microsoft, Default, brucethemoose, and Aikar JVM configurations over one iteration of 120 seconds.

of the columns of the table, with a minimum of 1.06 GB, a 25th percentile of 1.56 GB, a median of 1.79 GB, a 75th percentile of 2.05 GB, and a maximum of 2.41 GB. Aikar shows a 37% decrease in the maximum heap memory usage, and Microsoft shows a 42% decrease in the maximum heap memory usage compared to the default configuration. In comparison, brucethemoose shows an 11% decrease in maximum heap memory usage compared to the Default configuration.

Overall, Aikar shows the most stable and low tick time, along with a consistent stable garbage collection pattern, low CPU usage, and low memory usage. Microsoft shows better memory usage along with a similar pattern of garbage collection; however, the Microsoft configuration is worse in CPU usage and tick time. We will examine the configuration used by Aikar and touch upon the used options and their purpose; subsequently, we will provide a summary of the findings where we compare Aikar against Default.

In this section, we discuss the configuration used by Aikar, which focuses on improving the garbage collector’s behavior. We categorize the GC options into categories: sizing, aging, and collection. We discuss categories as a whole rather than each option to maintain the scope of this thesis.

The JVM is configured to resize with a larger section of reserve memory. **Sizing:** The configuration changes the size of each section in the regions and significantly increases the minimum size of the young generation memory region. **Aging:** The time after which

## 5.5 MF4: MVE Specific JVM Configurations Can Positively Impact Performance

---

an object is transferred to the old generation is decreased. The size of the transition section between the young and old generation is drastically decreased. **Collection:** Manual triggering of garbage collection through the application code is disabled. The garbage collection is configured to use more threads during its marking stage, and the marking stage is set to trigger at a lower heap usage percentage.

This configuration focuses on increasing the size of the young generation and swiftly cleaning up the young generation. While it decreases old generation garbage collection, it increases the rate at which objects are transferred to the old generation. This suits our workload, especially because our bots are constantly moving, and thus, most objects will be short-lived.

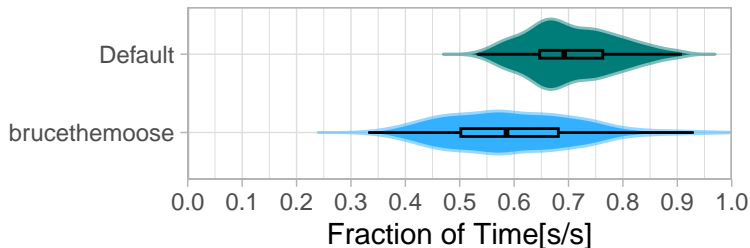
We argue the performance improvements achieved by the Aikar configuration can be attributed to the optimized garbage collection behavior and management of the heap. The configuration adapts the garbage collector to clean up short-lived objects quickly, reducing memory usage and the need for old generation collection. This all reduces the impact of garbage collection on the execution of the program. Allowing the CPU to focus more on the execution of the program rather than garbage collection, improving the tick time and CPU load. In summary, the configuration reduces memory usage, decreases the load on the CPU, and has a positive impact on tick time.

We ran an additional experiment testing the maximum amount of players, and how the configurations impact this number. We run a single iteration using the jump bot behavior. The jump bot behavior is used in place of idle behavior to increase the representatives of the workload. This is done by ensuring that the player count holds even if the players are not idle. This experiment is done to find out whether the speculated boosts to scalability convert to an increased maximum player count. During this test, one configuration rose above the rest: brucethemoose, which supported 190 players, whereas each other configuration crashed at this bot count. We compare the statistics of brucethemoose at 190 bots to the out-of-the-box configuration at 180 bots.

In Figure 5.13, we compare the average CPU usage in the utilized cores in seconds per second for the Default and brucethemoose configuration, with 180 and 190 bots, respectively, over one iteration of 120 seconds. In the figure, we see that Default has a much higher median than brucethemoose: 69% compared to brucethemoose's 59%. Furthermore, brucethemoose hits much lower lows, 33% minimum for brucethemoose compared to 53% for Default. However, the brucethemoose hits a higher maximum of 97% compared to Default's 91%. This leads us to speculate that the brucethemoose configuration reduces the CPU load, which allows the server to keep up with the demand better. Other metrics are

## 5. WHAT IS THE PERFORMANCE IMPACT OF JVM CONFIGURATION ON MVES?

---



**Figure 5.13:** Violin plot of the average CPU Usage in the used cores in fractions of a second for the Default JVM configuration with 180 bots and brucethemoose JVM configuration with 190 bots over one iteration of 120 seconds.

similar and reveal no major insights to us. This is a significant positive marker and shows potential; we believe further work in applying such configurations in real-world situations can be fruitful. In summary, the brucethemoose configuration can increase the maximum bot count.

This leads us to conclude that a JVM configuration developed for MVEs can positively impact MVE performance in terms of CPU usage and heap memory usage. Furthermore these configurations have shown to be able to increase the maximum bot count by 10 bots compared to the out-of-the-box configuration. The used configuration can lead to an 8% decrease in maximum CPU usage, a 5% decrease in the 25th percentile, and a 22% decrease in minimum CPU usage. Furthermore, the same configuration can lead to a 37% decrease in maximum heap usage when compared to an out-of-the-box configuration. Lastly, using such a configuration can increase the maximum player count by 10.

### 5.6 Limitations and/or Threat to Validity

In this section, we discuss this thesis’s limitations and the results it has produced. Additionally, we discuss threats to its validity.

Although we have designed and implemented our workloads to be representative of real-world scenarios, a real-world scenario with unique players will not have consistent and repetitive behavior like the simulated bots do, which may lead to a difference in the achieved JVM configuration impact.

Additionally, we are missing an evaluation of the game’s IO operations and networking elements. While the JVM configurations may improve the metrics we evaluated, they could negatively impact these.

Also missing from our framework and evaluation are detailed garbage collection metrics. There is a wide range of tools specifically made to analyze the garbage collector, which have not been used in our framework and evaluation, while these tools have the potential to improve our understanding of the observed impact.

Metrics specific to the execution and working of the JIT were not evaluated by our framework, potentially leaving a hole in our understanding of the performance impact of the JVM configuration. However, we argue that due to the nature of most of our experiments where the JIT stays consistent, the details will not be important to understand the performance impact. In the experiment discussed in section 5.2, where the JIT does change, we found no meaningful results; thus, we argue leaving out JIT metrics does not substantially hinder our understanding of the JVM configuration's performance impact.

Lastly, we do not analyze our framework's overhead on the performance of the MVE. However, we argue that this does not invalidate the differences in performance measured since every experiment is conducted using the same framework.

## 5.7 Evaluation Summary

The gathered results directly address research question 3: What is the performance impact of JVM configuration on MVEs? Our experiments have shown that JVM configurations can have a significant impact on the performance of a MVE. The choice between the OpenJDK JVM and GraalVM CE JVM has led to no significant performance impact in our experiments. However, we have shown that the chosen garbage collector can significantly impact performance. In our experiments, we compare the performance of ZGC against that of G1GC and have shown that ZGC can lead to a 390% increase in maximum observed tick time. We found that the choice of a maximum heap size of 32GiB over a maximum size of 4 GiB can lead to a reduction in maximum CPU usage of 8%, a 5% decrease of the 25th percentile, and a 22% decrease in minimum CPU usage. Lastly, we found that JVM configurations developed for MVEs can have a significant impact on the performance of the MVE when compared to the default JVM configuration. Leading to an 8% decrease in CPU usage, a 5% decrease in the 25th percentile, and a 22% decrease in the maximum CPU usage. As well as a 37% decrease in maximum heap usage. Most importantly, these configurations have led to an increase in maximum bot count of 10 during our experiments. Following these findings, we conclude that JVM configurations can have a meaningful impact on the performance of MVEs and are worth further research.

**5. WHAT IS THE PERFORMANCE IMPACT OF JVM  
CONFIGURATION ON MVES?**

---

## 6

# Related Work

A range of academic works researches the impact of JVM configurations on specific applications. These works have shown that the configuration of the JVM can have a significant impact on the performance of an application (8, 9, 10, 40, 41).

Some works have explored the automation of the configuration process; these works implement models, algorithms, and auto-tuners to automatically search for an optimal configuration of the JVM (8, 9, 10, 18, 41). Such tools have potential to improve the performance of an application. Due to time constraints, we do not evaluate results that can be achieved with an auto-tuner. The aim of this thesis is to understand more about the performance impact, not to figure out what configuration has the most impact. By evaluating different parts of the JVM rather than an auto-tuned configuration, we can learn more about the reason for the performance impact.

Furthermore, a range of resources exists to assist in understanding and configuring the JVM (45, 46, 47). We leverage these resources to deepen our understanding of the JVM and its configurations.

These works from academic and industry sources focus on specific applications or the JVM in general, and to our best knowledge, none of these works have targeted MVEs. We utilize these papers and the lessons learned in them as guidance for this thesis.

Within the MVE community, individuals have investigated what an optimal JVM configuration for Minecraft looks like (43, 44). Two of these configurations have some notoriety in the community and are mentioned on the Fandom Wiki page for setting up a server (42). One of these works by an individual who uses the moniker brucethemoose developed a benchmark to validate his results. However, this benchmark requires modifications to Minecraft to function and does not meet several key design requirements for Solarstick (outlined in chapter 3). We leverage the work done by these 2 individuals to

## 6. RELATED WORK

---

improve the design of Solarstick, and we evaluate the impact of the JVM configurations they developed on the performance of MVEs.

Lastly, the performance of MVEs has been studied in several academic works (6, 17, 48, 49, 50). These works each focus on different parts of MVEs, for example, Meterstick focuses on performance variability in cloud and self-hosted environments, while Yardstick introduces a benchmark for MVEs. Our work builds on this research by designing and prototyping a framework loosely based on Yardstick (6) and the Opencraft tutorial (22) to evaluate the performance impact of JVM configuration. To the best of our knowledge, this area has not been addressed by earlier academic work. Identifying how JVM configurations can positively impact MVE performance can improve the scalability of MVEs. Our findings, when combined with the findings that improve scalability from previous works, can further the scalability of MVEs

To summarize, the work done on JVM configurations for MVEs in this thesis is novel to the best of our knowledge. However, the MVE performance and JVM configuration research fields are well established. Our work leverages existing research and incorporates it into our design to produce results in this novel field of research.



# 7

## Conclusion

In this section, we conclude the work done in this thesis with a conclusion and discuss this work's limitations and the opportunities for future work.

### 7.1 Answering Research Questions

This section will offer a conclusion to this thesis; this will be done in two parts: firstly, we will answer the research questions posed in section 1.2, and then we will give a summary.

**Research question 1: How to design a performance evaluation framework to evaluate JVM configuration impact on MVE performance?** To address this question, we went through the process of designing a framework; we constructed a list of requirements and designed a framework called Solarstick around the requirements. The design and the design process can be found in chapter 3.

**Research question 2: How to implement such a framework in practice?** To address this question, we created a prototype called Solarstick. Solarstick is based on the design we created for research question 1 in chapter 3. The implementation is covered in chapter 4.

**Research question 3: What is the performance impact of JVM configuration on MVEs?** To answer this question we first design a set of experiments to give us a broad overview of the performance impact. Consequently, we conduct the experiments using the Solarstick prototype we implemented for research question 2. This process is described in chapter 5. We subsequently evaluate the results gathered during the experiments.

The experiments have shown that JVM configurations can significantly impact the performance of the MVE. Switching from OpenJDK to GraalVM CE did not lead to an observed performance impact. However, selecting the ZGC garbage collector instead of

## 7. CONCLUSION

---

the OpenJDK default G1GC led to an observed increase in maximum tick time of 390% from 33.11ms using G1GC to 162.09 ms using ZGC. Furthermore, we observed that increasing the maximum heap size can positively impact CPU load; increasing the maximum heap size from 4 GiB to 32 GiB led to an 8% decrease in maximum CPU usage and a 22% decrease in minimum CPU usage. Lastly, comparing a JVM configuration developed for MVEs against the default JVM configuration led to an observed 8% decrease in maximum CPU usage, a 22% decrease in minimum CPU usage, a 37% decrease in maximum heap memory usage, and an increase in maximum bot count of 10.

We design and implement Solarstick, an evaluation framework for evaluating the impact of JVM configurations on MVE performance. Subsequently, we use Solarstick to conduct experiments to evaluate the performance impact. The experiments lead to valuable insights into the impact of JVM configurations on MVE performance. These insights can be applied to future work and in practice to enhance the scalability of MVEs.

### 7.2 Limitations and Future Work

In this section, we will discuss the limitations of this thesis. Additionally, we will address opportunities for future work. We start by discussing the limitations of Solarstick.

The limitations of Solarstick are as follows:

- **Crash handling:** Solarstick does not gracefully handle crashes; the only crashes that are handled are crashes of the MVE/JVM. If any other program crashes, the experiment will fail, and no data will be available.
- **GC inspection tools** Solarstick does not include tools to evaluate the garbage collector; in its current state, only the metrics exposed by JMX (21, 29) are available. The available metrics do include GC-specific metrics, but a tool specifically made for investigating GC behavior could potentially lead to new insights and is thus a worthwhile direction for future work.

The lack of proper crash handling does not affect runs where no crashes happen and thus does not affect our results. However, proper crash handling would increase the quality of life for Solarstick users. The lack of GC inspection tools could mean we are missing insights and/or details in our evaluation. However, we argue that this does not invalidate the existing insights but rather is an opportunity for future work.

In this thesis, we give an overview of the general impact of JVM configurations on the performance of MVEs, however, there are limitations to the scope of our evaluation:

- **MVE implementations evaluated:** We only evaluate the performance impact on one MVE implementation: Vanilla Minecraft. It is worthwhile to evaluate the impact of these configurations on various MVE implementations. Previous research has shown that there are performance differences between implementations (6)
- **JVM implementations evaluated:** This thesis has only tested a single version of 2 JVM implementations: GraalVM CE version 22.0.1 and OpenJDK version 22.0.1. However, previous works have shown that there are differences between implementations and versions of these implementations. Both the implementations themselves and the execution of programs on these implementations have shown differences DBLP:conf/icse/ChenSS19,5445810,DBLP:journals/computers/LambertMC22, this leads us to argue that evaluating the impact of JVM implementations on MVE performance could be a valid subject for future work.

These limitations could restrict the effective scope of our results. The observed impact may be specific to the tested MVE and JVM implementation. However, regardless of a restricted scope, we argue that the findings are still a valuable basis for future work.

We have investigated the impact of multiple components of the JVM on the performance of MVEs. However, this evaluation has limitations which leave room for future work:

- **JIT tuning:** We have not tuned the JIT in this thesis, due to time constraints. Due to these constraints, we have chosen to focus on a more general overview of the impact that JVM configurations have. However, tuning the JIT has positively impacted the performance of applications in previous works (18), and thus could be an interesting subject for future work.
- **GC configuration:** The garbage collector and configurations that focus on the garbage collector have been shown to significantly impact the performance of MVEs in the following sections 5.3, and 5.5. However, this thesis does not provide an in-depth evaluation and exploration of GC settings. The impact that the GC and its configuration have on the performance of the MVE make this a worthwhile subject for future work.
- **Missing metrics** This thesis does not fully evaluate the performance of the MVE; multiple metrics are missing, such as network and I/O operations. Future work can be done on the impact that the JVM configurations have on the network and I/O operations of the MVE.

## 7. CONCLUSION

---

The absence of JIT tuning and GC configuration exploration means that we may have missed potential significant configuration options. However, the focus of this thesis is to evaluate the impact of the configurations, not to find an optimal configuration. Thus we argue that our work is still valid and can be used as a basis for future work.

Although our evaluation does not include all metrics and thus does not give a complete picture, the findings in this thesis are still valid. The results offer insights into the impact that JVM configurations have on MVE performance and can be used and built upon by future research.

During our experiments, we use simulated player behavior and try to model the behavior to be representative of a real-world scenario. However, the simulation will not be 100% representative of a real-world scenario; if the same experiments were run using real players, the results may differ. Since players will not move and perform actions in the same way that our bots do, this might change the specifics of the results. However, this does not change the fact that the impact achieved is possible.

Our experiments use a low number of iterations, and there is no reporting on the significance of these numbers. This is due to time constraints. Running the experiments and gaining a satisfying confidence level did not fit within the time limits for this thesis. However, the achieved results are still possible, and the speculation and discussion on causes are not invalidated but might not show the full picture. This is an opportunity for future work.

Lastly, we will discuss approaches/experiments that have not been included in this thesis but show potential for future work:

- **Auto-tuner:** This thesis does not evaluate the performance impact that can be achieved using an auto-tuner for the JVM. Auto-tuners have shown to be able to positively impact the performance of applications and at times be even better than hand-picked configurations (8, 10, 18, 41). Additionally this could lead to insights on which JVM configuration settings impact MVE performance.
- **Maximum player count:** This thesis has not done an in-depth evaluation of the potential for JVM configuration to increase the maximum amount of players supported by a single instance. We have done a single maximum player count test however, we mostly opted for other experiments to learn more about the impact of the JVM configurations in terms of different metrics rather than focus on the maximum number of players.

## 7.2 Limitations and Future Work

---

The auto tuner could have led to a more optimal configuration and could offer insights into the settings that affect performance. Evaluating whether the configurations affect the maximum player count could have helped us gain insights into the scalability of MVEs. However both of these limitations do not invalidate the findings in this thesis, but could be interesting opportunities for future work.

## 7. CONCLUSION

---

# References

- [1] ZACHARY BODDY. **Minecraft crosses 300 million copies sold as it prepares to celebrate its 15th anniversary**, Oct 2023. (Accessed on 07/08/2024). 1
- [2] ZACHARY BODDY. **Minecraft boasts over 141 million monthly active users and other impressive numbers**, Oct 2021. (Accessed on 07/08/2024). 1, 5
- [3] STEVE NEBEL, SASCHA SCHNEIDER, AND GÜNTER DANIEL REY. **Mining Learning and Crafting Scientific Experiments: A Literature Review on the Use of Minecraft in Education and Research**. *J. Educ. Technol. Soc.*, **19**(2):355–366, 2016. 1
- [4] SUNGWOONG LEE, WOYOUNG WILLIAM JANG, AND MINNA ROLLINS. **Using Minecraft Education Edition to Enhance 21st Century Skills in the College Classroom: A Mixed Methods Study**. In TUNG X. BUI, editor, *57th Hawaii International Conference on System Sciences, HICSS 2024, Hilton Hawaiian Village Waikiki Beach Resort, Hawaii, USA, January 3-6, 2024*, pages 5339–5346. ScholarSpace, 2024. 1
- [5] MOJANG AB. **Get Minecraft for Your Classroom | Minecraft Education**. <https://education.minecraft.net/en-us>. (Accessed on 07/26/2024). 1
- [6] JEROM VAN DER SAR, JESSE DONKERVLIET, AND ALEXANDRU IOSUP. **Yardstick: A Benchmark for Minecraft-like Services**. In VARSHA APTE, AN-TINISCA DI MARCO, MARIN LITOIU, AND JOSÉ MERSEGUER, editors, *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*, pages 243–253. ACM, 2019. 1, 17, 20, 42, 45
- [7] MICROSOFT. **Microsoft Realms Servers: Bedrock & Java | Minecraft**. 1

## REFERENCES

---

- [8] MILINDA FERNANDO, THARINDU RUSIRA PATABANDI, CHALITHA PERERA, AND CHAMARA PHILIPS. **CGO:U:Auto-tuning the HotSpot JVM**, 2015. 1, 2, 7, 33, 41, 46
- [9] DELELI MESAY ADINEW, SHIJIE ZHOU, AND YONGJIAN LIAO. **Spark Performance Optimization Analysis in Memory Tuning On GC Overhead for Big Data Analytics**. In *Proceedings of the 8th International Conference on Networks, Communication and Computing, ICNCC 2019, Luoyang, China, December 13-15, 2019*, pages 75–78. ACM, 2019. 2, 7, 41
- [10] FELIPE CANALES, GEOFFREY HECHT, AND ALEXANDRE BERGEL. **Optimization of Java Virtual Machine Flags using Feature Model and Genetic Algorithm**. In JOHANN BOURCIER, ZHEN MING (JACK) JIANG, COR-PAUL BEZEMER, VITTORIO CORTELLESA, DANIELE DI POMPEO, AND ANA LUCIA VARBANESCU, editors, *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021, Companion Volume*, pages 183–186. ACM, 2021. 2, 41, 46
- [11] ALEXANDRU IOSUP, LAURENS VERSLUIS, ANIMESH TRIVEDI, ERWIN VAN EYK, LUCIAN TOADER, VINCENT VAN BEEK, GIULIA FRASCARIA, AHMED MUSAAFIR, AND SACHEENDRA TALLURI. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 1765–1776. IEEE, 2019. 3
- [12] R. HAMMING. *The Art of Doing Science and Engineering: Learning to Learn*. CRC Press, 1997. 3
- [13] KEN PEFFERS, TUURE TUUNANEN, MARCUS A. ROTHENBERGER, AND SAMIR CHATTERJEE. **A Design Science Research Methodology for Information Systems Research**. *J. Manag. Inf. Syst.*, **24**(3):45–77, 2008. 3
- [14] RAJ JAIN. **The Art of Computer Systems Performance Analysis**. In LEO F. ZIMMERMAN, JOHN P. PILCH, LINDA J. CARROLL, MARK SORKIN, DANIEL KABERON, DOUG MCBRIDE, FRANK M. BEREZNAVAY, DALE DOOLITTLE, JOEL GOLDSTEIN, DAVE THORN, HARRY ZIMMER, ELLEN E. ROBERTSON, PEGGY DE ROSSETT, BERNARD DOMANSKI, PHILIP CLARK, ROBERT L. MORRISON, AND



## REFERENCES

---

- EDGAR A. ORTIZ, editors, *17th International Computer Measurement Group Conference, Nashville, TN, USA, December 9-13, 1991, Proceedings*, pages 1233–1236. Computer Measurement Group, 1991. 3
- [15] GERNOT HEISER. **Systems Benchmarking Crimes**. <http://www.cse.unsw.edu.au/~Gernot/benchmarking-crimes.html>, 2019. Accessed: 2020. 3
- [16] JOHN K. OUSTERHOUT. **Always measure one level deeper**. *Commun. ACM*, **61**(7):74–83, 2018. 3
- [17] JERRIT EICKHOFF, JESSE DONKERVLIET, AND ALEXANDRU IOSUP. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games**. In MARCO VIEIRA, VALERIA CARDELLINI, ANTINISCA DI MARCO, AND PETR TUMA, editors, *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15-19, 2023*, pages 173–185. ACM, 2023. 5, 6, 25, 42
- [18] KENNETH HOSTE, ANDY GEORGES, AND LIEVEN EECKHOUT. **Automated just-in-time compiler tuning**. In ANDREAS MOSHOVOS, J. GREGORY STEFFAN, KIM M. HAZELWOOD, AND DAVID R. KAELI, editors, *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 62–72. ACM, 2010. 7, 41, 45, 46
- [19] ORACLE CORPORATION. **OpenJDK**. <https://openjdk.org/>. (Accessed on 07/20/2024). 8
- [20] ORACLE. **GraalVM**. <https://www.graalvm.org/>. (Accessed on 07/20/2024). 8
- [21] ORACLE. **Java Management Extension(JMX)**. 15, 20, 44
- [22] ATLARGE RESEARCH. **Public Opencraft Tutorial. Get familiar with Opencraft and Yardstick**. <https://github.com/atlarge-research/opencraft-tutorial>. (Accessed on 07/12/2024). 17, 18, 42
- [23] ANACONDA. **Miniconda — Anaconda documentation**. <https://docs.anaconda.com/miniconda/>. (Accessed on 07/01/2024). 17
- [24] ANACONDA. **Download Anaconda Distribution | Anaconda**. <https://www.anaconda.com/download/>. (Accessed on 07/01/2024). 17

## REFERENCES

---

- [25] ANSIBLE COLLABORATIVE. **Homepage | Ansible Collaborative**. <https://www.ansible.com/>. (Accessed on 07/01/2024). 18, 56
- [26] HENRI E. BAL, DICK H. J. EPEMA, CEES DE LAAT, ROB VAN NIEUWPOORT, JOHN W. ROMEIN, FRANK J. SEINSTRA, CEES SNOEK, AND HARRY A. G. WISHOFF. **A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term**. *Computer*, **49**(5):54–63, 2016. 18, 24, 55
- [27] MINECRAFT WIKI. **server.properties – Minecraft Wiki**. <https://minecraft.fandom.com/wiki/Server.properties>. (Accessed on 07/02/2024). 19
- [28] PROMETHEUS AUTHORS. **Prometheus Node Exporter: Exporter for machine metrics**. [https://github.com/prometheus/node\\_exporter](https://github.com/prometheus/node_exporter). (Accessed on 07/02/2024). 19
- [29] PROMETHEUS AUTHORS. **Prometheus JMX Exporter: A process for exposing JMX Beans via HTTP for Prometheus consumption**. [https://github.com/prometheus/jmx\\_exporter](https://github.com/prometheus/jmx_exporter). (Accessed on 07/02/2024). 19, 20, 44
- [30] PRISMARINEJS. **mineflayer: Create minecraft bots with a powerful, stable, and high level javascript API**. 19, 25
- [31] ROLAND HUSS. **Jolokia – Overview**. <https://jolokia.org/>. (Accessed on 07/03/2024). 20
- [32] PROMETHEUS AUTHORS. **Prometheus - Monitoring system & time series database**. <https://prometheus.io/>. (Accessed on 07/04/2024). 20
- [33] INHAZE. **Hillside Manor World [1.8] 4 Year Anniversary Minecraft Map**. <https://www.planetminecraft.com/project/hillside-manor/>. (Accessed on 06/30/2024). 21, 22
- [34] MICROSOFT. **Minecraft Server Download | Minecraft**. <https://www.minecraft.net/en-us/download/server>. (Accessed on 06/30/2024). 25
- [35] MICROSOFT. **Release Changelogs – Minecraft Feedback**. <https://feedback.minecraft.net/hc/en-us/sections/360001186971-Release-Changelogs>. (Accessed on 06/30/2024). 25

- 
- [36] YUTING CHEN, TING SU, AND ZHENDONG SU. **Deep differential testing of JVM implementations**. In JOANNE M. ATLEE, TEVFIK BULTAN, AND JON WHITTLE, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1257–1268. IEEE / ACM, 2019. 25
- [37] ORACLE. **Graal Compiler**. <https://www.graalvm.org/latest/reference-manual/java/compiler/>. (Accessed on 06/29/2024). 26
- [38] ORACLE. **Why GraalVM?** <https://www.graalvm.org/why-graalvm/>. (Accessed on 06/29/2024). 26
- [39] ALBERT MINGKUN YANG AND TOBIAS WRIGSTAD. **Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK**. *ACM Trans. Program. Lang. Syst.*, 44(4):22:1–22:34, 2022. 28
- [40] SEMIH SAHIN, WENQI CAO, QI ZHANG, AND LING LIU. **JVM Configuration Management and Its Performance Impact for Big Data Applications**. In CALTON PU, GEOFFREY C. FOX, AND ERNESTO DAMIANI, editors, *2016 IEEE International Congress on Big Data, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 410–417. IEEE Computer Society, 2016. 33, 41
- [41] VENKTESH V, POOJA B. BINDAL, DEVESH SINGHAL, A. V. SUBRAMANYAM, AND VIVEK KUMAR. **OneStopTuner: An End to End Architecture for JVM Tuning of Spark Applications**. *CoRR*, abs/2009.06374, 2020. 33, 41, 46
- [42] MINECRAFT FANDOM. **Tutorials/Setting up a server – Minecraft Wiki**. [https://minecraft.fandom.com/wiki/Tutorials/Setting\\_up\\_a\\_server](https://minecraft.fandom.com/wiki/Tutorials/Setting_up_a_server). (Accessed on 07/08/2024). 33, 41, 59
- [43] BRUCE THE MOOSE. **Minecraft Performance Flags Benchmarks: Sane, Benchmarked Java Flags and Tweaks for Minecraft**. <https://github.com/brucethemoose/Minecraft-Performance-Flags-Benchmarks#server-g1gc>. (Accessed on 07/01/2024). 33, 41, 60
- [44] DANIEL ENNIS (AIKAR). **JVM Tuning: Optimized G1GC for Minecraft - Aikar’s Thoughts**. <https://aikar.co/2018/07/02/tuning-the-jvm-g1gc-garbage-collector-flags-for-minecraft/>. (Accessed on 07/01/2024). 33, 41, 61

## REFERENCES

---

- [45] ORACLE. **Automating Java Performance Tuning**. <https://www.oracle.com/technical-resources/articles/javaee/automating-java-performance-tuning.html>. (Accessed on 07/12/2024). 41
- [46] ORACLE. **Java Tuning White Paper**. <https://www.oracle.com/java/technologies/java-tuning.html>. (Accessed on 07/12/2024). 41
- [47] SCOTT OAKS. *Java performance: In-depth advice for tuning and programming java 8, 11, and beyond*. O'Reilly Media, Sebastopol, CA, 2 edition, 2020. 41
- [48] JESSE DONKERVLIET, JAVIER RON, JUNYAN LI, TIBERIU IANCU, CRISTINA L. ABAD, AND ALEXANDRU IOSUP. **Servo: Increasing the Scalability of Modifiable Virtual Environments Using Serverless Computing**. In *43rd IEEE International Conference on Distributed Computing Systems, ICDCS 2023, Hong Kong, July 18-21, 2023*, pages 829–840. IEEE, 2023. 42
- [49] JESSE DONKERVLIET, JIM CUIJPERS, AND ALEXANDRU IOSUP. **Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency**. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 126–137. IEEE, 2021. 42
- [50] MATT COCAR, RENEISHA HARRIS, AND YOURY KHMELEVSKY. **Utilizing Minecraft bots to optimize game server performance and deployment**. In *30th IEEE Canadian Conference on Electrical and Computer Engineering, CCECE 2017, Windsor, ON, Canada, April 30 - May 3, 2017*, pages 1–5. IEEE, 2017. 42
- [51] DILANO EMANUEL JERMAINE DOELWIJT. **Evaluating the performance impact of JVM configurations on MVE performance**. 55, 56

# Appendix A

## Reproducibility

### A.1 Abstract

Ansible based performance evaluation framework meant for use on DAS-5. Evaluates running MVE server with specified JVM configuration.

### A.2 Artifact check-list (meta-information)

- **Program:** Solarstick (51).
- **Compilation:** Python3, Java.
- **Run-time environment:** Experiments are executed on CentOS, Solarstick has not been tested on other operating systems.
- **Hardware:** DAS-5 (26) is used during the experiment, but other supercomputers with the same architecture also work, e.g., DAS-6.
- **Execution:** Configuration present on GitHub pins the Java process to the first four processor cores. Execution per iteration has a maximum of 900 seconds. Amount of iterations can be configured.
- **Metrics:** What metrics can be collected can be configured. The metrics exposed by JMX and Node are available. Tick time is always gathered.
- **Output:** The framework outputs plots and raw data.
- **Experiments:** Experiments are prepared by configuring the desired experiment in the `experiment_configuration.yml` (and/or adding the desired world to the `worlds` folder), then reserving a node and running the `run.sh` shell script on the node.
- **How much disk space required (approximately)?:** 250 MB
- **How much time is needed to prepare workflow (approximately)?:** 5 Minutes

## A. REPRODUCIBILITY

---

- **How much time is needed to complete experiments (approximately)?:** max 15 minutes per experiment iteration.
- **Publicly available?:** Yes on GitHub (51)
- **Workflow framework used?:** Ansible (25)
- **Archived (provide DOI)?:** No

### A.3 Description

#### A.3.1 How to access

Cloning the GitHub repository (51). Out of the box, Solarstick uses approximately 250 MB, *Obligatory*

#### A.3.2 Hardware dependencies

The framework depends on the structure of DAS-5; it heavily relies on the presence of dedicated compute nodes with a shared file-system.

#### A.3.3 Software dependencies

The system must contain the dependencies for Miniconda, Ansible, Prometheus, Jolokia, NodeJS, and Java.

### A.4 Installation

Installation is done by cloning the GitHub repository (51), configuring the experiment using the `experiment_configuration.yml` file, and then running the `'run.sh'` shell script on a reserved compute node.

### A.5 Experiment workflow

Configure the experiment in the YML file, then execute the `'run'` shell script on a reserved compute node. The experiment will be executed using Ansible. Ansible reserves nodes that are used for the bot simulation and the MVE server. After the experiment, results will be available in a time-stamped folder:

- The server log: The log created by the MVE server

- The configuration: A copy of the experiment configuration YAML file used for the run.
- The information regarding the node: This contains information about the hardware and software on the node.
- Raw data: The raw data collected during the experiment is in a JSON file.
- Plots: The raw data turned into plots. Each plot is available in PDF and PNG format.

## A.6 Evaluation and expected results

The results heavily depend on the configuration and system used. On DAS-5, using the configuration provided in GitHub, the expectation is for the tick time to be around 22-25 ms on average. Peaks are expected to be around 35-40 at most.

## A.7 Experiment customization

The experiment can be customized in the experiment configuration file. This offers the following options:

- `experiment_duration_s`: The period where metrics will be collected. Actual run time differs.
- `server_runtime_s`: The maximum length the server and monitoring programs can run. This is capped at 900 by the duration of the node reservation.
- `iterations`: Number of experiment iterations.
- `node_count`: Minimum 2, no maximum. One node will be the node that runs the MVE; the other nodes will be bot simulation nodes.
- `players_per_node`: The number of bots each bot simulation node simulates. Total players =  $(\text{node\_count} - 1) * \text{players\_per\_node}$
- `bot_behavior_script`: Selects the script the bot simulation nodes execute. Any script in the `bot_scripts` folder can be used. New behaviors can be added to this folder.
- `world_name`: Selects the world used during execution. Any world present in the `worlds` folder can be used. New worlds can be added to this folder.

## A. REPRODUCIBILITY

---

- `JVM_config_parameters`: Contains the entire Java command executed to run the MVE. The JVM configuration can be configured here. Different MVE implementations can be used by altering the download link for the server JAR downloaded in the `before.yml` file.

### A.8 Methodology

Submission, reviewing, and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-badging>
- <http://cTuning.org/ae/submission-20201122.html>
- <http://cTuning.org/ae/reviewing-20201122.html>

...



## Appendix B

# JVM configurations

During the experiments in chapter 5, we used JVM configurations sourced from the community. The full configurations are included here. Note that each configuration is run with additional flags for the heap size, GC, and processor pinning, as stated in the experiments and flags for the JMX and Jolokia agents. We will add the full command used for the default configuration as a reference. Options in the default configurations will be excluded from the community configurations. The full default configuration, including flags that we do not change, can be found in the GitHub, in the file `full_java_config`.

### **Default:**

```
taskset -c 0-3:1 java
-Dcom.sun.management.jmxremote.port=9999
-javaagent:./jmx_prometheus_javaagent-0.20.0.jar=12345:jmxexp_config.yml
-javaagent:./jolokia-agent.jar=port=7777,host=localhost
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-XX:+UseG1GC
-Xms2G
-Xmx4G
-jar server.jar nogui
```

### **Microsoft (42):**

```
-XX:+UnlockExperimentalVMOptions
-XX:G1NewSizePercent=20
-XX:G1ReservePercent=20
```

## B. JVM CONFIGURATIONS

---

-XX:MaxGCPauseMillis=50

-XX:G1HeapRegionSize=32M

### **brucethemoose (43):**

-XX:+UnlockExperimentalVMOptions

-XX:+UnlockDiagnosticVMOptions

-XX:+AlwaysPreTouch

-XX:+DisableExplicitGC

-XX:+UseNUMA

-XX:NmethodSweepActivity=1

-XX:ReservedCodeCacheSize=400M

-XX:NonNMethodCodeHeapSize=12M

-XX:ProfiledCodeHeapSize=194M

-XX:NonProfiledCodeHeapSize=194M

-XX:-DontCompileHugeMethods

-XX:MaxNodeLimit=240000

-XX:NodeLimitFudgeFactor=8000

-XX:+UseVectorCmov

-XX:+PerfDisableSharedMem

-XX:+UseFastUnorderedTimeStamps

-XX:+UseCriticalJavaThreadPriority

-XX:ThreadPriorityPolicy=1

-XX:AllocatePrefetchStyle=3

-XX:MaxGCPauseMillis=37

-XX:+PerfDisableSharedMem

-XX:G1HeapRegionSize=16M

-XX:G1NewSizePercent=23

-XX:G1ReservePercent=20

-XX:SurvivorRatio=32

-XX:G1MixedGCCCountTarget=3

-XX:G1HeapWastePercent=20

-XX:InitiatingHeapOccupancyPercent=10

-XX:G1RSetUpdatingPauseTimePercent=0

-XX:MaxTenuringThreshold=1

-XX:G1SATBBufferEnqueueingThresholdPercent=30

---

-XX:G1ConcMarkStepDurationMillis=5.0  
-XX:G1ConcRSHotCardLimit=16  
-XX:G1ConcRefinementServiceIntervalMillis=150  
-XX:GCTimeRatio=99  
-XX:+UseLargePages  
-XX:LargePageSizeInBytes=2m

**Aikar (44):**

-XX:+ParallelRefProcEnabled  
-XX:MaxGCPauseMillis=200  
-XX:+UnlockExperimentalVMOptions  
-XX:+DisableExplicitGC  
-XX:+AlwaysPreTouch  
-XX:G1NewSizePercent=30  
-XX:G1MaxNewSizePercent=40  
-XX:G1HeapRegionSize=8M  
-XX:G1ReservePercent=20  
-XX:G1HeapWastePercent=5  
-XX:G1MixedGCCountTarget=4  
-XX:InitiatingHeapOccupancyPercent=15  
-XX:G1MixedGCLiveThresholdPercent=90  
-XX:G1RSetUpdatingPauseTimePercent=5  
-XX:SurvivorRatio=32  
-XX:+PerfDisableSharedMem  
-XX:MaxTenuringThreshold=1  
-Dusing.aikars.flags=https://mcflags.emc.gs  
-Daikars.new.flags=true