SoftwareX 28 (2024) 101933

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

JVM optimization: An empirical analysis of JVM configurations for enhanced web application performance

Darlan Noetzold [a,b], Anubis Graciela de Moraes Rossetto [b], Luis Augusto Silva [c], Paul Crocker [d], Valderi Reis Quietinho Leithardt [e,*]

[a] *University of Vale do Rio dos Sinos (UNISINOS), São Leopoldo, Brazil*
[b] *Federal Institute of Education, Science and Technology Sul-rio-grandense, Passo Fundo, RS, 99.064-440, Brazil*
[c] *Department of Computer Science, Faculty of Science, Universidad de Salamanca, Salamanca, 37008, Spain*
[d] *Instituto de Telecomunicações and Departamento de Informática, Universidade da Beira Interior, Covilhã, Portugal*
[e] *Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Avenida das Forças Armadas, 1649-026, Lisboa, Portugal*

ARTICLE INFO

Keywords:
JVM optimization
Garbage collection
Performance metrics
Resource management
Web application efficiency

ABSTRACT

This research presents software for empirically analyzing Java Virtual Machine (JVM) parameter configurations to enhance web application performance. Using tools like JMeter and cAdvisor in a controlled hardware environment, it collects and analyzes performance metrics. Tailored JVM settings for high request loads improved CPU efficiency by 20% and reduced memory usage by 15% compared to standard configurations. For I/O intensive operations with large files, optimized JVM configurations decreased response times by 30% and CPU usage by 25%. These findings highlight the impact of tailored JVM settings on application responsiveness and resource management, providing valuable guidance for developers and engineers.

Code metadata

Current code version	v1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00346
Permanent link to Reproducible Capsule	https://github.com/DarlanNoetzold/JVMParamsCompare
Legal Code License	MIT License.
Code versioning system used	git
Software code languages, tools, and services used	Java
Compilation requirements, operating environments & dependencies	Maven, Java 17, Spring, Docker
If available Link to developer documentation/manual	https://documenter.getpostman.com/view/16000387/2sA3XSAgXs
Support email for questions	darlan.noetzold@gmail.com

1. Motivation and significance

The optimization of Java Virtual Machine (JVM) parameters is essential for improving the performance of web applications, directly impacting the efficiency and response to resource demands [1,2].

This work focuses on customizing JVM parameters for web applications that are executed under various simulated scenarios. To ensure comprehensive performance evaluation, load generation, and testing were conducted under various specific scenarios. This involved simulating various levels of traffic and operational scenarios to measure how the application handled different loads and stresses, for instance when an application is operated under an HTTP request processing load scenario or in a scenario where the requests generate intensive I/O operations. It has been shown that adjusting the JVM parameters to each scenario can significantly improve the application's performance and efficiency [3].

This study presents results aimed at increasing the efficiency of JVM resource usage in the following six scenarios: High Request Load, where the goal is to maximize response under intense demands; Intensive I/O

* Corresponding author.
E-mail addresses: dnoetzold@edu.unisinos.br (Darlan Noetzold), anubisrossetto@ifsul.edu.br (Anubis Graciela de Moraes Rossetto), luisaugustos@usal.es (Luis Augusto Silva), paul.crocker@lx.it.pt (Paul Crocker), valderi.leithardt@iscte-iul.pt (Valderi Reis Quietinho Leithardt).

https://doi.org/10.1016/j.softx.2024.101933
Received 11 July 2024; Received in revised form 30 September 2024; Accepted 8 October 2024
Available online 21 October 2024

Operations and Large Files, adapting the JVM to handle large volumes of data; CPU Efficiency and Low Memory Consumption, focused on systems with limited resources; High Robustness in Production Environments, ensuring stability and resilience; A Mixed Workload scenario, where versatility is crucial; and a scenario balancing Efficient Garbage Collection Management and memory operations. Each configuration was tested in a controlled and isolated environment, and tools such as JMeter and cAdvisor were used to collect and analyze performance metrics [4,5]. The results show a direct comparison between the scenarios, revealing how different JVM adjustments can enhance performance. For instance, in the "High Request Load" scenario, the JVM configuration focusing on CPU efficiency and low memory consumption showed a 20% increase in CPU load and a 15% reduction in memory usage compared to the standard configuration. In the "Intensive I/O Operations and Large Files" use case, it was observed that the JVM optimized for I/O reduced response time by 30% and CPU usage by 25% compared to other configurations. These results underscore the importance of selecting the appropriate JVM configuration for each specific scenario to maximize performance and minimize resource consumption.

2. Software description

2.1. Software architecture

For the development, deployment, testing, and monitoring of a complex web application, various specialized tools are necessary. The software application was built using the Spring Boot framework and the architecture adopted followed the MVC (Model-View-Controller) pattern, providing a clear separation between business logic, user interface, and database interaction. For data management, PostgreSQL and JPA with Hibernate were chosen for the persistence layer, offering abstraction over SQL operations and reducing error susceptibility. Furthermore, integration with the RabbitMQ messaging broker was used to enhance scalability and efficiency by handling the asynchronous operations, non blocking operations, that do not require immediate processing. For issues related to data security and privacy, we used the work developed in [6], to manage the criteria and parameters of these topics Docker provided an isolated and replicable environment for each instance of the application. The JVM, along with its Garbage Collector, can then be tuned to optimize execution and memory management. For testing, Postman and JMeter were used to simulate requests and loads, while cAdvisor and JProfiler monitored resource usage and performance. These technologies together allowed for a detailed evaluation of the system in the selected operational scenarios.

2.2. Software functionalities

In addition to the basic operations (GET, POST, PUT, DELETE on 'NormalData'), the application features specialized modules for intensive data processing ('DataIntensive POST'), heavy calculations ('HeavyCalculation POST'), and simulation of I/O operations ('IOSimulation POST'). These additional functionalities allow for a comprehensive evaluation of performance under various JVM configurations, exploring aspects such as response time, memory usage, and CPU load across different scenarios. This framework was designed to assess the impact of different JVM configurations under various operational conditions, enabling specific analysis of each use case.

2.2.1. Performance evaluation metrics

The metrics CPU load, memory consumption, GC activity, runnable threads, and errors were used to understand how each configuration influences aspects such as application response, resource consumption, and overall efficiency. Below are the formulas for how the main metrics used are calculated.

- **Throughput:** Number of operations that a system can process in a unit of time [7].

$$\text{Throughput} = \frac{\text{Total Number of Requests}}{\text{Total Test Time}}$$

- **CPU Load:** Percentage of CPU utilization by the system during the tests [8].

$$\text{CPU Load} = \frac{\sum_{i=1}^{n} \text{CPU Load}_i}{n}$$

- **Average Memory Consumption:** Amount of RAM used by the application [9].

$$\text{Average Memory Consumption} = \frac{\sum_{i=1}^{n} \text{Memory Used}_i}{n}$$

- **GC Activity (%):** Percentage of total test time spent on garbage collection activities [10].

$$\text{GC Activity (\%)} = \left(\frac{\text{Total GC Time}}{\text{Total Test Time}}\right) \times 100$$

- **Runnable Threads:** Average number of threads in the running state [9].

$$\text{Runnable Threads} = \text{Average Number of Runnable Threads}$$

- **Error (%):** Percentage of requests that resulted in errors during the test.

$$\text{Error (\%)} = \left(\frac{\text{Number of Errors}}{\text{Total Number of Requests}}\right) \times 100$$

2.3. Selection of the use cases

The selection of the scenarios used in this study is grounded in a combination of established industry practices, insights from relevant academic literature, and practical experiences in the development and operation of large-scale web applications. These scenarios were chosen based on their prevalence and significance in the field, reflecting common challenges that web applications frequently encounter in real-world environments.

Specifically, the scenarios were informed by widely recognized performance considerations for web applications, such as handling high traffic loads, managing intensive I/O operations, and optimizing resource usage in constrained environments.

The selection process also incorporated practical insights gained from analyzing performance issues reported by developers and system operators in real-world applications. These insights helped identify key areas where JVM parameter tuning can have a substantial impact on application performance. The scenarios selected for this study thus represent a synthesis of theoretical knowledge and practical experience [11–13].

The chosen scenarios cover a broad spectrum of performance concerns, making them applicable to a wide range of web application contexts. This approach ensures that the findings of this study can be generalized and applied to various types of web systems, providing valuable guidance for developers and engineers seeking to optimize JVM performance in similar environments.

2.3.1. High request load

This case focused on applications that need to handle a large number of simultaneous requests, optimizing for quick responses and efficient thread management. The selected parameters are presented and justified in Table 1.

Table 1
High request load.

Parameter	Value	Justification
`-XX:+UseParallelGC`	Default	Optimizes the application for multicore environments by reducing garbage collector pause times.
`-XX:MaxGCPause Millis`	200	Aims to minimize GC pauses to maintain quick response times under load.
`-XX:NewRatio`	2	Balances allocation between the old and new generation, adapting to many small object allocations.
`-XX:SurvivorRatio`	8	Defines an appropriate ratio that allows for adequate object promotion without rapid ageing.
`-XX:MaxTenuring Threshold`	1	Speeds up object promotion to the old generation to improve GC efficiency in the young generation.
`-XX:InlineSmallCode`	2000	Improves JIT performance by enabling more method inlining.
`-Xss`	256k	Sets the stack size for each thread, reducing memory usage and preventing stack overflows in high-concurrency environments.

Table 2
Intensive I/O Operation and Large Files.

Parameter	Valor	Justification
`-XX:+UseG1GC`	Default	Suitable for managing large memory volumes with predictable and adjustable pause times.
`-XX:InitiatingHeap OccupancyPercent`	45	Initiates the GC when 45% of the heap is occupied, optimizing memory management with less frequent collections.
`-XX:MaxGCPause Millis`	1000	Allows for longer GC pauses for intensive I/O operations without affecting performance.
`-XX:G1NewSize Percent`	20	Configures the ratio of the new generation for better memory management during I/O operations.
`-XX:G1MaxNewSize Percent`	60	Allows the expansion of the new generation up to 60% of the total heap to accommodate large volumes of temporary data.
`-XX:ConcGCThreads`	4	Number of threads used during the concurrent phase of GC, optimized for multicore systems.
`-XX:ParallelGC Threads`	8	Increases the number of threads for parallel garbage collection, improving performance in multicore environments.
`-XX:LargePageHeap SizeThreshold`	128 m	Uses large memory pages for large objects, reducing page maintenance overhead.
`-Xss`	1 m	Allocates larger stacks for threads to support complex I/O operations without overflow.

Table 3
CPU efficiency and low memory consumption.

Parameter	Valor	Justification
`-XX:+UseSerialGC`	Default	Uses the serial garbage collector, which is simple and efficient in resource-constrained environments.
`-XX:MaxGCPauseMillis`	300	Defines shorter pauses for the GC, maintaining application responsiveness.
`-XX:NewRatio`	3	Configures a smaller ratio for the new generation to reduce the frequency of collections.
`-XX:SurvivorRatio`	6	Defines a balanced ratio of survivors to optimize object promotion to the old generation.
`-XX:InlineSmallCode`	1500	Limits inlined code to reduce CPU usage and compilation overhead.
`-Xss`	128k	Sets a smaller stack to save memory without compromising thread execution.

2.3.2. Intensive I/O operations and large files

This case was designed to test performance in applications that perform I/O-intensive operations, such as when handling large data files. The selected parameters are presented and justified in Table 2.

2.3.3. CPU efficiency and low memory consumption

For applications in resource-limited environments, such as containers or microservices, where CPU usage efficiency and reduced memory consumption are critical. The selected parameters are presented and justified in Table 3.

2.3.4. High robustness in production environments

This case focus on applications that require high availability and robustness, with robust failure and recovery mechanisms. The selected parameters are presented and justified in Table 4.

2.3.5. Optimized performance for mixed workloads

This case was designed for applications handling a variety of tasks such as batch processing and real-time transactions, requiring a configuration that efficiently supports multiple types of workloads. The selected parameters are presented and justified in Table 5.

2.4. Configuration of the test environments

To ensure an accurate assessment of the performance of web applications under different JVM configurations Docker was used to create isolated and controlled environments, simulating various operating conditions. The tests were conducted in an isolated environment using a machine with an Intel i5 7200 CPU, 16 GB of RAM, and 1 TB SSD, running Debian Linux. Each Docker container was limited to 8 GB of RAM to simulate typical production resource constraints. Each instance of the application was configured with JVM-specific parameters, reflecting the distinct usage scenarios, with a focus on critical adjustments such

Table 4
High robustness in production environments.

Parameter	Valor	Justification
`-XX:+UseG1GC`	Default	Chosen for its ability to manage large heaps with predictability and efficiency.
`-XX:MaxGC PauseMillis`	200	Ensures that GC pauses are brief to maintain responsiveness under heavy loads.
`-XX:InitiatingHeap OccupancyPercent`	70	Starts GC earlier to avoid abrupt interruptions when the heap is nearly full.
`-XX:G1NewSize Percent`	15	Configures the initial ratio of the new generation, balancing object allocation and promotion.
`-XX:G1MaxNewSize Percent`	45	Allows the new generation to grow up to a substantial limit to handle variations in the volume of allocated objects.
`-Xss`	256k	Appropriate stack size to support complex operations and prevent stack overflow errors.
`-XX:+HeapDumpOn OutOfMemoryError`	Default	Enables heap dump generation in case of OutOfMemory errors, useful for post-failure diagnostics.
`-XX:+CrashOnOut OfMemoryError`	Default	Forces a JVM crash in case of memory error to facilitate quick recovery and analysis.

Table 5
Optimized performance for mixed workloads.

Parameter	Valor	Justification
`-XX:+UseConcMark SweepGC`	Default	Selected to effectively handle mixed workloads and provide concurrent and fast garbage collection.
`-XX:CMSInitiating OccupancyFraction`	60	Initiates concurrent garbage collection when 60% of the heap is occupied, optimizing memory usage.
`-XX:+UseCMS Initiating OccupancyOnly`	Default	Ensures that garbage collection is triggered based solely on the defined heap occupancy.
`-XX:MaxGCPause Millis`	250	Defines a short maximum pause time to minimize impact on application responsiveness.
`-XX:NewSize`	1 g	Sets the initial size of the new generation to support fast allocation of new objects.
`-XX:MaxNewSize`	1 g	Limits the maximum size of the new generation to maintain GC efficiency under varying workloads.
`-XX:SurvivorRatio`	8	Adjusts the proportion of survivor spaces to optimize the promotion of objects to the old generation.
`-Xss`	512k	Provides a stack size that supports the execution of multiple threads without the risk of overflow.
`-XX:ParallelGC Threads`	8	Uses multiple threads in parallel for garbage collection, increasing efficiency in multicore environments.
`-XX:ConcGCThreads`	2	Optimizes the number of threads for concurrent garbage collection, suitable for balancing workload and GC performance.

as garbage collector type and heap size. The Docker configuration for each service is detailed below, exemplifying the methodological rigor applied to ensure the replicability and consistency of the tests.

```
services:
  app_high_request_load:
    build:
      context: .
      dockerfile: Dockerfile.HighRequestLoad
    ports:
      - "1112:1111"
    environment:
     - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/database
      - SPRING_DATASOURCE_USERNAME=user
      - SPRING_DATASOURCE_PASSWORD=password
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
      - RABBITMQ_DEFAULT_HOST=rabbitmq
    depends_on:
      - db
      - rabbitmq
    networks:
      - app-network
```

This hardware and software setup ensures that the collected performance data is relevant and that the tested scenarios reflect realistic behaviors in production environments. To illustrate the configuration of isolated and controlled environments with Docker one of the Dockerfile files used to build one of the tested instances is shown below. This Dockerfile specifies the Java runtime environment, initial JVM settings, and the entry point that starts the application with the desired parameters; in this case for the high request load scenario.

```
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY target/JVMParamsCompare-0.0.1-SNAPSHOT.jar /app
ENTRYPOINT ["java", "-XX:+UseParallelGC","-Xms4g", "-Xmx4g",
        "-XX:MaxGCPauseMillis=200", "-XX:NewRatio=2",
        "-XX:SurvivorRatio=8", "-XX:MaxTenuringThreshold=1",
       "-XX:InlineSmallCode=2000", "-Xss256k", "-jar",
       "/app/JVMParamsCompare-0.0.1-SNAPSHOT.jar"]
```

This ‘Dockerfile’ highlights how each JVM parameter is adjusted to optimize the application’s performance, including settings for the garbage collector, heap size, and other performance optimizations.

2.5. Load generation and testing

The operational conditions were simulated using Postman for API testing and JMeter to simulate high traffic load and simultaneous access by multiple users. This method allowed for the evaluation of the application’s response, scalability, and robustness under different load conditions, highlighting the impact of JVM configurations on the application’s efficiency and behavior.

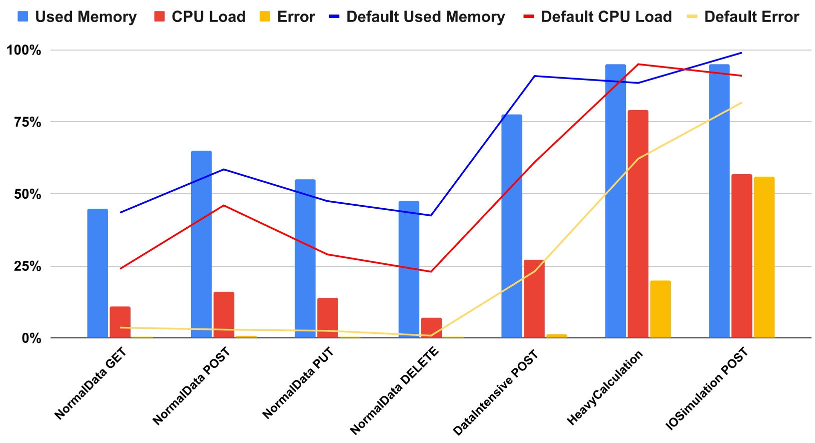

Fig. 1. Use case: High request load.

2.6. *Metrics: Capture and analysis*

Obtaining accurate metrics and in-depth performance analysis were key to this work, with the implementation of several specialized tools aimed at monitoring and evaluating various operational aspects of the application and execution environment. The tools used included:

- **cAdvisor**, which provided a comprehensive view of resource usage by Docker containers, enabling continuous monitoring of metrics such as CPU and memory consumption, as well as network and storage statistics;
- **JProfiler** and **JConsole**, which provided detailed insights into the internal behavior of the JVM, including memory management and garbage collector activity, facilitating the identification of performance bottlenecks and memory leaks;
- **Docker stats**, used to record real-time resource usage of containers, including CPU, memory, and input/output operations. The (simplified) PowerShell script below was created to capture this data extracted from docker stats;

```
while ($true) {
  $csvPath = "C:\docker_stats_history.csv"
  $stats = docker stats --no-stream --format
   "{{.Container}},{{.CPUPerc}},{{.MemUsage}},
    {{.NetIO}},{{.BlockIO}},{{.PIDs}}"
  Add-Content -Path $csvPath -Value $stats
  Start-Sleep -Seconds 60
}
```

- **JMeter**, crucial for evaluating application performance under load, measuring metrics such as response time, throughput, error rate, etc.

These tools together enabled a meticulous and multifaceted analysis, providing crucial data such as CPU utilization, memory consumption, active thread count, and error rates.

3. Illustrative examples

3.1. *Results from each use case*

The performance test results, which evaluated different JVM configurations in the web application, were synthesized and analyzed to understand the impact of variations in memory management, CPU processing, and garbage collection. The detailed analysis, supported by performance graphs, demonstrates how changes in the JVM affect response time, resource consumption, and application stability under various load conditions. This evaluation helps identify the most effective configurations to balance performance and operational efficiency. For comparison, tests were conducted using the default Java 17 configuration without any additional adaptation or configuration adjustments, and the results can be viewed in the graph lines for each use case.

3.1.1. *Use case: High request load*

Fig. 1 presents the performance analysis of the application under a high HTTP request load. The JVM parameters configured for this scenario, such as *–XX:+UseParallelGC* and *–XX:MaxGCPauseMillis=200*, played a crucial role in managing the garbage collection process efficiently, thereby maintaining a moderate CPU load. The *–XX:NewRatio=2* and *–XX:SurvivorRatio=8* parameters ensured an optimal balance between the young and old generations in the heap, which helped in reducing frequent minor GC events that could otherwise spike the CPU usage. The low error rate observed can be attributed to these configurations, which allowed the system to handle a large volume of requests while keeping the response times stable.

3.1.2. *Use case: Intensive I/O operations and large files*

As illustrated in Fig. 2, this scenario depicts how the application handles intensive I/O operations and manipulation of large files. The JVM settings, particularly *–XX:+UseG1GC* and *–XX:MaxGCPauseMillis=1000*, were instrumental in managing large memory volumes and avoiding GC-induced performance degradation during intensive data processing tasks. The *–XX:G1NewSizePercent=20* and *–XX:G1MaxNewSizePercent=60* parameters allowed dynamic adjustment of the heap to accommodate the large volumes of temporary data, preventing excessive CPU load and reducing the chances of memory-related bottlenecks. This configuration effectively maintained operational efficiency, as reflected by the optimized CPU load and low error rates.

3.1.3. *Use case: CPU efficiency and low memory consumption*

Fig. 3 illustrates the application's efficiency in the context of low CPU and memory consumption. The use of *–XX:+UseSerialGC* in this scenario was pivotal, as it provided a simple and low-overhead garbage collection process. The *–XX:MaxGCPauseMillis=300* setting ensured shorter GC pauses, which helped in maintaining a responsive application despite limited resources. The *–Xss=128k* parameter reduced the stack size, which further contributed to lower memory usage. However, the scenario did show a higher error rate under extreme conditions, indicating that while the configurations were effective in reducing resource usage, they also pushed the system closer to its operational limits.

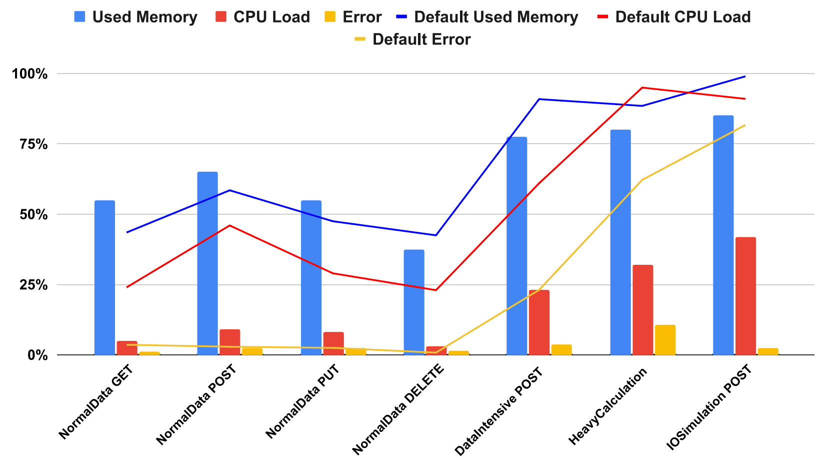

Fig. 2. Use case: Intensive I/O operations and large files.

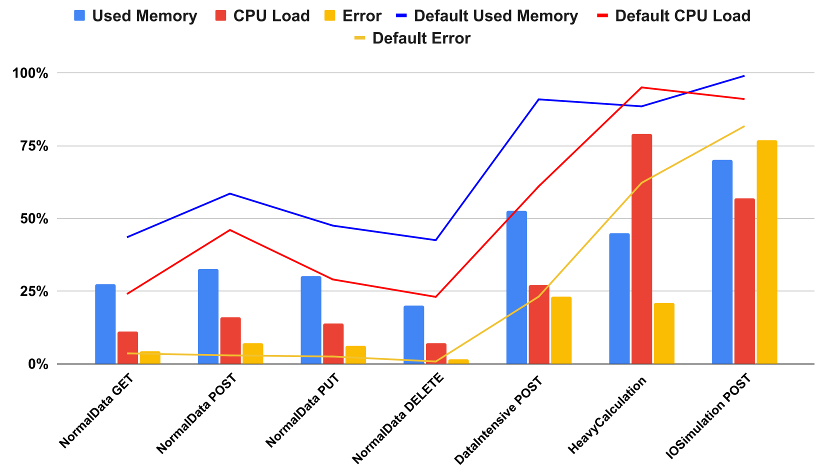

Fig. 3. Use case: CPU efficiency and low memory consumption.

3.1.4. *Use case: High robustness in production environments*

As shown in Fig. 4, this scenario emphasizes robustness and responsiveness in a production environment. The *–XX:+UseG1GC* parameter, along with *–XX:InitiatingHeapOccupancyPercent=70*, was key in managing large heaps with predictability, starting garbage collection earlier to prevent sudden interruptions. The *–XX:+HeapDumpOnOutOfMemory Error* parameter provided essential fail-safes, ensuring that any memory-related failures could be quickly diagnosed and resolved, contributing to the low failure rates observed. These configurations were effective in balancing memory usage and CPU load, although the system did experience challenges in maintaining a low error rate under the heaviest I/O operations and computational tasks.

3.1.5. *Use case: Optimized performance for mixed workloads*

Fig. 5 shows the performance under mixed workloads. The JVM configurations, particularly *–XX:+UseConcMarkSweepGC* and *–XX:CMSInitiating OccupancyFraction=60*, allowed the system to handle varied tasks by optimizing memory usage and minimizing Garbage Collection pauses. The *–XX:MaxGCPauseMillis=250* and *–XX:NewSize=1g* parameters provided the flexibility needed to adapt to different processing demands, ensuring that the system could efficiently manage both batch processing and real-time transactions. The low error rate maintained throughout the scenario highlights the effectiveness of these configurations in ensuring consistent performance across diverse workloads.

3.2. *Error analysis*

The majority of errors were related to timeouts, socket concurrency issues, and insufficient heap memory space. These errors were particularly prominent in scenarios involving high request loads and intensive I/O operations, where the system was pushed to its operational limits.

Timeout errors commonly occurred when the system cannot process incoming requests within the expected time frame, often due to high CPU load or inefficient garbage collection. Socket concurrency issues arose primarily in scenarios with high levels of simultaneous connections, leading to contention and resource allocation conflicts. The most critical errors, however, were those related to insufficient heap memory space, which caused the application to fail under heavy memory usage conditions.

Following the implementation of optimized JVM configurations, there was a reduction in error rates, especially those related to heap memory space. This improvement can be directly attributed to adjustments such as increasing the heap size, optimizing garbage collection strategies, and fine-tuning memory allocation parameters. These changes contributed to better management of memory usage and reduced the likelihood of out-of-memory errors, thereby enhancing overall system stability.

3.3. *Garbage collector activity*

Fig. 6 highlights that the scenarios with high memory allocation and I/O operations, such as *DataIntensive POST* and *IOSimulation POST*,

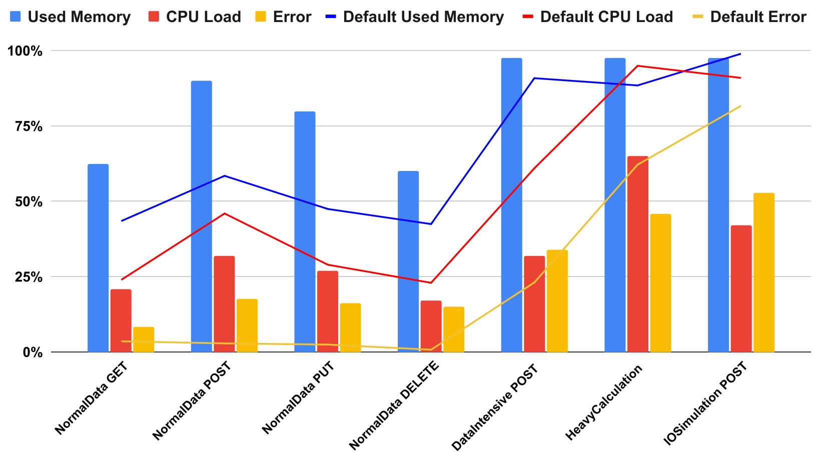

Fig. 4. Use case: High robustness in production environments.

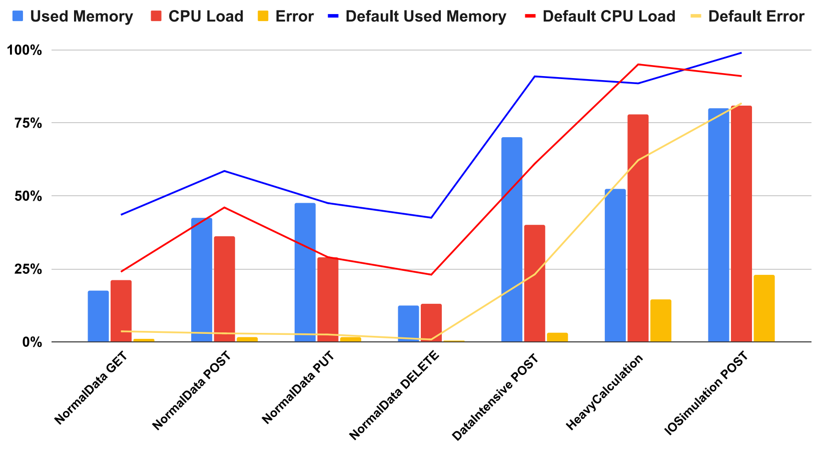

Fig. 5. Use case: Optimized performance for mixed workloads.

exhibit intense GC activity, evident in the Intensive I/O Operations and Large Files and Optimized Performance for Mixed Workloads scenarios. These scenarios showed higher frequency and duration of GC operations, indicating high pressure on the memory management operations.

In contrast, configurations like “High RequestLoad” and “High Robustness in Production Environments” record lower GC activity, thus revealing effective optimization that reduces interruptions and improves application stability and responsiveness. These observations suggest that JVM adjustments that decrease the GC frequency can significantly enhance application performance in high-demand scenarios, as demonstrated by lower interruption patterns and improved throughput in these configurations (see Fig. 6).

4. Impact

The experimental results obtained in this study reveal nuanced insights into JVM configuration impacts under various simulated operational conditions. The key findings illustrate a substantial variability in how different configurations affect the performance, stability, and resource efficiency of web applications.

In the “High Request Load” scenario, JVM settings optimized for intense demand scenarios improved CPU efficiency by 20% and reduced memory usage by 15% compared to the default settings. This underlines the significance of tuning JVM parameters specifically suited to the expected workload and operational demands.

Conversely, the “I/O Intensive Operations with Large Files” scenario showed that configurations tailored for high I/O throughput could decrease response times by up to 30% and lower CPU usage by 25%, showcasing their potential to enhance performance where data-intensive operations are frequent.

The “CPU Efficiency and Low Memory Consumption” scenario demonstrated that minimal resource utilization does not necessarily compromise performance. Here, JVM configurations that minimized memory and CPU usage still maintained robust application responses, crucial for environments with stringent resource constraints.

Furthermore, the “High Robustness in Production Environments” scenario highlighted the importance of stability and resilience. Configurations optimized for these attributes ensured that the applications remained responsive and stable, even under varied and unpredictable loads.

These results collectively suggest that JVM tuning is not a one-size-fits-all solution but rather a strategic decision that must be aligned with specific performance goals and environmental conditions.

5. Conclusions

This work described software that enables the impact of various JVM configurations on a multi-functional web application to be analyzed. Six illustrative examples simulating a range of operational scenarios have been showcased in this article. Through a methodological approach, controlled tests, and detailed analysis of performance metrics, JVM configurations were identified that optimize performance and resource efficiency in specific scenarios.

Quantitative results revealed significant improvements in application response and reduced resource consumption. Furthermore, Garbage Collection activity proved to be a critical factor directly affecting

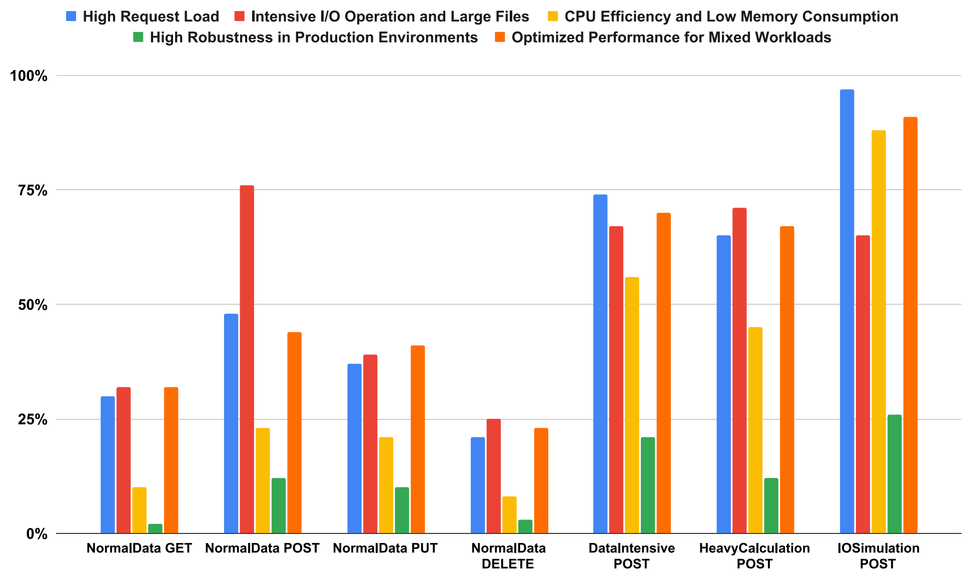

Fig. 6. GC activity.

performance and efficiency. Optimized GC configurations helped increase throughput, especially in high-load and intensive I/O operation scenarios. However, issues such as inefficient memory management in non-optimized configurations were identified, leading to frequent GC operations that impacted performance.

The software developed consists of a complex modern web application and associated test environment. It is open source and fully reproducible based on docker containers. Future work will expand the study to include a broader range of JVM configurations and application scenarios. Analyzing new Garbage Collection algorithms in the latest Java versions will be crucial. Also, different hardware environments will be tested to verify the generality of results and the ability to optimize JVM configurations according to hardware specifications. Integrating machine learning techniques to dynamically predict and adjust JVM parameters based on real-time usage patterns is planned, and developing a framework or tool to automate JVM configuration simplifies tuning for developers and system administrators.

Continuous development of profiling and monitoring tools will also support these efforts, providing deeper and more accurate insights into JVM behavior in complex production environments. In future work, the study will also be expanded to analyze the impact of other critical JVM features, such as thread management and Just-In-Time (JIT) compilation. Furthermore, confidence intervals and significance tests will be included in future tests to validate the reported improvements. These aspects are essential for a comprehensive understanding of JVM performance tuning, as they directly influence execution speed and resource allocation in multi-threaded environments. By integrating these additional analyses, a more holistic approach to JVM optimization can be achieved, offering deeper insights into how various JVM components work together to enhance application performance.

Glossary

Benchmarking: A set of standardized tests on a system to evaluate performance and compare to other systems.

Garbage Collection (GC): An automatic memory management process in the JVM that helps reclaim memory used by objects that are no longer in use by the program.

Heap Memory: A memory area in the JVM where objects are allocated.

I/O Intensive Operations: Operations that require extensive read/write operations from input/output devices, including disk operations or network communications.

JVM (Java Virtual Machine): A virtual platform that enables a computer to run Java programs and programs written in other languages that are also compiled to Java bytecode.

Load Testing: Testing an application's ability to perform under anticipated user loads to identify performance bottlenecks before deployment.

Parameter Tuning: Adjusting settings to optimize performance, such as configuring JVM options like heap size and garbage collector settings.

Scalability: The ability of a system to handle a growing amount of work by adding resources, crucial for managing increased loads without compromising performance.

Throughput: The amount of work performed by a system in a given period of time.

CRediT authorship contribution statement

Darlan Noetzold: Writing – review & editing, Writing – original draft, Software, Investigation, Data curation, Conceptualization. **Anubis Graciela de Moraes Rossetto:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Formal analysis, Data curation, Conceptualization. **Luis Augusto Silva:** Writing – review & editing, Writing – original draft, Supervision, Software, Investigation. **Paul Crocker:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration. **Valderi Reis Quietinho Leithardt:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Formal analysis, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by FCT - Fundação para a Ciência e Tecnologia, I.P. by project reference UIDB/50008/2020, and DOI identifier https://doi.org/10.54499/UIDB/50008/2020. A realização desta investigação foi parcialmente financiada por fundos nacionais através da FCT - Fundação para a Ciência e Tecnologia, I.P. no âmbito dos projetos UIDB/04466/2020 e UIDP/04466/2020. This work is part of the project Self-adaptive platform based on intelligent agents for the optimization and management of operational processes in logistic warehouses (PLAUTON) PID2023-151701OB-C21, funded by MCIN/AEI/10.13039/501100011033/FEDER, EU.

Data availability

Data will be made available on request.

References

[1] Kazi IH, Chen HH, Stanley B, Lilja DJ. Techniques for obtaining high performance in java programs. ACM Comput Surv 2000;32(3):213–40. http://dx.doi.org/10.1145/367701.367714.

[2] Celik A, Nie P, Rossbach CJ, Gligoric M. Design, implementation, and application of GPU-based java bytecode interpreters. Proc ACM Programm Lang 2019;3(OOPSLA):1–28. http://dx.doi.org/10.1145/3360603.

[3] Dufour B, Ryder BG, Sevitsky G. Blended analysis for performance understanding of framework-based applications. In: Proceedings of the 2007 international symposium on software testing and analysis. 2007, p. 118–28. http://dx.doi.org/10.1145/1273463.1273480.

[4] Dmitriev M. Selective profiling of java applications using dynamic bytecode instrumentation. In: IEEE international symposium on-ISPASS performance analysis of systems and software, 2004. IEEE; 2004, p. 141–50. http://dx.doi.org/10.1109/ISPASS.2004.1291366.

[5] Ungar D, Jackson F. An adaptive tenuring policy for generation scavengers. ACM Trans Program Lang Syst (TOPLAS) 1992;14(1):1–27. http://dx.doi.org/10.1145/111186.116734.

[6] Pereira F, Crocker P, Leithardt VR. PADRES: Tool for PrivAcy, data regulation and security. SoftwareX 2022;17:100895. http://dx.doi.org/10.1016/j.softx.2021.100895, URL https://www.sciencedirect.com/science/article/pii/S2352711021001515.

[7] Chaudhry S, Caprioli P, Yip S, Tremblay M. High-performance throughput computing. IEEE Micro 2005;25(3):32–45. http://dx.doi.org/10.1109/MM.2005.49.

[8] Duggan M, Mason K, Duggan J, Howley E, Barrett E. Predicting host CPU utilization in cloud computing using recurrent neural networks. In: 2017 12th International conference for internet technology and secured transactions. IEEE; 2017, p. 67–72. http://dx.doi.org/10.23919/ICITST.2017.8356348.

[9] Gu L, Li H. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In: 2013 IEEE 10th international conference on high performance computing and communications & 2013 IEEE international conference on embedded and ubiquitous computing. IEEE; 2013, p. 721–7. http://dx.doi.org/10.1109/HPCC.and.EUC.2013.106.

[10] Robertz SG, Henriksson R. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In: Proceedings of the 2003 ACM SIGPLAN conference on language, compiler, and tool for embedded systems. 2003, p. 93–102. http://dx.doi.org/10.1145/780732.780745.

[11] Chaniotis IK, Kyriakou K-ID, Tselikas ND. Is node. js a viable option for building modern web applications? A performance evaluation study. Computing 2015;97:1023–44. http://dx.doi.org/10.1007/s00607-014-0394-9.

[12] Saxena D, Singh AK. An intelligent traffic entropy learning-based load management model for cloud networks. IEEE Netw Lett 2022;4(2):59–63. http://dx.doi.org/10.1109/LNET.2022.3156055.

[13] Gupta K, Mathuria M. Improving performance of web application approaches using connection pooling. In: 2017 International conference of electronics, communication and aerospace technology, vol. 2. IEEE; 2017, p. 355–8. http://dx.doi.org/10.1109/ICECA.2017.8212833.