Chapter 1

EXPERIMENTAL STUDY FOR EVALUATING THE PERFORMANCE OF JAVA VIRTUAL MACHINES IN APPLICATION INTEGRATION

Daniela L. Freire *[1], **Rafael Z. Frantz**[1], **Eldair F. Dornelles**[1], **Fabricia Roos-Frantz**[1], **and Sandro Sawicki**[1]

[1]Unijuí University, Department of Exact Sciences and Engineering, Ijuí, RS, Brazil

PACS 05.45-a, 52.35.Mw, 96.50.Fm. **Keywords:** Benchmark, API Java concurrency, Java Virtual Machine, Multithread, Performance.

Abstract

To be competitive, companies face the challenge of keeping their business processes running efficiently. Integration platforms are tools designed to provide integration solutions to achieve these goals. Many of these platforms are developed in the Java language, which offers a complete class library to concurrent execution of tasks that compose the solutions. Thus, the better the performance of the platforms, the more efficient the business process becomes. The technology used by these platforms is one of the factors that influences this improvement, including the Java virtual machine. Meanwhile, there are different implementations of the Java virtual machine,

*Corresponding Author Email: dsellaro@unijui.edu.br

and the choice of the most appropriate is a challenge for enterprises. Such choice should consider non-deterministic factors, such as the compiler used, the scheduling policy, and the interruptions caused by other software. In our literature review, we did not identify proposals that help decision making concerning the Java virtual machine that best manages the concurrent execution of the tasks of an integration solution. This article evaluates the behaviour of the concurrent execution of tasks in different implementations of Java virtual machines. The performance measurements were analysed by rigorous statistical techniques: Analysis of Variance and HSD comparison test of Tukey averages. The study showed that the Oracle virtual machine was the most efficient. However, the difference between the performances concerning the concurrent execution of tasks was not significant in statistical perspective.

1. Introduction

The set of software applications, which support the business processes of companies, is usually quite heterogeneous since the applications were developed or acquired over time without any concern for integration between them. Besides, the incorporation of cloud computing services has made this set even more heterogeneous [56]. For business processes to receive quick and reliable responses, the applications need to work together and meet performance-quality requirements.

Enterprise Application Integration is the research field that provides methodologies, techniques and tools to support the development of integration solutions, which allow the different applications to work synchronously [21]. Integration platforms are specialised software tools that provide practical means for software engineers to design, deploy, execute, and monitor integration solutions [22, 23]. The integration solutions orchestrate a suite of applications to keep data synchronised or create new functionality; ideally, such solutions should have no impact on applications [26]. In the last decade, open-source integration platforms implemented with the Java language, inspired by the architectural style of *pipes-and-filters* [4] and the conceptual patterns of integration documented by Hohpe and Woolf [27]. Integration patterns document a set of best practices for performing tasks that help solve recurring application integration issues. In an integration solution, filters are implemented by tasks, which carry an integration pattern; and pipes are implemented by communication channels through which data flows, encapsulated in messages. Amongst the elements that make up an integration platform, the runtime system is the responsible for executing integration solutions, so its performance is one of the most observed factors in the decision making process concerning the choice of an integration platform [7, 16, 34].

With the growth of multi-processor computer architectures, also known as multicore, software development has adopted multithread programming in applications [51]. Besides, several tools have been released to improve the performance of existing applications [31]. The first experiments in programming with parallel execution of tasks were based on the principles of concurrency used in the operating systems. Later, new languages have been introduced bringing multithread programming, as Java language, which provides a complete class library (API) to create and make use of threads [12]. Threads are basic CPU units that concurrently execute parts of the same program, so a thread is the smallest sequence of programme statements that can be managed by runtime system [53].

The Java language continues to evolve and expand, including new features, to extend its support to the concurrency to execute software tasks and thus, to follow the trends of contemporary computing. The *concurrency utilities* package was a significant contribution introduced in version 5 of Java and contains the Executor API, which encapsulates the thread creation and management functions [48]. Many integration platforms have been developed in the Java language, such as Mule [15], Camel [29], Spring Integration [18], Petals [52], WSO2 [30] and Guaraná [20], Fuse [44], ServiceMix [35, 45].

Despite the advantages, multithread programs are more complicated because they must meet both the quality attributes and the previously configured performance parameters. Such programs should provide adequate performance so that they can achieve a shorter response time and a higher workload in the execution of tasks. In order to do so, it is also necessary to know the program code, and to consider the technology used in its development [11]. This technology is related to the Java Virtual Machine (JVM), which has different implementations developed by various companies and software communities. Comparing the performance of these JVMs, concerning the management of the threads in the concurrent execution of tasks, is a challenge since there are several non-deterministic sources, such as the compiler used, the scheduling policy, interruptions caused by other programs that influence it [25].

This chapter presents an experiment with the Executor API and extends previous researches on JVMs [14]. First, a detailed study regarding the Executor API, and then a performance analysis of three different implementations of JVMs. The selection of the JVMs follows the following criteria: (i) Windows operating system and the x64 based processor, (ii) open source, (iii) free license and (iv) updated version. The implementations of JVMs analysed were: Oracle HotSpot Java [1], RedHat OpenJDK [2], Zulu Blue Systems [3]. The performance metrics of the study were CPU times, system times and use times. Such metrics were collected in the execution of the same program, which represents an integration solution, running in the three JVMs. Then, the statistical analysis of the resulting data was performed, following the techniques of analysis of variance - ANOVA [40] and averages comparison - Tukey test [54].

The remainder of this chapter is organised as follows. Section 2. presents a brief theoretical reference on JVMs, multithreaded programming, and the statistical techniques used in the chapter. Section 3. discusses work related to performance evaluation of JVMs. Section 4. reports the experiment on the performance of the JVMs concerning thread management in the concurrent task execution of an integration solution simulation. Finally, Section 5. sets forth our conclusions.

2. Background

In this section, we discuss the JVM concept, multithread programming in Java, especially in API Executor and present a brief approach regarding Variance Analysis and Tukey test for comparison of averages.

2.1. Java Virtual Machine

Java is a complete development and execution platform, composed of three elements: the JVM, a set of APIs and the programming language. JVM is an application that abstracts both the hardware layer and the communication layer with the operating system. Thus, by using the virtual machine concept, the Java compiler generates an executable program for a generic, non-physical virtual machine, which has its machine instructions and its APIs. The function of a JVM is to execute instructions of generic machines on the operating system, and on the specific hardware under which the JVM is running. Thus, it is necessary to install the JVM suitable for the operating system and device used. Figure 1 exemplifies the execution flow of a Java program that can run on both the Windows and Linux operating systems [50].

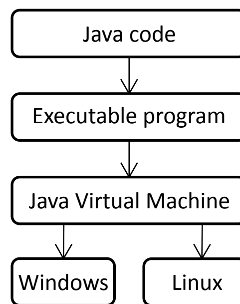

Figure 1. Execution flow of a Java program.

A JVM is a specification, i.e., a set of standards for software development so that a manufacturer or even a developer can write their virtual machine for Java. Java HotSpot by Oracle, OpenJDK by RedHat, Zulu by Azul Systems are examples of these JVMs for the Windows operating system. Each of these manufacturers tries to improve their implementation by using different strategies such as compiler improvement, memory management, thread scheduling, thus promoting market competitiveness and offering more options for developers [50].

Three elements compose a JVM: a ≪class loader≫, a ≪heap≫, and a ≪runtime system≫. The ≪class loader≫ loads Java API classes and those of the program to be executed; the ≪heap≫ is the data memory region that stores the objects and classes; and, the ≪runtime system≫ is the element responsible for interpreting instructions encoded in the JVM format. This architecture is shown in Figure 2.

The core of the JVM is the runtime system, which behaviour is defined by a set of instructions. Each program thread is an instance of the runtime system that executes *bytecodes* or native methods. The first is the abstract encoding of a program produced by a compiler when the source code is processed. The latter are codes written in another language and compiled into machine code native to a particular processor. In addition to the threads used to run the program, the JVM uses threads for different purposes, such as garbage collector threads, which do not have to be instances of the runtime system [43].

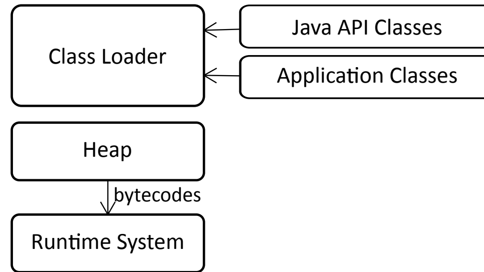

Figure 2. Architecture of a JVM

2.2. Multithreads Java

The use of threads in applications is mainly intended to enable the concurrency of the tasks contained in those applications so that when some of the threads are blocked, others may be running. Threads can share the same address space for data and are also easily created and destroyed as there are no resources associated with them. This ease of creation and destruction is useful when the required number of threads changes dynamically and quickly. Besides, threads provide a performance gain in applications that have high computational effort tasks and tasks that have input and output (I/O) interactions because they allow the execution of these two types of tasks to overlap [53].

The resources for specifying concurrent tasks, offered by some programming languages, were not considered comfortable by software engineers [13]. Historically, programming with parallel execution of tasks was based on the competition principles of operating systems. Subsequently, languages that facilitate such programming have appeared, such as the Java language. Thus, the developer can build programs that contain execution threads. This concurrent programming using threads is called multithread.

Multi-threaded programming support has been one of the most important innovations in Java and is still one of its key strengths. However, until version 5, the features offered left the management of the threads under the responsibility of the programmer, in the programs that allowed the competition of tasks. In version 5, the *concurrency utilities* package, also referred to as the *API concurrency* package, was added, which includes features for synchronisation that facilitated the use of threads, such as synchronisers, thread pools, blockers. This package was significantly extended in version 7 when it introduced the *Framework Fork/Join*, an important feature that facilitates the creation of programs that use multiple processors, thus allowing the actual parallelism in the execution of the threads. In version 8 new features related to API *concurrency* were added, such as the classes: `ConcurrentHashMap`, `ConcurrentLinkedQueue` and `CopyOnWriteArrayList` [48].

Version 9 has added API Flow to concurrency utilities, which implements reactive programming that handles the publisher/subscriber pattern (*publisher/subscriber pattern*). Until version 9, for a thread to be interrupted, there should be a global stop for all threads; as of version 10, the interrupt can be done individually for a thread. The Java language continues to evolve and expand to meet the needs of the contemporary computing environment.

2.2.1. Executors Java

In Java, Executors are objects that encapsulate the threads creation and management functions and compose the *java.util.concurrent* package from the *API concurrency*. Figure 3 shows the class diagram containing the interfaces and most used classes of the Executor API. At the top of the hierarchy is the `Executor` interface, which contains the execute() method to start a thread. The ExecutorService interface extends the Executor interface with methods that manage and provide a complete set for executing asynchronous tasks. This interface provides methods to manage queues and schedule tasks and allows the cancelling of tasks by using the shutdown() method. There are three implementations for the `ExecutorService` interface: `ThreadPoolExecutor`, `ScheduledThreadPoolExecutor`, and `ForkJoinPool`. The first two classes allow the configuration of thread pools in different ways. The `ForkJoinPool` class provides an executor designed to handle the instance of ForkJoinTask and its subclasses, which achieve high throughput for high-computation tasks using parallel processing. The ScheduledExecutorService subinterface and associated interfaces add support for scheduled task executions and periodic executions.

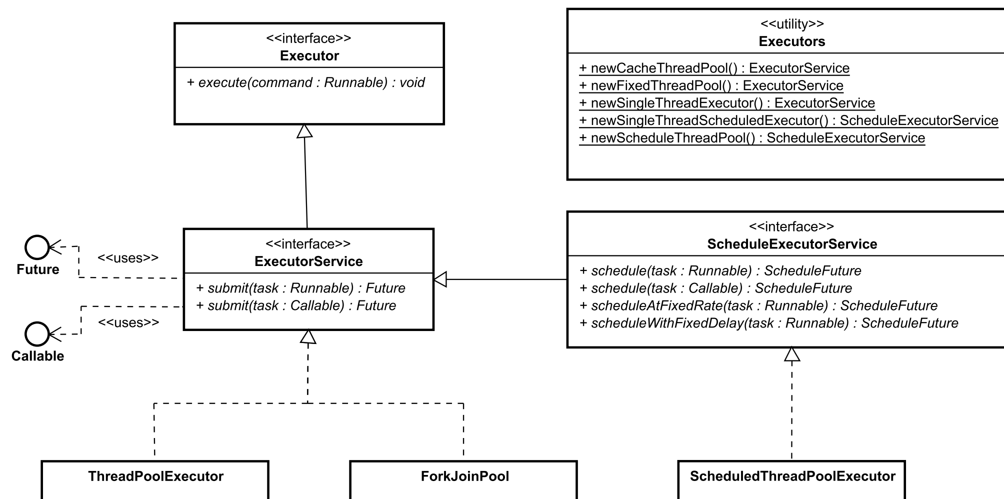

Figure 3. Executor API Class Diagram

The *java.util.concurrent* package also defines the utility class `Executors`, which includes static methods to simplify the creation of executor objects. Related to the performers are the interfaces `Future` and `Callable`. The first allows the cancelling of the execution of a task and finds out if the execution ended successfully or error. The latter returns a value for a thread at the end of this thread's execution. An application can use Callable objects to compute results that are returned by the executed thread that is very useful in coding many types of numerical computations, in which partial results are computed simultaneously. Besides, such objects can be used to execute a thread that returns a status, indicating if the thread execution has been completed. A Callable task is executed by an `ExecutorService`, through the submit() method, and the result is returned through an object of type Future. The `ScheduledExecutorService`

interface provides methods such as: schedule(), scheduleAtFixedRate(), and scheduleWithFixedDelay(). The schedule() method creates and executes a ScheduledFuture object, which is activated after the defined delay and is used to extract a result or cancel execution. The scheduleAtFixedRate() and scheduleWithFixedDelay() methods, on the other hand, create tasks that periodically run until they are cancelled. The first with a fixed time and the last, with delays between the executions.

2.2.2. Factory Metrics and Utility Methods

The Executors class extends the `Object` root class and has factory methods and utility methods. Factory methods create objects with predefined settings, while utility methods allow the developer to customise the thread pool configuration. The factory methods of the Executors class return instances of the ThreadPoolExecutor class and are detailed as follows:

- `newSingleThreadExecutor()`: Creates an executor that uses a single thread running on a non-limited task queue. Tasks are processed sequentially, and only one task is active for a previously determined time. Generally, this type of thread pool configuration is used in simple scenarios or in those where subsequent processing is desired. It is not suitable for scenarios where many tasks need to be processed concurrently or simultaneously.
- `newSingleThreadScheduledExecutor()`: Creates an executor, which uses a single thread, but can schedule tasks to run after a predefined delay or run periodically at previously determined intervals of time. Like the newSingleThreadExecuto() method, the tasks are processed sequentially, and only one task is active for a previously determined time.
- `newFixedThrheadPool()`: Creates an Executor with a fixed number of threads, which is passed as a parameter. It is possible to reuse threads that have already been created, and all threads operate over the same unlimited task queue. If additional tasks are submitted when all threads are active, the tasks will wait in the queue until a thread becomes available. The threads in a pool will exist until they are explicitly cancelled. This thread pool configuration allows greater control of the computational resources used, since the number of threads is predetermined.
- `newScheduledThreadPool()`: Creates a thread pool, usually with more than one thread, and can schedule tasks to run after a predefined delay or run periodically at predetermined intervals of time.
- `newCachedThreadPool()`: Creates a thread pool that accepts new tasks, increasing the number of threads when none is available, or reusing previously created threads. Threads that are idle for sixty seconds will be terminated and removed from the cache. This type of thread pool configuration typically improves the performance of programs that execute many short-lived asynchronous tasks. In scenarios where many tasks are being processed simultaneously, there is a risk that the executor tries to create more threads than is possible with the number of physical resources available on the machine.

Instances of the class `ThreadPoolExecutor` returned by the factory methods of class `Executors`, are previously configured. So when a custom instance of ThreadPoolExecutor to suit a specific situation is needed, the constructor methods and configuration items should be used, because they are more flexible for this configuration, such as `corePoolSize` , `maximumPoolSize`, `keepAliveTime`, `unit`, and `workQueue`. The `corePoolSize` indicates the number of threads held in the pool, even though the thread is idle. The `maximumPoolSize` indicates the maximum number of threads allowed in a pool. The `keepAliveTime` determines the maximum number of time a surplus thread is idle, waiting for a new task. Surplus threads are those created in addition to the quantity defined in `corePoolSize`. The `unit` indicates the time unit for the `keepAliveTime` argument. Finally, `workQueue` defines the type of queue used to hold jobs before execution.

2.2.3. Blocking Queue

The *java.util.concurrent* package includes the `BlockingQueue` interface that allows the definition of different types of task queues for different styles of software architectures, such as: messaging, producer-consumer, parallel tasks and other related competition. The main types of queues are: `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, and `DelayQueue`. The first works as a limited *buffer*, the tasks are kept as elements of an array and are selected by the *First-In-First-Out* policy. In the second type, each insertion operation of a task must wait for a corresponding removal operation from another thread and vice-versa. The third type queues are unlimited and organised in a specified priority order, where, at the beginning of each of these queues, is the highest priority task. In the fourth and last type, the queue's tasks have a predefined delay for execution and can only be retrieved from that queue when the queue expires, at the beginning of the queue is the task that has expired the longest. In Java Executors objects, the use of the task queue interacts with the size of the thread pool as follows:

- if the thread number running in the pool is fewer than the thread number set in `corePoolSize`, then, the executor will always prefer to add a new thread than to queue more tasks;
- if the thread number running in the pool is equal to or greater than the thread number set in `corePoolSize`, the executor will always prefer to queue more tasks than adding a new thread;
- if a task cannot be queued, a new thread will be created; if the thread number running is less than the thread number set to `maximumPoolSize`, the task will be rejected.

2.3. Statistical Reference

The methodology for evaluating and analysing the performance of a runtime system should consider random sources to avoid distorted or even wrong results [6]. The reference literature suggests the use of statistical theory in the analysis of experimental data on the performance of runtime systems [25] because statistical reasoning is an appropriate resource to deal with the non-determinism present in engines such as Java [19].

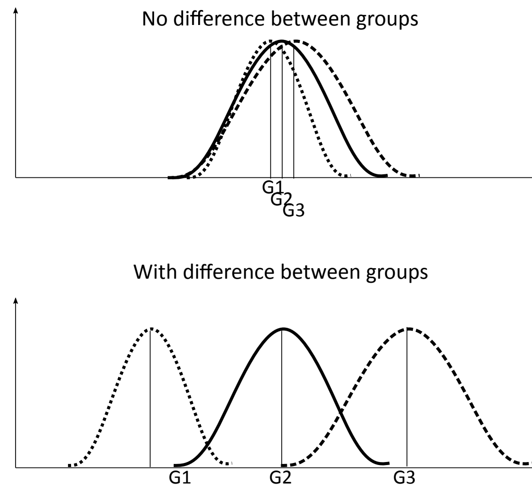

Figure 4. Analysis of variance

This theory classifies the errors of an experiment into two types: the thematic and the random. Thematic errors are those caused by an experimental error or some incorrect procedure during measurements. It is up to the investigator to control and eliminate such errors, since they can invalidate the results, even in the cases where the statistical analysis was performed. Random errors are unpredictable and non-deterministic and may come from external sources unrelated to what one wants to measure. Although it is not possible to predict random errors, it is possible to develop a statistical model to describe its effect on experimental results. ANOVA is a statistical technique that allows differentiating amongst the variations found in a set of measurements of an experiment that are derived from random factors and which are due to real differences between the alternatives being analysed. For the application of the analysis of variance, it is assumed that the observations are independent, the groups compared have the same variance and the errors are independent and come from a normal distribution with average zero and constant variance.

Therefore, the ANOVA technique separates the total variation in two: the first is the variation observed within each alternative studied, which is presumed to be a result of non-deterministic factors, and the second is the difference found between the alternatives considered. If the variation between the alternatives is greater than the variation within each alternative, it is concluded that there is a statistically significant difference between the alternatives [38].

The two graphs, in Figure 4, show the distribution of the averages of any three groups, called: G1, G2, G3. The first graph shows the average of distributions, where there is no statistically significant difference between the three groups and the second graph shows the average of distributions, where there is a difference between the groups.

When there is this difference in the results of the experiment, there are techniques to find which alternatives differ and which ones are similar to the others. Some of these techniques are considered stricter, others less. Those considered stringent are those where there is a significant difference between alternatives, while the least rigorous are those that find the least difference between them. The Tukey test (Honestly Significantly Different-HSD) is a

Table 1. Literature Review.

Year	Collected	Select	References
2010	5	1	Krieger and Strout [36]
2011	15	2	Chen et al. [8], Kulkarni [37]
2012	5	1	Sartor and Eeckhout [47]
2013	5	-	-
2014	10	2	Ganesan et al. [24], Ueno [55]
2015	18	2	Huang et al. [28], Patros et al. [42]
2016	15	2	Magalhães et al. [39], Mostafa et al. [41]
2017	1	-	-
2018	1	-	-

test considered more rigorous and is adopted in the literature in experiments with the JVMs since it is a simple approach [25].

3. Related Works

In this section, we present our literature review, which identified proposals that analyse the performance of Java virtual machines. Initially, we describe the research methodology used to select the works, and then, each related works is mainly discussed. This review had three activities: ≪collection of references≫, ≪selection of articles≫ and ≪discussion of related works≫. In the first activity, a scientific database was queried using a search string. In following, articles were selected for the studies using predefined inclusion criteria. Then, the selected articles were analysed and compared with the proposal of this chapter.

3.1. Reference Collection

In this activity, the SCOPUS database was used to search for titles, abstracts and keywords of articles, using the following search string:

(“java virtual machine”) AND (“performance” OR “benchmark” OR “comparison” OR “evaluation”) AND (“thread” OR “executor”)

This study sought for articles published from 2010 to 2018, written in English, in the disciplinary area of Computer Science. The survey returned 75 different results, covering several journals and conferences.

3.2. Article selection

The inclusion criterion “to be an experimental article about JVM performance” was used to select the articles. The titles and abstracts of these 75 articles were carefully read and, in the end, 65 were excluded, leaving ten selected articles. The selected articles were grouped per year, cf. shown in Table 1.

3.3. Discussion of related works

Magalhães et al. [39] compared the implementation of multithread in procedural programming languages with implementation in oriented languages. The authors also investigated the computational overhead in virtual machines and their impact on the performance of these machines with the use of the Just-In-Time (JIT) compiler. Their work compares performance between types of programming languages, with or without the use of the JIT compiler; while the proposal of this chapter focuses on the comparison between JVMs, concerning multithread programming. Mostafa et al. [41] proposed a fairer approach to scheduling CPU usage by assigning weights to virtual machines, in which each JVM uses the CPU in proportion to the weight assigned to it. An experiment was carried out through simulations, in which the execution time and CPU time in the execution of tasks by three virtual machines were measured to prove the effectiveness of the proposal in relation to the current scheduling. Each virtual machine receives the same CPU time regardless of the complexity of the task being executed. Their work concerns fair scheduling for virtual machines and uses a virtual machine implementation for the tests, while this proposal is concerned with the performance of the virtual machines regarding thread management. Besides, the experiment was performed with JVMs implemented by companies and the Java community.

Patros et al. [42] proposed a tool to measure data from contention of threads blocks and incorporated it into the IBM JVM. The measured data is used to identify bottlenecks and, to accelerate the execution of the programs in the JVM. The authors affirm that the tool can be incorporated into other JVMs and performed experiments to validate the proposed tool, achieving a good precision in the measurement of the average waiting time of the blocks and detection of bottlenecks. The author's work addresses one aspect of multithread programming, the time a thread waits until it is released from the lock, differing from the proposal of this chapter, which intends to compare the performance of thread management in three current implementations of JVMs. Huang et al. [28] performed experiments comparing the execution of a new compilation policy to JVM, which interrupts interpretation when the compilation queue is too long. The JVM used in the tests was Oracle HotSpot, in which the authors propose a new policy with the current JIT compilation policy and found that the new one achieves better performance when the number of processor cores increases. Their objective was to explore the impact between the number of processor cores and the performance of the JVMs, with different compilation policies; the purpose of this chapter is to verify that different implementations of JVMs have the same performance in the execution of the same multithread program.

Ueno [55] analysed performance characteristics of multicore computers in running multithreaded programs on virtual machines over real machines using the Linux operating system and the results showed that the performance difference is related to the number of processor cores. The author's work compared the performance of virtual machines with real machines, while the proposal of this chapter compares JVMs with each other. Ganesan et al. [24] addressed the bottlenecks that arise in multicore task scheduling using a single JVM. They also explored the use of the Java *concurrency utilities* package, leveraging their resources for multithreaded programs, and experimented with using the Java class `ConcurrentHashMap`. The experiment compared the implementation of that class in

version 8 with the implementation in version 7 and found a significant improvement in program scalability of about 57% in version 8. Their work compared the implementation of the class `ConcurrentHashMap` between Java versions, while the proposal of this chapter examines the implementation of another class of the Java concurrency utilities package between different implementations of JVMs.

Sartor and Eeckhout [47] experiments in a JVM called Jikes Research Virtual Machine to investigate the performance of Java programs by mapping program threads to virtual machines in a multicore environment and `multi-socket`. The authors tested and compared various strategies to optimise runtime and minimise energy consumption. The strategies were: variation of the number of threads over a single socket and over two sockets; variation of the core frequency and the Java program's clock speed; isolation of threads during compilation during startup and during steady state time; fixing the threads to a socket and isolating the *garbage collector*. Their work varied the execution strategies for which JVMs concerning program performance; whereas, the proposal of this chapter evaluates different implementations of JVMs regarding the performance of thread management. Kulkarni [37] investigated JIT compilation strategies to verify the impact on single-processor and multi-processor performance in JVMs. Through new configurations in the JVM HotSpot, the author experiment varied JIT compilation strategies, also changing the number of threads. Their work has tried strategies to improve compilation performance in a JVM; while this proposal, experienced different implementations of JVMs, comparing their performances.

Chen et al. [8] studied the performance and scalability of Java multithread programs on multicore machines, identifying possible bottlenecks. Experiments were performed with HotSpot OpenJDK, varying the numbers of processor cores and varying the number of threads of the programs. The authors analysed the performance of a JVM with several programs; while the proposal in this chapter analyses different implementations of JVMs with a single program. Krieger and Strout [36] shared an experience of parallelizing an irregular scientific program written in Java, executed in a multicore machine, using a JVM implementation, called *Molecular Workbench*. The authors report the experience with the parallelism of the JVM in multicore hardware using the libraries of the Java *java.util.concurrent*. The purpose of this chapter is to study the behaviour of JVMs in the execution of a multithread program that uses concurrency utilities from Java.

4. Performance Analysis

This section describes the environment, the variables, and the methodology used in the experiment to answer the research question. This section also presents the statistical study of the results through the analysis of variance and Tukey test. The methodology adopted in this experiment was based on the works of Jedlitschka and Pfahl [32], Wohlin et al. [57] and Basili et al. [5], and the procedures for controlled experiments in software engineering study field. The literature classifies this type of experiment as a terminating simulation, in which the output is a function of the initial conditions. Terminating simulations are usually statistically analysed by the method of the replicates, where 20 to 30 repetitions are sufficient to obtain a population average, in the use of the distribution with more extreme values than a normal distribution [46].

4.1. Search Question

The experiment tried to answer the following research question:

Is the performance difference between different implementations of Java virtual machines statistically significant concerning thread management?

The hypothesis is that there is a difference in thread management performance between JVMs since different companies or development communities implement them, and the Java specification does not include quality and performance parameters. Differences in performance are expected to be statistically significant since distinct optimisation strategies are adopted in the implementation and execution of the codes of the different JVMs.

4.2. Environment Configuration

The experiment was performed on a 2.50GHz Intel ® Core i5-2450M 2.50GHz, with 2 physical cores and 4 logical processors, 4G of RAM, with the Windows 8.1 Pro operating system.

4.3. Variables

In this section, the independent and dependent variables of the experiment are described.

An independent variable is an attribute that defines the study configuration and is varied and controlled in the experiment. The independent variable of our study is:

JVM: Java virtual machine implementation. The tested values were Java HotSpot 25.101-b13-JDK.8.0.101 from Oracle, OpenJDK 25.102-14-jdk1.8.0.102 from RedHat, Zulu 8.17.0.3-jdk8.0.102 from Azul Systems.

A dependent variable is influenced by the values of the independent variables and it is measured for later analysis. In this study they are:

System Time: time consumed by the operating system code to run the experiment code.

Use time: time spent running the lines of code that are in the experiment.

CPU time: sum of the times described above: system time and use time. It does not count I/O or time spent running other programs [49].

4.4. Execution and Data Collection

The experiment consisted of developing a program that simulated the behaviour of the runtime system of an integration platform in the execution of tasks of an integration solution. The program has a task generator, a thread pool, and a monitor. The first one generates ten tasks simultaneously for a task queue; the second performs the tasks; and the third captures and records time measurements; according to Figure 5.

In this experiment, a task is a Java class that identifies prime numbers within the range of 1 to 10000; the thread pool consists of two threads, implemented through the

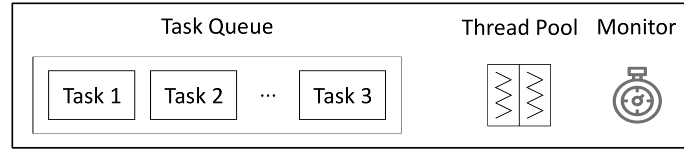

Figure 5. Java program elements.

newFixedThrheadPool() method of the *java.util.concurrent* package; the monitor is a class that captures and records the CPU time, system time, and use time consumed in running the experiment. The experiment was repeated 25 times, and with each repetition, the thread pool performs ten tasks, and the monitor records the times. The source codes of classes developed and used in this experiment are publicly available [1].

The software *Genes* [9], version 2015.5.0, was used to process the descriptive statistics, ANOVA techniques and the Tukey technique of the times measured in this study. *Genes* is a program for processing and analysing data based on experimental statistics.

4.5. Results

In this section, we present the results of the experiment and the statistical analysis applied to these results. The values of the averages of the times in seconds of the three JVMs, in the 25 repetitions, are illustrated in the graphs of Figure 6, 7, and 8. In these graphs, the abscissa axis contains the JVMs and the ordinate axis, the measured times.

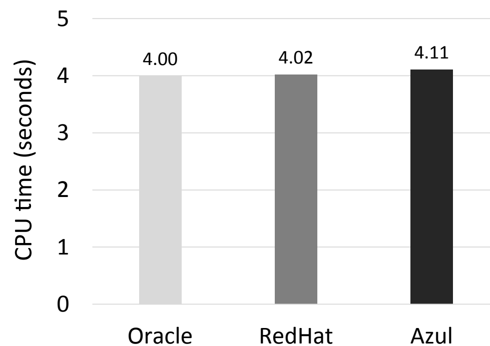

Figure 6. Average of CPU time.

The average CPU times consumed were 4 seconds by JVM Oracle, 4.02 seconds by JVM RedHat and 4.11 seconds by JVM Blue, cf. Figure 6. Figure 7 shows the average system times, consumed 2.97 seconds by JVM Oracle, 2.96 seconds by JVM RedHat and 3.04 seconds by JVM Blue. Figure 8 displays the average use times, consumed 1.03 seconds by JVM Oracle, 1.06 seconds by JVM RedHat and 1.07 seconds by JVM Blue.

[1] www.gca.unijui.edu.br/publication/data/ns-jvm.zip

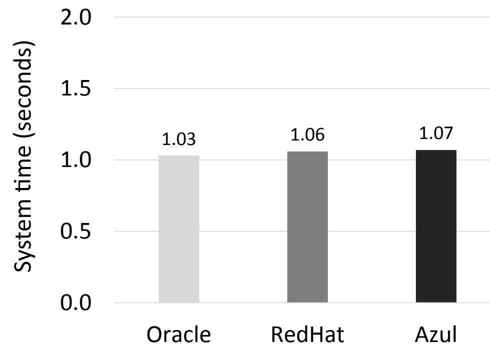

Figure 7. Average of system time.

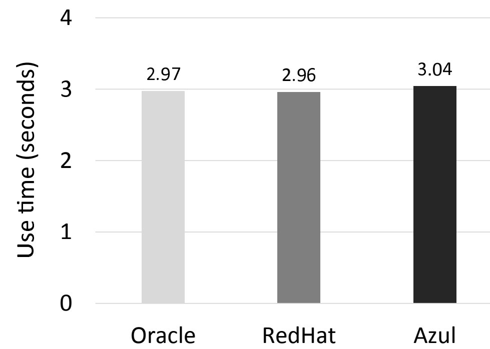

Figure 8. Average of use time.

Tables 2, 3 and 4 present the analysis of variance dependent variables, where ns means that the result is not significant by Fisher-Snedecor F Probability and 5 % error probability level; GL means the degree of freedom for the experiment; QM means average square; and CV, coefficient of variation. The total degree of freedom equals 74, calculated by subtracting one from total results, where the total of results is the number of repetitions multiplied by the total number of JVMs, $25\ x\ = 375$. The degree of freedom equals 2 for JVMs, calculated by subtracting one from the total number of JVMs. The degree of freedom for error is equal to 72, calculated by the difference between the degree of freedom and the degree of freedom of the JVMs. The Table 2 presents the analysis of variance of CPU time, showing the average square of 0.092 for the JVMs and 0.081 for the error. Overall, the average was 4.042 seconds, and the coefficient of variation was 7.066 %. The Table 3 presents the analysis of system time variance, showing the average square of 0.013 for the JVMs and 0.014 for the error. The overall average was equal to 1.054 seconds, and the coefficient of variation was equal to 11.357 %. The Table 4 presents the analysis of use

time variance, showing the average square of 0.049 for the JVMs and 0.047 for the error. The overall average was equal to 2.987 seconds, and the coefficient of variation was equal to 7.280 %.

Table 2. Analysis of variance of CPU time.

Sources of variation	GL	QM
JVM	2	0,092 ns
Error	72	0,081
TOTAL	74	
Overall average	4,042	
CV (%)	7,066	

Table 3. Analysis of variance of system time.

Sources of variation	GL	QM
JVM	2	0,013 ns
Error	72	0,014
TOTAL	74	
Overall average	1,054	
CV (%)	11,357	

Table 4. Analysis of variance of use time.

Sources of variation	GL	QM
JVM	2	0,049 ns
Error	72	0,047
TOTAL	74	
Overall average	2,987	
CV (%)	7,280	

Tables 2, 3 and 4 present the test of comparison of averages by the test of Tukey dependent variables, where averages followed by the same letter in the “Average” column do not differ statistically from each other, at a 5 % error probability level by the Tukey model.

The Table 5 presents the comparison test of averages for CPU time, where the JVM Oracle averaged 3.998 seconds, JVM RedHat averaged 4.018 seconds, and JVM Blue obtained an average of 4.011 seconds. Table 6 presents the test of comparison of averages for system time, where the JVM Oracle obtained the average of 1,028 seconds, JVM RedHat obtained the average of 1,062 seconds, and Blue JVM obtained the average of 1,072 seconds. Table 7 presents the test of comparison of averages for the use time, where the JVM Oracle obtained the average of 2.999 seconds, JVM RedHat obtained the average of 2.955 seconds, and Blue JVM obtained the average of 3.038 seconds.

Table 5. Averages of CPU time.

JVM	Average (seconds)
HotSpot Java - Oracle	3,998 a
OpenJDK - RedHat	4,018 a
Zulu - Azul Systems	4,110 a

Table 6. Averages of system time.

JVM	Average (seconds)
HotSpot Java - Oracle	1,028 a
OpenJDK - RedHat	1,062 a
Zulu - Azul Systems	1,072 a

Table 7. Averages of use time.

JVM	Average (seconds)
HotSpot Java - Oracle	2,969 a
OpenJDK - RedHat	2,955 a
Zulu - Azul Systems	3,038 a

4.6. Analysis of Results

The graph of the CPU time averages in Figure 6, shows that the best performance amongst the JVMs was the JVM Oracle, with the lowest average CPU time, 4 seconds. The most important difference between averages of CPU times was between Oracle JVM and Blue JVM of 0.11 seconds and between JVM Oracle, and JVM RedHat was only 0.02 seconds. In the average system time, the Oracle JVM reached the shortest time of 1.03 seconds, as shown in Figure 7 and in the average use time, the RedHat JVM achieved the

quickest time of 2.96 seconds, as shown in Figure 8. The JVM Blue obtained the highest average system time, 1.07 seconds, and the highest average use time, 3.04 seconds, and consequently the highest average CPU time, 1.07 + 3.04 = 4.11 seconds. These graphs show that the average of the times were quite similar,and this is the reason why more rigorous techniques such as the analysis of variance and the Tukey test were used.

In Table 2, the average square, the ratio of the sum of squares to the degrees of freedom, was 0.092 for the JVMs and 0.081 for the random factors, called the errors. This table shows that the overall average obtained in the experiment was 4.042 seconds for CPU times with a coefficient of variation of 7.066 % and there was no significant difference between these times in the three JVMs. In Table 3, the average square was 0.013 for the JVMs and 0.014 for the error. This table shows that the overall average obtained in the experiment was 1.054 seconds for the system time, with the coefficient of variation of 11.357 % and there was no significant difference between these times in the three JVMs. In Table 4, the average square was 0.0049 for the JVMs and 0.047 for the error. This table shows the overall average obtained in the experiment was 2.987 seconds for use time, with a coefficient of variation of 7.280 % and there was no significant difference between these times in the three JVMs. In the three cases, coefficients were reduced, indicating empirical adequacy and high reliability of the experiment.

For the comparison test of Tukey averages, Table 5 shows that the best performance amongst the JVMs was the JVM Oracle, with the lowest average CPU time being 3.998 seconds. The biggest difference involves the averages of CPU times between Oracle JVM and Blue JVM being 0.112 seconds, while that difference between JVM Oracle and JVM RedHat was only 0.013 seconds. In the system time-average comparison test, the Oracle JVM obtained the shortest time, 1.028 seconds, as shown in Table 6. In the comparison test of use time averages, the RedHat JVM got the most efficient with 2.955 seconds, as shown in Table 7. In this test, the Blue JVM got the highest average system time, 1.072 seconds, the highest average use time, 3.038 seconds, and the highest average CPU time, 4.11 seconds. In the analysis of CPU time, system time and use time, the letter a in the column of averages in all three JVMs, indicates that the JVMs do not present statistically significant differences between themselves, by the comparison test by the Tukey model, at a level of 5 % of error probability.

4.7. Threats to validity

In this section, we evaluated the factors that influence the results obtained and the main limitations of the experiment. Our goal is to mitigate the validity threats since they are present in any empirical research [10].

4.7.1. Constructor Validity

We followed the guidelines of studies of the software engineering [5, 32, 57] to plan and perform the experiment. In this planning, we provide information about the execution environment, supporting tools, variables, execution and data collection. Then, we simulate a real-world integration process in two hundred different scenarios and used ANOVA and Tukey statistical techniques to evaluate the results.

4.7.2. Conclusion Validity

Conclusion validity aims to guarantee that the treatment used in the experiment is related to the actual outcome observed [17]. We used ANOVA and the Tukey statistical techniques to assure that the actual result observed in our experiment is concerning the JVMs, and not to factors that we do not control or have not measured, and we verified that there was a significant difference in the outcome.

4.7.3. Internal Validity

Internal validity aims to ensure that the treatment caused the outcome, mitigating effects of other uncertain factors or not measured ones [17]. We used the Windows security model, so that programs are avoided to run concurrently with the experiment. This competition for the use of processors has negative impacts on the times measured by the analysis. However, Windows's security model does not eliminate this competition, as other tasks are running under the operating system, which is not possible to control. The results may be different, in regards to another operating system and another environment configuration.

4.7.4. External Validity

External validity addresses the generalisation of the results outside the scope of the study [17]. We have studied the source code of the three JVMs in relation to the classes that compose the program, using a file comparison tool called "WinMerge" version 2.14.0.0 [33], we verified that these classes were implemented with the same features of the Java language, differing only in programming comments, which do not influence the execution. Thus, other implementations of JVMs that implement such classes with features other than those used by the tested JVMs, may result in different results. The study is valid for comparison of Oracle HotSpot JVMs, RedHat OpenJDK, Zulu from Azul Systems and can be re-used to compare other implementations, but it is not possible to generalise the results obtained for all JVM implementations.

5. Conclusions

Companies are looking for tools to make their business processes more competitive. Integration platforms are software tools that implement integration solutions for applications, which make up the business processes, communicate in an efficient and synchronised manner. The way platforms manage threads, which perform the tasks of an integration solution, have a direct impact on their performance. Many of the integration platforms have been developed in the Java programming language that offers multi-threaded programming. This type of programming allows tasks to be performed simultaneously, allowing an increased performance in implementing integration solutions. The interface `Executor` of the Java language *concurrency utilities* package provides features that facilitate this type of programming, such as: *(i)* configuration of the threads through the predefined parameters; *(ii)* factory methods for creating thread pools; *(iii)* different queue types for the tasks; *(iv)* policy for creating threads and queuing tasks.

The experiment presented in this article sought to verify the performance of three different implementations of Java virtual machines, simulating the behaviour of the execution of an integration solution by an integration platform, regarding the management of threads, using Java language resources. For the treatment of the time measurements, extracted from the experiment, we used the statistical analysis, using ANOVA and Tukey's test, where we verified that there is no statistically significant difference with 95 % confidence concerning the CPU time, the system time and use time between the tested machines. The implementation of Oracle's JVM HotSpot Java was the one that presented the lowest average CPU time, which consumed on average 4 seconds, followed by the JVM OpenJDK of RedHat that wasted in average 4.02 seconds and, finally, the Zulu of Azul Systems, which consumed on average 4.11 seconds. As there was no statistical difference, we concluded that these small differences between the measurements are not related to thread management, but to oscillations of the environment, entailed by the execution of other programs of the operating system itself and the internal implementation of the JVMs, such as the optimisation done by the compiler. The study is valid for the comparison of the three JVMs tested and can be reused to compare other implementations of JVMs concerning the performance of thread management in the execution of tasks of an integration solution.

Acknowledgements

This work was partially supported by the following Brazilian Research Funding Agencies: the Research Support Foundation of the State of Rio Grande do Sul (FAPERGS), under grant 17/2551-0001206-2; and, the Coordination for the Improvement of Higher Education Personnel (CAPES), under grants 88881.136207/2017-01 and 73318345415.

References

[1] Hotspot, 2016. URL `https://www.oracle.com/technetwork/systems/vmoptions-jsp-140102.html`. (Accessed at 14/10/2016).

[2] Openjdk redhat, 2016. URL `https://developers.redhat.com/products/openjdk/overview/`. (Accessed at 14/10/2016).

[3] Zulu, 2016. URL `https://www.azul.com/downloads/zulu/`. (Accessed at 14/10/2016).

[4] Christopher Alexander, Sara Ishikawa, and Murray Silvertein. *A pattern language: towns, buildings, construction*. Oxford University Press, 1977.

[5] Victor R. Basili, Dieter Rombach, Kurt Schneider Barbara Kitchenham, Dietmar Selby, and Richard W. Pfahl. *Empirical Software Engineering Issues*. Springer Berlin Heidelberg, 2007.

[6] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton,

Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. *ACM Sigplan Notices*, 41(10):169–190, 2006.

[7] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: A survey. *Future Generation Computer Systems*, 56:684–700, 2016.

[8] Kuo-Yi Chen, J. Morris Chang, and Ting-Wei Hou. Multithreading in Java: Performance and scalability on multicore systems. *IEEE Transactions on Computers*, 60(11):1521–1534, 2011.

[9] Cosme Damião Cruz. *Genes program: experimental statistics and matrices*. Federal University of Viçosa, 2006.

[10] Daniela Soares Cruzes and Lotfi ben Othman. Threats to validity in empirical software security research. In *Empirical Research for Software Security*, pages 295–320, 2017.

[11] Gleydson de Azevedo Ferreira Lima. Performance analysis of large distributed systems on the Java platform. Master's thesis, Federal University of Rio Grande do Norte, 2007.

[12] Alcione de Paiva Oliveira and Vinícius Valente Maciel. *Java in practice*. Fábrica de Livros, 2003.

[13] Paul J. Deitel and Harvey Deitel. *Java How to Program, Early Objects, Student Value Edition*. Pearson, 2017.

[14] Eldair F. Dornelles. Performance analysis of the executor API at zulu and oracle JVM on windows and linux operating systems. *Scientific Seminar on Technological Training*, 14, 2017.

[15] David Dossot, John DÉmic, and Victor Romero. *Mule in action*. Manning Publications Co., 2014.

[16] Nico Ebert, Kristin Weber, and Stefan Koruna. Integration platform as a service. *Business & Information Systems Engineering*, 59:375–379, 2017.

[17] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research-an initial survey. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 374–379, 2010.

[18] Mark Fisher, Jonas Partner, Marius Bogoevice, and Iwein Fuld. *Spring integration in action*. Manning Publications Co., 2012.

[19] Rafael Z. Frantz, Rafael Corchuelo, and José L. Arjona. An efficient orchestration engine for the cloud. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 711–716, 2011.

[20] Rafael Z. Frantz, Rafael Corchuelo, and Carlos Molina-Jiménez. A proposal to detect errors in Enterprise Application Integration solutions. *Journal of Systems and Software*, 85(3):480–497, 2012.

[21] Rafael Z. Frantz, Rafael Corchuelo, and Fabricia Roos-Frantz. On the design of a maintainable software development kit to implement integration solutions. *Journal of Systems and Software*, 111:89–104, 2016.

[22] Daniela L. Freire, Rafael Z. Frantz, and Fabricia Roos-Frantz. Ranking Enterprise Application Integration platforms from a performance perspective: An experience report. *Software: Practice and Experience*, 49(5):921–941, 2019.

[23] Daniela L. Freire, Rafael Z. Frantz, Fabricia Roos-Frantz, and Sandro Sawicki. Survey on the run-time systems of Enterprise Application Integration platforms focusing on performance. *Software: Practice and Experience*, 49(3):341–360, 2019.

[24] Karthik Ganesan, Yao-Min Chen, and Xiaochen Pan. Scaling Java virtual machine on a many-core system. In *International Symposium on Integrated Circuits (ISIC)*, pages 336–339, 2014.

[25] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[26] Inma Hernández, Sandro Sawicki, Fabricia Roos-Frantz, and Rafael Z. Frantz. Cloud configuration modelling: A literature review from an application integration deployment perspective. *Procedia Computer Science*, 64:977–983, 2015.

[27] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[28] Mingkai Huang, Xianhua Liu, Tingyu Zhang, and Xu Cheng. Exploration of the relationship between just-in-time compilation policy and number of cores. In *International Conference on Algorithms and Architectures for Parallel Processin (ICA3PP)*, pages 293–307, 2015.

[29] Claus Ibsen and Jonathan Anstey. *Camel in action*. Manning Publications Co., 2010.

[30] Kasun Indrasiri. *Introduction to the WSO2 ESB*. Springer, 2016.

[31] Corporation Intel. Get faster performance for many demanding business applications, 2018. URL `https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`. (Accessed at 12/01/2018).

[32] Andreas Jedlitschka and Dietmar Pfahl. Reporting guidelines for controlled experiments in software engineering. In *International Symposium on Empirical Software Engineering (ESEM)*, pages 95–104, 2005.

[33] Chris Kemper and Ian Oxley. I have a conflict: What can I do? In *Foundation Version Control for Web Developers*, pages 221–241. Springer, 2012.

[34] Khalil Khoumbati, Marinos Themistocleos, and Zahir Irani. Evaluating the adoption of Enterprise Application Integration in Health-Care organizations. *Journal of Management Information Systems*, 22:69–108, 2006.

[35] Henryk Konsek. *Instant Apache ServiceMix How-to*. Packt Publishing, 2013.

[36] Christopher D. Krieger and Michelle Mills Strout. Performance evaluation of an irregular application parallelized in Java. In *International Conference on Parallel Processing Workshops (ICPPW)*, pages 227–235, 2010.

[37] Prasad A. Kulkarni. JIT compilation policy for modern machines. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 46, pages 773–788, 2011.

[38] David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005.

[39] Guilherme Grunewald Magalhães, Anderson Luis Sartor, Arthur Francisco Lorenzon, Philippe Olivier Alexandre Navaux, and Antonio Carlos Schneider Beck. How programming languages and paradigms affect performance and energy in multithreaded applications. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 71–78, 2016.

[40] Douglas C. Montgomery and George C. Runger. *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

[41] Samih M. Mostafa, Shigeru Kusakabe, and Hirofumi Amano. Fairness scheduler for multithreaded programs in virtual machine environment. In *Japan-Egypton Conference Electronics, Communications and Computers (JEC-ECC)*, pages 79–82, 2016.

[42] Panagiotis Patros, Eric Aubanel, David Bremner, and Michael Dawson. A Java util concurrent park contention tool. In *Programming Models and Applications for Multicores and Manycores (PMAM)*, pages 106–111, 2015.

[43] Francis Rangel and Anderson Faustino Silva. The Java virtual machine and inline optimization: A case study. *Revista Tecnológica*, 21(1):103–118, 2012.

[44] Jesse Russell and Ronald Cohn. *Fuse ESB*. Book on Demand, 2012.

[45] Jesse Russell and Ronald Cohn. *Jitterbit Integration Server*. Book on Demand, 2012.

[46] Robert G. Sargent. Verification and validation of simulation models. *Journal of Simulation*, 7(1):12–24, 2013.

[47] Jennfer B. Sartor and Lieven Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, volume 47, pages 281–296, 2012.

[48] Herbert Schildt and Danny Coward. *Java: the complete reference*. McGraw-Hill Education, 2014.

[49] Gabriel P. Silva. Evaluating a unix environment with multiple processors. In *Brazilian Symposium on Computer Architecture Parallel Processing*, volume 2, 1988.

[50] Paulo Silveira, Guilherme Silveira, Sérgio Lopes, Guilherme Moreira, Nico Steppat, and Fábio Kung. *Introduction to architecture and software design: an overview of the Java platform*. Elsevier Editora Ltda., 2012.

[51] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. *ACM SIGARCH Computer Architecture News*, 36(1):277–286, 2008.

[52] Lambert M. Surhone, Miriam T. Timpledon, and Susan F. Marseken. *Petals ESB*. Betascript Publishing, 2010.

[53] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems: Global Edition*. Pearson Education, 2014.

[54] John W. Tukey. Comparing individual means in the analysis of variance. *Biometrics*, 5:99–114, 1949.

[55] Hitoshi Ueno. A performance evaluation of multi-programming model on a multicore system with virtual machines. In *International Symposium on Embedded Multicore/Manycore SoCs (MCSoc)*, pages 321–328, 2014.

[56] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[57] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslen. *Experimentation in software engineering*. Springer Science & Business Media, 2012.