A Measurement-based Aging Analysis of the Java Virtual Machine

Domenico Cotroneo, Salvatore Orlando,
Roberto Pietrantuono, Stefano Russo*

Universitá di Napoli Federico II,
Dipartimento di Informatica e Sistemistica
Via Claudio 21, 80125 - Napoli, Italy

SUMMARY

In this work, a software aging analysis of Java-based software systems is conducted.
The Java Virtual Machine (JVM) is the core layer in Java-based systems, and its dependability greatly affects the overall system quality. Starting from an experimental campaign on a real world testbed, this work isolates the contribution of the Java Virtual Machine (JVM) to the overall aging trend, and identifies, through statistical methods, which workload parameters are the most relevant to aging dynamics.
Results revealed the presence of several aging dynamics in the JVM, including: *i)* a throughput loss trend mainly dependent on the execution unit, *ii)* a *slow* memory depletion drift, due to the JIT-compiler activity, *iii)* a *fast* memory depletion drift caused by dynamics inside the garbage collector.
The outlined procedure and obtained results are useful in order to *i)* identify the presence of aging phenomena, *ii)* perform on-line aging detection and time-to-exhaustion prediction, and *iii)* define optimal rejuvenation techniques.

keywords: Software Aging, Software Reliability Evaluation, Workload Characterization, Java Virtual Machine.

1. Introduction

Software aging is a term that refers to a condition in which the state of the software or its environment degrades with time [1]. The primary causes of this degradation are the exhaustion of operating system resources, data corruption, and numerical error accumulation.
Recent studies showed that a large number of software systems, employed also in business-critical or safety-critical scenarios, are affected by software aging. The Patriot missile defense system employed during the first gulf war, responsible for the scud incident in Dhahran, is perhaps the most representative

*E-mail: (cotroneo, saorland, roberto.pietrantuono, stefano.russo)@unina.it

Prepared using stvrauth.cls *[Version: 2007/09/24 v1.00]*

example of a critical system affected by software aging. To project a target's trajectory, the weapon control computer requires a conversion from integer into a real value, which causes a round-off error in the calculation of target's expected position. System users were warned that "very long runtime" could negatively affect system's targeting capabilities. Unfortunately, they were not given a quantitative evaluation of such "very long runtime", thus leading to the well-known incident in which 28 people were killed [2].

In order to reduce the development efforts and the time-to-market, there is an increasing use of Off-The-Shelf (OTS) items in the development of complex software systems. Examples are virtual machines or middleware layers, which are increasingly being used also in critical contexts. It is unrealistic to assume that such complex systems are not affected by software aging. On the contrary, since they often lack proper testing in the target environment, they are more subject to these phenomena. Thus, in order to employ OTS in critical scenarios, it is very important to characterize their behavior from a software aging perspective.

Software systems including such items may be regarded as a stack of software layers, each of them using services from the lower layer and offering services to the upper one. Among the wide range of this kind of layered systems, Java-based systems represent a relevant class to study, for the following reasons:

1. they are currently employed in a wide range of contexts, including critical ones. For instance, Java has been used to develop the ROver Sequence Editor (ROSE), a component of the Rover Sequencing on Visualization Program (RSVP), used to control the Spirit Robot in the exploration of Mars [3];
2. there is a growing interest in the scientific and industrial community toward the employment of Java in safety and mission critical scenarios, as shown by a Java Specification Request (JSR-302 [4]), which aims at defining those capabilities needed to use Java technologies in safety critical applications.

Contribution. While performance aspects of the JVM have been extensively explored in the past [5, 6, 7], software dependability issues received less attention. An example to augment fault tolerance in the JVM has been proposed by Alvisi et *al.* [8], who conducted an interesting study on how to apply state machine replication to the JVM. A similar approach has been used by Friedman et *al.* [9], applied to the *Jikes Research Virtual Machine*. Regarding software aging, a work by Silva et *al.* [10] highlighted the presence of aging phenomena in a Java-based SOAP server. Several research studies analyzed aging introduced by long-running applications (such as web servers [11] and DBMS servers [12]) measuring the aging contribution at operating system level. They neglected the contribution of intermediate layers, such as middleware, virtual machines, and, more in general, third-party off-the-shelf (OTS) items. Such layers may impact resource exhaustion dynamics and become a significant source of aging.

The methodology presented in this paper allows to identify and analyze the contribution of such intermediate layers in a Java-based software system.

Moreover, even though past studies highlighted the presence of aging independently of the workload applied to the system (e.g., [13]), successive works showed that the software aging phenomena are strictly related to the workload. Some studies addressed the relationships between workload and aging trends (e.g., [14, 15, 11]), but the selection of workload parameters and the assessment of their effect on aging trends have been partially addressed only by Matias and Filho [16], and Hoffmann et *al.* [17]. The

former addressed the evaluation of the effects of workload parameters on the response variable, whereas the latter coped with the selection of the most relevant workload parameters. However, the work by Matias and Filho [16] encompassed only controllable workload factors, whereas workload parameters to monitor in layered systems are often uncontrollable. Variable selection techniques presented by Hoffmann et *al.* [17] do not focus on the influence of workload parameters on aging trends.
From the above considerations, it is clear that in order to develop a methodology to analyze software aging in Java-based systems, two challenging issues have to be addressed.
First, **new methods are needed to isolate the contribution of the JVM to the aging trends from the ones of other software layers**.
Second, **since there is a strict relationship between workload and aging trends, it is crucial to investigate how these trends are affected by changes in the workload being applied**.
By exploiting statistical techniques such as cluster analysis, principal component analysis and multiple regression, this paper proposes a methodology to analyze workload-correlated aging phenomena by *i)* pinpointing software layers in which aging phenomena are introduced; *ii)* identifying which workload parameters are more relevant to the development of aging trends, and *iii)* evaluating the relationships between workload parameters and aging trends. The methodology includes three phases:

1. *Design and Realization of Experiments*, in which several long-running experiments are executed with different workload levels thus allowing to collect data about system resource usage (at different layers) and workload.
2. *Workload Characterization*, in which workload data are analyzed in order to characterize the behavior of the observed component as a function of the workload level imposed during the experiments.
3. *Software Aging Analysis*, in which aging trends exhibited at different layers and existing relationships between applied workload and aging phenomena are evaluated.

By applying this methodology, software engineers are able to gain insights about the sources of aging, identifying the components more affected by this phenomenon. Moreover, results of such an analysis allow engineers to design strategies to counteract aging during operation, by obtaining predictions of aging trends, and by determining the best time schedule to rejuvenate the system and to clean its state.
The rest of the paper is organized as follows. Section 2 surveys the research on software aging and provides background on the JVM architecture. Section 3 discusses the presence of aging in the JVM, and presents the tool implemented to monitor and collect data relevant to the subsequent aging analysis, namely *JVMMon*. Section 4 outlines the steps of the proposed methodology, while Section 5 shows its application to the case-study, discussing the results. Section 6 concludes the paper.

2. Research on Software Aging

Software Aging can be defined as a continued and growing degradation of software internal state during its operational life. It leads to progressive performance degradation, occasionally causing system lockout or crashing. Due to its cumulative property, it occurs more intensively in continuously running processes that are executed over a long period. Software aging phenomena are due to the activation of *aging-related bugs* [18]; typical examples of these bugs are memory bloating and leaking, unreleased file locks, data corruption, storage space fragmentation, and accumulation of round-off errors.

Software aging has been widely observed, reported and documented for a consistent number of operational software systems. It has been observed in telecommunications billing applications, as well as in the related switching software [19]; it caused a 55 meters errors in the target trajectory calculation after 8 hours of execution, in the cited Patriot missile control software [2]; it was also detected in the Apache Web Server [20], and even in the Linux Kernel code [21].
Moreover, it is well known that a consistent number of systems progressively slow down until they need to be rebooted. To counteract aging, a proactive approach to environment diversity has been proposed, in which the operational software is occasionally stopped and then restarted in a "clean" internal state. This technique is called *software rejuvenation*, and it was first proposed in 1995 by Huang et *al.* [22].
Countermeasures against aging during operation require the assessment of the current health of the system, the estimation of the expected time to resources exhaustion (a measure often called *Time to Exhaustion*, or TTE), and the determination of the optimal rejuvenation schedule. Solutions proposed to counteract software aging can be broadly classified into two approaches: the analytic modeling and the measurement-based approach.

Analytic Modeling Approach

Analytic modeling generally aims at determining the optimal time to perform software rejuvenation in operational software systems. The optimal rejuvenation schedule is determined starting from analytical models, which are conceived to represent the aging phenomenon, and the accuracy depends on the assumptions that are made for the model construction. Approaches in this category assume failure and repair time distributions of a system and obtain optimal rejuvenation schedule to maximize availability, or minimize loss probability or downtime cost.
Analytical models were first employed in order to prove that, when dealing with systems affected by software aging, software rejuvenation allows reducing the cost due to system downtime [22] and minimizing program completion time [23]. Regarding the kind of the considered aging effects, several works take into account only failures causing total unavailability of software [22, 23, 24], whereas Pfening et *al.* [25] consider a gradually decreasing service rate (i.e., a performance degradation); a model that takes into account both the effects together in a single model is reported by Garg et *al.* [15].
Different probability distributions were also chosen for time-to-failure (TTF). Some works, such as the ones by Garg *al.* [24] and by Huang et *al.* [22], are restricted to a hypo-exponential distribution, whereas other papers employ more general distributions for TTF, like the Weibull distribution (e.g., [23]). However these TTF models are unable to capture the effect of load on aging.
Although some authors, like Andrzejak and Silva [26], Bao et *al.* [14], and Garg et *al.* [15], considered workload in their analyses, it is still unclear how the impact of workload variation on software aging should be evaluated; the selection of relevant workload parameters that need to be taken into account and the level at which they have to be observed are still open issues.
In analytic modeling approaches, stochastic processes are commonly used. In the work by Pfening et *al.* [25] a Markov Decision Process (MDP) was used to build a software rejuvenation model in a telecommunication system including the occurrence of buffer overflows. Garg et *al.* [24] used Markov semi-ReGenerative Processes (MGRP), in conjunction with Stochastic Petri Nets (SPN), in order to build a simple but general model to estimate the optimal rejuvenation schedule in a software system. Petri Nets, in particular Stochastic Deterministic Petri Nets (SDPN) were employed by Wang et *al.* [27], in order to build a model to analyze performability of cluster systems under varying workload. A recent approach based on Petri nets has been presented by Salfner and Wolter [28]. In their work, authors focused on the evaluation of time-triggered system rejuvenation policies using a queuing

model, formulated as an extended stochastic Petri net. Non-homogeneous, continuous time Markov Chains were instead used by Garg et *al.* [15]. Semi-Markovian Processes were also used to model proactive fault management by Bao et *al.* [14]. A common shortcoming with analytic modeling is that the accuracy of the derived rejuvenation schedule deeply depends on the goodness of the model (i.e., how good the stochastic model used to represent the system approximates the real behavior of the system) and on the accuracy of the parameters used to solve the model (e.g., failure rate distribution expected value, probability of transition from the "steady" state to the "degraded" state).

Measurement-Based Approach

The measurement-based approach applies statistical analysis to data collected from systems and applies trend analysis or techniques to determine a time window over which to perform rejuvenation. The basic idea of measurement-based approaches is to directly monitor attributes subject to software aging, trying to assess the current "health" of the system and to obtain predictions about possible impending failures due to resource exhaustion or performance degradation. A measurement-based software aging analysis performed on a set of Unix workstation is reported by Garg et *al.* [13]. In this paper, a set of 9 Unix Workstations was monitored for 53 days using an SNMP-based monitoring tool. During the observation period, 33% of reported outages were due to resource exhaustion, highlighting how much software aging is a non-negligible source of failures in software systems.

An interesting workload-based software aging analysis is reported in the work by Vaidyanathan and Trivedi [11]. The paper presented results of an analysis conducted on the same set of Unix workstation used by Garg et *al.* [13]. While the latter considered only time-based trend detection and estimation of resource exhaustion without considering the workload, the former took the system workload into account and built a model to estimate resource exhaustion times. It considered some parameters to include the workload in the analysis, such as the number of CPU context switches and the number of system call invocations. Different workload states were first identified through statistical cluster analysis and a state-space model was built, determining sojourn time distributions; then, a reward function, based on the resource exhaustion rate for each workload state, was defined for the model. By solving the model, authors obtained resource depletion trends and TTE for each considered resource in each workload state. The methodology presented by Vaidyanathan and Trivedi [11] allows carrying out a workload-driven characterization of aging phenomena, more useful and powerful than the workload-independent characterization presented by Garg et *al.* [13]. A further improvement of this work was presented by Vaidyanathan and Trivedi [29], where a hybrid approach was adopted (i.e., analytical and measurements-based), by *i)* building a measurement-based semi-markovian model for system workload, *ii)* estimating TTE for each considered resource and state (using reward functions), *iii)* and finally building a semi-Markov availability model, based on field data rather than on assumptions about system behavior.

The work by Vaidyanathan and Trivedi [29] extends the previous one [11] by performing transient analysis, formulating the TTE as the mean time to accumulated reward in a semi-Markov reward model, and developing an availability model that accounts for failure and rejuvenation, useful to derive optimal rejuvenation schedules.

Another interesting measurement-based approach to software rejuvenation, based on a closed-loop design, was presented by Hong et *al.* [30].

Although several measurement-based analyses dealt with resource exhaustion, only a few of them deal with performance degradation. Software aging that manifests itself as a progressive loss of performance was deeply studied for OLTP servers [12] and for the Apache Web Server [31, 20].

In the work by Gross et *al.* [12] pattern recognition methods were applied in order to detect aging phenomena in shared memory pool latch contention in large OLTP servers. Results of this work showed that these methods allowed to detect significant deviations from "standard" behavior with a 2 hours early warning. On the other hand, Trivedi et *al.* [31, 20], analyzed performance degradation in the Apache Web Server by sampling web server's response time to predefined HTTP requests at fixed intervals. Collected data were analyzed using the same techniques employed by Garg et *al.* [13]. Results showed a 0.061 ms/hr degradation for response time in the Apache Web Server, and a 8.377 Kb/hr depletion trend for physical memory. Used swap space, on the other hand, showed a seasonal pattern, as a direct consequence of built-in rejuvenation mechanisms in the Apache web Server. Software Aging in a SOAP-based server was also analyzed by Silva et *al.* [10], where authors presented a dependability benchmarking study to evaluate some SOAP-RPC implementations, focusing in particular on Apache Axis, where they revealed the presence of aging.
An analysis addressing the impact of workload parameters on aging trends was presented by Matias and Filho [16], where the memory consumed by an Apache Web Server was observed together with three controllable workload parameters: page size, page type (dynamic or static), and request rate. Applying the Design Of Experiments (DOE) technique, several experiments were performed with different level of the three workload parameters; effects of single and combined workload parameters on the output variable (memory used) were evaluated through the Analysis of Variance (ANOVA) technique. A closed-loop software rejuvenation agent was also implemented. Finally, Hoffmann et *al.* [17], proposed a best practice guide for building empirical models to forecast resource exhaustion. This best practice guide addresses the selection of both resource and workload variables, the construction of an empirical system model, and the sensitivity analysis.

3. Aging in the JVM

3.1. The Architecture of the JVM

In this section, the main components* of the JVM architecture are described. The specification for the Java Virtual Machine [32] has been implemented in different ways by many vendors. JVM implementations differ from one another not only with regard to the interface toward the operating system but also in the implementation of internal components. In this paper, focus is on the Sun Hotspot 1.5 VM. In order to understand the internal behavior of the Sun Hotspot VM, its source code has been carefully analyzed, due to the lack of appropriate documentation concerning the implementation of VM components. The resulting schema is depicted in Figure 1. The JVM is composed of four main components: **i)** The *Execution Unit*, which includes the core components of the JVM needed for executing Java programs, e.g., the bytecode interpreter and the Java Native Interface (JNI); **ii)** The *Memory Management Unit*, which manages memory operations (e.g., object allocation, object reference handling, garbage collection); **iii)** The *System Services Unit*, which offers Java Applications "higher level" services, such as thread synchronization management and class loading; and **iv)** the

*In this context, the tern "component" is intended as a logically independent unit of an architecture performing a well-defined function

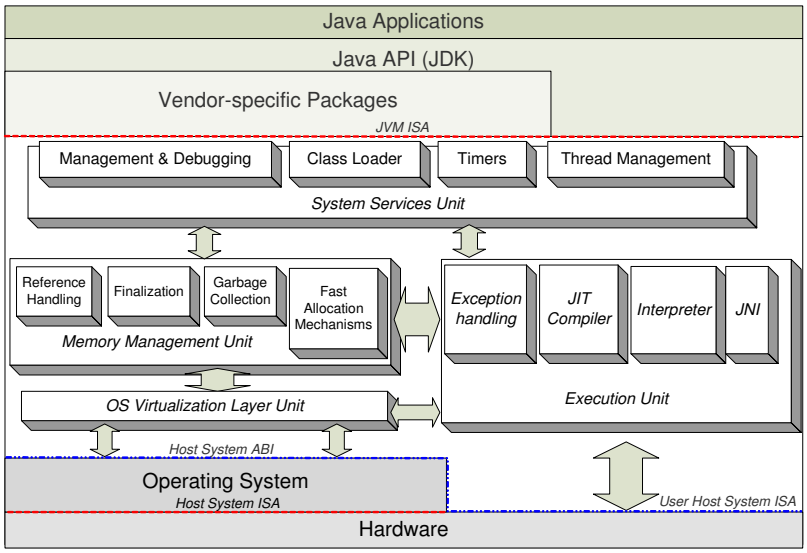


Figure 1. Architectural Model of the Java Virtual Machine

OS abstraction layer, which provides a platform-independent abstraction of the host system's ABI (Application Binary Interface).

The Execution Unit

It dispatches and executes operations, acting like a CPU. An operation could be a bytecode instruction, a JIT-Compiled method or a native instruction. The *Interpreter* translates single bytecode instructions into native machine code, whereas the *Just-In-Time(JIT)* compiler optimizes the execution of a set of instructions (methods) translating it into native code. Methods to JIT-compile are automatically selected by the JVM by exploiting the code locality principle: the JVM is able to identify "hotspots" in the application, i.e., pieces of code that are executed more frequently. Native instructions need no translation: they are dynamically loaded, linked and executed by the *Java Native Interface (JNI)*. Furthermore, the *Exception Handler* deals with exceptions thrown by both Java Applications and the Virtual Machine. In particular, exceptions thrown by the VM are called *unchecked* and are related to errors originated inside the virtual machine.

The Memory Management Unit

It handles the JVM heap area, managing object allocation, reference handling, object finalization, and garbage collection. The heap area is organized into three generations, i.e., three memory pools holding objects of different ages: the young generation, the tenured generation, and the permanent generation. Objects are first allocated in the young generation, and when the latter becomes full, a minor collection occurs; surviving objects are moved to the tenured generation. The young generation consists of *eden* and *survivor* space. The latter is divided into two further survivor spaces (called *semispaces*) that are

used to implement the garbage collection copying algorithm.
Whenever a new object is allocated, it is stored in the eden space. Objects that survive garbage collection are first promoted to the survivor space and then to the tenured generation. Due to the high "infant mortality" of Java objects, only a few of them reach the tenured generation. This way, the performance of the JVM is improved, since the frequency of full garbage collection cycles, which involve both the young and the tenured generation, is reduced. The young and the tenured generations are used to store Java objects and are therefore subject to garbage collection; instead, the permanent generation is mainly used to store Java classes loaded into the JVM: objects in this area are not subject to garbage collection. Even if the maximum size of these generations is fixed at JVM startup, their actual dimension depends upon the memory requirements of the application, since the JVM has the ability to dynamically grow or shrink the size of each generation.
The Sun Hotspot JVM provides several garbage collectors. By default, the JVM employs a serial, stop-the-world garbage collector; this collector uses a copying algorithm on the young generation, using semispaces in the survivor space, and a compact, mark-and-sweep algorithm on the tenured generation. It is also possible to use multi-threaded garbage collectors to improve either program completion time or throughput. Furthermore, *Fast Allocation Mechanisms* are provided to allocate memory areas for internal VM operations.
The System Services Unit
Components included in this unit offer services to Java Applications. The *Thread Management* component handles Java threads as specified by the *Java Virtual Machine Specification*[†] and the *Java Language Specification (JLS)* [33]. The *Class Loader* is in charge of dynamically loading, verifying and initializing Java classes. Finally, the *Management and Debugging* component includes functionalities for debugging Java applications and for management of the JVM.
The OS abstraction layer
This component provides a platform-independent abstraction of the host system's Application Binary Interface. It represents a common gateway for all JVM components to access host system's resources.

3.2. Is the JVM affected by Software Aging?

Software aging manifests itself as a continuous and progressive degradation of performances with time, due to resource exhaustion or throughput loss; a JVM is affected by aging when its activity contributes to the resource exhaustion and/or to the throughput loss trend (e.g., when the activity of an internal JVM component, such as the JIT compiler, contributes to a progressive memory consumption, it is possible to state the JVM is getting aged).
In previous authors' work [34], a failure analysis of the JVM have been shown, in order to figure out whether aging failures are really a threat to JVM's dependability. In that analysis, failure data were extracted, in the form of reports, from publicly available bug databases, where developers and users of Java applications usually submit failures/bugs. Data were extracted from *Sun*[‡] and *Jikes*[§], resulting in 700 bug submissions related to JVM failures. Other JVM implementations, such as *Kaffe*, *J9*, and

[†] *http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html*
[‡] Sun Hotspot Bug Database: *http://bugs.sun.com*
[§] Jikes RVM bug database: *http://jikesrvm.org*

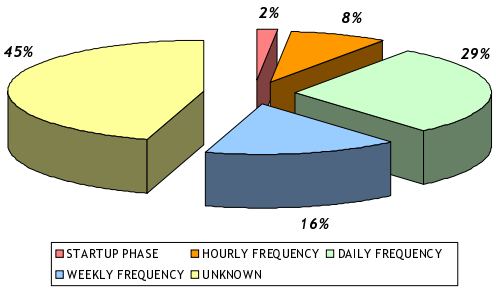


Figure 2. Distribution on non-deterministic failures by time-to-failure

JRockit had no public bug databases or very poor ones[¶]. Data extracted from bug reports allowed classifying failures along several dimensions, i.e., Failure Manifestation, Failure Source, Severity, Environment, Workload. Such data have been used to start an in-depth study of the development of aging phenomena. In particular, the procedure adopted to perform an aging-oriented analysis of failure reports was the following:

1. Exclude failures marked as "Always Reproducible": failures that are due to aging phenomena are usually strictly dependent on the workload applied to the JVM, and are therefore not easily reproducible in a different environment. Obviously, remaining failures (64% of the total number of failures) exhibit a non-deterministic behavior.
2. Among the remaining failures, exclude those that occurred when a non-relevant workload was applied: without a relevant workload it is generally impossible to observe the development of aging phenomena. Only a 2% of failures occurred with a negligible workload, although workload estimation was not possible for 30% of occurred failures (due to lack of data in reports).
3. Analyze the distribution of the remaining failures (32% of the total number of failures) with regard to the estimation of the time to failure.

The pie chart in Figure 2 reports the distribution of non-deterministic failures occurring with relevant workload according to the time to failure. Unfortunately, there is a high level of uncertainty, since data (even qualitative) about TTF was missing in a consistent number of reports. Only 2% of these failures occurred during the startup phase, whereas 45% of these failures occurred after a significant time (16% daily, 29% weekly). This means that in about half of the considered failures there are significant clues of software aging phenomena.

Summarizing, it is possible to state that:

- About 40% of failures are absolutely NOT due to aging phenomena;

[¶]Further information on these JVMs can be found at: *http://www.kaffe.org/, http://wiki.eclipse.org/index.php/J9*, and *http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html*, for *Kaffe*, *J9*, and *JRockit*, respectively

- There is a high probability that 15% of failures are due to aging phenomena;
- For the remaining 45% of failures, it is not possible to state anything with certainty: however, by analyzing the reports, it has been possible to deduce that a part of these failures is probably related to aging.

Therefore, results of this study indicate that software aging is not an irrelevant phenomenon in the JVM, since it is responsible for a non-negligible percentage of failures. Moreover, further analyses carried out on the JVM (presented in a previous work [35]), confirmed the presence of software aging phenomena. Based on the outcome of such studies, this paper defines and experiments a measurements-based methodology for carrying out aging analyses in Java software systems. It first presents the methodology; then shows results of its application to a Java-based software system, and finally discusses in detail the key findings about aging dynamics evolution inside the JVM.

3.3. JVM Monitoring

In order to analyze aging phenomena in Java-based systems, it should be possible to collect enough information, expressed by proper parameters, in each of the involved layers. In particular, parameters at application level are set by the experimenter (i.e., they are controllable parameters), whereas parameters at OS level are collected by well-known and available tools (two well-know aging indicators are collected, i.e., throughput loss and memory depletion). However, to gain insights about the particular aging contribution of the intermediate layer, i.e., the JVM, a set of internal parameters needs to be identified, and then monitored. To this aim, an ad-hoc infrastructure, named *JVMMon*, has been developed. Unlike other systems conceived to collect failure data, *JVMMon* has been designed to intercept each event related to changes in the state of the JVM, including errors and failures.
The proposed monitoring infrastructure allows on-line analysis of the JVM state evolution through a three- steps process: **i)** a monitoring agent, developed by using the JVM Tool Interface (JVMTI) (which stems from the *Java Platform Profiling Architecture* [36]) and Bytecode Instrumentation (BCI), intercepts events generated inside the JVM and collects data about its state; **ii)** a monitoring daemon processes this information and updates the state of the virtual machine; **iii)** a data collector stores collected data in a database, allowing on-line and off-line analysis. Since JVMMon is built upon JVMTI, it may be employed with all JSR-163-compliant virtual machines.
Data Sources:

Data are collected using the following information sources:
JVMTI events - Several events raised from the JVM are intercepted by implementing JVMTI callbacks. Events intercepted by *JVMMon* are reported in Table I.
JVMTI functions - JVMTI callbacks use these functions in order to update the state of the JVM component, which the raised event is related to. The functions used to determine the state of the Java Virtual Machine are reported in Table II.
Java Objects (through BCI) - Java methods are instrumented in order to obtain further information about the virtual machine.
Events raised by the JVM are useful, along with the JVMTI functions and with Java Objects, to track the state evolution of the JVM. In fact, starting from this information sources, workload parameters describing the state of each JVM subsystem are derived. These allow figuring out how the internal

Table I. VM-related events intercepted to collect data about JVM evolution.

Event	Raised when	Additional information supplied
Class Load/Prepare	Generated when the class is first loaded (Load) or when the class is ready to be instanced in applications (Prepare)	Thread loading the class, java.lang.Class instance associated with the class
Compiled Method Load/Unload	Generated when a method is JIT-Compiled and loaded (or moved*) into memory or unloaded from memory	Method being compiled and loaded/unloaded, compiled code size and absolute address where code is loaded
Exception	Generated when an exception is detected by a java or native method	Thread throwing the Exception, location where it was detected, java.lang.Throwable instance describing the Exception
Exception Catch	Generated whenever a thrown exception is caught	Thread catching the Exception, location where it was caught java.lang.Throwable instance describing the Exception
Monitor Contended Enter/Entered	Generated when a thread is attempting to enter (enters) a of the Java monitor acquired by another thread	Thread attempting to enter (or entering) the monitor, instance monitor
Monitor Wait/Waited	Generated when a thread is entering (leaving) Object.wait()	Thread entering (leaving) Object.wait(), instance of Object, waiting timeout (if applicable)
Object Free	Generated when the Garbage Collector frees an object	Tag** of the freed object
Thread Start/End	Generated immediately before the run method is entered (exited) for a Java Thread	Thread starting (terminating)
VMStart	Generated w hen the JNI subsystem of the JVM is started. At this point the VM is not yet fully functional.	
VMInit	Generated when JVM inizialization has completed.	Thread executing the public static void main method.
VMDeath	Generated w hen the VM is terminated. No more events will be generated.	

*When a JIT-compiled method is moved a Compiled Method Unload Event is generated, followed by a Compiled Method Load event
**Object Free events are generated only for tagged object. A tag is a 64-bit integer variable connectable to each object in JVM Heap

JVM components are evolving.

Table II. Functions Employed to retrieve data about Java Virtual Machine state

Function	Retrieved Information
`GetThreadState`	Bitmask describing the state of a Java Thread
`GetThreadInfo`	Thread priority and context class loader
`GetOwnedMonitorInfo`	Monitor owned by a thread
`GetStackTrace`	Thread's stack trace
`IterateOverHeap`	Information about organization of object in Heap Area (through an Heap Iteration callback function)
`GetClassStatus`	Status of a Class
`GetClassModifiers`	Access flags for a Class Instance
`GetObjectSize`	Object size in byte
`GetObjectHashcode`	Unique identifier associated with the object

Monitoring the state of the JVM: The state of the JVM may be defined as the union of the states of its components. Some of these components, namely the JIT compilers, the Interpreter and the OS Abstraction Layer, may be regarded as being stateless, whereas the state of the *Class Loader*, the *Thread Management Unit*, and the *Memory Management Unit* is defined as follows:

Class Loader - Its state is defined by the list of classes loaded in the permanent generation. A Java Class could be `loaded`, `prepared` (the code is available in method area), `initialized`, `unloaded` or in an `erroneous` state, if there was an error during preparation or initialization.
Thread Management Unit - The state of this component is characterized by the state of each Java thread. Internal VM threads are not managed by this component. In order to characterize the state of each Java Thread, *JVMMon* keeps track of the following information:
- *State*: current state of the thread (i.e., Runnable, Waiting, Blocked, Suspended, etc.)
- *Stack trace*: stack trace of the thread.
- *Owned monitors*: a list of monitors owned by a thread. According to the Java Language Specifications (JLS) [33], only one thread at a time may own a monitor.
- *Contended monitor* and *Waiting monitor*: the monitor on which the thread is currently blocked.
- *Scheduling timestamps*: a timestamp is taken each time a thread is scheduled on an available processor and each time the same thread yields the processor to another thread. This allows collecting scheduling information also in multiprocessor systems.
Memory Management Unit: - Since the aim is to monitor the integrity of data structures on which the reference handler and the garbage collector operate, the heap of the JVM is defined as the set of objects allocated in the Java Heap since the VM has been started. *JVMMon* has been implemented to distinguish the amount of memory committed to the application from the space actually allocated into the heap of the JVM, thus allowing to obtain resource usage information both at the application and at the JVM layer. This will be useful to isolate the layer in which prospective aging trends are introduced.
Monitoring JVM workload:

Since it is impossible to understand how aging phenomena evolve inside the JVM without characterizing its workload, *JVMMon* is conceived to also monitor JVM workload parameters. This task is accomplished by using **i)** JVMTI functions, **ii)** Bytecode Injection, and **iii)** performance counters accessible through the `jstat`[‖] interface.
A number of 30 workload parameters (reported in Table III) were monitored, in order to describe the behavior of each relevant part of the JVM architecture. These parameters are related to the components of the JVM described in Section 3.1.
JVMMon Architecture: Figure 3 shows the main components of *JVMMon* and their interconnections. The JVMTI agent on the `Monitored JVM` collects data about JVM state, handling events triggered by the JVM itself. This information is then sent to the `Local Monitor Daemon` which computes the state of the monitored JVM. These two components are deployed on the same host and communicate through shared memory. Each event raised by the monitored JVM is also logged on a local file. Moreover, the `JVMTI agent` sends a heartbeat message at a fixed rate in order to make the *Local Monitor Daemon* aware of JVM crash/hang failures.
The *Local Monitor Daemon* notifies the *Data Collector* about JVM failures and relevant state changes. These events are then written in the `Event Database`. Each time the `Data Collector` is notified, a snapshot of JVM state is retrieved and stored in the `State Snapshots Database`.
JVMTI Agent
This component is a shared library loaded at JVM startup. It is in charge of:

[‖] The jstat tool displays performance statistics for an instrumented HotSpot JVM

Table III. JVM workload parameters captured by JVMMon

COMPONENT	PARAMETER	DESCRIPTION	U.M.
Runtime Unit	**MET_INV**	Method Invocation Rate	*methods / min*
	OBJ_ALL	Object Allocation Rate	*objects / min*
	ARR_ALL	Array Allocation Rate	*arrays / min*
Class Loader	**CLS_INIT**	Time spent in class identification	*ms*
	CLS_VER	Time spent in class verification	*ms*
	CLS_LOAD	Time spent in class loading	*ms*
	INIT_CLS	Number of initialized classes	*classes / min*
Memory Management	**SM**	Object size mean	*bytes*
	SV	Object size variance	*bytes*
	COLLECTOR0_INV	Number of young generation collector (copying) invocations	*invocations/min*
	COLLECTOR0_TIME	Time spent during young generation collection	*ms*
	COLL0_TIME_PER_INV	Young collection duration	*ms*
	COLLECTOR1_INV	Number of tenured generation collector (compacting) invocations	*invocations/min*
	COLLECTOR1_TIME	Time spent during tenured generation collection	*ms*
	COLL1_TIME_PER_INV	Tenured collection duration	*ms*
	SAFEPOINTS	Number of safepoints(*) reached	*Safepoints/min*
Jit Compiler	**CI_THR0_EVENTS**	Compiler thread 0 events	*Events/min*
	CI_THR0_TIME	Compiler thread 0 time	*ms*
	CI_NATIVE_COMPILES	Native JIT-compilations	*Events/min*
	CI_NATIVE_TIME	Native JIT-compilation time	*ms*
	CI_OSR_COMPILES	On-Stack-Replacement (OSR) (OSR) JIT-compilations	*Events/min*
	CI_OSR_TIME	OSR JIT-compilations time	*ms*
	CI_STD_COMPILES	Standard JIT-compilations	*Events/min*
	CI_STD_TIME	Standard JIT-compilation time	*ms*
	CI_TIME_PER_COMP	Time per compilation	*ms*
Thread Management Unit	**TE**	Threading Events	*Events/min*
	WM	Waiters (threads waiting on a mutex) mean	*# of threads*
	WV	Waiters (threads waiting on a mutex) variance	*# of threads*
	NWM	Notify waiters (threads waiting on a condition variable) mean	*# of threads*
	NWV	Notify waiters (threads waiting on a condition variable) variance	*# of threads*

**The Hotspot JVM uses a safepointing mechanism to halt program execution only when the locations of all objects are known. When a safepoint is reached, each thread stops its execution, enters the VM and stores its current Java Stack Pointer.*

1) Handling events generated by the JVM, implementing JVMTI callback functions.
2) Retrieving data about JVM state through both JVMTI API and BCI.

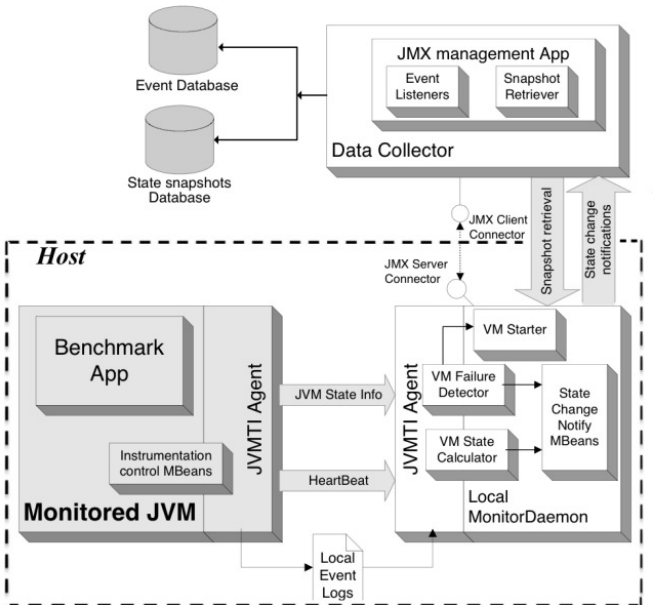


Figure 3. JVMMon Architecture

3) Sending retrieved data to the `Local Monitor Daemon`.
4) Storing data about events in the `Local Event Log`.
5) Storing resource usage data in the `Resource Usage Log`.
6) Storing workload information in the `Workload parameters Log`.

Local Monitor Daemon

This component runs on a separate, non-instrumented, JVM. It communicates with the monitored JVM through a shared memory and sends notifications of failures or relevant state changes to the Data Collector. The `VM state calculator` sub-component is responsible for computing the current state of the JVM; the `State Change Notify MBeans` sub-component copes with the notifications towards the Data Collector.

The `VM Failure Detector` sub-component is in charge of detecting JVM failures. The detection is performed at three different layers: i) *Process Layer*: JVM crashes (i.e., a SIGSEGV failure) are detected by checking for its PID; ii) *Local log Layer*: Hang failures (i.e., a deadlock between Java or VM threads) are detected by checking whether events are being written on the local log or not; iii) *Communication Layer*: the failure detector listens on a socket for heartbeat messages and checks if the monitored VM is still able to communicate.

Data Collector

This component collects data from multiple instrumented Virtual Machines. A connection is

established with each `Local Monitor Daemon`. `Event listeners` handle state change notifications whereas the `Snapshot retriever` performs state snapshot retrieval. Data is then stored in the `Event Database` and in the `State Snapshot Database`.

4. Aging Analysis Steps

When an OTS item providing a complete virtualization of the execution environment, like the JVM, is employed, it may happen that aging trends are not detected at all, since they develop at the JVM level, and are not visible at the OS level. For instance, since the JVM pre-allocates system memory required for its heap area, there is no chance at the OS level to distinguish how much of such heap area is actually used by Java objects.
Moreover, even isolating the JVM aging contribution, its relationship with JVM-specific workload parameters, as those in Table III, should also be taken into account. In fact, such parameters describe the usage of the JVM subsystems, hence providing insights about potential sources of aging within the JVM. However, the number of such parameters can easily reach an order of magnitude that makes it difficult to evaluate the influence of each parameter on the measured aging trends. Therefore, when targeting software aging analysis in Java-based systems, two challenging questions arise:

- *Is it possible to isolate the contribution of each layer to the overall aging trends?*
- *Is there a way to select only those workload parameters that are relevant to the development of aging phenomena?*

As for the first question, it has been previously argued that monitoring system resources only at the OS layer does not allow to gain insights about the behavior of each layer. Thus, an approach where these resources are monitored at each layer is preferable. In this way, given a particular resource (e.g., used memory), it will be possible to compare its usage at different layers, and locate the layer(s) in which aging phenomena are introduced.
As for the second question, it is important to address the selection of the smallest set of workload parameters that have the greatest influence on aging phenomena. This problem, also known as *dimension reduction* or *feature selection*, is one of the most prevalent topics in the machine learning and pattern recognition community. In the approach adopted in this paper, *Principal Component Analysis* [37] is employed, which is a technique to transform original data into a set of (first-order) uncorrelated data, and then statistical *Null Hypothesis* testing to select only those workload parameters that have a real influence on aging phenomena. Finally, *Partial Linear Regression* is employed in order to estimate relationships between workload and aging trends. The choice of linear models is reasonable for aging phenomena, since each experiment is conducted with a constant workload and invariable conditions for a long time, with the goal of activating repeatedly aging bugs and depleting resources progressively, i.e., proportionally to time. Several studies [13, 11, 20] have shown that linear models are able of describing such relationships in a reliable way.
To address these issues, the approach presented here is characterized by three phases, described in the next sections:

1. **Design and Realization of Experiments**, where several long-running experiments are planned and executed under different workload levels, in order to stress the system and observe its

behavior. In this phase, data about system resource usage and about workload parameters at different layers are collected.
2. **Workload Characterization**: in this phase, workload data are analyzed in order to characterize the system behavior as a function of workload level imposed during the experiments.
3. **Software Aging Analysis**, in which aging trends at different layers and their relationships with relevant workload parameters are evaluated.

4.1. Design of Experiments

The first step of the conducted aging analysis is the experimental design. The goal of this step is to define and to execute the list of experiments needed to evaluate aging dynamics, and their relationship with workload parameters. Considering the *Design of Experiments* (DOE) approach [38, 39], the main points of this phase are the definition of application-level load that will stress the system, and the definition of what information has to be collected during experiments. The latter concerns both workload parameters indicating the usage of the observed layer, and system resource usage indicating potential aging trends development. In the following, details of this step are provided.
The Design of Experiments (DOE) is a systematic approach to investigate systems or processes. A series of structured measurement experiments are designed, in which planned changes are made to one or more input factors of a process or system. The effects of these changes on one or more response variables are then assessed. For the purpose of the analyses conducted in this paper, a limited form of DoE is considered, in which the number of experiments is determined by one input factor, representing the application-level load.
The first step in planning such experiments (also called factorial experiments) is to formulate a clear statement of the objectives of the experimental campaign; the second step is concerned with the choice of **response variables**; the third step is to identify the set of all **factors** that can potentially affect the value of response variables; a particular value of a factor is usually called *level*. A factor is said to be *controllable* if its level can be set by the experimenter, whereas the levels of an **uncontrollable** or **observable** factor cannot be set, but only observed. Given m controllable factors, an m-tuple of assignments of a level to each of those factors is called a **treatment**. The number of treatments required to estimate the effects of factors on response variables is determined by the number of controllable factors and the number of levels assigned to each factor. Given m factors and q levels, the number of required treatments is $N = q^m$. A response variable y can then be written, using several kinds of model. The most common models are the **effects model** and the **regression model**. In particular, assuming a two-factor factorial experiment, the response variable can be written, according to an *effects model*, as:

$$y_{i,j,p} = \mu + \alpha_i + \delta_j + (\alpha\delta)_{ij} + \epsilon_{i,j,p} \qquad for \begin{cases} i = 1, 2, \ldots, a; \\ j = 1, 2, \ldots, b; \\ p = 1, 2, \ldots, c. \end{cases} \tag{1}$$

where μ is the overall mean effect, α_i is the effect of the ith level of the first factor (which has a levels), δ_j is the effect of the jth level of the second factor (which has b levels), $(\alpha\delta)_{ij}$ is the effect of the interaction between α_i and δ_j; $\epsilon_{i,j,p}$ is the random error term referred to the pth observation (out of c observations). The $\epsilon_{i,j,p}$ variables are typically assumed to be mutually independent, normally distributed random variables with zero means and common variance σ^2 (as also assumed in the following experiments). Given Equation 1 describing the *effects model*, the corresponding *regression*

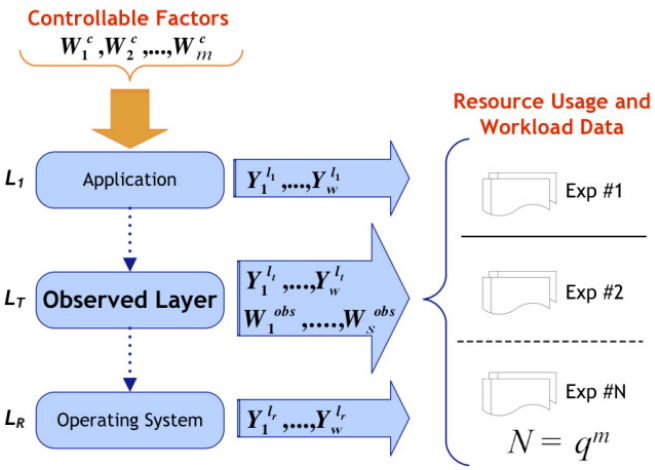


Figure 4. Design of Experiments for Software Aging Analysis in OTS-based systems

model representation of the factorial experiment is the following:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \epsilon \tag{2}$$

that is a linear combination of the factors x_1, x_2 and their products, where:

- the parameter β_0 is called the *intercept parameter* and does not describe the effect of any factor onto the response variable;
- the parameters β_i are the *main effect* parameters and describe the effect of each factor on the response variable;
- the parameter β_{12} is the *interaction* parameter and describes the effect of the combination of the two factors on the response variable;
- the ϵ term is the random variable error term that captures the effect of all uncontrollable parameters.

This representation, whose parameters are derived by the **least squares** method, is particularly useful when all the factors in the experiment are quantitative. Since the presented analysis deals with quantitative factors, the *regression model* representation is adopted in the following.
Figure 4 depicts the approach to employ the DOE technique for software aging analysis. Among the several layers in which a Java-based software system may be divided, a particular layer (l_t) is chosen as the target of the analysis. The goal of the conducted experiments is to analyze the effects of changes in the workload parameters of the target layer on software aging trends. The latter, measured in terms of memory depletion and performance degradation, will be the response variables.
As shown in Figure 4, application-level workload parameters can be controlled by the experimenter, in order to stress the whole system with synthetic workloads, whereas component-level workload

parameters may only be observed. In this study, a series of experiments $Exp\#1 \dots Exp\#N$ will be conducted, in which levels of the m controllable parameters $W_1^c, \dots, W_m^c$ are set to ensure that each experiment will be executed with a different workload (i.e., with a different combination of levels). For each experiment, the following data will be collected: **i)** resource usage data for layers $l_1, \dots, l_r$, needed for aging trends estimation; they are denoted, in Figure 4, as $Y_1^{l_j} \dots Y_w^{l_j}$, for a given layer l_j, and w resource usage indicators; **ii)** values of observable workload parameters $W_1^{obs}, \dots, W_s^{obs}$ related to the observed target layer l_t.
Most of the employed factors are uncontrollable. We are therefore more interested in the ϵ term of the Expression 1 and 2 rather than in the effects of controllable parameters. Standard statistical analysis techniques, such as *Analysis of Variance (ANOVA)*, which analyze only the effect of controllable factors on response variables (adopting the model of Equation 1), cannot be employed. Given this scenario, for a fixed number of experiments, the number of controllable factors has been reduced to only 1 factor, incrementing at the same time the number of levels for it, and focusing the attention on the effects of uncontrollable factors through subsequent regression analyses. The experiments will then be performed in the following way:

1. Choose a controllable factor to drive the synthetic workload applied for each experiment;
2. Select the target layer and its workload parameters that have to be monitored, grouped by the particular component or sub-system they belong to;
3. Define the number of levels q to assign to the above mentioned controllable factor;
4. Choose an interval to sample resource usage and workload information during each experiment;
5. Run the $N = q$ experiments, collecting resource usage information for each layer, and workload information for the target layer.

4.2. Workload Characterization

Once data have been collected, the next step of the methodology is concerned with workload characterization. Specifically, the purpose of this phase is to identify which workload parameters are more relevant for the subsequent aging analysis. Parameters evolution is first observed within each experiment, then as function of the application-level workload. Finally, redundant information is removed: the output is a minimal set of uncorrelated variables describing workload evolution, to be related with software aging in the last step. The specific goals of the workload characterization are the following:

- ***Intra-Experiment Characterization***: perform a statistical characterization of the behavior of the target layer for each experiment. For each workload parameter, a cluster analysis is performed to identify the different workload states that can be visited during the experiment. Then, mean and trend of each workload parameter in each identified cluster (i.e., workload state) are computed.
- ***Inter-Experiment Characterization***: identify variations among experiments by relating synthetic data extracted from each experiment with the controllable workload parameter W^c.
- ***Principal Component Analysis***: reduce the complexity of the analysis by minimizing, through the Principal Component Analysis technique, the number of variables to consider when assessing the relationships with aging trends.

The three phases, reported in the conceptual diagram of Figure 5, are described in the following.

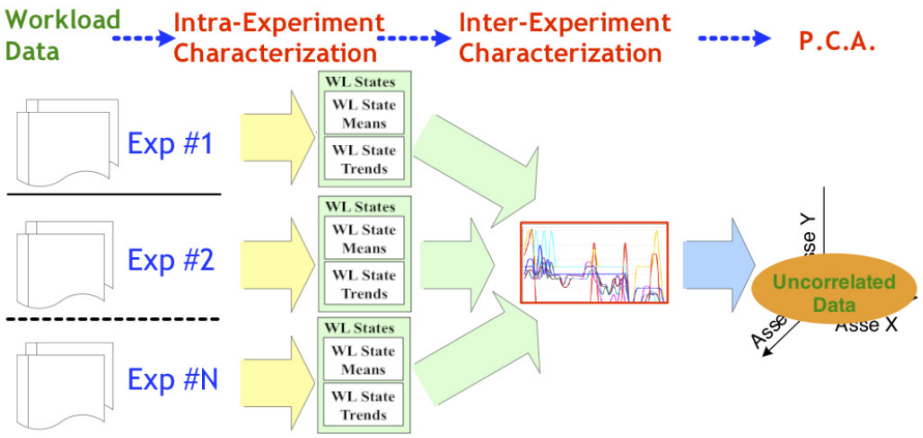


Figure 5. Workload characterization phases

Intra-Experiment Characterization

The first step of the conducted analysis deals with the **detection of clusters**, in order to identify the different workload states traversed by the observed layer during the experiment. The workload parameters $W_1^{obs}, \ldots, W_s^{obs}$ are grouped according to the component they belong to; for each of these groups a cluster analysis is performed by using the *Hartigan's k-means clustering algorithm* **.

If the variables for clustering are not expressed in homogeneous units, a normalization must be applied. In the developed methodology, the following normalization method is used:

$$x_i' = \frac{x_i - min_i\{x_i\}}{max_i\{x_i\} - min_i\{x_i\}} \tag{3}$$

where x_i' is the normalized value of x_i. Through this transformation, all time series $W_i^{obs}(t)$ are transformed into normalized time series $W_i^{obs'}(t)$ whose values range between 0 and 1. In order to augment the effectiveness of cluster analysis, outliers in data are eliminated by removing samples whose (euclidean) distance from the mean value of the time series was higher than 2 times its standard deviation.

The clustering algorithm starts by assigning an initial value to each centroid. Then, it iteratively updates centroids and assigns points in normalized data series to the closest centroid until centroids no longer move. The choice of the number of clusters is a crucial point in the workload characterization for a given component C, since it defines the number of states traversed by the component during the experiment. To this aim, an approach based on *frequency count* [40] has been used. Given a component C in the target layer l_t, whose related workload parameters are $W_1^{obs,C}, \ldots, W_g^{obs,C}$, the range of

**The Hartigan's algorithm is a well-known algorithm for cluster analysis ("Clustering Algorithms, John Wiley and Sons, 1975")

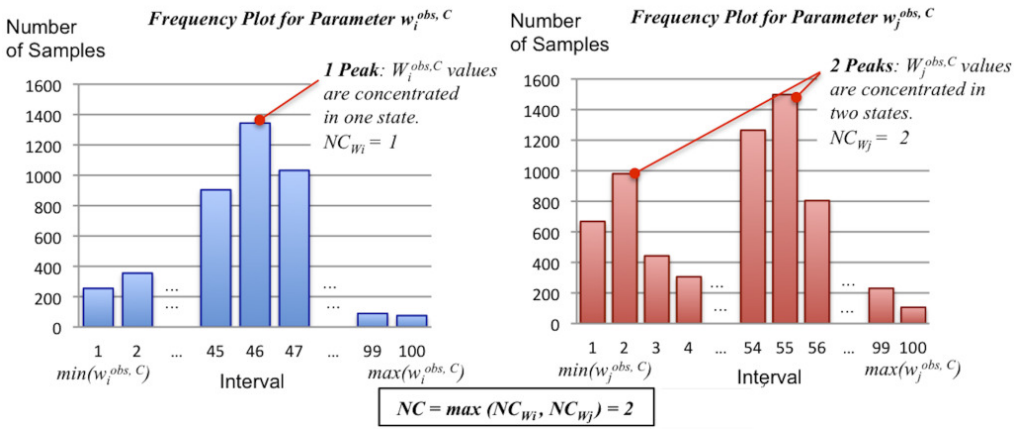


Figure 6. An example of the frequency count approach, for a component C with two parameters $W_i^{obs,C}$, $W_j^{obs,C}$

.

values $[w_{i_{min}}^{obs,C}; w_{i_{max}}^{obs,C}]$ is considered for each parameter $W_i^{obs,C}$, that is then divided into 100 equally sized intervals (the choice of 100 intervals is tied to the chosen clustering algorithm). For each interval, the number of $w_i^{obs,C}$ samples falling into that interval is counted; in this way the number of sample occurrences is described as a function of the interval. In order to infer the number of clusters, the number $N_{C_{W_i}}$ of relative peaks in the frequency count function is first calculated, for each workload parameter. Then, the number of cluster is determined as:

$$NC = max\{N_{C_{W_1}}, N_{C_{W_2}}, \ldots, N_{C_{W_g}}\} \tag{4}$$

An example of the frequency count process is described in Figure 6.

It is worth noting that the number of clusters affects the complexity of subsequent analyses, since it represents the number of workload states and it determines the number of data series to build. To reduce such a complexity, this number could be reduced by merging clusters, e.g., by choosing the two clusters that result in the smallest increase in total SSE (sum of the squared error), or by merging clusters with the closest centroids. The choice of how many clusters to merge depends on the desired trade-off between the complexity of the analysis and its accuracy, that is related to the accuracy of clustering. In this analysis, no clusters reduction was applied.

Once the cluster analysis has been carried out, the intra-experiment characterization is completed by calculating the **expected values** for each workload parameter and for each cluster, and by estimating the potential **linear trends** in time series. This step is carried out in order to have a synthetic measure of the average behavior of a workload parameter in each workload state, and a measure of how it evolves during successive visits in that state, for a given experiment. In the next step, these measures act as independent variables in the inter-experiment characterization, where their evolution across different experiments is observed.

Since the cluster analysis splits the original time series into several non contiguous intervals, each one representing a different visit in a particular workload state, an approach for trend estimation, which is similar to the one adopted in time series with seasonal patterns, has been adopted. In particular, the trend for each interval is first estimated, and then a weighted average of the trend estimated for each interval is computed.
Assuming that the cluster analysis revealed the presence of V clusters for the component C, the intra-experiment characterization will therefore calculate, for each workload parameter $W_i^{obs,C}$, the expected value:

$$E[W_i^{obs,C}]_j = \frac{1}{n}\sum_{f=1}^{n} w_{i,f}^{obs,C} \; with\, f \in j^{th}\, Cluster,\; for\, j = 1 \ldots V. \tag{5}$$

where $w_{i,f}^{obs,C}$ denotes the number of samples of the corresponding workload parameter in the interval f, with n denoting the number of intervals. Regarding the linear trend, given a set of *r* non contiguous intervals, and called $m_1, \ldots, m_r$ the number of data samples in each one of these intervals, the linear trend in data, for the jth cluster, may be expressed as:

$$T[W_i^{obs,C}]_j = \frac{1}{n}\sum_{f=1}^{r} m_f\, TREND(W_i^{obs,C})_f \tag{6}$$

where `TREND` is the autoregressive function employed to calculate the trend in a single interval, and m_f denotes the number of data samples in each interval f. To assess if a trend exists in f, *statistical hypothesis testing* is applied. Null hypothesis testing is a common procedure in which a first hypothesis, called the *null hypothesis* and denoted with H_0, is tested against an alternative hypothesis, denoted with H_1. The null hypothesis usually refers to a condition in which a particular treatment does not have any effect on the output variable. In this case, the null hypothesis of *no trend* in data is tested against the alternative hypothesis stating the presence of a linear trend in data, using the *student's t* statistic.
The *student's t* test is a hypothesis test in which the *test statistic* (i.e., a random variable that is function of the random samples, and such that its distribution is known) follows a *student's t* distribution if the null hypothesis is supported. The null hypothesis has to be rejected when the calculated value for the statistic falls into the tails of the t distribution. The border between the center and the tail of the distribution is set up according to the chosen significance level: the lower is the confidence level, the higher is the probability of being in the tail of the distribution.
Thus, the value of `TREND` for an interval f is 0 in the case that the null hypothesis of no trend in data cannot be rejected, as it often happens when dealing with intervals with a low number of data sample (i.e., intervals related to short visits in a particular workload state).
More in detail, in order to verify the presence of a trend, the test considers the linear regression of the dependent variable Y on the independent one X (that is time in trend analysis), namely $Y = a + bX$. If a trend exists, b should be different from 0. The test is based on the consideration that the statistic $t = r(\sqrt{n-2})/\sqrt{1-r^2} = (b-\beta)/S_b$ follows a *student's t* distribution with $n-2$ degrees of freedom [41], where:

- n is the sample size;
- r is the Pearson correlation coefficient;
- b is the regression coefficient;

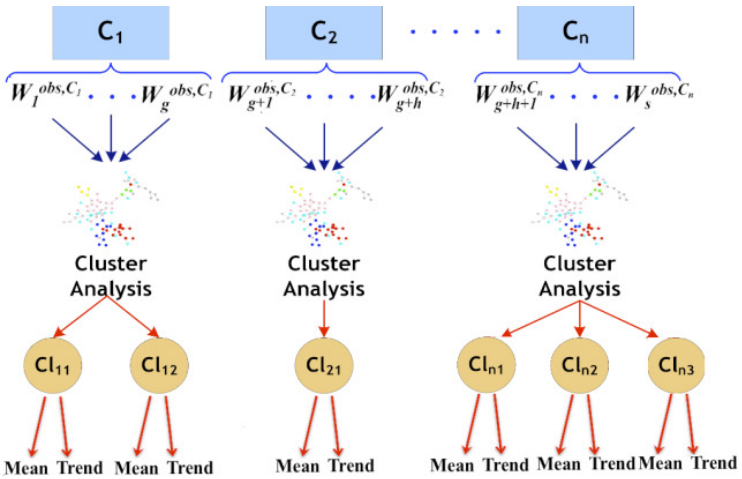


Figure 7. The Intra-Experiment Characterization Process, with n components and s workload parameters

- β is the regression coefficient value hypothesized in H_0, which is $\beta = 0$ in this case;
- S_b is the standard error of the coefficient estimate. The latter is equal to the standard deviation of residuals, s, over the square root of the sums of squares of the independent variable, SS_x: $s/\sqrt{SS_x}$.

The hypothesis $H_0 : \beta = 0$ is tested against the hypothesis $H_1 : \beta \neq 0$ at the chosen level of significance α (that is $\alpha = 0.05$ in this case). The test is based on the idea that if the null hypothesis is supported (i.e., $H_0 : \beta = 0$, that means no trend in data) the statistic t follows a *student's t* distribution; whereas if the regression coefficient b is systematically different from 0 (i.e., $H_1 : \beta \neq 0$ supported), than the t value progressively diverges from the center of the distribution. When the t value becomes greater than the critical value $t_{\alpha/2}$ (since a *two-tailed* test is considered), than this means that the difference between b and $\beta = 0$ is significant, and the hypothesis that there is no trend in data is rejected.

The sequence of operations needed to perform the intra-experiment characterization is reported in Figure 7: once workload parameters are grouped according to the layer's components, a cluster analysis is performed on each of these groups; after that, the expected values and the linear trend in data are calculated for each cluster found in the previous phase, by using Equation 5, Equation 6, and the described *student's t* test.

Inter-Experiment Characterization

This phase aims at analyzing the impact of workload parameters on aging trends, by building new data series from synthetic data obtained in the previous phase. By this way, it will be possible to observe the evolution of the workload parameters $W_1^{obs}, \ldots, W_s^{obs}$ as a function of the controllable workload parameter W^c, which regulates the synthetic workload imposed on the overall system.

For instance, in Java-based systems, where the target layer is supposed to be the Java Virtual Machine, a typical workload parameter to be monitored is the *Object Allocation Frequency*, i.e., the number of objects inserted into the Java Heap in a given unit of time. Assuming that no clusters are detected, the intra-experiment analysis returns, for each experiment, the average object allocation frequency and its trend. The inter-experiment characterization, in turn, builds two new data series describing the average object allocation rate and its trend as a function of the synthetic workload imposed on the application, thus allowing to gain some insights about the JVM's heap behavior. Assuming s observable parameters and V clusters, the number of data series constructed in this phase is $s * V * 2$. Indeed, it is necessary to build a data series not only for each observable workload parameter, but also for each workload state visited by this parameter. Since cluster analysis is performed for each experiment, it may happen that two different experiments reveal a different number of clusters. This situation simply indicates that one or more workload states have not been visited during some experiments. In other words, it may happen that data series built during the inter-experiment phases have some missing points. Moreover, if one or more workload states are visited only in a few experiments, they may be treated as outliers and excluded from subsequent analysis.

Principal Component Analysis

It is very unlikely that data series returned by the inter-experiment characterization are uncorrelated. Correlation among data may distort the analysis of the effects of workload parameters on aging trend: indeed a higher weight would be given to correlated variables, thus actually amplifying the effects of such variables on aging trends.

In order to partially remove correlation among data (first-order correlation), Principal Component Analysis (PCA) [37] is applied, which transforms original data into a set of new uncorrelated data. As for cluster analysis, data have to be normalized first, on order to have all data series with the same weight.

PCA computes new variables, called *Principal Components*, which are linear combination of the original variables, such that the first-order correlation among all principal components is removed. PCA transforms m variables $X_1, X_2, \ldots, X_m$ into m principal components $PC_1, PC_2, \ldots, PC_m$ such that:

$$PC_i = \sum_{j=1}^{m} a_{ij} X_j \qquad with \quad 1 \leq i \leq m \tag{7}$$

The values of the a_{ij} coefficients are in the range $[-1; 1]$. This transformation has the following properties:

i) $Var[PC_1] > Var[PC_2] > \ldots > Var[PC_m]$, which means that PC_1 contains the most of information and PC_m the least;

ii) $Cov(PC_i, PC_j) = 0 \ \forall i \neq j$ this means that there is no information overlap among the principal components. Note that the total variance in the data remains the same before and after the transformation, i.e., $\sum_{i=1}^{m} Var[X_i] = \sum_{i=1}^{m} Var[PC_i]$.

Principal components are decreasingly ordered according to their variance. It is therefore possible to remove the last components, which have the lowest variance, thus reducing the number of variables to take into account for assessing the relationships between workload and aging trends. Removing the last components guarantees that only a minimum percentage of information contained in the original data is thrown away. Typically, a very small percentage of original variables (e.g., 10%) are able to explain from 85% to 90% of the original variance. Therefore, for each component in the observed layer, a subset z of the calculated m principal components is selected, being $z \ll m$. Each of these

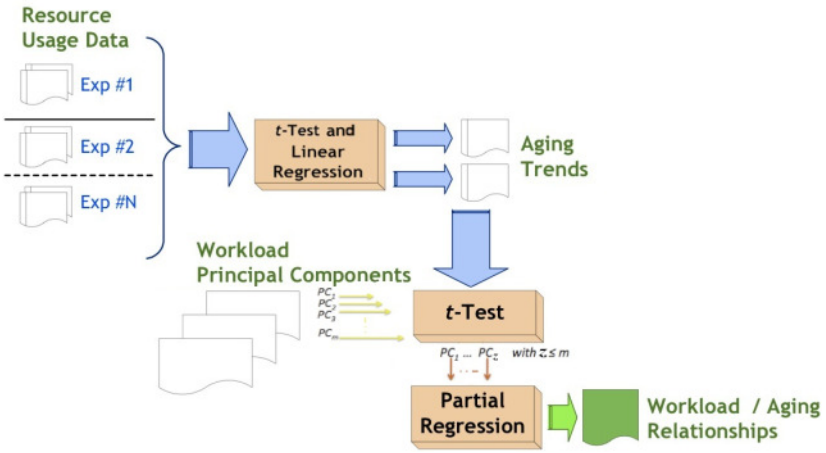


Figure 8. Software Aging Analysis process

components is expressed as in the Equation 7.
Thus, principal components are linear combinations of parameters; parameters contribute to each PC through the a_{ij} coefficients, which express the weight that each workload parameter has in the principal component(s), where it appears. A weight that is close to 1 (or -1) means that the workload parameter has a very high impact on the principal component, whereas a weight that is close to 0 means that the workload parameter has a negligible impact on the principal component. The software aging analysis will select the principal components that have the greatest influence on measured aging trends; after that, analyzing the composition of each principal component, it will be possible to identify workload parameters that are more relevant to aging trends.

4.3. Software Aging Analysis

The purpose of the last step is to detect and estimate aging trends in resource usage data, and then to analyze relationship between such trends, if any, and the workload characterized in the previous step. Variables describing the resource usage, represented by $Y_j^{l_i}$ in Figure 4, are monitored in each experiment in order to collect resource usage data. They act as response variables of the aging trend analysis over time, and they are measured in terms of memory depletion and performance degradation. Then, the relationships between detected trends and workload parameters are analyzed.

The process to accomplish this step is depicted in Figure 8. The first phase deals with the detection and the estimation of aging trends in resource data, collected during the experimental campaign. The second phase deals with the assessment of the relationship between the estimated aging, if any, and component-layer workload parameters.
To this aim, the following steps are carried out: **i) Hypothesis testing**, to assess the presence of trend in

data, **ii) Linear regression**, to estimate this trend, if present, and **iii) Hypothesis testing on principal components** and **multiple regression**, for the assessment of the aging-workload relationship.

The same test as the one adopted for trend detection in workload data series (see Section 4.2) has been used for assessing the presence of aging trends in resource usage data: namely, the *student's* test. If the presence of aging trend in data is confirmed by this test, linear regression is applied to estimate such a trend. Linear regression is a method that models the relationship between a dependent variable Y, independent variables $X_i, i = 1, \ldots, n$, and a random term ϵ. Linear regression has many practical uses. Commonly, it is used for prediction, i.e., to find the best model fitting an observed data set of Y and X values in order to predict value of Y given a new value of X. In this case, the X_i variables are called *predictor variables*. Linear regression is also used to quantify which variables out of a given set $X_i, i = 1, \ldots, n$, is more or less related to the response variable Y, i.e., which of them better "explain" the variation of Y. In such a case, the X_i variables are also referred to as *explanatory variables*.
The linear regression model can be written as $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots \beta_n X_n + \epsilon$, where β_0 is the intercept ("constant" term), the β_i are the respective parameters of independent variables, and n is the number of parameters to be estimated in the linear regression. The ϵ term represents the unpredicted or unexplained variation in the response variable; it is conventionally called the "error", regardless of whether it is really a measurement error or not, and it is assumed to be independent of the X_i variables. As far as resource usage data are concerned, only the time is taken into account as independent variable, thus the regression model can be reduced to $y = a + bx + \epsilon$. In order to carry out the parameters estimation, *least squares analysis* is employed. *Least squares analysis* is a technique that allows finding the curve that best fits a given set of data, where "best" means the curve that minimizes the sum of the squares of the y-coordinate deviations from it.
Aging trends are estimated for each monitored resource, for each layer in the system, and for each cluster detected in the intra-experiment analysis. It is important to repeat parameters estimation for each cluster since, given that each cluster corresponds to a different workload state, there is a high probability that software aging phenomena develop in a different way too.

Once the aging trends have been calculated, linear regression (in this case **multiple regression**) is employed again, in order to assess the relationships between aging trends and the principal workload components built in the workload characterization step. The parameters of regression model, determined by the least square analysis, are validated using the *student's* t test. In particular, for the parameter β_j, the test statistic has the following form: $t_j = \beta_j / SE(\beta_j) = \beta_j / \sqrt{s^2 (X'X)^{-1}_{jj}}$, where SE is the standard error of the least-square estimate, computed as the square root of the diagonal elements of the estimated variance-covariance matrix (i.e., $s^2 (X'X)^{-1}_{jj}$).
Partial regression parameters whose *t-values* do not fall into the tails of the distribution will be discarded: indeed, if the null hypothesis cannot be rejected, there is no effect of the workload principal component on the aging trend. For instance, suppose that in the regression model described by $Y = \beta_0 + \sum_{i=1}^{n} \beta_i PC_i$, only the β_1 coefficient is significant at 95% confidence level; it means that only PC_1 contributes to aging (i.e., to the Y variable). After discarding invalid parameters, the regression model is solved again with the remaining parameters, until all parameters show a *t-value* falling into the tails of the t distribution.
Finally, once the effect of PCs on aging trends has been assessed, the impact of original workload components can be evaluated by analyzing the structure of each principal component. Since each principal component has the form $PC_i = \sum_{j=1}^{m} a_{ij} X_j$, the goal is to express X_j variables as a

function of PC_i components. In general, this is not possible, since the number of principal components z is usually far less than the number of original workload parameters s. To perform a meaningful estimation of the relationships between the original workload parameters and aging trends, given s workload parameters and z principal components, it is possible to proceed as follows:

1. Delete, if possible, s-z variables having a little influence on the principal components, i.e., those variables whose coefficient is relatively close to 0 for each principal component. In this way, workload parameters can be expressed as a function of principal components.
2. For each principal component, consider only the workload parameters with the highest coefficient. Even if it will not be possible to accurately express the relationships between workload and aging trends, an overestimation of such relationships can be obtained. Overestimating the effects of workload on aging trends will lead to predict a shorter time for resources exhaustion, which is always better than predicting a time longer than the actual TTE.

5. Case Study

The presented methodology is used to characterize software aging dynamics in Java-based software systems, isolating the contribution of the JVM layer to the overall aging trends and estimating the influence of the workload on aging. In the following section, the methodology is applied to a case study, namely a Java-based system constituted by the Apache JAMES mail server, the Sun Hotspot JVM, and the Linux Operating System. Aging trends are estimated by evaluating two well-known software aging indicators, i.e., **throughput loss** and **memory depletion**.

5.1. Experimental Setup

JVMMon has been employed to collect resource usage and system activity data from a workstation running JAMES mail server on a Sun Hotspot JVM v.1.5.0_09. The workstation was a dual-Xeon server equipped with 5GB RAM and running Linux OS (kernel v.2.6.16). The JVM was started with the typical `server` configuration[††] and a maximum heap size of 512MB; it was configured to run serial, stop-the-world collectors both on the young and on the tenured generation. No other application was competing for system resources with the JVM: the server workstation was started with just minimal system services. A mail server has been chosen as the benchmark application for the analysis, since it represents an important class of long running server applications usually stressed by significant workloads. The server was stressed by using a load generator, which acts as an email-client, sending and receiving mails at a constant pace for the whole experiment. The load generator allows to control the imposed workload by specifying the number of mails per minute and their size. In the experimental campaign, the number of mails per minute is chosen as the controllable workload parameter, whereas the size of the e-mails is kept constant among all experiments. Figure 9 depicts a single experiment scenario. *JVMMon* collects the greatest part of information required for the analysis. Only data about

[††]The Sun Hotspot VM has been tuned to maximize peak operating speed. It is intended for long-running server applications, for which having the fastest operating speed is generally more important than having the fastest start-up time

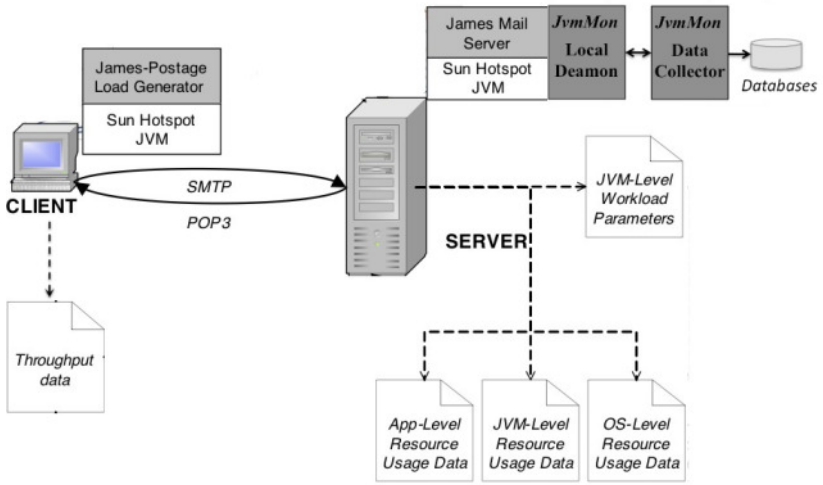


Figure 9. Experimental setup for data collection

server's throughput are extrapolated from logs produced by the load generator.
Moreover, *JVMMon* has been extensively tested in order to guarantee that it is aging-bug free, thus ensuring that the monitoring agent does not introduce any aging phenomenon or overdraws existing ones. In order to choose a proper range for the controllable workload parameter, the capacity of the mail server has been determined first, by identifying the highest workload (in terms of emails per minute) it is able to sustain without refusing any connection. The estimated limit has been of about 1550 mails per minute (i.e., the SMTP server is able to deliver up to 1550 mails/min without errors). A set of 33 experiments has been performed, with the number of mails per minute ranging from 330 mails/min to 1530 mails/min, increasing by 40 mails/min per experiment. From this set, four experiments (namely the experiment number 1,2, 8 and 9) have been discarded, which yield non-analyzable results, getting to a final set of 29 experiments (reported in Table IV). These experiments were discarded because the system failed before the established experimental time for causes not due to aging (i.e., for loss of power). The lower bound of 330 mails/min is because it has been observed that the experiments performed under that threshold, no significant work by the garbage collector has been observed, as well as by the JVM. Hence those experiments were not considered.

Table IV reports a summary of these experiments, including the throughput achieved in the first 4 hours of execution. The throughput is measured by considering the number of (K)Bytes processed by both the SMTP and POP3 server in each minute. The value reported in Table IV can be assumed as the throughput achieved by the mail server in *"Normal operation"* mode, when the throughput loss due to aging is not yet noticeable. Indeed, aging manifests its effect only after a long period; in the conducted experiments, it has been observed that in the first 4 hours of execution the throughput loss is not yet noticeable. Hence the throughput value of the first 4 hours has been assumed as the throughput in a

Table IV. Experiment Summary.

Exp	WKL (mail/min)	EXEC TIME (min)	*NORMAL OPERATION* SMTP (KB/min)	POP3 (KB/min)
3	330	6001,19	11105	11481
4	370	6002,05	12430	12857
5	410	6000,79	13808	14306
6	450	6001,61	14885	15396
7	490	6001,75	15913	16450
10	610	6001,52	19602	20301
11	650	6001,47	20789	21525
12	690	6001,73	21803	22562
13	730	6002,07	24042	24882
14	770	6002,38	24980	25846
15	810	6001,78	26050	26967
16	850	6001,86	27082	28037
17	890	6002,35	28736	29746
18	930	6002,42	29656	30704
19	970	6001,98	30698	31779

Exp	WKL (mail/min)	EXEC TIME (min)	*NORMAL OPERATION* SMTP (KB/min)	POP3 (KB/min)
20	1010	6002,51	31855	32998
21	1050	6002,72	33086	34258
22	1090	5469,67	33932	35134
23	1130	5467,69	34968	36179
24	1170	5464,97	35702	36998
25	1210	5463,97	36340	37621
26	1250	5460,86	37.231	38522
27	1290	6002,51	36527	37876
28	1330	6002,89	38742	39994
29	1370	6002,52	39334	40718
30	1410	6001,98	39346	40778
31	1450	6003,02	39726	41132
32	1490	6002,97	42658	43949
33	1530	6002,75	40567	42124

Normal operation mode.
Each experiment runs for 6000 minutes (i.e., 100 hours), and one sample is collected each minute. Fixing the sample collection interval to 1 minute allows capturing dynamics in resource usage and workload parameters that have a relatively small duration; on the other hand, the chosen interval is large enough to avoid noise due to transient phenomena like garbage collections, which occur several times per minute. As for the duration of the experiments, a period of 6000 minutes has been preliminarily observed as sufficient to reveal aging phenomena manifestation. Clearly, a longer period would yield even more accurate results, but 6000 minutes is a good trade-off between the ability to observe significant aging trends and the cost of lab resources usage.

5.2. Workload Characterization

This section discusses the characterization of workload applied to the JVM, by considering the parameters shown in Table III. Thus, the following subsections present and discuss the relevant trends observed for workload parameters. As already discussed, such results describe the behavior of the JVM internal components across the experiments; the trends are then related to the observed aging, in order to figure out which one is relevant from the aging point of view. A separate characterization is performed for each JVM component.
Class Loader
Class loader activity is mainly focused in the startup phase of the JVM, in which the greatest part of classes required by the application is loaded. Since the workload applied for each experiment is constant and rather static, it is possible to expect little activity in this component. This is confirmed by the inter-experiment characterization of classloader-related workload parameters. On average, during 100 hours of execution, less than 1 millisecond is spent in classloading activity. In particular, no class

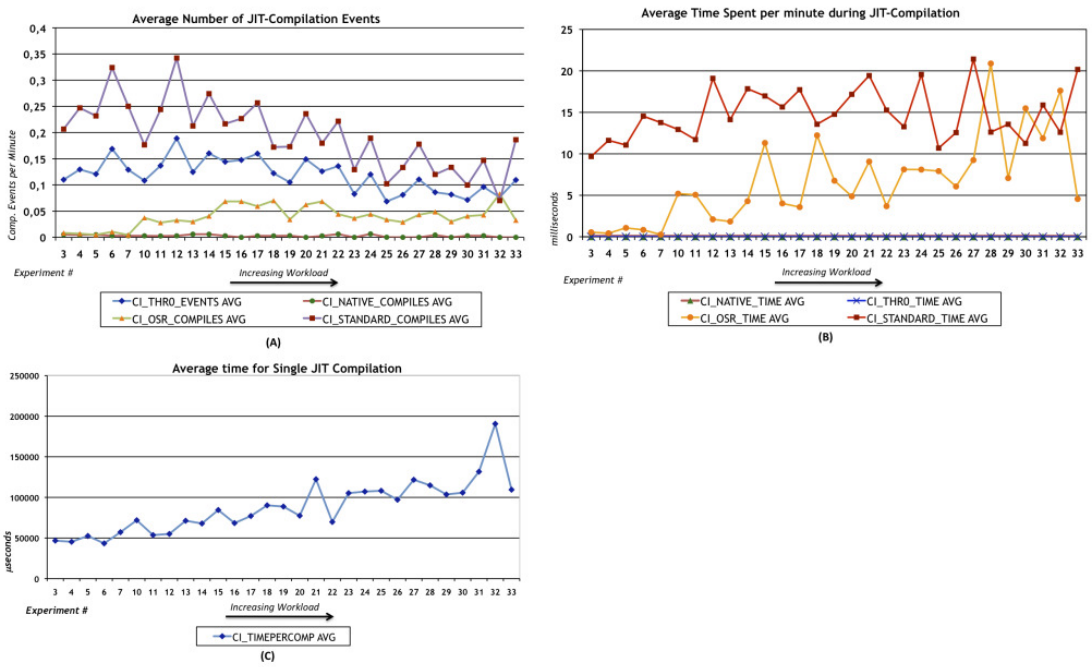


Figure 10. A - Average number of JIT compilation events per minute for each experiments; B - Average time per minute spent during JIT compilation for each experiment; C - Average time per JIT compilation for each experiment

loader activity was observed for 17 out of 29 experiments (58,62%). For the remaining 12 experiments, the null hypothesis of no trend in data cannot be rejected at a significance level of $\alpha = 0.01$[‡‡] and $\alpha = 0.05$. Given this scenario, it is possible to argue that the Class Loader has a negligible impact on aging trends: therefore classloader-related workload parameters will be excluded from the analysis of relationships between workload and aging phenomena.

Just-In-Time Compiler

In all the experiments, none of the JIT-related workload parameters has shown the presence of multiple clusters in data. Moreover, none of these parameters has exhibited trends, except for the ones concerning on-stack replacement compilation (`CI_OSR_COMPILES` and `CI_OSR_TIME`), which have experienced a trend at the $\alpha = 0.01$ significance level.

Figure 10-a shows the average number of JIT-compilation events occurring each minute for each experiment. The high degree of correlation between the counter of JIT-compiler events (`CI_THR0_EVENTS`)

[‡‡]Given a certain α, the percentage of the confidence interval for the estimation is given by $1 - \alpha$

and the number of compilations performed (CI_STD_COMPILES) parameter is evident, whereas the number of native compilations is very small compared with the number of standard and on-stack-replacement (OSR) compilations. Moreover, while the average number of standard JIT-compilations tends to decrease with higher workloads, the number of OSR-JIT-compilation increases with higher workloads (recall that workload is increased linearly across experiments). From the aging's point of view, this means that if aging also increases with higher workload, the usage of the OSR compilation feature *may* be related to aging. However, this will be confirmed or rejected only in the last step of the analysis, when i) the inter-correlations among variables are removed by the PCA, ii) the aging is observed against the workload increase, and iii) a multiple regression analysis identifies the most relevant variables related to aging.

Figure 10-b shows the average time spent during JIT-compilation. While the time spent in OSR compilation (CI_OSR_TIME) clearly shows an increasing trend (thus showing a strong correlation with the number of OSR compilations, CI_OSR_COMPILES), it is not possible to detect a trend for the time spent during standard JIT compilations (CI_STD_TIME). This is mainly due to the behavior of the parameter describing the time spent for each JIT-compilation (CI_TIME_PER_COMP, in Figure 10-c), which increases as the applied workload increases, thus showing a dependence between applied workload and the time required to perform a single JIT compilation. It is therefore possible to expect a certain impact of JIT compilation workload parameters on aging dynamics inside the JVM. In order to remove intercorrelation among these parameters, the PCA is applied, obtaining 4 principal components, reported in Table V-a. These 4 principal components account for 80.05% of variance in the original sample set.

Table V.

(a) Principal components for JIT-compiler Workload Parameters

	CI_PC_1 40,78%	**CI_PC_2 15,91%**	**CI_PC_3 14,80%**	**CI_PC_4 8,56%**
CI_THR0_EVENTS_AVG	-0,131	0,161	0,208	0,226
CI_THR0_TIME_AVG	0,023	0,036	0,249	-0,223
CI_NATIVE_COMPILES_AVG	-0,076	0,260	-0,290	0,203
CI_NATIVE_TIME_AVG	-0,022	0,325	-0,280	0,171
CI_OSR_COMPILES_AVG	0,105	0,136	0,230	0,272
CI_OSR_COMPILES_TREND	0,132	0,227	0,080	0,005
CI_OSR_TIME_AVG	0,153	0,086	-0,047	0,119
CI_OSR_TIME_TREND	0,152	0,141	-0,093	-0,033
CI_STD_COMPILES_AVG	-0,157	0,101	0,134	0,124
CI_STD_COMPILES_TREND	0,105	0,096	-0,074	-0,422
CI_STD_TIME_AVG	0,000	0,258	0,339	0,012
CI_STD_TIME_TREND	0,100	-0,099	-0,039	0,335
CI_TIME_PER_COMP_AVG	0,159	0,004	0,066	0,036
CI_TIME_PER_COMP_TREND	0,070	-0,237	0,027	0,517

(b) Principal components for Execution Unit and Threading Workload Parameters

	EXEC_PC_1 82,64%	**EXEC_PC_2 11,19%**
MET_INV_AVG	0,145	0,260
MET_INV_TREND	-0,137	-0,183
OBJ_ALL_AVG	0,148	0,201
OBJ_ALL_TREND	-0,138	0,334
ARR_ALL_AVG	0,147	0,213
ARR_ALL_TREND	-0,139	0,230
TE_AVG	0,145	0,214
TE_TREND	-0,091	0,849

Execution Unit and Thread Management Unit

Figure 11-a highlights a direct relationship between the number of mails per minute and workload parameters such as method invocation rate (MET_INV), object allocation rate (OBJ_ALL), and array allocation rate (ARR_ALL). Also the average number of threading events (TE) follows the same pattern, indicating a direct relationship between method execution and synchronization events. All these parameters show a decreasing trend (which even becomes negative) as the applied workload

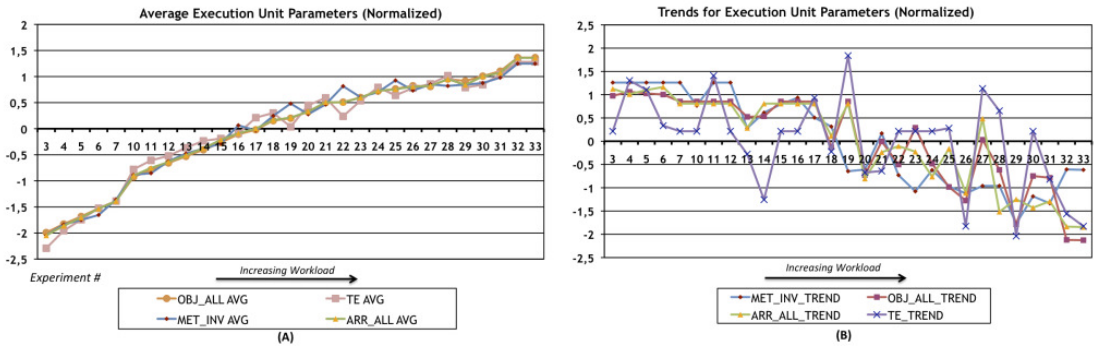


Figure 11. A - Average number of execution related events per minute for each experiment; B - Trends detected for execution related parameters for each experiment; normalized data are reported due to significant differences in data order of magnitude.

increases, as it is shown in Figure 11-b. This indicates a loss of throughput during experiment execution that is proportional to the workload applied to the mail server. Synthetic parameters about threads waiting on monitors and condition variables (`WM`, `WV`, `NWM`, `NWV` parameters) are instead negligible, and they are excluded from the analysis of the impact of workload on aging trends. Figure 11 also shows a high degree of correlation among these parameters: only 2 principal components, reported in Table V-b (namely, `EXEC_PC_1` and `EXEC_PC_2`), account for 93.82% of variance in data (82.64% for the first component). The high degree of correlation of these workload parameters is shown by the small differences in coefficients for the first principal component (`EXEC_PC1`).

Memory Management Unit

These parameters deal with the activity of Garbage Collectors. Unlike previously discussed parameters, memory-related parameters exhibited the presence of clusters. Cluster analysis revealed that garbage collector activity can be divided into two main clusters, defining two well-distinct workload states: the *Normal Collection* state and the *Low Collection* state. 25 experiments out of 29 visited both states, whereas 4 experiments exhibited only the normal collection state.

A similar behavior has been observed in the remaining experiments visiting both states: Table VI

Table VI. Garbage Collection Workload Parameters

	NORMAL COLLECTION				*LOW COLLECTION*			
	AVG	**ST.DEV**	**MIN**	**MAX**	**AVG**	**ST.DEV**	**MIN**	**MAX**
COLLECTOR0_INV	83,686	18,928	42,333	104,549	10,286	3,449	4,701	18,062
COLLECTOR1_INV	0,650	0,185	0,272	0,860	0,028	0,015	0,008	0,062
COLLECTOR0_TIME	226,309	58,259	109,188	290,785	91,812	31,363	40,238	167,666
COLLECTOR1_TIME	83,492	26,900	31,892	116,330	3,657	2,037	0,917	9,334
SAFEPOINTS	92,187	20,854	46,612	115,180	11,787	3,833	5,530	20,157
COLL0_TIME_PER_INV	2,732	0,147	2,491	3,043	11,118	0,494	9,823	12,452
COLL1_TIME_PER_INV	128,058	6,769	114,564	139,000	127,627	7,910	117,182	147,661

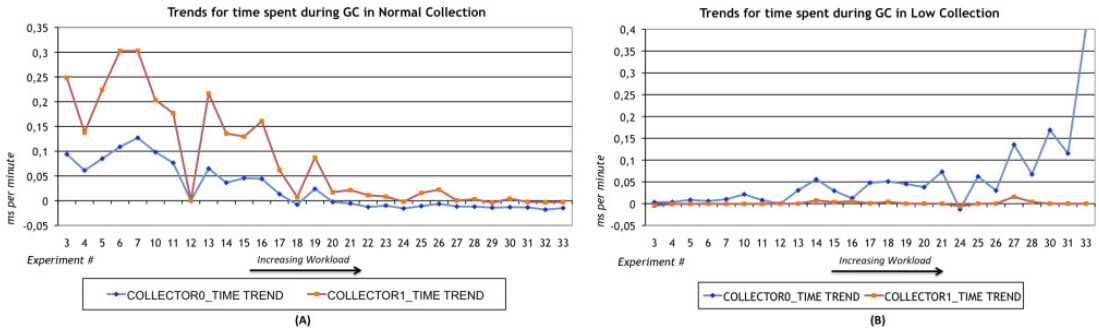


Figure 12. Trends for time spent during garbage collection during normal collection periods (A), and during low collection periods (B)

reports the average values over all the experiments of each memory-related parameter, in both clusters (the difference in the averages also indicates the distinction between clusters).

The *Low collection* state is characterized by low garbage collector invocation rates (especially for the tenured generation collector) and long young generation collection times, whereas the *Normal collection* state is characterized by high collection invocation rates and short young generation collection times. Safepoints reached by the JVM are strongly correlated with young generation collections. Instead, no noticeable variation has been observed for the time spent for each tenured generation collection (`COLL1_TIME_PER_INV`) between normal and low collection periods. Since in *Low collection* state no significant variation in object and array allocation rates is observed, this state represents a potential source of memory depletion of the JVM: objects are allocated with the same frequency, but collections occur less frequently. Furthermore, although the time spent to reclaim unreachable objects, i.e., the time to reclaim memory occupied by objects that can no longer be referenced by the program (hence no longer in use), during normal collection periods is about 3 times higher than the time spent during low collection periods, no throughput increase was observed during visits into the low collection state.

Figure 12 shows the trend exhibited by parameters describing the time spent during young and tenured collections (`COLLECTOR0_TIME` and `COLLECTOR1_TIME`); the remaining parameters show similar trends, except the ones related to the duration of each collection (`COLL0_TIME_PER_INV` and `COLL1_TIME_PER_INV`), for which none of the experiments showed a trend. During normal collection periods, there are impressive trends for both collectors when the applied workload is small; the trend then decreases, becoming negligible for experiments with high workloads. Instead, during low collection periods, there are no noticeable trends for the tenured generation collector (the null hypothesis cannot be rejected), whereas the young generation collector trend increases as the applied workload increases.

Moreover, if the average time spent in garbage collection in both Normal and Low collection state is considered, the different behaviour in both states becomes clear. In particular, Figure 13 shows

Table VII. Principal Components for Garbage Collection Parameters

	NORM _COLL_PC_1 85,25%	NORM _COLL_PC_2 6,46%	LOW_COLL_PC1 61,92%	LOW_COLL_PC2 15,53%	LOW_COLL_PC3 13,34%
COLLECTOR0_INV_AVG	0,092	0,398	0,127	0,032	-0,113
COLLECTOR0_INV_TREND	-0,094	0,066	0,120	-0,145	0,121
COLLECTOR0_TIME_AVG	0,095	0,175	0,122	-0,088	0,012
COLLECTOR0_TIME_TREND	-0,095	0,077	0,115	-0,207	0,111
COLLECTOR1_INV_AVG	0,091	0,330	0,111	0,186	-0,218
COLLECTOR1_INV_TREND	-0,092	0,209	0,058	0,361	0,371
COLLECTOR1_TIME_AVG	0,092	0,211	0,116	0,148	-0,231
COLLECTOR1_TIME_TREND	-0,092	0,209	0,060	0,372	0,347
SAFEPOINTS_AVG	0,092	0,398	0,127	0,031	-0,111
SAFEPOINTS_TREND	-0,094	0,065	0,120	-0,144	0,116
COLL0_TIME_PER_INV_AVG	0,071	-0,749	0,011	-0,311	0,437
COLL1_TIME_PER_INV_AVG	0,080	-0,354	0,112	-0,143	-0,092

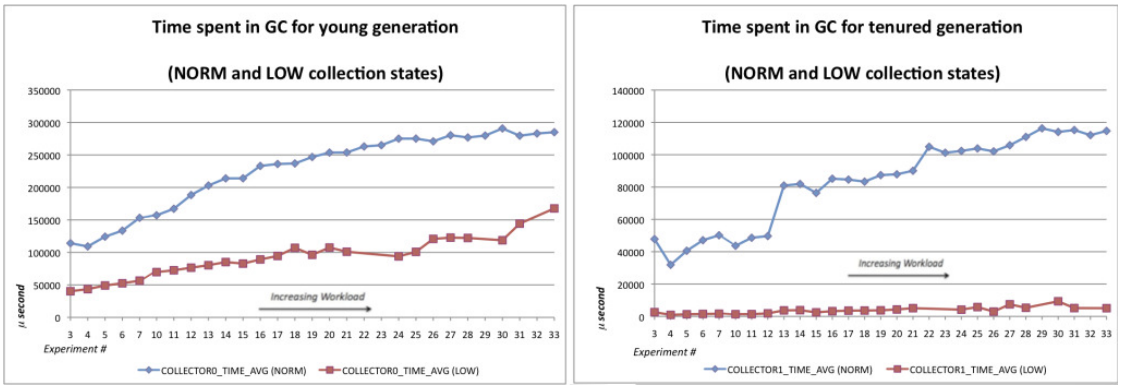


Figure 13. Average Time Spent in Garbage Collection for (A) Young Generation, and for (B) Tenured Generation

that the average time spent in garbage collection in the state LOW is much shorter than the time spent during the NORMAL collection period, for both collectors (young and tenured). This leads the semispaces of the young collector to be always full, and the space of the tenured generation to dramatically increase during the LOW collection period. This causes the observed aging dynamics in the LOW collection state, since while the space occupied by collectors increases, the object allocation rate remains approximately the same (in fact, no presence of clusters was revealed in the execution unit analysis). This behavior is confirmed also by observing the graphs of the average number of invocations of collectors, which is much lower in the LOW state than in the NORMAL state, while the object allocation rate is the same. Figure 14 shows the average number of invocations of both collectors in both states, highlighting this difference in the two states.

Summarizing, the analysis of these workload parameters tells that **i)** trend for memory depletion should be determined for both normal and low collection workload states, and **ii)** there is a high degree of correlation among the remaining parameters. Therefore, a principal component analysis on these parameters in both normal and low collection states was performed, whose results are shown in Table

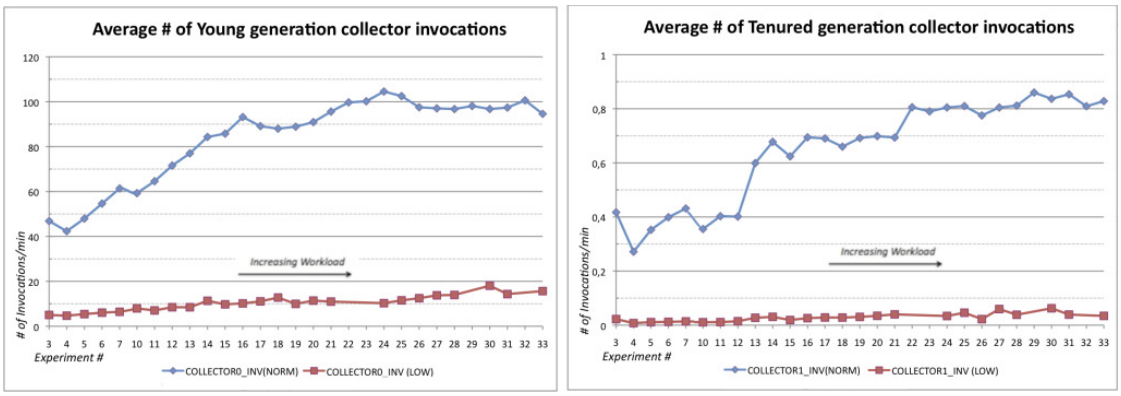


Figure 14. Average Number of Garbage Collector Invocations per Minute in both States, for (A) Young Generation Collector, and for (B) Tenured Generation Collector

VII. Due to the high degree of correlation, only 2 principal components are able to explain 91.71% of the original variance in the *Normal Collection* state, whereas 3 principal components explain 90.79% of the original variance in the *Low Collection* state.

5.3. Throughput Loss analysis

In the following two sections the aging analysis step takes place. They deal with *i)* detection and estimation of aging trends by considering the two adopted indicators, i.e., the throughput loss (presented in this section) and the memory depletion (in the next section), and with *ii)* the analysis of relationships between the workload parameters and the aging trends.
Dots in Figure 15 represent the trend of the loss of bytes per minute observed in each experiment, i.e., each point is the trend along one experiment. In one experiment, samples are collected each minute, hence the loss of bytes observed in one sample refers to that time interval (1 minute), in which the number of mails that are processed depends on the experiment (for instance, in the experiment number 3, it is 330 mails per minute, meaning that the loss of bytes of one sample refers to 330 mails).
Performed experiments report an evident loss of throughput, which is not affected by periods of *Normal* and *Low Collection*. For instance, experiments #10, #20 and #30 experienced a throughput loss trend for the SMTP server of 0.76KB/min, 1.56KB/min, and 1.96KB/min, respectively. Results reported in Figure 15 highlight the presence of a linear relationship between the throughput loss and the application-level load. Table VIII reports results of a linear regression analysis applied to throughput loss trends (shown in Figure 15). The first row of this Table reports the value of the *student's t* statistic; the probability reported in the second row indicates at which confidence level the null hypothesis can be rejected. The third row reports the estimated slope, referred, respectively, to *i)* the throughput loss trend as linear function of the number of mails per minute (specifically, how much the throughput loss increases for an increment of 100 mails per minute), and to *ii)* the throughput loss trend as

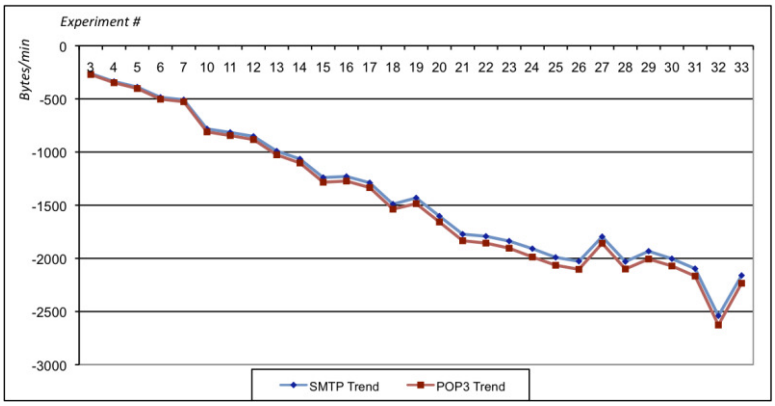


Figure 15. Throughput loss trends for SMTP and POP3 servers among different experiments

function of the *Normal Operation* throughput (i.e., how much the throughput loss trend increases as the *Normal Operation* throughput increases by 1MB per minute; cf. Section 5.1 for a definition of "Normal Operation"). The fourth row reports 95% confidence intervals.
Increasing the workload by 100 mails per minute causes an increment of about 175 bytes per minute in the throughput loss for the SMTP server and of about 185 bytes per minute for the POP3 server. This means that, for instance, if the *Normal Operation* throughput is about 30MB per minute, it is possible to expect the throughput to be halved in about 8 days and 13 hours. In order to obtain useful insights into the relationships between throughput loss and JVM workload parameters, the multiple regression step of the software aging analysis was carried out, presented in section 4.3. Results of this analysis are reported in Table IX, which highlights the significance of principal components. From

Table VIII. Throughput loss trend as a linear function of the number of emails/min and of the *Normal Operation* throughput

	SMTP Server		***POP3 Server***	
	Mail/min	**Normal Operation Throughput**	**Mail/min**	**Normal Operation Throughput**
Student's t	-31,29	-46,35	-31,09	-45,41
Pr $> \|t\|$	$<$0,0001	$<$0,0001	$<$0,0001	$<$0,0001
Estimated Slope	-0,174 KB/100 mail/min	-0,067 KB/MB/min	-0,18 KB/100 mail/min	-0,067 KB/MB/min
95% Confidence Interval	[-0,186 KB/100 mail/min; -0,163 KB/100 mail/min]	[-0,07 KB/MB/min; -0,064 KB/MB/min]	[-0,192 KB/100 mail/min; -0,169 KB/100 mail/min]	[-0,07 KB/MB/min; -0,064 KB/MB/min]

these, only the principal components that have a real influence on aging trends are selected, i.e., the principal components showing a probability of being in the tail of t distribution lower than 5%. They are `EXEC_PC_1` and `NORM_COLL_PC2`. For each of these PCs, original variables mostly contributing to its composition have to be selected for estimating relationship between original workload parameters

Table IX. Results for multiple regression analysis of throughput loss against principal components

	SMTP Server		POP3 Server	
	Student's t	**Pr $>\|t\|$**	**Student's t**	**Pr $>\|t\|$**
CI_PC1	-1,69	0,119	-1,68	0,104
CI_PC2	0,65	0,525	0,67	0,508
CI_PC3	-0,23	0,818	-0,17	0,866
CI_PC4	1,55	0,136	-1,56	0,130
EXEC_PC1	**-6,19**	**0,000**	**-6,25**	**0,000**
EXEC_PC2	-2,01	0,054	-2,03	0,052
NORM_COLL_PC1	-1,61	0,123	-1,62	0,116
NORM_COLL_PC2	**-2,42**	**0,025**	**-2,52**	**0,018**
LOW_COLL_PC1	-1,22	0,233	-1,27	0,233
LOW_COLL_PC2	-0,84	0,408	-0,91	0,371
LOW_COLL_PC3	-0,44	0,663	-0,45	0,656

and aging trend, according to the procedure defined in Section 4.3.
Note that it may happen, as in the case of EXEC_PC1, that all, or almost all, of the original variables contribute approximately equally to the PC. In such a case, it is reasonable to not consider all the variables, for two reasons: *i)* it would be redundant and pretty useless, since the original variables are often strongly correlated to each other (e.g., for SAFEPOINTS_AVG and COLLECTOR0_INV_AVG, the JVM creates a "safepoint" whenever the young collector is invoked; hence, almost the same results would be obtained by analyzing one of the twos); *ii)* it would be expensive, with respect to the benefits obtained (because of redundancy). Rather, in these cases only the most relevant variables could be considered, e.g., the ones whose events occurrence during execution is significantly more frequent than others.
In the case of the two selected PCs, it is possible to notice, recalling their composition, that while the main contribution to the NORM_COLL_PC2 principal component is mainly due to the duration of each young generation collection (described by the workload parameter COLL0_TIME_PER_INV, whose weight is -0.749), all the execution-related workload parameters, except TE_TREND, contributed to the EXEC_PC1 principal component.
Thus, for NORM_COLL_PC2, the parameter COLL0_TIME_PER_INV has been selected. Whereas, for EXEC_PC_1, among the six contributing parameters (namely, MET_INV_AVG and TREND, OBJ_ALL_AVG and TREND, ARR_ALL_AVG and TREND), MET_INV_AVG and OBJ_ALL_AVG have been considered. Indeed, their occurrence frequency is orders of magnitude greater than the one of ARR_ALL_AVG; hence, they are much more relevant. Moreover, ARR_ALL_AVG is strongly correlated with OBJ_ALL_AVG, its analysis would provide similar results.
Then, the impact of the 3 above mentioned workload parameters on throughput loss has been evaluated, obtaining results shown in Table X. This impact has been estimated by linear regression methods, identifying the ideal slope between the throughput loss and each of the selected workload parameters. Values of the *Student's t* confirm that it is possible to reject the *no trend* null hypothesis. However, the COLL0_TIME_PER_INV parameter exhibits a larger confidence interval (95%), thus suggesting that this is less confident than MET_INV AVG and OBJ_ALL AVG parameters. Results of linear regression analysis tells that it is possible to expect an increase of 0.0047KB (about 5 bytes) in throughput loss for an increment of 1 million in method invocation rate, or an increase of 0.055KB (about 55 bytes) for an increment of 100K in object allocation rate. These results are not surprising, since the average method invocation rate of the performed experiments is about 3×10^8 methods per minute and the average

object allocation rate is of about 3×10^6 objects per minute.

Execution-related workload parameters are therefore the most relevant for throughput loss, which

Table X. Throughput loss as a linear function of most relevant JVM workload parameters.

	POP3 Server		
	OBJ_ALL_AVG	**MET_INV_AVG**	**COLLECTOR0_TIME_PER_INV**
Student's t	-35,84	-25,4	-5,17
Pr $> \|t\|$	<0,0001	<0,0001	<0,0001
Estimated Slope	-0.055 KB/100KAll **	-0.0047 KB/Minv *	-3.002 KB/ms
95% Confidence Interval	[-0.058; -0.052]	[-0.005; -0.0043]	[-4.195; -1.81]
	POP3 Server		
	OBJ_ALL_AVG	**MET_INV_AVG**	**COLLECTOR0_TIME_PER_INV**
Student's t	-35,91	-25,46	-5,15
Pr $> \|t\|$	<0,0001	<0,0001	<0,0001
Estimated Slope	-0.057 KB/100KAll **	-0.0048 KB/Minv *	-3.177 KB/ms
95% Confidence Interval	[-0.06; -0.054]	[-0.0052; -0.0044]	[-4.34; -1.866]

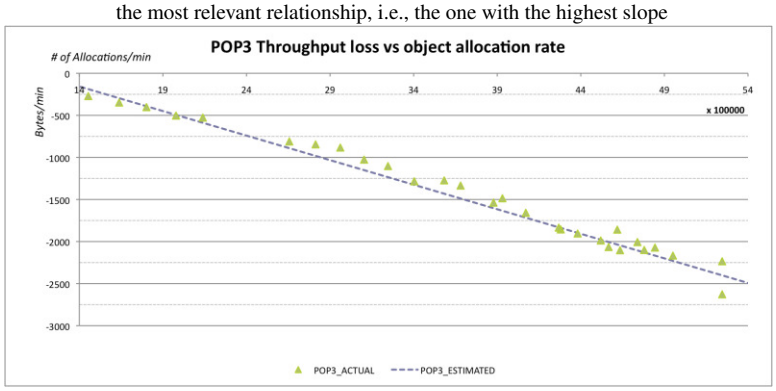
** Millions of method invocations per minute*
*** Number of object allocation per minute * 100.000*

increases linearly with their values. However, although regression analysis highlighted a relationship between throughput loss and the time required for a single collection, it is not possible to claim that the garbage collectors have a real impact on throughput loss, since collection times are higher because there are more objects in the heap area. Based on these results, relationships between JVM workload parameters and throughput loss can be excluded; as a consequence, being the application layer aging-free, the observed throughput loss is attributable to the OS abstraction layer and to its interaction with the operating system. This is also supported by the I/O-bound nature of the workload, since the JVM forwards any I/O operation to the underlying operating system through the OS abstraction layer. However, further investigations are needed to confirm this hypothesis.

5.4. Memory Depletion Analysis

In this Section, results of aging analysis referred to the *memory depletion* aging indicator are reported. Memory depletion has been measured as the amount of physical memory available in a given time, queried by means of the Linux *free* utility, plus the page cache size. The page cache contains a copy of recently accessed files in kernel memory, and can get all the free memory not allocated by the kernel or by user processes; since the page cache is preemptable if needed, it has to be considered as available memory, and added to the actual free physical memory. The value of the available memory is periodically sampled and stored in a trace. Moreover, for memory depletion, unlike for throughput loss, it has been possible to split the contribution at the application layer from the contribution at JVM layer. In particular, the developed monitoring tool, i.e., *JVMMon*, is able to distinguish, by using the *JVM Tool Interface* facilities (events, callbacks, functions; cf. Section 3.3) and *Bytecode Instrumentation*, the amount of memory committed to the application (by intercepting application's objects allocation/deallocation), from the space actually allocated into the heap of the JVM, thus allowing to obtain resource usage information both at the application and at the JVM layer.

Figure 16. Throughput loss as a linear function of most relevant JVM workload parameters. The Figure reports, as an example, the most relevant relationship, i.e., the one with the highest slope



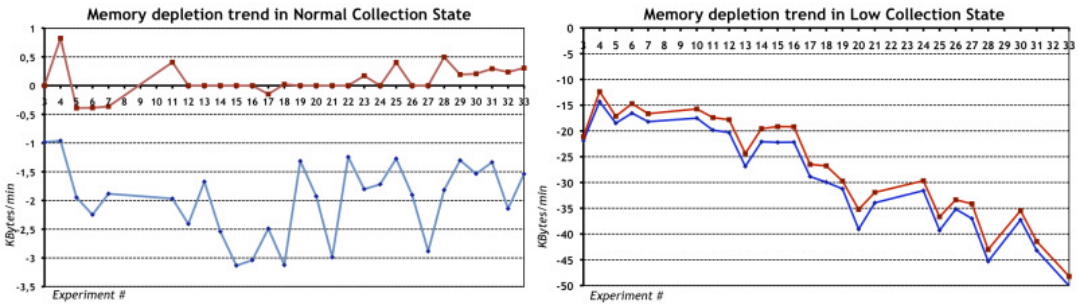


Figure 17. Memory depletion trends during (A) Normal and (B) Low collection periods

For throughput, it is not easy to make this distinction, since throughput is the number of bytes processed by the pair server-JVM when the application sends or receives emails; this contribution (i.e., bytes per minute processed) cannot be easily split between the two.

Memory depletion trends for the application and the JVM are reported in Figure 17. In the *Normal Collection* state, the conducted analysis, which focused on workload-correlated aging, revealed no memory depletion, i.e., aging trend seems to be independent of the load imposed on the mail server; whereas, in the *Low Collection* state, the memory depletion trend generally increases as the workload increases. As regards memory depletion trends in *Normal Collection* state at the application layer, no aging trend was found for the greatest part of experiments, thus proving that the benchmark application layer does not suffer from memory depletion. Moreover, trends measured at the JVM layer are always higher than trends measured at the application layer, thus suggesting a contribution of the JVM to

memory depletion. The estimated Time-To-Exhaustion (TTE) in the *Low Collection* state ranges from 22 days and 4 hours (-13.96 KB/min) to 6 days and 11 hours (-48.89 KB/min), whereas JVM-level memory depletion trends exhibited in the *Normal Collection* state are less concerning, given that the corresponding TTE is equal to about 105 days in the worst case (-3.06 KB/min). It is therefore possible to claim that, although memory depletion has to be considered in both the workload states, it becomes a serious threat during low collection periods. During the experiments, the average duration of visits in the *Low Collection* state was 590 minutes, with a peak of 2234 minutes.

Table XI. Regression analysis of memory depletion at application and JVM layer *vs.* principal components

	NORMAL COLLECTION STATE						**LOW COLLECTION STATE**					
	JVM		***APP***		***DIFF***		***JVM***		***APP***		***DIFF***	
	t	Pr $> \|t\|$	t	Pr $> \|t\|$	t	Pr $> \|t\|$	t	Pr $> \|t\|$	t	Pr $> \|t\|$	t	Pr $> \|t\|$
CI_PC1	**-2,48**	**0,02**	0,44	0,66	**2,96**	**0,01**	-0,27	0,79	-0,24	0,81	0,17	0,87
CI_PC2	-1,25	0,23	-0,67	0,51	0,98	0,34	-0,03	0,98	-0,02	0,98	0,05	0,96
CI_PC3	**-4,21**	**0,00**	-0,51	0,62	**4,00**	**0,00**	0,27	0,79	0,56	0,58	**2,19**	**0,05**
CI_PC4	-1,49	0,15	0,43	0,67	1,88	0,08	1,66	0,12	1,58	0,14	0,94	0,36
EXEC_PC1	1,88	0,08	-0,51	0,62	-1,35	0,19	-1,30	0,22	-1,27	0,23	0,14	0,89
EXEC_PC2	0,85	0,41	-0,13	0,90	-1,00	0,33	-0,61	0,55	-0,65	0,55	0,31	0,76
NORM_COLL_PC1	-0,82	0,42	0,79	0,44	1,35	0,19	-0,19	0,85	-0,16	0,87	0,22	0,83
NORM_COLL_PC2	1,07	0,30	-1,61	0,12	0,08	0,94	0,30	0,77	0,26	0,80	-0,27	0,79
LOW_COLL_PC1	-0,39	0,70	-0,41	0,69	-0,04	0,97	**-5,20**	**0,00**	**-4,81**	**0,00**	-0,53	0,61
LOW_COLL_PC2	0,07	0,94	0,03	0,98	0,01	0,99	**3,11**	**0,01**	**2,77**	**0,02**	1,71	0,11
LOW_COLL_PC3	-0,16	0,87	-0,04	0,97	-0,02	0,98	-1,69	0,10	-1,43	0,18	0,04	0,97

Table XI reports results of the regression analysis, for both *Normal* and *Low collection* state; the first column reports partial regression results for trends at the JVM layer, the second one refers to the application layer, and the third one reports the difference between the two previous trends. These results reveal the following insights:

i) No workload parameter gives information for investigating memory depletion at the application layer in the *Normal Collection* state, since the value of the t statistic is always under its critical value; indeed, in this state memory depletion at the application layer has been noticed.

ii) Memory depletion trends at the JVM layer in the normal workload state are due to the `CI_PC1` and `CI_PC3` principal components; therefore JIT compiler can be addressed as the main source of software aging in this workload state. Indeed, each time a Java method is JIT-compiled, generated native code is stored in a reserved area in the Java Heap, the *Native Method Cache*. The size of this area progressively increases during experiment duration.

iii) In the *Low Collection* state, memory depletion trends, both at the application and JVM layers, are due to the activity of the garbage collector. The most relevant principal components are `LOW_COLL_PC1` and `LOW_COLL_PC2`, which reported significant scores for the t statistic. This confirms that, in the *Low Collection* state, memory depletion is mainly due to the downfall of garbage collector activity.

iv) The difference between memory depletion trends at the application and JVM layer can be explained taking into account JIT Compiler activity in both collection states. The `CI_PC3` principal component has a relevant impact on the trend difference between application and JVM layers. Recalling Table V, the time spent per minute in standard JIT compilation (`CI_STD_TIME AVG`) gives the most significant contribution to this principal component.

Given the results of partial regression analysis, 4 JIT-related parameters are chosen to estimate

workload-aging relationship in the *Normal Collection* state, and 3 GC-related parameters in the *Low Collection* state. Results of linear regression analysis applied to these parameters are reported in Table XII. Among the 4 selected JIT-compiler parameters, only the time spent in standard

Table XII. Memory depletion in Java Heap as a linear function of most relevant JVM workload parameters

	NORMAL COLLECTION STATE-JVM FREE			
	CI_STD_TIME_AVG	**CI_STD_COMPILES_AVG**	**CI_OSR_TIME_AVG**	**CI_TIMEPERCOMP_AVG**
Student's t	-3,11	-1,3	-0,86	-1,99
Pr $> \|t\|$	0,0045	0,2063	0,3977	0,0572
Estimated Slope	-0.097 KB/ms	-2.384 KB/comp *	-0.02 KB/ms	-0.0014 KB/ms
95% Confidence Interval	[-0.167; -0.027]	[-6.167; 1.39]	[-0.067; 0.027]	[-0.002; -0.0007]

	LOW COLLECTION STATE-JVM FREE		
	COLLECTOR0_INV_AVG	**COLLECTOR1_TIME_AVG**	**COLLECTOR1_TIME_TREND**
Student's t	-7,66	-5,95	-0,71
Pr $> \|t\|$	<0,0001	<0,0001	0,4876
Estimated Slope	-2.41 KB/inv **	-3.74 KB/ms	-0.347 KB/(ms/min)
95% Conf. Int.	[-3.05; -1.76]	[- 5.16; -2.44]	[-1.36; 0.67]

	LOW COLLECTION STATE-APP FREE		
	COLLECTOR0_INV_AVG	**COLLECTOR1_TIME_AVG**	**COLLECTOR1_TIME_TREND**
Student's t	7,29	-5,85	-0,54
Pr $> \|t\|$	<0,0001	<0,0001	0,5926
Estimated Slope	-2.35 KB/inv **	-3.68 KB/ms	-0.265 KB/(ms/min)
95% Conf. Int.	[-3.01; -1.68]	[- 4.99; -2.38]	[-1.27; 0.747]

compilation (`CI_STD_TIME AVG`) shows a strong linear relationship with memory depletion in the *Normal Collection* state. The null hypothesis (no linear relationship) cannot be rejected for the remaining parameters, which altogether account for 33.86% of the information contained in the `CI_PC1` principal component. Among the 3 selected GC parameters, the average number of young collector invocations (`COLLECTOR0_INV AVG`) and the average time spent during tenured generation collection (`COLLECTOR1_TIME AVG`) are linearly correlated with memory depletion both at the JVM and at the Application layer, whereas the null hypothesis cannot be rejected for the trend exhibited by the tenured generation collector. It is therefore possible to claim that: **i)** there is no noticeable memory depletion trend at the application layer in *Normal Collection* state, and the trend observed at the JVM layer is due to the growth of the native method cache size; **ii)** memory depletion is much higher in the *Low Collection* state, and it is due to the downfall of garbage collector invocations (and phases of normal collection do not even out the loss), and **iii)** the differences between the trends observed at the application and JVM layer are due to the activity of JIT compiler.

5.5. Key Findings

The conducted campaign revealed that the JVM is affected by software aging, which manifested both as throughput loss and memory depletion. In particular, the analysis of collected data proved that:
i) Regarding memory depletion, in both normal and low collection states, there is a slow drift whose source is located in the JIT compiler. This drift is mainly due to data stored in the *Native Code Cache*. However, these depletion dynamics cannot be regarded as a serious threat for the JVM, since the

estimated TTE is considerably high.
ii) Sudden downfalls in Garbage Collector activity shift the JVM from the normal to the low collection state, causing free memory to decrease at a high rate.
iii) The interface between the JVM and the operating system seems to be critical from a throughput loss perspective. Results showed that, under stressing workload, throughput is cut down by half after about 1 week of execution. In different scenarios, with higher and more stressful workloads, TTE could be consistently lower, thus becoming a serious problem. However, further investigations are required to assess if these aging phenomena are really located in the interface between the JVM and the OS.
Summarizing, the experimental campaign highlighted the presence of three distinct aging dynamics.

Table XIII. Time to exhaustion estimation for detected aging phenomena

	Related Workload Parameters			*TTE*	
	Parameter	**Slope**	**U.M.**	**Best Case**	**Worst Case**
Throughput Loss	OBJ_ALL_AVG	-0,055	KB/10^5 Alloc.	36 days	5 days
	MET_INV_AVG	-0,0047	KB/10^6 Inv.	10 hours	23 hours
"Slow" Memory Depletion Drift	CI_STD_TIME_AVG	-0,097	KB/ms	342 days	104 days
	CI_OSR_TIME_AVG	-0,02	KB/ms	11 hours	22 hours
"Fast" Memory Depletion Drift	COLLECTOR0_INV_AVG	-2,41	KB/inv	48 days	13 days
	COLLECTOR1_TIME_AVG	-3,74	KB/ms	21 hours	23 hours

Table XIII reports, for each of these dynamics, the related workload parameters, the estimate of the relationships between the workload parameter and the aging trend, and the estimated TTE. As regards throughput loss, TTE has been calculated assuming the system failed when the throughput is halved. In particular, Table XIII reports TTEs estimated both in the worst and in the best case. The most relevant aging dynamic is the one related to throughput loss. This dynamic gets worse when the activity of the execution unit increases. For instance, if the method invocation rate increases by 10 millions per minute (keep in mind that in the conducted experiments an average method invocation rate ranging from 170 to 650 millions of method per minute was observed), it is possible to expect an increase in throughput loss of 0.5 KBytes per minute.
On the other hand, two distinct dynamics are responsible for memory depletion in the JVM. The *fast* drift is associated with a very low TTE (about 14 days in the worst case). However, since low collection periods do not usually last for a long period of time, this aging dynamic does not usually cause a failure of the JVM. Moreover, since low collection periods may be detected by monitoring Garbage Collection workload parameters, it is possible to bring the JVM out of this workload state by forcing garbage collections. This task may be accomplished through monitoring and management tools, such as `JConsole` (JConsole is a GUI tool compliant with the Java Management Extensions). The *slow* drift, instead, exhibits very high TTEs (about 105 days in the worst case, and about 1 year int the best case). However, unlike the fast drift, this dynamic is present in both collection states. Therefore, even if TTEs estimated during experiments do not represent a significant threat to JVM dependability, this dynamic may become a serious source of JVM failure whenever the JIT compilation activity significantly increases. Results discussed in this paper may be useful for both JVM designers and final users; in particular they may allow: i) designers to identify internal JVM components mainly causing aging dynamics (e.g., the garbage collector, and the JIT-compiler); this drives the investigation of aging sources and corrections inside such components; ii) users to plan effective rejuvenation strategies based

on the TTE predictions (i.e., rejuvenate the system just before the TTE expires, saving additional rejuvenations in the time interval [0, TTE]), or to prolong the TTE by applying proactive actions (for instance, bringing the JVM out of the low collection state by forcing the garbage collections). Moreover, the methodology can be applied to other VMs and layered systems, in order to isolate aging contribution of each layer, and to monitor their activities at runtime, or to correct them against aging-related bugs. The empirical analysis presented in this study can be extended to address the following limitations and threats to its validity:

i) this study analyzed the relationship between software aging and workload parameters in the JVM layer, while varying one application-level workload parameter. Although the interest is on "uncontrollable" workload parameters of a layer (in this case the JVM), considering more than on application-level parameters could improve results by highlighting further aging sources not currently identified.

ii) The conducted analysis has identified sources of aging in the JVM with a given application-level workload type; however it is not exhaustive. Further experiments with different workload could enhance the achieved findings, by identifying additional sources of aging that have not been stressed by the current type of workload.

iii) Experiments have been conducted on one JVM implementation; different JVM implementations can yield different results. Thus, the methodology defined in this paper could be applied also to other JVM implementations, in order to draw more general conclusions about the aging of the JVM.

iv) As in the previous point, experiments could be extended to different JVM / Operating System pairs, since the behavior may differ also in this case.

For these issues, further experiments are required, which, applying the same methodology, could lead to the identification of additional aging sources, other than those identified by the conducted analysis.

6. Conclusions

In this paper, a methodology aimed at evaluating the software aging phenomenon in Java-based systems has been presented. It allows discovering aging phenomena, evaluating their relationship with the workload, and locating those components that suffer from aging. The methodology has been applied to a Java software system composed by an Apache JAMES mail server at the application layer, the Sun Hotspot JVM, at intermediate layer, and a Linux OS at lowest layer. An experimental campaign consisting of a series of 29 experiments with synthetic workload, accounting for about 3000 hours of execution, has been performed; during this campaign, data about resource usage and workload were collected by means of an ad-hoc monitoring tool, named *JVMMon*. The analysis of data revealed the presence of several distinct aging dynamics, which manifested as throughput loss; and as memory depletion, mainly attributable to activities performed by the JIT-compiler. With regard to throughput loss, a consistent aging trend, ranging from 0.08 KBytes per minute to 2.48 KBytes per minute has been found, mainly dependent on execution unit activity: indeed, it has been found that such trend is strongly correlated with workload parameters, such as method invocation frequency and object allocation frequency. On the other hand, memory depletion phenomena involved two distinct aging dynamics. The first one manifested itself as a slow, but constant, drift and is mainly due to the activity of the JIT-compiler. The second one was due to sudden down-falls in garbage collector activity and manifested itself as a fast drift. These aging trends may be treated as follows: regarding the slow drift,

the JVM may be properly configured in order to avoid excessive JIT compilation (in particular, it has been found that On-Stack-Replacement compilations are the most expensive ones in terms of required memory); regarding the fast drift, radical changes in Garbage Collector activity may be easily detected by monitoring invocations of the Young Generator Collector. Whenever a downfall in garbage collector frequency is detected, it is possible to force Garbage Collector invocations in order to limit memory depletion. The obtained results were quite unexpected, since the JVM has been designed to reduce effects of aging phenomena. Suffice it to recall that the garbage collector, which is designed to free developers from manually handling memory management, is the most important source of aging due to memory leaking or bloating.

Although these specific results are valid only for the Sun Hotspot JVM implementation (and different implementations of the JVM may exhibit different aging dynamics), the methodology adopted in this paper is general and may be applied to different JVM implementations, and to different Java-based software systems. Indeed, different Java applications can stress the underlying JVM components in different ways, thus causing aging to evolve in various ways. By applying the outlined methodology, it is possible to isolate the JVM components' contribution to software aging when stressed by some applicative workload parameters, thus allowing an in-depth analysis of aging dynamics.

REFERENCES

1. Grottke M, Matias R, Trivedi KS. The Fundamentals of Software Aging. *Proceedings of the 1st International Workshop on Software Aging and Rejuvenation/19th IEEE International Symposium on Software Reliability Engineering*, November 2008, 1–6. DOI: 10.1109/ISSREW.2008.5355512.
2. Marshall E. Fatal Error: How Patriot Overlooked a Scud. *Science* 1992, **255** (5050):1347. DOI: 10.1126/science.255.5050.1347
3. Hartman F, Maxwell S. Driving the Mars Rover. *Linux Journal* 2004; **125**: 68–70.
4. Java Community Process (JCP). *JSR-302: Safety Critical Java Technology*, 2006.
5. Georges A, Buytaert D, Eeckout L, De Bosschere K. How Java Programs Interact with Virtual Machines at the Microarchitectural Level. *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2003; 169–186. DOI: 10.1145/949305.949321.
6. Georges A, Buytaer D, Eeckhout L, De Bosschere K. Method-Level Phase Behavior in Java Workloads. *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, 2004; 270–287. DOI: 10.1145/1028976.1028999.
7. Sweeney PF, Hauswirth M, Cahoon B, Cheng P, Diwan A, Grove D, Hind M. Using Hardware Performance Monitors to Understand the Behavior of Java Applications. *Proceedings of the third Usenix Virtual Machine Research and Technology Symposium*, 2004; 5–5.
8. Napper J, Alvisi L, Vin H. A Fault-Tolerant Java Virtual Machine, *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, June 2003; 425–434. DOI: 10.1.1.13.7072.
9. Friedman R, Kama A. Transparent fault-tolerant java virtual machine, *Proceedings of the 22st Symposium on Reliable Distributed Systems*, October 2003; 319–328. DOI: 10.1109/RELDIS.2003.1238083.
10. Silva L, Madeira H, Silva JG, Software aging and rejuvenation in a soap-based server. *Proceedings of the 5th International Symposium on Network Computing and Applications*, July 2006; 56–65. DOI: 10.1109/NCA.2006.51.
11. Vaidyanathan K, Trivedi KS. A measurement-based model for estimation of resource exhaustion in operational software systems. *Proceedings of the 10th International Symposium on Software Reliability Engineering*, November 1999; 84–93. DOI: 10.1109/ISSRE.1999.809299.
12. Cassidy KJ, Gross KC, Malekpour A. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002; 478–482. DOI: 10.1109/DSN.2002.1028933
13. Garg S, Van Moorsel A, Vaidyanathan K, Trivedi KS. A methodology for detection and estimation of software aging. *Proceedings of the 9th International Symposium on Software Reliability Engineering*, November 1998; 283. DOI: 10.1109/ISSRE.1998.730892

14. Bao Y, Sun X, and Trivedi KS. A workload-based analysis of software aging, and rejuvenation. *IEEE Transactions on Reliability* 2005; **54** (3): 541–548. DOI: 10.1109/TR.2005.853442
15. Garg S, Puliafito A, Telek M, Trivedi KS. Analysis of Preventive Maintenance in Transactions Based Software Systems. *IEEE Transactions on Computers* 1998; **47** (1); 96–107. DOI: 10.1109/12.656092.
16. Matias R, Filho PJF. An experimental study on software aging and rejuvenation in web servers. *30th Annual International Computer Software and Applications Conference*, September 2006; 189–196. DOI: 10.1109/COMPSAC.2006.25.
17. Hoffmann GA, Trivedi KS, Malek M. A best practice guide to resources forecasting for the apache webserver. *IEEE Transactions on Reliability* 2007, **56** (4): 615–628. DOI: 10.1109/TR.2007.909764.
18. Grottke M, Trivedi KS. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *IEEE Computer* 2007; **40** (2): 107-109. DOI: 10.1109/MC.2007.55.
19. Balakrishnan M, Puliafito A, Trivedi KS, Viniotis Y. Buffer losses vs. deadline violations for ABR traffic in an ATM switch: A computational approach. *Telecommunication Systems* 1997; **7** (1-3): 105–123. DOI: 10.1.1.44.1321
20. Grottke M, Li L, Vaidyanathan K, Trivedi KS. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 2006; **55** (3): 411–420. DOI: 10.1109/TR.2006.879609.
21. Cotroneo D, Natella R, Pietrantuono R, Russo S. Software Aging Analysis of the Linux Operating System. *Proceedings of the 21st International Symposium on Software Reliability Engineering*, November 2010; 71–80. DOI: 10.1109/ISSRE.2010.24
22. Huang Y, Kintala CMR, Kolettis N, Fulton ND. Software rejuvenation: Analysis, module and applications. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995; 381–390. DOI: 10.1109/FTCS.1995.466961.
23. Garg S, Huang Y, Kintala C, Trivedi KS. Minimizing completion time of a program by checkpointing and rejuvenation. *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1996; 252–261. DOI: 10.1145/233013.233050
24. Garg S, Puliafito A, Telek A, and Trivedi KS. Analysis of software rejuvenation using markov regenerative stochastic petri nets. *Proceedings of the 6th International Symposium on Software Reliability Engineering*, October 1995; 24–27. DOI: 10.1109/ISSRE.1995.497656.
25. Pfening A, Garg S, Puliafito A, Telek M, Trivedi KS. Optimal Software Rejuvenation for Tolerating Soft Failures. *Performance Evauation* 1996; **27-28** (4): 491–506. DOI: 10.1016/S0166-5316(96)90042-5
26. Andrzejak A, Silva L. Deterministic Models of Software Aging and Optimal Rejuvenation Schedules. *Proceeding of the10th IFIP/IEEE International Symposium Integrated Network Management*, May 2007. DOI: 10.1.1.150.3409
27. Wang D, Xie W, Trivedi KS. Performability analysis of clustered systems with rejuvenation under varying workload. *Performance Evaluation* 2007; **64** (3): 247–265. DOI: 10.1016/j.peva.2006.04.002.
28. Salfner A, Wolter K. Analysis of Service Availability for Time-triggered Rejuvenation Policies. *The Journal of Systems & Software* 2010; **9**(83): 1579–1590. DOI: 10.1016/j.jss.2010.05.022.
29. Vaidyanathan K, Trivedi KS. A comprehensive model for Software Rejuvenation. *IEEE Transactions on Dependable and Secure Computing* 2005; **2** (2): 124–137. DOI: 10.1109/TDSC.2005.15.
30. Hong Y, Chen D, Li L, Trivedi KS. Closed loop design for Software Rejuvenation. *Proceedings of SHAMAN Workskop "Security for mobile systems beyond 3G"*, June 2002.
31. Li L, Vaidyanathan K, Trivedi KS. An approach for estimation of software aging in a web server. *Proceedings of the International Symposium on Empirical Software Engineering* October 2002; 91–102. DOI: 10.1109/ISESE.2002.1166929.
32. Lindholm T, Yellin F, *The Java (TM) Virtual Machine Specification, 2nd ed*, Sun Microsystems Press, 1999.
33. Gosling J, Joy B, Steele G, Bracha G. *The java language specification, 3rd ed.*. Sun Microsystems Press, 2005.
34. Cotroneo D, Orlando S, Russo S, Failure classification and analysis of the java virtual machine. *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, July 2006; 17. DOI: 10.1109/ICDCS.2006.37
35. Cotroneo D, Orlando S, Russo S, Characterizing Aging Phenomena of the Java Virtual Machine, *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, October 2007; 127–136. DOI: 10.1109/SRDS.2007.22
36. Java Community Process (JCP). *JSR-163: Java Platform Profiling Architecture (JPPA)*, 2004.
37. Jolliffe IT. *Principal Component Analysis, second ed.*. Springer-Verlag: New York, 2002.
38. Trivedi KS. *Probability and Statistics with Reliability, Queuing and Computer Science Applications, second ed.*. John Wiley and Sons Inc., 2002.
39. Montgomery DC. *Design and Analysis of Experiments, fifth ed.*. John Wiley and Sons Inc., 2001.
40. Xu R, Wunsch DC. *Clustering*. John Wiley and Sons Inc., 2009.
41. Ross Sheldom M. *Introduction to Probability and Statistics for Engineers and Scientists, Third Edition*. Elsevier Academic Press, 2004.