A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

SOFTWARE AGING ANALYSIS OF OFF THE SHELF SOFTWARE ITEMS

SALVATORE ORLANDO

Tesi di Dottorato di Ricerca

Novembre 2007

Dipartimento di Informatica e Sistemistica

via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

SOFTWARE AGING ANALYSIS OF OFF THE SHELF SOFTWARE ITEMS

SALVATORE ORLANDO

Tesi di Dottorato di Ricerca

(XX Ciclo)

Novembre 2007

Il Tutore
Prof. Stefano Russo

Il Coordinatore del Dottorato
Prof. Luigi P. Cordella

Dipartimento di Informatica e Sistemistica

via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

SOFTWARE AGING ANALYSIS OF OFF THE SHELF SOFTWARE ITEMS

By

Salvatore Orlando

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
"FEDERICO II" UNIVERSITY OF NAPLES
VIA CLAUDIO 21, 80125 – NAPLES, ITALY
NOVEMBER 2007

“ ’A vecchie ’e cient anne ricette:

M’aggia ’mpara’ ancora! ”

(Neapolitan Proverb)

A 100-year old woman said:

I still have to learn!

Table of Contents

List of Tables

List of Figures

Acknowledgements

Ed eccomi qui, dopo tre anni, a scrivere di nuovo una pagina di ringraziamenti...
Un capitolo della mia vita si sta per chiudere, un altro si sta per aprire. Come si suol dire, *per una porta che si chiude si apre un portone*.
E prima di chiudere 'sta porta allora mettiamoci a tirare un po' le somme: dunque dopo tre anni sono più grasso, più calvo e più sedentario. Il quadro non è decisamente positivo! Meglio passare allora ad altri aspetti...
In questi tre anni ho avuto modo di constatare una notevole maturazione sia dal punto di vista culturale che lavorativo, il cui principale effetto è una socratica consapevolezza dell'essere totalmente ignorante! Per tale maturazione (vera o presunta che sia), devo ringraziare principalmente il mio tutor, Prof. Stefano Russo, ed il mio tutor-in-second, Prof. Domenico Cotroneo, ma anche i miei colleghi "nonni" Armando, Marcello e Generoso, e le "spine" Gabriella, Roberto, Christian e Lelio.
Al di là delle porte del "MobiLab" ci sono d'altro canto tante persone cui è doveroso dedicare in queste pagine il mio pensiero: i miei genitori, Titina ed Emanuele, con cui so di avere un grosso debito (non nel senso economico della parola!), che spero di poter ripagare nei prossimi anni; i miei amici, e principalmente Peppe Cocorito e Marco 'o Ciabibbo, compagni di tante avventure e sventure, e da sempre miei primi tifosi; e *last but not least*, Titty, la mia fidanzata, che ormai da 4 anni e 4 giorni sopporta questo povero ingegnere polemico e brontolone...
A tutti voi, ed anche a quelli che ho dimenticato di menzionare (non me ne vogliano,sono in preda ad una precoce demenza senile!), va il mio più sentito ringraziamento!

Vorrei concludere queste pagine con una serie di "auguri" per tutti i membri del glorioso gruppo *MobiLab*:

- Che il Prof. Russo impari al più presto come si esegue correttamente uno stop al calciotto;

- Che i pensieri che annebbiano la testa di Domenico svaniscano al più presto, al fine di fargli recuperare le piene facoltà mentali;
- Che il barone Generoso resti sempre uno dei 4 giovani ricercatori più bravi d'Italia;
- Che il Prof. Cinque finalmente riesca ad arrivare in laboratorio prima delle 11 (così magari riesce anche ad andar via prima delle 20);
- Che Gabriella possa coronare tutti i suoi sogni, incluso quello di raggiungere il peso di 35 Kg;
- Che schiavo Lelio vinca il nobel per il pezzotto;
- Che schiavo Roberto finalmente acquisisca il dono della parola;
- Che schiavo Christian capisca che stavamo scherzando quando gli dicevamo che doveva lavorare 25 ore al giorno;

Salvatore

30-11-2007

Introduction

Off-The-Shelf (OTS) items are technology or computer products, ready-made and available for sale or license to the general public. Software OTS items (such as libraries, virtual machines or application servers) are nowadays starting to be widely employed also in business, mission and safety critical scenarios: industries are looking at them as an attractive way to reduce software development costs and time-to-market. As a real-world example of this trend, it is possible to consider the roadmap outlined by EuroControl for the European's Air Traffic Management (ATM) and Air Traffic Control (ATC) evolution. Unfortunately, whilst notably reducing development efforts, the employment of OTS item poses several dependability-related issues. Indeed:

1. OTS items often lack a proper evaluation of their dependability attributes;
2. Interactions between different items may have unpredictable effects which are not easy to foresee.

Therefore, the impact of software defects on system dependability becomes even more critical when OTS items are employed in system design. The use of such items introduces additional problems. They may come with known development faults and may contain unknown faults as well. Moreover, their specification may be incomplete or event incorrect.

Beyond "traditional" errors like exceptions and wrong computations, some development faults affecting software can cause Software aging [2]. Software aging is a phenomenon in which progressively accrued error conditions lead to either performance degradation or transient failures or both. Examples are memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage space fragmentation, and accumulation of round-off errors.

Several recent studies showed that a large number of software systems, employed also in business-critical or safety-critical scenarios [3, 4, 5, 6, 7], are affected by Software Aging. The Patriot missile defense system employed during the First Gulf War, responsible for the Scud incident in Dhahran [8], is perhaps the most representative example of critical system affected by software aging.

Although several approaches have already been proposed to study the development of software aging phenomena [9, 10, 11, 12], there are still some open issues, especially in the field of OTS-based software systems. All measurement-based software aging analyses consider aging introduced by long-running applications, such as web servers and DBMS servers, as measured at the operating system level, neglecting the contribution of intermediate layers, such as middleware, virtual machines, and, more in general, third-party OTS items. These layers might worse resource exhaustion dynamics or become an additional source of aging. In order to develop a methodology to analyze Software Aging in OTS-based systems two challenging issues arise:

1. ***New methods are needed to isolate the contribution of each intermediate layer to aging trends****, i.e., the one introduced by the presence of a virtual machine from the one due to the application running on top of it.*

2. ***Since there is a strict relationship between workloads and aging trends, it is crucial to investigate how these are affected by changes in the applied workload****. Although several work addressed the relationships between workload and aging trends, the selection of workload parameters and the assessment of their effect on aging trends have been partially addressed only recently in [12] and [13].*

In this dissertation we focus on Software Aging phenomena in OTS-based system exploring the possibility of assessing a measurement-based methodology capable of characterize thoroughly the dependability of OTS items from a Software Aging perspective.

By exploiting statistical techniques such as cluster analysis, principal component analysis and multiple regression, the proposed methodology addresses the above mentioned issues, thus allowing to pinpoint software layers in which aging phenomena are introduced, identify which workload parameters are more relevant to the development of aging trends, and evaluate the relationships between workload parameters and aging trends.

This methodology has been adopted to characterize the development of aging phenomena inside the Java Virtual Machine, which is a relevant example of the items which may be employed in OTS-based system. Indeed: **i)** it provides a complete virtualization of the underlying execution environment, **ii)** it is currently widely employed in a wide range of applications, including critical ones, and **iii)** there is a lack of research about the characterization of the dependability of the JVM.

In this thesis we discuss the results of two experimental campaigns aimed at characterize the dependability of the JVM from a software aging perspective. The former,

extends results previously published in [14], addresses aging phenomena which develop inside the JVM; on the other hand the latter takes into account aging phenomena which develop in the interface between the JVM and the underlying OS; results obtained from both the Windows and the Linux OS are compared.

The dissertation is organized as follows:

Chapter 1 provides the needed background on software faults and dependability evaluation, focusing particularly on techniques and methodologies aimed at estimating software aging phenomena.

Previous relevant work dealing with the assessment of the dependability of OTS items are discussed in chapter 2. In particular, as far as software aging analysis is concerned, this chapter deeply discusses open research issues about software aging analysis in OTS-based system and summarizes the contributions of this dissertation.

Chapter 3 presents a preliminary characterization of JVM failure behavior performed analyzing failure reports extracted from publicly available Bug Databases. Results discussed in this chapter extend the ones previously published in [15].

The proposed approach to evaluate Software Aging in OTS-based system is presented in Chapter 4. Moreover, in this chapter we discuss the result of a massive experimental campaign performed on the Java Virtual Machine. Field data have been collected using an ad-hoc developed monitoring tool, described in section 4.5.1,

and more extensively in [16].

Chapter 5 describes the results of an experimental campaign aimed at estimating Software Aging phenomena at the Operating Systems. In this chapter we compare experimental results for both the Windows and the Linux Operating System.

The dissertation concludes with final remarks and the indication of the lesson learned. In particular, as regards the JVM, we provide several hints to augment its resilience to software aging phenomena.

"Chi se mette pe' mare
adda sape' primma nata' "
Who is going to sail
has to know how to swim first

Neapolitan Proverb

Chapter 1

Dependability Assessment Techniques

People have become aware, often by bitter experience, that not only they must know how much service a computer system can deliver, but also how often it actually delivers that intended level of service. Similar to other products, a computer system becomes far less attractive if it frequently deviates from its nominal performance or becomes totally unavailable.
Therefore there is a definite need to assess how long the system is capable to offer the desired level of service, but also to assess how long the system, after an outage, takes to recover back to its level of service. The above two concepts fit together into a measure, the dependability, which describes the effect of outages on system efficiency.
This chapter first describes the concept of dependability toghether with its attributes, then discusses development software faults, also know as software defects or simply "bugs". In particular they are classified according to their semantic and reproducibility.
In the latter part of the chapter, focus is on software dependability evaluation. Several techniques for analyzing the dependability of software systems are discussed and compared according to parameters such as cost of analysis, and level of detail of collected data.

1.1 Basic Notions of Dependability

The first attempt to give a formal definition to the notion of dependability may be traced back to 1960, when it was defined by as *"the probability that a system will*

operate when needed" [17]. The notion of reliance is totally absent in this definition: in order to offer a correct service, a system has only to operate. A substantial effort toward the definition of the basic concepts and terminology for computer systems dependability dates back to 1979, when, during the IFIP [1] Working Conference on Reliable Computing and Fault Tolerance, the concept of a technical committee on Dependable Computing and Fault Tolerance was first formulated.

This technical committee, namely the IFIP 10.4 Working Group, was established in 1980 and held its first meeting on June 22,1981. A principal theme since the first meeting has been the understanding and exposition of the fundamental concepts of dependable computing. A synthesis of this work was presented at the 15th symposium on fault-tolerant computing (FTCS) in 1985 [18], where computer system dependability was defined as *"the quality of the delivered service such that reliance can justifiably be placed on this service"*, thus introducing the concept of reliance in the definition of dependability. This notion, together with its attributes and their definition has changed over time. Continued intensive discussions led in 1992 to a book [19], in which *security* was added as an attribute of dependability. More recently, in a paper appeared on the IEEE transactions on dependable and secure computing (TDSC) in 2004 [1], the notion of dependability was formulated as follows: *"the ability to avoid service failures that are more frequent and more sever than acceptable"*, thus putting focus on the definition of service failure rather than on the justification

[1] International Federation for Information Processing

of trust. Moreover, security has no more been characterized as a single attribute of dependability.

Although in this thesis we will adhere to the notion of dependability given in [1], several different definitions were given for the notion of dependability. ISO[2], in 1992, gave the following availability-oriented definition: "*The collective term used to describe the availability performance and its influencing factors: reliability performance and maintenance support performance*". This definition recalls the one given in 1984 by the international organization for telephony, the CCITT[3], since availability is the main concern for telephone systems. Another definition, closer to the one adopted in this thesis has been given by the International Electrotechnical committee (IEC): "*The extent to which the system can be relied upon to perform exclusively and correctly the system task(s) under operational and environmental conditions over a defined period of time, or at a given instant of time*". The latter definition, like the one given in 1985, puts emphasis on the concept of reliance.

The dependability concept is used in all stages of the life cycle of a computer system, from the requirement stage, where it provides a customer orientation in developing systems requirements, to the operational stage, in which dependability aids in selecting effective measures for operator response to failure-related incidents. These

[2]ISO: International Standards Organization

[3]CCITT: *Comité consultatif international téléphonique et télégraphique* (International Telephone and Telegraph Consultative Committee)

measures allow to quantitatvely evaluate several aspects of system dependability; indeed rather then being a monolithic concept, dependability may be regarded as an integrating concept that includes the following attributes:

- **Availability**: readiness for correct service;
- **Reliability**: continuity of correct service;
- **Safety**: absence of catastrophic consequences on the user(s) and the environment;
- **Integrity**: absence of improper system alterations;
- **Maintainability**: ability to undergo modifications and repairs.

1.1.1 Measures

Several measures have been defined in order to assess the dependability level of a system. Some of these measures are enough general to be applied to every system, and are based on few parameters which are widely used in order to characterize a system from a dependability perspective. These parameters are:

- **Mean Time To Failure** `MTTF`: Mean interval of time between a system recovery (or system start) and the occurrence of a failure.
- **Mean Time To Repair** `MTTR`: Mean time required to perform a repair. It

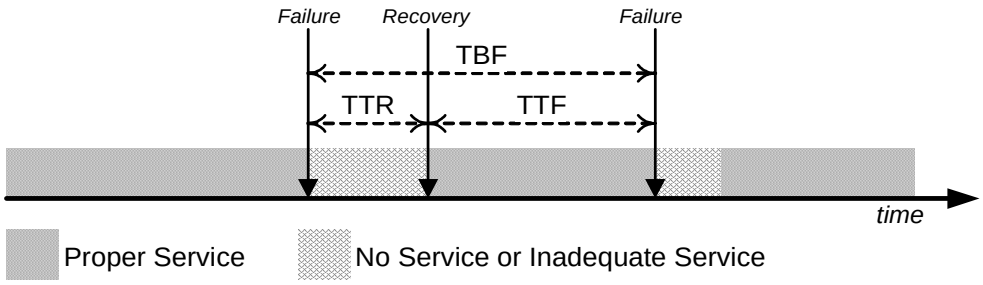

Figure 1.1: Relationships between Time-To-Failure, Time-To-Repair and Time-Between-Failures

can be regarded also as the mean time between a failure and its consequent recovery.

- **Mean Time Between Failures** `MTBF`: Mean interval of time between two consequent failures. It can be described as the sum of `MTTF` and `MTTR`.

- **Failure Rate**: Describes the rate at which failures occurs. It is usually the expected value of a random variable.

- **Coverage**: A value describing which percentage of system failures are covered by fault tolerance mechanisms built into the system.

Figure 1.1 depicts the relationships between time to failure, time to recovery and time between failures. As regards maintainability and safety attributes, `MTTR` is usually used as an indicator to measure this attribute, whereas measures to quantify the safety of a system are usually regulated by domain-specific standard.

Another interesting kind of measures used to quantify system dependability are *Task Completion* metrics introduced by Trivedi in [20]. These metrics show the likelihood

that a user will receive a correct service or, equivalently, the proportion of users who receive adequate service.

Availability Measures

A system is said to be available at a the time t if it is able to provide a correct service at that instant of time. System availability can therefore be expressed as the the following *A(t)* function:

$$A(t) = \begin{cases} 1 & \textit{if proper service at } t \\ 0 & \textit{otherwise} \end{cases} \tag{1.1.1}$$

The measuring of the availability became important with the advent of time-sharing systems. These systems brought with it an issue for the continuity of computer service and thus minimizing system down time became a priority. Availability is a function not only of how rarely a system fails but also of how soon it can be repaired upon failure.

The *Instantaneous Availability* is the probability that a system is up at a give instant t, and it therefore represent a single point of the *A(t)* function. The expected value of the *A(t)* function is refereed as *Steady State Availability* and it may also be computed as the total probability that the system is in an "up" state. Finally, *Interval Availability* is defined as the fraction of time, elapsed from system start until t, in which the system is up. If *A(t)* is a decreasing function of time, *Interval Availability* is higher than the Instantaneous Availability since it is averaged over time. Moreover, both *Interval* and *Instantaneous Availability* converge to the value of *Steady-state Availability*.

Table 1.1: Availability classes and Annual Downtime

Class #	Availability	Annual Downtime
1	90%	36,5 days/year
2	99%	3,65 days/year
3	99,9%	8,76 hours/year
4	99,99%	52 minutes/year
5	99,999%	5 minutes/year
6	99,9999%	31 seconds/year
7	99,99999%	3 seconds/year

Given the value of `MTTF` and `MTTR` parameters, it is possible to compute the steady-state availability as the ratio of these two parameters:

$$A_{ss} = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} \tag{1.1.2}$$

Downtime per year is a more intuitive way of understanding the availability. Table 1.1 depicts several availability classes and the corresponding downtime. Availability classes are named after the number of nines in the annual system uptime percentage.

Reliability Measures

System reliability emphasizes the occurrence of undesirable events in the system. Reliability is an essential feature in systems where no down time can be tolerated. Although the `MTTF` and the *Failure Rate* are useful reliability indicators, the best way to measure system availability is to use the *Reliability Function*. This function represents the probability that an incident has not yet occurred since the beginning

of current system operation. It is usually denoted as *R(t)*:

$$R(t) = P(no\ failures\ in\ [0, t[\ \mid\ correct\ service\ at\ t = 0) \tag{1.1.3}$$

System unreliability, the cumulative distribution of the failure time is expressed as *F(t) = 1 - R(t)*.

Reliability was the only measure of interest to early designers of dependable computer systems. Since reliability is a function of the mission duration T, mean time to failure (MTTF) is often used as a single numeric indicator of system reliability. Another widely adopted measure of reliability is the failure rate, that is, the frequency with which a system fails. Failure rates can be expressed using any measure of time, but hours is the most common unit in practice. The Failures In Time (FIT) rate of a device is the number of failures that can be expected in one billion (10^9) hours of operation. This term is used particularly by the semiconductor industry. Usually, the failure rate of a system is not constant during all the system life-time, but it follows the so-called bath-tube form, i.e., a system experiences a decreasing failure rate when it is firstly deployed, due to infant-mortality failures, then it follows a rather constant failure rate during the operational life, and, finally, it experiences an increasing failure rate at the end of its life, due to wear-out failures.

Task Completion Measures

Task completion measures indicate the likelihood that a task (or job or customer) will be successfully completed. Since the task is the fundamental unit by which work is carried out on a system, the likelihood of successful completion of a task gives a precise assessment of user's perception of system dependability. These measures are very effective in situations where system usage can indeed be broken down into individual tasks, such as a transaction processing system.

Unlike availability and reliability measures, which only take into account the systems itself, task completion measures also include the nature of the task and their interaction with the system. Therefore, these measures take not only into account incidents with their occurrence rates and repair time, but also the effects of such incidents on tasks. The effects are functions of aspects such as the incident profile, the length of time of the task or the sensitivity of the task to interruptions.

Due to their task-dependent nature, it is not possible to express these measures with a general form like availability and reliability measures. For each system different task completion measures have to be defined.

1.1.2 Threats

There are several causes which may lead a system to deliver an incorrect service, i.e., a service deviating from its function. Hardware faults and design errors are just an example of the possible sources of failure. These causes, along with the

manifestation of incorrect service, are recognized in the literature as dependability threats, and are commonly categorized as *failures*, *errors*, and *faults* [1]. A **failure** is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. A service failure is a transition from correct service to incorrect service. The period of delivery of incorrect service is a service outage. The transition from incorrect service to correct service is a service recovery or repair. The deviation from correct service may assume different forms that are called service failure modes and are ranked according to failure severities. Examples of criteria for determining the classes of failure severities are:

1. for availability, the outage duration;

2. for safety, the possibility of human lives being endangered;

3. for integrity, the extent of the corruption of data and the ability to recover from these corruptions.

As far as software systems are concerned, the **CRASH** scale [4] [21] represents an example of user-centric failure severity classification for software systems. **CRASH** is an acronym where each letter is representative of a different failure severity level. *Catastrophic* failures (the failure is not contained within a single task, but other

[4]CRASH is the acronym for Catastrophic, Restart, Abort, Silent, Hindering. Each letter in the acronym represent a different level for a software failure severity.

tasks or the whole system crashed or hung) are the more severe, whereas *Hindering* failures (the correct diagnosis of a trivial problem is made difficult or impossible by an incorrect error code returned by the system) are the less severe.

An **error** can be regarded as the part of a system's total state that may lead to a failure. In other words, a failure occurs when the error causes the delivered service to deviate from correct service. The adjudged or hypothesized cause of an error is called a fault. **Faults** can be either internal or external of a system. Depending on their nature, faults can be classified as:

- *Development faults*: include all the internal faults which originate during the development phase of a system's hardware and software.
- *Physical faults*: include all the internal faults due to physical hardware damages or misbehaviors.
- *Interaction faults*: include all the external faults deriving from the interaction of a system with the external environment.

[1] provides an exhaustive taxonomy of faults. All faults that may affect a system during its life are classified according to eight basic viewpoints, leading to 31 *elementary faults classes*.

Dimension is one of the above mentioned viewpoints: faults can be divided into **Hardware Faults**, which originate in hardware and **Software Faults** which affect

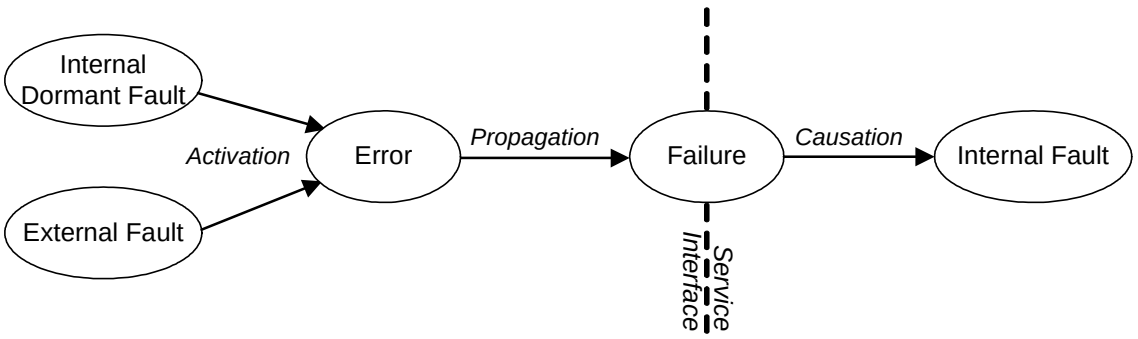

Figure 1.2: Chain of threats

programs or data. Focus in this thesis is on software faults, which are deeply discussed in the following section.

Failures, errors, and faults are related each other in the form of a chain of threats [1], as sketched in figure 1.2. A fault is *active* when it produces an error; otherwise, it is *dormant*. An active fault is either i) an internal fault that was previously dormant and that has been activated, or ii) an external fault. A failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. An error which does not lead the system to failure is said to be a latent error. A failure of a system component causes an internal fault of the system that contains such a component, or causes an external fault for the other system(s) that receive service from the given system. The ability to identify the activation pattern of a fault that has caused on or more errors is the *fault activation reproducibility*.

1.1.3 Means

Over the course of the past 50 years many means have been developed to attain the various attributes of dependability. These means can be grouped into four major categories [1]:

- **Fault Prevention**, to prevent the occurrence or introduction of faults. Fault prevention is enforced during the design phase of a system, both for software (e.g., information hiding, modularization, use of strongly-typed programming languages) and hardware (e.g., design rules).

- **Fault Tolerance**, to avoid service failures in the presence of faults. It takes place during the operational life of the system. A widely used method of achieving fault tolerance is redundancy, either temporal or spatial. Temporal redundancy aims to re-establish proper operation by bringing the system in a error-free state and by repeating the operation which caused the failure, while spatial redundancy exploits the computation performed by multiple system's replicas. The former is adequate for transient faults, whereas the latter can be effective only under the assumption that the replicas are not affected by the same permanent faults. Both temporal and spatial redundancy requires error detection and recovery techniques to be in place: upon error detection (i.e., the ability to identify that an error occurred in the system), a recovery action is performed. Such a recovery can assume the form of rollback (the system is brought back to

a saved state that existed prior the occurrence of the error; system state must be periodically saved, via checkpointing techniques [22]), rollforward (the system is brought to a new, error-free state), and compensation (a deep knowledge of the erroneous state is available to enable error to be masked). Fault masking, or simply masking, results from the systematic usage of compensation. The measure of effectiveness of any given fault tolerance technique is called its coverage, i.e, the percentage of the total number of failures that are successfully recovered by the fault tolerance mean.

- **Fault removal**, to reduce the number and severity of faults. The removal activity is usually performed during the verification and validation phases of the system development, by means of testing and/or fault injection [23]. However, fault removal can also be done during the operational phase, in terms of corrective and perfective maintenance.

- **Fault forecasting**, to estimate the present number, the future incidence, and the likely consequences of faults. Fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. Evaluation has two aspects: qualitative, or ordinal, evaluation, that aims at identifying, classifying, and ranking the failure modes that would lead to system failures; and quantitative, or probabilistic, evaluation, that aims to evaluate in terms of probabilities the extent to which some of the attributes are

satisfied; those attributes are then viewed as measures.

1.2 Software Faults

Among the 31 fault classes identified in [1], only 13 cope with software faults. A detailed view of these software fault classes is reported in figure 1.3.

While 8 of these 13 classes deal with problems occurring at the operational stage,

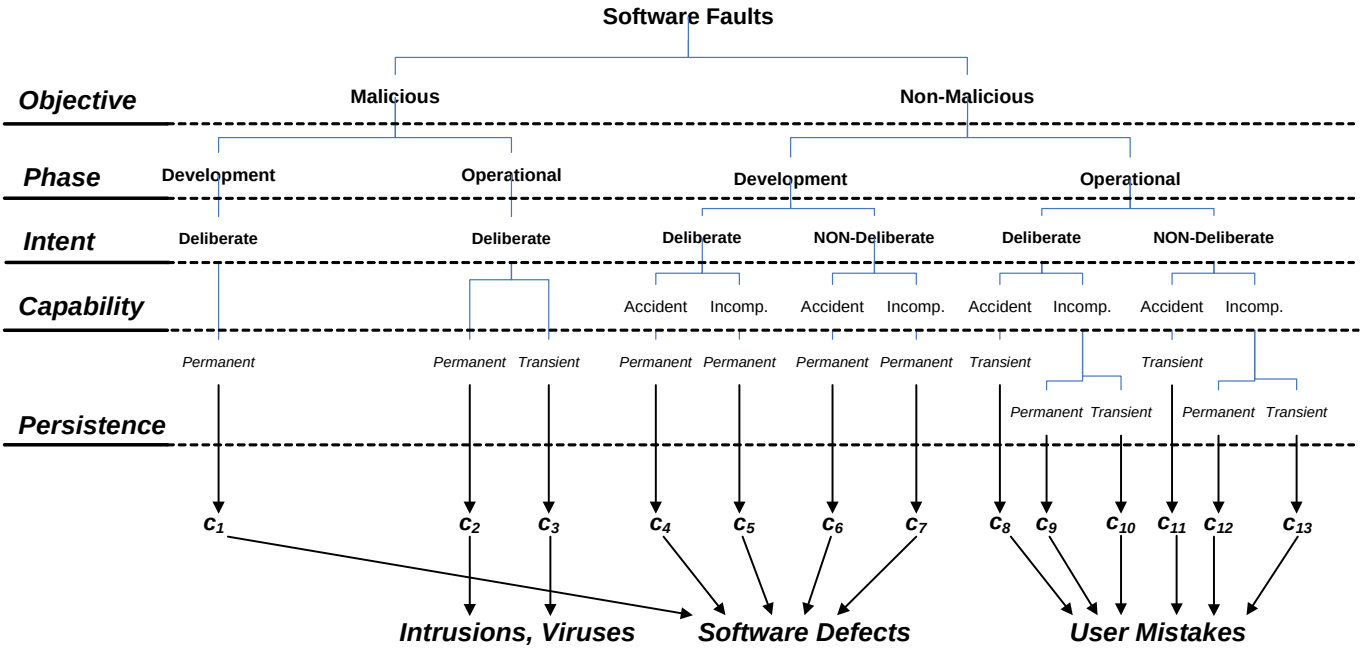

Figure 1.3: Classification of Software Faults according to Laprie and Avizienis [1]

either for user mistakes or attacks, the remaining 5 classes deal with faults introduced in the development stage which can manifest their effect during system operation. As remarked by figure 1.3, software defects are always *permanent* faults, since the fault

lies in application's source code.

These faults can be either *malicious* (logic bombs, trapdoors) or *non-malicious*. Usually the terms "software defect" and "bug" are used to refer to these kind of faults. In this work, focus is on software defects. Several studies confirm that nowadays the greatest percentage of system failures are due to these defects [24]. As stated in [25] "Software failure is the nightmare of the Information Age". Software defects are more critical when Off-the-shelf (OTS) items are used in system design. The use of such items introduces additional problems. They may come with known development faults and may contain unknown faults as well. Moreover, their specification may be incomplete or event incorrect. This problem is especially serious when legacy OTS components are employed. Beyond "traditional" errors like exceptions and wrong computations, some development faults affecting software can cause Software aging [2]. Software aging is a phenomenon in which progressively accrued error conditions lead to either performance degradation or transient failures or both. Examples are memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage space fragmentation, and accumulation of round-off errors.

Software defects will be classified according to their semantics, following the Orthogonal Defect Classification (ODC), and to their reproducibility, distinguishing bugs which exhibit a deterministic behavior from others which instead seem to be non-deterministic.

In the rest of this section, the term "software fault" will refer exclusively to faults introduced in the development phase, either malicious or not malicious. Moreover we will use either the terms "software defect" or "bug" to refer to software faults.

1.2.1 Orthogonal Defect Classification

Orthogonal Defect Classification (ODC) [26] has been presented in 1992. It brought a scientific approach to measurement in a difficult area that, up to the early '90s, was based on *ad hoc* solutions suitable for a single product or a single line of products or, even worse, demanded to opinion-based classifications. It may be regarded as the first attempt to define a standardized methodology to classify software defects and provide useful feedbacks to software developers. It represents a fundamental milestone in the analysis of the dependability of software systems, since it provides a method to describe each development software fault from a semantic perspective. Software defects are grouped into orthogonal *defect types* thus avoiding confusion in defect classification; defect type must also be simple, in that they should be obvious for a programmer, and general, in that the should be applicable to every class of software and to every stage of the software life cycle. In each case a distinction is made between something *missing* and something *incorrect*. For instance, having a missing function is something really different with respect to having an erroneous function. In [26] the following defect types have been defined:

- *Function* - The fault affects significant capability, end-user interfaces, interface

with hardware architecture or global data structures and should require a formal design change. Usually these faults affect a considerable amount of code and refer to capabilities either implemented incorrectly or not implemented at all.

- *Interface* - This defect type corresponds to errors in interacting with other components, modules or device drivers, via macros, call statements, control blocks or parameters lists.

- *Assignment* - The fault involves a few lines of code, such as the initialization of control blocks or data structures. The assignment may be either missing or wrongly implemented.

- *Checking* - This defect addresses program logic that has failed to properly validate data and values before they are used. Examples are missing or incorrect validation of parameters or data in conditional statements.

- *Timing/Serialization* - Missing or incorrect necessary serialization of shared resources, wrong resources serialized or wrong serialization technique employed. Examples are deadlocks or missed deadline in hard real time systems.

- *Algorithm* - This defect includes efficiency and correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change.

- *Build/package/merge* - Describe errors that occur due to mistakes in library systems, management of changes, or version control. Rather than being related to the product under development, this defect type is mainly related to the development process, since it affects tools used for software development such as code versioning systems.

- *Documentation* - This defect type affects both publication and maintenance notes. It has a significant meaning only in the early stages of software life cycle (Specification and High Level Design)

Beyond *Defect Types*, Defect Triggers also have an important role in ODC. A *Defect trigger* is a condition that allows a defect to be activated. Even if extensive testing has been performed, a series of circumstances may allow a defect to surface after that a software system has been deployed. Ideally, the defect trigger distribution exhibited on the field should be similar to the distribution observed in the test environment: significant discrepancies between the two identified potential problems in the system test environment (e.g.: the test environment fails in assessing the robustness of the components or system under test). The most used defect trigger categories are:

- *Boundary Conditions* - Software defects were triggered when the systems ran in particularly critical conditions (e.g.: low memory).

- *Bug Fix* - The defect surfaced after another defect was corrected. This may

happen either because the bug fixed allowed users to executed a previously untested (and buggy) area of the system, because in the same component where the bug was fixed there was another undiscovered bug, or because the fix was not successfully, in that it caused another defects on the same (or on a different) component.

- *Recovery* - The defect surfaced after the system recovered from a previous failure.
- *Exception Handling* - The defect surfaced after an unforeseen exception handling path was executed.
- *Timing* - The defect emerged when particular timing conditions were met (e.g.: the application was deployed on a system with a different thread scheduler).
- *Workload* - The defect surfaced only when particular workload condition were met (e.g.: only after the number of concurrent requests to serve was higher than a particular threshold).

Extensions to the original ODC classification

Although ODC provides an important basis to understand software faults, it relates faults to the way they are corrected: in order to fully understand the nature of the fault and its activation path it is necessary to extend the ODC classification taking into account other factors, such as those related to language programming constructs

being used.

Given that the same faults can be usually corrected in different ways, a closer look into the exact nature of the faults is necessary for accurate fault emulation.

Madeira and Durães in [27] proposed a novel fault classification methodology extending the ODC classification. This classification used ODC as a first step; then, in a second step, faults were grouped according to the nature of the defect, defined from a building block programming perspective. For each ODC class, a software fault is characterized by one programming language constructs that may be either missing, wrong or superfluous (instead, in ODC, the cause of software defect can be an incorrect or a missing construct); finally, in the third and last step, faults were further refined and classified in specific types.

Since fault types provided by ODC are too broad, it may happen that several different faults are encompassed by the same type and therefore classified in the same category, even if the nature and the activation path of these is fault is totally different.

In order to provide a detailed classification of software faults, a set of fault types, representative of the most common types of software faults, was identified. As an example of such fault types it is possible to consider *Missing function calls (MFC)*, which is a particular kind algorithm fault in which a required function call is missing.Field data collected in [27] reported more than 20% of faults belong to this category.

1.2.2 Bohrbugs, Heisenbugs and Aging-related bugs

ODC classification and its extensions are very useful in order to classify a software defects after it has been detected.

Unfortunately, detection and diagnosis of software faults often becomes a very hard task: the main problem is the reproducibility of the software defect, that is the ability to identify the activation pattern of fault that has caused one or more errors. Faults whose activation is easily reproducible (e.g., through a debugger) are called *solid* or *hard* faults; faults whose activation is not systematically reproducible are called *elusive* or *soft* faults. They are intricate enough that their activation conditions depend on complex combinations of the internal state and the external environment (i.e., the set made by up other programs, services, libraries, virtual machines, middleware and operating systems the applications interact with). The conditions which activate the fault occur very rarely and can be very difficult to reproduce.

Software fault reproducibility was first discussed in [28]. In this paper, Jim Gray claimed that the greater part of faults in the operational phase are transient just like hardware faults: if the program state is reinitialized and the failed operation retried, the operation will usually not fail the second time.

Industrial software products, usually, before reaching the operational phase (or being placed on the market) undergo several steps aimed at removing each defect; among

these steps it is possible to mention structured design, design review, quality assurance, unit testing, component, integration testing, alpha and beta test. In this way, the greatest part of "hard" software bugs, which are always activated on retry, can be easily fixed. The residual bugs are rare cases, typically related to strange environmental conditions, limit conditions (e.g: out of storage, out of memory, buffer overflows, etc.) or race conditions.

For these reasons the number of hard faults decreases over time (as depicted in figure

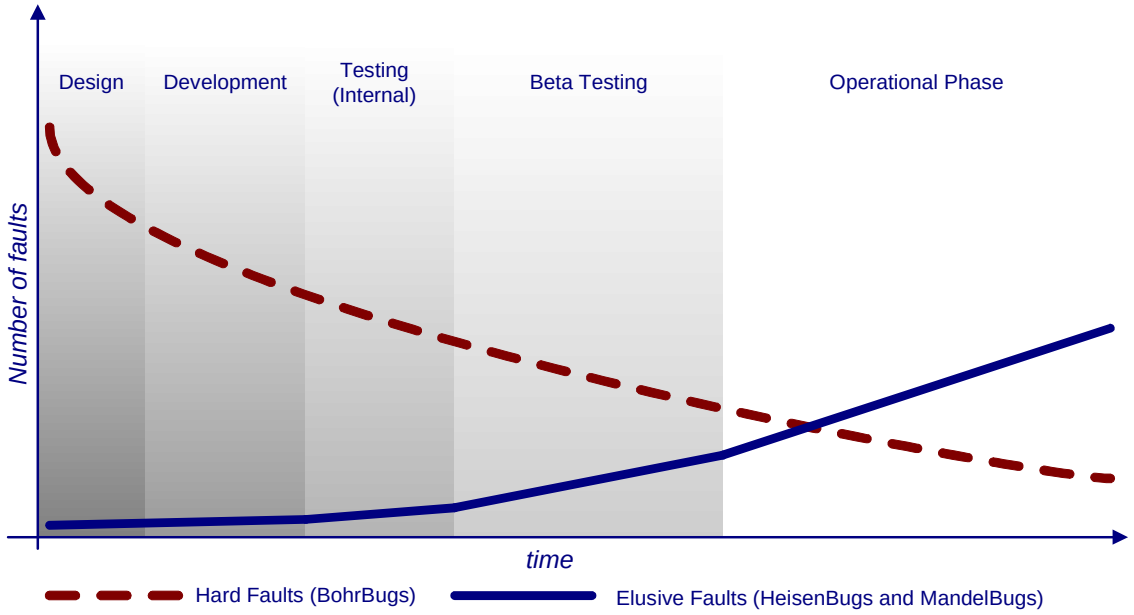

Figure 1.4: Evolution over time and software life cycle phase of reproducible and non-reproducible software faults

1.4, becoming negligible after a long period of production (although the release of a new version of the same software often causes a burst in this curve). On the other hand, the number of elusive faults increases with time: during the development phase

this number is very low, since the system is yet under development; internal testing phases (such as unit testing and alpha testing) are able to discover just a few elusive faults, since the system under test runs always in the same environment; once the system is delivered out of the production environment for beta testing, a consistent number of elusive faults are reported, since the system runs in several different environments with workloads very different from the ones applied in the testing phase; the number of elusive faults further increases once the system has been brought to the operational phase, since it has to interact with even more different environments, and, more important, the components (mainly Off-the-Shelf) which the software system interacts with, change over time (new features, new versions, etc. etc.).

Gray named these two broad classes of faults respectively **BohrBugs** and **Heisenbugs**.

Bohrbugs, which recall the Bohr atom model, are bugs that manifests reliably under a well-defined set of conditions. Thus a bohrbug does not disappear or alter its characteristics when it is activated. These include the easiest bugs to fix (where the nature of the problem is obvious), but also bugs that are hard to find and to fix, which remain in the software during the operational phase. A software system with a Bohrbug is analogous to a faulty deterministic finite state machine.

Heisenbugs were named after the Heisenberg uncertainty principle, a quantum

physics term which is commonly used to refer to the way in which observers affect the measurements of the things that they are observing; they are computer bugs that disappear or alter their characteristics when the software is debugged. A software system with an Heisenbug is analogous to a faulty non-deterministic finite state machine. One common example is a bug that occurs in a release-mode compile of a program, but not when researched under debug-mode; another is a bug caused by a race condition. One common reason for heisenbug-like behaviour is that executing a program in debug mode often cleans memory before the program starts, and forces variables onto stack locations, instead of keeping them in registers. Another reason is that debuggers commonly provide watches or other user interfaces that cause code (such as property accessors) to be executed, which can, in turn, change the state of the program. Moreover, many heisenbugs are caused by uninitialized variables.

However, software developers and testers often encounter failures that cannot be reproduced, since under seemingly exact conditions, the actions that a test case describes can sometimes, but not always, lead to a failure. Apparently this is a typical heisenbug-like behavior. Instead these faults often have a different nature. There may be a long delay between fault activation and final failure occurrence (because several error state are traversed before the failure or because the fault progressively accrues an abnormal condition until the system fails). Then it is difficult to identify the actions that actually caused the failure. Fault activation is just apparently

non-deterministic: actually, there exist a particular exact condition under which the fault is deterministically activated, but detecting this condition is so difficult that the bug is label as non-deterministic. This usually happens with complex software systems employing one or more *Off-The-Shelf (OTS) items*. Indeed, technical documentation and source code for these items is often incomplete or totally unavailable; furthermore, interactions between an application and the employed OTS items may lead to unpredictable effect not foreseen during development and testing of the OTS item itself. In scientific literature these software defects are usually named **Mandelbugs**(which name derives from the name of fractal innovator Benoit Mandelbrot) [29]. Some authors use this term as a synonym for **Heisenbugs**, since they claim that there is no way for a to distinguish a bug whose behavior *appears* chaotic and a bug whose behavior is *actually* chaotic. However these two kinds of software faults are somewhat different:

- A **Heisenbug** is a computer bug that disappears or alters its characteristics when it is researched.
- A **Mandelbug** is a computer bug whose causes are so complex that its behavior appears chaotic.

Summarizing, three different classes of computer bugs, depicted in figure 1.5, whose edges are not sharply marked, were defined. **Bohrbugs** and **Mandelbugs** are deterministic, even if the latter are so complex that appear to be chaotic; on the other

hand **Heisenbugs** are totally non-deterministic.

Often software systems running continuously for a long time tend to show a degraded performance and an increased failure occurrence rate. This phenomenon is usually called *Software Aging* [9]. Tracing back to the root cause of the failure (or of the degraded performance) is really hard and sometimes impossible. This happens because these failures are caused by accrued error conditions, such as round-off errors, data corruption or unreleased physical memory. Software defects which cause software

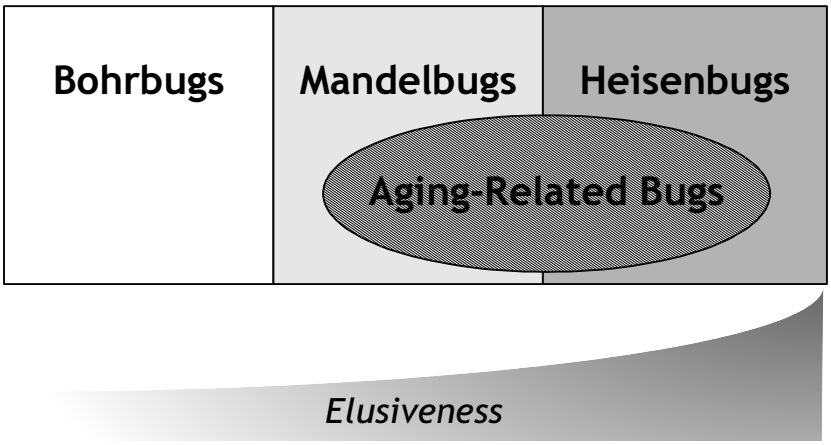

Figure 1.5: Categories of software faults and their elusiveness

aging phenomena are called **Aging Related Bugs**. A typical example of an aging related bug is an unreleased memory region inside a program's heap area: memory allocated with `new` or `malloc` is never freed using `delete` or `free`. This kind of software faults may be either environment-independent (i.e.: it occurs despite of the host environment) or dependent on the environment (e.g.: a fault which is dependent on the message arrival order). In the former case, the faults fall into the category of

Mandelbugs, whereas in the latter case the faults fall into the category of Heisenbugs. Actually, nothing prohibits aging related bugs to be also classified as Bohrbugs: for instance, a missing `delete` statement is a clearly deterministic fault; however, assuming that industrial software system undergo intensive testing before the production stage, it is very unlikely that easily traceable defects are still in the product. Therefore, deterministic aging related bugs in the operational phase should fall into the Mandelbug category. Figure 1.5 depicts the placement of Aging Related Bugs with regards to other bug categories.

1.2.3 Dealing with Software Development Faults

Figure 1.6 reports, for each of the above defined class of software faults, the techniques which may be used to remove or tolerate the above mentioned faults. Except *Debugging*, all the techniques are concerned with the operational phase. Bohrbugs are easily reproducible and hence can be easily removed. These faults should have ideally been removed during the debugging phase. If such faults remain in the operational phase, then *Design Diversity* represents the best solution. In this way several applications providing the same functionality but using different design/implementations are used to mask faults in individual implementations. *Design diversity* is also a valuable solution in order to tolerate Mandelbugs: although they are so complex that their behavior seem to be unpredictable, they are always deterministic and therefore tolerated by design diversity just like bohrbugs.

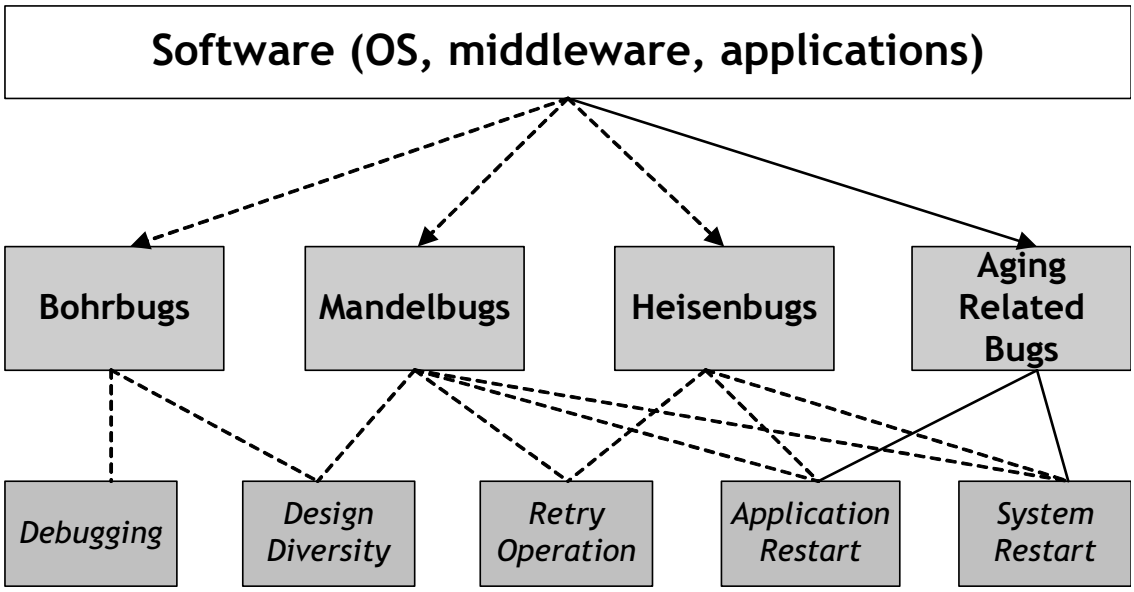

Figure 1.6: Fault model for software components

Design diversity techniques [30] are redundancy techniques, where independently developed versions of software are concurrently executed in order to augment the reliability of the software system. *N-version programming* and Recovery Blocks are representative examples of design diversity techniques. Different versions of the same software are usually developed using different design methodologies, algorithms, compilers and run-time systems. This means that reliability comes with a non-negligible cost during. Moreover, a *software driver* is required to decide which version's outcome choose. This software is usually not replicated, thus becoming a single point of failure for the whole systems. Furthermore, software developers, even the more skilled, tend to make the same errors even if they are working on totally different

algorithms (e.g.: errors in memory management). Therefore, although design diversity techniques are a valuable solution to augment system reliability, they cannot be considered as a *panacea* for software reliability.

Even mature software can be expected to incur in a Heisenbug, leading to intermittent application failures. Simply retrying a failed operation, or if the application process has crashed, restarting the process might resolve the problem. These solutions may work not only in presence of Heisenbugs, but also in presence of Mandelbugs: given their unpredictable behavior, retrying the same operation or restarting the whole application the condition which triggered the failure may be removed.

Process pairs, introduced in [28], are a redundancy technique capable of tolerating also Heisenbugs. There are several approaches to designing and implementing Process Pairs, such as *Locksteps* and *State, Automatic or Delta Checkpointing*. Y.Y.Zhou and others [31] implemented a checkpoint based rollback-recovery mechanisms based on shadow processes capable of tolerating Heisenbugs by modifying the environment in which the process executes before its recovery.

When coping with aging related bugs, debugging,design diversity and operation retrying do not help. Aging related bugs are usually too difficult to discover through debugging, where as design diversity will not avoid resource exhaustion. Moreover retrying the operations will only accrue error conditions. **Rebooting** the application or the whole system is the best way to free exhausted resources or reset accrued error

conditions; in certain cases it is possible to restart only a single component of the entire software system.

The estimation of the rate of resource exhaustion and consequently the expected time of software failure has been the focus of research on *Software rejuvenation* techniques. It has been shown [2] that periodically restarting a process, rebooting a node, or doing a prediction-based rejuvenation based on the observed rate of resource exhaustion may help prevent the software from crashing, and increment system availability.

1.3 Assessing the Dependability of Software Systems

In recent years, ensuring certain levels of availability and reliability has become one of the more relevant tasks for software engineers and developers.

Answering to questions like *"Is this system going to satisy all my business needs?"* or *"does this software work for my organization?"* is no more sufficient in order to persuade a customer to choose a software product. People involved in software development processes are even more forced to answer questions like *"May I trust operations computed by this software system?"*, *"How much time I can expect the system to run before a failure occurs?"*, or *"If the system fails, how much should I wait before it returns available?"*.

Given the previous considerations about software faults, it is not possible to answer these questions in a predictable way. On the other hand, people are not willing to

invest their money in something they do not know how much it is reliable.

Therefore it becomes necessary to establish methods and techniques to evaluate how much a software system, or a software component, can meet users' requirements about availability, safety, reliability, and security. In other words, it is necessary to establish methods and technologies to assess and certify the dependability of software systems. When talking about software dependability assessment, software and dependability engineers usually have different point of views: the formers are mainly focused on *software testing* and intend dependability as the likelihood the software is defect-free, whereas the latter are mainly focused on the software system or component in the operational phase, in order to:

- Derive an analytical model of the system, using techniques such as Markov chains and Petri Nets, and compute the expected dependability level by solving the model;
- Assess, through field-based measurement, the current dependability level achieved by software system or by a particular software component;
- Measure the effectiveness of fault tolerance mechanisms, if present;
- Identify dependability bottlenecks in the system, i.e.: pathological conditions under which the system is more failure-prone, or weak components negatively affecting the dependability of the whole system.

- By analyzing collected data or solving system models, encompass solutions to improve the dependability of the system (e.g.: by rejuvenation policies, replication, checkpointing, etc.)

There are several approaches to assess the dependability of a software. Each of these adopts a different methodology or technique to evaluate system dependability, although other approaches uses more than one technique at the same time.
In last decades a number of techniques and methodologies to evaluate software dependability ,described in section 1.4, have been proposed in scientific literature.
These techniques may be classified according to the criteria reported in table 1.2. For each criterion reported in this table, we fix three possible levels: *Low*,*Medium*, and *High*. The meaning of these levels for each specific criterion is reported in the third column of table 1.2.

1.4 Dependability Assessment techniques

This section is focused on techniques and methodologies for evaluating and assessing the dependability of software systems. Particular emphasis will be given to Software Aging Analysis, which is the main topic of this thesis.
After describing the main features of these methodologies and techniques and some relevant works in the respective fields, these will be compared using the classification criteria reported in table 1.2. This comparison aims at giving a sort of guidance in the

Table 1.2: Criteria for the classification of dependability assessment techniques and methodologies

Cost	Cost required for dependability evaluation, both from an economical and a temporal perspecitve	LOW	The analysis may be performed solving analytic models or using simulation tools.
		MEDIUM	A limited number of machines is required. Time required for evaluation is in the order of weeks
		HIGH	Several machines are required. Time required to perform the analysis is in the order of months.
Detail	Level of detailed of dependability measure	LOW	Only sinthetic measures, such as MTTF and MTTR, are returned
		MEDIUM	Indicators about system dependability, such as MTTF are enriched with data such as information about failures and workload
		HIGH	Full details about the dependability behavior of single components or functions are given
Coverage	Percentage of system functions and components covered by the evaluation campaign	LOW	The evaluation covers only a part of the whole system (e.g.: a single layer in a network stack)
		MEDIUM	The evaluation covers each interface exposed by the system, but not its internal state (or viceversa)
		HIGH	The evaluation throughly covers each function or component of the system
Management	The capability of controlling experiments performed in order to evaluate system dependability	LOW	Nothing, or very little, can be controlled (E.g.: the evaluation is performed on an already operational system)
		MEDIUM	A few parameters can be controlled (E.g.: the number and size of e-mails in evaluating the dependability of a mail servers)
		HIGH	A consistent number of experiment parameters may be controlled

selection of the proper technique(s) and methodologie(s) to adopt when evaluating the dependability of a software system.

1.4.1 Field Failure Data Analysis

The Field Failure Data Analysis (FFDA) of a computer system embraces all measurement-based techniques which are performed in the operational phase of the system's life cycle. This analysis aims at measuring dependability attributes of the actual and deployed system, under real workload conditions. By measuring it is meant to monitor and record natural occurring errors and failures while the normal system operation. In other words, the failing behavior is not forced or induced in the systems. The objective of a FFDA campaign mainly concerns the detailed characterization of the actual

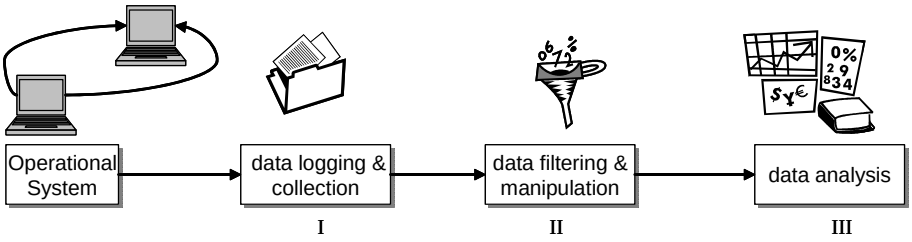

Figure 1.7: The FFDA methodology

dependability behavior of the operational system. More in detail, FFDA studies main objectives can be summarized as the following:

- identification of the classes of errors/failures as they manifest in the field, along with their relative severity and correlation among them. In other terms, FFDA is useful to derive the actual failure model of an operational system;
- analysis of failure and recovery times statistical distributions;
- correlation between failures and system workload;
- modeling of the failing behavior and recovery mechanisms, if any;
- identification of the root causes of outages, and indication of dependability bottlenecks;
- provision of figures useful to validate or to populate simulated failure models;
- derivation of general results which a crucial to guide research and development of fault avoidance, masking and tolerance means.

Although FFDA studies are useful for evaluating real systems, they are limited to manifested failures, such as the ones that can be traced. In addition, the particular conditions under which the system is observed can vary from an installation to another, thus casting doubts on the statistical validity of the results. FFDA studies may require a long period of observation of the target system, especially when the system is robust and failure events are rare. FFDA studies usually account three consecutive steps, as shown in figure 1.7: **i)** data logging and collection, where data are gathered from the actual system, **ii)** data filtering and manipulation, concerning the extraction of the information which are useful for the analysis, and **iii)** data analysis, that is the derivation of the intended results from the manipulated data.

Data Logging

Common techniques for data logging and collection are **failure reports** and **event logging**.

Failure reports may be human or machine-generated. The problem with human-generated reports is that operators are responsible for the detection of the failure, hence some failure may remain undetected. Moreover, the information contained in the report can vary from one operator to another, depending on his experiences and opinions. However, as it will be shown later in this section, studies based on failure reports allowed researchers to build trustworthy software reliability growth models or discovers bottlenecks in the dependability of software systems.

Unlike failure reports, event logs are always machine-generated. Hence, from these logs, it is possible to extract useful information about system failures. A limit of event logging is that the detection of a failure event depends on whether or not the application/system module logs that particular event.

Data Filtering and Manipulation

Data filtering and manipulation consist in analyzing collected data for correctness, consistency, and completeness. This concerns the **filtering** of invalid data and the **coalescence** of redundant or equivalent data. This is especially true when event logs are used. Logs, indeed, contain many information which are not related to failure events. In addition, events which are close in time may be representative of one single failure events. They thus need to be coalesced into one failure event.

Filtering is used to reduce the amount of information to be stored, and to concentrate the attention only on a significant set of data, thus simplifying the analysis process. Coalescence techniques can be distinguished into temporal, spatial, and content-based. Temporal coalescence, or *tupling* [32], exploits the heuristic of the *tuple*, i.e., a collection of events which are close in time. The heuristic is based on the observation that often more than one failure events are reported together, due to the same underlying fault. Indeed, as the effects of the fault propagate through a system, hardware and software detectors are triggered resulting in multiple events. Moreover, the same fault may persist or repeat often over time. Spatial coalescence is used to relate

events which occur close in time but on different nodes of the system under study. It allows to identify failure propagations among nodes, resulting particularly useful when targeting distributed systems. Finally, content-based coalescence groups several events into one event by looking at the specific content of the events into the event log.

Data Analysis

The data analysis step consists in performing statistical analysis on the manipulated data to identify trends and to evaluate quantitative dependability measures.

Failure classification is a first analysis step, which aims at categorizing all the observed failures on the basis of their nature and/or location. In addition, descriptive statistics can be derived from the data to analyze details such as the location of faults, errors and failures, the severity of failures, the impact of the workload on the system behavior. These statistics are used to quantify system dependability, by means of the general measures described in section 1.1.1, or using more detailed measures like probability distributions (in particular the "time to failure" distribution) or system-specific measures such as error propagation time and the distribution of failures among the components of the system.

Relevant Works

Several FFDA studies have been proposed in the literature over the past three decades, each of them addressing different systems, collecting data from different data sources,

and proposing different results. The importance of FFDA studies of computer systems has been recognized since many years. The first seminal contributions date back to the 70s with studies on the Chi/OS for the Univac [33], and CRAY-1 systems. The research has then broadened its scope over the years addressing a wide set of systems and pursuing several objectives. The 80s and the early 90s have been characterized by FFDA studies on mainframe and multicomputer systems, such as the IBM 370 with the MVS OS [34], the DEC VAX [35], and Tandem systems [36]. In particular, the latter work, written Jim Gray on the availability of Tandem Systems, pointed out that software was responsible for the largest part of system failures. Therefore FFDA studies shifted their focus on software, mainly operating systems, especially Windows [37] and Unix/Linux [38]. At the same time, as the Internet increased in popularity, many studies emerged, trying to assess the dependability of the network of networks [39]

The present decade has witnessed an even broader spectrum of research, adding contributions on virtual machines, applications, embedded systems, large-scale and parallel systems, and mobile distributed systems. Over the years, many objectives have been pursued, from the mere statistical classification and modeling of failure events, to the identification of trends and correlations, and the experimental evaluation of malicious attacks.

1.4.2 Dependability Benchmarking

The idea of benchmarking dependability features of computer systems or computer components has caught the attention of researchers in recent years, whereas in the last two decades the term "benchmarking" has been usually referred to performance benchmarking. Unfortunately the seek for pure peak performance also have caused that, in many cases, the systems and configurations used to achieve the best performance are very far from the systems that are actually used in practice. Since many application have strict dependability requirements, it is compulsory to shift focus from performance to the measurement of both performance and dependability.

Dependability benchmarking is a dependability evaluation technique based on fault-injection [23]. Here faults are injected to evaluate dependability features of a component or sub-system of the whole system rather than evaluating the coverage and the effectiveness of fault-tolerance mechanisms.

A dependability benchmark can be defined as *the specification of a standard procedure to assess dependability related measures of a computer system or computer component [40]*; unlike other evaluation techniques, is a reproducible and cost-effective way of performing such a characterization, especially for comparative purposes. The idea of benchmarking the dependability of a software system has been first proposed by D.P. Siewiorek in 1997 [41], and evolved over years. More recently, the *DBench* project,

financed under the European Union's 6th Frame Program, has extended and standardized the dependability benchmarking process, defining the main dimensions that are decisive for defining dependability benchmarks and the way experimentation can be conducted in practice.

In order to perform a dependability benchmark on a software system (called *System Under Benchmark (SUB)*) the benchmark performer has first to choose: **i)** the *Benchmark Target (BT)*, that is the component or subsystem we are interested in evaluating, and **ii)** the benchmark measurements, which can be classified according to the following criteria:

1. *Performance-related*: These measures allow to evaluate system performance in faulty conditions

2. *Comprehensive*: These measures characterize the system at the service delivery level, taking into account all events impacting its behavior.

3. *Specific*: The measures evaluate robustness and or dependability of single features or functions of the BT.

Figure 1.8, drawn from [40], depicts a typical dependability benchmarking scenario, and highlights the most important components of a dependability benchmark. The **Benchmark Target (BT)** is the component or subsystem which is the target of the

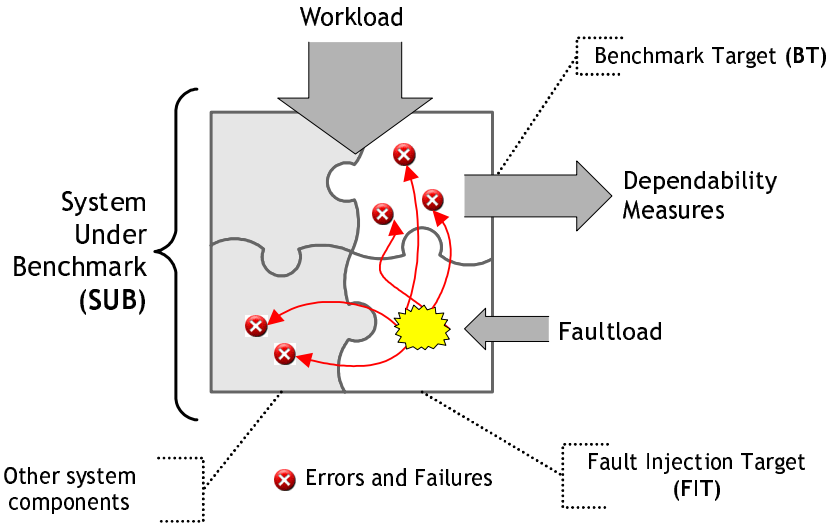

Figure 1.8: Dependability Benchmarking Components

benchmark with respect to its application area and operating environment. Dependability measure (i.e.: the results of the dependability benchmark) are taken on the BT (by either direct on indirect measurement). In order to obtain representative data from a dependability benchmark, it is important that the BT is not altered. Altering the BT (either by injecting faults or by installing an invasive monitoring system) is very likely to produce unreliable dependability measures, since the behavior of the BT is different from the one it should exhibit in "normal" conditions.

The **System Under Benchmarking (SUB)** is the wider system which includes the above described BT. For instance the SUB may be an Operating System while the Benchmark Target may be a particular driver.

The **Workload** represents a typical operational profile applied to the SUB in order to benchmark the dependability of the BT. Workload selection is a very important

task for dependability benchmarking as well as for performance benchmarking. The selected workload should be representative of real workloads applied to the SUB and also portable, in case the benchmark is performed in order to compare different benchmark targets. Determining the "optimal" workload for a specific benchmark is basically impossible. Therefore benchmark performers can use industry-standard benchmark applications used also for performance benchmarking. On the other hand, assuming that workload parameters (or a part of them) are controllable, it is possible to excite the SUB with different operational profiles, thus obtaining dependability measures as a function of the applied workload.

The **Faultload** consists of a set of faults and exceptional conditions that are intended to emulate the real threats the system would experience. Faults are applied to one or more components of the SUB (different from the BT) which constitute the *Fault Injection Target (FIT)*. The reliability of the dependability measures carried out by a dependability benchmark is strictly related with the representativeness of the selected faultload. FFDA studies, discussed in the previous section, supply a consistent amount of data about software faults for a wide range of software systems.

Relevant Works

Many scientific works on dependability benchmarking have been published in the last years; in particular, many of these works have been published in the framework of the *DBench* project [42]. Due to its characteristics, this dependability evaluation

technique is particularly suitable for OTS items: indeed a considerable number of dependability benchmarks has been performed on systems such as Operating Systems, Database Managements System, or Transaction Processing System.

As regards Operating Systems, their dependability has been benchmarked with respect to faulty drivers [43] and with respect to faults in applications [44, 45, 46]. In the former case software faults are injected into a particular driver, according to a "commonly observed" distribution of these faults, whereas in the latter case faults are injected into the interface between the Operating System and the application, by corrupting system call parameters

Moreover, the dependability of a number of server applications (DBMS, OLTP, HTTP, ...) has been benchmarked [47, 48, 49]: software faults are usually injected directly into OS system class. OS profilers are employed to select the System Calls in which faults should be injected. Therefore the OS plays as the FIT and the server application as the BT.

1.4.3 Robustness Testing

Robustness testing is a technique aimed at evaluating the robustness of a system, i.e. the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. This technique has been designed specifically to improve robustness of OTS items. Unlike methodologies such as FFDA, it has

not been designed to be performed on operational systems. Robustness testing may be useful in order to i) evaluate the robustness of one or more OTS items before integrating them into an existing software system or ii) evaluate the robustness of a whole software system before moving it to the operational stage.

Robustness testing is sometimes confused with dependability benchmarking. Indeed one of the faultload chosen for OS dependability benchmarking, discussed in one of the final deliverables of the DBench project [42], was generated using Ballista [50], which is probably the most famous robustness testing suite.

In this thesis, when talking about robustness testing, we will refer to *Interface Robustness testing*, which consists of bombarding the public interface(s) of the application/system/API with valid and exceptional inputs. The success criteria is in most cases: "if it does not crash or hang, then it is robust", hence no oracle is needed for the testing. The CRASH scale, already discussed in section 1.1.2, is usually adopted to describe robustness testing results.

Figure 1.9 depicts a typical robustness testing scenario. Once system interfaces to test have been chosen, both valid and invalid inputs are selected according to the expected behavior of the system (which can be retrieved from system specification or API reference manuals). Invalid inputs may include boundary input values, exceptional input values, such as NULL values, or erroneous values (e.g.: an empty string on the file name parameter of an `fopen` call). The behavior of the system is then

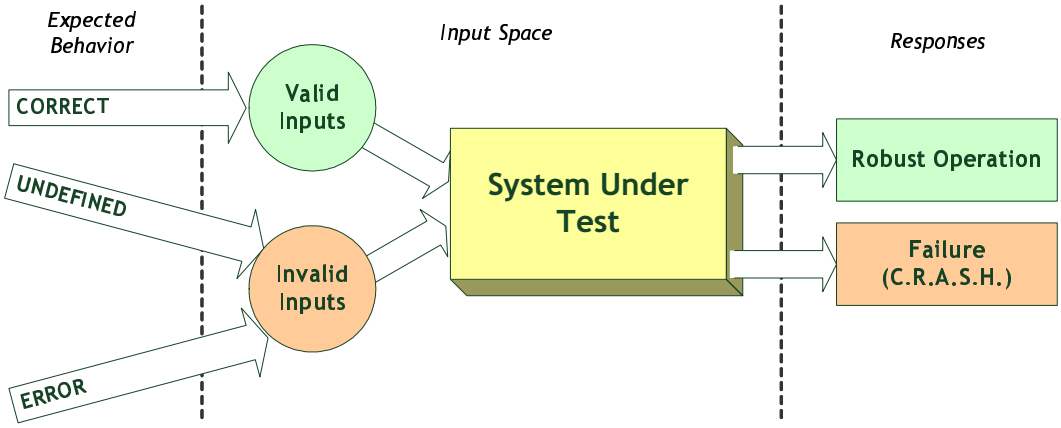

Figure 1.9: A Robustness Testing Scenario

observed. If a failure occurs, this failure is classified according to a failure severity scale (usually the CRASH scale).

Relevant Works

Even if the first paper on robustness testing dates back to 1993 [51], the first papers which actually applied the above described robustness testing technique were published in 1997 [41, 21]. While the former presents the various features that are desirable in a benchmark of system robustness and presents a novel approach to build robustness benchmarks, the latter defines the CRASH scale and reports the results of a robustness test which involved 5 different Unix-Based operating systems.

The paper presented in 1998 at the Fault-Tolerant Computing Symposium [52], further extended the work presented the previous year, formalizing the methodology and introducing the Ballista suite. [53] discusses a comparison between the robustness of

different families of Operating Systems, namely Windows and Linux. This paper presents a novel approach to define benchmarks which are portable across OTS item with deeply different interfaces. Indeed, while previous work compared the robustness of Operating System with a similar System Call Interface (Unix-based OSes), in this work the robustness of several version of the Windows Operating System is compared to Linux's robustness, by identifying common groups of system calls and then analyzing the robustness for each of these groups. Finally, more recently, robustness testing moved from Operating Systems to other OTS items, such as the implementations of CORBA-compliant Object Request Brokers [54].

1.4.4 Fault Emulation Through Error Injection

As discusses in 1.4.2, dependability benchmarking allows to evaluate the dependability of a system or component by injecting faults in a different component (FIT). However, it is often really difficult to inject software faults. It usually happens because there are too many point of injections, or too many possible faults to inject. Chillarege et al. [55] showed that it is possible to directly inject the error caused by such faults in the system. Given some field data about observed faults (described according to ODC classification, described in section 1.2.1), errors are classified according to application-specific criteria. Then faults are mapped onto real system errors.

Field data help in selecting where inject the error and which kind of error should be

injected. This methodology ensures that injected errors are representative of software faults rather than hardware faults. This methodology allows to perform dependability measurements in relatively short time, whereas classical measurement-based dependability analysis would require longer periods. Differences between fault injection and error injection were deeply investigated in [56].

1.4.5 Software Aging Analysis

As already mentioned in section 1.2.2 Software Aging can be defined as a continued and growing degradation of software internal state during its operational life. These problems lead 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 of time. Software aging phenomena occur due to the activation of aging related bugs; typical examples of these bugs are memory bloating and leaking, unreleased file locks, data corruption, storage space fragmentation and accumulation of round-off errors.

The existence of software aging has been widely reported and observed. It has been observed in AT&T billing applications as well as in their telecommunications switching software [6]; during the first gulf war, an aging-related bug in the Patriot missiles software system caused a 55 meters errors in target trajectory calculation after 8 hours of execution [8]; software aging phenomena were also detected in the Apache Web Server [4], even if it had a sort of built-in rejuvenation policy; software aging

phenomena have been documented for a consistent number of operational software systems. Moreover from an anecdotal point of view, it is well known that a consistent number of systems progressively slow down until they have 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 has been called *Software Rejuvenation* and first proposed in 1995 by Huang et al. [2].

The dependability evaluation methodologies analyzed so far are not suitable when coping with aging phenomena. All these methodologies, except FFDA, are injection-based methodologies: a fault, an error, or an "exceptional" input value, is injected, and the behavior of the system is then observed. Even if these techniques allow to highlight the presence of Bohrbugs and Heisenbugs they are not able to discover the presence of aging phenomena. On the other hand, FFDA, according to system log's detail level, once a failure has been observed, has the potential to address software aging as a cause of that failure. However FFDA has not been specifically designed to address software aging phenomena. The study of software aging phenomena requires the assessment of the current health of the system, the estimation of the expected time to system resource exhaustion (a measure often called *Time to Exhaustion (TTE)*), and the determination of the optimal rejuvenation schedule.

The study of software aging can be broadly classified into two approaches: the *Analytic Modeling* approach and the *Measurement Based*. The first assumes failure and repair time distributions of a system and obtains optimal rejuvenation schedule to maximize availability, or minimize loss probability or downtime cost. The Measurement based approach applies statistical analysis to data collected from systems and applies trend analysis or other techniques to determine a window of time over which to perform rejuvenation in order to prevent unplanned outages. The rest of this section is concerned with the discussion of these two approaches.

Analytic Modeling Approach

Analytic modeling generally deals with determining the optimal times to perform software rejuvenation in operational software systems. The optimal rejuvenation schedule is determined starting from analytical models and the accuracy of the determined schedule is determined by the assumptions made in the model for capturing aging.

Analytic models were first employed in order to prove that, when dealing with systems affected by software aging, software rejuvenation allows to reduce the cost due to system downtime [2] and minimizing program completion time [57].

As regards the kind of failures considered, several papers take into account only failures causing total unavailability of the software [57, 58, 2],whereas in [59] a gradually decreasing service rate is considered; a of model which takes into account both kinds

of failures is reported in [5].

Different probabilistic distributions were also chosen for time-to-failure (TTF). Some papers, such as [58] and [2], are restricted to an hypo-exponential distribution, whereas other papers, such as [57] employs more general distributions for TTF, like the Weibull distribution. However these TTF models are not able to capture the effect of load on aging, as it has been done in [5] and [60].

Stochastic processes are always used to analytically modeling software systems. In [59] a Markov Decision Process (MDP) is used to build a software rejuvenation model in telecommunication system including the occurrence of buffer overflows.In [58] Markov semi-ReGenerative Processes (MGRP), in conjunction with Stochastic Petri Nets (SPN), are used to build a simple but general model for estimating the optimal rejuvenation schedule in a software system. Petri Nets, in particular Stochastic Deterministic Petri Nets (SDPN) are employed in [7], in order to build a model to analyze the performability of cluster systems under varying workload. Non-homogeneous, continuous time Markov Chains are instead used in [5]. Semi-Markovian Processes have also been used to model proactive fault management in [60].

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). Recently, Trivedi and Vaidyanathan in [61] addressed this problem, by building a measurement-based semi-markovian model for system workload, defining a set of workload states through cluster analysis, estimating TTE for each considered resource and state using reward functions, and finally building a semi-Markov availability model, based on field data rather than on assumptions about system behavior.

Measurement-Based Approach

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 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 in [9]. In this paper, a set of 9 Unix Workstations has been 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. The aging analysis was performed using the Mann-Kendall statistic test to reject the null hypothesis of no trend in data, and the non-parametric Sen's algorithm [62] to perform slope estimation. Reported result showed that for all the considered workstations, free memory

and swap space showed the highest aging trends. However, different machines showed consistent differences in TTE for the same resources, thus highlighting the need to take into account the applied workload when studying software aging phenomena.

An interesting workload based software aging analysis can be found in [10]. This paper presents the results of an analysis conducted on the same set of Unix workstation of the previous paper which takes into account also some workload parameters, such as the number of CPU context switches and the number of system call invocations. The approach adopted in this paper is slightly different from the one adopted in the previous paper. Different workload state are first identified through statistical cluster analysis; then a state-space model is built determining sojourn time distributions, i.e.: the statistical distribution of the time spent in each workload state; a reward function, based on the resource exhaustion rate for each workload state, is then defined for the model. By solving the model, authors obtained resource depletion trends and TTE for each considered resource in each workload state. Results of this analysis confirmed that, given a particular resource, on different workstations it exhibited similar depletion trends in the same workload state, thus confirming the validity of the approach. The methodology presented in this paper allows to perform a workload-driven characterization of aging phenomena, which more useful and powerful than the workload-independent characterization presented in [9].

An interesting measurement-based approach to software rejuvenation based on a

closed-loop design, is presented in [63]. As for many papers on software aging and rejuvenation, free physical memory is the monitored resource. A supervisor process monitors the real process and automatically performs rejuvenation actions whenever free physical memory goes below fixed thresholds. Two rejuvenation levels are implemented into the supervisor: system level, which implies a complete reboot of the whole software system, and service level, which instead allows to restart only a single service.

Although a consistent number of measurement-based analysis deal with resource exhaustion, only a few of them deal with performance degradation. Software aging which manifests as a progressive loss of performance has been deeply studied for OLTP servers [64] and for the Apache Web Server [65, 4].

In [64], Gross et al. applied pattern recognition methods 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 in [65, 4], Trivedi et al., 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 in [9]. 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 patterns, as a direct consequence of rejuvenation mechanisms built-in in the Apache web Server. Extending trend detection and estimation techniques used for the previous resources, authors determined a 7.714 Kb/hr depletion trend for used swap space.

Software Aging in a SOAP-based server was analyzed in [66]. A SOAP-based web server running on top of a Java Virtual Machine, has been stressed with different workload distribution. For each considered distribution, throughput loss and memory depletion was graphically observed, thus proving the presence of software aging, even if it is not clearly understandable whether aging phenomena are due to the SOAP server or to the underlying virtual machine.

An analysis addressing the impact of workload parameters on aging trends has been presented in [12]. In this paper, 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 have been performed with different level of the three workload parameters; effects of single and combined workload parameters on the output variable (memory used) have been evaluated through Analysis of Variance (ANOVA) methods. A closed-loop software rejuvenation agent has then been implemented.

Finally, in [13], Malek et al., propose 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.

1.5 Comparison

This section is intended to provide a qualitative comparison of the dependability evaluation techniques and methodologies discussed in this chapter. The comparison is performed along the dimension described in section 1.3.

The main goal of such comparison is to provide a simple and intuitive framework for the choice of the right methodologies and techniques to assess the dependability of a software system, according to dependability requirements and other constraints such as the cost of the analysis and the time available to perform it. In the rest of this section, we will first classify dependability evaluation techniques and methodologies on single dimensions; then we will classify these techniques and methodologies in the following 2-dimensions spaces:

1) Cost vs. Detail;

2) Cost vs. Management;

3) Detail vs. Coverage.

1.5.1 The *Cost* Dimension

Among the discussed evaluation techniques and methodologies, *Robustness Testing* and *Dependability Benchmarking* require a low effort in order to be performed. In particular, Robustness Testing appear to be cheaper than Dependability Benchmarking, since it does not require to define fault injection profiles and implement tools for fault injection. The time required to perform such evaluations depends on the number of defined injection profiles or on the number of functions exposed on component interfaces to test, but is generally very little when compared with the time required to perform a *FFDA* or a *Software Aging analysis*.

On the other hand, *FFDA* appears to be the most expensive technique, since it generally requires a long period of observations. In particular, when coping with systems which have a relatively high time to failures, several months or years of observation may be required to obtain enough data to characterize system's failure behavior. *Software Aging Analysis*, instead, may be performed in lower times since an observation period ranging from 5 to 50 days allow to characterize faithfully resource depletion and performance degradation dynamics.

Finally, Error Injection cost should be placed in the middle of our evaluation scale, since while on the one hand it makes injection experiments easier, on the other hand it requires a non-negligible amount of field data in order to map real faults on injected errors.

1.5.2 The *Detail* Dimension

Although the level of detail reached by a dependability evaluation is usually dependent on the quality and effectiveness of the monitoring and analysis tools employed, and therefore it has an impact also on the cost of the evaluation, in this section the level of detail achieved with the various dependability evaluation techniques discussed in this chapter will be compared based on the experience made with scientific literature.

Injection based techniques such as *Dependability Benchmarking* and *Error Injection* usually achieve a Medium level of detail: once the injection campaign has been conducted, results are capable of giving a complete picture of the weaknesses of the component under test, without giving a detailed description of such weaknesses. *Robustness Testing* instead reaches an higher detail level, since it maps software failures to erroneous input values supplied, thus allowing to infer some information about the dependability behavior of the component under test, even if its internal details are not known.

Theoretically there is no lower or upper limit on the detail level of *FFDAs*: it depends on many parameters related to system monitoring, data filtering and data analysis. For instance, an *FFDA* for the JVM may achieve an higher level of detail if thread stack traces are periodically collected, whereas a very low detail level is achieved if only crash dumps are collected, since information contained in crash dump is often

not sufficient to assess the root cause of a failure.

Finally, the level of detailed achieved by *Software Aging Analysis* varies from Low to Medium: analyses taking into account only system resources achieve a very low detail level, whereas analyses taking into account also the impact of the workload on aging trend may reach higher level of details. However, this kind of analysis is not usually capable of pinpoint the root cause of the aging phenomena, i.e.: the aging related bug.

1.5.3 The *Coverage* Dimension

As far coverage is concerned, dependability evaluation techniques which monitor the system during its operational phase achieve higher levels of coverage. Therefore one can expect *FFDA* and *Software Aging Analysis* to be the best dependability evaluation techniques from a coverage perspective.

Instead, coverage levels of Dependability Benchmarks are quite lower. Indeed, Dependability Benchmarks are usually designed to test a single feature or component of a software system (e.g.: benchmarking the dependability of Linux against faulty drivers). The same consideration apply also to *Error Injection*: in this case the coverage is worsened by the fact that there is no real fault injected, but errors which are representative of real faults.

Further on lower levels of coverage are achieved by *Robustness Testing*, which is limited only to the interfaces of the system or component under test and does not take

into account its internal state.

1.5.4 The *Management* Dimension

The last dimension considered in this comparison is the capability, offered by the various techniques and methodologies described in this chapter, to be managed by people who want to assess the dependability of a system. Clearly, techniques dealing with pre-deployment phases, such as *Robustness Testing* and *Dependability Benchmarking*, achieve the highest levels in terms of manageability of experiments. A good level of manageability is achieved also by *Error Injection*.

On the other hand, techniques concerned with the operational phases achieve lower level of manageability. In particular, little or nothing can be done with *FFDA*, expect for imposing ad-hoc generated artificial workloads. Software Aging Analysis experiments are a little more manageable, since it is possible to employ techniques such as Design Of Experiments in order to extract useful insights about the relationships between aging trends and workload. DOE has been used in [12] and a similar approach is proposed in the fourth chapter of this thesis.

1.5.5 Cost versus Detail

As depicted in figure 1.10, injection-based techniques show a good trade-off between cost and detail. In particular, *Robustness Testing* is capable of giving the higher level of detail with the lower cost, whereas *Error Injection* has slightly higher costs with a

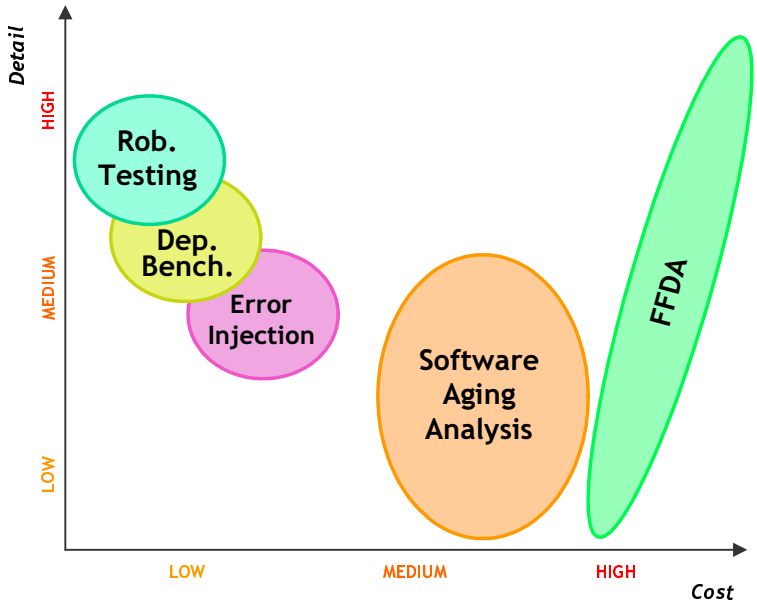

Figure 1.10: Comparison of cost and detail of different dependability evaluation techniques and methodologies

lower level of detail.

As far as *FFDA* is concerned, its costs are generally high. However the achieved level of detail may be very high, even if an increasing detail level is associated with an increasing cost. *Software aging analyses* usually allow to achieve a Low or Medium detail level with a Medium cost.

Summarizing, if there is no concern about Coverage and there are some restrictions on Cost, injection-based techniques are the most suitable dependability evaluation techniques.

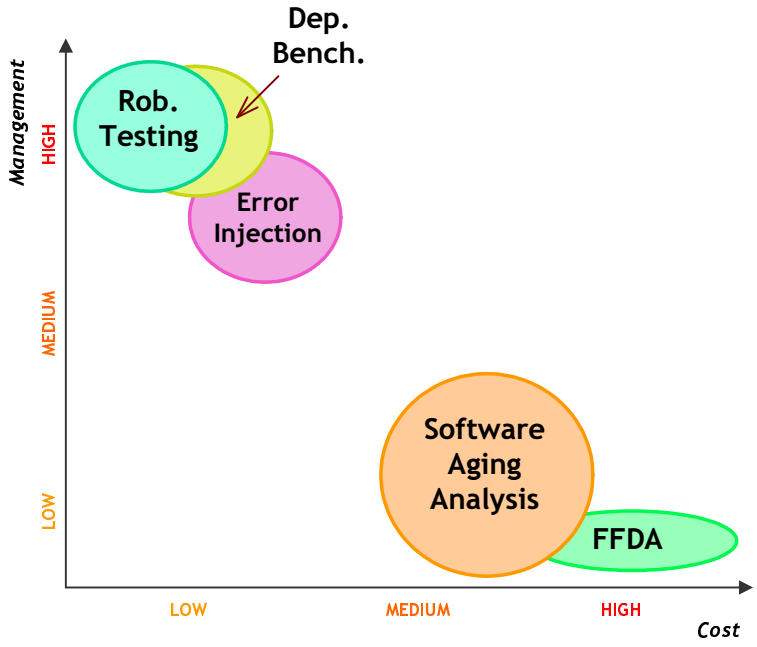

Figure 1.11: Comparison of cost and management of different dependability evaluation techniques and methodologies

1.5.6 Cost versus Management

When considering Management against cost, *FFDA* becomes the worst possible choice (as depicted in figure 1.11), since it has a very Low management capability in spite of an high cost. On the other hand, *Software Aging Analysis* achieves acceptable management capabilities. Again, injection-based techniques are the most manageable ones.

Summarizing, when it is required to manage dependability evaluation experiments, *FFDA* has to be discarded; instead it is still possible to keep an acceptable level of manageability in software aging studies; injection-based techniques achieve the best trade-off between cost and management.

1.5.7 Detail versus Coverage

In this case, showed in figure 1.12 injection-based techniques do not exhibit a good trade-off, since they achieve a very low coverage. For instance, given a component whose dependability has to be evaluated, *Robustness Testing* measures only dependability of a part of the interfaces exposed by such exponent, without taking into account at all its internal state: in general it is not possible to think that the outcome of a function exposed by an interface is independent by the internal state of the component.

In contrast, *FFDA* and *Software Aging Analysis* showed better trade offs. In particular *FFDA* ranges from a low detail level with a medium coverage to a high detail level with a high coverage. Similarly, coverage *Software Aging Analysis* is above the average as well as its detail level. Its coverage is lower than *FFDA*'s one, since this kind of analysis is aimed at detecting only a particular type of software pathological behavior, whereas FFDA is aimed at determining all the failures modes of a software system.

Summarizing, when there is concern about coverage and there are no budget or time restrictions, it is better to avoid using injection based techniques, and switch to deeper, longer, kinds of evaluations, such as *Software Aging Analysis* and *FFDA*.

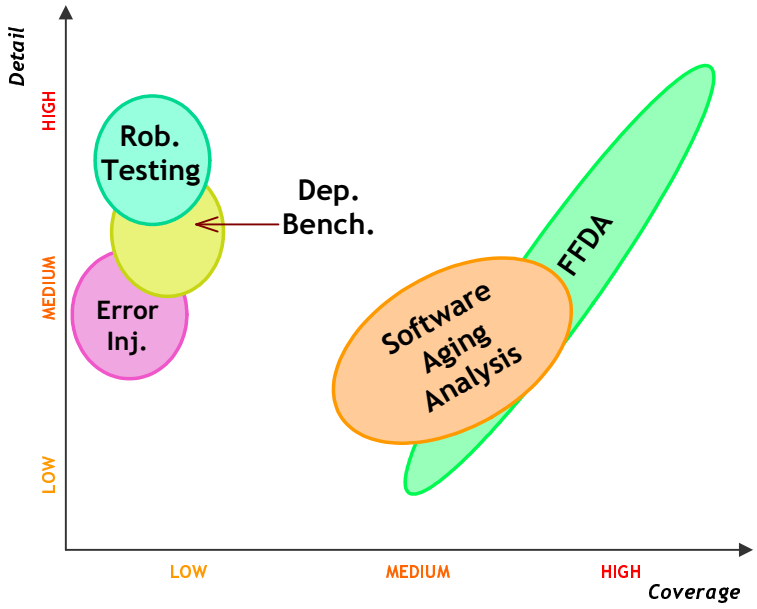

Figure 1.12: Comparison of coverage and detail of different dependability evaluation techniques and methodologies

"Chi vo' mettere 'o pede
'ncopp 'a ttutt' 'e pprete,
nun arriva maje"
Who wants to make a step on
every stone, does never arrive

Neapolitan Proverb

Chapter 2

Thesis Contributions

This chapter describes previous relevant work dealing with the assessment of the dependability of OTS items (a generalization of the more common term "COTS", Commercial Off-The-Shelf, which refers to any code that is not developed specifically for a new system).
OTS items are software components ready-made, and available for sale or license to the general public,and integrable into larger software systems. These items are widely employed in today's software systems. Moreover, they are starting to be employed also in business critical, mission-critical, and safety-critical software systems.
Many papers deal with the dependability of OTS items, using the techniques and methodologies described in the previous chapter. However, scientific literature currently lacks a "standard" approach to evaluate the dependability of an OTS item.
The goal of this thesis is to propose a general approach to evaluate the dependability, from a software aging perspective, of a software system employing one or more OTS items. In this chapter we first discuss some examples of dependability evaluation of OTS items, pointing out the lack in scientific literature of a general approach to evaluate software aging in OTS items. Then we will introduce the Java Virtual Machine, which will be used as a case study throughout the thesis. In particular, we will describe its architecture and discuss previous relevant work dealing with its dependability issues.

2.1 Evaluating the dependability of OTS items

Among the various kinds of OTS items, dependability issues have been deeply studied for Operating Systems, but also for Database Management Systems and transaction-based server systems. More recently, the range of OTS items whose dependability has been studied, has widely broadened, including Web Servers, File Servers, Network Protocol Stacks, etc. etc.

Operating Systems

M. Kaâniche and C. Simache in [38] present an FFDA regarding the availability of Unix Systems in a distributed environment. This work is focused on the identification of machine reboots and the evaluation of statistical measures characterizing the distribution of reboots and the availability of the monitored workstations. Reboots have been identified by implementing a pattern recognition algorithm to extract information about system reboots (and time to recovery), from system log files, whereas the availability for each node has been calculated by taking into account the first and last event of the reboot sequence.

Koopman et al. in [53] performed a robustness testing analysis of six different Win32-based operating systems, using the Ballista testing methodology [50]. For testing an OS, this methodology involves selecting a set of functions and system calls to test, with each module under test being exercised until a desired portion of the API is tested. Non-robust responses are classified according to the CRASH scale described

in the previous chapter. Using this methodology, authors were capable of find robustness vulnerabilities in these Operating Systems, and compare their robustness level with the one exhibited by the Linux operating system.

On-Line Transaction Processing Servers (OLTP)

OLTP servers, such as Database Management Systems, represent a category of OTS items which is largely employed also in critical software systems. Madeira et al. in [27], presented a dependability benchmark for OLTP servers, aimed at choosing the best choice for a typical OLTP application, considering both performance and dependability aspects. Results of the dependability benchmark allowed the authors to rank different OLTP server according to criteria such as baseline performance, performance with faults, and availability.

Web Servers

Several papers addressed the characterization of the dependability of this kind of OTS item from a software aging perspective. In [4], Trivedi et al. studied aging phenomena inside the Apache Web Server. In this paper resource data collected from a web sever subjected to a synthetic load are analyzed, in order to provide a better understanding of web server aging phenomena, leading to a more appropriate scheduling of rejuvenation actions. The development of response time and memory usage was examined in detail, revealing the influence of settings related to both the Linux OS, as well as the Apache Web Server itself, on the aging phenomena. However, although authors

made a deep effort toward the modeling of seasonal patterns, the performed analysis did not take into account the relationships between workload and aging phenomena. These relationships were taken into account by Matias and Filho in [12]. In this paper the Design Of Experiments (DOE) [67] technique was adopted to characterize aging phenomena. Authors adopted this technique in order to determine the aging factors and their degrees of influence on the web server. DOE is a structured, organized method for determining the relationship between factors affecting a process and the output of that process. The selection of factors and their respective levels were partially based on previous work in the field ([11, 65]). Factors employed were: *Page Size*, *Page type*, *Request rate*; two levels were defined for each factor, accounting for respectively 50% and 90% of the capacity of the web server, leading to a total number of 8 treatments. Authors analyzed memory usage for each treatment and solved a regression model to infer both main effects and combined effects of the three factors. Results of such treatments showed that: **i)** a premature aging of the web server was always occurring regardless of the *page size* factor, and **ii)** the *page size* and *page type* factors were responsible for over 99% of memory variation in the server process. The problem of the selection of the most relevant workload parameters has been addressed for the first time by Malek et al. in [13]. In this paper several variable selection techniques (Forward Selection, Backward Elimination, and Probabilistic Wrappers) have been evaluated on a data set related to Apache Web Server's workload parameters.

Moreover, sensitivity analysis has been employed in order to assess how sensitive are response variables to changes in the assumptions.

2.1.1 Remarks

Due to their nature, OTS items allow to dramatically reduce software development costs: companies are thus seeking the use of these items as an attractive way to shorten time-to-market, hence increasing business opportunities.

Nowadays, for these reasons, OTS items are are increasingly used in application areas with high dependability requirements. Nevertheless, the adoption of OTS items significantly hampers the dependability and safety of the whole system. Indeed they are generally developed without taking into account the specific system operational profile on which they will be integrated. Moreover, they often lack proper testing. Furthermore, interactions between different OTS items may have unpredictable effects, which are not easy to foresee and may lead to the failure of the whole software system.

Given this scenario, it becomes a priority to develop methodologies and techniques to evaluate, from different perspectives, the dependability of OTS items and the influence that their integration into larger systems may on system dependability.

As regards Dependability Benchmarking, as well as robustness testing, significant efforts have been made in recent years toward dependability assessment techniques suitable for several classes of OTS items. In particular, the DBench project [42], funded

under the fifth Framework of the Information Society Technologies Programme, defined standardized dependability benchmarks for general purpose Operating Systems, OLTP environments, Real-Time Kernels, and Engine Control Applications.

On the other hand, much efforts have still to be done in order to achieve a unified approach to evaluate OTS items' dependability through FFDA or Software Aging analysis. As it has been shown in the previous chapter, these techniques are of vital importance in the assessment of the dependability of software systems; unlike other techniques, they have the capability of revealing a larger number of insights about the behavior of the analyzed component or system.

In particular, software aging analysis allow to highlight the presence of aging-related bugs, and to assess, in a relatively simple way, rejuvenation techniques to treat these faults.

2.2 Contributions

In this dissertation we focus on the study of Software Aging phenomena and explore the possibility of assessing a measurement-based methodology capable of characterize thoroughly the dependability of OTS items from a Software Aging perspective.

Although many work, described in the previous chapter, already coped with software aging and rejuvenation, there are still some open issues, especially in the field of OTS-based software systems. All measurement-based work consider aging introduced by long-running applications, such as web servers and DBMS servers, as measured at

the operating system level, neglecting the contribution of intermediate layers, such as middleware, virtual machines, and, more in general, third-party OTS items. These layers might worse resource exhaustion dynamics or become an additional source of aging. In order to develop a methodology to analyze Software Aging in OTS items two challenging issues, addressed in this dissertation, arise:

First, **new methods are needed to isolate the contribution of each intermediate layer to aging trends**, i.e., the one introduced by the presence of a virtual machine from the one which is due to the application running on top of it.

Second, **since there is a strict relationship between workloads and aging trends, it is crucial to investigate how these are affected by changes in the applied workload**. Although several work addressed the relationships between workload and aging trends, the selection of workload parameters and the assessment of their effect on aging trends have been partially addressed only recently in [12] and [13]. The former work addressed the evaluation of the effects of workload parameters on the response variable, whereas the latter addressed the selection of the most relevant workload parameters. However [12] encompasses only controllable workload factors, whereas workload parameters to be monitored in OTS-based systems are often uncontrollable. Furthermore, variable selection techniques presented in [13] does not take into account influence of workload parameters on aging trends.

2.3 The JVM as a case study

Operating Systems such as Windows or Linux, Database Management Systems such as Oracle or Microsoft SQL Server, Middleware platforms such as CORBA or DCOM, or Application Servers such as Bea Weblogic or Red Hat JBoss, are only some examples of OTS items. Among the extremely wide range of available OTS items, we chose Virtual Machines, and in particular the Java Virtual Machine (JVM), as the case study to employ throughout the dissertation.

The JVM has been choosen for the following reasons:

1. Virtual Machines represent a relevant example of the above mentioned *intermediate layers*, since they provide a virtualization of a complete execution environment;

2. The JVM is currently widely employed in a wide range of applications, 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 [68];

3. There is a growing interest in the scientific and industrial community toward the employment of Java in safety and mission critical scenarios, as proved by a recent issue of a Java Specification Request (JSR-302 [69]), which aims at defining those capabilities needed to use Java technologies in safety critical

applications;

4. There is a lack of research about the characterization of the dependability of the JVM; moreover, as regards Software Aging, a recent paper [66], discussed in the previous chapter, highlighted the presence of aging phenomena in a Java-based SOAP server.

2.3.1 Research on JVM dependability

During the last decade, research on the JVM has been progressed along two directions: performance related issues and, more recently, dependability issues. Performance aspects of the Java Virtual Machine have been extensively explored in [70, 71, 72]. Even though these studies did not address dependability issues specifically, they represent fundamental milestones for the pursuit of a deep analysis of JVM runtime behavior, in terms of the definition of workload profiles and instrumentation strategies, which have to be taken into account.

As shown in figure 2.1, only a small percentage of the research work on the JVM (about 6%) coped with its dependabilty and fault tolerance issues.

Memory errors effects in the JVM have been studied in [73, 74]. Authors investigated trade-offs between performance of the detection module (in terms of errors detected), heap space occupancy, and impact on the overall application's performance. However, these studies focused on issues related to memory errors, such as bit-flip or stuck-at

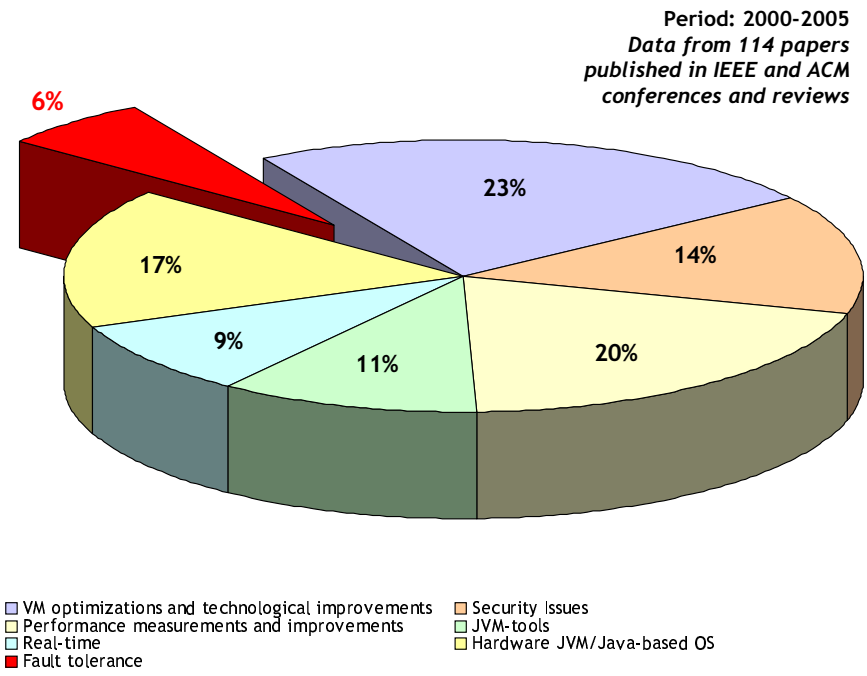

Figure 2.1: Classification by topic of the scientific literature dealing with the Java Virtual Machine

errors, which are typical examples of hardware faults. Protection against such kind of faults may be implemented in hardware in several ways (e.g.: by using ECC memories), and implementing checksum-based protection schemes inside the JVM makes sense only when it is deployed on resource-constrained devices.

In [75], Alvisi et al. conducted an interesting study on how to apply state machine replication to the Java Virtual Machine. The work focused on Sun Hotspot VM 1.2, applying restrictions to the execution environment in order to remove sources of non-determinism, and then implemented a primary-backup replication schema. In [76] a similar approach has been used and it has been applied to the Jikes Research virtual

machine.

Both these approaches are based on *Hypervisor-based fault tolerance* [77], a layer of software that implements a Virtual Machine having the same instruction-set architecture as the machine on which it executes. Therefore, although it is possible to find some important attempts aimed at augment fault tolerance in the JVM, there is still a lack of research about its dependability issues. In order to use the JVM in scenarios with stringent dependability requirements, it is crucial to identify dependability bottlenecks in order to implement effective strategies to improve the overall dependability of this virtual machine.

2.3.2 The Architecture of the JVM

The specification for the Java Virtual Machine [78] has been implemented in different ways by many vendors. JVM implementations differ from one another not only with regard to the interface to the operating system but also in the implementation of its various components. In this thesis, focus is on the Sun Hotspot 1.5 Virtual Machine. In order to understand the internal behavior of the Sun Hotspot VM, its source code has been analyzed. The resulting schema is depicted in figure 2.2. This analysis has become a necessity due the lack of appropriate technical documentation concerning the implementation of VM components. As depicted in figure 2.2, the analyzed JVM is composed by four main components: **i)** The *Execution Unit*, which includes the core components of the JVM needed for executing java programs, for instance the bytecode

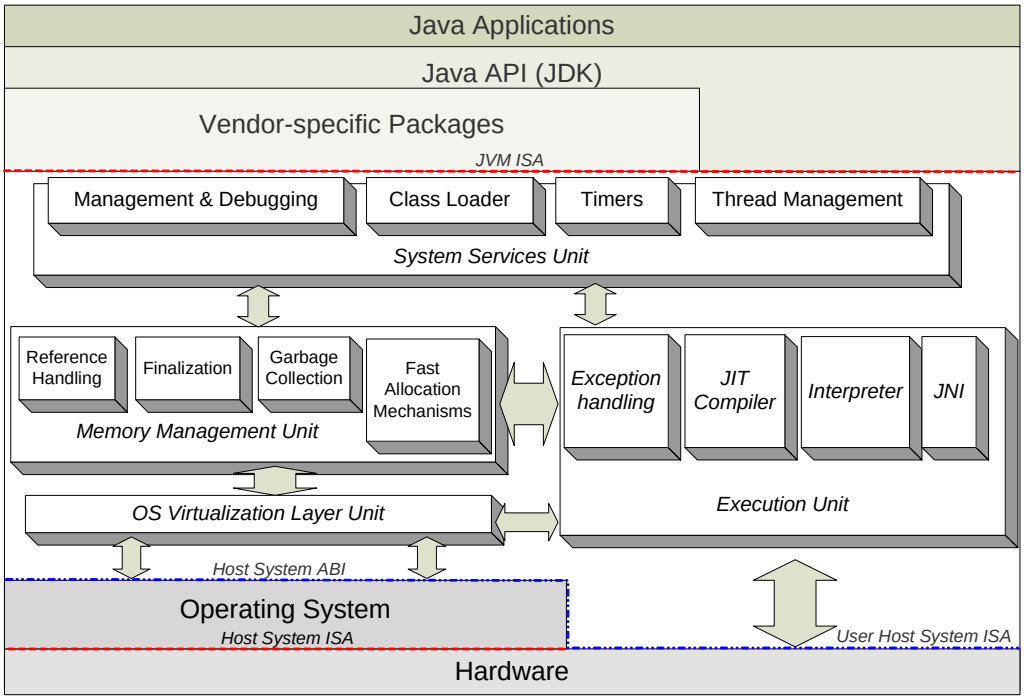

Figure 2.2: Architectural Model of the Java Virtual Machine

interpreter and the Java Native Interface (JNI); **ii)** The *Memory Management Unit*, which is in charge of managing 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 OS abstraction layer, which provides a platform-independent abstraction of the host system's ABI. By looking inside each of the above mentioned components it's possible to isolate several subcomponents. In the following we give a brief description of these subcomponents.

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 whole methods translating them into native code. Methods to JIT-compile are automatically selected by the JVM by exploiting the code locality principle. The JVM is capable of identify "hotspots" in the application, i.e. the pieces of code which are executed more frequently. Finally, native instructions need no translation: they are dynamically loaded, linked and executed by the *Java Native Interface (JNI)*. Furthermore, the *Exception Handler* handles 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, as depicted in figure 2.3. Even if the maximum size of these generations is fixed at JVM startup, their actual dimension depend upon the memory requirements of the application, since the JVM has the capability of dynamically grow or shrink the size

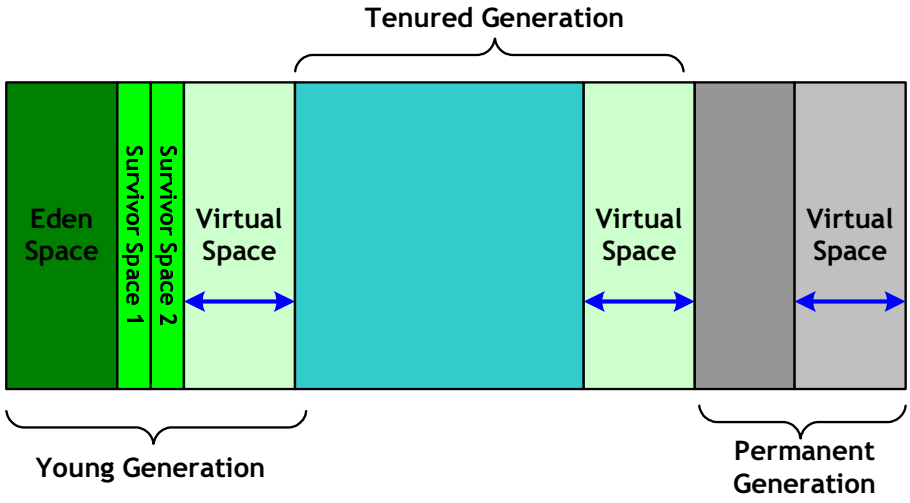

Figure 2.3: Internal organisation of the JVM heap

of each generation.

The young and the tenured generation are used to store Java objects and are therefore subject to garbage collection; the permanent generation, instead, is mainly used to store java classes loaded into the JVM: objects in this area are not subject to garbage collection. Whenever a new object is allocated, it is stored in the eden space. Objects which survive to 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 manage to reach the tenured generation. In this way the performance of the JVM is consistently improved, since the frequency of full garbage collection cycles, which involve both the young an the tenured generation, is reduced.

The Sun Hotspot JVM provide 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 and a compact, mark and sweep algorithm on the tenured generation. Moreover, it is possible to use multi-threaded garbage collectors to improve either program completion time or application throughput. Furthermore, *Fast Allocation Mechanism* 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)* [79]. 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 the management of the JVM.

The OS abstraction layer

These 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.

2.4 Thesis Structure

So far, we discussed previous relevant work regarding the dependability evaluation of OTS items, and outlined some research open issues which will be addressed by the studies presented in this dissertation. The rest of the thesis is organized as follows:

The next chapter discusses results of an analysis of JVM failure reports extracted from publicly available Bug Databases; from this analysis it arises that there are actual clues of aging phenomena in the JVM as a consistent number of failures occurred when relevant workloads were applied with a daily or weekly frequency Characterization.

Chapter 4 describes a methodology to study software aging phenomena in an OTS item and their relationships with workload parameters, and discusses the result of a massive experimental campaign performed on the JVM.

Chapter 5 further explores aging phenomena into the JVM taking into account also resources as measured at the operating systems level.

Finally, chapter 5.6 concludes the thesis and explores solutions to make the JVM resilient to aging-related bugs.

"Chi fraveca e sfraveca nun perde mai tiempo."
Who does and undoes never loses time.

Neapolitan Proverb

Chapter 3

Failure Behavior Characterization through Failure Reports Analysis

Bug databases are a precious source of information related to reliability and robustness of software systems. Despite of their qualitative nature, they represent a convenient way to perform a preliminary characterization of the dependability behavior of a software system. Such characterization may then be used in order to i) extract measures about its failure behavior and ii) identify the main dependability bottlenecks thus narrowing directions for more detailed analyses.
The work described in this chapter reports the results of a preliminary analysis of the dependability of the JVM, gathering information from failure reports provided in bug databases. Results of this analysis clearly indicate that much more efforts have still to be done in order to improve the dependability of the JVM. In particular, the conducted analysis revealed that aging-related bugs may be addressed as the root cause of a non-negligible percentage of failures showing a non-deterministic behavior.

3.1 The Importance of Failure Reports

As already discussed in the first chapter, field failure data analysis represents the most suitable technique when dealing with the characterization of the failure behavior of a system. Unfortunately, the cost of FFDA is usually very high. Data have to be collected for a long period of time and often from a consistent number of workstation

in order to obtain a significant number of failures to analyze.

Bug reports, instead, may be regarded as a ready-to-use source of failure data available to extract useful insights about the failure modes of a system. Despite of their prominently qualitative nature, these reports should be considered well-founded, since they are submitted and then evaluated by skilled people, which usually are expert users or even developer of the system itself.

Previous works such as [55] and [80] showed that user-submitted data can represent a valuable starting point in order to study the dependability of a system. In particular, Strigini et al. in [80], were able to prove how design diversity with OTS items may improve system reliability, by analyzing 181 bug reports for two commercial database servers (Oracle and Microsoft SQL Server) and two open-source database servers (PostgreSQL and Interbase); the found that only 4 out the 181 bugs caused identical failures in different server.

A typical flaw of bug reports is that they are often elusive or incomplete, thus not allowing to completely characterize the failure they describe. For this reason, even if bug databases contain thousands of reports only a small percentage of them can be actually considered for the analysis. A careful filtering of such information is therefore required in order to extract meaningful information about dependability features.

3.2 Data selection and classification approach

3.2.1 Data Selection and Filtering

Before starting a failure analysis based on bug reports, it is important to select proper data sources and define criteria to filter out those reports which are not suitable for the analysis. Reports contained in the selected data sources will be filtered according to the following criteria:

- The bug has been marked as *Fixed*. These reports are not useful and misleading, since they are related to faulty conditions already fixed.

- The reported failure is related to a version of the software still under development or testing. Since our research is aimed to discover information about failures of operational systems, these must be dropped.

- The report is elusive or it does not contain enough information to characterize the failure.

- The failure report is related to a fault or error originated in a component different from the observed one. This means that the root cause of the failure (i.e.: the fault) is located outside the observed component, whereas the goal of this study is to analyze failures which are due to faults originated in the component itself.

3.2.2 Data Classification

Data contained in bug reports allow to classify failures according to several dimension, since they usually contain (completely or not) information such as the configuration of the system and its environment when the failure occurred, the time elapsed from system startup (or even the frequency of the failure), a detailed description of the failure (including crash dumps and stack traces), and sometimes directions to reproduce the failure.

In our approach, we will employ the following failure classification criteria:

- ***Failure Manifestation*** - Failures are classified according to their manifestations (i.e: the message printed on the console). Four failure manifestation types were defined:

  - *Error Message* - The failure was reported to the user through a message generate either from the observed component or from another component interacting with it;

  - *Hang/Deadlock* - The component did not crash, but it stopped executing.

  - *Silent Crash* - The component crashed silently, without printing any error message.

  - *Computation Error* - Results obtained were different from the expected ones.

- ***Failure Source*** - By analyzing information attached to failure reports it is possible to pinpoint the subcomponent(s) where the failure source is located.

Table 3.1: Categories for the reliance by the environment

Dependency	Description	Types of Software Faults
Platform-Independent	The failure occurs regardless of the environment	Bohrbugs, Mandelbugs (rarely)
OS-dependent	The failure occurs only on a specific Operating System	Heisenbugs, Mandelbugs
Platform-Dependent	The failure occurs only on a specific Hardware Platform	Heisenbugs, Mandelbugs (rarely)
OS&Platform-Dependent	The failure occurs only on a specific combination of Hardware Platform and Operating System	Heisenbugs

Table 3.2: Workload levels

CPU-Bound	The workload consistently stresses CPU, performing significant floating-point calculations and/or spanning several concurrent tasks, thus causing significant synchronization activity
	Examples: Application Servers, Parallel Systems
Memory-Bound	The workload requires a significant amount of memory. Moreover, memory management operations occurr with a very high frequency
	Examples: Scientific applications, OLAP applications
IO-Bound	The workload mainly performs Input-Output operations (on files, databases, and network connections)
	Examples: Web Servers, Database Management Servers
Common	No particular workload is imposed.
	Examples: web browsers, e-mail clients, utilities

- ***Severity*** - A failure is defined *Catastrophic* if it leads to the crash of the whole system or *non-Catastrophic* if the system is still capable of executing its functions (or a part of them) despite of the failure.

- ***Environment*** - Failure reports were classified according to the reliance by the environment in which the observed component is integrated. Four categories (described in table 3.1) were defined. In order to collect such information, the bug report must contain data about several failures on different environments.

- ***Workload*** - Failure reports were classified according to the workload imposed on the component when the failure was reported. Table 3.2 shows the qualitative

Table 3.3: Failure Timing categories

REGULAR	The failure occurs reguarly despite of the sequence of operations performed (clear proof of the presence of a Bohrbug)
STARTUP	The failure occurs during the initialization of the observed component
HOURLY	The failure occurs within one hour of execution of the observed component
DAILY	The failure occurs on average once a day
WEEKLY	The failure occurs on average once a week

workload levels defined.

- ***Failure Frequency*** - Failure reports were classified according to the frequency of failure occurrences. This information may be directly derived when the TTF or the failure frequency is specified in the report, or indirectly from the description of the failure included in the bug report. Table 3.3 describes the categories defined.

3.3 JVM Failure Reports Analysis

Java Virtual Machine failure reports, extracted from publicly available bug database, have been classified according to the approach outlined in the previous section.

This study aims at performing a preliminary analysis of the dependability of the Java Virtual Machine, and it is the first work in scientific literature addressing this topic. Failures are analyzed with respect to their manifestation, the host system environment, JVM components, their frequency and the kind of workload imposed.

The analysis of extracted failure reports allowed us to give useful insights into **i)** the

nature of the failure of the JVM, **ii)** the sources of the error, and **iii)** the characterization of the failures with regards to workload and frequency.

3.3.1 Data Set

Bug databases are a precious source of information related to reliability and robustness of software systems: development software faults always occur when buggy code is executed. Some kinds of bugs, namely Heisenbugs, Mandelbugs, are particularly likely to elude all testing phases, since they usually disappear or alter their characteristics when they are inspected. Failure data analyzed in this study are extracted by *Sun*[1] and *Jikes*[2]. Other JVM implementations, such as *Kaffe*, *J9*, and *JRockit* had no public bug databases or very poor ones. Data were collected from these Bug Databases between June 2005 and June 2006. Among thousands of submissions related to the whole Java Platform, 698 bug submissions related to JVM failures were selected and analyzed. This set was further refined by excluding submissions which met the criteria described in section 3.2.1.

Among the initially selected bug reports, 147 (29 from Jikes Database, 118 from Sun) were selected; 191 distinct failures were described in these reports.

Each submission reports the environment on which the JVM was running,the configuration of the virtual machine (i.e.: heap configuration,JIT compiler used) and stack traces. Many failure reports also contained a detailed description of the source of the

[1]Sun Hotspot Bug Database: *http://bugs.sun.com*
[2]Jikes RVM bug database: *http://jikesrvm.org*

Table 3.4: Structure of a bug report taken from the Sun Hotspot Bug Database

Section	*Description*
Bug ID	Unique ID for the Bug Report
Votes	Number of votes given by users of the Sun Developer Network to urge developers to fix the bug
Synopsis	A brief description of the Bug
Category	Pinpoints the component of the JVM where the bug is located
Reported Against	The version(s) of the JVM where the bug was found
Fixed in	The version of the JVM in which the fix for the bug has been included
State	Current state of the bug (only bugs in progress have been selected)
Submit Date	The date in which the bug report was first submitted
Reproducibility	Tells whether the bug is reproducible or not. Sometimes this section contains detailed instruction about how to reproduce the failure or even source code fragments
Description	The "core" of the bug report. This section contains the greatest part of the data used to characterize the failure. Usually users submit a JVM crash dump, which includes: a description of the error reported by the JVM (e.g.: OutOfMemoryException, EXCEPTION_ACCESS_VIOLATION or SIGSEGV), the name, the address and the stack trace of the currently executing thread, the operation the JVM was performing when the failure occurred, a list of all the threads running in the address space of the JVM, a summary of JVM heap state, the configuration of the JVM, the host Operating System and Hardware platform, and the time (in second) elapsed from JVM startup.
Work Around	A solution to avoid the failure without having to install a patch or upgrade to a different version of the JVM. For instance, if the JIT compilation of a particular methods causes a failure of the JVM, the workaround usually suggests to tell the JVM to skip the compilation of that method by a proper command-line switch
Evaluation	An evaluation of the bug given by a developer or JVM specialist
Comments	User-submitted comments about the bug (sometimes contains very useful information, such as other crash dumps related to similar failures of the JVM).

failure (given by specialists in the evaluation section of the report itself) and information related to the frequency of the failure and its reproducibility. The structure of a bug report (taken from the Sun Hotspot Bug Database) is shown in table 3.4.

Failure sources where classified, according to the architectural view described in section 2.3.2, in the following way:

- *Execution Unit* - This category is further divided into *Shared Runtime*, *JIT Compilers*, *Interpreter* and *JNI* subcategories.

- *OS Virtualization Layer.*
- *Memory Management Unit* - This category is further divided into *Garbage Collector* and *Reference Handling* subcategories.
- *System Services Unit* - This category is further divided into *Thread Management*, *Class Loader* and *Monitoring* subcategories

It this worth noting that more than 20% of failures contained in selected reports were due to faulty conditions in more than one component, thus highlighting the presence of error propagation phenomena inside the JVM.

3.3.2 Results

This section discusses results obtained from the analysis of selected failure reports. The first part analyzes the distribution of failure manifestations and their relationship with the environment on which the JVM runs. The second part highlights the role of internal JVM components in reported failures, whereas the third part shades some light on the relationships between internal JVM components, failure frequency and workloads imposed on the JVM itself. To this aim, starting from the 147 selected bug reports (accounting for 191 failures), 108 failure reports (56.54%) were selected for frequency analysis, 114 failure reports (59.69%) for workload analysis and 101 submissions (68.71%) for environment dependency analysis.

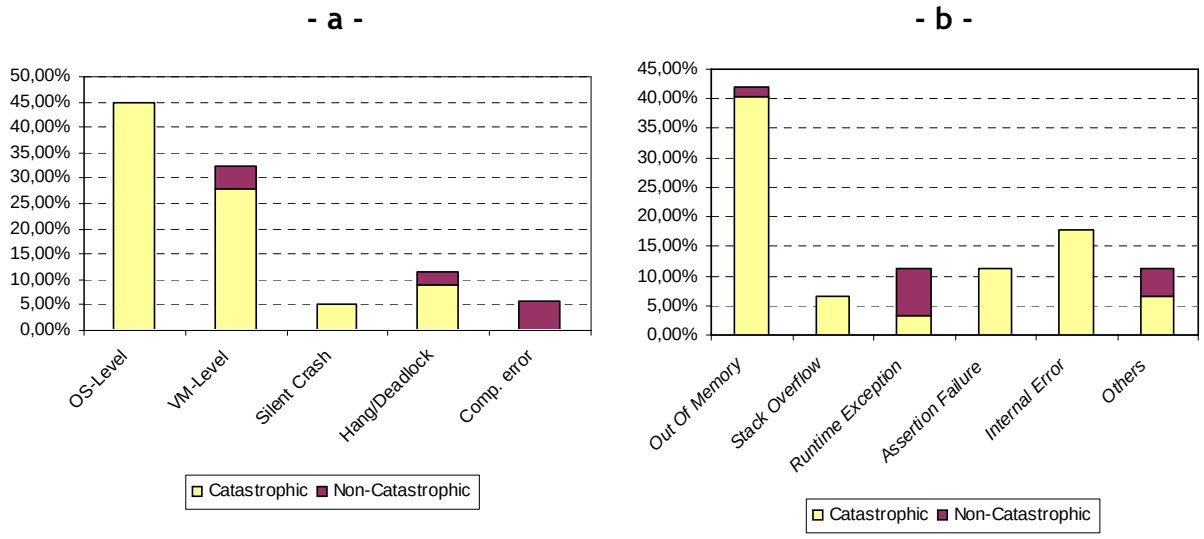

Figure 3.1: (a) Failure manifestations distribution (b) detailed view of VM-level failure manifestations. Computation errors were captured comparing the "Expected Output" against the "Actual Output" in the failure report.

Failure Manifestation Analysis

Figure 3.1-a depicts a bar chart of failure manifestations and severity. The most recurrent manifestation is an *OS-level* message (45.03%), followed by *VM-level* messages (32.46%), *hangs* or *deadlocks* (11.52%), *computation errors* (5.76%) and *silent crashes* (5.24%). Almost all failures lead to VM crash (86.06%). Only computation errors are usually non catastrophic. A quarter of hang/deadlocks are non catastrophic (not all threads of the virtual machine are blocked), whereas just a little part of VM-level manifestations (13.09%) does not lead to VM crash. Almost an half of the failures manifested as OS-level messages (i.e: `SIGBUS`, `SIGSEGV` or `ACCESS VIOLATION`). **Therefore built-in error detection mechanisms are not able to**

pinpoint a considerable percentage of faulty conditions inside the virtual machine. When one of such OS signals is raised, the JVM just dispatches the signal to a special `Signal Handler Thread`, which produces the failure report, and then exits.

VM-level manifestations appear when built-in error detection mechanisms pinpoint faulty conditions. In this cases an unchecked exception can be thrown from the virtual machine, thus giving a chance to handle the faulty condition in application code. With respect to VM-level manifestation, figure 3.1-b depicts a bar chart of the various error messages reported and their severity. Among VM-level manifestations, the most recurrent is `OutOfMemoryError` (44.07%). `InternalError` (15.25%), `RuntimeException` and `AssertionFailure` (11.86%), `StackOverflow` (6.78%) and others exceptions (10.17%) (e.g.: `NullPointerException`) are less frequently thrown from the JVM. Even if applications could handle these conditions through Java exception handling mechanism, we found that in the greater part of cases (with the exception of `RuntimeException` manifestations) the consequences were catastrophic. This indicates that **either the state of the virtual machine gets corrupted and no recovery action can be achieved anymore, or no recovery action was taken in Java applications**, since developers did not expect that these error conditions would be occurred.

To gain an understanding of the relationship between failures and the underlying

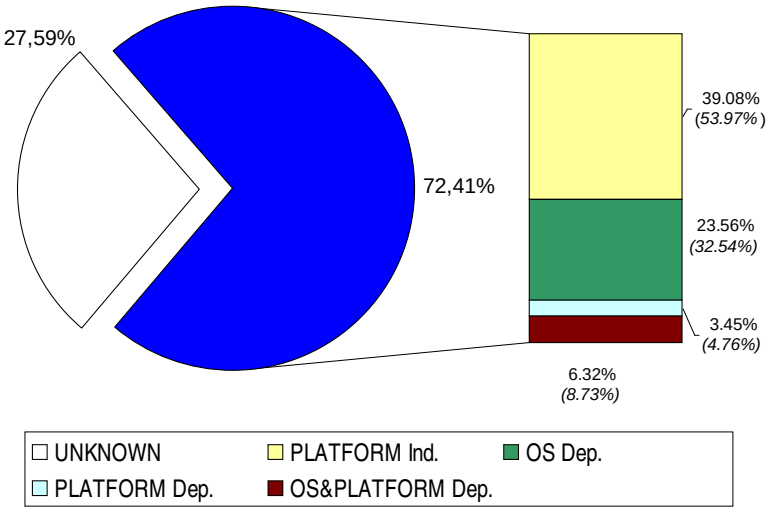

Figure 3.2: Relationships between failures and environment. In the bar reported on the right the value without parentheses represents the absolute percentage of environment-dependent (or independent) failures, whereas the value in parentheses represents the relative percentage of failures which have been classified as OS dependent.

environment, we analyzed the dependency of the reported failures on the Operating System and the Hardware Platform, as depicted in figure 3.2. In some cases it was not possible to distinguish whether the failures was environment-dependent or not. As far as the remaining cases (more than 70%) are concerned, we observed that 53.97% of the failures occurred regardless of the specific environment (OS and Hardware) on which the JVM was running. Thus these failures are not due the interactions between the JVM and the underlying platform, but they are *common-mode* failures which are likely to occur independently from the underlying Operating System or Hardware architecture.

Table 3.5: Detailed view of OS-dependent failures

OS	OS-DEP %	OS-IND %	UNKNOWN %
Windows	27,03%	38,71%	32,43%
Linux	40,00%	52,46%	12,00%
Solaris	19,35%	43,86%	40,32%

Even if only a little percentage of failures (4.76%) depended exclusively on the hardware platform, a more considerable percentage were dependent on the Operating System (32.54%) or both OS and Hardware (8.73%). These results indicate that there is a substantial dependency on the Operating System. Therefore, **it is not possible to claim that Java applications keep the same levels of dependability across different operating systems**.

To gain a more detailed view of the relationship between failures and operating systems, we further analyzed OS-dependent failures reported in Windows, Linux and Solaris. The results are described in table 3.5. Reported percentages are determined with respect to the total number of failures observed for each Operating System, thus avoiding bias in results. Even if the percentage of environment-independent failures has different values among different operating system, its absolute number (68) is the same on all considered OSes. These results indicate that the dependency on the underlying operating system seems to be more critical in Linux than in Windows and Solaris. However the fourth column of table 3.5 highlights that there is still a large margin of uncertainty (especially in Solaris), since in many cases it was not possible to distinguish whether the failure is OS-dependent or OS-independent.

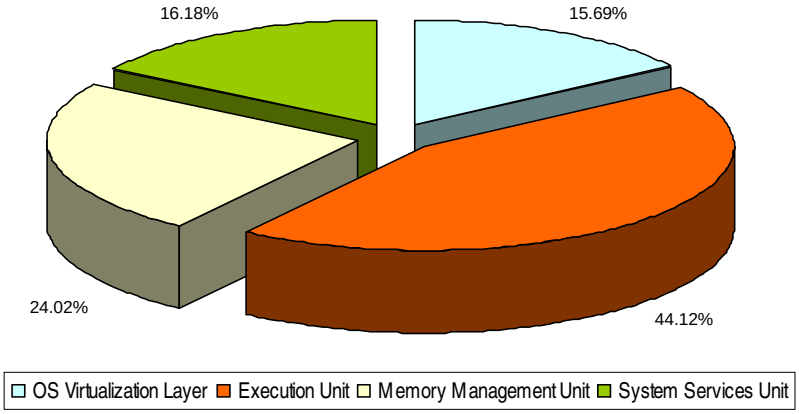

Figure 3.3: Distribution of failures with respect to JVM components

Failure Sources Analysis

By analyzing stack traces and core dumps attached to bug submissions it is possible to provide useful insights into failure sources, nailing JVM components in which errors were located. Often the source of a failure is located in more than one component. Among the reported failures, 22.47% of them were due to errors in more than one component.

The percentage of failures for each component of the JVM is depicted in figure 3.3. The greatest part of failures is due to the Execution unit. Moreover, looking at the details about the failures of the subcomponents, reported in table 3.6 it is straightforward that:

- The greatest part of failures in the memory management unit (72.73%) is due to the Garbage Collector.

Table 3.6: Detailed view of failure sources

Execution Unit	*Execution*	**19,33%**
	Optimizing JIT	**15,13%**
	JNI	**3,78%**
	Base JIT	**4,20%**
	Interpeter	**2,10%**
Memory Management Unit	*GC*	**16,81%**
	Ref Handler	**5,04%**
	Other Memory-Related	**1,26%**
System Services	*Thread Management*	**10,92%**
	Class Loader	**2,52%**
	Monitoring	**0,84%**
OS Virtualization Layer		**18,07%**

- Runtime support operations and optimized just-in-time compilation tasks cover the 77.36% of Execution unit failures.

- The greatest part of failures in the System Services Unit (76.41%) is due to the Thread Management sub-component.

By analyzing these results it is possible to argue that:

- Runtime support operations, such as method invocation, stack frame allocation and deallocation or exception handling, seem to be the most critical dependability bottleneck in the JVM.

- The optimizing JIT compiler, even if improves prominently the performance of Java applications, is one of the major sources of failures in the JVM; therefore Java developers have to cope with a trade-off between performance and reliability.

- The Garbage Collector still remains one of most error-prone components in the JVM. In particular low-pause or high-throughput garbage collectors seem to be critical for JVM reliability; therefore there is another trade-off between the performance of the collector and its reliability.

- Also the OS Virtualization layer has a deep impact on the dependability of the JVM. In particular this component is responsible for 15.91% of Solaris failures, 14.29% of Windows Failures and 24.21% of Linux Failures.

These information show that the JVM is a complex system characterized by several dependability bottlenecks. In particular, performance-enabling components of the JVM represent a serious threat for JVM failure-free execution. Another way to gain useful insights into dependability features of the JVM is to analyze the relationship between failure sources and failure manifestations. Table 3.7 reports the percentage of failures with respect to JVM components for each failure manifestation kind. Looking at this table it is possible to deduce that:

- The memory management unit is accountable only for less than half of the `OutOfMemory` errors. These errors are often caused also by the execution unit

Failure Manifestations vs. Failure Sources	OS-Level	Out Of Memory	Stack Overflow	Runtime Exception	Assertion Failure	Internal Error	Others Exc.	Silent Crash	Hang/ Deadlock	Comp. error
OS Virtualization Layer	24.10%	15.15%	0.00%	0.00%	20.00%	18.18%	0.00%	10.00%	3.70%	9.09%
Execution	42.17%	36.36%	80.00%	75.00%	50.00%	18.18%	0.00%	70.00%	37.04%	81.82%
Memory Management	22.89%	42.42%	20.00%	12.50%	20.00%	9.09%	0.00%	20.00%	33.33%	0.00%
System Services	10.84%	6.06%	0.00%	12.50%	10.00%	54.55%	100.00%	0.00%	25.93%	9.09%

Table 3.7: Distribution of failure sources for each type of failure manifestation

(mainly by JIT compilers);

- The source for the greatest part of Runtime Exception and Computation errors is the Execution Unit;
- The System Services unit is the main reason of Virtual Machine Internal Errors and Deadlocks;
- Just a few VM-level failure manifestations are attributable to the OS virtualization layer, whereas the greater part of error in this component manifested as OS-level messages. This confirms that built-in detection mechanism are not able to monitor the lower layers of the JVM.

Relationships between Frequency and Workload

We finally classify failure according to their frequency and to the workload applied when the failure itself occurred. Figure 3.4-a reports the percentage of errors with respect to JVM components for each frequency category.

Regular Failures are most recurrent ones (39.81%). Since regular failures always occur at the same point in program execution, Java developers can avoid them by adopting proper workarounds. *Regular failures* are mainly attributable to the Execution Unit (22.21%) and to the Memory Management Unit (12.96%).

Startup (11.11%) and *Hourly* (10.19%) failures occur at the first stages of Java Program Execution. The OS Virtualization Layer and the Execution Unit are the main

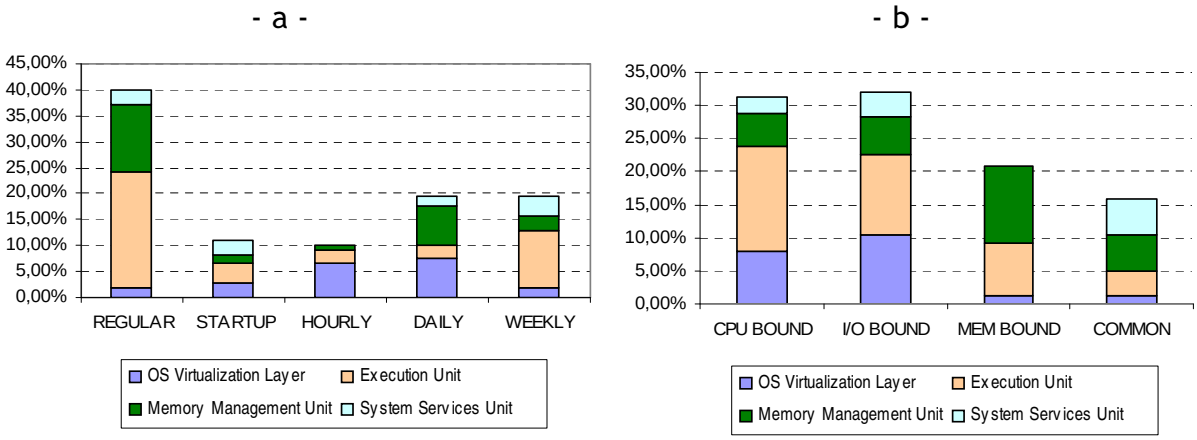

Figure 3.4: Frequency and workload classification of failures with respect to JVM components

causes of hourly failures (6.48%, 2.78%), whereas each component plays an equivalent role in startup failures. Many non-regular failures shows a daily or weekly frequency (19.44%).

It is worth noting that Execution Unit and System Services Unit failures increase when frequency decreases, whereas Memory Management Unit and OS Virtualization Layer Unit failures decrease when frequency decreases. This suggests the presence of software aging phenomena in this components (especially in JIT compilers, Shared Runtime Support and Thread Management sub-components). Figure 3.4-b reports the percentage of error with respect to JVM components for each workload level defined. The greatest percentage of failures occurred under CPU Bound Workloads (32.00%), followed by

I/O Bound Workloads (26.40%). Less failures occured under Memory Bound Workloads (21.60%) or "Common" Workloads (20.00%).

The greatest part of failures (80%) occur when significant workloads are imposed on the JVM, moreover CPU Bound and I/O Bound applications seem to be more critical for the JVM than Memory Bound applications. These results indicate that CPU Bound and I/O Bound applications, such as Web Servers, stress mainly the Execution unit (50.98% CPU Bound; 38.46% I/O Bound) and the OS Virtualization Layer (25.49% CPU Bound; 32.69% I/O Bound). On the other hand, the most relevant percentage of failures with non-significant workloads are attributable to errors in the Memory Management Unit (33.03%) and in the System Services Unit (37.04%). **Therefore, since the JVM suffers mainly CPU Bound and I/O Bound applications, it is possible to argue that the development of strategies and mechanisms aimed to improve the reliability of the virtual machine should first address these kinds of applications.**

The analysis of the relationships between failure frequency and workload (reported in table 3.8) also suggests the presence of aging phenomena. Regular and Startup failures usually occur when non significant workloads are applied, where as non regular failures usually occur when significant workloads are applied. For instance, 80% of weekly failures occur when CPU Bound or I/O Bound workloads are applied.

Table 3.8: Relationships between failure frequencies and workload levels

	CPU BOUND	I/O BOUND	MEM BOUND	COMMON
STARTUP	20,00%	30,00%	10,00%	40,00%
HOURLY	50,00%	40,00%	10,00%	0,00%
DAILY	20,00%	55,00%	20,00%	5,00%
WEEKLY	40,00%	40,00%	5,00%	15,00%
REGULAR	15,15%	15,15%	27,27%	42,42%

3.3.3 Clues about software aging in the JVM

In order to perform an effective study of the development of aging phenomena, it is compulsory to analyze the development of the state of the system under observation. Therefore it is very difficult to extract such information analyzing only failure reports. Anyway, in order to extract at least some clues about the presence of aging phenomena in the JVM, we first selected only failure reports reporting an exact indication of the time to failure. This information was available only for 23 reports. Unfortunately this number is too small to perform a meaningful analysis. Therefore we had to consider also reports with a qualitative indication of the TTF.

The procedure adopted to perform an aging-oriented analysis of failure reports is the following:

1. Exclude failures marked as "Always Reproducible": failures which are due to aging phenomena are usually strictly dependent on the workload applied on 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 these failures, exclude those which 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, due to lack of data in the reports, for a 30% of occurred failures.

3. Analyze the distribution of the remaining failures (32% of the total number of failures) with regards to the estimation of the time to failure.

The pie chart in figure 3.5 reports the distribution of non-deterministic failures occurring with relevant workload according to the time to failure.Unfortunately, there is an high level of uncertainty, since data (even qualitative) about TTF was missing in a consistent number of reports. Only 2% of these failures occur during the startup phase, whereas 45% of these failures occur after a significant time (16% daily, 29% weekly). This means that in about half of the considered failures we have significant clues of software aging phenomena.

Summarizing, it is possible to state that:

- About 40% of failures are absolutely NOT due to aging phenomena;
- There is an high probability that 15% of failures are due to aging phenomena;
- It is not possible to state anything for the remaining 45%;

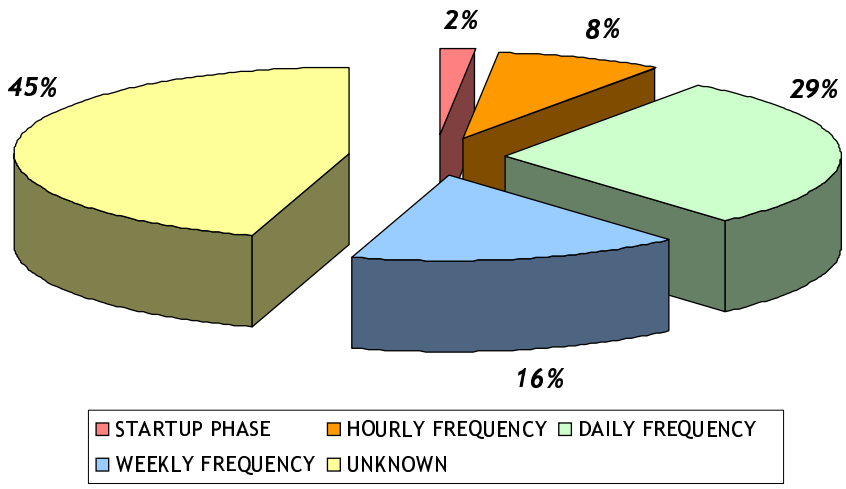

Figure 3.5: Distribution on non-deterministic failures by time-to-failure

Therefore, even if software aging is not the main cause of failures in the JVM, it is a phenomenon which is responsible for a non negligible percentage of failures: therefore it will be worth performing an extensive experimental campaign aimed at exploring the development of these phenomena.

3.4 Final Remarks

This chapter reports the results of a bug reports based classification of the failures behavior of the Java Virtual Machine. Previous work showed this kind of analysis to be very effective when performing preliminary dependability evaluations, especially for OTS items.

The approach adopted for this analysis is very simple and can be resumed in the following steps:

1. Collection of a consistent number of bug reports from bug database and extraction of the failure reports contained inside;

2. Careful filtering of failure reports in order to drop those ones which result to be inaccurate or even misleading;

3. Classification of failure reports according to their manifestation, the source of the failure and the relationships between failure frequency and workload.

Since the adopted approach is not dependent on specific features of the JVM it can be applied to perform a preliminary dependability evaluation for a wide range of OTS items.

The analysis of JVM failure reports showed that there is a non-negligible dependency of JVM reliability on the underlying Operating System, and that the Execution Unit is responsible for the greatest part of reported failures. The analysis also showed that failures are often due to hardly reproducible bugs (Mandelbugs and Heisenbugs): these failures can be classified into *environment dependent* and *environment independent*. Approaches based on active replication ([75, 76]) make the JVM robust only with respect to hardware faults and software faults in the OS. If there is a component in the primary VM, the same fault would be activated also on the backup.

Finally, the analysis showed a strict relationship between failures and the workload applied on the JVM, showing that there are several clues that software aging phenomena may develop inside the JVM. An experimental campaign is mandatory in

order to detect and estimate this phenomena. Aging phenomena inside the JVM are extensively explored in the following chapter, whereas aging caused by the JVM at the Operating System level is studied in chapter 5.

"Dicette 'o pappic 'nfaccia 'a noce: damme tiempo ca te spertoso"
The worm said to the nut: give me some time, and I will pierce you.

Neapolitan Proverb

Chapter 4

Aging Phenomena Characterization

This chapter presents a measurement-based methodology to study software aging phenomena as a function of the workload. This methodology, specifically designed for OTS-based systems, addresses the issues outlined in section 2.2, by providing a solution to isolate the contribution to aging trend of a particular layer, selecting workload parameters which are more relevant to aging trends, and assessing the influence of such paramters on detected aging trends.
This methodology has been adopted to characterize the development of aging phenomena inside the Java Virtual Machine, which is a relevant example of the items which may be employed in OTS-based system since it provides a complete virtualization of the underlying execution environment. Therefore this chapter, after introducing the above mentioned methodology, discusses the results of an analysis performed on data collected through a massive carried out on the JVM.

4.1 Rationale and Approach

Several 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 example of critical system affected by software aging.

To project a target's trajectory, the weapon control computer required two floating point input values: its velocity and the time. However, the time was stored into the system as an integer, counting tenths of seconds and storing them in a 24-bit register. The necessary conversion into a real value caused a round-off error in the calculation of target's expected position. For a given velocity of the target, these errors were proportional to the length of time the system had been running. As a consequence, the risk of failing a target increased with the time the system operated without rebooting. Users of the system 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", i.e.: a rejuvenation frequency, thus leading to the famous incident in which 28 people were killed.

OTS items, which are starting to be employed also in critical contexts, cannot be thought as not being affected by Software Aging phenomena. On the contrary, since they often lack proper testing (as already mentioned in section 2.1.1), they are more subject to these phenomena. In order to employ such components in critical scenarios, it is therefore very important to characterize first their behavior from a software aging perspective.

Studying Software Aging dynamics in OTS-based software systems becomes more difficult due to the interconnection of application-specific logic and several off-the-shelf components. OTS-based systems may be regarded as being made of a sequence

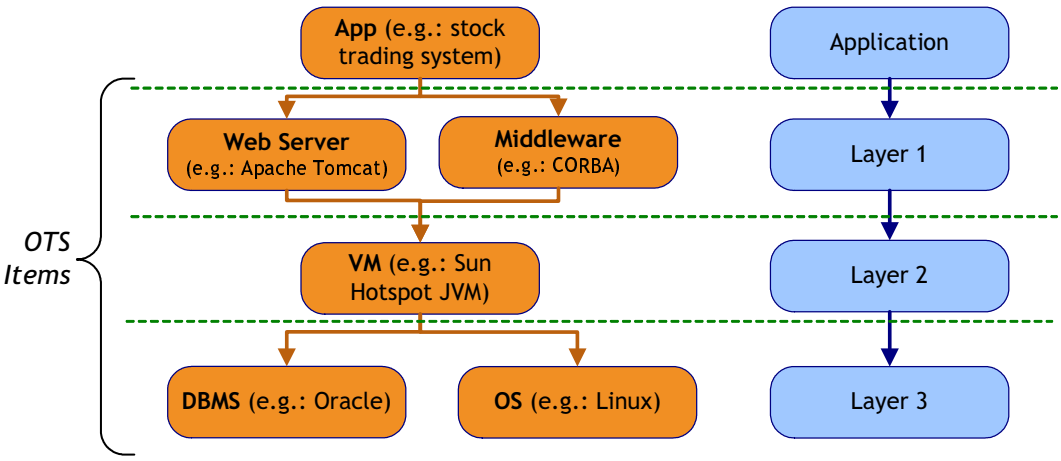

Figure 4.1: Stratification in layers of an OTS-based software system

of software layers, each of them using services from the lower layer and offering services to the upper one. Components offering basic services such as operating system are at the lowest layer, whereas application-specific business is located at the highest layer. A typical example of such situation is depicted in figure 4.1.

In this figure, the application, a Stock Trading System, uses services offered by two OTS Items: a Web Server and a Middleware platform; these OTS items run upon a Java Virtual Machine, another OTS item. Finally, the virtual machine interacts with the underlying operating system, and a Database Management System. The stratified organization of the software system can be characterized by the following layers:

- ***Application*** - The application-specific business logic;
- ***Layer 1*** - The Web Server and the Middleware platform

- ***Layer 2*** - The Java Virtual Machine;
- ***Layer 3*** - The DBMS and the Operating System.

All previous measurement-based works concerning Software Aging analysis consider aging trends as measured at the Operating System level. In this way, given the scenario depicted in figure 4.1, it is impossible to distinguish in which component the source of the aging phenomenon is located. In particular, when OTS items providing a complete virtualization of the execution environment, like the JVM, are 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. Indeed, since the JVM preallocates 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, when dealing with the relationships between aging trends and workload, OS-level workload parameters are always chosen, either randomly or according to previous experience. In order to perform an effective assessment of Software Aging dynamics in OTS-based systems, it is compulsory to take into account component-specific workload parameters. Indeed, once the component has been chosen, the analysis of the relationships between aging and workload cannot be performed without considering the workload parameters describing the specific operations performed. Given the complexity of industrial software systems the number of such parameters can easily reach an order of magnitude, which makes it difficult, if not impossible, to

evaluate the influence of each parameter on the measured aging trends.

Therefore, when targeting Software Aging analysis for OTS-based systems, two challenging questions arise:

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

As regards 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 component. Therefore 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 the usage of such resource at the different layers, and locate the layer(s) in which aging phenomena are introduced.

As for the second question, it is compulsory to address the selection of the smallest set of workload parameters which has the greatest influence of aging phenomena. This problem, also known as the *dimension reduction* or the *feature selection* problem, is one of the most prevalent topics in the machine learning and pattern recognition community. In our approach, we will employ *Principal Component Analysis* in order to reduce the number of workload parameters, and then statistical *Null Hypothesis* testing in order to select only those workload parameters which have a real influence

on aging phenomena. Finally, *Partial Linear Regression* will be employed in order to estimate the relationships between workload and aging trends. Several works in scientific literature [9, 10] have shown that linear models are capable of describing in a reliable way such relationships.

Our approach to evaluate software aging will be therefore divided into 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 (for a particular OTS component).

2. **Workload Characterization**: it is well known that workload impact on aging trends represents a key point in software aging studies. In this step 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 we evaluate i) the aging trends exhibited at the different layers, ii) the influence of workload parameters on such trends, iii) the relationships between relevant workload parameters and aging trends.

4.2 Design of Experiments

The Design of Experiments (DOE), described in [81] and [67], is a systematic approach to investigation of a system or process. 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.

The first step in planning such measurement 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 the response variable; 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, a 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 the response variables is determined by the number of controllable factors and the number of levels assigned to each factor. Given k factors and l levels, the number of treatments n required is $n = l^k$. A response variable y can then be written, using the regression model representation of the factorial experiment, as a linear combination of the factors $x_1,\ldots,x_k$ and their products. For instance, for a two-factor factorial experiment, the response variable can be written as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \epsilon \tag{4.2.1}$$

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 factor on the response variable;
- The ϵ term is a random variable that captures the effect of all uncontrollable parameters.

Figure 4.2 depicts our approach to employ the DOE technique for Software Aging Analysis. Among the several layers in which an OTS-based software system may be divided, we choose a particular layer (l_t) as the target of our analysis. The goal of our experiments is to analyze the effects of changes in workload parameters of the target layer on Software aging trends. The latters, measured in terms of memory depletion on performance degradation will be our response variable.

As shown in figure 4.2, 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 our study, we will conduct a series of experiments using $W_1^c, \ldots, W_n^c$ parameters in order to ensure that each one will be executed with a different workload level; the number of experiments

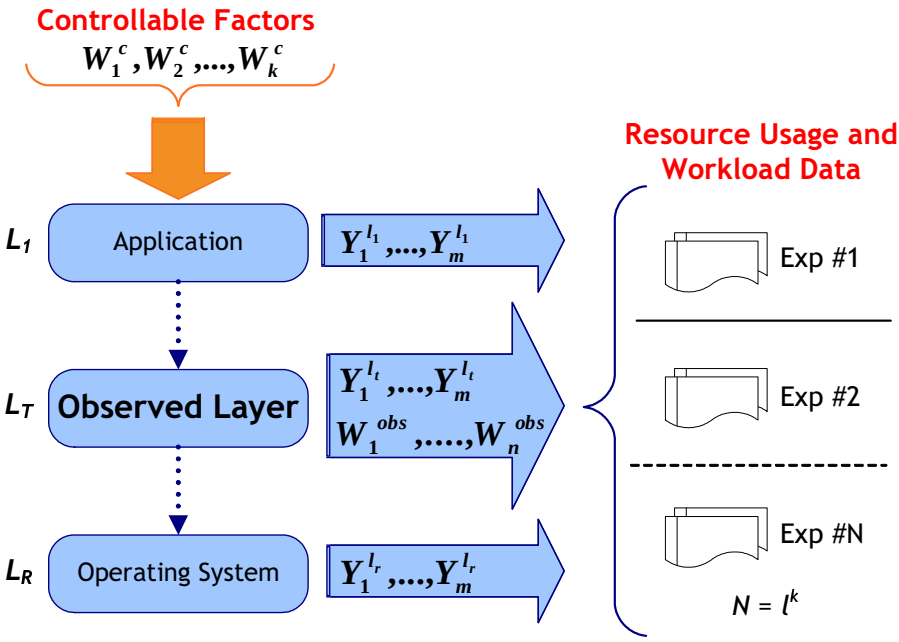

Figure 4.2: Design of Experiments for Software Aging Analysis in OTS-based systems

to perform, n_{exp}, is generally application-specific: for instance it may be chosen taking into account the maximum workload sustainable by the benchmark application. For each experiment we will collect **i)** resource usage information for layers $l_1, \ldots, l_r$, in order to estimate aging trends, and **ii)** observable workload parameters $W_1^{obs}, \ldots, W_n^{obs}$ related to the target layer l_t.

The greatest part of the factors that will be employed in our analysis are uncontrollable. We are therefore more interested in the ϵ component of the expression 4.2.1 rather than in the effects of controllable parameters. Standard statistical analysis techniques, such as *Analysis of Variance (ANOVA)*, which take into account only the effect of controllable factors on response variables, cannot be employed. Given this

scenario, we chose to reduce the number of controllable factors to only 1 factor, thus avoiding factorial design and keeping the number of treatments low, and to increment the number of level for this controllable factor, thus obtaining more information to evaluate the contribution of uncontrollable factors on aging trends.

The experiments will be then performed in the following way:

1. Choose a controllable factor to drive the synthetic workload applied on the system for each experiments;

2. Select the target layer and its workload parameters which have to be monitored, grouped by the particular component or sub-system they belong to;

3. Define the number of levels n to assign to the above mentioned controllable factors;

4. Choose an interval to sample resource usage and workload information during each experiments;

5. Perform the n_{exp} experiments, collecting resource usage information for each layer and workload information for the target layer.

4.3 Workload Characterization

Once data have been collected, the next step of our methodology is concerned with the characterization of the workload. This step has the following goals:

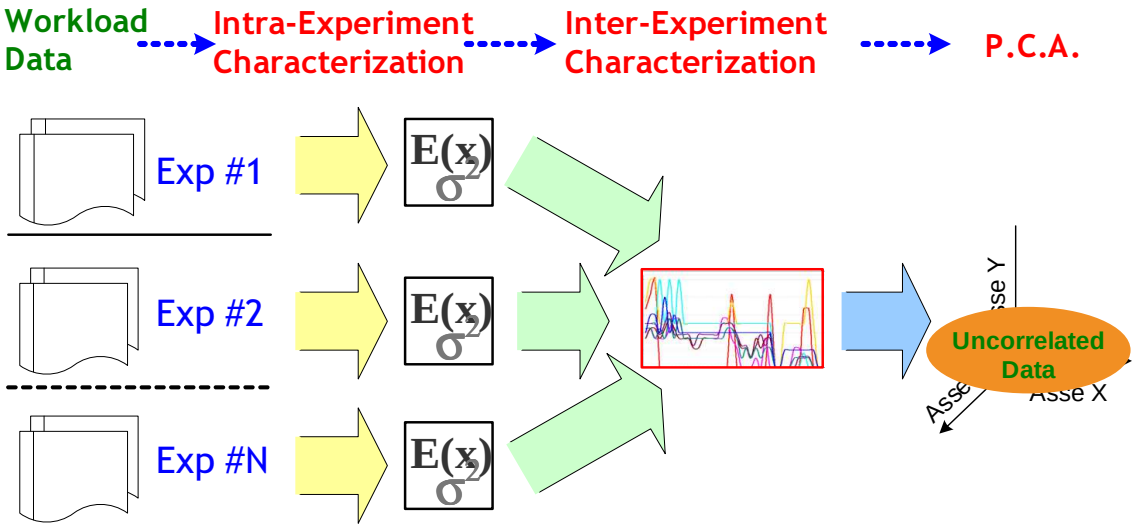

Figure 4.3: Workload characterization phases

- Perform a statistical characterization of the behavior of the target layer for each experiment and identify variations among experiments by relating synthetic data extracted from each experiment in order to draw an overall figure of the evolution of workload parameters.

- Reduce the complexity of the analysis by reducing the number of variables to consider when assessing the relationships with aging trends.

The workload characterization process may divided into three phases, depicted in the conceptual diagram reported in figure 4.3, and described in the following subsections.

4.3.1 Intra-Experiment Characterization

The first step of our analysis deals with the **detection of clusters** in data, in order to identify the different workload states traversed by the observed layer during the experiment. The workload parameters $W_1^{obs}, \ldots, W_n^{obs}$ are grouped according to the component they belong to; for each of these groups a cluster analysis is performed using the *Hartigan's k-means clustering algorithm* [1].

If the variables for clustering are not expressed in homogeneous units, a normalization must be performed. In our methodology, we use the following normalization method:

$$x_i' = \frac{x_i - min_i\{x_i\}}{max_i\{x_i\} - min_i\{x_i\}} \tag{4.3.1}$$

where x_i' is the normalized value of x_i. Through this transformation all the 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 the cluster analysis, we eliminated the outliers in data by removing samples whose 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 (the centroid of a cluster is the average point in the multidimensional space defined by the dimensions. In a sense, it is the center of gravity for the respective cluster). The clustering algorithm iteratively updates centroids and assigns points in normalized data series to the closest centroid until centroids no longer move. The choice of

[1] The Hartigan's algorithm is a well-known iterative algorithm for cluster analysis published in "Clustering Algorithms, John Wiley and Sons, 1975"

the number of clusters is a vital point in the characterization of the workload for a given component C, since it defines the number of states traversed by the component during the experiment. In our analysis, we used an approach based on *frequency count*. Given a component C in the target layer l_t, whose related workload parameters are $W^{obs}_{C_1}, \ldots, W^{obs}_{C_m}$, we consider the range of values $[w^{obs}_{C_{imin}}; w^{obs}_{C_{imax}}]$ for each parameter $W^{obs}_{C_i}$, and then divide this range into 100 equally sized intervals; for each interval we count the number of $w^{obs}_{C_i}$ samples falling into that interval; in this way the number of sample occurrences is described as a function of the interval. In order to infer the number of clusters, we first calculate the number NM_{C_i} of relative peaks in the frequency count function for each workload parameter and then determine the number of cluster as:

$$NC = max\{NM_{C_1}, NM_{C_2}, \ldots, NM_{C_m}\} \tag{4.3.2}$$

Clusters whose centroids are close each other can be merged, thus reducing the total number of clusters and the complexity of the subsequent analysis. Another viable approach to select the number of clusters is proposed in [61] and it is based on the evaluation of ratio of the within-cluster sum of squares obtained with i and $i + 1$ clusters: the number of clusters is iteratively incremented until the ratio goes below a predefined threshold.

Once the cluster analysis has been performed, the intra-experiment characterization is completed by calculating the **expected values** for each workload parameter and

for each cluster, and by estimating potential **linear trends** in time series for each workload parameter and for each cluster. 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, in order to perform the trend estimation we adopted an approach which is similar to the one adopted for the estimation of trends in time series with seasonal patterns. In particular, we estimate the trend for each interval, and then calculate a weighted average of the trend estimated for each interval.

Assuming that the cluster analysis revealed the presence of j clusters for the component C, the intra-experiment characterization will therefore calculate, for each workload parameter $W_{C_i}^{obs}$, the expected value in the following way:

$$E[W_{C_i}^{obs}]_j = \frac{1}{n}\sum_{k=1}^{n} w_{C_{ik}}^{obs} \; with\, k \in j^{th}\, Cluster \tag{4.3.3}$$

As regards 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 interval, the linear trend in data may be expressed as:

$$T[W_{C_i}^{obs}]_j = \frac{1}{n}\sum_{k=1}^{r} m_k\, TREND(W_{C_i}^{obs})_k \tag{4.3.4}$$

where `TREND` is the autoregressive function employed to calculate the trend in a single interval. This value 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.

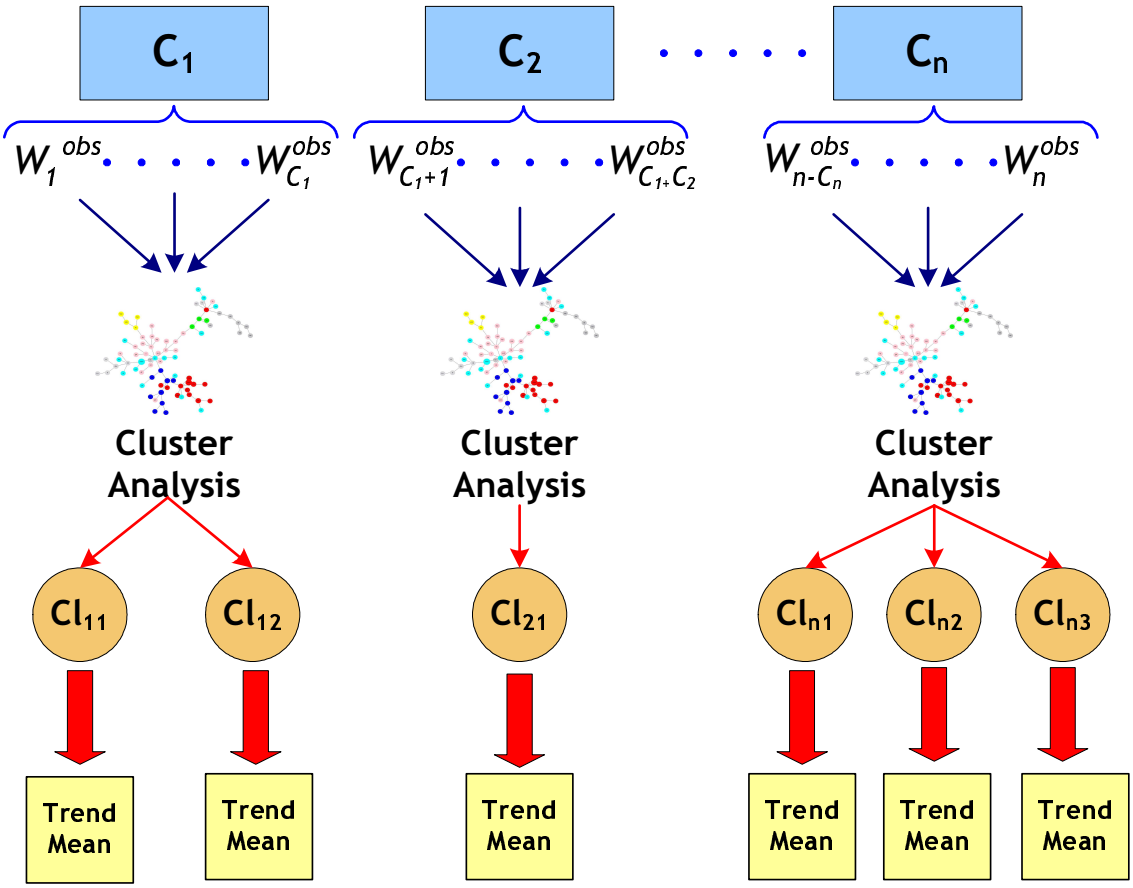

Figure 4.4: The Intra-Experiment Characterization Process

The sequence of operations needed to perform the intra-experiment characterization is reported in figure 4.4: 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.

4.3.2 Inter-Experiment Characterization

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

For instance, let us consider an OTS-based system like the one depicted in figure 4.1; let us suppose our target layer to be the java Virtual Machine layer. 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 m observable parameters and k clusters, the number of data series constructed in this phase is $m * k * 2$. Indeed, it is necessary to build a data serie not only for each observable workload parameter, but also for each workload state visited by such parameter.

Since cluster analysis is performed for each experiment, it may happen that two different experiments reveal a different number of clusters. Thi 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 the subsequent analysis.

4.3.3 Principal Component Analysis

It is not possible to assume 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 an higher weight would be given to correlated variables, thus actually amplifying the effects of such variables on aging trends.
In order to remove correlation among data, we apply Principal Component Analysis (PCA) which transforms original data into uncorrelated data. As for cluster analysis, data have to be normalized first, since non normalized data give an higher weight to data series with an higher variance. Through normalization all data series have the same weight.

PCA computes new variables, called *Principal Components*, which are linear combination of the original variables, such that all principal components are uncorrelated. PCA transform m variables $X_1, X_2, \ldots, X_n$ into m principal components $PC_1, PC_2, \ldots, PC_n$ with $PC_i = \sum_{j=1}^{m} a_{ij} X_j$. The values of the a_{ij} coefficients are in the range $[-1; 1]$. This transformation has the following properties:

1. $Var[Z_1] > Var[Z_2] > \ldots > Var[Z_m]$ which means that Z_1 contains the most information and Z_m the least;

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

Principal components are decreasingly ordered according to their variance. Is 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 small and negligible percentage of the information contained in original data is thrown away. Typically a very small percentage of original variables (e.g.: 10%) are able to explain 85% to 90% of the original variance.

Therefore, in our methodology, for each component in the observed layer, we select a subset q of the calculated m principal components, having $q \ll m$. Each of these

components is expressed as:

$$PC_i = \sum_{j=1}^{m} a_{ij} X_j \; with \, 1 \leq i \leq q \tag{4.3.5}$$

Therefore each principal component has a meaningful interpretation in terms of the workload data series calculated in the inter-experiment phase, since the a_{ij} coefficient express the weight that each workload parameter has in the principal component(s) where it appears. A weight close to 1 (or -1) means that the workload parameter has a very high impact on the principal component, whereas a weight 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 which 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 which are more relevant to aging trends.

4.4 Software Aging Analysis

The last step of our methodology deals with the detection and estimation of aging trends in resource usage data, and with the analysis of the relationship between the workload, characterized in the previous step, and the aging trends themselves.

The process employed to perform this step is depicted in figure 4.5. The first phase deals with the detection and the estimation of aging trends in resource data collected

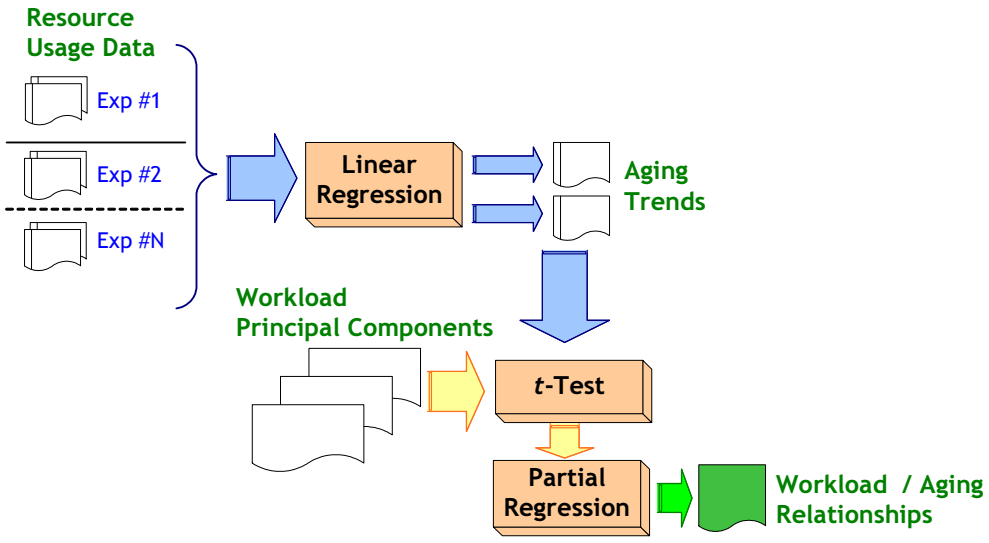

Figure 4.5: Software Aging Analysis process

during the experimental campaign. To this aim, we will apply **i) Hypothesis testing**, to assess the presence of a trend in data, and **ii) Linear regression**, to estimate such trend if present.

Null hypothesis testing is a common statistical procedure in which a first hypothesis, called the *Null Hypothesis* is tested against an alternate hypothesis. The null hypothesis usally refers to a condition in which a particular treatment does not have any effect on the output variable. In our methodology we test the null hypothesis of no trend in data against the alternate hypothesis stating the presence of a linear trend in data, using the student's t statistic. The null hypothesis cannot be rejected if 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,which is comprised between 0 and 1: the lower is the confidence

level, the higher is the probability of being in the tail of the distribution.

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. The model can be written as $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots \beta_n X_n$, 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" whether it is really a measurement error or not, and is assumed to be independent of the X_i.

As far as resource usage data are concerned, we have to take into account only the time as an independent variable, thus the regression model can reduced to $y = a + bx + \epsilon$. In our analysis we will employ *least squares analysis* in order to perform parameter estimation. Aging trends must be estimated for each monitored resource, for each layer in the OTS-based system, and for each cluster detected in the intra-experiment analysis. It is very important to repeat parameter 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.

After calculating aging trends, linear regression, in particular **multiple regression** will be employed in order to assess the relationships between aging trends and the principal workload components built in the workload characterization step.

The parameters of the regression model, determined with the least square analysis, will be validated again using the student's t test. Partial regression parameters, whose *t-values* falls 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. After discarding invalid parameters, the regression model has to be solved again with the remaining parameters, until all parameters show a *t-value* falling in the center of the t distribution.

Finally, once the influence of principal workload components on aging trends has been evaluated, the impact of original workload components may be evaluated by analyzing the structure of each principal component. Since each principal component has the form $PC_i = \sum_{j=1}^{m} a_{ij} X_j$, our goal is to express X_js as a function of PC_is. In general, this is not possible, since the number of principal components q is usually far less than the number of original workload parameters n. In order to perform a meaningful estimation of the relationships between the original workload parameters and the aging trends, given n workload parameters and q principal components, it is possible to proceed in the following ways:

1. Delete, if possible, $n-q$ variables having a little influence on the principal components, i.e. variables whose coefficient is relatively close to 0 for each principal component. In this way it will be possible to express workload parameters 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 possible to accurately express the relationships between workload and aging trends, it will be possible to obtain an overestimation of such relationships. Overestimating the effects of workload on aging trends will lead to predict a lower time for resource exhaustion, which is always better than predict a time higher than the actual TTE.

Summarizing, the presented methodology allow to completely characterize software aging dynamics into an OTS item (or a group of OTS items). This characterization, which includes the contribution of the target layer to the overall aging trends and the estimation of the influence of the workload imposed on the target layer on aging trends, may be very useful in several scenarios:

- During the development of an OTS-based system, the OTS selection and acquisition process may be improved by information related to OTS items' software aging behavior;
- During the testing phase of an OTS-based system, the detection of aging-related bugs may be improved by locating the particular sub-component(s) where such bugs are located;
- During the operational phase of an OTS-based system, the development of proper rejuvenation strategies may be improved by the knowledge about the

relationships between workload and aging trends.

4.5 Characterization of Aging Phenomena in the JVM

In this section we discuss the results of a significant experimental campaign aimed to estimate aging trends introduced by the JVM, and to investigate the relationships between workload parameters and aging trends.

Aging trends have been estimated by evaluating two well-known software aging indicators, **throughput loss** and **memory depletion**. Workload information, as well as resource usage data, have been collected using an ad-hoc implemented monitoring infrastructure for the Java Virtual Machine, presented in the next sub-section.

4.5.1 Java Virtual Machine Monitoring

A monitoring infrastructure is a key component in each dependability evaluation campaign: this component should collect enough information about the behavior of the monitored system when proper workloads are applied or faults are injected.

In order to perform an experimental campaign aimed at studying Software Aging Phenomena in the JVM, we developed an infrastructure, named *JVMMon*, to monitor its behavior. 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 thus collecting the evolution of JVM state together with errors and failures.

Table 4.1: 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 (or unloaded), compiled code size and absolute address where code is loaded in memory
Exception	Generated when an exception is detected by a java or native method	Thread throwing the Exception, location where the exception was detected, java.lang.Throwable instance describing the Exception
Exception Catch	Generated whenever a thrown exception is caught	Thread catching the Exception, location where the exception was caught, java.lang.Throwable instance describing the Exception
Monitor Contended Enter/Entered	Generated when a thread is attempting to enter (enters) a Java monitor acquired by another thread	Thread attempting to enter (or entering) the monitor, instance of the 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 when 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 when 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

The proposed monitoring infrastructure allow on-line analysis of JVM state evolution through a three-step process: **i)** a monitoring agent, developed using the JVM Tool Interface (JVMTI) (which stems from the *Java Platform Profiling Architecture* [82]) and Bytecode Instrumentation (BCI), intercepts events generated inside the JVM and collects data about its state; **ii)** a monitoring daemon processes these 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.

Field Data Sources

Field data are collected using the following information sources:

Table 4.2: 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 flas for a Class Instance
GetObjectSize	Object size in byte
GetObjectHashcode	Unique identifier associated with the object

- ***JVMTI events*** - Several events raised from the JVM are intercepted implementing JVMTI callbacks. Events intercepted by JVMMon are reported in Table 4.1.

- ***JVMTI functions*** - JVMTI callbacks use these functions in order to update the state of the component of the JVM which the raised event is related to. The functions used to determine the state of the Java Virtual Machine are reported in Table 4.2.

- ***Java Objects (through BCI)*** - Java Methods are instrumented in order to obtain further information about the virtual machine.

Monitoring the state of the JVM

The state of the JVM may be defined as the union of the state of its components. Some of these components, namely the JIT compilers, the Interpreter and the OS Abstraction Layer, can 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 the classes loaded in the permanent generation of the JVM. 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 we keep 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 JLS only a 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 us to collect scheduling information also in multiprocessor systems.

Table 4.3: JVM workload parameters captured by JVMMon

COMPONENT	PARAMETER	DESCRIPTION	U.M.	COMPONENT	PARAMETER	DESCRIPTION	U.M.
Runtime Unit	MET_INV	Method Invocation Rate	*methods/ min*	Jit Compiler	CI_THR0_EVENTS	compiler thread 0 events	*events/min*
	OBJ_ALL	Object Allocation Rate	*objects/min*		CI_THR0_TIME	compiler thread 0 time	*ms*
	ARR_ALL	Array Allocation Rate	*arrays/min*		CI_NATIVE_COMP	native JIT-compilations	*events/min*
Class Loader	CLS_INIT	Time spent in class Initialization	*ms*		CI_NATIVE_TIME	native JIT-compilation time	*ms*
	CLS_VER	Time spent in class verification	*ms*		CI_OSR_COMPILES	On-Stack-Replacement (OSR) JIT-compilations	*events/min*
	CLS_LOAD	Time spent in class loading	*ms*		CI_OSR_TIME	OSR JIT-compilation time	*ms*
	INIT_CLS	Number of Initialized Classes	*classes/min*		CI_STD_COMP	standard JIT-compilations	*events/min*
Memory Management	SM	Object Size Mean	*bytes*		CI_STD_TIME	standard JIT-compilation time	*ms*
	SV	Object Size Variance	*bytes*		CI_TIMEPERCOMP	time per compilation	*ms*
	COLL0_INV	Number of Young generation Colletor (Copying) Invocations	*invocations/min*	Thread Management Unit	TE	Threading Events	*events/min*
	COLL0_TIME	Time spent during Young Generation Collection	*ms*		WM	Waiters (threads waiting on a mutex) Mean	*# of threads*
	COLL0_TIMEPERINV	Young Collection Duration	*ms*		WV	Waiters (threads waiting on a mutex) Variance	*# of threads*
	COLL1_INV	Number of Tenured Generation Collector (Compacting) Invocations	*invocations/min*		NWM	Notify Waiters (Threads waiting on a condition variable) Mean	*# of threads*
	COLL1_TIME	Time spent during Tenured Generation Collection	*ms*		NWV	Notify Waiters (Threads waiting on a condition variable) Variance	*# of threads*
	COLL1_TIMEPERINV	Tenured Collection Duration	*ms*	* 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.			
	SAFEPOINTS	Number of Safepoints(*) reached	*safepoints/min*				

- ***Memory Management Unit*** - Since we aim at monitor the integrity of data structures on which the reference handler and the garbage collector operate, we define the heap of the JVM is defined as the set of objects allocated in Java Heap since VM has been started. JVMMon has been implemented in order 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 either at the application and the JVM layer, in order to isolate the layer in which prospective aging trends are introduced.

Monitoring JVM workload

Capturing the state of the JVM allows to assess its current health: by the analysis of system logs it is indeed possible to understand whether anomalous conditions are verified inside the JVM. Since it is impossible to understand how aging phenomena evolve inside the JVM without characterizing its workload, JVMMon is also capable of monitoring JVM workload parameters. This task has been accomplished by using **i)** JVMTI functions, **ii)** Bytecode Injection and **iii)** Using performance counters accessible through the `jstat` [2] interface.

A number 30 workload parameters (reported in table 4.3) were monitored. These parameters are related to the components of the JVM described in section 2.3.2.

JVMMon Architecture

Figure 4.6 shows the main components of JVMMon and their interconnections. The JVMTI agent on the `Monitored JVM` retrieves data about JVM state handling events raised from 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 each other through a shared memory. Each event raised by the Monitored JVM is also logged on a file on the local file system. Moreover, the `JVMTI agent` sends an heartbeat message at a fixed rate in order to make the *Local Monitor Daemon* aware of JVM failures.

The *Local Monitor Daemon* notifies the *Data Collector* about failures and relevant

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

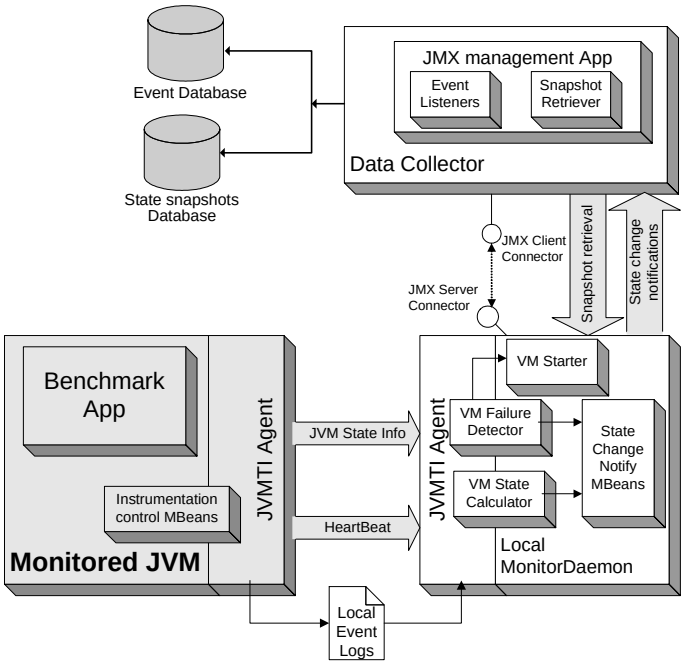

Figure 4.6: JVMMon Architecture

changes in the state of the monitored virtual machine. This 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:

1) Handling events generated by the JVM implementing JVMTI callback functions.

2) Retrieve data about JVM state through both JVMTI API and BCI.

3) Send retrieved data to the `Local Monitor Daemon`.

4) Store data about events in the `Local Event Log`.

5) Store resource usage data in the `Resource Usage Log`.

6) Store workload information in the `Workload parameters Log`.

As regards BCI, Java classes are instrumented in order to:

i) Detect context switches: each time a method is entered the current thread is checked.

ii) Detect object finalization: `finalize()` methods are instrumented thus detecting when the JVM releases resources held by an object.

iii) List system resources such as file descriptors and sockets, instrumenting JDK core classes. Moreover, in order to keep overhead and intrusiveness in the instrumented JVM as little as possible, not all loaded classes are instrumented. A pattern-based rule is used to define which classes are to be instrumented.

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 related to failures or relevant state changes to the Data Collector. The `VM state calculator` sub-component copes with the first task, where as the `State Change Notify MBeans` sub-component copes with the second one.

The `VM Failure Detector` sub-component is in charge of detecting Monitored JVM failures. The detection is performed at three different layers: i) *Process Layer*: JVM crashes (i.e.: a SIGSEGV failure) are detected checking for its PID; ii) *Local log Layer*: Hang failures (i.e.: a deadlock between Java or VM threads) are detected

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 `Event Database` and `State Snapshot Database`.

4.5.2 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 [3] and a maximum heap size of 512 Megabytes; it was configured to run serial, stop-the-world collectors both on the young and the tenured generation. No other application is competing for system resources with the JVM: the server workstation is started with just minimal system services.

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

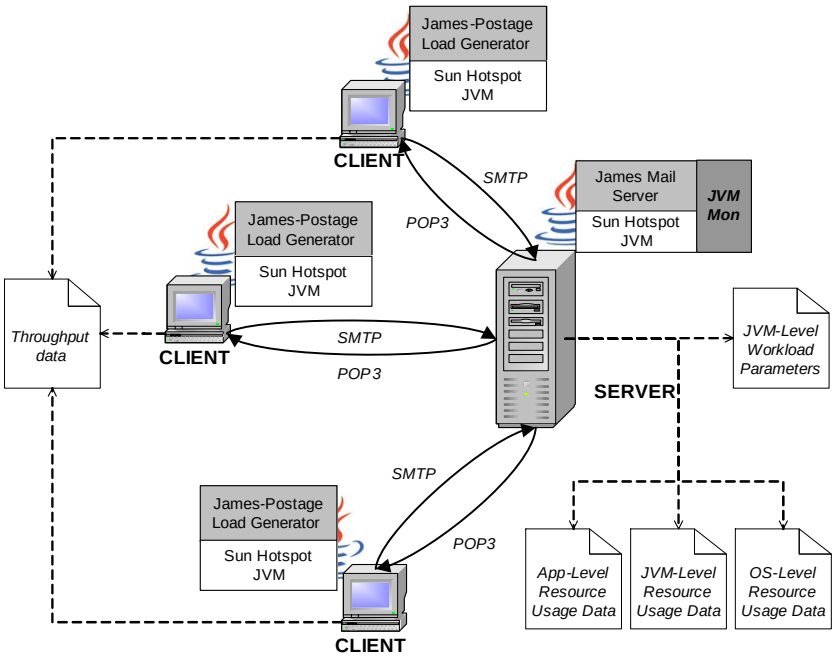

Figure 4.7: Experimental setup for data collection

We chose a mail server as the benchmark application for our analysis since it represents an important class of long running server applications usually stressed by significant workloads.

The server workstation is stressed using a load generator, which acted as an email-client, sending a receiving mails at a constant pace for the whole experiment. The load generator employed allows to control the imposed workload by specifying the number of mails per minute and the size of such mails. In our experimental campaign, the number of mails per minute has been chosen as the controllable workload parameter, whereas the size of the e-mails will be kept constant among all experiments Figure 4.7 depicts a single experiment scenario. JVMMon collects the greatest part

of the information required for the analysis. Only data about server's throughput are extrapolated from logs generated 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.

4.5.3 Experimental Campaign

In order to choose a proper range for the controllable workload parameter, the capacity of the mail server has been determined first, identifying the highest workload (in terms of email per minute) it is capable of sustaining without refusing any connection. We estimated this limit to be about 1550 mails per minute (i.e., the SMTP server is capable to deliver up to 1550 mails per minute without any error). We therefore performed 29 experiments, with the number of mail per minute ranging from 330 mail/min to 1530 mail/min., increasing by 40 mail/min per experiment.

Table 4.4 reports a summary of these experiments, including the throughput achieved (for both SMTP and POP3 protocols) in the first 4 hours of execution: this value can be assumed as the throughput achieved by the mail server in *"Normal operation"* mode, when the throughput loss due to software aging is not yet appreciable. Each experiment runs from 6000 minutes, and a sample is collected each minute. Fixing the sample collection interval to 1 minute allows us to capture dynamics in resource usage and workload parameters which have a relatively small duration; on the other

Table 4.4: Experiment Summary

EXP. #	WKL (mail/min)	EXEC TIME (min)	NORMAL OPERATION THR. SMTP (KBytes/Min)	NORMAL OPERATION THR. POP3 (Kbytes/min)
1	330	6.001,19	11.105	11.481
2	370	6.002,05	12.430	12.857
3	410	6.000,79	13.808	14.306
4	450	6.001,61	14.885	15.396
5	490	6.001,75	15.913	16.450
6	610	6.001,52	19.602	20.301
7	650	6.001,47	20.789	21.525
8	690	6.001,73	21.803	22.562
9	730	6.002,07	24.042	24.882
10	770	6.002,38	24.980	25.846
11	810	6.001,78	26.050	26.967
12	850	6.001,86	27.082	28.037
13	890	6.002,35	28.736	29.746
14	930	6.002,42	29.656	30.704
15	970	6.001,98	30.698	31.779
16	1.010	6.002,51	31.855	32.998
17	1.050	6.002,72	33.086	34.258
18	1.090	5.469,67	33.932	35.134
19	1.130	5.467,69	34.968	36.179
20	1.170	5.464,97	35.702	36.998
21	1.210	5.463,97	36.340	37.621
22	1.250	5.460,86	37.231	38.522
23	1.290	6.002,51	36.527	37.876
24	1.330	6.002,89	38.742	39.994
25	1.370	6.002,52	39.334	40.718
26	1.410	6.001,98	39.346	40.778
27	1.450	6.003,02	39.726	41.132
28	1.490	6.002,97	42.658	43.949
29	1.530	6.002,75	40.567	42.124

hand, the chosen interval is enough large to avoid noise due to transitory phenomena like garbage collections, which indeed occur several times per minute.

In order to shorten the time required to run all the experiments (3200 hours), 10 identical workstations, with the same configuration, were employed. All the machines used in this analysis (clients and servers) were on a private LAN, in order to avoid any perturbation due to other applications consuming network bandwidth.

4.5.4 Workload Characterization

This section discusses the characterization of the workload applied to the JVM in terms of its workload parameters. A separate characterization is performed for each JVM component.

Class Loader

Class loader activity is mainly is focused in the startup phase of the Java Virtual Machine, 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 just a 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 class-loading activity. In particular, no class loader activity was observed for 17 out of 29 experiments (58,62%). For the remaining 12 experiments, data always assumed an exponential distribution, and the null hypothesis of no trend in data cannot be rejected at a significance level of α=0.01[4] and α=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

During our experiments, although the JVM is requested to do the same actions for the whole duration of the experiment (and therefore it is possible to expect that once critical hot spots are identified JIT-compiler activity progressively decreases), a consistent JIT compilation activity has been noticed throughout the entire experiment.

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

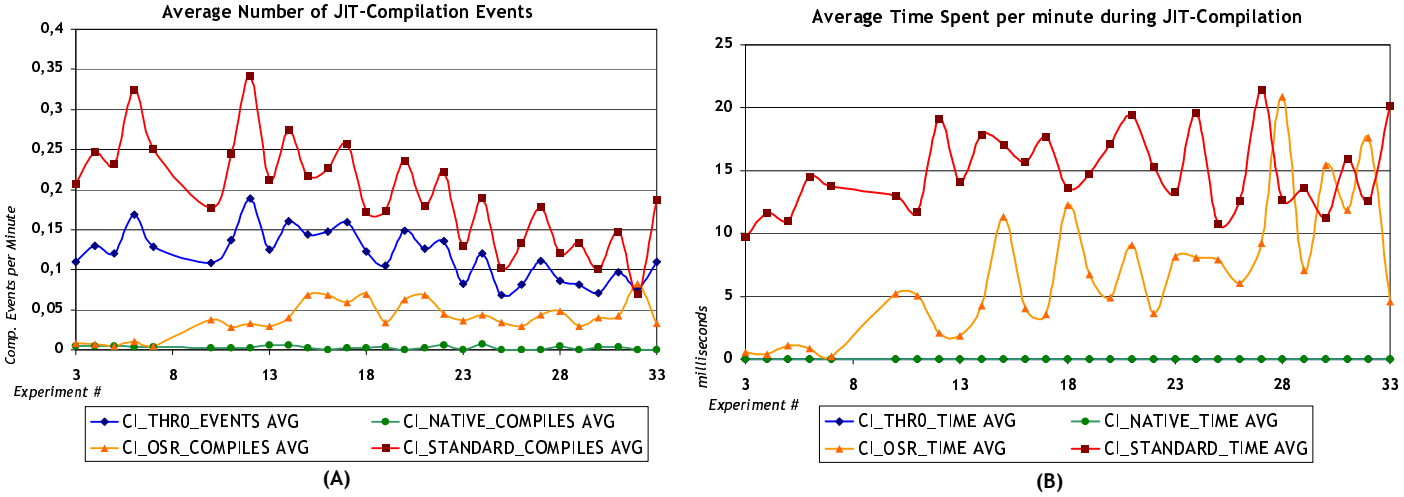

Figure 4.8: A - Average number of JIT compilation events per minute for each experiments; B - Average time per minute spent during JIT compilation for each experiment

For each experiment all JIT-related workload parameters are exponentially distributed, thus excluding the presence of clusters. Moreover, none of the JIT-related workload parameters exhibited trends, except for the ones concerning on-stack replacement compilation (`CI_OSR_COMPILES` and `CI_OSR_TIME`), which experienced a trend at the $\alpha = 0.01$ significance level.

Figure 4.8-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`) 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

Table 4.5: 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_STANDARD_COMPILES_AVG	-0,157	0,101	0,134	0,124
CI_STANDARD_COMPILES_TREND	0,105	0,096	-0,074	-0,422
CI_STANDARD_TIME_AVG	0,000	0,258	0,339	0,012
CI_STANDARD_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

tends to decrease with higher workloads, the number of OSR-JIT-compilation increases with higher workloads.

Figure 4.8-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 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 attributable to the behavior of the parameter describing the time spent for each JIT-compilation (`CI_TIME_PER_COMP`), 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. After applying PCA to these workload parameters, we obtained 4 principal components, reported in Table 4.5. These 4 principal components account for 80.05% of variance in the originall sample set.

Execution Unit and Thread Management Unit

Figure 4.9-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 follows the same pattern, indicating a direct relationship between method execution and synchronization events. All these parameters are normally distributed around their expected value, and they show a negative trend for each experiment, which decreases when the applied workload increases, as shown in figure 4.9-b. This indicates of a loss of throughput during experiment execution, which is

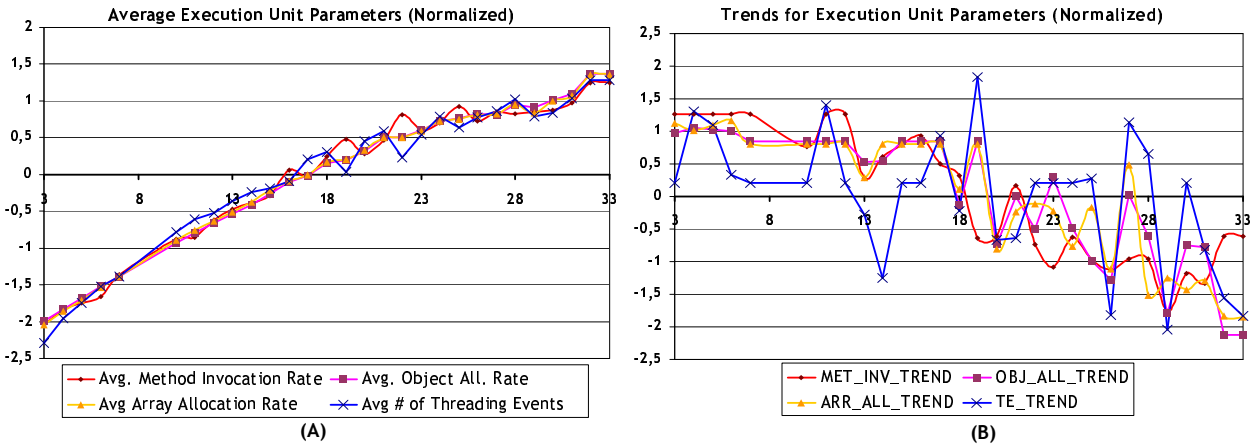

Figure 4.9: A - Average number of execution related events per minute for each experiment; B - Trends detected for execution related parameters for each experiment; normalized data are reported due to significant differences in data order of magnitude.

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 will be excluded from the analysis of the impact of workload on aging trends. Figure 4.9 also shows an high degree of correlation among these parameters: only 2 principal components, reported in Table 4.6 account for 93.82% of variance in data (82.64% for the first component). The high degree of correlation of these workload parameters is proved by the small differences in the coefficients for the first principal component (`EXEC_PC1`).

Table 4.6: 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

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 only 4 experiments exhibited only the normal collection state.

Table 4.7 reports average values for each memory-related parameter in both clusters.

Table 4.7: 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
COLLECTOR0_TIME_PER_INV	2,732	0,147	2,491	3,043	11,118	0,494	9,823	12,452
COLLECTOR1_TIME_PER_INV	128,058	6,769	114,564	139,000	127,627	7,910	117,182	147,661

The *Low collection* state is marked by low garbage collector invocation rates (especially for the tenured generation collector) and longer young generation collection times, whereas the *Normal collection* state is marked by high collection invocation rates and low young generation collection times. Safepoints reached by the JVM are strongly correlated with young generation collections. Instead, no appreciable variation has been observed for the time spent for each tenured generation collection (`COLLECTOR1_TIME_PER_INV`) between normal and low collection periods. Since in *Low collection* state we did not observed significant variation in object and array allocation rates, 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 during normal collection periods is about 3 times higher than the time spent during low collection periods, we did not observe any throughput increase during visits into the low collection state. Figure 4.10 shows the trend exhibited by the 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

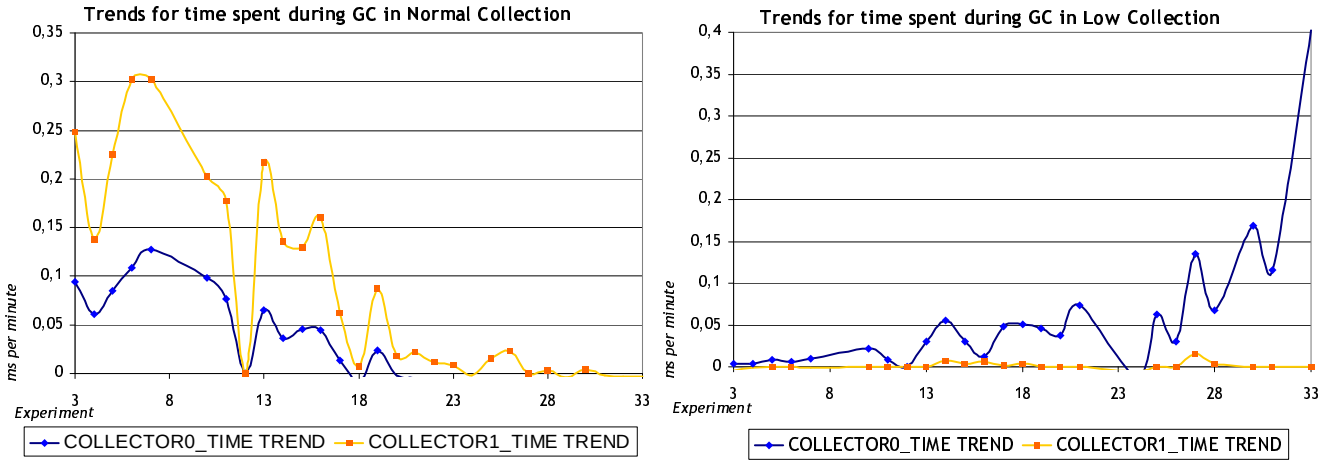

Figure 4.10: Trends for time spent during garbage collection during normal collection periods (A), and during low collection periods (B)

of each collection (`COLLECTOR0_TIME_PER_INV` and `COLLECTOR1_TIME_PER_INV`), which did not show any trend in each experiment. During normal collection periods, there are impressive trends for both collectors when the applied workload is small; the trend then decreases, becoming negligible for experiment with high workloads. Instead, during low collection periods, there are no appreciable trends for the tenured generation collector (the null hypothesis cannot be rejected), whereas the young generation collector trend increases as the applied workload increases. This behavior can be explained taking into account the duration of visits in the two workload states (not shown due to lack of space). Visits in the *Normal Collection* state are shorter when the applied workload is small, whereas visits in the *Low Collection* state are shorter when in applied workload is bigger: therefore transitions from low to normal collection state determine the higher trend in low-workload experiments, and transitions

Table 4.8: 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
COLLECTOR0_TIMEPERINV_AVG	0,071	-0,749	0,011	-0,311	0,437
COLLECTOR1_TIMEPERINV_AVG	0,080	-0,354	0,112	-0,143	-0,092

from normal to low collection periods determine the higher trend in high-workload experiments.

Summarizing, the analysis of these workload parameters tells us that **i)** trend for memory depletion should be determined for both normal and low collection workload states, **ii)** `SM` and `SV` parameters can be excluded from software aging analysis, as well as trends for single collection times, and **iii)** there is an high degree of correlation among the remaining parameters. Therefore we performed a principal component analysis on these parameters in both normal and low collection states, whose result are shown in Table 4.8. 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.

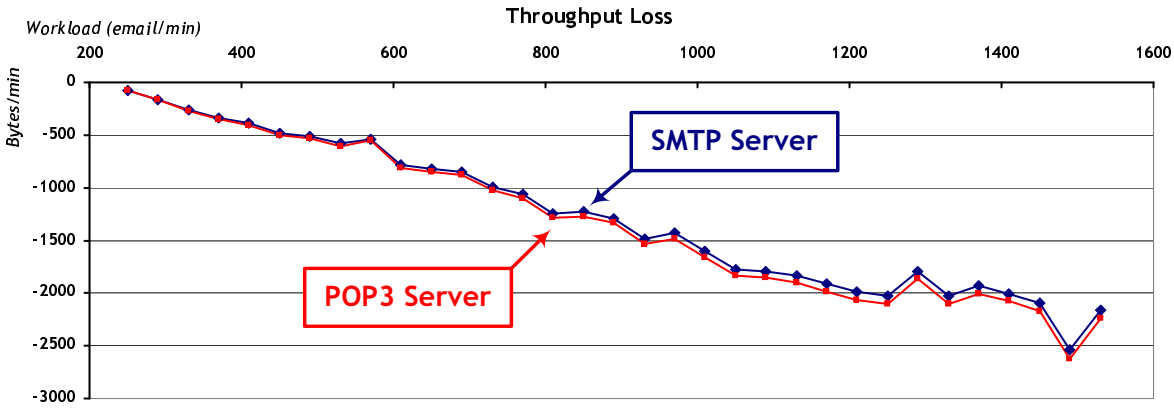

Figure 4.11: Throughput loss trends for SMTP and POP3 servers among different experiments

4.5.5 Throughput Loss analysis

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 0.76 KB/min, 1.56 KB/min, and 1.96 KB/min respectively. Results reported in figure 4.11 highlight the presence of a linear relationship between the throughput loss and the JVM workload. Table 4.9 reports the results of a linear regression analysis applied to throughput loss trends (shown in Figure 4.11). 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 reported the estimated slope, whereas the fourth row reports 95% confidence intervals for the estimation. Increasing the workload by 100 mails per minute causes an increment

Table 4.9: Throughput loss as a linear function of the number of email sent per minute and of the *Normal Operation* throughput

	SMTP Server		*POP3 Server*	
	Mail/min	Normal Operation throghput	Mail/min	Normal Operation Throghput
Student's t	-31,29	-46,35	-31,09	-45,41
Pr > |t|	<.0001	<.0001	<.0001	<.0001
Estimated Slope	-0.174 KB/100 mail/min	-0.067 KB/MB	-0.18 KB/100 mail/min	-0.067 KB/MB
95% Confidence Interval	-0.186 KB/100 mail/min; -0.163 KB/100 mail/min	[-0.07 KB/MB; -0.064 KB/MB]	[-0.192 KB/100 mail/min; -0.169 KB/100 mail/min]	[-0.07 KB/MB; -0.064 KB/MB]

of about 175 bytes in the throughput loss for the SMTP server and of about 185 bytes for the POP3 server. Moreover, if the *Normal Operation* throughput increases by 1MB per minute, it is possible to expect an increment of about 70 bytes in the throughput loss trend for both the SMTP and 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 about the relationships between throughput loss and JVM workload parameters, we performed the multiple regression step of the Software Aging Analysis presented in section 4.4.

Results of such analysis are reported in Table 4.10. The next step of the software aging analysis is concerned with the selection of the principal components which have a real influence on aging trends. We then selected only principal components which showed a probability of being in the tail of t distribution lower than 5%, namely `EXEC_PC_1` and `NORM_COLL_PC2`. Recalling the composition of these principal components, we noticed that while the main contribution to the `NORM_COLL_PC2` principal

Table 4.10: 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

component is mainly due to the duration of each young generation collection (described by the workload parameter `COLLECTOR0_TIMEPERINV`), all execution related workload parameters contributed to the `EXEC_PC1` principal component. Among these parameters, we chose the 2 most representative ones, the average method invocation rate (`MET_INV_AVG`) and the average object allocation rate (`OBJ_ALL_AVG`).

We then evaluate the impact of the 3 above mentioned workload parameters on throughput loss, obtaining results shown in Table 4.11. This impact has been estimated through 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 `COLLECTOR0_TIMEPERINV` 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

Table 4.11: Throughput loss as a linear function of most relevant JVM workload parameters

	SMTP Server		
	OBJ_ALL_AVG	MET_INV_AVG	COLLECTOR0_ TIMEPERINV
Student's t	-35,84	-25,4	-5,17
Pr > |t|	<.0001	<.0001	<.0001
Est. Slope	-0.055 KB/100KAll **	-0.0047 KB/Minv *	-3.002 KB/ms
95% Conf. Int.	[-0.058; -0.052]	[-0.005; -0.0043]	[-4.195; -1.81]
	POP3 Server		
	OBJ_ALL_AVG	MET_INV_AVG	COLLECTOR0_ TIMEPERINV
Student's t	-35,91	-25,46	-5,15
Pr > |t|	<.0001	<.0001	<.0001
Est, Slope	-0.057 KB/100KAll **	-0.0048 KB/Minv *	-3.177 KB/ms
95% Conf. Int.	[-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*

of 0.0047 KB (about 5 bytes) in throughput loss for an increment of 1M in method invocation rate, or an increase of 0.055 KB (about 55 bytes) for an increment of 100K in object allocation rate. These result are not surprising, since the average method invocation rate of the performed experiments is of 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 increases linearly with their values; although regression analysis highlighted a relationship between the 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: collection times are higher since there are more objects in heap area. Since the workload applied to the JVM is mainly I/O bound, the source of the

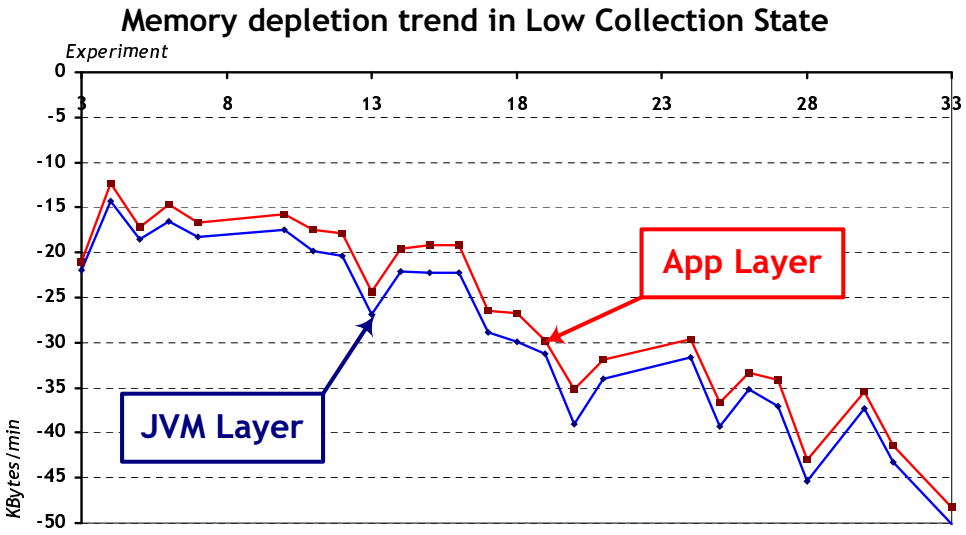

Figure 4.12: Memory depletion trends during low collection periods

throughput loss has to be searched in the `OS abstraction layer` component; indeed the JVM forwards any I/O operation to the underlying operating system through this component. This also suggests that throughput loss values are affected by the host operating system. In order to confirm this hypothesis it is necessary to shift focus from the JVM to the OS layer. In the next chapter an analysis of the relationships between OS workload parameters and aging trends measured at the OS layer will be presented.

4.5.6 Memory Depletion Analysis

Memory depletion trends for the application and the JVM, in the *Low Collection* state, are reported in Figure 4.12. In the *Normal Collection* state (not shown in figure) the aging trend seem 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, we was not able of finding any aging trend for the greatest part of experiments, thus proving that the benchmark application 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 in the *Low Collection* state ranges from 22 days and 4 hours for the best case (-13.96 KB/min) to 6 days and 11 hours for the worst case (-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 bot workload states,it becomes a serious threat during low collection periods. During our experiments the average duration of visits in the *Low Collection* state was of 590 minutes,with a peak of 2234 minutes.

Table 4.12 reports the 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

Table 4.12: Results for partial regression analysis of memory depletion at application and JVM layer against 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

the difference between the two previous trends. These results reveal the following insights:

1. No workload parameter give 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 we did not notice any memory depletion at the application layer.

2. Memory depletion trends at the JVM layer in the normal workload state are attributable 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 Java Heap, the *Native Method Cache*. Size of this area progressively increases during experiment duration.

3. 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.

4. 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 4.5 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, we chose 4 JIT related parameters to estimate the relationships with aging trends in the Normal Collection state and 3 GC related parameters to estimate them in the *Low Collection state*. The results of a linear regression analysis applied to these parameters are reported in Table 4.13. Among the 4 selected JIT-compiler parameters only the time per minute spent in standard 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

Table 4.13: 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
Est. Slope	-0.097 KB/ms	-2.384 KB/comp *	-0.02 KB/ms	-0.0014 KB/ms
95% Conf. Int.	[-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	
Est. 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	
Est. 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]	

the 3 selected Garbage Collection 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 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 appreciable memory depletion trend at the application layer in *Normal Collection* state, and the trend observed at the JVM layer is attributable to the growth of the native method cache size; **ii)** memory depletion is much higher in the *Low Collection* state, and it is attributable to the downfall of garbage collector invocations, and **iii)** the differences between the trends observed at the application and JVM layer are due to the activity of the JIT

compiler.

4.5.7 Key Findings

The conducted campaign revealed that the JVM suffers from Software Aging phenomena, which manifested both as throughput loss and memory depletion. In particular the analysis of collected data proved that:

i) As regards 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 with a very high slope.

iii) The interface between the JVM and the operating system seem to be really critical from a throughput loss perspective. Results presented in this paper showed that, under stressing workload, achieved throughput is halved 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, which constitute the focus of the next chapter, are required to assess whether these aging phenomena are really located in the interface between the JVM and the OS. Summarizing, the experimental campaign highlighted the presence of three dis-

Table 4.14: 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/10^5Alloc.	36 days	5 days
	MET_INV_AVG	-0,0047	KB/10^6Inv.	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

tinct software aging dynamics. Table 4.14 reports, for each of these dynamics, the related workload parameters, the estimation of the relationships between the workload parameter and the aging trend, and the estimated Time To Exhaustion. As regards throughput loss, TTE has been calculated assuming the system failed when the throughput is halved. In particular, table 4.14 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 worses 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 our experiment we observed an average method invocation rate ranging from 170 to 650 millions of method per minute), 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 usually do not last for so much

time, this aging dynamic usually does not 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` [5]. 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 both in the normal and in the low collection state. Therefore, even if TTEs estimated during our experimental campaign do not represent a significant threat to JVM dependability, this dynamic may become a serious source of JVM failure whenever the JIT compilation activity increases significantly. ...

4.6 Conclusions

In this chapter we presented a novel methodology aimed at evaluating the dependability of OTS items from a software aging perspective.

This methodology has been applied to the JVM, conducting of an experimental campaign on a simple OTS-based system constituted by the Apache James mail server, the Sun Hotspot JVM, and the Linux Operating System. Data have been collected employing a self-developed monitoring infrastructure for the JVM, named JVMMon.

[5] JConsole is a GUI tool compliant with the Java Management Extensions which is capable of connecting to a running JVM. The management agent must be started in the monitored JVM by specifying a proper command-line option.

The analysis of collected data allowed us to discover several aging phenomena in the Sun Hotspot JVM, evaluate the relationships between workload and these phenomena, and locate the components where aging-related bugs were located.

Although results presented in the previous section 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 not only to different JVM implementations, but also to each OTS-based system which may be structure in layers as reported in figure 4.1.

"Ogni scarpa addeventa scarpone"
Each new shoe eventually becomes a tatty one.

Neapolitan Proverb

Chapter 5

Sensitivity Analysis of the OS layer against Aging Faults

In the previous chapter software aging phenomena at the JVM layer were deeply analyzed. We found that aging in the JVM manifests both as memory depletion and throughput loss. We addressed the interface between the JVM and the Operating System as the source of throughput loss phenomena. In particular, since the application running on top of the JVM imposed an I/O bound workload, we claimed I/O activities performed by such interface to be the root cause of the observed throughout loss. In order to confirm this hypothesis, it is necessary to analyze software aging behavior at the OS level. In this chapter we analyze the relationships between OS-level workload parameters and aging phenomena as measured at the OS layer. The analysis has been performed on a different testbed, both for Windows and Linux.
Results of such analysis are significantly different from the ones presented in the previous chapter: no throughput loss was observed, whereas a consistent memory depletion has been observed for the Windows OS. This suggests us that previously observed throughput loss was due the particular combination of JVM version, OS system libraries, OS kernel and also hardware platform.

5.1 Motivations

The analysis presented in the previous chapter allowed us to thoroughly characterize the development of Software Aging phenomena at the JVM layer. Results of the analysis clearly showed the JVM suffers from Software Aging. Aging-related bugs

manifested both as memory depletion and throughput loss.

As far as memory depletion is concerned, we found two distinct contributions to this phenomenon: a slow drift due to the activity of the Just-In-Time compiler, and a fast drift due to sudden downfalls in garbage collection frequency. This means that JVM components are responsible for the observed depletion trends in JVM heap. On the other hand, we observed a strict correlation between throughput loss and workload parameters related to the execution unit, such as the *Method Invocation Rate* and the *Object Allocation Rate*. We did not observe an higher throughput loss when the JIT activity was higher. Moreover, since the employed benchmark is a pure-java application, it does not execute any user-defined native method. Therefore we claimed that the source of this aging phenomena has to searched in the interface between the JVM and the underlying OS.

In order to validate this hypothesis, it is required to shift focus from the JVM layer to the OS layer, and perform a new data analysis taking into account OS-level workload parameters (such as system call invocation frequency, number of bytes read per second, or number of bytes written per second), and evaluating their relationship with throughput loss and memory depletion trends observed at the OS layer.

Beyond confirming or rejecting our hypothesis about throughput loss source, an OS-level analysis will be useful to investigate whether there are memory depletion phenomena introduced in the OS layer which are not observable in the JVM layer. Indeed

the JVM provides a complete abstraction of system memory: Java applications do not see other memory than the one available in the heap area. It may therefore happen that memory depletion phenomena introduced by the JVM are not observable monitoring only memory usage in the JVM heap.

Moreover, in order to augment the effectiveness of an OS-level analysis, it will be worth to collect data from different Operating Systems. In this way it will be possible to compare the development of aging phenomena in different Operating Systems, thus allowing to choose the best operating system taking into account not only performance aspects, but also dependability ones.

In order to perform such comparison, particular carefulness is required for workload parameters and resource usage variables selection. It is indeed necessary to select workload parameters which are comparable among different operating systems, since there should be Operating System specific information which are not comparable with other Operating Systems or require a conversion in order to be compared.

This chapter presents the results of an experimental campaign aimed at analyzing the development of aging phenomena in the OS layer of the same simple OTS-based system employed in the previous chapter. The analysis has been performed using the methodology presented in the previous chapter. The next section discusses experimental design, whereas section 5.3 describes the testbed used for this experimental

campaign. The problem of monitoring OS-level workload parameters has been addressed in section 5.4. Section 5.5 discusses the results of such experimental campaign, reporting results for both the Windows and the Linux Operating System. Finally, section 5.6 concludes the chapter discussing the key findings of this analysis.

5.2 Design of Experiments

As for the JVM level analysis, the number of e-mails sent per minute has been chosen as the controllable parameters for our experiments. The workload applied to the James Mail Server ranges from 150 mail/min to 600 mail/min. We was not able to perform experiments with higher workload since the mail server capacity test on the Windows OS revealed that the James Mail Server, on this Operating System, was capable of delivering no more than 650 mail/min without refusing any connection. For this analysis we performed 4 experiments for both Linux and Windows, increasing the number of mails for each experiment by 150 mail/min.

Each experiment runs from 15000 minutes, and a sample is collected each 2 minutes. The duration of each experiment has been consistently increased with regards to the JVM-level analysis in order to highlight the presence of particularly slow aging trends which may be not appreciable with 6000 minutes experiments, and to evaluate whether there are aging phenomena which start to develop after a considerable execution time is elapsed. Moreover, since the JVM heap, which represents the greatest part of the memory used at the OS layer, grows (shrinks) only when heap memory

usage goes above (below) certain thresholds, we chose a larger interval for sample collection, thus reducing the time required to perform the subsequent analysis.

We considered the same response variables employed in the JVM level analysis: throughput and memory usage. In particular, rather than measuring system-wide memory usage, i.e., total memory allocated by the OS, we measured memory actually used by the `java` process, thus allowing to discard transient phenomena due to the concurrent activity of other process, which, especially in the Windows OS, is unavoidable.

As far as workload parameters are concerned, unlike the JVM level analysis, we restricted our focus only to I/O workload parameters. This choice has been made since our original goal was to confirm or reject the hypothesis on the relationships between throughput loss and I/O activity.

Among the broad spectrum of workload parameters concerning Operating System activity, only 2 workload parameters were selected: *Total number of bytes written per minute (B_W)*, and *total number of bytes read per minute (B_R)*. Given the meagre number of workload parameters, the PCA step of the workload characterization may be skipped, since there is no need to perform reduce the number of independent variables in the regression model. The values collected for B_R and B_W must concern only the I/O activity performed by the `java` process, in order to avoid perturbation due to I/O activity performed by system processes (such as the `syslogd` process in Linux

Table 5.1: Experimental design parameters summary

Controllable Parameter	Response Variables	Observable Parameters
Number of mails per minute	Java process memory usage *(KBytes)*	B_R: Bytes Read per minute *(KBytes/min)*
Levels: 150, 300, 450, 600 mail/min	Application throughput *(KBytes/min)*	B_W: Bytes Written per minute *(KBytes/min)*

or the `svchost` process in Windows).

Table 5.1 summarizes the levels chosen for the controllable workload parameter,the selected response variables, and the selected observable workload parameters.

5.3 Monitoring OS workload parameters

Since JVMMon is capable of capturing data only about JVM state and workload, it has been necessary to find a solution to monitor resource usage and I/O workload parameters at the OS level. This section describes the solutions employed to monitor these information in Windows and Linux.

5.3.1 Windows OS

As far as the Windows Operating System is concerned, both resource usage and workload parameters have been monitored using performance counter exposed by the *Windows Management Instrumentation (WMI)* [83]. WMI is a technology based on the *Web-Based Enterprise Management (WBEM)* standard. Its architecture, shown in figure 5.1, is composed of four components: *Management Applications,WMI infrastructure, data providers, and managed objects.* Developers typically must target management applications, such as `perfmon`, to collect data from specific objects. An

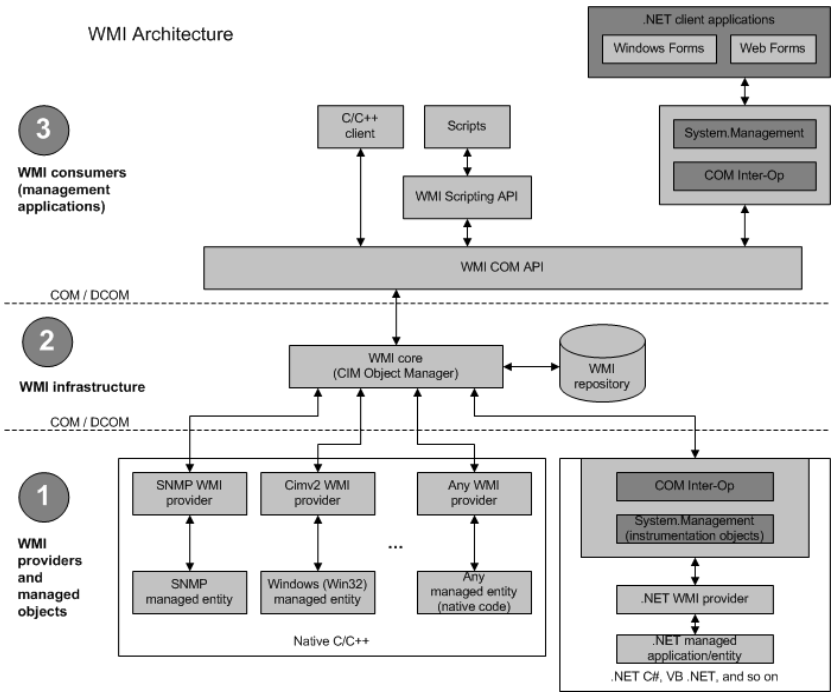

Figure 5.1: WMI architecture

object might represent a component, such as a network adapter device, a system service such a web server, and an OS kernel element such a windows process or thread. Providers need to define and export the representation of the objects that management applications are interested in.

The WMI infrastructure, the heart of which is the *Common Information Model (CIM) Object Manager (CIMOM)*, is the glue that binds management applications and providers. As part of its infrastructure, WMI supports several APIs through which management applications access object data and providers supply data and class definitions. To collect data about system resources, we access the managed object related to the `java` process and sample performance counters related to the

memory actually committed to the process and the virtual memory it has been assigned. Since we did not have any managed object for the James Mail Server, we had to measure throughput by analyzing logs produced by the load generator.

The same managed object has been employed to collect workload data. Samples collected from WMI performance counters are stored in a log using the *Performance Logs and Alerts* service offered by the Windows OS.

5.3.2 Linux OS

Although the Linux OS lacks a sophisticated management infrastructure like WMI, the `proc` virtual file system contains many information about the state of processes and threads.

The `proc` filesystem contains a virtual filesystem. It does not exist on a disk. Instead, the kernel creates it in memory. It is used to provide information about the system. In the root directory of this filesystem it is possible to found several files describing the state of the system, and a subdirectory for each active process.

In order to collect data about system resources we developed an `awk` script periodically parsing `status` file of the `java` process, and extracting data about memory usage from this file.

Unfortunately, the Linux file system does not provide any mechanism to monitor I/O activity performed by a single process. Indeed utilities such as `iostat` return data related to I/O activity of the whole system.

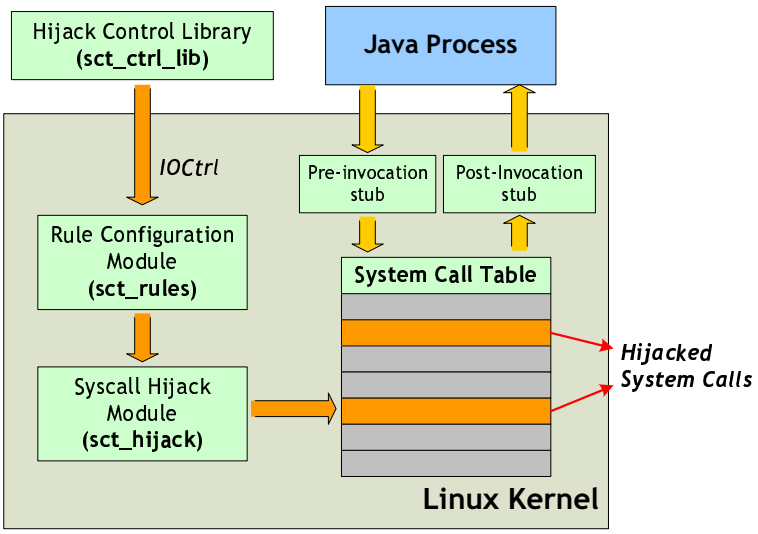

Figure 5.2: SysCallTrack architecture

Each Linux process performs I/O operation, either on a socket on a file, through the `sys_write` and the `sys_read` system calls. Therefore, in order to monitor I/O activity of the `java` process, these system calls were hooked via the `syscalltrack` [84] system call hijacker. `SysCallTrack`, whose architecture is depicted in figure 5.2, is composed by 2 kernel modules and a user-space control library. The `sct_hijack` module perform the actual hijacking of the system call by replacing entries in the system call table with the address of the hijacking stub. The `sct_rules` module control the previous module realizing the injection of hijacking rules specified by to the user by means of the `sct_ctrl_lib` library. An hijacking rule is composed by **1)** the name of the system call to hijack, **2)** a mnemonic rule name, **3)** a filter expression (e.g. the PID of the process whose system call invocation must be hijacked), and **4)** an action, which could be `LOG` or `FAIL`. Once system calls have been hijacked, each

system call invocation is redirected to a pre-invocation stub, which in turns calls the actual system call; before returning to the user-space process, a post-invocation stub is called, as depicted in figure 5.2.

In order to monitor I/O activity of the `java` process, rules for hijacking the `sys_read` and the `sys_write` system calls were defined with a proper filter. Since the number of bytes read or written is one of the parameters of these system calls it is logged whenever the system call is invoked. A simple application counts bytes read and written and writes these information in another log upon sample collection.

5.4 Experimental Setup

The methods described in the previous section were employed to monitor OS-level resource usage data and workload parameters in the same simple OTS-based system used in the previous chapter. This system is constituted by a pure-java mail server, running on top of a Sun Hotspot JVM. As for the JVM level analysis, the JVM was started with the typical `server` configuration.

Experiments were executed on an Intel Pentium4 workstation at 3.4GHz, equipped with 2 GB RAM; the Sun Hotspot JVM version 1.5.0_09 has been employed for both Windows and Linux experiments. Windows XP Service Pack 2 has been employed for Windows Experiments, whereas the Linux experiments have been performed on a Fedora Core 5 distribution with kernel v.2.6.11.12. We was not able to use the same kernel version employed for JVM-level analysis due to technical problems with the

system call injector.

5.5 Experimental Results

Quite surprisingly, we obtained significantly different results with respect to the ones obtained in the JVM analysis. Throughput loss was never observed, neither in Windows or Linux. Bytes sent and received kept a constant average during the whole duration of the experiment, with a much lower variance.

Moreover, while we still observed transitions from the *Normal Collection State* to the *Low Collection State* in the Linux OS, we did not observed any of these transitions in the Windows OS, where the garbage collection frequency is almost constant. This means that the behavior which generated the fast memory depletion drift in JVM-level analysis depends on some Linux-specific issues of the garbage collector. However, since in this chapter we are interested in aging phenomena which develop at the OS layer and we already pointed out that this behavior is confined into the JVM heap, we will not discuss it in detail.

As regards memory depletion at the OS layer, a totally different behavior has been observed between the two Operating Systems. By analyzing memory usage of the `java` process, no aging trend has been detected in Linux, since it was not possible to reject the null hypothesis for the Linux OS, whereas a non-negligible aging trends has been observed for the Windows OS, in spite of actions aimed at reducing physical memory used by the `java` process automatically performed by the Operating System.

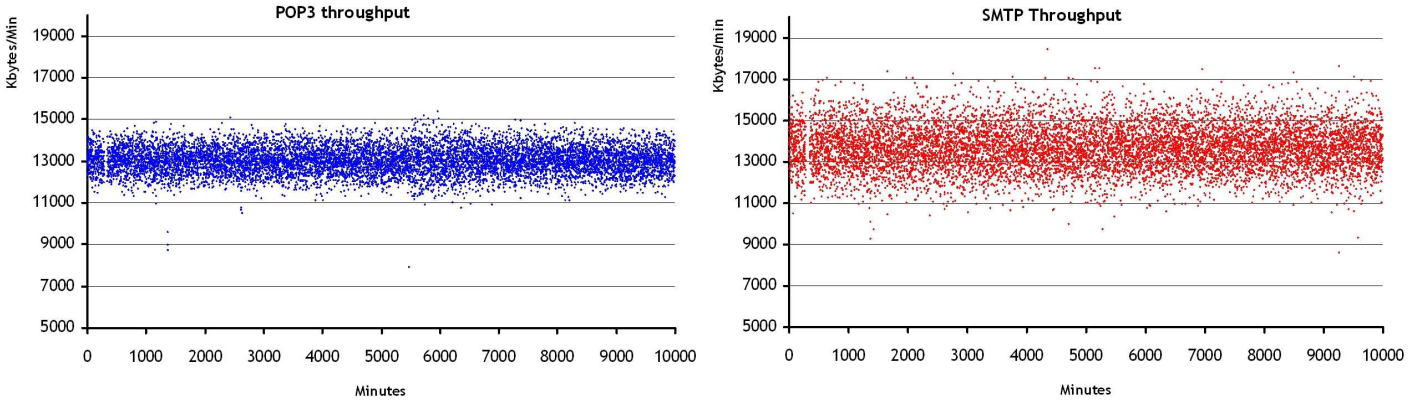

Figure 5.3: Throughput achieved during experiment 3 (450 mail/min) in Windows

In the following experimental results are discussed in detail for each Operating System, analyzing exhibited aging trend, and then characterizing I/O workload parameters; finally, the relationships between measured aging trends and such workload parameters are analyzed.

5.5.1 Windows

Throughput loss was never observed in the 4 performed experiments. The null hypothesis of no trend in data cannot be rejected either at the $\alpha = 0.05$ and $\alpha = 0.01$ significance level. Figure 5.3 reports the development of bytes sent and received for the third experiment, performed with a workload of 450 mail/min. It is therefore possible to exclude the presence of this kind of aging phenomena.

As regards memory depletion,WMI provides 3 different counters to measure process system memory usage: *Virtual Bytes*,*Private Bytes*, and *Working Set*. The *Working*

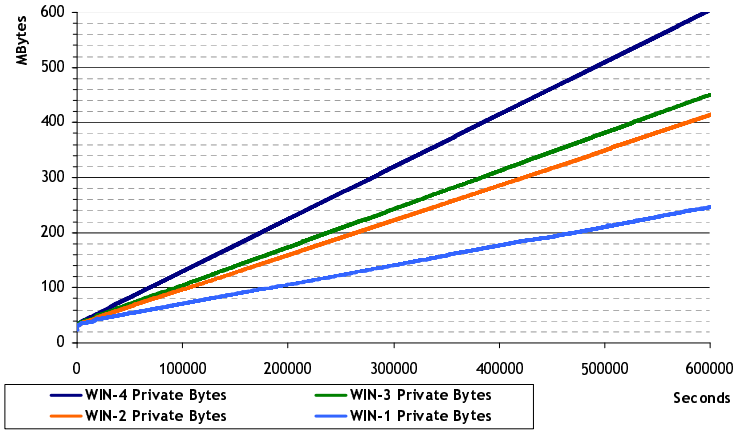

Figure 5.4: Private Bytes measured during Windows Experiments

Set counter reports the amount of memory currently allocated to the process, where the *Private Bytes* counter reports the amount of memory in memory pages exclusively assigned to the process. The *Virtual Bytes* counter instead have not been taken into account, since the fact that a process is assigned with a certain amount of virtual memory does not imply that this memory is entirely committed to the process. .

Experimental results clearly show a constant increasing trend for the *Private Bytes* counter. The graph reported in figure 5.4 shows the trend of this counter for each experiment, and table 5.2 reports the entity of such trend. The corresponding Time-To-Exhaustion depends upon the amount of available system memory and the minimal amount of memory the Operating System requires in order to work correctly. For instance, for a machine equipped with 2GB RAM memory and requiring about 100MB for the OS, the estimated TTE ranges from about 67 days for the first experiment (150 mail/min), to about 25 days for the fourth (600 mail/min).

Table 5.2: Intra-experiment characterization for *Private Bytes* and *Working Set* in Windows

	Private Bytes (Mbytes)			Working Set (Mbytes)		
	AVG	*DEV.ST*	*TREND (KB/min)*	*AVG*	*DEV.ST*	*TREND (KB/min)*
EXP-1 Windows	136,47	59,12	20,56	138,47	59,17	20,58
EXP-2 Windows	216,33	106,57	37,06	101,10	59,25	37,84
EXP-3 Windows	235,81	116,59	40,55	237,90	116,66	40,58
EXP-4 Windows	310,44	159,90	55,61	312,52	160,00	55,65

The *Working Set* counter shows behavior similar to the one exhibited by the *Private Bytes* counter, except for experiment 2 in which we found a typical saw-tooth trend. This behavior can be explained recalling the way in which the Windows OS manages working sets: memory pages are left in the working set of the process even if they are not actually used by the application as long as system free physical memory does not go below a particular threshold; once this threshold is overshot, less frequently used pages are swapped out thus freeing some physical memory; in this way the Windows OS performs a sort of "spontaneous rejuvenation". Table 5.2 reports also trends (calculated as a weighted average of the trend exhibited for each segment of the saw-tooth graph) for the *Working Set* counter. Trends estimated using this counter are also very similar than the ones measured using the *Private Bytes* counter, as well as the corresponding TTEs.

Once aging trends have been estimated, the next steps of our analysis deal with the characterization of I/O workload parameters and the evaluation of their influence on memory depletion trends.

The intra-experiment characterization of the two considered workload parameters,

Table 5.3: I/O workload parameters characterization

	B_R (Kbytes/min)			B_W (Kbytes/min)		
	AVG	*DEV.ST*	*RATIO*	*AVG*	*DEV.ST*	*RATIO*
EXP-1 Windows	26181,18	3376,85	12,90%	21291,75	2376,40	11,16%
EXP-2 Windows	46214,18	1933,31	4,18%	40569,71	1489,36	3,67%
EXP-3 Windows	63319,00	4990,29	7,88%	53266,48	3722,36	6,99%
EXP-4 Windows	91458,63	3280,69	3,59%	80927,51	2677,48	3,31%

B_W and B_R, reported in table 5.3, reveals that I/O activity of the `java` process is almost constant for the whole duration of the experiments. Moreover, standard deviation is below 15% of the average value for each experiment, and over 10% only for the first one. For this reasons, it arises that these parameters may be considered as controllable workload parameters. Unfortunately, since only 4 experiments have been performed, factorial design and multiple regression are not employable, since there is a serious risk of obtaining a type II error [1]

Nevertheless, useful insights about the relationships between I/O activity and aging trends may still be obtained by analyzing the Pearson's correlation index. Pearson correlation assumes that the two variables are measured on at least interval scales, and it determines the extent to which values of the two variables are proportional to each other. The value of the correlation (i.e., the correlation coefficient) does not depend on the specific measurement units used.

The values found for the Pearson Correlation index are: `0.9728` **for the** B_W **parameter, and** `0.9712` **for the** B_R **parameter**. It is therefore possible to claim that

[1]Type II errors occur when researchers conclude in favor of the Null Hypothesis, when in fact the other one is true.

aging trends are strictly linked to I/O activity measured at the Operating System layer. However, it is important to remark that:

- Although this correlation index proves that there is a strong dependence of aging trends upon I/O activity, it does not imply that there exist a cause-and-effect relationship between I/O activity and aging trends
- This correlation index may return values close to 1 even in presence of non-linear relationships. More experiments are required to assess whether the detected relationships are linear or not.

5.5.2 Linux

As for the Windows analysis, throughput loss has never been observed in the 4 experiments. The null hypothesis of no trend in data cannot be rejected either at the $\alpha = 0.05$ and $\alpha = 0.01$ significance level. These results contrast with the ones found for the JVM level analysis performed in the previous chapter, in which a consistent throughput loss was observed in a JVM running on top of the Linux Operating System. In the following section we will discuss in detail this behavior.

As for memory depletion, scripts described in section 5.3 extract two counters from the `status` file in the `proc` filesystem: `VmSize` and `RssSize`. Since the `VmSize` reports the amount of virtual memory assigned to the process, it will not be taken into account. On the other hand the `RssSize` reports the amount of physical memory used

by the process, and is therefore equivalent to the *Working Set* counter monitored in Windows.

The analysis of the `RssSize` counter revealed the presence of a slight aging trend for 3 out of 4 experiments; it was not possible to reject the null hypothesis of no trend in data for the third experiment (450 mail/min). Moreover, memory depletion trends measured for the remaining experiments resulted in a Time-To-Exhaustion ranging from more than 100 years for the first experiment to about 10 years for the fourth. It is therefore possible to claim that memory depletion at the OS layer, in Linux, is negligible. These result do not contrast with the ones found in the JVM-level analysis: memory depletion trends detected in the previous chapter are related to Java Heap Usage, whereas the ones measured in this chapter are related to system memory usage of the `java` process. Since the JVM allocates memory required for its heap area upon its startup, by measuring used memory at the OS layer it is not possible to see how much of the heap area is actually used for Java objects. Memory depletion phenomena, such the ones which due to low collection periods, may happen without being detected at all at the OS layer.

5.6 Conclusions

The original goal of this OS-level analysis was to confirm whether I/O activities performed by the interface between the JVM and the underlying Operating System was the root cause of the observed throughout loss.

Unfortunately, in this new experimental campaign, we did not observe any throughput loss. This result is particularly surprising especially for the Linux OS, since the previous experimental campaign, performed on this operating system, revealed the presence of a consistent throughput loss. Such a different behavior can be explained only considering that the two experimental campaign have been performed using different versions of the OS kernel. In the JVM-level campaign experiments were performed running Linux kernel v.2.6.16, whereas in the OS-level campaign the JVM ran on Linux kernel v.2.6.11.12. A different version has been employed due to technical problem on system call hijacking with the newer version of the kernel.

However, even if it is not possible to state anything about the relationships between throughput loss and I/O activity, this apparently perplexing results indirectly confirm our hypothesis. Indeed, since throughput loss has been removed by changing the version of the operating system kernel, this aging phenomenon was necessarily caused by the interface between the JVM and the operating system.

As far as memory depletion is concerned, we was not able to found any significant OS-level trends in the Linux Operating System, whereas we noticed a consistent memory depletion trend in the Windows OS. In particular we observed a strong correlation between the measured trends and I/O workload parameters, measured as number of bytes read (written) per minute. I/O activity may be therefore addressed as the root cause of memory depletion at the Operating System layer in Windows.

Summarizing, it is possible to conclude that:

- The interface between the JVM and the Operating System is a potential source of aging phenomena which evolve independently from the aging phenomena detected at the JVM layer.

- The integration of several OTS items may have a significant impact on the development of aging phenomena. Indeed using the same JVM implementation of two different versions of the Linux Kernel, we obtained deeply different results regarding throughput loss.

- Linux seem to be more reliable than Windows from a software aging perspective. While the former does not introduces any aging phenomenon at the OS layer, the latter introduces a consistent memory depletion trend. This trend may become a serious concern when relevant I/O bound workloads are applied.

"Chi ha avuto, ha avuto, ha avuto... chi ha dato, ha dato, ha dato... scurdámmoce 'o ppassato, simmo 'e Napule paisá!"

Neapolitan Song

Conclusions

This dissertation addressed Software Aging issues in OTS based systems. Software aging progressively lead to service failure due to excessive performance degradation or transient failures. These phenomena are due to the activation of a particular class of software faults, called *Aging-Related bugs* (see section 1.2). Typical examples of such faults are unreleased memory regions and file handles.

It has been shown that Software Aging is one of the prominent causes of failures for a consistent number of software systems, including also systems employed in critical scenarios. Due to their nature, software systems based on Off-The-Shelf items are more exposed to Software Aging Phenomena. Since OTS items often lack proper testing, several aging bugs may reach the operational stage; moreover interactions between different OTS items may have unpredictable effects which may lead to the failure of the whole software system.

Since existing techniques and methodologies aimed at detecting and estimating aging phenomena are not capable of thoroughly characterize the development of these phenomena in OTS-based system, this thesis proposed a novel approach to evaluate

aging trends and estimate their relationships with workload.

In brief, the proposed methodology, described in chapter 4, assumes that an OTS-based system may be divided into several layers, in which application-specific business logic is at the top layer and the operating system is at the bottom one; each of these layers may include one or more OTS items. The analysis process may then be split into two steps: the first one deals with workload characterization, whereas the second one is concerned with the estimation of aging trends and their relationships with workload parameters.

Given an OTS-based system and a particular OTS item chosen as the target of the analysis, this approach allows to:

1. Identify in which component(s) aging phenomena are introduced;

2. Select only workload parameters which are more relevant to the development of aging phenomena, and estimate the linear dependence, if any, between them and aging trends.

This approach has been employed to characterize software aging phenomena inside the Java Virtual Machine, which has been adopted as a case study throughout the dissertation.

Prior to performing any experimental campaign, a preliminary characterization of the failure behavior of the JVM has been performed by analyzing failure reports contained in bug databases. Results of this analysis showed that a non-negligible

percentage of JVM failures was attributable to software aging phenomena. However, these preliminary results did not allowed us to state anything about the development of software aging inside the virtual machine. For this reason, a massive experimental campaign on real world testbeds, 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. The analysis of collected data revealed the presence of three distinct aging dynamics. The first one manifests as throughput loss and is mainly dependent on execution activities performed by the JVM; the second one manifests as memory depletion, and is mainly attributable to activities performed by the Just-In-Time compiler; finally, also the third dynamic manifests as memory depletion, but it is mainly attributable to sudden downfalls in garbage collection frequency.

After that, the same approach has been adopted to study aging phenomena introduced in the interface between the JVM and the underlying operating system as a function of the operating system. In this case, the analysis of collected data revealed the presence a consistent memory depletion trend in the Windows OS; this trend is mainly attributable to I/O activity performed by the application running on top of the JVM.

In both cases, the approach presented in this dissertation allowed us not only to detect the presence of aging phenomena, but also to locate its source, and identify its

relationships with applied workload.

In the rest of this final chapter, we discuss lessons learned from the work presented in this dissertation, and also directions to improve the dependability of the JVM against aging-related bugs.

Lessons learned

The value of failure reports

As discussed in the first chapter, several methodologies and techniques have been developed in order to assess the dependability of a software system. Each of these techniques is concerned with a particular dependability aspect; for instance Robustness Testing is aimed at evaluating the robustness of a system or a component with respect to invalid or unexpected inputs, whereas Software Aging Analysis is aimed at detecting and estimating software aging phenomena.

Given a system whose dependability has to be assessed, failure report analysis represents the easiest and fastest solution to achieve a preliminary dependability characterization, since it allows to obtain insights about the behavior of the system which may be used either to infer conclusions about its failure behavior, and to choose the proper technique to adopt in order to perform a detailed assessment of the dependability of the system.

In chapter 3, a preliminary characterization of the dependability behavior of the Java

Virtual Machine has been performed by analyzing failure reports extracted from publicly available bug databases. Despite of the their qualitative nature, failure reports allowed us to extract very useful insights about the dependability of the Java Virtual Machine.

As an example, we found that almost half of JVM failures (45%) are due errors not detected by JVM built-in error detection mechanisms, and that the JVM attains different reliability levels on different Operating Systems. Finally, we found that a consistent percentage of failures with non-deterministic behavior (about 15%) occurred with a daily or weekly frequency when relevant workloads are applied. Therefore the preliminary analysis of failure reports addressed software aging as a relevant source of failures for the JVM, thus suggesting a detailed analysis of software aging phenomena inside the JVM.

On system monitoring

In order to perform an effective measurement-based software aging analysis, a particular care should be placed toward field data collection, which should be aimed at assessing the current health of the system rather than focusing exclusively on failure reporting. Current health system should be described in terms of its internal state and applied workload.

Unfortunately, existing JVM monitoring tools usually are not capable of collecting enough data to characterize system state and workload. To this aim, an ad-hoc

monitoring tool for the JVM, named JVMMon (presented in chapter 4), has been developed. JVMMon allows to collect data about the current state of internal JVM components by intercepting relevant virtual machine events, and to collect workload information by periodically sampling JVM performance counters.

The results of the JVM level analysis showed that such a monitoring tool can provide a more thorough picture of aging phenomena and their relationships with JVM workload parameters.

On experimental campaigns

A key point of the methodology presented in chapter 4 is that experiments are performed varying the level associate with a controllable workload parameters. In the experimental campaigns discussed in section 4.5, and chapter 5, the workload imposed to the JVM has been controlled by manipulating the number of e-mails sent per minute. In this way a constant workload is imposed on the JVM throughout the entire experiment.

By performing experiments in this way we managed to perform a statistical characterization of JVM workload parameters for each experiment. An overall figure of the evolution of workload parameters was then drawn by relating synthetic data extracted for each experiment. As regards aging trends, performing each experiment with a different level of a controllable factor allowed us to describe aging trends as a function of the workload applied on the Java Virtual Machine. Finally, by relating aging trends

and workload parameter figures, we were able to provide a detailed characterization of the development of software aging dynamics in the JVM.

These remarks suggest that a proper planning of the experimental campaign may significantly improve the effectiveness of the analysis. When dealing with systems in the operational stage, data must be collected only using the real workload imposed on the target system: there is no chance to manipulate any workload parameter. Although workload characterization is feasible also with operational systems (as done in [61]), our experimental campaign proved that performing experiments by varying a controllable workload parameters allows to obtain a more thorough and general characterization of aging phenomena.

On Software Aging in the JVM

The results presented in section 4.5 and in chapter 5 clearly indicate that the JVM is affected by software aging phenomena. These results were quite unexpected, since the JVM has been designed to reduce effects of aging phenomena. It is be sufficient to recall the garbage collector, which frees developers from manually handling memory management, which in turn is the most important source of "aging bugs" due to memory leaking or bloating.

Experimental results showed that software aging in the JVM manifests both as throughput loss and memory depletion.

As regards throughput loss, in the JVM-level analysis we found a consistent aging

trend, ranging from 0.08 KBytes per minute to 2.48 KBytes per minute. This trend is mainly dependent on execution unit activity: indeed we found that it is strongly correlated with workload parameters such as method invocation frequency and object allocation frequency. Moreover, this aging phenomenon disappeared when we performed the OS-level analysis, thus suggesting us that it was dependent on the particular combination of JVM implementation and Operating System employed in the JVM-level analysis. In other words, we observed that the aging related bug causing this phenomenon was removed by simply changing the version of the Linux kernel employed in the testbed, whereas we did not notice any throughput loss in the windows OS. These remarks prove that aging phenomena may arise as a consequence of the integration of several OTS items.

On the other hand, memory depletion phenomena were detected both in the JVM-level and in the OS-level analysis. In the JVM-level analysis we detected two distinct contributions to memory depletion. The first contribution manifests as a slow, but constant, drift (ranging from 0.94 KBytes/min to 3.06 KB/min) and is mainly due to the activity of the JIT-compiler. The second contribution is due to sudden downfalls in garbage collector activity and manifest as a fast drift (ranging from 13.96 KBytes/min to 48.96 KBytes/min).

These aging phenomena may be detected and treated as follows:

- As regards the slow drift, the JVM may be properly configured in order to avoid

excessive JIT compilation. In particular, we found that On-Stack-Replacement compilations are the most expensive ones in terms of required memory;

- As regards 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.

Moreover, in the OS-level analysis we noticed a further memory depletion trend. This phenomenon affects memory required by the `java` process and it is not visible at the JVM level, since it does not affect java heap. It is strictly OS-dependent, since it was observed only in the Windows OS, and measured aging trends range from 20.56 KB/min to 55.61 KB/min. By analyzing its relationships with workload parameters, we found that this aging phenomenon is mainly dependent on I/O activity performed by the JVM. This remarks suggest us that, at least for I/O bound applications, the Linux OS offers a more reliable environment with regards to the Windows OS from a software aging perspective.

Bibliography

[1] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] Y. Huang, C.M.R. Kintala, N. Kolettis, and Fulton N.D. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena,CA (USA)*, pages 381–390, 1995.

[3] Yujuan Bao, Xiaobai Sun, and Kishor S. Trivedi. Adaptive software rejuvenation: Degradation model and rejuvenation scheme. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA*, pages 241–248, 2003.

[4] M. Grottke, L. Li, K. Vaidyanathan, and K.S. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55(3):411 – 420, Sept. 2006.

[5] Sachin Garg, Antonio Puliafito, Miklos Telek, and Kishor S. Trivedi. Analysis of Preventive Maintenance in Transactions Based Software Systems. *IEEE Transactions on Computers*, 47(1):96–107, 1998.

[6] Meera Balakrishnan, Antonio Puliafito, Kishor S. Trivedi, and Yannis Viniotis. Buffer losses vs. deadline violations for ABR traffic in an ATM switch: A computational approach. *Telecommunication Systems*, 7(1-3):105–123, 1997.

[7] Dazhi Wang, Wei Xie, and Kishor S. Trivedi. Performability analysis of clustered systems with rejuvenation under varying workload. *Performance Evaluation*, 64(3):247–265, 2007.

[8] E. Marshall. Fatal Error: How Patriot Overlooked a Scud. *Science*, 255:1347, 1992.

[9] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K.S. Trivedi. A methodology for detection and estimation of software aging. In *Proc. of The 9th International Symposium on Software Reliability Engineering (ISSRE 1998), Paderborn, Germany*, page 283, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[10] K. Vaidyanathan and K.S. Trivedi. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proceedings of 10th International Symposium on Software Reliability Engineering (ISSRE 1999), Boca Raton, USA*, pages 84–93, 1999.

[11] Kishor S. Trivedi, Kalyanaraman Vaidyanathan, and Katerina Goseva-Popstojanova. Modeling and analysis of software aging and rejuvenation. In *Annual Simulation Symposium*, pages 270–. IEEE Computer Society, 2000.

[12] Rivalino Matias and Paulo J. F. Filho. An experimental study on software aging and rejuvenation in web servers. In *30th Annual International Computer Software and Applications Conference (COMPSAC 2006), 17-21 September 2006, Chicago, Illinois, USA*, pages 189–196. IEEE Computer Society, 2006.

[13] Gunther A. Hoffmann, Kishor S. Trivedi, and Miroslaw Malek. A best practice guide to resources forecasting for the apache webserver. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*, pages 183–193. IEEE Computer Society, 2006.

[14] D.Cotroneo, S.Orlando, and S.Russo. Characterizing Aging Phenomena of the Java Virtual Machine. In *The 26th International Conference on Reliable Distributed Systems (SRDS 07), Beijing, China*, October 2006.

[15] D.Cotroneo, S.Orlando, and S.Russo. Failure Classification and Analysis of the Java Virtual Machine. In *The 26th International Conference on Distributed Computing Systems (ICDCS 06), Lisboa, Portugal*, July 2006.

[16] S. Orlando and S. Russo. Java Virtual Machine Monitoring for Dependability Benchmarking. In *Proc. of the International Symposium on Object and component-oriented Real-time distributed Computing (ISORC), Gyeongju, S.Korea*, April 2006.

[17] J.E. Hosford. Measures of dependability. *Operations Research*, 8(1):204–206, 1960.

[18] J.C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11, 1985.

[19] J.C. Laprie. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, 1992.

[20] Heimann D.I., Mittal N., and Trivedi K.S. Dependability modeling for computer systems. In *Proceedings of the 1991 Reliability and Maintainability Symposium (RMS91)*, pages 120–128, 1991.

[21] Philip J. Koopman Jr., John Sung, Christopher P. Dingman, Daniel P. Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16st Symposium on Reliable Distributed Systems (SRDS 1997)*, pages 72–79, 1997.

[22] Y.-M. Wang, Y.Huang, K.P Vo, P.-Y. Chung, and C.Kintala. Checkpointing and its applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena,CA (USA)*, June 1995.

[23] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

[24] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. Technical Report 90.1 (Part Number 33579), Tandem Computers Inc., January 1990.

[25] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73–83, 1997.

[26] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification - a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

[27] João Durães and Henrique Madeira. Definition of software fault emulation operators: A field data study. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA*, pages 105–114. IEEE Computer Society, 2003.

[28] J. Gray. Why do computers stop and what can be done about it? In *Proc. of the Symposium on Reliability in Distributed Software and Database Systems (SRDS), Los Angeles, USA*, pages 3–12, 1986.

[29] Michael Grottke and Kishor S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer Communications*, 40(2):107–109, 2007.

[30] D.P. Siewiorek and R.S..Swarz. *Reliable Computer Systems*. A K Peters, third edition, 1998.

[31] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK*, pages 235–248. ACM, 2005.

[32] J.P Hansen and D. P. Siewiorek. Models for Time Coalescence in Event Logs. In *Proceedings of the 22nd IEEE Symposium on Fault Tolerant Computing (FTCS-22)*, June 1992.

[33] W.C. lynch, W. Wagner, and M.S. Schwartz. Reliability Experience with Chi/OS. *IEEE Transactions on Software Engineering*, 1(2):253–257, June 1975.

[34] Ravishankar K. Iyer, Steven E. Butner, and Edward J. McCluskey. A statistical failure/load relationship: Results of a multicomputer study. *IEEE Transactions on Computers*, 31(7):697–706, 1982.

[35] Dong Tang and Ravishankar K. Iyer. Dependability measurement and modeling of a multicomputer system. *IEEE Transactions on Computers*, 42(1):62–75, 1993.

[36] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, October 1990.

[37] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *Proceedings of the 21th International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[38] Cristina Simache and Mohamed Kaâniche. Measurement-based availability of unix systems in a distributed environment. In *12th International Symposium on Software Reliability Engineering (ISSRE 2001), 27-30 November 2001, Hong Kong, China*, pages 346–355. IEEE Computer Society, 2001.

[39] D. Oppenheimer and D.A. Patterson. Studying and Using Failure Data from Large-Scale Internet Services. In *Proc. of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, July 2002.

[40] Joao Duraes and Henrique Madeira. Generic faultloads based on software faults for dependability benchmarking. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 285, Washington, DC, USA, 2004. IEEE Computer Society.

[41] Arup Mukherjee and Daniel P. Siewiorek. Measuring software dependability by robustness benchmarking. *IEEE Transactions on Software Engineering*, 23(6):366–378, 1997.

[42] The DBench Dependability Benchmarking Project (IST-2000-25425). http://www2.laas.fr/dbench/.

[43] Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy*, pages 867–876, 2004.

[44] Karama Kanoun, Yves Crouzet, Ali Kalakech, Ana-Elena Rugina, and Philippe Rumeau. Benchmarking the Dependability of Windows and Linux using PostMark Workloads. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*, pages 11–20, 2005.

[45] Ali Kalakech, Karama Kanoun, Yves Crouzet, and Jean Arlat. Benchmarking the dependability of windows NT4, 2000 and XP. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 681–686, 2004.

[46] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computers*, 51(2):138–163, 2002.

[47] Diamantino Costa, Tiago Rilho, and Henrique Madeira. Joint evaluation of performance and robustness of a cots dbms through fault-injection. In *2000 International Conference on Dependable Systems and Networks (DSN 2000) (formerly FTCS-30 and DCCA-8), 25-28 June 2000, New York, NY, USA*, pages 251–260, 2000.

[48] Eric Marsden, Jean-Charles Fabre, and Jean Arlat. Dependability of CORBA systems: Service characterization by fault injection. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS 2002), 13-16 October 2002, Osaka, Japan*, pages 276–285, 2002.

[49] Marco Vieira and Henrique Madeira. Joint evaluation of recovery and performance of a COTS DBMS in the presence of operator faults. *Performance Evaluation*, 56(1-4):187–212, 2004.

[50] The Ballista Robustness Testing Service. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/edrc-ballista/www/index.html.

[51] Daniel P. Siewiorek, John J. Hudak, Byung-Hoon Suh, and Zary Segall. Development of a benchmark to measure system robustness. In *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 88–97, 1993.

[52] Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS-28), Munich (Germany)*, pages 230–239, 1998.

[53] Charles P. Shelton, Philip Koopman, and Kobey Devale. Robustness testing of the microsoft win32 api. In *2000 International Conference on Dependable Systems and Networks (DSN 2000) (formerly FTCS-30 and DCCA-8), 25-28 June 2000, New York, NY, USA*, pages 261–270, 2000.

[54] Jiantao Pan, Philip Koopman, Daniel P. Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. Robustness Testing and Hardening of CORBA ORB implementations. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN 2001),Göteborg, Sweden, June 30th - July 4th, 2001*, pages 141–150, 2001.

[55] Jorgen Christmansson and Ram Chillarege. Generation of error set that emulates software faults based on field data. In *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 304–313, 1996.

[56] J. Christmansson, M. Hiller, and M. Rimen. An experimental comparison of fault and error injection. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE), 1998.*, pages 369–378, 1998.

[57] S. Garg, Y. Huang, C.M.R. Kintala, and K.S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Philadelphia,PA (USA)*, pages 252–261, 1996.

[58] S. Garg, A. Puliafito, M. Telek, and K.S. Trivedi. Analysis of software rejuvenation using markov regenerative stochastic petri nets. In *6th International Symposium on Software Reliability Engineering (ISSRE 1995)*, pages 24–27, 1995.

[59] András Pfening, Sachin Garg, Antonio Puliafito, Miklós Telek, and Kishor S. Trivedi. Optimal Software Rejuvenation for Tolerating Soft Failures. *Performance Evauation.*, 27-28(4):491–506, 1996.

[60] Yujuan Bao, Xiaobai Sun, and Kishor S. Trivedi. A workload-based analysis of software aging, and rejuvenation. *IEEE Transactions on Reliability*, 54(3):541–548, 2005.

[61] Kalyanaraman Vaidyanathan and Kishor S. Trivedi. A comprehensive model for Software Rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2):124–137, 2005.

[62] P.K. Sen. Estimates of the regression coefficient based on Kendall's tau. *Journal of the American Statistical Association*, (63):1379–1389, 1968.

[63] Y. Hong, D. Chen, L. Li, and K.S. Trivedi. Closed loop design for Software Rejuvenation. In *Proceedings of SHAMAN Workskop "Security for mobile systems beyond 3G", June 2002,Egham, UK*, 2002.

[64] Karen J. Cassidy, Kenny C. Gross, and Amir Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 478–482. IEEE Computer Society, 2002.

[65] Lei Li, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. An approach for estimation of software aging in a web server. In *2002 International Symposium on Empirical Software Engineering (ISESE 2002), 3-4 October 2002, Nara, Japan*, pages 91–102, 2002.

[66] L. Silva, H. Madeira, and J.G. Silva. Software aging and rejuvenation in a soap-based server. In *Proc. of 5th International Symposium on Network Computing and Applications (NCA06), Cambridge,MA (USA)*, pages 56 – 65, 2006.

[67] D.C. Montgomery. *Design and Analysis of Experiments*. John Wiley and Sons Inc., fifth edition, 2001.

[68] Frank Hartman and Scott Maxwell. Driving the Mars Rover. *Linux Journal*, (125):68–70, september 2004.

[69] Java Community Process (JCP). JSR-302: Safety Critical Java Technology, 2006.

[70] A. Georges, D. Buytaert, L. Eeckout, and K. De Bosschere. How Java Programs Interact with Virtual Machines at the Microarchitectural Level. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[71] A. Georges, D. Buytaert, L. Eeckout, and K. De Bosschere. Method-Level Phase Behavior in Java Workloads. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2004.

[72] P.F. Sweeney and M. Hauswirth. Using Hardware Performance Monitors to Understand the Behavior of Java Applications. In *Proceedings of the third Usenix Virtual Machine Research and Technology Symposium (VM '04)*, 2004.

[73] Guilin Chen, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Anand Sivasubramaniam, and Mary Jane Irwin. Analyzing heap error behavior in embedded jvm environments. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2004, Stockholm, Sweden, September 8-10, 2004*, pages 230–235, 2004.

[74] DeQing Chen, Alan Messer, Philippe Bernadat, Guangrui Fu, Zoran Dimitrijevic, David Jeun Fung Lie, Durga Mannaru, Alma Riska, and Dejan S. Milojicic. JVM Susceptibility to Memory Errors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, pages 67–78. USENIX, 2001.

[75] J.Napper, L.Alvisi, and H. Vin. A Fault-Tolerant Java Virtual Machine. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA*, 2003.

[76] Roy Friedman and Alon Kama. Transparent fault-tolerant java virtual machine. In *Proceedings of the 22st Symposium on Reliable Distributed Systems (SRDS 2003), 6-8 October 2003, Florence, Italy*, pages 319–328, 2003.

[77] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[78] T.Lindholm and F.Yellin. *The Java(TM) Virtual Machine Specification*. Sun Microsystems Press, 2nd edition, 1999.

[79] J.Gosling, B.Joy, G.Steele, and G.Bracha. *The Java Language Specification*. Sun Microsystems Press, 3rd edition, 2005.

[80] Ilir Gashi, Peter T. Popov, and Lorenzo Strigini. Fault diversity among off-the-shelf sql database servers. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy*, pages 389–398. IEEE Computer Society, 2004.

[81] K.S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Inc., second edition, 2002.

[82] Java Community Process (JCP). JSR-163: Java Platform Profiling Architecture (JPPA), 2004.

[83] Windows Management Instrumentation (WMI). http://www.microsoft.com/whdc/system/pnppwr/wmi/default.mspx.

[84] SysCallTrack System Call Hijacker. http://syscalltrack.sourceforge.net/.