Performance Evaluation 70 (2013) 231–250

Contents lists available at SciVerse ScienceDirect

Performance Evaluation

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

A comparative experimental study of software rejuvenation overhead

J. Alonso [b,*], R. Matias [a], E. Vicente [a], A. Maria [a], K.S. Trivedi [b]

[a] School of Computer Science, Federal University of Uberlandia, Uberlandia, Brazil

[b] Electrical and Computer Eng., Duke University, Durham, United States

ARTICLE INFO

Article history:
Available online 20 September 2012

Keywords:
Memory leaks
Software aging
System availability
Software rejuvenation
Virtualization

ABSTRACT

In this paper we present a comparative experimental study of the main software rejuvenation techniques developed so far to mitigate the software aging effects. We consider six different rejuvenation techniques with different levels of granularity: (i) physical node reboot, (ii) virtual machine reboot, (iii) OS reboot, (iv) fast OS reboot, (v) standalone application restart, and (vi) application rejuvenation by a hot standby server. We conduct a set of experiments injecting memory leaks at the application level. We evaluate the performance overhead introduced by software rejuvenation in terms of throughput loss, failed requests, slow requests, and memory fragmentation overhead. We also analyze the selected rejuvenation techniques' efficiency in mitigating the aging effects. Due to the growing adoption of virtualization technology, we also analyze the overhead of the rejuvenation techniques in virtualized environments. The results show that the performance overheads introduced by the rejuvenation techniques are related to the granularity level. We also capture different levels of memory fragmentation overhead induced by the virtualization demonstrating some drawbacks of using virtualization in comparison with non-virtualized rejuvenation approaches. Finally, based on these research findings we present comprehensive guidelines to support decision making during the design of rejuvenation scheduling algorithms, as well as in selecting the appropriate rejuvenation mechanism.

1. Introduction

Seventeen years ago, the notion of software aging was formally introduced in [1]. Since then, much theoretical and experimental research has been conducted in order to characterize and understand this important phenomenon. Software aging can be understood as being a continual and growing degradation of the software's internal state and/or its operating environment during its execution. A general characteristic of this phenomenon is the gradual performance degradation and/or an increase in the software failure rate [2]. Aging in a software system, as in human beings, is an accumulative process. The accumulating effects of successive internal error occurrences [3] directly influence the aging-related failure manifestation. Software aging effects are the practical consequence of errors caused by software fault activations. They work by gradually leading the system's erroneous state towards a failure occurrence. This gradual shifting as a consequence of aging effects accumulation is the fundamental nature of the software aging phenomenon [2]. It is important to highlight that a system fails due to the consequences of aging effects accumulated over time. For example, a given aged application fails due to insufficiency of available physical memory caused by the accumulation of memory leaks. In this case, the software fault causing memory leaks is a defect in the program code that prevents the use of previously allocated memory, but no longer in use; thus the memory leak is the observed effect of an aging-related fault being activated. The input patterns that exercise

* Corresponding author.
E-mail addresses: alonso@ac.upc.edu, javier.alonso@duke.edu (J. Alonso), rivalino@fc.ufu.br (R. Matias), elder@mestrado.ufu.br (E. Vicente), anamaria@mestrado.ufu.br (A. Maria), kst@ee.duke.edu (K.S. Trivedi).

0166-5316/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2012.09.002

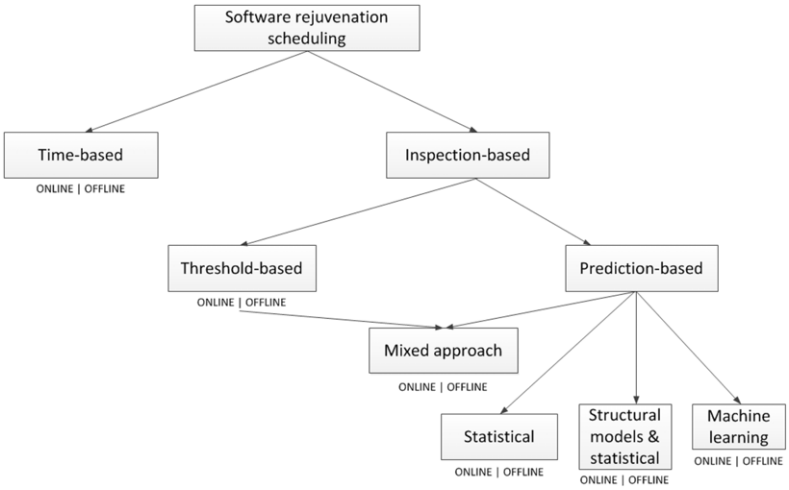

Fig. 1. Software rejuvenation scheduling strategies.

the code fragment where the aging-related fault is located are called the aging factors [2]. Hence, aging-related faults may remain dormant until their activation by the aging factors. The activation time can be represented as a random variable. This unpredictability in the manifestation of aging effects explains why locating and removing aging-related faults is very costly in terms of time and human resources [2,3]. Common causes of software aging are the accumulation of numerical errors, greedy resource allocation policies, non-safe resource releasing strategies, and also degradation problems such as file system and memory fragmentations. Most of these problems are caused by bad software design or faulty code. Regardless of the cause of aging, the presence of aging factors has a deleterious effect on the dependability of software systems.

In order to mitigate the effects of software aging, the concept of software rejuvenation was proposed in [1]. This is a preventive maintenance technique that helps to postpone or prevent the occurrence of failures attributable to the aging effects. Many ideas have been proposed to implement software rejuvenation. For example, stopping and restarting an application process that is suffering from aging (e.g., accumulated memory leaks). This approach aims to prevent or postpone an unexpected application failure that could cause data loss or even major consequences. The main concern, then, is focused on deciding the instant to trigger the rejuvenation mechanism. The software rejuvenation approaches could be classified in two main groups: Time-based and inspection-based strategies. In time-based strategies, rejuvenation is applied regularly and at predetermined time intervals. Time-based strategies are widely used in real environments, such web servers [4].

In contrast to this approach, inspection-based rejuvenation is based on measuring the progress of the aging effects and, when it crosses a certain prespecified limit, triggers the chosen rejuvenation mechanism. In inspection-based strategies, we found three different approaches to determine the optimal moment of triggering the rejuvenation based on the system state: threshold-based, prediction-based, and a mixed approach. In threshold-based approaches, a threshold is pre-fixed by a human expert for every metric under consideration that is an aging indicator [5,6]. In the prediction-based approaches, some prediction method is applied to predict the time to the exhaustion of resources or the time to failure caused by the software aging. Then, the rejuvenation trigger epoch is decided based on the predicted time to exhaustion. In this case, we can find different prediction methods in use: machine learning, statistical approaches, or structural models [7–10]. More recently, we also find some papers combining both these approaches, using prediction methods to determine the optimal threshold to trigger the rejuvenation [11]. Fig. 1 graphically presents this classification of rejuvenation scheduling. There is also an orthogonal classification based on whether the whole process of prediction and scheduling of rejuvenation is carried out off-line [8,9,12] or on-line [5,13–15].

All rejuvenation strategies, in general, have in common the fact that the rejuvenation mechanism usually involves stopping the aged software to refresh its internal state. During this process, it is not uncommon to expect service downtime when the rejuvenation is being carried out. Hence, many papers in this field have concentrated on reducing (e.g., [12,13,16]) or even avoiding (e.g., [5,7,14]) service downtime during a software rejuvenation execution. This is the reason for the importance of properly scheduling software rejuvenation. In order to determine the best time epochs for triggering software rejuvenation, the use of analytic models [17], monitoring system resources followed by statistical analysis [12,18], or their combination [4,15] have been advocated.

As seen from the literature, many different approaches (e.g., [4,7,19,20]) have been proposed to deal with important issues when implementing software rejuvenation. However, to the best of our knowledge, there is no study comparing the effectiveness of these techniques under the same experimental conditions. In all the previous studies, the main goal was to determine the optimal time epoch to trigger specific rejuvenation mechanisms, however, these studies do not take into account the differences in rejuvenation overhead.

To contribute to the body of knowledge in this area, this paper presents a comparative experimental study of different rejuvenation techniques, covering all different levels of rejuvenation granularity investigated so far: application level,

operating system (OS) level, virtualization level, and physical node level. The main purpose of this study is a comparative evaluation of the overhead caused by the rejuvenation strategies according to their granularity. Our experimental evaluation is focused on performance overhead, and memory fragmentation overhead caused by the six rejuvenation strategies under evaluation. We also analyze the effectiveness of the rejuvenation techniques to remove the aging effects. We note that beneficial effects of rejuvenation have been well quantified in previous works; here we are quantifying and comparing the overhead accruing upon triggering rejuvenation. Rejuvenation scheduling methods should attempt to balance its beneficial effect with its overhead. This paper is a major extension of our previously published paper in WoSAR 2011 [21]. The main new contributions of this paper are: (i) analysis of the rejuvenation overhead with respect to memory fragmentation and its consequences; (ii) a more comprehensive analysis of the results, a guideline of the pros and cons of the different rejuvenation strategies in dealing with the aging effects and in minimizing the rejuvenation overhead, and finally (iii) a detailed discussion of different rejuvenation scheduling options available together with guidelines in helping design effective rejuvenation scheduling algorithms.

The rest of this paper is organized as follows. Section 2 revisits the fundamentals of memory-related aging effects, particularly memory leak and memory fragmentation that are investigated in our experimental study. Section 3 provides the basics of the selected rejuvenation strategies. Section 4 describes the methodology used to conduct the experiments, emphasizing the experimental plan and the instrumentation. Section 5 discusses comparative results. Section 6 presents different rejuvenation maintenance policies known in their ability to improve the behavior of systems suffering the effects of aging. Finally, Section 7 presents our conclusions and final remarks.

2. Revisiting memory-related software aging

The most prevalent aging effects investigated in the literature are memory related, specifically memory leaks. Another important memory-related aging effect is the memory fragmentation, but unlike the case of memory leaks, it has not been extensively investigated in the context of software aging. The difficulty involved in experimentally measuring memory fragmentation in a real system is significant, and that is probably one reason for the lack of experimental studies on this topic. Note that both the above mentioned aging effects may occur either at the user-level [22] or at the kernel-level [23].

At the user level, the effects of memory leak and memory fragmentation are located inside the application process, which means they are removed as soon as the aged process terminates. That is why killing the aged application process and restarting it is a widely adopted rejuvenation mechanism. On the other hand, when the memory leak or memory fragmentation occur in the kernel, their impact is higher than that at the user level since the aging effects affect all application processes running under the aged OS. Note that for this case, the general rejuvenation approach is the node reboot, so as to load a new OS kernel instance and then to have the OS kernel's structures re-initialized. In [22,23] specific techniques to measure memory leak and memory fragmentation at the user-level and at the kernel-level, respectively, are presented.

Ref. [22] explains the influence of the user-level memory allocator design on the memory-related aging effects. It describes in detail how the standard Linux user-level memory allocator (i.e., ptmallocv2 [24]) works and how memory leak and memory fragmentation affect it. Understanding these mechanics is very important, particularly because the majority of research papers in this field uses Linux in their studies. In this regard, our experimental study (see Section 4) also uses the Linux OS with the standard user-memory allocator. Our experiments cover aging at the user level, injecting memory leaks inside a Java application and then measuring their effects on the system performance. Most importantly, we evaluate how different rejuvenation techniques, used to mitigate user-level aging effects, affect the rejuvenated system with respect to its performance and availability.

Complementally, [23] explains how to measure memory-related aging effects inside the operating system kernel. Specifically, it describes two approaches to deal with memory leaks and memory fragmentation at the kernel level. Similar to what occurs at the user level, the design of the kernel memory allocator has an influence on the aging effects inside the kernel. The standard Linux kernel currently supports three memory allocators: SLAB [25], SLOB [26], and SLUB [26]. SLUB is the default memory allocator in the current Linux kernel, and thus we use it in our experimental setup. In addition to measuring the effects of memory leaks at the user level, in this paper we apply the kernel instrumentation technique described in [23] to measure (at the kernel level) the intensity of memory fragmentation in a Java application server system under different workload scenarios.

Besides the user and kernel levels, the virtualization technology introduces a new layer where memory-related aging effects can take place. This third layer is the hypervisor. The hypervisor controls the virtualized environment, allowing the virtual machines to share the physical resources (e.g., RAM, CPU). Current implementations of hypervisor are based on specialized microkernels or modified existing operating system kernels to support virtualization. In both cases this new layer is also vulnerable to aging effects such as memory leaks and memory fragmentation. Furthermore, in a virtualized environment there are multiple virtual machines running under the same hypervisor, so the effects of aging at the hypervisor level may affect the multiple VMs and consequently their operating systems and applications. Fig. 2 presents a generic view of all three levels mentioned above.

Many previous papers (e.g., [5,6,20,27,28]) have proposed the use of virtualization to reduce or even avoid the performance degradation during the rejuvenation execution. However, none of them have investigated the drawbacks of this decision especially in terms of the effects of aging inside the hypervisor level. Hence, in this paper in addition to monitoring the level of memory fragmentation inside the OS kernel we also monitor aging at the hypervisor level.

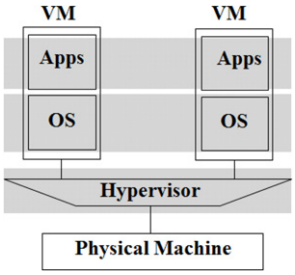

Fig. 2. Three levels susceptible to memory-related aging.

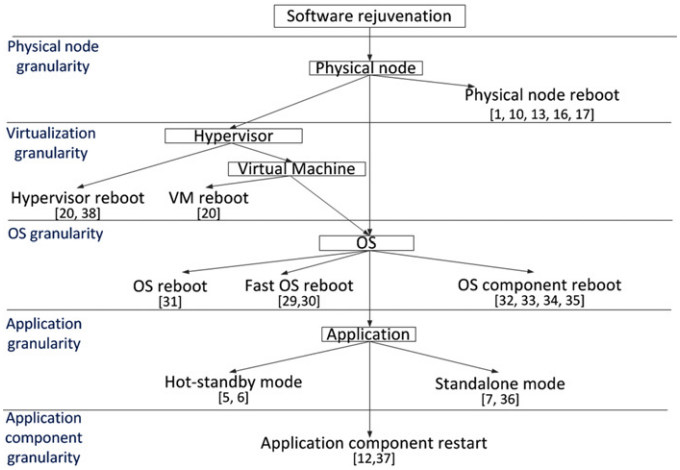

Fig. 3. Rejuvenation granularities.

3. Rejuvenation granularities

Our study is focused on the experimental comparison of different rejuvenation techniques under the same operational conditions, measuring their overhead on the performance, as well as their effectiveness in removing the aging effects. We have classified the rejuvenation techniques based on their granularity. We define granularity as the level that the rejuvenation mechanism directly targets. Based on the system architecture, we can define five main rejuvenation granularity levels while referencing the relevant literature: (i) node level [1,10,13,16,17], (ii) operating system (OS) level [29–31], (iii) OS component level [32–35], (iv) application level [5–7,36] and (v) application component level [12,37]. Finally, due to the growing adoption of virtualization, a new layer between node level and OS level is added: (vi) virtualization level [20,38]. It is important to note that a rejuvenation at the OS level usually implies the rejuvenation of all the subsumed layer (i.e., application) as well. There are several ongoing initiatives [32–35] to develop a more reliable OS, where it is possible to restart an OS component without needing to restart its subsumed layers. However, these solutions are not yet in general use. Fig. 3 depicts the proposed rejuvenation granularity classification and references to research papers for each of them.

In this paper, we restrict ourselves to the first four granularity levels. In order to cover these four levels, we evaluate six different rejuvenation strategies: standalone application restart, application rejuvenation by hot-standby server, OS reboot, fast OS reboot, VM reboot, and physical node reboot. The rejuvenation strategies at the application-component level are outside the scope of this paper since they are usually designed in an ad hoc manner based on the architecture and the characteristics of the application, and hence are not easy to generalize.

3.1. Application-level granularity

3.1.1. Application standalone restart

The first rejuvenation technique under evaluation is a simple automatic application restart. This rejuvenation technique basically stops and restarts the application process. This technique has been widely adopted (e.g., [1,12,13,16]) to mitigate the aging effects in many general and special purpose systems. The main overhead cost of using it is the service downtime, which we evaluate in comparison with other rejuvenation approaches tested.

3.1.2. Application rejuvenation by hot standby server

As an alternative to the simple restart of application processes, more sophisticated techniques have been proposed to offer rejuvenation at the application level. These techniques focus on gracefully restarting the application, so as to reduce the downtime perceived by end-users as much as possible. Some techniques [37] apply the rejuvenation at application component level. However, these approaches require re-engineering the application, hence are not as cost-effective a solution. In this study we have selected the application rejuvenation technique discussed in [5,14]. This approach uses virtualization technology so that it can be used for legacy as well as new applications without requiring application re-engineering. The most significant contribution of this approach is that it reduces failed requests while carrying out rejuvenation using a hot-standby application server. A load balancer is used as a proxy of the target application server and a hot-standby application server is ready to take over from the one being rejuvenated. When the rejuvenation is triggered, the new requests to the target application are redirected to the hot-standby application server. When all processing of the ongoing requests has finished on the primary application server, the server is restarted. So, during a short time duration both applications are running simultaneously, guaranteeing the service availability; assuming that other essential components do not fail in the meantime.

However, note that this approach introduces a new level of complexity by adding a new software replica, additional software components (e.g., load balancer), and added technology (virtualization). Hence, to correctly evaluate the benefits of this approach, we also have to consider the performance overhead of this increased complexity, besides the reduction of failed requests.

3.2. OS granularity

The previous two approaches are based on applying the rejuvenation at the application level. However, they are only effective when the aging effects are confined in the application processes. If the aging effects are accumulating inside the operating system kernel [22,39], then the previous approaches become ineffective. For this reason, it is also important to apply rejuvenation at the OS granularity. We consider two OS rejuvenation techniques in our experimental study.

3.2.1. OS reboot

In this approach, the OS is rebooted (full reboot) using the standard reboot mechanism of the operating system. Generally, it involves reloading the firmware (i.e., BIOS), re-executing the POST (power-on self test) routines, reloading and restarting the OS kernel code, restarting the basic OS services, and finally restarting the application.

3.2.2. Fast OS reboot

Since the sequence of stages executed during an OS reboot introduces considerable delay before the system is able to run the application processes again, we also test the recent advance known as the fast OS reboot. To the best of our knowledge, two main approaches have been proposed in order to speed up the OS reboot process [29,30]. In [30], *phase-based* reboot is presented. This approach divides the booting process into phases and skips those phases that can be skipped based on the state of the system. In this paper, we use the *kexec* system call [29] to reload a new kernel and restart it without re-loading all firmware and without re-executing POST related steps. This approach replaces the current kernel, at runtime, keeping the same internal state by refreshing as the usual full reboot, but without the delay caused by the hardware initialization related routines. We have chosen this technique due to the fact that it is well known and available in many Linux distributions.

Regardless of the OS rejuvenation approach chosen, the chain of tasks required to be implemented is: (i) stop the target aged application processes, (ii) reload the operating system (full reboot or fast reboot), and (iii) restart the new target application process.

3.2.3. Virtual machine reboot

As mentioned in Section 3.1, one of the selected application rejuvenation techniques is based on virtualization. The virtualization technology is increasingly being adopted to reduce the IT infrastructure costs [40]. This means that virtualized computing platforms have become a popular option to deploy long-running applications in data centers. In these environments, usually every application is deployed on its own virtual machine, and thus called a virtual appliance. Hence, our experimental study also considers the effectiveness of restarting a virtual appliance (destroy and recreate the target virtual machine) to mitigate potential aging in the virtual machine, where an ordinary OS reboot inside the virtual appliance cannot remove it.

3.3. Physical node reboot

Finally, we test the most common rejuvenation approach — the physical machine reboot. This approach is based on stopping the machine (power off) and starting it again (power on). In this case, we physically restart the machine (cold reboot) instead of using a reboot instruction (hot reboot) from the OS (see Section 3.2.1). This last strategy is studied as a baseline to compare with other less drastic approaches. Furthermore, we wish to determine if the use of virtualization introduces some improvements. The virtual machine restart could be seen like the node restart. However, in this case the

Table 1
Test bed configuration.

System	OS	Memory (MB)	Software	Hardware
M1	CentOS 2.6.18-238.el5	2048	JDK 1.6.0_17; TPC-W	Intel Quad Core Q8400; 2.66 GHz
M2	OpenSuSe 2.6.37.6-0.5-xen	512	JDK 1.6.0_25; Tomcat 5.5.26	Intel Quad Core Q6600; 2.40 GHz
M2	OpenSuSe 2.6.37.6-0.5-default	4096	JDK 1.6.0_25; Tomcat 5.5.26	Intel Quad Core Q6600; 2.40 GHz
M3	CentOS 2.6.18-238.el5	2048	Mysql Server 5.0.77	Intel Pentium Dual E2180; 2.00 GHz
VM1	CentOS 2.6.18-238.el5xen	256	Ipvsadm 1.24	Intel Quad Core Q6600; 2.40 GHz
VM2	CentOS 2.6.18-238.el5xen	1152	JDK 1.6.0_25; Tomcat 5.5.26	Intel Quad Core Q6600; 2.40 GHz
VM3	CentOS 2.6.18-238.el5xen	1152	JDK 1.6.0_25; Tomcat 5.5.26	Intel Quad Core Q6600; 2.40 GHz

node is a software object, no hardware component is involved. So, we wish to study if the use of virtualization reduces the downtime impact of the node reboot.

4. Experimental setup

In this section we present the instrumentation and experimental plan adopted in our study.

4.1. Instrumentation

The test environment reproduces a typical web application composed of a web application server, a database server, and a set of web clients. We have focused on a web application because it is a well-known example of a long running application where the aging phenomenon is especially evident [7,11,12]. Note that the rate of aging is highly application and workload dependent, but it can be argued that the rejuvenation overhead is not dependent as much on the application and the workload. We have used a multi-tier e-commerce site that simulates an on-line bookstore, following the standard configuration of TPC-W benchmark [41]. We use Java *servlets* and MySQL as database server. As the application server we use the Apache Tomcat [42]. TPC-W allows us to run different experiments using different parameters and under a controlled workload. TPC-W clients, called Emulated Browsers (EBs), access the emulated book store web site, in sessions. Between two consecutive requests from the same EB, TPC-W computes a think time, representing the time between the user receiving the requested web page and submitting the next request. The think time is a random variable, generated following a truncated exponential distribution with an average of 7.5 s and maximum of 75 s [43]. After the think time has elapsed, the TPC-W Emulated browser selects the next TPC-W navigation option (next web page) randomly, following a uniform distribution between the legal navigation options based on the last web page visited. In all of our experiments we have used the default configuration of TPC-W. Moreover, following the TPC-W specification, the number of concurrent EBs is kept constant during the experiment (in our case 700EBs running simultaneously). In order to simulate the aging phenomenon, we inject memory leaks at the application level. To emulate memory depletion caused by aging-related bugs, we have modified the TPC-W implementation through the injection of memory leaks. So, in our experiments, the memory leak phenomenon is embedded at the application layer to avoid potential interference from other applications as would occur if we were injecting memory leaks at the OS level directly. However, note that memory injection at the application level may cause the entire system to crash due to system resource exhaustion. To simulate a random memory leaking rate, we modify a servlet (*TPCW_search_request_servlet*) which generates a random number between 0 and n. This random variable follows a uniform distribution between 0 and n (in our experiments $n = 30$). This random value determines how many requests are allowed to use the servlet before the next memory leak is injected. Therefore, the variation of memory consumption depends on the number of clients and the frequency of servlet visits. According to the TPC-W specification, this frequency depends on the workload chosen. Thus, our servlet injects memory leaks more quickly under high workload than under low workload. This rationale has been proposed in other related papers [4]. In order to have a fair comparison between experiments, we have selected the same shopping workload characterization for all experiments. For more details about this workload see [44].

The test bed setup used in our experiments is described in Table 1. Note that M2 has two different kernel configurations for experiments with or without virtualization. We used the Xen virtualization middleware.

4.2. Experimental plan

We conducted a total of eight experiments, since two out of the six granularity strategies described in Section 3 (standalone mode application restart and OS reboot) are experimented with and without virtualization. Each experiment is replicated five times, in the same environmental and workload conditions. Replication is necessary to reduce the experimental error. Each replication was running for 30 min, where the first 10 min are dedicated to a warm-up period. For all the experiments, a rejuvenation is triggered approximately 10 min after the warm-up period, and the application continues to run for 10 more minutes. These time epochs were chosen in order to simplify the comparison of strategies. The goal of the paper is comparing different rejuvenation techniques under the same scenario. So, it is necessary to build a very controlled experiment. We also note that we compute the rejuvenation overhead over five repetitions. In order

Table 2
Measured performance metrics.

Client-side metrics	Server-side metrics
Throughput (req/s)	App. memory depletion (MB)
Response time (ms)	Memory fragmentation (# Events)
# Requests (total)	
# Failed requests	
# Slow requests	

to simplify the computation of average and plotting the results, we decided to conduct the rejuvenation at a previously determined time instant. We choose the instant to be after 10 min (excluding the warm-up period) for simplicity. We note that the rejuvenation triggering instant is not relevant in our study due to the fact that we only conduct one rejuvenation per experiment replication. So, the results, in terms of overhead, would be the same, independent of the instant of time we trigger the rejuvenation.

We monitor the system during the 30 min of test. This monitoring occurs on both sides (i.e., the client and server).

On the client side, we monitor the throughput (requests/second), mean response time (milliseconds), total number of requests, number of failed requests, and the number of slow requests. The definition of slow requests is based on the well-known "eight-seconds rule", which defines slow requests as those with response time greater than 8 s [45]. At the sampling interval of 15 s, we calculate the average throughput and the mean response time. We also calculate how many requests are sent to the server and the number of requests that failed during the same period, which gives us the total number of failed requests during the rejuvenation time. We have ensured that the workload selected never caused failed requests due to any other reasons such as the server overload. In other words, the request is assumed failed in only two situations: The request timeout expires before any answer arrives from the server or the response from the server indicates some HTTP error indicated by 4XX and 5XX codes.

On the server side, we measure the aging effects in terms of memory depletion and memory fragmentation, using the approaches described in [23,22], respectively. Our goal is to analyze the effectiveness of the investigated rejuvenation approaches in terms of their ability to mitigate these two memory-related aging effects. The monitoring sampling interval on the server side is 30 s. Note that memory depletion is measured from the outside of the JVM. The memory used by a Java application may look quite different if monitored at the operating system level than from inside the JVM. When a Java application frees up memory objects, it is not possible to observe the changing trend in the memory used from the OS. However, if we observe it from inside the JVM, then we can obtain accurate measurements. This is because the JVM starts by reserving a maximum amount of memory to be used for dynamic memory allocation. Subsequently, the JVM manages dynamic memory without involving the OS. Important to note here is that this behavior deviation (between the OS and JVM perspectives) is only relevant in determining when to trigger rejuvenation. As we have focused our study on evaluating the overhead and the effectiveness of rejuvenation strategies to mitigate the aging effects, we only need to observe if the injected memory leak effects are removed after conducting the rejuvenation. Moreover, since we run our experiments for 30 min, this time is not enough to inject memory leaks to trigger aggressive JVM garbage collector activities, which can cause the behavior deviation between the OS perspective and JVM internal perspective [9] to be significant. So without limited duration experiments, the injected memory leaks are observable from the OS perspective thus reducing the monitoring infrastructure needed, and their associated overhead as well. Table 2 shows the performance metrics used in this study.

The eight designed experiments cover all rejuvenation granularities discussed in Section 3. At the application level, we have conducted three experiments: (1) applying standalone application restart in a virtualized environment, (2) applying a hot standby rejuvenation solution in a virtualized environment, and (3) applying standalone application restart, where the OS runs directly on the physical machine. For the all three experiments, the aged application process (Tomcat) is stopped and restarted by automatically killing and re-starting its JVM.

At the OS level, we have conducted another set of three experiments: (4) applying OS reboot in a physical machine, (5) applying fast OS reboot in a physical machine environment, and (6) applying OS reboot inside a virtual machine OS guest. Note that fast OS reboot approach is not tested inside a virtual machine because technically there is no difference when executing it on physical and virtual machines. For the OS rejuvenation experiments, first we stop the Tomcat and next reboot the OS, and finally restart Tomcat. This chain of steps is executed automatically without human intervention.

At the virtualization level, we have conducted one experiment: (7) applying a VM reboot inside the hypervisor. It is done by stopping Tomcat, shutting down the OS and then destroying the VM hosting the Tomcat application. Subsequently, we create a new VM instance, boot its OS and finally restart the Tomcat application. Note that all of these steps are carried out automatically without human intervention.

Finally, at the physical node level, experiment (8) applies a physical node reboot. In this case, we manually execute a power switch off, which forces the OS to shut down sending a signal to all the application processes to terminate. After that, the system turns off and then manually we turn it back on. All these manual steps follow the same protocol to make sure that consistency among the replications is maintained. Table 3 summarizes the experimental plan. Note that the experiment names are based on a mnemonic convention to help the reader follow the rest of the paper. The experiment name is composed by: granularity (APP-application, OS-operating system, VM-virtualization, and PH-physical node) + technique

Table 3
Summary of the test plan.

Exp.	Exp. name	Granularity	Environment	Rejuvenation technique
1	APPStandVM	App. process	Virtual machine	Standalone
2	APPHotVM	App. process	Virtual machine	Hot standby
3	APPStandPH	App. process	Physical node	Standalone
4	OSRebootPH	OS	Physical node	OS reboot
5	OSFastPH	OS	Physical node	fast OS reboot
6	OSRebootVM	OS	Virtual machine	OS reboot
7	VMRebootVM	Virtual machine	Virtual machine	VM reboot
8	PHRebootPH	Physical node	Physical node	Node cold reboot

Table 4
Application granularity — results on the client side.

Exp.	Exp. name	#Total requests	#Failed requests	#Slow requests
1	APPStandVM	118 203	174	0
2	APPHotVM	117 034	0	2
3	APPStandPH	118 140	193	0

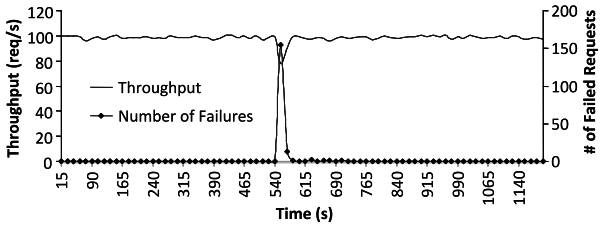

Fig. 4. Throughput and number of failed requests in Exp. *APPStandVM*.

used (Stand-standalone, Hot-hot stand by, Fast-fast OS reboot, and Reboot) + environment (VM-virtualization, PH-non virtualization).

5. Analysis of experimental results

In this section we present the results obtained from our experiments. We compare them mainly in terms of the execution overhead that each evaluated rejuvenation technique causes on the target application, especially on the client side. We also investigate the effectiveness of the rejuvenation execution on the server side in terms of memory consumption and fragmentation. Note that the values presented in this section are averages over five replications for each experiment.

5.1. Results of application granularity level

The client-side results for the application level rejuvenation are summarized in Table 4.

We observe that the hot standby approach (*APPHotVM*) is very effective in avoiding failed requests. It also shows good results in terms of throughput (the total number of requests processed). It processes 1169 requests (about 1%) less than the standalone approach in the same virtualized environment (*APPStandVM*). These differences are caused possibly by the impact of the extra routines and pieces of software added (e.g., load balancing) to guarantee no failed requests due to rejuvenation. Note that the hot standby approach causes, on the average, two slow requests. The migration process from aged application to the standby one causes the slow requests. The hot standby application has to react to a sudden workload and this fact causes some slow requests. In the case of scenarios *APPStandVM* and *APPStandPH*, the rejuvenation only causes failed requests but no slow requests.

Comparing the application restart executed inside (*APPStandVM*) and outside (*APPStandPH*) the VM, the virtualized environment shows a lower number of failures (approximately 18 failures). The difference only represents about 0.14% of the total number of requests, that is not substantial. To show the client perception during rejuvenation, Fig. 4 presents the throughput and the number of failed requests for experiment *APPStandVM*. Note that all the figures related to client side performance (throughput and response time) only present the last 20 min of experiments. For the sake of simplicity, we have removed the warm-up period. The overhead effect of an application restart is clearly observed. We see a throughput drop and, at the same time, we observe failed requests. The same pattern is observed in experiment *APPStandPH*.

Fig. 5 shows the client side results for experiment *APPHotVM*. We do not observe any overhead in terms of throughput loss. The throughput is slightly affected by the extra software used to guarantee the application availability. This overhead

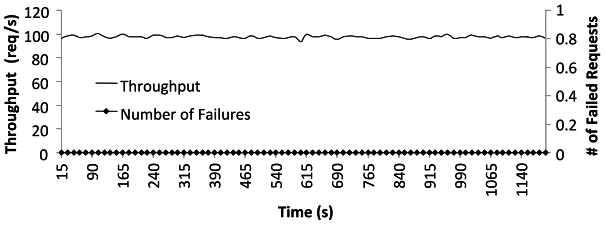

Fig. 5. Throughput and number of failed requests in Exp. *APPHotVM*.

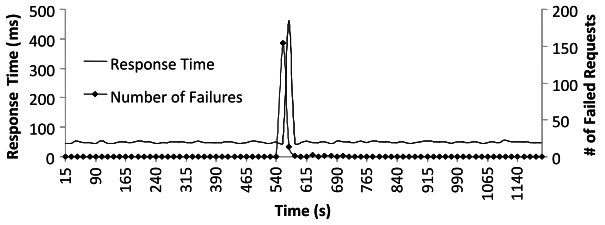

Fig. 6. Response time and failed requests in Exp. *APPStandVM*.

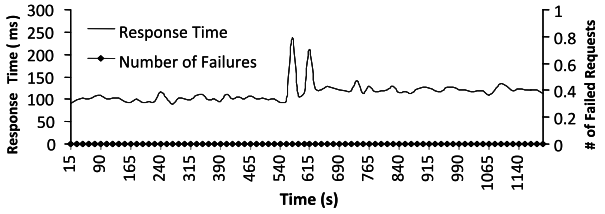

Fig. 7. Response time and failed requests in Exp. *APPHotVM*.

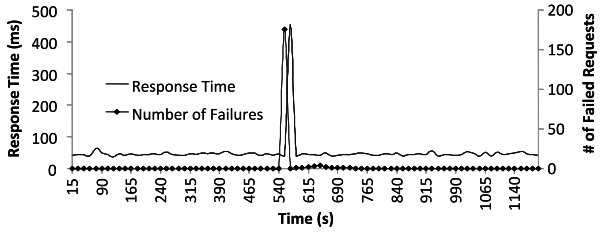

Fig. 8. Response time and failed requests in Exp. *APPStandPH*.

impact is only visible in the number of requests processed during the execution time (see Table 4), but the gain in terms of non-failed requests is clear.

Fig. 6 presents the response time for experiment *APPStandVM*. We observe a peak just after the application restart, and then the response time returns to the normal level.

By contrast, experiment *APPHotVM* presents a more unstable response time behavior, mainly around the time of fail-over to the standby (see Fig. 7). Furthermore, the response time is higher than those values observed in experiments *APPStandVM* and *APPStandPH*, indicating the performance overhead caused by the extra fault tolerance mechanisms. We also clearly observe, on the client side, two peaks of response time. The migration of requests caused these two peaks. The hot standby application reacts to a sudden workload peak, causing an overhead during the application startup period. This shows the performance overhead introduced by this rejuvenation technique. However, we note that this evident performance impact was hardly observable in the throughput measures (see Fig. 5).

Fig. 8 presents the response time for experiment *APPStandPH*. We observe a pattern very similar to the experiment *APPStandVM*. This is reasonable since the unique difference between both experiments is the incorporation of the virtualization layer.

On the server-side we measure memory depletion by monitoring the resident set size of the aged application process as in [7,23]. We observe that for all the three application-level experiments (*APPStandVM*, *APPHotVM*, *APPStandPH*) the aging process behaves as expected, since the aging-related fault injection is applied in the same way for all the experiments. In terms of rejuvenation effectiveness, we observe that all three scenarios have practically the same results (see Figs. 9–12). This means that there is no difference between executing application restart in a virtual machine or in a physical machine.

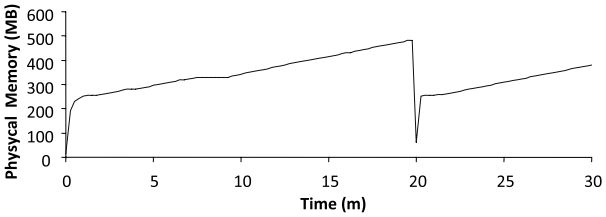

Fig. 9. JVM resident set size and rejuvenation in Exp. *APPStandVM*.

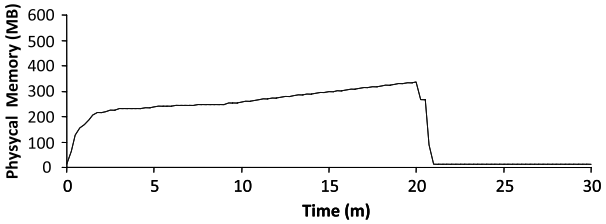

Fig. 10. JVM resident set size in and rejuvenation in Exp. *APPHotVM* (VM1).

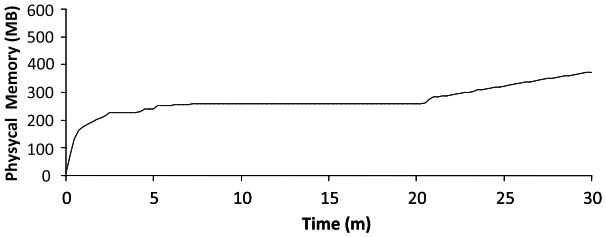

Fig. 11. JVM resident set size in and rejuvenation in Exp. *APPHotVM* (VM2).

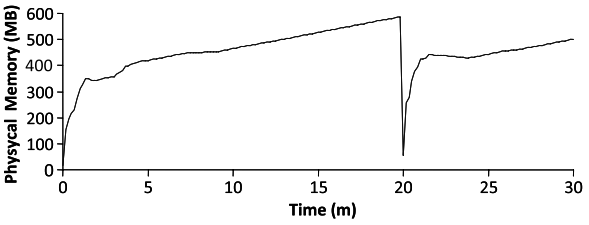

Fig. 12. JVM resident set size in and rejuvenation in Exp. *APPStandPH*.

We note that memory depletion (see Figs. 9 and 12) does not show the entire application memory release during the rejuvenation. When we stop the application, the monitoring probes are also stopped and they are restarted only after the application is restarted since they need to know the process identifier (*pid*) to monitor it. So, when the monitoring probes start, the application is already created and some initial memory is used.

In the case of experiment *APPHotVM*, the memory usage pattern is quite different as presented in Figs. 10 and 11. Fig. 10 presents the memory depletion of the primary application process running in VM1. We observe clearly the rejuvenation instant (20th min). After that, all new requests are redirected to the hot standby application running on VM2, thus increasing its memory usage as shown in Fig. 11. We also observe that memory usage in VM1 goes down (see Fig. 10). We note that Fig. 10 represents the average of the five repetitions of the same experiment. In this case, we observe two different instances of replacing VM1 for VM2 (20th and 21st min, approximately). This is caused by the time window where both applications are running simultaneously along with the monitoring probes, before the application (in VM1) is finally restarted. We highlight that the time window depends mainly on the time needed by VM1 to finish processing all the ongoing requests, which takes a random amount of time. Hot standby application (in VM2) is not rejuvenated in this experiment.

We observe that the scenarios using virtualization, experiments *APPStandVM* and *APPHotVM*, introduce a high overhead in terms of memory fragmentation when compared with the scenario without virtualization (experiment *APPStandPH*). Table 5 presents the number of memory fragmentation events captured on the server-side for the application level experiments. Analyzing the collected data, we observe that system processes responsible for the virtualization and load-balancing

Table 5
Application granularity — results on the server side.

Exp.	Exp. name	# Memory fragmentation events
1	APPStandVM	135 230
2	APPHotVM	188 777
3	APPStandPH	6

Table 6
OS granularity — results on the client side.

Exp.	Exp. name	#Total requests	#Failed requests	#Slow requests
4	OSRebootPH	111 717	5509	913
5	OSFastPH	113 094	3585	926
6	OSRebootVM	111 473	5758	527

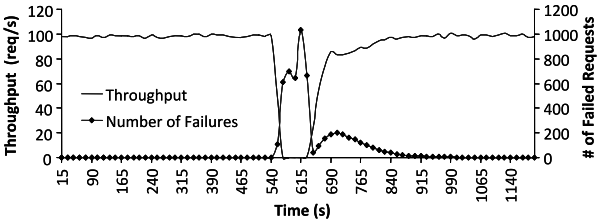

Fig. 13. Throughput and failed requests in Exp. *OSRebootPH*.

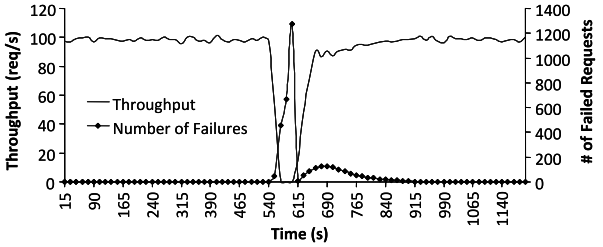

Fig. 14. Throughput and failed requests in Exp. *OSFastPH*.

infrastructures mainly caused these fragmentation events. In comparison with experiment *APPStandPH*, the fragmentation overhead caused by these processes is enormous.

5.2. OS granularity level results

The client side results obtained from the three experiments conducted at OS granularity level are presented in Table 6.

Experiments *OSRebootPH* and *OSRebootVM* are based on OS reboot outside and inside of a virtual machine, respectively. Experiment *OSFastPH* is based on fast OS reboot outside of the virtual machine. The results show the effectiveness of fast OS reboot strategy; it incurs only 3585 failed requests, a reduction of 34.9% compared with the (full) OS reboot in the same environment (*OSRebootPH*).

In the experiment labeled *OSRebootVM*, the application completes only 0.2% fewer requests than in the experiment *OSRebootPH* (the same solution, different environment). But, by contrast, it suffers from 4.3% more failed requests. However, if we compare application reboot (Table 4) and OS reboot, the OS reboot incurs slow requests while previously discussed methods (i.e., application restart approaches *APPStandVM* and *APPStandVM*), did not. This is caused by the difference between TCP/IP time-out mechanisms when the node is available (application restart) and when it is not (OS reboot).

Finally, the throughput and response time results for all three experiments follow the same pattern and they are presented in Figs. 13–18. The main difference is based on the downtime period and response time behavior. Figs. 13–15, respectively, present the evolution (in terms of throughput) of experiments *OSRebootPH*, *OSFastPH*, and *OSRebootVM*. We clearly observe the downtime period and the number of failed requests. We also observe how the startup period after the rejuvenation impacts the throughput, and thus the application needs a period of time to reach its steady state.

In Figs. 16–18, we observe two peaks in the response time. The first peak is caused by the failed requests during the time that the OS is stopped and the second peak is caused by the workload peak when the new application is started after rebooting the operating system.

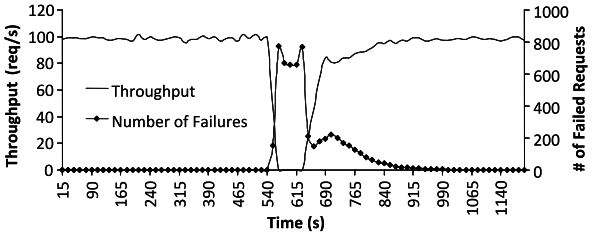

Fig. 15. Throughput and failed requests in Exp. *OSRebootVM*.

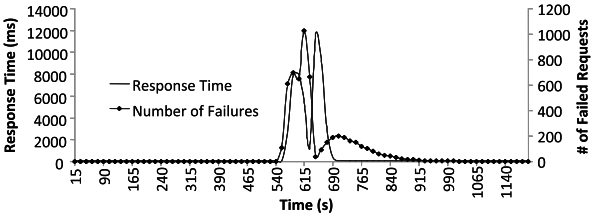

Fig. 16. Response time and failed requests in Exp. *OSRebootPH*.

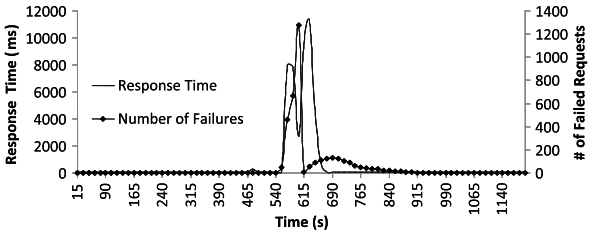

Fig. 17. Response time and failed requests in Exp. *OSFastPH*.

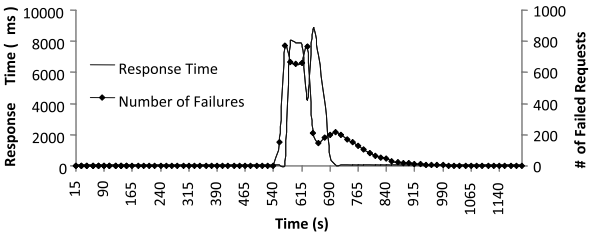

Fig. 18. Response time and failed requests in Exp. *OSRebootVM*.

Figs. 13–18 also show an interesting second peak of failed requests, a part of the first peak that corresponds with the downtime period. This second peak is caused by the startup period. The application is not able, immediately after restarting, to process all the arriving requests, thus causing connection timeouts for a short period of time.

However, this is not observed in the application restart strategies. So, we believe that this occurs due to the overhead caused by the OS rejuvenation. It is related to the OS initialization period, when the OS is starting its basic services. Note that experiments *OSRebootPH* and *OSFastPH* have a similar number of slow requests (difference of 13 requests between the two techniques). But, the OS reboot inside virtual machine only has 527 slow requests. This difference is because the downtime period, in this case, is longer than the other experiments and some of the slow requests in experiments *OSRebootPH* and *OSFastPH* are counted as failed requests. The slow requests are caused after rejuvenation is completed. When the application is restarted, the application starts a warm-up phase. This period can cause slow requests.

On the server side, the effect of these three mechanisms on the memory depletion is practically the same, given that all of them refresh the system memory through rebooting the OS (see Figs. 19–21).

In terms of memory fragmentation, we observe that experiments *OSRebootPH* and *OSFastPH* (without virtualization) show lower numbers of memory fragmentation events when compared to the experiment *OSRebootVM* (with virtualization); it corroborates our prior observation that the added virtualization infrastructure introduces important overhead, which results in high levels of memory fragmentation. In this case, the OS reboot has an important role in removing this type of aging effect, since it is confined to the OS kernel and thus it is not removed by the application restart. Table 7 shows the numerical results.

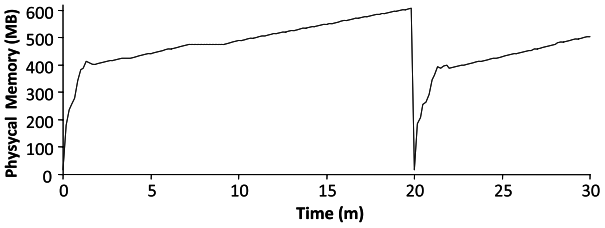

Fig. 19. JVM resident set size and rejuvenation in Exp. *OSRebootPH*.

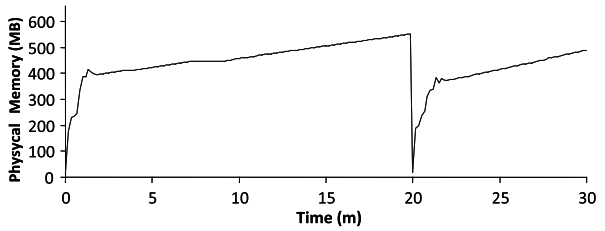

Fig. 20. JVM resident set size and rejuvenation in Exp. *OSFastPH*.

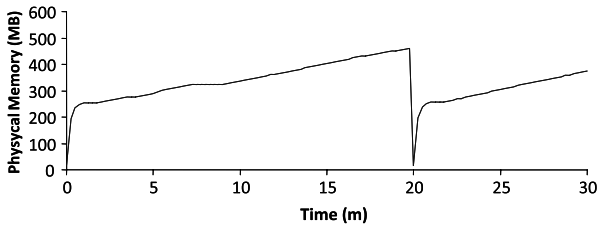

Fig. 21. JVM resident set size and rejuvenation in Exp. *OSRebootVM*.

Table 7
OS granularity — results on the server side.

Exp.	Exp. name	# Memory fragmentation events
4	OSRebootPH	5.0
5	OSFastPH	4.6
6	OSRebootVM	9270

Table 8
Virtualization and physical node granularity — results on the client side.

Exp.	Exp. name	#Total requests	#Failed requests	#Slow requests
7	VMRebootVM	111 550	6091	544
8	PHRebootPH	111 592	5625	994

Observe that in experiment *OSRebootVM* (with virtualization) the OS reboot brings about a considerable reduction in the system fragmentation level when compared with experiments *APPStandVM* and *APPHotVM* from application granularity (see Table 5), which also use virtualization. This reduction is on the average 93.4% and 95%, respectively. Given that the rejuvenation granularity used in experiments *APPStandVM* and *APPHotVM* is at the application-level, the fragmentation events are not affected and still continue to accumulate over the duration of the experiments.

5.3. VM granularity level and physical node level results

Finally, we analyze the results obtained on the client side while applying VM reboot and physical node reboot. Table 8 summarizes the results.

In some sense, experiments *VMRebootVM* and *PHRebootPH* are based on the same idea: stop the system completely and start it again. In *VMRebootVM*, this is done via the virtual machine destruction and recreation of standard routines. In *PHRebootPH*, this is conducted via the power button (switch off/on). The total number of processed requests in both scenarios is quite similar (only 42 more requests in *PHRebootPH*). On the other hand, experiment *PHRebootPH* is able to reduce the number of failed requests by 8%. This is consistent with results presented in Table 6. The virtualization scenarios have more

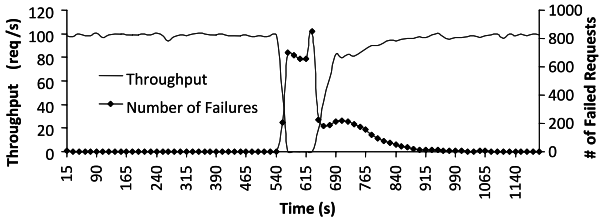

Fig. 22. Throughput and failed requests in Exp. *VMRebootVM*.

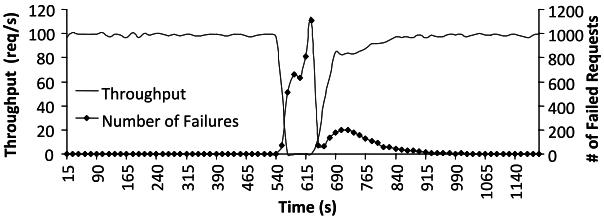

Fig. 23. Throughput and failed requests in Exp. *PHRebootPH*.

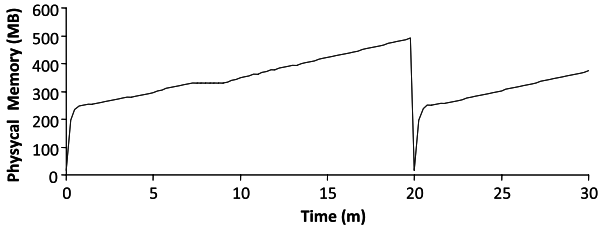

Fig. 24. JVM resident set size and rejuvenation in Exp. *VMRebootVM*.

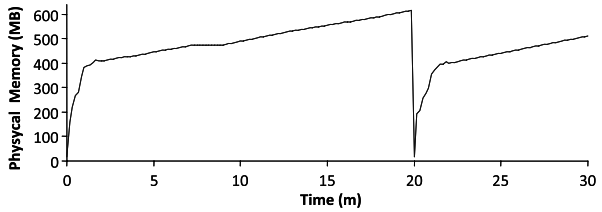

Fig. 25. JVM resident set size and rejuvenation in Exp. *PHRebootPH*.

failed requests than no virtualization. Following the same reasoning, the number of slow requests using virtual machine reboot is smaller than physical node reboot. This result follows the same pattern as in the OS rejuvenation analysis. A longer downtime increases the number of failed requests and reduces the number of slow requests.

Figs. 22 and 23 show the throughput of both experiments (*VMRebootVM* and *PHRebootPH*). In this case, we observe the clear downtime period and a second failed requests peak after rebooting the system. This tells us that this set of failed requests is caused by the OS initialization overhead and not by the application startup overhead. If the application startup overhead were the root-cause, then we would have observed similar patterns in experiments *APPStandVM* and *APPStandPH* (see Figs. 4 and 8), where the application also restarts.

The server-side analysis for experiments *VMRebootVM* and *PHRebootPH* just confirms that the effect of the virtual machine reboot (VM destroy & creation) and physical machine cold reboot (power switch off/on), on the memory depletion (see Figs. 24 and 25), is practically the same as was observed in previous experiments (application and OS granularity level's rejuvenation).

In terms of memory fragmentation, these last two experiments also corroborate the previous results, where experiment *VMRebootVM* (VM based) shows a high value of accumulated memory fragmentation events that is very close to that observed in experiment *APPStandVM*. Experiment *PHRebootPH* shows the same effectiveness as the case using OS reboot (full or fast). Table 9 summarizes the results of experiments *VMRebootVM* and *PHRebootPH*.

Table 9
Virtualization and physical node granularity — results on the server side.

Exp.	Exp. name	# Memory fragmentation events
7	VMRebootVM	13 412
8	PHRebootPH	6.3

5.4. Comprehensive experimental comparison

Based on the results of this experimental study we summarize our analysis as follows. All evaluated rejuvenation strategies are similarly efficient in terms of removing the accumulated (injected) memory leak effects. From the client-side viewpoint, implementing the application process restart with a hot-standby application server is the best option evaluated in terms of the number of failed requests. However, we conclude that this redundancy might be preferentially implemented through different physical servers. We verified that using virtualization to implement any rejuvenation technique introduces an important overhead that causes harmful effects on the server side in terms of memory fragmentation. If virtualization has to be used, then it is necessary to mitigate the aging effects related to memory fragmentation in some way. This becomes critical because for some specific memory allocation requests, mainly to support kernel-level services, physically contiguous memory pages are required. In this case memory fragmentation is a major concern in both the hypervisor and the OS Guest. If the memory is significantly fragmented, then large allocation requests — for contiguous memory — will start failing at some point. This could cause performance degradation mainly on the hypervisor in a long-running system, and subsequently affecting virtual machines running on it. Note that mitigating the aging effects inside the hypervisor is not trivial, requiring very sophisticated rejuvenation mechanisms. Also, rejuvenation at this level impacts all subsumed layers, such as the virtual machines running on it, and their guest operating systems and applications (see Fig. 2). This finding is very relevant since the broad adoption of virtualization technology gave rise to many proposals using this technology to support rejuvenation operations [5,6,14,20,27,28]. However, so far no quantitative evaluation has been carried out to evaluate their drawbacks. Our results demonstrate that virtualization considerably impacts the levels of memory fragmentation, especially at the OS and the hypervisor layers. In this case, based on this study, the recommended approach is to reboot the OS rather than the VM. Importantly, we demonstrate that a VM reboot (destroy & create routines) causes approximately 30% more memory fragmentation than only rebooting the OS inside the VM. Since fast OS is the best option to reduce the rejuvenation overhead at the OS level, if we employ a fast OS reboot inside the VM, it will be the best option if virtualization is required. Finally, rebooting a virtual machine and a physical machine (cold) has similar effects on the client side, but very different effects on the server side, where the mechanism of rebooting VM is not effective against memory fragmentation. Hence, surprisingly, executing a physical machine cold reboot is better than a virtual machine reboot from both the client (number of failed requests) and server side points of view.

Finally, we remark about the extrapolation of our experimental study with a web application to other software applications. The numerical results are indeed dependent on the application, in this case on the workload. However the comparison between rejuvenation strategies may be easily generalized, since the difference between the techniques seems quite independent of the application under study.

6. Rejuvenation scheduling

In order to enhance the availability and mitigate the software aging effects, it is critical to design rejuvenation maintenance policies. Two main rejuvenation scheduling approaches can be defined: time-based and inspection-based.

Time-based approaches trigger rejuvenation at predetermined points of time. If based on monitoring or root cause detection techniques, we are able to figure out the state of various aging indicators, then we can apply rejuvenation on the exact aged layer (i.e., application layer, OS layer, or so on). However, this could be difficult in a real environment due to the growing complexity of the systems introduced by recent technologies (e.g., cloud computing) and heterogeneous environments where the systems have to interact with each other. In these scenarios, it is important to combine a time-based rejuvenation scheduling with granularity. This approach is based on applying different rejuvenation mechanisms based on their granularities, at different time intervals. Fig. 26 depicts the logic of this approach. The goal of this approach is focused on guaranteeing that after a complete rejuvenation maintenance cycle, the aging effects would be removed from all levels. In this type of approach, the main issue is how to determine the optimal rejuvenation trigger interval [46] and also, how to judiciously combine the different rejuvenation granularities in order to reduce as much as possible the rejuvenation overhead.

Inspection-based rejuvenation requires the observation of the state of the system and applying the most suitable rejuvenation mechanism based on the state determined by the inspection. As before, if we are able to determine the aged layer, the rejuvenation should be carried out at that level. However, if we are not able to determine the proper rejuvenation layer, a backup policy of maintenance needs to be defined. In this case, the recommended maintenance approach is rejuvenation followed by further inspection. After applying a rejuvenation mechanism, the system enters in an inspection time period, where it is monitored to evaluate if the aging effects are removed. If they are not removed, a new

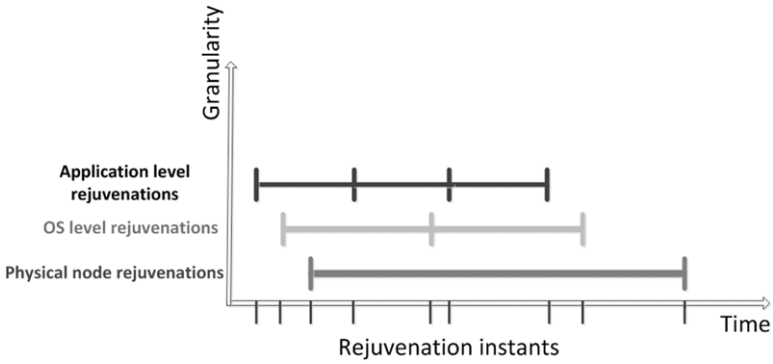

Fig. 26. Intuitive process of time-based rejuvenation scheduling with granularity.

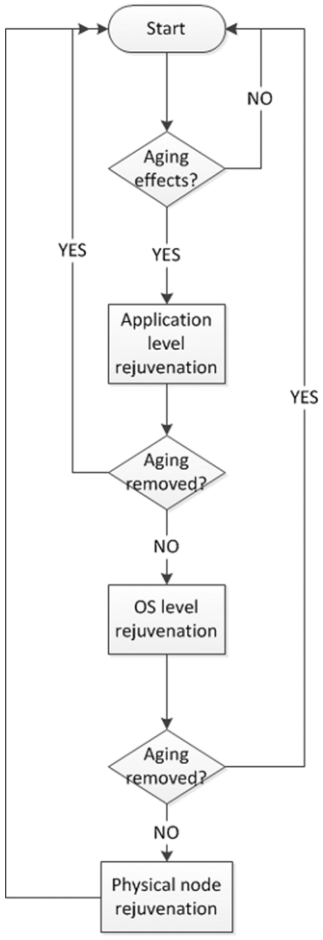

Fig. 27. Inspection-based escalated levels of rejuvenation.

rejuvenation mechanism (presumably at a higher layer) is triggered. This process needs to be repeated until the aging effects have been totally removed. The idea of inspection-based escalated levels of rejuvenation was first introduced in [47], and we improve it by adding priorities to the flow of rejuvenation, based on the rejuvenation granularity concept as well as the experimental findings obtained in this paper. The rationale of this approach is presented as a flow chart in Fig. 27.

Based on the results presented in Section 5, a guideline can be derived for designing rejuvenation scheduling, mainly based on the inspection-based escalated levels of the rejuvenation approach. We note that the rejuvenation overhead per technique, in terms of failed requests, could be assumed to be time independent. This means that the overhead of the rejuvenation at application granularity would be the same no matter at what instance we apply it. Based on this, we estimate the total number of failed requests of different inspection-based escalated levels of rejuvenation using the six rejuvenation

Table 10
Rejuvenation maintenance scheduling proposals.

Maintenance scheduling	Chain of rejuvenation techniques
Non VM environment 1	Standalone + OS reboot + Physical reboot
Non VM environment 2	Standalone + fast OS reboot + Physical reboot
Non VM environment 3	Hot standby + OS reboot + Physical reboot
Non VM environment 4	Hot standby + fast OS reboot + Physical reboot
VM environment 1	Standalone + OS reboot + VM reboot + Physical reboot
VM environment 2	Standalone + fast OS reboot + VM reboot + Physical reboot
VM environment 3	Hot standby + OS reboot + VM reboot + Physical reboot
VM environment 4	Hot standby + fast OS reboot + VM reboot + Physical reboot
Bypass 1	Standalone + Physical reboot
Bypass 2	Standalone in VM + Physical reboot
Bypass 3	Hot standby + Physical reboot
Bypass 4	Standalone + VM reboot + Physical reboot
Bypass 5	Hot standby + VM reboot + Physical reboot

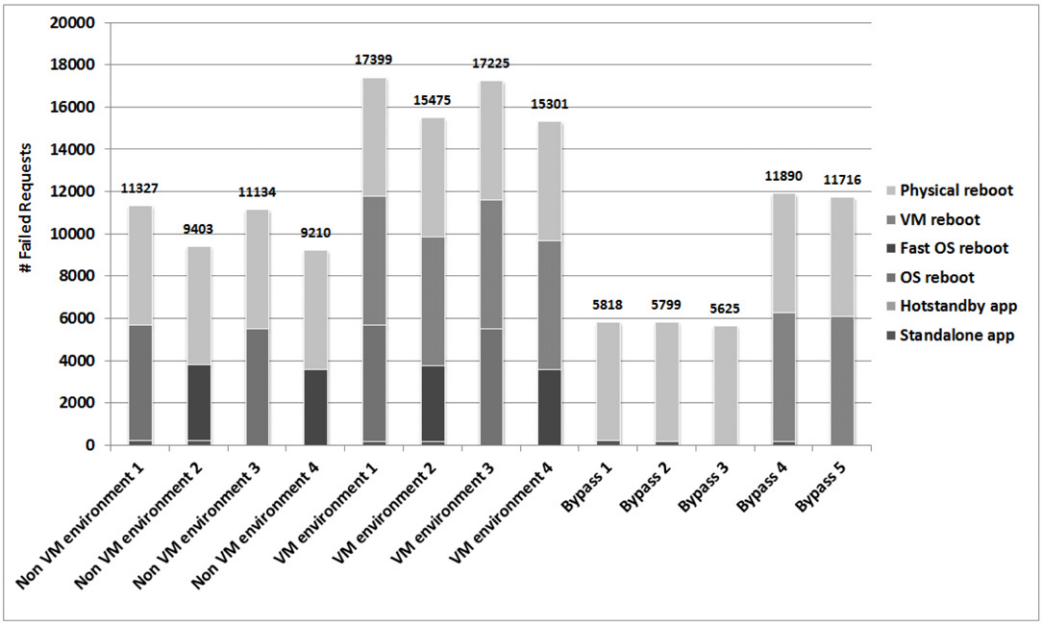

Fig. 28. Comparison of different escalated levels of rejuvenation based on the total failed requests.

mechanisms evaluated. The total number of failed requests represents the sum of the failed requests per technique used in the rejuvenation flow. Table 10 presents all the inspection-based escalated levels of rejuvenation policies analyzed.

We note that five extra rejuvenation scheduling policies have been added labeled bypass scheduling. The notion of bypass scheduling is to omit some rejuvenation granularity levels in the overall rejuvenation chain. Basically, these five proposals are defined considering that the number of failed requests caused by OS reboot and physical node reboot are practically the same (approx. 2% of difference). We, thus, bypass the OS rejuvenation granularity level in order to reduce the estimated rejuvenation scheduling overhead. Hence, our proposal of inspection-based escalated levels of rejuvenation hinges two main levels: application and physical node. If using virtualization, we add a third level: VM Reboot. However, based on the results obtained in our experimental study, we recommend to avoid VM rejuvenation approaches to reduce the memory fragmentation at the hypervisor level.

Fig. 28 summarizes the estimated results of every maintenance scheduling technique proposed. We clearly see how the bypass scheduling is more effective in reducing the rejuvenation overhead.

7. Conclusions

This paper presents an experimental evaluation of six rejuvenation strategies categorized in terms of granularity. We have conducted eight experiments with virtualized and non-virtualized environments in order to quantify the influence of this technology on the overhead of the rejuvenation strategies under consideration. The results show that the overhead impact of the rejuvenation techniques is related to their granularity. Fine-grain techniques such as application-level rejuvenation

strategies are better as a first tentative approach to mitigate the aging effects. If this first approach fails, we can use a rejuvenation technique from the next higher level of granularity. As a part of the evidence of the overhead introduced by every strategy under analysis, we observe a very important result related to the virtualization technology. Virtualization was the main cause of the drastic increase in memory fragmentation, which can cause aging-related failures in long-running systems [23]. This means that in the long term, virtualization technology could degrade the performance of the host operating system where the hypervisor is running. This finding is especially important nowadays, since virtualization is a core technology of the new cloud-computing paradigm. Hence, selecting the right rejuvenation approach to be used in cloud-based systems in order to reduce the aging-related effects in the virtualization layer is a major requirement. Finally, we present guidelines for the use of the appropriate scenario for each rejuvenation technique under consideration.

Acknowledgments

This research was supported in part by the NASA Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) under the JPL subcontract # 1440119. We also thank CNPq (National Research Council of Brazil) for the financial support. We also thank Daniel Tes for his help during the test bed setup.

References

[1] Y. Huang, C. Kintala, N. Kolettis, N. Fulton, Software rejuvenation: analysis, module and applications, in: The 15th International Symposium on Fault-Tolerant Computing, FTCS '95, 1995, pp. 381–390.
[2] M. Grottke, R. Matias, K. Trivedi, The fundamentals of software aging, in: The 1st Intl. Workshop on Software Aging and Rejuvenation/19th IEEE Symp. on Software Reliability Engineering, 2009, pp. 1–6.
[3] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (2004) 11–33.
[4] K. Vaidyanathan, K. Trivedi, A comprehensive model for software rejuvenation, IEEE Transactions on Dependable and Secure Computing 2 (2) (2005) 124–137.
[5] J. Alonso, L. Silva, A. Andrzejak, P. Silva, J. Torres, High-available grid services through the use of virtualized clustering, in: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, GRID '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 34–41.
[6] L. Silva, J. Alonso, J. Torres, Using virtualization to improve software rejuvenation, IEEE Transactions on Computers 58 (11) (2009) 1525–1538.
[7] R. Matias, P.J.F. Filho, An experimental study on software aging and rejuvenation in web servers, in: Proceedings of the 30th Annual International Computer Software and Applications Conference - Vol. 01, IEEE Computer Society, Washington, DC, USA, 2006, pp. 189–196.
[8] A. Andrzejak, L. Silva, Using machine learning for non-intrusive modeling and prediction of software aging, in: IEEE/IFIP Network Operations & Management Symposium, NOMS 2008, 2008, pp. 7–11.
[9] J. Alonso, J. Torres, J. Berral, R. Gavalda, Adaptive on-line software aging prediction based on machine learning, in: Dependable Systems and Networks, DSN, 2010 IEEE/IFIP International Conference on, 2010, pp. 507–516.
[10] K. Vaidyanathan, R.E. Harper, S.W. Hunter, K.S. Trivedi, Analysis and implementation of software rejuvenation in cluster systems, in: Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '01, ACM, New York, NY, USA, 2001, pp. 62–71.
[11] R. Matos, J. Araujo, P. Maciel, F. Vieira de Souza, R. Matias, K. Trivedi, Software rejuvenation in eucalyptus cloud computing infrastructure: a hybrid method based on multiple thresholds and time series prediction, International Transactions on Systems Science and Applications 8 (2012) 1–16.
[12] M. Grottke, L. Li, K. Vaidyanathan, K.S. Trivedi, Analysis of software aging in a web server, IEEE Transactions on Reliability 55 (2006) 411–420.
[13] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan, W.P. Zeggert, Proactive management of software aging, IBM Journal of Research and Development 45 (2) (2001) 311–332.
[14] L.M. Silva, J. Alonso, P. Silva, J. Torres, A. Andrzejak, Using virtualization to improve software rejuvenation, in: Sixth IEEE International Symposium on Network Computing and Applications, NCA 2007, IEEE Computer Society, Cambridge, MA, USA, 2007, pp. 33–44.
[15] T. Dohi, K. Goseva-Popstojanova, K.S. Trivedi, Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule, in: Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing, PRDC '00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 77–84.
[16] A.T. Tai, H. Hecht, S.N. Chau, L. Alkalaj, On-board preventive maintenance: analysis of effectiveness and optimal duty period, in: The 3rd Workshop on Object-Oriented Real-Time Dependable Systems, WORDS '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 40–47.
[17] D. Wang, W. Xie, K.S. Trivedi, Performability analysis of clustered systems with rejuvenation under varying workload, Performance Evaluation 64 (2007) 247–265.
[18] S. Garg, A. Van Moorsel, K. Vaidyanathan, K.S. Trivedi, A methodology for detection and estimation of software aging, in: The 9th International Symposium on Software Reliability Engineering, ISSRE '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 283–292.
[19] C.M. Kintala, Software rejuvenation in embedded systems, Journal of Automata, Languages and Combinatorics 14 (2009) 63–73.
[20] K. Kourai, S. Chiba, Fast software rejuvenation of virtual machine monitors, IEEE Transactions on Dependable and Secure Computing 8 (2011) 839–851.
[21] J. Alonso, R. Matias, E. Vicente, A.M. Carvalho, K. Trivedi, A comparative evaluation of software rejuvenation strategies, in: IEEE Third International Workshop on 2011, Software Aging and Rejuvenation, WoSAR, 2011, pp. 26–31.
[22] R. Matias, I. Beicker, B. Leitao, P. Maciel, Measuring software aging effects through OS kernel instrumentation, in: IEEE Second International Workshop on 2010, Software Aging and Rejuvenation, WoSAR, 2010, pp. 1–6.
[23] A. Macedo, T. Ferreira, R. Matias, The mechanics of memory-related software aging, in: IEEE Second International Workshop on 2010, Software Aging and Rejuvenation, WoSAR, 2010, pp. 1–5.
[24] W. Gloger, ptmalloc @ONLINE, Feb. 2012. URL http://www.malloc.de/en/.
[25] M.T. Jones, Anatomy of the linux slab allocator @ONLINE, Feb. 2012. URL http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/#N102CA.
[26] P. Mundt, Asymmetric numa: Multiple-memory management for the rest of us @ONLINE, Feb. 2012. URL http://mirror.averse.net/pub/linux/kernel/people/lethal/papers/elce2007-numa.pdf.
[27] K. Su, H. Fu, J. Li, D. Zhang, Software rejuvenation in virtualization environment, in: Information Technology and Artificial Intelligence Conference, ITAIC, 2011 6th IEEE Joint International, Vol. 2, 2011, pp. 329–331.
[28] T. Thein, S.-D. Chi, J.S. Park, Improving fault tolerance by virtualization and software rejuvenation, in: Second Asia International Conference on Modeling Simulation, 2008, AICMS 08, 2008, pp. 855–860.
[29] A. Pfiffer, Reducing system reboot time with kexec @ONLINE, Nov. 2011. URL http://devresources.linux-foundation.org/andyp/kexec/whitepaper/kexec.pdf.
[30] K. Yamakita, H. Yamada, K. Kono, Phase-based reboot: reusing operating system execution phases for cheap reboot-based recovery, in: Dependable Systems Networks, DSN, IEEE/IFIP 41st International Conference on 2011, 2011, pp. 169–180.

[31] J.N. Herder, H. Bos, B. Gras, P. Homburg, A.S. Tanenbaum, Reorganizing unix for reliability, in: Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture, ACSAC'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 81–94.
[32] S. Sundararaman, S. Subramanian, A. Rajimwale, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, M.M. Swift, Membrane: operating system support for restartable file systems, in: Proceedings of the 8th Conference on File and Storage Technologies, FAST '10, San Jose, California, 2010, pp. 281–294.
[33] M.M. Swift, M. Annamalai, B.N. Bershad, H.M. Levy, Recovering device drivers, ACM Transactions on Computer Systems 24 (2006) 333–360.
[34] J.N. Herder, H. Bos, B. Gras, P. Homburg, A.S. Tanenbaum, Failure resilience for device drivers, in: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 41–50.
[35] M.M. Swift, S. Martin, H.M. Levy, S.J. Eggers, Nooks: an architecture for reliable device drivers, in: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10, ACM, New York, NY, USA, 2002, pp. 102–107.
[36] O. Laadan, S.E. Hallyn, Linux-cr : transparent application checkpoint-restart in linux, in: Proceedings of the 12th Annual Linux Symposium, no. July, Ottawa, Canada, 2010, pp. 159–172.
[37] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox, Microreboot — a technique for cheap recovery, in: The 6th Conference on Symposium on Operating Systems Design & Implementation, OSDI'04, USENIX Association, Berkeley, CA, USA, 2004, pp. 31–41.
[38] J. Araujo, R. Matos, P. Maciel, R. Matias, I. Beicker, Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure, in: Proceedings of the Middleware 2011 Industry Track Workshop, Middleware'11, ACM, New York, NY, USA, 2011, pp. 1–7.
[39] D. Cotroneo, R. Natella, R. Pietrantuono, S. Russo, Software aging analysis of the linux operating system, in: IEEE 21st International Symposium on 2010, Software Reliability Engineering, ISSRE, 2010, pp. 71–80.
[40] R. Figueiredo, P. Dinda, J. Fortes, Guest editors' introduction: resource virtualization renaissance, Computer 38 (5) (2005) 28–31.
[41] Tpc-w benchmark java version @ONLINE, Feb. 2012. URL http://pharm.ece.wisc.edu/.
[42] Apache tomcat @ONLINE, Feb. 2012. URL http://tomcat.apache.org/.
[43] D. Garcia, J. Garcia, Tpc-w e-commerce benchmark evaluation, Computer 36 (2) (2003) 42–48.
[44] D. Menasce, Tpc-w: a benchmark for e-commerce, Internet Computing, IEEE 6 (3) (2002) 83–87.
[45] D.A. Menasce, V.A.F. Almeida, Scaling for E Business: Technologies, Models, Performance, and Capacity Planning, first ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.
[46] W. Xie, Y. Hong, K. Trivedi, Analysis of a two-level software rejuvenation policy, Reliability Engineering & System Safety 87 (1) (2005) 13–22.
[47] K. Trivedi, G. Ciardo, B. Dasarathy, M. Grottke, A. Rindos, B. Varshaw, Achieving and assuring high availability, in: IEEE International Symposium on 2008, Parallel and Distributed Processing, IPDPS 2008, 2008, pp. 1–7.

Javier Alonso received the master's degree in Computer Science in 2004 and the Ph.D. degree from the Technical University of Catalonia (Universitat Politecnica de Catalunya, UPC) in 2011, respectively. From 2006 to 2011 he was an assistant lecturer at the Computer Architecture Department at UPC. He is currently a postdoctoral associate under the supervision of Professor K.S. Trivedi, at Duke University, Durham, NC. Dr. Alonso has served as a reviewer for IEEE TRANSACTIONS ON COMPUTERS, PERFORMANCE EVALUATION, and several international conferences. His research interests in are in dependability, reliability, availability, performance, performability and survivability modeling of computer and communication systems.

Rivalino Matias, Jr. received his B.S. (1994) in informatics from the Minas Gerais State University, Brazil. He earned his M.S (1997) and Ph.D. (2006) degrees in Computer Science, and Industrial and Systems engineering from the Federal University of Santa Catarina, Brazil, respectively. In 2008 he was with the Department of Electrical and Computer Engineering at Duke University, Durham, NC, working as a research associate under the supervision of Dr. Kishor Trivedi. He also works for IBM Research Triangle Park in a research related to embedded system availability and reliability analytical modeling. He is currently an Associate Professor in the Computer School at Federal University of Uberlândia, Brazil. Dr. Matias has served as a reviewer for IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, JOURNAL OF SYSTEMS AND SOFTWARE, and several international conferences. His research interests include dependability applied to computing systems, software aging theory, and diagnosis protocols for computing systems.

Elder Vicente received his B.S.(2007) in Control and Automation Engineering from the Polytechnic School of Uberlândia, Brazil. He earned his M.S.(2012) degree in Computer Science from the Federal University of Uberlândia, Brazil. His recent research interests include techniques for optimizing and evaluating the performance of operating systems.

Ana Maria Martins Carvalho received her B.S. (1998) in Informatics from Uberaba University, Brazil. She earned her M.S. (2012) degree in Computer Science from Federal University of Uberlândia, Brazil. Her recent research interests include statistical techniques applied to computing systems and computer network traffic analysis.

Kishor S. Trivedi (M'86–SM'87–F'92) holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He has been on the Duke faculty since 1975. He is the author of a well known text entitled, Probability and Statistics with Reliability, Queuing and Computer Science Applications, published by Prentice-Hall; a thoroughly revised second edition (including its Indian edition) of this book has been published by John Wiley. He has also published two other books entitled Performance and Reliability Analysis of Computer Systems, published by Kluwer Academic Publishers; and Queueing Networks and Markov Chains, published by John Wiley. He is a Fellow of the Institute of Electrical and Electronics Engineers. He is a Golden Core Member of IEEE Computer Society. He has published over 420 articles, and has supervised 42 Ph.D. dissertations. He is on the editorial boards of IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, JOURNAL OF RISK AND RELIABILITY, INTERNATIONAL JOURNAL OF PERFORMABILITY ENGINEERING, AND INTERNATIONAL JOURNAL OF QUALITY AND SAFETY ENGINEERING. He is the recipient of IEEE Computer Society Technical Achievement Award for his research on Software Aging and Rejuvenation.

His research interests in are in reliability, availability, performance, performability and survivability modeling of computer and communication systems. He works closely with industry in carrying out reliability/availability analysis, providing short courses on reliability, availability, performability modeling, and in the development and dissemination of software packages such as SHARPE and SPNP.