Software Aging Analysis of the Linux Operating System

Domenico Cotroneo*, Roberto Natella*, Roberto Pietrantuono*, Stefano Russo*†
*Dipartimento di Informatica e Sistemistica,
Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
†Laboratorio CINI-ITEM "Carlo Savy",
Complesso Universitario Monte Sant'Angelo, Ed. 1, Via Cinthia, 80126, Naples, Italy
{cotroneo, roberto.natella, roberto.pietrantuono, sterusso}@unina.it

***Abstract*—Software systems running continuously for a long time tend to show degrading performance and an increasing failure occurrence rate, due to error conditions that accrue over time and eventually lead the system to failure. This phenomenon is usually referred to as *Software Aging*. Several long-running mission and safety critical applications have been reported to experience catastrophic aging-related failures. Software aging sources (i.e., aging-related bugs) may be hidden in several layers of a complex software system, ranging from the Operating System (OS) to the user application level.**

This paper presents a software aging analysis at the Operating System level, investigating software aging sources inside the Linux kernel. Linux is increasingly being employed in critical scenarios; this analysis intends to shed light on its behaviour from the aging perspective. The study is based on an experimental campaign designed to investigate the kernel internal behaviour over long running executions. By means of a kernel tracing tool specifically developed for this study, we collected relevant parameters of several kernel subsystems. Statistical analysis of collected data allowed us to confirm the presence of aging sources in Linux and to relate the observed aging dynamics to the monitored subsystems behaviour. The analysis output allowed us to infer potential sources of aging in the kernel subsystems.

***Keywords*-Software aging, Linux kernel, trend analysis**

I. Introduction

Software Aging can be defined as a continued and growing degradation of software's internal state during its operational life. This problem leads to progressive performance degradation, occasionally causing system crashing. Due to its cumulative property, it occurs more intensively in continuously running systems that execute over a long period of time. It is typically caused by accrued error conditions, such as round-off errors, data corruption, storage space fragmentation, or unreleased memory regions.

Detecting and removing the sources of software aging (i.e., the so-called *aging-related bugs* [1]) is very difficult at testing time, since aging becomes evident only after a long operational time. For this reason, software aging represents one of the most subtle dependability threats in today's business- and safety-critical software systems.

Past research work reported software aging phenomena that manifested as the increasing consumption of Operating System (OS) resources, such as free memory and swap space exhaustion [2]–[4]. Subsequent studies found software aging sources in several software applications, such as web servers [4], telecommunication systems [5], and SOAP servers [6]. Therefore, several approaches were developed to predict the time to failure at operational time, in order to plan proper actions (that are usually referred to as *software rejuvenation*) with an optimal schedule (i.e., neither too early, because it would be expensive, nor too late, because a failure may occur before rejuvenation).

Although relevant, the analysis of software aging sources at application level represents only a partial view of the issue. In fact, the OS itself can be a source of software aging phenomena, since it is a large and bug-prone part of complex software systems [7]. Being able to detect and isolate the aging contribution of the OS would yield insights about aging trends for a wide number of applications based on it. Moreover, these insights can be exploited for planning software rejuvenation strategies tailored to the OS [8], as well as for identifying aging-related bugs in the OS code.

In this work we carry out an experimental campaign to analyze software aging inside the Linux OS kernel. First, the study tests the presence of aging sources at the OS level. The goal of this phase is to statistically confirm if and in what extent the Linux kernel is actually affected by aging-related bugs. A deeper analysis is then carried out, with the goal of figuring out how the usage of each internal subsystem impacts on aging trends. By means of a kernel tracing tool specifically developed for this study, we collected usage information about several subsystems, such as memory management and the filesystem. Usage information has been related with the observed aging trends, by means of *multiple linear regression* and *principal components analysis*; these relationships were exploited to find out kernel subsystems responsible for aging phenomena.

From our experimental analysis, it results that aging sources actually exist in the Linux kernel; they manifested as a statistically significant aging trend of memory consumption in all our experiments. This result is of practical importance for final users, which can benefit from a rejuvenation schedule that individually takes into account aging at the OS

and the application layer. Moreover, the analysis of internal subsystems identified a set of potential aging sources in the *filesystem* and *process management subsystem*, which manifested a significant contribution to the overall aging trend. A further experiment allowed us also to quantify a non-negligible contribution of the filesystem to the memory consumption trend, which impacts the overall aging effects in a large, complex software system.

The rest of this paper is organized as follows: Section II surveys the past work related to software aging. Section III describes the experimental procedure and setup, outlining the steps followed to carry out the analysis. Section IV reports the statistical analysis results and discusses potential sources of software aging, while section V concludes the paper.

II. Related work

Until some years ago, software aging was judged as an occasional phenomenon experienced only by few badly designed systems. As more and more studies reported experiences of systems failed due to software aging, it has been progressively recognized as a systematic, non-negligible problem of long-running systems.

Software aging has been observed in a number of systems massively employed for both business- and safety-critical scenarios. Relevant examples include the Apache Web Server [4], [9], the Java Virtual Machine [10], the AT&T billing applications and telecommunications switching software [5], military software, e.g., the famous incident occurred during the first gulf war [1], where an accrued round-off error caused a miscalculation of the target's expected position.

Currently, the best way to counteract aging is *software rejuvenation*, in which the operational software is occasionally stopped and then restarted in a "clean" internal state [11]. As a consequence, several studies on software aging focused on how to assess the system's runtime health in order to estimate the expected time to resource exhaustion (i.e., the *Time to Exhaustion (TTE)*), and determine the optimal rejuvenation schedule (i.e., *when* it is better to rejuvenate). Proposed solutions are broadly classified into two classes: *analytic modelling* and *measurement-based* approaches.

Analytic modelling typically determines the optimal rejuvenation schedule starting from models. Stochastic processes representing the system's states are adopted to model the software affected by aging. Models representing the aging phenomenon assume failure and repair time distributions, and compute the rejuvenation schedule that maximizes the system availability, or, equivalently, that minimizes the downtime cost. Modelling approaches differ from one another in the chosen distributions and in the adopted stochastic process (e.g., Markov Decision Processes, Stochastic Petri Nets) [12]–[14].

The **measurement-based** approach applies statistical analysis to data collected from system execution to determine a time window over which to perform rejuvenation. The idea is to directly monitor the system affected by aging in order to obtain predictions about possible impending failures due to resource exhaustion or to performance degradation.

In [2] and [3], authors report results of a measurement-based analysis, carried out on a network of 9 UNIX workstations; they monitored OS resources for 53 days by using an SNMP-based tool. During the observation period, the 33% of reported outages were due to resource exhaustion, highlighting that software aging is a non-negligible source of failures. In [3], some workload parameters are also taken into account (e.g., the number of CPU context switches, the number of system call invocations). The work highlighted the need to consider the workload variation when studying software aging phenomena, since different results are observed under different workload conditions. Authors first identified workload states through statistical cluster analysis; then they built a state-space model determining sojourn time distributions, and defined a reward function, based on the resource exhaustion rate, for each workload state. By solving the model, they obtained resource depletion trends and TTE for each considered resource in each workload state. Although [2], [3] considered OS resources and OS-related parameters to model the workload, authors analyzed the software aging of the *whole system*, not focusing on the investigation and isolation of the aging contribution from the OS.

In [15], Gross et *al.* deal with performance degradation, rather than with the (most common) resource exhaustion. They applied pattern recognition methods to detect aging in shared memory pool latch contention in large OLTP servers. In [4], Trivedi et al. analyzed performance degradation in the Apache Web Server by sampling web server's response time to HTTP requests at fixed intervals. Software Aging in a SOAP-based server running on top of a Java Virtual Machine was analyzed in [6]. For each considered distribution, throughput loss and memory depletion highlighted the presence of software aging. An analysis addressing the impact of workload parameters on aging trends has been presented in [9], where the memory consumed by an Apache Web Server was observed together with three controllable workload parameters: page size, page type (dynamic or static), and request rate.

Trivedi and Vaidyanathan in [16] proposed a combined solution, by composing analytic modelling with a measurement-based approach. They built a measurement-based semi-Markovian model for system workload, estimating the time to exhaustion (TTE) for each considered resource and workload state, and finally building a semi-Markovian availability model, based on field data rather than on assumptions about system behaviour.

Among the mentioned papers, [4], [6], [10] investigated software aging sources at the user application level, e.g., in the Apache Web Server [4], or in the SOAP-based server [6]. Past work focused on the intermediate levels of the software stack, also confirming the presence of aging sources, respectively in the JVM [10] and at middleware layer [17]. Moreover, in [4], [10], aging phenomena are attributed to software aging sources in the OS. In this paper, we test this hypothesis by analyzing aging sources *inside* the OS. Our work relies on a *workload- and measurement-based* approach, in that we monitor the OS's health under different controlled workloads, collect data characterizing its behaviour, and then analyze them to identify aging sources inside the Linux kernel.

III. Experimental Framework

A. Overview of the approach

The characterization of software aging phenomena we carried out consists of two parts: (i) the detection of aging phenomena at the OS layer, and (ii) the detection of potential correlations between aging phenomena and system workload. The former is devoted to isolate and quantify the contribution of the OS to software aging in the system. The latter aims at identifying relevant relationships between system workload parameters, describing the usage of OS subsystems, and the observed aging, if any.

To correctly isolate the aging contribution at OS layer, we need to prevent the occurrence of additional aging at the application layer. For example, if both applications and the OS are affected by memory leaks, it is difficult to quantify the contribution of the OS to the overall leaked memory. Therefore, we adopted an approach based on *controlled stress tests*, in which the applications running in the system under test are carefully selected. In particular, we removed from the system all processes that could interfere with experiments and contribute to software aging (e.g., system daemons). Moreover, the system was stressed by means of a *load generator*, which was screened before experiments in order to assure that it is aging-free. The load generator stresses several subsystems (e.g., process management, memory management) by using system calls provided by the OS (e.g., by allocating memory and by writing to the disk).

In order to highlight correlations between the system workload and the aging phenomena, we designed a stress testing campaign that includes several experiments. An experiment is characterized by a set of *application-level workload conditions* that can be varied by the experimenter. Each experiment is executed for a long time period, during which the application-level workload conditions are kept constant. Workload conditions vary between experiments, in order to point out how system usage is related to aging phenomena.

The setup of a stress test experiment is shown in Figure 1, and the collected information is detailed in Section III-B. During an experiment, we periodically collect information about aging phenomena, by analyzing indicators in the system about performance and resource usage (*aging indicators*) that may exhibit aging symptoms. Past studies also show that software aging manifests itself as resource depletion and performance degradation [6], [9], [18]. Data about the usage of OS subsystems' functionalities (i.e., the *workload parameters*, such as number of interrupts or disk operations processed in a time period), are also collected and exploited to discover correlations between the system workload and aging phenomena.

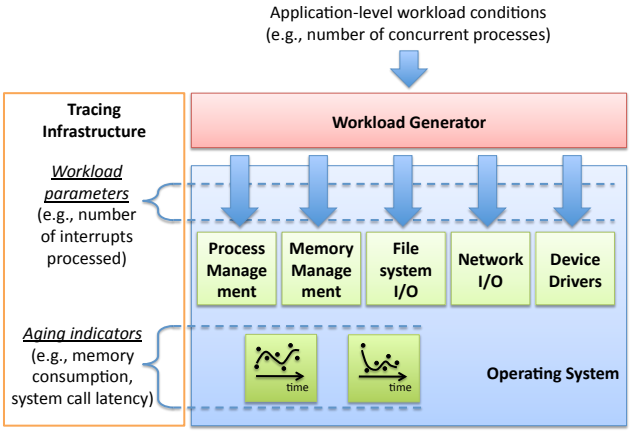

Figure 1. Setup of a stress test.

After the experimental campaign, a software aging analysis is performed on the collected data (Section III-C). First, the occurrence of aging phenomena is detected by means of *statistical trend analysis* on aging indicators, which aims to assess whether the data exhibit a trend with statistical significance. If an aging trend is detected, it is then related to the system workload parameters by means of *multiple linear regression* and *principal component analysis*. This step points out linear relationships between workload parameters and aging trends. In turn, a workload parameter correlated to aging phenomena can be a symptom of aging bugs in a specific subsystem, therefore these relationships can be exploited to diagnose aging phenomena in OS subsystems.

B. Aging indicators and workload parameters

We identified two aging indicators in our analysis, namely the *memory consumption* (MC) and the *system call latency* (SCL) (Table I). Memory consumption is considered since memory is the resource most affected by aging phenomena and with the lowest TTE, as shown by several past studies [2], [4]. Memory consumption is given by:

$$MC = TM - FM - PC \quad (1)$$

where TM, FM, and PC are total memory, free memory, and page cache size respectively. This metric is provided by

the standard Linux kernel; it can be queried by means of the *free* utility. The page cache contains a copy of recently accessed files in kernel memory. Since the page cache can get all the free memory not allocated by the kernel or user processes, its memory consumption is quite large and would bias our analysis; therefore, it is subtracted from MC. The MC is periodically sampled and stored in a trace.

The SCL indicator is considered since bugs that affect the system performance, such as aging-related bugs, may affect performance of services at the OS interface. Moreover, performance degradation is a common symptom of aging phenomena due to the accumulation of errors and stale resources, as shown in [6], [9], [18]. Latency is the time duration between the invocation and the completion of a system call, which we measure in CPU clock counts using performance monitoring registers within the CPU. We analyze the latency of a subset of system calls (the ones included in Table II). For each system call, we periodically collect a sample of the latency distribution (i.e., an histogram) of all invocations made since the previous sample. In particular, a set of n counters is allocated for each traced system call; if the latency l of an invocation is between 2^i and 2^{i+1} clock cycles, then the i-th counter is incremented. In order to collect this data, we included the *OSProf* profiler [19] in our tracing infrastructure (detailed in Section III-D).

Table I
AGING INDICATORS.

Indicator	Unit of measurement
Memory Consumption (MC)	kB
System Call Latency (SCL)	# clock cycles per invocation

Along with aging indicators, data about the subsystems usage (i.e., the system workload parameters) is also collected. We identified the major 5 subsystems within the Linux kernel [20], [21]; each subsystem was analyzed to identify a set of workload parameters to characterize the amount of activity in the subsystem. Kernel subsystems and the corresponding workload parameters implemented in our tracing infrastructure are shown in Table II (they are discussed in Section IV-B).

C. Aging analysis

The analysis of experimental data is split in three steps: **i)** *trend detection*, **ii)** *workload-aging correlation analysis*, and **iii)** *analysis of potential aging sources*.

We adopt statistical **hypothesis testing** to assess the presence of a trend in the data. Trend detection has been performed by means of the Mann-Kendall test, which was adopted also in [2], [4]. The trend, if any, has been estimated by means of the non-parametric Sen's procedure [2], [4], which calculates the median slope of all pairs of data points. The Sen's procedure is known to be robust, i.e., it does not assume normally distributed measurement errors, and resistant, i.e., it is not sensitive to outliers.

We observed that workload parameters stressed by the experiments were normally distributed around their mean value. However, to relate the system workload parameters to the revealed aging trends, we need to consider the

Table II
KERNEL SUBSYSTEMS AND THEIR WORKLOAD PARAMETERS.

Subsystem	Parameter	Description
Process Management	TSK-CTS	# of context switches
	TSK-CRT	# of tasks created
	TSK-DEL	# of tasks removed
	TSK-MIG	# of tasks migrated among CPUs
	TSK-WAIT	# of tasks on a waitqueue
	TSK-PMPT	# of tasks preempted
	TSK-RUN	# of tasks entering in the *runnable* state
	TSK-URUN	# of tasks entering in the *un-runnable* state
	TSK-STP	# of tasks entering in the *stopped* state
	TSK-FRK	# of *fork* system calls
	TSK-CLN	# of *clone* system calls
	TSK-EXEC	# of *exec* system calls
Memory Management	MM-MMAP	# of *mmap* system calls
	MM-MUMP	# of *munmap* system calls
	MM-BRK	# of *brk* system calls
	MM-PGM	# of page misses
	MM-PFL	# of page cache flushes
	MM-PIN	# of page cache insertions
	MM-PRM	# of page cache removals
	MM-MALL	# of memory allocations
	MM-MDAL	# of memory deallocations
	MM-BREQ	# of bytes requested
	MM-BALL	# of bytes allocated
	MM-SWIN	# of swap-ins
	MM-SWOUT	# of swap-outs
Filesystem	FS-RD	# of file reads
	FS-WR	# of file writes
	FS-OPN	# of files opened
	FS-CLS	# of files closed
	FS-ACC	# of *access* system calls
	FS-SEK	# of *seek* system calls
Networking	NET-RD	# of socket reads
	NET-WR	# of socket writes
	NET-OPN	# of sockets opened
	NET-CLS	# of sockets closed
	NET-ACPT	# of *accept* system calls
	NET-LST	# of *listen* system calls
	NET-BND	# of *bind* system calls
	NET-CNT	# of *connect* system calls
	NET-TTR	# of TCP packets transmitted
	NET-TRC	# of TCP packets received
	NET-UTR	# of UDP packets transmitted
	NET-URC	# of UDP packets received
Device Drivers	DR-IRQ	# of interrupt requests
	DR-TSK	# of *tasklet*s executed
	DR-SIRQ	# of *softirq*s executed
	DR-WQIN	# of workqueue insertions
	DR-WQEX	# of workqueue executions
	DR-WQCR	# of workqueues created
	DR-WQDL	# of workqueues deleted

correlation among them. Correlation among data may distort the analysis due to the issue of multicollinearity, caused by the existence of inter-correlations among the data. For instance, the parameter FS-WR (number of writes) not only can correlate with aging, but it strongly correlates also with FS-OPN (number of files opened), and with others. Such an inter-correlation can lead to an *inflated variance* in the estimation of the dependent variable—that is, aging trend. This would cause correlated variables to be given a higher weight, thus actually amplifying the effects of such variables on aging trends. To overcome this issue, we used a standard statistical approach, namely the **principal component analysis** (PCA) [22], which is a technique that transforms the original data into uncorrelated data. To assure that all data series have the same weight, a normalization step has been first carried out, by the following method:

$$x'_i = \frac{x_i - min_i\{x_i\}}{max_i\{x_i\} - min_i\{x_i\}} \quad (2)$$

where x_i is the mean value of he *i-th* workload parameter during the whole experiment, and x'_i is the normalized value of x_i. Through this transformation, all the time series have been transformed into series whose values range between 0 and 1. From the original set of variables, the PCA computes new variables, called Principal Components (PCs), which are linear combination of original variables, such that all PCs are uncorrelated. From these new set of (normalized) variables, a subset of them able to explain the most of variance of original data is usually selected. Typically, a very small percentage of original variables (e.g., 10%) are able to explain from 85% to 90% of the original variance. Each of the principal components is expressed as:

$$PC_i = \sum_{j=1}^{m} a_{ij} x'_j \quad (3)$$

where $x'_j s$ are the original variables, m is the number of variables and a_{ij}s are coefficients expressing the weights that the *j-th* original variables has on the *i-th* principal component. With this technique, once the m PCs are obtained, a subset of them that contains as much variance as possible is selected. The chosen PCs will be the independent variables of the **multiple linear regression** model [23]. For each PC, a statistical hypothesis test is performed, to check whether there is a relation between the PC and trends. If not, the PC is excluded from the fitted model. The multiple regression step will assess the relationships between aging trends and the principal workload components; hence its output is the list of PCs with the greatest influence on measured aging.

A limitation of the PCA is that PCs do not have a physical meaning; they are a combination of original variables. Thus, once found an influence of a PC on the response variable, we need to analyze the major contributors to that PC, i.e., identifying those original variables that mostly impact that PC. Through the analysis of the composition of each principal component, it is possible to identify workload parameters more relevant to aging trends.

The identification of the most relevant subsystems' workload parameters drives the final step, i.e., the *aging sources analysis*. Subsystems whose parameters (or a combination of them) exhibit a strong correlation with aging are identified as potential sources of aging; then, depending on the specific system, tailored solutions can be taken. It should be noted that this analysis can identify actual software aging sources in the kernel, which are workload-independent; however, it is able to detect only those aging sources stimulated by the specific workload.

D. Experimental setup

The experiments were distributed among 8 servers[1] equipped with 2 Xeon 2.8 GHz CPUs with Hyper-Threading (4 CPUs are seen by the OS), 5 GB of RAM, Smart Array 5i Plus Disk Controller, and a 36 GB 15Krpm Ultra320 SCSI hard disk.

The kernel source code (version 2.6.30.1) was instrumented to collect the data required for aging analysis (Section III-B). The instrumentation of the system has been kept simple with intent, in order to prevent the instrumentation from affecting the aging indicators. The instrumentation consists of instructions in specific kernel code locations that increase a counter. The software counters are stored in a fixed-size kernel memory buffer. These counters are accessed through the `/debug` virtual filesystem [20] by a user-space process, which periodically stores them in a log and resets them to zero. The overall tracing infrastructure is sketched in Figure 2.

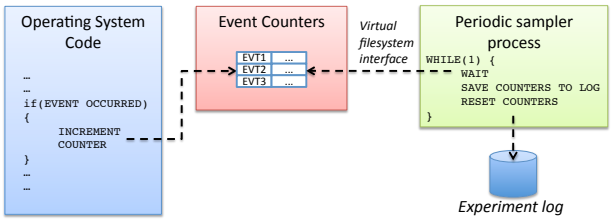

Figure 2. Tracing infrastructure.

Before each experiment, the system is restarted, and a minimal set of processes is started at boot. After booting, a workload generator is started and the tracing infrastructure is enabled. We adopted the Postmark[2] benchmark in our tests. This benchmark emulates a large email server, by performing a mix of data- and metadata-intensive operations on a pool of random text files [24], [25]. It first creates the pool of files with uniformly distributed sizes, and then

[1] http://h18000.www1.hp.com/products/quickspecs/11473_div/11473_div.html

[2] http://fsbench.filesystems.org/bench/postmark-1_5.c

it performs a sequence (namely *transaction*) of randomly selected I/O operations (e.g., file creation, delation, read, and append). We adopted the default configuration of Postmark parameters. The standard Postmark benchmark is single-threaded; however, since concurrency is an important aspect of complex OSs, we extended the benchmark by running several concurrent instances of Postmark, similarly to [26]. Each instance executes 10^5 consecutive transactions; the instance is then terminated and restarted. Finally, in order to ensure that the application layer is aging-free, we carefully inspected the workload source code; this was possible since the workload is implemented by a single small source file made up of 1500 lines of code.

The load generator allows us to control the application-level workload imposed to the OS by specifying the *number of concurrent processes* (N) executing in each experiment. To choose a proper range for the controllable parameter N, we performed a preliminary *capacity test* of the system, by executing different experiments with increasing N (starting from $N = 2$, which is the minimum to get a concurrent benchmark). We observe (Figure 3) that the server capacity (measured in I/O throughput) does not increase anymore after $N = 8$. The throughput is initially decreasing for $N \leq 4$ due to the complex interactions between concurrent processes in our multiprocessor systems (e.g., cache conflicts, lock contentions). We selected the range $2 \leq N \leq 9$ for our experiments, which encompasses several relevant workload scenarios for our analysis.

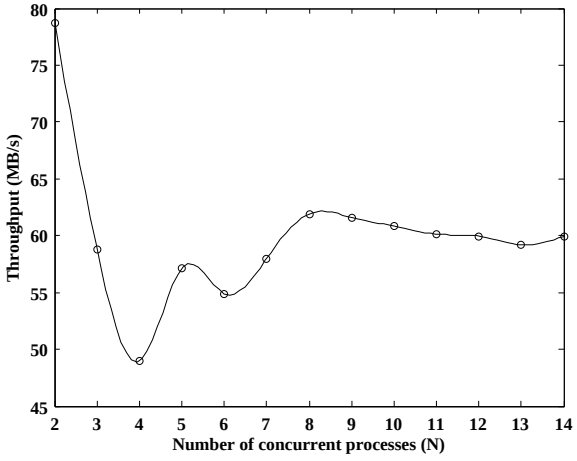

Figure 3. Capacity test.

As for the experiment duration, we do not know a priori how much time is needed to highlight aging phenomena in the Linux kernel. In general, this time duration is dependent on several factors such as the workload and the application type. A tentative duration of 4 days of execution was chosen on the basis of past work on software aging [10], [17]. After observing aging phenomena in the first experiment ($N = 1$), we perfomed the remaining experiments for that duration.

IV. Result analysis

In this section, we analyze the results provided by the experimental campaign. First, we provide evidence of aging phenomena manifested during the experiments; we then relate aging trends with system usage, and try to identify sources of aging into the OS.

A. Aging trend detection

In our experimental campaign, *we observed a memory consumption trend in every experiment*. Here and in the following, we say that a trend is detected when a confidence level of 95% or more is reached. Table III shows the estimated slopes of memory consumption trends, with the corresponding 95% confidence interval. Instead, *we did not find any significant trend in the system call latency indicator* (SCL). This result may be due to the specific workload we considered, which does not highlight appreciable software aging effects on performance. However, these results point out that memory consumption is a more significant problem than performance loss, therefore in the following we focus on aging phenomena related to the memory consumption.

Table III
Memory consumption trends.

	Slope	**p-value**	**95% conf. interval**
2	7.391 kB/h	7.667e-32	[6.393, 8.396] kB/h
3	3.148 kB/h	2.001e-08	[2.233, 4.066] kB/h
4	1.978 kB/h	7.494e-04	[1.018, 2.932] kB/h
5	10.153 kB/h	1.896e-67	[9.280, 11.036] kB/h
6	14.845 kB/h	1.530e-92	[13.817, 15.893] kB/h
7	11.892 kB/h	2.749e-71	[10.909, 12.872] kB/h
8	19.817 kB/h	3.023e-42	[17.468, 22.215] kB/h
9	16.090 kB/h	4.325e-114	[15.123, 17.056] kB/h

The memory consumption trend for the experiment $N = 7$ is shown in Figure 4, along with the collected samples. The memory consumption trend is most probably due to software aging phenomena: in fact, the OS operates in constant workload conditions for the whole experiment, and we excluded from the analysis the memory allocated for the page cache (see Section III-B). Therefore, memory consumption should not increase in the absence of software aging phenomena. The observed trends are relatively small if compared to aging trends described in past work [10]. However, as shown in Figure 4, the trend is non-negligible and it adds up to the memory consumption at the application layer, which can be significant and also be affected by aging phenomena, thus exacerbating the software aging issue. This motivates a more detailed analysis for the identification of aging sources in the OS (e.g., memory management bugs) and the need for software rejuvenation strategies tailored to the OS.

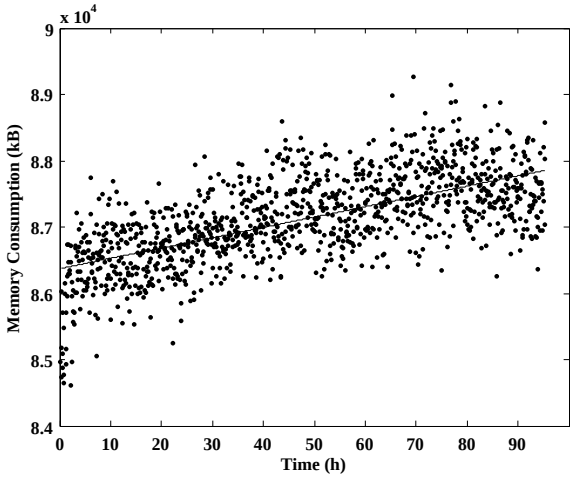

Figure 4. Memory consumption trend during the $N = 7$ test.

B. Workload characterization

In this section, we provide a workload characterization of the experiments, in order to support the analysis of software aging. A large set of workload parameters is initially considered (Section III-B), in order to include the factors that are potentially related to software aging. However, to minimize the effect of multicollinearity, we performed a PCA (Section III-C) obtaining a small set of representative principal components (PC).

In particular, workload parameters were grouped by subsystem, and each group was separately characterized. For each subsystem, we identified the minimal number of principal components accounting for *at least* 90% of the variance in the data. From the initial set of workload parameters, we identified a subset of 9 principal components. The low number of principal components (compared to workload parameters in Table II) is due to the inherent redundancy of information in the workload parameters. In particular, only 2 PCs were obtained for the Memory Management, the Device Driver, and the Filesystem subsystems (Tables IV, V, and VII), since the considered workload uniformly stressed factors related to I/O and memory management; in both cases, the weight of workload parameters is about the same (in absolute value) except for some parameters which were zero or close to zero (e.g., MM-SWIN and MM-SWOUT, since the physical memory was not saturated).

In the Process Management and Device Driver subsystems, respectively three and two PCs were identified (Tables VI and VII). In the case of Process Management, almost all workload parameters contributed to each PC, with the exception of TSK-PMPT (the number of preempted tasks was very low since the workload was I/O bound) and TSK-URUN (since we didn't explicitly stop any process during an experiment).

Table IV
PRINCIPAL COMPONENTS RELATED TO THE MEMORY MANAGEMENT SUBSYSTEM.

	MEM-PC1	**MEM-PC2**
MM-MMAP	-0.49289	0.20383
MM-MUMP	-0.48244	0.21586
MM-BRK	-0.3287	0.30394
MM-PGM	0.040484	0.41911
MM-PFL	3.6979e-18	2.3393e-18
MM-PIN	0.044984	0.4169
MM-PRM	0.045682	0.41676
MM-MALL	0.11301	0.39067
MM-MDALL	0.27469	0.31592
MM-BREQ	-0.40028	-0.15454
MM-BALL	-0.40256	-0.1415
MM-SALL	0	0
MM-SDAL	0	0

Table V
PRINCIPAL COMPONENTS RELATED TO THE FILESYSTEM SUBSYSTEM.

	FS-PC1	**FS-PC2**
FS-RD	-0.43127	0.23118
FS-WR	-0.4317	0.23165
FS-OPN	-0.43153	0.23138
FS-CLS	-0.43154	0.23039
FS-ACC	0.23514	0.72445
FS-SEK	0.44712	0.51132

Table VI
PRINCIPAL COMPONENTS RELATED TO THE PROCESS MANAGEMENT SUBSYSTEM.

	TSK-PC1	**TSK-PC2**	**TSK-PC3**
TSK-CTS	0.13336	-0.43439	0.57759
TSK-CRT	0.37447	-0.31228	-0.21547
TSK-DEL	0.37452	-0.31264	-0.21511
TSK-MIG	-0.28049	-0.32948	-0.35243
TSK-WAIT	-0.28391	-0.33014	-0.34553
TSK-PMPT	-6.8013e-18	1.8667e-18	4.086e-18
TSK-RUN	0.34706	-0.46638	-0.018913
TSK-URUN	0	0	0
TSK-STP	0.22042	0.23902	-0.55882
TSK-FRK	-0.37781	-0.2142	0.030124
TSK-CLN	-0.37505	-0.21322	0.025204
TSK-EXEC	-0.3	-0.18032	-0.12614

In the case of Device Driver subsystem, hardware interrupt requests (*IRQs*), *softirqs*, and *workqueue* parameters provide the most significant contributions to the PCs. Tasklets do not significantly contribute since the considered device driver (Section III-D) does not adopt this kernel mechanism. The processing of I/O disk requests is then based on both *IRQs*, *softirqs* and *workqueues*. *IRQs* enable the device driver to manage events from the hardware device (e.g., an operation is completed); *softirqs* are adopted to postpone non-critical work (e.g., data processing) that otherwise should be performed by IRQ handlers; *workqueues* are adopted by the block I/O layer (a device-independent component that interfaces the kernel with block device

drivers) to manage pending I/O requests. For each I/O operation from the user, a sequence of *IRQ, softirq*, and *workqueue* events occur; therefore, these workload parameters are related each other, and the PCs take into account these relationships. It is finally worth noting that since our workload does not encompass network activity, we did not consider the Networking subsystem; we plan to perform further experimental campaigns in the future, to extend the analysis to other subsystems.

Table VII
PRINCIPAL COMPONENTS RELATED TO THE DEVICE DRIVER SUBSYSTEM.

	DR-PC1	DR-PC2
DR-IRQ	0.54346	-0.56596
DR-TSK	9.2541e-18	1.0458e-16
DK-SIRQ	0.4797	0.81545
DR-WQIN	-0.4871	0.0858
DR-WQEX	-0.48711	0.085825
DR-WQCR	0	0
DR-WQDL	0	0

C. Workload-Aging Correlation and Aging Sources Analysis

Multiple linear regression has been adopted to identify the relationships between aging trends and workload parameters of each subsystem. Table VIII provides the results of partial linear regression with respect to memory consumption trends. In particular, TSK-PC2, TSK-PC3, and FS-PC1 exhibit a statistically significant relationship with memory consumption trends (with 95% confidence). This reveals that the *process management* and the *filesystem* subsystems (which the mentioned PCs refer to) are subject to aging phenomena, since the aging trends vary with the usage of these subsystems.

Looking at the composition of these PCs, we can see that for both subsystems, almost all of the considered parameters contribute to the PCs (except the TSK-PMPT and TSK-URUN for the process management and FS-SEK for the filesystem). Therefore, there is no particular system call that contributes much more than others.

Table VIII
MULTIPLE LINEAR REGRESSION OF MEMORY CONSUMPTION TRENDS WITH RESPECT TO PRINCIPAL COMPONENTS.

	Coef.	p-value	Correlation
TSK-PC1	-0.132	0.739	No
TSK-PC2	-8.793	0.000	Yes
TSK-PC3	2.238	0.041	Yes
DR-PC1	0.486	0.659	No
DR-PC2	-6.432	0.388	No
FS-PC1	4.149	0.001	Yes
FS-PC2	0.409	0.636	No
MEM-PC1	-0.382	0.472	No
MEM-PC2	0.350	0.769	No

To confirm whether the subsystems identified by our methodology are actually involved in the aging phenomena, we carried out a more precise analysis on the filesystem. This analysis also serves as an example of how developers can delve into the results after applying the proposed experimental procedure. We selected the filesystem since it is one of the most critical and bug-prone component of the OS [27], [28].

In particular, the approach that we adopt to identify aging sources is to collect and to analyze more detailed data about memory consumption of the considered subsystem. From the analysis of the literature on the architecture of the Linux kernel [20], [21], we identified three data structures that are intensively used for filesystem activity; in turn, the management of these data structures can potentially be related to memory consumption trends:

- *Directory entry*: it used in path-related operations, such as searching the *inode* associated to a path; it represents a specific component in a path (e.g., a directory in the path or a file);
- *Inode*: it contains the information needed to manipulate a file or a directory (e.g., file metadata);
- *Buffer head*: it represents a set of disk blocks for I/O operations.

To check if the filesystem is involved in the memory consumption trends, we executed an experiment (running the Postmark workload with $N = 8$) in which we collected information about the mentioned data structures. More specifically, we periodically sampled the number of instances of these data structures stored in memory, by tracing the requests of the filesystem to the memory allocator for allocating or deallocating them.

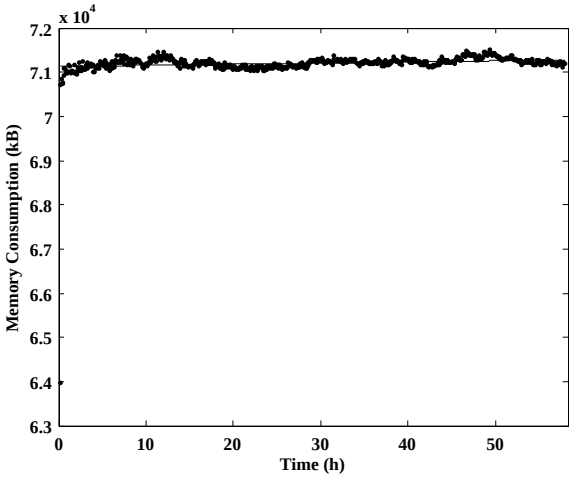

Figure 5. Memory consumption due to filesystem data structures.

Figure 5 shows the memory consumption due to the considered filesystem data structures. We observe an increasing trend (3.418 kB/h) that is statistically significant; it accounts for 17.25% of the overall memory consumption trend. We

then analyzed the source code of the filesystem, in order to identify memory management bugs related to filesystem data structures. We found that the memory consumption trend is due to the caching of these data structures by the filesystem. In particular, they are not immediately deallocated by the kernel, which can reuse them in subsequent filesystem operations. In the considered filesystem (*ext3*), there is no garbage collection mechanism to periodically reclaim this memory; as a result, these data structures are deallocated only when the available memory is too low. While this mechanisms can be desirable in some scenarios, it may not be the case for performance-critical application, since it causes degraded performance due to the increased memory pressure (i.e., the kernel needs more time than usual to allocate memory, since other data structures need to be deallocated). We foresee two strategies to prevent such a scenario:

- To proactively deallocate data structures before memory saturation; this mechanism could be triggered periodically;
- To impose a maximum size to the filesystem cache; newly allocated data structures could replace oldest ones on a least-recently-used basis.

V. Conclusion

The study reported in this paper focused on the analysis of software aging in the Linux OS kernel. The experimental procedure revealed that the Linux OS suffers from software aging phenomena, manifested as statistically significant memory consumption trends; instead, there were no statistically significant effect on system call latency. The analysis of collected data at subsystem level showed that aging dynamics are present in the *process management* and in the *filesystem* subsystems.

These results can be an aid for Linux kernel developers to find software aging bugs in such a large software. As an example of how these results can be exploited, we performed a deeper analysis of the filesystem. It revealed that a significant contribution of the experienced aging is due to this subsystem, caused by delays in crucial data structures deallocation. At the same time, we observe that the filesystem analysis does not fully account for the overall memory consumption trend; there are more aging sources within the kernel. From the statistical analysis of PCs, we hypothesized some additional sources of software aging that are worth being analyzed and reported to kernel developers:

- Since the Process Management subsystem was also related to the aging trend, the manifested aging dynamics suggest that potential aging sources are also in that subsystem; for example, memory could be leaked when a new process is started, or when a process exits;
- The conducted analysis on the filesystem could be extended to more data structures within the filesystem (for example, the filesystem journal);
- The analysis in this work could be repeated by increasing the set of workload parameters (Table II); more parameters may reveal additional relationships between kernel subsystems and aging trends, which can be followed to track down other software aging sources.

Carrying out additional experiments with the methodology outlined in this paper, by using different workload patterns, could lead to identify additional aging sources inside the kernel, other than those found in this work. Our future work will be focused on:

- *Exploring the Network Subsystem*, by designing experiments with a network-stressing applicative workload;
- *Looking for further potential aging sources* in the kernel, designing additional experiments according to the outlined methodology;
- *Defining effective software rejuvenation strategies*: the cost (both in time and complexity) of software rejuvenation techniques at the application (e.g., application restart) and OS layer (e.g., whole-system reboot) can differ by orders of magnitude. If the extent of software aging at each layer is precisely evaluated, then software rejuvenation techniques can be scheduled to maximize system availability (e.g., by scheduling rejuvenation at the application layer frequently, and rejuvenation at the OS layer only when needed). This would enable the design of more effective rejuvenation strategies.

References

[1] M. Grottke and K. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate," *IEEE Computer*, vol. 40, no. 2, 2007.

[2] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi, "A Methodology for Detection and Estimation of Software Aging," in *Proc. of the 9th Intl. Symp. on Software Reliability Engineering (ISSRE)*, 1998.

[3] K. Vaidyanathan and K. Trivedi, "A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems," in *Proceedings of the International Symposium on Software Reliability Engineering*, 1999.

[4] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of Software Aging in a Web Server," *IEEE Trans. Reliability*, vol. 55, no. 3, pp. 480–491, 2006.

[5] M. Balakrishnan, A. Puliafito, K. S. Trivedi, and Y. Viniotis, "Buffer losses vs. deadline violations for ABR traffic in an ATM switch: A computational approach," *Telecommunication Systems*, vol. 7, no. 1-3, pp. 105–123, 1997.

[6] L. Silva, H. Madeira, and J. Silva, "Software Aging and Rejuvenation in a SOAP-based Server," in *Proc. of the 5th IEEE Intl. Symp. on Network Computing and Applications (NCA)*, 2006, pp. 56–65.

[7] A. Chou et al., "An Empirical Study of Operating System Errors," in *Proc. ACM Symp. on Operating Systems Principles*, 2001.

[8] K. Kourai and S. Chiba, "A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines," in *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007, pp. 245–255.

[9] R. Matias Jr and P. Freitas, "An Experimental Study on Software Aging and Rejuvenation in Web Servers," in *Proc. of the 30th Intl. Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2006, pp. 189–196.

[10] D. Cotroneo, S. Orlando, and S. Russo, "Characterizing Aging Phenomena of the Java Virtual Machine," in *Proc. of the 26th IEEE Symp. on Reliable Distributed Systems (SRDS)*, 2007, pp. 127–136.

[11] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," in *The 25th International Symposium on Fault-Tolerant Computing*, 1995.

[12] A. Pfening, S. Garg, A. Puliafito, M. Telek, and K. Trivedi, "Optimal Software Rejuvenation for Tolerating Soft Failures," *Performance Evaluation*, vol. 27, 1996.

[13] S. Garg, A. Puliafito, and K. Trivedi, "Analysis of Software Rejuvenation using Markov Regenerative Stochastic Petri Nets," in *In Proceedings of the 6th International Symposium on Software Reliability Engineering*, 1995.

[14] Y. Bao, X. Sun, and K. Trivedi, "A Workload-Based Analysis of Software Aging, and Rejuvenation," *IEEE Transactions on Reliability*, vol. 54, no. 3, p. 541, 2005.

[15] K. J. Cassidy, K. C. Gross, and A. Malekpour, "Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2002.

[16] K. Vaidyanathan and K. Trivedi, "A Comprehensive Model for Software Rejuvenation," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 124–137, 2005.

[17] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory Leak Analysis in Mission-Critical Middleware," *Journal of Systems and Software*, vol. 83, no. 9, 2010.

[18] A. Avritzer, A. Bondi, M. Grottke, K. Trivedi, and E. Weyuker, "Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms," in *Intl. Conference on Dependable Systems and Networks*, 2006.

[19] N. Joukov, A. Traeger, R. Iyer, C. Wright, and E. Zadok, "Operating System Profiling via Latency Analysis," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 89–102.

[20] R. Love, *Linux Kernel Development*. Novel Press, 2005.

[21] D. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly, 2005.

[22] I. T. Jolliffe, *Principal Component Analysis*. Springer, 2002.

[23] N. R. Draper and H. Smith, *Applied Regression Analysis*.

[24] A. Traeger, E. Zadok, N. Joukov, and C. Wright, "A Nine Year Study of File System and Storage Benchmarking," *ACM Transactions on Storage*, vol. 4, no. 2, p. 5, 2008.

[25] K. Kanoun, Y. Crouzet, A. Kalakech, A. Rugina, and P. Rumeau, "Benchmarking the Dependability of Windows and Linux using PostMark™ Workloads," in *In Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, 2005.

[26] R. Bryant, R. Forester, and J. Hawkes, "Filesystem Performance and Scalability in Linux 2.4.17," in *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[27] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using Model Checking to Find Serious File System Errors," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[28] L. N. Bairavasundaram, M. Rungta, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Analyzing the Effects of Disk-Pointer Corruption," in *Intl. Conference on Dependable Systems and Networks*, 2008.